Permissionless Blockchains: Bitcoin and Proof of Work

19/3/2021

1 Model definitions

Within the model of the original Nakamoto's paper [Nak09], we consider two kinds of processes: *clients* and *nodes* [1].

Assumption 1 (Synchrony)**.** *We assume a* synchronous *network, i.e., every message sent by a process is delivered within a known* a priori *delay. The important assumption is that this delay is negligible compared to the other scale of time, which we denote as "delay between blocks", that we will define.*

Permissionless membership. Processes can join or leave the protocol at any time. A physical entity can emulate as many clients and/or nodes as it wants for free [2]. The useful observation for what follows, is that the total computing power of the emulated processes, is lower than the computing power of the physical entity emulating them.

2 Object implemented: a ledger

The objective of the Bitcoin protocol is to provide a linearizable implementation of the following distributed object, denoted as a *ledger*.

Definition 2. *A ledger is an ordered set of values*

$$Ledger = \{v_1 < v_2 < \cdots < v_m\}. \tag{1}$$

which is accessible by the clients by two operations: read *the contents of the ledger and* append *a value.*

Recall that the definition of linearizability ([HW90]) implies that the operations applied to the implemention of a ledger can be put in a total order respecting this sequential behavior. Moreover, this order should respect the real-time relation across operations: if o_1 returns before o_2 was invoked, then o_2 cannot be ordered before o_1. The *liveness* guarantee implied is that every read or append invocation performed by an honest client

[1] Clients are sometimes also called light nodes or wallets, while nodes are sometimes also called miners, replicas or full nodes.

[2] In practice, an entity running a node wants to run also a client, in order to receive rewards for its work. [Nak09] even recommends to create one new client per transaction received, in order to enhance privacy

eventually returns. Let us however examplify more concrete consequences of these definitions.

The *liveness* guarantee implies that for every append(tx) done by a client, for some input tx (called a "transaction") then after a variable finite time, every client doing read will eventually be returned a ledger containing append(tx). More precisely, we will see that the Bitcoin protocol achieves liveness, as long as, on average, the total size of values for which a append request is made per 10 minutes, is lower than 1Mb (=the size of one block). In practice, the Bitcoin system manages to respect this limitation by imposing clients to pay a transaction fee per append (currently 20$), which is adjusted depending on the demand.

The other guarantee is implied by safety and is called *consistency*. It states that if two clients read the ledger, then there must be one client who is returned a *prefix* of what is returned to the other client. In particular, we cannot have that one client reads $v_1 < \cdots < v_6$ while the other client reads $v_1 < \cdots < v'_6$. As we will see, the Bitcoin protocol achieves those safety conditions, up to some probability of failure. The concrete parameters aim at making small this probability of failure.

3 Bitcoin protocol: Implementing a ledger with a tree of blocks

3.1 Proof of work: data structures that are difficult to compute

Values are organized in trees of blocks. A "block" B_i consists in

(2)
$$B_i = \begin{cases} \mathcal{V}_i & = \{v_{i,1} < v_{i,2} < \dots\} \text{ a certain number of totally ordered values (of total size } <\text{1Mb)} \\ m_i & = \text{some metadata, including an adjustable bitstring: the "nonce"} \end{cases}$$

The number of values in a block is chosen so that the block's size is around 1Mb. Blocks are *partially ordered* forming a *tree*: see Figure 3.1. All possible trees have the same common root: the *genesis* block B_0, which specifically contains no value, and a metadata m_0 equal to : "Times 03/Jan/2009 Chancellor on brink of second bailout for banks". Partial order means that we have total order between the consecutive blocks in a branch, e.g.:

$$B_0 < B'_1 < B'_2 < B'_3 \ .$$

But we do not have any order relation between blocks that are on different branches, and which we call *conflicting* blocks. For instance B_2 and in B_2". Actually, it can happen that the same value v appears both in B_2 and B_2".

The height of a block $B"_m$ in the tree is the length of the branch from the genesis block B_0 to B_m. For instance if $B_0 < B_1 < \cdots < B"_m$ is of length m, then $B"_m$ has height m.

A validity condition on blocks which is achieved by a time-consuming "mining" program There exists a function f_{valid} which is public and (for simplicity) fixed, that defines the eligibility of a block in the tree as follows. A block B'_i is a valid child of a block B_i if and only if $f_{valid}(B_i, B'_{i+1}) =$ accept (otherwise f_{valid} returns reject). We then say that a block B_j is *valid* if it is on a branch with origin the genesis block, and such that

