ECE419S: Distributed Systems

Winter 2013

Assignment 4

Submission deadline: Thursday April 4^{th} 2013 11:59 PM

1 Introduction

In this assignment you will design and implement a distributed data processing framework. Ideally, we would like a generic framework, capable of executing any data processing job. However, for the purpose of this assignment, you will build a framework for reversing hashes using a dictionary attack. This can be viewed as a password cracking job – that is, assuming we obtained a password hash, it would be nice if we figured out the actual password.

A dictionary attack relies on a list of common words (dictionary). For a given password hash, a process computes the hash for each word in the dictionary and compares each word hash against the password hash. If a match is found, the respective word is the password we were looking for. As this search is usually computationally intensive, finding the password can be slow with a single process. Fortunately, the search can be easily ran in parallel by having multiple processes perform the search on different parts of the dictionary. In this assignment, you will deploy multiple worker processes on different machines to speed up the search. Following, we will refer to a password hash search as a Job.

2 Framework Components

The data processing framework should consist of the following components: ZooKeeper service, Client Driver, Job Tracker service, File Server service, Worker Pool. ZooKeeper is primarily used to achieve *coordination* among the other framework components. You have to implement all components in the framework, but for ZooKeeper. Figure 1 shows a high-level overview of the framework architecture. Following, we provide a description for the functionality of each component.

ZooKeeper: You will run the ZooKeeper service with a single server. The steps for setting up the ZooKeeper service are provided in Section 6. ZooKeeper should be used to coordinate the framework components. All components should be aware of *ZooKeeper's IP and port* – passed as arguments. Moreover, framework components should make use of ZooKeeper according to the requirements – as lookup service, for leader election, group membership, failure detection, etc.

Client Driver: Framework component that can run on User machines (ideally your laptop, but in this assignment all processes run on the *ug* machines). Provides a command line *user interface* for Users to submit Job requests and Job status queries. Each Job is identified by the password hash. The Client Driver uses ZooKeeper as lookup service to discover the IP address of the Primary JobTracker – see below for details about the Job Tracker service. All User requests are forwarded to the Primary JobTracker and replies are returned to Users. For instance for a new Job request submission, the Client Driver sends the request to the Primary JobTracker and informs the User that the Job has been submitted for processing. Job status queries should return either "job in progress" or "job finished" & the result – either the *password* if found, or *password not found*. Upon Primary JobTracker failure, the Client Driver should *transparently* reconnect to the new Primary JobTracker, by fetching the new Primary's IP address from ZooKeeper.

Job Tracker service: Fault tolerant service deployed as Primary/Backup – 2 JobTracker processes on 2 different machines. The 2 JobTracker processes rely on ZooKeeper for leader election (to decide the Primary). Upon failure of the current Primary JobTracker, the Backup JobTracker should take over as the new Primary. Primary JobTracker receives Job requests, Job status queries from Client Drivers. For each Job request, the Primary JobTracker splits the Job into Tasks to be processed by Worker processes in parallel – see below for details on the Worker Pool. A Task is composed of the password hash and a Dictionary partition id. For simplicity, you can assume that the JobTracker knows the Dictionary size (in number of words). Dictionary partition ids should be derived based on the number of words in a partition (e.g., 1,000 words per Dictionary partition). For example, if the Dictionary has 100,000 words, and each Dictionary partition has 1,000 words, partition ids are from 1 to 100. Partition ids are used by Workers to fetch corresponding Dictionary partition from the File Server service – see below for details.

