Enhancing Coordination in Cloud Infrastructures with an Extendable Coordination Service *

Tobias Distler[1], Frank Fischer[1], Rüdiger Kapitza[2], and Siqi Ling[1]
[1]Friedrich–Alexander University Erlangen–Nuremberg [2]TU Braunschweig

ABSTRACT

With application processes being distributed across a large number of nodes, coordination is a crucial but inherently difficult task in cloud environments. Coordination middleware systems like Chubby and ZooKeeper approach this problem by providing mechanisms for basic coordination tasks (e. g., leader election) and means to implement common data structures used for coordination (e. g., distributed queues). However, as such complex abstractions still need to be implemented as part of the distributed application, reusability is limited and the performance overhead may be significant.

In this paper, we address these problems by proposing an *extendable coordination service* that allows complex abstractions to be implemented on the server side. To enhance the functionality of our coordination service, programmers are able to dynamically register high-level *extensions* that comprise a sequence of low-level operations offered by the standard coordination service API. Our evaluation results show that extension-based implementations of common data structures and services offer significantly better performance and scalability than their state-of-the-art counterparts.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Distributed Systems

General Terms Design, Performance, Reliability

Keywords Cloud, Coordination Service, ZooKeeper

1. INTRODUCTION

Large-scale applications running on today's cloud infrastructures may comprise a multitude of processes distributed over a large number of nodes. Given these circumstances, fault-tolerant coordination of processes, although being an essential factor for the correctness of an application, is difficult to achieve. As a result, and to facilitate their design, fewer and fewer of such applications implement coordination primitives themselves, instead they rely on external coordination services. Large-scale distributed storage systems like BigTable [6] and HBase [2], for example, do not provide means for leader election but perform this task using the functionality of Chubby [5] and ZooKeeper [7], respectively.

However, instead of implementing more complex services (e. g., leader election) directly, state-of-the-art coordination middleware systems only provide a basic set of low-level functions including file-system–like access to key-value storage for small chunks of data, a notification-based callback mechanism, and rudimentary access control. On the one hand, this approach has several benefits: Based on this low-level functionality, more complex services and data structures for the coordination of application processes (e. g., distributed queues) can be implemented. Furthermore, the fact that state-of-the-art coordination services are replicated frees applications developers from the need to deal with fault-tolerance–related problems, as the coordination service does not represent a single point of failure. On the other hand, this flexibility comes at a price: With more complex services being implemented at the coordination-service client (i. e., as part of the distributed application), reusability is limited and maintenance becomes more difficult. In addition, there is a performance overhead for cases in which a complex operation requires multiple remote calls to the coordination service. As we show in our evaluation, this problem gets worse the more application processes access a coordination service concurrently.

To address the disadvantages of current systems listed above, we propose an *extendable coordination service*. In contrast to existing solutions, in our approach, more complex services and data structures are not implemented at the client side but within modules ("*extensions*") that are executed at the servers running the coordination service. As a result, implementations of coordination-service clients can be greatly simplified; in fact, in most usage scenarios, only a single remote call to the coordination service is required.

In our service, an extension is realized as a sequence of regular coordination-service operations that are processed atomically. This way, an extension can benefit from the flexibility offered by the low-level API of a regular coordination service while achieving good performance under contention.

Besides enhancing the implementation of abstractions already used by current distributed applications, extensions also allow programmers to introduce new features that cannot be provided based on the functionality of traditional coordination services: By registering custom extensions, for example, it is possible to integrate assertions into our ex-

*This work was partially supported by an IBM Ph.D. Fellowship for T. Distler, and by the European Union's 7th Framework Programme (FP7/2007-2013) under grant agreement n°257243 (TClouds: `http://www.tclouds-project.eu/`).

SDMCMM '12, December 3-4, 2012, Montreal, Quebec, Canada.

tendable coordination service that perform sanity checks on input data, improving protection against faulty clients. Furthermore, extensions may be used to execute automatic conversion routines for legacy clients, supporting scenarios in which the format of the coordination-related data managed on behalf of an application differs across program versions.

