Chapter 1

Fault-Tolerance & Paxos

How do you create a fault-tolerant distributed system? In this chapter we start out with simple questions, and, step by step, improve our solutions until we arrive at a system that works even under adverse circumstances, Paxos.

1.1 Client/Server

Definition 1.1 (node)**.** *We call a single actor in the system* ***node****. In a computer network the computers are the nodes, in the classical client-server model both the server and the client are nodes, and so on. If not stated otherwise, the total number of nodes in the system is n.*

Model 1.2 (message passing)**.** *In the* ***message passing model*** *we study distributed systems that consist of a set of nodes. Each node can perform local computations, and can send messages to every other node.*

Remarks:

- We start with two nodes, the smallest number of nodes in a distributed system. We have a *client* node that wants to "manipulate" data (e.g., store, update, . . .) on a remote *server* node.

Algorithm 1.3 Naïve Client-Server Algorithm

```
1: Client sends commands one at a time to server
```

Model 1.4 (message loss)**.** *In the message passing model with* ***message loss,*** *for* ***any*** *specific message, it is not guaranteed that it will arrive safely at the receiver.*

Remarks:

- A related problem is message corruption, i.e., a message is received but the content of the message is corrupted. In practice, in contrast to message loss, message corruption can be handled quite well, e.g. by including additional information in the message, such as a checksum.

- Algorithm 1.3 does not work correctly if there is message loss, so we need a little improvement.

Algorithm 1.5 Client-Server Algorithm with Acknowledgments

```
1: Client sends commands one at a time to server
2: Server acknowledges every command
3: If the client does not receive an acknowledgment within a reasonable time,
   the client resends the command
```

Remarks:

- Sending commands "one at a time" means that when the client sent command c, the client does not send any new command c' until it received an acknowledgment for c.
- Since not only messages sent by the client can be lost, but also acknowledgments, the client might resend a message that was already received and executed on the server. To prevent multiple executions of the same command, one can add a *sequence number* to each message, allowing the receiver to identify duplicates.
- This simple algorithm is the basis of many reliable protocols, e.g. TCP.
- The algorithm can easily be extended to work with multiple servers: The client sends each command to every server, and once the client received an acknowledgment from each server, the command is considered to be executed successfully.
- What about multiple clients?

Model 1.6 (variable message delay)**.** *In practice, messages might experience different transmission times, even if they are being sent between the same two nodes.*

Remarks:

- Throughout this chapter, we assume the variable message delay model.

Theorem 1.7. *If Algorithm 1.5 is used with multiple clients and multiple servers, the servers might see the commands in different order, leading to an inconsistent state.*

Proof. Assume we have two clients u_1 and u_2, and two servers s_1 and s_2. Both clients issue a command to update a variable x on the servers, initially $x = 0$. Client u_1 sends command $x = x + 1$ and client u_2 sends $x = 2 \cdot x$.

Let both clients send their message at the same time. With variable message delay, it can happen that s_1 receives the message from u_1 first, and s_2 receives the message from u_2 first.[1] Hence, s_1 computes $x = (0 + 1) \cdot 2 = 2$ and s_2 computes $x = (0 \cdot 2) + 1 = 1$.

□

[1]For example, u_1 and s_1 are (geographically) located close to each other, and so are u_2 and s_2.

Definition 1.8 (state replication)**.** *A set of nodes achieves **state replication**, if all nodes execute a (potentially infinite) sequence of commands $c_1, c_2, c_3, \ldots$, in the same order.*

Remarks:

- State replication is a fundamental property for distributed systems.
- Since state replication is trivial with a single server, we can designate a single server as a *serializer*. By letting the serializer distribute the commands, we automatically order the requests and achieve state replication!

Algorithm 1.9 State Replication with a Serializer

```
1: Clients send commands one at a time to the serializer
2: Serializer forwards commands one at a time to all other servers
3: Once the serializer received all acknowledgments, it notifies the client about
   the success
```

Remarks:

- This idea is sometimes also referred to as *master-slave replication*.
- What about node failures? Our serializer is a single point of failure!
- Can we have a more *distributed* approach of solving state replication? Instead of directly establishing a consistent order of commands, we can use a different approach: We make sure that there is always at most one client sending a command; i.e., we use *mutual exclusion*, respectively *locking*.

Algorithm 1.10 Two-Phase Protocol

```
   Phase 1
1: Client asks all servers for the lock
   Phase 2
2: if client receives lock from every server then
3:    Client sends command reliably to each server, and gives the lock back
4: else
5:    Clients gives the received locks back
6:    Client waits, and then starts with Phase 1 again
7: end if
```

Remarks:

- This idea appears in many contexts and with different names, usually with slight variations, e.g. *two-phase locking (2PL).*
- Another example is the *two-phase commit (2PC)* protocol, typically presented in a database environment. The first phase is called the *preparation* of a transaction, and in the second phase the transaction is either *committed* or *aborted.* The 2PC process is not started at the client but at a designated server node that is called the *coordinator.*
- It is often claimed that 2PL and 2PC provide better consistency guarantees than a simple serializer if nodes can *recover* after crashing. In particular, alive nodes might be kept consistent with crashed nodes, for transactions that started while the crashed node was still running. This benefit was even improved in a protocol that uses an additional phase (3PC).
- The problem with 2PC or 3PC is that they are not well-defined if exceptions happen.
- Does Algorithm 1.10 really handle node crashes well? No! In fact, it is even worse than the simple serializer approach (Algorithm 1.9): Instead of having a only one node which must be available, Algorithm 1.10 requires *all* servers to be responsive!
- Does Algorithm 1.10 also work if we only get the lock from a subset of servers? Is a majority of servers enough?
- What if two or more clients concurrently try to acquire a majority of locks? Do clients have to abandon their already acquired locks, in order not to run into a deadlock? How? And what if they crash before they can release the locks? Do we need a slightly different concept?

1.2 Paxos

Definition 1.11 (ticket). *A* ***ticket*** *is a weaker form of a lock, with the following properties:*

- **Reissuable:** *A server can issue a ticket, even if previously issued tickets have not yet been returned.*
- **Ticket expiration:** *If a client sends a message to a server using a previously acquired ticket t, the server will only accept t, if t is the most recently issued ticket.*

Remarks:

- There is no problem with crashes: If a client crashes while holding a ticket, the remaining clients are not affected, as servers can simply issue new tickets.
- Tickets can be implemented with a counter: Each time a ticket is requested, the counter is increased. When a client tries to use a ticket, the server can determine if the ticket is expired.
- What can we do with tickets? Can we simply replace the locks in Algorithm 1.10 with tickets? We need to add at least one additional phase, as only the client knows if a majority of the tickets have been valid in Phase 2.

Algorithm 1.12 Naïve Ticket Protocol

```
   Phase 1
1: Client asks all servers for a ticket
   Phase 2
2: if a majority of the servers replied then
3:    Client sends command together with ticket to each server
4:    Server stores command only if ticket is still valid, and replies to client
5: else
6:    Client waits, and then starts with Phase 1 again
7: end if
   Phase 3
8: if client hears a positive answer from a majority of the servers then
9:    Client tells servers to execute the stored command
10: else
11:    Client waits, and then starts with Phase 1 again
12: end if
```

Remarks:

- There are problems with this algorithm: Let u_1 be the first client that successfully stores its command c_1 on a majority of the servers. Assume that u_1 becomes very slow just before it can notify the servers (Line 7), and a client u_2 updates the stored command in some servers to c_2. Afterwards, u_1 tells the servers to execute the command. Now some servers will execute c_1 and others c_2!
- How can this problem be fixed? We know that every client u_2 that updates the stored command after u_1 must have used a newer ticket than u_1. As u_1's ticket was accepted in Phase 2, it follows that u_2 must have acquired its ticket after u_1 already stored its value in the respective server.

- Idea: What if a server, instead of only handing out tickets in Phase 1, also notifies clients about its currently stored command? Then, u_2 learns that u_1 already stored c_1 and instead of trying to store c_2, u_2 could support u_1 by also storing c_1. As both clients try to store and execute the same command, the order in which they proceed is no longer a problem.

- But what if not all servers have the same command stored, and u_2 learns multiple stored commands in Phase 1. Which command should u_2 support?

- Observe that it is always safe to support the most recently stored command. As long as there is no majority, clients can support any command. However, once there is a majority, clients need to support this value.

- So, in order to determine which command was stored most recently, servers can remember the ticket number that was used to store the command, and afterwards tell this number to clients in Phase 1.

- If every server uses its own ticket numbers, the newest ticket does not necessarily have the largest number. This problem can be solved if clients suggest the ticket numbers themselves!

Algorithm 1.13 Paxos

```
Client (Proposer)                              Server (Acceptor)

Initialization ...........................................................

c          ◁ command to execute                $T_{max} = 0$   ◁ largest issued ticket
$t = 0$    ◁ ticket number to try
                                               $C = \perp$     ◁ stored command
                                               $T_{store} = 0$ ◁ ticket used to store C

Phase 1 ..................................................................

1: $t = t + 1$
2: Ask all servers for ticket $t$
                                               3: if $t > T_{max}$ then
                                               4:     $T_{max} = t$
                                               5:     Answer with ok($T_{store}$, $C$)
                                               6: end if

Phase 2 ..................................................................

7: if a majority answers ok then
8:     Pick ($T_{store}$, $C$) with largest $T_{store}$
9:     if $T_{store} > 0$ then
10:        $c = C$
11:    end if
12:    Send propose($t$, $c$) to same
       majority
13: end if
                                               14: if $t = T_{max}$ then
                                               15:     $C = c$
                                               16:     $T_{store} = t$
                                               17:     Answer success
                                               18: end if

Phase 3 ..................................................................

19: if a majority answers success
    then
20:     Send execute($c$) to every server
21: end if
```

Remarks:

- Unlike previously mentioned algorithms, there is no step where a client explicitly decides to start a new attempt and jumps back to Phase 1. Note that this is not necessary, as a client can decide to abort the current attempt and start a new one *at any point* in the algorithm. This has the advantage that we do not need to be careful about selecting "good" values for timeouts, as correctness is independent of the decisions when to start new attempts.
- The performance can be improved by letting the servers send negative

replies in phases 1 and 2 if the ticket expired.

