UNIVERSITÄT
BERN

Analysing FastPay

Functionality and Formalization of FastPay

Bachelor Thesis

Noah Maggio

from

Bern, Switzerland

Faculty of Science, University of Bern

13. December 2023

Prof. Christian Cachin
David Lehnherr
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

FastPay is used to settle pre-funded payments with high-integrity and availability by using a committee, where some members can be Byzantine. As its core primitive, it uses Byzantine consistent broadcast instead of relying on total order broadcast, i.e., consensus, or reliable broadcast. Recent work has shown that it is not necessary to use consensus in order to provide high security. Due to this fact, FastPay achieves low latency by using a less strictly broadcast. This work looks at the functionality and implementation of FastPay, explaining the process up to the finalization of a transaction, discussing some security aspects, and explaining how the protocol handles crashes.

Contents

Chapter 1

Introduction

Cryptocurrencies, especially those based on blockchains, like bitcoin or the somewhat newer currency ethereum, have gained popularity the past couple years. Because of this, the number of users increased, leading to higher usage and increasingly more transactions, need for higher scalability (c.f. Croman et al. [10]) and with all of that came the need for higher standards. These mentioned currencies are based on blockchains, which allow account holders to interact directly with an online and highly secure distributed ledger. This means centralized banks (i.e., trusted third parties) are unnecessary and would only slow down transactions due to the way those transactions are handled via netting systems. In a netting system, the settlement of payments is deferred for some period of time, usually until the end of the business day (c.f. Bech and Hobijn [6]). Blockchains, on the other hand, need high amount of time to check and verify transactions because the system is public, which means there is always the possibility of being a victim of malicious attackers who sabotage some messages in their favour. At the heart of blockchain lies the problem of preventing double spending, which is usually solved using consensus. In an earlier time, when those cryptocurrencies were not that famous, those long waiting times were somewhat accepted as a small price for high security and being decentralized. But as the saying goes, "time is money". With the ever-changing world and the necessity for inventions, especially in technology, everything has to become better and faster. One wants to have the least amount of delay, everything should work in real time. The same applies to the transactions with cryptocurrencies and other digital currencies, but without sacrificing the security aspect. The question now may arise, is there such a technology to decrease the time needed for such transactions? The answer is yes, and it is called FastPay. It covers every aspect of the so-called trilemma of blockchain, is decentralized, has high security and allows for high scalability, where the classical blockchain approach suffers from this, as observed by Bano et al. [2] and Danezis et al. [11].

FastPay is a Byzantine fault tolerance real-time gross settlement system (RTGS) that uses Byzantine consistent broadcast and has a committee-based approach, containing authorities. Typically, in a classical blockchain approach, broadcast-type consensus or equivalently, total order broadcast, is being used (to achieve order of transfer among the participants), which is the solution to prevent a famous attack called double spending. Guerraoui et al. [12] have shown that it is in fact not necessary to use full consensus in order to prevent double spending, which means a weaker and faster broadcast type can be used. Baudet et al. used this fact and came up with the protocol FastPay, which represents the basis of this thesis. It should be mentioned that FastPay is not the one and only protocol that improves payment in cryptocurrencies. There exist many different protocols that all try to achieve similar goals, for example, Astro relies on Byzantine reliable broadcast (c.f. Collins et al [9]) or Brick which is in fact very similar to FastPay which also uses consistent broadcast and has also some sort of committee called wardens (c.f. Avarikioti et al.[1]). FastPay represents an interface for transactions in the form the FastPay committee, i.e., the authorities. This is also the reason why RTGS are fast since the handling of transactions happens at one secure "point", without everyone having his hands in it (like in blockchain). It allows for a very low-latency confirmation (in the area of sub seconds) of eventual transaction finality. FastPay can tolerate up to f Byzantine failures out of 3f+1 authorities and can still provide security, liveness and high

performance, unlike traditional RTGS as shown by Baudet et al. [3]. It can also be implemented in many different forms. Here, we focus on the implementation as a side chain, which means it is implemented on the blockchain and used like an add-on. With this implementation, all the possible functions that FastPay can deliver are shown.

