Chapter 15

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.

15.1 Client/Server

Definition 15.1 (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 15.2 (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 15.3 Naive Client-Server Algorithm

1. Client sends commands one at a time to server

Model 15.4 (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 15.5 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:
- Algorithm 15.3 does not work correctly if there is message loss, so we need a little improvement.

Theorem 15.7. If Algorithm 15.5 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 \( x_1 \) are (geographically) located close to each other, and so are \( u_2 \) and \( x_2 \).
**Definition 15.8 (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.
- For people working in the financial tech industry, state replication is often synonymous with the term blockchain. The Bitcoin blockchain we will discuss in Chapter 22 is indeed one way to implement state replication. However, as we will see in all the other chapters, there are many alternative concepts that are worth knowing, with different properties.
- 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 15.9 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 15.10 Two-Phase Protocol**

1. Client asks all servers for the lock
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
3. end if

**Remarks:**
- Does Algorithm 15.10 really handle node crashes well? No! In fact, it is even worse than the simple serializer approach (Algorithm 15.9): Instead of needing one available node, Algorithm 15.10 requires all servers to be responsive!
- Does Algorithm 15.10 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?
- Bad news: It seems we need a slightly more complicated concept.
- Good news: We postpone the complexity of achieving state replication and first show how to execute a single command only.

### 15.2 Paxos

**Definition 15.11 (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.
15.2. PAXOS

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 15.10 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 15.12 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:

• Idea: What if a server, instead of only handing out tickets in Phase 1, also notifies clients about its currently stored command? Then, u2 learns that u1 already stored c1 and instead of trying to store c2, u2 could support u1 by also storing c1. 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 u2 learns multiple stored commands in Phase 1. What command should u2 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 15.13 Paxos

Client (Proposer)        Server (Acceptor)
Initialization
\[ T_{\text{max}} = 0 \, \text{a largest issued ticket} \]
\[ c \in \text{command to execute} \]
\[ t = 0 \, \text{a ticket number to try} \]
\[ C = \perp \, \text{a stored command} \]
\[ T_{\text{store}} = 0 \, \text{a ticket used to store } C \]

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

Phase 2
7: If a majority answers \( \text{ok} \) then
8: Pick \((T_{\text{store}}, C)\) with largest \( T_{\text{store}} \)
9: If \( T_{\text{store}} > 0 \) then
10: \( c = C \)
11: end if
12: Send \( \text{propose}(t, c) \) to same majority
13: end if
14: If \( t = T_{\text{max}} \)
15: \( C = c \)
16: \( T_{\text{store}} = t \)
17: Answer success
18: end if

Phase 3
19: If a majority answers \( \text{success} \) then
20: Send \( \text{execute}(c) \) 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 15.14. We call a message \( \text{propose}(t,c) \) 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 \( \text{propose}(t',c') \) with \( t' > t \) holds that \( c' = c \), if there was a chosen \( \text{propose}(t,c) \).

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

Assume that there is at least one \( \text{propose}(t',c') \) with \( t' > t \) and \( c' \neq c \), of such proposals, consider the proposal with the smallest ticket number \( c' \). Since both this proposal and also the \( \text{propose}(t,c) \) 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 \( \text{propose}(t,c) \) has been chosen, this means 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 \( t' \) before \( t \). \( \square \)

Theorem 15.15. If a command \( c \) is executed by some servers, all servers (eventually) execute \( c \).

Proof. From Lemma 15.14 we know that once a proposal for \( c \) is chosen, every subsequent proposal is for \( c \). As there is exactly one first \( \text{propose}(t,c) \) 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 \( c \). \( \square \)

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.

• For state replication as in Definition 15.8, we need to be able to execute multiple commands, we can extend each instance with an instance number, that is sent around with every message. Once the 1st command is chosen, any client can decide to start a new instance and compete for the 2nd command. If a server did not realize that the 1st instance already 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].

Leslie Lamport is an eminent scholar when it comes to understanding distributed systems, and we will learn some of his contributions in almost every chapter. Not surprisingly, Lamport has won the 2013 Turing Award for his fundamental contributions to the ‘theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency’ [Lam]. One can add arbitrarily to this official citation, for instance Lamport’s popular LaTeX typesetting system [Lam94], based on Donald Knuth’s TeX.

This chapter was written in collaboration with David Stolz.

Bibliography