In particular, this paper makes the following three contributions: First, it proposes a coordination service whose functionality can be enhanced dynamically by introducing customized extensions. Second, it provides details on our prototype of an extendable coordination service based on ZooKeeper [7], a coordination middleware widely used in industry. Third, it presents two case studies, a priority queue and a quota-enforcement service, illustrating both the flexibility and efficiency of our approach.

2. BACKGROUND

This section provides background information on the basic functionality of a coordination service and presents an example of a higher-level abstraction built on top of it.

2.1 Coordination Services

Despite their differences in detail, coordination services like Chubby [5] and ZooKeeper [7] expose a similar API to the client (i. e., a process of a distributed application, see Figure 1). Information is stored in *nodes* which can be created (`create`[1]) and deleted (`delete`) by a client. Furthermore, there are operations to store (`setData`) and retrieve (`getData`) the data assigned to a node. In general, there are two different types of nodes: *ephemeral nodes* are automatically deleted when the session of the client who created the node ends (e. g., due to a fault); in contrast, *regular nodes* persist after the end of a client session.

Besides managing data, current coordination services provide a callback mechanism to inform clients about certain events including, for example, the creation or deletion of a node, or the modification of the data assigned to a node (see Figure 1). On the occurrence of an event a client has registered a *watch* for, the coordination service performs a callback notifying the client about the event. Using this functionality, a client is, for example, able to implement failure detection of another client by setting a deletion watch on an ephemeral node created by the client to monitor.

2.2 Usage Example: Priority Queue

Based on the low-level API provided by the coordination service, application programmers can implement more complex data structures to be used for the coordination of processes. Figure 2 shows an example implementation of a distributed priority queue (derived from the queue implementation in [13]) that can be applied to exchange data between two processes running on different machines: a *producer* and a *consumer*. New elements are added to the queue by the producer calling `insert`; the element with the highest priority is dequeued by the consumer calling `remove`.

To insert an element `b` into the queue, the producer creates a new node and sets its data[2] to `b` (L. 8). The priority `p` of the element is thereby encoded in the node name by appending `p` to a default name prefix (L. 5). To remove the head element of the queue, the consumer queries the coordination service to get the names of all nodes matching the default name prefix (L. 13). From the result set of node names, the consumer then locally determines the head of the queue by selecting the node name indicating the highest priority (L. 14). Knowing the head node, the consumer is able to retrieve its data from the coordination service (L. 17) before removing the node from the queue (L. 18).

Note that the priority-queue implementation in Figure 2 has two major drawbacks: First, while the `insert` operation involves only a single remote call to the coordination service (L. 8), the `remove` operation requires three remote calls (L. 13, 17, and 18), resulting in additional latency. Second, the implementation does not scale for multiple consumer processes: In order to prevent different consumers from returning the same element, entire `remove` operations would either have to be executed sequentially (which is difficult to achieve when consumer processes run on different machines) or they would have to be implemented optimistically; that is, if the `delete` call (L. 18) aborts due to a concurrent `remove` operation already having deleted the designated head node, a consumer must retry its `remove` (omitted in Figure 2). In Section 5.1, we show that the performance of the optimistic variant suffers from contention when multiple consumer processes access the queue concurrently.

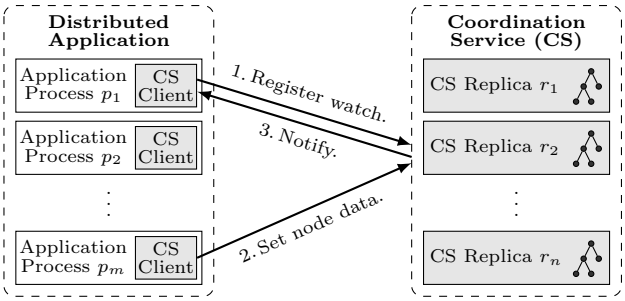


Figure 1: Callback mechanism usage example: An application process p_1 registers a data watch on a node; when the node's data is updated, the coordination service notifies p_1 about the modification.

```
1  CoordinationService cs = establish connection;
3  void insert(byte[] b, Priority p) {
4    /* Encode priority in node name. */
5    String nodeName = "/node-" + p;
7    /* Create node and set its data to b. */
8    cs.create(nodeName, b);
9  }
11 byte[] remove() {
12   /* Find the node with the highest priority. */
13   String[] nodes = get node names from cs;
14   String head = node from nodes
           with highest priority according to its name;
16   /* Get node data and remove node. */
17   byte[] b = cs.getData(head);
18   cs.delete(head);
19   return b;
20 }
```

Figure 2: Pseudo-code implementation of a priority-queue client (ZooKeeper): an element is represented by a node, the priority is encoded in the node name.

