Dist - Systems - Notes
08 November 2020 17:25

As the notes for this course are particularly well done, and due to the little time I have recently, this will look mainly as a collage:

Chapter 1 - Fault-tolerance and Paxos

The problem set-up would look as follows in the chapter

- Problem setup:
  - N distributed, trusted nodes
  - All-to-all asynchronous messages
  - Variable message transmission time
  - Messages may be lost
  - Some nodes may crash
- One fundamental goal: *state replication*
  - Same sequence of commands in the same order

} We are interested in solving this.

Notice that with such a setting a simple ack-mechanism as follows is not sufficient

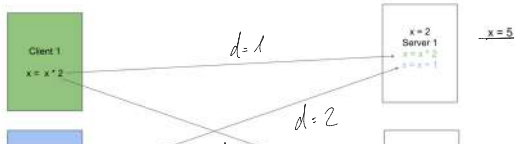
Algorithm 15.5 Client-Server Algorithm with Acknowledgments
1: Client sends commands one at a time to server
2: Server acknowledges every command
3: If the client does not receive an acknowledgment within a reasonable time, the client resends the command

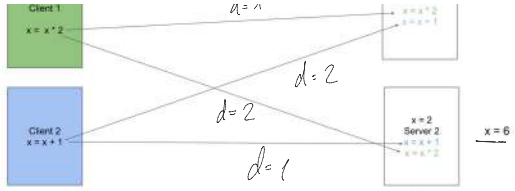
I.e.: -

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.*

!

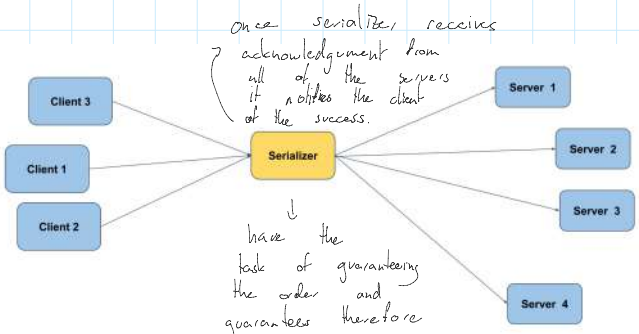
⇒ To see that consider, for instance the following:





so you see that the asynchronous messages with different possible delays leads to major issues.

A solution would be to elect a serializer / master



Issue: Single-Point-of-Failure

One other possible solution is the one of working with locks.

↳ Idea of "il signore delle mosche"
↳ chi ha la conchiglia può parlare.

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 server, and gives the lock back
4: **else** (is with ACK.)
5: Clients gives the received locks back
6: Client waits, and then starts with Phase 1 again
7: **end if**

Issues: 1. what happens if a client that acquired locks dies?

↳ locks acquired forever and the

↳ locks acquired forever and the algo fails.

(2) 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!

↳ As otherwise no chance of obtaining the lock.

The issues above leads to the idea of Paxos. This is a fault-tolerant algorithm.

↳ It explains how to run a single command in a fault tolerant way.

to solve the issues above Paxos used the following new concept of ticket:

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.*

Notice that therefore playing with tickets can possibly lead to a solution that avoids the crash

We look first at the following simple algorithm that is propedeutic to paxos in its ideas:

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**

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**

Notice that here we introduced a 3rd phase - i.e. one phase more in contrast to the two phase protocol with locks previously seen.

↳ This is necessary as in contrast to the previous algo we introduced a fault-tolerant system that just relies on a majority.

↳ We need a 2nd phase to check that a majority of servers answers before sending it command to it.

(Notice therefore that each client must know the # servers in the system).

↳ In a third phase the client checks then that a majority of servers stored the command, and tells them to execute it.

⚠ Notice: This algorithm is just propedeutic to Paxos and does not guarantee state replication.

↳ To see that consider the following:

Idea: as ticket is more flexible you cannot guarantee between

↳ receival of ticket from server and server store of command

} } and { {

sending of command that no other client

sending of command that no other client already requested and committed another command to one of these servers due to a faster bandwidth and message propagation.

↳ Notice therefore problem in phase 3.

If v_1 := node that stored c_1 first

v_2 := node that executed c_2 as first command.

↳ Then it is obvious in the above setting that

(if it is the last issued)

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.

↳ So the idea of Paxos is basically the following:

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.

- In Paxos just the first client updates the command
  - ↳ all the subsequent clients that arrive endorse this command if it was not executed yet

Why did we require a majority in the above algo?

Why did we require a majority in the above algo?

But what if not all servers have the same command stored, and u_2 learns multiple stored commands in Phase 1. What 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.

↳ as due to majority you know that the value supported by the majority will stay constant through time.

↳ i.e. u_2 cannot get any other command as it waits for majority

ticket + stored command

As a final remark notice that in the basic naive ticket algorithm above you might have a different ticket system for each server

↳ however this cannot be in Paxos as otherwise it is impossible to guarantee highest number = most recent number

↳ Solution: Client has to propose ticket.

This has basically set a strong fundation for understanding Paxos

Algorithm 15.13 Paxos

Client (Proposer)	Server (Acceptor)
Initialization	
c ◁ *command to execute* $t = 0$ ◁ *ticket number to try*	$T_{\max} = 0$ ◁ *largest issued ticket* $C = \perp$ ◁ *stored command* $T_{\text{store}} = 0$ ◁ *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}	

Recall client selects ticket

Server notifies about last stored command

guarantees consistency

take last saved command

take last saved command

guarantees consistency

```
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                    } updates the stored command
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
```

Notice once this step is done by the majority of the servers we have reached the point of no return and the command will eventually be computed.

Chapter 2 - Consensus

In the previous chapter we have seen some protocols for reaching an agreement that could satisfy state-replication.

This chapter extends this idea, by setting more stringent constraints. Will call such stricter agreement Consensus

Definition 16.1 (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, satisfying the following properties:*

More stringent requirements ↳ have to satisfy them if you want to reach consensus

- **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.*

} will have to prove these for algo claiming to reach consensus.

In this chapter we furthermore make the following assumptions:

- We assume that every node can send messages to every other node, and that we have reliable links, i.e., a message that is sent will be received.
- There is no broadcast medium. If a node wants to send a message to multiple nodes, it needs to send multiple individual messages. If a node crashes while broadcasting, not all nodes may receive the broadcasted message. Later we will call this best-effort broadcast.

- We assume that every node can send messages to every other node, and that we have reliable links, i.e., a message that is sent will be received.
- There is no broadcast medium. If a node wants to send a message to multiple nodes, it needs to send multiple individual messages. If a node crashes while broadcasting, not all nodes may receive the broadcasted message. Later we will call this best-effort broadcast.

Notice moreover:

- Does Paxos satisfy all three criteria? If you study Paxos carefully, you will notice that Paxos does not guarantee termination. For example, the system can be stuck forever if two clients continuously request tickets, and neither of them ever manages to acquire a majority.
- One may hope to fix Paxos somehow, to guarantee termination. However, this is impossible. In fact, the consensus problem of Definition 16.1 cannot be solved by any algorithm.

Notice that what we will actually show next is that no deterministic protocol can reach consensus.

In order to verify the above we must first give some temporal structure to the system - such that we can define the termination requirement correspondingly.

Model 16.2 (asynchronous). *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 unbounded time.*

Definition 16.3 (asynchronous runtime). *For algorithms in the asynchronous model, the* ***runtime*** *is the number of time units from the 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.*

For the proof of the impossibility of consensus in deterministic protocols we will introduce some notation:

Definition 16.4 (configuration). *We say that a system is fully defined (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).*

Definition 16.5 (univalent). *We call a configuration C* ***univalent*** *if the decision value is determined independently of what happens afterwards.*

Definition 16.6 (bivalent). *A configuration C is called* ***bivalent*** *if the nodes might decide for 0 or 1.*

Notice that we will make the problem harder, meaning that we will focus on

Notice that we will make the problem harder, meaning that we will focus on binary input $\{0, 1\}$, and show that the impossibility theorem holds for it.

↳ As any input that allows more than two values can be reduced into a binary system you can understand now that the impossibility theorem also holds for the generalized system.

We will go on with this setting and show that a bivalent initial configuration always exists

C_0 := initial configuration, i.e. nodes executed their initialization code and possibly sent some messages based on it

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

↳ Where, notice f = # of nodes that can fails, without failing the model

Proof. As explained in the previous remark, C_0 only depends on the input values of the nodes. Let $V = [v_0, v_1, \ldots, v_{n-1}]$ denote the array of input values, where v_i is the input value of node i.

We construct $n + 1$ arrays $V_0, V_1, \ldots, V_n$, where the index i in V_i denotes the position in the array up to which all input values are 1. So, $V_0 = [0, 0, 0, \ldots, 0]$, $V_1 = [1, 0, 0, \ldots, 0]$, and so on, up to $V_n = [1, 1, 1, \ldots, 1]$.

Note that the configuration corresponding to V_0 must be 0-valent so that the validity requirement is satisfied. Analogously, the configuration corresponding to V_n must be 1-valent. Assume that all initial configurations with starting values V_i are univalent. Therefore, there must be at least one index b, such

we will prove this assumption wrong.

that the configuration corresponding to V_{b-1} is 0-valent, and configuration corresponding to V_b is 1-valent. Observe that only the input value of the b^{th} node differs from V_{b-1} to V_b.

Since we assumed that the algorithm can tolerate at least one failure, i.e., $f \geq 1$, we look at the following execution: All nodes except b start with their initial value according to V_{b-1} respectively V_b. Node b is "extremely slow"; i.e., all messages sent by b are scheduled in such a way, that all other nodes must assume that b crashed, in order to satisfy the termination requirement. Since the nodes cannot determine the value of b, and we assumed that all initial configurations are univalent, they will decide for a value v independent of the initial value of b. Since V_{b-1} is 0-valent, v must be 0. However we know that V_b is 1-valent, thus v must be 1. Since v cannot be both 0 and 1, we have a contradiction.

11

↳ Such that it always exists a bivalent initial configuration.

We introduce the concept of transition and configuration tree next.

We introduce the concept of transition and configuration tree next.

Definition 16.8 (transition). *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.*

↳ Notice this is nothing else than the formal definition of an event in the asynchronous model.

↳ Notice that a transition can trigger a local computation on node v and a new message sent by the node leading to a new configuration C_τ

Definition 16.9 (configuration tree). *The **configuration tree** is a directed 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.*

Notice that the following holds for configuration trees:

- 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. Note that we use termination in the sense that the system as a whole terminated, i.e., there will not be any transition anymore.
- 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.
- If a node u crashes when the system is in C, all transitions $(u, *)$ are removed from C in the configuration tree.

i.e. all the following configurations triggered by all of the messages related to the group

all config ⇒ so any message/transaction would trigger the system into a critical

Definition 16.11 (critical configuration). *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.*

Given these new definitions it is possible to prove the following three Lemmas, which will lead consequently to the impossibility of consensus via deterministic algos.

will " lead consequently to the impossibility of consensus via deterministic algos.

1st Lemma

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 $C_{\tau_2\tau_1}$ be defined analogously. It holds that $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$.*

Proof: I.e. you must prove that τ_1, τ_2 are disjoined. Then the above will follow immediately.

↳ The fact of independence follows then immediately by noting that τ_2 can be applied to C_{τ_1} as it cannot change the state of u_1 but rather just the one of u_2 which is independent of C_{τ_1} as soon as m_2 arrives to u_2 sooner than any generated message from C_{τ_1}.

2nd Lemma:

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.*

The proof is trivial and therefore skipped.

↳ The only way to reach consensus is to reach a critical config in a finite amount of time.

3rd Lemma ! Key Lemma

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.*

Proof:

What we are going actually prove is that in the critical configuration all of

What we are going actually prove is that in the critical configuration all of the transitions must operate on a single node.

↳ It follows then immediately that by crashing this single node you would actually have distroyed the critical configuration

The proof goes as follows:

First note that the following must hold as the critical config C is bivalent:

Let C denote critical configuration in a configuration tree, and let T be the set of transitions applicable to C. Let $\tau_0 = (u_0, m_0) \in T$ and $\tau_1 = (u_1, m_1) \in T$ be two transitions, and let C_{τ_0} be 0-valent and C_{τ_1} be 1-valent.

A now follows immediately with the previous lemma that $u_0 = u_1$, i.e. they can just occur on the same node

Assume that $u_0 \neq u_1$. Using Lemma 16.10 we know that C has a following configuration $C_{\tau_0\tau_1} = C_{\tau_1\tau_0}$. Since this configuration follows C_{τ_0} it must be 0-valent. However, this configuration also follows C_{τ_1} and must hence be 1-valent. This is a contradiction and therefore $u_0 = u_1$ must hold.

You can now generalize this concept to any transition:

Applying the same argument again, it follows that all transitions in T that lead to a 0-valent configuration must take place on u as well, and since C is critical, there is no transition in T that leads to a bivalent configuration. Therefore *all* transitions applicable to C take place on the *same* node u!

This concludes the proof a.s.

If this node u crashes while the system is in C, *all transitions are removed*, and therefore the system is stuck in C, i.e., it terminates in C. But as C is critical, and therefore bivalent, the algorithm fails to reach an agreement. □

It is now clear the impossibility of

It is now clear the impossibility of consensus algorithm:

Theorem 16.14. *There is no deterministic algorithm which always achieves consensus in the asynchronous model, with $f > 0$.*

Proof. We assume that the input values are binary, as this is the easiest non-trivial possibility. From Lemma 16.7 we know that there must be at least one bivalent initial configuration C. Using Lemma 16.12 we know that if an algorithm solves consensus, all executions starting from the bivalent configuration C must reach a critical configuration. But if the algorithm reaches a critical configuration, a single crash can prevent agreement (Lemma 16.13). □

notice the always ⇐ if the starting config would always be univalent then we would be a-priori fine.

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.

↳ but as there is always at least 1 bivalent initial config we cannot have an algo that satisfies consensus always ⇔ ∀ initial configs.

we will however show next that by rendering the algorithm probabilistic we can in fact reach consensus. (notice the nice benchmark with cryptography and the work of Micali).

Randomized Consensus

Here the idea is basically the following.

Instead of relying on critical configuration where the messages processed by a critical node will lead to different univalent configurations reach a more fault-tolerant system that is able to reach consensus by introducing a probabilistic component through which at each round there is a positive probability of entering an univalent configuration.

This can be done easily in the following way:

This can be done easily in the following way:

Algorithm 16.15 Randomized Consensus (assuming $f < n/2$)

```
1: v_i ∈ {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:       Broadcast propose(⊥, 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
```

Notice that due to the majority requirement at each turn there is just 1 possible value which is proposed by different Nodes at each round. Upon this follows agreement!

here you endorse the majority, if the majority endorse the majority

vote.

then you settle for this value and sponsor it

(if at least 1 majority endorse)

here the probabilistic term.

↳ idea: all nodes that do not get a majority proposal will toss a coin, this happens until by chance a large number of nodes toss the same coin.

We will show next that the above algorithm manage to reach consensus, i.e. that it satisfies:

1. Validity
2. Termination
3. Agreement

(1.) Validity:

↳ This is straightforward to see, as it all start with the same value all propose the same value and decide for it.

If there is at least 1 node that starts with a different value, the two values are acceptable, and this also always settle down for one of the two.

(2.) Agreement (with some elements of Termination).

⇒ First of all notice that as long as $f < \frac{n}{2}$ then the algorithm never gets stuck

long as $f < \frac{n}{2}$ then the algorithm never gets stuck and continues, assuming that no node already decided and terminated

⇒ Then what happens if one node [v] decides and terminates?

For any other node $v' \neq v$, it is easy to see:

v' got propose v from a majority ⇒ settle on v.

v' does not get a majority.

↳ however, it must have received at least 1 proposal for v, this because of the majority of messages it collected.

↳ Noticing that v received by v' must be the same received by v and because of lines 16-17 it follows immediately that in the next round all nodes will start with the same value and terminates.

(3) Termination:

We showed that as soon as one node decides the algorithm above of Randomized Consensus terminates in the next round.

↳ We therefore have to prove that we reach the state where 1 node decides in a finite amount of time.

node decides in a finite amount of time.

↳ It is now straightforward to see that the probability that all of the nodes choose the same value is $\frac{1}{2^n}$. (Notice this is the worst case that guarantees that 1 node decides for a state).

↳ It follows then that asymptotically the runtime of the algorithm is $O(2^n)$

This proves that Randomized Consensus with $f < \frac{n}{2}$ number of possible crashing nodes manages to reach consensus.

Next we will see the impossibility of reaching consensus if $f \geq \frac{n}{2}$.

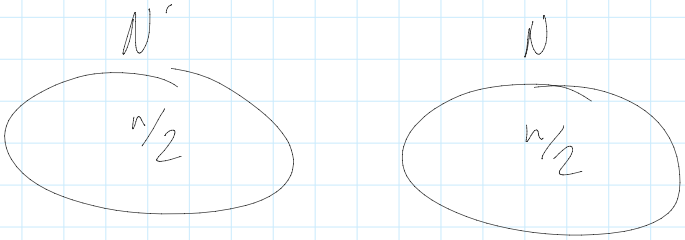
We will then start to question on how to improve the poor runtime of the Randomized Consensus mechanism.

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

Proof:

The proof is quite simple assume to have $f = n/2$.

Then define two disjoint node sets:



$\{ v_i = 0 \quad \forall v_i \in N$

Th

Then assume in V_{Half} $\begin{cases} v_i = 0 & \forall v_i \in N \\ v_j = 1 & \forall v_j \in N' \end{cases}$

It follows immediately that if messages across the two nodes sets are heavily delayed and you would just read $n/2$ messages you would have

↯ no agreement $\Rightarrow$ All nodes in N' $\rightarrow$ choose 1 and all nodes in N $\rightarrow$ choose 0.

Hence in order to reach consistency in the above you would have to wait for at least $n/2 + 1$ messages.

↳ Hence you can immediately see that if you let $t = n/2$ an entire set of nodes that could be represented by the above system can fail and there would be no termination as the system must need $\frac{n}{2}+1$ messages to be sure to always reach agreement. You would therefore not satisfy the termination requirement.

Randomized Consensus

- Algorithm 16.15 solves consensus with optimal fault-tolerance – but it is awfully slow. The problem is rooted in the individual coin tossing: If all nodes toss the same coin, they could terminate in a constant number of rounds.
- Can this problem be fixed by simply always choosing 1 at Line 19?!
- This cannot work: Such a change makes the algorithm deterministic, and therefore it cannot achieve consensus (Theorem 16.14). Simulating what happens by always choosing 1, one can see that it might happen that there is a majority for 0, but a minority with value 1 prevents the nodes from reaching agreement.
- 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. Of course, such a coin is not a magic device, but it is simply an algorithm. To improve the expected runtime of Algorithm 16.15, we replace Line 19 with a function call to the shared coin algorithm.

By

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 $n - f$ coins 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 one coin is 0 among all coins in the coin sets **then**
7: return 0

4: Broadcast mySet(C_u)
5: Wait for $n - f$ coin sets
6: **if** at least one coin is 0 among all coins in the coin sets **then**
7: return 0
8: **else**
9: return 1
10: **end if**

We will now show with this algorithm you have a much higher probability of settling down to a single value for all of the nodes, and hence has a much stronger advantage in comparison to the Randomized Consensus Algorithm where each node tosses a coin individually.

To show this consider the following: (we will assume $n = 3f + 1$)

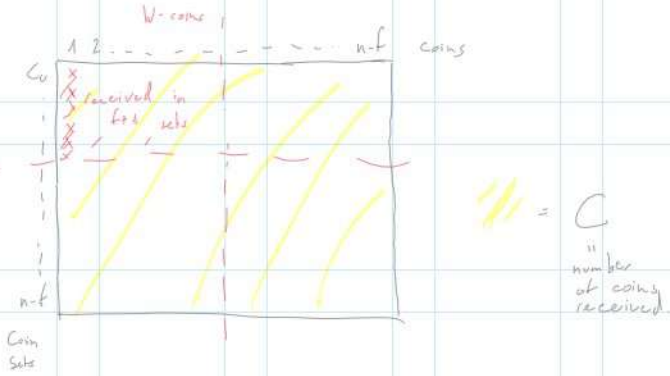
Lemma 16.23. *Let u be a node, and let W be the set of coins that u received in at least $f + 1$ different coin sets. It holds that $|W| \geq f - 1$.*

↳ This basically means that

$$W = \{ coin_i \text{ if } coin_i \in C_u : \Sigma C_{u_j} \geq f+1 \}$$

Such that W is a set of coins that appear at least in $f+1$ received coin sets (the one at line 3).

Visually this means the following



Proof Assume that the lemma does not hold, then it will hold that in $(n-f)$ coin sets ($> f+1$ as $n > 3f+1$) you would have at most f coins.

↳ this implies that all of the others $n-f$ coins are in f coin sets

↳ this implies that all of the others n-f coins are in f coin sets.

but then it holds, that the total number of received coins

$$|C| \leq \underbrace{(n-f)}_{\text{coin sets}} \cdot f + \underbrace{f}_{\text{coin sets}} \cdot (n-f)$$

all of coin sets multi with coins inside them

however, this would actually yield

$$|C| \leq 2(n-f) \cdot f$$

given that $n > 3f \leftrightarrow n-f > 2f$ it follows

$$|C| \leq (n-f)^2$$

↳ This is a contradiction by construction of C.

□

With the above schema it is now possible to see with the following lemma that using the shared coin we are able to highly reduce the termination runtime of the adapted Ben-Or algorithm due to the faster probability of selecting the same variable for all of the nodes.

In order to see that understand that

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

Proof. Let $w \in W$ be such a coin. By definition of W we know that w is in at least $f+1$ sets received by u. Since every other node also waits for $n-f$ sets before terminating, each node will receive at least one of these sets, and hence w must be seen by every node that terminates. □

Given the above it now follows that assuming that 0 is in $\geq f+1$ coin sets such that it is observed by all of the nodes; then every node will select 0.

It follows from the Lemmas above that such a probability is expressed by:

$$1 - \left(1 - \frac{1}{n}\right)^{|W|}$$

= all in the W set.

$1 - \left(1 - \frac{1}{n}\right)$ = the ψ set.

and given the above Lemma $|W| \geq f+1$, such that in the best case (i.e. 0 in exactly $f+1$)

$$1 - \left(1 - \frac{1}{n}\right)^{|W|} = 1 - (1 - 1/n)^{f+1}$$

given now $f+1 \approx n/3$ for large n it follows that all of the nodes will choose 0 with at least (recall many other cases)

$$1 - \left(1 - \frac{1}{n}\right)^{n/3} \approx 1 - \left(\frac{1}{e}\right)^{1/3} = 0{,}28$$

probability.

In a similar way you can compute the prob of selecting 1 as

$$\left(1 - \frac{1}{n}\right)^{n} \approx \frac{1}{e} = 0{,}37$$

(again only a lower bound in the prob).

This general ideas can be easily extended to see the following theorem for any $f < n/3$

We conclude therefore the chapter stating

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.*

Chapter 17 - Byzantine Agreement

First of all understand that

Byzantine Agreement = Consensus with f = byzantine.

So f do not simply crash but rather may implement any protocol. It follows that a protocol satisfing byzantine agreement must be resilient to a subset of nodes f implementing a wrong protocol trying to fool the original one.

implementing a wrong protocol trying to fool the original one.

⇒ We assume that nodes cannot forge a fake address here.

Notice that while the definition of validity was straightforward it is not now in the byzant. agreement case.

↳ There are different meaningful options, depending on the use case you would choose & and have to implement a protocol that respects it.

Definition 17.3 (Any-Input Validity). *The decision value must be the input value of* ***any*** *node.*

⇒ This basically validity of consensus.

Definition 17.4 (Correct-Input Validity). *The decision value must be the input value of a* ***correct*** *node.*

Definition 17.5 (All-Same Validity). *If* ***all*** *correct nodes start with the same input v, the decision value must be v.*

Definition 17.6 (Median Validity). *If the input values are orderable, e.g. $v \in \mathbb{R}$, byzantine outliers can be prevented by agreeing on a value close to the* ***median*** *of the correct input values – how close depends on the number of byzantine nodes f.*

} if ordered. Then here you can choose the value of a byzantine, as long as it is not totally off and makes generally sense.

Keeping this in mind we will now analyze the byzantine agreement model, first in the synchronous and then in the asynchronous case.

↳ Note that when describing an algorithm you should make explicit which validity condition the protocol satisfies, when solving byzantine agreement.

↳ We will focus in this sense on all-same-validity and any-input validity. Hence the lecture skips the very interesting correct-input validity case.

Necessary Condition for Reaching Byzantine Agreement

This section will show the following Thm:

Necessary Condition for Reaching Byzantine Agreement

This section will show the following Thm:

Theorem 17.13. *A network with n nodes cannot reach byzantine agreement with $f \geq n/3$ byzantine nodes.*

The idea is the usual, in order to agree all nodes must refer to either a King value or a Majority, such that you have a fix point that cannot be fooled.

We will see now show that in the case of $n = 3$ and $f = 1$ agreement is not possible and then we will generalize the concept to show the Thm above.

To understand this better look at the following algo

Algorithm 17.9 Byzantine Agreement with $f = 1$.

```
Code for node u, with input value x:
Round 1
Send tuple(v, x) to all other nodes v
Receive tuple(v, y) from all other nodes v
Store all received tuple(v, y) in a set S_u
Round 2
Send set S_u to all other nodes
Receive sets S_v from all nodes v
T := set of tuple(v, y) seen in at least two sets S_v, including own S_u
Let tuple(v, y) ∈ T be the tuple with the smallest value y
Decide on value y
```

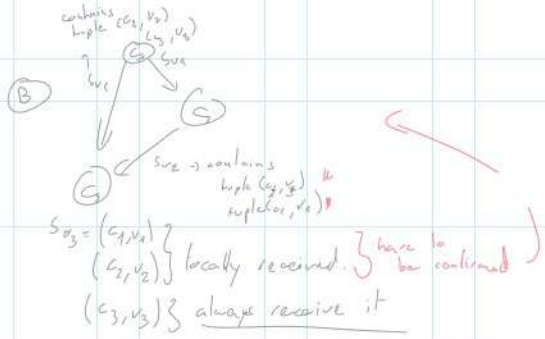
why king $f < n/3$ seems to work also with $f = n/3$ → look at the proof lo fatti aggiungendo due values diverse non facendo cambio → questo vale pure per king?

This is a way to solve agreement leveraging the sheer size and the concept of a majority for avoiding being fooled.

↳ It is then possible to see that

Lemma 17.10. *If $n \geq 4$, all correct nodes have the same set T.*

↳ This can be easily seen as follows:



$(c_2, v_2)\}$ locally received. } have to be confirmed
$(c_3, v_3)\}$ always receive it

So that in the above protocol you need a second node to support your value sent directly in the first round, such that it manages to reach T.

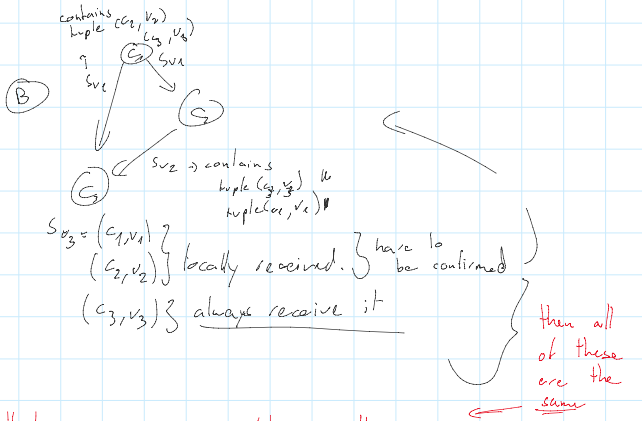
The idea now is that in the case the number of true nodes is ≥ 3 you have always a second correct node supporting your value such that it always makes it into T.

Moreover if the byzantine node sends a false message different to every node $\Rightarrow$ none of the correct nodes will have such values in T. If in contrast the byzantine node will send the same value 2 times then it will enter in all of the correct nodes T.

↳ Given now that all of the correct nodes have the same T it is straightforward to see that the algo above satisfies any-input validity

Notice moreover that this can be generally easily extended to accomodate all-same-validity input.

↳ In this case you would have the value of the correct nodes being represented more than 1 time in the sense that



such that you can obtain all same validity by deciding on the tuple with a specific value appearing at least two times.

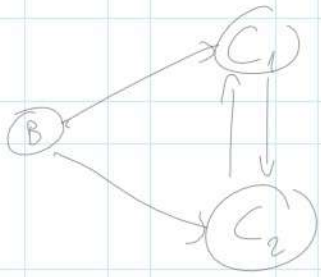
are the same

such that you can obtain all same validity by deciding on the tuple with a specific value appearing at least two times.

Lemma 17.12

Note now that in case of $n=3$ it is impossible to reach byzantine agreement with (all-the same validity)

↳ Proof:



receives c_2 and b

receives c_1 and b

then the byzantine node can sponsor two different values to c_1 and c_2 supporting the values they both have.

↳ Then each of the two nodes will decide on a different value and there is no agreement.

The proof with all-same-validity that comes in the script goes as follows (albeit the above proof seems more general to me.)

Proof. We will assume that the three nodes satisfy all-same validity and show that they will violate the agreement condition under this assumption.

assumption ↓

In order to achieve all-same validity, nodes have to deterministically decide for a value x if it is the input value of every correct node. Recall that a Byzantine node which follows the protocol is indistinguishable from a correct node. Assume a correct node sees that $n-f$ nodes including itself have an input value x. Then, by all-same validity, this correct node must deterministically decide for x.

In the case of three nodes $\{n-f=2\}$, a node has to decide on its own input value if another node has the same input value. Let us call the three nodes u, v and w. If correct node u has input 0 and correct node v has input 1, the byzantine node w can fool them by telling u that its value is 0 and simultaneously telling v that its value is 1. By all-same validity, this leads to u and v deciding on two different values, which violates the agreement condition. Even if u talks to v, and they figure out that they have different assumptions about w's value, u cannot distinguish whether w or v is byzantine. □

goes more in the direction previously discussed → but then it assumes that you are interested in any input validity, and that you choose by majority

Given this it is straightforward to see the theorem introduced at the beginning of the subchapter

Theorem 17.13. *A network with n nodes cannot reach byzantine agreement with $f \geq n/3$ byzantine nodes.*

Proof. Assume (for the sake of contradiction) that there exists an algorithm

So straight

Theorem 17.13. *A network with n nodes cannot reach byzantine agreement with $f \geq n/3$ byzantine nodes.*

Proof. Assume (for the sake of contradiction) that there exists an algorithm A that reaches byzantine agreement for n nodes with $f \geq \lceil n/3 \rceil$ byzantine nodes. We will show that A cannot satisfy all-same validity and agreement simultaneously.

Let us divide the n nodes into three groups of size $n/3$ (either $\lfloor n/3 \rfloor$ or $\lceil n/3 \rceil$, if n is not divisible by 3). Assume that one group of size $\lceil n/3 \rceil \geq n/3$ contains only Byzantine and the other two groups only correct nodes. Let one group of correct nodes start with input value 0 and the other with input value 1. As in Lemma 17.12, the group of Byzantine nodes supports the input value of each node, so each correct node observes at least $n - f$ nodes who support its own input value. Because of all-same validity, every correct node has to deterministically decide on its own input value. Since the two groups of correct nodes had different input values, the nodes will decide on different values respectively, thus violating the agreement property. □

So straightforward generalization of the above concept

We will now show two protocols for reaching byzantine agreement in the case of synchronous and asynchronous communication.

Recall

- asynchronous comm.:

→ upon receiving a message do ... (the do part takes no time).

- synchronous comm.:

→ sequentiality is respected; there are rounds that are equal for all nodes (here idea of synchronization).
↳ in each round a node might:
- receive & process messages
- send

Synchronous Model – The King Algorithm

We will show in this section that the following King algorithm solves byzantine agreement with all-same validity.

Algorithm 17.14 King Algorithm (for $f < n/3$)

```
1: x := my input value
2: for phase := 1 to f + 1 do
       Vote
3:     Broadcast value(x)
       Propose
4:     if some value(y) received at least n − f times then
5:         Broadcast propose(y)
6:     end if
7:     if some propose(z) received more than f times then
8:         x = z
9:     end if
       King
10:    Let node v_i be the predefined king of this phase i
11:    The king v_i broadcasts its current value w
12:    if received strictly less than n − f propose(y) then
13:        x = w
14:    end if
15: end for
```

the king value will converge here

so over

Lemma 17.15. *Algorithm 17.14 fulfills the all-same validity.*

so once all nodes have the same value at some phase they will simply propagate this all of the rest of the phases.

Lemma 17.15. *Algorithm 17.14 fulfills the all-same validity.*

Proof. If all correct nodes start with the same value, all correct nodes propose it in Line 5. All correct nodes will receive at least $n - f$ proposals, i.e., all correct nodes will stick with this value, and never change it to the king's value. This holds for all phases. □

Lemma 17.16. *If a correct node proposes x, no other correct node proposes y, with $y \neq x$, if $n > 3f$.*

Proof. Assume (for the sake of contradiction) that a correct node proposes value x and another correct node proposes value y. Since a good node only proposes a value if it heard at least $n - f$ value messages, we know that both nodes must have received their value from at least $n - 2f$ distinct correct nodes (as at most f nodes can behave byzantine and send x to one node and y to the other one). Hence, there must be a total of at least $2(n - 2f) + f = 2n - 3f$ nodes in the system. Using $3f < n$, we have $2n - 3f > n$ nodes, a contradiction. □

otherwise must also've got all f byzantine, so that would help

Lemma 17.17. *There is at least one phase with a correct king.*

Proof. There are $f + 1$ phases, each with a different king. As there are only f byzantine nodes, one king must be correct. □

Lemma 17.18. *After a phase with a correct king, the correct nodes will not change their values v anymore, if $n > 3f$.*

So either all change to the king value or they keep their value

Proof. If all correct nodes change their values to the king's value, all correct nodes have the same value. If some correct node does not change its value to the king's value, it received a proposal at least $n - f$ times, therefore at least $n - 2f$ correct nodes broadcasted this proposal. Thus, all correct nodes received it at least $n - 2f > f$ times (using $n > 3f$), therefore all correct nodes set their value to the proposed value, including the correct king. Note that only one value can be proposed more than f times, which follows from Lemma 17.16. With Lemma 17.17, no node will change its value after this phase. □

Theorem 17.19. *Algorithm 17.14 solves byzantine agreement.*

Proof. The king algorithm reaches agreement as either all correct nodes start with the same value, or they agree on the same value latest after the phase where a correct node was king according to Lemmas 17.17 and 17.18. Because of Lemma 17.15 we know that they will stick with this value. Termination is guaranteed after $3(f + 1)$ rounds, and all-same validity is proved in Lemma 17.19. □

as you have 3 broadcasts taking one round each time for each of the f+1 phases

↳ Any input validity is also guaranteed by the phase with the correct king there everything will converge.

! Notice that the above needs f+1 predefined kings. This is a task requiring byzantine agreement by itself, we will see later how to solve this issue.

We will now see that this algorithm in the synchronous model has optimal runtime.

Lower bound on Runtime for Consensus

notice special and more restrictive

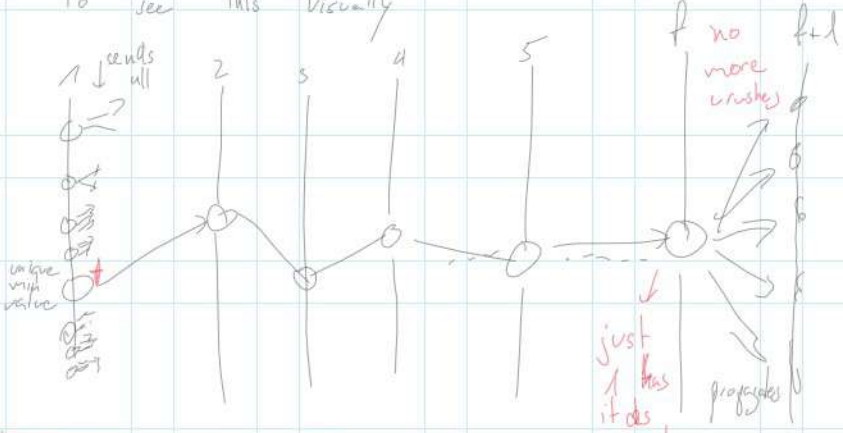
We will show that in the case of consensus the optimal runtime f+1, in the synchronous case.