When deployed as a side chain, one could look at FastPay like how "money" is handled in a casino. Instead of betting with different kinds of currencies directly (which requires time to evaluate the different currencies) or with bank transfers, which can take some time and would then hinder the flow of the game, everything happens over the casino chips, where the casino itself has control over them and monitors them (for example, the dealer). A player enters the casino, changes some of his money at the counter to some casino chips (where the long process of transferring money is no problem), and then can start to play immediately at every game the casino has to offer and either make a fortune quickly or lose it all. At the same time, the casino itself observes everything (in case there are some illegal actions, a card counter, etc.) and keeps track of how much money is around. If the player then wants to leave the casino and take his money back, he goes back to the casino counter and exchanges his casino chips for his currency. FastPay works similarly (besides the gambling part). One transfers their cryptocurrencies to FastPay (similar to exchanging your money for casino chips) and then can do transactions within FastPay, which happen in sub-seconds. The casino resembles a bit the position of the FastPay committee, where they keep track of the value and validate if a "bet" i.e., a transfer, is in fact valid. When one wants the money out of FastPay, they communicate with the committee and transfer their FastPay value back.

The main contribution of this work is, i) formalizing pseudo code from the implementation of FastPay (c.f. Baudet [4]) as a side chain, explaining them in detail and also presenting each of their properties, ii) discussing, explaining and formalizing how the FastPay protocol and its clients behave in failures or Byzantine cases and try to proceed the protocol, i.e., how a transaction still gets completed when the sender has crashed, or preventing a transaction from completing if it is Byzantine.

In chapter 2, we go over pre-required knowledge and abstractions in order to understand the underlying functionality of FastPay. Chapter 3 shows FastPay in detail, like who is involved in the protocol, how FastPay works, i.e., showing and explaining the steps of different phases, and also discussing how FastPay circumvents a common attack known as double spending. Chapter 4 provides detailed pseudo code for the FastPay protocol as discussed in Chapter 3, showing the structure, functions and properties FastPay has. Also, we look at how clients can drive the protocol to an end in case of a failure and prove the properties FastPay consists of. Finally, chapter 5 contains the conclusion of this work.

Chapter 2

Background

In order to fully grasp the concept and functionality of FastPay, it is necessary to understand some basic mechanics and abstractions (like some already hinted at in the introduction), which will be discussed and elaborated on in the following section.

2.1 Byzantine Process

FastPay is a Byzantine fault tolerance system, but what does Byzantine fault tolerance mean? The name Byzantine originates from an allegory called the "Byzantine general's problem". Some generals want to attack a fortress. They have to agree as a group on the same decision, i.e., attacking the fortress or retreating. The problem now is that if not all do the same actions, the outcome is the worst of all possibilities, so that is to be avoided. A difficulty lies in the voting, which must be sent via unreliable messages to the other generals, which may be lost during the act of delivery. Also, a general can send deliberately or unconsciously wrong messages, which can lead the group of generals to make catastrophic decisions. An example would be if there are 11 generals, where 5 want to attack and 5 want to retreat, the eleventh general may send a message to attack to the attacker group and a retreat message to the retreating group, which leads to the wrong decisions of the whole group and the worst possible outcome. The same can be projected onto a system with processes. A Byzantine process can arbitrarily deviate from its instructions and manipulate the algorithm to reach its goal. In order to circumvent the Byzantine problem, one uses something called a quorum or quorums.

2.2 Quorums

A quorum is the minimum number of processes in a network that need to agree with each other in order to accept some sort of message or event. A majority quorum is formed if, in a network containing N nodes, $\frac{(N+1)}{2}$ or more agree with each other, i.e., more than half. That is why in a network, it has to be guaranteed that at most f nodes can crash, so $N - f$ nodes need to be on-line all the time.

Example 1. *Let $N = 7$, then at least $\frac{(N+1)}{2} = \frac{8}{2} = 4$ nodes have to agree with each other, and at most 3 of the 7 nodes can crash, such that a quorum is still working/trustworthy. As soon as more than 3 nodes crash, then the cluster of nodes decays and is not suitable any more to verify/agree on data/information. This is because it would not be possible to reach an agreement of at least 4 nodes, i.e., there are less than 4 nodes running.*

In Byzantine cases, there is even a more restrictive condition. In a Byzantine environment, there can be at most $N = 3f + 1$ processes, where f is the number of faulty processes. $3f + 1$ is derived from the following facts: There can be two possible problems when processes undergo Byzantine failure. One such problem is when the process does not send any message at all. In particular, as mentioned before, for a distributed system with N nodes to function properly while having f nodes become Byzantine, it

is necessary to achieve a consensus of $N - f$ messages, so $N - f$ processes are required for a quorum. The second problem is when a process equivocates different messages. The most extreme scenario is when a quorum of $N - f$ processes is reached, and f were sent by Byzantine processes. In such a case, the system must still operate normally, which means that $(N - f) - f$ messages must be greater than f messages sent from the Byzantine processes. In order to solve those problems, it is necessary that $(N - f) - f > f$. With a little reshaping, we get $N > 3f$, which means there have to be more than 3 times the amount of f Byzantine processes in order for the system to work properly. The smallest N we can then get is $3f + 1$, where f is the number of Byzantine processes (c.f. Castro et al. [8]). A small example would be that when $N = 7$, there are only $f = 2$ nodes allowed to crash/act Byzantine.
Several algorithms rely on quorums and exploit the fact that two quorums overlap in at least one process. The following theorem shows an even stronger case of overlapping for Byzantine quorums.