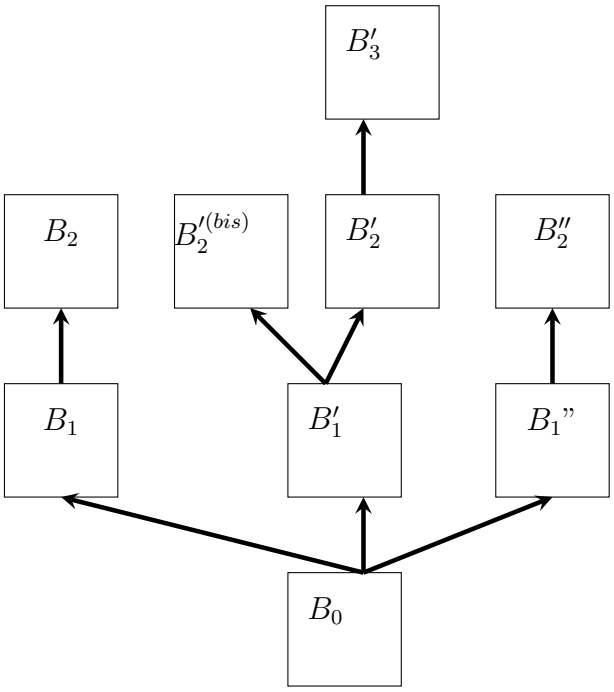

Figure 1: Blocks of values, partially ordered in a tree

every block in the branch until B_j is a valid child of his father. f_{valid} is easy to compute, is deterministic but its output is unpredictable in the following sense. Namely, the only way to create a valid child B_{i+1} of an existing block B_i, containing some fixed values $\mathcal{V}_{i+1}$, is to try many possibilities $\widetilde{B_{i+1}}$ of potential childs. Namely, repeat the following: build a potential child block $\widetilde{B'_{i+1}}$ and test if $f_{valid}(B_i, \widetilde{B'_{i+1}}) = \mathsf{accept}$. If not, then build another potential child $\widetilde{B"_{i+1}}$ and repeat. Notice that it is indeed possible to build many different potential childs of B_i, all of them containing the same ordered values $\mathcal{V}_{i+1}$, because one can play on the metadata $\widetilde{m_{i+1}}$. This time-consuming procedure is denoted as "mining".

The computational power of a node We denote $|N|$ the computational power currently devoted by a node N to computing this mining function. To make ideas concrete, one could make the normalization choice of defining $|N|$ as the number of calls to f_{valid} per 10 minutes that N does. But any other choice of normalization is also ok. The more N is currently allocating power in computing the mining function, the higher $|N|$. On the contrary, when N is currently sleeping, then $|N| = 0$. A process P which emulates 100 nodes N_i in parallel will thus have to divide his mining power between the nodes: $\sum |N_i| = |P|$.

Processes that follow the protocol are called *honest*. The other ones are called "byzantine/malicious". We will sometimes consider an entity $\mathcal{A}$, called "the Adversary" that coordinates the actions of malicious processes, and note $|\mathcal{A}|$ their total mining power.

Concrete parameters, and simplified assumptions The function f_{valid} is currently calibrated such that it takes on average 2^{72} attempts before finding a valid child. Given

all the computations currently done by the total nodes $\mathcal{C}$ on earth, a valid block is statistically found somewhere on earth every 10 minutes. The following statement is underlying [Nak09, §11], and is proved in §4.3. It states states "when", in a probabilistic sense, the mining program returns.

Theorem 1. *Let $t_1 < t_2 < \cdots < t_j$ be the sequence moments of time when, somewhere in the world, the mining program outputs a valid block. Consider a process N that is running one instance of the mining program with its full mining power $|N|$ allocated to it. Then the probability p_N that N* is *the* first *node N' in the world that outputs a new block after t_j, so at t_{j+1}, is equal to the* fraction of mining power *of N in the world. Namely, note $|\mathcal{C}|$ the total mining power allocated in the world after t_j, then:*

$$p_N = \frac{|N|}{|\mathcal{C}|}.$$

The proof is intuitive since it is as if all mining nodes on earth were concurrently tossing coins ($f_{valid}(B_i, \widetilde{B'_{i+1}})$) with probability 2^{-72} to land on the right side (accept). So the more a node tosses his coin quickly, the more likely he is to be the next winner. In practice the validity function f_{valid} is recalibrated every two weeks to maintain a constant average delay of 10 minutes between two blocks, provided the total mining power engaged $|\mathcal{C}|$ does not vary too much. *For the exam we can make the following simplistic assumption*, see §4.4 for a correct statement:

Assumption 3. *For every j, we have that the "delay between two blocks" $t_{j+1} - t_j$ is exactly equal to* 10 *minutes.*

Observation 2. *At this point we can make the following informal observation: consider a mining process, which has a tree consisting of one branch of blocks B_0, B_1, ... B_{z+1} , as the one in the left on Figure 3.1. Suppose that the process wants to change a value v in the block B_2, leaving the other values unchanged. Then it needs to create a valid child of B_1: a new block B'_2, containing the same values as in B_2, except v that is modified. This represents some work, at least* 10 *minutes. Then, it needs to create valid descendents of B'_2 which contain the same values as in B_3, B_4 etc. Creating each of these valid descendents B'_3, B'_4,...,B'_{z+1} represents some additional work, at least* 10 *minutes each. Notice also that this can only be done* sequentially, *since in order to mine every B'_{i+1}, the process needs to know a valid ancestor B'_i so that it can give them as input to the f_{valid}.*

Remark 3.1. The mining function is "memoryless" by several aspects. First, we see from Theorem 1 that a process N that is mining since a very long time, and didn't succeed to mine any block so far at t_j, will *not* be priviledged in any manner after t_j. It will *not* find a valid block quicker in the future. In fact, a new process N' that started allocating the same mining power much later than N, say at t_j, will have the *same* probability than N of being the lucky process at t_{j+1}. [3]

[3] Actually N', could have start mining just before t_{j+1}, its odds of being the winning process at t_{j+1} would even be the same. Of course this is not a winning strategy, because in reality, the time t_{j+1} is random: see §4.4. So that every further second of lazyness of N', increases the risk of N' not having started to mine when the (unpredictable random) time t_{j+1} happens.

