Almost-decentralized fast payments in Bitcoin

Ludovic Barman

School of Computer and Communication Sciences

Semester Project

December 2014

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Sebastian Faust
EPFL / LASEC

Abstract

We propose a solution for fast payment processing in Bitcoin. It relies on selecting periodically a group of *Fast Payments Operators* responsible for agreeing on fast transactions, and enforcing their decision on the Bitcoin network.

We show that under reasonable assumptions, we achieve with high probability to select the operators such that the majority of them are honest, and have agreed on a Public Key Infrastructure. Given the PKI, we can then use efficient cryptographic protocols for Byzantine Agreement to achieve consensus, used to decide upon fast payments. These operators have a reasonable work load, and allow transaction to be processed much faster than with the current Bitcoin protocol. In this paper, we describe one main theoretical approach and one alternative for this problem.

Contents

Chapter 1

Introduction

1.1 Objectives

At the current state, the Bitcoin network is relatively slow to process payments : 60 minutes are recommended to get sufficient confidence on a processed transaction (**?**) (see Chapter **??**).

The main objective is to allow fast payment in Bitcoin, with respect to the inherent constraints of the Bitcoin system. We want to keep our design as decentralized as possible; our solution needs to work with minimal changes to the current Bitcoin protocol, to be viable both theoretically and economically.

The reader may ask... Why not just lower the difficulty of the proof-of-work and hence reduce the 10 minutes per generated block rate in Bitcoin? It appears that as explained in **?**, we need the ratio

$$\frac{\text{network propagation time}}{\text{block generation time}}$$

to be "negligible [...] so that transaction persistence can be shown [...]". This paper further argues why lowering the block generation time, as done in others altcoins[1] **?**, is a straightforward idea but with unclear negative consequences.

10-minutes block generation time. Hence, we need to keep the block generation time as it is. Note that even if we could reduce the *transaction confirmation time* to the *generation time of one block*, it would still be non-satisfactory. We aim to have a system as fast as Paypal or other instantaneous payment schemes; hence, 10 minutes is still too long for our needs. As a result, we need some sort of sub-protocol that is able to decide and broadcast information faster than the Block chain. In our solution, this is made possible by the *Fast Payments Operators* making periodic public claims, much faster than the block generation time.

Sub protocol. This sub-protocol will be used for the network to *agree* on the fast transactions. Once the members of this sub-protocol agreed that a transaction is accepted or not, it broadcasts the decision to the normal Bitcoin network, enforcing the decision. To sum up, we want this protocol to perform large-scale *Byzantine Agreement*.

Differences with classical Byzantine Agreement. In our setting, we require the BA protocol to scale well[2], to decide in time $\Delta_{BA} \ll 10\text{min}$, and in general we want our protocol to allow

[1] An altcoin is an Alternative-Coin, in practice a derivate of Bitcoin.

[2] The 90-day running average of the number of active nodes in the Bitcoin network is of 6200 nodes in late December 2014.

nodes to join and leave (but not necessarily a specific instance of the Byzantine Agreement[3]). In addition, we do not have a bound on the number of identities controlled by the adversary, but a bound on his processing power (less than 50% of the total power, by requirement of the Bitcoin protocol (**?**)). This is really similar to what Bitcoin is doing, with the extra requirement to decide faster, and with the all the Bitcoin ecosystem usable as a basis for our protocol; contrarily to what is done is (**?**), we do not aim to replace Bitcoin, but to build on top of it.

1.2 Structure of this paper

First of all, we give some background information about the Bitcoin system (see Chapter **??**).

Then, we start the presentation of our protocol with two transformations and their proof (see Section **??** and **??**); they will help us to select a set of miners with desirable properties. We concatenate these two constructions into one (see Section **??**) for simplicity; we will use it as black box in the second part of this paper. We also present an alternative solution (see Section **??**) based on Byzantine Agreement on the whole Bitcoin network, which can also be used as a basis for the second part of the paper.

Following this, we present the fast payment protocol (see Section **??**), and propose a list of concrete changes in Bitcoin's client source code to implement this protocol (see Chapter **??**).

We conclude with an analysis of the proposed solution (see Chapter **??**); we also discuss the problem of using Bitcoin as a source of randomness (see Annex **??**).

[3]We could have a two-phases protocol: in the first phase, nodes are allowed to join and leave, and not in the second, where we run Byzantine Agreement.

Chapter 2

Bitcoin Basics

e-money. The concept of e-money is not new at all: the first milestone was laid in 1990 by David Chaum, with his paper on anonymous electronic cash **?**; nowadays, credit card are a perfect example of widespread electronic cash. However, Bitcoin, created in 2009 by Satoshi Nakamoto **?**, is different from previous solutions in several ways: it is completely decentralized; meaning that it does not have a central trusted bank, but rather relies on a fully peer-to-peer network. Some of the advantages of Bitcoin are the fact that it is not bound to a country or a jurisdiction, but is intrinsically worldwide; that it has lower fees than centralized electronic cash systems; and probably more interestingly, that any computing device can *create* Bitcoins by running a special program.

Block chain. Bitcoin's main data structure is the *block chain*, a distributed append-only linked-list that can be seen as a global ledger, upon which every computer of the network agrees. The block chain is made of *blocks*, themselves containing several *transactions*. Blocks are linked together by the hash of the previous block, thus forming the chain. Blocks can be seen as pages in the global ledger, and transactions as individual records on the page. A transaction has one or more inputs, one or more outputs, and a script initially used for safety check and transaction processing, but now widely used to extend Bitcoin's functionalities[1].

The chain is growing with new transactions: Miners are creating new blocks, which contains the transactions that users from the network want to perform, validating them by adding a proof of work, and broadcasting them to the network; if the block is accepted by the other miners, the miner that validated it also receives a reward in Bitcoins; hence the denomination "mining Bitcoins", process in which you are in fact validating others' transactions, and getting rewarded.

Forks happen when two miners try to add two (or more) different blocks to the chain at the same position. As this is not possible by design, only one will remain, the other one being destroyed, its transactions to be later re-included in other blocks. Decision on which block is legitimate and which is not is made by comparing the proof-of-work, the validation added by the miner. These proofs-of-work are computationally moderately hard problems, even by clusters of computer working together[2]. In Bitcoin, there is a mechanism to ensure that solving these problems takes 10 minutes on average; hence, the time needed to create a block with new transactions is also close to 10 minutes, and this constitutes a strict lower bound for the total confirmation time of a transaction. It is interesting to notice that this very process is a

[1] MPC, for example see (**???**); another example : decentralized time-stamping on Bitcoin (**?**).
[2] Called "pools" in Bitcoin.

weaker[3] form of Byzantine Agreement in a fully distributed manner; this is the first algorithm of this kind, see (**?**) for a presentation.

