Extensible Distributed Coordination

Tobias Distler[1] Christopher Bahn[1] Alysson Bessani[2] Frank Fischer[1] Flavio Junqueira[3]

[1]Friedrich–Alexander–Universität Erlangen–Nürnberg (FAU)

[2]Faculdade de Ciências/LaSIGE Universidade de Lisboa

[3]Microsoft Research Cambridge

Abstract

Most services inside a data center are distributed systems requiring coordination and synchronization in the form of primitives like distributed locks and message queues. We argue that extensibility is a crucial feature of the coordination infrastructures used in these systems. Without the ability to extend the functionality of coordination services, applications might end up using sub-optimal coordination algorithms, possibly leading to low performance. Adding extensibility, however, requires mechanisms that constrain extensions to be able to make reasonable security and performance guarantees. We propose a scheme that enables extensions to be introduced and removed dynamically in a secure way. To avoid performance overheads due to poorly designed extensions, it constrains the access of extensions to resources. Evaluation results for extensible versions of ZooKeeper and DepSpace show that it is possible to increase the throughput of a distributed queue by more than an order of magnitude (17x for ZooKeeper, 24x for DepSpace) while keeping the underlying coordination kernel small.

Categories and Subject Descriptors C.2.4 [*Computer Systems Organization*]: Distributed Systems; D.4.5 [*Operating Systems*]: Reliability

General Terms Design, Algorithms, Performance

Keywords Coordination Services, Extensibility, Distributed Algorithms, ZooKeeper, DepSpace

1. Introduction

Modern Web-scale services are complex and difficult to design and maintain. Part of such complexity comes from satisfying scalability and fault tolerance; the latter implies the use of sophisticated distributed protocols that are notoriously hard to implement correctly [17, 20, 58]. Coordination services have been proposed and applied to provide such functionality while exposing simpler interfaces [3, 5, 14, 17, 31].

In a nutshell, coordination services provide a consistent and highly-available data store with enough synchronization power [29] for client processes to execute fundamental tasks, such as mutual exclusion and leader election, and to store important system configuration. Such clients are often service processes deployed on clusters of hundreds or thousands of servers. Two key features of coordination services explain their success: (1) the fact that they provide a trust anchor for a much larger distributed system, using robust implementations of state machine replication protocols [35, 39, 46] to avoid any single point of failure; and (2) their accessible and limited interface, also called *coordination kernel* [31], which can be accessed through simple remote procedure calls (RPCs) that are intuitive even for programmers who are not experts in distributed computing [17].

Although simplifying usage, limited coordination kernels have a significant drawback: being confined to a particular set of primitives, more complex coordination tasks (e.g., distributed queues) have to be implemented as a combination of multiple RPCs, following coordination *recipes* [31], which is in general not optimal with respect to performance. For example, due to ZooKeeper [31] not providing a primitive to update the value of a data node based on its current value, such an operation must be realized using a read that is followed by a conditional write, which leads to poor performance under contention. Such deficiencies are an inherent property of limited coordination kernels (see §2). One way to address this issue would be to increase the size of kernels by adding new primitives. However, this would also mean to lose the benefits of limited interfaces discussed above.

In this paper, we therefore follow a different approach: to make coordination services extensible. For this purpose, we present a model that allows clients to dynamically and securely extend a coordination service by introducing small pieces of custom code, which are executed atomically at the server side (see §3). In addition, we devise a sandbox for constraining such extensions in a way that they do not degrade or disrupt the performance of the system (see §4).

© Distler et al. 2015. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*.
EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741954

In order to show the flexibility of our model, we implemented it in two rather different state-of-the-art coordination services: ZooKeeper [31] and DepSpace [14] (see §5). ZooKeeper provides a hierarchical namespace with sequencer capabilities; it uses the primary-backup approach to tolerate crashes [16], in which all state updates are processed by a master replica and then disseminated to the backups. In contrast, DepSpace implements an augmented tuple-space service with "test-and-set-like" operations; resilience against Byzantine faults is ensured using state machine replication [13, 52], in which all operations are ordered by a Byzantine fault-tolerant protocol and executed deterministically by all correct replicas. The systems have quite different targets: ZooKeeper aims for read scalability (i.e., adding replicas increases the read throughput of the system) while DepSpace is designed for untrusted environments, thus implementing several access control mechanisms to cope with malicious clients.

We evaluate our approach by reimplementing several coordination recipes in our extensible versions of ZooKeeper and DepSpace (see §6). The results show performance gains of more than an order of magnitude for crucial coordination mechanisms such as shared counters and distributed queues when using extensions, while the underlying coordination kernel remains small. Perhaps equally important, extensibility enables new use cases for coordination services, for example, as a basis for load balancing in software-defined networks or as storage for file-system metadata (see §7).

Although extensions have been applied to many kinds of systems both for improving performance and enriching their features (e.g., [9, 11, 42, 50]), to the best of our knowledge, no previous work discussed the extensibility of coordination services and its implications (see §8). A key insight of this work is that extensions bring performance benefits for clients as well as servers: with extensions executing multiple operations atomically, most coordination tasks require clients to make only a single RPC to the server. As a result of clients issuing fewer RPCs and extensions being small pieces of code with negligible effect on processing load, servers can provide better throughput.

In summary, the key contributions of this work are:

1. A model for extensible coordination services and its implementation in two systems, ZooKeeper and DepSpace.
2. A sandboxing mechanism for constraining extensions that protects servers against misbehaving extensions.
3. A set of extension-based coordination recipes providing support for a shared counter, a distributed queue, a distributed barrier, and leader election, which show significantly better performance than the state of the art.

All source code for the extensible versions of ZooKeeper and DepSpace together with the extension-based recipes used in the paper are publicly available via our project website at `http://www4.cs.fau.de/Research/EDC/`.

2. Background and Problem Statement

In recent years, several coordination services have been proposed [6, 14, 17, 31, 44] to facilitate the implementation of coordination tasks such as locking, leader election, and message ordering. These services are also often used for storing configuration [17, 31] or metadata [12]. Each of these services provides different programming interfaces, also called *coordination kernels*, that applications can use for running coordination tasks, following some recipes [31].

Background. Independently from the specifics of the supported coordination kernel and underlying implementation details, coordination services implement three main features:

- *Highly-available small storage*: The coordination service is often an anchor of trust in a much bigger distributed system and, amongst other things, is responsible for reliably managing small chunks of important data. Thus, it must be fault tolerant, which is usually achieved based on state machine replication [52].
- *Interface with synchronization power*: Coordination tasks usually can be reduced to the problem of consensus [48]. To solve this problem, simple read/write (or put/get) operations are not enough [43]; more powerful primitives are necessary. Some older systems rely on the lock/lease abstraction [17, 44], which is not wait-free since a client that fails with the lock can block other clients, at least for some time, until its lease expires. More recent systems provide transactions [6] or even wait-free operations [14, 31] based on primitives with infinite synchronization power [30].
- *Client failure detection*: For implementing important coordination tasks such as leader election and fault-tolerant task assignment, it is essential that clients learn about the failure of other clients. Some systems explicitly provide such failure detection (e.g., by maintaining client sessions and notifying registered clients when such sessions terminate [5, 31]), while others enable the expiration of objects, which can be interpreted as failures of the clients responsible for renewing the objects' time to live [3, 14].

Table 1 summarizes the main characteristics of several coordination services proposed in recent years. The table shows that these services provide different abstractions with regard to their data model and synchronization power.

Problem Statement. One important limitation of these systems is that their coordination kernels suit some coordination tasks better than others. For example, implementing mutual exclusion with Chubby is trivial since it already provides lock objects. In contrast, the best practice for acquiring a lock in ZooKeeper is to create a sequential node and to set a watch to the adjacent node with a smaller sequence number. This corresponds to three RPCs to the service while Chubby requires just one. On the other hand, adding an element to a queue can be performed very efficiently in ZooKeeper but

System	Data Model	Sync. Primitive	Wait-free
Boxwood [44]	Key-Value store	Locks	No
Chubby [17]	(Small) File system	Locks	No
Sinfonia [6]	Key-Value store	Microtransactions	Yes
DepSpace [14]	Tuple space	cas/replace ops	Yes
ZooKeeper [31]	Hierar. of data nodes	Sequencers	Yes
etcd [3]	Hierar. of data nodes	Sequen./Atomic ops	Yes
LogCabin [5]	Hierar. of data nodes	Conditions	Yes

Table 1. Coordination services and their characteristics.

includes multiple operations on shared locks in Chubby. This kind of tradeoff is inherent to the choice of synchronization primitives. As far as we know there is no "silver bullet" primitive that would allow the implementation of all coordination tasks in an optimal way, that is, using a single RPC.

