Chapter 6

Eventual Consistency & Bitcoin

Notes:

- First I discuss the ATM example, and the CAP theorem (all very slowly, can be done in about 15'). Then I present the bitcoin addresses, transactions, network, twix, doublespends. Then 45' are over, and the break is here.

- After the break the leaders, blocks, then proof-of-work, blockchain, forks, reorg. Students ask many questions along the way. In the end I have about 15' left to explain micropayment channels, and the network, which is probably not quite enough. I don't do the other weak consistency definitions at the end. Sometimes I do Decker/MtGox, sometimes not.

- Maybe weak and strong consistency should be its own chapter? Not sure, but clearly Bitcoin by itself is good and plenty.

How would you implement an ATM? Does the following implementation work satisfactorily?

Algorithm 6.1 Naïve ATM

```
1: ATM makes withdrawal request to bank
2: ATM waits for response from bank
3: if balance of customer sufficient then
4:     ATM dispenses cash
5: else
6:     ATM displays error
7: end if
```

Remarks:

- A connection problem between the bank and the ATM may block Algorithm 6.1 in Line 2.

- A *network partition* is a failure where a network splits into at least two parts that cannot communicate with each other. Intuitively any non-trivial distributed system cannot proceed during a partition *and* maintain consistency. In the following we introduce the tradeoff between consistency, availability and partition tolerance.
- There are numerous causes for partitions to occur, e.g., physical disconnections, software errors, or incompatible protocol versions. From the point of view of a node in the system, a partition is similar to a period of sustained message loss.

6.1 Consistency, Availability and Partitions

Definition 6.2 (Consistency)**.** *All nodes in the system agree on the current state of the system.*

Definition 6.3 (Availability)**.** *The system is operational and instantly processing incoming requests.*

Definition 6.4 (Partition Tolerance)**.** *Partition tolerance is the ability of a distributed system to continue operating correctly even in the presence of a network partition.*

Theorem 6.5 (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. □

Algorithm 6.6 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
```

Remarks:

- Algorithm 6.6 is partition tolerant and available since it continues to process requests even when the bank is not reachable.
- The ATM's local view of the balances may diverge from the balances as seen by the bank, therefore consistency is no longer guaranteed.

- The algorithm will synchronize any changes it made to the local balances back to the bank once connectivity is re-established. This is known as eventual consistency.

Definition 6.7 (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 example of eventual consistency is the Bitcoin cryptocurrency system.

6.2 Bitcoin

Definition 6.8 (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.*

Remarks:

- The lack of structure is intentional: it ensures that an attacker cannot strategically position itself in the network and manipulate the information exchange. Information is exchanged via a simple broadcasting protocol.

Definition 6.9 (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.*

Remarks:

- The terms public key and address are often used interchangeably, since both are public information. The advantage of using an address is that its representation is shorter than the public key.
- It is hard to link addresses to the user that controls them, hence Bitcoin is often referred to as being *pseudonymous*.
- 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 Bitcoin network collaboratively tracks the balance in bitcoins of each address.

- 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".

- The hashing algorithm produces addresses of size 20 bytes. This means that there are 2^{160} distinct addresses. It might be tempting to brute force a target address, however at one billion trials per second one still requires approximately 2^{45} years in expectation to find a matching private/public key pair. Due to the birthday paradox the odds improve if instead of brute forcing a single address we attempt to brute force any address. While the odds of a successful trial increase with the number of addresses, lookups become more costly.

Definition 6.10 (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.

Definition 6.11 (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 6.12 (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.

Algorithm 6.13 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
```

Remarks:

- 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 2.10), 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 6.19)
- Notice that so far we only described a local acceptance policy. Nothing prevents nodes to locally accept different transactions that spend the same output.
- 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*.

Definition 6.14 (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 6.13. 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 6.15 (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 6.16 (Bitcoin PoW function)**.** *The Bitcoin PoW function is given by*

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

Remarks:

- This function concatenates the challenge c and nonce x, and hashes them twice using SHA256. The output of SHA256 is a cryptographic hash with a numeric value in $\{0, \ldots, 2^{256} - 1\}$ which is compared to a target value $\frac{2^{224}}{d}$, which gets smaller with increasing difficulty.
- SHA256 is a cryptographic hash function with pseudorandom output. No better algorithm is known to find a nonce x such that the function $\mathcal{F}_d(c, x)$ returns true than simply iterating over possible inputs. This is by design to make it difficult to find such an input, but simple to verify the validity once it has been found.
- If the PoW functions of all nodes had the same challenge, the fastest node would always win. However, as we will see in Definition 6.19, each node attempts to find a valid nonce for a node-specific challenge.

Definition 6.17 (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 6.18 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 $b_t = (memory\ pool, b_{t-1}, x)$

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.
- 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 6.19 (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 sum of the fees of transactions confirmed in the block.*

Remarks:

- A reward transaction is the sole exception to the rule that the sum of inputs must be at least the sum of outputs.
- The number of bitcoins that are minted by the reward transaction and assigned to the miner is determined by a subsidy schedule that is part of the protocol. Initially the subsidy was 50 bitcoins for every block, and it is being halved every 210,000 blocks, or 4 years in expectation. Due to the halving of the block reward, the total amount of bitcoins in circulation never exceeds 21 million bitcoins.
- It is expected that the cost of performing the PoW to find a block, in terms of energy and infrastructure, is close to the value of the reward the miner receives from the reward transaction in the block.