1 hour confirmation time. We will now explain why the time to confirm a transaction is usually one hour, as advised on the Bitcoin FAQ (**?**). When Alice wants to pay Bob for, say, an online newspaper, she will create a transaction with the correct details and broadcast it to the network. Miners that receive this new transaction have an incentive to include it on the block they are trying to validate, because of the fee they collect upon validation. The expected time to validate a block is 10 minutes, after which we expect Alice's transaction to be in the block[4]. But as we saw earlier, the network might very well be in a state where the chain has a fork. Bob would be careless to consider Alice's transaction to be definitive, as it can be on a fork that will be deleted later on. As a result, Bob will wait to have several blocks on top of the one with Alice's transaction: the probability for Alice's transaction to be on a secondary branch is decreasing exponentially with the number of blocks above it. In a typical scenario, Bob waits for at least five blocks on top of the block with Alice's transaction; with one new block each 10 minutes, this takes 60 minutes on average.

[3]Where the *validity* property is not fulfilled with high probability. See (**?**).

[4]it might also be the case that Alice transaction is still pending.

Chapter 3

Protocol

3.1 Model

Groups of miners. We have the following groups :

- The Bitcoin network is made of T honest miners. The number T is not known to the honest parties. It may be known to the adversary.
- The *Fast Payments Operators* network, or simply the *Operators* network is made of N miners. N is a public parameter known by everyone.

Sets and size. A set is denoted by "blackboard bold" notation, e.g. $\mathbb{A}$. The size of this set is usually denoted by the corresponding capital letter, i.e. A.

Each party has the same computational power. For simplicity, and because it is used in the proof of Construction A, we model the network as T honest parties all having the same hashing[1] power π. It easy to see that we can represent parties with different hashing power by having the more powerful of them run several instances of the protocol. We will try to find more efficient protocols, but for now the simplicity of this idea helps. This is similar to what is done in (**??**).

Adversarial model. In this scenario, we consider a polynomial time attacker that controls an arbitrary number of entities, such that the sum of the computing power (the *hash rate* if we talk about proof-of-works) of these entities is π_A. We assume an active adversary, able to deviate arbitrarily from the protocol. He may read all the communications channels, and is able to modify messages (but not to forge signatures). He cannot delete a message from two directly connected honest parties.

Unless specified otherwise, we do not include the adversary when we mention *parties* or *miners*; we model the adversary as an external entity that has access to the communication channels, following what is done in (**?**). This captures what the above attacker can do, and at the same time makes the model simpler.

Communications. We assume a network that provides guaranteed delivery of honest parties' messages; in particular, honest parties can broadcast messages with the guarantee that all

[1] We talk about *hashing* since the computational power is measured with Proof-of-Works, which in Bitcoin is based on a Hash function.

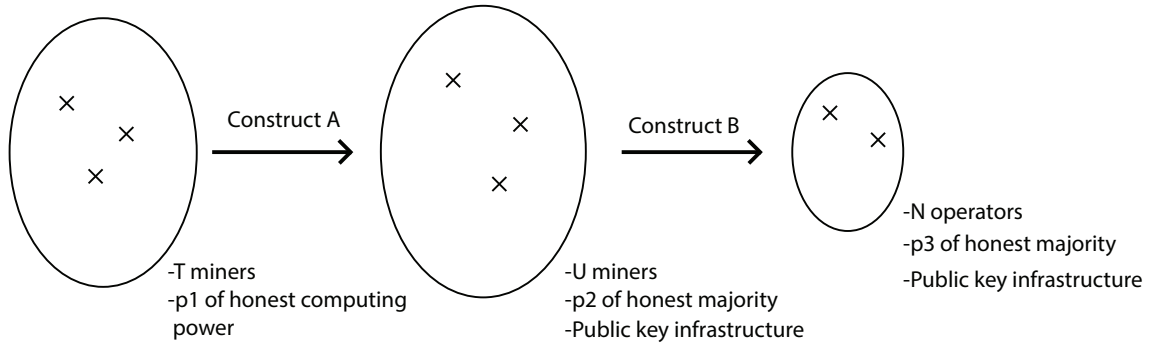

Figure 3.1: Illustration of the two constructions

honest parties will receive this message in bounded time $\Delta_{\text{propagation}}$. In particular, honest parties' broadcast (signed) messages cannot be deleted, modified or forged by the adversary, nor can it be delayed more than $\Delta_{\text{propagation}}$.

3.2 Operators selection process

3.2.1 Introduction

This alternative (see **??** for the other option) is based on the work done by Marcin Andrychowicz and Stefan Dziembowski **?**.

3.2.2 Definitions.

The selection of the operator is made in two steps.

- First, the construction A (see **??**) is used to obtain, assuming that the Bitcoin network is made of T miners and that $p_1 = [\frac{1}{2}, 1[$ of the computing power in Bitcoin is honest, a set of U miners with public-key infrastructure, such that we have at least $p_2 = [\frac{1}{2}, 1[$ of honest majority.

- Then, the construction B (see **??**) is used to get, from a set with at least $p_2 = [\frac{1}{2}, 1[$ honest nodes, a small group of size N, e.g. $N = 100$, with at least $p_3 = [\frac{1}{2}, 1[$ honest nodes.

3.2.3 Construction A

Introduction. For this construction, we use the protocol described in (**?**) to create a static set of miners with at least p_2 of honest nodes and a public key infrastructure. For the convenience of the reader, we describe this protocol hereafter.

RankedKey protocol

Assumptions. The protocol works without any trusted setup. We assume the existence of a Proof-of-work scheme with low variance (unlike Bitcoin Proof-of-work), similar to what is presented in (**??**).

Definition 1. We use the model presented in Section **??**. Let $\Sigma = (\text{Gen, Sign, Verify})$ be a signature scheme, let $l \in \mathbb{N}$ be a public parameter, and let Π be a multiparty protocol (where the number of participating node is *not* known in advance). We denote $(P_1 ... P_n)$ the parties executing Π, each of them with a hash rate π. Each P_i takes as input a security parameter 1^K, and outputs a tuple $(\text{sk}_i, \text{pk}_i, K_i, \text{rank}_i)$, where $(\text{sk}_i, \text{pk}_i)$ is the key pair of P_i, $\mathbb{K}_i$ is a set of public keys, and rank_i is a function from $\mathbb{K}_i$ to $\{0, 1, ...l\}$.

