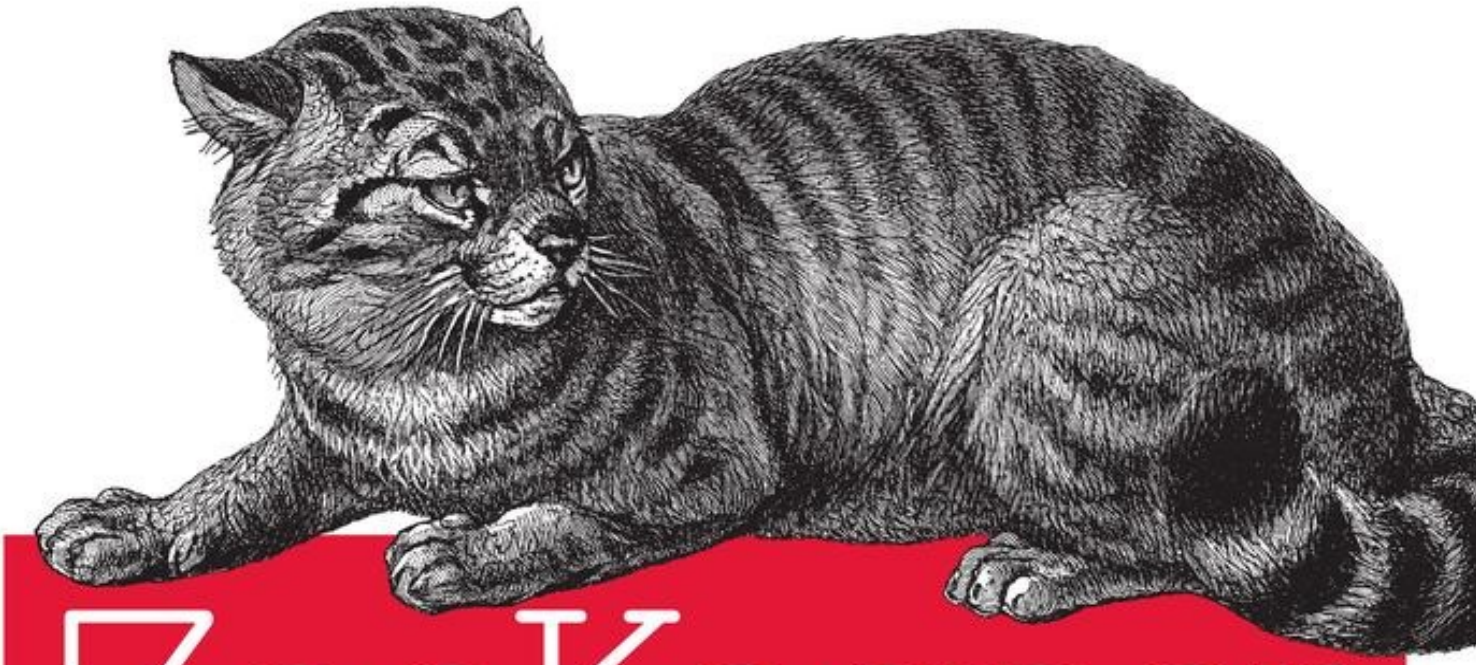
O'REILLY®



ZooKeeper

DISTRIBUTED PROCESS COORDINATION

Flavio Junqueira & Benjamin Reed

ZooKeeper

Flavio Junqueira

Benjamin Reed

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Special Upgrade Offer

If you purchased this ebook directly from oreilly.com, you have the following benefits:

- DRM-free ebooks—use your ebooks across devices without restrictions or limitations
- Multiple formats—use on your laptop, tablet, or phone
- Lifetime access, with free updates
- Dropbox syncing—your files, anywhere

If you purchased this ebook from another retailer, you can upgrade your ebook to take advantage of a
these benefits for just $4.99. Click here to access your ebook upgrade.

Please note that upgrade offers are not available from sample content.

Preface

Building distributed systems is hard. A lot of the applications people use daily, however, depend on such systems, and it doesn't look like we will stop relying on distributed computer systems any time soon. Apache ZooKeeper has been designed to mitigate the task of building robust distributed systems. It has been built around core distributed computing concepts, with its main goal to present the developer with an interface that is simple to understand and program against, thus simplifying th task of building such systems.

Even with ZooKeeper, the task is not trivial—which leads us to this book. This book will get you up t speed on building distributed systems with Apache ZooKeeper. We start with basic concepts that wil quickly make you feel like you're a distributed systems expert. Perhaps it will be a bit disappointing to see that it is not that simple when we discuss a bunch of caveats that you need to be aware of. But don't worry; if you understand well the key issues we expose, you'll be on the right track to building great distributed applications.

Audience

This book is aimed at developers of distributed systems and administrators of applications using ZooKeeper in production. We assume knowledge of Java, and try to give you enough background in the principles of distributed systems to use ZooKeeper robustly.

Contents of This Book

Part I covers some motivations for a system like Apache ZooKeeper, and some of the necessary background in distributed systems that you need to use it.