Definition 6.20 (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.*

Remarks:

- The path length from the genesis block to block b is the height h_b.
- Only the longest path from the genesis block to a leaf is a valid transaction history, since branches may contradict each other because of doublespends.
- 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.
- The mining incentives quickly increased the difficulty of the PoW mechanism: initially miners used CPUs to mine blocks, but CPUs were quickly replaced by GPUs, FPGAs and even application specific integrated circuits (AS-ICs) as bitcoins appreciated. This results in an equilibrium today in which only the most cost efficient miners, in terms of hardware supply and electricity, make a profit in expectation.
- 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.

Algorithm 6.21 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**

Remarks:

- Algorithm 6.21 describes how a node updates its local state upon receiving a block. Notice that, like Algorithm 6.13, this describes the local policy and may also result in node states diverging, i.e., by accepting different blocks at the same height as current head.
- 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.

- 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 6.22. *Forks are eventually resolved and all nodes eventually agree on which is the longest blockchain. The system therefore guarantees eventual consistency.*

Proof. In order for the fork to continue to exist, pairs of blocks need to be found in close succession, extending distinct branches, otherwise the nodes on the shorter branch would switch to the longer one. The probability of branches being extended almost simultaneously decreases exponentially with the length of the fork, hence there will eventually be a time when only one branch is being extended, becoming the longest branch. □

6.3 Smart Contracts

Definition 6.23 (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.*

Remarks:

- 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 spend the funds associated with an output and when.

Definition 6.24 (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.*

Remarks:

- Transactions with a timelock are not released into the network until the timelock expires. It is the responsibility of the node receiving the transaction to store it locally until the timelock expires and then release it into the network.
- 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 6.25 (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.*

Remarks:

- 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.

Algorithm 6.26 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

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.

Algorithm 6.27 Simple Micropayment Channel from S to R with capacity c

1: $c_S = c$, $c_R = 0$
2: S and R use Algorithm 6.26 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

Remarks:

- Algorithm 6.27 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.

6.4 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 6.28 (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.*

Remarks:

- Users are annoyed if they receive a notification about a comment on an online social network, but are unable to reply because the web interface does not show the same notification yet. In this case the notification acts as the first read operation, while looking up the comment on the web interface is the second read operation.

Definition 6.29 (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).*

Remarks:

- The ATM must replay all operations in order, otherwise it might happen that an earlier operation overwrites the result of a later operation, resulting in an inconsistent final state.

Definition 6.30 (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 6.31 (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.*

Remarks:

- The first rule ensures that writes by a single node are seen in the same order. For example if a node writes a value in one variable and then signals that it has written the value by writing in another variable. Another node could then read the signalling variable but still read the old value from the first variable, if the two writes were not causally related.

Definition 6.32 (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.*

Chapter Notes

The CAP theorem was first introduced by Fox and Brewer [FB99], although it is commonly attributed to a talk by Eric Brewer [Bre00]. It was later proven by Gilbert and Lynch [GL02] for the asynchronous model. Gilbert and Lynch also showed how to relax the consistency requirement in a partially synchronous system to achieve availability and partition tolerance.

Bitcoin was introduced in 2008 by Satoshi Nakamoto [Nak08]. Nakamoto is thought to be a pseudonym used by either a single person or a group of people; it is still unknown who invented Bitcoin, giving rise to speculation and conspiracy theories. Among the plausible theories are noted cryptographers Nick Szabo [Big13] and Hal Finney [Gre14]. The first Bitcoin client was published shortly after the paper and the first block was mined on January 3, 2009. The genesis block contained the headline of the release date's The Times issue "*The Times 03/Jan/2009 Chancellor on brink of second bailout for banks*", which serves as proof that the genesis block has been indeed mined on that date, and that no one had mined before that date. The quote in the genesis block is also thought to be an ideological hint: Bitcoin was created in a climate of financial crisis, induced by rampant manipulation by the banking sector, and Bitcoin quickly grew in popularity in anarchic and libertarian circles. The original client is nowadays maintained by a group of independent core developers and remains the most used client in the Bitcoin network.

Central to Bitcoin is the resolution of conflicts due to doublespends, which is solved by waiting for transactions to be included in the blockchain. This however introduces large delays for the confirmation of payments which are undesirable in some scenarios in which an immediate confirmation is required. Karame et al. [KAC12] show that accepting unconfirmed transactions leads to a non-negligible probability of being defrauded as a result of a doublespending attack. This is facilitated by *information eclipsing* [DW13], i.e., that nodes do not forward conflicting transactions, hence the victim does not see both transactions of the doublespend. Bamert et al. [BDE+13] showed that the odds of detecting a doublespending attack in real-time can be improved by connecting to a large sample of nodes and tracing the propagation of transactions in the network.

Bitcoin does not scale very well due to its reliance on confirmations in the blockchain. A copy of the entire transaction history is stored on every node in order to bootstrap joining nodes, which have to reconstruct the transaction history from the genesis block. Simple micropayment channels were introduced by Hearn and Spilman [HS12] and may be used to bundle multiple transfers between two parties but they are limited to transferring the funds locked into the channel once. Recently Duplex Micropayment Channels [DW15] and the Lightning Network [PD15] have been proposed to build bidirectional micropayment channels in which the funds can be transferred back and forth an arbitrary number of times, greatly increasing the flexibility of Bitcoin transfers and enabling a number of features, such as micropayments and routing payments between any two endpoints.

This chapter was written in collaboration with Christian Decker.

Bibliography

[BDE+13] Tobias Bamert, Christian Decker, Lennart Elsen, Samuel Welten, and Roger Wattenhofer. Have a snack, pay with bitcoin. In *IEEE Internation Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, 2013.