The protocol Π is a π_A-secure l-ranked Σ-key generation protocol if for any polynomial time adversary A with hash rate π_A attacking the protocol, the following properties hold :

- **Key generation algorithm.** For every party P_i, the tuple $(\text{sk}_i, \text{pk}_i)$ is such that $\forall m \in \{0,1\}^*$, Verify(pk_i, Sign(sk_i, m)) holds; In addition, the adversary A has negligible probability to forge a valid message in the "chosen message attack" model.
- **Consistency.** The sets $\mathbb{K}_i$ created by the honest parties are identical. Denote this set $\mathbb{K}$.
- **Validity.** For all honest party P_i with a key pk_i, we have $\text{pk}_i \in \mathbb{U}$.
- **Bounded creation of identities** The number of created identities is at most $T + \lceil \pi_A / \pi \rceil$ except with negligible probability.

Protocol RankedKey. This is an summary of the protocol **RankedKey** presented in (**?**). The interested reader may read the details on **Fig 3** in (**?**). This protocol is divided in three phases :

- **Challenges phase**
  This phase consists of $l + 2$ rounds, each lasting $\Delta_{\text{propagation}}$, with l a public parameter.
  - Round 0 : Each party P_i draws a random challenge c_i, and broadcasts a message $\text{Challenge}_0(c_i)$ to other parties.
  - Round $k \in [1, l+1]$: Each party P_i wait, for at most $\Delta_{\text{propagation}}$, for the messages $\text{Challenge}_{k-1}(c)$. There is at least n such messages, where n is the number of honest parties, but there can be much more. To compute c_k, P_i uses a hash function F, and the set A_i^{k-1} of all the received challenges a in round $k-1$; $c_k = F(A_i^{k-1})$.
- **Proof-of-work phase**
  Each P_i generates a fresh key pair $(\text{sk}_i, \text{pk}_i)$, then solves $\text{Sol}_i = \text{POW}(F(\text{pk}, A_i^{l+1}))$, where F, A_i^{l+1} is the same as above, and POW is the low-variance proof-of-work described in the assumptions. Then, each P_i broadcast the message $\text{Key}_0(\text{pk}_i, A_i^{l+1}, \text{Sol}_i)$
- **Key ranking phase**
  This phase consists of $l + 1$ rounds. Each P_i starts with an empty set of ranked keys $\mathbb{K} = \emptyset$.
  - Round 0 : Each party waits $\Delta_{\text{propagation}}$ for messages $\text{Key}_0(\text{pk}, B^{l+1}, \text{Sol})$. For each received message, P_i checks that the proof-of-work is valid, and that his latest challenge c_i^l is $\in B^{l+1}$. If so, P_i broadcasts $\text{Key}_1(\text{pk}, A_i^l, B^{l+1}, \text{Sol})$, and adds pk to $\mathbb{K}$ with rank 0 (ie. sets $\text{rank}_i(\text{pk}) = 0$)
  - Round $k \in [1, l]$: Each P_i waits (for at most $\Delta_{\text{propagation}}$) for messages of the form $\text{Key}_k(\text{pk}, B^{l-k+1}, ..., B^{l+1}, \text{Sol})$. For each received message, he checks that the proof-of-work is valid, that pk $\notin \mathbb{K}$, and that c_i^{l-k} is $\in B^{l+1-k}$ and $\forall i \in [l+1-k, k]$, we have $F(B^i) \in B^{i+1}$. If so, P_i broadcasts $\text{Key}_{k+1}(\text{pk}, A_i^{l-k}, B^{l+1-k}, ..., B^{l+1}, \text{Sol})$ and adds pk to $\mathbb{K}$ with rank k (ie. sets $\text{rank}_i(\text{pk}) = k$).
  - End of protocol : Each party P_i outputs $(\text{sk}_i, \text{pk}_i, K_i, \text{rank}_i)$.

This protocol is a π_a-secure l-ranked key generation protocol in the sense of **Definition 1**.

RankedBroadcast protocol

Assumptions. The protocol works without any trusted setup. We assume the total hash rate of the system (all parties combined) is bounded by some $\pi_{\max}$, and that the number of messages the adversary can send is bounded. In addition to this, we need the same assumptions than for the RankedKey protocol, i.e. the existence of a Proof-of-work scheme with low variance; indeed, each P_i first runs RankedKey, and uses the output $(\mathrm{sk}_i, \mathrm{pk}_i, \mathbb{K}_i, \mathrm{rank}_i)$ as an input for RankedBroadcast.

Definition 2. We use the model presented in Section **??**. Let Π be a multiparty protocol (where the number of participating node is *not* known in advance). We denote $(P_1 ... P_n)$ the honest parties executing Π, each of them with a hash rate π. Each P_i takes as input a value $x_i \in \{0,1\}^*$, and outputs a set $Y_i \subset \{0,1\}^*$.

The protocol Π is a $\pi_{\max}$-secure broadcast protocol if for any polynomial time adversary A with hash rate $\pi_A < \pi_{\max}$ attacking the protocol, the following properties hold :

- **Consistency.** The sets Y_i created by the honest parties are identical. Let us call this set Y.
- **Validity.** For all honest party with an input x_i, we have $x_i \in Y$.
- **Bounded creation of identities.** The size of Y is at most $T + \lceil \pi_A / \pi \rceil$.

Protocol RankedBroadcast. Each party P_d runs RankedBroadcast in parallel. The *dealer* initiates the broadcast, with its value x_d. Each party maintains a set Z_i^d initialized with $\emptyset$. The protocol consists of $l+1$ rounds that each lasts $\Delta_{\text{propagation}}$.

- Round 0 : The *dealer* P_d broadcasts $(x_d, \mathrm{Sign}_{\mathrm{sk}_d}(x_d, \mathrm{pk}_d))$
- Round $k \in [1, l]$: Each $P_i, i \neq d$, waits for the messages of the form

$$(v, \mathrm{Sign}_{\mathrm{sk},a_1}(x_d, \mathrm{pk}_d)), ...\mathrm{Sign}_{\mathrm{sk},a_k}(x_d, \mathrm{pk}_d))$$

  The message is accepted if :
  - all signatures are valid,
  - $\mathrm{pk}_{a1} = \mathrm{pk}_d$
  - $\mathrm{pk}_{aj} \in \mathbb{K}_i$ and $\mathrm{rank}_i(\mathrm{pk}_{aj}) \leq k \ \forall j \in [1, l]$
  - $v \notin Z_i^d$ and $\|Z_i^d\| \leq 2$

  If so, P_i adds v to Z_i^d; if $k < l$ she broadcasts

$$(v, \mathrm{Sign}_{\mathrm{sk},a_1}(x_d, \mathrm{pk}_d), ...\mathrm{Sign}_{\mathrm{sk},a_k}(x_d, \mathrm{pk}_d), \mathrm{Sign}_{\mathrm{sk},i}(x_d, \mathrm{pk}_d))$$

- End of protocol : P_i outputs $v \in Z_i^d$ if $\|Z_i^d\| = 1, \perp$ otherwise.

This protocol is a $\pi_{\max}$-secure broadcast protocol in the sense of **Definition 2**.