[1]Note that we use the ZooKeeper terms here as our prototype is based on this particular coordination service.

[2]The ZooKeeper API allows a client to assign data to a node at creation time. Otherwise an additional `setData` call would be necessary for setting the node data.

3. ENHANCING COORDINATION

The priority-queue example discussed in Section 2.2 illustrates the main disadvantage of state-of-the art coordination services: With implementations of higher-level data structures and services being a composition of multiple low-level remote calls to the coordination service, performance and scalability become a major concern. We address this issue with an *extendable coordination service* that provides means to implement additional functionality directly at the server.

3.1 Basic Approach

To add functionality to our coordination service, programmers write *extensions* that are integrated via software modules. Depending on the mechanism an extension operates on, we distinguish between the following three types:

- During integration, a **node extension** registers a *virtual node* through which the extension will be accessible to the client. In contrast to a regular node, client operations invoked on a virtual node (or one of its sub nodes) are not directly executed by the coordination-service logic; instead, such requests are intercepted and redirected to the corresponding node extension.
- A **watch extension** may be used to customize/overwrite the behavior of the coordination service for a certain watch. Such an extension is executed each time a watch event of the corresponding type occurs.
- A **session extension** is triggered at creation and termination of a client session and is therefore suitable to perform initialization and cleanup tasks.

Note that an extension module providing additional functionality may be a composition of multiple extensions of possibly different types.

In general, an extension is free to use the entire API provided by the coordination service. As a consequence, a stateful extension, for example, is allowed to create own regular nodes to manage its internal state. Furthermore, a complex node extension, for example, may translate an incoming client request into a composite request comprising a sequence of low-level operations. Note that, in such a case, our coordination service guarantees that low-level operations belonging to the same composite request will be executed atomically (see Section 4.3).

3.2 Usage Example: Enhanced Priority Queue

Figure 3 shows how the implementation of the priority-queue client from Figure 2 can be greatly simplified by realizing the queue as a node extension that is accessed via a virtual node `/queue`. In contrast to the traditional implementation presented in Section 2.2, our extension variant only requires a single remote call for the removal of the head element from the queue.

When a client inserts an element into the queue by creating a sub node of `/queue` (L. C5), the request is forwarded to the queue extension, which in turn processes it without any modifications (L. E5); that is, the extension creates the sub node as a regular node. To dequeue the head element, a client issues a `getData` call to a (non-existent) sub node `/queue/next` (L. C10). On the reception of a `getData` call using this particular node name, the extension removes the head element and returns its data to the client (L. E10-E17).

Client Implementation

```
C1  CoordinationService cs = establish connection;
C3  void insert(byte[] b, Priority p) {
C4    /* Create node and set its data to b. */
C5    cs.create("/queue/node-" + p, b);
C6  }
C8  byte[] remove() {
C9    /* Remove head node and return its data. */
C10   return cs.getData("/queue/next");
C11 }
```

Coordination Service Extension Implementation

```
E1  CoordinationServiceState local = local state;
E3  void create(String name, byte[] b) {
E4    /* Process request without modifications. */
E5    local.create(name, b);
E6  }
E8  byte[] getData(String name) {
E9    if("/queue/next".equals(name)) {
E10     /* Find the node with the highest priority. */
E11     String[] nodes = get node names from local;
E12     String head = node from nodes
              with highest priority according to its name;
E14     /* Get node data and remove node. */
E15     byte[] b = local.getData(head);
E16     local.delete(head);
E17     return b;
E18   } else {
E19     /* Return data of regular node. */
E20     return local.getData(name);
E21   }
E22 }
```

Figure 3: Pseudo-code implementation of a priority queue in our extendable coordination service: the extension is represented by a virtual node `/queue`.

