INFO8002 Large-scale data systems
Exercise session II - Possible solution

Instructions

For the main questions, be ready to:

- Describe the architecture of your solution and its operations.
- Define all its components and their interactions.
- Motivate your design decisions.
- Make diagrams whenever necessary.
- State clearly your choices and assumptions

For feedback, feel free to e-mail joeri.hermans@doct.uliege.be

Question 1.

You are responsible for a messaging system. It is your responsibility that a message sent by a user is eventually delivered to all users currently present in the messaging room. Assume a *total* or *atomic* ordering of the messages, i.e., all users should see the same set of messages in the same order. Discuss a *fault-tolerant* centralized or decentralized architecture.

Solution

The most important question when starting to sketch the solution to a question like this, and in fact any system design problem, is to ask yourself the question what distributed system models do you want to design for. Even though most networks are unreliable and asynchronous in practice (https://queue.acm.org/detail.cfm?id=2655736), which is also why the FLP impossibility result is not only academic in nature. Choosing your assumed distributed system model effectively defines the algorithms you can (safely) use. Choosing stronger assumptions on the distributed system model obviously simplifies the design and influences the performance (e.g., in terms of exchanged messages) of your system.

The most frequent appearing distributed system models in this course are:

- Fail-stop (synchronous)
    - Crash-stop failures
    - Perfect links
    - Perfect failure detector
- Fail-noisy (partially synchronous)
    - Crash-stop failures
    - Perfect links

    - Eventually perfect failure detector
- Fail-silent (asynchronous)
    - Crash-stop failures (no failure detector) unless specified otherwise
    - Perfect links

Decentralized architecture

The question imposes an *atomic* restriction on the ordering of the messages of all involved users (processes). Causal ordering can be solved by reliable broadcast. But a total-ordering of the messages in a decentralized setting, in itself, can't. We therefore need *consensus*.

The idea behind consensus-based involves 2 necessary components; (i) reliable broadcast to dissipate the messages to all involved parties (no ordering guarantees), and (ii) consensus to determine the set of the messages and their order, which are subsequently delivered to the user by the application.

Since there are no restrictions on the assumed distributed system model, we can assume a synchronous system. However, we list the algorithms from the course for completeness:

- Asynchronous (fail-silent)
    - Paxos or Raft.
    - Although it should be noted that for performance reasons, it might be a good idea to initially use Paxos to agree upon a leader for a limited time, i.e. using a lease. Without a lease, it would be impossible to re-initiate leader election and progress in the consensus. Because due to the fail-silent setting, we can't tell whether the leader has crashed etc. With the lease, the leadership will automatically expire and leader election with Paxos or Raft can be re-initiated.
- Partially synchronous (fail-noisy)
    - Note the crash-stop failures
    - By combining:
        - An eventual leader detector (to determine the detection of the leader)
        - Epoch-change (to trigger the election of a new leader)
        - Epoch-consensus (an attempt to reach consensus within the epoch)
    - => Leader-Driven consensus (uniform agreement, see below)
- Synchronous (fail-stop)
    - Under the additional assumption that we can impose an **ordered** hierarchy among the processes. This could be done by, for instance, randomly generating universally unique identifiers, which have a low collision-probably (https://en.wikipedia.org/wiki/Universally_unique_identifier).
    - Hierarchical consensus.
    - Hierarchical uniform consensus (uniform agreement is preferred here, no actor and its delivered ordering may diverge, even if it fails).

=> Demonstrate a run of sending a message in your setting (say fail-stop, with current leader being process *n*).

1) A user hits send. The application puts m into the internal buffer and reliably broadcasts its message to all participants.
2) The application keeps the message m in the internal buffer, and so do the other processes which receive m.
3) There are several options now:
   a) The leader has received m but there are other messages in the buffer as well.
   b) The leader has received m and no other messages are in the buffer.
   c) The leader has not received m but other messages are in the buffer.
   d) No messages are in the buffer.
4) Cases a-c are actually the same since there are messages in the buffer. Because process n is the leader, it can arbitrarily select what message is going to be next, e.g., by randomly picking a message from the buffer or by using some other metric that might be available in your system. Say message m' is picked to be decided. The leader first checks if the message m' has already been decided previously (see next). If that's the case it will repeat the procedure until there are no more messages which have not been decided previously. The leader will subsequently decide m' and rb it's decision (including the message m') to all other processes, this includes the processes which not yet have received m'. In case (d) the leader can simply decide(none). The rounds will therefore still increase monotonically and therefore satisfy the integrity property.
5) If the leader crashes, it can be detected due to the availability of the perfect failure detector for all involved processes. In that case, the round simply progresses and process n + 1 becomes leader.

Centralized architecture

Alternatively, the question asked for a *fault-tolerant* centralized approach.

This can be solved using a single server process p which collects the messages to be broadcast using point-to-point link (with perfect links). The ordering of the incoming messages can be decided arbitrarily by the server process. The server process then broadcasts the ordered message using ordered perfect links.

Perfect links guarantee delivery, not ordered guaranteed delivery => You need to ensure they arrive in the same order.

-> Ordered perfect links can be implemented in a rather trivial fashion.
1) Messages are assigned an order number in which they were put into the queue.
2) On the receiving end, deliver messages which have the proper index number, keep messages in a buffer otherwise. Once a message has been delivered (on the receiving end).
3) Check if there are other messages in the buffer consecutive index numbers.

This setup is fine whenever the server or coordinator process does not crash. The question specifically asks a fault-tolerant centralized architecture. This means we need additional coordinator processes, which require a synchronized state. We will assume we can tolerate a single failure of the server processes.

The set of server processes therefore need a synchronized state machine (i.e., ordering of messages). Rely on total order broadcast to synchronize the delivered messages. For instance, server process 1 receives the message. TOB (*if you haven't explained the above, explain TOB and what is necessary)* the next message. The server processes agree on the next message. Since server process 1 has the lowest rank, it will broadcast the message to the users (using the ordered perfect link). Once the users detect a crash of server process 1, they will automatically switch to server 2. As the states between server process 1 and 2 are synced, there is no issue. Messages that were in transit to server process 1, will have to be retransmitted to server process 2. The detection of the crash is only possible within a fail-stop of fail-noisy setting.

Question 2.

Does the FLP impossibility result state that it is impossible to *achieve* consensus? If so, why? If not, what behaviour can happen in practice and why?

Nice read: https://www.the-paper-trail.org/post/2009-02-03-consensus-protocols-paxos/

Solution

No, it is still possible to *achieve* consensus. Imagine an asynchronous environment where everything works nicely, and every actor interacts correctly and in a timely manner. In such a situation it is possible to achieve consensus. However, there is no *guarantee* to make the distinction between a process that crashed, and one that is taking a long time to respond. It is therefore possible that the consensus state machine keeps repeating without ever deciding on a proposed value -> no progression.