Theorem 17.20. *A synchronous algorithm solving consensus in the presence of f crashing nodes needs at least $f + 1$ rounds, if nodes decide for the minimum*

Theorem 17.20. *A synchronous algorithm solving consensus in the presence of f crashing nodes needs at least $f+1$ rounds, if nodes decide for the minimum seen value.*

Proof. Let us assume (for the sake of contradiction) that some algorithm A solves consensus in f rounds. Some node u_1 has the smallest input value x, but in the first round u_1 can send its information (including information about its value x) to only some other node u_2 before u_1 crashes. Unfortunately, in the second round, the only witness u_2 of x also sends x to exactly one other node u_3 before u_2 crashes. This will be repeated, so in round f only node u_{f+1} knows about the smallest value x. As the algorithm terminates in round f, node u_{f+1} will decide on value x, all other surviving (correct) nodes will decide on values larger than x. □

To see this visually



So you assume in the above crashing behavior send to exactly 1 and then fail.

It is now clear that the crashing nodes is a subset of byzantine node thus bound also applies to byzantine nodes.

We turn to the asynchronous case now.

Asynchronous Byzantine Agreement

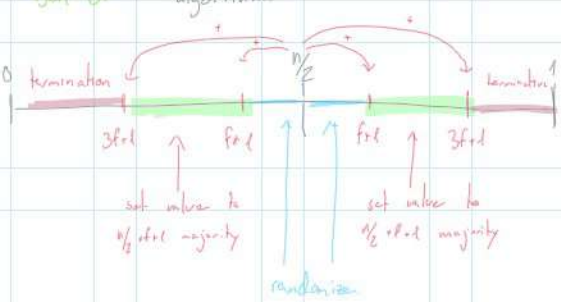
We will see that the following adaption of the Ben-Or algorithm seen the last week for solving consensus in a probabilistic way, will manage to solve byzantine agreement.

Algorithm 17.21 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/10$) → more restrictive

```
1: x_u ∈ {0,1}                    ◁ input bit
2: round = 1                      ◁ round
3: while true do
4:   Broadcast propose(x_u,round)
5:   Wait until n − f propose messages of current round arrived
6:   if at least n/2 + 3f + 1 propose messages contain same value x then
7:     Broadcast propose(x,round + 1)
8:     Decide for x and terminate
9:   else if at least n/2 + f + 1 propose messages contain same value x then
10:    x_u = x
11:  else
12:    choose x_u randomly, with Pr[x_u = 0] = Pr[x_u = 1] = 1/2
13:  end if
14:  round = round + 1
```

```
9:  else if at least ⌈(n+f+1)/2⌉ propose messages contain same value x then
10:   x_u = x
11: else
12:   choose x_u randomly, with Pr[x_u = 0] = Pr[x_u = 1] = 1/2
13: end if
14: round = round + 1
15: end while
```

The idea to understand the above is the same of the last chapter as we are again dealing with a Ben-Or algorithm.



The idea is now that you cannot be more than 2f apart, in the sense that for instance if:

- a node that observes n-f propose(z) values, must have obtained them from at least n-2f different correct nodes.

- assume now that a different node observes again n-f propose(z) (worst case) then we know again that these should come from n-2f different correct nodes.

↳ As we know that there are in total n-f different correct nodes it follows that at most f nodes might differ in their two sets.

⇒ $n-f-(n-2f) = f$

So that in general the difference in proposals among two nodes can be max 2f (one of different correct and f of byzantine).

It follows that you will always stay in adjacent regions in the pic above.

in adjacent regions in the pic. / above.

↳ Through this it follows that:

- if a node terminates, all of the other either terminates or are in the adjacent region and will terminate in the following round

- it now follows that if all the correct nodes start with the same value you have

$n - 2f$ prop (2).

Noticing that $5f < \frac{n}{2}$ by def it follows

$$5f < n/2 \quad | + n/2$$

$$n/2 + 5f < n \quad | - 2f$$

$$n/2 + 3f < n - 2f$$

such that you always enter the termination phase and the algo reaches all-same-validity conclusion.

- it finally exists in the case nodes are in or to have a positive chance of having all the same values such that with the above the algorithm terminates and reaches agreement.

- Notice moreover that also any-input validity and correct-input validity (for the binary case)

↳ i.e. as a byzantine node cannot push a completely different value in the algo above any-input validity is guaranteed

↳ Given the all-same-validity it follows that if you are not in the above case you must

↳ Given the all-same-validity it follows that if you are not in the above case you must have at least $\{0, 1\}$ as starting values in different correct nodes. such that both $0 \wedge 1$ will be fine as selection.

⚠ Issue with Ben-Or algorithms:

Termination time is ~~exponential~~

So we only need to worry about termination: We have already seen that as soon as one correct node terminates (Line 8) everybody terminates in the next round. So what are the chances that some node u terminates in Line 8? Well, we can hope that all correct nodes randomly propose the same value (in Line 12). Maybe there are some nodes not choosing randomly (entering Line 10 instead of 12), but according to Lemma 17.22 they will all propose the same.

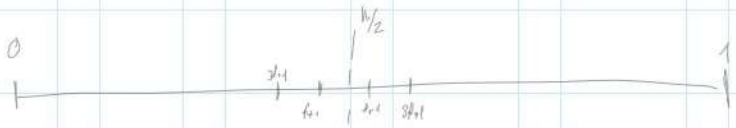
Thus, at worst all $n - f$ correct nodes need to randomly choose the same bit, which happens with probability $2^{-(n-f)+1}$. If so, all correct nodes will send the same propose message, and the algorithm terminates. So the expected running time is exponential in the number of nodes n in the worst case. □

cannot have two different proposals $x_u = x$ $x_v = y$ as that would imply $(n/2 + 1)$ 2 different correct nodes $> \#$ ↯

We conclude the section with the following two important remarks

- This Algorithm is a proof of concept that asynchronous byzantine agreement can be achieved. Unfortunately this algorithm is not useful in practice, because of its runtime.
- Note that for $f \in O(\sqrt{n})$, the probability for some node to terminate in Line 8 is greater than some positive constant. Thus, Algorithm 17.21 terminates within expected constant number of rounds for small values of f.

idea: if f is small:



⇒ prob of having random samples $> \frac{n}{2} + 3f + 1$ is then given by the CLT

$$n \cdot \frac{1}{2} \pm \frac{1}{\sqrt{n}} \cdot n \cdot \sqrt{\frac{1}{4}} \cdot \Phi \text{ quantile}$$

which is very high.

We will show now that relying on some oracle on which all of the nodes will rely when setting their value can improve the runtime.

We will show now that relying on some oracle on which all of the nodes will rely when setting their value can improve the runtime.

Random Oracle and Bitstring

Definition 17.24 (Random Oracle). *A random oracle is a trusted (non-byzantine) random source which can generate random values.*

Algorithm 17.25 Algorithm 17.21 with a Magic Random Oracle
1: Replace Line 12 in Algorithm 17.21 by
2: **return** c_i, where c_i is ith random bit by oracle

↳ Notice that the bit must be random otherwise we are in the impossibility of consensus.

Theorem 17.26. *Algorithm 17.25 plugged into Algorithm 17.21 solves asynchronous byzantine agreement in expected constant number of rounds.*

Proof. If there is a large majority for one of the input values in the system, all nodes will decide within two rounds since Algorithm 17.21 satisfies all-same-validity; the coin is not even used.

If there is no significant majority for any of the input values at the beginning of algorithm 17.21, all correct nodes will run Algorithm 17.25. Therefore, they will set their new value to the bit given by the random oracle and terminate in the following round.

If neither of the above cases holds, some of the nodes see an $n/2 + f + 1$ majority for one of the input values, while other nodes rely on the oracle. With probability $1/2$, the value of the oracle will coincide with the deterministic majority value of the other nodes. Therefore, with probability $1/2$, the nodes will terminate in the following round. The expected number of rounds for termination in this case is 3. □

Remarks:

- Unfortunately, random oracles are a bit like pink fluffy unicorns: they do not really exist in the real world. Can we fix that?

Definition 17.27 (Random Bitstring). *A **random bitstring** is a string of random binary values, known to all participating nodes when starting a protocol.*

Algorithm 17.28 Algorithm 17.21 with Random Bitstring
1: Replace Line 12 in Algorithm 17.21 by
2: **return** b_i, where b_i is ith bit in common random bitstring

Remarks:

- But is such a precomputed bitstring really random enough? We should be worried because of Theorem 16.13.

i.e. does it become deterministic?

↳ **Theorem 17.29.** *If the scheduling is worst-case, Algorithm 17.28 plugged into Algorithm 17.21 does not terminate.*

↳ we will prove this next the idea is however that if the byzantine nodes are very well aware of the entire random bitstring and the state of the system, then it is easy to fool it.

Proof. We start Algorithm 17.28 with the following input: $n/2 + f + 1$ nodes have input value 1, and $n/2 - f - 1$ nodes have input value 0. Assume w.l.o.g. that the first bit of the random bitstring is 0.

If the second random bit in the bitstring is also 0, then a worst-case scheduler will let $n/2 + f + 1$ nodes see all $n/2 - f + 1$ values 1, these will therefore deterministically choose the value 1 as their new value. Because of scheduling (or byzantine nodes), the remaining $n/2 - f - 1$ nodes receive strictly less than

that the first bit of the random bitstring is 0.

If the second random bit in the bitstring is also 0, then a worst-case scheduler will let $n/2+f+1$ nodes see all $n/2+f+1$ values 1, these will therefore deterministically choose the value 1 as their new value. Because of scheduling (or byzantine nodes), the remaining $n/2-f-1$ nodes receive strictly less than $n/2+f+1$ values 1 and therefore have to rely on the value of the shared coin, which is 0. The nodes will not come to a decision in this round. Moreover, we have created the very same distribution of values for the next round (which has also random bit 0).

If the second random bit in the bitstring is 1, then a worst-case scheduler can let $n/2-f-1$ nodes see all $n/2+f+1$ values 1, and therefore deterministically choose the value 1 as their new value. Because of scheduling (or byzantine nodes), the remaining $n/2+f+1$ nodes receive strictly less than $n/2+f+1$ values 1 and therefore have to rely on the value of the shared coin, which is 0. The nodes will not decide in this round. And we have created the symmetric situation for input value 1 that is coming in the next round.

So if the current and the next random bit are known, worst-case scheduling will keep the system in one of two symmetric states that never decide.

□

In the next chapter we will investigate the issue further and we will explore ways to achieve to achieve a constant runtime without having to rely on fluffy unicorns but rather based on solid implementable ideas.

Chapter 18 - Broadcast and Shared Coins

Recall that so far we have the following situation

	Syncronous	Asynchronous
Consensus	- King (must also apply here, notice however f< n/3.)	- Paxos (issue with termination) - Ben-Or (issue with runtime) - Ben-Or with Shared Coin (good runtime not resistant to bad scheduler)
Byzantine	- King (good runtime)	- Ben-Or (issue with runtime/adaption)

By 2

We will focus on the right hand side of the table and how to solve the runtime issue there.

↳ The idea is again to create a shared coin that is resilient to a worst case scheduler.

First define properly what a shared coin is:

Definition 18.1 (Shared Coin). *A* ***shared coin*** *is a binary random variable shared among all nodes. It is 0 for all nodes with constant probability and 1 for all nodes with constant probability. The shared coin is allowed to fail (be 0 for some nodes and 1 for other nodes) with constant probability.*

It is now straightforward to see that the modified ben-or looked in the consenses case with $C_v = 0$ with $1/n$ prob else $C_v = 1$ is nothing else than a shared coin.

↳ Notice however that it is not resilient to worst case scheduler:

Lemma 18.2. *Algorithm 16.21 has exponential expected running time under worst-case scheduling.*

Proof. In Algorithm 16.21, worst-case scheduling may hide up to f rare zero coinflips. In order to receive a zero as the outcome of the shared coin, the nodes need to generate at least $f+1$ zeros. The probability for this to happen is $(1/n)^{f+1}$, which is exponentially small for $f \in \Omega(n)$. In other words, with worst-case scheduling, with probability $1-(1/n)^{f+1}$ the shared coin will be 1. The worst-case scheduler must make sure that some nodes will always deterministically go for 0, and the algorithm needs n^{f+1} rounds until it terminates. □

not anymore 1 due to the worst case scheduler

worst case scheduler does not pass 0 when these happens ⇓ i.e. they always crash so that to generate the first 0 need to generate n^f 0.

⇓ as otherwise if you always choose 1 algo fully deterministic

So we see that despite theoretically solving the runtime issue, the shared-coin cannot be implemented by a simple broadcast action, due to the exponential runtime for the worst-scheduler case.

↳ We will now propose an adapted shared coin that relies on a common ground truth table each node might

shared coin that relies on a common ground truth table each node might refer to.

↳ We will then see that through such common table it is possible to implement a shared coin which runtime is unaffected by worst-case schedulers.

The idea is the one of using a so called blackboard:

Definition 18.3 (Blackboard Model). *The* ***blackboard*** *is a trusted authority which supports two operations. A node can* ***write*** *its message to the blackboard and a node can* ***read*** *all the values that have been written to the blackboard so far.*

Remarks:

- We assume that the nodes cannot reconstruct the order in which the messages are written to the blackboard since the system is asynchronous.

It follows now that the following algorithm implements a crash-resilient blackboard implementing a shared coin

Algorithm 18.4 Crash-Resilient Shared Coin with Blackboard (for node u)

```
1: while true do
2:   Choose new local coin c_u = +1 with probability 1/2, else c_u = -1
3:   Write c_u to the blackboard
4:   Set C = Read all coinflips on the blackboard
5:   if |C| ≥ n^2 then
6:     return sign(sum(C))
7:   end if
8: end while
```

recall returns finishes the while loop.

Remarks:

- In Algorithm 18.4 the outcome of a coinflip is -1 or $+1$ instead of 0 or 1 because it simplifies the analysis, i.e., "$-1 \approx 0$".

Notice that the algorithm above is wait-free you do not wait for x-messages before continuing.

→ Notice that in the above a single node might write down the n^2 values and terminate.

It is now possible to see that for n^2 values written down we have

Theorem 18.5 (Central Limit Theorem). *Let $\{X_1, X_2, \ldots, X_N\}$ be a sequence of independent random variables with $Pr[X_i = -1] = Pr[X_i = 1] = 1/2$ for all $i = 1, \ldots, N$. Then for every positive real number z,*

$$\lim_{N\to\infty} \Pr\left[\sum_{i=1}^{N} X_i \geq z\sqrt{N}\right] = 1 - \Phi(z) > \frac{1}{\sqrt{2\pi}} \frac{z}{z^2+1} e^{-z^2/2},$$

where $\Phi(z)$ is the cumulative distribution function of the standard normal distribution evaluated at z.

Notice in the above

Notice in the above

$$E(X) = \tfrac{1}{2} \cdot -1 + \tfrac{1}{2} \cdot 1 = 0$$

$$Var(X) = E(x^2) - E(x)^2$$

$$= 1 \cdot \tfrac{1}{2} + 1 \cdot \tfrac{1}{2} = 1$$

So that $E(\sum x_i) = 0$

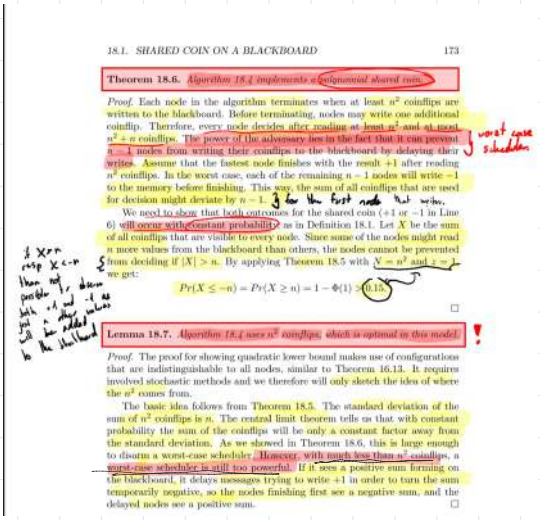
$$Var(\sum_i^{N^2} X_i) = N^2$$

(so that you see that the above is the very standard CLT, i.e.

$$\sqrt{N}\left(\sum \frac{x_i - \mu}{\sigma}\right) \to N(0,1)$$

where you already plugged in the stuff.

It now follows:



18.1. SHARED COIN ON A BLACKBOARD 173

Theorem 18.6. *Algorithm 18.4 implements a* [illegible] *shared coin.*

Proof. Each node in the algorithm terminates when at least n^2 coinflips are written to the blackboard. Before terminating, nodes may write one additional coinflip. Therefore, every node decides after reading at least n^2 and at most $n^2 + n$ coinflips. The power of the adversary lies in the fact that it can prevent $n - 1$ nodes from writing their coinflips to the blackboard by delaying their writes. Assume that the fastest node finishes with the result $+1$ after reading n^2 coinflips. In the worst case, each of the remaining $n - 1$ nodes will write -1 to the memory before finishing. This way, the sum of all coinflips that are used for decision might deviate by $n - 1$. [illegible]

We need to show that both outcomes for the shared coin ($+1$ or -1 in Line 6) will occur with constant probability as in Definition 18.1. Let X be the sum of all coinflips that are visible to every node. Since some of the nodes might read more values from the blackboard than others, the nodes cannot be prevented from deciding if $|X| > n$. By applying Theorem 18.5 with $K = n^2$ and $\varepsilon = 1$ we get:

$$Pr[X \leq -n] = Pr[X \geq n] = 1 - \Phi(1) > 0.15.$$

Lemma 18.7. *Algorithm 18.4 uses n^2 coinflips, which is optimal in this model.*

Proof. The proof for showing quadratic lower bound makes use of configurations that are indistinguishable to all nodes, similar to Theorem 16.13. It requires involved stochastic methods and we therefore will only sketch the idea of where the n^2 comes from.

The basic idea follows from Theorem 18.5. The standard deviation of the sum of n^2 coinflips is n. The central limit theorem tells us that with constant probability the sum of the coinflips will be only a constant factor away from the standard deviation. As we showed in Theorem 18.6, this is large enough to disarm a worst-case scheduler. However, with much less than n^2 coinflips, a worst-case scheduler is still too powerful. If it sees a positive sum forming on the blackboard, it delays messages trying to write $+1$ in order to turn the sum temporarily negative, so the nodes finishing first see a negative sum, and the delayed nodes see a positive sum. □

So basically the idea of this algorithm is to leverage the CLT for having a prob of returning 1 or 0 with constant probability that cannot be easily disrupted by a worst case scheduler.

Important

Remarks:

- Algorithm 18.4 cannot tolerate even one byzantine failure: assume the byzantine node generates all the n^2 coinflips in every round due to worst-case scheduling. Then this byzantine node can make sure that its coinflips always sum up to a value larger than n, thus making the outcome -1 impossible.

the byzantine node generates all the n^2 coinflips in every round due to worst-case scheduling. Then this byzantine node can make sure that its coinflips always sum up to a value larger than n, thus making the outcome -1 impossible.

- In Algorithm 18.4, we assume that the blackboard is a trusted central authority. Like the random oracle of Definition 17.24, assuming a blackboard does not seem practical. However, fortunately, we can use advanced broadcast methods in order to implement something like a blackboard with just messages.

↳ So basically we are still in the unicorns world.

↳ We will however see next how it is possible to implement such a blackboard through standard broadcasting messages.

18.2 Broadcast Abstractions

Definition 18.8 (Accept). *A message received by a node v is called* ***accepted*** *if node v can consider this message for its computation.*

Definition 18.9 (Best-Effort Broadcast). ***Best-effort broadcast*** *ensures that a message that is sent from a correct node u to another correct node v will eventually be received and accepted by v.*

Remarks:

- Note that best-effort broadcast is equivalent to the simple broadcast primitive that we have used so far.
- Reliable broadcast is a stronger paradigm which implies that byzantine nodes cannot send different values to different nodes. Such behavior will be detected.

Definition 18.10 (Reliable Broadcast). ***Reliable broadcast*** *ensures that the nodes eventually agree on all accepted messages. That is, if a correct node v considers message m as accepted, then every other node will eventually consider message m as accepted.*

In order to implement such broadcast you can use the following algo:

Algorithm 18.11 Asynchronous Reliable Broadcast (code for node u)

1: Broadcast own message $\texttt{msg}(u)$
2: **upon** receiving $\texttt{msg}(v)$ from v or $\texttt{echo}(w, \text{msg}(v))$ from $n - 2f$ nodes w:
3: Broadcast $\texttt{echo}(u, \text{msg}(v))$
4: **upon** receiving $\texttt{echo}(w, \text{msg}(v))$ from $n - f$ nodes w:
5: Accept $\texttt{msg}(v)$

more than f

covers some of f very slow correct messages

Theorem 18.12. *Algorithm 18.11 satisfies the following properties:*

1. *If a correct node broadcasts a message reliably, it will eventually be accepted by every other correct node.*
2. *If a correct node has not broadcast a message, it will not be accepted by any other correct node.* !?
3. *If a correct node accepts a message, it will be eventually accepted by every correct node*

This algorithm can tolerate $f < n/3$ byzantine nodes or $f < n/2$ crash failures.

easy to verify. Proof skipped

↓ also assure point 3

if you accept in one node you will eventually have this echo from $n-2f$ correct nodes.

Remarks:

- Algorithm 18.11 does not solve consensus according to Definition 16.1. It only makes sure that all messages of correct nodes will be accepted eventually for correct nodes, this corresponds to sending and receiving messages in the asynchronous time model (Model 16.2).
- The algorithm has a linear message overhead since every node again broadcasts every message. $n \to n^2$
- Note that byzantine nodes can issue arbitrarily many messages. This may be a problem for protocols where each node is only allowed to send one message (per round). Can we fix this, for instance with sequence numbers?

fix

Definition 18.13 (FIFO Reliable Broadcast). *The* ***FIFO (reliable) broadcast*** *defines an order in which the messages are accepted in the system. If a node u broadcasts message m_1 before m_2, then any node v will accept message m_1 before m_2.*

We will now see that the following implements a FIFO reliable broadcast.

Algorithm 18.14 FIFO Reliable Broadcast (code for node u)

1: Broadcast own round r message $\texttt{msg}(u, r)$
2: **upon** receiving first message $\texttt{msg}(v, r)$ from node v for round r or $n - 2f$ $\texttt{echo}(w, \texttt{msg}(v, r))$ messages:
3: Broadcast $\texttt{echo}(u, \texttt{msg}(v, r))$
4: **upon** receiving $\texttt{echo}(w, \texttt{msg}(v, r))$ from $n - f$ nodes w:
5: **if** accepted $\texttt{msg}(v, r - 1)$ **then**
6: Accept $\texttt{msg}(v, r)$
7: **end if**

you just accept when previous round is acc.

Theorem 18.15. *Algorithm 18.14 satisfies the properties of Theorem 18.12. Additionally, Algorithm 18.14 makes sure that no two messages* $\texttt{msg}(v, r)$ *and* $\texttt{msg'}(v, r)$ *are accepted from the same node. It can tolerate* $f < n/3$ *byzantine nodes or* $f < n/2$ *crash failures.*

↳ Proof: in the case of crashing nodes no-one proposes a msg'(v,r).

↳ You can then see that with $f < n/2$ the algo still holds. but with $f = n/2$ everything crashes in line 2.

For the byzantine case it is clear that if echo(v, r) and echo(v', r) should be received by $n-2f$ correct nodes correspondingly, given now

$$2n - 4f > n - f$$

(n − f: number of total correct nodes.)

Notice that there exists a more strict ordered broadcasting, which essentially corresponds to stable replication.

This is

Definition 18.16 (Atomic Broadcast). ***Atomic broadcast*** *makes sure that all messages are received in the same order by every node. That is, for any pair of nodes* u, v*, and for any two messages* m_1 *and* m_2*, node* u *receives* m_1 *before* m_2 *if and only if node* v *receives* m_1 *before* m_2*.*

↳ So the difference is that the FIFO reliable broadcast just holds for 1 node sending two messages (same node) then a second node always accepts them in order.

in order.

Given this new way of reliable broadcast it is easy to see that; the following implements a blackboard:

Algorithm 18.17 Simulating a write to the blackboard (code for node u)

1: FIFO-broadcast $\texttt{msg}(u)$ to all nodes
2: Participate in FIFO-broadcast of all messages and write all received messages to a local blackboard
3: Wait until accepted own $\texttt{msg}(u)$

Algorithm 18.18 Simulating a read from the blackboard (code for node u)

1: Request local blackboards from all nodes
2: Wait for $n - f$ responses and add all newly seen messages to own local blackboard

Theorem 18.19. *Algorithm 18.4 together with the write operation from Algorithm 18.17 and the read operation from Algorithm 18.18 solves asynchronous binary agreement for $f < n/2$ crash failures.*

Proof. The upper bound for the number of crash failures results from the upper bound in Theorem 18.15.

Algorithm 18.17 simulates a write operation: by accepting the own message/coinflip, a node verifies that $n - f$ nodes have received its most recent generated coinflip $\texttt{coin}(c_u)$ and written it to their local blackboard. At least $n - 2f > 1$ of these nodes will never crash and the value therefore can be considered as stored on the blackboard. While u's own value is not accepted and therefore not stored, node u will not generate new coinflips. Therefore, at any point of the algorithm, there are at most n additional generated coinflips next to the accepted coins. This corresponds to the write operation in Algorithm 18.4.

Algorithm 18.18 simulates a read operation from the blackboard. A node reads a value by receiving the local views of the blackboards from at least $n - f$ nodes. Note that each written value is stored on $n - f$ local blackboards. Therefore, a node will see at least $(n - f) + (n - f) - n = n - 2f > 1$ copies of each value that was written to the blackboard so far.

This shows that the read and write operations are equivalent to the same operations in Algorithm 18.4. Assume now that some correct node has terminated after reading n^2 coinflips. Since each node reads the stored coinflips before generating a new one in the next round, there will be at most n additional coins accepted by any other node before termination. This setting is equivalent to Theorem 18.6 and the rest of the analysis is therefore analogous to the analysis in that theorem. □

this not entirely clear.

I try to understand it at some point.

Remarks:

- So finally we can deal with worst-case crash failures *and* worst-case scheduling

⇒ Notice that therefore we have solved the runtime issue for

1. Worst-case crash-failures
2. Worst-case scheduling

in the asynchronous case.

Notice moreover: noone has solved asynchronous byzantine agreement in constant runtime without relying on cryptography ⇒ area of research.

We will now show constant asynchronous

We will now show constant asynchronous byzantine agreement leveraging cryptography.

Byzantine Agreement - Asynchron Using Cryptography

Definition 18.20 (Threshold Secret Sharing). *Let $t, n \in \mathbb{N}$ with $1 \leq t \leq n$. An algorithm that distributes a secret among n participants such that t participants need to collaborate to recover the secret is called a (t, n)-**threshold secret sharing** scheme.*

Definition 18.21 (Signature). *Every node can **sign** its messages in a way that no other node can forge, thus nodes can reliably determine which node a signed message originated from. We denote a message x signed by node u with $\texttt{msg}(x)_u$.*

Algorithm 18.22 (t, n)-Threshold Secret Sharing

1: Input: A secret s, represented as a real number.

Secret distribution by dealer d

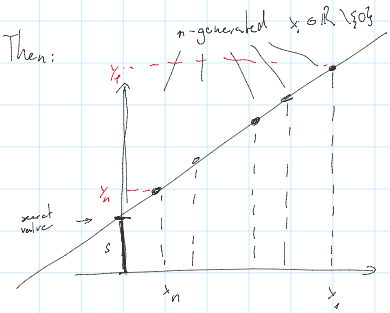
2: Generate $t-1$ random numbers $a_1, \ldots, a_{t-1} \in \mathbb{R}$
3: Obtain a polynomial p of degree $t-1$ with $p(x) = s + a_1 x + \cdots + a_{t-1} x^{t-1}$
4: Generate n distinct $x_1, \ldots, x_n \in \mathbb{R} \setminus \{0\}$
5: Distribute share $\texttt{msg}(x_1, p(x_1))_d$ to node v_1, ..., $\texttt{msg}(x_n, p(x_n))_d$ to node v_n

Secret recovery

6: Collect t shares $\texttt{msg}(x_u, p(x_u))_d$ from at least t nodes
7: Use Lagrange's interpolation formula to obtain $p(0) = s$

Pretty simple idea, think in low dimensions i.e. a polynom of degree = 1.

↳ Then:



then obvious that colluding in two

$$y_n = s + a_1 \cdot x_n$$
$$y_1 = s + a_1 x_1$$

⇒ two unknown, two equations ⇒ solve the system.

Remarks:

- Algorithm 18.22 relies on a trusted dealer, who broadcasts the secret shares to the nodes.
- Note that the communication between the dealer and the nodes must be private, i.e., a byzantine party cannot see the shares sent to the correct nodes.
- Using an $(f+1, n)$-threshold secret sharing scheme, we can encrypt messages in such a way that byzantine nodes alone cannot decrypt them.

⇒ so the idea is basically to assume the above two points and then use a $(f+1, n)$-threshold secret such that the byzantine node cannot get to

above two points and then use a $(f+1, n)$-threshold secret such that the byzantine nodes cannot get to the secret by colluding among them

It is now straightforward to see that the following basic concept solves byzantine agreement as byzantine nodes become harmless, i.e. might not alter shared coins

Algorithm 18.23 Preprocessing Step for Algorithm 18.24 (code for dealer d)

1: According to Algorithm 18.22, choose polynomial p of degree f
2: **for** $i = 1, \ldots, c$ **do**
3: Choose coinflip c_i, where $c_i = 0$ with probability 1/2, else $c_i = 1$
4: Using Algorithm 18.22, generate n shares $(x_1^i, p(x_1^i)), \ldots, (x_n^i, p(x_n^i))$ for c_i
5: **end for**
6: Send shares $\mathtt{msg}(x_1^1, p(x_1^1)), \ldots, \mathtt{msg}(x_n^c, p(x_n^c))_d$ to node u

Algorithm 18.24 Shared Coin using Secret Sharing (cd. iteration)

1: Replace Line 12 in Algorithm 17.21 by
2: Request shares from all but $f-1$ nodes.
3: Using Algorithm 18.22, let c_r be the value reconstructed from the shares
4: **return** c_r.

Proof. In Line 4 of Algorithm 18.24, the nodes collect shares from $f+1$ nodes. Since a byzantine node cannot forge the signature of the dealer, it is restricted to either send its own share or decide to not send it at all. Therefore, each correct node will eventually be able to reconstruct secret c_r of round r correctly in Line 4 of the algorithm. The running time analysis follows then from the analysis of Lemma 17.26. □

Notice now that leveraging cryptography we can achieve a lower bound for the synchronous runtime $(f+1)$ previously shown this will be shown next

On cryptographic lower bound - Synchronous Case

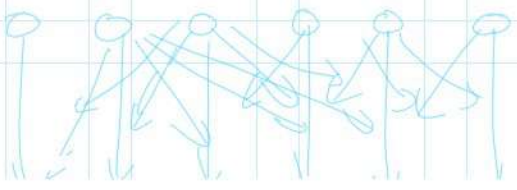
The idea relies on hash-functions:

Definition 18.26 (Cryptographic Hash Function). *A hash function $hash : U \to S$ is called **cryptographic**, if for a given $z \in S$ it is computationally hard to find an element $x \in U$ with $hash(x) = z$.*

Algorithm 18.27 Simple Synchronous Byzantine Shared Coin (for node u)

1: Each node has a public key that is known to all nodes.
2: Let r be the current round of Algorithm 17.21
3: Broadcast $\mathtt{msg}(r)_u$, i.e., round number r signed by node u
4: Compute $h_v = \mathrm{hash}(\mathtt{msg}(r)_v)$ for all received messages $\mathtt{msg}(r)_v$
5: Let $h_{min} = \min_v \ h_v$
6: **return** least significant bit of h_{min}

↳ Notice this is a shared coin as



min(hash)

↑ this will be a correct node with given probability

} and will return hence 0 or 1 with given prob.

} if byzantine then might be that two nodes different hash

happens with some pos prob.

⇒ Overall: it is clear that the above is the exact def of a shared coin.

Note:

Hashing helps to restrict byzantine power since a byzantine node cannot compute the smallest hash.

↳ and send it to some nodes.

↳ notice moreover that you need to be in the synchronous case for the same reason; i.e. at each round byzantine cannot observe min. hash.

Theorem 18.28. *Algorithm 18.27 plugged into Algorithm 17.21 solves synchronous byzantine agreement in expected 5 rounds for up to $f < n/10$ byzantine failures.*

Proof. With probability 1/3 the minimum hash value is generated by a byzantine node. In such a case, we can assume that not all correct nodes will receive the byzantine value and thus, different nodes might compute different values for the shared coin.

With probability 2/3, the shared coin will be from a correct node, and with probability 1/2 the value of the shared coin will correspond to the value which was deterministically chosen by some of the correct nodes. Therefore, with probability 1/3 the nodes will reach consensus in the next iteration of Algorithm 17.21. The expected number of rounds is:

$$1 + \sum_{i=0}^{\infty} 2 \cdot \left(\frac{2}{3}\right)^i = 5$$

Chapter 19

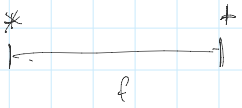
This module confronts with the notion of logical

This module contrasts with the notion of logical time. The idea is to reason about temporal dependencies in our systems.

The idea is that when constructing distributed systems you should be able to reason on these, and make explicit what type of time consistency your system should guarantee.

Definition 19.1 (Object). *An **object** is a variable or a data structure storing information.*

Definition 19.2 (Operation). *An **operation** f accesses or manipulates an object. The operation f starts at wall-clock time f_* and ends at wall-clock time $f_\dagger$.*



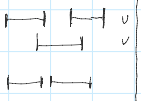
You do not know exactly the exact point/interval when the operation occurs.

If $f_\dagger < g^*$, we write $f \lessdot g$.

Definition 19.3 (Execution). *An **execution** E is a set of operations on one or multiple objects that are executed by a set of nodes.*

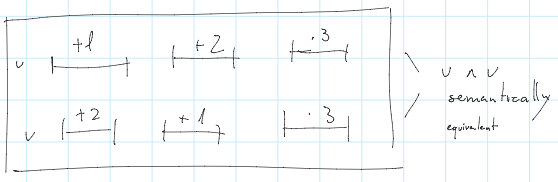
Definition 19.4 (Sequential Execution). *An execution restricted to a single node is a **sequential execution**. All operations are executed sequentially, which means that no two operations f and g are concurrent, i.e., we have $f < g$ or $g < f$.*

Definition 19.5 (Semantic Equivalence). *Two executions are **semantically equivalent** if they contain exactly the same operations. Moreover, each pair of corresponding operations has the same effect in both executions.*



This is giving me headache, it does not simply mean that it contains the same set of operations that can be performed at different times.

It also means that pairs of operations performed at different point in time should give the same result.



- For example, when dealing with a stack object, corresponding pop

- For example, when dealing with a stack object, corresponding pop operations in two different semantically equivalent executions must yield the same element of the stack.
- In general, the notion of semantic equivalence is non-trivial and dependent on the type of the object.

