Multi-Decree Paxos Implementation

Matthew Nappo[1]

University of Rochester mnappo@u.rochester.edu

1 Implementation and Assumptions

My implementation makes the following assumptions:

1. A majority never fails
2. There is only one leader proposer, and it never fails
3. If an acceptor or proposer fails, it may rejoin the network successfully
4. New acceptors/proposers/learners cannot join the network after it was initially created

Based on these assumptions, my implementation provides the following features:

1. Any number of Acceptors/learners
2. All acceptors are also learners
3. Any non-majority of acceptors/learners can fail at any time, and rejoin later on (as long as they have the same port)
4. If the leader does fail, the network will not be usable by clients while the leader is down, but the network state will remain. If the leader recovers (with knowledge of who the acceptors are), then the network will become usable again as if it were never down. Key-Value pairs inserted before the failure will be available after the leader goes back online.

My implementation `main` functions (and hence four kinds of processes):

1. Client: responsible for connecting to the leader to make get/put requests on the network. Entrypoint: `src/agents/client.rs`
2. Proposer: responsible for listening from client connections, and running the Paxos consensus algorithm with acceptors. Entrypoint: `src/agents/proposer.rs`
3. Acceptor: listens for connections from Proposers, and engages in the Paxos consensus algorithm with them. Also, handle `Accepted` messages, acting as a learner. Entrypoint: `src/agents/acceptor.rs`
4. Orchestrator: Start a new Paxos network. That is, start multiple processes for a Leader and an arbitrary number of Acceptors. Entrypoint: `src/orchestrator.rs`

2 Differences from Paxos

I assume that the leader never fails. This is a vast simplification of Paxos. In real Paxos, this is not assumed. I could, however, easily change my implementation to allow for multiple proposers to vote on a new leader when one fails. Also, Mutli-Paxos supports multiple leaders. However, changing my implementation to support Multi-Paxos would require significant work.

3 Network Protocol

A message on my implementation of Paxos, at its core, is the following enum:

```
pub enum MessageBody {
    Prepare(u32),
    Promise(u32, Option<Entry<Key, Value>>),
    Accept(u32, Entry<Key, Value>),
    Accepted(u32, Entry<Key, Value>),
    Nack,
}
```

The `u32` is for proposal numbers. The Promise message has an optional value, since acceptors who have never accepted a proposal before will return `None` for the value in their Promise message. If they have accepted a proposal before, this will be the accepted value. The rest of the messages follow the protocol as described in documentation.

This enum is serialized using the `serde` library, and sent over the network along with a small message header.

4 Correctness

My implementation of Paxos is still correct with respect to my assumption of a leader not failing. This is because Proposers and Acceptors engage in the two-phase process of sending Prepare, Propose, Accept, and Accepted messages with each other in accordance to the logic described in the documents studied in class. There are no "shortcuts" taken in my implementation: Paxos consensus is present in my implementation.

5 Usage

To build my project, make sure to compile with `cargo build --release`. Then, a Paxos network can be initialized with

```
./target/release/paxos258kv <base port> <num acceptors>
```

To enable logging, run

```
RUST_LOG=info ./target/release/paxos258kv <base port> <num acceptors>
```

Then the client can be run as described in the project prompt:

```
./target/release/kvclient <leader port> put key some value
./target/release/kvclient <leader port> get key
```

Additionally, each binary in `./target/release` can be run individually, without the use of the orchestrator (`./target/release/paxos258kv`).