Possible Solutions. A possible solution would be to implement and provide a rich and extensive API with all practically-relevant primitives. However, this approach presents two fundamental limitations. First, it is hard to define the set of operations such API must provide, probably requiring regular changes to account for new uses of the system. Second, this would create a huge coordination kernel, making it difficult for programmers to figure out how to correctly implement their tasks. A good coordination kernel is small, elegant, expressive, and stable, providing underpinnings for normal programmers to create coordination libraries and custom coordination tasks.

Notice that some systems provide extensive libraries of coordination recipes, which are implemented based on their respective coordination kernel (e.g., Apache Curator [2] for Zookeeper). Despite the ease of use of such libraries, the performance of the coordination tasks are still constrained by the underlying coordination kernel, as we show in §6.

An entirely different approach is to build custom services tailored to the specific needs of an application (e.g., as done in [33]). This way, all coordination tasks can be natively supported via a single RPC to the service. The basic idea is to implement a highly-available service with the required interface and features from ground up, based on a consensus protocol (e.g., Paxos [39], RAFT [46], Zab [35]) or a replication library (e.g., BFT-SMaRt [13], JZab [4]). The first approach is time-consuming since reimplementing a replicated state machine is quite complex [20]. The second approach, although much simpler than the first, can be extremely error prone, since ensuring determinism and predictable performance in replicated state machines is inherently difficult.

Our Approach. We advocate the use of *extensions at the server side* for making (fixed-kernel) coordination services as efficient as possible for any coordination task. The objective is to get the best features of fixed-kernel (simple and expressive programming model) and custom (optimal coordination tasks) coordination services.

3. A Model for Extensible Coordination

In this section, we introduce a conceptual model to discipline the extensibility of current and future coordination services.

3.1 System Model

A coordination service CS is a stateful service accessed through a set of operations op_1, ..., op_n that read or modify its state S, which consists of a set of *data objects*. These operations define the API of the coordination kernel. If an operation op can change the state S (depending on its parameters and the actual state) it is called an *update*, otherwise it is called a *read*. The modification of the state by an update triggers an *event* v. An extension $e = \langle P, A \rangle$ contains a pattern P and a sequence of operations A that are executed *atomically*. The extension is triggered when an operation or an event matches the pattern P defined for the extension.

3.2 Requirements

Our extension mechanism must satisfy these requirements:

- *No changes to the API*: A fundamental principle of our extension model is to not change the coordination kernel.
- *Security*: An extension should run with the privileges of the client that has invoked it. To prevent attacks, an extension should only be executed if a client has acknowledged the use of the extension, either by registering the extension itself or by sending an explicit one-time request.
- *Bounded resource consumption*: Extensions should consume a bounded amount of memory and CPU in order to not degrade or disrupt system performance as well as to support performance predictability.
- *Determinism*: To ensure consistency, extensions in actively-replicated systems have to be deterministic: given a state and an operation, applying an extension must always generate the same reply and resulting state.

3.3 Types of Extensions

We distinguish between two main categories of extensions. An *operation extension* is invoked as the direct result of a client issuing a request to the coordination service. In contrast, an *event extension* is executed in reaction to the state of the coordination service being changed. Extensions of both types may be combined, as shown in §6.1.4.

Operation Extensions. Operation extensions allow clients to invoke multiple operations on the coordination service by using only a single RPC. In the typical use-case scenario, operations called by the same extension are dependent on each other (e.g., because they access and/or modify the same data objects), and a client combines them in order to exploit atomic execution. Operation extensions offer one key benefit: instead of shipping the data to be processed to the client, such extensions allow complex operations to be performed at the server side, directly on the data objects.

```
1 /* Operation and event subscriptions */
2 OPERATIONSUBSCRIPTION[] getOpsSubscriptions();
3 EVENTSUBSCRIPTION[] getEventSubscriptions();

5 /* Extension execution */
6 void handleOperation(REQUEST request);
7 void handleEvent(EVENT event);
```

Figure 1. Basic extension interface.

Execution model: If a client request matches an operation extension, the call will be made to this extension, *instead of being executed normally* by the system. If a request matches multiple extensions, only the last registered will be executed.

Event Extensions. Event extensions allow clients to customize the handling of events including, for example, the creation, modification, or deletion of a data object. In contrast to an operation extension, which replaces a normal operation, an event extension is triggered after an operation is executed, in response to a change in the service state.

Execution model: Once the system state has changed, all event extensions registered with the particular change will be triggered, one after another, in the order of their registration.

3.4 Basic Extension Interface

Extensions must implement the basic interface presented in Figure 1 in order to be registered with an extensible coordination service. The interface requires an extension to provide a set of *subscriptions* indicating the operations and events for which the extension is to be executed (L. 2–3). In general, a subscription may contain any criteria that can be matched to the characteristics of an operation or event including, for example, operation and event types, ids of data objects accessed, and/or contents of operation parameters.

As described in §3.7, if a request matches a subscription of an extension, the extension is invoked by a call to its `handleOperation` method (L. 6), allowing it to perform complex operations on behalf of the original request. In the same way, an extension is able to provide a custom reaction to events by implementing the `handleEvent` method (L. 7).

Based on this interface, particular extensible coordination services may use derived interfaces for their extensions that reflect the specifics of their respective APIs. As a result, extension programmers may customize the behavior of specific operations and events directly, as illustrated by the examples in §6, instead of implementing the generic `handleOperation` and `handleEvent` methods.

3.5 Extension Manager

Making a coordination service extensible requires additional functionality at the server side, which is provided by a component we refer to as the *extension manager*. A replica of the extension manager is integrated with each replica of an extensible coordination service. The extension manager's main tasks include handling the lifecycle of extensions (i.e., registration and deregistration – see §3.6) as well as the execution of extensions (see §3.7). For this purpose, the extension manager must be able to intercept and (when necessary) suppress or modify the requests received by and events occurring on its local replica. In §5.1 and §5.2 we present details on how this can be ensured for different coordination services.

Initially, only a single built-in extension is registered with the service: it allows clients to communicate with the extension manager by invoking standard operations on a special data object (e.g., `/em`) that represents the extension manager.

3.6 Extension Registration and Deregistration

Meeting the requirement of §3.2, registration and deregistration of extensions are performed using the standard API of the coordination service, without adding new operations to the service interface: to register an extension, a client issues a standard create operation for the extension manager's data object, passing the extension code as well as additional relevant information (e.g., the extension name) as data.

When the extension manager intercepts the call, it verifies, compiles, and instantiates the extension and retrieves the extension's operation and event subscriptions. Furthermore, the extension manager adds a new data object (e.g., `/em/ext` for an extension named `ext`) to the coordination service state, which from then on acts as a surrogate of the extension. Note that this approach has two main benefits. First, a client is able to deregister an extension by issuing a standard delete operation for the extension's data object to the service. Second, with data objects being protected by the general fault-tolerance mechanisms of the coordination service, storing registration information inside a data object frees the extension manager from providing its own mechanisms, as further discussed in §3.8.

For security reasons, extensions by default can only be triggered by the client that has registered them. However, if a client wants to use an extension registered by another client, it can do this by acknowledging the extension once through a call to the extension's data object.

3.7 Extension Execution

While the system is running, the extension manager constantly monitors incoming requests as well as occurring events, trying to match them to subscriptions of extensions registered. This is practical as, following our requirements (see §3.2), for each operation/event, the extension manager only needs to take extensions into account that have been acknowledged by a particular client and is able to ignore the subscriptions of all unacknowledged extensions.

If an operation/event matches none of the subscriptions, the extension manager acts as a relay and forwards it to the corresponding handlers. Otherwise, the extension manager suppresses the operation/event for the clients affected and executes the matching extension(s). For this purpose, the

extension manager first creates a sandbox environment for the extension to run in (see §4.1.2) and then invokes the corresponding methods of the extension interface (Figure 1).

