Addressing the ZooKeeper Synchronization Inefficiency

Babak Kalantari and André Schiper

Ecole Polytechnique Fédérale de Lausanne (EPFL)
{babak.kalantari,andre.schiper}@epfl.ch

Abstract. In this paper we discuss the problem of synchronization in ZooKeeper, a fault-tolerant distributed coordination framework. One of the key features of ZooKeeper is to move away from blocking API such as locks, in order to avoid problems with slow or faulty clients. Instead, it provides an event like synchronization mechanism, allowing clients to be notified upon state change on the server. However, such a mechanism leads to very inefficient implementation of synchronization objects such as queues or barriers. We propose a new solution to this problem.
The solution is to handle a sequence of client operations completely on the server. This means that the client implements the required sequence of operations as a single request, which is sent to the server for execution via a generic API. We present a prototype that shares some of the concepts of ZooKeeper but, contrary to ZooKeeper, allows a very efficient implementation of synchronization objects. The solution requires a deterministic multi-threaded server, which we implement thanks to a coroutine mechanism. Experiments show the significant gain in efficiency of our solution on producer-consumer queues and synchronization barriers.

Keywords: Coordination service; Synchronization; Deterministic scheduler.

1 Introduction

ZooKeeper [1] is a reliable service for coordinating distributed applications that consist of several processes. A *coordination* service is a middleware that provides high level primitives and abstractions to such distributed applications. Examples of coordination are: managing application states, providing locks and queues, leader election, etc. Coordination services themselves achieve fault tolerance by replication, i.e., active replication (state machine replication) [2], [3] or passive replication (primary-backup replication) [4], [5]. In the recent years there has been several developments to address distributed coordination. Boxwood [6] provided a lock service, reliable state management and a failure detection service, each in a separate module. Later the Chubby [7] fault tolerant locking service, which was partly inspired by Boxwood, provided a high level API that unifies locks and state management into file-like operations, i.e., *open/close*, *read/write*, etc. Similarly to Boxwood, Chubby also uses Paxos [8] to achieve fault tolerance.

The most recent interesting development along the same line, while providing a more powerful API, is ZooKeeper [9]. The fault tolerance of the ZooKeeper service is ensured using an atomic broadcast algorithm [10] slightly different from Paxos. More important, contrary to Boxwood and Chubby, ZooKeeper moves away from a blocking API (e.g., locks). The reason is to avoid having to address the problem of slow or faulty clients, hence providing what the authors of ZooKeeper call a *wait-free* client API. However synchronization, as an essential coordination mechanism, cannot be left out. Therefore, ZooKeeper provides an event-based synchronization mechanism called *watch*. A watch allows client processes to block, waiting for the update of some server's data. As we point out in the paper, a watch is a synchronization mechanism that leads to inefficiency in many applications. Watches also cause secondary problems such as herd effects, discussed in the paper.

Solving the problem by adding a blocking API to ZooKeeper is not an option, since this has been excluded by design. We show in the paper that another solution is possible. Indeed, the problem is not so much the blocking operation performed by the client; it is rather the fact that the corresponding unblocking operation might not be invoked in case of the crash of the client. We prevent this from happening by allowing clients to send a sequence of operations to the server, including blocking and unblocking operations. This guarantees that, once the first operation is executed on the server, all operations are available for execution on the server, even if the client crashes. We have designed and implemented a prototype of such a coordination framework, which shares some of its concepts with ZooKeeper (e.g., a file system based shared memory). Our server handles the clients' requests in a minimal kernel with a deterministic scheduler, providing semaphores to allow clients synchronization in an elegant and efficient manner.

The paper is organized as follows. Section 2 provides an overview of ZooKeeper. In Section 3 we discuss limitations of the synchronization mechanisms of ZooKeeper. In Section 4 we present our solution. Section 5 is devoted to an experimental evaluation: we compare the cost of the ZooKeeper synchronization mechanisms and our prototype on a queue object and a barrier object. Finally, Section 6 concludes the paper.