Theorem 1. *Two Byzantine quorums overlap in at least one **correct** process.*

Proof. $N-f$ processes are required to form a Byzantine quorum. Now we have two Byzantine quorums: $(N - f) * 2 = 2N - 2f$. If we now plug in the requirement of $N = 3f + 1$ in the Byzantine scenario, we get: $6f + 2 - 2f = 4f + 2 \geq 3f + 1$. Subtracting the right-hand side: $4f + 2 - 3f + 1 = f + 1$. This result shows that at least one process is not Byzantine, i.e., is a correct process. □

2.3 System Model

It is also important to understand how broadcasts in general work and one of the most fundamental abstractions used in them, the link abstractions, in order to realize what FastPay is and how it works. Before we dive into the broadcast, we should learn more about those link abstractions. In the upcoming sections, the important link abstractions will be discussed in more detail. Some of the abstractions which the others are based on, will be mentioned, but will not go further into detail or even be left out if not necessary. The representative source for the following chapter is the book *Introduction to Reliable and Secure Distributed Programming* by Cachin et al. [7].

Every pair of processes is connected with a bidirectional link, those provide full connections between every process. Messages are sent over these links. Multiple processes, each connected with such links, form a network. For a receiver to identify from whom it got the message, the sender has to add enough information to it such that the receiver recognizes the sender. Because of that, each message exchanged over those links is unique. There can be a possibility that an adversary inserts a wrong message into a link between two processes with malicious intent. This can be prevented if cryptographic methods are used to provide the correct sender information. There is also always the possibility that messages get lost during transmission, but this can easily be circumvented by retransmitting the message over and over until it is received. There exist different kinds of link abstractions, implementing different failure abstractions, starting from simple ones and going to more complex ones with higher reliability guarantees. In advance, FastPay with its certain broadcast type, which will be discussed later, uses the Authenticated Perfect Links, abbreviated to *al*. In order to understand how these abstractions work, one has to somewhat understand the previously simpler link abstractions, since they build up on each other.

2.3.1 Link-Abstractions

The first and simplest one, as well as the weakest one in terms of reliability, is the *fair-loss link*, which can retransmit messages or even lose them and works in a crash-stop environment (when a process crashes, it never recovers and stays "dead"). Fair-loss provides three properties. The first one is called *"fair-loss"*. This property guarantees that a link does not systematically drop every message. This means that if both the sender and receiver process are correct (correct in terms of not being adversaries or faulty) and the sender keeps retransmitting a message if it is lost during the process, then the message is eventually delivered. In short, it guarantees that some (but not all) messages get delivered. The second property,

called *"finite duplication"* ensures that, if it happens that the sender has to retransmit a message, it only occurs a finite amount of time. The third property is called *"no creation"* which ensures that no message is created or corrupted by the network, every delivery must be part of a send request.

The *stubborn link* abstraction is an upgraded version of the fair-loss link abstraction. It eliminates the possibility that messages may be lost (since in fair-loss, the messages only get retransmitted a finite amount of times). It does this with an implemented algorithm called "retransmit forever", which exactly does what its name implies. The stubborn link has only two properties, one property called *"stubborn delivery"* replaces the *"fair-loss"* and *"finite duplication"* in the fair-loss link abstraction. This property causes a message sent through such a link to be retransmitted infinitely to the receiver. The receiver then must always check if an incoming message has already been delivered or not. Obviously, the performance takes a massive hit because of the infinite retransmission of messages, that's also why it is not practical in a real-world scenario. The second property is *"no creation"* which is the same as in the fair-loss abstraction.

Perfect link abstraction is one of the higher abstractions, which means it is built upon the simpler (and also less reliable) link abstractions. Its predecessor is the stubborn link abstraction. The perfect link abstraction eliminates the problem of infinitely delivering a message, which the stubborn link abstraction had. It does this by adding mechanisms like detecting and suppressing duplicate messages. This abstraction provides three properties. The *"no creation"* property is the same as in the previous abstractions. The *"reliable delivery"* and *"no duplication"* properties together guarantee that every message sent by a correct process is delivered by the receiver exactly once, if the receiver is also correct. The additional algorithm used in this abstraction is called "eliminate duplicates". What this algorithm essentially does is keeps records of what messages have been delivered. This means that when a message is received, it is only delivered if it is not a duplicate in the list. Note that we are still in the crash-stop scenario, where a crashed process stays crashed. Otherwise, it would be problematic if the list of delivered messages was in the volatile memory of the process, which would be lost if the process restarted and continued to do its work.