- The contention between different clients can be alleviated by randomizing the waiting times between consecutive attempts.

Lemma 1.14. *We call a message* propose(t,c) *sent by clients on Line 12 a* ***proposal for (t,c)***. *A proposal for (t,c) is* **chosen**, *if it is stored by a majority of servers (Line 15). For every issued* propose(t',c') *with* $t' > t$ *holds that* $c' = c$, *if there was a chosen* propose(t,c).

Proof. Observe that there can be at most one proposal for every ticket number τ since clients only send a proposal if they received a majority of the tickets for τ (Line 7). Hence, every proposal is uniquely identified by its ticket number τ.

Assume that there is at least one propose(t',c') with $t' > t$ and $c' \neq c$; of such proposals, consider the proposal with the smallest ticket number t'. Since both this proposal and also the propose(t,c) have been sent to a majority of the servers, we can denote by S the non-empty intersection of servers that have been involved in both proposals. Recall that since propose(t,c) has been chosen, this means that that at least one server $s \in S$ must have stored command c; thus, when the command was stored, the ticket number t was still valid. Hence, s must have received the request for ticket t' *after* it already stored propose(t,c), as the request for ticket t' invalidates ticket t.

Therefore, the client that sent propose(t',c') must have learned from s that a client already stored propose(t,c). Since a client adapts its proposal to the command that is stored with the highest ticket number so far (Line 8), the client must have proposed c as well. There is only one possibility that would lead to the client not adapting c: If the client received the information from a server that some client stored propose(t^*,c^*), with $c^* \neq c$ and $t^* > t$. But in that case, a client must have sent propose(t^*,c^*) with $t < t^* < t'$, but this contradicts the assumption that t' is the smallest ticket number of a proposal issued after t. □

Theorem 1.15. *If a command c is executed by some servers, all servers (eventually) execute c.*

Proof. From Lemma 1.14 we know that once a proposal for c is chosen, every subsequent proposal is for c. As there is exactly one first propose(t,c) that is chosen, it follows that all successful proposals will be for the command c. Thus, only proposals for a single command c can be chosen, and since clients only tell servers to execute a command, when it is chosen (Line 20), each client will eventually tell every server to execute c. □

Remarks:

- If the client with the first successful proposal does not crash, it will directly tell every server to execute c.
- However, if the client crashes before notifying any of the servers, the servers will execute the command only once the next client is successful. Once a server received a request to execute c, it can inform every client that arrives later that there is already a chosen command, so that the client does not waste time with the proposal process.

- Note that Paxos cannot make progress if half (or more) of the servers crash, as clients cannot achieve a majority anymore.

- The original description of Paxos uses three roles: Proposers, acceptors and learners. Learners have a trivial role: They do nothing, they just learn from other nodes which command was chosen.

- We assigned every node only one role. In some scenarios, it might be useful to allow a node to have multiple roles. For example in a peer-to-peer scenario nodes need to act as both client and server.

- Clients (Proposers) must be trusted to follow the protocol strictly. However, this is in many scenarios not a reasonable assumption. In such scenarios, the role of the proposer can be executed by a set of servers, and clients need to contact proposers, to propose values in their name.

- So far, we only discussed how a set of nodes can reach decision for a single command with the help of Paxos. We call such a single decision an *instance* of Paxos.

- If we want to execute multiple commands, we can extend each instance with an instance number, that is sent around with every message. Once a command is chosen, any client can decide to start a new instance with the next number. If a server did not realize that the previous instance came to a decision, the server can ask other servers about the decisions to catch up.

Chapter Notes

Two-phase protocols have been around for a long time, and it is unclear if there is a single source of this idea. One of the earlier descriptions of this concept can found in the book of Gray [Gra78].

Leslie Lamport introduced Paxos in 1989. But why is it called Paxos? Lamport described the algorithm as the solution to a problem of the parliament of a fictitious Greek society on the island Paxos. He even liked this idea so much, that he gave some lectures in the persona of an Indiana-Jones-style archaeologist! When the paper was submitted, many readers were so distracted by the descriptions of the activities of the legislators, they did not understand the meaning and purpose of the algorithm. The paper was rejected. But Lamport refused to rewrite the paper, and he later wrote that he *"was quite annoyed at how humorless everyone working in the field seemed to be"*. A few years later, when the need for a protocol like Paxos arose again, Lamport simply took the paper out of the drawer and gave it to his colleagues. They liked it. So Lamport decided to submit the paper (in basically unaltered form!) again, 8 years after he wrote it – and it got accepted! But as this paper [Lam98] is admittedly hard to read, he had mercy, and later wrote a simpler description of Paxos [Lam01].

This chapter was written in collaboration with David Stolz.

Bibliography

[Gra78] James N Gray. *Notes on data base operating systems.* Springer, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.