Although the steps executed during the dequeuing of the head element are identical to the corresponding procedure in the traditional priority-queue implementation (L. 12-19 in Figure 2), there is an important difference: the calls for learning the node names of queue elements (L. E11), for retrieving the data of the head element (L. E15), and for deleting the head-element node (L. E16) are all local calls with low performance overhead. Furthermore, with these three calls being processed atomically, the implementation does not suffer from contention, as shown in Section 5.1.

4. EXTENDABLE ZOOKEEPER

In this section, we present details on the implementation of *Extendable ZooKeeper (EZK)*, our prototype of an extendable coordination service, which is based on ZooKeeper [7].

4.1 Overview

EZK relies on actively-replicated ZooKeeper for fault tolerance. At the server side, EZK (like ZooKeeper) distinguishes between client requests that modify the state of the coordination service (e. g., by creating a node) and read-only client requests that do not (e. g., as they only read the data of a node). A read-only request is only executed on the server replica that has received the request from the client. In contrast, to ensure strong consistency, a state-modifying request is distributed using an atomic broadcast protocol [8] and then processed by all server replicas.

For EZK, we introduce an *extension manager* component into each server replica which is mainly responsible for redirecting the control and data flow to the extensions registered. The extension manager performs different tasks for different types of extensions (see Section 3.1): On the reception of a client request, the extension manager checks whether the request accesses the virtual node of a node extension and, if this is the case, forwards the request to the corresponding extension. This way, a node extension is able to control the behavior of an incoming request before the request had any impact on the system. In addition, the extension manager intercepts watch events and, if available, redirects them to the watch extensions handling the specific events, allowing the extension to customize callbacks to the client. Finally, the extension manager also monitors ZooKeeper's session tracker and notifies the session extensions registered about the start and end of client sessions.

4.2 Managing an Extension

For extension management in EZK we provide a *built-in management extension* that is accessible through a virtual node `/extensions`. To register a custom extension, a client creates a sub node of `/extensions` and assigns all necessary configuration information as data to this management node. For a node extension, for example, the configuration information includes the name of the virtual node through which the extension can be used by a client, and a Java class containing the extension code to execute when a request accesses this virtual node. Furthermore, a client is able to provide an *ephemeral* flag indicating whether the extension should be automatically removed by EZK when the session of the client who registered the extension ends; apart from that, an extension can always be removed by explicitly deleting its corresponding management node.

When EZK's management extension receives a request from a client to register an extension, it verifies that the extension code submitted is a valid Java class, and then distributes the request to all server replicas. By treating the request like any other state-modifying request, EZK ensures that all server replicas register the extension in a consistent manner. After registration is complete the extension manager starts to make use of the extension.

4.3 Atomic Execution of an Extension

Traditional implementations of complex operations comprising multiple remote calls to the coordination service (as, for example, removing the head element of a priority queue, see Section 2.2) require the state they operate on not to change between individual calls. As a consequence, such an operation may be aborted when two clients modify the same node concurrently, resulting in a significant performance penalty (see Section 5.1). We address this problem in EZK by executing complex operations atomically.

In ZooKeeper, each client request modifying the state of the coordination service is translated into a corresponding *transaction* which is then processed by all server replicas. In the default implementation a single state-modifying request leads to a single transaction. To support more complex operations, we introduce a new type of transaction in EZK, the *container transaction*, which may comprise a batch of multiple regular transactions. EZK guarantees that all transactions belonging to the same container transaction will be executed atomically on all server replicas, without interfering with other transactions. By including all transactions of the same extension-based operation in the same container transaction, EZK prevents concurrent state changes during the execution of an extension.

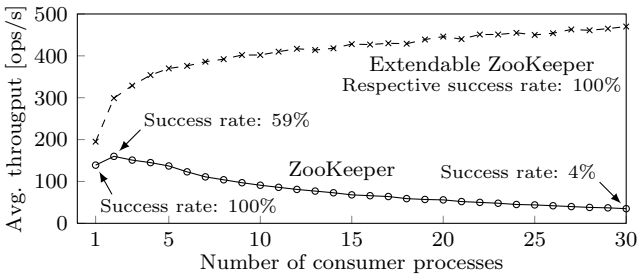


Figure 4: Throughput (i. e., successful dequeue operations) for different priority-queue implementations for different numbers of consumer processes.