All previously observed links work with the assumption that all processes are correct; could crash (because of failure) or something went wrong during the transmission with the links. But if now a process acts wrongly or arbitrarily on purpose, i.e., is Byzantine, then the previous abstraction may violate some of the properties.

Starting with fair-loss point-to-point link abstraction, the property *"fair-loss"* is sound with respect to Byzantine abstraction. The *"finite duplication"* and *"no creation"* properties are not fine, since they permit, even with just one faulty process, that a correct process will deliver any legal message an unbound number of times even without any correct process previously having seen it. A Byzantine process can easily insert a message infinitely and pretend it is from an arbitrary sender process. The upgraded version of the fair-loss link, the stubborn link, would also work the same way as the fair-loss link in a Byzantine scenario. They just add the function of repeatedly retransmitting a message . But the same problem appears in the *"no creation"* property as before. So those two link abstractions are not very useful on their own, but with adding more security in the form of cryptographic authentication, they turn into the so-called *authenticated perfect link* abstraction (see module 1), which is very useful. With this authentication, it eliminates the possibility of manipulating the messages between the two correct processes.

Authenticated perfect link has three properties. The first two, *"reliable delivery"* and *"no duplication"* are the same as in the perfect link abstraction. The third one, called *"authenticity"*, replaces the *"no creation"* property since it is a stronger version of it. The cryptographic authentication method used to authenticate perfect point-to-point links over the stubborn link abstraction is MAC (Message Authentication Code).

Module 1 Interface and properties of authenticated perfect point-to-point links

Name: AuthPerfectPointToPointLinks, **instance** *al*.

Events:
Request: $\langle al,\ Send \mid q,\ m\rangle$: Request to send message m to process q.
Indication: $\langle al,\ Deliver \mid p,\ m\rangle$: Delivers message m sent by process p.

Properties:
AL1: *Reliable delivery*: If a correct process sends a message m to a correct process q, then q eventually delivers m.
AL2: *No duplication*: No message is delivered by a correct process more than once.
AL3: *Authenticity*: If some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p.

2.3.2 Broadcast

There are plenty of different broadcast types, each suited for different environments and error handling. Three of the probably most important types of error handling are fail-silent, fail-crash and fail-arbitrary. In fail-silent, processes do not know or can not detect if another process crashed. Every process is in the unknown, so there has to be some sort of precautionary mechanism to handle such cases. In fail-crash, also known as fail-stop, every process can recognize if the sender process crashes. In such an environment, one assumes that processes can crash. In fail-arbitrary situations, every process can suddenly act maliciously or crash consciously to pursue some ill-natured intent. Here, processes are distinguished into two types: faulty and correct processes. It should be noted that this differentiation is static, which means once a process acts faulty, it stays faulty forever. This static definition is also justified because one cannot make any statements about the behaviour of a Byzantine process. Since the errors/problems can appear arbitrary, there is no uniform procedure to handle those errors/problems.

Byzantine consistent Broadcast A broadcast that can handle Byzantine cases, i.e., Byzantine processes and corresponds to the fail-arbitrary model is called a Byzantine consistent broadcast (module 2). Some simpler broadcasts, like the most basic one, the best-effort broadcast, would fail in this scenario since they assume that processes are correct or crash but do not arbitrarily deviate from their actions. Since they can act arbitrarily, there is no mechanism that can guarantee anything related to their actions. Meaning that it is not possible to define a "uniform" primitive in the fail-arbitrary model.

In some simpler broadcasts, there is a property called "agreement": If a message m is delivered by some correct process, then m is eventually delivered by every correct process. This property would not hold in Byzantine cases and has to be redefined, since in broadcasts that have this property, an application can interpret messages by their sender and from their content and also distinguish the messages from the same sender by adding appropriate tags to the messages when it broadcasts them. But in fail-arbitrary, the sender can act arbitrarily, which means it is not bound to add the correct tags to an application message and can even pretend to have broadcast any message.

In the Byzantine consistent broadcast, in short *bcb*, every instance has a designated sender process called p who broadcasts a message m. If the sender p is correct, then every correct process will deliver m at some point. If the sender is faulty, then some correct processes may deliver the message and others do not, but if two correct processes deliver a message, it is unique. This property is called "consistency" and is a safety property, which is related to the agreement property.

