Chapter 8: Consensus

- Algo Ben-Or : line 7 (if all messages contain the same value v) : do you take your own value into account ? Or do you only work with others ? Same question with propose. Also, does the empty value count as a value or if there is a propose with a real value you can choose this one ?
    - Yes, your own value can be counted as well. To see this, note that this clearly does not interfere with the validity and termination properties, and the proof for agreement only assumes that you have knowledge of more than n/2 input values.
    - For propose, it only makes sense to keep the own proposal message for further processing if it contains a value. However, the algorithm does not require this (check proofs of lemmas 8.17-8.19).
    - The empty value ("bottom", LaTeX: \bot) is not a value in the sense of line 17.
- page 63, the algorithm 8.22, I can't understand why this is a shared coin, this is an algorithm for node u, but why we have 0 or 1 for all nodes with constant probability? What does the return 0/ return 1 mean? And for the lemma 8.23, could you give me an example to illustrate what W is?
    - It is a shared coin algorithm if all nodes each run this code individually.
    - We have constant probabilities > 0 for both outcomes 0 and 1 to be attained unanimously as shown in Theorem 8.25. There is a chance > 0, however, that nodes terminate with different values.
    - "return 0 / 1" decides on the actual value that a "shared coin flip" shows to a specific node u running this code. In most cases (all 1 or all 0, probability for this is roughly 0.65) the nodes will see the same resulting value.
    - Draw an n x (n-f) matrix and make (n-f) crosses in each column. The rows represent all nodes, the columns represent different coin sets you received, each containing (n-f) coin values. Now take a highlighter and mark every row that contains at least (f+1) crosses. These are all the coin values that you received in at least (f+1) coin sets, thus, W. The claim is, that there must be at least (f+1) such rows. Thus, as n = 3f + 1, we can deduce Lemma 8.24, that is, from the perspective of a single node u running the code, it can be entirely sure that all nodes will be aware of all the coins that u finds to be in its proper set W (which may differ for every node) as they have received it in at least one coin set. Intuitively, this comes from the fact that every node waits for (n-f) coin sets and W is contained in (f+1) of them, (n-f)+(f+1) > n.
- Aufgabe 2.2 Assignment 3 Deterministic Random Consensus?!: Ich verstehe nicht ganz wieso dieses Argument der Lösung gültig ist. Der letzte Satz sagt: «…this scheduling might happen in every round and the algorithm does not terminate.». Das «bad scheduling» hört sich nach einem Ereignis an, dass ede Runde mit einer gewissen Wahrscheinlichkeit passiert. Wenn das der Fall wäre könnte ich auch beim Randomized Consensus von Ben-Or auch annehmen, dass immer das Ereignis v_i = 1 eintritt in Zeile 23 + immer dieses bad scheduling. Das ist im worst-case auch unendlich lange Laufzeit bzw. keine Termination. Im Lemma 8.19 wird auch nur endliche expected termination bewiesen und nicht Termination in worst-case.

Ich verstehe, dass es mit einem node fail nicht zum Agreement kommt und man das ähnlich wie Theorem 8.14 machen kann, aber wieso macht man das denn nicht?
Die Musterlösung macht für mich auch Sinn, wenn das bad scheduling Etwas ist, dass einmal passiert und danach fix ist. Aber der Satz müsste anders formuliert sein, oder?

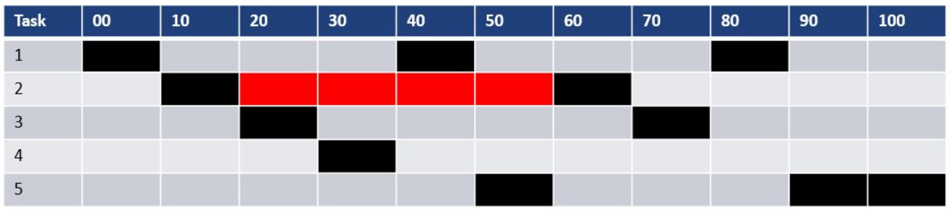
- The short answer: this is just how the asynchronous model is defined. We use it in order to build systems that are robust towards any mal-functioning such as "evil scheduling".
- Furthermore, this scheduling is much more practical than you might think: Consider a real-world scenario, where some nodes are geographically clustered much closer to each other and thus a similar scheduling (that is potentially biased as in the exercise) might, indeed, occur in every round. We still want our algorithm to work and, eventually, terminate. I don't personally see how a random coin toss (that is designed to behave randomly and not evil) will behave just like an evil attacker forever.