5. CASE STUDIES

In this section, we evaluate the priority-queue extension introduced in Section 3.2. Furthermore, we present an additional example of how extensions can be used in our coordination service to efficiently provide more complex functionality. All experiments are conducted using a coordination-service cell comprising five server replicas (i. e., a typical configuration for ZooKeeper), each running in a virtual machine in Amazon EC2 [1]; coordination-service clients are executed in an additional virtual machine. As in practice distributed applications usually run in the same data center as the coordination service they rely on [5], we allocate all virtual machines in the same EC2 region (i. e., Europe).

5.1 Priority Queue

Our first case study compares a traditional priority-queue implementation (see Section 2.2) against our extension-based EZK variant (see Section 3.2). For both implementations, we measure the number of successful dequeue operations per second for a varying number of consumer processes accessing the queue concurrently. At all times during the experiments, we ensure that there are enough producer processes to prevent the queue from running empty. As a result, no dequeue operation will fail due to lack of items to remove.

Figure 4 presents the results of the experiments: For a single consumer process, the priority queues achieve an average throughput of 139 (ZooKeeper variant) and 195 (EZK) dequeue operations per second, respectively. The difference in performance is due to the fact that in the ZooKeeper implementation the `remove` operation comprises three (i. e., two read-only and one state-modifying) remote calls to the coordination service, whereas the extension-based EZK variant requires only a single (state-modifying) remote call.

Our results also show that for multiple consumer processes the ZooKeeper priority queue suffers from contention: Due to its optimistic approach a dequeue operation may be aborted when issued concurrently with another dequeue operation (see Section 2.2), causing the success rate to decrease for an increasing number of consumers. In contrast, dequeue operations in our EZK implementation are executed atomically and therefore always succeed on a non-empty queue. As a result, the extension-based EZK variant achieves better scalability than the traditional priority queue.

```
1  CoordinationService cs = establish connection;
3  void allocate(int amount) {
4    do {
5      /* Determine free quota and node version. */
6      (int free, int version) = cs.getData("/memory");
8      /* Retry if there is not enough quota. */
9      if(free < amount) sleep and continue;
11     /* Calculate and try to set new free quota. */
12     cs.setData("/memory", free - amount, version);
13   } while(setData call aborted);
14 }
```

Figure 5: Pseudo-code implementation of a quota-server client in ZooKeeper: the current amount of free quota is stored in the data of `/memory`; to release quota, `allocate` is called with a negative `amount`.

5.2 Quota Enforcement Service

Our second case study is a fault-tolerant quota enforcement service guaranteeing upper bounds for the overall resource usage (e. g., number of CPUs, memory usage, network bandwidth) of a distributed application [3]. In order to enforce a global quota, each time an application process wants to dynamically allocate additional resources, it is required to ask the quota service for permission. The quota service only grants this permission in case the combined resource usage of all processes of the application does not exceed a certain threshold; otherwise the allocation request is declined and the application process is required to wait until additional free quota becomes available, for example, due to another process terminating and therefore releasing its resources.

5.2.1 Traditional Implementation

Figure 5 illustrates how to implement a quota service based on a state-of-the-art coordination service. In this approach, information about free resource quotas (in the example: the amount of free memory available) is stored in the data assigned to a resource-specific node (i. e., `/memory`). To request permission for using additional quota, an application process invokes the quota client's `allocate` function indicating the amount of quota to be allocated (L. 3). Due to the traditional coordination service only providing functionality to get and set the data assigned to a node, but lacking means to modify node data based on its current value, the quota client needs to split up the operation into three steps: First, the client retrieves the data assigned to `/memory` (L. 6), thereby learning the application's current amount of free quota. Next, the quota client checks whether the application has enough free quota available to grant the permission (L. 9). If this is the case, the client locally computes the new amount of free quota and updates the corresponding node data at the coordination service (L. 12).