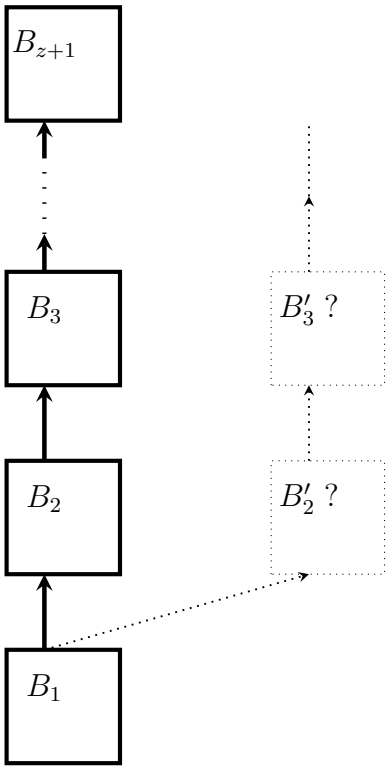

Figure 2:

Likewise, N' could well change several time its desired ancestor B_i of which it is trying to find a valid child, or values $\mathcal{V}_{i+1}$ that it is trying to include into the valid child, several times between t_j and t_{j+1}, he will still have an *equal* probability of succeeding at t_{j+1} as N. In case of success, the output of the mining function to N' is a valid block matching the ancestor + values he was currently mining on at t_{j+1}.

So the expressions: "start or finish to mine a block" or the "work necessary to mine a block" are faux amis, as well as the word "proof of work" itself actually. A good analogy is that processes are gambling many times in a casino —at the same game table or not—, each time with a very small chance to win. It is not because a process gambled a lot of times and never won, that he will have more chances to be the next winner in the casino.

3.2 Bitcoin protocol

We describe in Figure 3 the Bitcoin protocol, with the memory-inefficient simplification that every process maintains forever a local copy of the tree of all the blocks that it could download from the network so far.

The goal of the protocol, informally, is to guarantee that (1) every value v requested as append by clients, will appear exactly once in the *longest branch of the local tree* of every honest client and node, and that (2) when removing the b last blocks of the longest branch of any tree of an honest node—a parameter to be adjusted— then the remaining prefix will always appear identically in the longest branch of all the trees recovered in the future by any client of node. Thus, this longest branch prefix could be seen as a correct

Bitcoin protocol

Security parameter We fix b a positive integer.

Join/read To join the protocol and/or to read the Ledger, a process *queries* all the nodes to forward it the longest branch in their respective trees. Then it merges these branches into a tree that it stores locally. The process then read*s* the state of the Ledger as: the ordered sequence of values contained in the longest branch of its local tree, *minus* the b last blocks.

Append 1) A client requests the append*ing* of a new value v, by multicasting it to the nodes. Each node:

2) Collects pending new values $v_{pending,j}$, i.e. those that are not yet on the longest branch of its local tree.

3) Gathers them into a prospective new block B_{i+1} extending the last block B_i of its longest branch.

4) Launches the "mining" program, hoping to find a *valid* new successor B_{i+1}

5) When a node succeeds in mining a new block, it broadcasts it to all nodes

6) If receiving a valid new block B'_{i+1}, then a node adds it to its local tree, possibly querying the predecessors of B'_{i+1} if it hasn't them yet.

7) A value is append*ed* when all honest nodes have it in their longest local branch minus the b last blocks.

Figure 3: Bitcoin protocol, instantiated with the longest branch rule and an appending delay of b blocks

read operation.

About the rule to mine to extend the longest branch Notice that this rule is not present in the [Nak09] original paper. See the exercice in §5.2 for an explanation of why it is important for safety. Notice also that this rule is *not* the best possible one when the delay between consecutive blocks comes closer to network delays, see the end of §4.4.

A consequence of this rule is that, if an honest node N which is currently mining to extend B_i, receives a block B'_{i+1} from another node, then:

- If B'_{i+1} is a successor of B_i, and thus becomes the leaf of the longest branch of the N's tree. Thus N will from now on mine to extend B'_{i+1};
- Else if B'_{i+1} is another leaf in N's local tree, such that B_i is still the leaf of the strictly longest branch. Then N continues to mine to extend B_i;
- Or we have an undertermined situation when there are several longest branches of equal length. In this case, we will always consider the pessimistic scenario where honest nodes mine on the branch chosen by the adversary.

Introducing the tradeoff between efficiency and safety Notice also that the protocol of [Nak09] does not specify that a new process joining the system or reading the Ledger should request trees from *all* nodes in the protocol. Likewise it does not define when a value should be considered as append*ed*. Our specification of b blocks delay —a parameter to be adjusted— will be motivated by the next theorem.[4]

Notice that, even with the synchrony assumption, an adversary node could possibly send a very long branch to a single honest node N_1. As long as N_1 does not read the Ledger, he will then possibly see a different longest branch in its local tree, than the one of other honest nodes. The choice of b is designed in particular to avoid this kind of situation.

3.3 Safety properties and choice of the security parameter/delay b

Consider the situation of Figure 3.1, where the set of honest nodes $\mathcal{H}$ all have the branch on the left: $B_0, B_1, \ldots, B_{z+1}$. Let us call $\mathcal{A}$ the set of adversary nodes, so that the total set of computers in the world $\mathcal{C} = \mathcal{H} \cup \mathcal{A}$. Recall that the honest nodes are assumed to always mine on the longest branch. From the point where $\mathcal{A}$ enters the protocol and allocate its full mining power, we have thus that the percentage of mining power of honest nodes is

$$p_{\mathcal{H}} = \frac{|\mathcal{H}|}{|\mathcal{H}| + |\mathcal{A}|}$$

and the one of the adversary nodes is $1 - p_{\mathcal{H}}$. Then the following theorem bounds the probability to observe the scenario sketched in Observation 2. The first claim is proven at the end of [Nak09, p6], and in §4.5. The second one follows from the argument in [Nak09, p7], the interested reader can first read §4.4 to understand it.