[Big13] John Biggs. Who is the real satoshi nakamoto? one researcher may have found the answer. http://on.tcrn.ch/l/R0vA, 2013.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 2000.

[DW13] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE International Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, September 2013.

[DW15] Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2015.

[FB99] Armando Fox and Eric Brewer. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems*. IEEE, 1999.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[Gre14] Andy Greenberg. Nakamoto's neighbor: My hunt for bitcoin's creator led to a paralyzed crypto genius. http://onforb.es/1rvyecq, 2014.

[HS12] Mike Hearn and Jeremy Spilman. Contract: Rapidly adjusting micro-payments. https://en.bitcoin.it/wiki/Contract, 2012. Last accessed on November 11, 2015.

[KAC12] G.O. Karame, E. Androulaki, and S. Capkun. Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin. In *Conference on Computer and Communication Security (CCS)*, 2012.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[PD15] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network. 2015.

Chapter 7

Advanced Blockchain

In this chapter we study various advanced blockchain concepts, which are popular in research.

7.1 Selfish Mining

Satoshi Nakamoto suggested that it is rational to be altruistic, e.g., by always attaching newly found block to the longest chain. But is it true?

Definition 7.1 (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.*

Algorithm 7.2 Selfish Mining

```
1: Idea: Mine secretly, without immediately publishing newly found blocks
2: Let d_p be the depth of the public blockchain
3: Let d_s be the depth of the secretly mined blockchain
4: if a new block b_p is published, i.e., d_p has increased by 1 then
5:    if d_p > d_s then
6:       Start mining on that newly published block b_p
7:    else if d_p = d_s then
8:       Publish secretly mined block b_s
9:       Mine on b_s and publish newly found block immediately
10:   else if d_p = d_s − 1 then
11:      Publish both secretly mined blocks
12:   end if
13: end if
```

Remarks:

- If the selfish miner is more than two blocks ahead, the original research suggested to always answer a newly published block by releasing the oldest unpublished block. The idea is that honest miners will then split their mining power between these two blocks. However, what matters is how long it takes the honest miners to find the next block,

to extend the public blockchain. This time does not change whether the honest miners split their efforts or not. Hence the case $d_p < d_s - 1$ is not needed in Algorithm 7.2.

Theorem 7.3 (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*

$$\frac{\alpha(1-\alpha)^2(4\alpha+\gamma(1-2\alpha))-\alpha^3}{1-\alpha(1+(2-\alpha)\alpha)}.$$

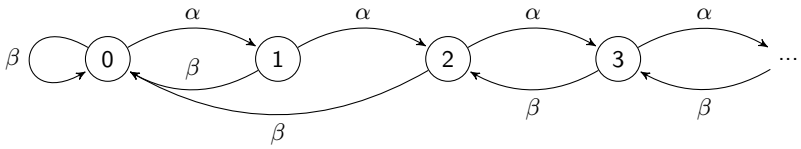

Figure 7.4: Each state of the Markov chain represents how many blocks the selfish miner is ahead, i.e., $d_s - d_p$. In each state, the selfish miner finds a block with probability α, and the honest miners find a block with probability $\beta = 1 - \alpha$. The interesting cases are the "irregular" β arrow from state 2 to state 0, and the β arrow from state 1 to state 0 as it will include three subcases.

Proof. We model the current state of the system with a Markov chain, see Figure 7.4.

We can solve the following Markov chain equations to figure out the probability of each state in the stationary distribution:

$$\begin{aligned} p_1 &= \alpha p_0 \\ \beta p_{i+1} &= \alpha p_i, \text{ for all } i > 1 \\ \text{and } 1 &= \sum_i p_i. \end{aligned}$$

Using $\rho = \alpha/\beta$, we express all terms of above sum with p_1:

$$1 = \frac{p_1}{\alpha} + p_1 \sum_{i \geq 0} \rho^i = \frac{p_1}{\alpha} + \frac{p_1}{1-\rho}, \text{ hence } p_1 = \frac{2\alpha^2 - \alpha}{\alpha^2 + \alpha - 1}.$$

Each state has an outgoing arrow with probability β. If this arrow is taken, one or two blocks (depending on the state) are attached that will eventually end up in the main chain of the blockchain. In state 0 (if arrow β is taken), the honest miners attach a block. In all states i with $i > 2$, the selfish miner eventually attaches a block. In state 2, the selfish miner directly attaches 2 blocks because of Line 11 in Algorithm 7.2.

State 1 in Line 8 is interesting. The selfish miner secretly was 1 block ahead, but now (after taking the β arrow) the honest miners are attaching a competing block. We have a race who attaches the next block, and where. There are three possibilities:

- Either the selfish miner manages to attach another block to its own block, giving 2 blocks to the selfish miner. This happens with probability α.
- Or the honest miners attach a block (with probability β) to their previous honest block (with probability $1-\gamma$). This gives 2 blocks to the honest miners, with total probability $\beta(1-\gamma)$.
- Or the honest miners attach a block to the selfish block, giving 1 block to each side, with probability $\beta\gamma$.

The blockchain process is just a biased random walk through these states. Since blocks are attached whenever we have an outgoing β arrow, the total number of blocks being attached per state is simply $1+p_1+p_2$ (all states attach a single block, except states 1 and 2 which attach 2 blocks each).

As argued above, of these blocks, $1-p_0+p_2+\alpha p_1-\beta(1-\gamma)p_1$ are blocks by the selfish miner, i.e., the ratio of selfish blocks in the blockchain is

$$\frac{1-p_0+p_2+\alpha p_1-\beta(1-\gamma)p_1}{1+p_1+p_2}.$$

□

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 7.3 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.