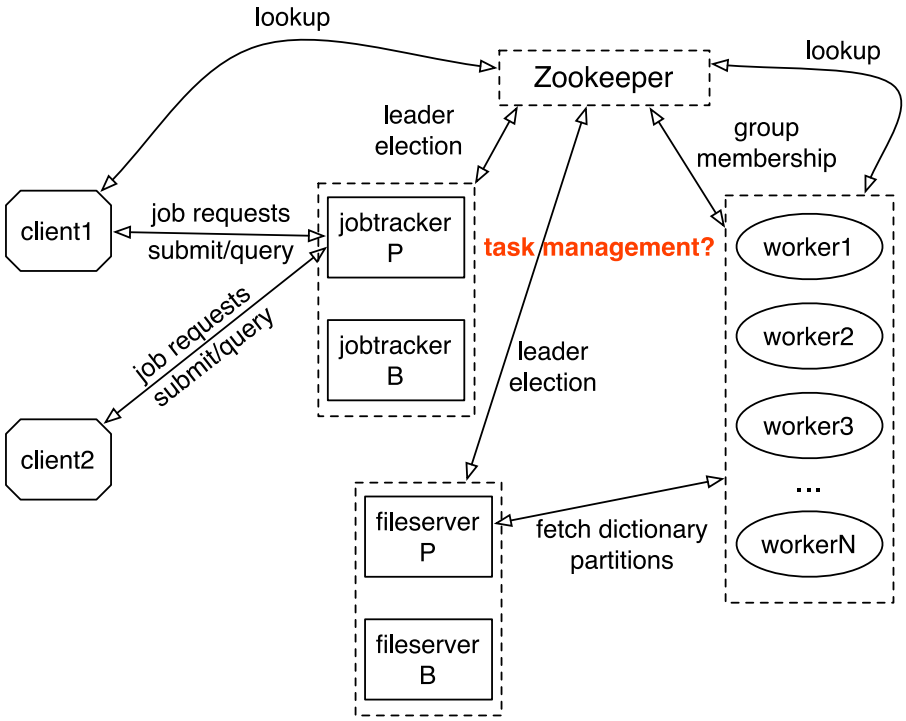

Figure 1: **Data Processing Framework Architecture.**

Furthermore, the JobTracker service should support concurrent Jobs from multiple Client Drivers/Users. That is, multiple Client Drivers should be able to use the data processing framework concurrently.

File Server service: Fault tolerant service deployed as Primary/Backup – 2 FileServer processes on 2 different machines. The 2 FileServer processes rely on Zookeeper for leader election (to decide the Primary). Upon failure of the current Primary FileServer, the Backup FileServer should take over as the new Primary. Each FileServer process stores the Dictionary file (passed as argument). The Dictionary can be read into memory on process startup. The Primary FileServer receives requests from Worker processes for various portions of the Dictionary (various Dictionary partitions, identified by partition ids). Primary FileServer sends Dictionary partitions to workers on worker requests.

Worker Pool: Multiple independent Worker processes on different machines searching for a hash match in parallel – parallel Job processing. Workers receive Tasks from the Primary JobTracker, directly or indirectly, depending on your design. For each Task, a Worker fetches the Dictionary partition associated with the Task from the Primary FileServer. Then, the Worker processes the Task, by computing the hash for each word in the Dictionary partition, and comparing the word hash against the password hash. When a match is found, the Worker should report the solution. Workers rely on ZooKeeper as a lookup service to discover the IP address/port of the Primary FileServer. Each Worker should transparently handle Primary FileServer failures/changes, through ZooKeeper. All Worker processes are more or less identical – there is no leader for the Worker Pool.

3 Design Issues

A number of design choices are suggested to you, at least at the high level – ZooKeeper as lookup service, fault tolerant Primary/Backup services using Zookeeper for leader election, failure transparency, Task structure (Dictionary partitions). However, some important design choices are not stated clearly, and you should come up with your own architecture. Namely, we do not specify how the management of Tasks should be done. Following we provide a couple of options for you to reason about and use as guidelines when designing your solution.

One option is to implement *stateful* JobTrackers, by having the Primary and the Backup JobTracker store Job related information, locally, in memory, using synchronous replication to handle failures. Workers can talk to the Primary JobTracker for fetching Tasks and returning Task results. The Primary JobTracker should also understand when Workers fail and Tasks are lost and need to be reassigned to other Workers. To achieve this, ZooKeeper can be used to maintain group membership for Workers. This design renders the JobTrackers a bit more complex, as they need to maintain replicated state.

Another possibility is to go with *stateless* JobTrackers and store all Job state in ZooKeeper. The Primary JobTracker is in charge of populating ZooKeeper with Job Tasks, and the Workers fetch Tasks from ZooKeeper. Workers also use ZooKeeper to update Task status or when a solution is found. Note that you can rely on watches for JobTrackers to notify Workers indirectly through ZooKeeper about new Tasks to be processed. Job status can also be stored in ZooKeeper, with the Primary JobTracker being notified when a solution is found. The advantage with stateless JobTrackers is a simple recovery.