Being processed instead of a regular operation, an operation extension returns a result as soon as execution is complete. In such case, the extension manager forwards the result to the client whose request invoked the extension.

3.8 Fault Tolerance

State-of-the-art coordination services such as the ones described in §2 provide resilience against faults and, for example, offer mechanisms that allow replicas to recover from crashes and/or new replicas to join the system. Managing the set of registered extensions, the extension manager is a stateful component and therefore has to be protected against data loss. We address this problem by maintaining the state of the extension manager in data objects as part of the regular coordination-service state. This way, the state of the extension manager is protected by the fault-tolerance mechanisms already in place (which are usually a combination of replication, persistent logging, and state-transfer protocols), thereby greatly reducing the implementation overhead for making a coordination service extensible.

As discussed in §3.6, when registering an extension, the extension manager creates a new data object in which it stores all information necessary to recover the extension after a fault. In particular, this includes the extension's name and code as well as the id of the client that has registered the extension. In addition, the extension manager updates an index data object containing a set of pointers to the data objects of all extensions currently registered. As a consequence, when a coordination-service replica joins the system or recovers after a fault, its local extension manager only needs to query the index data object in order to be able to find and then (re)load all registered extensions.

Using the same approach, an extensible coordination service can also provide support for stateful extensions: if an extension manages its internal state in coordination-service data objects (instead of local variables), existing fault-tolerance mechanisms ensure that the state is protected against data loss and readily available after recovery.

4. Limiting Extensions

Having extensions deployed in coordination services raises issues related to performance and security. In this section, we present an execution model for extensions that protects the service from inappropriate extensions that contain programming errors, are deployed with malicious intent, or have been implemented based on a bad understanding of the feature.

4.1 Basic Approach

Some systems define guidelines for extensions design and let programmers/administrators be responsible for any problem caused by the extensions deployed. This is the case, for example, with portable interceptors in CORBA middleware [9] or filters in Servlet containers [32]. Although this approach is valid and quite reasonable in a closed environment with well-informed developers, the experience with drivers and modules of operating systems has shown that extensibility can bring a lot of problems for a widely-used system [27]. Therefore, we address the problem with an approach that puts the focus on the system protecting itself against harmful extensions. It consists of two parts: first, prior to instantiating an extension, the server invokes a verification procedure that analyzes the extension code to ensure that only extensions performing non-critical operations are registered; second, extensions are executed in a sandbox to be able to monitor (and if necessary constrain) their behavior at runtime.

4.1.1 Extension Verification

Before compiling an extension, the extension manager verifies that the extension does not exceed a certain size and that it is compliant with a number of constraints set up to protect the system from rogue extensions. Note that the verification procedure does not need to decide whether an extension is innocuous or harmful. Instead, an extension is responsible for being compliant by only using a white-listed set of APIs and language constructs. If this verification fails the extension is rejected and the registration aborts immediately.

The rationale behind using a white-list approach is that extensions are not intended to be means to run arbitrary code at a server. Instead, as shown by the examples in §6, an extension in essence is a collection of coordination-service API calls linked by relatively simple glue code. Consequently, in addition to the service interface, the white list only contains basic math, boolean, and string operations. For actively-replicated systems, the white list is required to only contain deterministic operations. This is necessary to ensure that replicas remain consistent when executing an extension [52]. In contrast, in passively-replicated systems the fact that only one replica executes an extension leaves room for adding nondeterministic operations to the white list.

To bound the execution time of extensions, they must not perform recursive calls and are generally restricted to using non-loop control structures. An exception to this rule is that we allow extensions to iterate through data structures of constant size (e.g., using a for-each loop in Java to access the elements of an array or a list), as such an operation terminates by definition.

4.1.2 Extension Sandbox

By complying with the constraints discussed before, we ensure that extensions do not invoke blocking operations, have no access to the file system, cannot open and/or use network sockets, and are not able to create, pause, or terminate threads. Although this greatly limits the spectrum of problems an extension can cause, our verification procedure does not prove extensions to be free of errors. Therefore, each extension is executed inside a sandbox that prevents extension crashes from affecting the rest of the service.

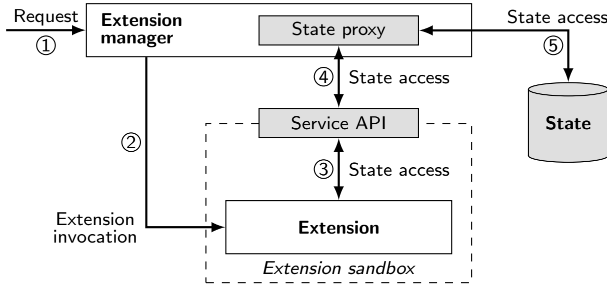

Figure 2. Sandbox environment for extensions.

Besides handling crashes, the sandbox is also responsible for monitoring accesses of extensions to the service state. As illustrated in Figure 2, after being invoked to handle an operation (①, ②) or event, an extension does not have direct access to the state of the service. Instead, all state operations performed by an extension are handled by the extension manager (③ – ⑤), which provides a proxy for extensions to access the coordination service state. The state proxy implements an interface similar to the one of the coordination service, which brings three advantages. First, it reduces the implementation overhead for extensions as existing client code can be reused. Second, it simplifies the task of making a coordination service extensible as no additional operations need to be integrated into the service API. Third, limiting extensions to the same operations clients are able to invoke regulates the interaction with the service and potentially reduces the damage that bad extensions can do.

Serving as a proxy for state operations enables the extension manager to implement access-control mechanisms ensuring that a client cannot gain privileges by invoking an extension. Furthermore, the extension manager may apply policies aimed at bounding the resource consumption of extensions, for example, by enforcing an upper limit on the number of data objects an extension is allowed to create or the maximum CPU time an extension can use per invocation.

4.2 Discussion

The focus of our approach lies on limiting the effects an extension can have on the performance of the system. This is achieved due to the following reasons: first, the verification of an extension is done at registration time, resulting in no verification overhead during execution; second, the limited size of an extension ensures that the verification process is fast; and third, the white list of methods and constructs an extension is allowed to use guarantees that an extension is not able to degrade or disrupt system performance.

Our experience with writing extensions as well as the examples in §6 show that the constraints enforced upon extensions do not hinder their utilization. In particular, the limitation to for-each loops turned out to not pose a problem as loops are usually only required for iterating through operation parameters or entities stored in the coordination service. Nevertheless, there might be scenarios in which such constraints are too restrictive. (Up to this point, we did not find any.) For these situations, we open the possibility for the extension verification being disabled. Alternatively, it is possible to (statically) extend the interface of the sandbox proxy with additional helper methods that safely implement the required functionality.

5. Implementations

To show the effectiveness and practicality of extensions, we have implemented two prototypes: EXTENSIBLE ZOOKEEPER, an extensible variant of the crash-tolerant ZooKeeper [31], and EXTENSIBLE DEPSPACE, an extensible variant of DepSpace [14], a coordination service that provides resilience against Byzantine faults. The source code of both EXTENSIBLE ZOOKEEPER and EXTENSIBLE DEPSPACE is publicly available via the project website at `http://www4.cs.fau.de/Research/EDC/`.

5.1 EXTENSIBLE ZOOKEEPER

Below, we give an overview of the original ZooKeeper implementation and discuss our modifications for EXTENSIBLE ZOOKEEPER (EZK). Like ZooKeeper (see §2), EZK manages information using wait-free data objects that are accessible via a hierarchical namespace.

5.1.1 ZooKeeper Architecture

ZooKeeper clients access the coordination service using a library that handles all communication with the server side. In order to provide resilience against up to f crashes, the server side comprises $2f+1$ replicas. Each client is connected to a single replica, to which the client issues all of its requests and from which it also receives the corresponding replies. Different clients usually communicate with different replicas to balance load. If a client's replica crashes, the client establishes a connection to another replica.

One of the replicas in a ZooKeeper deployment serves as primary while the others are backups [16]. The primary is responsible for assigning unique sequence numbers to incoming updates[1] and for translating such requests into state transactions, which are then processed by all replicas, including the primary itself. In contrast, a read is only executed by a single replica: the one a client is connected to.

As shown in Figure 3, a replica in ZooKeeper is implemented as a chain of request processors handling different tasks including, for example, preprocessing, ordering, persistent logging, and execution of requests. The composition of the request-processor chain reflects the specific responsibilities of a replica and therefore differs between primary and backups. In particular, the primary participates as proposer in the protocol responsible for the reliable distribution of state transactions [35], while backups act as learners.