2 ZooKeeper overview

The goal of ZooKeeper is to provide fault tolerant coordination services for distributed applications, which consist of multiple processes. These processes are *clients*[1] of the ZooKeeper coordination *service*. ZooKeeper coordination is achieved by allowing processes to access a shared hierarchical name space of data registers called *znodes* [9]. The name space structure is very similar to that of a standard file system, with files and directories. Every znode has a unique, fully qualified name (i.e. absolute node path), and can store application data and

[1] A client can itself act as a server in the user application; nevertheless for the coordination service it remains a client.

children nodes. For fault tolerance, the ZooKeeper name space is replicated over an ensemble of servers (hosts) using state machine replication: atomic broadcast is used to ensure consistency of the state of the server replicas [10]. Whenever coordination within the distributed application is required, the appropriate operation on a ZooKeeper object is invoked, using the API provided. Invoking an operation requires to open a connection to a single ZooKeeper server to establish a session. Once the session is established, a client can submit operations.

The basic ZooKeeper operations allow the client to add/delete znodes (operations *create* and *delete*), to read/write the data stored in znodes (*getData* and *setData*) and to obtain the children of a znode (*getChildren*).

For synchronization, ZooKeeper provides an event mechanism. Events are provided through the *watch* mechanism. A client can attach a watcher (an event handler) to a znode and wait. The client is later notified upon state change of the znode through the triggering of the event handler. The notification does not contain any information about the change itself.

3 Limitations of synchronization in ZooKeeper

The synchronization mechanism provided by ZooKeeper is rather poor because: (a) by watching a znode, clients cannot reliably see every change that happens, (b) notifications do not contain adequate information for most cases, and (c) upon a change all the watchers are notified.[2] In order to understand deficiencies of ZooKeeper, we discuss the implementation of a very basic coordination object, namely a producer-consumer queue. The queue object itself is not the main issue here: the main issue is the mechanisms allowing an efficient queue implementation (which are not specific to queues). As such we do not claim by any means that our queue implementation is the most efficient one nor that it is novel. But we demonstrate an alternative to ZooKeeper that can be significantly more efficient.

A producer-consumer queue object has two operations *enqueue()*, to insert element at the tail, and *dequeue()*, to remove an element from the head. Moreover, the dequeue() operation must block the client if the queue is empty, and the enqueue() operation must block the client if the queue is full. Implementing such a queue requires to solve two problems: (1) a communication problem, and (2) a synchronization problem. The communication problem requires the consumer to get the right queue element according to the FIFO policy. The synchronization problem requires to correctly handle blocking and unblocking of client processes.

3.1 Queue example: Solving the communication problem

The communication problem can be solved using the shared znode data space. This includes control data, i.e., to use the znode space for data that allows to

[2] Watches are one-time triggers and there is a delay between getting the watch event and setting the watch to get the next one. Moreover, a watch notification which was set by *getData* does not provide the new data; the same is true for *getChildren*.

locate the head of the queue and the tail of the queue. However, such data must be accessed in mutual exclusion, while ZooKeeper does not provide such a mechanism.[3] For this reason, we have to solve the communication mechanism differently, relying on the atomicity provided by ZooKeeper when creating and naming znodes.

A queue in ZooKeeper can be represented by a regular znode, say *zn*, and its children. The name *zn* is the queue name and each child of *zn* represents an item in the queue. Therefore, creating or deleting a child to/from *zn* is in fact adding or removing a queue item. To produce an item, a znode child (with a unique sequence number appended to its name) is created under *zn*. This sequence number is incremented atomically by ZooKeeper whenever a child is created. This is done by using the *create* operation with *mode=SEQUENTIAL*. To consume an item from the queue, the client has to read all the children of *zn* using the *getChildren* operation, and sort them according to their sequence numbers (these sequence numbers are extracted from the names). The child with the smallest sequence number is the queue head. This child is then removed using the *delete* operation. Note that the *delete* operation may fail if in the meantime this child was removed by another client. If this happens, removing the next (smallest) child in the list has to be tried.

3.2 Queue example: Solving the synchronization problem

We discuss only the synchronization of a consumer when the queue is empty. If the queue is empty (*zn* has no child), then a *watch* is left on *zn* to get notified whenever *zn* is modified (child added). After notification, the clients execute again the above procedure (*getChildren*, *sorting* and *deleting*). If several consumers are blocked by an empty queue, all of them concurrently execute the procedure (but only one will succeed).

3.3 Queue example: discussion

From the above description, we see several sources of inefficiency in the queue implementation. When consuming an item from the queue, requiring the client to read all the queue items plus to order them is highly inefficient. When several consumer clients are blocked because the queue is empty, awaking all of them when an item is added to the queue is inefficient: all consumers — except one — will have to block again. In Section 5 we will experimentally measure the cost of this solution.