Definition 19.6 (Linearizability). *An execution E is called* ***linearizable*** *(or atomically consistent), if there is a sequence of operations (sequential execution) S such that:*

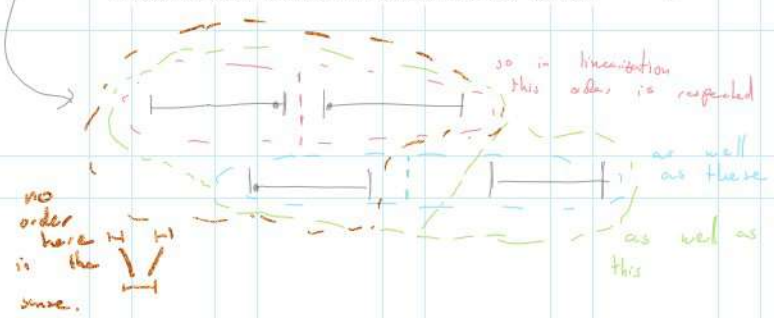
- *S is correct and semantically equivalent to E.*
- *Whenever $f < g$ for two operations f, g in E, then also $f < g$ in S.*

Notice:

Definition 19.7. *A* ***linearization point*** *of operation f is some $f_\bullet \in [f_\vee, f_\dagger]$.*

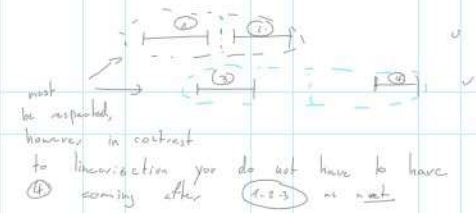
Lemma 19.8. *An execution E is linearizable if and only if there exist linearization points such that the sequential execution S that results in ordering the operations according to those linearization points is semantically equivalent to E.*

Proof. Let f and g be two operations in E with $f_\dagger < g_\bullet$. Then by definition of linearization points we also have $f_\bullet < g_\bullet$ and therefore $f < g$ in S. □



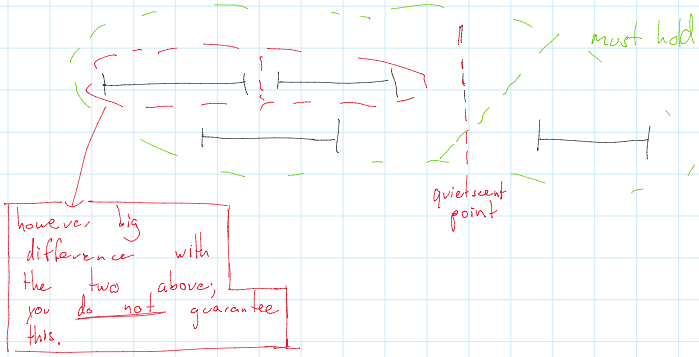
Definition 19.9 (Sequential Consistency). *An execution E is called* ***sequentially consistent****, if there is a sequence of operations S such that:*

- *S is correct and semantically equivalent to E.*
- *Whenever $f < g$ for two operations f, g* ***on the same node*** *in E, then also $f < g$ in S.*



Definition 19.11 (Quiescent Consistency). *An execution E is called* ***quiescently consistent****, if there is a sequence of operations S such that:*

- *S is correct and semantically equivalent to E.*
- *Let t be some quiescent point, i.e., for all operations f we have $f_\uparrow < t$ or $f_* > t$. Then for every t and every pair of operations g, h with $g_\uparrow < t$ and $h_* > t$ we also have $g < h$ in S.*



Lemma 19.12. *Every linearizable execution is also quiescently consistent, i.e., linearizability $\implies$ quiescent consistency.*

Proof. Let E be the original execution and S be the semantically equivalent sequential execution. Let t be a quiescent point and consider two operations g, h with $g_\uparrow < t < h_*$. Then we have $g < h$ in S. This is also guaranteed by linearizability since $g_\uparrow < t < h_*$ implies $g < h$. □

follows immediately from image above

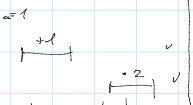
↳ more formal proof, but same idea

Lemma 19.13. *Sequentially consistent and quiescent consistency do not imply one another.*

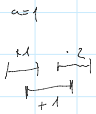
Proof. There are executions that are sequentially consistent but not quiescently consistent. An object initially has value 2. We apply two operations to this object: *inc* (increment the object by 1) and *double* (multiply the object by 2). Assume that $inc < double$, but *inc* and *double* are executed on different nodes. Then a result of 5 (first double, then inc) is sequentially consistent but not quiescently consistent.

There are executions that are quiescently consistent but not sequentially consistent. An object initially has value 2. Assume to have three operations on two nodes u and v. Node u calls first *inc* then *double*, node v calls *inc* once with $inc^u_* < inc^u_\uparrow < double^u_* < inc^v_\uparrow$. Since there is no quiescent point, quiescent consistency is okay with a sequential execution that doubles first, resulting in $((2 \cdot 2) + 1) + 1 = 6$. The sequential execution demands that $inc^u < double^u$, hence the result should be strictly larger than 6 (either 7 or 8). □

also pretty clear looking the above.



↳ sequential consistent any order; quiescent +1 before ·2.



↓

Quiescent might easily be consistency

a=4 however sequential

a=5

a=6

Given such definitions it is now possible to question about the possible properties of distributed systems.

↳ i.e. we should ask ourself what type of logical time makes sense for our application and how to implement it.

Definition 19.14. *A system or an implementation is called* ***linearizable*** *if it ensures that every possible execution is linearizable. Analogous definitions exist for sequential and quiescent consistency.*

Remarks:

- In the introductory social media example, a linearizable implementation would have to make sure that the comment is immediately visible on any device, as the *read* operation starts after the *write* operation

Remarks:

- In the introductory social media example, a linearizable implementation would have to make sure that the comment is immediately visible on any device, as the *read* operation starts after the *write* operation finishes. If the system is only sequentially consistent, the comment does not need to be immediately visible on every device.

Hence you see that for instance in social media it does not make sense to have linearizability as that would be awefully expensive.

Definition 19.15 (restricted execution)**.** *Let E be an execution involving operations on multiple objects. For some object o we let the* ***restricted execution*** *$E|o$ be the execution E filtered to only contain operations involving object o.*

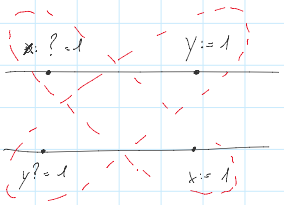
Definition 19.16. *A consistency model is called* ***composable*** *if the following holds: If for every object o the restricted execution $E|o$ is consistent, then also E is consistent.*

This is powerful. If the execution is composable you can implement, verify and execute concurrent objects independently.

Lemma 19.17. *Sequential consistency is not composable.*

Proof. We consider an execution E with two nodes u and v, which operate on two objects x and y initially set to 0. The operations are as follows: u_1 reads $x = 1$, u_2 writes $y := 1$, v_1 reads $y = 1$, v_2 writes $x := 1$ with $u_1 < u_2$ on node u and $v_1 < v_2$ on node v. It is clear that $E|x$ as well as $E|y$ are sequentially consistent as the write operations may be before the respective read operations. In contrast, execution E is *not* sequentially consistent: Neither u_1 nor v_1 can possibly be the initial operation in any correct semantically equivalent sequential execution S, as that would imply reading 1 when the variable is still 0. □

translated:



⇒ : $E|_x$ and $E|_y$

Notice that here all such restricted executions are consistent.

⇒ However, for that the entire execution is not sequential consistent.

↳ Neither u_1 nor v_1 can be the first correct execution in any correct semantical equivalent execution S.

Notice, however that:

Theorem 19.18. *Linearizability is composable.*

Proof. Let E be an execution composed of multiple restricted executions $E|x$. For any object x there is a sequential execution $S|x$ that is semantically consistent to $E|x$ and in which the operations are ordered according to wall-clock-linearization points. Let S be the sequential execution ordered according to all linearization points of all executions $E|x$. S is semantically equivalent to E as $S|x$ is semantically equivalent to $E|x$ for all objects x and two object-disjoint executions cannot interfere. Furthermore, if $f_\dagger < g_*$ in E, then also $f_\bullet < g_\bullet$ in

linearization points. Let S be the sequential execution ordered according to all linearization points of all executions $E|x$. S is semantically equivalent to E as $S|x$ is semantically equivalent to $E|x$ for all objects x and two object-disjoint executions cannot interfere. Furthermore, if $f_\dagger < g_*$ in E, then also $f_\bullet < g_\bullet$ in E and therefore also $f < g$ in S. □

↳ Follows immediately if you carefully think about it.

Such that when having a linearizable system the nice composability feature results.

We turn now to the topic of logical clocks and on how to collect temporal dependencies such as

$$g < h$$

in distributed systems.

Logical Clocks

This as mentioned try to capture so called happen-before-relations $g<h$.

Definition 19.19. *Let S_u be a sequence of operations on some node u and define "$\to$" to be the* ***happened-before relation*** *on $E := S_1 \cup \cdots \cup S_n$ that satisfies the following three conditions:*

1. *If a local operation f occurs before operation g on the same node ($f < g$), then $f \to g$.*
2. *If f is a send operation of one node, and g is the corresponding receive operation of another node, then $f \to g$.*
3. *If f, g, h are operations such that $f \to g$ and $g \to h$ then also $f \to h$.*

↳ So happen-before-relations respect so to say logical temporal conditions.

If for two distinct operations f, g neither $f \to g$ nor $g \to f$, then we also say f and g are *independent* and write $f \sim g$. Sequential computations are characterized by $\to$ being a total order, whereas the computation is entirely concurrent if no operations f, g with $f \to g$ exist.

Definition 19.20 (Happened-before consistency)**.** *An execution E is called* ***happened-before consistent****, if there is a sequence of operations S such that:*

- *S is correct and semantically equivalent to E.*
- *Whenever $f \to g$ for two operations f, g in E, then also $f < g$ in S.*

} follows exactly from definition.

Lemma 19.21. *Happened-before consistency = sequential consistency.*

Proof. Both consistency models execute all operations of a single node in the sequential order. In addition, happened-before consistency also respects messages between nodes. However, messages are also ordered by sequential consistency because of semantic equivalence (a receive cannot be before the corresponding send). Finally, even though transitivity is defined more formally in happened-before consistency, also sequential consistency respects transitivity.

In addition, sequential consistency orders two operations o_u, o_v on two dif-

between nodes. However, messages are also ordered by sequential consistency because of semantic equivalence (a receive cannot be before the corresponding send). Finally, even though transitivity is defined more formally in happened-before consistency, also sequential consistency respects transitivity.

In addition, sequential consistency orders two operations o_u, o_v on two different nodes u, v if o_v can see a state change caused by o_u. Such a state change does not happen out of the blue, in practice some messages between u and v (maybe via "shared blackboard" or some other form of communication) will be involved to communicate the state change. □

We will now inspect how to construct logical clocks for which the happen-before relation is satisfied.

Definition 19.22 (Logical clock). *A logical clock is a family of functions c_v that map every operation $f \in E$ on node u to some logical time $c_u(f)$ such that the happened-before relation $\rightarrow$ is respected, i.e., for two operations g on node u and h on node v*

$$g \rightarrow h \implies c_u(g) < c_v(h).$$

Definition 19.23. *If it additionally holds that $c_u(g) < c_v(h) \implies g \rightarrow h$, then the clock is called a* ***strong logical clock.***

Remarks:

- In algorithms we write c_u for the current logical time of node u.

Algorithm 19.24 Lamport clock

1: {Code for node u}
2: Initialize $c_u := 0$.
3: Upon local operation: Increment current local time $c_u := c_u + 1$.
4: Upon send operation: Increment $c_u := c_u + 1$ and include c_u as T in message.
5: Upon receive operation: Extract T from message and update $c_u := \max\{c_u, T\} + 1$.

Theorem 19.25. *Lamport clocks are logical clocks.*

Proof. If for two operations f, g it holds that $f \rightarrow g$, then according to the definition three cases are possible.

1. If $f < g$ on the same node u, then $c_u(f) < c_u(g)$.
2. Let g be a receive operation on node v corresponding to some send operation f on another node u. We have $c_v(g) \geq T + 1 = c_u(f) + 1 > c_u(f)$.
3. Transitivity follows with $f \rightarrow g$ and $g \rightarrow h \Rightarrow g \rightarrow h$, and the first two cases.

□

Remarks:

- Lamport logical clocks are not strong logical clocks, which means we cannot completely reconstruct $\rightarrow$ from the family of clocks c_u.
- To achieve a strong logical clock, nodes also have to gather information about other clocks in the system, i.e., node u needs to have a idea of node v's clock, for every u, v. This is what *vector clocks* in Algorithm 19.26 do: Each node u stores its knowledge about other node's logical clocks in an n-dimensional vector c_u.

to see this: ① ② ③

Here there is a clear happen before relation ① → ②, however ③ has no idea of the logical clock of node v so that its logical clock can be $c_{③} < c_{②}$.

↳ It would however be wrong to infer from it $c_{③} < c_{②} \Rightarrow ③ \rightarrow ①$

⇒ In order to ensure strong logical clocks use vector clocks:

Algorithm 19.26 Vector clocks

1: {Code for node u}
2: Initialize $c_u[v] := 0$ for all other nodes v.
3: Upon local operation: Increment current local time $c_u[u] := c_u[u] + 1$.
4: Upon send operation: Increment $c_u[u] := c_u[u] + 1$ and include the whole vector c_u as d in message.
5: Upon receive operation: Extract vector d from message and update $c_u[v] := \max\{d[v], c_u[v]\}$ for all entries v. Increment $c_u[u] := c_u[u] + 1$.

Theorem 19.27. *Define $c_u < c_v$ if and only if $c_u[w] \leq c_v[w]$ for all entries w, and $c_u[x] < c_v[x]$ for at least one entry x. Then the vector clocks are strong logical clocks.*

Proof. We are given two operations f, g, with operation f on node u, and operation g on node v, possibly $v = u$.

If we have $f \rightarrow g$, then there must be a happened-before-path of operations and messages from f to g. According to Algorithm 19.26, $c_v(g)$ must include at least the values of the vector $c_u(f)$, and the value $c_v(g)[u] > c_u(f)[u]$.

This means that in the above we would not compare the local clocks of two different nodes owing to wrong conclusions but we would rather know the state of the other

local clocks of two different nodes running to wrong conclusions but we could rather know the state of the other node by looking directly at its entry.

Proof. We are given two operations f, g, with operation f on node u, and operation g on node v, possibly $v = u$.

If we have $f \to g$, then there must be a happened-before path of operations and messages from f to g. According to Algorithm 19.26, $c_v(g)$ must include at least the values of the vector $c_u(f)$, and the value $c_v(g)[v] > c_u(f)[v]$.

If we do not have $f \to g$, then $c_v(g)[u]$ cannot know about $c_u(f)[u]$, and hence $c_v(g)[u] < c_u(f)[u]$, since $c_u(f)[u]$ was incremented when executing f on node u.

□

Next we go into the topic of Consistent Snapshots and we will question about algorithms to obtain a picture of a distributed system such that we know its current state.

Consistent Snapshot

Definition 19.28 (cut). *A **cut** is some prefix of a distributed execution. More precisely, if a cut contains an operation f on some node u, then it also contains all the preceding operations of u. The set of last operations on every node included in the cut is called the **frontier** of the cut.*

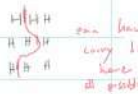
Definition 19.29 (consistent snapshot). *A cut C is a **consistent snapshot**, if for every operation g in C with $f \to g$, C also contains f.*

Remarks:

- In a consistent snapshot it is forbidden to see an effect without its cause.

Notice how that the number of consistent snapshots gives information on the degree of concurrency of a system.

- One extreme is a sequential computation, where stopping one node halts the whole system. Let q_u be the number of operations on node $u \in \{1, \ldots, n\}$. Then the number of consistent snapshots (including the empty cut) in the sequential case is $\mu_s := 1 + q_1 + q_2 + \cdots + q_n$.



- One the other hand, in an entirely concurrent computation the nodes are not dependent on one another and therefore stopping one node does not impact others. The number of consistent snapshots in this case is $\mu_c := (1 + q_1) \cdot (1 + q_2) \cdots (1 + q_n)$.

Definition 19.30 (measure of concurrency). *The concurrency measure of an execution $E = (S_1, \ldots, S_n)$ is defined as the ratio*

$$m(E) := \frac{\mu - \mu_s}{\mu_c - \mu_s}, \quad \in [0,1]$$

where μ denotes the number of consistent snapshot of E.

It is clear that in order to obtain the degree of concurrency we have to compute the number of consistent snapshots in a system.

We will next see an algorithm to collect a consistent snapshot.

Algorithm 19.32 Distributed Snapshot Algorithm

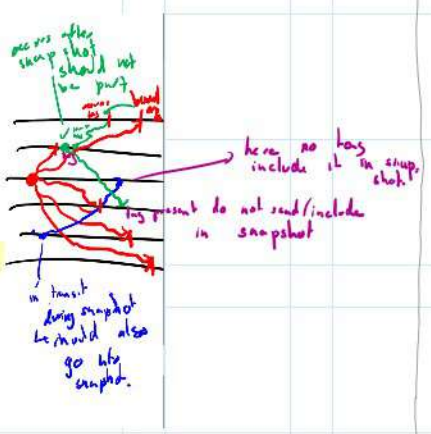
1: Inititntor: Save local state, send a snap message to all other nodes and collect incoming status and messages of all other nodes.
2: All other nodes:
3: Upon receiving a snap message for the first time: send own state (before message) to the initiator and propagate snap by adding snap tag to future messages.
4: If afterwards receiving a message m without snap tag: Forward m to the initiator.

Theorem 19.33. *Algorithm 19.32 collects a consistent snapshot.*

Proof. Let C be the cut induced by the frontier of all states and messages forwarded to the initiator. For every node u, let t_u be the time when u gets the first snap message m (either by the initiator, or as a message tag). Then C contains all of u's operations before t_u, and none after t_u (also not the message m which arrives together with the tag at t_v).

Assume for the sake of contradiction we have operations f, g on nodes u, v respectively, with $f \to g$, $f \notin C$ and $g \subset C$, hence $t_u \leq f$ and $g < t_v$. If $u = v$ we have $t_u \leq f < g < t_u = t_v$, which is a contradiction. On the other hand, if $u \neq v$: Since $t_u \leq f$ we know that all following send operations must have included the snap tag. Because of $f \to g$ we know there is a path of messages between f and g, all including the snap tag. So the snap tag must have been received by node v before or with operation g, hence $t_v \leq g$, which is a contradiction to $t_v > g$. □

So basically graphically:



Note now that all of these ideas can be easily translated into industrial microservices architectures, where you have similar ideas and open source software implementing distributed tracing:

Definition 19.34 (Microservice Architecture). *A **microservice architecture** refers to a system composed of loosely coupled services. These services communicate by various protocols and are either decentrally coordinated (also known as "choreography") or centrally ("orchestration").*

same idea of object

Definition 19.35 (Span). *A **span** s is a named and timed operation representing a contiguous sequence of operations on one node. A span s has a start time $s_\vdash$ and finish time $s_\dashv$.*

same ideas of relations

Definition 19.36 (Span Reference). *A span may causally depend on other spans. The two possible relations are **ChildOf** and **FollowsFrom** references. In a ChildOf reference, the parent span depends on the result of the child (the parents asks the child and the child answers), and therefore parent and child span must overlap. In FollowsFrom references parent spans do not depend in any way on the result of their child spans (the parent just invokes the child).*

ideas of relations

spans. The two possible relations are ***ChildOf*** *and* ***FollowsFrom*** *references. In a ChildOf reference, the parent span depends on the result of the child (the parents asks the child and the child answers), and therefore parent and child span must overlap. In FollowsFrom references parent spans do not depend in any way on the result of their child spans (the parent just invokes the child).*

Definition 19.37 (Trace)**.** *A* ***trace*** *is a series-parallel directed acyclic graph representing the hierarchy of spans that are executed to serve some request. Edges are annotated by the type of the reference, either ChildOf or FollowsFrom.*

so fairly basic algorithm

- Algorithm 19.38 shows what is needed if you want to trace requests to your system.

Algorithm 19.38 Inter-Service Tracing

1: Upon requesting another service: Inject information of current trace and span (IDs or timing information) into the request header.
2: Upon receiving request from another service: Extract trace and span information from the request header and create new span as child span.

Physical Time
23 November 2020 17:39

Chapter 20 - Time and GPS

This Chapter contrasts with how time is physically measured.

↳ The idea is that if we manage to create more precise time structures and synchronization techniques we might be able to move from asynchronous to synchronous communication.

We will hence go in this chapter into state of the art time measurement and synchronization technologies.

Definition 20.2 (Wall-Clock Time). *The* ***wall-clock time*** t^* *is the true time (a perfectly accurate clock would show).*

Definition 20.2 (Wall-Clock Time). *The **wall-clock time** t^* is the true time (a perfectly accurate clock would show).*

Definition 20.3 (Clock). *A **clock** is a device which tracks and indicates time.*

Definition 20.4 (Clock Error). *The **clock error** or clock skew is the difference between two clocks, e.g., $t-t^*$ or $t-t'$. In practice the clock error is often modeled as $t = (1+\delta)t^* + \xi(t^*)$.*

systematic error → $(1+\delta)t^*$

idiosyncratic error → $\xi(t^*)$

⇒ same as finance

Definition 20.5 (Drift). *The **drift** δ is the predictable clock error.*

Stable clock sources, which offer a low drift, are generally preferred, but also more expensive, larger and more power hungry, which is why many consumer products feature inaccurate clocks.

Definition 20.6 (Parts Per Million). *Clock drift is indicated in **parts per million (ppm)**. One ppm corresponds to a time error growth of one microsecond per second.*

↳ micro - 10^{-6} ⇒ so 1 ppm 1 time unit error every million units.

Definition 20.1 (Second). *A **second** is the time that passes during 9,192,631,770 oscillation cycles of a caesium-133 atom.*

Usual time drifts in comparison to above second definition.

- In PCs, the so-called *real-time clock* normally is a crystal oscillator with a maximum drift between 5 and 100 ppm.
- Applications in signal processing, for instance GPS, need more accurate clocks. Common drift values are 0.5 to 2 ppm.

Definition 20.7 (Jitter). *The **jitter** ξ is the unpredictable, random noise of the clock error.*

↳ idiosyncratic error, captures error not measured by the drift.

Given such definitions it is possible to explore some basics time synchronization mechanism.

Definition 20.9 (Clock Synchronization). ***Clock synchronization** is the process of matching multiple clocks (nodes) to have a common time.*

- A trade-off exists between synchronization accuracy, convergence time, and cost.
- Different clock synchronization variants may tolerate crashing, erroneous or byzantine nodes.

Algorithm 20.10 Network Time Protocol NTP

1: Two nodes, client u and server v

2: **while** true **do**
3: Node u sends request to v at time t_u
4: Node v receives request at time t_v
5: Node v processes the request and replies at time t'_v
6: Node u receives the response at time t'_u

7: Propagation delay $\delta = \frac{(t'_u - t_u) - (t'_v - t_v)}{2}$ (assumption: symmetric)
8: Clock skew $\theta = \frac{(t_v - (t_u + \delta)) - (t'_u - (t'_v + \delta))}{2} = \frac{(t_v - t_u) + (t'_v - t'_u)}{2}$
9: Node u adjusts clock by $+\theta$
10: Sleep before next synchronization

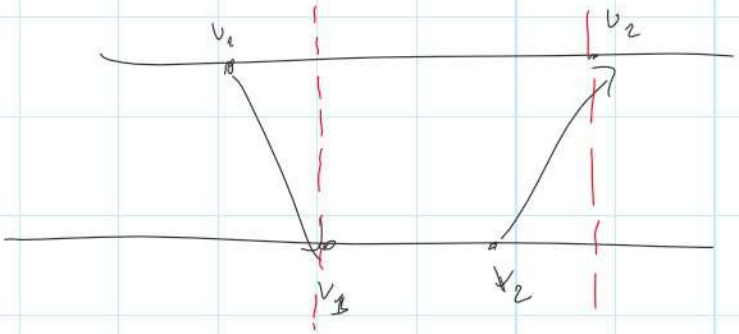
8: Clock skew $\theta = \frac{(t_v-(t_u+\delta))-(t_u-(t_v+\delta))}{2} = \frac{(t_v-t_u)+(t_v-t_u)}{2}$

9: Node u adjusts clock by $+\theta$

10: Sleep before next synchronization

11: **end while**

↳ Fairly simple mechanism, just a lot of notation that makes it difficult.



propagation time $\frac{(v_1 - v_1) + (v_2 - v_2)}{2}$

drift $\theta = \frac{v_1 - (v_1 + \delta) - (v_2 - (v_2 + \delta))}{2}$

:-: = if both equal after correction; no drift.

↳ Notice above you take the time of v as being good time and you adjust time of v to it; i.e. compute the drift to it.

This is why the clock of v

- The most accurate NTP servers derive their time from atomic clocks, synchronized to UTC. To reduce those server's load, a hierarchy of

the clock of ↓ should be very precise.

- The most accurate NTP servers derive their time from atomic clocks, synchronized to UTC. To reduce those server's load, a hierarchy of NTP servers is available in a forest (multiple trees) structure.
- The regular synchronization of NTP limits the maximum error despite unpredictable clock errors. Synchronizing clocks just once is only sufficient for a short time period.

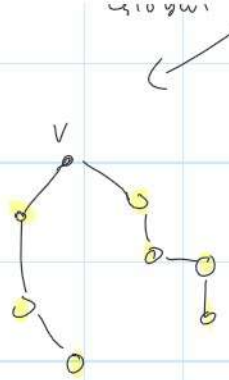
Definition 20.11 (PTP). *The **Precision Time Protocol (PTP)** is a clock synchronization protocol similar to NTP, but which uses **medium access control (MAC)** layer timestamps.*

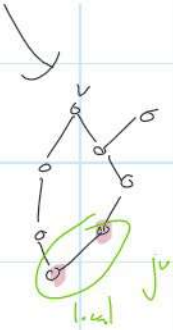
↳ same as Network Time Synchronization but you set the timestamp as low as possible in the Network Architecture (level 2) so that the time to go down and up the architecture does not influence the measured time.

Notice that there are two types of nodes synchronization. Local time synchronization and global time synchronization.

Definition 20.12 (Global Synchronization). ***Global** synchronization establishes a common time between **any** two nodes in the system.*

Global vs. local





= all v in the above.

why?

because you have central servers.

Tree structure and all of the nodes refer to these.

- NTP and PTP are both examples of clock synchronization algorithms that optimize for global synchronization.

- However, two nodes that constantly communicate may receive their timestamps through different paths of the NTP forest, and hence they may accumulate different errors. Because of the clock skew, a message sent by node u might arrive at node v with a timestamp in the future.

Algorithm 20.13 Local Time Synchronization

```
1: while true do
2:     Exchange current time with neighbors
3:     Adapt time to neighbors, e.g., to average or median
4:     Sleep before next synchronization
5: end while
```

Local synchronization is the method of choice to establish *time-division multiple access (TDMA)* and coordination of wake-up and sleeping times in wireless networks. Only close-by nodes matter as far-away nodes will not interfere with their transmissions.

While global synchronization algorithm such as NTP usually synchronize to an external time standard, local algorithms often just synchronize among themselves, i.e., the notion of time does not reflect any time standards.

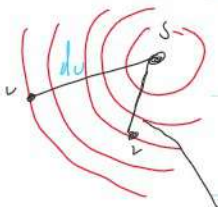
Algorithm 20.14 Wireless Clock Synchronization with Known Delays

1: Given: transmitter s, receivers u, v, with known transmission delays d_u, d_v from transmitter s, respectively.

2: s sends signal at time t_s
3: u receives signal at time t_u
4: v receives signal at time t_v

5: $\Delta_u = t_u - (t_s + d_u)$
6: $\Delta_v = t_v - (t_s + d_v)$

7: Clock skew between u and v: $\theta = \Delta_v - \Delta_u = t_v - d_v + d_u - t_u$



wireless propagates as a wave.

known as you know distance and when signal should arrive $:= d_v$

We will introduce now the time standards we use today, and we will discuss in the next section how these are propagated.

Definition 20.15 (TAI). *The **International Atomic Time (TAI)** is a time standard derived from over 400 atomic clocks distributed worldwide.*

- Using a weighted average of all involved clocks, TAI is an order of magnitude more stable than the best clock.
- The involved clocks are synchronized using simultaneous observations of GPS or geostationary satellite transmissions using Algorithm 20.14.
- While a single satellite measurement has a time uncertainty on the order of nanoseconds, averaging over a month improves the accuracy by several orders of magnitude.

- Using a weighted average of all involved clocks, TAI is an order of magnitude more stable than the best clock.
- The involved clocks are synchronized using simultaneous observations of GPS or geostationary satellite transmissions using Algorithm 20.14.
- While a single satellite measurement has a time uncertainty on the order of nanoseconds, averaging over a month improves the accuracy by several orders of magnitude.

Definition 20.16 (Leap Second). *A leap second is an extra second added to a minute to make it irregularly 61 instead of 60 seconds long.*

- Time standards use leap seconds to compensate for the slowing of the Earth's rotation. In theory, also negative leap seconds can be used to make some minutes only 59 seconds long. But so far, this was never necessary.
- For easy implementation, not all time standards use leap seconds, for instance TAI and GPS time do not.

so idea keep it more less consistent with earth cycle.

Definition 20.17 (UTC). *The **Coordinated Universal Time (UTC)** is a time standard based on TAI with leap seconds added at irregular intervals to keep it close to mean solar time at 0° longitude.*

Definition 20.18 (Time Zone). *A **time zone** is a geographical region in which the same time offset from UTC is officially used.*

Remarks:

- Time zones serve to roughly synchronize noon with the sun reaching the day's highest apparent elevation angle.

Propagation/measurement of time that you can use in your distributed systems.

Definition 20.19 (Atomic Clock). *An **atomic clock** is a clock which keeps time by counting oscillations of atoms.*

Remarks:

- Atomic clocks are the most accurate clocks known. They can have a drift of only about one second in 150 million years, about 2e-10 ppm!
- Many atomic clocks are based on caesium atoms, which led to the current definition of a second. Others use hydrogen-1 or rubidium-87.
- In the future, atoms with higher frequency oscillations could yield

- Many atomic clocks are based on caesium atoms, which led to the current definition of a second. Others use hydrogen-1 or rubidium-87.
- In the future, atoms with higher frequency oscillations could yield even more accurate clocks.
- Atomic clocks are getting smaller and more energy efficient. Chip-scale atomic clocks (CSAC) are currently being produced for space applications and may eventually find their way into consumer electronics.

Definition 20.20 (System Clock). *The **system clock** in a computer is an oscillator used to synchronize all components on the motherboard.*

- Usually, a quartz crystal oscillator with a frequency of some tens to hundreds MHz is used.
- Therefore, the system clock can achieve a precision of some ns!
- The *CPU clock* is usually a multiple of the system clock, generated from the system clock through a clock multiplier.
- To guarantee nominal operation of the computer, the system clock must have low jitter. Otherwise, some components might not get enough time to complete their operation before the next (early) clock pulse arrives.
- Drift however is not critical for system stability.
- If a computer is shut down, the system clock is not running; it is reinitialized when starting the computer.

Definition 20.21 (RTC). *The **real-time clock (RTC)** in a computer is a battery backed oscillator which is running even if the computer is shut down or unplugged.*

Remarks:

- The RTC is read at system startup to initialize the system clock.
- This keeps the computer's time close to UTC even when the time cannot be synchronized over a network.
- RTCs are relatively inaccurate, with a common maximum drift of 5, 20 or even 100 ppm, depending on quality and temperature.

Definition 20.22 (Radio Time Signal). *A **Radio Time Signal** is a time code transmitted via radio waves by a time signal station, referring to a time in a given standard such as UTC.*

Remarks:

- Time signal stations use atomic clocks to send as accurate time codes as possible.
- Radio-controlled clocks are an example application of radio signal time

- Time signal stations use atomic clocks to send as accurate time codes as possible.
- Radio-controlled clocks are an example application of radio signal time synchronization.
- In Europe, most radio-controlled clocks use the signal transmitted by the *DCF77* station near Frankfurt, Germany.
- Radio time signals can be received much farther than the horizon of the transmitter due to signal reflections at the ionosphere. DCF77 for instance has an official range of 2,000 km.

↳ same concept seen when studying sailing, there especially strong as no possible obstruction for the signal in normal atmospheric conditions.

Definition 20.23 (Power Line Clock). *A **power line clock** measures the oscillations from electric AC power lines, e.g. 50 Hz.*

idea use this alternate current oscillations to measure time.

Remarks:

- Clocks in kitchen ovens are usually driven by power line oscillations.
- AC power line oscillations drift about 10 ppm, which is remarkably stable.
- The magnetic field radiating from power lines is strong enough that power line clocks can work wirelessly.
- Power line clocks can be synchronized by matching the observed noisy power line oscillation patterns.
- Power line clocks operate with as little as a few ten µW.

these are extremly stable as it energy provides mass with them the entire circuit might easily go down.

Definition 20.24 (Sunlight Time Synchronization). ***Sunlight time synchronization** is a method of reconstructing global timestamps by correlating annual solar patterns from light sensors' length of day measurements.*

↳ Very inaccurate.

→ Very inaccurate.

Today, the most commonly used technique to measure time is GPS, this is the basis of TAI tracking as discussed above and poses today the basis for time synchronization for most of our distributed systems.

Definition 20.25 (Global Positioning System). *The **Global Positioning System (GPS)** is a **Global Navigation Satellite System (GNSS)**, consisting of at least 24 satellites orbiting around the Earth, each continuously transmitting its position and time code.*

- Positioning is done in space and *time*!
- GPS provides position and time information to receivers anywhere on Earth where at least four satellite signals can be received.
- Line of sight (LOS) between satellite and receiver is advantageous. GPS works poorly indoors, or with reflections.

Algorithm 20.26 GPS Satellite

1: Given: Each satellite has a unique 1023 bit (± 1, see below) *PRN* sequence, plus some current navigation data D (also ± 1).
2: The code below is a bit simplified, concentrating on the digital aspects, ignoring that the data is sent on a carrier frequency of 1575.42 MHz.

3: **while** true **do**
4: **for all** bits $D_i \in D$ **do**
5: **for** $j = 0 \dots 19$ **do** } 20 times the information
6: **for** $k = 0 \dots 1022$ **do** {this loop takes exactly 1 ms}
7: Send bit $PRN_k \cdot D_i$
8: **end for**

```
      for k = 0...1022 do {this loop takes exactly 1 ms}
        Send bit PRN_k · D_i
      end for
    end for
  end for
end while
```

Definition 20.27 (PRN). ***Pseudo-Random Noise (PRN)*** *sequences are pseudo-random bit strings. Each GPS satellite uses a unique PRN sequence with a length of 1023 bits for its signal transmissions.*

Remarks:

- The GPS PRN sequences are so-called *Gold codes*, which have low cross-correlation with each other.
- To simplify our math (abstract from modulation), each PRN bit is either 1 or -1.

} means you should be able to distinguish them easily; key concept.

Definition 20.28 (Navigation Data). ***Navigation Data*** *is the data transmitted from satellites, which includes orbit parameters to determine satellite positions, timestamps of signal transmission, atmospheric delay estimations and status information of the satellites and GPS as a whole, such as the accuracy and validity of the data.*

- As seen in Algorithm 20.26 each bit is repeated 20 times for better robustness. Thus, the navigation data rate is only 50 bit/s.

1 bit every 20 ms so that

20 ms · 50 = 1 sec

50 bit/s

- Due to this limited data rate, timestamps are sent every 6 seconds, satellite orbit parameters (function of the satellite position over time) only every 30 seconds. As a result, the latency of a first position estimate after turning on a receiver, which is called *time-to-first-fix (TTFF)*, can be high.

↳ i.e. in case of bad luck (just transmitted) you might have to wait 30 sec.

Definition 20.29 (Circular Cross-Correlation). *The* ***circular cross-correlation*** *is a similarity measure between two vectors of length N,* ***circularly*** *shifted by a given displacement d:*

$$xcorr(\boldsymbol{a}, \boldsymbol{b}, d) = \sum_{i=0}^{N-1} a_i \cdot b_{i+(d \bmod N)}$$

↳ So recall that $PNR_k \in \{1, -1\}$.

and that by construction

$PNR_{k_i} \cdot PNR_{k_i} = 1$ one time

$PNR_{k,i} \cdot PNR_{k,j} = 1$ one time and -1 another time for $PNR_{k,i+\ell} \cdot PNR_{k,j+\ell}$ such that the sum is 0 and these are low cross-correlated.