[1] If a backup receives an update, it forwards the request to the primary.

ZooKeeper allows a client to be informed about certain state-related events (e.g., the deletion of a data object) by registering *watches*. If an event that is monitored by a watch occurs, a replica sends a notification to the associated client.

5.1.2 Making ZooKeeper Extensible

Below, we present the most important modifications and additions we made to ZooKeeper to implement EZK.

Client Library. EZK introduces two methods for registering and deregistering extensions into the ZooKeeper client library. Internally, these methods are mapped to standard ZooKeeper update operations (see §3.6) creating and deleting sub-objects of the data object representing the extension manager, for example, `/em/ext` for an extension `ext`.

Operation Extensions. In order to customize the behavior of the coordination service via extensions, the extension manager must be able to monitor and (when necessary) control the handling of operations (see §3.5). As shown in Figure 3, we meet this requirement in EZK by invoking the extension manager at the preprocessor stage of ZooKeeper's request-processor chain. This allows the extension manager to intercept requests issued by clients and to redirect them to extensions. In addition, we modify the final processor stage to enable a replica to control the results it sends to clients.

If a request entering the preprocessor stage matches the subscription of an extension, the extension manager invokes the corresponding extension by handing over the request. While execution is in progress, the extension manager (via its state-proxy subcomponent, see Figure 2) records all state modifications performed by the extension. This way, once execution is complete, the extension manager is able to construct a multi-transaction (i.e., a batch of state updates that is processed atomically) reflecting all state changes caused by the extension. After its creation, the multi-transaction is treated like any regular state transaction and passed along the chain of request processors. Note that at this point the implementation of EZK has been greatly simplified by the fact that ZooKeeper natively supports the batching of transactions. As a result, making ZooKeeper extensible does not require modifications to the stage responsible for persistent logging as well as the stages involved in the replication protocol, to which extensions are transparent.

EZK allows operation extensions to customize the results they return to clients. For this purpose, the extension manager at the leader replica piggybacks the value produced by an extension during execution at the preprocessor stage on the corresponding multi-transaction. When such transaction reaches the final processor stage, the replica connected to the client that has issued the operation request detaches the result and includes it in the reply.

Event Extensions. For EZK, we modify ZooKeeper's event-handling mechanism to invoke the extension manager each time a watch triggers. In consequence, the extension

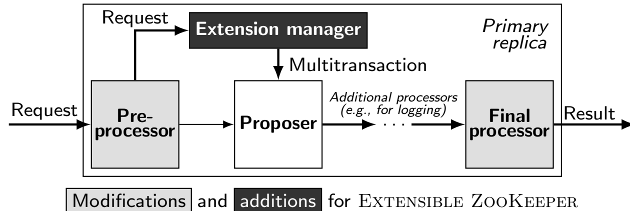

Figure 3. Basic architecture (simplified) of a primary replica in ZooKeeper and EXTENSIBLE ZOOKEEPER.

manager is able to execute event extensions with matching subscriptions. If at least one of such extensions exists for a particular watch, the original notification to the client will be suppressed. However, an event extension may still choose to send a notification of its own.

5.2 EXTENSIBLE DEPSPACE

In the following, we present the architecture of DepSpace and our additions for EXTENSIBLE DEPSPACE (EDS). Both systems rely on the tuple-space abstraction [26] to provide coordination services for clients.

5.2.1 DepSpace Architecture

DepSpace provides resilience against f Byzantine faults and therefore requires a minimum of $3f+1$ replicas, which are kept consistent using the BFT-SMaRt [13] library that handles all tasks related to state machine replication (e.g., request ordering and state transfer). As shown in Figure 4, the implementation of a DepSpace server replica comprises multiple layers with different responsibilities providing, for example, means to control the access of clients to tuples. Being an actively-replicated system, all replicas in DepSpace process all requests, starting at the bottom layer.

As in ZooKeeper, DepSpace clients use a distinct library to invoke operations at the service. However, DepSpace does not expose a notification-service interface to clients that could be used to get information about events such as the creation of an object. Instead, a DepSpace client for this purpose performs a read operation that, if the corresponding data object does not exist, blocks until the object is created. In consequence, the client only needs to wait for its read to complete and does not have to actively poll the service.

5.2.2 Making DepSpace Extensible

Below, we present an overview of our measures to derive EDS from DepSpace.

Client Library. Similar to EZK, we add convenience methods for the registration and deregistration of extensions to DepSpace's client library. In EDS, calls to these methods are translated to DepSpace operations creating and deleting tuples of a tuple space that is dedicated to the extension manager and not accessible via regular operations.

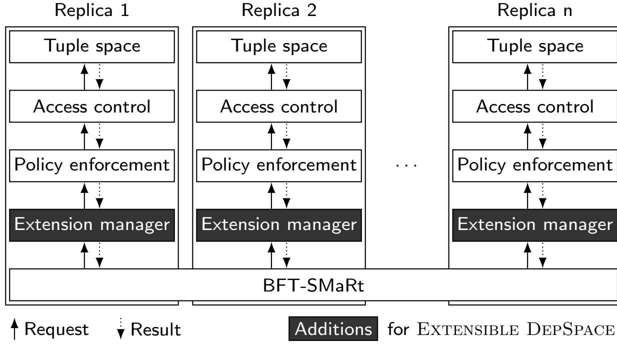

Figure 4. Overview of the replicated DepSpace server architecture and the additions for EXTENSIBLE DEPSPACE.

Operation Extensions. As shown in Figure 4, at the server side, we implement EDS by introducing a new *extension layer* containing the extension manager at the bottom of the stack. This location has two benefits. First, all incoming client requests have to pass the extension layer, allowing the extension manager to redirect them to operation extensions. Second, the extension manager does not need to provide additional access-control mechanisms for operations invoked by extensions as this task is performed by upper layers.

Event Extensions. As discussed in §5.2.1, an event in DepSpace occurs when a blocked operation unblocks. At this point, EDS hands over control to the extension manager, which is then able to execute matching event extensions. In the following, an extension may decide to block the operation again, which results in no reply being sent to the client. A blocked call only returns if the associated event occurs and all the extensions triggered by it let it proceed.

6. Evaluation

In this section, we evaluate EZK and EDS for several use cases to analyze the impact of extensions on clients that invoke them (see §6.1) as well as the side effects on clients that perform regular read and write operations on the coordination service (see §6.2). All experiments are conducted on a cluster of 4-core servers (2.3 GHz, 8 GB RAM) that are connected with switched Gigabit Ethernet. Clients are executed on separate 12-core machines (2.4 GHz, 24 GB RAM) and run a stress test by continuously invoking the operation under test at the coordination service; as a consequence, each client has at most one request pending at a time. All systems evaluated are configured to tolerate a single faulty server, which means that three servers are used for EZK and ZooKeeper and four servers for EDS and DepSpace. Each data point in the graphs represents the average of five runs.

6.1 Coordination Recipes

In the following, we evaluate the usefulness and efficiency of extensions based on four recipes included in the Apache Curator library [2] that provide support for a shared counter, a distributed queue, a distributed barrier, and leader election, respectively. To show their generality, we present the recipes using an abstract coordination-service API, which can be mapped to both ZooKeeper and DepSpace (see Table 2). In all the examples discussed in this section, a client's remote reference to the coordination service is denoted as `remote`; in contrast, an extension executed at the server side can access the service via a local reference `local`.

6.1.1 Shared Counter

Shared counters are used for different purposes in distributed systems including, amongst other things, the implementation of semaphores, the generation of unique ids and the collection of statistics. Figure 5 shows how the `increment` operation of such a counter can be realized based on the operations provided by a traditional coordination service. The current value of the counter is managed in an object `/ctr`. To increment the counter, a client first retrieves the current value (T4), then locally adds one, and finally updates the counter object (T5). As multiple clients may perform the same operation concurrently, to ensure atomicity, the counter object may only be updated if its value has remained unchanged since a client's last read. If this is not the case, the entire `increment` operation needs to be retried (T2, T6).

Utilizing the atomicity of extensions, the client implementation for an extensible coordination service in contrast only comprises a single remote call (C2) to a data object that triggers the extension (i.e., `/ctr-increment`). If invoked, the extension performs similar steps to increment the counter as the standard client implementation (E2–E4).