7.2 DAG-Blockchain

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.

Definition 7.5 (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.*

Definition 7.6 (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.*

Theorem 7.7. *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 7.8 (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.*

Remarks:

- In other words, every tree-something is also a dag-something, but not necessarily vice versa.
- Blocks do not specify who is their tree-parent, or the order of their dag-parents. Instead, tree-parents are implicitly defined as follows.

Definition 7.9 (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 7.10 (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 7.11 (Tree-Parent)**.** *The tree-parent of b is the first dag-parent in b's parent order.*

Remarks:

- Now we can totally order all the blocks in the DAG-Blockchain.

Theorem 7.13. *Let p be the tree-parent of b. The order of blocks $<_b$ computed by Algorithm 7.12 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. □

Algorithm 7.12 DAG-Blockchain Ordering

We totally order all dag-ancestors of block b as $<_b$ as follows:
Initialize $<_b$ as empty
for all dag-parents p of b, in their parent order **do**
    Compute $<_p$ (recursively)
    Remove from $<_p$ any blocks already included in $<_b$
    Append $<_p$ at the end of $<_b$
end for
Append block b at the end of $<_b$

Remarks:

- 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 7.13 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 7.14 (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).*

7.3 Smart Contracts

Definition 7.15 (Ethereum)**.** *Ethereum is a distributed state machine. Unlike Bitcoin, Ethereum promises to run arbitrary computer programs in a blockchain.*

Remarks:

- Like the Bitcoin network, Ethereum consists of nodes that are connected by a random virtual network. These nodes can join or leave the network arbitrarily. There is no central coordinator.

- Like in Bitcoin, users broadcast cryptographically signed transactions in the network. Nodes collate these transactions and decide on the ordering of transactions by putting them in a block on the Ethereum blockchain.

Definition 7.16 (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.

Definition 7.17 (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.*

Definition 7.18 (Ethereum Transaction)**.** *An Ethereum transaction is sent by a user who controls an EOA to the Ethereum network. A transaction contains:*

- *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.*
- *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.*

Remarks:

- There are three types of transactions.

Definition 7.19 (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 7.20 (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 7.21 (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 7.22 (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.*

Remarks:

- As Ethereum contracts are programs (with loops, function calls, and recursions), end users need to pay more gas for more computations. In particular, smart contracts might call another smart contract as a subroutine, and StartGas must include enough gas to pay for all these function calls invoked by the transaction.
- The product of StartGas and GasPrice is the maximum cost of the entire transaction.
- Transactions are an all or nothing affair. If the entire transaction could not be finished within the StartGas limit, an Out-of-Gas exception is raised. The state of the blockchain is reverted back to its values before the transaction. The amount of gas consumed is not returned back to the sender.

Definition 7.23 (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)*

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.

7.4 Payment Hubs

How to we enable many parties to send payments to each other efficiently?

Definition 7.24 (Payment Hub)**.** *Multiple parties can send payments to each other by means of a payment hub.*

Remarks:

- While we could always call the smart contract to transfer money between users that joined the hub, every smart contract call costs as it involves the blockchain. Rather, we want a frugal system with just few blockchain transactions.

Definition 7.25 (Smart Contract Hub)**.** *A smart contract hub is a payment hub that is realized by a smart contract on a blockchain and an off-chain server. The smart contract and the server together enable off-chain payments between users that joined the hub.*

Remarks:

- The smart contract lives forever, but the server can disappear anytime. If it does, nodes can show their recent balance proofs to the smart contract and withdraw their balances.

- The server can be scaled to in terms of latency and number of users. The smart contract does not need to scale as it only needs to just accept one commitment per epoch.

- In case the server disappears, the smart contract will be flooded with withdrawal requests, and could be subject to delays based on the delays of the underlying blockchain.

Algorithm 7.26 Smart Contract Hub

1: Users join the hub by depositing some native currency of the blockchain into the smart contract
2: Funds of all participants are maintained together as a fungible pool in the smart contract
3: Time is divided into epochs: in each epoch users can send each other payment transactions through the server
4: The server does the bookkeeping of who has paid how much to whom during the epoch
5: At the end of the epoch, the server aggregates all balances into a commitment, which is sent to the smart contract
6: Also at the end of the epoch, the server sends a proof to each user, informing about the current account balance
7: Each user can verify that its balance is correct; if not the user can call the smart contract with its proof to get its money back

7.5 Proof-of-Stake

Almost all of the energy consumption of permissionless (everybody can participate) blockchains is wasted because of proof-of-work. Proof-of-stake avoids these wasteful computations, without going all the way to permissioned (the participating nodes are known a priori) systems such as Paxos or PBFT.

Definition 7.27 (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.*

Remarks:

- Literally, "the rich get richer".
- Ethereum is expected to move to proof-of-stake eventually.
- There are multiple flavors of proof-of-stake algorithms.

Definition 7.28 (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 (a la byzantine fault tolerance, see Chapter 4).