Note that the optimistic procedure described above is only correct as long as the data assigned to `/memory` does not change between the `getData` (L. 6) and `setData` (L. 12) remote calls. However, as different quota clients could invoke `allocate` for the same resource type concurrently, this condition may not always be justified. To address this problem, state-of-the-art coordination services like Chubby [5] and ZooKeeper [7] use node-specific version counters (which are incremented each time the data of a node is reassigned) to provide a `setData` operation with compare-and-swap semantics. Such an operation only succeeds if the current version matches an expected value (L. 12), in this case, the version number that corresponds to the contents the quota client has retrieved (L. 6). If the two version numbers differ, the `setData` operation aborts and the quota client retries the entire allocation procedure (L. 13).

Client Implementation

```
C1 CoordinationService cs = establish connection;
C3 void allocate(int amount) {
C4   do {
C5     /* Issue quota demand. */
C6     cs.setData("/memory-quota", amount);
C7   } while(setData call aborted);
C8 }
```

Coordination Service Extension Implementation

```
E1  CoordinationServiceState local = local state;
E3  void setData(String name, int amount) {
E4    if("/memory-quota".equals(name)) {
E5      int free = local.getData("/memory");
E7      /* Abort if there is not enough quota. */
E8      if(free < amount) abort;
E10     /* Calculate and set new free quota. */
E11     local.setData("/memory", free - amount);
E12   } else {
E13     /* Set data of regular node. */
E14     local.setData(name, amount);
E15   }
E16 }
```

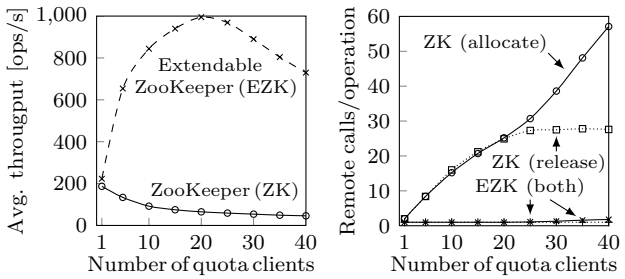
Figure 6: Pseudo-code implementation of a quota server in our extendable coordination service: a call to `setData` only aborts if there is not enough quota.

5.2.2 Extension-based Implementation

In contrast to the traditional implementation presented in Section 5.2.1 where remote calls issued by a quota client may be aborted due to contention, allocation requests in our extension-based EZK variant of the quota enforcement service (see Figure 6) are always granted when enough free quota is available. Here, to issue an allocation request, a client invokes a `setData` call to the virtual `/memory-quota` node passing the amount of quota to allocate as data (L. C6). In the absence of network and server faults, this call only aborts if the amount requested exceeds the free quota currently available (L. E8), in which case the quota client retries the procedure (L. C7) after a certain period of time (omitted in Figure 6). At the EZK server, the quota enforcement extension functions as a proxy for a regular node `/memory`: For each incoming `setData` call to the virtual `/memory-quota` node (L. E4), the extension translates the request into a sequence of operations (i. e., a read (L. E5), a check (L. E8), and an update (L. E11)) that are processed atomically.

5.2.3 Evaluation

We evaluate both implementations of the quota enforcement service varying the number of quota clients accessing the service concurrently from 1 to 40. During a test run, each client repeatedly requests 100 quota units, and when the quota is granted (possibly after multiple retries), immediately releases it again. In all cases, the total amount of quota available is limited to 1500 units. As a consequence, in scenarios with more than 15 concurrent quota clients, allocation requests may be aborted due to lack of free quota.



(a) Overall throughput (b) Costs per operation

Figure 7: Throughput (i. e., successful allocation and release operations) and costs (i. e., remote calls per operation) for different quota-server variants; the total quota is limited to the demand of 15 clients.

The throughput results for this experiment presented in Figure 7a show that our EZK quota server provides better scalability than the state-of-the-art ZooKeeper variant. For a small number of concurrent clients, the fact that the total amount of quota is limited has no effect: As in the priority-queue experiment (see Section 5.1), the ZooKeeper implementation suffers from contention, whereas the throughput of the EZK quota server improves for multiple quota clients. For more than 15 quota clients, the fraction of aborted allocation requests increases in both implementations with every additional client, leading to an observable throughput decrease for the EZK quota server for more than 20 clients.