Traditional Client Implementation

```
T1 int increment() {
T2   while(true) {
T3     /* Read current counter value and try to write back new value. */
T4     int c = remote.read("/ctr");
T5     boolean success = remote.cas("/ctr", c, c+1);
T6     if(success) return c+1;
T7   }
T8 }
```

Extension-based Client Implementation

```
C1 int increment() {
C2   return remote.read("/ctr-increment");
C3 }
```

Extension Implementation

```
E1 OBJECT read(OBJECTID oid) {
E2   int c = local.read("/ctr");
E3   local.update("/ctr", c+1);
E4   return c+1;
E5 }
```

Figure 5. Implementation of the `increment` operation of a shared counter without (top) and with (bottom) extension.

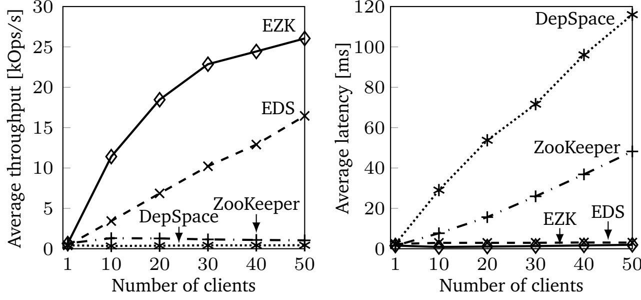

Figure 6. Evaluation results for the shared-counter recipe.

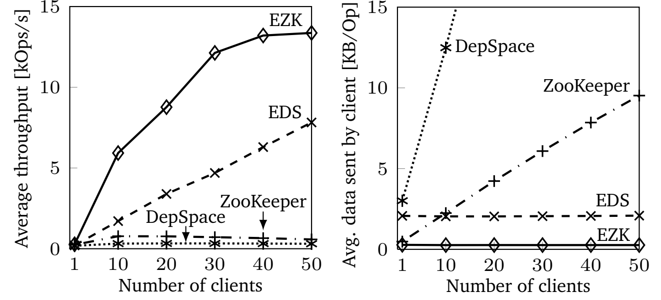

Figure 8. Evaluation results for the distributed queue.

The results of our experiments presented in Figure 6 show that the ZooKeeper and DepSpace implementations of the shared counter reach modest throughput values. This happens due to the fact that increasing the number of clients increases also the number of tries required for executing a counter increment. EZK and EDS, on the other hand, are less susceptible to contention and therefore able to achieve considerably higher throughputs (e.g., an increase of 20× for EZK over ZooKeeper). This is possible as clients in both extension-based systems only need to perform a single remote call to successfully increment the counter, resulting in low latencies of about 2 milliseconds (EZK) and 3 milliseconds (EDS) for 50 concurrent clients, respectively.

6.1.2 Distributed Queue

Distributed queues play an important role in the exchange of information between producer and consumer processes running on different hosts. Figure 7 presents the basic recipe to implement such a data structure using a coordination service. To add an element to the end of a queue, a client creates a new sub-object of the queue's main object (T3). Removing the head element from the queue in contrast takes multiple steps. First, the client learns all elements that are currently in the queue (T9) and sorts them by creation time (T10). After this step, the client is able to remove the head element from the queue (i.e., the element with the lowest creation timestamp) by deleting the corresponding data object (T14). In case of `remove` being invoked concurrently by different clients, only the client that successfully performs the deletion is allowed to return the result (T15). All other clients first try to remove subsequent elements (T13) before starting the entire operation all over again (T7).

In our extension-based implementation, adding an element to the queue is identical to the non-extension variant (C2). However, removing the head element only requires a single remote call by the client (C6), which leads to the extension deleting the head element atomically (E2–E4) and returning the element's data (E5).

To evaluate the different queue implementations, we run an experiment in which each client repeatedly first adds a new element to the (initially empty) queue and then, in a separate call, removes the current head element. As a result, due to calls of different clients interleaving, the size of the queue may vary between zero and the number of clients. However, as each `remove` is preceded by an `add`, it is ensured that at the time of the `remove` call the queue contains at least one element. In order to be able to properly assess the coordination overhead involved, elements in our experiment carry an empty data payload. Consequently, the operation costs represent the minimum amount of data required to send an element through the queue in each of the scenarios evaluated. Notice that clients need to send much more data with DepSpace (resp. EDS) than with ZooKeeper (resp. EZK), mainly because the Byzantine fault-tolerant protocol of the former requires requests to be sent to all service replicas.

Figure 8 shows that the ZooKeeper and DepSpace implementations of the queue are subject to the same problem as the corresponding shared counter variants in §6.1.1. More specifically, when multiple clients remove elements from

Method	Description	ZooKeeper	DepSpace
`create(o)`	Creates a data object o.	`create(o)`	`out(o)`
`delete(o)`	Deletes data object o.	`delete(o, ANY_VERSION)`	`inp(o)`
`read(o)`	Reads the content of data object o.	`getData(o)`	`rdp(o)`
`update(o, c)`	Sets the content of data object o to c.	`setData(o, c, ANY_VERSION)`	`replace(o, ANY, nc)`
`cas(o, cc, nc)`	`update` with compare-and-swap semantics: Only sets the content of object o to nc if the current content is cc.	`int v` = object version observed by last `read(o)` `setData(o, nc, v)`	`replace(o, cc, nc)`
`subObjects(o)`	Reads the contents of all sub-objects of data object o.	1. OBJECTID[] `oids = getChildren(o)` 2.* For each `oid` in `oids`: `getData(oid)` * Step 2 can be omitted if only the ids of sub-objects are of interest.	`rdAll(<o, SUB_ANY>)`
`block(o)`	Waits until data object o is created.	1. Set `exists` watch on data object o. 2. Unblock on receiving watch-event notification.	`rd(o)`
`monitor(x, o)`	Creates object o and instructs the service to monitor client x. If client x terminates or fails, the service deletes object o.	1. Client x creates object o as an ephemeral node. 2. ZooKeeper deletes o if client x's session ends.	1. Client x creates o as a lease tuple. 2. o is deleted if x fails to renew it.

Table 2. Overview of the coordination-service methods used in §6 and their equivalences in ZooKeeper and DepSpace.

Traditional Client Implementation

```
T1  void add(ElementID eid, byte[] data) {
T2    /* Create object storing the element's data. */
T3    remote.create("/queue/" + eid, data);
T4  }
T6  byte[] remove() {
T7    while(true) {
T8      /* Learn queue elements. */
T9      Object[] objs = remote.subObjects("/queue/");
T10     Sort objs ascending by creation timestamp;
T12     /* Try to remove the current head of the queue. */
T13     For each obj in objs {
T14       boolean success = remote.delete(obj);
T15       if(success) return obj.data;
T16 } } }
```

Extension-based Client Implementation

```
C1  void add(ElementID eid, byte[] data) {
C2    remote.create("/queue/" + eid, data);
C3  }
C5  byte[] remove() {
C6    return remote.read("/queue/head").data;
C7  }
```

Extension Implementation

```
E1  Object read(ObjectID oid) {
E2    Object[] objs = local.subObjects("/queue/");
E3    Object head = object from objs with lowest creation timestamp;
E4    local.delete(head);
E5    return head.data;
E6  }
```

Figure 7. Implementation of a distributed queue without (left) and with (right) extension.

Traditional Client Implementation

```
T1  void enter() {
T2    /* Register this client at the barrier using the client's id. */
T3    remote.create("/barrier/" + this.id);
T5    /* Check whether all clients have entered the barrier. */
T6    Object[] objs = remote.subObjects("/barrier/");
T7    if(|objs| < barrier threshold) {
T8      /* Block until /ready object is created. */
T9      remote.block("/ready");
T10   } else {
T11     /* Create /ready object. */
T12     remote.create("/ready");
T13   }
T14 }
```

Extension-based Client Implementation

```
C1  void enter() {
C2    remote.block("/ready/" + this.id);
C3  }
```

Extension Implementation

```
E1  void block(ObjectID oid) {
E2    local.create("/barrier/" + client id encoded in oid);
E3    Object[] objs = local.subObjects("/barrier/");
E4    if(|objs| < barrier threshold) {
E5      local.block("/ready");
E6    } else {
E7      local.create("/ready");
E8  } }
```

Figure 9. Implementation of the `enter` operation of a distributed barrier without (left) and with (right) extension.