HonestMaj protocol

Assumptions. The protocol works without any trusted setup. We assume the model presented in Section **??**. We assume the total hash rate of the system (all parties combined) is bounded by some $\pi_{\max}$, and that the number of messages the adversary can send is bounded. In addition to this, we need the same assumptions than for the RankedKey and RankedBroadcast protocols, which are used as a black-box here, i.e. the existence of a Proof-of-work scheme with low variance.

Protocol HonestMaj. The protocol consists of four steps:

1. Each party P_i runs RankedKeys, with output $(\text{sk}_i, \text{pk}_i, K_i, \text{rank}_i)$. Let K_i^0 denote the set of keys k with rank 0, that is $\text{rank}_i(k) = 0$.
2. Each P_i runs RankedBroadcast with input K_i^0. Denote the (shared) output $\{X_1, ...X_m\}$.
3. Define $K = \{k : k \text{ belongs to more than } m/2 \text{ of the sets } X_1, ...X_m\}$.
4. Output K

This protocol is a key generation protocol in the sense of **Definition 1**.

Overview

HonestMaj as black-box. In the rest of the paper, we will use HonestMaj as a black-box. Given the model (See **??**) and the assumptions stated above (bounded hash-rate polynomial time adversary, bounded total hash-rate of the system, bounded number of messages sent by the adversary, proof-of-work scheme with low-variance), we are able to generate in a decentralized manner a set of public keys $\mathbb{U}$ such that :

- Every honest party has his public key in the set.
- Every honest party shares the same set.
- Given an adversary A with hash-power π_A, the size of the set is at most $T + \lceil \pi_A / \pi \rceil$ except with negligible probability.

Lemma 1. Assuming that the Bitcoin network (*set of miners*) $\mathbb{T}$ has at least p_1 of honest computational power, we can create a set of miners $\mathbb{U}$ where there is a lower bound p_2 on the honest majority (*of parties*). This set has a PKI. With high probability, $p_1 \approx p_2$.

Proof of Lemma 1. We assume that the Bitcoin network is made of T honest miners, each miner having the same hash-rate π; more powerful miners are "split" into several parties with the same hash-rate π. We have p_1 of honest computational power; the total power of both honest participants and the adversary is $\pi_{\max}$. In addition, we have that the hash rate of the adversary is bounded by

$$\pi_A \leq (1 - p_1) * \pi_{\max}$$

In addition, we have

$$T * \pi \geq p_1 \pi_{max}$$

Each miner P_i runs HonestMaj. In the end, they all share a set $\mathbb{U} = K$ including all honest miners. Following the property of **bounded identity generation**, the size of $\mathbb{U}$ is at most $T + \lceil \pi_A / \pi \rceil$. We have

$$\begin{aligned}
& U \leq T + \lceil \pi_A / \pi \rceil \\
\iff \quad & U \leq T + \left\lceil (1 - p_1) \pi_{max} / \tfrac{(p_1)\pi_{max}}{T} \right\rceil \\
\iff \quad & U \leq T + \left\lceil T \left(\tfrac{1-p_1}{p_1} \right) \right\rceil \\
\iff \quad & U \leq T + T \left(\tfrac{1-p_1}{p_1} \right) + 1
\end{aligned}$$

From this, we calculate the percentage of honest parties p_2 in $\mathbb{U}$:

$$p_2 = \frac{T}{\text{size of } \mathbb{U}}$$

$$p_2 \geq T / \left(T + T \left(\frac{1-p_1}{p_1} \right) + 1 \right)$$

$$p_2 \geq 1 / \left(1 + \left(\frac{1-p_1}{p_1} \right) + 1/T \right)$$

Since T is large, we approximate $1/T \approx 0$ for convenience.

$$p_2 \geq 1 / \left(1 + \left(\frac{1-p_1}{p_1} \right) \right)$$

$$p_2 \geq \frac{p_1}{p_1 + 1 - p_1}$$

$$p_2 \geq p_1$$

Hence the lower bound

$$p_2 \approx p_1$$

3.2.4 Construction B

Introduction. In this part, we present a protocol to create a static set of miners $\mathbb{O}$, i.e. a set of Operators, of relatively small size N with p_3 honest majority (i.e. at least $p_3 N$ party are honest) and a public key infrastructure.

Assumptions. We start with a set $\mathbb{U}$ of parties with at least $p_2 * U$ honest parties.

Definition 3. Let $\mathbb{U}$ be a set of (distinct) public keys, let $r \in \{0,1\}^*$ be some randomness, let $N \in \mathbb{N}$ be a public parameter, and Π a protocol. We denote $(P_1 ... P_Q)$ the parties executing Π. Each P_i takes as input $(\mathbb{U}, r, N)$, and outputs a set $\mathbb{O}_i$ of size N.

The protocol Π is a *random subset selection* protocol if the following conditions holds, even with a polynomial time adversary A attacking the protocol.

1. **Consistency** The sets $\mathbb{O}_i$ produced by the honest parties are identical. Denote this set $\mathbb{O}$.
2. **Random subset** In the random oracle model, each public key pk_i has the same chance to be in $\mathbb{O}$.

Protocol DiscriminatePk. Each party P_i runs the protocol with input $(\mathbb{U}, r, N)$. Let H be a hash function, $H : \{0,1\}^* \to \{0,1\}^q$; parties have access to domain expansion and reduction functions by using standards methods and padding, such that $q = \log_2(N)$. Finally, let Sort:$\{0,1\}^q \to \{0,1\}^q$ be a deterministic permutation over the space of binary string of size q, e.g. a lexicographic sort. Each party P_i maintains one set $\mathbb{O}$ initially equal to $\emptyset$.

- Round 0 :
  Compute $\mathbb{U}' = \text{Sort}(\mathbb{U})$. Let $\mathbb{U}'^{(i)}$ be the i^{th} public key in $\mathbb{U}'$.
  Compute $n_0 = H(r), n_0 \in [0, N[$. Add $\mathbb{U}'^{(n_0)}$ to $\mathbb{O}$.

- Round $i \in [1, N-1]$:
  Compute $n_i = H(n_{i-1}), n_i \in [0, N[$.
  While $\mathbb{U}'^{(n_i)} \in \mathbb{O}$, compute $n_i = H(n_i)$.
  Add $\mathbb{U}'^{(n_0)}$ to $\mathbb{O}$.
- End of protocol : Outputs $\mathbb{O}$.

Theorem 2. The protocol DiscriminatePk is a random subset selection protocol.

Proof of Theorem 2. We prove the **Consistency** part. Trivially, since DiscriminatePk is a deterministic protocol that invokes no communications between the parties, and since all honest parties P_i starts with the same inputs $(\mathbb{U}, r, N)$, all honest parties outputs the same set $\mathbb{O}$.

