AGORA: A Dependable High-Performance Coordination Service for Multi-Cores

Rainer Schiekofer[1], Johannes Behl[2], and Tobias Distler[1]
[1]Friedrich-Alexander University Erlangen-Nürnberg (FAU) [2]TU Braunschweig

***Abstract*—Coordination services are essential building blocks of today's data centers as they provide processes of distributed applications with means to reliably exchange data. Consequently, coordination services must deliver high performance to ensure that they do not become a bottleneck for the applications depending on them. Unfortunately, the design of existing services such as ZooKeeper prevents them from scaling with the number of cores on a machine. In this paper, we address this problem with AGORA, a high-performance coordination service that is able to both effectively and efficiently utilize multi-core machines. AGORA relies on a primary-backup replication architecture that partitions the workload on each server to achieve parallelism while still providing similar consistency guarantees as ZooKeeper. Our evaluation shows that AGORA scales with the number of cores and thus can fully utilize the network resources available.**

I. INTRODUCTION

Coordination services such as ZooKeeper [1] and Chubby [2] greatly facilitate the operation of distributed applications by providing basic functionalities that would otherwise have to be integrated into each application individually. Examples of typical functionalities offered by coordination services include the provision of message queues, support for leader election, as well as reliable storage for small chunks of data. Focusing on such basic tasks, coordination services support a wide spectrum of client applications ranging from large-scale storage and data-processing systems [3], [4], over distributed network controllers in software-defined networks [5], to cloud-backed file systems [6]. As coordination services are essential building blocks of today's data-center infrastructures, they have been an active area of research in recent years, resulting in systems with improved availability [7], response times [8], composability [9], [10], and resilience [11], [12], [13], [14].

Providing an anchor of trust for the client applications relying on them, coordination services in general are replicated for fault tolerance. ZooKeeper and Chubby for this purpose, for example, apply a primary-backup approach [15] in which a leader server processes all state-modifying requests and then forwards the corresponding state updates to a set of follower servers using an atomic broadcast protocol [16], [17]. While this approach prevents the data stored by client applications from being lost in the presence of server crashes, it also comes at the cost of a performance penalty. As a consequence, some client applications currently only access the coordination service outside of their critical processing paths [5].

Acknowledgments: This work was partially supported by the German Research Council (DFG) under grants no. DI 2097/1-2 and KA 3171/1-2 ("REFIT").

Having analyzed state-of-the-art coordination services, we identified two main limitations preventing existing systems from achieving a higher read and write performance: First, the current internal architecture of coordination servers does not allow them to efficiently use the processing resources available on the respective machines. In particular, the fact that requests are forwarded along a chain of different worker threads leads to a high synchronization overhead. Second, the replication protocols used by state-of-the-art coordination services do not scale with the number of cores per server which, as our experiments with ZooKeeper show, results in a server not being able to utilize the entire available network bandwidth. A key reason for this property is the design decision to establish a global total order on all write operations, which creates a bottleneck that prevents existing coordination services from exploiting the full potential of modern multi-core machines.

To address these problems, in this paper we present AGORA, a dependable coordination service for client applications with high performance demands for both read and write operations. In contrast to existing approaches, AGORA is able to effectively and efficiently utilize multi-core machines due to relying on a novel parallelized primary-backup replication protocol. In order to parallelize system operations, AGORA dynamically partitions the coordination-service state and, on each of its servers, executes each partition on a dedicated core. With the vast majority of operations only accessing a single partition, in the normal case an AGORA server receives, handles, and answers a client request in the same thread, thereby significantly minimizing synchronization overhead.

Instead of totally ordering all state modifications at a global scale, each partition in AGORA orders its writes locally and reliably replicates them across multiple servers for fault tolerance. If demanded by a client, AGORA relies on an inter-partition protocol that allows the system to provide the client with a causally consistent view of the coordination-service state across partitions. Applying this mechanism, client applications are able to depend on AGORA for the same use-case scenarios as for ZooKeeper. In addition, AGORA offers a fast path for client requests with weaker consistency demands.

Our evaluation of the AGORA prototype shows that due to the reduced synchronization overhead AGORA achieves a significantly higher single-core efficiency than ZooKeeper for both read as well as write operations. Furthermore, in contrast to ZooKeeper AGORA scales with the number of cores per server and is therefore able to fully utilize the network resources available even for reads as small as 8 bytes.

In summary, this paper makes the following contributions: 1.) It presents AGORA, a coordination service for applications requiring high performance for both reads and writes. 2.) It discusses THOLOS, AGORA's parallelized primary-backup replication protocol, which enables the system to scale with the number of cores on a server. 3.) It evaluates AGORA in comparison to the state-of-the-art coordination service ZooKeeper.

II. BACKGROUND

In this section, we give an overview of existing coordination services and the consistency guarantees they provide.

Coordination Services. Coordination services [1], [2], [11], [18], [19] are essential for today's data-center infrastructures as they offer means to deliver information to and/or exchange data between processes of large-scale distributed applications. Use cases for such services, for example, include the distribution of configuration parameters, the implementation of message queues, as well as support for leader election [1], [2]. To provide such functionality, coordination services commonly comprise built-in storage capabilities designed for handling small chunks of data on behalf of clients (i.e., application processes). For this purpose, different coordination services rely on different abstractions, however, they all support basic operations such as creating, reading, writing, and deleting data nodes. In addition, many coordination services offer clients the possibility to subscribe to data-related events (e.g., the creation, modification, or deletion of a particular data node), for example, by registering a corresponding *watch* [1].

Being crucial for the well-functioning of other applications, coordination services are typically replicated for reliability and availability. As shown in Figure 1, a common approach in this context is a distributed primary-backup replication architecture in which one server, the *leader*, processes the requests issued by clients and then brings the other servers, the *followers*, up to speed by using a replication protocol [16], [17] to consistently forward state updates carrying the resulting state changes.

Consistency Guarantees. Besides a typical architecture, Figure 1 also shows a (simplified) example of how clients commonly exchange data using a coordination service [1]. To publish content, a sending client first stores the new information in multiple data nodes (`/data_*`) and then creates a special `/ready` node to complete the procedure. The rationale behind the `/ready` node is the following: First, the node serves as a confirmation by the sender that the contents of all the information-carrying data nodes involved have been successfully updated and that they consequently represent a consistent view of the data to be exchanged. This property is especially important in scenarios where a client fails during the sending process and another client takes over as sender, potentially forwarding data of different content. Second, by registering a watch informing them of changes to the `/ready` node, receiving clients can be notified about the presence of new data without needing to repeatedly poll the coordination service. Instead, receiving clients can wait until the watch on the `/ready` node triggers and then read the updated contents of the data nodes containing relevant information for them.

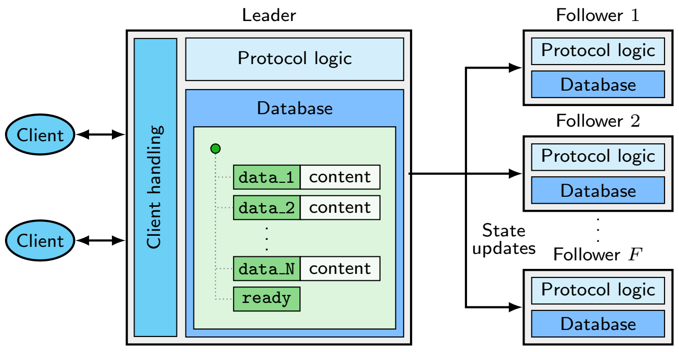


Figure 1. Basic coordination service architecture

In order for the approach discussed above to work properly, the consistency guarantees provided by a coordination service in general and its replication protocol in particular must ensure that at the moment a client learns of the creation of the `/ready` node, it must also be able to read the updated states of the data nodes that previously have been modified by the sender. Otherwise, a receiving client could be in danger of obtaining inconsistent data due to reading stale values.