Traditional Client Implementation

```
T1  void becomeLeader() {
T2    /* Instruct the coordination service to monitor this client. */
T3    remote.monitor(this.id, "/leader/" + this.id);
T5    /* Return as soon as this client has become leader. */
T6    if(amILeader()) return;
T7    Wait for I_AM_LEADER signal;
T8  }
T10 void objectDeletionEvent() { /* Event handler */
T11   if(amILeader()) Send I_AM_LEADER signal;
T12 }
T14 boolean amILeader() { /* Local helper method */
T15   Object[] objs = remote.subObjects("/leader/");
T16   Sort objs ascending by creation timestamp;
T17   ClientID leaderID = client id encoded in objs[0];
T18   return (leaderID == this.id);
T19 }
```

Extension-based Client Implementation

```
C1  void becomeLeader() {
C2    remote.block("/leader/" + this.id);
C3  }
```

Extension Implementation

```
E1  void block(ObjectID oid) {
E2    ClientID cid = client id encoded in oid;
E3    local.monitor(cid, "/clients/" + cid);
E4    local.block(oid);
E5  }
E7  void objectDeletionEvent() { /* Event handler */
E8    Object[] objs = local.subObjects("/clients/");
E9    Object ldr = object from objs with lowest creation timestamp;
E10   ClientID newLeaderID = client id encoded in ldr;
E11   local.create("/leader/" + newLeaderID);
E12 }
```

Figure 11. Implementation of the leader-election recipe without (left) and with (right) extension.

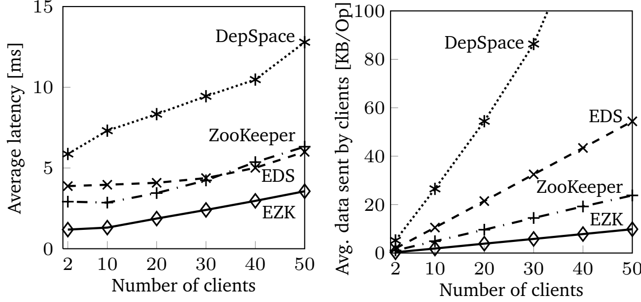

Figure 10. Evaluation results for the distributed barrier.

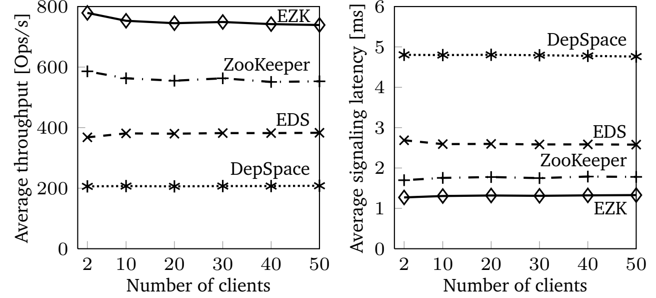

Figure 12. Evaluation results for the leader-election recipe.

the queue concurrently, only one of them is successful. This means that with more clients accessing the queue, the costs per successful operation (in terms of data to be transmitted by a client) rise due to the increasing number of retries; the `add` operation, on the other hand, is not affected by contention and therefore still always succeeds at the first try. In contrast, the costs of both queue operations in the extension-based variants are independent of the number of concurrent accesses, allowing EZK and EDS to outperform ZooKeeper and DepSpace by a factor of 17 and 24, respectively.

6.1.3 Distributed Barrier

Distributed barriers are abstractions used to introduce synchronization points for a group of processes executed on different hosts. Each process that enters the barrier early is blocked until all other processes have also arrived at the barrier. Figure 9 shows how to implement the `enter` operation based on a traditional coordination service. First, a client registers at the barrier by creating an object containing its id (T3). Next, the client checks whether all processes have entered the barrier by counting the id objects of all clients (T6–T7). If the barrier is not complete yet, the client blocks until a special `/ready` object exists (T9). This object is created as soon as the barrier is complete (T12), thereby unblocking all clients that are waiting at the barrier.

Using an extension, the client side of the barrier can be greatly simplified: here, a client only invokes a single remote call that blocks until the barrier is complete (C2). For each such call of a client, at the server side, the extension performs the same steps as the client of the traditional implementation (E2–E7). However, there is an important difference: waiting for the `/ready` object to be created does not actually block the extension (E5). Instead, the `block` operation at the server side is implemented to only submit a registration for the object-creation event, which results in the extension terminating afterwards. When the barrier is complete, the standard event-handling mechanism then takes care of sending the unblock notification to the client.

Figure 10 shows that the synchronization overhead for a distributed barrier in EZK and EDS is significantly lower than in ZooKeeper and DepSpace, respectively, both in terms of latency as well as the amount of data to be sent. This is due to the fact that entering a barrier using an extensible coordination service requires only a single remote call. As soon as the last client enters the barrier, the extension immediately creates the `/ready` object and thereby causes the coordination service to send out unblock notifications to clients informing them that the barrier is complete. In contrast, in ZooKeeper and DepSpace, after the last client has entered the barrier, two additional remote calls are required to inform the other clients: one to find out that the barrier is complete (T6) and one to trigger the unblock notifications (T12). As a result, more messages are sent and latency increases.

6.1.4 Leader Election

Appointing a leader from a group of processes and electing a new leader after the old one has terminated or failed are two of the most common coordination tasks in distributed systems. Figure 11 shows how to use a traditional coordination service to implement (handling of corner cases omitted) a `becomeLeader` method that blocks until the caller takes over as acting leader. First, the client creates an object with its id that is automatically deleted by the service as soon as the client terminates or fails (T3). Then, the client determines the current leader by learning the id objects of all clients registered and selecting the object with the lowest creation timestamp (T15–T17). If the client has been elected, the `becomeLeader` method returns instantly (T6). Otherwise, the procedure is repeated each time the client learns that the object of another client has been deleted (T10–T11).[2]

Using an extensible coordination service, the client implementation comprises a single operation that blocks until the client has been elected as leader (C2). The server-side implementation is a combined operation-event extension. When a client registers to become leader, the extension instructs the service to monitor the client (E3) and then performs the client's original `block` call (E4); as discussed in §6.1.3, this operation is non-blocking at the server side. If the current leader terminates or fails, the extension handles the deletion event for the corresponding id object by appointing a new leader (E8–E10) and unblocking the `becomeLeader` call of the client affected (E11).

[2] ZooKeeper avoids a herd effect by notifying only a particular client on such an event. Nevertheless, this client then still has to call `amILeader()`.

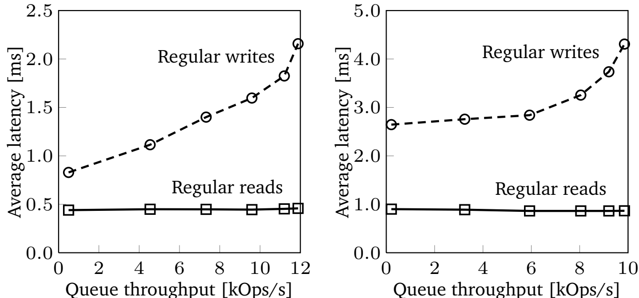

Figure 13. Impact of the queue extension on regular clients accessing EZK (left) and EDS (right).

For our evaluation, we have performed stress tests in which a newly appointed leader immediately abdicates by deleting its id object. The results in Figure 12 show that EZK and EDS are able to achieve more leader changes per second than their respective counterparts. The reason for this lies in the differences in notification overhead after a leader change: while obtaining the confirmation of being the new leader in traditional implementations requires an additional remote call (T15), a client using an extension-based coordination service directly learns of its election. As a result, the signaling latency in EZK and EDS is about 25% and 45% lower than in ZooKeeper and DepSpace, respectively.

6.2 Impact on Regular Clients

Our evaluation of different recipes in §6.1 has shown that extensions offer significant performance benefits to clients. We analyze next their impact on regular clients accessing the coordination service without triggering any extension.