We prove now the **Random subset** part. In the random oracle model, the output of $H(.)$ is uniformly distributed in the set $[0, N[$.

1. Round 0:
   The random variable n_0 is uniformly distributed in the set $[0, N[$, hence the first pick is uniformly distributed in $\mathbb{U}'$: each public key $\text{pk}_i \in \mathbb{U}'$ has an equal chance to be picked.
2. Round $i \in [1, N-1]$:
   Define $\mathbb{U}'' = \mathbb{U}' \setminus \{n_0, ...n_{i-1}\}$. n_i is uniformly distributed in $[0, N[$, but we recompute a new value for n_i if $n_i \in \{n_0, ...n_{i-1}\}$; as a result, we exclude $\{n_0, ...n_{i-1}\}$ from the range of possible values for n_i. We have n_i uniformly distributed in the range $[0, N[\setminus\{n_0, ...n_{i-1}\}$, hence the i^{th} pick is uniformly distributed in $\mathbb{U}''$, the set of remaining keys.

Theorem 3. Given a lower-bound $p_2 \in [\frac{1}{2}, 1]$ on the proportion of public keys belonging to a honest party P_i in $\mathbb{U}$, we can model the probability p_3 of having a honest majority in $\mathbb{O}$ with the Hypergeometric Distribution.

Proof of Theorem 3. Let the fact of picking a honest party in $\mathbb{U}$ be represented as a *success*, and the fact of picking a dishonest party be represented as a *failure*. The hypergeometric distribution describes the probability to have k successes in n draws when picking in a finite population of size N containing exactly K successes.

Our protocol does exactly the same thing, i.e. picking N public key from the set $\mathbb{U}$ in which at least $p_2 * U$ are honest (*success*). To match with the hypergeometric distribution definition, we have $X \sim \textsc{Hypergeometric}(n = N,\ N = U,\ K = p_2 * U)$, and we are interested in the probability $p_3 \geq \Pr(k \geq N/2)$. The first inequality appears because we have *at least* $p_2 * U$ honest parties.

Corollary 1. Given $p_2 = \frac{2}{3}$ of honest majority in $\mathbb{U}$, $N = 100$, $U = 6000$, we achieve a honest majority $\mathbb{O}$ with good probability.

Simulation of corollary 1. After running the simulation (see Figure **??**), the probability that the majority is not honest is roughly $2 * 10^{-4}$. We reach 10^{-10} with $N \simeq 330$, and 10^{-20} with $N \simeq 690$.

If we plot the probability to have an honest set versus the number of operators picked, we get the confirmation that for $N > 50$, the probability to have an honest majority becomes close to 1.

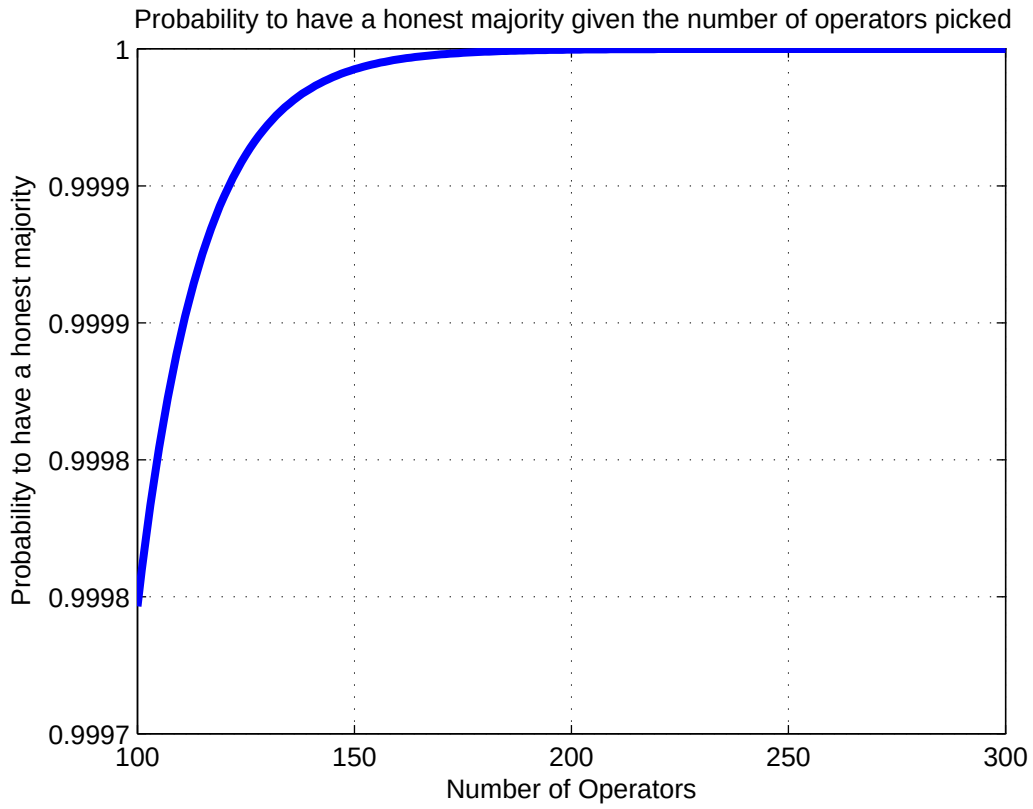

Figure 3.2: Evolution of the probability to end up with an honest majority

The reader may ask... Why not use the one of the Leader Election (cf. (**?**)) algorithm (which elects a winning node at random from a set of node) and repeat the operation N times ? The main reason in addition to inefficiency is that these algorithms usually require a hard upper bound on the number of dishonest parties in the set. We are not in this scenario : we have a (good) *probabilistic* bound on the ratio of honest/dishonest parties, but we have no strong guarantees. In addition, since we rely on the Bitcoin block chain, we have a trusted random beacon that we can use for more efficient protocols.

3.2.5 Construction C

Introduction. Construction A can be used as it is. However, Construction B takes as input some randomness $r \in \{0,1\}^*$. We emphasis that *all parties* should use the same r for the **Consistency** property of Definition 3 to hold. In the meantime, r should be random, i.e. if r is known in advance, some adversary can create a public key that will be selected by the picking process in DiscriminatePk, e.g. that will fall at the position n_0 in $\mathbb{U}'$, thus breaking the **Random subset** property. Hence, we need to release a random r to every party before the beginning of the protocol, in a decentralized fashion. This is the problem of the *random beacon*, discussed in (**?**).