Chapter 9: Scheduling

- Ich hab beim Lösen der Aufgabe 1.1 von assignment 4 gemerkt, dass ich nicht genau weiss was «response time» ist. Dies schnell und einfach gerade an der Aufgabe zu erklären, wäre ein Anliegen von mir. Ein allgemeines Rezept für die Berechnung wäre auch hilfreich.
    - The response time depends on the scheduling algorithm, but in general "longest time until a process responds in the worst case"

      For RR this is when the task just lost it's time slice i.e. if we have N tasks and the scheduling decision isch performed each 10s then the response time is (N -1)* 10ms.

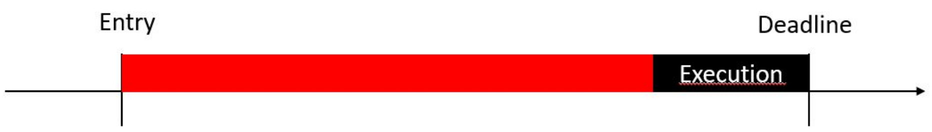
RR



Task	00	10	20	30	40	50	60	70	80	90	100
1	⬛				⬛				⬛		
2		⬛	🟥	🟥	🟥	🟥	⬛				
3			⬛					⬛			
4				⬛							
5						⬛				⬛	⬛

For EDF the response time is (if the schedule is feasible) at maximum the time between entry time and deadline without the execution time (deadline - entry - exec). This means that the worst case for response time is when a task is scheduled right at the time when it is barely able to finish.

EDF



For SRTF we simply cannot compute the response time since a long running task might never be scheduled when there are many short running tasks arriving all the time.

Chapter 10: I/O

- Assignment 4, Exercise 2.2 c): A DMA controller (or DMA engine) has multiple channels that can be used by device drivers to request a DMA transfer. The controller itself is capable of requesting a 32-bit word every 100 nsec. A response takes equally long. How fast does the bus have to be to avoid being a bottleneck?
  Answer: Each bus transaction has a request and a response each taking 100 nsec, or 200 nsec per bus transaction. This gives 5 million bus transactions / sec. If each one is four bytes, the bus should be able to handle 20 MB/sec. The fact that these transactions may be distributed over four I/O devices (four channels) in round-robin fashion is irrelevant. A bus transaction takes 200 nsec, regardless of whether consecutive requests are to the same device or different device, so the number of channels the DMA controller has does not matter.
  I don't understand why the bus transactions can't be pipelined, why it isn't possible to request a word every 100 nsec.
  - DMA request is the request to access the shared bus, the reply is the transfer itself. The bus is a resources that has to be multiplexed through requesting the bus (like a lock) for each access and thus can not be pipelined.

  I'm also a bit confused about why a round-robin fashion is chosen to allow access for multiple devices, wouldn't it be possible for the DMA controller to multiplex the bus access for the devices?
  - There is not only one DMA controller but multiple (Each device might have one + additional others). In most of these case we have some kind of bus arbiter which decies the scheduling algorithm. In many cases round robin is the algorithm used to multiplex the bus since RR is simple to implement in hardware.

Chapter 11: Byzantine Agreement

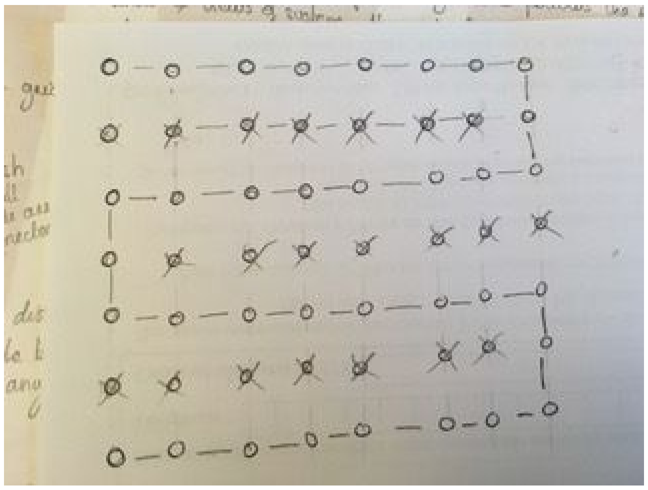
- Assignment 5, 1.1 c: why is the answer w + h + 1 and not just w + h? Can it not see last new value and compute or is it just one compute per time unit?So for runtime Analysis we Always hafe to count number of msg exchanges plus an extra step for calulation/Termination?
  - It takes w+h rounds for the node to receive all values. The node can however not know that it has received all values, since it does not know w and h. Therefore, it needs to wait for another round and see that it there is no new values coming.

