Aidan Rohm
Alex Brooke
CISC-5597

# Lab 2: Basic Paxos Distributed File System

## Creating Paxos: Replacing add/sub RPCs with Paxos RPCs

The starter server.py only exposed two basic RPCs (add() and sub()), which didn't perform any distributed coordination. We reused the existing RPCHandler and rpc_server() structure from the original example and registered new Paxos functions:

```
handler = RPCHandler()
handler.register_function(prepare)
handler.register_function(accept)
handler.register_function(SubmitValue)
handler.register_function(get_value)
```

Each new function mapped directly to a phase of the Paxos algorithm:
- prepare(n) — Phase 1, promises not to accept lower-numbered proposals
- accept(n, v) — Phase 2, accepts proposals if they're valid
- SubmitValue(value) — client entry point that coordinates both phases
- get_value() — returns the current local replica's value

This kept the original RPC pattern intact while extending the logic to handle distributed consensus rather than local computation.

## Multi-Node Cluster Connectivity

During initial testing, proposers consistently failed Phase 1 with messages like:

```
SubmitValue FAILED in Phase 1 (only 1 promises).
```

This indicated that the node could only reach itself — the other two nodes were either not running, misconfigured, or using different authentication keys. Without responses from a majority of nodes, Paxos could not progress beyond the prepare phase.

We added explicit cluster configuration variables to the top of server.py:

```
NODE_ID = 1  # Change for each node: 1, 2, or 3
ALL_NODES = [
   ('10.128.0.2', 17000),
   ('10.128.0.3', 17000),
   ('10.128.0.4', 17000),
]
AUTHKEY = b'peekaboo'
MAJORITY = 2
```

Each node runs the same script with a unique NODE_ID but shares the same ALL_NODES list and AUTHKEY.

We also added a small helper to send RPCs between nodes:

```
from multiprocessing.connection import Client
```

```
def call_remote(addr, func, *args):
    try:
        c = Client(addr, authkey=AUTHKEY)
        c.send(pickle.dumps((func, args, {})))
        reply = pickle.loads(c.recv())
        c.close()
        return reply
    except Exception as e:
        print(f"[Node {NODE_ID}] Failed to reach {addr}: {e}")
        return None
```

This allowed each node to reach all peers and collect promises/accepts during SubmitValue. Once all three nodes were properly configured and running the Paxos version of server.py, proposals consistently reached a majority of promises and accepts, completing both phases successfully.

**Demonstrating Competing Proposers and Livelock**

We needed a way to simulate multiple proposers submitting values at nearly the same time to show that only one value ultimately gets chosen.
We created two separate client scripts — clientA and clientB — that each connect to different nodes in the cluster and submit distinct values:

```
# clientA connects to Node 1
proxy.SubmitValue('Hello from clientA')
# clientB connects to Node 3
proxy.SubmitValue('Hello from clientB')
```

To simulate real network timing and allow competition between proposers, a randomized delay was added at the start of SubmitValue:

```
time.sleep(random.uniform(0, 2))
```

This produces realistic timing variations where either A or B might reach the majority first, allowing both 'A wins' and 'B wins' outcomes on repeated runs.

Running both clients simultaneously demonstrated the correct Paxos safety property — only one value is ever chosen, even with multiple competing proposers.