Definition 7.29 (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.*

Remarks:

- Proof-of-stake can be attacked in various ways. Let us discuss the two most prominent attacks.
- Most importantly, there is the "nothing at stake" attack: In blockchains, forks occur naturally. In proof-of-work, a fork is resolved because every miner has to choose which blockchain fork to extend, as it does not pay off to mine on a hopeless fork. Eventually, some chain will end up with more miners, and that chain is considered to be the real blockchain, whereas other (childless) blocks are just not being extended. In a proof-of-stake system, a user can trivially extend all prongs of a fork. As generating a block costs nothing, the miner has no incentive to not extend all the prongs of the fork. This results in a situation with more and more forks, and no canonical order of transactions. If there is a double-spend attack, there is no way to tell which blockchain is valid, as all blockchains are the same length (all miners are extending all forks). It can be argued that honest miners, who want to preserve the value of the network, will extend the first prong of the fork that they see. But that leaves room for a dishonest miner to double spend by moving their mining opportunity to the appropriate fork at the appropriate time.
- Long range attack: As there are no physical resources being used to produce blocks in a proof-of-stake system, nothing prevents a bad player from creating an alternate blockchain starting at the genesis block, and make it longer than the canonical blockchain. New nodes may have difficulties to determine which blockchain is the real established blockchain. In proof-of-work, long range attacks takes an enormous amount of computing power. In proof-of-stake systems, a new node has to check with trusted sources to know what the canonical blockchain is.

Chapter Notes

Selfish mining has already been discussed shortly after the introduction of Bitcoin [RHo10]. A few years later, Eyal and Sirer formally analyzed selfish mining [ES14]. Similarly, Courtois and Bahack [CB14] study subversive mining strategies. Nayak et al. [NKMS15] combine selfish mining and eclipse attacks. Algorithm 7.2 is not optimal for all parameters, e.g., sometimes it may be beneficial to risk even a two-block advantage. Sapirshtein et al. [SSZ15] describe and analyze the optimal algorithm.

Vitalik Buterin introduced Ethereum in the 2013 whitepaper [But13]. In 2014, Ethereum Foundation was founded to create Ethereum's first implementation. An online crowd-sale was conducted to raise around 31,000 BTC (around USD 18 million at the time) for this. In this sense, Ethereum was the first ICO (Initial Coin Offering). Ethereum has also attempted to write a formal specification of its protocol in their yellow paper [Gav18]. This is in contrast to Bitcoin, which doesn't have a formal specification.

Bitcoin's blockchain forms as a chain, i.e., each block (except the genesis block) has a parent block. The longest chain with the highest difficulty is

considered the main chain. GHOST [SZ15] is an alternative to the longest chain rule for establishing consensus in PoW based blockchains and aims to alleviate adverse impacts of stale blocks. Ethereum's blockchain structure is a variant of GHOST. Other systems based on DAGs have been proposed in [SLZ16], [SZ18], [LLX+18], and [LSZ15].

Khalil and Gervais [KG18] introduced the notion of a payment hub that is a combination of a smart contract and an online server. Plasma [JP17] is another family of systems that uses a smart contract on the Ethereum blockchain and one or more off-chain operators to enable off-chain transactions.

Proof of Stake was first introduced in PPCoin [KN12]. The most well known Proof of Stake system is the one being implemented for Ethereum, which involves a transition phase from PoW to PoS [BG17], and finally, on to a more formally constructed PoS [VZ18]. More details are available in this article [Cho18] by Jon Choi.

Bibliography

[BG17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[But13] Vitalik Buterin. A Next-Generation Smart Contract and Decentralized Application Platform, 2013. Available from: `https://github.com/ethereum/wiki/wiki/White-Paper`.

[CB14] Nicolas T. Courtois and Lear Bahack. On subversive miner strategies and block withholding attack in bitcoin digital currency. *CoRR*, abs/1402.1718, 2014.

[Cho18] Jon Choi. Ethereum casper 101. 2018. Available from: `https://medium.com/@jonchoi/ethereum-casper-101-7a851a4f1eb0`.

[ES14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.

[Gav18] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, Byzantium Version, 2018. Available from: `https://ethereum.github.io/yellowpaper/paper.pdf`.

[JP17] Vitalik Buterin Joseph Poon. Plasma: Scalable autonomous smart contracts, 2017.

[KG18] Rami Khalil and Arthur Gervais. Nocust - a non-custodial 2nd-layer financial intermediary. Cryptology ePrint Archive, Report 2018/642, 2018. `https://eprint.iacr.org/2018/642`.

[KN12] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[LLX+18] Chenxing Li, Peilun Li, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. Scaling nakamoto consensus to thousands of transactions per second. *CoRR*, abs/1805.03870, 2018.

[LSZ15] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[NKMS15] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. Technical report, IACR Cryptology ePrint Archive 2015, 2015.

[RHo10] RHorning. Mining cartel attack, 2010.

[SLZ16] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. Cryptology ePrint Archive, Report 2016/1159, 2016. `https://eprint.iacr.org/2016/1159`.

[SSZ15] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. *arXiv preprint arXiv:1507.06183*, 2015.

[SZ15] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[SZ18] Yonatan Sompolinsky and Aviv Zohar. Phantom: A scalable blockdag protocol. Cryptology ePrint Archive, Report 2018/104, 2018. `https://eprint.iacr.org/2018/104`.

[VZ18] Aditya Asgaonkar Georgios Piliouras Vlad Zamfir, Nate Rush. Introducing the "minimal cbc casper" family of consensus protocols. 2018. Available from: `https://github.com/cbc-casper/cbc-casper-paper/blob/master/cbc-casper-paper-draft.pdf`.

Chapter 8

Authenticated Agreement

Notes:

- I start out with the 3-node example of chapter 3, and motivate signatures by showing that 3 nodes can probably solve byzantine agreement, if the byzantine node cannot lie about its input. Then we try to solve binary synchronous authenticated byzantine consensus, using a primary just sending 1_p to all. Long and good discussion what can go wrong. Some suggest to also do 0_p instead of nothing, some want to do it in two rounds. I always ask what can go wrong, until somebody suggests to use $f + 1$ rounds. From there we develop the protocol with the set S, and formally prove it. 10' before break, I start with Zyzzyva, the good cases (well-motivated).