[4] In practice, to join and read the Ledger one queries sufficiently many nodes until the probability that there is an honest up-to-date node among them is very high. Likewise, one could consider a value v to be append*ed* when the probability that every process read*ing* the Ledger, by querying some number of other nodes, sees v, is very high.

Theorem 3. *In the previous situation, we have that:*

Easy case, seen in class *Suppose that $\mathcal{A}$ joins the protocol —i.e. starts allocating mining power— when the honest nodes have* already *the branch on the left until B_{z+1}. Then the probability $\epsilon(z,p)$ that $\mathcal{A}$* ever *manages to build a concurrent branch B'_i starting from B_1 with the* same length *as the honest branch, is:*

- 1 *of* $1-p_{\mathcal{H}} \geq p_{\mathcal{H}}$, *and*
- $\left(\frac{1-p_{\mathcal{H}}}{p_{\mathcal{H}}}\right)^z$ *otherwise.*

Hard case, *not* seen in the lectures *Consider here that $\mathcal{A}$ had already been allocating mining power,* since *B_1 was created, so* before *the honest chain was extended to B_{z+1}. Consider situation where honest nodes would observe the same local tree as before in Figure 3.1. From the honest nodes' point of view, there only one branch $(B_i)_i$ in the tree. It is the one on the left, it reached the block denoted B_{z+1}. But the adversary may possibly have secretly mined a concurrent branch.* Assume *that the mining power of $\mathcal{A}$ is in* minority*:*

$$1 - p_{\mathcal{H}} < p_{\mathcal{H}}$$

Then for every $\eta > 0$, there exists a $z := z(\eta, p_{\mathcal{H}})$ such that the probability that $\mathcal{A}$ ever *manages to build a concurrent branch B'_i, starting from B_1 and reaching the* same length *as the honest branch (which is concurrently being extended by honest players), is smaller than η.*

However when $\mathcal{A}$ has the majority of mining power, then it follows from the easy case that he can catch up the honest branch with probability one.

Notice that we cannot say anything about scenarios where the adversary would have started allocating mining power *before* the honest nodes joined the protocol. In particular he could have secretly mined its adversary branch $(B'_i)_i$ in advance, send B_1 to the honest nodes so that they can start mining on it. Then, once honest nodes have read the Ledger from their honest branch $(B_i)_i$, $\mathcal{A}$ sends to them its longer adversary branch $(B'_i)_i$. Thus honest nodes will read a new state of the Ledger which does not extend what they read previously, which is a safety violation.

Corollary 4. *Assume that the fraction $p_{\mathcal{H}}$ of mining power of honest nodes in the world is fixed and strictly greater than* 51% *since the beginning of the protocol. Then for every $\eta > 0$ "the target probability of failure", there exists a security parameter $b(\eta, p_{\mathcal{H}})$ such that, the Bitcoin protocol in Figure 3 with parameter b greater or equal than $b(\eta, p_{\mathcal{H}})$ realizes a linear implementation of a Ledger, except with probability of violating safety $\eta > 0$.* [5]

For instance, the computations in [Nak09, p6] shows that if $p_H = 70\%$, then we can achieve a probability of failure η smaller than 10^{-6} by choosing a security parameter of $b := b(10^{-6}, 0.7) \sim 50$.

[5] By this we mean that a read operation can be in a conflict with a future read operation with probability η. In particular, an append operation which was assumed to terminate, could actually have not terminated up to probability η.

Another example is that the common usage is to wait for a delay of $b = 6$ blocks. The computations in [Nak09, p6] show that this guarantees a probability of failure smaller than 15%, in case honest nodes control more than 70% of the mining power.

Finally, when the network delay is not considered anymore as very small compared to 10 minutes, then the optimal adversary mining rate tolerated by Bitcoin decreases[2]. A tight bound is given in [GKR20].

4 Auxiliary material (not required for the exam)

4.1 A hash function

will be defined, for simplicity, as a "random oracle". Note $\{0,1\}^*$ the set of all binary strings and fix an output length of 256 bits[6]. Then Bellare and Rogaway CCS'93 define a random oracle as a map from $\{0,1\}^*$ to $\{0,1\}^{256}$ chosen by selecting each bit of $H(s)$ uniformly and independently for every $s \in \{0,1\}^*$. For convenience of the reader we will give a more concrete formulation, following the equivalent definition of [KL14], last paragraph of page 434. Following [GKL15, p8], we also model that querying this function on a new string s costs time, but no time if the string s was already queried [7].

Definition 4. *A hash function is a function:*

$$H : \{0,1\}^* \longrightarrow \{0,1\}^{256} \tag{3}$$

that takes as input a string s of arbitrary length, and outputs a string $H(s)$ of 256 bits.

The function H is such that: let $\mathcal{C}$ be the set of all computers in the world since H was invented, and $\mathcal{X}$ the table of values $(s, H(s))$ computed by $\mathcal{C}$ so far. Then for any string $s \in \{0,1\}^*$, we have that:

Determinism *either $H(s)$ was already computed by $\mathcal{C}$ before, then H returns the* same *value $H(s)$.*

Unpredictability *or $H(s)$ was not computed by $\mathcal{C}$ before. Then H returns a random value $H(s)$ sampled uniformly in $\{0,1\}^{256}$.*

Work *each call to H takes time τ for one computer[8], unless the value was already computed: $s \in \mathcal{X}$, in which case we assume it is returned in no time.*

Example 4.1. Let $\mathcal{N}_{40}$ the set of strings s in $\{0,1\}^{1000}$ such that $H(s)$ begins with 40 zeros. Let us compute the average time for one computer to find a string s in $\mathcal{N}_{40}$. Let us assume that initially $\mathcal{X} = \varnothing$ for simplicity. Thus, by definition, all the 2^{1000} values

$$H(0), H(1), \ldots, H(2^{1000} - 1)$$

[6] Which is the one of the Bitcoin's protocol, which uses the function SHA256

[7] The interested reader will notice that what we actually need is just a hash function with "preimage resistance", in the sense of [KL14, p. 4.6.2]. By contrast, a random oracle is a strong abstraction which is unimplementable, see e.g. Maurer-Renner-Holenstein TCC'04.