4 Failures/Scaling

Your framework should sustain the following failure/scaling scenarios:

- Primary JobTracker failure – Backup JobTracker should take over and continue with Job requests from where the Primary left off; All the other components that interact with the Primary JobTracker (Client Drivers, Workers?) should transparently reconnect to the new Primary.
- Primary FileServer failure – Backup FileServer should take over and continue serving Worker requests for Dictionary partitions; All the other components that interact with the Primary FileServer (Workers) should transparently reconnect to the new Primary.
- Dynamic Worker addition/removal – for scaling (more parallelism) or when dealing with Worker failures. Your framework should make progress even if N-1 Workers fail, where N is the initial number of Workers. The Task management implementation should support this dynamic Worker pool. That is, when Workers fail, corresponding Tasks should be *reassigned*. In addition, when Workers are added, they should start processing Tasks, for an increased parallelism, and faster overall Job execution.

5 ZooKeeper

ZooKeeper is a high-performance coordination service for distributed applications. It exposes common services – such as naming (lookup), configuration management, synchronization, and group services – in a simple interface so you don't have to write them from scratch. You can use it off-the-shelf to implement group management, leader election, distributed locking, notifications, etc. And you can build on it for your own, specific needs. It is designed to be easy to program to, and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

The ZooKeeper service is intended to run on a set of several machines (3 or 5 are common), which prevents the loss of individual nodes from bringing down the cluster. All nodes in a ZooKeeper cluster provide the same consistent answer for a query, regardless of which ZooKeeper server you contact. Internally, ZooKeeper implements a quorum consensus algorithm. For this assignment, however, we will deploy a single ZooKeeper server (no replication), as we are mainly interested in building the rest of the data processing framework.

For more details about ZooKeeper, refer to the Lecture slides and the links below.

Useful links:
ZooKeeper Getting Started (You don't need to download anything – refer to Section 6 for setting up and using the ZooKeeper server):
`http://hadoop.apache.org/zookeeper/docs/current/zookeeperStarted.html`

ZooKeeper overview:
`http://hadoop.apache.org/zookeeper/docs/current/zookeeperOver.html`

ZooKeeper Programmer's Guide (Sample code has been provided to you for interacting with ZooKeeper. The code is written in Java and implements leader election. – refer to Section 6):
`http://hadoop.apache.org/zookeeper/docs/current/zookeeperProgrammers.html`

High-level ZooKeeper recipes:
`http://hadoop.apache.org/zookeeper/docs/current/recipes.html`

Tutorial showing simple implementations of barriers and producer-consumer queues using ZooKeeper:
`http://hadoop.apache.org/zookeeper/docs/current/zookeeperTutorial.html`

ZooKeeper internals:
`http://hadoop.apache.org/zookeeper/docs/current/zookeeperInternals.html`

6 ZooKeeper setup/Starter Code

You are required to use the *ug* computers in the GB 243 lab to implement and run the data processing framework for cracking passwords.

The starter source code and scripts are placed on the *ug* machines, under `${ECE419_HOME}/labs/lab4/`. `${ECE419_HOME}` is `/cad2/ece419s/`.

6.1 ZooKeeper setup

First, copy `myzk` under your `HOME` directory on the *ug* machines. To achieve this, issue:

```
cp -R /cad2/ece419s/labs/lab4/myzk $HOME/.
```

`$HOME` is your home directory environment variable

Then change directory to `myzk`, by issuing:

```
cd $HOME/myzk
```

Next you need to run a configuration script:

```
conf/INITCFG.sh
```

The above script creates the ZooKeeper server configuration file, `conf/zoo.cfg`, by assigning a random `clientPort` number for your server. You should use the `clientPort` to connect to your ZooKeeper server. The configuration file contains other parameters, such as `tickTime` or `dataDir`. `dataDir` is the location on disk where ZooKeeper

stores data, and it should be `/tmp/$USER/server1/data`, where `$USER` is your username on the *ug* machines.

Now you can start the ZooKeeper server, by issuing:

```
bin/zkServer.sh start
```

Once the ZooKeeper server is up and running, you can connect to it using a command line interface (this CLI can be used for debugging purposes):

```
bin/zkCli.sh -server ZooKeeperIPaddress:clientPort
```

`ZooKeeperIPaddress` is the IP address or hostname of *ug* machine you started the ZooKeeper server on, and `clientPort` is the random port the configuration script generated for you. Do a `cat conf/zoo.cfg` to find the port number.

Once connected to the ZooKeeper server, you can issue operations: `ls, create, delete`, etc. Use `help` to find the full list of commands.