- Chapter 1, *Introduction*, explains what ZooKeeper can accomplish and how its design supports its mission.
- Chapter 2, *Getting to Grips with ZooKeeper*, goes over the basic concepts and building blocks. It explains how to get a more concrete idea of what ZooKeeper can do by using the command line.

Part II covers the library calls and programming techniques that programmers need to know. It is useful but not required reading for system administrators. This part focuses on the Java API because is the most popular. If you are using a different language, you can read this part to learn the basic techniques and functions, then implement them in a different language. We have an additional chapte covering the C binding for the developers of applications in this language.

- Chapter 3, *Getting Started with the ZooKeeper API*, introduces the Java API.
- Chapter 4, *Dealing with State Change*, explains how to track and react to changes to the state of

ZooKeeper.

- Chapter 5, *Dealing with Failure*, shows how to recover from system or network problems.
- Chapter 6, *ZooKeeper Caveat Emptor*, describes some miscellaneous but important considerations you should look for to avoid problems.
- Chapter 7, *The C Client*, introduces the C API, which is the basis for all the non-Java implementations of the ZooKeeper API. Therefore, it's valuable for programmers using any language besides Java.
- Chapter 8, *Curator: A High-Level API for ZooKeeper*, describes a popular high-level interfaces to ZooKeeper.

Part III covers ZooKeeper for system administrators. Programmers might also find it useful, in particular the chapter about internals.

- Chapter 9, *ZooKeeper Internals*, describes some of the choices made by ZooKeeper developers tha have an impact on administration tasks.
- Chapter 10, *Running ZooKeeper*, shows how to configure ZooKeeper.

Conventions Used in this Book

The following typographical conventions are used in this book:

Italic
: Used for emphasis, new terms, URLs, commands and utilities, and file and directory names.

`Constant width`
: Indicates variables, functions, types, parameters, objects, and other programming constructs.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user. Also used for emphasis command output.

`Constant width italic`
: Indicates placeholders in code or commands that should be replaced by appropriate values.

> **TIP**
>
> This icon signifies a tip, suggestion, or a general note.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at http://bit.ly/zookeeper-code.

This book is here to help you get your job done. In general, if example code is offered with this book you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*ZooKeeper* by Flavio Junqueira and Benjamin Reed (O'Reilly). Copyright 2014 Flavio Junqueira and Benjamin Reed, 978-1-449-36130-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information.

You can access this page at http://oreil.ly/zookeeper-orm.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Acknowledgments

We would like to thank our editors, initially Nathan Jepson and later Andy Oram, for the fantastic jo they did of getting us to produce this book.

We would like to thank our families and employers for understanding the importance of spending so many hours with this book. We hope you appreciate the outcome.

We would like to thank our reviewers for spending time to give us great comments that helped us to improve the material in this book. They are: Patrick Hunt, Jordan Zimmerman, Donald Miner, Henry Robinson, Isabel Drost-Fromm, and Thawan Kooburat.

ZooKeeper is the collective work of the Apache ZooKeeper community. We work with some really excellent committers and other contributors; it’s a privilege to work with you all. We also want to give a big thanks to all of the ZooKeeper users who have reported bugs and given us so much feedba and encouragement over the years.

Part I. ZooKeeper Concepts and Basics

This part of the book should be read by anyone interested in ZooKeeper. It explains the problems tha ZooKeeper solves and the trade-offs made during its design.

Chapter 1. Introduction

In the past, each application was a single program running on a single computer with a single CPU. Today, things have changed. In the Big Data and Cloud Computing world, applications are made up o many independent programs running on an ever-changing set of computers.

Coordinating the actions of these independent programs is far more difficult than writing a single program to run on a single computer. It is easy for developers to get mired in coordination logic and lack the time to write their application logic properly—or perhaps the converse, to spend little time with the coordination logic and simply to write a quick-and-dirty master coordinator that is fragile an becomes an unreliable single point of failure.

ZooKeeper was designed to be a robust service that enables application developers to focus mainly o their application logic rather than coordination. It exposes a simple API, inspired by the filesystem API, that allows developers to implement common coordination tasks, such as electing a master server, managing group membership, and managing metadata. ZooKeeper is an application library with two principal implementations of the APIs—Java and C—and a service component implemente in Java that runs on an ensemble of dedicated servers. Having an ensemble of servers enables ZooKeeper to tolerate faults and scale throughput.

When designing an application with ZooKeeper, one ideally separates application data from control o coordination data. For example, the users of a web-mail service are interested in their mailbox content, but not on which server is handling the requests of a particular mailbox. The mailbox conten is application data, whereas the mapping of the mailbox to a specific mail server is part of the coordination data (or metadata). A ZooKeeper ensemble manages the latter.

The ZooKeeper Mission

Trying to explain what ZooKeeper does for us is like trying to explain what a screwdriver can do for us. In very basic terms, a screwdriver allows us to turn or drive screws, but putting it this way does n really express the power of the tool. It enables us to assemble pieces of furniture and electronic devices, and in some cases hang pictures on the wall. By giving some examples like this, we can give a sense of what can be done, but it is certainly not exhaustive.