An example of an algorithm using Byzantine consistent broadcast would be the so-called "authenticated echo broadcast" (Algorithm 1). This broadcast relies on the authenticated perfect links and exploits the properties of Byzantine quorums to guarantee consistency, meaning $N > 3f$.

Module 2 Interface and properties of Byzantine consistent broadcast

Name: ByzantineConsistentBroadcast, **instance** *bcb*, with sender p.

Events:
Request: $\langle bcb,\ Broadcast\ |\ m\rangle$: Broadcasts a message m to all processes. Executed only by process p.
Indication: $\langle bcb,\ Deliver\ |\ p,\ m\rangle$: Delivers message m broadcast by process p.

Properties:
BCB1: *Validity*: If a correct process p broadcasts a message m, then every correct process eventually delivers m.
BCB2: *No duplication*: Every correct process delivers at most one message.
BCB3: *Integrity*: If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcasted by p.
BCB4: *Consistency*: If some correct process delivers a message m and another correct process delivers a message m', then $m = m'$.

Algorithm 1 Authenticated Echo Broadcast

1: **Implements:**
2: ByzantineConsistentBroadcast, **instance** *bcb*, with sender p.

3: **Uses:**
4: AuthPerfectPointToPointLinks, **instance** *al*.

5: **upon event** $\langle bcb,\ Init\rangle$ **do**
6: $sentecho := FALSE$
7: $delivered := FALSE$
8: $echos := [\perp]^N$

9: **upon event** $\langle bcb,\ Broadcast\ |\ m\rangle$ **do** //only process s
10: **forall** $q \in \Pi$ **do**
11: **trigger** $\langle al,\ Send\ |\ q,\ [SEND, m]\rangle$

12: **upon event** $\langle al,\ Deliver\ |\ p,\ [SEND, m]\rangle$ **such that** $p = s$ **and** $sentecho = FALSE$ **do**
13: $sentecho := TRUE$
14: **forall** $q \in \Pi$ **do**
15: **trigger** $\langle al,\ Send\ |\ q,\ [ECHO, m]\rangle$

16: **upon event** $\langle al,\ Deliver\ |\ p,\ [ECHO, m]\rangle$ **do**
17: **if** $echos[p] = \perp$ **then**
18: $echos[p] := m$

19: **upon exist** $m \neq \perp$ **such that** $\#(p \in \Pi\ |\ echos[p] = m) > \frac{N+f}{2}$ **and** $delivered = FALSE$ **do**
20: $delivered := TRUE$
21: **trigger** $\langle bcb,\ Deliver\ |\ s,\ m\rangle$

In this algorithm, an instance of Byzantine consistent broadcast exchanges messages in two rounds. In the first message exchange, the sender p who bcb-broadcasts a message m, distributes m to all processes. In the second message exchange, every process that has received the message m from the sender acts as a witness and thus resends the message m in an ECHO to all other processes. Once a process receives more than $\frac{N+f}{2}$ ECHOS for the same message m, it bcb-delivers that m. With this procedure, it is not possible for a Byzantine process to cause a correct process to bcb-deliver a message $m' \neq m$.

All properties hold in this algorithm. The *"validity"* property is satisfied because if the sender is correct, then every correct process al-sends an ECHO message, and every correct process al-delivers at least $N - f$ of them. And since $N - f > \frac{N+f}{2}$, with the assumption $N > 3f$, every correct process also bcb-delivers the message m contained in the ECHO messages. The *"no duplication"* property follows from the algorithm, since a correct process only sends a message m once to all other processes, and each correct process will echo this message, i.e., it does not create a duplicate message, only an echo. The *"integrity"* property holds because if a correct process delivers a message m, received from a correct process p, means that a quorum was reached, i.e., that the message was echoed from more than $\frac{N+f}{2}$ other processes, which results in the message being indeed broadcast by p. The *"consistency"* property follows from the same fact as integrity. In order for a correct process to bcb-deliver a message m, it needs to receive more than $\frac{N+f}{2}$ echo messages containing such m. Such a set of more than $\frac{N+f}{2}$ processes forms a Byzantine quorum, where two such Byzantine quorums will overlap in one correct process. Assume a different process q' has bcb-delivered a message m'. Since q' delivered m', it means it has received a Byzantine quorum of echo messages containing m'. Now the correct process at the intersection of those two Byzantine quorums has sent the same echo message to q and q', which then results in m being equal to m', i.e., $m = m'$.
Performance-wise, the algorithm uses $O(N^2)$ message transmission. This follows from the already mentioned two-round message exchange, where in the second round it involves all-to-all communication.