Most coordination services [2], [11], [18], [19] address this issue by providing strong consistency, that is, they ensure that a read issued by a client sees all previous modifications. As a consequence, these services can guarantee that, if a client sees the `/ready` node, it also sees the new contents of all other data nodes involved as their modifications preceded the creation of the `/ready` node. As the example shows, strong consistency guarantees are often convenient for application programmers due to leading to intuitive semantics. On the downside, however, strong consistency usually also comes with a performance overhead: To provide such guarantees in the Chubby coordination service [2], for example, not only writes but also all reads have to be executed at the leader, causing this server to become a bottleneck. To mitigate this problem, Chubby utilizes client caches to reduce the number of read requests reaching the leader in the first place and relies on a latency-intensive cache-invalidation protocol that is run by the leader prior to executing each modification.

The ZooKeeper coordination service [1] circumvents the performance issues associated with strong consistency by relaxing the consistency guarantees offered to clients, using a combination of linearizable writes and FIFO client orders. In a nutshell, this means that all ZooKeeper servers apply all writes (i.e., state modifications) in the same total order; furthermore, ZooKeeper ensures that all (read or write) requests issued by the same client are executed in the order in which the client has sent them. In combination with the fact that the coordination service delivers notifications in the order of the corresponding events and with respect to regular replies, these guarantees are sufficient to fulfill the requirements discussed in the context of the data-exchange example illustrated in Figure 1. As a key benefit, the relaxed consistency guarantees enable ZooKeeper to employ a system architecture in which, in contrast to Chubby, reads can be performed by any server, offering the possibility to load-balance reads across machines.

III. Problem Statement and Approach

As ZooKeeper already allows to parallelize reads across different servers, in this section, we further analyze its use of multi-core machines. Based on our findings, we then formulate a number of requirements for Agora and discuss the basic approach our coordination service applies to fulfill them.

A. Problem Analysis

As explained in Section II, using relaxed consistency guarantees allows ZooKeeper to improve read performance. Nevertheless, the need to ensure linearizable writes prevents the coordination service from scaling with the number of cores on a server, because all writes still need to be executed sequentially at both the leader as well as the followers.

Figure 2 shows the basic architecture of a ZooKeeper server, which is based on a pipeline of different request processors and handles read and write requests differently: Incoming read requests bypass the pipeline and are directly handed over to the database containing the service state (i.e., the data nodes), where they are executed sequentially. In contrast, write requests are first forwarded to the protocol logic [17], which is responsible for processing the requests on the leader and reliably distributing the corresponding state updates to all servers in the same order. Once the protocol logic delivers a state update, a server applies the update to its database.

As illustrated in Figure 2, in order to be able to use more than one core, a ZooKeeper server executes most of its request processors in separate threads. Note that although such a server-internal organizational structure offers some parallelism, it also has several major disadvantages [14]: First, the approach leads to increased synchronization overhead each time two request processors have to communicate with each other, for example, to forward a message. Second, as soon as one of the threads completely saturates a core, the corresponding request processor becomes the bottleneck of the entire pipeline, thereby limiting the performance of the overall ZooKeeper server. Third, this server-internal architecture at best scales until the point at which the number of cores reaches the number of request-processor threads; if the number of available cores is increased beyond this point, a ZooKeeper server is unable to use them due to a lack of additional threads.

In summary, server architectures relying on a single pipeline of request processors by design provide limited scalability and only enable performance gains until a certain number of cores. Evaluation results show that this threshold in case of ZooKeeper lies at 4 cores [20], as also confirmed in Section V.

B. Goals and Challenges

Our analyses in the previous sections have shown that existing state-of-the-art coordination services are unable to exploit the full potential of today's multi-core machines, either because of executing an expensive replication protocol (e.g., Chubby, see Section II) or due to applying a server architecture with limited scalability (e.g., ZooKeeper,

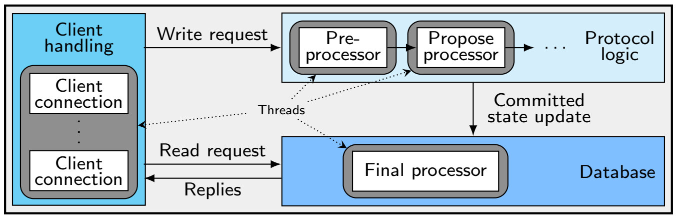


Figure 2. Basic architecture of a leader server in ZooKeeper

see Section III-A). This creates the need for a replication architecture that allows a coordination service to scale on multi-core machines and to efficiently use the processing resources available, while still providing consistency guarantees that are strong enough to be useful to clients. In the following, we further discuss each of these goals in more detail.

Scalability with the Number of Cores. In order to prevent a coordination service from becoming a bottleneck during periods of high workload, the service must provide high performance for both read and write operations. To achieve this goal while relying on a replication protocol for dependability, it is crucial for a coordination service to be able to scale with the number of cores available on its servers. Most importantly, this requires a replication protocol that, in contrast to the protocols used by existing coordination services, allows both read as well as write requests to be executed in parallel.

Efficient Resource Usage. Besides achieving high performance, a coordination service should efficiently use the available cores and, for example, minimize the synchronization overhead between threads. In particular, client requests should not be forwarded along a chain of different threads, leading to inter-thread communication at every handover. Instead, the coordination service should rely on a parallelization scheme that aims at reducing the number of synchronization points a request needs to pass while being handled by the service.

Useful Consistency Guarantees. Despite parallelizing the execution of both read and write operations on a server, a coordination service should still provide consistency guarantees that are strong enough to support the common use-case scenarios of these services. In particular, clients should still be able to exchange data between each other using the method discussed in Section II, requiring the presence of basic ordering guarantees with regard to state modifications and event notifications. This rules out weaker forms of consistency such as eventual consistency [21], which on the one hand would greatly simplify the goal of increasing performance due to introducing a loose coupling between servers, but on the other hand cannot ensure the necessary ordering properties.

C. Approach

Agora achieves scalability with the number of cores on a server by dynamically partitioning the coordination-service state and executing each partition on a separate core. As a result, reads and writes operating on different partitions can be processed in parallel. By running the entire request-processor pipeline of a partition in the same thread, AGORA eliminates partition-internal synchronization points and is consequently able to efficiently use the resources available. To support the same use-case scenarios as ZooKeeper, AGORA offers similar consistency guarantees and relies on an inter-partition protocol providing causal serializability, tracking and respecting causal dependencies between write operations on different partitions.

IV. AGORA

In this section, we present AGORA and its parallelized primary-backup replication protocol THOLOS in detail.

A. Overview

AGORA manages information using a tree of data nodes, which is accessible through a global namespace; for example, the path `/a/b/c` denotes a data node `c` that is the child node of another data node `/a/b`. At the start of the service, only the tree's root node / exists. After this point, clients can store and access information by creating, reading, updating, and deleting data nodes and their contents. In addition, AGORA enables clients to register watches for data-related events.

While the client interface of AGORA is similar to those of existing coordination services (cf. Section II), its architecture is not: As illustrated in Figure 3, internally AGORA is divided into several partitions, which are each responsible for managing a different part of the data-node tree. To balance load across partitions, the mapping of data nodes to partitions can be changed dynamically. For fault tolerance, each partition is handled by multiple servers executing the THOLOS replication protocol to ensure consistency. For this purpose, one of the servers assumes the role of a leader while the others act as followers. In case the current leader server crashes, the leader role is reassigned to a different server in the system.