[8] τ is very small compared to 10 minutes. For typical Bitcoin computers, which are "antminers s9", then τ equals 10^{-14} seconds.

are all initially random variables

$$X_0, X_1, \ldots, X_{2^{1000}-1}$$

which are independent and vary uniformly in $[0, \ldots, 2^{256}-1]$.

Each time the computer calls H on a value s not computed before, the function H returns $H(s)$ a random sample of X_s. The random variable X_s is then equal forever to this fixed value $H(s)$. Let us note $\mathcal{H}_{40}$ the set of values in $[0, \ldots, 2^{256}-1]$ beginning with 40 zeros. The variables X_s being independent, the probability that $H(s)$ is in $\mathcal{H}_{40}$ is thus

$$p_s := P\big(X_s \in \mathcal{H}_{40} | \text{previous samples of } X_{s' \neq s}\big) = P\big(X_s \in \mathcal{H}_{40}\big) = \frac{|\mathcal{H}_{40}|}{|[0, \ldots, 2^{256}-1]|}$$

But we have that:

$$|\mathcal{H}_{40}| = \frac{2^{256}}{2^{40}} \ ,$$

left as an exercice. Thus $p_s = 2^{-40}$.

The situation is thus that the computer performs successive samples of independent binary variables —also known as coin tosses—, which output success with probability $p_s = 2^{-40}$ and failure otherwise. The average number of trials before success is thus $1/p_s = 2^{40}$. Multiplying by τ, we get an average time of $2^{40}\tau$ for finding a s in $\mathcal{N}_{40}$.

4.2 A chained data structure for blocks

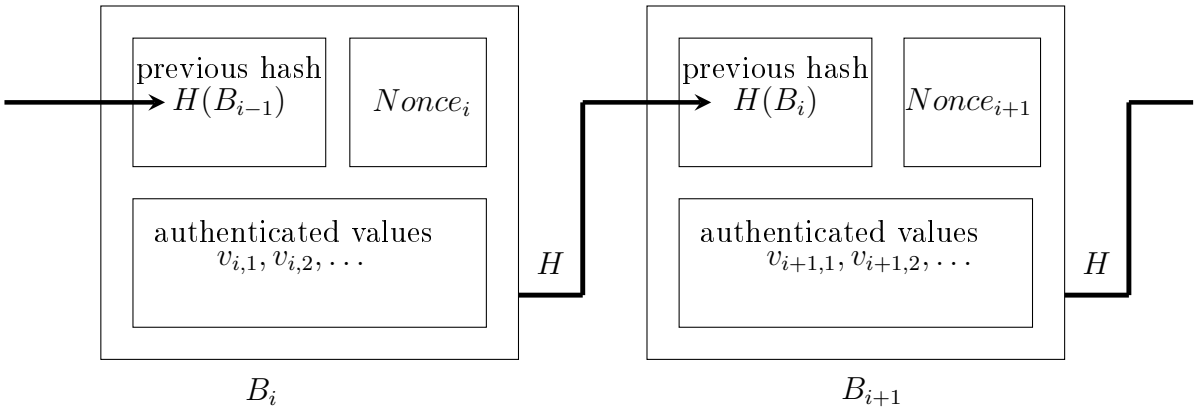

Figure 4: Chaining relation between two consecutive blocks in a tree

As explained in §3, see Figure 3.1 authenticated values are ordered within "blocks" B_i, B'_j..., which are themselves organized in trees. A valid tree must be such that:

- it starts with a specific root block B_0, the same for all trees, which is fixed at the beginning of the protocol;
- authenticated values within blocks of the same branch are all different;

- Let H be a fixed public hash function on 256 bits. Then the successor B_{i+1} of a block B_i is structured as the concatenation

$$B_{i+1} = H(B_i)||Nonce_{i+1}||(v_{i,j})_j \tag{4}$$

where $Nonce_i \in \{0,1\}^*$ is a string of bits such that

$$H(B_{i+1}) \in \{0,1\}^{256} \text{ begins with 72 zeros} \tag{5}$$

The last condition is difficult to satisfy, as we will quantify[9] [10].

4.3 Proof of Theorem 1

Exercise 4.2. (Cultural) (a) From the value $\tau = 10^{-14}$, and the fact that one mining computer in the world solves on average (5) every 10 minutes, estimate the order of magnitude of the total current mining power in the world.

(b) Deduce the order of magnitude of all hashes $H(s)$ computed in the world since 2008.

Exercise 4.3. (a) Consider the total merged trees $\mathcal{B}_{\in\prime\in\prime}$ of valid blocks ever created since 2008. Estimate an order of magnitude of the probability that two *distinct* blocks in $\mathcal{B}$ have the same hash (search the "birthday paradox").

(b) Estimate the order of magnitude of the probability that in 2100, two valid blocks in distinct places of $\mathcal{B}_{\in\infty\prime\prime}$ the have the same hash. We can e.g. proceed as follows. Suppose it is the case up to time t. Then every potential valid new leaf of $\mathcal{B}_\sqcup$: $Nonce_{i+1}$ is equal to a string of the form $B_{i+1} = H(B_i)||Nonce_{i+1}||(v_{i,j})_j$ defined in (4). By the recurrence assumption, it is thus distinct from all the other existing blocks B in $\mathcal{B}_\sqcup$. Conditioned to this state, estimate the probability that the hash of a fixed valid new leaf B_{i+1} equals the one of existing blocks in B in $\mathcal{B}$. Estimate the probability that this holds until 2100 by the approximation done in the birthday paradox.