3.4 Why not locks or semaphores?

In Section 3.1 we have pointed out the absence of mutual exclusion mechanism in ZooKeeper (which is not a queue specific problem). This leads to the inefficient queue implementation that we described. Therefore, why not add locks or

[3] In [9], the authors of ZooKeeper suggest two ways to implement locks. Apart from the overhead of the solution, the crash of a client holding the lock is not handled properly (partial state update is not undone).

semaphores to ZooKeeper, in order provide mutual exclusion when needed, in order to allow for a more efficient implementation?

To answer this question, assume for a while that ZooKeeper provides locks with *lock/unlock* operations. A client could typically, using a *mutex* lock, execute the following operations: (1) *lock(mutex)*, (2) *read/write* znodes, (3) *unlock(mutex)*. The problem is the crash of the client between (1) and (3): ZooKeeper would have to handle the problem. For this reason, by design, ZooKeeper excludes providing locking mechanisms. ZooKeeper calls the lock operation a non *wait-free* operation, and requires that all operations provided to clients are wait-free.

4 Addressing the ZooKeeper inefficiency

To explain the solution, we start from the example introduced above, where a client has to execute the following sequence of operations:

$$\textit{lock(mutex)};\quad \textit{read/write znodes};\quad \textit{unlock(mutex)}.$$

As pointed out, this sequence of operations must be executed atomically (all or nothing property). This property is violated if the client crashes before executing *unlock(mutex)*. The problem is not solved using transactions: transactions also would require ZooKeeper to detect and handle the crash of clients, which is excluded by design. The solution we propose is to *send the whole sequence of operations to the server*. Even if the client crashes afterwards, the server is able to execute the full sequence of operations.

However, the solution requires the server to be multi-threaded. Indeed, assume a single-threaded server executing the operation *lock(mutex)*. If the operation blocks the server thread, the server cannot handle a future *unlock(mutex)* operation from another client. Unfortunately, a multi-threaded server leads to a non-deterministic execution, which is incompatible with state machine replication [3]. We address the problem by relying on a deterministic scheduler. In the rest of the section we describe first how we handle the execution of sequences of operations in a prototype server written in Java, and then we describe our deterministic Java thread scheduler.

4.1 Server request handling

Consider a client that wants to execute a sequence of operations on the server. The sequence of operation corresponds to a method of some Java object *obj* of class C. As an example, consider the methods *produce*, *consume* and object $Q1$ of class *Queue*. A request to *produce* item *value1* in queue *Q1* has the following format:

$$\textit{req1} = \{\textit{byteCode, Queue, Q1, produce, value1}\}$$

The first argument *byteCode* is the Java byte code of class *Queue*, which has two methods: *produce* and *consume*. The second argument *Queue* is the name of the class. The third argument *Q1* is the name of the object. The fourth argument *produce* is the name of the method invoked. The last argument *value1* is the parameter of the method called.

Assume that the server, upon receiving $req1$, has no code for class *Queue* and no object $Q1$ exists. After execution of *req1*, the server state is the following:[4]

- File */byteCodes/Queue.class* is created with content *byteCode*.
  In other words, the server maintains the byte codes (provided by clients) under the path */byteCodes/* in the file system hierarchy. As explained later, this allows the class code not to be sent with each client request.
- The *Queue* class is loaded into the running JVM;
- A reference to class *Queue* is entered in *classTable*, an in-memory table;
- File */Queue/Q1/* is created: */Queue/Q1/* contains file *headTail* with an index to the head of $Q1$ (initially 0) and an index to the tail of $Q1$ (initially 0).
- A reference to object *Q1* is entered in *objectTable*, another in-memory table;
- File *item-1* with content *value1* is created under */Queue/Q1/*; the index to the head and the index to the tail of *Q1* are both set to 1.

Next consider a second request, *req2*, delivered to the server:

$$req2 = \{byteCode,\ Queue,\ Q1,\ produce,\ value2\}$$

The server finds object $Q1$ in the in-memory *objectTable*, and executes the *produce* method. After execution the new server state is as follows:

- */Queue/Q1/* contains the new file *item-2* with content *value2*;
- file */Queue/Q1/headTail* contains now 2 as the head index.