As depicted in Figure 3, the number of partitions P usually equals the number of available cores per server. To minimize synchronization overhead, each partition on a server executes in a dedicated thread, utilizes separate network connections, and comprises its own instances of the client handling, protocol logic, and database modules, respectively. This way, most requests can be received, processed, and answered within a single thread. AGORA further increases resource efficiency by keeping direct interactions between partitions, which for example are required to perform load balancing, at a minimum.

A client communicates with AGORA by selecting a server from the group (i.e., the leader or a follower) to be its *contact server* and submitting all subsequent requests to this particular machine. In case a client assumes the contact server to have crashed, it switches to another server. To prevent partitions from having to compete against each other for the same network connection when interacting with the same client, a client establishes a separate connection to each partition. Using these links, a client can send requests for any data node to any partition. However, the reply to a request will always be returned by the partition currently comprising the data node accessed. Clients use this knowledge gained through replies to reduce synchronization overhead and improve performance by

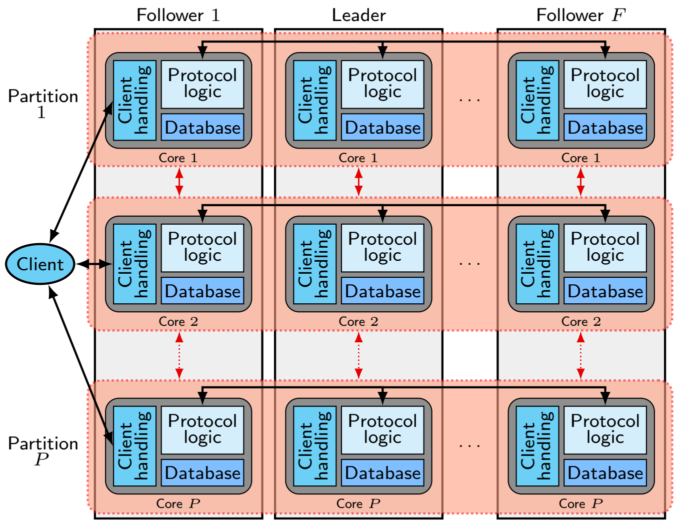


Figure 3. Architecture overview of AGORA

caching the partition IDs of data nodes and sending subsequent requests directly to the respective partitions. If a data node in the meantime has moved to another partition, for example as the result of a load-balancing procedure, the server side will forward the request accordingly and the client will update its cache after having received the reply from this partition.

In general, AGORA distinguishes between two different categories of client requests: reads, which do not modify the service state, and writes, which do. While a read request is directly handled by the contact server that receives it, a write request is forwarded to the leader and there processed by the corresponding partition. Having executed the request, the leader then relies on a partition-internal protocol to reliably distribute the resulting state update in order to bring the followers up to speed. Finally, based on this state update, the contact server creates a reply and sends it to the client that has issued the write. In summary, while writes affect the entire group of servers, reads are handled by only a single server.

B. System Model

Clients and servers, as well as servers amongst each other, are linked via (application-level) FIFO channels that deliver messages in the order in which they have been sent. If executed on top of an unreliable network, a protocol such as TCP is used to ensure reliable message delivery for connections. In order to be able to tolerate up to f server crashes, AGORA is replicated across a total of $2f + 1$ servers, each hosting replicas of all partitions. To minimize the synchronization overhead for requests operating on multiple partitions (see Section IV-E), the server assuming the leader role is the simultaneous leader of all partitions. If, for example due to problems with individual network connections, the leader is unable to fulfill its responsibilities for one or more partitions, AGORA treats the entire server as faulty and reassigns the leader role for all partitions to another server in the group.

C. THOLOS

Ensuring consistency in a distributed system in general requires some form of (logical) timeline representing the order in which relevant events such as state changes occurred [22]. In contrast to existing coordination services, however, AGORA's replication protocol THOLOS does not use a global totally ordered timeline for the entire system, but an individual timeline for each partition. The rationale behind giving each partition its own timeline is that this approach significantly reduces the synchronization overhead between partitions, as it allows partitions to make progress without having to order the requests they process with respect to the requests executed by other partitions. In the following, we first discuss how THOLOS replicates data within a partition and then present details on its handling of dependencies between partitions.

1) Intra-Partition Replication: AGORA replicates the contents of each partition across different servers to ensure availability in the presence of server crashes. For this purpose, each partition independently applies a replication scheme in which the leader establishes an order on all writes issued to the partition, creates corresponding state updates, and then distributes these updates to all servers (including itself) using the crash-tolerant atomic broadcast protocol Zab [17]. Once Zab delivers such an update, a server applies the state modification to its local database storing the data nodes of the partition. Due to the fact that Zab guarantees reliable and in-order delivery of transactions even in case of faults, the partition databases of different servers remain consistent as all servers apply all state modifications to a partition in the same order. As a result, the sequence numbers Zab assigns to state updates establish a partition-specific timeline representing the progress of the partition state. In contrast to writes, reads are not part of this timeline as they are only processed by the respective contact server and therefore not subject to intra-partition replication.

2) Handling Inter-Partition Dependencies: While handling the timelines of different partitions in isolation increases performance for many use cases (e.g., independent writes), there are some cases in which partitions cannot be treated separately due to the existence of dependencies between requests. In the scenario discussed in Section II, for example, it is crucial that the `/ready` node only becomes visible to a client, if the client also sees the updated states of all other `/data_*` data nodes, even if they are currently managed by other partitions. Fulfilling such a requirement is complicated by the fact that by design different partitions may advance at different speeds.

Timestamps. To support scenarios that introduce dependencies between partitions, AGORA relies on timestamps $t = (s_1, s_2, ..., s_P)$ that are composed of the state-update sequence numbers s_p of individual partitions p; as such, the timestamps are similar to vector clocks [23]. In particular, the p-th element of a timestamp represents the sequence number up to which partition p has advanced. Two timestamps t_1 and t_2 can be compared using an "is smaller than" relation $\prec$ that pairwise compares the timestamps' elements and is defined as follows: $t_1 \prec t_2 \Leftrightarrow (\forall i : t_1[i] \leq t_2[i]) \wedge (\exists i : t_1[i] < t_2[i])$.

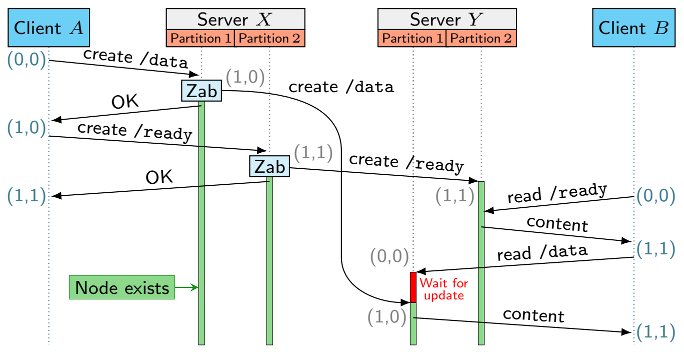


Figure 4. Use of timestamps in THOLOS (simplified example)

To avoid a central synchronization point, each partition maintains a local timestamp, that is, its own view of the overall system progress, and includes this timestamp in all messages exchanged with clients, other partitions, as well as other servers. When a partition learns another timestamp t_x through a state update or write request, it merges its local timestamp with t_x by pairwise computing the maximum of elements. Consequently, the local timestamps of partitions can only increase over time. With partitions rarely interacting with each other directly, the common way for a partition to learn that another partition has made progress is through communication with clients that have previously accessed other partitions.