Exercise 4.3 motivates the following assumption:

Assumption 5. *Consider the total merged trees $\mathcal{B}$ of valid blocks ever created. Then no two valid blocks in distinct places of $\mathcal{B}$ the have the same hash.*

Let us also make the following assumption, which seems not far from reality:

Assumption 6. *Computers in the world have so far exclusively dedicated all their mining power to compute hashes of strings of the form* (4)*, where B_i are valid blocks already created.*

[9] In practice the 72 zeros threshold is adjusted every two weeks: at the creation of Bitcoin it was only 32 zeros. The "mining difficulty" (search on Google) is the ratio between these two numbers.

[10] Actually the Nonce is only 32 bits long, so all possibilities of strings of format (4) are quickly exhausted if one leaves unchanged all other data of the prospective block. A big mining farm would exhaust all of these 2^{32} hashes in less than a millisecond. And the probability to find a succesful Nonce matching (5) in this set of strings is only $2^{32}/2^{72} = 10^{11}$. So in practice miners play on other variables in the block, as the time stamp or values, to test new strings. This is why we simplified and allowed that $Nonce_{i+1}$ can be of arbitrary length.

To summarize, we can now assume that all mining computers are successively calling $H(s)$ on distinct strings $s \in \{0,1\}^*$ on which H has not been called already, until they find one which satisfies (5): we define this as the "mining" program. We define this as the For each such string, the value $X_s := H(s)$ is independent from the previous calls of H, and varies uniformly in $\{0,1\}^{256}$, until it is actually computed.

In addition, even if miners have similar deterministic procedures to test strings satisfying (5), one can still make the assumption that no two different honest miners ever call H on the same string. Indeed in practice, the potential block B_{i+1} of each honest miner includes a specific value that depends on him.

The situation boils down to the following: every mining computer in the world tosses successive independent coins Y_s, each equal to

- $Y_s = success$ iff X_s matches (5)
- $Y_s = failure$ otherwise

Where the probability of success of each Y_s is equal to $p_s = p = 2^{-72}$, by the rule (5) and a straightforward adaptation of Exercise 4.1.

Let us model the time as a succession of tiny elementary intervals of duration $\tau/|\mathcal{C}|$, where in each of them, one of the mining computers $\mathcal{C}$ in the world tosses a new coin Y_s. Let us model that for each of these tiny intervals, a given node N with mining power $|N|$ has probability:

$$p_N = \frac{|N|}{|\mathcal{C}|} \ .$$

to be the one that tossed the coin. Consider, as in the theorem, the event of the first success after t_j. This happens in a certain fixed such tiny interval. Conditioned on this event, the probability that N is the computer that tossed the coin during this interval is thus p_N.

4.4 Random delays between blocks and discussion on synchrony

Let us consider the time delay between two blocks mined in the world:

$$T_j = t_{j+1} - t_j$$

The average number n of total coin tosses of Y_s in the world until a success is $n = 2^{72}$. Considering that $|\mathcal{C}|$ computers in the world are running in parallel, and that each computer takes τ time to toss a coin, then the expectation of every T_j is:

$$E(T_j) = 2^{72}\frac{\tau}{|\mathcal{C}|}.$$

Every toss being independent from the previous, the variables T_j are also independent. Let us assume that $|\mathcal{C}|$ does not vary, we thus have that the variables T_j are also equidistributed. This approximation is justified —to a certain extent— by the fact the number $n = 2^{72}$ is recalibrated every two weeks, such that we have:

$$E(T_j) = 10\ minutes$$

which we will assume from now on.

Consider now a percentage $0 \leq \lambda \leq 1$ and a time window W_λ:

$$W_\lambda = [t_j, \ldots, t_j + \lambda E(T_j)]$$

The number of tosses during this time window W_λ is λn, and satisfies $(\lambda n)p \leq 1$. We are thus in the good regime of large numbers, to model by a Poisson law the number of successes of tosses of (Y_s) occuring in W_λ. Namely we have :

$$\text{Number of successes in the world during } W_\lambda \sim \mathcal{P}(\lambda) \ .$$

Concretely, we have that the probability that k blocks are mined in W_λ is:

$$\frac{\lambda^k e^{-\lambda}}{k!} \ .$$

Likewise, during the same time window W_λ, a node with fraction p_N of the total mining power will perform $(p_N\lambda)n$ tosses during the time window W_λ, so that:

$$\text{Number of successes by N during } W_\lambda \sim \mathcal{P}(p_N\lambda) \ . \tag{6}$$

Exercise 4.4. Estimate the probability that more than 6 blocks are computed in less than 10 seconds by a node controling 50% of the total mining power.

Let us emphasize that short delays between blocks can lead to safety violations in the Bitcoin protocol, see §5.3 for a caricatural case. Let us mention for the culture that, in the other proof of work blockchain Ethereum, the delay between blocks is 15 seconds on average. To improve safety, the rule to mine on the longest chain is chain replaced by a rule consisting in mining on the densiest subtree.

4.5 Proof of Theorem 3

4.5.1 Easy case, seen in class: catching up from a fixed number of blocks behind

Let us consider the situation in Figure 3.1. We assume that initially, only honest nodes $\mathcal{H}$ are running the Bitcoin protocol, and that all of them have the same tree, made of the branch on the left (in bold): B_0, B_1... to B_{z+1}. Let us call it the *honest branch*. Then an Adversary node $\mathcal{A}$ joins the protocol, with a percentage of the total mining power that we note $p_{\mathcal{A}}$.