The argument for what a system like ZooKeeper can do for us is along the same lines: it enables coordination tasks for distributed systems. A coordination task is a task involving multiple processes Such a task can be for the purposes of cooperation or to regulate contention. Cooperation means that processes need to do something together, and processes take action to enable other processes to make progress. For example, in typical master-worker architectures, the worker informs the master that it i available to do work. The master consequently assigns tasks to the worker. Contention is different: it refers to situations in which two processes cannot make progress concurrently, so one must wait for the other. Using the same master-worker example, we really want to have a single master, but multip

processes may try to become the master. The multiple processes consequently need to implement *mutual exclusion*. We can actually think of the task of acquiring mastership as the one of acquiring a lock: the process that acquires the mastership lock exercises the role of master.

If you have any experience with multithreaded programs, you will recognize that there are a lot of similar problems. In fact, having a number of processes running in the same computer or across computers is conceptually not different at all. Synchronization primitives that are useful in the conte of multiple threads are also useful in the context of distributed systems. One important difference, however, stems from the fact that different computers do not share anything other than the network i a typical shared-nothing architecture. While there are a number of message-passing algorithms to implement synchronization primitives, it is typically much easier to rely upon a component that provides a shared store with some special ordering properties, like ZooKeeper does.

Coordination does not always take the form of synchronization primitives like leader election or locks. Configuration metadata is often used as a way for a process to convey what others should be doing. For example, in a master-worker system, workers need to know the tasks that have been assigned to them, and this information must be available even if the master crashes.

Let's look at some examples where ZooKeeper has been useful to get a better sense of where it is applicable:

Apache HBase
: HBase is a data store typically used alongside Hadoop. In HBase, ZooKeeper is used to elect a cluster master, to keep track of available servers, and to keep cluster metadata.

Apache Kafka
: Kafka is a pub-sub messaging system. It uses ZooKeeper to detect crashes, to implement topic discovery, and to maintain production and consumption state for topics.

Apache Solr
: Solr is an enterprise search platform. In its distributed form, called SolrCloud, it uses ZooKeeper to store metadata about the cluster and coordinate the updates to this metadata.

Yahoo! Fetching Service
: Part of a crawler implementation, the Fetching Service fetches web pages efficiently by caching content while making sure that web server policies, such as those in *robots.txt* files, are preserved This service uses ZooKeeper for tasks such as master election, crash detection, and metadata storage.

Facebook Messages
: This is a Facebook application that integrates communication channels: email, SMS, Facebook Chat, and the existing Facebook Inbox. It uses ZooKeeper as a controller for implementing sharding and failover, and also for service discovery.

There are a lot more examples out there; this is a just a sample. Given this sample, let's now bring th discussion to a more abstract level. When programming with ZooKeeper, developers design their applications as a set of clients that connect to ZooKeeper servers and invoke operations on them

through the ZooKeeper client API. Among the strengths of the ZooKeeper API, it provides:

- Strong consistency, ordering, and durability guarantees
- The ability to implement typical synchronization primitives
- A simpler way to deal with many aspects of concurrency that often lead to incorrect behavior in real distributed systems

ZooKeeper, however, is not magic; it will not solve all problems out of the box. It is important to understand what ZooKeeper provides and to be aware of its tricky aspects. One of the goals of this book is to discuss ways to deal with these issues. We cover the basic material needed to get the reade to understand what ZooKeeper actually does for developers. We additionally discuss several issues w have come across while implementing applications with ZooKeeper and helping developers new to ZooKeeper.

THE ORIGIN OF THE NAME "ZOOKEEPER"

ZooKeeper was developed at Yahoo! Research. We had been working on ZooKeeper for a while and pitching it to other groups, so we needed a name. At the time the group had been working with the Hadoop team and had started a variety of projects with the names of animals, Apache Pig being the most well known. As we were talking about different possible names, one of the group members mentioned that we should avoid another animal name because our manager thought it was starting to sound like we lived in a zoo. That is when it clicked: distributed systems *are* a zoo. They are chaotic and hard to manage, and ZooKeeper is meant to keep them under control.

The cat on the book cover is also appropriate, because an early article from Yahoo! Research about ZooKeeper described distributed process management as similar to herding cats. ZooKeeper sounds much better than CatHerder, though.

How the World Survived without ZooKeeper

Has ZooKeeper enabled a whole new class of applications to be developed? That doesn't seem to be the case. ZooKeeper instead simplifies the development process, making it more agile and enabling more robust implementations.

Previous systems have implemented components like distributed lock managers or have used distributed databases for coordination. ZooKeeper, in fact, borrows a number of concepts from these prior systems. It does not expose a lock interface or a general-purpose interface for storing data, however. The design of ZooKeeper is specialized and very focused on coordination tasks. At the sam time, it does not try to impose a particular set of synchronization primitives upon the developer, bein very flexible with respect to what can be implemented.

It is certainly possible to build distributed systems without using ZooKeeper. ZooKeeper, however, offers developers the possibility of focusing more on application logic rather than on arcane distributed systems concepts. Programming distributed systems without ZooKeeper is possible, but more difficult.

What ZooKeeper Doesn't Do

The ensemble of ZooKeeper servers manages critical application data related to coordination.