Determining Dependencies Between Partitions. Maintaining partition-local views of the system progress using the timestamps described above has a key benefit: It enables partitions to track causal dependencies to other partitions. In particular, by comparing the timestamp attached to a request with the local timestamp, a partition can determine the point in time at which it is able to process the request in order to provide a causally consistent reply. Figure 4 illustrates how this mechanism works in practice, continuing the use-case example of Section II, in which the creation of a `/ready` data node represents the completion of a client's information upload procedure, indicating to other clients that all modified data nodes are up to date. For clarity, the figure only shows two servers, two data nodes (each in a separate partition), and two clients: a client A uploading data, whose contact server is server X, and a client B acting as reader, which is connected to server Y. Initially, all timestamps are assumed to be $(0, 0)$.

In the example, when client A issues a request to create a data node `/data`, AGORA decides to assign the data node to partition 1, which then executes an instance of the Zab protocol to distribute the corresponding state update within the partition. Being the first modification to the contents of this partition, the state update is assigned sequence number 1. Once Zab delivers the state update to partition 1 at server X, the partition sets its local timestamp to $(1, 0)$, applies the update to its database, and returns a reply carrying the new timestamp to client A. In a similar way, the node `/ready` is then created on partition 2. However, in this case due to client A providing the timestamp $(1, 0)$ it has obtained in the first step, the resulting merged and updated timestamp on partition 2 is $(1, 1)$.

As shown in Figure 4, the loose coupling between partitions may lead to different servers executing the state changes of different partitions in different orders. This effect is usually caused by the fact that Zab delivers a state update when a majority of servers (i.e., $f+1$ of the $2f+1$) has committed the update, creating scenarios in which a contact server confirms the execution of a write at a point in time at which most servers, but not necessarily all, have acknowledged the operation. Consequently, if partitions advance at different speeds it is, for example, possible that at server Y the update for node `/ready` in partition 2 is delivered earlier than the update for node `/data` in partition 1. In general, this does not pose a problem, however, as discussed below, it requires additional measures to handle causally dependent state modifications.

When client B reads node `/ready` from server Y after partition 2 of the server has created the node in its database, the client receives a reply indicating that the node exists. In this context, client B also obtains the partition's current timestamp $(1, 1)$, which it includes in a subsequent read request for node `/data`, expecting to read the latest information. In the example in Figure 4, however, the read request arrives before partition 1 of server Y has created node `/data`, which means that the partition must not process the request right away in order to preserve the semantics of the `/ready` node. A partition p in AGORA is able to detect such scenarios by comparing the p-th element $t_{req}[p]$ of the timestamp t_{req} attached to a request with the p-th element $t_{local}[p]$ of its current local timestamp t_{local}. If $t_{req}[p] \leq t_{local}[p]$, the partition immediately executes the request as the timestamp comparison confirms the partition to be sufficiently up to date. On the other hand, if $t_{req}[p] > t_{local}[p]$, the partition learns that the client has already seen effects of state modifications (such as the creation of the `/ready` node), which may be causally dependent on other state changes the partition itself has not yet applied (such as the creation of the `/data` node). In such case, the partition delays the processing of the request until having advanced to the point at which $t_{req}[p] = t_{local}[p]$. Note that the delayed execution only affects requests with higher $t_{req}[p]$; requests from other clients with lower $t_{req}[p]$ arriving in the meantime remain unaffected and are processed immediately.

Summary. THOLOS offers scalability by dividing the service state into disjoint partitions and independently executing Zab within each partition to order local state modifications. To keep the number of synchronization points between partitions low, THOLOS relies on timestamps that, without the need for interaction with other partitions, allow a partition to determine when it is able to give a causally consistent reply to a request.

D. Consistency Guarantees

In the following, we discuss the consistency guarantees provided by AGORA for read and write operations.

Atomic Operations on Data Nodes. All operations on data nodes in AGORA are executed atomically. For operations that access only a single data node (e.g., reads and writes), this is ensured by the fact that at each point in time each data node is assigned to a single partition, which executes all operations on its local data nodes in the same thread (see Section IV-A). In addition, AGORA guarantees atomic execution of operations accessing multiple data nodes (e.g., creates and deletes), which besides affecting the specified data nodes also involve their respective parent nodes. For create operations, AGORA achieves this by assigning the responsibility of creating a new data node to the partition that currently comprises the parent node. This partition then atomically handles the creation of the data node. After that, the new data node may be migrated to a different partition. For delete operations, additional measures have to be taken as a data node and its parent are possibly assigned to different partitions. Here, AGORA relies on a dedicated mechanism for multi-partition operations further explained in Section IV-E that splits the delete request into two partition-local operations, one for the child node and one for the parent node, and ensures a consistent execution across partitions.

FIFO Client Order. Besides invoking the coordination service synchronously (i.e., having at most a single outstanding request per client at a time), AGORA also offers clients the possibility of asynchronous calls. As a key benefit, asynchronous calls enable clients to use the service more efficiently by submitting multiple operations at once without having to wait for the system to respond between operations. For both synchronous as well as asynchronous calls, AGORA guarantees that the requests issued by the same client are processed in the order in which the client has sent them. As shown by the example described in Section II, for some use-case scenarios of coordination services providing this property is crucial for correctness, for example, when the creation of a particular data node (i.e., `/ready`) marks the end of a batch of client actions.

In case of synchronous calls, ensuring request execution in FIFO client order is trivial, because a client only submits a subsequent request after having received the reply to its previous one. For asynchronous calls to the same partition, keeping the FIFO order is also straightforward as each partition processes all requests sequentially. However, the same does not apply to cases in which subsequent asynchronous operations issued by the same client access different partitions. To handle such scenarios, an AGORA server, for each client, stores the latest timestamp it has sent to the client and updates the timestamp on each reply, thereby reproducing the view each client has on the progress of the coordination-service state. As a consequence, the server is able to retroactively attach the same timestamps to the asynchronous requests that the client would have used if it had issued the requests synchronously. That is, in essence AGORA transforms asynchronous requests at the client side into synchronous requests at the server side.

Causal Serializability. In AGORA, write requests to the same partition are serialized by the leader and afterwards processed by all servers in the determined order. If a write (to one partition) causally depends on a previous write (to another partition), AGORA ensures that all clients will observe the two writes in the order in which they occurred. In combination with the guarantee of atomic operations on data nodes, this results in write operations in AGORA being causally serializable [24].

As discussed in Section IV-C2, to provide these consistency guarantees a partition in some cases may have to wait until being able to execute a read request, for example, if a client has already witnessed progress on another partition. While this temporary delay is necessary for reads requiring causally consistent results, there is no need to penalize clients with weaker consistency demands. To address this issue, AGORA besides normal (causally consistent) reads also offers *fast reads*. As normal reads, fast reads always return state that has been committed by the replication protocol of a partition. However, unlike normal reads, fast reads are immediately executed after their reception and therefore do not consider causal dependencies between data nodes on different partitions.

Comparison with ZooKeeper. AGORA and ZooKeeper both guarantee that all (read as well as write) operations on data nodes are executed atomically and that requests issued by the same client are processed in FIFO order [1]. However, while ZooKeeper establishes a global order on all state modifications in the system, AGORA only orders writes within each partition and in addition respects causal dependencies between partitions. To understand the difference between the two systems, it is helpful to assume the existence of two applications that both rely on the same coordination service but operate on disjoint sets of data nodes. Using ZooKeeper, all writes are serialized even though there are no dependencies between the two data-node sets that would require them to be brought into a unique order. In contrast, AGORA allows the two independent data-node sets to be assigned to different partitions, thereby enabling a parallel execution of requests issued by the two applications. Furthermore, if a data-node set were to be distributed across different partitions, AGORA would respect causal dependencies between operations of the corresponding application. In summary, AGORA contrary to ZooKeeper enables write parallelism, while still providing guarantees that are strong enough to coordinate the actions of clients operating on the same set of data nodes.