As the protocol goes on, the honest nodes continue to follow the protocol and mine for blocks extending the longest branch, which is currently the honest branch. The goal of $\mathcal{A}$ is to change the values that $\mathcal{H}$ read in the Ledger, for example change a value contained in the second block B_2. For this, $\mathcal{A}$ needs first to mine an alternative block B_2' extending B_1. Then he needs to extend it into an *adversary branch* $(B_i')_i$, until it reaches the length of the honest branch $(B_i)_i$. When this happens, $\mathcal{A}$ will need only sending this adversary branch (B_i') to the honest nodes, so that they will include this adversary branch in their tree. They will have to decide which of this two longest branches they should try to

extend. In the worst case scenario (see §3.2), honest nodes will from now on all mine on extending the adversary branch.

Starting from any situation between the two concurrent chains $(B_i)_i$ and $(B'_j)_j$, then by Theorem 1, the probability that the next block is mined by an honest node is

$$p = p_{\mathcal{H}} = \frac{|\mathcal{H}|}{|\mathcal{H}| + |\mathcal{A}|}$$

and by the adversary is $1 - p_{\mathcal{H}}$.

The goal of $\mathcal{A}$ is to catch up its late of z blocks behind the honest branch. The following exercise shows that the probability $\epsilon(z, p)$ that this ever happens is: one is $1 - p_{\mathcal{H}} \geq p_{\mathcal{H}}$, and

$$\Big(\frac{1 - p_{\mathcal{H}}}{p_{\mathcal{H}}}\Big)^z$$

otherwise.

Exercise 4.5. (Gambler's ruin, reverted) Consider two players $\mathcal{H}$ and $\mathcal{A}$ playing several coin flips with a biased coin: $\mathcal{H}$ has probability p to win at each round, whereas $\mathcal{A}$ has probability $1 - p$ to win. Suppose that $\mathcal{H}$ starts with an advantage of z points. Then the probability that $\mathcal{A}$ ever catches its initial late of z points behind $\mathcal{H}$, is:

- 1 if $1 - p > p$.
- $\left(\frac{1-p}{p}\right)^z$ if $1 - p < p$.

Hint: consider P_z the probability to catch up a late of z points. Consider that $P_0 = 1$, and that from the next coin toss we have $P_z = pP_{z+1} + (1 - p)P_{z-1}$.

5 Attacks and exercises

5.1 Sybil attacks are useless

clients's influence on the protocol is limited by the money they can spend. The goal of this paragraph is to explain why denial of service attacks from clients is limited. This will not be considered for the exam. In practice, a client must have enough money on its account, typically 70 Dollars, to send one valid request. We do not discuss how this is possible without revealing the identity of the physical person controling the client, see Appendix A for more information. This charge of 70 Dollars is blocked on the client's account until the request is executed. Then they are transfered to the node that mined the block containing the request. client can possibly pay more to be prioritized. In conclusion, even if a physical person emulates many clients, then its ability to send many valid Append requests to the nodes is limited by the money it can spend.

nodes' influence on the protocol is limited by their computational power

Exercise 5.1. Consider the situation of §4.5. The Adversary node $\mathcal{A}$ runs a set of computers to compute the mining program. It starts with a gap of 6 blocks late behind the branch mined by honest nodes. Explain whether or not its strategy would be more efficient, if instead it emulated 100 nodes, using the same set of computers $\mathcal{A}$.

5.2 Why mining on the longest branch

Exercise 5.2. In the protocol, remove the condition "if the new Block B'_{i+1} becomes the leaf of its local longest branch". Thus now, we assume that a node automatically starts mining on the top of the last new blocks he received —be it in its longest branch or not. Explain briefly a strategy, for an adversary $\mathcal{A}$ having only a minority of the mining power, say 40%, to ensure that it will ultimately fully control the content of the longest branch. Hint: divide the adversary in three nodes performing different tasks.

5.3 Why synchrony is important

Safety fails without synchrony Suppose that the message propagation time is comparable to the time between two blocks. For example in Ethereum, a new block is created every 10-15 seconds. Then honest nodes might not all agree on the same longest chain on which to build. Thus their mining power will be dispersed over several branches, while a powerful adversary will concentrate on extending one single branch, and ultimately impose it. This is why Ethereum's "Ghost" protocol replaces the mine-over-the-longest-chain rule, by another rule: mine over the densiest subtree.

The Gramoli-Natoli's "balance attack" considers an adversary $\mathcal{A}$ controling the network, that isolates a client C during a certain period of time. During this period, $\mathcal{A}$ extends the tree saw by C by a chain of blocks that $\mathcal{A}$ forges. These blocks typically include values stating that $\mathcal{A}$ sends money to C. Once C is convinced that this adversary chain represents the state of the Ledger, he takes actions in real life: like sending goods to $\mathcal{A}$ in exchange for the money. Once connectivity is reestablished between C and the rest of the world, C catches up with the longest chain, mined by honest nodes. He then discovers, too late, that $\mathcal{A}$ didn't send money to him in this chain.

We formalize this in the following exercise:

Exercise 5.3. Consider that the world is composed of three nodes N_1, N_2, N_3, with respective mining powers in proportion of 60%, 30% and 10%. Consider an honest client V (the "victim") and consider that N_2 is a dishonest node who:

- Runs a client C_2 that can make as many valid requests he wants.
- And has the power to isolate V and N_3 from the network during, say, one day. That is, during one day, N_2 can block all the incoming messages to V and N_3, except the ones he decides.

