EECS 591 – Distributed Systems – Winter 2018 Team maximum size:2
Implementation project
Due: Monday, February 26, 2018, 11:59:59pm

Implementing Paxos

In this project, you are asked to implement a replicated chat service based on Paxos. The service is a simple one: clients connect to the service and send chat messages. The service replicas run Paxos to agree on the order in which messages should be delivered. Once the order of a message is determined (i.e. the sequence number of the corresponding client command has been chosen), the replicas "execute" the command; that is, they add it to the chat log. The requirements on the service are simple:

1. **Safety**: All replicas should add the client messages to the chat log in the same order. For convenience, you can have the replicas print the chat log every time they add a new message to it. The service should maintain safety despite asynchrony and despite any number of failures.
2. **Liveness**: The service should remain available despite the failure of **f** replicas, as long as messages are delivered synchronously.

Failure and network model We are only considering benign failures in an asynchronous network where messages may be dropped, reordered or delayed arbitrarily.

Parameters Your implementation should allow for the number of tolerated failures to be configured. In particular, each replica should be provided with:

1. f, the number of tolerated failures.
2. A unique ID (0, 1, …, 2f)
3. The (IP, port) pairs of all replicas in the system
4. Any other parameters you need for specific test cases (e.g. skip slot, message loss, etc.)

Details For convenience, you should use a view-based approach to leader election. That is, replica 0 is initially the primary. If at some point other replicas think the primary is faulty, replica 1 will become the primary and so on.

Each replica and client should be a **separate process**. You don't have to run each process on a separate machine, but they should be able to run on different machines, if necessary---i.e. don't bind them all to localhost.

When a client makes multiple requests, each request should be identifiable by a *client sequence number*. This should not be confused with the sequence number that the replicas use to determine the order of execution.

A client should wait for a response to its current request before being able to send another request.

Deliverable and demonstration Send us your source code, along with instructions on how to use it, by the deadline. We will schedule a time for each team to demonstrate that their code is working properly and answer questions about it.

Your code should work in two modes:

- Script mode: a single script that spawns 2f+1 replicas, given an input configuration file. After the replicas have started, the script should start a configurable number of batch mode clients (see below). In script mode, you should not print the chat log every time, but you should have a way to demonstrate that the replicas' logs are consistent. For example, you could have a special client request that forces replicas to print a hash of their logs.
- Manual mode: in this mode, we should be able to run the replica processes separately.

Language choice You can choose the language of implementation. However, if you choose to implement in a language other than C, C++, Java or Python, you need to run it by us first---and you need to have a good reason for it.

Test cases Here are some test cases we will try during demonstration. You should make sure that your code can handle these cases (at least).

1. Normal operation. Clients propose requests and all requests are executed in the right order. The clients should have a batch mode where they propose a request as soon as they receive the response for the current request (when enough clients are present, this will create concurrently submitted requests, which are otherwise hard to emulate by typing). This will increase the size of the chat log, so don't have the replicas print the chat log every time they add a new message to it any more. Instead, they can write it to a log file, print the

log periodically, or even print a hash of the log at periodic, pre-defined intervals. As long as we have a way of telling whether the chat logs are identical, any approach is fine.

2. The primary dies. The system should successfully detect the failure, and the next replica in the "line of succession" should become the new primary.
3. The primary dies, a new primary is elected and the new primary dies, too. We should be able to repeat the view change process as many as **f** times.
4. The skipped slot. You should have a special parameter that forces the primary to "skip" the proposal for sequence number **x** and propose a value for **x+1** instead. Sometime after sequence number **x+1** is learned, the primary dies, and the new primary is responsible for detecting the "holes" and taking appropriate action.
5. Simulating message loss: you should have a parameter **p** for each process (replicas and clients) that causes it to randomly drop **p**% of its outgoing messages. Your code should remain live for small values of **p**.

Bonus points

Use stable storage and an appropriate recovery mechanism to be able to recover replicas after they fail. In that case, we should be able to kill an infinite number of replicas; as long as f+1 replicas are up and the network is synchronous, the service should be safe and live.