↳ The idea is then when you receive a signal containing the PNR to check which PNR this signal is most similar to. If then follows that due to the construction of the different PNRs you should find high cross-correlation for just 1 of these.

Remarks:

have to do this job for all PNR.

- The two vectors are most similar at the displacement d where the sum (cross-correlation value) is maximum.
- The vector of cross-correlation values with all N displacements can efficiently be computed using a fast Fourier transform (FFT) in $\mathcal{O}(N \log N)$ instead of $\mathcal{O}(N^2)$ time.

↓
have to look at multiple displacement as you do not know when the signal starts.

This leads us to the acquisition

This leads us to the acquisition algorithm through which it is possible to understand the signal arrival time for each satellite.

Algorithm 20.30 Acquisition

1: Received 1 ms signal $\boldsymbol{s}$ with sampling rate $r \cdot 1,023$ kHz
2: Possible Doppler shifts F, e.g. {-10 kHz, -9.8 kHz, ..., +10 kHz}
3: Tensor $A = 0$: Satellite × carrier frequency × time
4: **for all** satellites i **do**
5: $PRN_i' = PRN_i$ stretched with ratio $\tilde{r}$ (atmospheric condition, obtained from navigation data every 30 sec.)
6: **for all** Doppler shifts $f \in F$ **do** (have to go over all possible doppler shifts)
7: Build modulated PRN_i'' with PRN_i' and Doppler frequency f
8: **for all** delays $d \in \{0, 1, \ldots, 1,023 \cdot r - 1\}$ **do**
9: $A_i(f, d) = |xcorr(\boldsymbol{s}, \boldsymbol{PRN_i''}, d)|$
10: **end for**
11: **end for**
12: Select d^* that maximizes $\max_d \max_f A_i(f, d)$
13: Signal arrival time $\tau_i = d^*/(r \cdot 1,023 \text{ kHz})$
14: **end for**

- Multiple milliseconds of acquisition can be summed up to average out noise and therefore improve the arrival time detection probability.

Definition 20.31 (Acquisition). ***Acquisition*** *is the process in a GPS receiver that finds the visible satellite signals and detects the delays of the PRN sequences and the Doppler shifts of the signals.*

Note that the nested loop and search over all possible delays

search over all possible delays and doppler effects makes acquisition computationally expensive.

Algorithm 20.32 Classic GPS Receiver

1: h: Unknown receiver *handset* position
2: θ: Unknown handset time offset to GPS system time 3 drift
3: r_i: measured signal arrival time in *handset time system*
4: c: signal propagation speed (GPS: speed of light)

known i, r_i obtained in acquisition.

5: Perform Acquisition (Algorithm 20.30)
6: Track signals and decode navigation data
7: **for all** satellites i **do**
8: Using navigation data, determine signal transmit time s_i and position p_i
9: Measured satellite transmission delay $d_i = r_i - s_i$
10: **end for**
11: Solve the following system of equations for h and θ:
12: $||p_i - h||/c = d_i - \theta$, for all i

+1

given

known

known

4 unknowns: - h → 3d position in space i.e. $\mathbb{R}^3$.

- θ → time offset $\in \mathbb{R}$

It is clear that we have 4 unknown and as soon as we get signal from 4 satellites we can easily solve the system of equations above.

- Since the equations are quadratic (distance), with as many observations as variables, the system of equations has two solutions in princi-

- Since the equations are quadratic (distance), with as many observations as variables, the system of equations has two solutions in principle. For GPS however, in practice one of the solutions is far from the Earth surface, so the correct solution can always be identified without a fifth satellite.

- More received signals help reducing the measurement noise and thus improving the accuracy.

- Since the positioning solution, (which is also called position fix,) includes the handset's time offset Δ, this establishes a global time for all handsets. Thus, GPS is useful for global time synchronization.

it simply says that when solving the above you get time offset ⇒ global time synch.

- For a handset with unknown position, GPS timing is more accurate than time synchronization with a single transmitter, like a time signal station (cf. Definition 20.22). With the latter, the unknown signal propagation delays cannot be accounted for.

Notice that if position known no such a big difference ⇒ think of wireless example previously shown.

Definition 20.33 (A-GPS). *An **Assisted GPS (A-GPS)** receiver fetches the satellite orbit parameters and other navigation data from the Internet, for instance via a cellular network.*

Definition 20.33 (A-GPS). *An **Assisted GPS (A-GPS)** receiver fetches the satellite orbit parameters and other navigation data from the Internet, for instance via a cellular network.*

Remarks:

- A-GPS reduces the data transmission time, and thus the TTFF, from a maximum of 30 seconds per satellite to a maximum of 6 seconds.

- Smartphones regularly use A-GPS. However, coarse positioning is usually done based on nearby Wi-Fi base stations only, which saves energy compared to GPS.

check energy usage in mountains if using GPS. should be terribly high.

- Another GPS improvement is *Differential GPS (DGPS)*: A receiver with a fixed location within a few kilometers of a mobile receiver compares the observed and actual satellite distances. This error is then subtracted at the mobile receiver. DGPS achieves accuracies in the order of 10 cm.

Definition 20.34 (Snapshot GPS Receiver). *A **snapshot receiver** is a GPS receiver that captures one or a few milliseconds of raw GPS signal for a position fix.*

mmm... not the best explanation here.

Remarks:

- Snapshot receivers aim at the remaining latency that results from the transmission of timestamps from the satellites every six seconds. § do not have it.
- Since time changes continuously, timestamps cannot be fetched together with the satellite orbit parameters that are valid for two hours.
- A snapshot receiver can determine the ranges to the satellites modulo 1 ms, which corresponds to 300 km. An approximate time and location of the receiver is used to resolve these ambiguities without a timestamp from the satellite signals themselves.

so instead of observing 20 times the signal just one.
↳ approximative.

Definition 20.35 (CTN). ***Coarse Time Navigation (CTN)** is a snapshot receiver positioning technique measuring sub-millisecond satellite ranges from correlation peaks, like conventional GPS receivers.*

Remarks:

- A CTN receiver determines the signal transmit times and satellite positions from its own approximate location by subtracting the signal propagation delay from the receive time. The receiver location and time is not exactly known, but since signals are transmitted exactly at whole milliseconds, rounding to the nearest whole millisecond gives the signal transmit time.

so here only difference.

so here you compute s_i and p_i and do not fetch it from navigation data.

↳ so you can well understand the coarse in the name once you understand the functioning.

Notice that with coarse time navigation:

- With only a few milliseconds of signal, noise cannot be averaged out well and may lead to wrong signal arrival time estimates. Such wrong measurements usually render the system of equations unsolvable, making positioning infeasible.

measurements usually render the system of equations unsolvable, making positioning infeasible.

Algorithm 20.36 Collective Detection Receiver

1: Given: A raw 1 ms GPS sample $\boldsymbol{s}$, a set H of location/time hypotheses
2: In addition, the receiver learned all navigation and atmospheric data

3: **for all** hypotheses $h \in H$ **do**
4: Vector $\boldsymbol{r} = 0$
5: Set V = satellites that should be visible with hypothesis h
6: **for all** satellites i in V **do**
7: $\boldsymbol{r} = \boldsymbol{r} + \boldsymbol{r_i}$, where $\boldsymbol{r_i}$ is expected signal of satellite i. The data of vector $\boldsymbol{r_i}$ incorporates all available information: distance and atmospheric delay between satellite and receiver, frequency shift because of Doppler shift due to satellite movement, current navigation data bit of satellite, etc.
8: **end for**
9: Probability $P_h = cxcorr(\boldsymbol{s}, \boldsymbol{r}, 0)$
10: **end for**
11: Solution: hypothesis $h \in H$ maximizing P_h

Definition 20.37 (Collective Detection)**.** ***Collective detection (CD)*** *is a maximum likelihood snapshot receiver localization method, which does not determine an arrival time for each satellite, but rather combine all the available information and take a decision only at the end of the computation.*

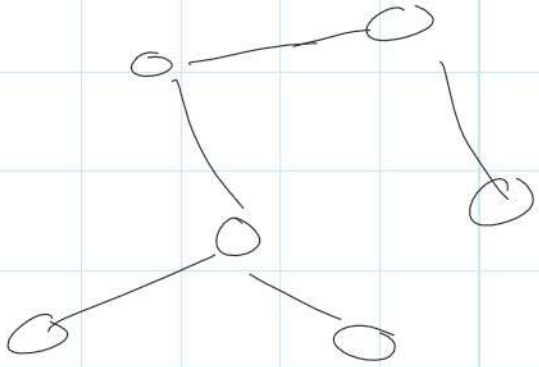
Remarks:

- CD can tolerate a few low quality satellite signals and is thus more robust than CTN.
- In essence, CD tests how well position hypotheses match the received signal. For large position and time uncertainties, the high number of hypotheses require a lot of computation power.
- CD can be sped up by a branch and bound approach, which reduces the computation per position fix to the order of one second even for uncertainties of 100 km and a minute.

This concludes our section on time measurement and propagation rules.

We will consider next the lower bounds for time synchronization

bounds for time synchronization in a network of nodes:



Lower bound on Clock Synchronization in Graph

Setting:

In the *clock synchronization* problem, we are given a network (graph) with n nodes. The goal for each node is to have a (logical) clock such that the clock values are well synchronized, and close to real time. Each node is equipped with a hardware (system) clock, that ticks more or less in real time, i.e., the time between two pulses is arbitrary between $[1-\epsilon, 1+\epsilon]$, for a constant $\epsilon \ll 1$. We assume that messages sent over the edges of the graph have a delivery time between $[0, 1]$. In other words, we have a bounded but variable drift on the hardware clocks and an arbitrary jitter in the delivery times. The goal is to design a message-passing algorithm that ensures that the logical clock skew of adjacent nodes is as small as possible at all times.

drift!

each hardware clock

Definition 20.38 (Local and Global Clock Skew)**.** *In a network of nodes, the* ***local clock skew*** *is the skew between neighboring nodes, while the* ***global clock skew*** *is the maximum skew between any two nodes.*

Remarks:

- Of interest is also the *average global clock skew*, that is the average

Remarks:

- Of interest is also the *average global clock skew*, that is the average skew between any pair of nodes.

Theorem 20.39. *The global clock skew (Definition 20.12) is $\Omega(D)$, where D is the diameter of the network graph.*

Proof. For a node u, let t_u be the logical time of u and let $(u \to v)$ denote a message sent from u to a node v. Let $t(m)$ be the time delay of a message m and let u and v be neighboring nodes. First consider a case where the message delays between u and v are $1/2$. Then, all the messages sent by u and v at time t according to the clock of the sender arrive at time $t + 1/2$ according to the clock of the receiver.

Then consider the following cases

- $t_u = t_v + 1/2$, $t(u \to v) = 1$, $t(v \to u) = 0$
- $t_u = t_v - 1/2$, $t(u \to v) = 0$, $t(v \to u) = 1$,

where the message delivery time is always fast for one node and slow for the other and the logical clocks are off by $1/2$. In both scenarios, the messages sent at time i according to the clock of the sender arrive at time $i + 1/2$ according to the logical clock of the receiver. Therefore, for nodes u and v, both cases with clock drift seem the same as the case with perfectly synchronized clocks. Furthermore, in a linked list of D nodes, the left- and rightmost nodes l, r cannot distinguish $t_l = t_r + D/2$ from $t_l = t_r - D/2$. □

↳ it follows $\Omega(D) = t_1$

$\Omega(g(n)) = f(n) \iff g(n) = o(f(n))$

where recall

$o(f(n))$ means $\lim_{n \to \infty} \frac{|f(n)|}{g(n)} = 0$

i.e.

$\Omega(g(n))$ means $\lim_{n \to \infty} \frac{|f(n)|}{g(n)} < \infty$

↳ so worst-case bounds.

- From Theorem 20.39, it directly follows that any reasonable clock synchronization algorithm must have a global skew of $\Omega(D)$.
- Many natural algorithms manage to achieve a global clock skew of $\mathcal{O}(D)$.
- As both message jitter and hardware clock drift are bounded by constants, it feels like we should be able to get a constant drift at least

- As both message jitter and hardware clock drift are bounded by constants, it feels like we should be able to get a constant drift at least between neighboring nodes.

Lemma 20.41. *The clock synchronization protocol of Algorithm 20.40 has a local skew of $\Omega(n)$.*

Algorithm 20.40 Local Clock Synchronization (at node v)

```
1: repeat
2:    send logical time t_v to all neighbors
3:    if Receive logical time t_u, where t_u > t_v, from any neighbor u then
4:       t_v = t_u
5:    end if
6: until done
```

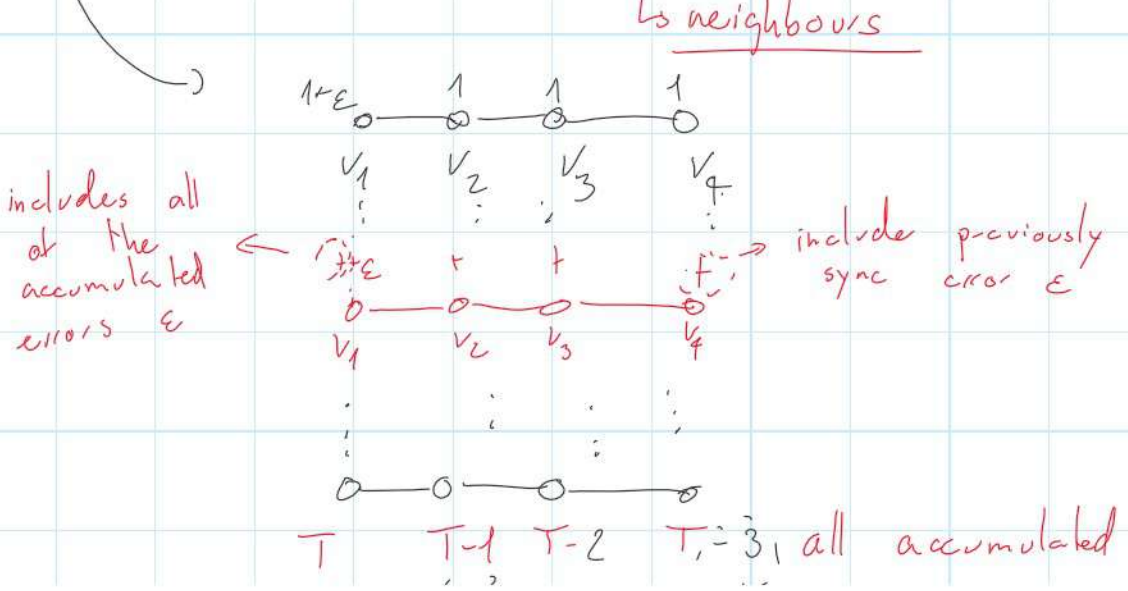
very simple and logic

Proof. Let the graph be a linked list of D nodes. We denote the nodes by $v_1, v_2, \ldots, v_D$ from left to right and the logical clock of node v_i by t_i. Apart from the left-most node v_1 all hardware clocks run with speed 1 (real time). Node v_1 runs at maximum speed, i.e. the time between two pulses is not 1 but $1-\epsilon$. Assume that initially all message delays are 1. After some time, node v_1 will start to speed up v_2, and after some more time v_2 will speed up v_3, and so on. At some point of time, we will have a clock skew of 1 between any two neighbors. In particular $t_1 = t_D + D - 1$.

Now we start playing around with the message delays. Let $t_1 = T$. First we set the delay between the v_1 and v_2 to 0. Now node v_2 immediately adjusts its logical clock to T. After this event (which is instantaneous in our model) we set the delay between v_2 and v_3 to 0, which results in v_3 setting its logical clock to T as well. We perform this successively to all pairs of nodes until v_{D-2} and v_{D-1}. Now node v_{D-1} sets its logical clock to T, which indicates that the difference between the logical clocks of v_{D-1} and v_D is $T - (T - (D-1)) = D - 1$. □

So above important here just construct to make it explicit

so at the end just two nodes have the delay above (neighbours)



T T-1 T-2 T_i = 3, all accumulated error = 1

exactly

So you see above that you have maximum error of n = number of linked nodes.

- The introduced examples may seem cooked-up, but examples like this exist in all networks, and for all algorithms. Indeed, it was shown that any natural clock synchronization algorithm must have a bad local skew. In particular, a protocol that averages between all neighbors (like Algorithm 20.13) is even worse than Algorithm 20.40. An averaging algorithm has a clock skew of $\Omega(D^2)$ in the linked list, at all times.

lower bound

- It was shown that the local clock skew is $\Theta(\log D)$, i.e., there is a protocol that achieves this bound, and there is a proof that no algorithm can be better than this bound!

↑ this is why Ω notation and not O-notation.

- Note that these are worst-case bounds. In practice, clock drift and message delays may not be the worst possible, typically the speed of hardware clocks changes at a comparatively slow pace and the message transmission times follow a benign probability distribution. If we assume this, better protocols do exist, in theory as well as in practice.

Chapter 21 - Quorum Systems

28 November 2020 16:03

This chapter investigates and questions the topic of consensus and byzantine agreement through quorum systems.

The basic idea is that despite we have looked at various algorithms to deal the above cases we had to rely on some majority voting schema (think for instance at Paxos).

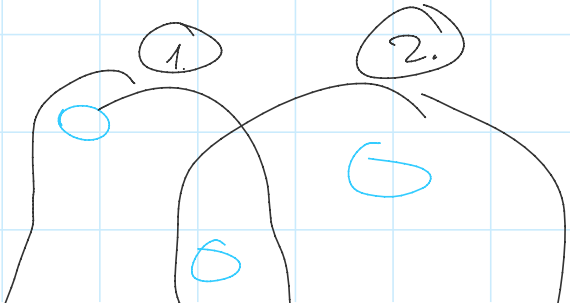
The question is then if some system exists for which a client would have to access less than a majority in order to reach consensus. (here focus do. the first section).

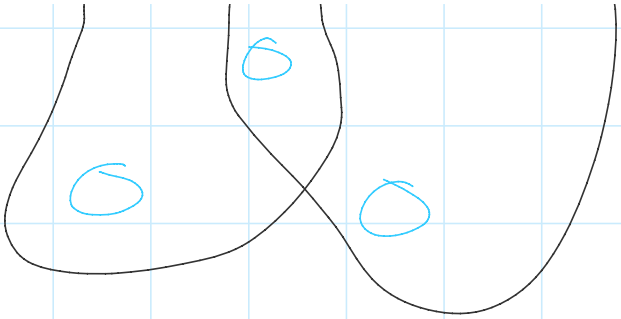
Let us take a step back: We used majorities because majority sets always overlap. But are majority sets the only sets that guarantee overlap? In this chapter we study the theory behind overlapping sets, known as quorum systems.

We will now consider quorums and quorums systems. There we will also have the overlapping property and we will see that working with such quorums systems we might be able to get better properties for our distributed system keeping the same effect.

↳ The question is therefore on efficiency in this chapter

Definition 21.1 (quorum, quorum system). *Let $V = \{v_1, \ldots, v_n\}$ be a set of nodes. A **quorum** $Q \subseteq V$ is a subset of these nodes. A **quorum system** $\mathcal{S} \subset 2^V$ is a set of quorums s.t. every two quorums intersect, i.e., $Q_1 \cap Q_2 \neq \emptyset$ for all $Q_1, Q_2 \in \mathcal{S}$.*





(1.) ∨ (2.) = quorums

(1.) ∧ (2.) = quorum system.

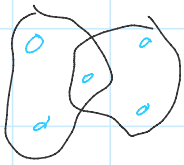
Notice moreover the following three important quorum systems:

(1.) • A quorum system $\mathcal{S}$ is called **minimal** if $\forall Q_1, Q_2 \in \mathcal{S} : Q_1 \not\subset Q_2$.

(2.) • The simplest quorum system imaginable consists of just one quorum which in turn just consists of one server. It is known as **Singleton**.

(3.) • In the **Majority** quorum system, every quorum has $\lfloor \frac{n}{2} \rfloor + 1$ nodes.

Such that

(1.)  minimal, but  not minimal.

(2.)  C. ||

2. ⓪ Singleton

3. This was in fact the quorum used so far.

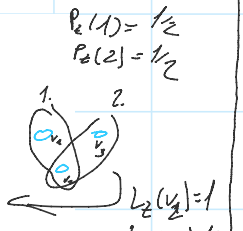
In the next sections we will consider the efficiency of different quorum systems. This is measured in terms of their load and work:

Definition 21.2 (access strategy). *An* ***access strategy*** *Z defines the probability $P_Z(Q)$ of accessing a quorum $Q \in \mathcal{S}$ s.t. $\sum_{Q \in \mathcal{S}} P_Z(Q) = 1$.*

How much each node is utilized in the quorum system.

Definition 21.3 (load).

- *The* ***load*** *of access strategy Z on a node v_i is $L_Z(v_i) = \sum_{Q \in \mathcal{S}; v_i \in Q} P_Z(Q)$.*
- *The* ***load*** *induced by access strategy Z on a quorum system $\mathcal{S}$ is the maximal load induced by Z on any node in $\mathcal{S}$, i.e., $L_Z(\mathcal{S}) = \max_{v_i \in \mathcal{S}} L_Z(v_i)$.*
- *The* ***load*** *of a quorum system $\mathcal{S}$ is $L(\mathcal{S}) = \min_Z L_Z(\mathcal{S})$.* ↳ maximal load of best possible strategy



↳ In this case 1. gives us that $\sum P_Z(Q) = 1$, as no way to reduce it with any strategy.

So that work represents the average number of accessed nodes by a client.
↳ the lower the more efficient

Definition 21.4 (work).

- *The* ***work*** *of a quorum $Q \in \mathcal{S}$ is the number of nodes in Q, $W(Q) = |Q|$.*
- *The* ***work*** *induced by access strategy Z on a quorum system $\mathcal{S}$ is the expected number of nodes accessed, i.e., $W_Z(\mathcal{S}) = \sum_{Q \in \mathcal{S}} P_Z(Q) \cdot W(Q)$.*
- *The* ***work*** *of a quorum system $\mathcal{S}$ is $W(\mathcal{S}) = \min_Z W_Z(\mathcal{S})$.*

Given the

Given the definitions above it is immediate to see that

good but at the expense of load.

Primary Copy vs. Majority		Singleton	Majority
How many nodes need to be accessed?	(Work)	**1**	$> n/2$
What is the load of the busiest node?	(Load)	1	$> \mathbf{1/2}$

bad

Notice that it is immediate to get lower bounds on the quorum system load given the quorum system work.

The idea is the following:

Theorem 21.6. *Let $\mathcal{S}$ be a quorum system. Then $L(\mathcal{S}) \geq 1/\sqrt{n}$ holds.*

Proof. Let $Q = \{v_1, \ldots, v_q\}$ be a quorum of minimal size in $\mathcal{S}$, with $|Q| = q$. Let Z be an access strategy for $\mathcal{S}$. Every other quorum in $\mathcal{S}$ intersects in at least one element with this quorum Q. Each time a quorum is accessed, at least one node in Q is accessed as well, yielding a lower bound of $L_Z(v_i) \geq 1/q$ for some $v_i \in Q$.

Furthermore, as Q is minimal, at least q nodes need to be accessed, yielding $W(\mathcal{S}) \geq q$. Thus, $L_Z(v_i) \geq q/n$ for some $v_i \in Q$, as each time q nodes are accessed, the load of the most accessed node is at least q/n.

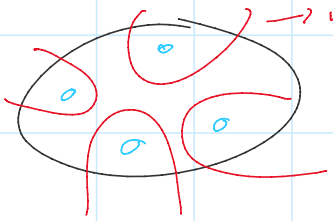
Combining both ideas leads to $L_Z(\mathcal{S}) \geq \max(1/q, q/n) \Rightarrow L_Z(\mathcal{S}) \geq 1/\sqrt{n}$. Thus, $L(\mathcal{S}) \geq 1/\sqrt{n}$, as Z can be *any* access strategy. □

Remarks:

- Can we achieve this load?

↳ ideal case (minimal)

↳ idea if quorum minimal



→ worst case match with other quorums for 1 node.

↳ so that load on any of these nodes when q quorums are accessed is in this best case

$$L_t(v_i) \geq \frac{1}{q}$$

Given now that Quorum Q is the minimal Quorum, i.e. the one of minimal site, to access a quorum you need to access at least q nodes.

↳ Such that each node can expect to be accessed at least q/n times, which represents its load.

Summing up you have for the load

$$\max\left(\frac{1}{q}, \frac{q}{n}\right)$$

$$\max\left(\frac{1}{q}, \frac{q}{n}\right)$$

We will see next some systems to achieve this lower bound in terms of node load.

Definition 21.7 (Basic Grid quorum system). *Assume $\sqrt{n} \in \mathbb{N}$, and arrange the n nodes in a square matrix with side length of $\sqrt{n}$, i.e., in a grid. The basic* ***Grid*** *quorum system consists of $\sqrt{n}$ quorums, with each containing the full row i and the full column i, for $1 \leq i \leq \sqrt{n}$.*

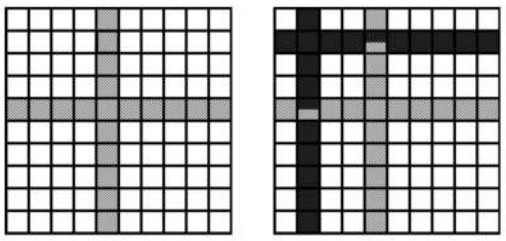
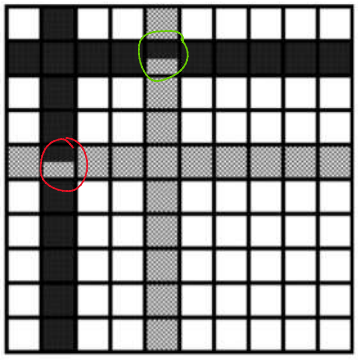


Figure 21.8: The basic version of the Grid quorum system, where each quorum Q_i with $1 \leq i \leq \sqrt{n}$ uses row i and column i. The size of each quorum is $2\sqrt{n}-1$ and two quorums overlap in exactly two nodes. Thus, when the access strategy Z is uniform (i.e., the probability of each quorum is $1/\sqrt{n}$), the work is $2\sqrt{n}-1$, and the load of every node is in $\Theta(1/\sqrt{n})$.

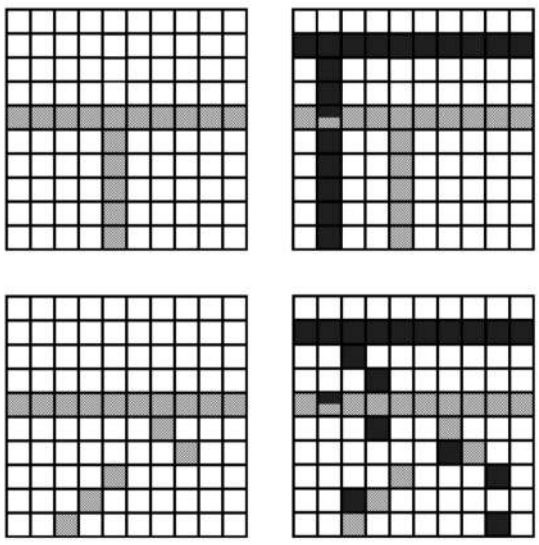
↳ reaches the lower bound

Notice however how in the above you can easily end up in a dead-lock.

For instance assume that as in the image left 1 quorum acquires ○ and one ○. Then both will be stucked forever waiting to acquire the last piece.



↳ One way to solve the issue is to slightly modify the quorum system such that there is intersection on just 1 node; for any two quorums



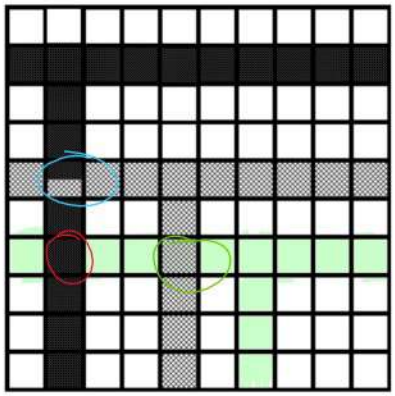
if highest quorum in first row.

it just take last row

Figure 21.9: There are other ways to choose quorums in the grid s.t. pairwise different quorums only intersect in one node. The size of each quorum is between $\sqrt{n}$ and $2\sqrt{n}-1$, i.e., the work is in $\Theta(\sqrt{n})$. When the access strategy Z is uniform, the load of every node is in $\Theta(1/\sqrt{n})$.

uniform, the load of every node is in $O(1/\sqrt{n})$.

However you can see that as soon as you have 3 clients accessing 3 quorums the entire system breaks down.



↳ each acquires a lock and waits for the others.

So we need another mechanism to ensure the avoidance of deadlocks:

Algorithm 21.10 Sequential Locking Strategy for a Quorum Q

1: Attempt to lock the nodes one by one, ordered by their identifiers
2: Should a node be already locked, release all locks and start over

2: Should a node be already locked, release all locks and start over

quite logical step

Theorem 21.11. *If each quorum is accessed by Algorithm 21.10, at least one quorum will obtain a lock for all of its nodes.*

But then we are back to the linear sequential access in a distributed system.

↳ How can we solve that?

Algorithm 21.12 Concurrent Locking Strategy for a Quorum Q

Invariant: Let $v_Q \in Q$ be the highest identifier of a node locked by Q s.t. all nodes $v_i \in Q$ with $v_i < v_Q$ are locked by Q as well. Should Q not have any lock, then v_Q is set to 0.

```
1: repeat
2:    Attempt to lock all nodes of the quorum Q
3:    for each node v ∈ Q that was not able to be locked by Q do
4:       exchange v_Q and v_Q' with the quorum Q' that locked v
5:       if v_Q > v_Q' then
6:          Q' releases lock on v and Q acquires lock on v
7:       end if
8:    end for
9: until all nodes of the quorum Q are locked
```

so all the quorums that have locked less number of nodes gives way to the one that already locked the most nodes.

Theorem 21.13. *If the nodes and quorums use Algorithm 21.12, at least one quorum will obtain a lock for all of its nodes.*

↳ the proof is again quite intuitive.

Theorem 21.13. *If the nodes and quorums use Algorithm 21.12, at least one quorum will obtain a lock for all of its nodes.*

↳ the proof is again quite intuitive.

Given now that we have found quorum system that reach a good load-work trade-off we will address now the question of fault tolerance and check at the properties of Grid-quorum systems in there.

Fault-Tolerance

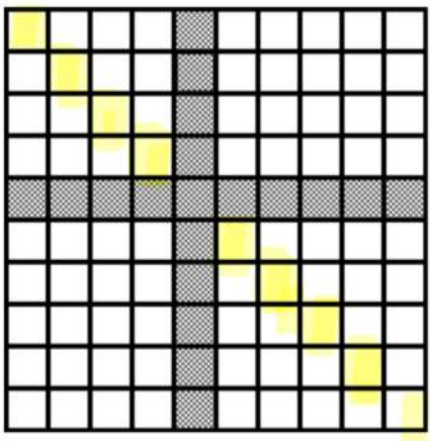
There are essentially two types of fault-tolerance, one is more deterministic, the other one probabilistic

Definition 21.14 (resilience). *If any f nodes from a quorum system $\mathcal{S}$ can fail s.t. there is still a quorum $Q \in \mathcal{S}$ without failed nodes, then $\mathcal{S}$ is f-**resilient**. The largest such f is the **resilience** $R(\mathcal{S})$.*

↳ notice that it can also be a very single one.

Theorem 21.15. *Let S be a Grid quorum system where each of the n quorums consists of a full row and a full column. S has a resilience of $\sqrt{n} - 1$.*

↳ Proof if everything fails on the diagonal but one cell then you are left with a single quorum without failed nodes.



We turn now to the probabilistic fault tolerance definition and we will see that the grid quorum systems

Note the grid quorum systems seen so far fail to guarantee a sound fail tolerance in expectation.

Definition 21.16 (failure probability). *Assume that every node works with a fixed probability p (in the following we assume concrete values, e.g. $p > 1/2$).*

*The **failure probability** $F_p(\mathcal{S})$ of a quorum system $\mathcal{S}$ is the probability that at least one node of every quorum fails.*

Given the above and the Chernoff-bounds it is immediate to derive the asymptotic failure probabilities for our quorum systems.

in our case fail vs. no-fail.

Facts 21.17. *A version of a **Chernoff bound** states the following:*
Let $x_1, \ldots, x_n$ be independent Bernoulli-distributed random variables with $Pr[x_i = 1] = p_i$ and $Pr[x_i = 0] = 1 - p_i = q_i$, then for $X := \sum_{i=1}^{n} x_i$ and $\mu := \mathbb{E}[X] = \sum_{i=1}^{n} p_i$ the following holds:

$$\text{for all } 0 < \delta < 1\text{: } Pr[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2}.$$

It now follows:

Theorem 21.18. *The asymptotic failure probability of the Majority quorum system is 0.*

Proof. In a Majority quorum system each quorum contains exactly $\lfloor \frac{n}{2} \rfloor + 1$ nodes and each subset of nodes with cardinality $\lfloor \frac{n}{2} \rfloor + 1$ forms a quorum. The Majority quorum system fails, if only $\lfloor \frac{n}{2} \rfloor$ nodes work. Otherwise there is at least one quorum available. In order to calculate the failure probability we define the following random variables:

$$x_i = \begin{cases} 1, & \text{if node } i \text{ works, happens with probability } p \\ 0, & \text{if node } i \text{ fails, happens with probability } q = 1 - p \end{cases}$$

define the following random variables:

$$x_i = \begin{cases} 1, & \text{if node } i \text{ works, happens with probability } p \\ 0, & \text{if node } i \text{ fails, happens with probability } q = 1-p \end{cases}$$

and $X := \sum_{i=1}^{n} x_i$, with $\mu = np$,

whereas X corresponds to the number of working nodes. To estimate the probability that the number of working nodes is less than $\lfloor \frac{n}{2} \rfloor + 1$ we will make use of the Chernoff inequality from above. By setting $\delta = 1 - \frac{1}{2p}$ we obtain $F_P(\mathcal{S}) = Pr[X \leq \lfloor \frac{n}{2} \rfloor] \leq Pr[X \leq \frac{n}{2}] = Pr[X \leq (1-\delta)\mu]$.

With $\delta = 1 - \frac{1}{2p}$ we have $0 < \delta \leq 1/2$ due to $1/2 < p \leq 1$. Thus, we can use the Chernoff bound and get $F_P(\mathcal{S}) \leq e^{-\mu\delta^2/2} \in e^{-\Omega(n)}$. as $\mu = n \cdot p$ □

Such that asymptotic failure probability of the majority quorum system goes to 0 exponentially fast in the number of nodes and is therefore resiliant.

Theorem 21.19. *The asymptotic failure probability of the Grid quorum system is 1.*

Proof. Consider the $n = d \cdot d$ nodes to be arranged in a $d \times d$ grid. A quorum always contains one full row. In this estimation we will make use of the Bernoulli inequality which states that for all $n \in \mathbb{N}, x \geq -1 : (1+x)^n \geq 1 + nx$.

The system fails, if in each row at least one node fails (which happens with probability $1 - p^d$ for a particular row, as all nodes work with probability p^d). of row

Therefore we can bound the failure probability from below with:

$F_p(\mathcal{S}) \geq Pr[\text{at least one failure per row}] = (1-p^d)^d \geq 1 - dp^d \underset{n\to\infty}{\longrightarrow} 1.$ □

Bernoulli Inequality

It therefore follows that a Grid quorum system is optimal for the load of the system however is not asymptotically failure tolerant.

↳ In contrast the Majority quorum system is not optimal in terms of load, however is failure-tolerant.

↳ Can we reach the best of both worlds?

Yes, solution.