- Assignment 5, 1.1 d): I dont agree with the master solution. In a) the master solution said "agree on the smallest value". So if it is a binary system, the question is only is there a 0 in the system? It is not stipulated that in this broadcasting set up, that the recipients dont know from whom the messages are sent. As soon as it is information Sender, value, the corner node will Always recieve also the 0. So the only Problem is in all-same value validity… than if all have 1 and just the byzantine 0 it would lead to a disagreement. Would the solution not rather be that you have to at least partition the network and assume, that byzantine nodes change through coming information? And that is another question: byzantine nodes are not allowed identity Theft or change, but can they change another nodes values they have to broadcast in such a grid set up?

   - All-same validity is not the only problem. If the byzantine node tells one node that it has input 0 and all other nodes that it has input 1, then the node that is cut off will not be able to decide which value to take. The only algorithm which would work in such a setting would be to always agree on 0 (independent of the input), which is a trivial system. The trivial system is however not very interesting. Byzantine agreement that we consider until Chapter 12.5 (which is the standard byzantine agreement problem) does not allow to use signatures. Therefore, if a node on a grid does not receive a message from another node directly, it cannot verify if the forwarded message is correct. So the nodes can change the values of other nodes without pretending to be other nodes.

- Assignment 5, 1.2 b: why is it 3(w+h) ? And when is it doable? (because the master solution is around 2 times w + h.
   - The idea is that you would move vertically up, down and up, and horizontally you would move to the right, the left and the right again. The bound should hold for larger values of w and h.

- Exercise 5) 1.2 Synchronous Consensus in a Grid - Crash Failures: question b. The solution states that the largest l you can achieve is 27 by arranging the faulty nodes in the diagonals. I found 34 using this strategy : cf image below. What am I missing here?
   - You are only allowed to crash 13 nodes in the task.



Chapter 12: Broadcast & Shared Coins

- I am having a hard time seing the difference between best-effort broadcast and reliable 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. So, for me, at some point, all correct node will accept message m (as it is a broadcast) : so it is a reliable broadcast. What am I missing ?
    - You are missing the Byzantine nodes in your system. Assume you have a byzantine nodes in the system: with reliable broadcast, this node would be detected if it sends different messages to different correct nodes, but with best effort broadcast this node will not be detected.

- Could you explain the third paragraph of the proof of the Theorem 12.24 on Page 109 (Algorithm 12.23 solves asynchronous binary agreement for $f < n/2$ crash failures.)
    - The main Idea is to just read all stored values. In Algorithm 12.10, all such values were written on the blackboard. In this algorithm, each of the “written” values is saved at at least n-f correct nodes. If f of these nodes crash, only n-2f nodes are left that have this value. In order to read it, you would request all nodes, and since f might have crashed, you wait for n-f responses. The only question left is whether the nodes which gave you a response (n-f nodes) intersect with the set of nodes which hold the value (n-2f nodes). The calculation in the lecture notes shows that there is an intersection and that the node will therefore see all written values.

- Could you tell me why the expected round is 5 on Page 112?
    - The Ben-Or algorithm in the synchronous case consists of two rounds: in the first round, the input values of the current round are exchanged, and in the second round, the shared coin is computed. These two rounds are repeated until agreement is established. It might be that all correct nodes finish after exchanging their values in the very first communication round of the algorithm (due to all-same-validity). If this is not the case, a shared coin round is needed, and another “first round” of the Ben-Or algorithm. This gives the described formula.

- Solution to assignment 5, 2.3 b), the third last line, I don't understand "Since the intervals I of the nodes do not intersect in any value", what is the interval I?
    - I is the local interval of a node in Algorithm 2. Since each correct node can wait for at most n-f values, it will not receive 2f correct values in the worst case, and f byzantine values instead. 3 of the nodes will have a local interval I = {-3,-3,-3,-2,-1,0,1}, 3 nodes will have I = {-1,0,1,2,3,3,3}, and one node will only receive correct values, i.e. I = {-3,-2,-1,0,1,2,3}. If each of the nodes removes 2 largest and smallest values, you get the new inputs as described in task

- Chapter 12: Ich verstehe im Algorithmus 12.17 in Zeile 5 die Formulierung «…but not msg(v)…» nicht. Ich dachte zuerst, dass wäre ein Fehler und es muss v statt msg(v) stehen. Doch dann sah ich, dass etwas Ähnliches in Zeile 6 des Algorithmus 12.10 vorkommt. Was für eine Rolle spielt das für den Algorithmus? Eine weitere Frage gerade zu dem: Wieso terminiert er (Algo 12.17) nicht? Das steht später im Remark.