Obviously, it is stupid to send the byte code of a class in each request. By default, in our implementation *byteCode* is not sent. When the server receives a request invoking a method of some class C, if */byteCodes/C.class* does not exist, then the server asks the client the byte code. The current implementation does not allow overloading: when some class C is loaded, a second class with the same name cannot be loaded.

4.2 Multi-threaded deterministic scheduling

Figure 1 shows the overall architecture of our server. The server executes the following tasks:

1. Reception of client requests;
2. Ordering of client requests by interacting with the other server replicas (atomic broadcast);
3. Execution of client requests;
4. Sending results to clients.

[4] Although we assume here the same hierarchical file space as in ZooKeeper, this is not necessary. The application state, in this case queue items and meta data, can be more efficiently maintained as class variables, i.e., static class fields shared among instances of a class.

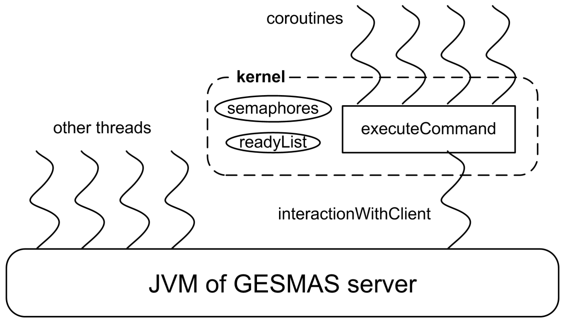

Fig. 1: GESMAS server architecture

Item 3 is usually done by one single thread, in order to ensure deterministic execution. In our server this is done by several threads that are scheduled deterministically, as we explain below. In Figure 1, *other threads* and *interactionWithClient* are scheduled by the JVM. The box called *executeCommand* represents a procedure called by the thread *interactionWithClient*. The box called *kernel*, dispatches deterministically the CPU allocated by the JVM to *interactionWithClient* to the threads that execute client requests. To differentiate these thread scheduled deterministically by the *kernel* from the threads scheduled by the JVM, we call the former *coroutines*. The *kernel* in Figure 1 implements also semaphores, which allow to synchronize client processes. Revisiting the queue example using semaphores, the *produce* method becomes:

$$P(semaphore_1);\quad deposit\ the\ item;\quad V(semaphore_2),$$

where $semaphore_1$ is initialized to the size of the queue, and $semaphore_2$ is initialized to 0. The *consume* method becomes:

$$P(semaphore_2);\quad consume\ an\ item;\quad V(semaphore_1).$$

In the rest of the section we present the coroutine library that we have used, the *interactionWithClient* thread, and finally the implementation of *semaphores*.

a) Coroutines

For coroutines, we use Java *Continuations* provided by *JavaFlow*, an Apache Common library [11]. A Continuation is like a coroutine: it has the following methods: *startSuspendedWith*, *run*, *continueWith* and *suspend*. The method *run* is the code executed by the Continuation. The method *startSuspendedWith* creates a Continuation and suspends it immediately. The method *continueWith(co)* transfers execution to the Continuation *co*: the execution of *co* resumes at the point where it was suspended. The *suspend* method suspends the running Continuation: if Continuation *co1* has executed *continueWith(co2)*, when *co2* executes *suspend*, the execution of *co1* continues immediately after *continueWith(co2)*. In the sequel we use term Coroutine instead of Continuation.

Algorithm 1 Interaction with client

```
 1: Global variables:
 2:   readyList initially empty          {List of coroutineWithRequest not blocked}
 3:   current initially null             {currently executing coroutineWithRequest}
 4:   requestWanted
 5:   req

 6: thread interactionWithClient
 7:   requestWanted = true
 8:   while true do
 9:     req = null;
10:     if requestWanted then
11:       req = get new client request
12:     end if
13:     executeCommand(req)
14:   end while
15: end

16: function executeCommand (req)
17:   if req ≠ null then
18:     current = new CoroutineWithRequest(req)
19:     continueWith(current.continueRef)
20:   else
21:     current = remove coroutine at the head of readyList
22:     continueWith(current.continueRef)
23:   end if
24:   if readyList is empty then
25:     requestWanted = true
26:   else
27:     requestWanted = false
28:   end if
29: end
```

b) Thread "interactionWithClient"

The thread *interactionWithClient* is given by Algorithm 1. The boolean variable *requestWanted* is true if the thread is ready to get a new request from a client; otherwise there are still client requests ready to be executed. Consider the queue example and request *consume* of client c_1 that is blocked due to an empty queue. Request *produce* of client c_2 unblocks the request of c_1. When the request of c_2 is fully handled, the request of c_1 can be resumed: no new client request needs to be requested. If *requestWanted*, then the thread *interactionWithClient* gets a new client request.

