Lab 4: Zookeeper

Spring 2022

Contents

1 Introduction **2**
- 1.1 Zookeeper Overview . . . 2
- 1.2 Key Concepts in Zookeeper . . . 2
  - 1.2.1 Znodes . . . 2
  - 1.2.2 The Zookeeper Ensemble vs. Your Cluster . . . 2
  - 1.2.3 Ephemeral vs. Sequential Znodes . . . 3
- 1.3 Distributed Locks . . . 3

2 Zookeeper Setup **4**
- 2.1 Run Using Docker (Preferred) . . . 4
- 2.2 Local Install . . . 4

3 The Go Zookeeper Client **4**
- 3.1 Basic Operations . . . 4
- 3.2 Permissions . . . 5
- 3.3 Version Number of a Znode . . . 5
- 3.4 The Ephemeral & Sequential Flags . . . 5

4 Getting Started **5**

5 Your Task **5**
- 5.1 Reading & Updating a Counter . . . 5
- 5.2 Distributed Lock . . . 6

6 Handin **6**

1 Introduction

1.1 Zookeeper Overview

Zookeeper is a distributed coordination service by Apache. One common challenge when building a distributed application is managing centralized information without introducing a single point of failure. For example, consider building a distributed file system; we need a central source of truth on the directory structure — which files are in the system, and where — agreed on by all nodes. Of course, this information needs to be updated constantly: files are added, modified, and deleted all the time. Suppose we choose to store this information on a server: if this server fails, our entire system will too. If only the system storing this information can be distributed, and hence more fault-tolerant, too!

In fact, we have already seen a solution to this problem: passive replication. To avoid a single point of failure, we can use passive replication replicate our centralized information across multiple servers. Moreover, you have already implemented a system that accomplishes just this: Raft. In essence, Zookeeper is a service that achieves the same goal and provides an easy API. Under the hood, Zookeeper relies on a consensus protocol similar to Raft.

For the final CS1380 project, you will implement a distributed file-system called Puddlestore. You will use Zookeeper as a central source of truth, storing the structure of the file-system and critical metadata. To aid you in this task, the primary goal of this lab is to help you become familiar with Zookeeper and its Go client.

For more information, see here for a list of applications that use Zookeeper, and here for a blog on how Zookeeper is used at Twitter.

1.2 Key Concepts in Zookeeper

The Zookeeper Programmer's Guide is the definitive reference for design-level documentation of Zookeeper. Be sure to check it out for a deeper dive, but the key concepts relevant to this lab (and Puddlestore) are explained in this section.

1.2.1 Znodes

Zookeeper has a hierarchical name space structured like a file-system. Znodes are data registers in Zookeeper with a filepath-like name. For example, a znode may have the name `\a` and associated data "hello"; another may have the name `\a\b` and associated data "world". Naturally, `\a\b` is referred to as the *child* of `\a`. Given the file-system like structure of Zookeeper, we sometime use the word *path* to refer to the name of a znode (*path* and *name* are used interchangeably in both our assignments and the Zookeeper documentation).

Finally, we note that the data field in znode accepts *byte arrays* with size limit 1 MB, This means that the actual data can have any type: string, int, arrays, etc. Given the size limit, znodes are almost always used to store *metadata*, instead of actual data.

1.2.2 The Zookeeper Ensemble vs. Your Cluster

A cluster of Zookeeper servers is called an **ensemble**. Via a Raft-like protocol, they conduct leader elections in order to maintain consensus. On the other hand, you will be building distributed applications that *connect* to the Zookeeper ensemble. Specifically, each node in your application will connect to the Zookeeper ensemble, and it can be helpful to think of the Zookeeper ensemble as a centralized service.

For simplicity, both this lab and Puddlestore will actually only use one Zookeeper server; you can assume that the Zookeeper server has 100% uptime. In a real application, you will likely want

to spin up an ensemble of Zookeeper servers for fault-tolerance. (For those interested, docker-compose is an easy tool that you can use to spin up multiple servers at once; see the **via docker stack deploy or docker-compose** section here. If you do this, please remember to switch your implementation back to the single server default before you submit.)