- after the break, I continue Zyzzyva, with arrows. We discuss what complete really means (and show the proof that complete commands must be ordered correctly), and then we discuss what a byzantine primary can do to us. From then we go into view changes, and I explain the way to find the new view (by picture). At the very end I discuss safety and liveness, and quickly mention a few other protocols. This year I have no time to do strong consistency (4.3).

- This chapter still needs a few final brushes, in particular that part $h_new \geq$ "$3f + 1$". I would suggest to really not include 4.3 in the chapter in the future, but I am not sure (have to try again to be sure). also still needed: chapter notes and stuff.

- ATTENTION: I (Roger) do have some notes about this, i.e. what should be changed. It sits in the red DistSys folder, very first page.

- Currently, liveness argument is missing from the chapter as claimed in the original paper [CL+99]. If the delay is finite but not bounded, then it can happen that a request is never satisfied as follows. Basically, the timer value can be too small and it fires even when the primary is correct. The liveness section in the paper suggests to increase the timer value exponentially to avoid triggering the view changes too soon. Even under that assumption it might happen that by the time pre-prepare, prepare, commit messages for a given view are received, the views have changed.

Then, one can schedule the delivery of the delayed pre-prepare, prepare, commit messages. However, these will be discarded since the view has changed. Although the delay of the messages increases exponentially, its still finite as they are delivered. So, the delays must be bounded for liveness after all, as also claimed in Castro's thesis [Cas01].

In Section 5.6 we have already had a glimpse into the power of cryptography. In this Chapter we want to build a *practical* byzantine fault-tolerant system using cryptography. With cryptography, Byzantine lies may be detected easily.

8.1 Agreement with Authentication

Definition 8.1 (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* $\texttt{msg}(x)$ *signed by node* u *with* $\texttt{msg}(x)_u$*.*

Remarks:

- Algorithm 8.2 shows a synchronous agreement protocol for binary inputs relying on signatures. We assume there is a designated "primary" node p that all other nodes know. The goal is to decide on p's value.