The Bitcoin Blockchain as a Random Beacon. Let H be a hash function, let $\mathrm{Bc}(t) : \emptyset \to \{0,1\}^q$ be a function that yields the value of the latest block on the main chain; each $t \approx 10$min, this value is replaced with a new one. It is believed that $\mathrm{Bc}(t)$ is hard-to-predict, even with the knowledge of the previous $\mathrm{Bc}(0), ..., \mathrm{Bc}(t-1)$. See Appendix **??** for a discussion on the

subject. We will assume here that for $t' < t$, $\mathrm{Bc}(t)$ is unpredictable, and use $r(t) = H(\mathrm{Bc}(t))$ as a distributed source of randomness.

Merging the constructions A and B. We call the following protocol **Construction C**

1. Define $t_{\text{second part}}$ as some time in the future. Set $N = 100$.
2. Each party P_i run **HonestMaj**, and gets the same output $\mathbb{U}$.
3. Each party P_i wait for $t_{\text{second part}}$
4. Each party computes $r = H(\mathrm{Bc}(t_{\text{second part}}))$.
5. Each party computes $\mathbb{O} =$**DiscriminatePk**$(\mathbb{U}, r, N)$

3.3 Operations selection process - alternative

Introduction. Another promising line of work is large-scale dynamic Byzantine Agreement, already proposed and currently being implemented in LDA, EPFL by Rachid Guerraoui, Florian Huc, Anne-Marie Kermarrec **?**.

General idea. The main idea is to divide the network, of size T, in clusters of size $\log T$. Each cluster runs locally a Byzantine Agreement algorithm, and broadcasts the results to the other clusters. Given the fact that the adversary controls less than $\frac{1}{3}$ of the nodes, the construction leads to global agreement. The protocol allows a polynomial number of nodes joining or leaving the network. The required global properties (e.g. the fraction of nodes the adversary may control) are maintained for each cluster, by a clever way of shuffling between clusters when another node joins.

Comparison to our needs. The first very good news is that it fits quite well the requirements of the fast payment operation : every cluster basically has the same properties as the output of $\text{Construction}_{\mathbf{C}}$. We do not really need the ability for nodes to join or leave at all times, since we want a static set of Operators, but we could accommodate that fact : instead of running the protocol presented in part 1 at some time t, we could run continuously this protocol, and at some time t, randomly select a cluster, and publicly select all the nodes currently in the cluster to be part of the Fast Payments Operators.

However, there is two main problems with this solutions : first, there is the need to do a *Setup* phase, where one (potentially non-trusted) node (conceptually, the initiator of the Byzantine Agreement) start the protocol, and allows other nodes to join. In the Bitcoin system, it is unclear who this node must be.

In addition, the protocol does not prevent Sybil attacks; i.e. an adversary that can create several identities (and exceed $\frac{1}{3}$ of the total number of identities) may break the protocol. This is clearly bad in our scenario, and we need to adapt it and add the property of *bounded identity generation* (see **??**). It turns out that this addition is rather orthogonal to the development done in (**?**), and can be achieved rather easily.

Fixing the Sybil attack. Concretely, every time the random beacon outputs a new value (i.e. the block chains has a new block), every party in each cluster start to work on a proof of work including the new value of the random beacon. They broadcast to their own cluster the solutions found; In every cluster, honest nodes discard the parties from their cluster if they do not produce satisfying solutions every round. Note that this correspond to the *exact normal mining process*, if we add the requirement for the miners to broadcast their partial solutions[2] to their cluster. With minimal changes, and minimal extra work, we are able to force every node in a cluster to compute proof-of-works to participate in the Byzantine decision.

Comparison of the efficiency. This second protocol maintains clusters with the desired properties with a time and communication cost polylogarithmic in the number of nodes joining / leaving the network, itself polynomially bounded. In comparison, the protocol HonestMaj has a complexity polynomial in $\Delta_{\text{propagation}}$ plus linear in the time needed to compute the low-variance proof-of-work. The complexity of this second protocol is lower, but with the need to keep the protocol running, or to regrow the clusters each new instance of the protocol (which requires a node that initiates the protocol). We do not include the time needed for the proof-of-work in this second protocol, since it would just be publishing the computation already done by the miners. As a word of conclusion, this alternative is promising in terms of efficiency, but is not a straightforward solution.

3.4 Fast payment operation

Abstract We pick a small set of miners with a honest majority, able to run standard protocols for Byzantine Agreement, that we call Fast Payments Operators. Then, we use this set to do a binary decision (ACCEPTED or REJECTED) for each fast transaction. This result is then made public (broadcasted by the operators), and becomes a reference for the miners and the payer/payee. By design, we enforce this decision to be written in one of the future blocks. Therefore, the payee is certain to receive the money as soon as the operators finish their Byzantine Agreement, which is much faster than waiting for a certain number of blocks.

3.4.1 Setup protocol

The protocol **??** works as follow :

Let the set $\mathbb{T} = \{m_1, m_2 ... m_T\}$ of size T be the current set of Bitcoin miner.
We construct $\mathbb{O} \leftarrow \text{Construction}_{\mathbf{C}}(\textit{public parameters})$. Each miner $m_i \in T$ constructs is own $\mathbb{O}$, and by the Consistency property of Definition 3 (see Section **??**), every honest party share the same set $\mathbb{O}$.
This set has the following properties :

1. This set has a public key infrastructure set up; all honest parties share the same set of public keys.
2. This set is made of at least $p_3 * N$ honest parties; p_3 is the minimum ratio of honest parties in the set. Following the numerical example of $\text{Construction}_{\mathbf{B}}$, we have $p_3 = 1/2$ with good probability, hence we have honest majority in $\mathbb{O}$.

[2] And this is currently used in pooled mining, for pool member to prove to the pool operator that they are working on the problem.

Algorithm 1 Protocol run collectively by the network

1: **function** OPERATORSETUP(miners M)
2: $O \leftarrow$ **Construction**$_C(M)$
3: **return** O
4: **end function**

3.4.2 ProcessFastPayment protocol

The protocol described in **??** can be explained with different tasks :

Listening to Fast Payments. These operators must then watch the network for special transaction with the *Fast* flag[3].

Making a decision. When such a transaction is detected[4], the operator(s) which saw the transaction run the protocol :

1. Broadcast the fast transaction to every other operator.
2. Start a Byzantine Agreement with the transaction as input, and $d \in \{\text{accept}, \text{reject}\}$ as output. Notice that since we reached a small group of Operators, fully aware of each others, assumptions such as almost-synchronicity and fault detection make sense and we are able to run efficient Byzantine Agreement. For instance, we suggest running (**?**), although there might exists newer, more efficient protocols.

More precisely, each operator first decide locally on a d_{local} which reflects if the transaction is acceptable given the current block chain (this is precisely the exact process run by all miners when receiving a new transaction)[5]. Then, each of these N operators agree on a common $d = \textbf{ByzantineAgreement}(d_{\text{o1}}, d_{\text{o2}}, ...d_{\text{oN}})$.