    - Line 5 of algorithm 12.17 says “but not msg(v)”, this is the reason: If a node has received msg(v), it has already send an echo for this message in line 3 of the algorithm. The second if-clause captures the case where a node has not received a message directly, but many echoes for this message instead, such that it can be sure that enough correct nodes saw the same message.
      Algorithm 12.17 does not terminate, since the messages can be arbitrarily delayed and some participating nodes in the broadcast might crash or be byzantine. You can think of it as the normal broadcast routine in the asynchronous case: if we assume that a node waits n and not n-f messages, this node might never terminate, because it might not receive the last f messages it is waiting for.

- Ex. 5, 2.3 : question b & c: byzantine are able to delay the arrival of 2 values. But can they delay everything and block the execution ? When trying to find how many byzantine we can tolerate, we show that with 2 byzantine that can delay it does not work (in case of 9 nodes). The result of question c is f < n/4, so in our case, it makes f=2 possible. I also did not quite get the proof. The nodes are only supposed to remove the largest and the smallest f values (so for two nodes : one large, one small. What happens for 3 ?), so why does it considers 2f + 2f ? It should only be 2f ? Or does it considers the possibility to delay?
    - The correct result of question c) is f<n/5, so it does not contradict the counter example from task b).
      Byzantine nodes can delay all messages arbitrarily long. Any correct node can only wait for n-f messages in the asynchronous case (otherwise byzantine nodes might decide not to send any messages at all, and the correct node will wait forever). However, in the worst case, the correct node will not receive f correct values, because these were delayed by the byzantine scheduler. Moreover, it will have f byzantine values among the n-f received values. In total, any correct node can expect to receive at most n-2f correct values. From the received n-f values, each node must remove f too large and f too small values, since these values might be arbitrarily large/small. This way, each correct node removes at least another f correct values.
      In total, each node removed 2f correct values. What we need to take into account now, is that the views of the correct nodes might differ. The nodes might also receive different byzantine values, and not know about that since they do not broadcast the values reliably. This is the reason, why any pair of correct nodes intersects in (n-f) - 2f - 2f = n-5f values.

Chapter 13: Consistency & Logical Time

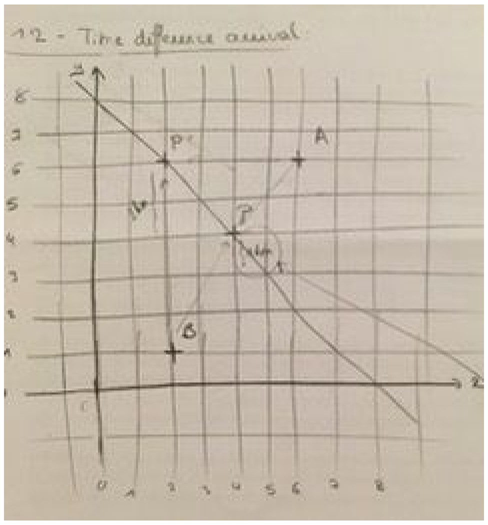
- I did not quite get how you count the number of consistent snapshot (mu) (even with the exercise 6 - 2.2, there only is the final result without details. If it is too difficult to explain the general idea, can you explain the solution of the 6 - 2.2, only the mu part, not mus and muc)
    - This is a combinatorial problem: Given two threads with a message sent between them, the number of consistent snapshots before the message was sent is equal to all possible combinations of partial executions. Let's say we have two threads A and B with one message sent from B to A. The number of consistent snapshots is equal the number of partial executions of A before the message was received times the number of partial executions of B (all partial executions of B are possible since after sending the message B can continue the execution before A receives the message). To this you add the number of partial executions of A after the message was received times the number of partial executions of B after the message was send, which is the number of consistent snapshots after the message was received. (No execution also counts as partial execution). This can be generalized to any number of messages exchanged between the two processes by summing all combinations of parts where the threads can act independently where for each combination the number of consistent snapshots is (n+1)*(m+1) where n is the number of independently executable instructions in thread A and m is the number of independently executable instructions in thread B and the +1 counts the empty set partial execution.
- Could you explain the proof of Lemma 13.13 on Page 117? (Sequentially consistent and quiescent consistency do not imply one another.)
    - This proof relies on the formal definitions. The first part uses the fact, that sequential consistency does not require the effect of two operations to be in the same order as the time of the operations, if the operations are on different nodes. Imagine a server with two clients: each client sends a request for an operation to the server, client A before client B. The operations on the clients are done as soon as the request is send, however, the server might choose to reorder the requests. The system is sequentially consistent, since each client doesn’t care whether its request got processed first or second, but not quiescent consistent consistent if the server executes request B before A since the send operation of A terminated before the send operation of B started.
    The second part relies on the fact, that if there is no quiescent point, any execution is quiescent consistent, even one that is not sequentially consistent.

