Computer Systems - Notes Week 8

Ruben Schenk, ruben.schenk@inf.ethz.ch

January 6, 2022

Chapter 14: Introduction To Distributed Systems

14.1 Why Distributed Systems?

Today's computing and information systems are inherently *distributed.* Many companies are operating on a global scale, with thousands or even millions of machines on all the continents. Data is stored in various data centers, computing tasks are performed on multiple machines. In summary, today almost all computer systems are distributed, for different reasons:

- Geography: Large organizations and companies are inherently geographically distributed, and a computer system needs to deal with this issue anyway.
- Parallelism: To speed up computation, we employ multicore processors or computing clusters.
- Reliability: Data is replicated on different machines to prevent data loss.
- Availability: Data is replicated on different machines to allow for access at any time, without bottlenecks, minimizing latency.

14.2 Distributed Systems Overview

We introduce some basic techniques to building distributed systems, with a focus on fault-tolerance. We will study different protocols and algorithms that allow for fault-tolerant operation, and we will discuss practical systems that implement these techniques. Furthermore, we will see different models that can be studies. The focus is on protocols and systems that matter in practice.

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

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: 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.

```
# Algorithm 15.3: Naive Client-Server Algorithm
1:  Client sends commands one at a time to the server
```

Model 15.4: In the message passing model with **message loss,** for any specific message, it is not guaranteed that it will arrive safely at the receiver. Algorithm 15.3 does not work correctly if there is message loss, so we need a little improvement.

```
# Algorithm 15.5: Client-Server Algorithm with Acknowledgments
1:  Client sends commands one at a time to the server
```

```
2:  Server acknowledges every command
3:  If the client does not receive an acknowledgment within a reasonable time, the client resends the comma
```

Remark: 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.

Model 15.6: In practice, messages might experience different transmission times, even if they are being sent between the same two nodes.

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

> ***Theorem 15.7:*** If Algorithm 15.5 us used with multiple clients and multiple servers, the servers might see the commands in different order, leading to an inconsistent state.

A set of nodes achieves **state replication** if all nodes execute a potentially infinite sequence of commands c_1, c_2, c_3, ... in the same order. 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 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
```

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
    # 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 other 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:
```

15.2 Paxos

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 a new ticket.
- 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.

```
# Algorithm 15.12: Naive 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 9), 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 other c_2!
- 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.

```
# Algorithm 15.13: Paxos
Client (Proposer)                          Server (Acceptor)
    # Initialization
    c <- command to execute                    T_max = 0 <- largest issued ticket
    t = 0 <- ticket number to try
                                               C = bot <- 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 answer with 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
```

Unlike previously mentioned algorithms, there is no step where a client explicitly decides to start a new attempt and jumps back to Phase 1. 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.

Lemma 15.14: We call a message `propose(r, 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 $t' > t$, it holds that $c' = c$, if there was a chosen `propose(t, c)`.

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

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

Chapter 16: Consensus

16.1 Two Friends

Alice wants to arrange dinner with Bob. She sends a text message suggesting meeting for dinner at 6pm. However, Alice cannot be sure that the message arrives at Bob's phone, hence she will only go to the meeting point if she receives a confirmation message from Bob. But Bob cannot be sure that his confirmation message is received. Therefore, Bob demands a confirmation message from Alice, to be sure that she will be there. But as this message can also be lost...

Such a protocol cannot terminate. Can Alice and Bob use Paxos?

16.2 Consensus

There are n nodes, of which at most f might crash, i.e. at least $n - f$ nodes are *correct.* Node i starts with an input value v_i. The nodes must decide for one of those values, staisfying the following properties:

- *Agreement:* All correct nodes decide for the same value.
- *Termination:* All correct nodes terminate in finite time.
- *Validity:* The decision value must be the input value of a node.

Remarks:

- We assume that every node can send messages to every other node, and that we have reliable links.
- There is no broadcast medium. If a node wants to send a message to multiple nodes, it needs to send multiple individual messages.
- Does Paxos satisfy all three criteria? No, one will notice that Paxos does no guarantee termination.
- In fact, the consensus problem state above cannot be solved by any algorithm.

16.3 Impossibility Of Consensus

Model 16.3: In the **asynchronous model,** algorithms are event based ("upon receiving message ..., do ..."). Nodes do not have access to a synchronized wall-clock. A message sent from one node to another will arrive in a finite but unbound time.

For algorithms in the asynchronous model, the **runtime** is the number of time units from start of the execution to its completion in the worst case (every legal input, every execution scenario), assuming that each message has a delay of *at most* one time unit.

We say that a system is fully define at any point during the execution by its **configuration** C. The configuration includes the state of every node, and all messages that are in transit (sent but not yet received). We call a configuration C **univalent,** if the decision value is determined independently of what happens afterwards. We call a configuration that is univalent for value v **v-valent**. A configuration C is called **bivalent** if the nodes might decide for 0 or 1.

We call the initial configuration of an algorithm C_0. When nodes are in C_0, all of them executed their initialization code and possibly, based on their input values, sent some messages. These initial messages are also included in C_0. In other words, in C_0 the nodes are now waiting for the first message to arrive.

Lemma 16.7: There is at least one selection of input values V such that the according initial configuration C_0 is bivalent, if $f \geq 1$.

A **transition** from configuration C to a following configuration C_τ is characterized by an event $\tau = (u, m)$, i.e. node u receiving message m. A transition $\tau = (u, m)$ is only applicable to C, if m was still in transit in C. C_τ differs from C as follows: m is no longer in transit, u has possibly a different state (as u can update its state based on m), and there are potentially new messages in transit, sent by u.

The **configuration tree** is a direct tree of configurations. Its root is the configuration C_0 which is fully characterized by the input values V. The edges of the tree are the transitions. Every configuration has all applicable transitions as outgoing edges.

Remarks:

- For any algorithm, there is exactly *one* configuration tree for every selection of input values.
- Leaves are configurations where the execution of the algorithm terminated.
- Every path from the root to a leaf is one possible asynchronous execution of the algorithm.
- Leaves must be univalent, or the algorithm terminates without agreement.

> ***Lemma 16.10:*** Assume two transitions $\tau_1 = (u_1, m_1)$ and $\tau_2 = (u_2, m_2)$ for $u_1 \neq u_2$ are both applicable to C. Let $C_{\tau_1\tau_2}$ be the configuration that follows C by first applying transition τ_1 and then τ_2, and let $C_{\tau_2\tau_2}$ be defined analogously. It holds that $C_{\tau_1\tau_2} = C_{\tau_2\tau_2}$.

We say that a configuration C is **critical,** if C is bivalent, but all configurations that are direct children of C in the configuration tree are univalent. Informally, C is critical, if it is the least moment in the execution where the decision is not yet clear. As soon as the next message is processed by any node, the decision will be determined.

> ***Lemma 16.12:*** If a system is in a bivalent configuration, it must reach a critical configuration within finite time, or it does not always solve consensus.
>
> ***Lemma 16.13:*** If a configuration tree contains a critical configuration, crashing a single node can create a bivalent leaf, i.e. a crash prevents the algorithm from reaching agreement.
>
> ***Theorem 16.14:*** There is no deterministic algorithm which always achieves consensus in the asynchronous model, with $f > 0$.

Remarks:

- If $f = 0$, then each node can simply send its value to all others, wait for all values, and choose the minimum.
- But if a single node may crash, there is no deterministic solution to consensus in the asynchronous model.

16.4 Randomized Consensus