Having examined a wide spectrum of use-case scenarios for ZooKeeper (e.g., distributed locks, barriers, leader election [1]), we are not aware of any use case that would actually require a total order on writes and therefore could not be handled by AGORA. Of all the scenarios investigated, we identified the use of a `/ready` node as discussed in Section II to be the most challenging with regard to consistency guarantees.

E. Multi-Partition Operations

As for single-node operations, AGORA also guarantees atomicity for multi-node operations, even when the data nodes involved are handled by different partitions. For this purpose, AGORA splits a multi-partition operation into different single-partition operations and assigns them a common timestamp that prevents clients from observing intermediate states.

Basic Concept. Figure 5 illustrates the basic concept behind this approach using an example in which a client A deletes a data node `/a/b` on a partition 1 whose parent node `/a` is handled by a partition 2. For clarity all timestamps initially are $(0, 0)$. When partition 1 on the leader receives the delete request, it creates a new timestamp $(1, 0)$ and sends it attached to the request to partition 2. On the reception of the timestamp, partition 2 merges it with its local timestamp and increments the resulting timestamp. As a result of these first steps, both partitions have a) each assigned a sequence number of their local timelines to the request and b) created a timestamp $(1, 1)$ that reflects these sequence numbers and can be used by partition 2 to order the operation. When Zab delivers the operation, partition 2 updates the parent node `/a` and instructs partition 1 to remove the child node `/a/b` from its database.

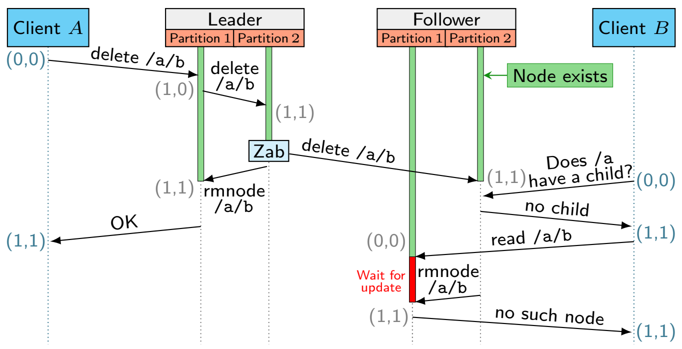


Figure 5. Handling of multi-partition operations (basic concept)

The right-hand side of Figure 5 shows how AGORA's handling of multi-partition operations guarantees atomicity and in particular ensures that clients cannot observe intermediate execution states. If, for example, a client B queries the data nodes affected by a deletion while the operation is in progress and learns about the absence of a child node, the client afterwards will not be able to access it, independent of which partition the node had been assigned to and of how much progress the partition has made on the client's contact server. This is due to the fact that, as illustrated in Figure 5, if a client observes the effects of a multi-partition operation on one partition, it at the same time also obtains the timestamp of the operation. By including this timestamp in subsequent requests, the client enables other partitions on slow servers to determine that they have fallen behind and first need to catch up before processing the client's request (cf. Section IV-C2).

Ensuring Consistency Across Partitions. AGORA ensures that the database of each partition remains consistent across servers by executing a separate Zab protocol in each partition to order state modifications within the partition. In addition, AGORA guarantees that the databases of different partitions remain consistent with regard to the effects of multi-partition operations; that is, if for example a delete request completes at the parent-node partition, it also completes at the child-node partition. To provide such a guarantee in the presence of faults, it is necessary to address scenarios in which one of the partitions advances to the point at which it is able to complete its part of a multi-partition operation, while another partition involved in the same operation does not, for example, due to the leader server having crashed in the meantime. In AGORA, such scenarios can occur as partitions are largely independent and do not necessarily always advance at the same speed.

To handle such scenarios, a partition only updates its database in the context of a multi-partition operation if a) the operation has been delivered by Zab and b) the partition has the confirmation that on all partitions involved in the operation, the respective Zab instance has delivered all state updates with sequence numbers lower than $s_{p,o}$, with $s_{p,o}$ being the local sequence number a partition p has assigned to the multi-partition operation o. The latter requirement demands additional local interaction between the partitions involved on each server (omitted in Figure 5), but it guarantees that databases remain consistent across partitions even if the leader changes: As all previous operations on the affected partitions are committed by their respective Zab instances, a new leader cannot retroactively change the local timelines; therefore, the effects of the multi-partition operation remain stable.

F. Load Balancing

AGORA allows the mapping of data nodes to partitions to change at runtime in order to perform load balancing. For this purpose, each partition continuously records workload statistics such as the number of data-node accesses or the amount of network traffic. To minimize memory and analysis overhead, a partition divides its data nodes into groups and manages a single set of combined statistics per group. By default, AGORA uses hash values of node paths to assign data nodes to groups. However, if provided, a partition can also exploit knowledge about clients to form groups, for example, combining data nodes that are often accessed together.

To prepare load-balancing decisions, partitions periodically report their workload statistics to a load-controller component running on the leader, which is consequently able to determine if a load imbalance between partitions exists. If this is the case, the load controller identifies partitions with high load and instructs them to migrate certain fractions of their load to specific partitions with low load to minimize the differences between partitions. Although the load controller decides on the partitions to participate in load balancing, it is the responsibility of a partition to select the data nodes to be reassigned to the other partition. This allows a partition to only include data nodes that can currently be migrated safely, for example, due to not being involved in a multi-partition operation.

G. Watches

To avoid repeated polling, AGORA offers a client the possibility to register watches for the creation, modification, and deletion of data nodes, resulting in the client to be notified by AGORA when the corresponding event occurs. As for reads and writes, AGORA also respects causal dependencies for watches, ensuring for example that if a client in the use-case scenario in Section II has registered watches for both data nodes `/data_1` and `/ready`, the client will observe events in the order in which the data nodes have been created, even if they are in different partitions. To achieve this, each server locally manages the watches of its clients, separated into partitions. When an event occurs for which a watch has been registered, the corresponding partition sends a notification carrying the timestamp of the state update that triggered the event to the client. Furthermore, the partition forwards the notification to all other partitions on the same server. In reaction, the other partitions wait until their local timestamps are at least as high as the notification timestamp and then also send the notification to the client. The client handles an event once a) it has received a notification from every partition and b) there are no notifications of unhandled events with lower timestamps.

The mechanism described above ensures that if an event e_2 causally depends on another event e_1 (i.e., the timestamp of e_1 is smaller than the timestamp of e_2), at least the partition handling event e_1 will first send the notification for event e_1 and then the notification for event e_2. With clients and servers being linked via FIFO channels, it is guaranteed that the client learns about event e_1 before obtaining all notifications for event e_2, consequently handling event e_1 prior to event e_2. As a counterpart to fast reads, AGORA allows clients with weaker consistency demands to register *fast watches*, that is, watches for which only the affected partition returns a notification.

H. Implementation

Our AGORA prototype is based on the code of ZooKeeper (version 3.4.6). For AGORA, we refactored ZooKeeper's request-processor pipeline by separating the modules containing the protocol logic from the threads executing them, following the concept of actor-based programming [25]. As a key benefit, this allows us to execute the entire protocol logic of a partition in a single thread, thereby eliminating all internal synchronization points from the request-processor pipeline. To create multiple partitions, we instantiate the pipeline multiple times and execute each partition in a dedicated thread.