Chapter 14: Time, Clocks & GPS

- 14.6: Lower Bound Proof of Lemma 14.41: Why does the Clock speed up? And how to you get to the clock skew of 1?
    - Since v_1 runs faster than v_2, t_1 > t_2. Therefore, v_2 speeds up in the sense that it will continuously make forward time jumps whenever a message from v_1 arrives. The same holds for all neighbors. The clock skew of 1 assumes a

worst-case message delivery time, which is 1 according to the section introduction.

- Ex. 6, 1.2 : question b. Isn't it more likely to be in (4,4) ? I drew it, set up the different points and it seems more likely to me to be in (4,4) : cf. image below.
    - Algebraically:
      r_(4,4) = sqrt((2-4)^2+(1-4)^2) - sqrt((6-4)^2+(6-4)^2) - 1 = sqrt(13) - sqrt(8) - 1 = -0.22
      r_(2,6) = sqrt((2-2)^2+(1-6)^2) - sqrt((6-2)^2+(6-6)^2) - 1 = 5 - 4 - 1 = 0
      So the *magnitude* of the residual for (2,6) is smaller.
    - Geometrically:
      The point (2,6) is at a distance 4 to A and distance 5 to B, which exactly matches the observed delay difference of 1 km.



Chapter 15: Quorum Systems

- Can you explain again the notion of load (how do you find it when you face an example, with or without access strategy) ? For example, I am not able to explain why the load is s in the exercise 1.3 - question a (assignment 7)
    - The load of a quorum system is the highest access probability of any node using the access strategy that gives the lowest maximum access probability of any node.
      If you know the access strategy, you can calculate the access probability for each node. You check in which quorums this node is and how often they get accessed. Then you select the node with the highest load.
      If you don't have the access strategy, you have to find the access strategy that minimizes the maximum load.
      In exercise 1.3 each access to a quorum system causes an access to s nodes. Therefore it is best for the load to distribute these accesses evenly among all nodes such that the load is the same for all.

- table 15.5, why in the majority quorum system, the load of the busiest node > 1/2 (I can't figure it out directly from the definition of load)?
    - In the majority quorum system more than half of the nodes have to be accessed each time, therefore there has to be at least one node that is accessed more than ½ of the times
- Theorem 15.6, the first paragraph I understand "Each time a quorum is accessed, at least one node in Q is accessed as well", but why this yields a lower bound of Lz(vi)>=1/q? And the second paragraph I understand "at least q nodes need to be accessed", but I don't understand why Lz(vi)>=q/n.
    - Each time a quorum gets accessed, at least one node in Q is also accessed. If Q consists of only one node, this node has to be part of every quorum and its load is therefore 1. If Q has more nodes, these accesses can be better distributed among the q nodes of Q. The lowest possible load for the nodes in Q is therefore 1/q.
      The q accessed nodes for each quorum access can ideally be equally distributed among all nodes, achieving a load of q/n. Therefore, at least one node (of V?) has to have a load of at least q/n.
- Assignment 7, 1.3 a), in the solution, why intuitively “Summed up over all servers we reach a total load of s”? And I don't understand the formal proof in the next question.
    - In exercise 1.3 each access to the quorum system causes an access to s nodes as each quorum contains s nodes. Therefore it is best for the load to distribute these accesses evenly among all nodes such that the load is the same for all. The equation in the solution of b) shows that the sum of the load of all nodes has to be s. To minimize the maximum load of any node, we have to distribute this load evenly among all nodes, such that each node has load s/n which can be achieved with a balanced access strategy.