We discuss now *executeCommand(req)*. If *req* $\neq$ *null* a new coroutine is created (in suspended mode) to handle the request: the context of each coroutine is a request, namely the request that the coroutine handles, see Algorithm 2. Then the new coroutine starts executing (line 19).

Algorithm 2 Class CoroutineWithRequest

```
Instance variable:
    req                                                  {request handled by coroutine}
    coroutineRef

function CoroutineWithRequest (request)                  {constructor}
    req = request
    coroutineRef = Coroutine.startSuspendedWith (this)

function run():
    results = handleRequest(req)
    send(results, req.requestId)
```

If *req* is equal to *null*, the *readyList* is used. As indicated by its name, the *readyList* contains coroutines that are ready to be executed, i.e., that are not blocked. So, when *req* is equal to *null*, a coroutine is removed from the head of the *readyList*, and its execution is resumed.[5]

When executing, i.e., handling a client request, the coroutine can call a synchronization operation, namely P or V on a semaphore. Operation V can unblock a coroutine, in which case the unblocked coroutine is added to the *readyList*. Operation P may suspend the coroutine, in which case the reference of the coroutine is added to the waiting queue of the semaphore.[6] The execution of *suspend* resumes the execution of the thread *interactWithClient* at line 24 (Alg. 1). If the *readyList* is empty, then *requestWanted* is set to *true*, otherwise it is set to *false*, and the function *executeCommand()* terminates.

5 Performance evaluation

In this section we evaluate experimentally the cost of the ZooKeeper synchronization mechanisms and the alternate implementation we have described, called *GESMAS* (GEneric State Machine and Application Service). After describing our experimental setup we present the performance evaluation for two objects: the *queue* object (see Sect. 2 and 4) and the *barrier* object. We also use the queue object to study the herd-effect in the two solutions.

5.1 Setup

Our hardware setup consists of three machines with single-core 2.8 GHz cpu, 1GB RAM running Linux-2.6.18 which are connected via a 1 Gigabit/s Ethernet switch. On each machine we run one replica. Clients are also run on similar machines.

[5] In our implementation, the coroutine currently executing is not in the readyList.

[6] Note that a coroutine looses the CPU only when executing a blocking operation. In other words, there are no race issues among coroutines executing client operations.

We have used ZooKeeper release *3.3.4* (the latest stable release at the time of our experiments) with Java bindings. Our ZooKeeper server consists of three replicas. Our GESMAS state machine replication (with three replicas) is implemented using JPaxos [12], a Java implementation of Paxos.

5.2 Queue performance results

We did experiments to measure latency and throughput of *produce* and *consume* operations.

Latency: We measured the latency of the *produce* and the *consume* operations, including the latency as a function of the number of items[7] in the queue. All the measurements were done after a warm-up period. Figure 2 shows the latency

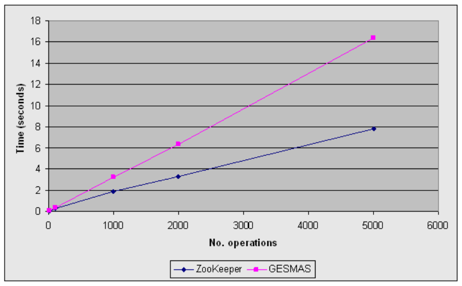

Fig. 2: Latency of *produce* operation.

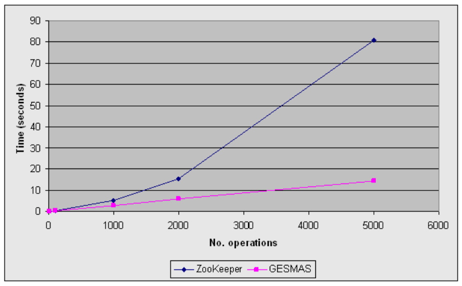

Fig. 3: Latency of *consume* operation.

measurements for the *produce* operation. To measure the cost of N *produce* operations, the client invokes the operation on the queue, waits until the response is received, performs the next *produce* operation, etc. This is repeated N times and the total duration is measured. We can see in Figure 2 that ZooKeeper has lower latency compared to GESMAS. This can be explained by the fact that the *produce* operation in ZooKeeper requires only the creation of one file, while in GESMAS it involves additionally to write to the *headTail* file.

