ECE419 - Lab 4 Design Report

Tahia Khan 998897216
Kevin Gumba 997585117
Submitted April 3, 2014 (Early Bonus)

1 Overview

This system implements a distributed data processing framework for unhashing passwords with the use of ZooKeeper for coordination, leader election, group membership and failure handling. The four main components, the Client Driver, the Job Tracker, the File Server and the Workers are discussed in detail below.

2 Client Driver

2.1 Submitting Tasks

The client submits a task (ie, a job/query) by entering the following into the command line interface:

- To submit a job:job [hash]
- To submit a query: status [hash]

The ClientDriver places the task into ZooKeeper under the path /task/[task_id] and sets the data on the task's znode to "client_id;request_type;hash". The JobTracker watches the /task znode to detect new jobs and queries. The client watches for responses to queries under /task/[task_id]/response.

3 Job Tracker

3.1 Job Handling

Clients submit tasks, which can be either jobs or queries, under the /task znode in ZooKeeper. The JobTracker watches the /task znode and whenever the number of children changes (ie, new tasks are submitted), the JobTracker handles the incoming task located at /task/[task_id]. If the task is a job, it is placed on the path /jobs/[hash] so that Workers may handle it appropriately. If the task is a query, the JobTracker checks /results/[hash] to see if a success or failure has been reported by a Worker in the data field of this znode. The result is reported to the client by placing it under /task/[task_id]/result. Query tasks are removed from ZooKeeper as soon as the client is finished reading the result, and job tasks are removed from ZooKeeper when clients are no longer waiting on the job (ie, the requesting client disconnects). In this way, entries in ZooKeeper are always being cleaned.

Some efficiency considerations are implemented for the case where multiple clients request jobs on the same hash. Since jobs are identified by the hash, this event means that multiple clients are submitting the same job at the same time. This event is detected by keeping track of how many clients are waiting on any particular job by keeping track of a "use count" in the data field of each /jobs/[job name], which is incremented if clients submit the same job and decremented when waiting clients disconnect or quit. Resources are saved this way since multiple Workers will never process data for the same hash.

3.2 Leader Election/Failure Handling

Upon startup, the JobTracker checks for the existences of the /tracker znode. If it does not exist, then there are no other trackers currently in the system and this JobTracker makes itself the primary by creating the znode /tracker/primary in ZooKeeper. If the /tracker node does exist however, the JobTracker must check for the existence of a primary tracker at /tracker/primary. If a primary currently exists, this JobTracker sets itself as the backup and puts a watch on the primary. If the primary JobTracker fails, the backup watch will detect that the /tracker/primary znode is deleted and the backup will set itself as the new primary.

4 File Server

4.1 Leader Election/Failure Handling

Zookeeper is used to determine leader election between file server primary or backup. On startup, all file servers connect to the Zookeeper. Using Zookeeper, the primary file server creates a /fserver at ephemeral state. Backup file servers wait until the primary file has been deleted. On file server exit of failure, the /fserver is deleted and one backup file server is elected as the next primary via Zookeeper.

The hostname and port of the primary file server is stored as data within /fserver. This allows all workers to determine the location of the primary file server.

4.2 Scaling and Dictionary partition

The primary file server creates a socket for incoming partition packets. Workers connect to the file server providing information about their ID and the number of workers currently present. The primary file server calculates which part of the dictionary to be partitioned by dividing the indexes equally (partition size = size of dictionary / number of workers). An ArrayList is chosen to handle library partitions since it allows indexing of each word in the dictionary and easily determine sublists. The partition ID is signified by the index on the ArrayList.

5 Worker

5.1 Concurrent Job Executions

When the Job Tracker places a new job within the /jobs znode via Zookeeper, a worker handler thread is spawned. The worker handler thread creates an ephemeral node at /jobs/[job name]. The ephemeral node is used by clients to post their completion in traversing their partition. If all workers post a "-1" within their node, meaning no workers found the hash code, an agreement is reached and the workers post a "fail" on the /results/[job name].

5.2 Dynamic worker join/exit

On startup, workers register within /workers via Zookeeper using ephemeral state. This is done to keep track of group membership. Workers watch /workers in order to determine any changes in group membership. If a worker joins, exits, or fails, the workers shall reconnect to the file server and request for a new dictionary partition to evenly distribute the workload.

5.3 Failure Handling

In the event where workers fail, those remaining shall request a new partition to ensure that all parts of the dictionary are traversed by checking that all workers at the start of the job are the same workers at the end of the job. This is done by keeping track of any changes in /workers. To ensure that failed workers are handled, workers wait until the job is complete and a "success" or "fail" before exiting.

In the event where the file server fails, the workers shall try and reconnect via polling. The following events are kept monotonic to ensure robust file server handling: a worker must retrieve the hostname and port from Zookeeper, connect to the server, request a partition, and receive a partition.