ZooKeeper is not for bulk storage. For bulk storage of application data, there are a number of options available, such as databases and distributed file systems. When designing an application with ZooKeeper, one ideally separates application data from control or coordination data. They often have different requirements; for example, with respect to consistency and durability.

ZooKeeper implements a core set of operations that enable the implementation of tasks that are common to many distributed applications. How many applications do you know that have a master or need to track which processes are responsive? ZooKeeper, however, does not implement the tasks for you. It does not elect a master or track live processes for the application out of the box. Instead, it provides the tools for implementing such tasks. The developer decides what coordination tasks to implement.

The Apache Project

ZooKeeper is an open source project hosted by the Apache Software Foundation. It has a Project Management Committee (PMC) that is responsible for management and oversight of the project. Only *committers* can check in patches, but any developer can contribute a patch. Developers can become committers after contributing to the project. Contributions to the project are not limited to patches—they can come in other forms and interactions with other members of the community. We have lots of discussions on the mailing lists about new features, questions from new users, etc. We highly encourage developers interested in participating in the community to subscribe to the mailing lists and participate in the discussions. You may well find it also worth becoming a committer if you want to have a long-term relationship with ZooKeeper through some project.

Building Distributed Systems with ZooKeeper

There are multiple definitions of a *distributed system*, but for the purposes of this book, we define it as a system comprised of multiple software components running independently and concurrently across multiple physical machines. There are a number of reasons to design a system in a distributed manner. A distributed system is capable of exploiting the capacity of multiple processors by running components, perhaps replicated, in parallel. A system might be distributed geographically for strategic reasons, such as the presence of servers in multiple locations participating in a single application.

Having a separate coordination component has a couple of important advantages. First, it allows the component to be designed and implemented independently. Such an independent component can be shared across many applications. Second, it enables a system architect to reason more easily about the coordination aspect, which is not trivial (as this book tries to expose). Finally, it enables a system to run and manage the coordination component separately. Running such a component separately simplifies the task of solving issues in production.

Software components run in operating system processes, in many cases executing multiple threads. Thus, ZooKeeper servers and clients are processes. Often, a single physical server (whether a standalone machine or an operating system in a virtual environment) runs a single application process although the process might execute multiple threads to exploit the multicore capacity of modern processors.

Processes in a distributed system have two broad options for communication: they can exchange

messages directly through a network, or read and write to some shared storage. ZooKeeper uses the shared storage model to let applications implement coordination and synchronization primitives. But shared storage itself requires network communication between the processes and the storage. It is important to stress the role of network communication because it is an important source of complications in the design of a distributed system.

In real systems, it is important to watch out for the following issues:

Message delays
: Messages can get arbitrarily delayed; for instance, due to network congestion. Such arbitrary delays may introduce undesirable situations. For example, process *P* may send a message before another process *Q* sends its message, according to a reference clock, but *Q*'s message might be delivered first.

Processor speed
: Operating system scheduling and overload might induce arbitrary delays in message processing. When one process sends a message to another, the overall latency of this message is roughly the sum of the processing time on the sender, the transmission time, and the processing time on the receiver. If the sending or receiving process requires time to be scheduled for processing, then th message latency is higher.

Clock drift
: It is not uncommon to find systems that use some notion of time, such as when determining the time at which events occur in the system. Processor clocks are not reliable and can arbitrarily dri away from each other. Consequently, relying upon processor clocks might lead to incorrect decisions.

One important consequence of these issues is that it is very hard in practice to tell if a process has crashed or if any of these factors is introducing some arbitrary delay. Not receiving a message from a process could mean that it has crashed, that the network is delaying its latest message arbitrarily, tha there is something delaying the process, or that the process clock is drifting away. A system in which such a distinction can't be made is said to be *asynchronous*.

Data centers are generally built using large batches of mostly uniform hardware. But even in data centers, we have observed the impact of all these issues on applications due to the use of multiple generations of hardware in a single application, and subtle but significant performance differences even within the same batch of hardware. All these things complicate the life of a distributed systems designer.

ZooKeeper has been designed precisely to make it simpler to deal with these issues. ZooKeeper does not make the problems disappear or render them completely transparent to applications, but it does make the problems more tractable. ZooKeeper implements solutions to important distributed computing problems and packages up these implementations in a way that is intuitive to developers. at least, this has been our hope all along.

Example: Master-Worker Application

We have talked about distributed systems in the abstract, but it is now time to make it a bit more concrete. Let's consider a common architecture that has been used extensively in the design of distributed systems: a master-worker architecture (Figure 1-1). One important example of a system following this architecture is HBase, a clone of Google's Bigtable. At a very high level, the master server (HMaster) is responsible for keeping track of the region servers (HRegionServer) available an assigning regions to servers. Because we don't cover it here, we encourage you to check the HBase documentation for further details on how it uses ZooKeeper. Our discussion instead focuses on a generic master-worker architecture.