The same latency measurements were done for the *consume* operation. To measure the latency of N consecutive *consume* operations, the queue was first filled with N items. The result is shown in Figure 3. The figure shows that the latency of the *consume* operation in ZooKeeper is much higher than the latency of the *produce* operation, and also much higher than the latency of the *consume* operation with GESMAS. This result is not surprising, considering the ZooKeeper implementation, where each *consume* operation consists of getting the list of all queue items and then performing a linear search (to locate the head item). This is in contrast to GESMAS, where the *consume* operation involves steps similar to those of the produce *operation*.

[7] Each item in the queue stores only one integer, hence very few bytes.

Finally, in the last latency experiment we studied the latency of the *consume* operations as a function of the number of items in the queue. In this experiment, we measured the latency of 1000 *consume* operations, for different initial queue sizes, starting with a queue size of 1000. The results appear in Figure 4: with GESMAS the latency is constant, while with ZooKeeper it increases linearly with the initial size of the queue.

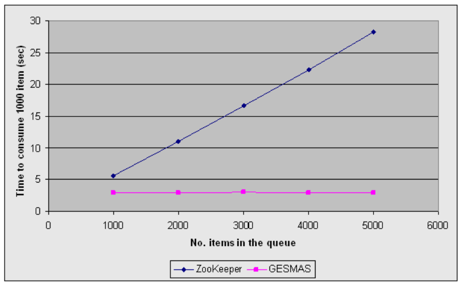

Fig. 4: Dependency to no. items in the queue. Latency of 1000 *consume* operation.

framework \ operation	*produce*	*consume*
ZooKeeper	1260	270
GESMAS	1041	1265

Table 1: Throughput (operations per second)

Throughput: In this experiment we measured the maximum number of *produce* and *consume* operations per second that can be executed on a queue. In order to generate high loads, we had to consider more than one client.

For *produce* operations, we measured the number of operations executed on the server during 1 second. The load was continuously increased by adding clients, and the number of operations per second was measured, until the maximum was reached. Table 1 shows the maximum throughput of 1260 for ZooKeeper and 1041 for GESMAS.

For *consume* operations, we could not proceed in the same way for ZooKeeper and GESMAS. For GESMAS, before running the experiments, we initialized the queue with "enough" elements such that *consume* would never block. Then, we did the measurements in the same way as for *produce* operations.

For ZooKeeper, the same procedure would lead to very bad results,[8] due to the fact that *consume* is very expensive if the number of the items in the queue is large. Therefore, we did the measurements differently. We considered four clients, two producers and two consumers, and a queue initially empty. All four clients were running on a different JVM and were started at the same time: each client sends a request, waits for the response, and then sends the next request immediately. We ran the same experiment also for GESMAS to show the relevance of the experiment. The results are shown in Figure 5. The figure shows, for both ZooKeeper and GESMAS, the number of operations per second, as a function of the total number of operations executed by each client, from 500

[8] About 40 operations per second when queue was filled with 10000 items and four clients.

to 5000.[9] Results for ZooKeeper in Figure 5 show the maximum of 442 at 1000 operations (combined *produce* and *consume*) per client. With four clients, this means 2000 ($= 4\cdot500$) *produce* and 2000 *consume*. Processing these 4000 requests took $4000/442 = 9$ seconds. From Table 1, 2000 *produce* take 1.6 seconds. This leads to 7.4 seconds (9secs−1.6secs) for 2000 *consume*, which means 270 *consume* operations per second, to be compared with the 1265 *consume* per second for GESMAS (see table 1).

We can perform a similar derivation for the *consume* operation with GESMAS, and compare the results with Table 1. Figure 5 shows a maximum of 1141 operations per second for GESMAS at 4000 operations per client. Having four clients (8000 produce and 8000 consume), that means that processing 16000 requests took 16000/1141=14 seconds. From Table 1, 8000 *produce* takes 7.7 seconds. This means in 6.3 seconds for 8000 *consume*, or 1269 *consume* operations per second. Table 1 reports 1265 *consume* operations per second.