V. EVALUATION

In this section, we compare the single-core efficiency and multi-core scalability of AGORA to the respective characteristics of ZooKeeper. In all cases, the coordination services run on three servers (8 logical cores, 2.27 GHz, 8 GB RAM), which are connected via switched Gigabit Ethernet to two servers with up to 500 clients (8 logical cores, 3.4 GHz, 8 GB RAM). Unless stated otherwise, the experiments take two minutes and data points represent the average of three runs.

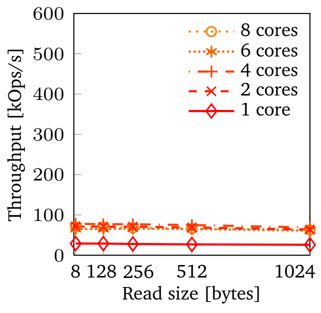
A. Throughput

In our first set of experiments, we evaluate the maximum read and write throughput achievable for ZooKeeper and AGORA depending on the number of cores the respective coordination service has at its disposal on each server. For this purpose, clients asynchronously query and modify data-node contents of different sizes ranging from 8 bytes to 1 kilobyte, which are typical sizes for coordination services [1], [2]. As reads in both systems are only handled by the server that receives them, in the read experiments we configure all clients to select the same contact server in order to be able to create a high read workload; as a consequence, we report per-server throughputs for reads. In contrast, writes in both systems affect the entire server group due to the need to replicate state updates across all servers. Therefore, we configure clients to use different machines as contact servers to modify data nodes and report overall throughputs for writes. Figures 6 and 7 present the results for read and write operations, respectively.

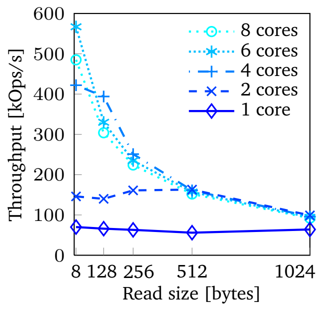
Single-Core Efficiency. To evaluate the resource efficiency of both systems, we limit each of the coordination services to only a single core per server. For ZooKeeper, this is achieved by instructing the operating system to execute all threads on the same core. In contrast, for AGORA this involves using a single partition that is executed in a single thread. Our results show that for the single-core configuration, ZooKeeper provides maximum throughputs of 26–29 kOps/s for reads and 7–8 kOps/s for writes of different sizes. As expected, writes are more expensive than reads due to being replicated to all servers. The same is true for AGORA, however, here the achieved maximum read (56–70 kOps/s) and write (17–31 kOps/s) throughputs are significantly higher than in ZooKeeper. With both system implementations sharing a common code base, these differences are the result of AGORA's approach to eliminate synchronization points within a partition by running the entire request-processor pipeline in a single thread (see Section IV-H), thereby efficiently using the resources available and improving maximum throughput by up to a factor of 2.4 (reads) and 4.3 (writes), respectively.

Multi-Core Scalability. When we increase the number of cores attributed to each server, ZooKeeper provides higher read and write throughputs until reaching a threshold at 4 cores. After this point, adding more cores has a negative effect on the performance of ZooKeeper as synchronization overhead increases; this observation confirms similar findings by Santos et al. [20]. In contrast to ZooKeeper, AGORA does not only scale up to 4 cores per server but is able to exploit additional resources available until saturating the network, which is the limiting factor in most of our experiments. Due to efficiently using processing resources, AGORA for example already reaches network saturation for reads as small as 8 bytes when relying on 6 or more partitions. In comparison, ZooKeeper is not even network-bound for 1-kilobyte reads, independent of the number of cores a server has at its disposal.

As shown in Figure 6b, for reads, AGORA configurations with fewer partitions (e.g., four) in some cases achieve higher throughputs than configurations with more partitions (e.g., six or eight). This is a result of a combination of a) the system being network bound and b) the fact that the sizes of timestamps in AGORA depend on the number of partitions (see Section IV-C2). With reads returning multiple meta-data timestamps (e.g., indicating the creation and latest modification of a data node), the replies of configurations with fewer partitions are therefore smaller, which results in a higher throughput if the network is saturated. As the performance of modifying operations is dominated by the replication protocol, our results do not show the same effect for writes, allowing for example the 8-partition configuration of AGORA for writes of 8 bytes to achieve a 1.9 times higher maximum throughput than the 4-partition configuration; this is a factor of 3.8 compared to the highest 8-bytes write performance of ZooKeeper.

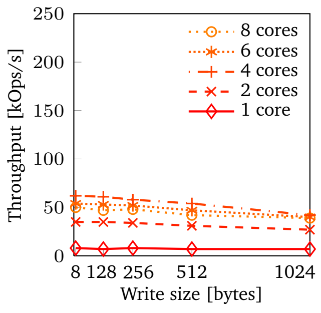


(a) ZooKeeper

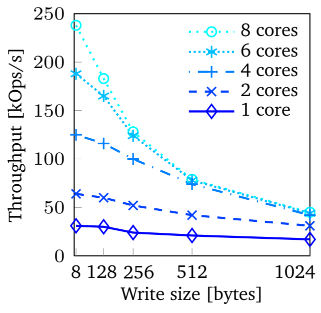


(b) AGORA

Figure 6. Read throughput per server

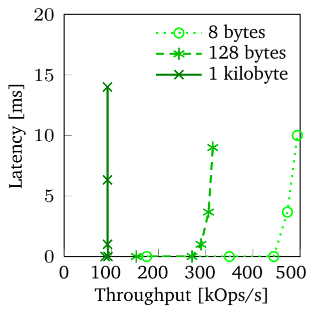


(a) ZooKeeper

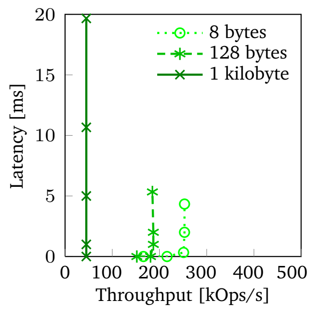


(b) AGORA

Figure 7. Overall write throughput



(a) Reads

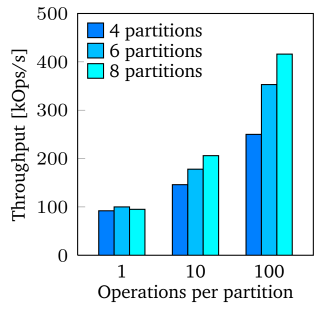


(b) Writes

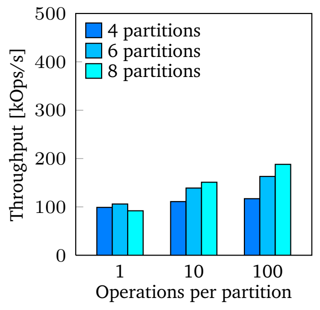
Figure 8. Latency provided by AGORA (8 partitions)

B. Latency

In the next experiment, we assess the response times of AGORA for different read and write operations by stepwise increasing the client load on a configuration with 8 partitions. As shown in Figure 8, for all data-node content sizes evaluated AGORA is able to provide low latencies as long as there is a sufficient amount of network resources available. In particular, in all cases AGORA delivers throughputs in under a millisecond that are higher than the respective maximum throughputs of ZooKeeper. For example, for reads and writes of 8 bytes the throughput AGORA can handle with sub-millisecond latency is 5.7 and 4.0 times higher than the maximum throughput achieved by ZooKeeper for these workloads, respectively.