Performance could be increased by deciding on a set of different transactions instead of one individual transaction (see (**?**)).

Making the result public. Each operator has two public lists ACCEPTEDTRANSACTIONS and REFUSEDTRANSACTIONS. Upon the decision d, it will add the tuple $(\text{transaction}, \textbf{Signature}(\text{transaction}))$ to one of these two lists, so everyone may access the current decisions;

In addition, we could make the Bitcoin node more passive by broadcasting the operators decision to them, instead of having them regularly poll the Operators. Hence, the operators could broadcast their decision (signed by a majority of Operators) to the network. Node only accept such broadcast if there is above $N/2$ signatures.

Bounded size of lists. The two lists ACCEPTEDTRANSACTIONS and REFUSEDTRANSACTIONS do not grow indefinitely : once a transaction t that was in ACCEPTEDTRANSACTIONS (W.L.O.G) appears on the block chain, and a sufficient number of block is stacked on top of it, operators can remove t from ACCEPTEDTRANSACTIONS.

[3]This might also be simply the presence of a fee to the Fast Payments Operators.
[4]or when a threshold is reached, or when a timer is triggered; both reasons are strictly for efficiency purposes.
[5]Some implementation simplify this process for efficiency purposes.

Algorithm 2 Tasks run continuously by operators

```
1: function OperatorRun( )
2:     t1 ← new thread(OperatorListen)
3:     t2 ← new thread(OperatorDecide)
4:     t1.run()
5:     t2.run()
6: end function

7: function OperatorListen(other operators O, threshold t)
8:     PrivatePendingTransactions ← [ ]
9:     loop: Message ← NetworkListenAndWait()
10:    switch(Message)
11:      case NewFastTransaction :
12:        PrivatePendingTransactions.add(Message)
13:        broadcast Message to O
14:        if PrivatePendingTransactions.length > t
15:          trigger event DoProcess  // in OperatorDecide
16:    goto loop
17: end function

18: function OperatorDecide(other operators O)
19:    PublicAcceptedTransactions ← [ ]
20:    PublicRefusedTransactions ← [ ]
21:    on event DoProcess :  // either on timer or PrivatePendingTransactions above a threshold
22:      foreach t ∈ PrivatePendingTransactions
23:        d_t ← ByzantineAgreement(O)
24:        if d_t == true
25:            PublicAcceptedTransactions.add( t, Sign(t))
26:        else
27:            PublicRefusedTransactions.add( t, Sign(t))
28: end function
```

3.4.3 Explanations

Using the decision to reach global fast consensus. Now, the key idea is that the operators have a higher decision power than the rest of the network; when they decide on an outcome for a pending transaction (either accepted or refused), no other miner can decide against them. All forks that do not comply with the operator's decisions are automatically considered invalid by the honest miners. Then, they are able to check for two (undesirable) situations :

1. They were working on creating a new block, but this block contains a transaction that is invalid regarding one of the operator's decision. In that case, they can immediately stop working on the block (it is never going to be accepted by other honest miners).

2. They accepted a fork that contains a transaction that is invalid regarding one of the operator's decision. In that case, they must re-obtain the latest main fork of the chain,

which will be broadcasted with the normal Bitcoin protocol; they however know that they have some invalid blocks at the end of the chain, and are able to discard them and start working on new block if they wish.

Work done by the operators. We emphasize that the operators do *not* have to mine and find a solution to any proof-of-work. They make *statements of truth* to the best of their abilities, which will become *realizations of truth* since by design, the network *cannot* disagree. These *statements of truth*, which represent pending-but-already-accepted-or-refused transactions, will be integrated in some block later on, by some miner, through the normal process.

More formally, suppose there exists a set of transaction $(t, t'_1, t'_2, ...)$ mutually exclusive. If t is **Accepted** by the operators, all other transactions $(t'_1, t'_2, ...)$ are implicitly **Refused**. Hence, any transaction t'_i that might reach the network will be **Refused** by all miner once they know that t was **Accepted**. Hence, no honest miner can accept t'_i and try to embed it in a block. As a result, t, which is still not in a block (only **Accepted** by the operators), will have no reason to be refused by any miner, and hence will be at some point embedded in the block chain.

Making the decision. We allow some of the operators to be malicious, but not more than the majority. Hence, we require the transaction to appear in more than half of the operator's AcceptedTransactions list to say it is accepted.

Speed improvement. Since there is no proof-of-work, the time required for processing a transaction is roughly equal to $2\Delta_{\text{propagation}} + \Delta_{\text{ByzantineAgreement}}$, believed to be much faster than the $k * 10$ minutes used for creating k blocks. In addition to the time required for the operators to run the Byzantine Agreement, we simply need the time for a transaction to reach the operators, and for the decision d to reach to the buyer.

Chapter 4

Modifications to current Bitcoin protocol and code

4.1 Operators selection process

We propose to embed into the Bitcoin clients code one of the selection protocol described in the first part of the paper; we describe the scenario with $Construction_C$, but the alternative construct could be used as well. Running this protocol is optional, the miners that want to become a Fast Payments Operators run the code in a subthread.

4.2 Miner's modifications

We take *bitcoinj*[1] release 0.11, a Java Bitcoin client, to have a concrete basis for our explanations.

Chains comparison modification. In the current Bitcoin protocol, miners have a way to compare two forks of the same chain, and decide which is the main fork they will continue to work with. This decision process is at the heart of our new fast payment protocol; in the current protocol, miners only take the amount of work done one the work as a criterion for deciding. We add *precedence to the Fast Payments Operators claims*.

Concretely, in *bitcoinj* we would change the line 498 in **AbstractBlockChain.java**[2]

```
boolean haveNewBestChain = newBlock.moreWorkThan(head);
```

With :

```
boolean haveNewBestChain = newBlock.moreWorkThan(head) && complyWithOperators(newBlock);
```

Conceptually, we do not accept the fork ended with *newBlock* if the Fast Payments Operators made a decision incompatible with a transaction in *newBlock*. By design, the decisions made by the Fast Payments Operators are enforced in the block chain, and this without extra proofs-of-works.

[1] https://github.com/bitcoinj/bitcoinj

[2] /core/src/main/java/org/bitcoinj/core/AbstractBlockChain.java

The function complyWithOperators is very close to the normal anti-double-spending check done in **Wallet.java**[3], line 1686, method **checkForDoubleSpendAgainstPending**().

```
Array[Transactions] operatorsDecisions; //storage of the received operator's decisions

function complyWithOperators(Block b)
disagree = false;
forall Transaction t in operatorsDecisions
    forall Transaction t' in b
        Array[TransactionOutPoint] quotedTx = t.getInputs().map(i => i.getOutpoint());
        Array[TransactionOutPoint] quotedTx' = t'.getInputs().map(i => i.getOutpoint());
        //we could be more precise in this check, but here we take no risks
        if quotedTx'.contains(quotedTx) && t ≠ t'
            disagree = true;
        end
    end
end
return disagree;
```

Here, if the block contains a transaction using (even partially) the same input transactions than a decision made by the operators, we delay it up to the next block to be completely sure of avoid problems. A finer decision rule could take into account the amount of bitcoins in the inputs, and the amount spent in the two clashing transactions.