Chapter 16: Eventual Consistency & Bitcoin

- consider that at some point two miners found a block at the same time, then the block-chain will fork.
    - am I right that both the blocks will be broadcasted to all the other nodes, and all nodes will have this fork to eventually agree on the longest path?
    - if 1.1 is true, consider that in the next round some miner is mining the next block, where is this block supposed to be inserted: so far both the branches have the same length. Do the miners pick up the branch to mine on randomly?
        - No. Bitcoin is a peer-to-peer network, and some nodes will see some updates before others, etc. Some nodes and miners will see fork prong-A, and some will see prong-B. The miners who see a specific prong will update their UTXO set with the block from that prong, and will make that block the parent block to the block that they are currently mining. A full node will just update its UTXO set similarly. If they see another block in the future which is longer, they will undo the previous updates to the UTXO set and make updates as per the longer chain.

        Miners who see both prongs together can chose to extend whatever prong they want. But there is a subtle catch here. The protocol rules say that the chain with the largest amount of work is the canonical chain. So, even if two chains are of equal length (height), they might have different total proof of work. Peter Wuille answers this question here: https://bitcoin.stackexchange.com/questions/5540/what-does-the-term-longest-chain-mean . The comment by Daira Hopwood on Wuille's answer specifies exactly how the "work" of a chain is calculated.

- Assignment 7, Question 2.4 a), why transactions are instantly finalized? We don't need to wait until that a block containing this transaction has been mined?
    - The opening transaction of a payment channel that creates the multi-sig output is already in the blockchain and is finalized. So, the source of the funds is secure. Now, the payment channel is updated by exchanging these commitment transactions that split the opening balance between the two parties, and is always signed by each party in such a way that if the channel were to be closed right now, both parties will get their current share of the opening transaction value. So, in this sense, the moment you have your version of the commitment (channel update) transaction, you can be sure that the money is yours, because you can always submit it to the Bitcoin network and get your share. It's true that this transaction has to be finalized in a mined block on the main blockchain, but there is no risk of double spend attacks here, because the multisig opening transaction output cannot be spent soley by the counter-party.
- Assignment 7, Question 2.4, c) the question says that "Bitcoin also allows to define timelocks relative to the time the spent outputs were created." What does this mean?
    - Bitcoin allows you to specificy outputs that can be spent only after some unix timestamp, or after a known number of blocks have been mined after the block in which the said transaction was mined. In that sense, it allows both “absolute” timelocks, and “relative timelocks”. Google for words like cltv, csv, nlocktime, etc.

Chapter 23: Game Theory

- assignment 11, 1.3 : what is the cost of accessing the file in case your demand is not 1 and you have to go through several nodes ? Your demand * (sum of costs) or do you have to consider that the middle nodes asked first and consider their demand (something like costv*demandv + costw*demandw). In the first example I thought that the only NE was (1,0,0) because the cost for w if u caches is 5/6…
    - The model in short again: Every node i caches or does not cache (Y_i = 1 or 0). If it caches, its’ cost is \alpha_i, otherwise it’s c*d_i where c is the length of the shortest path to a node that caches. The demand of other nodes, or any ordering does not come into play.
    - “the cost for w if u caches is 5/6…”: please verify that the cost for w if only u caches is 8 * ⅓ > \alpha_w = 2
    - There is a typo in the solution sheet: D_w = {v~~, u~~}, sorry for the confusion.

Chapter 24: Distributed Storage

- What is the motivation of k in the algo 24.1 "Consistent Hashing", instead of just having one specific hash function in use?
    - If we use k=1 then every item will be stored exactly once. If we want to store it more often we use a larger k.
- Can you quickly come back on DHT, I'm having a hard time understanding it.
    - The idea is that we have machines that can "move around" in the hypergraph, or in other words, change their neighbourhood. This is necessary if a very high number of machines is constantly failing. If we simply assign one machine to a node of a graph (e.g., 8 machines to the 8 corners of a 3D cube), then the graph can be "broken" pretty quickly. If we now just assign multiple physical machines to each node of the cube, the graph is still easy to be broken by an attacker, because she just needs to target a single corner of the cube. In the DHT (the virtual hypercube example in the script), the neighbouring nodes can help the node under attack by "sending" over machines. This makes it much more robust, i.e., more machines can fail in a worst case manner before either data is lost or the hypergraph becomes partitioned into subgraphs that cannot communicate anymore.
- solution of assignment 11 2.3: I do agree with the master solutions, but since it is written in the exercise that the iterative hashing method is used in practice as well ("There are several constructions for these hash functions, the most common being iterative hashing and salted hashing."), there must be a benefit for choosing it over the salted hashing, isn't it? But anyway, the question is actually more about pro/cons of the two different approaches.
    - An example of where iterative hashing is useful is for searching substrings in a text. Computing the hashes of, e.g., "hall" and "allo" is done consecutively, and since only one character changes, this can be done in constant time with iterative hashing. This can significantly speed up the search. Also see https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm

Chapter 25: Authenticated Agreement

- Let's say View Change occurred, so the algorithm 25.22 is being executed. If I understand the lecture right, the Pb set contains ALL prepare certificates the node b has collected from the beginning of time (since the system was started). With this assumption I don't understand the protocol and see several contradictions later on:
    - Alg. 25.22 Remark: "It is possible that V contains a prepared-certificate for a sequence number s while it does not contain one for some sequence number s` < s) - why? V is a collection of Pb's from 2f+1 nodes, and for each node Pb contains ALL certificates for the entire time of the system operation, so why can some seq. numbers be missed? No, Pb contains all the prepared certificates that only b has collected and not necessarily all the prepared certificates collected by every correct node. Due to asynchrony, it could happen that a correct node 'u'

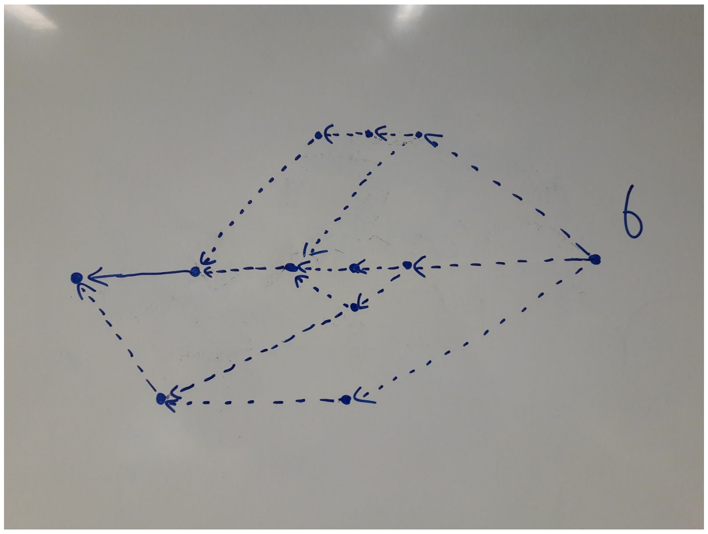
gathers a prepared certificate for s and another correct node ‘v’ gathers a prepared certificate for s’ < s. Then, the view-change messages from the correct nodes in the set V may not contain the view-change message from ‘v’.
    - If V indeed contains all certificates "since the system was started", then all possible (s, v) with s < s^max will qualify for the condition of the Alg. 25.23, line 2, so all pre-prepare messages (with all (s, r) starting 0 to s^max) will be added to O. See answer above.
    - I think something already went wrong in my intuition, since if O contains all pre-prepare messages, everything further is completely meaningless - the backups will just execute all requests from the beginning of time again ;) Again, see answer above.
- For the view change protocol, why we need the set O? O may contain the pre-prepare messages for requests executed in view v (algorithm 25.23, line 2), so since they are already executed, why we need them in the new view and execute them again (algorithm 25.24, line 5)? And why we have to add null-request for the s' which is smaller than s?
    - The set O may also contain pre-prepare messages for requests that have not been executed, since it could happen that a correct node collected a prepared certificate but triggered the view change before it could commit. Alg 25.24, line 5 ensures that these prepare-certificates that could not be executed in the previous view are executed in the new view. It could also happen that a request is re-executed but that is not a problem for correctness as long as every correct node (re-)execute the requests in the same order. One could optimize this away by including snapshots, but the lecture notes present a simplified version only.

      The null-request is a stand-in for no-operation and the primary p must sign the message so that the nodes agree later to do a no-operation for the corresponding sequence number.
- Assignment 12, 1.1 d), why omitting prepared-certificates for requests that no correct node executed cannot harm correctness of the system? The solution to c) is just an example, we could have a request r which some node collects a prepared-certificate of it but no nodes executed it. If the prepared-certificate for this request is omitted, then in the new view we still will not execute it, why it's not a problem?
    - It’s not a problem since for correctness we only need to ensure that requests are executed in the same order by every correct node and not that every request for which a prepared certificate is collected is executed.