(a) Reads (8 bytes)



(b) Writes (8 bytes)

Figure 9. Impact of asynchronous calls to different partitions

C. *Synchronization Overhead Between Partitions*

As discussed in Section IV-D, AGORA ensures FIFO order on the execution of requests issued by the same client even if subsequent requests access data nodes on different partitions. However, providing such a guarantee in the presence of asynchronous requests requires additional synchronization between partitions at the server side. To evaluate the overhead associated with this mechanism, we conduct experiments in which clients perform asynchronous reads and writes of 8 bytes on data nodes that are assigned to different partitions, varying the number of consecutive operations on the same partition before a client accesses another partition. To ensure valid results, we statically control the mapping of data nodes to partitions and also disable AGORA's load balancing for this experiment.

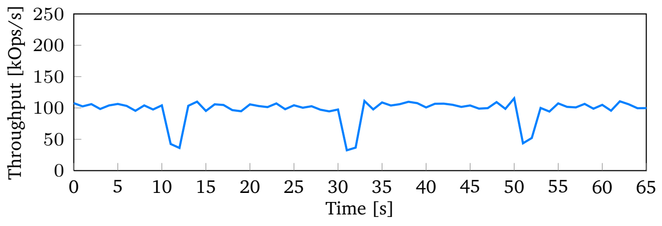
Figure 9 shows that for the worst-case scenario of all clients changing their respective partitions on each request, AGORA achieves a throughput of about 92–106 kOps/s for both reads and writes, almost independent of the total number of partitions. That is, despite the worst-case synchronization overhead, AGORA is still able to provide throughputs that are significantly higher than the maximum throughputs achieved by ZooKeeper for the same operations (cf. Section V-A). As further illustrated by the results presented in Figure 9, if clients switch partitions less frequently, a higher read and write performance is achievable. For example, if clients issue 10 consecutive requests to the same partition before accessing data nodes on a different partition, throughputs increase to 146–206 kOps/s for reads and 111–151 kOps/s for writes.

In summary, the results of this experiment allow us to draw two main conclusions: First, it is beneficial to collocate data nodes accessed by the same client on the same partition in order to minimize the costs for guaranteeing FIFO client order; in AGORA, this is possible by exploiting application-specific knowledge when configuring the coordination service's assignment of data nodes to load-balancing groups (see Section IV-F). Second, if a client does not need a read to return a causally consistent result, the client should exploit the fact that AGORA in addition to normal reads also offers the possibility to issue fast reads (see Section IV-D), as fast reads in no case require synchronization between partitions but instead are always processed immediately by a server.

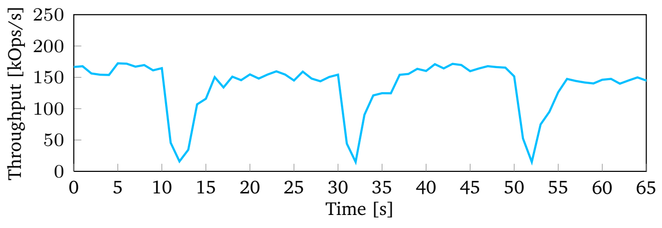
D. *Load Balancing*

In our last experiment, we evaluate the effectiveness of AGORA's load-balancing mechanism responsible for dynamically reassigning data nodes in scenarios where client requests are unevenly distributed across partitions (see Section IV-F). For this purpose, we instruct 250 clients to repeatedly perform writes of 8 bytes on different data nodes and to deliberately create worst-case load imbalances by simultaneously changing the data nodes they access at predefined points in time.

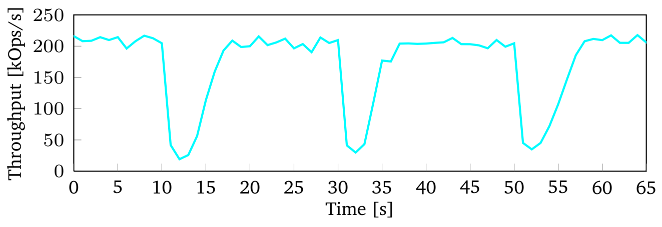
Figure 10 presents the results of this experiment for three AGORA configurations with different numbers of partitions. At the beginning of the experiment, the data nodes accessed by clients are evenly distributed across the available partitions. About ten seconds into the experiment, all clients shift their workload to a set of data nodes that at this point are all handled by the same partition, causing all other partitions to become idle. As a result of the entire write workload needing to be processed by a single partition, the overall throughput decreases to a fraction of its initial value. Having detected the load imbalance, AGORA starts to reassign most of the currently accessed data nodes to other partitions in order to once again have a balanced system. In the same way, AGORA reacts to similar load imbalances 30 and 50 seconds into the experiment. Our results of 16 test runs per setting show that it takes AGORA's load balancer on average 3.7 seconds (4 partitions), 5.4 seconds (6 partitions), and 7.5 seconds (8 partitions), respectively, to rebalance load in the worst-case scenarios.



(a) 4 partitions (writes of 8 bytes)



(b) 6 partitions (writes of 8 bytes)



(c) 8 partitions (writes of 8 bytes)

Figure 10. Impact of AGORA's load-balancing mechanism

VI. Discussion

Our evaluation has shown that due to efficiently using processing resources, AGORA in contrast to ZooKeeper in most experiments is able to fully utilize the network resources available, resulting in the network to be saturated even for small reads. A possible way to mitigate this problem is to introduce additional network hardware and to equip each AGORA server with additional network cards, ideally one network card per partition. In particular, such a measure would have two main benefits: First, as a consequence of the increased capacity more data could be exchanged between clients and servers as well as among servers, allowing the system to achieve a higher throughput for both reads and writes. Second, providing each partition with its own network card would eliminate an unnecessary synchronization point currently existing in our AGORA prototype implementation: the single network card all partitions use to send and receive messages. Note that the ability to benefit from additional network cards is a direct property of AGORA's parallelized replication architecture relying on largely independent partitions. In traditional architectures for example used for ZooKeeper, scaling with the number of network cards is not straightforward because the replication protocol still needs to establish a global timeline on events.

VII. Related Work

Below, we discuss existing works that address the performance and scalability of replicated (coordination) services.

Atomic Operations. To increase performance, some coordination services [1], [19], [26] offer clients the possibility to issue transactions containing several requests that are then executed atomically. Building on the idea of processing multiple operations at once at the server side, Kalantari et al. [27] presented an approach to handle sequences of operations submitted by different clients relying on a deterministic multi-threaded server. Going one step further, we proposed to make coordination services extensible [5], thereby enabling clients to register custom code that is then executed atomically by the coordination service. All three approaches are orthogonal to this work and could therefore also be integrated into AGORA.

ZooKeeper-specific Approaches. As state modifications have to be consistently replicated, adding servers to a ZooKeeper instance usually does not increase but decrease write throughput. Therefore, the common approach to address write-performance problems is to statically distribute the data across multiple ZooKeeper instances [1], thereby paying the maintenance costs associated with operating more than one deployment. Besides, a possible way to increase read throughput in ZooKeeper without noticeably harming write performance is to introduce observers [28], that is, servers that only passively learn committed state updates from others. However, this comes at the cost of additional network traffic. As our experiments have shown, by utilizing the resources available on multi-core machines more efficiently, AGORA achieves significantly higher read and write performance compared with ZooKeeper using the same hardware. Consequently, techniques such as the ones described above are less likely to become necessary.

Composition of Coordination Services. Lev-Ari et al. [10] proposed a system design that facilitates the development of applications that are distributed across different data centers, as it allows a coordination-service client to compose multiple service instances. As a key advantage, their approach can be implemented by extending the client library with a layer that enforces consistency across service instances by injecting synchronization requests into the stream of client operations. On the downside, handling this task entirely at the client comes at the cost of higher latency compared with AGORA's server-side consistency enforcement. Furthermore, relying only on the client complicates essential tasks such as dealing with server faults as well as load balancing between service instances.