Algorithm 8.2 Byzantine Agreement with Authentication

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
```

Theorem 8.3. *Algorithm 8.2 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 minimal among all rounds 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 minimal among all rounds 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 u' 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. □

Remarks:

- The algorithm only takes $f + 1$ rounds, which is optimal as described in Theorem 4.20.
- Using signatures, Algorithm 8.2 solves consensus for any number of failures! Does this contradict Theorem 4.12? Recall that in the proof of Theorem 4.12 we assumed that a byzantine node can distribute contradictory information about its own input. If messages are signed, correct nodes can detect such behavior – a node u signing two contradicting messages proves to all nodes that node u is byzantine.
- Does Algorithm 8.2 satisfy any of the validity conditions introduced in Section 4.1? No! A byzantine primary can dictate the decision value. 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.
- If the primary is a correct node, Algorithm 8.2 only needs two rounds! Can we make it work with arbitrary inputs? Also, relying on synchrony limits the practicality of the protocol. What if messages can be lost or the system is asynchronous?

8.2 Practical Byzantine Fault Tolerance

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 2.10 with byzantine nodes in an asynchronous network. We present a very simple version of it without any optimizations.

Definition 8.4 (System Model)**.** *There are $n = 3f + 1$ nodes and 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 ideas behind PBFT can roughly be summarized as follows:

- Signatures guarantee that every node can determine which node/client generated any given message.
- At any given time, every node will consider one designated node to be the *primary* and the other nodes to be *backups*. Since we are in the variable delay model, requests can arrive at the nodes in different orders. While a primary remains in charge (this timespan corresponds to what is called a *view*), it thus has the function of a serializer (cf. Algorithm 2.11).
- If backups detect faulty behavior in the primary, they start a new view and the next node in round-robin order becomes primary. This is called a *view change*.
- After a view change, a correct new primary makes sure that no two correct nodes execute requests in different orders. Exchanging information will enable backups to determine if the new primary acts in a byzantine fashion.

Definition 8.5 (View)**.** *A **view** is represented locally at each node i by a non-negative integer v (we say i **is in view** v) that is incremented by one whenever the node changes to a different view.*

Definition 8.6 (Primary; Backups)**.** *A node that is in view v considers node $v \bmod n$ to be the **primary** and all other nodes to be **backups**.*

Definition 8.7 (Sequence Number)**.** *During a view, a node relies on the primary to pick consecutive integers as **sequence numbers** that function as indices in the global order (cf. Definition 2.10) for the requests that clients send.*

Remarks:

- All nodes start out in view 0 and can potentially be in different views (i.e. have different local values for v) at any given time.
- 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 $r' \neq r$ with sequence number s, not unlike Lemma 2.16.
- Correct primaries choose sequence numbers such that they are *dense*, 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.

Definition 8.8 (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, whose components can be validated in the same way, and that also belong to view v.*

Lemma 8.9 (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. □

8.3 PBFT: Agreement Protocol

First we describe how PBFT achieves agreement on a unique order of requests within a view.

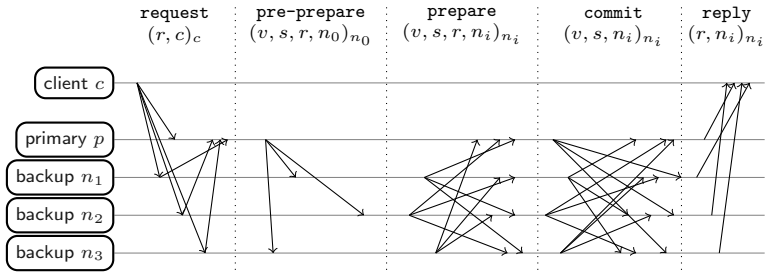

Figure 8.10: The agreement protocol used in PBFT for processing a client request, exemplified for a system with 4 nodes. Node n_0 is the primary in current view v. Time runs from left to right. Messages sent at the same time need not arrive at the same time.
indexrequest

Remarks:

- Figure 8.10 shows how the nodes come to an agreement on a sequence number for a client request. Informally, the protocal has these three steps:

  1. The primary sends a `pre-prepare`-message to all backups, informing them that he wants to execute that request with the sequence number specified in the message.
  2. Backups send `prepare`-messages to all nodes, informing them that they agree with that suggestion.
  3. All nodes send `commit`-messages to all nodes, informing everyone that they have committed to execute the request with that sequence number. They execute the request and inform the client.

- Figure 8.10 shows that all nodes can start each phase at different times.
- To make sure byzantine nodes cannot force the execution of a request, every node waits for a certain number of `prepare`- and `commit`-messages with the correct content before executing the request.
- Definitions 8.11, 8.14, 8.16 specify the agreement protocol formally. Backups run Phases 1 and 2 concurrently.

Definition 8.11 (PBFT Agreement Protocol Phase 1; Pre-Prepared Primary)**.** *In **phase 1** of the agreement protocol, the nodes execute Algorithm 8.12.*

Algorithm 8.12 PBFT Agreement Protocol: Phase 1

Code for primary p in view v:

1: accept $\texttt{request}(r, c)_c$ that originated from client c
2: pick next sequence number s
3: send $\texttt{pre-prepare}(v, s, r, p)_p$ to all backups

Code for backup b:

4: accept $\texttt{request}(r, c)_c$ from client c
5: relay $\texttt{request}(r, c)_c$ to primary p

Definition 8.13 (Faulty-Timer)**.** *When backup b accepts request r in Algorithm 8.12 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. We explain the view change protocol in Section 8.4.
- We leave out the details regarding for what timespan to set the faulty-timer as they are an optimization with several trade-offs to consider; the interested reader is advised to consult [CL+99].

Definition 8.14 (PBFT Agreement Protocol Phase 2; Pre-prepared Backups)**.** *In phase 2 of the agreement protocol, every backup b executes Algorithm 8.15. Once it has sent the* `prepare`*-message, b has **pre-prepared** r for (v, s).*

Definition 8.16 (PBFT Agreement Protocol Phase 3; Prepared-Certificate)**.** *A node n_i that has pre-prepared a request executes Algorithm 8.17. 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**.*

Algorithm 8.15 PBFT Agreement Protocol: Phase 2

Code for backup b in view v:

1: accept `pre-prepare`$(v, s, r, p)_p$
2: **if** p is primary of view v and b has not yet accepted a `pre-prepare`-message for (v, s) and some $r' \neq r$ **then**
3: send `prepare`$(v, s, r, b)_b$ to all nodes
4: **end if**

Algorithm 8.17 PBFT Agreement Protocol: Phase 3

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 (including n_i's own message, if it is a backup)
2: send `commit`$(v, s, n_i)_{n_i}$ to all nodes
3: wait until $2f + 1$ `commit`-messages (including n_i's own) matching (v, s) have been accepted
4: execute request r once all requests with lower sequence numbers have been executed
5: send `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 8.17 Line 4 until a request has been executed for all previous sequence numbers.

- The client only considers the request to have been processed once it received $f + 1$ `reply`-messages sent by the nodes in Algorithm 8.17 Line 5. Since a correct node only sends a `reply`-message once it executed the request, with $f + 1$ `reply`-messages the client can be certain that the request was executed by a correct node.

- We will see in Section 8.4 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.

- 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 8.12 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.

Lemma 8.18 (PBFT: Unique Sequence Numbers within View)**.** *If a node gathers a prepared-certificate for (v, s, r), then no node can gather a prepared-certificate for (v, s, r') with $r' \neq r$.*