4.3 Fast payment operation

Modifications to Transactions. We add a control bit **isFastTx** into the Transaction model. By default, **isFastTx** is **false**. Operators only process transactions when **isFastTx** is **true**. Normal miners ignore this flag.

Concrete fast payment process. Alice creates a new transaction t with **isFastTx** = **true**. She broadcasts it to the Bitcoin network. At this point, two simultaneous behaviors happens :

1. The normal miners, ignoring the flag **isFastTx**, start to process the transaction. The expected time for confirmation is 60 minutes.

2. The Fast Payments Operators, seeing the flag **isFastTx**, will run Byzantine Agreement between themselves to see if the transaction t is compatible with their respective current view of the block chain. The result is a common output bit $b \in \{\text{accepted, refused}\}$. They add it to their public lists, and also broadcast their decision to the Bitcoin network. At this point, Alice knows the definitive result b, and so does Bob, quoted in the outputs of t. The expected time for confirmation is therefore :

$$\Delta_{\text{network propagation time}} + \Delta_{\text{Byzantine Agreement}} + \Delta_{\text{network propagation time}}$$

   if we assume no workload on the operators, i.e. the Byzantine Agreement for t is run immediately.

[3] /core/src/main/java/org/bitcoinj/core/Wallet.java

Hence, we see that the new expected time for confirmation is lower bounded by $2\Delta_{\text{network propagation time}} + \Delta_{\text{Byzantine Agreement}} (\ll 60 \text{ minutes})$, and upper bounded by 60 minutes.

Chapter 5

Conclusion

We propose a solution that theoretically enables Fast Payments in Bitcoin, which is a major missing feature. Once the set of Fast Payments Operators is created, the fast payment operations are conceptually simple, and can be implemented without major changes in a future version of any Bitcoin's clients. In contrary to the trivial solutions found in some alt-coins, we emphasize that this protocol is analyzable from the point of view of security.

Rather than being a final immutable solution, this paper aims to present a novel way of processing fast payment : through several constructions, we reach a situation where we apply classical Byzantine Agreement. As for the constructions themselves, two versions were already presented in this paper, and more efficient alternatives may exists; although it was part of this research, optimization is far from finished.

In particular, we would like to further work on :

- model the parties *not* as having the same hashing power π, but rather allow each party p to have a different computational power of π_p. This would make the construction way more practical for big pool (clusters) of computers; in the present situation, they are forced to simulate multiple time the protocol, i.e. having several key-pairs, voting several times in the Byzantine Agreement, etc.

- further specify the *vector Byzantine Agreement*, e.g. how the N operators agree not only on one transaction, but on several in one instance of Byzantine Agreement; can we make the length of this vector variable, and adjust it to the current workload ? In addition, since we have a honest majority, under normal circumstances we are in a situation where at least $N/2$ nodes start with the same decision values; could we improve Byzantine Agreement in that specific case ?

- more concretely analyze the *economical incentives* that would drive nodes to become a Fast Payments Operators; i.e. dedicating a reward per fast transaction for the operators; while the fast payment decision process in itself is not computationally too intensive, the setup of the set of Operators is, and each unit of computing power dedicated to this process is not dedicated to mining, resulting in a potential loss of income for the node participating to the fast payment protocol. It would be logical to find a balance by rewarding them afterwards. In addition, it would discourage normal users to use the fast payment system when it is not needed, thus reducing the work load on the operators.

Appendices

Appendix A

Using Bitcoin as a random beacon

The main problem of using each bitcoin block as a source of randomness is that the property of *unpredictability* is not strictly respected. Rather than presenting a formal proof, we will argue here about the reason we think Bitcoin can be used as a random beacon.

Formalization. Each T minutes, T being a exponential random variable with a expectancy of 10, a new block B_t is created. We define the random beacon $rnd_{BC,t}$ as $H(B_t)$, with H being a hash function.

Analysis of one attack. In the Random Oracle setting (where H would be a random oracle), even 1 unknown bit in B_t is sufficient to make $rnd_{BC,t}$ unpredictable. An attacker trying to *guess without mining* would have to guess the correct order of transactions in the Merkle Tree, the correct miner (or mining pool) that will produce a solution[1], as well as guessing / computing the proof-of-work which is also part of the block.

Analysis of a smarter possibility. It is clear that the previously mentioned attack seems hard, especially in the time-bounded setting of 10 minutes (new value from the oracle). It should be also clear that the strategy followed by the attacker is far from optimal; he guesses the inputs, and still have to compute the result of the proof-of-work.

A more likely attack would come *from a miner*, preferably with high computational power : if the miner can find a solution to the proof-of-work at time t_1 less than 10 minutes, then by publishing the block, he will set the next value B_t. Since he has access to the random oracle (or the hash function), he is able to compute $rnd_{BC,t}$ at time t_1, before time $t_2 = t_1 + \Delta_{\text{propagation}}$, where a node far away becomes aware of B_t. There is also another more critical attack : by withholding the solution, and publishing it only at a time t_3, he gains $t_3 - t_1 + \Delta_{\text{propagation}}$ extra time with $rnd_{BC,t}$ while the other miners are not aware of the new value of the random beacon. This is a risky strategy, as some other miner might find *another* solution between t_1 and t_3, thus potentially invalidating the attack and resulting in a loss of income for the attacker.

Mitigation. The first thing to realize is that in a distributed setting where we run a Randomized Byzantine Agreement protocol[2], we *cannot* avoid the possibility where some honest node have access to the random become up to $\Delta_{\text{propagation}}$ after the others; this comes from the topology of the network; when one or several node make a decision, and they are aware to form a quorum (i.e., the decision will be definitely enforced), the time to propagate the

[1] since the public key of the miner is part of the block.

[2] It is another denomination for the Random Beacon.

information to an honest node not part of the quorum is only upper-bounded by $\Delta_{\text{propagation}}$.

To palliate the second attack, we think that the best mitigation is economic : indeed, taking the risk to by bypassed by another miner when you already found a solution results in a loss of income of 25 BTC[3]. If we can make sure that the result of the cost of withholding the block to alter the random beacon, time the probability to find a new block that is more desirable than the last one, leads to earning less than 25 BTC, this attack will not happens. This should be ensured by every system using the Block Chain as a random beacon. This is also the approach taken in **?**.

[3] 11 December 2014 : roughly 9000 USD.