Partitioning. ZooFence [29] partitions the coordination-service state and distributes it across multiple different ZooKeeper instances. Unlike AGORA, ZooFence requires additional proxy components to delegate requests to the service instances responsible. In general, state partitioning is a technique to achieve scalability with the number of servers, for example, in a distributed file system [30], [31], [32]. In addition, partitioning has also been proposed for ordering client requests in replicated systems [33]. For AGORA, we also rely on service-state partitioning for scalability. However, the goal we focus on is not scalability with the number of servers in the system, but with the number of cores on each server. As a result, compared to the communication costs that would be necessary if partitions were to be hosted by different servers, interaction between partitions in AGORA is comparably cheap. Nevertheless, to minimize synchronization overhead we still aim at reducing inter-partition communication where possible.

Parallelized Replication. Multi-Ring Paxos [34] partitions the service state of a replicated system and distributes the ordering of requests across multiple atomic-multicast groups so that requests are only delivered to the servers that execute them. S-SMR [35] builds on this idea and provides linearizability by coordinating servers that execute operations having dependencies on each other due to accessing the same part of the service state. A server in P-SMR [36] processes independent requests in parallel, synchronizing threads only for dependent operations. In AGORA, requests not only have dependencies with respect to the service state they operate on but also with respect to the client that has issued them. This means that to be able to guarantee both data-node consistency as well as client FIFO order, using these approaches, all requests would have to be handled by the same multicast group, thereby eliminating the advantages of having multiple independent groups.

COP [14] is a concept to parallelize the agreement of requests in Byzantine fault-tolerant systems, utilizing multiple threads per server that are each responsible for contributing a different part of the global sequence of requests. In contrast to COP, AGORA does not provide strong consistency and therefore does not establish a total order on all requests, but only orders writes affecting the same partition. Furthermore, AGORA in addition to exploiting multiple cores during request ordering also enables parallelism during request processing.

Rex [37] and Eve [38] parallelize the execution of requests in a replicated system in order to take advantage of multi-core machines at this system stage. To ensure consistency between servers, Rex records execution traces on the leader and deterministically replays them on the followers. Eve, on the other hand, executes requests on all servers and if necessary corrects the effects of nondeterminism afterwards. Both approaches are especially beneficial for use cases for which a) it is impossible or expensive to determine in advance whether requests might influence each other during execution or b) the actions that have an impact on other requests only comprise a small portion of the entire execution. In contrast, in AGORA it is straightforward to identify conflicts between requests by comparing the paths of the data nodes they access. Furthermore, due to the low complexity of coordination-service operations, there is no additional advantage in parallelizing requests that operate on the same data node.

Ordering of Event Notifications. Baldoni et al. [39] propose the introduction of vector-based timestamps to ensure the consistent delivery of notifications in a pub/sub system. AGORA clients for this purpose rely on the same timestamps that are used by servers to track causal dependencies between requests.

VIII. CONCLUSION

AGORA is a coordination service for applications with high performance demands that achieves scalability by dynamically partitioning the service state and executing each partition on a separate core. Despite the partitioning, AGORA guarantees FIFO client order and provides atomic operations on data nodes, independent of whether or not requests access multiple partitions. To support the same use cases as ZooKeeper, AGORA relies on an inter-partition protocol that tracks and respects causal dependencies between writes on different partitions. Our evaluation results show that AGORA provides a higher single-core efficiency than ZooKeeper due to minimizing the number of synchronization points within each partition. In addition, AGORA is able to scale with the number of cores on a server, resulting in low latencies for both reads and writes.

REFERENCES

[1] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proc. of ATC '10*, 2010.

[2] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. of OSDI '06*, 2006.

[3] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proc. of OSDI '06*, 2006.

[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. of NSDI '11*, 2011.

[5] T. Distler, C. Bahn, A. Bessani, F. Fischer, and F. Junqueira, "Extensible distributed coordination," in *Proc. of EuroSys '15*, 2015.

[6] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Veríssimo, "SCFS: A shared cloud-backed file system," in *Proc. of ATC '14*, 2014.

[7] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, "Detecting failures in distributed systems with the Falcon spy network," in *Proc. of SOSP '11*, 2011.

[8] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Hierarchical policies for software defined networks," in *Proc. of HotSDN '12*, 2012.

[9] A. Shakimov, H. Lim, R. Caceres, L. Cox, K. Li, D. Liu, and A. Varshavsky, "Vis-a-Vis: Privacy-preserving online social networking via virtual individual servers," in *Proc. of COMSNETS '11*, 2011.

[10] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer, "Modular composition of coordination services," in *Proc. of ATC '16*, 2016.

[11] A. N. Bessani, E. P. Alchieri, M. Correia, and J. Fraga, "DepSpace: A Byzantine fault-tolerant coordination service," in *Proc. of EuroSys '08*.

[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "UpRight cluster services," in *Proc. of SOSP '09*, 2009.

[13] T. Distler and R. Kapitza, "Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency," in *Proc. of EuroSys '11*, 2011.

[14] J. Behl, T. Distler, and R. Kapitza, "Consensus-oriented parallelization: How to earn your first million," in *Proc. of Middleware '15*, 2015.

[15] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems (2nd Edition)*. Addison-Wesley, 1993.

[16] L. Lamport, "The part-time parliament," *ACM Trans. on Computer Systems*, vol. 16, no. 2, 1998.

[17] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proc. of DSN '11*, 2011.

[18] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure," in *Proc. of OSDI '04*, 2004.

[19] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," in *Proc. of SOSP '07*, 2007.

[20] N. Santos and A. Schiper, "Achieving high-throughput state machine replication in multi-core systems," in *Proc. of ICDCS '13*, 2013.

[21] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, 2009.

[22] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, 1978.

[23] F. Mattern, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, 1989.

[24] M. Raynal, G. Thia-Kime, and M. Ahamad, "From serializable to causal transactions for collaborative applications," in *Proc. of EUROMICRO '97*, 1997.

[25] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[26] D. E. Bakken and R. D. Schlichting, "Supporting fault-tolerant parallel programming in Linda," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 3, 1995.

[27] B. Kalantari and A. Schiper, "Addressing the ZooKeeper synchronization inefficiency," in *Proc. of ICDCN '13*, 2013.

[28] ZooKeeper Observers, https://zookeeper.apache.org/doc/trunk/zookeeperObservers.html.

[29] R. Halalai, P. Sutra, E. Riviére, and P. Felber, "ZooFence: Principled service partitioning and application to the ZooKeeper coordination service," in *Proc. of SRDS '14*, 2014.

[30] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. of OSDI '02*, 2002.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. of SOSP '03*, 2003.

[32] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: The OceanStore prototype," in *Proc. of FAST '03*, 2003.

[33] M. Kapritsos and F. P. Junqueira, "Scalable agreement: Toward ordering as a service," in *Proc. of HotDep '10*, 2010.

[34] P. J. Marandi, M. Primi, and F. Pedone, "Multi-Ring Paxos," in *Proc. of DSN '12*, 2012.

[35] C. E. Bezerra, F. Pedone, and R. Van Renesse, "Scalable state-machine replication," in *Proc. of DSN '14*, 2014.

[36] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *Proc. of ICDCS '14*, 2014.

[37] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, "Rex: Replication at the speed of multi-core," in *Proc. of EuroSys '14*, 2014.

[38] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: Execute-verify replication for multi-core servers," in *Proc. of OSDI '12*, 2012.

[39] R. Baldoni, S. Bonomi, M. Platania, and L. Querzoni, "Efficient notification ordering for geo-distributed pub/sub systems," *IEEE Trans. on Computers*, vol. 64, no. 10, 2015.