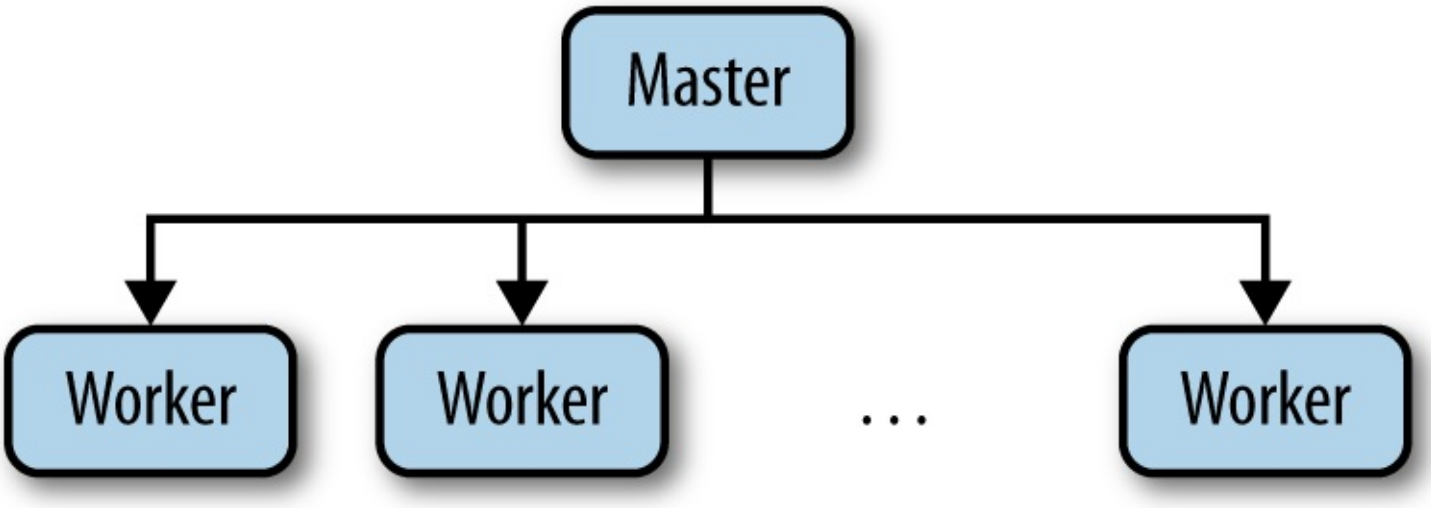


Figure 1-1. Master-worker example

In general, in such an architecture a master process is responsible for keeping track of the workers ar tasks available, and for assigning tasks to workers. For ZooKeeper, this architecture style is representative because it illustrates a number of popular tasks, like electing a master, keeping track c available workers, and maintaining application metadata.

To implement a master-worker system, we must solve three key problems:

Master crashes
: If the master is faulty and becomes unavailable, the system cannot allocate new tasks or reallocat tasks from workers that have also failed.

Worker crashes
: If a worker crashes, the tasks assigned to it will not be completed.

Communication failures
: If the master and a worker cannot exchange messages, the worker might not learn of new tasks assigned to it.

To deal with these problems, the system must be able to reliably elect a new master if the previous one is faulty, determine which workers are available, and decide when the state of a worker is stale with respect to the rest of the system. We'll look at each task briefly in the following sections.

Master Failures

To mask master crashes, we need to have a backup master. When the primary master crashes, the backup master takes over the role of primary master. Failing over, however, is not as simple as starting to process requests that come in to the master. The new primary master must be able to recover the state of the system at the time the old primary master crashed. For recoverability of the master state, we can't rely on pulling it from the faulty master because it has crashed; we need to hav it somewhere else. This somewhere else is ZooKeeper.

Recovering the state is not the only important issue. Suppose that the primary master is up, but the backup master suspects that the primary master has crashed. This false suspicion could happen because, for example, the primary master is heavily loaded and its messages are being delayed arbitrarily (see the discussion in Building Distributed Systems with ZooKeeper). The backup master will execute all necessary procedures to take over the role of primary master and may eventually star executing the role of primary master, becoming a second primary master. Even worse, if some workers can't communicate with the primary master, say because of a network partition, they may en up following the second primary master. This scenario leads to a problem commonly called *split-brain*: two or more parts of the system make progress independently, leading to inconsistent behavio As part of coming up with a way to cope with master failures, it is critical that we avoid split-brain scenarios.

Worker Failures

Clients submit tasks to the master, which assigns the tasks to available workers. The workers receive assigned tasks and report the status of the execution once these tasks have been executed. The master next informs the clients of the results of the execution.

If a worker crashes, all tasks that were assigned to it and not completed must be reassigned. The first requirement here is to give the master the ability to detect worker crashes. The master must be able t detect when a worker crashes and must be able to determine what other workers are available to execute its tasks. In the case a worker crashes, it may end up partially executing tasks or even fully executing tasks but not reporting the results. If the computation has side effects, some recovery procedure might be necessary to clean up the state.

Communication Failures

If a worker becomes disconnected from the master, say due to a network partition, reassigning a task could lead to two workers executing the same task. If executing a task more than once is acceptable, we can reassign without verifying whether the first worker has executed the task. If it is not acceptable, then the application must be able to accommodate the possibility that multiple workers may end up trying to execute the task.