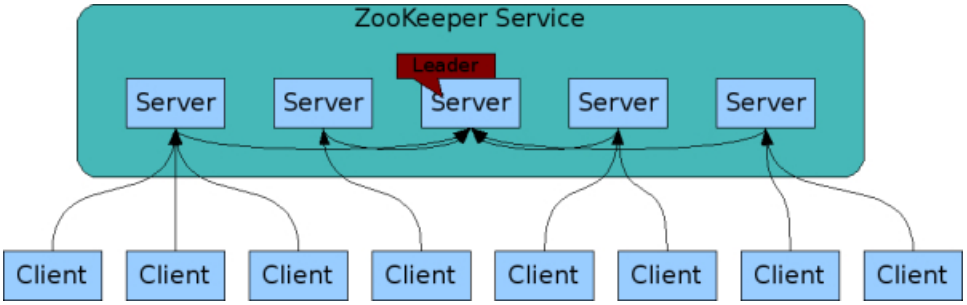

Figure 1: Relationship between the Zookeeper ensemble (labeled ZooKeeper Service) and nodes in your cluster (labeled Clients). The Zookeeper ensemble conducts leader elections internally, but from the perspective of your nodes, all Zookeeper servers are the same and they can contact any one. [2]

1.2.3 Ephemeral vs. Sequential Znodes

There are two flags that can be associated with a znode: ephemeral and sequential. They are neither required nor mutually exclusive.

Ephemeral znodes disappear when the Zookeeper connection used to create them drops. One use case for ephemeral znodes is maintaining cluster membership. Imagine a system like Tapestry that supports dynamic joining and leaving of nodes, and we'd like a way to know the list of nodes currently in our cluster. To do this, we can create a group znode and ephemeral children znodes under it that correspond to each node in our cluster. Concretely, we can create the non-ephemeral znode `\tapestry` and have tapestry nodes with IDs 001, 002, 003 create, upon joining the cluster, ephemeral znodes `\tapestry\001`, `\tapestry\002`, `\tapestry\003`, respectively. When asked, any node can list the children of the group znode and return the list of nodes currently in the cluster. If a node fails, its connection to Zookeeper will drop, and the ephemeral znode it created will disappear. As described, you will use Zookeeper as a cluster membership server in Puddlestore.

Sequential znodes are assigned a unique, monotonically increasing counter, appended to the end of their paths. The counter has 10 digits with 0 padding, e.g. 0000000001. One use case for sequential znodes is leader election. If we have each node in our cluster create an sequential znode, we can say: the node with the lowest sequence number is the leader. Since Zookeeper achieves consensus via an algorithm much like Raft, implementing the functionalities of Raft (i.e. leader elections) with Zookeeper is trivial. For more details on implementing leader elections with Zookeeper, see this recipe.

1.3 Distributed Locks

In a distributed file-system, what if two clients try to write to the same file concurrently? To avoid data corruption, we need distributed locks: before a client writes to a file, it needs to acquire a lock, and while the client is holding the lock, no other client can write to that file.

In essence, locks are merely another form of central information — in the form of, e.g. client X is currently reading file Y — that must be shared across all nodes. This is why Zookeeper is perfect implementing distributed locks.

[2] https://zookeeper.apache.org/doc/r3.6.3/zookeeperOver.html

In this lab, you will implement a distributed lock on the simplest piece of shared data: a counter. Zookeeper will be used to:

1. store the counter itself as information shared across all nodes
2. implement a distributed lock on the counter

These two tasks constitute the two parts of this lab, detailed in the Your Task section below.

2 Zookeeper Setup

2.1 Run Using Docker (Preferred)

Instead of manually installing the Zookeeper, we recommend using Docker to run Zookeeper. Docker packages code related to an application – in our case, Zookeeper – along with all its dependencies in a standardized way so that it can run easily on any machine. If you do not have Docker installed, you can follow these instructions.

To spin-up a Zookeeper instance using Docker, run:

```
docker run --rm -p 2181:2181 zookeeper
```

This command uses the publicly available Zookeeper Docker *image*. (You can think of a Docker image as containing all the information you need to run an application: code, config files, environment variables, libraries and runtimes.) Executing this command creates a Docker *container*: a process on your computer running Zookeeper according to the specifications in the image. The `--rm` flag tells Docker to cleanup after the container exits; `-p 2181:2181` tells docker to bind you local port 2181 to port 2181 of the container, which allows your code to use `localhost:2181` to access the Zookeeper instance.

More documentation on the Zookeeper Docker image can be found here.

