

Assignment 3: Fault Tolerance

Due date: Thursday November 25th at 11:00pm Waterloo time

ECE 454/751: Distributed Computing

Instructor: Dr. Wojciech Golab
wgolab@uwaterloo.ca

A few house rules

Collaboration:

- groups of 1, 2 or 3

Managing source code:

- do keep a backup copy of your code outside of ecelinux
- do not post your code in a public repository (e.g., GitHub free tier)

Software environment:

- test on eceubuntu and ecetesla hosts
- use ZooKeeper 3.4.13, Curator 4.3.0, Thrift 0.13.0, Java 1.11
- Guava is provided with the starter code
- no other third party code or external libraries are permitted
- **do not use oneway RPCs for replication**

Overview

- In this assignment, you will implement a fault-tolerant key-value storage system.
- A partial implementation of the key-value service and a full implementation of the client are provided in the starter code tarball.
- Your goal is to add primary-backup replication to the key-value service using Apache ZooKeeper for coordination.
- ZooKeeper will be used to solve two problems:
 1. determining which replica is the primary
 2. detecting when the primary crashes
- This assignment is worth **18% of your final course grade.**

Learning objectives

Upon successful completion of the assignment, you should know how to:

- interact with ZooKeeper using the Curator client
- create znodes in the ZooKeeper tree, including ephemeral and sequence znodes
- list the children of a znode
- query the data attached to a znode
- use watches to monitor changes in a znode
- analyze linearizability in a storage system that supports get and put operations

Step 1: set up ZooKeeper

- You do not need to set up your own ZooKeeper service since one is provided for you on **manta.uwaterloo.ca** on the default TCP port (2181). Therefore, you may skip ahead to step 2.
- You are welcome to set up ZooKeeper yourself. It is easiest to configure the service in standalone mode, meaning that only one ZK server is used. A production ZK cluster would use 3+ servers.
- Before launching ZK, modify the configuration file `zookeeper-3.4.13/conf/zoo.cfg`, particularly the **dataDir** and **clientPort** properties. Set `dataDir` to a subdirectory of your home directory. Also modify the `clientPort` to avoid conflicts with classmates.
- You may start and stop the standalone ZooKeeper by running `zookeeper-3.4.13/bin/zkServer.sh [start|stop]`

Step 2: create a parent znode

- You will need to **create a ZooKeeper node manually** before you can run your A3 code.
- If you set up ZooKeeper yourself then update the settings.sh script with the correct ZooKeeper connection string.
- **Create your znode using the provided script:**
./createznode.sh
- All the scripts are configured to use the same znode name, which defaults to **\$USER** (i.e., your Nexus ID, which is stored in the environment variable \$USER on ecolinux hosts).
- The Java programs accept the znode name as a command line parameter. **Do not hardcode the znode name!**

Step 3: study the client code

- The starter code includes a client in the file A3Client.java. **Your server code must work with this specific client.**
- The client determines the primary replica by listing the children of the designated znode. This is the same znode you created in Step 2.
- The client sorts the returned list of children in ascending lexicographical order, and **identifies the smallest child as the znode denoting the primary replica.** The client then parses the data from this node to extract the hostname and port number of the primary, and sets a watch.
- If the primary fails, the client receives a notification and executes the above procedure again to determine the new primary.

Step 4: write the server bootstrap code

- On startup, each server process must contact ZooKeeper and create a child node under a parent znode specified on the command line. This parent znode must be the same as the one queried by the client to determine the address of the primary.
- The newly created child znode must have both the **EPHEMERAL** and **SEQUENCE** flags set. Furthermore, the child znode must store as its data payload a host:port string denoting the address of the server process.
- The server whose child znode has the smallest name in the lexicographic order is the primary. The other server (if one exists) is the secondary or backup.

Step 5: add replication

- At this point, the key-value service is able to execute get and put operations, but there is no primary-backup replication.
- To implement replication, it is critical that each server process knows whether it is the primary or backup. This can be done by querying ZooKeeper, similarly to the client code.
- The primary server process must implement **concurrency control** above and beyond the synchronization provided internally by the ConcurrentHashMap.
- If in doubt, use a Java lock for concurrency control. Do not implement locking using a ZooKeeper recipe as that will make the code unnecessarily slow! Also **do not store all the key-value pairs in ZooKeeper.**

Step 6: implement recovery from failure

- If the primary server process crashes, the backup server process must **detect automatically** that the ephemeral znode created by the primary has disappeared.
- At this point, the backup must become the new primary, and begin accepting get and put requests from clients. The provided client code will automatically re-direct connections to the new primary.
- The new primary may execute without a backup for some period of time immediately after a crash failure until a new backup is started.
- When the new backup is started, it must **copy all the key-value pairs** over from the new primary to avoid data loss in the event that the new primary fails as well.

Step 7: test thoroughly

To test your code, run an experiment similar to the following:

1. Ensure that ZooKeeper is running, and create the parent znode.
2. Start primary and backup server processes.
3. Launch the provided client and begin executing a long workload.
4. Wait two or more seconds, and kill the primary or the backup.
5. Wait two or more seconds, and start a new backup server.
6. Repeat steps 4 and 5 for several iterations.

The key-value service should continue to process get and put operations after each failure, including between steps 4 and 5 when the new primary is running temporarily without a backup. The client may throw exceptions in step 4, but there should be **no linearizability violations**.

Packaging and submission

- All your Java classes must be in the default package.
- You may use multiple Java files but do not change the name of the client (A3Client) or the server (StorageNode) programs, or their command line arguments.
- Do not change the implementation of the client at all.
- You must modify the server code to complete its implementation.
- You may add new procedures to **a3.thrift**, but do not add services.
- Use the provided package.sh script to create a tarball for electronic submission, and upload it to the appropriate LEARN dropbox before the deadline.

Grading scheme

Evaluation structure:

Correctness of outputs (linearizability):	60%
Performance (throughput):	40%

Penalties will apply in the following cases:

- solution uses oneway RPCs for replication, and hence assumes that the network is reliable
- solution cannot be compiled or throws an exception during testing despite receiving valid input
- solution produces incorrect outputs (i.e., non-linearizable executions)
- solution is improperly packaged
- you submitted the starter code instead of your solution

Hints and tips for success

1. We will test your code with 1-2 server processes at a time. This means **one primary and at most one backup replica**.
2. Throughput of more than **25,000 ops/s** is achievable on ecelix hosts with 8 client threads and with a backup active.
3. Test with both small data sets (e.g., 1000 key-value pairs) and larger data sets (e.g., 1M key-value pairs), and spread your processes across multiple ecelix hosts. Do not run everything on one machine!
4. Be prepared to handle frequent failures (e.g., as in [slide 11](#)). Each failure event may terminate either the primary or the backup.
5. Failures will be simulated on linux using **kill -9**.
6. Be prepared to handle port reuse (e.g., primary fails, and is restarted as a backup on the same host with the same RPC port).