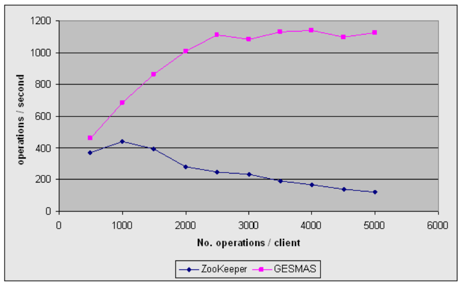

Fig. 5: Throughput of combined *produce*/*consume* operations.

N (clients)	ZooKeeper	GESMAS
50	498	212
100	1096	432
200	4144	810

Table 2: Time (ms) until N clients are unblocked.

Summary: The results for the *consume* operation point out the cost of the ZooKeeper approach. Even with a benchmark designed to penalize ZooKeeper as little as possible, the throughput is about five times smaller than with GESMAS.

The result for the *produce* operation show better performances for ZooKeeper than for GESMAS, which can be explained by the need to access an additional file. However, the small GESMAS overhead of *produce* operations is largely compensated by the ZooKeeper overhead of *consume* operations, and by the generality provided by GESMAS.

5.3 Herd effect experiment

The ZooKeeper queue implementation can cause another problem known as *thundering herd* effect.[10] The herd effect occurs when many processes or clients

[9] For example, in the first measurement, producers execute each 500 *produce* and consumers execute each 500 *consume*. The number of operations per second is obtained by dividing the $4\cdot500$ operations by the time it took the server to execute them.

[10] In [9] a recipe for a lock without herd effect has been proposed, which makes use of watch and other ZooKeeper API. Thus, one could argue that the problem is

are waiting (i.e. blocked) for an event, e.g., a resource to become available. When that event occurs, all processes are awaken but only one can proceed since processes compete for the same resource. Herd effect normally leads to huge overheads and is a serious source of inefficiency. To study this effect in ZooKeeper we did the following experiment.

First we created an empty queue. Then we started N clients in one JVM. Each client runs in a separate thread and has its own connection to the server. When started, each client invokes one *consume* operation followed by one *produce* operation. Hence at startup all N clients are blocked waiting for an item. Next, we produced one item, which causes one of the blocked clients to unblock. The unblocked client produces one item, which unblocks another client, and so on. We measured the time between the first produce operation until all N clients are unblocked. We did the same measurement with GESMAS. The results appear in Table 2.

We can observe that GESMAS performs clearly better than ZooKeeper. Moreover, the performance gap increases when increasing the number of clients. With GESMAS the time increases linearly with the number of clients, an expected result, considering that in GESMAS whenever an item is produced only one consumer is unblocked. As described in Section 4, this is achieved using semaphores. The time increase is not linear in the case of ZooKeeper. For example, increasing the number of client by a factor of 2 (from 100 to 200) leads to an increase of measured time by a factor of about 4 (from 1096 ms to 4144 ms).

5.4 Barrier object experiment

The second object we used for performance evaluation of synchronization mechanisms is a *barrier*. A barrier b is a synchronization object that enables a group of processes to block at b until all processes have reached barrier b. The barrier blocking operation is called here *wait*.

a) ZooKeeper barrier implementation: A barrier in ZooKeeper can be represented by a znode b. When invoking the *wait* operation, process p creates a child znode[11] under b. Processes can pass the barrier when the number of child znodes of b equals the barrier threshold. Otherwise they block using watches, until enough processes reach the barrier. Creation of each znode child under b triggers the watches, whereupon every process checks if the number of children has reached the barrier threshold. If yes, processes can pass the barrier, i.e., continue execution; otherwise processes set the watch again and wait for the next trigger.

solved. However, the proposed solution introduces a performance penalty, and not only because of the linear search it involves.

[11] The znode created is an *ephemeral* znode, which means that it is not stored on persistent memory.

b) GESMAS barrier implementation: In GESMAS, a barrier with name b, similarly to ZooKeeper, is represented by a file directory b. However, unlike ZooKeeper, there is only one file under b. This file contains a counter representing the number of client processes that have reached the barrier.[12] Synchronization is achieved using a semaphore, say $bSem$, with initial value 0.