2.2 Local Install

```
cd $HOME
wget https://downloads.apache.org/zookeeper/zookeeper-3.6.3/apache-zookeeper-3.6.3-bin.tar.gz
tar -xvzf apache-zookeeper-3.6.3-bin.tar.gz
cd apache-zookeeper-3.6.3-bin
cp conf/zoo_sample.cfg conf/zoo.cfg
bin/zkServer.sh start-foreground
```

If `wget` does not work for you, try `curl` instead.

```
curl -O https://downloads.apache.org/zookeeper/zookeeper-3.6.3/apache-zookeeper-3.6.3-bin.tar.gz
```

3 The Go Zookeeper Client

In this lab and in Puddlestore, we will use a Go Zookeeper client called **zk**. You can find its documentation here.

3.1 Basic Operations

Be sure to check out the basic read operations `Get`, `Exists`, `Children`, and the write operations `Create`, `Set`, `Delete`. Each read operation has a watch version (`GetW`, `ExistsW`, `ChildrenW`) that returns a channel; these channels act like watches and can be read from to receive updates on znode(s).

3.2 Permissions

Security is an important consideration when building any application, and Zookeeper provides built-in support for ACLs through the `acl` parameter to `Create`. Since access control isn't a focus of this class, you may use global permissions: `zk.WorldACL(zk.PermAll)` in both this lab and Puddlestore.

3.3 Version Number of a Znode

Each znode has a version number, which by default is incremented every time a znode is updated. Some applications use the version number for validation: when a client performs an update or a delete, it must supply the version of the data of the znode it is changing; if the version it supplies doesn't match the actual version of the data, the update will fail. Since neither the tasks in this lab nor Puddlestore require this validation, you can simply use `-1` as the version number to bypass version checking.

3.4 The Ephemeral & Sequential Flags

When creating a znode, you can pass in the flags `zk.Ephemeral` and/or `zk.Sequential`. To specify none, you can pass `0` into the flags parameter. To specify both, you can pass `zk.Ephemeral|zk.Sequential` into the flags parameter; alternatively, you can use the built-in function `CreateProtectedEphemeralSequential`.

4 Getting Started

Here is the link to the Github Classroom assignment. This assignment can be done individually or with your project partner [3].

5 Your Task

The code you must write is marked with `// TODO: students should implement this` comments.

5.1 Reading & Updating a Counter

Implement the following functions providing a basic `counter_client.go`:

```
func InitCounterPath(zkConn *zk.Conn) (err error)

func (cc *CounterClient) GetCount() (count int, err error)

func (cc *CounterClient) Inc() (err error)
```

After implementing these functions, your code should pass `TestBasic` provided in the stencil. This test checks that your counter functions are implemented correctly, without dealing with concurrency.

`TestConcurrentIncrement` creates 10 clients that call `Inc()` concurrently. Currently, this test should fail because these client operations are not executed in order.

[3] Download your assignment from Github Classroom. Don't clone our stencil repo. You have to work on the repo created by GitHub Classroom when you accept the homework. You will have to push your code to your own repo to get grades.

5.2 Distributed Lock

To fix this, we need to implement a distributed lock and use it to ensure that only one client can call `Inc()` at a time. Zookeeper provides a straightforward recipe for locks. On a high level, this recipe takes advantage of Zookeeper's ability to assign each child a unique sequence number and uses the rule: *the node that has the lowest sequences number has the lock*. The recipe is as follows:

Clients wishing to obtain a lock do the following:

1. Call `Create()` with a pathname of `<lock-root>/lock-` and the sequence and ephemeral flags set.
2. Call `Children()` on the lock root without setting the watch flag to obtain the list of clients—more precisely, the ephemeral nodes they created—waiting to access the lock root.
3. If the pathname created in step 1 has the lowest sequence number suffix, the client has the lock and the client exits the protocol.
4. The client calls `Exists()` with the watch flag set on the child that has the next lowest sequence number.
5. If `Exists()` returns `false`, the client associated with the lower sequence number child has released the lock, so we go to step 2 and try to acquire the lock. Otherwise, wait for a notification for the pathname from the previous step before going to step 2.

The unlock protocol is very simple: clients wishing to release a lock simply delete the node they created in step 1.

Based on this recipe, implement the following functions in `dist_lock.go`:

```
func (d *DistLock) Acquire() (err error)

func (d *DistLock) Release() (err error)
```

6 Handin

Once you finish the lab, you should submit to Gradescope from your Github repo. The autograder will run and give you feedback. You can resubmit the lab multiple times.

Feedback

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS138 document by filling out the anonymous feedback form.