Figure 7b shows that the costs for a single quota allocation greatly differ between both quota-service implementations: For 40 clients, due to contention and the limited amount of total quota, it takes a ZooKeeper client more than 57 remote calls to the coordination service to be granted the quota requested; an EZK quota client on average has to issue less than 2 remote calls for the same scenario. Note that in the ZooKeeper variant, release operations are also subject to contention, requiring up to 28 remote calls per successful operation. In contrast, the release operation in our EZK implementation always succeeds using a single remote call.

6. RELATED WORK

With the advent of large distributed file systems emerged the need to coordinate read and write accesses on different nodes. This problem was solved by distributed lock managers [11], the predecessors of current coordination services.

In contrast to the file-system–oriented coordination middleware systems Chubby [5] and ZooKeeper [7], DepSpace [4] is a Byzantine fault-tolerant coordination service which implements the tuple space model. As the tuple space abstraction does not provide an operation to alter stored tuples, in order to update the data associated with a tuple, the tuple has to be removed from the tuple space, modified, and reinserted afterwards. In consequence, implementations of high-level data structures and services built over DepSpace are expected to also suffer from contention for multiple concurrent clients. Note that, with our approach not being limited to a specific interface, this problem could be approached by an extension-based variant of DepSpace.

Boxwood [9] shares our goal of freeing application developers from the need to deal with issues like consistency, dependability, or efficiency of complex high-level abstractions. However, unlike our work, Boxwood focuses on storage infrastructure, not coordination middleware systems. In addition, the set of abstractions and services exposed by Boxwood is static, whereas our extendable coordination service allows clients to dynamically customize the behavior of existing operations and/or introduce entirely new functionality.

Relational database management systems rely on stored procedures [12] (i. e., compositions of multiple SQL statements) to reduce network traffic between applications and the database, similar to our use of extensions to minimize the number of remote calls a client has to issue to the coordination service. In active database systems [10], triggers (i. e., a special form of stored procedures) can be registered to handle certain events, for example, the insertion, modification, or deletion of a record. As such, triggers are related to watches in coordination services. The main difference is that in general a trigger is a database-specific mechanism which is transparent to applications. As a result, applications are not able to change the behavior of a trigger. In contrast, our extendable coordination service offers applications the flexibility to customize the service using a composition of extensions operating on nodes, watches, and sessions.

7. CONCLUSION

This paper proposed to enhance coordination of distributed applications by relying on an extendable coordination service. Such a service allows programmers to dynamically introduce custom high-level abstractions which are then executed on the server side. Our evaluation shows that by processing complex operations atomically, an extendable coordination service offers significantly better performance and scalability than state-of-the-art implementations.

8. REFERENCES

[1] Amazon EC2. `http://aws.amazon.com/ec2/`.
[2] Apache HBase. `http://hbase.apache.org/`.
[3] J. Behl, T. Distler, and R. Kapitza. DQMP: A decentralized protocol to enforce global quotas in cloud environments. In *Proc. of SSS '12*, pages 217–231, 2012.
[4] A. N. Bessani, E. P. Alchieri, M. Correia, and J. Fraga. DepSpace: A Byzantine fault-tolerant coordination service. In *Proc. of EuroSys '08*, pages 163–176, 2008.
[5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI '06*, pages 335–350, 2006.
[6] 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*, pages 205–218, 2006.
[7] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. of ATC '10*, pages 145–158, 2010.
[8] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proc. of DSN '11*, pages 245–256, 2011.
[9] 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*, pages 105–120, 2004.
[10] N. W. Paton and O. Díaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.
[11] W. Snaman and D. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, 5:29–44, 1987.
[12] M. Stonebraker, J. Anton, and E. Hanson. Extending a database system with procedures. *Transactions on Database Systems*, 12(3):350–376, 1987.
[13] ZooKeeper Tutorial: Queues. `http://wiki.apache.org/hadoop/ZooKeeper/Tutorial`.