When a client invokes the *wait* operation the counter is incremented. If the new value is smaller than the threshold the client blocks by invoking $P(bSem)$.[13] Otherwise the threshold has been reached, and $V(bSem)$ is invoked $threshold{-}1$ times, so that all blocked clients are unblocked and can pass the barrier.

c) Performance results: We performed the following 12 experiments. Each experiment is defined by a pair (c, b), where c represents the number of clients and b the number of consecutive barriers that each client has to pass, each barrier with threshold c. We did experiments with c equal 50, 100 and 200, and with b equal 200, 400, 800 and 1600. For each client, all *wait* invocations are done using the same connection to the server. The total accumulated time is reported in Table 3.

	ZooKeeper			GESMAS		
barrier threshold / no. barriers	50	100	200	50	100	200
200	12.6	41.2	138	3.2	6.1	12.9
400	24.7	78.1	266.2	6.3	12.4	25.6
800	42.2	140.6	497.1	12.7	24.5	53.6
1600	89.9	267.2	951.3	26.4	49.3	103.9

Table 3: Total accumulative time to pass all barriers (seconds)

As we can observe, GESMAS is significantly more efficient than ZooKeeper, and as the threshold gets larger the gap between ZooKeeper and GESMAS increases. This is not surprising, considering that with ZooKeeper at each *wait* invocation, all clients are awaken to check if the threshold has been reached. Almost always this condition is false, which leads to a high overhead (see also herd-effect experiment in Section 5.3). In GESMAS, synchronization is cheap: clients are awaken only when the threshold has been reached. Therefore, as expected GESMAS always performs better than ZooKeeper (by a factor of 4, 7 and 10 at thresholds of 50, 100 and 200 respectively).

6 Conclusive discussion

In the paper we have addressed an important source of inefficiency in the ZooKeeper coordination service with a radically different approach, which led us to develop GESMAS. We have concentrated on synchronization. Since in GESMAS client's

[12] As in GESMAS every operation is implemented by a class, the counter could be a class variable instead of a file.

[13] There is here no race issue to worry about, because of the coroutine schema for the execution of operations (see also Footnote 6).

request contains the object implementation (the byte code), any kind of (synchronization) object can be implemented. However, having clients send byte code to be executed on the server(s), raises a security issue. This problem is not difficult to address, as we now briefly sketch. A simple restrictive solution would be to only allow sending the code by well identified, trusted, expert, code developers. Another option can be signing the code. Note that the solution is not costly, considering the optimization mentioned at the end of Section 4.1, where by default the code is not sent (with the assumption that it has already been sent earlier). Another related issue is protection of resources accessed by the Java servers which can be controlled by sandboxing. One recent interesting sandboxing technique is presented in [13].

References

1. `http://hadoop.apache.org/zookeeper/`.
2. Lamport, L.: Using time instead of timeout for fault-tolerant distributed systems. ACM Trans. Program. Lang. Syst. **6** (April 1984) 254–280
3. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. **22**(4) (December 1990) 299–319
4. Guerraoui, R., Schiper, A.: Software-based replication for fault tolerance. Computer **30** (April 1997) 68–74
5. Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., Alonso, G.: Understanding replication in databases and distributed systems. In: In Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS2000. (2000) 264–274
6. MacCormick, J., et al.: Boxwood: abstractions as the foundation for storage infrastructure. In: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6. OSDI'04, Berkeley, CA, USA, USENIX Association (2004) 8–8
7. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th symposium on Operating systems design and implementation. OSDI '06, Berkeley, CA, USA, USENIX Association (2006) 335–350
8. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16** (May 1998) 133–169
9. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX conference on USENIX annual technical conference. USENIXATC'10, Berkeley, CA, USA, USENIX Association (2010) 11–11
10. Reed, B., Junqueira, F.: A simple totally ordered broadcast protocol. In: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware. LADIS '08, New York, NY, USA, ACM (2008) 2:1–2:6
11. `http://commons.apache.org/sandbox/javaflow/`.
12. Santos, N., Konczak, J., Zurkowski, T., Wojciechowski, P., Schiper, A.: Jpaxos: State machine replication based on the paxos protocol. Technical Report 167765, EPFL (July 2011)
13. Watson, R., et al.: A taste of capsicum: practical capabilities for unix. Commun. ACM **55**(3) (2012) 97–104