Proof. Assume two (not necessarily distinct) nodes gather 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 8.9. A correct primary only sends a single `pre-prepare`-message for each (v, s), see Algorithm 8.12 Lines 2 and 3. A correct backup only sends a single `prepare`-message for each (v, s), see Algorithm 8.15 Lines 2 and 3. Thus, $r' = r$. □

Remarks:

- Due to Lemma 8.18, once a node has a prepared-certificate for (v, s, r), no correct node will execute some $r' \neq r$ with sequence number s during view v because correct nodes wait for a prepared-certificate before executing a request (cf. Algorithm 8.17).
- However, that is not yet enough to make sure that no $r' \neq r$ will be executed by a correct node with sequence number s during some later view $v' > v$. How can we make sure that that does not happen?

8.4 PBFT: View Change Protocol

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. Nodes use their faulty-timer (and only that!) to decide whether they consider the primary to be faulty (cf. Definition 8.13).

Remarks:

- 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.
- How can we guarantee that this happens?

Definition 8.19 (PBFT: View Change Protocol)**.** *In the view change protocol, a node whose faulty-timer has expired enters the **view change phase** by running Algorithm 8.22. During the **new view phase** (which all nodes continually listen for), the primary of the next view runs Algorithm 8.23 while all other nodes run Algorithm 8.24.*

Remarks:

- The idea behind the view change protocol is this: during the view change protocol, the new primary gathers prepared-certificates from $2f + 1$ nodes, so for every request that some correct node executed, the new primary will have at least one prepared-certificate.
- After gathering that information, the primary distributes it and tells all backups which requests need to be to executed with which sequence numbers.

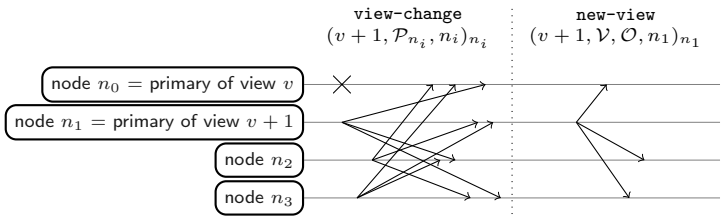

Figure 8.20: The view change protocol used in PBFT. Node n_0 is the primary of current view v, node n_1 the primary of view $v+1$. Once backups consider n_0 to be faulty, they start the view change protocol (cf. Algorithms 8.22, 8.23, 8.24). The X signifies that n_0 is faulty.

- Backups can check whether the new primary makes the decisions required by the protocol, and if it does not, then the new primary must be byzantine and the backups can directly move to the next view change.

Definition 8.21 (New-View-Certificate)**.** $2f+1$ `view-change`*-messages for the same view v form a* ***new-view-certificate****.*

Algorithm 8.22 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

Algorithm 8.23 PBFT View Change Protocol: New View Phase - Primary

Code for 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 $\mathcal{V}$
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 8.12 starting from sequence number $s^{\mathcal{V}}_{max}+1$

Remarks:

- It is possible that $\mathcal{V}$ contains a prepared-certificate for a sequence number s while it does not contain one for some sequence number $s' < s$. For each such sequence number s', we fill up $\mathcal{O}$ in Algorithm 8.23 Line 4 with `null`-requests, i.e. requests that backups understand to mean "do not do anything here".

Algorithm 8.24 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 8.22 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 8.23 Lines 2 and 4 **then**
6: respond to all `pre-prepare`-messages in $\mathcal{O}$ as in the agreement protocol, starting from Algorithm 8.15
7: start accepting messages for view $v+1$
8: **else**
9: trigger view change to $v+2$ using Algorithm 8.22
10: **end if**
11: **end if**

Theorem 8.25 (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 8.18 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 for s and a different $r' \neq r$.

Consider the new-view-certificate $\mathcal{V}$ (see Algorithm 8.23 Line 1). If any correct node executed request r with sequence number s, then due to Algorithm 8.17 Line 3, 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 8.17 Line 1.

The new-view-certificate contains `view-change`-messages from a set R_2 of $2f+1$ nodes. Thus according to Lemma 8.9, 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 8.23 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.

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 8.24 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 8.24 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 8.15 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. □

Remarks:

- We have shown that PBFT protocol guarantees safety or nothing bad ever happens, i.e., the correct nodes never disagree on requests that were commited with the same sequence numbers. But, does PBFT also guarantee liveness, i.e., a legitimate client request is eventually committed and receives a reply.

- To prove liveness, we make an additional assumption that message delays are finite and bounded. With infinite message delays in an asynchronous system and even one faulty (byzantine) process, it is impossible to solve consensus with guaranteed termination [FLP85].

- A faulty new primary could delay the system indefinitely by never sending a `new-view`-message. To prevent this, as soon as a node sends its `view-change`-message for $v + 1$, it starts its faulty-timer and stops it once it accepts a `new-view`-message for $v + 1$. If the timer runs out before being stopped, the node triggers another view change.

- However, the timer doubles to trigger the next view change because the message delays might be larger. Eventually, the timer values are larger than the message delays and the messages are received before the timer expires.

- Since at most f consecutive primaries can be faulty, the system makes progress after at most $f + 1$ view changes.

- We described a simplified version of PBFT; any practically relevant variant makes adjustments to what we presented. The references found in the chapter notes can be consulted for details that we did not include.

Chapter Notes

PBFT is perhaps the central protocol for asynchronous byzantine state replication. The seminal first publication about it, of which we presented a simplified version, can be found in [CL+99]. The canonical work about most versions of PBFT is Miguel Castro's PhD dissertation [Cas01]. Barbara Liskov was Miguel Castro's advisor, and show won a Turing award, partially because of her work on PBFT.

Notice that the sets $\mathcal{P}_b$ in Algorithm 8.22 grow with each view change as the system keeps running since they contain all prepared-certificates that nodes have collected so far. All variants of the protocol found in the literature introduce regular *checkpoints* where nodes agree that enough nodes executed all requests up to a certain sequence number so they can continuously garbage-collect prepared-certificates. We left this out for conciseness.

Remember that all messages are signed. Generating signatures is somewhat pricy, and variants of PBFT exist that use the cheaper, but less powerful Message Authentication Codes (MACs). These variants are more complicated because MACs only provide authentication between the two endpoints of a message and cannot prove to a third party who created a message. An extensive treatment of a variant that uses MACs can be found in [CL02].

Before PBFT, byzantine fault-tolerance was generally considered impractical, just something academics would be interested in. PBFT changed that as it showed that byzantine fault-tolerance can be practically feasible. As a result, numerous asynchronous byzantine state replication protocols were developed. Other well-known protocols are Q/U [AEMGG+05], HQ [CML+06], and Zyzzyva [KAD+07]. An overview over the relevant literature can be found in [AGK+15].

This chapter was written in collaboration with Georg Bachmeier.

Bibliography

[AEMGG+05] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74. ACM, 2005.

[AGK+15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[Cas01] Miguel Castro. *Practical Byzantine Fault Tolerance*. Ph.d., MIT, January 2001. Also as Technical Report MIT-LCS-TR-817.

[CL+99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.