EXACTLY-ONCE AND AT-MOST-ONCE SEMANTICS

Using locks for tasks (as with the case of master election) is not sufficient to avoid having tasks executed multiple times because we can have, for example, the following succession of events:

1. Master *M1* assigns Task *T1* to Worker *W1*.
2. *W1* acquires the lock for *T1*, executes it, and releases the lock.
3. Master *M1* suspects that *W1* has crashed and reassigns Task *T1* to worker *W2*.
4. *W2* acquires the lock for *T1*, executes it, and releases the lock.

Here, the lock over *T1* did not prevent the task from being executed twice because the two workers did not interleave their steps when executing the task. To deal with cases in which exactly-once or at-most-once semantics are required, an application relies on mechanisms that are specific to its nature. For example, if application data has timestamps and a task is supposed to modify application data, then a successful execution of the task could be conditional on the timestamp values of the data it touches. The application also needs the ability to roll back partial changes in the case that the application state is not modified atomically; otherwise, it might end up with an inconsistent state.

The bottom line is that we are having this discussion just to illustrate the difficulties with implementing these kinds of semantics for applications. It is not within the scope of this book to discuss in detail the implementation of such semantics.

Another important issue with communication failures is the impact they have on synchronization primitives like locks. Because nodes can crash and systems are prone to network partitions, locks can be problematic: if a node crashes or gets partitioned away, the lock can prevent others from making progress. ZooKeeper consequently needs to implement mechanisms to deal with such scenarios. First it enables clients to say that some data in the ZooKeeper state is *ephemeral*. Second, the ZooKeeper ensemble requires that clients periodically notify that they are alive. If a client fails to notify the ensemble in a timely manner, then all ephemeral state belonging to this client is deleted. Using these two mechanisms, we are able to prevent clients individually from bringing the application to a halt in the presence of crashes and communication failures.

Recall that we argued that in systems in which we cannot control the delay of messages it is not possible to tell if a client has crashed or if it is just slow. Consequently, when we suspect that a client has crashed, we actually need to react by assuming that it could just be slow, and that it may execute some other actions in the future.

Summary of Tasks

From the preceding descriptions, we can extract the following requirements for our master-worker architecture:

Master election
: It is critical for progress to have a master available to assign tasks to workers.

Crash detection
: The master must be able to detect when workers crash or disconnect.

Group membership management
: The master must be able to figure out which workers are available to execute tasks.

Metadata management

The master and the workers must be able to store assignments and execution statuses in a reliable manner.

Ideally, each of these tasks is exposed to the application in the form of a *primitive*, hiding completely the implementation details from the application developer. ZooKeeper provides key mechanisms to implement such primitives so that developers can implement the ones that best suit their needs and focus on the application logic. Throughout this book, we often refer to implementations of tasks like master election or crash detection as primitives because these are concrete tasks that distributed applications build upon.

Why Is Distributed Coordination Hard?

Some of the complications of writing distributed applications are immediately apparent. For example when our application starts up, somehow all of the different processes need to find the application configuration. Over time this configuration may change. We could shut everything down, redistribute configuration files, and restart, but that may incur extended periods of application downtime during reconfiguration.

Related to the configuration problem is the problem of group membership. As the load changes, we want to be able to add or remove new machines and processes.

The problems just described are functional problems that you can design solutions for as you implement your distributed application; you can test your solutions before deployment and be pretty sure that you have solved the problems correctly. The truly difficult problems you will encounter as you develop distributed applications have to do with *faults*—specifically, crashes and communication faults. These failures can crop up at any point, and it may be impossible to enumerate all the different corner cases that need to be handled.

BYZANTINE FAULTS

Byzantine faults are faults that may cause a component to behave in some arbitrary (and often unanticipated) way. Such a faulty component might, for example, corrupt application state or even behave maliciously. Systems that are built under the assumption that these faults can occur require a higher degree of replication and the use of security primitives. Although we acknowledge that there have been significant advances in the development of techniques to tolerate Byzantine faults in the academic literature, we haven't felt the need to adopt such techniques in ZooKeeper, and consequently we have avoided the additional complexity in the code base.

Failures also highlight a big difference between applications that run on a single machine and distributed applications: in distributed apps, partial failures can take place. When a single machine crashes, all the processes running on that machine fail. If there are multiple processes running on the machine and a process fails, the other processes can find out about the failure from the operating system. The operating system can also provide strong messaging guarantees between processes. All of this changes in a distributed environment: if a machine or process fails, other machines will keep running and may need to take over for the faulty processes. To handle faulty processes, the processes

that are still running must be able to detect the failure; messages may be lost, and there may even be clock drift.

Ideally, we design our systems under the assumption that communication is asynchronous: the machines we use may experience clock drift and may experience communication failures. We make this assumption because these things do happen. Clocks drift all the time, we have all experienced occasional network problems, and unfortunately, failures also happen. What kinds of limits does this put on what we can do?

Well, let's take the simplest case. Let's assume that we have a distributed configuration that has been changing. This configuration is as simple as it can be: one bit. The processes in our application can start up once all running processes have agreed on the value of the configuration bit.