```
# Algorithm 16.15: Randomized Consensus (assuming f < n/2)
1:  v_i in {0, 1} <- input bit
2:  round = 1
3:  while true do:
4:      Broadcast myValue(v_i, round)
    # Propose
5:      Wait until a majority of myValue messages of current round arrived
6:      if all messages contain the same value v then:
7:          Broadcast propose(v, round)
8:      else:
9:          Broadcastpropose(bot, round)
10:     end if
    # Vote
11:     Wait until a majority of propose messages of current round arrived
12:     if all messages propose the same value v then:
13:         Broadcast myValue(v, round + 1)
14:         Broadcast propose(v, round + 1)
15:         Decide for v and terminate
16:     else if there is at least one proposal for v then
17:         v_i = v
18:     else
19:         Choose v_i randomly, with Pr[v_i = 0] = Pr[v_i = 1] = 1/2
```

```
20:     end if
21:     round = round + 1
22: end while
```

The idea of Algorithm 16.15 is very simple: Either all nodes start with the same input bit, which makes consensus easy. Otherwise, nodes toss a coin until many nodes get - by chance - the same outcome.

Lemma 16.16: AS long as no node decides and terminates, Algorithm 16.15 does not get stuck, independent of which nodes crash.

Lemma 16.17: Algorithm 16.15 satisfies the validity requirement. ***Lemma 16.18:*** Algorithm 16.15 satisfies the agreement requirement. ***Lemma 16.19:*** Algorithm 16.15 satisfies the termination requirement, i.e. all nodes terminate in expected time $O(2^n)$.

Theorem 16.20: Algorithm 16.15 achieves binary consensus with expected runtime $O(2^n)$ if up to $f < n/2$ nodes crash.

Theorem 16.21: There is no consensus algorithm for the asynchronous model that tolerates $f \geq n/2$ many failures.

Algorithm 16.15 solves the consensus problem with optimal fault-tolerance - but it is awfully slow. Nevertheless, the algorithm can be improved by tossing a so-called *shared coin.* A shared coin is a random variable that is 0 for all nodes with constant probability, and 1 with constant probability. To improve the expected runtime of Algorithm 16.15, we replace line 19 with a function call to the shared coin algorithm.

16.5 Shared Coin

```
# Algorithm 16.22: Shared Coin (code for node u)
1:  Choose local coin c_u = 0 with probability 1/n, else c_u = 1
2:  Broadcast myCoin(c_u)
3:  Wait for n-f coinds and store them in the local coin set C_u
4:  Broadcast mySet(C_u)
5:  Wait for n-f coin sets
6:  if at least once coin is 0 among all coins in the coin sets then:
7:      return 0
8:  else:
9:      return 1
10: end if
```

Since at most f nodes crash, all nodes will always receive $n - f$ coins respectively coin sets in lines 3 and 5. Therefore, all nodes make progress and termination is guaranteed.

Lemma 16.23: Let u be a node, and let W be the set of coins that u received in at least $f + 1$ different coin sets. It holds that $|W| \geq f + 1$.

Lemma 16.24: All coins in W are seen by all correct nodes.

Theorem 16.25: If $f < n/3$ nodes crash, Algorithm 16.22 implements a shared coin.

Theorem 16.26: Plugging Algorithm 16.22 into Algorithm 16.15 we get a **randomized consensus algorithm** which terminates in a constant expected number of rounds tolerating up to $f < n/3$ crash failures.