# Engineering Notebook #4

*Kamryn Ohly + Grace Li*

**Link to Github:** https://github.com/kamrynohly/fault-tolerant-messaging-exercise

**Link to Live Notebook (with videos!):**
https://docs.google.com/document/d/1EH6GpqOW2bhChM0n0wXCLkDHBFdqEFugvsLFtZznH28/edit?usp=sharing

**Demo Video:**
https://drive.google.com/file/d/1RnO8f_iX5T2ciUfO4uRxzjkoIsu3JZ3D/view?usp=sharing

**Introduction**
This assignment truly gave us a new appreciation for building a fault tolerant system. Specifically, we went into the assignment blissfully unaware of the many edge cases that would cause the system to become out of sync. From the mind-bending challenge of managing multiple independent database in the same code repository and/or across multiple devices, to figuring out the dozens of architectural design choices that would all be permissible, but some more advantageous than others, we genuinely underestimated the sophistication of the assignment going in and now have an added layer of respect for all fault-tolerate, live communication systems 😅. It was, however, incredibly rewarding once we finally got the system up and running, and it was a near magical experience to be able to message each other from different machines with multiple backup servers that we could kill with no effect on the client-side.

**Architectural Design & Changes**
Something we had to think about explicitly for the technical implementation was **how to decide leader detection on the client-side,** specifically whether we wanted the client to be "aware" of who the leader server was. We had two options for making this decision: (1) the client could either be blissfully unaware of who the "leader" client is and responsible for iterating through all possible servers and testing a write, or (2) the server could be in charge of surfacing to the client who the leader was at any point of time, perhaps through an API or alternative gRPC endpoint. We ultimately decided to go with the former approach as we think it is a simpler implementation for the client to be responsible for finding a valid server than to ensure that there is a constantly valid endpoint between the leader server and the client. Therefore, we decided to loop through available servers, as surfaced by the config file, and then use try-except blocks to catch exceptions.

Another architectural design challenge that we faced was how we wanted to coordinate leader election between the servers. Specifically, we needed to solve two problems: (1) agreement on leader

election, and (2) leader-finding when a replica server joins. To be clear, we required the latter process to be distinct from the former because, when a new (replica) server joins a system where a leader already exists (as in the case of the extra credit), it must be able to recognize who the leader in the system is. We decided to solve the first problem to ensure each replica process had a list of all other servers in the system. Then, when it lost its heartbeat connection with the leader, we would trigger a leader election process that simply assigned the lowest UUID server as the leader, using each server's uuid. Since this is a deterministic assignment, we did not have to coordinate this agreement between servers, we simply allowed each server to recognize on its own who the lowest PID process would be. In the case where the leader system crashes, the first replica server that detects this loss of connection will simply begin another leader election process, and in this matter, each server in the network can be reliably expected to converge on the same leader server, if one exists.

```
def run_election(self):
    """Handle electing a new leader. Utilises the lowest UUID of the existing servers."""
    # If we are the only server remaining, we become the leader.
    if len(self.servers) == 0:
        self.leader["id"] = self.server_id
        self.leader["ip"] = self.ip
        self.leader["port"] = self.port
        channel = grpc.insecure_channel(f'{self.ip}:{self.port}')
        stub = service_pb2_grpc.MessageServerStub(channel)
        self.leader["stub"] = stub
        return

    # Otherwise, elect a new leader by finding the lowest UUID between this server's uuid and the
    # other servers.
    all_servers = list(self.servers.keys()) + [self.server_id]
    next_leader_id = min(all_servers)
    self.leader["id"] = next_leader_id

    if next_leader_id == self.server_id:
        logger.info("Server has become the new leader.")
        self.leader["ip"] = self.ip
        self.leader["port"] = self.port
        channel = grpc.insecure_channel(f'{self.ip}:{self.port}')
        stub = service_pb2_grpc.MessageServerStub(channel)
        self.leader["stub"] = stub
    else:
        logger.info("This process is still a replica. A new leader has been selected.")
        self.leader["ip"] = self.servers[next_leader_id]["ip"]
        self.leader["port"] = self.servers[next_leader_id]["port"]
        self.leader["stub"] = self.servers[next_leader_id]["stub"]
```