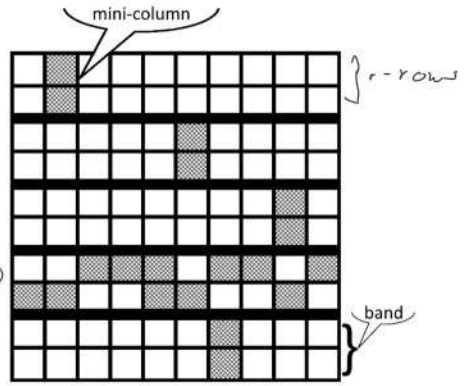
Definition 21.20 (B-Grid quorum system). *Consider $n = dhr$ nodes, arranged in a rectangular grid with $h \cdot r$ rows and d columns. Each group of r rows is a band, and r elements in a column restricted to a band are called a mini-column. A quorum consists of one mini-column in every band and one element from each mini-column of one band; thus every quorum has $d + hr - 1$ elements. The* ***B-Grid*** *quorum system consists of all such quorums.*

understand

for the band with 1 element per minicolumn

for all bands 1 minicolumn

as for the 1 minicol you do not have to cover a single element.



then clear that for two quorums you will always have at least 1 overlap given the mini-columns and the band with 1 element per minicolumn

Figure 21.21: A B-Grid quorum system with $n = 100$ nodes, $d = 10$ columns, $h \cdot r = 10$ rows, $h = 5$ bands, and $r = 2$. The depicted quorum has a $d + hr - 1 = 10 + 5 \cdot 2 - 1 = 19$ nodes. If the access strategy Z is chosen uniformly, then we have a work of $d + hr - 1$ and a load of $\frac{d+hr-1}{n}$. By setting $d = \sqrt{n}$ and $r = \log n$, we obtain a work of $\Theta(\sqrt{n})$ and a load of $\Theta(1/\sqrt{n})$.

↳ desired properties

Theorem 21.22. *The asymptotic failure probability of the B-Grid quorum system is* 0.

Proof. Suppose $n = dhr$ and the elements are arranged in a grid with d columns and $h \cdot r$ rows. The B-Grid quorum system does fail if in each band a complete mini-column fails, because then it is not possible to choose a band where in each mini-column an element is still working. It also fails if in a band an element in each mini-column fails. Those events may not be independent of each other, but with the help of the union bound, we can upper bound the failure probability with the following equation:

$$\begin{aligned} F_p(\mathcal{S}) &\leq Pr[\text{in every band a complete mini-column fails}] \\ &\quad + Pr[\text{in a band at least one element of every m.-col. fails}] \\ &\leq (d(1-p)^r)^h + h(1-p^r)^d \end{aligned}$$

We use $d = \sqrt{n}, r = \ln d$, and $0 \leq (1-p) \leq 1/3$. Using $n^{\ln x} = x^{\ln n}$, we have $d(1-p)^r \leq d \cdot d^{\ln 1/3} \approx d^{-0.1}$, and hence for large enough d the whole first term is bounded from above by $d^{-0.1h} \ll 1/d^2 = 1/n$.

Regarding the second term, we have $p \geq 2/3$, and $h = d/\ln d < d$. Hence we can bound the term from above by $d(1 - d^{\ln 2/3})^d \approx d(1 - d^{-0.4})^d$. Using $(1 + t/n)^n \leq e^t$, we get (again, for large enough d) an upper bound of $d(1 - d^{-0.4})^d = d(1 - d^{0.6}/d)^d \leq d \cdot e^{-d^{0.6}} = d^{(-d^{0.6}/\ln d)+1} \ll d^{-2} = 1/n$. In total, we have $F_p(\mathcal{S}) \in O(1/n)$. □

We finally have:

	Singleton	Majority	Grid	B-Grid*
Work	**1**	$> n/2$	$\Theta(\sqrt{n})$	$\Theta(\sqrt{n})$
Load	1	$> 1/2$	$\Theta(\mathbf{1/\sqrt{n}})$	$\Theta(\mathbf{1/\sqrt{n}})$
Resilience	0	$< \mathbf{n/2}$	$\Theta(\sqrt{n})$	$\Theta(\sqrt{n})$
F. Prob.**	$1-p$	$\to \mathbf{0}$	$\to 1$	$\to \mathbf{0}$

Table 21.23: Overview of the different quorum systems regarding resilience, work, load, and their asymptotic failure probability. The best entries in each row are set in bold.

* Setting $d = \sqrt{n}$ and $r = \log n$

**Assuming prob. $q = (1-p)$ is constant but significantly less than 1/2

This concludes our chapter, it is now possible to substitute the B-grid quorum system with the deadlock resolution mechanism of algo 21.12, in the majority sceme of Paxos and you

In the majority sceme of Paxos and you will obtain therein a more efficient algorithm.

We turn next to the question of Byzantine Quorum Systems - i.e. to the question about making quorum systems byzantine resistant.

21.4 Byzantine Quorum Systems

While failed nodes are bad, they are still easy to deal with: just access another quorum where all nodes can respond! Byzantine nodes make life more difficult however, as they can pretend to be a regular node, i.e., one needs more sophisticated methods to deal with them. We need to ensure that the intersection of two quorums always contains a non-byzantine (correct) node and furthermore, the byzantine nodes should not be allowed to infiltrate every quorum. In this section we study three counter-measures of increasing strength, and their implications on the load of quorum systems.

↳ If a node in Quorum byzantine then it can tell two clients that there is no conflict even if there is so that they will perform contradictory/not aligned actions and the system will break down

Method 1:

Definition 21.24 (f-disseminating). *A quorum system S is **f-disseminating** if (1) the intersection of two different quorums always contains $f+1$ nodes, and (2) for any set of f byzantine nodes, there is at least one quorum without byzantine nodes.*

and (2) for any set of f byzantine nodes, there is at least one quorum without byzantine nodes.

Remarks:

- Thanks to (2), even with f byzantine nodes, the byzantine nodes cannot stop all quorums by just pretending to have crashed. At least one quorum will survive. We will also keep this assumption for the upcoming more advanced byzantine quorum systems.
- Byzantine nodes can also do something worse than crashing - they could falsify data! Nonetheless, due to (1), there is at least one non-byzantine node in every quorum intersection. If the data is self-verifying by, e.g., authentication, then this one node is enough.
- If the data is not self-verifying, then we need another mechanism.

i.e. if we can identify that node it is immediate that we can trust it to know the state of the system.

Method (2)

Definition 21.25 (f-masking). *A quorum system $\mathcal{S}$ is f-**masking** if (1) the intersection of two different quorums always contains $2f+1$ nodes, and (2) for any set of f byzantine nodes, there is at least one quorum without byzantine nodes.*

Remarks:

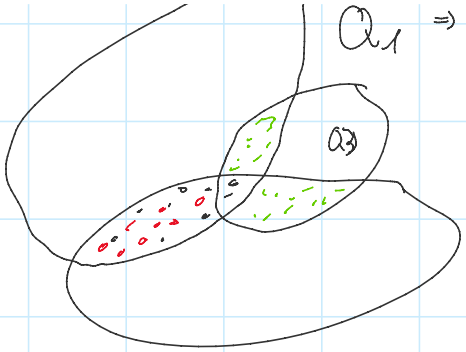
- Note that except for the second condition, an f-masking quorum system is the same as a $2f$-disseminating system. The idea is that the non-byzantine nodes (at least $f+1$ can outvote the byzantine ones (at most f), but only if all non-byzantine nodes are up-to-date!

as otherwise they will start to vote different stuff.

- This raises an issue not covered yet in this chapter. If we access some quorum and update its values, this change still has to be disseminated to the other nodes in the byzantine quorum system. Opaque quorum systems deal with this issue, which are discussed at the end of this section.

To understand that consider:

$Q_1 \Rightarrow$ outdated nodes, are not up-to-date



Q_1 ⇒ outdated nodes, are not up-to date with the latest update.

Q_2 ⇒ up-to-date nodes

∴ = f+1 nodes

∴ = f byzantine nodes.

∴ = intersection with other quorums such that sum ∴ = f+1

Idea then if among the f+1 nodes some are supporting the old values and some are supporting the new values then the byzantine nodes might be the majority in the quorum intersection and might still have power.

↳ This will be solved by opaque quorum systems.

Notice, moreover, that in order to reach a majority among the quorums so that just one can make progress

- f-disseminating quorum systems need more than $3f$ nodes and f-masking quorum systems need more than $4f$ nodes. Essentially, the quorums may not contain too many nodes, and the different intersection properties lead to the different bounds.

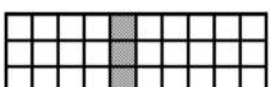
However, notice that if quorums share the same state it is immediate that you can follow the majority of the $f+1$ in each intersection to get to know the true state of the system.

Theorem 21.26. *Let $\mathcal{S}$ be a f-disseminating quorum system. Then $L(\mathcal{S}) \geq \sqrt{(f+1)/n}$ holds.*

Theorem 21.27. *Let $\mathcal{S}$ be a f-masking quorum system. Then $L(\mathcal{S}) \geq \sqrt{(2f+1)/n}$ holds.*

Proofs of Theorems 21.26 and 21.27. The proofs follow the proof of Theorem 21.6, by observing that now not just one element is accessed from a minimal quorum, but $f+1$ or $2f+1$, respectively. □

Definition 21.28 (f-masking Grid quorum system)**.** *A **f-masking Grid** quorum system is constructed as the grid quorum system, but each quorum contains one full column and $f+1$ rows of nodes, with $2f+1 \leq \sqrt{n}$.*



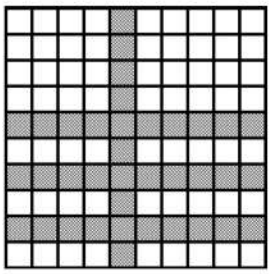
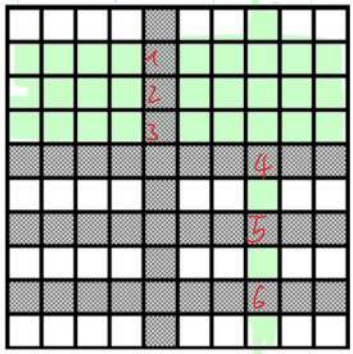


Figure 21.29: An example how to choose a quorum in the f-masking Grid with $f = 2$, i.e., $2 + 1 = 3$ rows. The load is in $\Theta(f/\sqrt{n})$ when the access strategy is chosen to be uniform. Two quorums overlap by their columns intersecting each other's rows, i.e., they overlap in at least $2f + 2$ nodes.

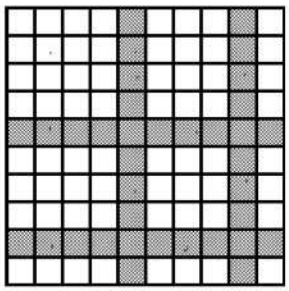


$6 = 2(f+1)$ (3 over the $f+1$)

Remarks:

- The f-masking Grid nearly hits the lower bound for the load of f-masking quorum systems, but not quite. A small change and we will be optimal asymptotically.

Definition 21.30 (M-Grid quorum system)**.** *The* M***-Grid*** *quorum system is constructed as the grid quorum as well, but each quorum contains* $\sqrt{f+1}$ *rows and* $\sqrt{f+1}$ *columns of nodes, with* $f \leq \frac{\sqrt{n}-1}{2}$.



$\sqrt{3+1}$

Figure 21.31: An example how to choose a quorum in the M-Grid with $f = 3$, i.e., 2 rows and 2 columns. The load is in $\Theta(\sqrt{f/n})$ when the access strategy is chosen to be uniform. Two quorums overlap with each row intersecting each other's column, i.e., $2\sqrt{f+1}^2 = 2f + 2$ nodes.

have to see finding these bounds.

Corollary 21.32. *The* f*-masking Grid quorum system and the* M*-Grid quorum*

Corollary 21.32. *The f-masking Grid quorum system and the M-Grid quorum system are f-masking quorum systems.*

We achieved nearly the same load as without byzantine nodes!

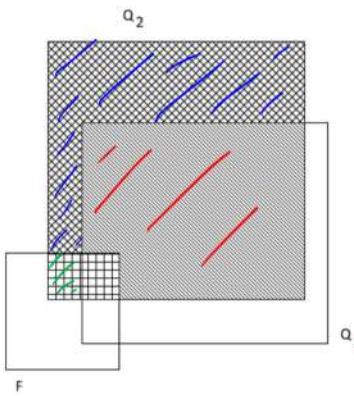
The question is now what happens if we access a quorum that is not up to date except for its intersection with an up-to-date quorum.

↳ We will show next a quorum system that guarantees for choosing up-to-date values:

Definition 21.33 (f-opaque quorum system)**.** *A quorum system $\mathcal{S}$ is f-**opaque** if the following two properties hold for any set of f byzantine nodes F and any two different quorums Q_1, Q_2:*

$$|(Q_1 \cap Q_2) \setminus F| > |(Q_2 \cap F) \cup (Q_2 \setminus Q_1)| \tag{21.33.1}$$

1 quorum w. byzantine { $(F \cap Q) = \emptyset$ *for some* $Q \in \mathcal{S}$ (21.33.2)



Q_1 - up to date quorum so that Q_2 intersecting will this up to date w. we'll.

Figure 21.34: Intersection properties of an opaque quorum system. Equation (21.33.1) ensures that the set of non-byzantine nodes in the intersection of Q_1, Q_2 is larger than the set of out of date nodes, even if the byzantine nodes "team up" with those nodes. Thus, the correct up to date value can always be recognized by a majority voting.

Theorem 21.35. *Let $\mathcal{S}$ be a f-opaque quorum system. Then, $n > 5f$.*

Proof. Due to (21.33.2), there exists a quorum Q_1 with size at most $n-f$. With (21.33.1), $|Q_1| > f$ holds. Let F_1 be a set of f (byzantine) nodes $F_1 \subset Q_1$, and with (21.33.2), there exists a $Q_2 \subset V \setminus F_1$. Thus, $|Q_1 \cap Q_2| \leq n - 2f$. With (21.33.1), $|Q_1 \cap Q_2| > f$ holds. Thus, one could choose f (byzantine) nodes F_2 with $F_2 \subset (Q_1 \cap Q_2)$. Using (21.33.1) one can bound $n - 3f$ from below: $n - 3f > |(Q_2 \cap Q_1)| - |F_2| \geq |(Q_2 \cap Q_1) \cup (Q_1 \cap F_2)| \geq |F_1| + |F_2| = 2f$. □

Remarks:

- One can extend the Majority quorum system to be f-opaque by setting the size of each quorum to contain $\lceil (2n+2f)/3 \rceil$ nodes. Then its load is $1/n \lceil (2n+2f)/3 \rceil \approx 2/3 + 2f/3n \geq 2/3$.
- Can we do much better? Sadly, no...

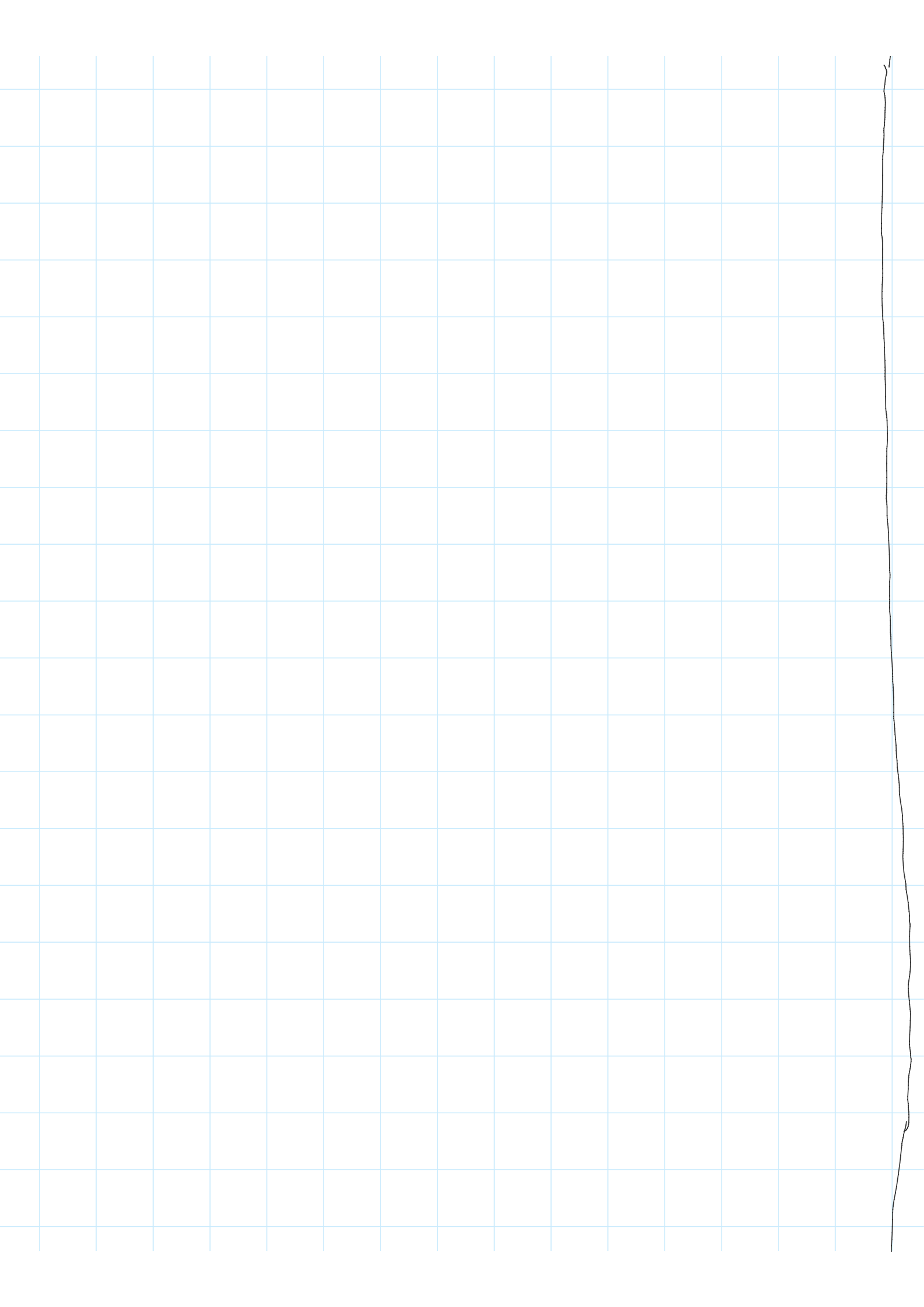
Theorem 21.36. *Let $\mathcal{S}$ be a f-opaque quorum system. Then $L(\mathcal{S}) \geq 1/2$ holds.*

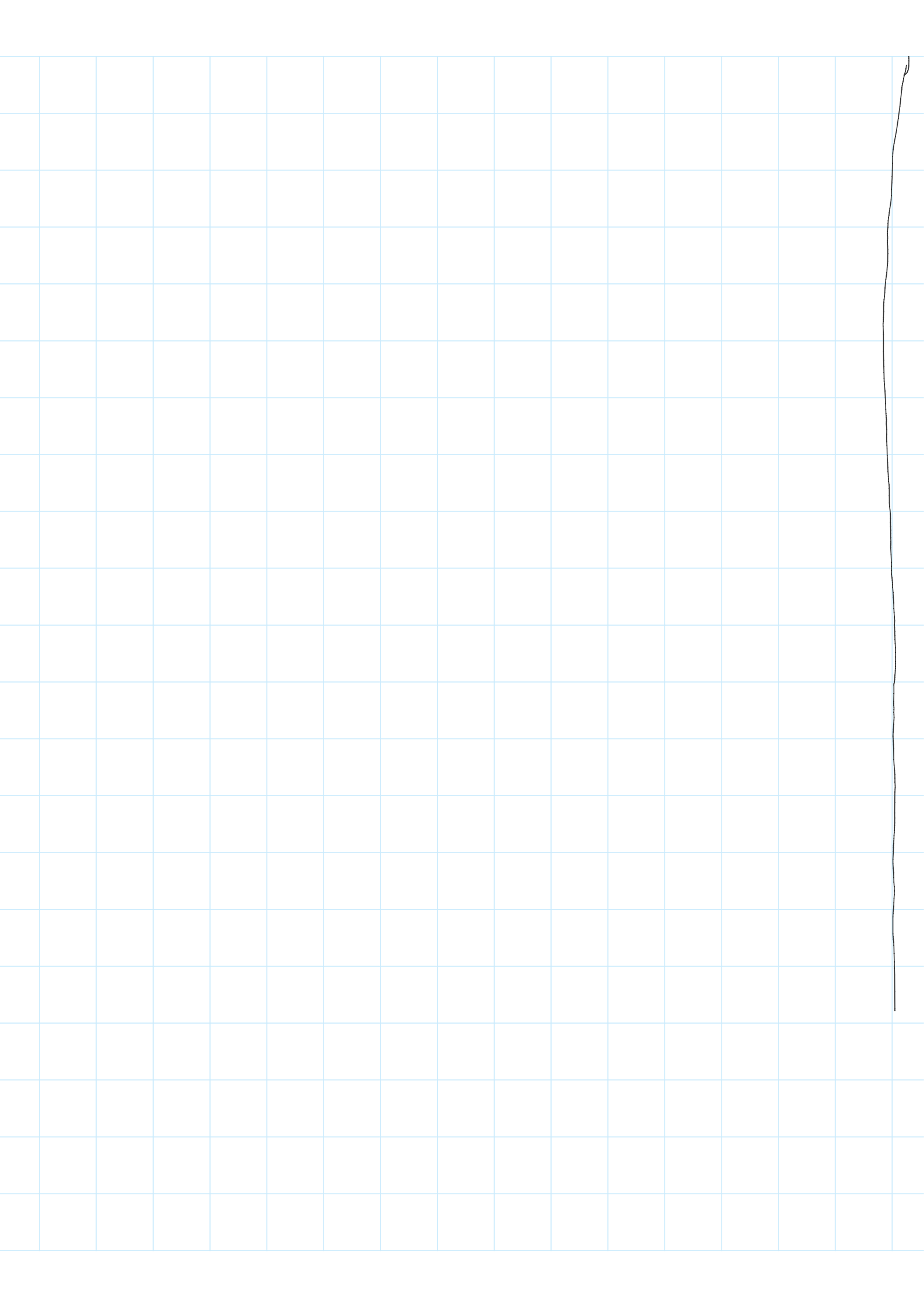
Proof. Equation (21.33.1) implies that for $Q_1, Q_2 \in \mathcal{S}$, the intersection of both Q_1, Q_2 is at least half their size, i.e., $|(Q_1 \cap Q_2)| \geq |Q_1|/2$. Let $\mathcal{S}$ consist of quorums $Q_1, Q_2, \ldots$. The load induced by an access strategy Z on Q_1 is:

$$\sum_{v \in Q_1} \sum_{v \in Q_i} L_Z(Q_i) = \sum_{Q_i} \sum_{v \in (Q_1 \cap Q_i)} L_Z(Q_i) \geq \sum_{Q_i} (|Q_1|/2)\, L_Z(Q_i) = |Q_1|/2 \,.$$

Using the pigeonhole principle, there must be at least one node in Q_1 with load of at least $1/2$. □

Such that we are essentially back to a majority scenario.





Consistency and Bitcoin

29 November 2020 14:46

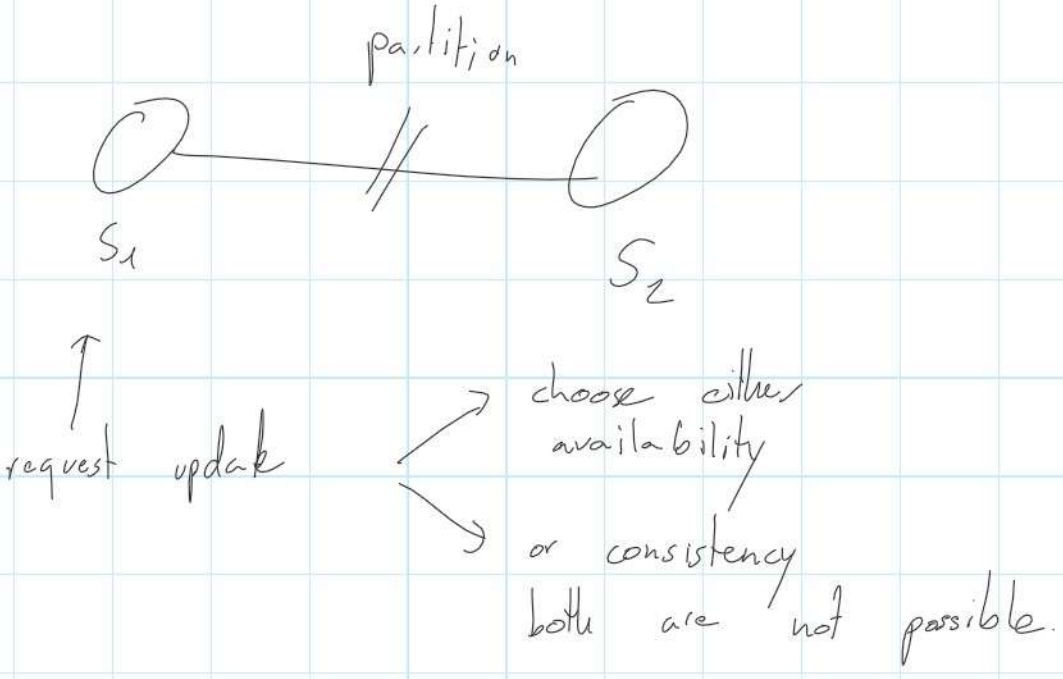
Definition 22.2 (Consistency). *All nodes in the system agree on the current state of the system.*

Definition 22.3 (Availability). *The system is operational and instantly processing incoming requests.*

Definition 22.4 (Partition Tolerance). *Partition tolerance is the ability of a distributed system to continue operating correctly even in the presence of a network partition.*

Theorem 22.5 (CAP Theorem). *It is impossible for a distributed system to simultaneously provide Consistency, Availability and Partition Tolerance. A distributed system can satisfy any two of these but not all three.*

Proof. Assume two nodes, sharing some state. The nodes are in different partitions, i.e., they cannot communicate. Assume a request wants to update the state and contacts a node. The node may either: 1) update its local state, resulting in inconsistent states, or 2) not update its local state, i.e., the system is no longer available for updates. □



give up on consistency

Algorithm 22.6 Partition tolerant and available ATM

give up on consistency

Algorithm 22.6 Partition tolerant and available ATM

```
1: if bank reachable then
2:     Synchronize local view of balances between ATM and bank
3:     if balance of customer insufficient then
4:         ATM displays error and aborts user interaction
5:     end if
6: end if
7: ATM dispenses cash
8: ATM logs withdrawal for synchronization
```

} otherwise.
↳ so loose consistency.

Definition 22.7 (Eventual Consistency). *If no new updates to the shared state are issued, then eventually the system is in a quiescent state, i.e., no more messages need to be exchanged between nodes, and the shared state is consistent.*

Remarks:

- Eventual consistency is a form of *weak consistency*.
- Eventual consistency guarantees that the state is eventually agreed upon, but the nodes may disagree temporarily.
- During a partition, different updates may semantically conflict with each other. A *conflict resolution* mechanism is required to resolve the conflicts and allow the nodes to eventually agree on a common state.

↳ One mechanism that ensures eventual consistency is bitcoin that we are going to address next.

Bitcoin

Definition 22.8 (Bitcoin Network). *The Bitcoin network is a randomly con-*

Definition 22.8 (Bitcoin Network). *The Bitcoin network is a randomly connected overlay network of a few thousand* ***nodes****, controlled by a variety of owners. All nodes perform the same operations, i.e., it is a homogenous network and without central control.*

Definition 22.9 (Address). *Users may generate any number of private keys, from which a public key is then derived. An address is derived from a public key and may be used to identify the recipient of funds in Bitcoin. The private/public key pair is used to uniquely identify the owner of funds of an address.*

- Not every user needs to run a fully validating node, and end-users will likely use a lightweight client that only temporarily connects to the network.

- The address is composed of a network identifier byte, the hash of the public key and a checksum. It is commonly stored in base 58 encoding, a custom encoding similar to base 64 with some ambiguous symbols removed, e.g., lowercase letter "l" since it is similar to the number "1".

Definition 22.10 (Output). *An output is a tuple consisting of an amount of bitcoins and a spending condition. Most commonly the spending condition requires a valid signature associated with the private key of an address.*

Remarks:

- Spending conditions are scripts that offer a variety of options. Apart from a single signature, they may include conditions that require the result of a simple computation, or the solution to a cryptographic puzzle.

- Outputs exist in two states: unspent and spent. Any output can be spent at most once. The address balance is the sum of bitcoin amounts in unspent outputs that are associated with the address.

- The set of unspent transaction outputs (UTXOs) and some additional global parameters are the shared state of Bitcoin. Every node in the Bitcoin network holds a complete replica of that state. Local replicas may temporarily diverge, but consistency is eventually re-established.

So very important recall that the Bitcoin network essentially keeps track of UTXO.

Definition 22.11 (Input). *An input is a tuple consisting of a reference to a previously created output and arguments (signature) to the spending condition, proving that the transaction creator has the permission to spend the referenced output.*

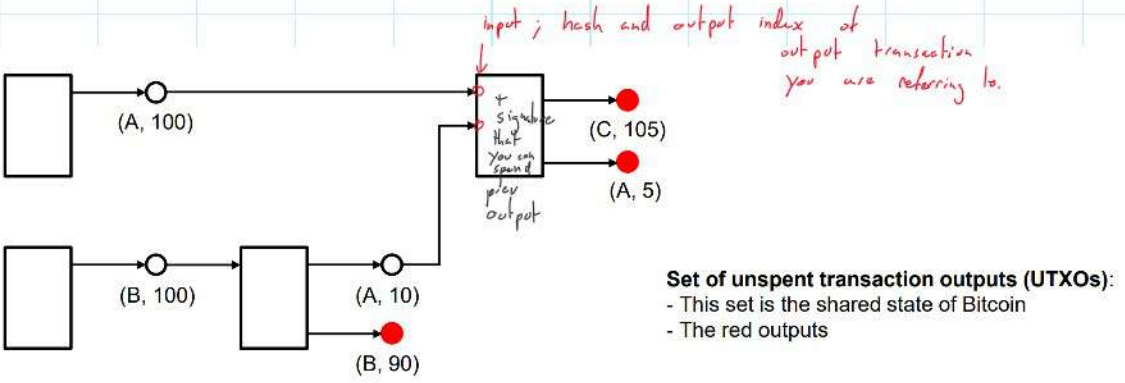
Definition 22.12 (Transaction). *A transaction is a data structure that describes the transfer of bitcoins from spenders to recipients. The transaction consists of a number of inputs and new outputs. The inputs result in the referenced outputs spent (removed from the UTXO), and the new outputs being added to the UTXO.*

Remarks:

- Inputs reference the output that is being spent by a (h, i)-tuple, where h is the hash of the transaction that created the output, and i specifies the index of the output in that transaction.
- Transactions are broadcast in the Bitcoin network and processed by every node that receives them.

necessary as a transaction can have multiple outputs

So summing it all up:



Concretely, you would perform a transaction in the following way:

Algorithm 22.13 Node Receives Transaction

1: Receive transaction t
2: **for each** input (h, i) in t **do**
3: **if** output (h, i) is not in local UTXO **or** signature invalid **then**
4: Drop t and stop
5: **end if**
6: **end for**
7: **if** sum of values of inputs $<$ sum of values of new outputs **then**
8: Drop t and stop
9: **end if**
10: **for each** input (h, i) in t **do**
11: Remove (h, i) from local UTXO
12: **end for**
13: Append t to local history
14: Forward t to neighbors in the Bitcoin network

means the node already received a transaction spending that output

no right to spend

not sufficient fundings for the transaction.

if conditions satisfied and carrying out the transaction.

Remarks:

note that the deterministic component is essential as all the nodes should eventually perform the same tasks and reach consensus.

- Note that the effect of a transaction on the state is deterministic. In other words if all nodes receive the same set of transactions in the same order (Definition 15.8), then the state across nodes is consistent.
- The outputs of a transaction may assign less than the sum of inputs, in which case the difference is called the transaction *fee*. The fee is used to incentivize other participants in the system (see Definition 22.19)
- Notice that so far we only described a local acceptance policy. Nothing prevents nodes to locally accept different transactions that spend the same output.

different

- Transactions are in one of two states: unconfirmed or confirmed. Incoming transactions from the broadcast are unconfirmed and added to a pool of transactions called the *memory pool*.

you will see later that just transactions that are in the longest chain of blocks and are within a block are "confirmed" / i.e. they must have been approved by the conflict resolution scheme.

Definition 22.14 (Doublespend). *A doublespend is a situation in which multiple transactions attempt to spend the same output. Only one transaction can be valid since outputs can only be spent once. When nodes accept different trans-*

Definition 22.14 (Doublespend). *A doublespend is a situation in which multiple transactions attempt to spend the same output. Only one transaction can be valid since outputs can only be spent once. When nodes accept different transactions in a doublespend, the shared state becomes inconsistent.*

Remarks:

- Doublespends may occur naturally, e.g., if outputs are co-owned by multiple users. However, often doublespends are intentional – we call these doublespend-attacks: In a transaction, an attacker pretends to transfer an output to a victim, only to doublespend the same output in another transaction back to itself.
- Doublespends can result in an inconsistent state since the validity of transactions depends on the order in which they arrive. If two conflicting transactions are seen by a node, the node considers the first to be valid, see Algorithm 22.13. The second transaction is invalid since it tries to spend an output that is already spent. The order in which transactions are seen, may not be the same for all nodes, hence the inconsistent state.
- If doublespends are not resolved, the shared state diverges. Therefore a conflict resolution mechanism is needed to decide which of the conflicting transactions is to be confirmed (accepted by everybody), to achieve eventual consistency.

Definition 22.15 (Proof-of-Work). *Proof-of-Work (PoW) is a mechanism that allows a party to prove to another party that a certain amount of computational resources has been utilized for a period of time. A function $\mathcal{F}_d(c, x) \to \{true, false\}$, where difficulty d is a positive number, while challenge c and nonce x are usually bit-strings, is called a Proof-of-Work function if it has following properties:*

1. *$\mathcal{F}_d(c, x)$ is fast to compute if d, c, and x are given.*
2. *For fixed parameters d and c, finding x such that $\mathcal{F}_d(c, x) = true$ is computationally difficult but feasible. The difficulty d is used to adjust the time to find such an x.*

Definition 22.16 (Bitcoin PoW function). *The Bitcoin PoW function is given by*

$$\mathcal{F}_d(c, x) \to \text{SHA256}(\text{SHA256}(c|x)) < \frac{2^{224}}{d}.$$

Notice:

- If the PoW functions of all nodes had the same challenge, the fastest node would always win. However, as we will see in Definition 22.19, each node attempts to find a valid nonce for a node-specific challenge.

Definition 22.17 (Block). *A block is a data structure used to communicate incremental changes to the local state of a node. A block consists of a list of transactions, a reference to a previous block and a nonce. A block lists some transactions the block creator ("miner") has accepted to its memory pool since the previous block. A node finds and broadcasts a block when it finds a valid nonce for its PoW function.*

Algorithm 22.18 Node Finds Block

1: Nonce $x = 0$, challenge c, difficulty d, previous block b_{t-1}
2: **repeat**
3: $\quad x = x + 1$
4: **until** $\mathcal{F}_d(c, x) = true$
5: Broadcast block $b_t = (memory\ pool, b_{t-1}, x)$

so that the miner that finds the solution has the right to impose it to all others.