Chapter 3

FastPay

FastPay is a stand-alone real-time gross settlement system (RTGS) that can be deployed as a side chain. Its main purpose is to settle pre-funded payments in cryptocurrencies or other digital payment currencies. It uses a set of distributed authorities, some of which can be Byzantine, that are responsible for holding client account states, modifying them if necessary, and checking transactions for validity. The Byzantine consistent broadcast is the main driver behind FastPay and is also part of the reason why it achieves low latency for both payment and confirmation finality.

Usually, transactions with cryptocurrencies are done through the blockchains themselves. But the way blockchains handle the guarantee of finality for a transaction takes time and represents a bottleneck. By shifting those transactions somewhere else, like to the side chain, i.e., the FastPay protocol, it increases the throughput. FastPay only relies on the information in the blockchain and its smart contracts.

3.1 Actors in FastPay

In FastPay, there are two to three main actors, depending on how one differentiates them. Every client can either be an account owner (a sender or a recipient) or an authority. The sender is the one that issues a transaction, and the recipient is the one who will receive the result of this transaction. The FastPay committee, consisting of a set of distributed authorities, has the job of checking if a transaction is valid and handing out signed certificates that prove, that such a transaction is indeed valid. Since we are in a Byzantine environment, we assume that at most one-third of the authorities can be malicious. If we reshape this statement, we get that the minimal possible amount of authorities to fulfil the inequality is $3f + 1$, as already shown in section 2.2. Also, FastPay or, respectively, Byzantine consistence broadcast relies on authenticated perfect links, so it is assumed every process knows from whom it gets messages. Since every authority acts on its own and only executes transactions when they are correct, means that even though they work independently, they are consistent across each other (everyone does the same). Since there is no consensus in terms of the order of execution of transactions, it may be possible that authorities may differ in their state, but this is only temporary since every process will execute all correct transactions. This means that every authority will eventually be consistent in a quiescent period, resulting in FastPay's eventual consistency. A short example would be:

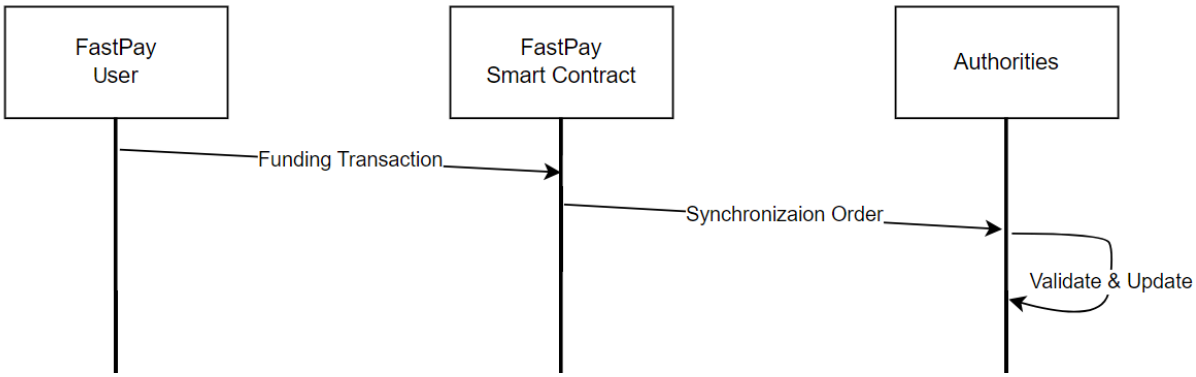
Example 2. *A client has 10 CHF on its account, he buys something for 8 CHF, sells something for 2 CHF and buys something again for 4 CHF. Assuming all transactions are valid. Some authorities get the transactions in the order mentioned, which results in State 1:* $10 - 8 = 2$*, State 2:* $2 + 2 = 4$ *and State 3:* $4 - 4 = 0$*. Some other authorities may get the transactions (due to various reasons) in a different order. State 1:* $10 - 8 = 2$*, State 2:* $2 - 4 = -2$ *and State 3:* $-2 + 2 = 0$*. They still execute the transaction even when the balance goes into the negative, since all the transactions have been validated as correct. In State 2 they are different from each other, but since every one will at some point execute every correct transaction, this is only temporary, and they will eventually end in the same state, i.e., State 3, with rest value of 0.*

3.2 Functionality

There are three scenarios in FastPay. The first one is adding funds from the Primary (i.e., the smart contract and the primary store of information would be the blockchain) to a FastPay account, the second one is the other way around, shifting funds back from the FastPay account to the Primary. The third scenario is the *main act*, a transaction between two FastPay accounts. Those scenarios, which are derived from the FastPay protocol from Baudet et al. [3], will be looked at in more detail in the following sections.