The final architectural challenge we had to overcome was figuring out how we wanted to propagate changes to a database. Once again, we had a few options here. We could either (1) allow the client to send a write request to any server, and then propagate that request up to the leader server, which would then propagate that write request down to the relevant replica servers to update their databases, or (2) we could only allow the client to write to the leader database. We decided to do the former, as we were weary of the atomicity of the code that would allow the client to detect who the

leader server was. Instead, we decided to add a "source" type to our database-manipulating proto requests to ensure that, if a server received a request from "client" but was not the leader, it would propagate the response up to the leader server. Else, if the source of the request was a "server", the recipient server would recognize that the request was a propagate request from the leader and would update its database in accordance with the request.

## Technical Implementation

### *Leader Assignment*

Once any replica leader discovers that the leader is dead (tragic), they can be depended on to run a leader election. This election is an independent process that does not depend on coordinating between replica servers. We decided on this approach to ensure we did not cause a deadlock where multiple processes are dependent on each other to resolve the leader assignment.

The election begins by first querying the replica server for all other servers that it's aware of. In the case where it is not aware of any other replicas, it knows that it must be the leader, therefore the system assigns itself the leader.

In the case where the replica server detects many other servers, it will find the minimum PID between all other servers and its own PID. Should the minimum and therefore leader PID be itself, the replica server will set leader port, id, and stub to its own port, id, and stub, respectively. Otherwise, the process will set the leader PID to be that of the respective server list.

### *Detect leader in an existing system*

We also need a way for replica servers to join and detect who the leader is, should there already be one. This is because, should we only use minimum PID assignment, there is a possibility that a new, supposedly replica process will join a system that already has a leader server but also possesses the minimum PID. In this case, we want to ensure that the new server is a replica of the leader, and does not accidentally assign itself leadership.

```
# MARK: Fault Tolerance
def setup(self, ip_connect, port_connect):
    # Connect to another server
    initial_channel = grpc.insecure_channel(f'{ip_connect}:{port_connect}')  # Replace with actual second server address
    initial_stub = service_pb2_grpc.MessageServerStub(initial_channel)
    leader_info_response = initial_stub.NewReplica(service_pb2.NewReplicaRequest(new_replica_id=self.server_id, ip_addr=self.ip,

    # Connecting to the leader
    leader_channel = grpc.insecure_channel(f'{leader_info_response.ip}:{leader_info_response.port}')
    leader_stub = service_pb2_grpc.MessageServerStub(leader_channel)
    self.leader["stub"] = leader_stub
    self.leader["id"] = leader_info_response.id
    self.leader["ip"] = leader_info_response.ip
    self.leader["port"] = leader_info_response.port

    print("Leader is currently", self.leader)
```

Therefore, what we implement is a setup function. Upon the joining of any non-leader server, the replica server will call this setup function to connect to ***any other server***, and it needn't be the leader server. This is because we use the stub of any existing server to call the NewReplica service, which is attended to by the lead server who returns the leader_info_response. This response includes the ip, id, and port of the leader, which the new replica server can use to configure the leader dictionary accordingly (which is to coordinate the stub, id, ip, and port of the leader).

In this manner, a new replica can join by initially communicating with any server in the system, and it can be depended on to detect who the leader server is.