Remarks:

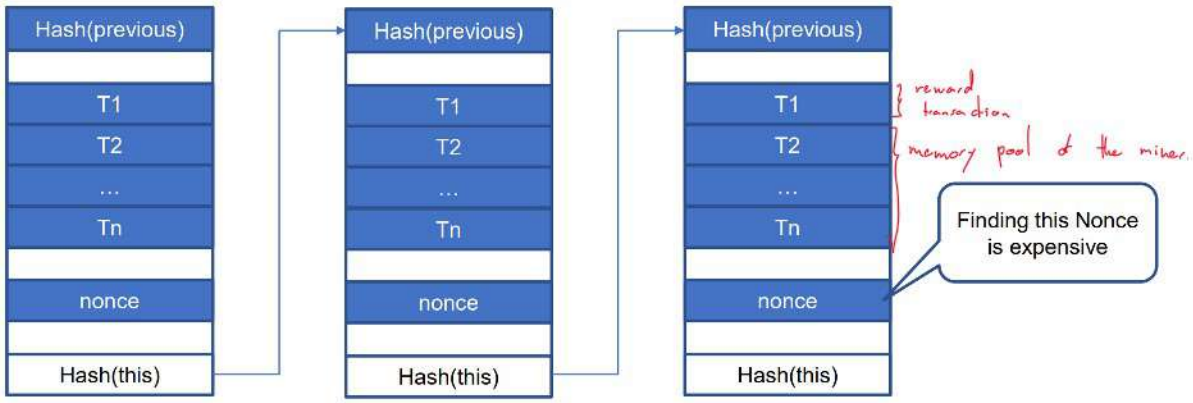
- With their reference to a previous block, the blocks build a tree, rooted in the so called *genesis block*.
- The primary goal for using the PoW mechanism is to adjust the rate at which blocks are found in the network, giving the network time to synchronize on the latest block. Bitcoin sets the difficulty so that globally a block is created about every 10 minutes in expectation. 3on expected
- Finding a block allows the finder to impose the transactions in its local memory pool to all other nodes. Upon receiving a block, all nodes roll back any local changes since the previous block and apply the new block's transactions.
- Transactions contained in a block are said to be *confirmed* by that block.

Definition 22.19 (Reward Transaction). *The first transaction in a block is called the reward transaction. The block's miner is rewarded for confirming transactions by allowing it to mint new coins. The reward transaction has a dummy input, and the sum of outputs is determined by a fixed subsidy plus the*

called the reward transaction. The block's miner is rewarded for confirming transactions by allowing it to mint new coins. The reward transaction has a dummy input, and the sum of outputs is determined by a fixed subsidy plus the sum of the fees of transactions confirmed in the block.

- A reward transaction is the sole exception to the rule that the sum of inputs must be at least the sum of outputs.

Summing Up:



↳ The longest of such chain is called blockchain:

Definition 22.20 (Blockchain)**.** *The longest path from the genesis block, i.e., root of the tree, to a leaf is called the blockchain. The blockchain acts as a consistent transaction history on which all nodes eventually agree.*

- Since only transactions in the longest path are agreed upon, miners have an incentive to append their blocks to the longest chain, thus agreeing on the current state.

have an incentive to append their blocks to the longest chain, thus agreeing on the current state.

- If multiple blocks are mined more or less concurrently, the system is said to have *forked*. Forks happen naturally because mining is a distributed random process and two new blocks may be found at roughly the same time.

Note now the process of updating the UTXO when receiving a block:

Algorithm 22.21 Node Receives Block

1: Receive block b
2: For this node the current head is block b_{max} at height h_{max}
3: Connect block b in the tree as child of its parent p at height $h_b = h_p + 1$
4: **if** $h_b > h_{max}$ **then**
5: $h_{max} = h_b$
6: $b_{max} = b$
7: Compute UTXO for the path leading to b_{max}
8: Cleanup memory pool
9: **end if**

so notice once the memory pool is cleaned if it was not inputted in a block "living" in a side-fork (shorter tree) if everyone cleans up memory no-one will know about potential transactions in the network (that would have but did not occur). This is an interesting

will not occur. This is an interesting point to me.

- Unlike extending the current path, switching paths may result in confirmed transactions no longer being confirmed, because the blocks in the new path do not include them. Switching paths is referred to as a *reorg*.

- Cleaning up the memory pool involves 1) removing transactions that were confirmed in a block in the current path, 2) removing transactions that conflict with confirmed transactions and 3) adding transactions that were confirmed in the previous path, but are no longer confirmed in the current path.

→ so already took place

→ unconfirmed } so they can change from confirmed to unconfirmed and be added to the memory pool in case of reorg.

interesting process!

- In order to avoid having to recompute the entire UTXO at every new block being added to the blockchain, all current implementations use data structures that store undo information about the operations applied by a block. This allows efficient switching of paths and updates of the head by moving along the path.

Theorem 22.22. *Forks are eventually resolved and all nodes eventually agree on which is the longest blockchain. The system therefore guarantees eventual consistency.*

↳ Clear proof omitted.

We turn now to Bitcoin Smart Contracts.

Notice that in Bitcoin the logic was constrained and so it is not possible to express everything with Smart Contracts in there.

with Smart Contracts in there.

↳ They are not complete turing machines so to say. as Bitcoin was designed giving up flow control so to say. in order to avoid infinite loops that would keep up the network from running and not reaching eventual consistency (in Ethereum this kind of solved through Gas).

↳ So while it is not possible to encode any computation into Bitcoin smart contracts some operations are still possible which we are going to discuss next.

Definition 22.23 (Smart Contract). *A smart contract is an agreement between two or more parties, encoded in such a way that the correct execution is guaranteed by the blockchain.*

- Contracts allow business logic to be encoded in Bitcoin transactions which mutually guarantee that an agreed upon action is performed. The blockchain acts as conflict mediator, should a party fail to honor an agreement.
- The use of scripts as spending conditions for outputs enables smart contracts. Scripts, together with some additional features such as timelocks, allow encoding complex conditions, specifying who may

- The use of scripts as spending conditions for outputs enables smart contracts. Scripts, together with some additional features such as timelocks, allow encoding complex conditions, specifying who may spend the funds associated with an output and when.

Definition 22.24 (Timelock). *Bitcoin provides a mechanism to make transactions invalid until some time in the future:* ***timelocks****. A transaction may specify a locktime: the earliest time, expressed in either a Unix timestamp or a blockchain height, at which it may be included in a block and therefore be confirmed.*

- Transactions with future timelocks are invalid. Blocks may not include transactions with timelocks that have not yet expired, i.e., they are mined before their expiry timestamp or in a lower block than specified. If a block includes an unexpired transaction it is invalid. Upon receiving invalid transactions or blocks, nodes discard them immediately and do not forward them to their peers.

- Timelocks can be used to replace or supersede transactions: a timelocked transaction t_1 can be replaced by another transaction t_0, spending some of the same outputs, if the replacing transaction t_0 has an earlier timelock and can be broadcast in the network before the replaced transaction t_1 becomes valid.

Definition 22.25 (Singlesig and Multisig Outputs). *When an output can be claimed by providing a single signature it is called a* ***singlesig output****. In contrast the script of* ***multisig outputs*** *specifies a set of m public keys and requires k-of-m (with $k \leq m$) valid signatures from distinct matching public keys from that set in order to be valid.*

Given these two options it is easy to set up shared wallets

to set up shared wallets from which to spend outputs just if the other person spends it part as well.

- Most smart contracts begin with the creation of a 2-of-2 multisig output, requiring a signature from both parties. Once the transaction creating the multisig output is confirmed in the blockchain, both parties are guaranteed that the funds of that output cannot be spent unilaterally.

→ inputs with valid UTXO belonging to B.

Algorithm 22.26 Parties A and B create a 2-of-2 multisig output o

1: B sends a list I_B of inputs with c_B coins to A
2: A selects its own inputs I_A with c_A coins
3: A creates transaction $t_s\{[I_A, I_B], [o = c_A + c_B \rightarrow (A, B)]\}$
4: A creates timelocked transaction $t_r\{[o], [c_A \rightarrow A, c_B \rightarrow B]\}$ and signs it
5: A sends t_s and t_r to B
6: B signs both t_s and t_r and sends them to A
7: A signs t_s and broadcasts it to the Bitcoin network

3 makes sure that if the output was not spent after a while it is returned.

here B could broadcast just t_r however if the funds in the common wallet has nothing to return. Te. final output does not exists.

Remarks:

- t_s is called a *setup transaction* and is used to lock in funds into a shared account. If t_s is signed and broadcast immediately, one of the parties could not collaborate to spend the multisig output, and the funds become unspendable. To avoid a situation where the funds cannot be spent, the protocol also creates a timelocked *refund transaction* t_r which guarantees that, should the funds not be spent before the timelock expires, the funds are returned to the respective party. At no point in time one of the parties holds a fully signed setup transaction without the other party holding a fully signed refund transaction, guaranteeing that funds are eventually returned.

- Both transactions require the signature of both parties. In the case of the setup transaction because it has two inputs from A and B respectively which require individual signatures. In the case of the refund transaction the single input spending the multisig output requires both signatures being a 2-of-2 multisig output.

Moreover, it is possible to set up micropayment channels that would work bilaterally next to the blockchain

bitaterally next to the blockchain and could be fired up just at the end.

Idea, you keep the accounting in the scripts and fire out at the end.

Algorithm 22.27 Simple Micropayment Channel from S to R with capacity c

1: $c_S = c$, $c_R = 0$
2: S and R use Algorithm 22.26 to set up output o with value c from S
3: Create settlement transaction $t_f\{[o], [c_S \to S, c_R \to R]\}$
4: **while** channel open **and** $c_R < c$ **do**
5: In exchange for good with value δ
6: $c_R = c_R + \delta$
7: $c_S = c_S - \delta$
8: Update t_f with outputs $[c_R \to R, c_S \to S]$
9: S signs and sends t_f to R
10: **end while**
11: R signs last t_f and broadcasts it

- Algorithm 22.27 implements a Simple Micropayment Channel, a smart contract that is used for rapidly adjusting micropayments from a spender to a recipient. Only two transactions are ever broadcast and inserted into the blockchain: the setup transaction t_s and the last settlement transaction t_f. There may have been any number of updates to the settlement transaction, transferring ever more of the shared output to the recipient.

- The number of bitcoins c used to fund the channel is also the maximum total that may be transferred over the simple micropayment channel.

- At any time the recipient R is guaranteed to eventually receive the bitcoins, since she holds a fully signed settlement transaction, while the spender only has partially signed ones.

- The simple micropayment channel is intrinsically unidirectional. Since the recipient may choose any of the settlement transactions in the protocol, she will use the one with maximum payout for her. If we were to transfer bitcoins back, we would be reducing the amount paid out to the recipient, hence she would choose not to broadcast that transaction.

only issue with the above

So it is rather intended to be used in a directed way due to

22.4 Weak Consistency

Eventual consistency is only one form of weak consistency. A number of different tradeoffs between partition tolerance and consistency exist in literature.

Definition 22.28 (Monotonic Read Consistency)**.** *If a node u has seen a particular value of an object, any subsequent accesses of u will never return any older values.*

Definition 22.29 (Monotonic Write Consistency)**.** *A write operation by a node on a data item is completed before any successive write operation by the same node (i.e., system guarantees to serialize writes by the same node).*

Definition 22.30 (Read-Your-Write Consistency)**.** *After a node u has updated a data item, any later reads from node u will never see an older value.*

Definition 22.31 (Causal Relation)**.** *The following pairs of operations are said to be causally related:*

- *Two writes by the same node to different variables.*
- *A read followed by a write of the same node.*
- *A read that returns the value of a write from any node.*
- *Two operations that are transitively related according to the above conditions.*

Definition 22.32 (Causal Consistency)**.** *A system provides causal consistency if operations that potentially are causally related are seen by every node of the system in the same order. Concurrent writes are not causally related, and may be seen in different orders by different nodes.*

Game Theory

05 December 2020 15:39

So far we have only considered a subset of nodes malfunctioning.

In particular, we saw:

- Crash Failures
- Byzantine Nodes.

This section tries to address a more general issue that repeatedly occurs when designing distributed systems protocols, i.e it deals with the selfish behaviour of nodes.

↳ When doing so it will borrow from game theory, for analyzing the

Game theory, for analyzing the possible behaviour of the nodes, and from mechanism design, in order to set up a protocol in the best possible functioning way.

We will start with the most basic behaviour you studied multiple times in your Eco classes!

PRISONER'S DILEMMA

	u	Player u	
v		Cooperate	Defect
Player v	Cooperate	1 1	0 3
	Defect	3 0	2 2

Table 23.1: The prisoner's dilemma game as a matrix.

- best reaction play v

- best reaction play u

It is then easy to identify the dominant strategies, the nash-equilibrium and the social optimum.

Definition 23.2 (game). *A game requires at least two rational players, and each player can choose from at least two options (**strategies**). In every possible outcome (**strategy profile**) each player gets a certain payoff (or cost). The payoff of a player depends on the strategies of the other players.*

*each player can choose from at least two options (**strategies**). In every possible outcome (**strategy profile**) each player gets a certain payoff (or cost). The payoff of a player depends on the strategies of the other players.*

Definition 23.3 (social optimum). *A strategy profile is called social optimum (SO) if and only if it minimizes the sum of all costs (or maximizes payoff).*

Definition 23.4 (dominant). *A strategy is dominant if a player is never worse off by playing this strategy. A dominant strategy profile is a strategy profile in which each player plays a dominant strategy.*

Definition 23.5 (Nash Equilibrium). *A Nash Equilibrium (NE) is a strategy profile in which no player can improve by unilaterally (the strategies of the other players do not change) changing its strategy.*

We immediately see that to defect is a dominant strategy for both players, that is the NE and that is the social optimum.

- The best response is the best strategy given a belief about the strategy of the other players. In this game the best response to both strategies of the other player is to defect. If one strategy is the best response to any strategy of the other players, it is a dominant strategy.

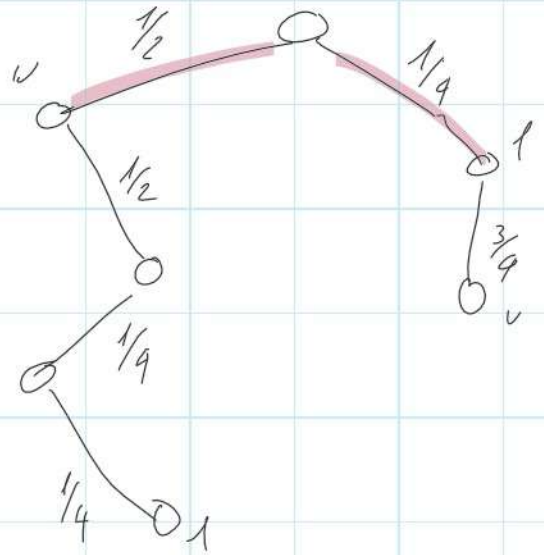
Given this quick intro we will apply now these concepts to distributed systems "games".

23.3 Selfish Caching

Game setting:

Computers in a network want to access a file regularly. Each node $v \in V$, with V being the set of nodes and $n = |V|$, has a demand d_v for the file and wants to minimize the cost for accessing it. In order to access the file, node v can either cache the file locally which costs 1 or request the file from another node u which costs $c_{v \leftarrow u}$. If a node does not cache the file, the cost it incurs is the minimal cost to access the file remotely. Note that if no node caches the file, then every node incurs cost ∞. There is an example in Figure 23.6.

Graphically:



= minimal cost to get it remotly $C_{u \leftarrow v}$

1 = cache

Notice also the demand element. As soon as demand > 2 the node will be

none will be better off caching.

We will see next in the following an Algorithm to obtain a NE in this setting.

Algorithm 23.7 Nash Equilibrium for Selfish Caching

1: $S = \{\}$ //set of nodes that cache the file
2: **repeat**
3: Let v be a node with maximum demand d_v in set V
4: $S = S \cup \{v\}, V = V \setminus \{v\}$
5: Remove every node u from V with $c_{u \leftarrow v} \leq 1$
6: **until** V = {}

Proof concept that above is NE:

1 = cache , ... = non caching nodes in between.

then it is obvious that (1) has no incentive on changing its caching

no incentive on changing its caching behaviour giving 1, as the decision of ① already in fact considered 1 as caching nodes.

Moreover, ① has no incentive to change its caching behaviour observing 1, as if 1 caches when ① does it means that given the demand · $c_{①,1}$ > 1 so that demand$_①$ · $c_{①,1}$ > demand$_①$ · $c_{①,1}$ by product we demand$_①$ > demand.

It follows that none of these nodes has the incentive in changing their behaviour. □

This proves

Theorem 23.8. *Algorithm 23.7 computes a Nash Equilibrium for Selfish Caching.*

Theorem 23.8. *Algorithm 23.7 computes a Nash Equilibrium for Selfish Caching.*

We finally introduce the following two concepts to see how much the distributed system degrades due to the presence of selfish nodes, in contrast to a social optimum with a benevolent central dictator.

Definition 23.9 (Price of Anarchy). *Let NE_- denote the Nash Equilibrium with the highest cost (smallest payoff). The* ***Price of Anarchy*** *(PoA) is defined as*

$$PoA = \frac{\text{cost}(NE_-)}{\text{cost}(SO)}.$$

Definition 23.10 (Optimistic Price of Anarchy). *Let NE_+ denote the Nash Equilibrium with the smallest cost (highest payoff). The* ***Optimistic Price of Anarchy*** *(OPoA) is defined as*

$$OPoA = \frac{\text{cost}(NE_+)}{\text{cost}(SO)}.$$

We will prove next that in the the worst case the optimistic price of anarchy is $\Theta(n)$.

is $\Theta(n)$.

Theorem 23.11. *The (Optimistic) Price of Anarchy of Selfish Caching can be* $\Theta(n)$.

To see this consider the following network, and assume $\forall$ nodes $v \in V$ $d_v = 1$.

Then:

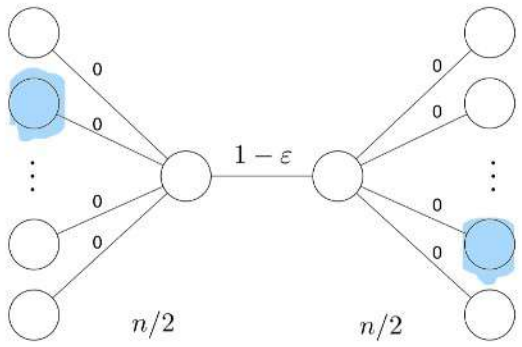


Figure 23.12: A network with a Price of Anarchy of $\Theta(n)$.

If 1 node on left and 1 node on right caches, then all nodes access the file at cost 0 and so $1 = 2$

file at cost 0, and $SO_{cost} = 2$.

This is however not an NE. 1 of the two nodes has the incentive to get the file from the other at cost $1-\varepsilon$.

↳ Once this 1 node change = NE, no one incentive to change.

↳ But there then the entire nodes on the one side will have $1-\varepsilon$ to access the file so that

$$NE_{cost} = 1 + (1-\varepsilon) \cdot \frac{n}{2}$$

and you see

$$\text{Optimal Price of Anarchy} = \frac{1 + (1-\varepsilon)\frac{n}{2}}{2}$$

$$= \frac{1}{2} + \frac{(1-\varepsilon)}{4} n$$

$= 1/2 + (1-\varepsilon) \cdot n \, 1/4$

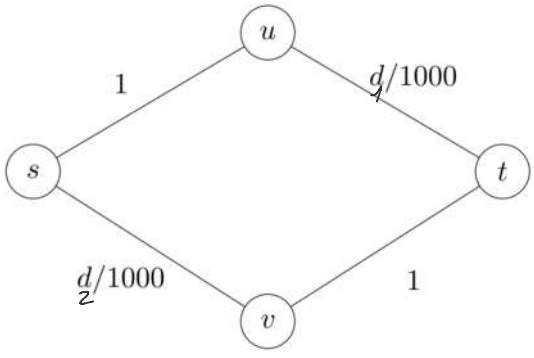
$= \Theta(n)$. □

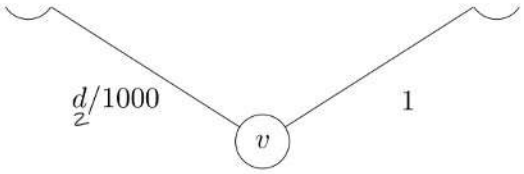
We turn next to an important Paradox in Distributed Systems

23.4 Braess' Paradox

This paradox goes as follows.

Consider the following Network:

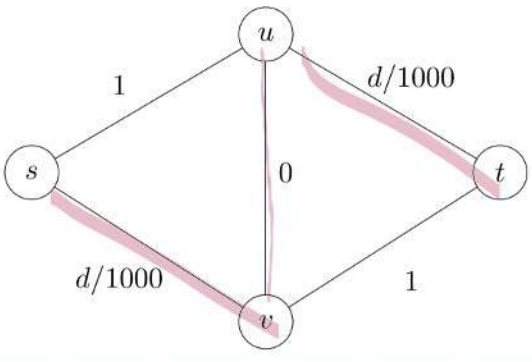




Then immediate to see that NE, $d_1 = 500$ and $d_2 = 500$, and total cost to transfer the file 1,5.

However notice:

Lemma 23.14. *Adding a super fast road (delay is 0) between u and v can increase the travel time from s to t.*



This is the paradox and it is immediate to see that in the above in NE

to see that in the above NE all have the incentive to take the $/1000 road.

You see therefore PoA = $\frac{2}{1,5}$

This concludes our introductory session about game theory. Before concluding and quickly checking at mechanism design, we note on mixed NE.

Consider the following:

v	u	Player u		
		Rock	Paper	Scissors
Player v	Rock	0 0	1 -1	-1 1
	Paper	-1 1	0 0	1 -1
	Scissors	1 -1	-1 1	0 0

Table 23.15: Rock-Paper-Scissors as a matrix.

You see above that there is no Nash Equilibrium in its pure form, i.e. one situation where no participant has the incentive to change its strategy.

↳ When you consider mixed strategies, i.e. mixing the strategies with some probability p_1, p_2, p_3 then you can see that $1/3 = p_1 = p_2 = p_3$ forms a NE.

I.e. there in expectation cost = 0 and no incentive to change.

Theorem 23.16. *Every game has a mixed Nash Equilibrium.*

We conclude by noting:

In distributed systems, players can be servers, routers, etc. Game theory can tell us whether systems and protocols are prone to selfish behavior.

↳ for instance TCP.

23.6 Mechanism Design

Whereas game theory analyzes existing systems, there is a related area that focuses on designing games – mechanism design. The task is to create a game where nodes have an incentive to behave "nicely".

We will show next some mechanism design for setting up an optimal auction.

Definition 23.17 (auction). *One good is sold to a group of bidders in an auction. Each bidder v_i has a secret value z_i for the good and tells his bid b_i to the auctioneer. The auctioneer sells the good to one bidder for a price p.*

Remarks:

- For simplicity, we assume that no two bids are the same, and that $b_1 > b_2 > b_3 > \dots$

Definition 23.19 (truthful). *An auction is truthful if no player v_i can gain anything by not stating the truth, i.e., $b_i = z_i$.*

Our goal for mechanism design will be to

Our goal to mechanism design will be to construct a truthful auction.

Algorithm 23.18 First Price Auction

1: every bidder v_i submits his bid b_i
2: the good is allocated to the highest bidder v_1 for the price $p = b_1$

Theorem 23.20. *A First Price Auction (Algorithm 23.18) is not truthful.*

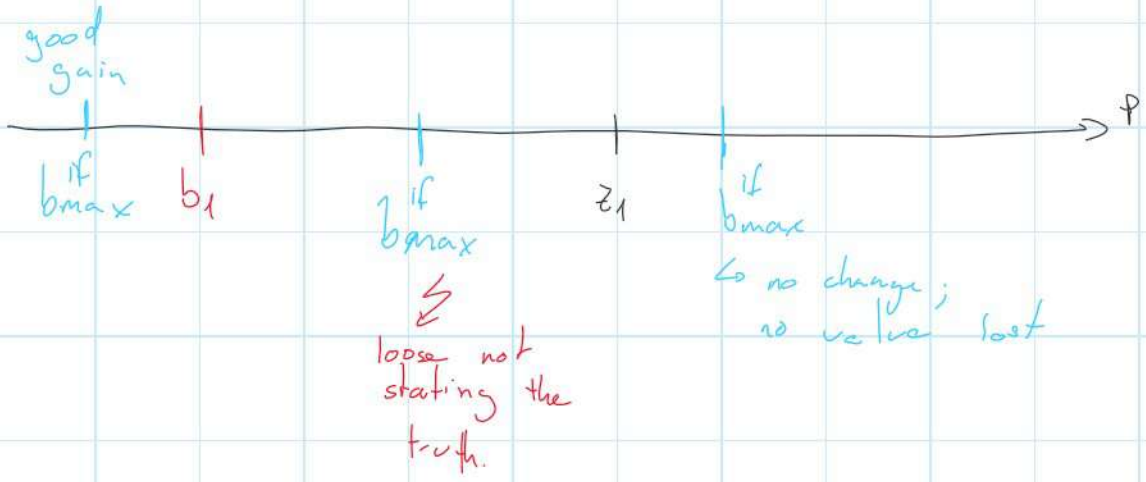
Proof. Consider an auction with two bidders, with bids b_1 and b_2. By not stating the truth and decreasing his bid to $b_1 - \varepsilon > b_2$, player one could pay less and thus gain more. Thus, the first price auction is not truthful. □

Algorithm 23.21 Second Price Auction

1: every bidder v_i submits his bid b_i
2: the good is allocated to the highest bidder v_1 for $p = b_2$

Theorem 23.22. *Truthful bidding is a dominant strategy in a Second Price Auction.*

↳ Proof:



This can then immediately by

This can then immediately be adopted for our selfish caching example.

Remarks:

- Let us use this for Selfish Caching. We need to choose a node that is the first to cache the file. But how? By holding an auction. Every node says for which price it is willing to cache the file. We pay the node with the lowest offer and pay it the second lowest offer to ensure truthful offers.

Another option will be to implement a mechanism design that would help to implement the NE of selfish caching.

Theorem 23.23. *Any Nash Equilibrium of Selfish Caching can be implemented for free.*

Proof. If the mechanism designer wants the nodes from the caching set S of the Nash Equilibrium to cache, then she can offer the following deal to every node not in S: "If any node from set S does not cache the file, then I will ensure a positive payoff for you." Thus, all nodes not in S prefer not to cache since this is a dominant strategy for them. Consider now a node $v \in S$. Since S is a Nash Equilibrium, node v incurs cost of at least 1 if it does not cache the file. For nodes that incur cost of exactly 1, the mechanism designer can even issue a penalty if the node does not cache the file. Thus, every node $v \in S$ caches the file. □

Distributed Storage

06 December 2020 18:30

This section explores distributed storage and techniques that aims to store files over a distributed system in an efficient way, such that it is easy to replicate and retrieve content from such system.

We will start with the most basic option, selfish caching and we will expand then on this idea by looking at different hypercubic networks and their merits for distributed storage.

How do you store 1M movies, each with a size of about 1GB, on 1M nodes, each equipped with a 1TB disk? Simply store the movies on the nodes, arbitrarily, and memorize (with a global index) which movie is stored on which node. What if the set of movies or nodes changes over time, and you do not want to change your global index too often?

Several variants of hashing will do the job, e.g. consistent hashing:

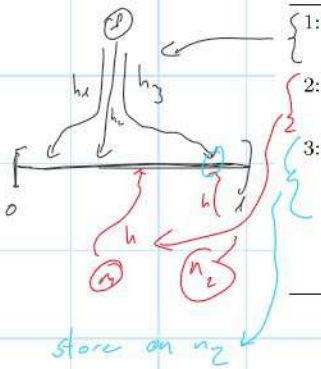
Algorithm 24.1 Consistent Hashing

1: Hash the unique file name of each movie x with a known set of hash functions $h_i(x) \to [0,1)$, for $i = 1, \ldots, k$

2: Hash the unique name (e.g., IP address and port number) of each node with the same hash function $h(u) \to [0,1)$

3: Store a copy of movie x on node u if $h_i(x) \approx h(u)$, for any i. More formally, store movie x on node u if

$$|h_i(x) - h(u)| = \min_v \{|h_i(x) - h(v)|\}, \text{ for any } i$$



Notice that the map $h: \{0,1\}$ can be easily constructed as a hash, i.e. a function...

Theorem 24.2 (Consistent Hashing). *In expectation, each node in Algorithm 24.1 stores km/n movies, where k is the number of hash functions, m the number*

Theorem 24.2 (Consistent Hashing). *In expectation, each node in Algorithm 24.1 stores km/n movies, where k is the number of hash functions, m the number of different movies and n the number of nodes.*

Proof. For a specific movie (out of m) and a specific hash function (out of k), all n nodes have the same probability $1/n$ to hash closest to the movie hash. By linearity of expectation, each node stores km/n movies in expectation if we also count duplicates of movies on a node. □

$\frac{m \cdot k}{n}$ hashes distributed uniformly o.ℓ.

Notice now that due to the probabilistic nature of the above mentioned algorithm, it makes sense to compute the probability of deviating from such expected value. This is important as deviating from it will pose a different storage burden on the nodes.

As seen multiple times in this course this can be computed as follows leveraging the Chernoff bounds:

Facts 24.3. *A version of a* ***Chernoff bound*** *states the following:*
Let $x_1, \ldots, x_n$ be independent Bernoulli-distributed random variables with $Pr[x_i = 1] = p_i$ and $Pr[x_i = 0] = 1 - p_i = q_i$, then for $X := \sum_{i=1}^{n} x_i$ and $\mu := \mathbb{E}[X] = \sum_{i=1}^{n} p_i$ the following holds:

$$\text{for any } \delta > 0\text{: } Pr[X \geq (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

Notice moreover, that slight adaptations of the configuration above exists:

configuration above exists:

- Instead of storing movies directly on nodes as in Algorithm 24.1, we can also store the movies on any nodes we like. The nodes of Algorithm 24.1 then simply store forward pointers to the actual movie locations.

↳ then this will allow you to deterministically choose a proper storage configuration that optimizes your infrastructure and just saves pointers to the storage disks.

↳ Notice however that this is not well suited for high churn systems.

- For better load balancing, we might also hash nodes multiple times.

↳ this reduces the probability that by bad luck all hashes of nodes concentrated in one region, such that just one/a few nodes will have to store a big region of the interval.

We will turn next to the analysis of high churn systems:

churn systems:

- With such a high churn, hundreds or thousands of nodes will change every second. No single node can have an accurate picture of what other nodes are currently in the system. This is remarkably different to classic distributed systems, where a single unavailable node may already be a minor disaster: all the other nodes have to get a consistent view (Definition 25.5) of the system again. In high churn systems it is impossible to have a consistent view at any time.

- Instead, each node will just know about a small subset of 100 or less other nodes ("neighbors"). This way, nodes can withstand high churn situations.

- On the downside, nodes will not directly know which node is responsible for what movie. Instead, a node searching for a movie might have to ask a neighbor node, which in turn will recursively ask another neighbor node, until the correct node storing the movie (or a forward pointer to the movie) is found. The nodes of our distributed storage system form a virtual network, also called an *overlay network*.

↳ this leads us to the exploration of hypercubic networks.

Ideally we want to achieve the following topology properties:

Definition 24.4 (Topology Properties)**.** *Our virtual network should have the following properties:*

- *The network should be (somewhat)* ***homogeneous****: no node should play a dominant role, no node should be a single point of failure.*
- *The nodes should have* ***IDs****, and the IDs should span the universe* $[0, 1)$*, such that we can store data with hashing, as in Algorithm 24.1.*
- *Every node should have a small* ***degree****, if possible polylogarithmic in* n*, the number of nodes. This will allow every node to maintain a persistent connection with each neighbor, which will help us to deal with churn.*
- *The network should have a small* ***diameter****, and routing should be easy. If a node does not have the information about a data item, then it should know which neighbor to ask. Within a few (polylogarithmic in* n*) hops, one should find the node that has the correct information.*

know which neighbor to ask. Within a few (polylogarithmic in n) hops, one should find the node that has the correct information.

So that you see that the bottom two properties are central.

d = degree = how many neighbouring nodes; should not be too large

D = diameter = how long does it take to reach the stored file for a node in the system by going through neighbouring nodes iteratively.

We will check now at different hypercubic networks observing their properties in terms of degree and diameter.

Remarks:

- Some basic network topologies used in practice are trees, rings, grids or tori. Many other suggested networks are simply combinations or derivatives of these.

or torl. Many other suggested networks are simply combinations or derivatives of these.

We will start with tree topologies, however notice that due to the root of the tree being a bottleneck, standard trees are not homogeneous. This leads to the notion of fat trees:

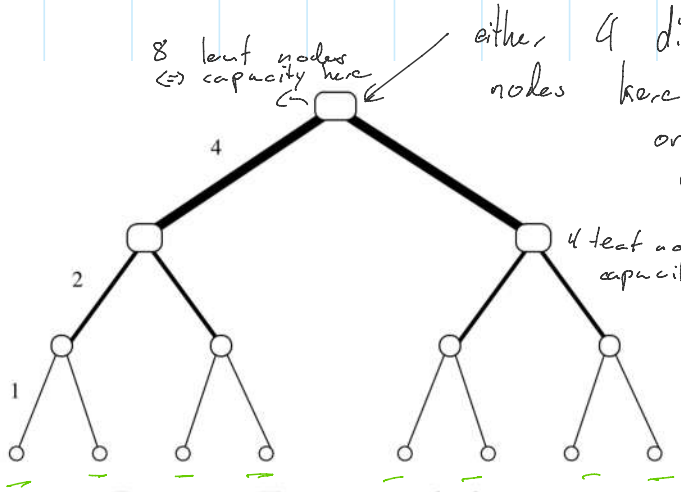
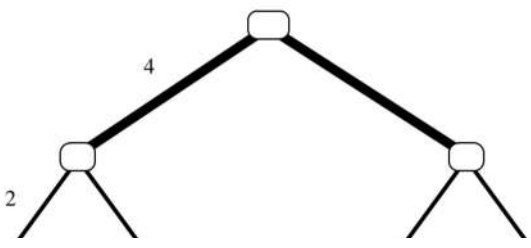


Figure 24.5: The structure of a fat tree.

Notice then that here routing is especially easy and of low order, while degree depends on chosen solution.



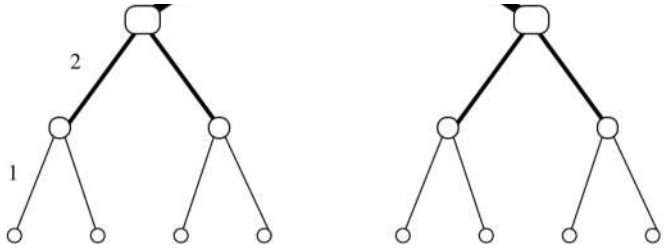


Figure 24.5: The structure of a fat tree.

In particular:

Diameter = max route from one node to the other = 2 · depth

↳ Given now that

$$2^{depth} = \text{leaf nodes}$$

it follows

$$depth = \mathcal{O}(\log(n))$$

For the degree it holds:

if multiple nodes with capacity 1 ⇒

degree = n } see top layer

if few nodes with capacity = leaf nodes

degree = 2.

↳ So bottom line, in order for the fat tree to be efficient you need nodes of non-uniform capacity, i.e. more capacity higher in the tree.

We turn now to networks with uniform capacity exploring torus and meshes:

Unless explicitly mentioned, we will treat all edges in the following to be of capacity 1.

