Assignment 1: Paxos-Based Reliable and Consistent Storage

Distributed Systems Class

March 30, 2012

In order to build an highly available service such as Facebook, one of the most important building blocks is to provide a way in which the underlying data is replicated and can be accessed even in the presence of failures. To this end, this assignment will build a replicated storage system based on Paxos to implement a narrow sliver of facebook functionality, i.e., making updates on a facebook wall page. The focus of this assignment will be on the backend Paxos processing rather than the front-end details of RPCs, so as to be manageable for a single person project.

1 The Storage System

Assume that the storage is organized as follows. Each facebook user has an associated file containing all of the wall posts, either by the user or by the friends of that user. The goal is to replicate these facebook "pages" (or files) on multiple machines and allow updates to the files such that all replicas contain the same sequence of updates. (Why would this matter in a real system? Because you would want causally related updates to be ordered in a consistent way across all replicas. Users would then see the same page irrespective of which replica they are redirected to and irrespective of failures to replicas.)

2 Communication Interface

Again, in order to minimize the development effort and focus your work just on Paxos, we will make the requirements for the communication interface extremely simple. You are to implement two RPC calls:

- Unreliable wall update: This RPC call takes two arguments: the name of the person whose wall is to updated and a string (of bounded length) that comprises of the status update. This RPC call can have any one of the following outcomes: (a) it updates all of the replicas using Paxos and returns success, (b) it updates all of the replicas using Paxos, but the response could be lost, (c) it fails completely and does not make any updates. The only requirement is consistency and not reliability; i.e., if an update is made, it is made consistently across all replicas.
- Unreliable wall fetch: This RPC call takes two arguments: the name of the person whose wall is to be read and an argument k that specifies the k^{th} update from the person's wall that needs to be fetched. This RPC call can have one of two outcomes: (a) it returns successfully with the response being the k^{th} update or with an error that the page does not contain k updates, or (b) it can fail silently without any response.

3 Consistent and Fault-tolerant Storage

Build a distributed storage system comprising of at least five replicas. Each replica maintains all of the files corresponding to the facebook users. Any replica can be contacted using the above two RPC calls, at which point it does the following:

- Update: if it is an update operation, the replicas run a Paxos round to determine an unique update number for the update. If two replicas are contacted simultaneously by different clients that each desire to make an update on a given page, then each update is assigned a non-deterministic but consistent update number. The file corresponding to the user is then updated with the status updates appended to the file in order and consistently across all replicas.
- Fetch: the fetch operation also needs to perform a Paxos operation to determine the k^{th} entry in a facebook page and responds to the request with the value of the entry if there are k entries. Note that a Paxos round is again required for this operation, because a replica might not be up-to-date with respect to all of the updates that have been made. If it were to respond to a request based on stale local state, it might erroneously respond with the result that the page does not contain k status updates.

You can follow the protocol from "Paxos made simple" paper by Leslie Lamport (which is available on the class website). You can organize your code in whatever mechanism that you think is appropriate. For example, you could have a single entity that serves all three roles of proposer, acceptor, and listener. While we have provided with a Java framework, you are welcome to use the language of choice, but with the following constraint:

- The message passing layer should have a configurable parameter p_l that controls the lossiness of messages sent through a channel. We would like you to demonstrate that your code works for various values of p_l.

4 Deliverables

4.1 Code

Students will need to provide the source code that implements the above requirements. The source code should be well-documented in order to ease grading and prevent misunderstandings about functionality. Compile/execution scripts must be included as well, and you can assume that they will run on support managed machines in the lab from the directory just above the jars/ and proj/ directories. The groups will meet with the TA to demonstrate the code and explain how it works.

4.2 Write-up

In addition to the source code, a major component of your grade will be based on a project write-up, in which your group should explain the protocol and any relevant implementation details. You should include, at a minimum, the following:

- a detailed description of your group's implementation
- any assumptions you made
- how to use your implementation

- any outstanding issues
- anything else that you feel is important to discuss (e.g. quirks, interesting behavior, performance characteristics, etc)