

# CSE 123b Spring 2005

## Project 2 Description

### Introduction

The goal of this assignment is to learn some of the commonly used techniques for achieving fault tolerance in distributed systems. One of the fundamental problems in this space is ensuring that services remain available in the face of failure.

The service you will be building attempts to collect the results of a poll from a set of clients. Periodically, each client will transmit its decision on a vote to the service. Each vote will be uniquely identified by a sequence number and will simply consist of clients voting yes or no on that sequence number. The vote will be initiated by the server requesting a vote from each client on a particular sequence number (e.g., ballot or initiative). The service will consist of exactly two replicas, each receiving a separate copy of the vote over a TCP connection. Each replica goes through the same logic to transmit a summary of the vote back to all clients, e.g., 55% of the clients voted yes. The primary also transmits a copy of the vote to the backup. While the backup does not transmit vote results to the clients, note that it has the exact same information as the primary and hence could transmit the vote results at any time.

### Client

It is fine for a given client to receive multiple copies of the results for a given vote. The vote results should have a globally unique identifier such that clients can easily detect the reception of a duplicate result. Each client should log the results of each vote to a local file. There should be no duplicate or missing entries from the file. The format of the log file should be as follows:

```
<Vote ID> <Yes Count> <No count> \n
```

That is each line should contain the vote sequence number, followed by the number of Yes votes, followed by the number of No votes followed by a newline. Each field on the same line should be delimited with spaces. Finally, you can assume that clients are dependable, meaning they will never crash and the link between clients and servers will never go down.

### Servers

In all cases, there will be two replicas of the server, a primary and a backup. Each will spawn a separate thread to exchange heartbeat messages with the other. These heartbeats might be exchanged once a second as a form of keep alive: when receiving a keep alive each instance of the server can assume that the other is still up and functioning correctly. If one machine does not receive a heartbeat for a specified minimum of intervals, e.g, 5 intervals, then it assumes that the other has failed.

If the primary detects the failure of the backup, its responsibility is to restart the backup. For simplicity, you may assume that the primary and backup are running on the same machine, but listening on different ports. In general of course, the primary and backup may be running on different machines across a cluster or even across the wide area Internet. Thus, restarting the backup will consist of appropriate calls to `fork()/exec()` and having the backup execute the appropriate join protocol. The primary must also inform the clients of the identity (e.g., IP address/port) of the new backup once the backup has restarted and executed a join protocol. Once the backup is restarted, clients may transmit their subsequent votes to the new backup.

If the backup detects the failure of the primary, it elevates itself to the status of primary and takes over the job of disseminating vote results to clients. For instance, if clients are still waiting for the results of a given vote after the failure of the primary, the newly elevated primary will transmit these results to all clients.

In general, servers are heavy weight pieces of software that cannot be restarted immediately. Hence, we will assume that a newly started backup takes a fairly long amount of time to restart (30 seconds, though you may use shorter values for testing). While a new backup is starting, the primary will inform the clients that it is the only person that should be collecting votes. Once the backup has successfully executed the join protocol, the primary will inform clients of the identity of the new backup and all votes will once again go to the primary and backup.

You may assume that the primary and the backup never fail simultaneously. Further, we assume that none of the participants, the clients or the servers, are malicious. However, you cannot make the assumption that the link between the primary and backup is always dependable. A valid scenario would be for a communication loss to happen between the primary and backup, and both the primary and backup would assume that the other has failed and try to become the new primary.

The same piece of code should be used to implement both the primary and the backup. There should simply be a switch to determine whether a particular server is acting in "primary"- or "backup"-mode at any particular point in time.

## Protocol

To summarize, here are the communication protocols that you need to design and implement as part of this assignment:

- The process of transmitting a vote from a client to both the primary and the backup from individual clients.
- Transmitting vote summaries from the primary to all clients.
- The keep alive messages between the primary and the backup.
- The join protocol for a new replica joining an existing primary as a backup.
- Communication of backup/primary identity from the primary to the clients in the face of failure.

## Milestones

For this assignment, the design of the protocol governing the interaction of the various components will be critical. Thus, we will proceed with this assignment in two parts as outlined below:

Part 1, due April 25, 2005 at 5 pm. You will submit a detailed description of the protocol that you will use for communication among the various entities. You may encode your protocol any way that you like, but consider using XML and/or XMLRPC for the communication as a learning exercise to gain some exposure to this technology. You should also have completed clients that transmit a vote to both a primary and a backup, wait for a response, write the response to disk, and then move on to the next vote. You do not need to have any failure detection, joins, redirection to particular primaries/replicas implemented. However, the protocol for all such communication needs to be fully documented. Include any relevant pictures to describe this communication process.

Part 2, due May 6, 2005 at 5 pm. You will complete the fully functional replicated service including all of the protocols mentioned above. You should also submit updated documentation reflecting your final protocol, including why you settled on the design that you did, how to run your system, etc.