As discussed in §5, making a coordination service extensible requires modifications to the existing architecture. A comparison of the latency of regular reads and writes in EZK and EDS to the latency of the same operations in ZooKeeper and DepSpace shows that the overhead caused by our modifications is negligible (i.e., less than 0.4%). This is mainly due to the extensions being accessible only to clients that have registered or acknowledged them (see §3.6). In contrast, requests of regular clients bypass most of the mechanisms introduced for extensibility.

From a server perspective, executing an extension (i.e., a composition of multiple operations) is usually more expensive than processing a regular read or write. To analyze the impact of extensions on the performance of regular operations, we repeat the distributed-queue experiment presented in §6.1.2. This time, however, we add 30 regular clients that access different data objects of size 256 bytes, a typical size for the objects stored in a coordination service, which are usually smaller than 1 kilobyte [17]. Half of these regular clients continuously read data from EZK and EDS while the remaining fifteen clients perform writes.

Figure 13 shows the latencies observed by regular clients according to the throughput for the distributed queue. For regular writes, latencies in both systems increase when more elements are sent through the queue, and there are two main reasons for this increase. First, the overall number of state modifications that need to be agreed upon across servers. Second, the queue `remove` operation becoming more expensive as the extension needs to access more data objects. The impact on latency is higher for EZK as latencies are generally lower than in EDS. In contrast to writes, the latency of read operations issued by regular clients is mainly unaffected by the queue extension. Both ZooKeeper and DepSpace, and consequently EZK and EDS, each provide a fast path for reads that only overlaps in small and comparably inexpensive parts with the longer and more costly path taken by writes and extensions.

6.3 Discussion

Below, we discuss a number of insights gained from the experiments presented in previous sections.

Comparison of Coordination Recipes. For all four coordination recipes evaluated, EZK and EDS achieve better performance than their respective counterparts, ZooKeeper and DepSpace. Analyzing the reasons for these results, we are able to divide the recipes into two different groups. On the one hand, the shared counter and the distributed queue mainly benefit from the fact that an extension allows clients to execute multiple operations atomically, thereby eliminating the need to retry operations in the face of contention. On the other hand, the recipes for the distributed barrier and leader election, which are both used by clients to wait for a specific event, take advantage of not requiring additional remote calls after the event has actually occurred.

In our evaluation, both clients and servers have been connected via a local network, as it is the case when coordination services are responsible for synchronizing different processes running in the same data center. With network latencies in such an environment being low, remote calls in general and retrying operations in particular are relatively cheap. However, this does not apply to scenarios in which clients access the coordination service via wide-area network links. As a consequence, for both groups of recipes discussed above, we expect extensions to provide even greater performance benefits in geographically distributed settings.

Comparison between EZK and EDS. EZK and EDS differ in many ways including, for example, their fault model. However, with regard to extensibility, the different techniques applied for replication are of particular significance. In EZK, extensions are only executed by the primary, which then distributes the corresponding state modifications. As a result, the amount of data to be exchanged between servers depends on the number and size of the state modifications issued by an extension. In contrast, EDS first distributes the client request that triggers the extension and then processes

the extension on all servers. Here, the size of the messages transmitted between servers depends on the size of the request, but is independent of the extent to which an extension modifies the service state. This advantage comes at the cost of not being able to support nondeterministic extensions.

7. Novel Use Cases for Coordination Services

Current applications mostly use coordination services outside their critical processing path to avoid the costs of accessing them. The performance gains obtained by using extensions may enable use cases that so far have not been considered practical. Two examples of novel uses for coordination services are shared persistent and atomic counters (required by many modern distributed systems such as Percolator [49] and CORFU [8]) and message queues, implementing a highly-available (and restricted) message-oriented middleware in the same line as ActiveMQ [1]. The biggest advantage in using coordination services for implementing these systems would be the reuse of a relatively stable and high-performance fault-tolerant service. Below, we describe two additional use cases related to network and storage.

7.1 Software-defined Networks

The emergence of software-defined networks gives unprecedented freedom to engineers to modify the network control plane via a *network controller*, which (among other things) is responsible for defining routes for every new flow on the network. Being a logically-centralized point of control for the network, the controller needs to satisfy non-functional properties such as dependability and scalability. To satisfy these properties, distributed controllers (e.g., [10, 38]) were developed. As a key requirement, they have to ensure that decisions taken by different controller nodes are based on an approximately consistent view of the network state. ZooKeeper is already employed in practical controllers, for example, for assigning responsibility to specific nodes for portions of the network [38]. However, the coordination service is always accessed outside of the flow processing path.

A fundamental tension in the design of distributed controllers is which part of the network state must be perfectly synchronized between the control nodes, and for which applications. Examples of network applications that do require such consistency are path-route establishment, in which divergent controllers can temporarily install block-hole routes, and load balancing, which requires consistency for an optimal resource utilization [41].

In particular, implementing optimal load balancing for a multi-server system requires each of the controller nodes to assign flows for different servers. For example, with a round-robin policy, each controller needs to get a different sequence number that will be projected to each one of the servers, which requires a shared counter. Without extensions, using ZooKeeper or DepSpace for implementing such a counter would create a bottleneck in the system that would result in the distributed controller being able to assign less than 2k flows/s to servers (see Figure 6). In contrast, employing EZK with our extension-based counter variant, the system can achieve up to 25k increments/s, which is more than what is reported for current distributed controllers [10, 15].

7.2 File-system Metadata Services

Recently, the SCFS file system [12] proposed the use of a coordination service for storing the metadata associated with files (i.e., directories, links, file names, permissions, etc.) that are maintained in cloud storage services like Amazon S3 or Microsoft Azure Blob Store. A fundamental challenge in this context is to map the POSIX semantics to the API provided by the coordination service. As already explained, DepSpace provides an unstructured tuple space that does not provide the notion hierarchy commonly found in a file system namespace. To implement this notion, SCFS developers made each tuple represent an object in the system, with a field containing the name of the parent node.

This design suffers from a fundamental limitation related to the implementation of the *move*/*rename* call used to atomically move files between directories or to change the name of a directory. In DepSpace, the main problem is that modifying the name of a node requires the modification of the parent field in all of its child nodes.[3] However, even if ZooKeeper was used, the operation could not be directly executed either, as it does not support the renaming of nodes.

The solution employed for SCFS was to modify DepSpace by adding a hook that is executed each time a tuple name changes to perform the necessary adjustments. This modification ensures the atomicity of the rename operation and can be easily implemented as an EDS extension.

Notice that without extending the coordination service (changing the code, as done in the case of SCFS, or with our proposed extension) it would be impossible to retain the POSIX semantics of the rename call. However, even if the atomicity of the rename operation were not an issue, using EDS would still bring the benefit of decreasing the number of RPCs from $k+1$ (k being the number of child nodes of the node being renamed) to 1, resulting in significant performance gains for this operation.

8. Related Work

Operating Systems. Extensibility is a common property of operating systems, mostly for dealing with the heterogeneity of the underlying hardware (e.g., device drivers) and deployment needs (e.g., Linux modules). However, some research systems made extensibility a first class property of their designs. For example, Exokernel [24] and JX [28] allow the replacement of fundamental OS components for giving applications direct control of resources. Other systems such as SPIN [11] and VINO [54] allow modular extensions in the

[3] The SCFS developers refrained from introducing an indirection level in order to avoid the additional RPCs such a measure would have required.

form of event handlers (and even function replacements) capable to directly interact with the kernel in response to low-level events. Our extensibility architecture is similar to this second approach, and many of the constraints defined for VINO extensions [54] are similar to the ones we use.

Databases. Extensibility is a well-known concept in database management systems that offer the possibility to introduce new functionality via user-defined functions [42] and stored procedures [57]. In addition, many of these systems support triggers [47], which are executed when the state of the database changes, similar to event extensions in our model. However, while triggers are commonly used by administrators to ensure the integrity of the database, and therefore are not user specific, event extensions are means for coordination-service clients to customize the reaction to events such as the failure of another client.

Mobile Code. Extensible coordination services implement the concept of remote evaluation [56] that aims at moving the computation in a system to where the data is by transferring program code. Besides minimizing the amount of data that has to be sent over the network, this approach also has the advantage of offering clients the possibility to dynamically add new functionality to the server. As a consequence, extensions are also related to mobile agents [37]. However, unlike a mobile agent, an extension is only transferred once, from client to server, and does not move between different servers. Furthermore, while mobile agents store their state internally in order to keep it during transfer, extensions manage their state in the coordination service. Implementations of mobile agents are usually based on interpreted programming languages. On the one hand, this allows an agent to use all the features provided by the language; on the other hand, this makes it more difficult and costly for the server to protect itself against misbehaving agents. In our approach, extensions are restricted to a small set of methods (i.e., the service API and the white-listed functionality) and access the server resources (i.e., the service state) using a sandbox mechanism that enables the server to monitor and possibly decline each access.