It turns out that a famous result in distributed computing, known as *FLP* after the authors Fischer, Lynch, and Patterson, proved that in a distributed system with asynchronous communication and process crashes, processes may not always agree on the one bit of configuration.[1] A similar result known as *CAP*, which stands for Consistency, Availability, and Partition-tolerance, says that when designing a distributed system we may want all three of those properties, but that no system can handle all three.[2] ZooKeeper has been designed with mostly consistency and availability in mind, although it also provides read-only capability in the presence of network partitions.

Okay, so we cannot have an ideal fault-tolerant, distributed, real-world system that transparently take care of all problems that might ever occur. We can strive for a slightly less ambitious goal, though. First, we have to relax some of our assumptions and/or our goals. For example, we may assume that the clock is synchronized within some bounds; we may choose to be always consistent and sacrifice the ability to tolerate some network partitions; there may be times when a process may be running, b must act as if it is faulty because it cannot be sure of the state of the system. While these are compromises, they are compromises that have allowed us to build some rather impressive distributed systems.

ZooKeeper Is a Success, with Caveats

Having pointed out that the perfect solution is impossible, we can repeat that ZooKeeper is not going to solve all the problems that the distributed application developer has to face. It does give the developer a nice framework to deal with these problems, though. There has been a lot of work over th years in distributed computing that ZooKeeper builds upon. Paxos[3] and virtual synchrony[4] have bee particularly influential in the design of ZooKeeper. It deals with the changes and situations as they arise as seamlessly as possible, and gives developers a framework to deal with situations that arise that just cannot be handled automatically.

ZooKeeper was originally developed at Yahoo!, home to an abundance of large distributed applications. We noticed that the distributed coordination aspects of some applications were not treated appropriately, so systems were deployed with single points of failure or were brittle. On the other hand, other developers would spend so much time on the distributed coordination that they wouldn't have enough resources to focus on the application functionality. We also noticed that these applications all had some basic coordination requirements in common, so we set out to devise a

general solution that contained some key elements that we could implement once and use in many different applications. ZooKeeper has proven to be far more general and popular than we had ever thought possible.

Over the years we have found that people can easily deploy a ZooKeeper cluster and develop applications for it—so easily, in fact, that some developers use it without completely understanding some of the cases that require the developer to make decisions that ZooKeeper cannot make by itself. One of the purposes of writing this book is to make sure that developers understand what they need to do to use ZooKeeper effectively and why they need to do it that way.

---

[1] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process." *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, (1983), doi:10.1145/588058.58806

[2] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." *ACM SIGACT News*, 33:2 (2002), doi:10.1145/564585.564601.

[3] Leslie Lamport. "The Part-Time Parliament." *ACM Transactions on Computer Systems*, 16:2 (1998): 133–169.

[4] K. Birman and T. Joseph. "Exploiting Virtual Synchrony in Distributed Systems." *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, (1987): 123–138.

Chapter 2. Getting to Grips with ZooKeeper

The previous chapter discussed the requirements of distributed applications at a high level and argue that they often have common requirements for coordination. We used the master-worker example, which is representative of a broad class of practical applications, to extract a few of the commonly used primitives we described there. We are now ready to present ZooKeeper, a service that enables th implementation of such primitives for coordination.

ZooKeeper Basics

Several primitives used for coordination are commonly shared across many applications. Consequently, one way of designing a service used for coordination is to come up with a list of primitives, expose calls to create instances of each primitive, and manipulate these instances directly For example, we could say that distributed locks constitute an important primitive and expose calls t create, acquire, and release locks.

Such a design, however, suffers from a couple of important shortcomings. First, we need to either come up with an exhaustive list of primitives used beforehand, or keep extending the API to introduc new primitives. Second, it does not give flexibility to the application using the service to implement primitives in the way that is most suitable for it.

We consequently have taken a different path with ZooKeeper. ZooKeeper does not expose primitives directly. Instead, it exposes a file system-like API comprised of a small set of calls that enables applications to implement their own primitives. We typically use *recipes* to denote these implementations of primitives. Recipes include ZooKeeper operations that manipulate small data nodes, called *znodes*, that are organized hierarchically as a tree, just like in a file system. Figure 2-1 illustrates a znode tree. The root node contains four more nodes, and three of those nodes have nodes under them. The leaf nodes are the data.