3.2.1 Adding funds from the Primary to FastPay

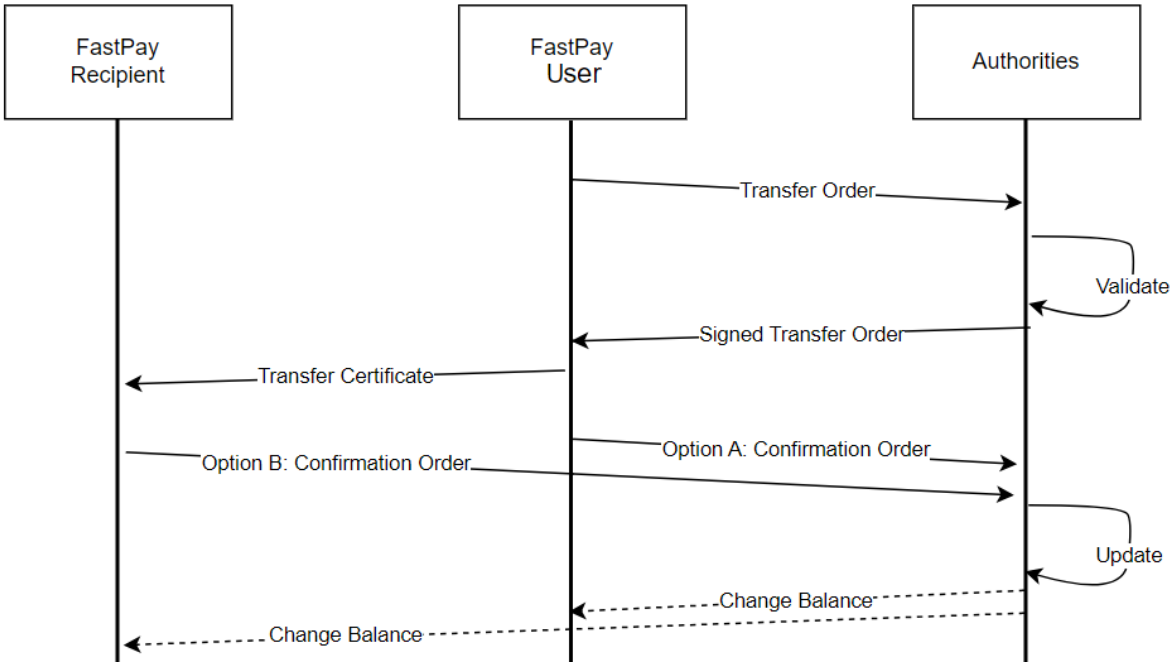
The client/owner of the FastPay account creates a Primary transaction, which starts by sending a payment to the FastPay smart contract in the blockchain. This payment is called a "funding transaction", and this transaction contains the recipien'st FastPay account address and the specified amount of value that has to be transferred. Once this Primary event is executed (i.e., the smart contract has validated it and decreased the sender's balance), the FastPay smart contract creates a Primary event. For simplicity, we model such an event as a synchronization order. This synchronization order informs the authorities of a change in the state of that smart contract. It also contains a unique and always-increasing sequential transaction index. This transaction index is necessary to prevent some possible problems that could appear without it, for example, missing transactions or the same transaction appearing multiple times, which would increase the funds multiple times to the price of one. Each authority, once it receives such a synchronization order, will check its content. Does the index match the expected one from the authorities? If that's the case, it increments their previous recorded index in the global state, so it is up-to-date for the next synchronization order from the smart contract. In the event that the assigned recipient does not exist, it will create a new FastPay account. Finally, it will increase the balance of the recipient's FastPay account accordingly to the specified amount and store this synchronization order in the synchronized list for this recipient (to keep track of the transferred funds from the Primary to the FastPay account).