Tuple Spaces. The tuple space model used by DepSpace has been introduced in the Linda coordination language [26]. Since then, several additional tuple-space primitives have been proposed (e.g., [7, 18, 34, 51, 53]) as the original set of primitives was not powerful enough to handle all coordination tasks in an efficient way, which supports our argument that there is no one-fits-all coordination kernel. In contrast to the approaches mentioned, our solution to the problem does not require changes to the service API and allows new functionality to be added dynamically.

Different authors proposed to combine mobile agents and tuple spaces [19, 22, 45, 50] in order to exploit the benefits of mobile code (see above) for coordination tasks. Unlike these works, our approach is not limited to the tuple-space model, as shown by EZK. In addition, we address the problem of making existing coordination services extensible and show that our model is suitable for both crash and Byzantine fault-tolerant systems.

Coordination Services. Several works have presented modifications and additions to ZooKeeper (e.g., [21, 23, 25, 36, 40, 55]), but (almost) none of them deals with changing the service's programming model. A notable exception is a recent short paper by Kalantari et al. [36] which identifies inefficiencies related to ZooKeeper's watch mechanism. As a solution, the authors present the prototype implementation of a coordination service that is able to execute critical sections (i.e., sequences of operations that are protected by a lock) relying on a deterministic multi-threaded server. In contrast to this work, we do not focus on a specific coordination service but show that inefficiencies are an inherent problem of limited coordination kernels. In addition, our model for extensible coordination considers security and performance issues and can be applied to existing services without the need to modify major parts of the system (e.g., mechanisms providing fault tolerance), as shown by EZK and EDS. One reason for this is that our approach does not require support for deterministic multithreading.

There are several coordination services that allow clients to submit a batch of operations in a transaction [6, 7, 31]. The service then either atomically executes all the operations belonging to the same transaction or, in case the transaction is aborted, none of them. Although powerful, such transactions have a significant drawback compared with extensions: clients are not able to create a transaction in which one operation uses the result of another operation as input; that is, unlike an extension, a transaction is only a collection of coordination-service operations, without any glue code. Consequently, in none of the use cases presented in §6 a transaction would be able to replace the extension.

9. Conclusion

We proposed a general model for extending coordination services that enables the implementation of very efficient coordination tasks. Our model has been implemented in two coordination services, ZooKeeper and DepSpace, and we showed that with comparably simple extensions it is possible to outperform them by an order of magnitude for distributed queues and atomic counters. We expect these benefits to enable new use cases for coordination services that today are deemed impractical due to performance limitations.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. We also thank Ricardo Mendes for helping us with the DepSpace codebase and Rüdiger Kapitza for interesting discussions and helpful comments on drafts of the paper. This work is partially supported by EC's FP7 project BiobankCloud (317871).

References

[1] Apache ActiveMQ. `http://activemq.apache.org/`.

[2] Apache Curator. `http://curator.apache.org/`.

[3] CoreOS etcd. `https://github.com/coreos/etcd/`.

[4] JZab. `https://github.com/zk1931/jzab/`.

[5] LogCabin. `https://github.com/logcabin/`.

[6] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP '07)*, pages 159–174, 2007.

[7] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, 1995.

[8] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A distributed shared log. *ACM Transactions on Computer Systems*, 31(4), 2013.

[9] R. Baldoni, C. Marchetti, and L. Verde. CORBA request portable interceptors: Analysis and applications. *Concurrency and Computation: Practice and Experience*, 15(6):551–579, 2003.

[10] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an open, distributed SDN OS. In *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*, pages 1–6, 2014.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP '95)*, pages 267–283, 1995.

[12] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Veríssimo. SCFS: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*, pages 169–180, 2014.

[13] A. Bessani, J. Sousa, and E. A. P. Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN '14)*, pages 355–362, 2014.

[14] A. N. Bessani, E. P. Alchieri, M. Correia, and J. Fraga. DepSpace: A Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys '08)*, pages 163–176, 2008.

[15] F. Botelho, F. Ramos, D. Kreutz, and A. Bessani. On the feasibility of a consistent and fault-tolerant data store for SDNs. In *Proceedings of the 2nd European Workshop on Software Defined Networks (EWSDN '13)*, pages 38–43, 2013.

[16] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In *Distributed Systems (2nd Edition)*, pages 199–216. Addison-Wesley, 1993.

[17] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[18] P. Butcher, A. C. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency – Practice and Experience*, 6(6): 505–516, 1994.

[19] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[20] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th Symposium on Principles of Distributed Computing (PODC '07)*, pages 398–407, 2007.

[21] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 277–290, 2009.

[22] E. Denti, A. Natali, A. Omicini, and M. Venuti. An extensible frame work for the development of coordinated applications. In *Proceedings of the 1st International Conference on Coordination Languages and Models (COORDINATION '96)*, pages 305–320, 1996.

[23] T. Distler and R. Kapitza. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys '11)*, pages 91–105, 2011.

[24] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, 1995.

[25] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Hierarchical policies for software defined networks. In *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networking (HotSDN '12)*, pages 37–42, 2012.

[26] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1): 80–112, 1985.

[27] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 103–116, 2009.

[28] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *Proceedings of the 2002 USENIX Annual Technical Conference (ATC '02)*, pages 45–58, 2002.

[29] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149, 1991.

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

[31] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC '10)*, pages 145–158, 2010.

[32] J. Hunter and W. Crawford. *Java servlet programming*. O'Reilly Media, 2001.

[33] M. Isard. Autopilot: Automatic data center management. *SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[34] B. Johanson and A. Fox. The event heap: A coordination infrastructure for interactive workspaces. In *Proceedings of the 4th Workshop on Mobile Computing Systems and Applications (WMCSA '02)*, pages 83–93, 2002.

[35] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st International Conference on Dependable Systems and Networks (DSN '11)*, pages 245–256, 2011.

[36] B. Kalantari and A. Schiper. Addressing the ZooKeeper synchronization inefficiency. In *Proceedings of the 14th International Conference on Distributed Computing and Networking (ICDCN '13)*, pages 434–438, 2013.

[37] N. M. Karnik and A. R. Tripathi. Design issues in mobile-agent programming systems. *IEEE Concurrency*, 6(3):52–61, 1998.

[38] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 351–364, 2010.

[39] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[40] 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 *Proceedings of the 23rd Symposium on Operating Systems Principles (SOSP '11)*, pages 279–294, 2011.

[41] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized? State distribution trade-offs in software defined networks. In *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networking (HotSDN '12)*, pages 1–6, 2012.

[42] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath. Design and implementation of an extensible database management system supporting user defined data types and functions. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB '88)*, pages 294–305, 1988.

[43] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4, 1987.

[44] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 105–120, 2004.

[45] A. Omicini and F. Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[46] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*, pages 305–320, 2014.

[47] N. W. Paton and O. Díaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

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

[49] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 251–264, 2010.

[50] A. I. T. Rowstron. Using mobile code to provide fault tolerance in tuple space based coordination languages. *Science of Computer Programming*, 46(1):137–162, 2003.

[51] A. I. T. Rowstron and A. M. Wood. Solving the Linda multiple rd problem using the copy-collect primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[52] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computer Survey*, 22(4):299–319, 1990.

[53] E. J. Segall. Resilient distributed objects: Basic results and application to shared tuple spaces. In *Proceedings of the 7th Symposium on Parallel and Distributeed Processing (SPDP '95)*, pages 320–327, 1995.

[54] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, 1996.

[55] 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 *Proceedings of the 3rd International Conference on Communication Systems and Networks (COMSNETS '11)*, pages 1–10, 2011.

[56] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4): 537–564, 1990.

[57] M. Stonebraker, J. Anton, and E. Hanson. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3):350–376, 1987.

[58] W. Vogels. Life is not a state-machine: The long road from research to production. In *Proceedings of the 25th Symposium on Principles of Distributed Computing (PODC '06)*, page 112, 2006.