Suppose that the initial state is such that everyone starts with the same initial Block B_0. Describe a strategy for N_2 which will guarantee him, with high probability, that:

- V will first Accept a branch B_i containing a value v_2 of client C_2 in the first block B_1.
- Then V will later change its choice, and Accept a concurrent branch B'_i *not* containing the value v_2 in the first block B'_1.

Selfish mining: synchrony minus epsilon => 33% adversary imposes 50% of blocks.

Exercise 5.4. We exemplify [GKL15, remark 3]. We consider an Adversary node $\mathcal{A}$ that has $\frac{|\mathcal{A}|}{|\mathcal{C}|} = 1/3$ of the total mining power in the world. The reste of the nodes: $\mathcal{H}$ play the Bitcoin protocol honestly. $\mathcal{A}$ has furthermore the *rushing* power : everytime it sees a honest node sending a message m, it has the power to send a message m' very quickly such that m' is delivered to all the nodes before m. It plays the following strategy: initially, the adversary mines on the same chain as every honest node. However, whenever it finds a new block, it keeps it private and keeps on extending a private chain from this new block. Whenever an honest party finds a new valid block and sends it to the network, the (rushing) adversary immediatly broadcasts one block from the private chain such that this block is received first. If the private chain is depleted the adversary returns to the public chain. Explain why the Adversary will produce on average 50% of the new blocks that will end up on the Ledger (=the longest branch minus b blocks) (and not only 33% of them).

5.4 Bitcoin protocol *does not* solve consensus

let us recall that an Adversary/malicious/byzantine process, client or node, is by definition one that deviates from the protocol. For example:

- Sends different values or blocks to different nodes
- Deliberately ignores a pending value v in the blocks he mines, although v is pending for a long time
- Does not mine on the longest chain
- Does not broadcast a block as soon as he mined it

Exercise 5.5. Try to understand [GKL15, §5.1], by explaining why there is a nonnegligible set of runs where the validity condition, as defined in this paper, is violated.

A A client is a digital signature

To avoid impersonation, a client signs every value that he request to append to the ledger, with his *digital signature*. In fact, he is exactly defined by his digital signature algorithm $(\text{Sign}_C, \text{Verif}_C)$, see the definition below. Verif_C is public information, which can be considered as the identity card of C. Whereas Sign_C is a secret C keeps for himself. In practice Sign_C and Verif_C are called the "public key" and the "private key" of C. Any person can generate as many different signature algorithms $(\text{Sign}_C, \text{Verif}_C)$ it wants, and thus run many different clients sequentially or in parallel. *The practical limitation being that, when a client wants to append a value to the Ledger —like making a transaction to another client— then it must pay 70 Dollars fees, so must have this sum on its account* Verif_C.

The digital signature algorithm of a client C consists in two algorithms $(\text{Sign}_C, \text{Verif}_C)$. Only C should know the first one, which, on input any string v : "the document to sign", outputs a valid signature of C on the document v. The requirement is that knowing the signature of C on a certain v, gives no additional information on what a valid signature of C on a different document $v' \neq v$ should look like

But everybody has access to the the second function, which enables to verify if a signature on a document is valid or not. To fix ideas we define below a signature algorithm as having a signature length of 160 bits. This is the one recommended for ECDSA, which is used in Bitcoin and, e.g., Whatsapp: for information see [Gal12, p. 22.2.2] then [JMV01]. [11]

Definition 7. *The digital signature of a client C is a pair of algorithms. First, a* signature algorithm Sign_C*, which takes as input any string $v \in \{0,1\}^*$ of any length, and outputs a string of* 160 *bits* $\text{Sign}_C(v)$*: the* signature of C on the document v*;*

then, a verification algorithm

$$\text{Verif}_C : \{0,1\}^* \times \{0,1\}^{160} \longrightarrow \{true, false\} \tag{7}$$

$$v,\ s \longrightarrow (\text{Sign}_C(v) == s)\ . \tag{8}$$

They must guarantee that, for any person that does not know Sign_C*, then for any document v' different from all those already signed by C, then the task of finding a valid signature s' on v' —i.e. such that* $\text{Verif}_C(s', v')$ *returns true—, is infeasible even with all the computing power on earth*[12]*.*

References

[Gal12] Steven D. Galbraith. *Mathematics of Public Key Cryptography.* Cambridge University Press, 2012. URL: `https://www.math.auckland.ac.nz/~sgal018/crypto-book/ch22.pdf`.

[GKL15] J. A. Garay, A. Kiayias, and N. Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications". In: *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques - Advances in Cryptology (EUROCRYPT).* 2015.

[GKR20] Peter Gaži, Aggelos Kiayias, and Alexander Russell. "Tight Consistency Bounds for Bitcoin". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security.* 2020.

[HW90] Maurice Herlihy and Jeannette M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems* 12.3 (June 1990), pp. 463–492.

[11] To generate a digital signature for Bitcoin, which is synonymous of a client, one just needs to produce a 256 bit string at random (64 hexa characters): the public key. Then call `openssl ec` on this public key with the curve `-name secp256k1` to generate the corresponding private key.

[12] See e.g. Pollard's rho attack mentionned [JMV01, p 29], which uses no memory and takes 2^{80} steps: this is equal to the number of particles in the universe.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *Int. J. Inf. Secur.* 1.1 (Aug. 2001), pp. 36–63.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. 2nd. Chapman & Hall/CRC, 2014.

[Nak09] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: *Cryptography Mailing list at https://metzdowd.com* (Mar. 2009).