Seminar 4 – Distributed Algorithms & Failures

This set of exercises requires the groups to implement an algorithm and try to make them fail proof. It is not expected to make the system 100% fail proof, especially with both message and node failures. Node failures (fail-stop[-return]) alone should be handled correctly. The combination with losing messages should be handled as good as possible.

Thereto, each group should consider and demonstrate two scenarios with **simulated** failures

1. Only fail-stop(-return) failures of nodes occur. Every node should have a certain probability to fail over an interval of time (e.g. the probability that at least one node fails after 30secs should be 90%). A 'failed' node will go silent and will not send any messages nor process any received messages for a random time interval (Choose appropriate upper and lower limits). Each failure has a low chance of being permanent (fail-stop instead of fail-stop-return). Failures happen independently, so it is possible that a second node fails while the first one is down.

2. Additionally to node failures, messages might get lost (simulated by executing **any** send operation only with a certain probability e.g. 95%). Make sure to apply this possible failure to ALL messages send.

Seminar 4 – Distributed Algorithms & Failures

The discussion of the group's solution in the lab should include a demo (showing that the system works) and a presentation (explaining how and why it works), with a special focus on costs. How much does the failure handling cost (time and messages) compared to the original algorithm?

All times and probabilities given in the exercise descriptions should be approached with common sense! If different numbers are required to show certain effects, **change them** accordingly!

Nodes should be distributed over at least two different physical machines.

You can either use UDP or RPC calls, just make sure that you wrap your send() operations to simulate failures.

We have 3 different scenarios, so each problem will be approached by two groups. So, we expect two independent solutions that we can discuss and compare.

Group 1 & 4
Token-ring algorithm with failure handling

Implement the dining philosopher's problem (beginning of ch.5 of in the slides, or original paper from Dijkstra https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF p. 21) with at least 5 philosophers on different machines and its solution using a token-ring (Slides Ch. 5 p.32, Book p.325) allowing multiple tokes.

Make sure you implement the forks with a mutex-like blocking ability so that the philosophers executing a take_fork() function will halt until the function returns with fork access.

The duration of the think and gormandize operations of philosophers should be randomly chosen from meaningful time intervals (e.g. [1..10] seconds).

Try to built a resilient synchronization system that is correct and avoids deadlocks and starvation in the two failures scenarios.

Group 2 & 5
Berkeley Time Server with Failures

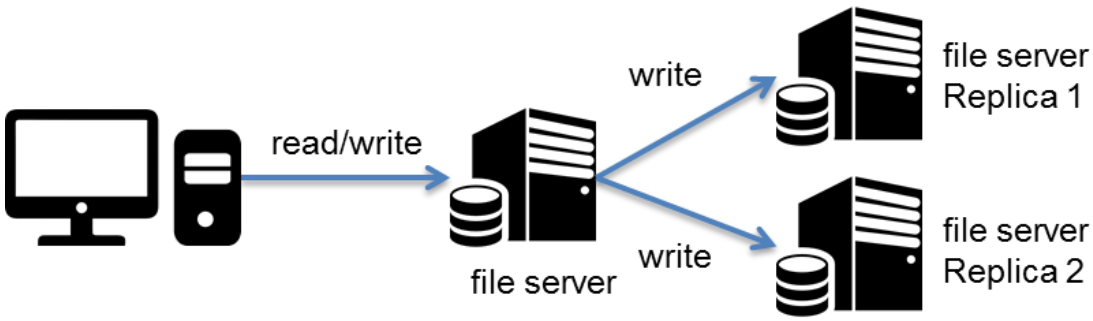
Implement the Berkeley Time Server (Slides Ch. 4 p.19/20, Book p.306) with at least 5 nodes on different machines. Implement a local clock for each node that

- is measured in seconds, but has nothing to do with the real time. You can e.g. initialize each clock with a random number from the interval [0...86400]
- has a variable, high drift in order to introduce the need for synchronization in the scenario. E.g. a local clock might count for each passed second only 0.95 seconds – adjust the values so that you can show something in the demonstration.

Try to built a resilient synchronization system that achieves a reasonable time synchronization in both failures scenarios.

Group 3 & 6
Fileserver with replicates

We want to implement a highly available fileserver that has two additional nodes with replicated data to achieve redundancy. To keep the server and its replicas in the same state, they synchronize via a *write-through* method. Read operations are handled on the original server, write operations are in parallel forwarded to the replicas. If the original fails, one of the replicas takes over its place.



The client will randomly call read and write operations (`read`(), `write`() – the operations are concerning a single string stored on the file server).

1. Focus on the first scenario with node failures for the three servers (failure of the client is not meaningful in this scenario and can be ignored). How can you make the replication transparent to the client? I.e. the client should not realize if either replicate fails, or if the main fail server is replaced by one replica.
2. Once you found a good, working solution to 1) consider losing messages too.