Daniel Kudrow & Dev Nath
CS 271 Final Project
Dec. 9, 2014 10:00am

Chimera: Distributed Bank Account Manager

In this document we describe the design and implementation details of our project. Chimera can be cleanly separated into logically distinct modules for communication, multi-paxos, replicated log and persistence. We will treat each of these in turn.

Communication

For communication we chose synchronous messaging with HTTP. The primary reason was to simplify failure detection. Because HTTP has well defined failure modes we can, for instance, distinguish between a node that is down (unable to connect) and one that is busy (request timeout). A second advantage of HTTP is ease of implementation. We used the Flask framework for building RESTful APIs to handle message passing. Flask is multi-threaded, queuing requests as they arrive and handling them asynchronously. This gives us the best of both worlds: the channels behave as though they are synchronous but in reality messages are queued and handled on separate threads.

We constructed a thin message passing layer over Flask by specifying message channels as routes on the HTTP server. As an example, all Paxos messages are exchanged over the '/paxos' route. Messages are simply JSON encoded objects allowing the messaging layer to be used by all other modules.

Multi-Paxos

Our multi-Paxos is a straightforward extension of basic Paxos. We simply extract the state information of a Paxos and store one copy for each instance. If the Paxos module detects that a majority of the nodes are down, it retries at random intervals until nodes come back online and it is able to proceed. We use random sleeps between prepare phases to prevent live-lock.

Replicated Log

Each operation in the log is chosen by a Paxos run initiated by the node at which the log lives. A node's log is not updated until it receives a deposit or withdraw operation. When a node receives an operation, it attempts to insert it in the last slot of the log by initiating a Paxos run. If another node has already chosen a value for the last slot, the slot is populated and the insert fails. The node advances the log and tries to insert the operation again. In this way, Chimera lazily populates the log when it receives an operation.

Persistence

Chimera stores two types of data on disk in order to recover from crash failures: log data and Paxos state. The log data is simply the serialized version of the nodes' log. This is written to disk every time the node updates the log. Recall that a node only updates its log

when it receives a operation so the stored log is not necessarily up to date. When the node comes back online, it will fill in the remaining slots once it receives an operation. The Paxos state is stored to make this recovery possible. When a node recovers from a crash, it must be able to participate in all Paxos runs that have not been completed by all nodes.

Optimization Simultaneous Deposits.

The ordering of consecutive deposits is unimportant as they are non-conflicting. In the event that deposits are requested simultaneously at multiple nodes, they can be chosen simultaneously and entered into the log in any order. Our implementation of Paxos detects when multiple nodes are vying for a slot in the prepare phase. A node can read from the accepted values returned by the other nodes whether a majority consensus has been reached. If a majority has not been reached and some of operations are deposits, they can be grouped into a single value and used in the accept phase. This saves a round of Paxos for each simultaneous deposit received. Because they do not conflict, the group of deposits can than be placed in the log in any order.

It should be noted however that this circumstance is quite rare as it depends on a delicate interweaving of prepare and accept phases by different nodes. Specifically, for each distinct simultaneous deposit at least one of it’s accepts must be preceded by a higher prepare from another node.