Chapter 26: Advanced Blockchain

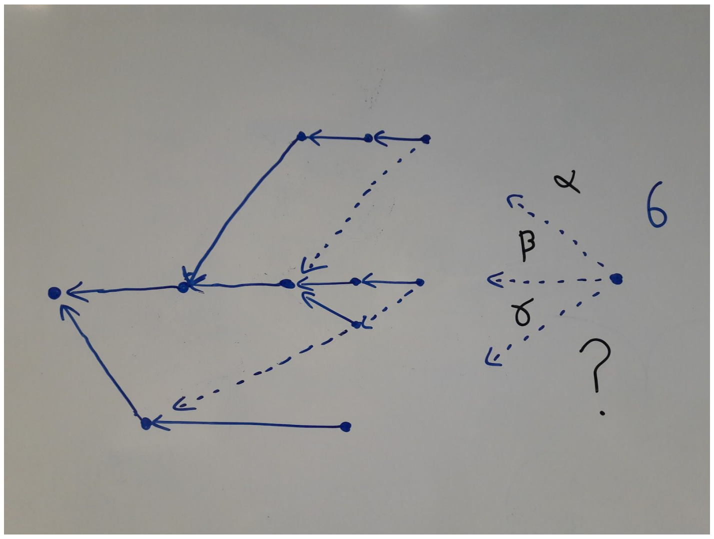
- Def 26.9 : How exactly do you find weights ? How do you know the number of tree descendant of a ?
    - Short answer: You construct the tree recursively and count the descendants.
    - Long answer: Given a block b, we look at all blocks reachable from it (dag-ancestors).

(Forgot to dash one edge...)

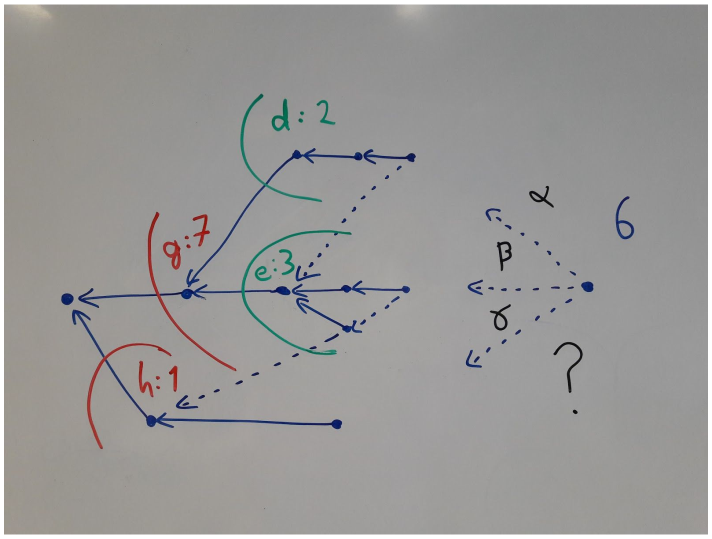


Order the references for other blocks just as we are now doing for b, and each block becomes a tree-child of the first reference.

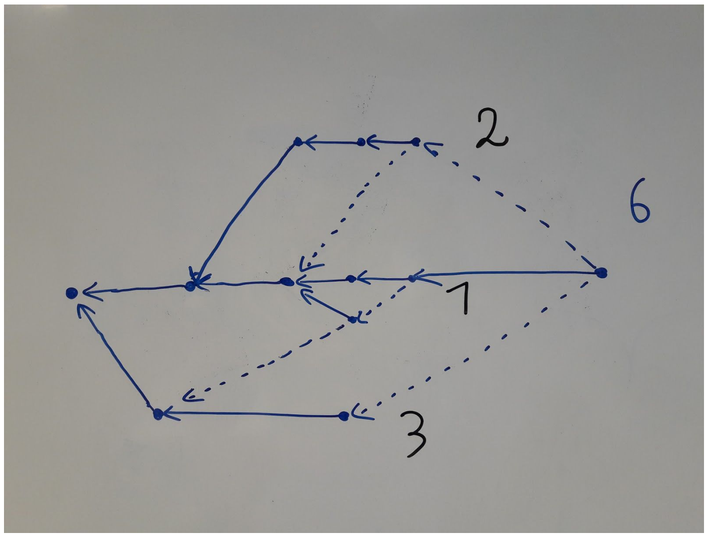
Tree edges are steady, and non-tree edges are dashed.



Compare b’s parents. Below you have the number of tree-descendants written for some blocks (in colour).



g has 7 tree-descendants and h has 1, so gamma is the last reference of b. e has 3 and d has 2, so beta becomes the first reference, alpha second, and gamma third.



Unknown Chapter

- Was bedütet das omega(D)?
  - O-notation: Complexity is at least on the order of D. (https://en.wikipedia.org/wiki/Big_O_notation#The_Knuth_definition)