6.2 ZooKeeper sample code

Copy the sample code provided, by issuing (you can copy the code anywhere in your `HOME` directory):

```
mkdir -p $HOME/lab4
cp -R /cad2/ece419s/labs/lab4/code $HOME/lab4/.
cd $HOME/lab4
```

We provide 2 examples written in Java on how to interact with the ZooKeeper server. Spend some time understanding these code examples.

The first example is located under `code/example1`. First, compile the code, by issuing `make`. Once you have the ZooKeeper server up and running, do `make runB`. You should see a usage message. Run `B` again, by specifying the ZooKeeper IP or hostname and `clientPort`. `B` waits for znode `/ece419` to be created. It sets a `watch` on the znode, and ZooKeeper will send an event to `B` when the znode is created. Now, in a different terminal/shell do `make runA`. Again, you need to re-run according to the usage message. `A` sleeps for 10 seconds, and then creates the `/ece419` PERSISTENT znode. When the znode is created, `B` receives the event.

The second example is located under `code/example2`. First, compile the code, by issuing `make`. Once you have the ZooKeeper server up and running, do `make run`. You should see a usage message. Run two processes of `Test` in two different terminals/shells, by specifying the ZooKeeper IP or hostname and `clientPort`. Both `Test` processes will sleep for 5 seconds, then check whether znode `/boss` exists with the `watch` set. Then, if the znode does not exist, the processes will try to create it with the EPHEMERAL flag set. Only one process will succeed. Let's call this guy *the boss*. Now both processes will block in an infinite loop – while(true). One is *the boss*, the other has a `watch` on the `/boss` znode. Now kill the process that's the current boss, by pressing CTRL+C in the corresponding shell. After a couple of seconds, the second process should receive an event that the znode was deleted. This second process now creates the znode and becomes *the new boss*. This second code sample is a simple implementation for leader election using ZooKeeper.

6.3 Hash function & Dictionary

As hash function you will use MD5. We have provided a sample class `MD5Test.java`, located under `code/md5`. `MD5Test.java` contains a sample method, `getHash`, to retrieve the MD5 hash from a given word (r41nb0w is the

hardcoded word value in the code). This method should be used by the Worker processes when computing hashes for Dictionary words. You can also use it to find out hash values for passwords, to be used as Job inputs when testing your framework. Another way to compute MD5 hashes for a given word is to issue in a terminal:

```
echo -n r41nb0w | md5sum
```

The Dictionary file is located under `code/md5/dictionary/lowercase.rand`. This file should be stored on the 2 FileServers. The Dictionary contains 265,744 possible password values (either English words, or variations obtained by replacing a with 4, e with 3, o with 0, i with 1, s with 5).

7 Deliverables

Each design team should produce a final design document of about 2 pages, where you describe your design choices and the overall framework architecture. The design document should build on the assignment handout and should detail each design aspect. *We are mainly interested in your design for task management and how you handle the failure scenarios.* As with Assignment 3, all members of a design team can discuss a common design, have the same design choices and produce the same design document.

Each *programming team* should implement the framework independently based on the design. You can implement the assignment in any programming language. However, the sample code is in Java and we recommend you stick with Java as well. The goal is to understand fundamental concepts and come up with a solid framework design. The programming language is less important. The final version of the design document, `design.pdf`, should be submitted together with the code by each *programming team* by the assignment deadline. In the design document, include the names and student IDs of the submitting programming team members (as well as the names and student IDs of the larger design team members).

You must implement your solution to:

1. Be usable – users can submit jobs and query job status
2. Be functional according to the high-level specifications
3. Handle listed failure scenarios
4. Support scaling by adding more workers
5. Support concurrent job executions

8 Submission

Only one team member needs to submit the assignment. Include the names and student numbers for your programming team members in a `README` file. Your submission should consist of the `design.pdf` document, source files, `Makefile`, scripts for running the framework, and the `README`.

Your submission must be in the form of one compressed archive named `Lastname1.Lastname2.tar.gz`. The directory structure of the archive should be as follows:

```
Lastname1.Lastname2.Lab4/
   design.pdf (don't forget to include names, student IDs for all your design team members)
   README (short description of how to run your code and other aspects you feel worth mentioning)
   Makefile
   <source files>
   scripts
```

Once you have a compressed archive in the `.tar.gz` format, you may submit your solution by the deadline using the submitece419s command, located under /local/bin on the *ug* machines:

```
/local/bin/submitece419s 4 Lastname1.Lastname2.tar.gz
```