***Client-side: Detect a writable server***

We implemented all of the logic to "test until finding a writable server" on the client side. Namely, given a server-modifying response, the client will first iterate through all possible servers, as defined by the client-side config file. It will then try to make its relevant request, like a login request, with that configured ip and host configuration. In the case where that configuration fails because the endpoint is not a writable endpoint/the connecting server is not the leader and therefore does not have write permissions, the client will catch that error in an exception and then iterate through all possible servers to find a connectable server.

```
# MARK: Check Servers
def check_servers(self):
    """Find a server that can be communicated with and handle changes in servers."""
    try:
        # If we already have a server that we are using to communicate, verify that it is still alive.
        if self.current_stub != None:
            # If there is no response after 2 seconds, assume the server has died.
            # Otherwise, no changes are needed.
            response = self.current_stub.Heartbeat(service_pb2.HeartbeatRequest(requestor_id="Client", server_id=""), timeout=2)
            return
        # Otherwise, look for a new server using the servers in the client_config.py file.
        else:
            self.current_stub = None
            for server in SERVERS:
                try:
                    channel = grpc.insecure_channel(f'{server["ip"]}:{server["port"]}')
                    stub = service_pb2_grpc.MessageServerStub(channel)
                    # Check if it exists, if there is no response after 2 seconds, move on
                    stub.Heartbeat(service_pb2.HeartbeatRequest(requestor_id="Client", server_id=""), timeout=2)
                    # If we got a valid response, use this server.
                    self.current_stub = stub
                    logger.info(f"Found server to use with info: {server["ip"]}:{server["port"]}")
                    break
                except Exception as e:
                    # This means that the given server is unavailable.
                    self.current_stub = None
                    continue
    except Exception as e:
        # If something has gone wrong, start over until a suitable server is discovered.
        self.current_stub = None
        time.sleep(1) # To give time for replicas to come back online and/or to waste unnecessary calls.
        self.check_servers()
```

***Spin-up independent databases***

In order to spin up an independent database for each independent process, we decided to use the combination of the port and the ip to create a new database for each database. This ensures

complete independence of the databases started by different processes. Since all changes catered to by the leader are propagated down to replica servers, we can ensure that if the system shuts down correctly (synchronized shutdown), all databases will be kept up-to-date so restarting the system will display persistent storage.

**Testing**

In order to test our program, we iterated with both client and server testing. We kept client-side testing pretty consistent with the previous setup, but we enhanced server-side testing with both replication and framework tests. Altogether, we added 13 new server-side tests to simulate adding a new replica, confirming that the leader information is updated, testing that the server is populated with the list of other servers (through test_get_servers).

```
def test_get_servers(self):
    # Populate server.servers with dummy entries.
    self.server.servers = {
        "server1": {"ip": "127.0.0.2", "port": "5002", "stub": None, "heartbeat": datetime.now()},
        "server2": {"ip": "127.0.0.3", "port": "5003", "stub": None, "heartbeat": datetime.now()}
    }
    request = SimpleNamespace(requestor_id="server1")
    context = DummyContext()
    responses = list(self.server.GetServers(request, context))
    for resp in responses:
        self.assertNotEqual(resp.id, "server1")
```

We also tested setting up message monitoring on a separate thread (through test_monitor_messages) and testing that election runs are done correctly so that lowest UUID is marked as the new leader. We have included some screenshots of those tests below.

```
def test_run_election(self):
    # Add dummy servers with fixed UUIDs.
    self.server.servers = {
        "a-replica": {"ip": "127.0.0.2", "port": "5002", "stub": None, "heartbeat": datetime.now()},
        "z-replica": {"ip": "127.0.0.3", "port": "5003", "stub": None, "heartbeat": datetime.now()}
    }
    # Force the leader to be a replica that is not the lowest.
    self.server.leader["id"] = "z-replica"
    self.server.run_election()
    expected_leader = min(["a-replica", "z-replica", self.server.server_id])
    self.assertEqual(self.server.leader["id"], expected_leader)
```

```
test_server.py .............                                                    [100%]
============================== 13 passed in 0.06s ==============================
```

```
test_client.py ...........                                                      [100%]
============================== 11 passed in 0.56s ==============================
```

**Conclusion**

Overall, this assignment was a truly intellectually stimulating and rewarding experience. We found it quite challenging and a great learning experience to engage in best-efforts to pre-emptively design the best multi-fault tolerant architecture theoretically, and then notice all of the small pitfalls, edge cases, and unexpected errors along the way of our implementation. Finally getting the system to run was such a thrill as testing a multi-fault tolerant system that actually behaves in the way you intend is nothing short of magical. If you have any questions, we are happy to answer them at Kamryn Ohly (kohly@college.harvard.edu) and Grace Li (graceli@college.harvard.edu).

**AI Disclosure**

- **Helped summarize bulleted notes for our engineering notebook into more coherent and concise sentences.**
- **Assisted with writing tests and debugging UI bugs.**