Definition 24.6 (Torus, Mesh). *Let $m, d \in \mathbb{N}$. The (m,d)-**mesh** $M(m,d)$ is a graph with node set $V = [m]^d$ and edge set*

$$E = \left\{ \{(a_1, \ldots, a_d), (b_1, \ldots, b_d)\} \mid a_i, b_i \in [m], \overbrace{\sum_{i=1}^{d} |a_i - b_i| = 1}^{\text{i.e. edge if they just differ in 1 dimension.}} \right\},$$

*where $[m]$ means the set $\{0, \ldots, m-1\}$. The (m,d)-**torus** $T(m,d)$ is a graph that consists of an (m,d)-mesh and additionally wrap-around edges from nodes $(a_1, \ldots, a_{i-1}, m-1, a_{i+1}, \ldots, a_d)$ to nodes $(a_1, \ldots, a_{i-1}, 0, a_{i+1}, \ldots, a_d)$ for all $i \in \{1, \ldots, d\}$ and all $a_j \in [m]$ with $j \neq i$. In other words, we take the expression $a_i - b_i$ in the sum modulo m prior to computing the absolute value. $M(m,1)$ is also called a **path**, $T(m,1)$ a **cycle**, and $M(2,d) = T(2,d)$ a d-**dimensional hypercube**. Figure 24.7 presents a linear array, a torus, and a hypercube.*

$a_i - b_i$ *in the sum modulo* m *prior to computing the absolute value.* $M(m,1)$ *is also called a* ***path****,* $T(m,1)$ *a* ***cycle****, and* $M(2,d) = T(2,d)$ *a* d***-dimensional hypercube****. Figure 24.7 presents a linear array, a torus, and a hypercube.*

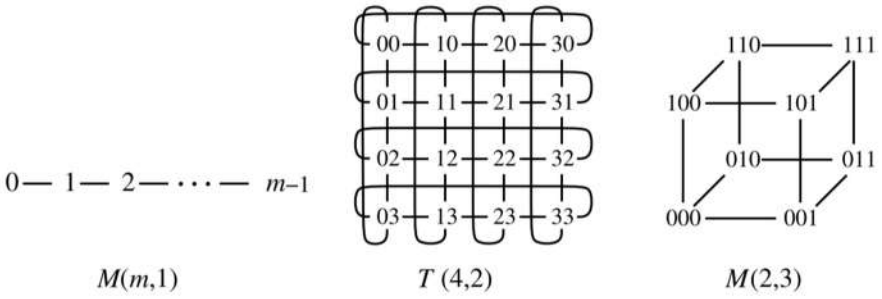
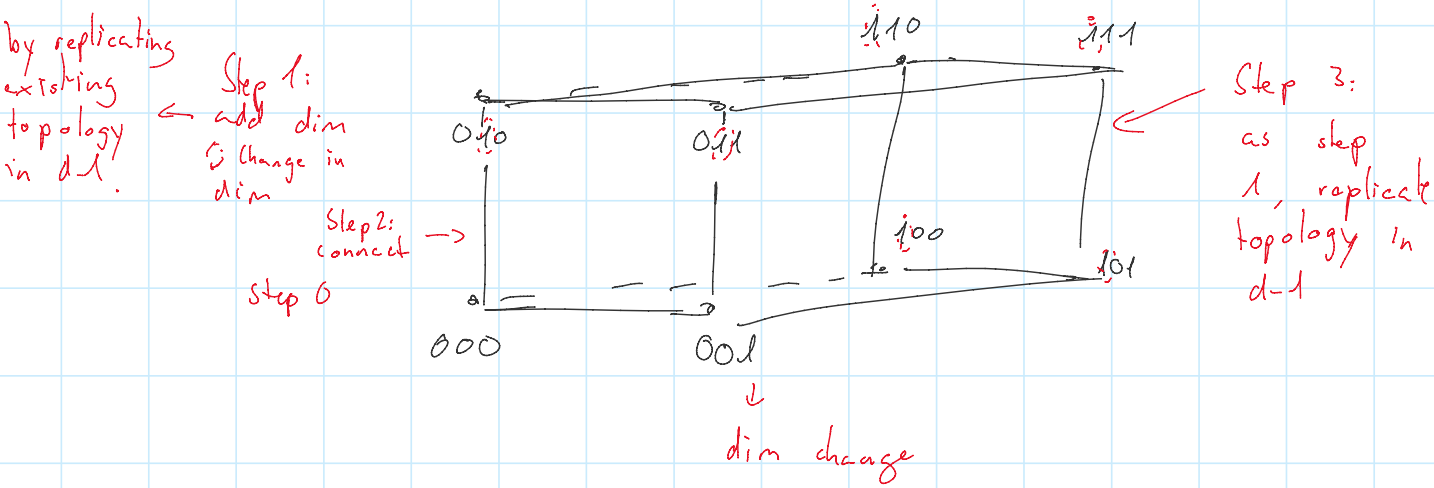


Figure 24.7: The structure of $M(m,1)$, $T(4,2)$, and $M(2,3)$.

Notice that the definition in mathematical terms is particularly challenging to remember.

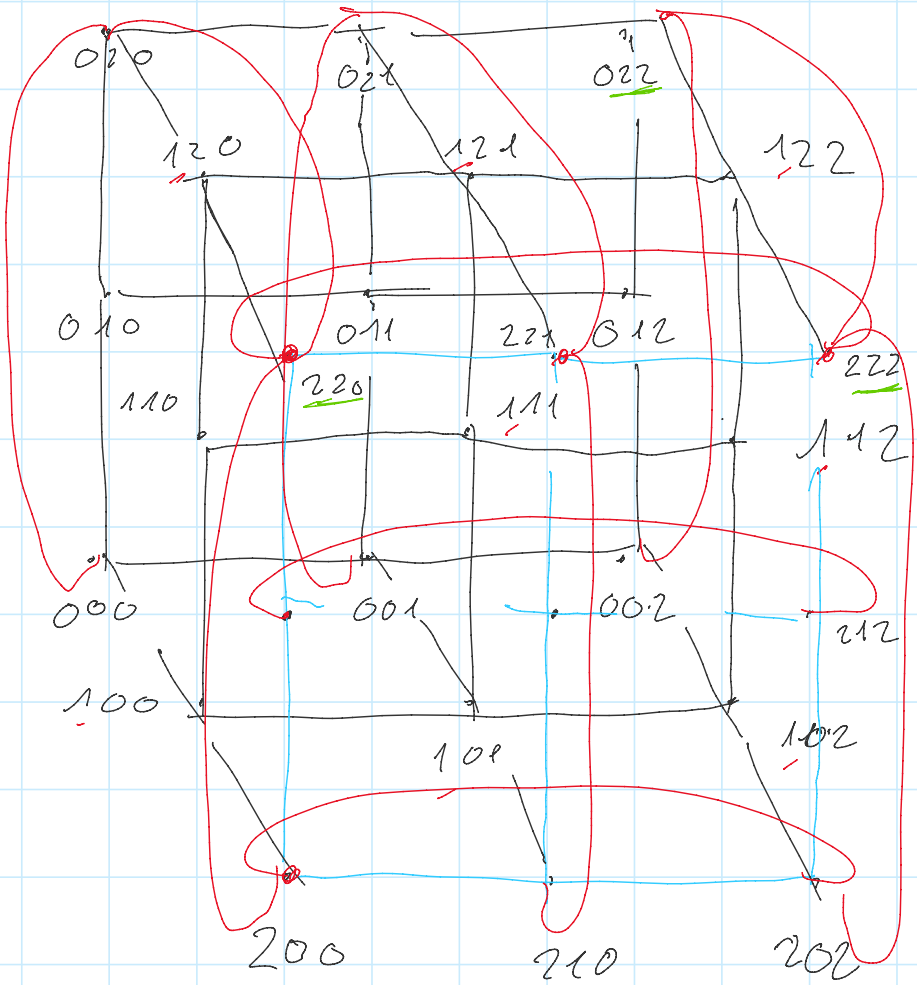
Simply understand that for a mesh in 2 = values = m and d = 3 = dimensions you construct it as follows:



Notice now that by definition each node is connected with # d = dimension nodes.

Notice now that by definition each node is connected with $\pm d$ = dimension nodes.

Torus definition is clear, consider this for $T(3,3)$, then first construct mesh then add wrappers.



C = taurus connections. from m-1 to 0 in all dim.

↳ for instance

222 ↓ connection 202 tau.

222 ↓ 022 and

222 ↓ 220

So in taurus you add for each m and d

for each m and q

1 if $m > 2$ · d

else 0

connections

- Routing on a mesh, torus, or hypercube is trivial. On a d-dimensional hypercube, to get from a source bitstring s to a target bitstring t one only needs to fix each "wrong" bit, one at a time; in other words, if the source and the target differ by k bits, there are $k!$ routes with k hops.
- As required by Definition 24.4, the d-bit IDs of the nodes need to be mapped to the universe $[0, 1)$. One way to do this is by interpreting an ID as the binary representation of the fractional part of a decimal number. For example, the ID **101** is mapped to $\mathbf{0.101_2}$ which has a decimal value of $\mathbf{0} \cdot 2^0 + \mathbf{1} \cdot 2^{-1} + \mathbf{0} \cdot 2^{-2} + \mathbf{1} \cdot 2^{-3} = \frac{5}{8}$.

Notice now,

degree Mesh = dimensions

Diameter Mesh = max d · m-1

max change d · m-1 bits

noticing how the dependence of

noticing how the dependence of dimension on nodes assuming a $m=2$ if follows

$$2^d = \text{nodes}, \quad \text{so that}$$

$$d = O(\log(n)) \quad \text{as well}$$

the diameter,

$$D = O(\log(n))$$

if $m > 2$ poly logarithmic degree easy to see from the above.

Notice how effectively the torus exchanges degree costs for Diameter costs, i.e you will decrease the diameter increasing the degree.

We note now that:

We note now that:

- The hypercube has many derivatives, the so-called *hypercubic networks*. Among these are the butterfly, cube-connected-cycles, shuffle-exchange, and de Bruijn graph. We start with the butterfly, which is basically a "rolled out" hypercube.

We will explore these next

Butterfly:

Definition 24.8 (Butterfly)**.** *Let $d \in \mathbb{N}$. The d-**dimensional butterfly** $BF(d)$ is a graph with node set $V = [d+1] \times [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$$E_1 = \{\{(i,\alpha),(i+1,\alpha)\} \mid i \in [d],\ \alpha \in [2]^d\}$$

fix one comb i.e. say 100

connect two adjacent dimensions

and

$$E_2 = \{\{(i,\alpha),(i+1,\beta)\} \mid i \in [d],\ \alpha,\beta \in [2]^d, \alpha \oplus \beta = 2^i\}.$$

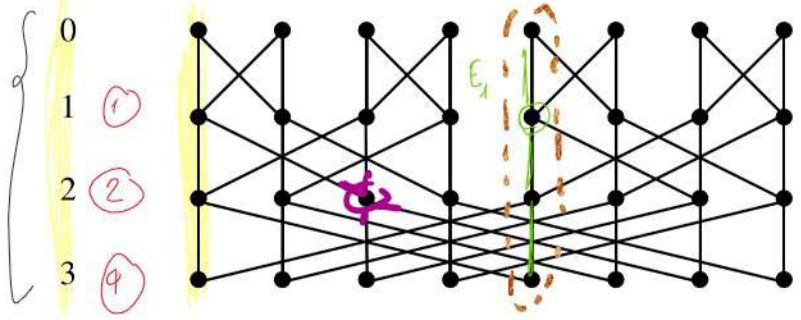
means that the difference between the two must be 2^i when converted into decimal numbers.

*A node set $\{(i,\alpha) \mid \alpha \in [2]^d\}$ is said to form **level** i of the butterfly. The d-**dimensional wrap-around butterfly** $W\text{-}BF(d)$ is defined by taking the $BF(d)$ and having $(d,\alpha) = (0,\alpha)$ for all $\alpha \in [2]^d$.*

Understand it again graphically:

$2^3 = 8$

000 001 010 011 100 101 110 111



d+1

i = fix

2^d

i+1

i

Figure 24.9: The structure of BF(3).

Figure 24.9: The structure of BF(3).

$2^2 = 4 = 2^1$ for $i=2$ etc..

i → 1 → E_1

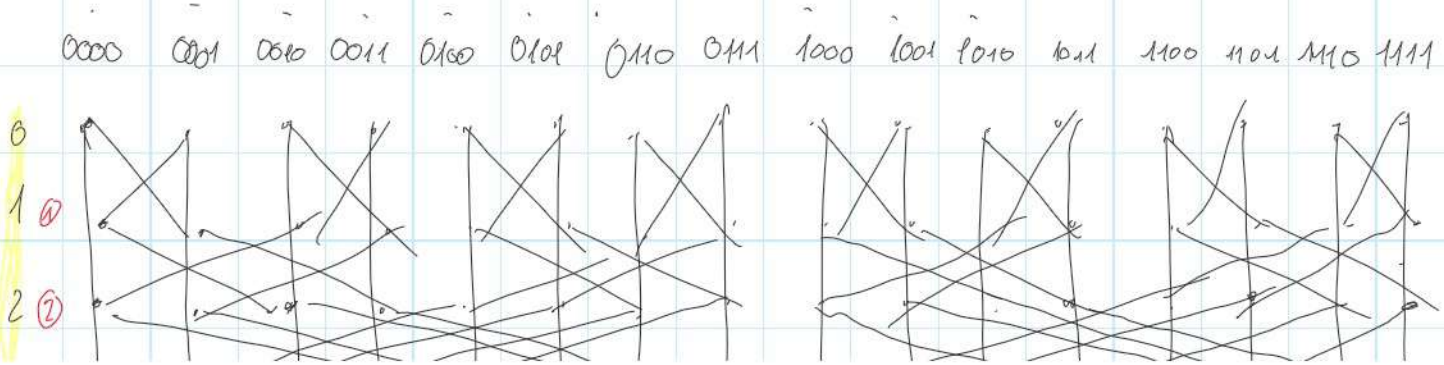
So # nodes = # · #

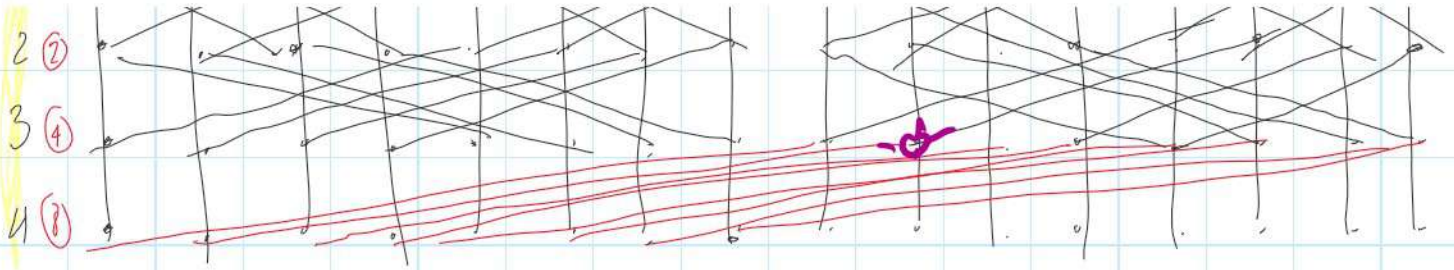
Notice that ultimately for E_2 connect in dimension $i+1$ by changing for column of interest $i+1$ bit. and keeping all other the same.

- Figure 24.9 shows the 3-dimensional butterfly $BF(3)$. The $BF(d)$ has $(d+1)2^d$ nodes, $2d \cdot 2^d$ edges and degree 4. It is not difficult to check that combining the node sets $\{(i, \alpha) \mid i \in [d]\}$ for all $\alpha \in [2]^d$ into a single node results in the hypercube.

degree = 4 see node

Notice that this not increases increasing the butterfly





So degree always 4.

Notice that in butterfly at each jump can change 1 leading bit and dim you are in bit.

↳ possible to see then that to change 1 bit possible worst number of moves ⇒ moves to dim which number of bit you want to change + 1.

so $(d+1-1)$ [worst] $+1 = d+1$, due to the short cuts I guess this should also be max in general.

It follows $\quad dim + 1 \cdot 2^{dim} = n$

$$\log(dim) \cdot dim = \log(n)$$

∴ possible to show $dim = \log(n)$, so

.. possible to show dim = log(n), so that Diameter = dim = dimension = log(n).

- Butterflies have the advantage of a constant node degree over hypercubes, whereas hypercubes feature more fault-tolerant routing.

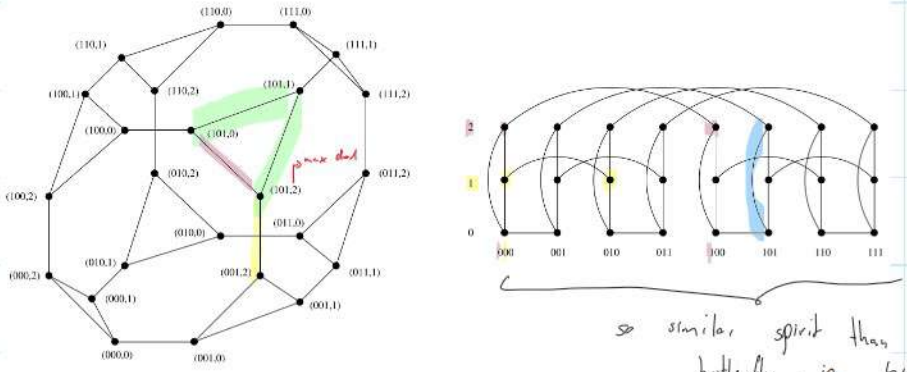
} as k! possible routs in hypercube.

while here just 4 links death is sufficient for network partition.

- Next we define the cube-connected-cycles network. It only has a degree of 3 and it results from the hypercube by replacing the corners by cycles.

Definition 24.10 (Cube-Connected-Cycles). *Let $d \in \mathbb{N}$. The **cube-connected-cycles** network CCC(d) is a graph with node set $V = \{(a,p) \mid a \in [2]^d, p \in [d]\}$ and edge set*

$$E = \{\{(a,p),(a,(p+1) \bmod d)\} \mid a \in [2]^d, p \in [d]\}$$
$$\cup \{\{(a,p),(b,p)\} \mid a,b \in [2]^d, p \in [d], |a-b| = 2^p\}$$



so similar spirit than butterfly, in butterfly however 1 missing.

Figure 24.11: The structure of CCC(3).

Definition 24.12 (Shuffle-Exchange). *Let $d \in \mathbb{N}$. The **d-dimensional shuffle-exchange** $SE(d)$ is defined as an undirected graph with node set $V = [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

Definition 24.12 (Shuffle-Exchange). *Let $d \in \mathbb{N}$. The **d-dimensional shuffle-exchange** $SE(d)$ is defined as an undirected graph with node set $V = [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$$E_1 = \{\{(a_1, \ldots, a_d), (a_1, \ldots, \bar{a}_d)\} \mid (a_1, \ldots, a_d) \in [2]^d,\ \bar{a}_d = 1 - a_d\}$$

and

$$E_2 = \{\{(a_1, \ldots, a_d), (a_d, a_1, \ldots, a_{d-1})\} \mid (a_1, \ldots, a_d) \in [2]^d\}\ .$$

Figure 24.13 shows the 3- and 4-dimensional shuffle-exchange graph.

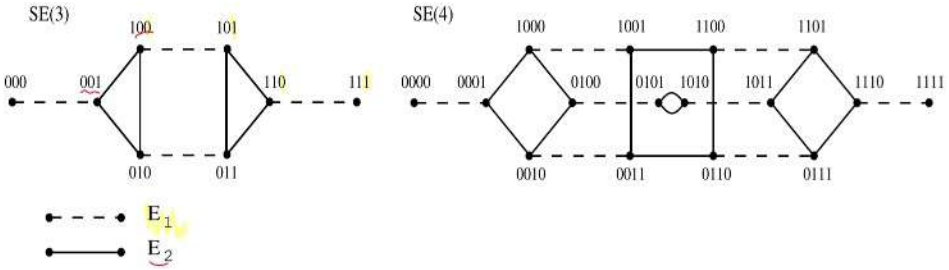


Figure 24.13: The structure of SE(3) and SE(4).

Another of such hypercubic structures is the de bruijn graph

Definition 24.14 (DeBruijn). *The b-**ary DeBruijn graph of dimension** d $DB(b, d)$ is an undirected graph $G = (V, E)$ with node set $V = \{v \in [b]^d\}$*

and edge set E that contains all edges $\{v, w\}$ with the property that $w \in \{(x, v_1, \ldots, v_{d-1}) : x \in [b]\}$, where $v = (v_1, \ldots, v_d)$.

should not be ordered as such just containing.

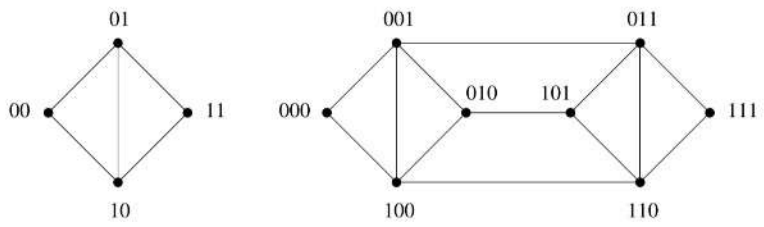


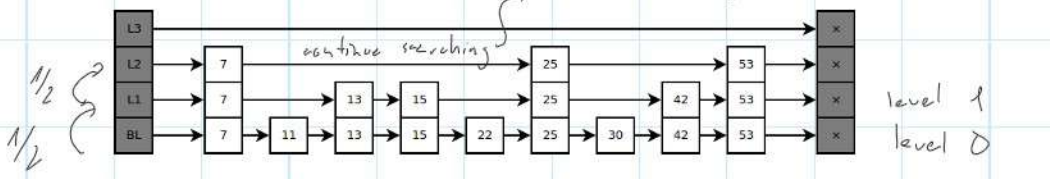
Figure 24.15: The structure of $DB(2, 2)$ and $DB(2, 3)$.

So notice no circular link here, as well as $\{0\}^d$ and $\{1\}^d$ disconnect.

Figure 24.15: The structure of $DB(2,2)$ and $DB(2,3)$.

- There are some data structures which also qualify as hypercubic networks. An example of a hypercubic network is the skip list, the balanced binary search tree for the lazy programmer:

Definition 24.16 (Skip List)**.** *The skip list is an ordinary ordered linked list of objects, augmented with additional forward links. The ordinary linked list is the level 0 of the skip list. In addition, every object is promoted to level 1 with probability 1/2. As for level 0, all level 1 objects are connected by a linked list. In general, every object on level i is promoted to the next level with probability 1/2. A special start-object points to the smallest/first object on each level.*



Remarks:

- Search, insert, and delete can be implemented in $\mathcal{O}(\log n)$ expected time in a skip list, simply by jumping from higher levels to lower ones when overshooting the searched position. Also, the amortized memory cost of each object is constant, as on average an object only has two forward links.
- The randomization can easily be discarded, by deterministically promoting a constant fraction of objects of level i to level $i+1$, for all

  i. When inserting or deleting, object o simply checks whether its left and right level i neighbors are being promoted to level $i+1$. If none of them is, promote object o itself. Essentially we establish a maximal independent set (MIS) on each level, hence at least every third and at most every second object is promoted.

We will now show that no matter the hypercubic network data structure you choose for your storage, the following holds:

Theorem 24.17. *Every graph of maximum degree $d > 2$ and size n must have a diameter of at least $\lceil (\log n)/(\log(d-1)) \rceil - 2$.*

Proof. Suppose we have a graph $G = (V, E)$ of maximum degree d and size n. Start from any node $v \in V$. In a first step at most d other nodes can be reached. In two steps at most $d \cdot (d-1)$ additional nodes can be reached. Thus, in general, in at most r steps at most

v-itself count it as reachable.

geom series formula with Erweiterung $\frac{-1}{-1}$

$$1 + \sum_{i=0}^{r-1} d \cdot (d-1)^i = 1 + d \cdot \frac{(d-1)^r - 1}{(d-1) - 1} \leq \frac{d \cdot (d-1)^r}{d-2}$$

nodes (including v) can be reached. This has to be at least n to ensure that v can reach all other nodes in V within r steps. Hence,

$$(d-1)^r \geq \frac{(d-2) \cdot n}{d} \quad \Leftrightarrow \quad r \geq \log_{d-1}((d-2) \cdot n/d) .$$

Since $\log_{d-1}((d-2)/d) > -2$ for all $d > 2$, this is true only if $r \geq \lceil (\log n)/(\log(d-1)) \rceil - 2$. □

$> \frac{\log(n)}{\log(d)}$

- In other words, constant-degree hypercubic networks feature an asymptotically optimal diameter D.

↳ As we saw for instance butterfly degree = 4 und diameter = $\log(n)$.

- There are a few other interesting graph classes which are not hypercubic networks, but nevertheless seem to relate to the properties of Definition 24.4. Small-world graphs (a popular representations for social networks) also have small diameter, however, in contrast to hypercubic networks, they are not homogeneous and feature nodes with large degrees.

- Expander graphs (an expander graph is a sparse graph which has

large degrees.

- Expander graphs (an expander graph is a sparse graph which has good connectivity properties, that is, from every not too large subset of nodes you are connected to an even larger set of nodes) are homogeneous, have a low degree and small diameter. However, expanders are often not routable.

We conclude this section by generally streching the idea of creating a distributed hash table from such networks.

This is a rather simple task.

Definition 24.18 (Distributed Hash Table (DHT)). *A **distributed hash table** (DHT) is a distributed data structure that implements a distributed storage. A DHT should support at least (i) a search (for a key) and (ii) an insert (key, object) operation, possibly also (iii) a delete (key) operation.*

It follows immediately that:

- A DHT can be implemented as a hypercubic overlay network with nodes having identifiers such that they span the ID space $[0, 1)$.
- A hypercube can directly be used for a DHT. Just use a globally known set of hash functions h_i, mapping movies to bit strings with d bits.
- Other hypercubic structures may be a bit more intricate when using it as a DHT: The butterfly network, for instance, may directly use the $d+1$ layers for replication, i.e., all the $d+1$ nodes are responsible for the same ID.

! then store at the bitstring 2^d in the hypercube

- A hypercube can directly be used for a DHT: Just use a globally known set of hash functions h_i, mapping movies to bit strings with d bits.
- Other hypercubic structures may be a bit more intricate when using it as a DHT: The butterfly network, for instance, may directly use the $d+1$ layers for replication, i.e., all the $d+1$ nodes are responsible for the same ID.

then store at the bitstring 2^d in the hypercube

We now turn to a more interesting question, the one of high churn distributed hash table

- Many DHTs in the literature are analyzed against an adversary that can crash a fraction of random nodes. After crashing a few nodes the system is given sufficient time to recover again. However, this seems unrealistic. The scheme sketched in this section significantly differs from this in two major aspects.

- First, we assume that joins and leaves occur in a worst-case manner. We think of an adversary that can remove and add a bounded number of nodes; the adversary can choose which nodes to crash and how nodes join.
- Second, the adversary does not have to wait until the system is recovered before it crashes the next batch of nodes. Instead, the adversary can constantly crash nodes, while the system is trying to stay alive. Indeed, the system is *never fully repaired* but *always fully functional*. In particular, the system is resilient against an adversary that continuously attacks the "weakest part" of the system. The adversary could for example insert a crawler into the DHT, learn the topology of the system, and then repeatedly crash selected nodes, in an attempt to partition the DHT. The system counters such an adversary by continuously moving the remaining or newly joining nodes towards the areas under attack.

- Clearly, we cannot allow the adversary to have unbounded capabilities. In particular, in any constant time interval, the adversary can at most add and/or remove $O(\log n)$ nodes, n being the total number of nodes currently in the system. This model covers an adversary

at most add and/or remove $O(\log n)$ nodes, n being the total number of nodes currently in the system. This model covers an adversary which repeatedly takes down nodes by a distributed denial of service attack, however only a logarithmic number of nodes at each point in time. The algorithm relies on messages being delivered timely, in at most constant time between any pair of operational nodes, i.e., the synchronous model. Using the trivial synchronizer this is not a problem. We only need bounded message delays in order to have a notion of time which is needed for the adversarial model. The duration of a round is then proportional to the propagation delay of the slowest message.

Algorithm 24.19 DHT

1: Given: a globally known set of hash functions h_i, and a hypercube (or any other hypercubic network)
2: Each hypercube virtual node ("hypernode") consists of $\Theta(\log n)$ nodes.
3: Nodes have connections to all other nodes of their hypernode and to nodes of their neighboring hypernodes.
4: Because of churn, some of the nodes have to change to another hypernode such that up to constant factors, all hypernodes own the same number of nodes at all times.
5: If the total number of nodes n grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively.

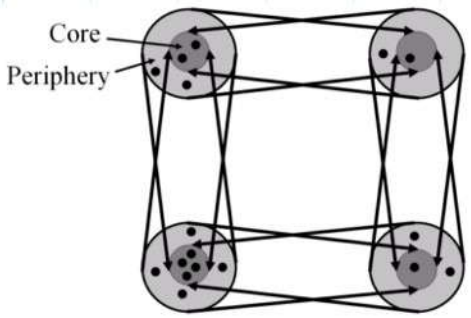


Figure 24.20: A simulated 2-dimensional hypercube with four hypernodes, each consisting of several nodes. Also, all the nodes are either in the core or in the periphery of a node. All nodes within the same hypernode are completely connected to each other, and additionally, all nodes of a hypernode are connected to the core nodes of the neighboring nodes. Only the core nodes store data items, while the peripheral nodes move between the nodes to balance biased adversarial churn.

churn.

- In summary, the storage system builds on two basic components: (i) an algorithm which performs the described dynamic token distribution and (ii) an information aggregation algorithm which is used to estimate the number of nodes in the system and to adapt the dimension of the hypercube accordingly:

Theorem 24.21 (DHT with Churn). *We have a fully scalable, efficient distributed storage system which tolerates* $O(\log n)$ *worst-case joins and/or crashes per constant time interval. As in other storage systems, nodes have* $O(\log n)$ *overlay neighbors, and the usual operations (e.g., search, insert) take time* $O(\log n)$.

Remarks:

- Indeed, handling churn is only a minimal requirement to make a distributed storage system work. Advanced studies proposed more elaborate architectures which can also handle other security issues, e.g., privacy or Byzantine attacks.

Advanced Block chain

12 December 2020 14:28

This module reasons about some key issues with the proposed bitcoin architecture.

It moreover introduce the core ideas of Ethereum, a DAG-based block chain being turing-complete and avoiding keeping stuck into while loops through the idea of Gas.

We will start by looking at selfish mining. This goes hand in hand with game theory and mechanism design that we previously see.

We will show that under the Bitcoin protocol mechanism following Nakatomo's idea of always publishing a block is not always the most rational behaviour for a miner.

In fact we will inspect the conditions under which it might well make sense for the miner to deviate from such behaviour.

Selfish Mining

Definition 26.1 (Selfish Mining). *A selfish miner hopes to earn the reward of a*

Definition 26.1 (Selfish Mining). *A selfish miner hopes to earn the reward of a larger share of blocks than its hardware would allow. The selfish miner achieves this by temporarily keeping newly found blocks secret.*

So the idea would be to apply a mechanism similar to the following.

Algorithm 26.2 Selfish Mining

```
Idea: Mine secretly, without immediately publishing newly found blocks
Let d_p be the depth of the public blockchain
Let d_s be the depth of the secretly mined blockchain
if a new block b_p is published, i.e., d_p has increased by 1 then
  if d_p > d_s then
    Start mining on that newly published block b_p
  else if d_p = d_s then
    Publish secretly mined block b_s
    Mine on b_s and publish newly found block immediately
  else if d_p = d_s − 1 then
    Publish both secretly mined blocks
  end if
end if
```

new block is published on public blockchain.

i.e. switch, no sense to mine on a short blockchain.

here you were 1 block in advance before. Means that now two blockchains of equal length ⇒ RACE

if now just 1 block in advance on secret blockchain, do not risk of getting caught up and starting a Race i.e. publish immediately all of your secret blocks and get the reward for it

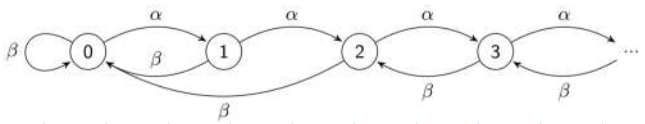
We will now see that given such behaviour you might well be better off than simply always immediately publishing mined blocks.

Theorem 26.3 (Selfish Mining). *It may be rational to mine selfishly, depending on two parameters α and γ, where α is the ratio of the mining power of the selfish miner, and γ is the share of the altruistic mining power the selfish miner can reach in the network if the selfish miner publishes a block right after seeing a newly published block. Precisely, the selfish miner share is*

$$\frac{\alpha(1-\alpha)^2(4\alpha+\gamma(1-2\alpha))-\alpha^3}{1-\alpha(1+(2-\alpha)\alpha)}.$$

In order to see this consider the following Markov-Chain representing the evolution of the secret

following Markov-chain representing the evolution of the secret chain d_s vs the public chain d_p.



Then it is easy to understand given basic Markov chain theory that a steady state distribution exists and that LLN applies so that we will converge to it.

Precisely, it is then possible to see that we can compute the steady state distribution by solving the following equations:

$$p_1 = \alpha p_0$$

assuming reversibility of the chain this must hold.

$$p_i \cdot \alpha = p_{i+1} \cdot \beta \qquad \forall\, i > 1$$

$$\sum_i p_i = 1$$

So that it follows with $p := \frac{\alpha}{\beta}$

$$1 = \frac{p_1}{\alpha} + p_1 \sum_{i \geq 0} p^i$$

$$\sim p_0$$

So that it follows with $\rho := \frac{\alpha}{\beta}$

$$1 = \frac{p_1}{\alpha} + p_1 \sum_{i \geq 0} \rho^i$$

$\sim$

p_0

(My question: how come it works for p_2?)

Then it is possible to see:

Using $\rho = \alpha/\beta$, we express all terms of above sum with p_1:

$$1 = \frac{p_1}{\alpha} + p_1 \sum_{i \geq 0} \rho^i = \frac{p_1}{\alpha} + \frac{p_1}{1-\rho}, \text{ hence } p_1 = \frac{2\alpha^2 - \alpha}{\alpha^2 + \alpha - 1}.$$

So that you can now express all of the probabilities in terms of p_1, i.e. α.

Given such computed steady state probabilities it is now possible to compute the share of mined blocks by the selfish miner.

It is clear that with the behaviour described above in the chain the expected number of mined blocks is:

$$1 + p_1 + p_2$$

$$1 + p_1 + p_2$$

This because at all of the states 1 block is released but in state 1 and 2.

↳ In state 1, 2 blocks are mined (1 of well behaving, 1 of selfish miners starting a race).

↳ In state 2, 2 blocks are appended by the selfish miner.

Important is to see that the share of the mined blocks of these appended by the selfish miner, is given by

$$1 - \text{non-selfish, i.e.}$$

↳ assume all selfish

$$1 - p_0 - p_1 \cdot (1-\gamma) \cdot \beta$$

at p_1 when good node win the race and append 2 blocks

$$+ a\, p_1$$

α p_1

at p_1 when selfish wins the race

So that total share of selfish miner

$$\frac{1 - p_0 - p_1(1-\gamma)\beta + \alpha p_1}{1 + p_1 + p_2} \qquad (I)$$

And when

$$(I) > \alpha$$

then it is convenient for the miner to be selfish.

Doing some analysis, it is possible to see that the inequality heavily depends on the parameter γ.

↳ Recall that this expresses how well the node is positioned in the network and how well it can propagate its block into the network if it has to start a RACE

network if it has to start a RACE.

Remarks:

- If the miner is honest (altruistic), then a miner with computational share α should expect to find an α fraction of the blocks. For some values of α and γ the ratio of Theorem 26.3 is higher than α.
- In particular, if $\gamma = 0$ (the selfish miner only wins a race in Line 8 if it manages to mine 2 blocks in a row), the break even of selfish mining happens at $\alpha = 1/3$.
- If $\gamma = 1/2$ (the selfish miner learns about honest blocks very quickly and manages to convince half of the honest miners to mine on the selfish block instead of the slightly earlier published honest block), already $\alpha = 1/4$ is enough to have a higher share in expectation.
- And if $\gamma = 1$ (the selfish miner controls the network, and can hide any honest block until the selfish block is published) any $\alpha > 0$ justifies selfish mining.

We turn now to DAG-based Blockchains and we will introduce the key ideas of Ethereum next.

DAG Blockchains

Traditional Bitcoin-like blockchains require mining blocks sequentially. Sometimes effort is wasted if two blocks happen to be mined at roughly the same time, as one of these two blocks is going to become obsolete. DAG-blockchains (where DAG stands for directed acyclic graph) try to prevent such wasted blocks. They allow for faster block production, as forks are less of a problem.

↳ Idea you do not revert the entire block to the longest chain, you just make a conflict resolution for the transactions that conflict in the DAG accepting the ones in the block with the heavier weight for the tree.