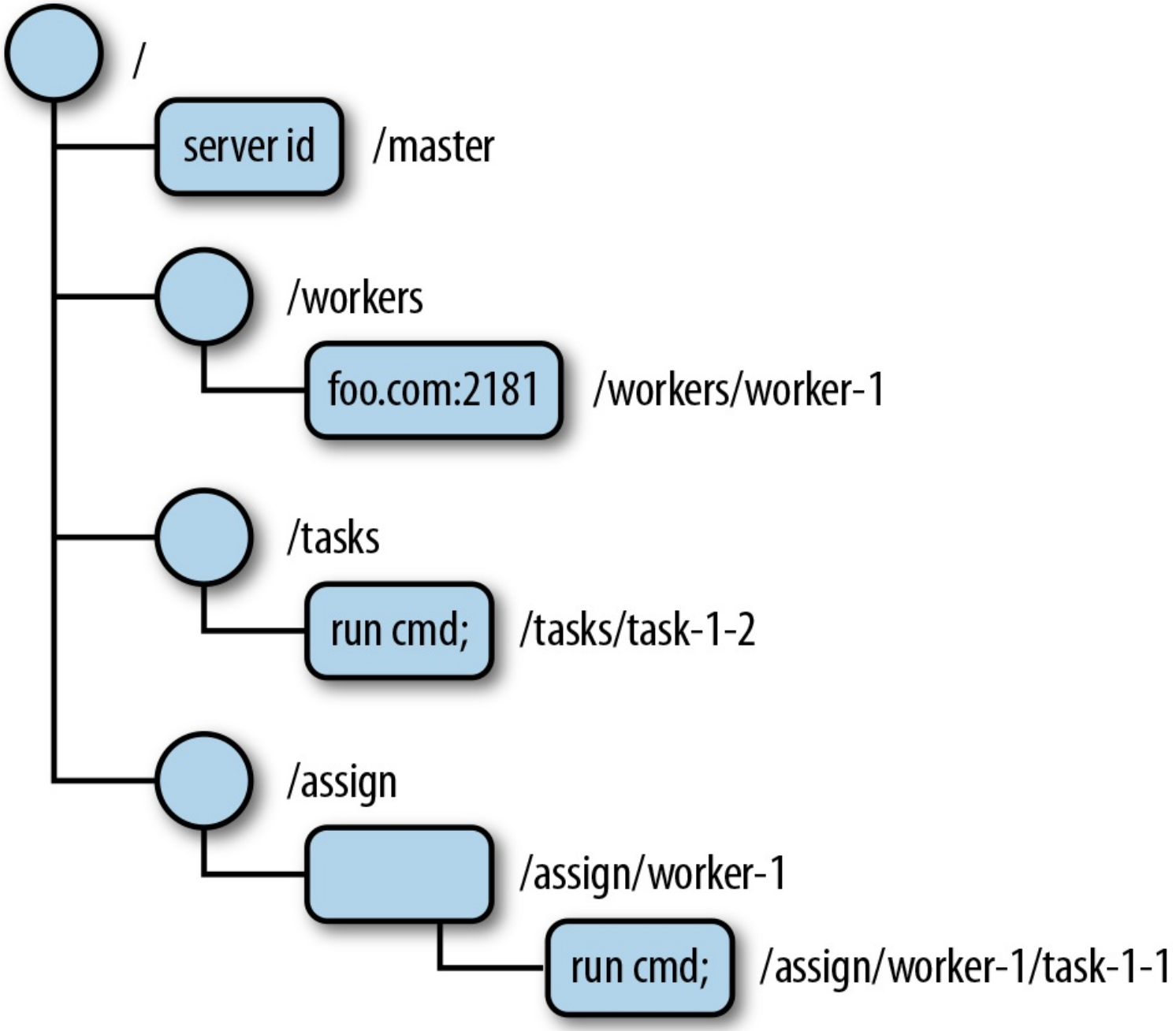


Figure 2-1. ZooKeeper data tree example

The absence of data often conveys important information about a znode. In a master-worker example for instance, the absence of a master znode means that no master is currently elected. Figure 2-1 includes a few other znodes that could be useful in a master-worker configuration:

- The `/workers` znode is the parent znode to all znodes representing a worker available in the system. Figure 2-1 shows that one worker (`foo.com:2181`) is available. If a worker becomes unavailable, its znode should be removed from `/workers`.

- The `/tasks` znode is the parent of all tasks created and waiting for workers to execute them. Clients of the master-worker application add new znodes as children of `/tasks` to represent new tasks and wait for znodes representing the status of the task.

- The `/assign` znode is the parent of all znodes representing an assignment of a task to a worker. When a master assigns a task to a worker, it adds a child znode to `/assign`.

API Overview

Znodes may or may not contain data. If a znode contains any data, the data is stored as a byte array. The exact format of the byte array is specific to each application, and ZooKeeper does not directly provide support to parse it. Serialization packages such as Protocol Buffers, Thrift, Avro, and MessagePack may be handy for dealing with the format of the data stored in znodes, but sometimes string encodings such as UTF-8 or ASCII suffice.

The ZooKeeper API exposes the following operations:

`create /path data`
: Creates a znode named with `/path` and containing `data`

`delete /path`
: Deletes the znode `/path`

`exists /path`
: Checks whether `/path` exists

`setData /path data`
: Sets the data of znode `/path` to `data`

`getData /path`
: Returns the data in `/path`

`getChildren /path`
: Returns the list of children under `/path`

One important note is that ZooKeeper does not allow partial writes or reads of the znode data. When setting the data of a znode or reading it, the content of the znode is replaced or read entirely.

ZooKeeper clients connect to a ZooKeeper service and establish a *session* through which they make API calls. If you are really anxious to use ZooKeeper, skip to Sessions. That section explains how to run some ZooKeeper commands from a command shell.

Different Modes for Znodes

When creating a new znode, you also need to specify a *mode*. The different modes determine how the znode behaves.

Persistent and ephemeral znodes

A znode can be either persistent or ephemeral. A *persistent* znode `/path` can be deleted only through