Definition 26.5 (DAG-blockchain). *In a DAG-blockchain the genesis block does not reference other blocks. Every other block has at least one (and possibly*

↳ primal by

Definition 26.5 (DAG-blockchain). *In a DAG-blockchain the genesis block does not reference other blocks. Every other block has at least one (and possibly multiple references) to previous blocks.*

trivial by DAG

Definition 26.6 (DAG-Relations). *Block p is a dag-parent of block b if block b references (includes a hash) to p. Likewise b is a dag-child of p. Block a is a dag-ancestor of block b, if a is b's dag-parent, dag-grandparent (dag-parent of dag-parent), dag-grandgrandparent, and so on. Likewise b is a's dag-descendant.*

Straightforward:

Theorem 26.7. *There are no cycles in a DAG-blockchain.*

Proof. A block b includes its dag-parents' hashes. These dag-parents themselves include the hashes of their dag-parents, etc. To get a cycle of references, some of b's dag-ancestors must include b's hash, which is cryptographically infeasible. □

Definition 26.8 (Tree-Relations). *We are going to implicitly mark some of the references in the DAG of blocks, such that these marked references form a tree, directed towards the genesis block. For every non-genesis block one edge to one of its dag-parents is marked. We use the prefix "tree" to denote these special relations. The marked edge is between tree-parent and tree-child. The tree also defines tree-ancestors and tree-descendants.*

↳ ie. tree = 1 path between any two nodes, and is ⊆ Dag.

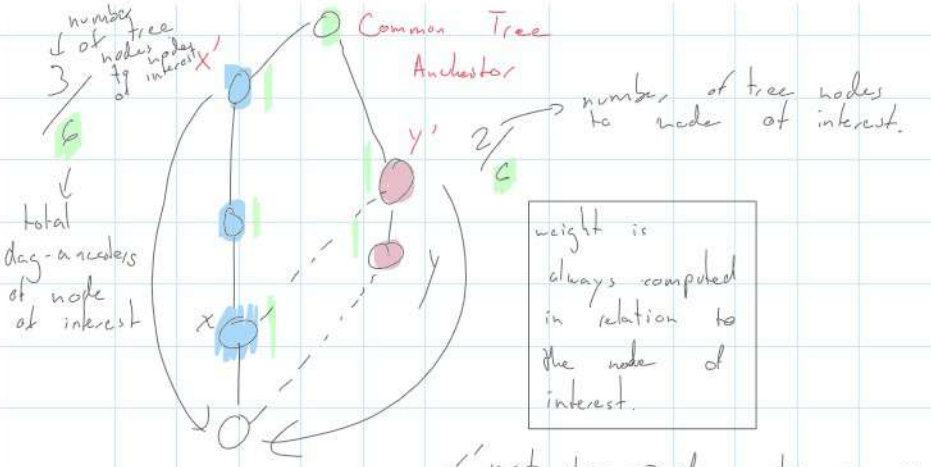
dag: more than 1 path possible

To specify which edge we mark we use the following key concepts:

Definition 26.9 (DAG Weight). *The weight of a dag-ancestor block a with respect to a block b is defined as the number of tree-descendants of a in the set of dag-ancestors of b. If two blocks a and a' have the same weight, we use the hashes of a and a' to break ties.*

Definition 26.10 (Parent Order). *Let x and y be any pair of dag-parents of b, and z be the lowest common tree-ancestor of x and y. x' and y' are the tree-children of z that are tree-ancestors of x and y respectively. If x' has a higher weight than y', then block b orders dag-parent x before y.*

Definition 26.11 (Tree-Parent). *The tree-parent of b is the first dag-parent in b's parent order.*



not tree ⇒ due to comp.

⇒ So that left has higher weight and comes sooner.

Theorem 26.13. *Let p be the tree-parent of b. The order of blocks $<_b$ computed by Algorithm 26.12 extends the order $<_p$ by appending some blocks.*

Proof. Block p is the first dag-parent of b, so in the first iteration of the loop, we have $<_b = <_p$. Further modifications of $<_b$ consist only of appending more blocks to $<_b$, ending with block b itself. □

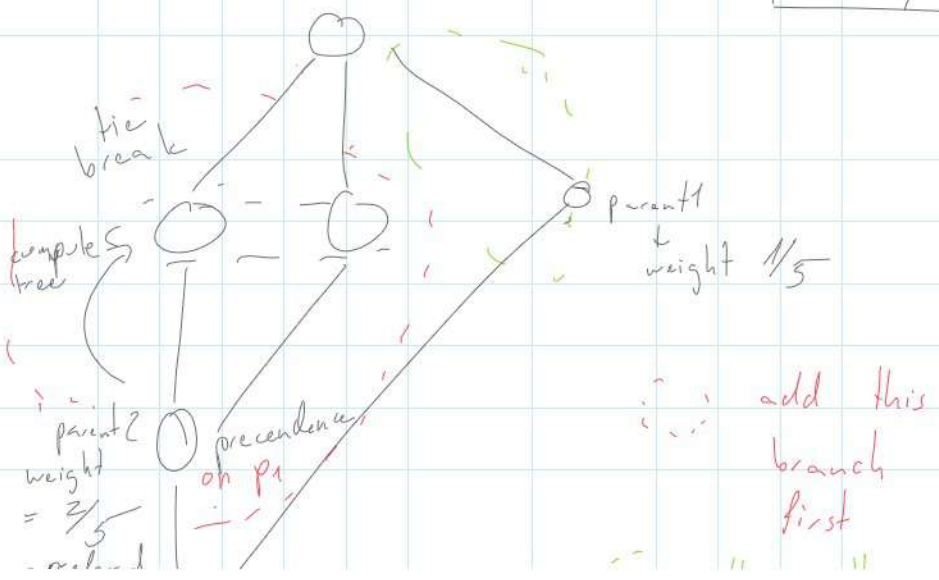
↳ i.e. block p in this case, second parent.

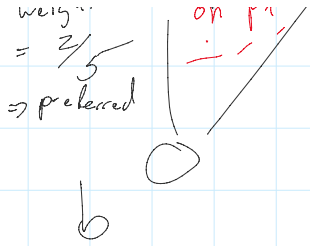
Algorithm 26.12 DAG-Blockchain Ordering

1: We totally order all dag-ancestors of block b as $<_b$ as follows:
2: Initialize $<_b$ as empty
3: **for** all dag-parents p of b, in their parent order **do**
4: Compute $<_p$ (recursively)
5: Remove from $<_p$ any blocks already included in $<_b$
6: Append $<_p$ at the end of $<_b$
7: **end for**
8: Append block b at the end of $<_b$

have to compute this first.
↳ then once it is clear you can compute $<_p$ of the parents in their order recursively

i.e. as done previously





first

then this branch

Note the following:

- Note that b is appended to the order only after ordering all its dag-ancestors. The genesis block is the only block where the recursion will stop, so the genesis block is always first in the total order.
- By Theorem 26.13 tree-children extend the order of their tree-parent, so appending blocks to the DAG preserves the previous order and new blocks are appended at the end.

Definition 26.14 (Transaction Order). *Transactions in each block are ordered by the miner of the block. Since blocks themselves are ordered, all transactions are ordered. If two transactions contradict each other (e.g. they try to spend the same money twice), the first transaction in the total order is considered executed, while the second transaction is simply ignored (or possibly punished).*

So this basically proves the thing mentioned at the beginning of the section, simple conflict resolution mechanism keeping all of the transactions that do not conflict

Remarks:

- Ethereum allows blocks to not only have a parent, but also up to two "uncles" (childless blocks). In contrast to above description, blocks must specify the main parent.
- In Ethereum, new blocks are mined approximately every 15 seconds (as opposed to 10 minutes in Bitcoin). New blocks being generated in such rapid succession leads to a lot of childless blocks. Uncles have been introduced to not "waste" those blocks.
- In Ethereum, the original uncle-miners get 7/8 of the block reward. The miner who references these uncle blocks also gets a small reward. This reward depends on the height-difference of the uncle and the included parent. Also, to be included, the uncle and the current block should have a common ancestor not too far in the past.

incentive to include such leftovers uncles.

Penalization to go against current of longest chain to, adding an "uncle" block so-to say.

an "uncle" block so-to say

Smart Contracts

Definition 26.15 (Ethereum). *Ethereum is a distributed state machine. Unlike Bitcoin, Ethereum promises to run arbitrary computer programs in a blockchain.*

↳ i.e. Turing complete

Definition 26.16 (Smart Contract). *Smart contracts are programs deployed on the Ethereum blockchain that have associated storage and can execute arbitrarily complex logic.*

Remarks:

- Smart Contracts are written in higher level programming languages like Solidity, Vyper, etc. and are compiled down to EVM (Ethereum Virtual Machine) bytecode, which is a Turing complete low level programming language.
- Smart contracts cannot be changed after deployment. But most smart contracts contain mutable storage, and this storage can be used to adapt the behavior of the smart contract. With this, many smart contracts can update to a new version.

idea can have a function of the contract which you can call with the proper key and change the address stored - which might reference another smart contract for instance.

Definition 26.17 (Account). *Ethereum knows two kinds of accounts. Externally Owned Accounts (EOAs) are controlled by individuals, with a secret key. Contract Accounts (CAs) are for smart contracts. CAs are not controlled by a user.*

CA - store deployed smart contracts

Definition 26.18 (Ethereum Transaction). *An Ethereum transaction is sent by a user who controls an EOA to the Ethereum network. A transaction contains:*

avoids double spending by tracking order.

- *Nonce: This "number only used once" is simply a counter that counts how many transactions the account of the sender of the transaction has already sent.*
- *160-bit address of the recipient.*
- *The transaction is signed by the user controlling the EOA.*
- *Value: The amount of Wei (the native currency of Ethereum) to transfer from the sender to the recipient.*
- *Data: Optional data field, which can be accessed by smart contracts.*

- *Value: The amount of Wei (the native currency of Ethereum) to transfer from the sender to the recipient.*
- *Data: Optional data field, which can be accessed by smart contracts.*
- *StartGas: A value representing the maximum amount of computation this transaction is allowed to use.*
- *GasPrice: How many Wei per unit of Gas the sender is paying. Miners will probably select transactions with a higher GasPrice, so a high GasPrice will make sure that the transaction is executed more quickly.*

avoids stucking network in while loops being turing complete

Definition 26.19 (Simple Transaction). *A simple transaction in Ethereum transfers some of the native currency, called Wei, from one EOA to another. Higher units of curency are called Szabo, Finney, and Ether, with 10^{18} Wei = 10^6 Szabo = 10^3 Finney = 1 Ether. The data field in a simple transaction is empty.*

Definition 26.20 (Smart Contract Creation Transaction). *A transaction whose recipient address field is set to 0 and whose data field is set to compiled EVM code is used to deploy that code as a smart contract on the Ethereum blockchain. The contract is considered deployed after it has been mined in a block and is included in the blockchain at a sufficient depth.*

Definition 26.21 (Smart Contract Execution Transaction). *A transaction that has a smart contract address in its recipient field and code to execute a specific function of that contract in its data field.*

Remarks:

- Smart Contracts can execute computations, store data, send Ether to other accounts or smart contracts, and invoke other smart contracts.
- Smart contracts can be programmed to self destruct. This is the only way to remove them again from the Ethereum blockchain.
- Each contract stores data in 3 separate entities: storage, memory, and stack. Of these, only the storage area is persistent between transactions. Storage is a key-value store of 256 bit words to 256 bit words. The storage data is persisted in the Ethereum blockchain, like the hard disk of a traditional computer. Memory and stack are for intermediate storage required while running a specific function, similar to RAM and registers of a traditional computer. The read/write gas costs of persistent storage is significantly higher than those of memory and stack.

Definition 26.22 (Gas). *Gas is the unit of an atomic computation, like swapping two variables. Complex operations use more than 1 Gas, e.g., ADDing two numbers costs 3 Gas.*

how much Gas you are willing to spend

- The product of StartGas and GasPrice is the maximum cost of the entire transaction.

Definition 26.23 (Block). *In Ethereum, like in Bitcoin, a block is a collection of transactions that is considered a part of the canonical history of transactions. Among other things, a block contains: pointers to parent and up to two uncles, the hash of the root node of a trie structure populated with each transaction of the block, the hash of the root node of the state trie (after transactions have been executed)*

Definition 26.23 (Block). *In Ethereum, like in Bitcoin, a block is a collection of transactions that is considered a part of the canonical history of transactions. Among other things, a block contains: pointers to parent and up to two uncles, the hash of the root node of a trie structure populated with each transaction of the block, the hash of the root node of the state trie (after transactions have been executed)*

Proof of Stake

Definition 26.27 (Proof-of-stake). *Proof-of-work awards block rewards to the lucky miner that solved a cryptopuzzle. In contrast, proof-of-stake awards block rewards proportionally to the economic stake in the system.*

So similar to the idea you had of giving the reward probabilistically at each node in the network. I discovered that such an idea was already implemented: Ouroboros.

Definition 26.28 (Chain based proof-of-stake). *Accounts hold lottery tickets according to their stake. The lottery is pseudo-random, in the sense that hash functions computed on the state of the blockchain will select which account is winning. The winning account can extend the longest chain by a block, and earn the block reward.*

Remarks:

- It gets tricky if the actual winner of the lottery does not produce a block in time, or some nodes do not see this block in time. This is why some suggested proof-of-stake systems add a voting phase.

maybe isolated block or block that just joined the network and did not register any transaction

Definition 26.29 (BFT based proof-of-stake). *The lottery winner only gets to propose a block to be added to the blockchain. A committee then votes (yes, byzantine fault tolerance) whether to accept that block into the blockchain. If no agreement is reached, this process is repeated.*

Authenticated Agreement

12 December 2020 14:28

This chapter exposes the ideas of Practical Byzantine Fault-Tolerant System.

The idea was that up to now the methods seem required quite a bit of overhead and where not that simple to apply in practice.

This section explains how with the use of cryptography you might well set up a PBFT system.

In order to see that we will start with a simple Authenticated Agreement Algorithm; leveraging signatures.

Definition 25.1 (Signature). *Every node can digitally* ***sign*** *its messages in a way that no other node can forge, thus nodes can reliably determine which node a signed message originated from. We denote a message* $\texttt{msg}(x)$ *signed by node* u *with* $\texttt{msg}(x)_u$.

Algorithm 25.2 Byzantine Agreement with Authentication

Code for primary p:

1: **if** input is 1 **then**

Code for primary p:

```
1: if input is 1 then
2:    broadcast value(1)_p
3:    decide 1 and terminate
4: else
5:    decide 0 and terminate
6: end if
```

Code for all other nodes v:

```
7: for all rounds i ∈ {1, ..., f + 1} do
8:    S is the set of accepted messages value(1)_u.
9:    if |S| ≥ i and value(1)_p ∈ S then
10:      broadcast S ∪ {value(1)_v}
11:      decide 1 and terminate
12:   end if
13: end for
14: decide 0 and terminate
```

synchronous

such that you know that there was at least 1 non-byzantine node, when message propagates from node to node.

Such that if all of the nodes receive $value(1)_p$ in round 1 then all terminate in the first round and decide on 1.

If none receives $value(1)_p$ in round 1 then they will all decide on 0.

Moreover notice that even if the primary is byzantine and sends different messages to different nodes, the nodes that received the 1 will propagate it in the network in further rounds so that it will eventually arrive to also nodes receiving a 0 and such that they all will agree.

Theorem 25.3. *Algorithm 25.2 can tolerate $f < n$ byzantine failures while terminating in $f + 1$ rounds.*

any number of byzantine not $\frac{1}{3}$.

Notice that such threshold

Theorem 25.3. *Algorithm 25.2 can tolerate $f < n$ byzantine failures while terminating in $f + 1$ rounds.*

Proof. Assuming that the primary p is not byzantine and its input is 1, then p broadcasts $\texttt{value}(1)_p$ in the first round, which will trigger all correct nodes to decide on 1. If p's input is 0, there is no signed message $\texttt{value}(1)_p$, and no node can decide on 1.

If primary p is byzantine, we need all correct nodes to decide on the same value for the algorithm to be correct.

Assume $i < f + 1$ is the minimal round in which any correct node u decides on 1. In this case, u has a set S of at least i messages from other nodes for value 1 in round i, including one of p. Therefore, in round $i + 1 \leq f + 1$, all other correct nodes will receive S and u's message for value 1 and thus decide on 1 too.

Now assume that $i = f + 1$ is the minimal round in which a correct node u decides for 1. Thus u must have received $f + 1$ messages for value 1, one of which must be from a correct node since there are only f byzantine nodes. In this case some other correct node v must have decided on 1 in some round $j < i$, which contradicts i's minimality; hence this case cannot happen.

Finally, if no correct node decides on 1 by the end of round $f + 1$, then all correct nodes will decide on 0. □

Notice that such threshold that was proved in a thm does not hold anymore as we are using cryptography now.

So that assuming the network is not down they will all agree on the same value:

Remarks:

- If the primary is a correct node, Algorithm 25.2 only needs two rounds! Otherwise, the algorithm terminates in at most $f + 1$ rounds, which is optimal as described in Theorem 17.20.
- Does Algorithm 25.2 satisfy any of the validity conditions introduced in Section 17.1? No! A byzantine primary can dictate the decision value.

So for instance any-input validity not guaranteed.

- Can we modify the algorithm such that the correct-input validity condition is satisfied? Yes! We can run the algorithm in parallel for $2f+1$ primary nodes. Either 0 or 1 will occur at least $f + 1$ times, which means that one correct process had to have this value in the first place. In this case, we can only handle $f < \frac{n}{2}$ byzantine nodes.

↓ as otherwise no chance of having $2f+1$ different primaries

having 2f+1 different primaries

↳ You would then decide on the value that was selected in at least f+1 runtimes.

- Relying on synchrony limits the practicality of the protocol. What if messages can be lost or the system is asynchronous?

↳ This brings us directly to PBFT protocol.

PBFT System

Practical Byzantine Fault Tolerance (PBFT) is one of the first and perhaps the most instructive protocol for achieving state replication among nodes as in Definition 15.8 with byzantine nodes in an asynchronous network. We present a simplified version of PBFT without any optimizations.

Such that this is a practically implementable algorithm that works under asynchronicity, in a permissioned network.

Definition 25.4 (System Model). *We consider a system with $n = 3f+1$ nodes, and additionally an unbounded number of clients. There are at most f byzantine nodes, and clients can be byzantine as well. The network is asynchronous, and messages have variable delay and can get lost. Clients send requests that correct nodes have to order to achieve state replication.*

The mechanism to achieve PBFT system is very similar to the one above.

It consists of:

Remarks:

- At any given time, every node will consider one designated node to be the *primary* and the other nodes to be *backups*.
- The timespan for which a node p is seen as the primary from the perspective of another node is called a view.

Definition 25.5 (View). *A **view** v is a non-negative integer representing the node's local perception of the system. We say that **node u is in view** v as long as node u considers node $p = v \bmod n$ to be the primary.*

Assume $n = 4$

views

$p=0$	0	5
$p=1$	1	6
$p=2$	2	7
$p=3$	3	8

Remarks:

- All nodes start out in view 0. Nodes can potentially be in different

Remarks:

- All nodes start out in view 0. Nodes can potentially be in different views (i.e. have different local values for v) at any given time.
- If backups detect faulty behavior in the primary, they switch to the next primary with a so-called *view change* (see Section 25.4).
- In the asynchronous model, requests can arrive at the nodes in different orders. While a primary remains in charge (sufficiently many nodes share the view v), it thus adopts the function of a serializer (cf. Algorithm 15.9).

Serializer function

Definition 25.6 (Sequence Number). *During a view, a node relies on the primary to assign consecutive* ***sequence numbers*** *(integers) that function as indices in the global order (cf. Definition 15.8) for the requests that clients send.*

↳ so you have then sequence numbers for the different client requests.

It is how important to keep consistency for such sequence numbers during a view change.

During a view change, we ensure that no two correct nodes execute requests in different orders. On the one hand, we need to exchange information on the current state to guarantee that a correct new primary knows the latest sequence number that has been accepted by sufficiently many backups. On the other hand, exchanging information will enable backups to determine if the new primary acts in a byzantine fashion, e.g. reassigning the latest sequence number to a different request.

So backups node will need to communicate with each other telling each other latest sequence

each other, telling each other latest sequence number so that a new byzantine primary cannot trick the system.

Based on this the idea is then to accept messages if the following holds:

Definition 25.7 (Accepted Messages). *A correct node that is in view v will only* ***accept*** *messages that it can authenticate, that follow the specification of the protocol, and that also belong to view v.*

Remarks:

- The protocol will guarantee that once a correct node has executed a request r with sequence number s, then no correct node will execute any request $r' \neq r$ with sequence number s, not unlike Lemma 15.14.
- Correct primaries choose sequence numbers in order, without gap, i.e. if a correct primary proposed s as the sequence number for the last request, then it will use $s + 1$ for the next request that it proposes.
- Before a node can safely execute a request r with a sequence number s, it will wait until it knows that the decision to execute r with s has been reached and is widely known.
- Informally, nodes will collect confirmation messages by sets of at least $2f + 1$ nodes to guarantee that that information is sufficiently widely distributed.

as this ensures that

Reason: standard Quorum arguments:

Lemma 25.8 ($2f + 1$ Quorum Intersection). *Let S_1 with $|S_1| \geq 2f + 1$ and S_2 with $|S_2| \geq 2f + 1$ each be sets of nodes. Then there exists a correct node in $S_1 \cap S_2$.*

Proof. Let S_1, S_2 each be sets of at least $2f + 1$ nodes. There are $3f + 1$ nodes in total, thus due to the pigeonhole principle the intersection $S_1 \cap S_2$ contains at least $f + 1$ nodes. Since there are at most f faulty nodes, $S_1 \cap S_2$ contains at least 1 correct node. □

i.e. there will always be a correct node in the intersection of S_1 and S_2 such that it will not execute a request $r' \neq r$ with same sequence number s.

Given this general idea we write it down now, in terms of pseudo-code

Informally the algorithm have 5 steps

1. The nodes receive a request and relay it to the primary.
2. The primary sends a `pre-prepare`-message to all backups, informing them that it wants to execute that request with the sequence number specified in the message.
3. Backups send `prepare`-messages to all nodes, informing them that they agree with that suggestion.
4. All nodes send `commit`-messages to all nodes, informing everyone that they have committed to execute the request with that sequence number.
5. They execute the request and inform the client.

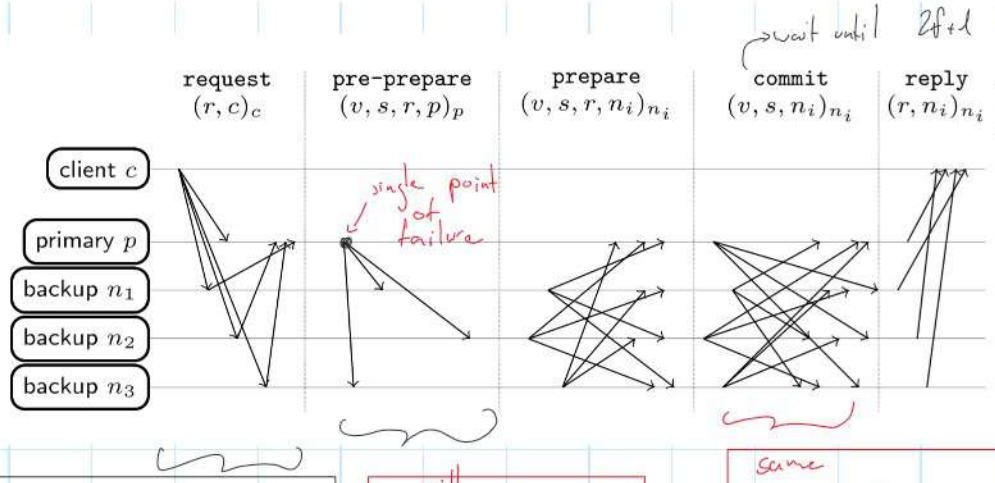
Validate primary not byzantine

serializer function.

and can be visualized as follows:

wait until 2f+1 before committing

`request` | `pre-prepare` | `prepare` | `commit` | `reply`



Formally the protocol would look as follows

Definition 25.10 (Pre-Prepare Phase)**.** *In the* ***pre-prepare phase*** *of the agreement protocol, the nodes execute Algorithm 25.11.*

Algorithm 25.11 PBFT Agreement Protocol: Pre-Prepare Phase

Code for primary p in view v:

1: accept **request**$(r, c)_c$ that originated from client c
2: pick next sequence number s
3: send **pre-prepare**$(v, s, r, p)_p$ to all backups

Code for backup b:

4: accept **request**$(r, c)_c$ from client c
5: relay **request**$(r, c)_c$ to primary p

} relay as maybe client byzantine and did not inform all of the nodes. If client informed with different requests → due to

nodes. If client informed with different requests → due to signature can check at it. Remove request.

Definition 25.12 (Prepare Phase). *In the* ***prepare phase*** *of the agreement protocol, every backup b executes Algorithm 25.13. Once it has sent the* `prepare`*-message, we say that b has* ***pre-prepared*** *r for (v, s).*

Algorithm 25.13 PBFT Agreement Protocol: Prepare Phase

Code for backup b in view v:

1: accept `pre-prepare`$(v, s, r, p)_p$
2: **if** b has not yet accepted a `pre-prepare`-message for (v, s, r') with $r' \neq r$ **then**
3: send `prepare`$(v, s, r, b)_b$ to all nodes
4: **end if**

Definition 25.14 (Prepared-Certificate). *A node n_i that has pre-prepared a request executes Algorithm 25.15. It waits until it has collected $2f$* `prepare`*-messages (including n_i's own, if it is a backup) in Line 1. Together with the* `pre-prepare`*-message for (v, s, r), they form a* ***prepared-certificate****.*

guarantee of quorum properties

Algorithm 25.15 PBFT Agreement Protocol: Commit Phase

Code for node n_i that has pre-prepared r for (v, s):

1: wait until $2f$ `prepare`-messages matching (v, s, r) have been accepted
2: create `prepared-certificate` for (v, s, r)
3: send `commit`$(v, s, n_i)_{n_i}$ to all nodes

Definition 25.16 (Committed-Certificate). *A node n_i that has created a prepared-certificate for a request executes Algorithm 25.17. It waits until it has collected $2f + 1$* `commit`*-messages (including n_i's own) in Line 1. They form a* ***committed-certificate*** *and allow to safely execute the request once all requests with lower sequence numbers have been executed.*

Algorithm 25.17 PBFT Agreement Protocol: Execute Phase

Code for node n_i that has created a prepared-certificate for (v, s, r):

1: wait until $2f + 1$ `commit`-messages matching (v, s) have been accepted
2: create `committed-certificate` for (v, s, r)
3: wait until all requests with lower sequence numbers have been executed
4: execute request r
5: send $\texttt{reply}(r, n_i)_{n_i}$ to client

Remarks:

- Note that the agreement protocol can run for multiple requests in parallel. Since we are in the variable delay model and messages can arrive out of order, we thus have to wait in Algorithm 25.17 Line 3 for all requests with lower sequence numbers to be executed.

Why is f+1 replies enough?

- We will see in Section ~~25.4~~ next that PBFT guarantees that once a single correct node executed the request, then all correct nodes will never execute a different request with the same sequence number. Thus, knowing that a single correct node executed a request is enough for the client.

across View proved will be through the change of view method.

Lemma 25.18 (PBFT: Unique Sequence Numbers within View)**.** *If a node was able to create a prepared-certificate for (v, s, r), then no node can create a prepared-certificate for (v, s, r') with $r' \neq r$.*

So summing up there is 1 correct at least and this

Proof. Assume two (not necessarily distinct) nodes create prepared-certificates for (v, s, r) and (v, s, r'). Since a prepared-certificate contains $2f + 1$ messages, a correct node sent a `pre-prepare-` or `prepare`-message for each of (v, s, r) and (v, s, r') due to Lemma 25.8. A correct primary only sends a single `pre-prepare`-message for each (v, s), see Algorithm 25.11 Lines 2 and 3. A correct backup only sends a single `prepare`-message for each (v, s), see Algorithm 25.13 Lines 2 and 3. Thus, $r' = r$. □

at least and this will never send a (v, s') with $r' \neq v$ given the if conditional!

pre-prepare-message for each (v, s), see Algorithm 25.11 Lines 2 and 3. A correct backup only sends a single prepare-message for each (v, s), see Algorithm 25.13 Lines 2 and 3. Thus, $r' = r$. □

Notice finally that you can address network failures on the client side in the following way:

- If the client does not receive at least $f+1$ reply-messages fast enough, it can start over by resending the request to initiate Algorithm 25.11 again. To prevent correct nodes that already executed the request from executing it a second time, clients can mark their requests with some kind of unique identifiers like a local timestamp. Correct nodes can then react to each request that is resent by a client as required by PBFT, and they can decide if they still need to execute a given request or have already done so before.

On top of it nodes can trigger a view change by themselves to deal with a faulty primary.

If the primary is faulty, the system has to perform a view change to move to the next primary so the system can make progress. To that end, nodes use a local faulty-timer (and only that!) to decide whether they consider the primary to be faulty.

No! By introducing a new variable and an incomplete world easy

Definition 25.19 (Faulty-Timer). *When backup b accepts request r in Algorithm 25.11 Line 4, b starts a local **faulty-timer** (if the timer is not already running) that will only stop once b executes r.*

Remarks:

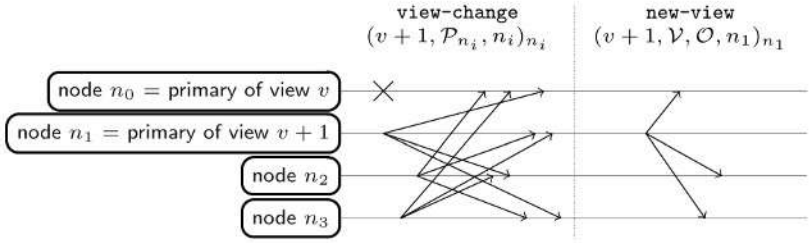
- If the faulty-timer expires, the backup considers the primary faulty and triggers a view change.

- If the faulty-timer expires, the backup considers the primary faulty and triggers a view change.
- We leave out the details regarding for what timespan to set the faulty-timer. This is a trade-off between patience and efficiency.

- During a view change, the protocol has to guarantee that requests that have already been executed by some correct nodes will not be executed with the different sequence numbers by other correct nodes.

This is solved in the following way:

Definition 25.20 (PBFT: View Change Protocol)**.** *In the view change protocol, a node whose faulty-timer has expired enters the* ***view change phase*** *by running Algorithm 25.23. During the* ***new view phase*** *(which all nodes continually listen for), the primary of the next view runs Algorithm 25.24 while all other nodes run Algorithm 25.25.*



Algorithm 25.23 PBFT View Change Protocol: View Change Phase

Code for backup b in view v whose faulty-timer has expired:

1: stop accepting `pre-prepare`/`prepare`/`commit`-messages for v
2: let $\mathcal{P}_b$ be the set of all prepared-certificates that b has collected since the system was started
3: send `view-change`$(v+1, \mathcal{P}_b, b)_b$ to all nodes

so you include all of the pre-prepared.

Algorithm 25.24 PBFT View Change Protocol: New View Phase - Primary

Code for new primary p of view $v+1$:

1: accept $2f+1$ `view-change`-messages (including possibly p's own) in a set $\mathcal{V}$ (this is the *new-view-certificate*)
2: let $\mathcal{O}$ be a set of `pre-prepare`$(v+1, s, r, p)_p$ for all pairs (s, r) where at least one prepared-certificate for (s, r) exists in $\mathcal{V}$

a important to the process message

$\mathcal{V}$ (this is the *new-view-certificate*)
2: let $\mathcal{O}$ be a set of `pre-prepare`$(v+1, s, r, p)_p$ for all pairs (s, r) where at least one prepared-certificate for (s, r) exists in $\mathcal{V}$
3: let $s^{\mathcal{V}}_{max}$ be the highest sequence number for which $\mathcal{O}$ contains a `pre-prepare`-message
4: add to $\mathcal{O}$ a message `pre-prepare`$(v+1, s', \texttt{null}, p)_p$ for every sequence number $s' < s^{\mathcal{V}}_{max}$ for which $\mathcal{O}$ does not contain a `pre-prepare`-message
5: send `new-view`$(v+1, \mathcal{V}, \mathcal{O}, p)_p$ to all nodes
6: start processing requests for view $v+1$ according to Algorithm 25.11 starting from sequence number $s^{\mathcal{V}}_{max} + 1$

Algorithm 25.25 PBFT View Change Protocol: New View Phase - Backup

Code for backup b of view $v+1$ if b's local view is $v' < v+1$:

1: accept `new-view`$(v+1, \mathcal{V}, \mathcal{O}, p)_p$
2: stop accepting `pre-prepare`-/`prepare`-/`commit`-messages for v `// in case` b `has not run Algorithm 25.23 for` $v+1$ `yet`
3: set local view to $v+1$
4: **if** p is primary of $v+1$ **then**
5: **if** $\mathcal{O}$ was correctly constructed from $\mathcal{V}$ according to Algorithm 25.24 Lines 2 and 4 **then**
6: respond to all `pre-prepare`-messages in $\mathcal{O}$ as in the agreement protocol, starting from Algorithm 25.13
7: start accepting messages for view $v+1$
8: **else**
9: trigger view change to $v+2$ using Algorithm 25.23
10: **end if**
11: **end if**

Theorem 25.26 (PBFT:Unique Sequence Numbers Across Views)**.** *Together, the PBFT agreement protocol and the PBFT view change protocol guarantee that if a correct node executes a request r in view v with sequence number s, then no correct node will execute any $r' \neq r$ with sequence number s in any view $v' \geq v$.*

Proof. If no view change takes place, then Lemma 25.18 proves the statement. Therefore, assume that a view change takes place, and consider view $v' > v$.

We will show that if some correct node executed a request r with sequence number s during v, then a correct primary will send a `pre-prepare`-message matching (v', s, r) in the $\mathcal{O}$-component of the `new-view`$(v', \mathcal{V}, \mathcal{O}, p)$-message. This guarantees that no correct node will be able to collect a prepared-certificate

number s during v, then a correct primary will send a `pre-prepare`-message matching (v', s, r) in the $\mathcal{O}$-component of the `new-view`$(v', \mathcal{V}, \mathcal{O}, p)$-message. This guarantees that no correct node will be able to collect a prepared-certificate for s and a different $r' \neq r$.

Consider the new-view-certificate $\mathcal{V}$ (see Algorithm 25.24 Line 1). If any correct node executed request r with sequence number s, then due to Algorithm 25.17 Line 1, there is a set R_1 of at least $2f + 1$ nodes that sent a `commit`-message matching (s, r), and thus the correct nodes in R_1 all collected a prepared-certificate in Algorithm 25.15 Line 1.

The new-view-certificate contains `view-change`-messages from a set R_2 of $2f + 1$ nodes. Thus according to Lemma 25.8, there is at least one correct node $c_r \in R_1 \cap R_2$ that both collected a prepared-certificate matching (s, r) and whose `view-change`-message is contained in $\mathcal{V}$.

Therefore, if some correct node executed r with sequence number s, then $\mathcal{V}$ contains a prepared-certificate matching (s, r) from c_r. Thus, if some correct node executed r with sequence number s, then due to Algorithm 25.24 Line 2, a correct primary p sends a `new-view`$(v', \mathcal{V}, \mathcal{O}, p)$-message where $\mathcal{O}$ contains a `pre-prepare`(v', s, r, p)-message.

here is the key!

Correct backups will enter view v' only if the `new-view`-message for v' contains a valid new-view-certificate $\mathcal{V}$ and if $\mathcal{O}$ was constructed correctly from $\mathcal{V}$, see Algorithm 25.25 Line 5. They will then respond to the messages in $\mathcal{O}$ before they start accepting other `pre-prepare`-messages for v' due to the order of Algorithm 25.25 Lines 6 and 7. Therefore, for the sequence numbers that appear in $\mathcal{O}$, correct backups will only send `prepare`-messages responding to the `pre-prepare`-messages found in $\mathcal{O}$ due to Algorithm 25.13 Lines 2 and 3. This guarantees that in v', for every sequence number s that appears in $\mathcal{O}$, backups can only collect prepared-certificates for the triple (v', s, r) that appears in $\mathcal{O}$.

Together with the above, this proves that if some correct node executed request r with sequence number s in v, then no node will be able to collect a prepared-certificate for some $r' \neq r$ with sequence number s in any view $v' \geq v$, and thus no correct node will execute r' with sequence number s. □

$\forall$ ϵ

this then basically is the same as within View.

Same concept for final step of the proof.