3.2.2 Transaction within FastPay

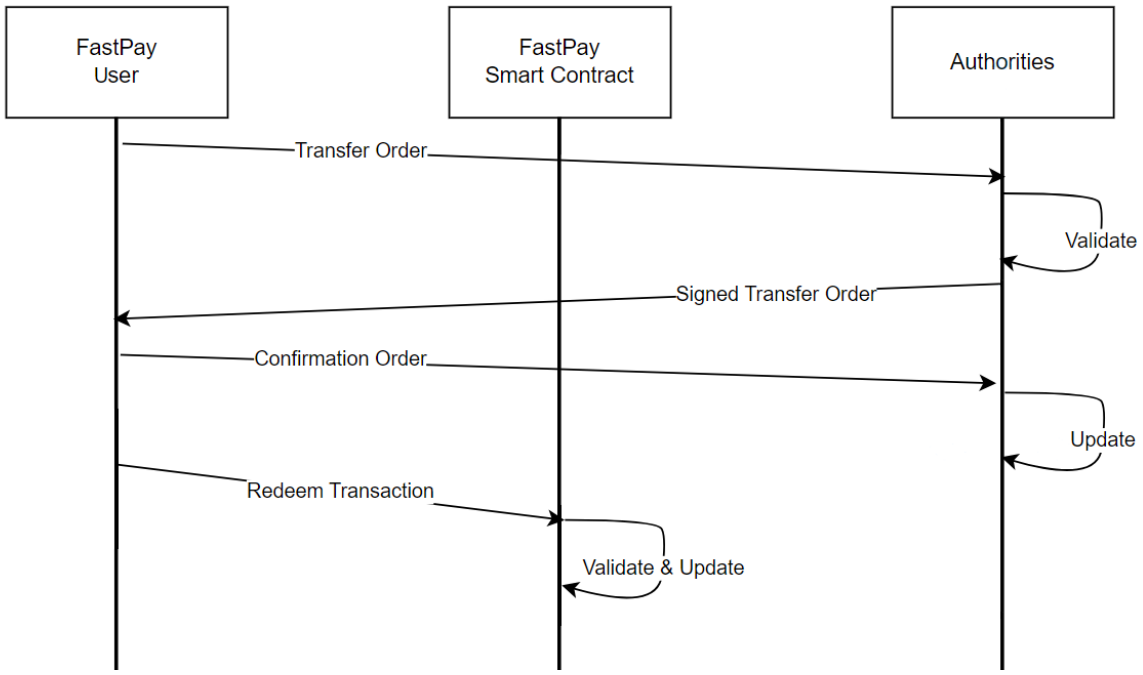
Now that there is some value on the FastPay account, it is possible to do transactions within FastPay. A typical transaction is as follows: The owner of the FastPay account creates a transfer order m, containing the information of the recipient's FastPay address, the specified amount of value to transfer to the recipient, the next sequence number in the sender's account, and the sender's signature (which may be trivial from the perspective of authorities because of authenticated links but is necessary for other clients who may proceed the protocol if the sender crashes). This transfer order then gets broadcasted via bcb by the client himself to all authorities. Each authority then checks that the signature is valid, that there is no pending transfer or is from the same person, that the specified amount is positive (no *negative* transactions), that the sequence number matches the next expected one, and that the balance of the sender is sufficient to make the desired payment, i.e., no overspending. If everything is all right, the authority sets this transfer as pending and sends back a signed version of the transfer order, which is also stored by the authority itself. The client, i.e., the one that initiated the transaction, collects those signed transfer

orders from the authorities. Once he has collected a quorum of those signed transfer orders, he forms with those signatures and together with the transfer order, a transfer certificate, which he will send to the recipient as proof that the payment will proceed. To conclude the transaction, either the sender client or the recipient broadcasts the transfer certificate to the authorities, this step is called a "confirmation order". Upon reception of a confirmation order for the current sequence number, each authority checks if a quorum of signatures has been reached. If that is the case, then it proceeds to decrease the balance of the sender according to the specified amount, increments the sequence number (to ensure that this transaction will only be executed once), and sets the pending order back to be $\emptyset$. Finally, each authority adds the certificate to the confirmed list for that certain client and increments the balance for the specified recipient.



3.2.3 Shifting funds from FastPay to the Primary

If the owner of the FastPay account now wants the funds back to the Primary (for various reasons), he has to create a Primary transfer order, which he signs using his own account key and broadcasts it to the authorities. This is basically a normal transfer order, but instead of a FastPay account as a recipient, it is the Primary address. The authorities do the same as in transactions within FastPay. They validate the transfer order, and if everything is all right, they send back a signed transfer order to the client. Once the client has collected a quorum of signatures, he creates a certified (Primary) order, also called a transfer certificate. This transfer certificate then gets broadcasted to the authorities again to confirm the transaction and unlock future spending for this account. Upon receiving a confirmation order (i.e., the transfer certificate), each authority checks that a quorum of signatures has been reached and the account sequence number corresponds to the expected one. If this is the case, they then set the pending order back to be $\emptyset$, increment the sequence number, and decrease the account balance. One of the last steps is for the recipient (which is most of the time the owner of the FastPay Account) to send a redeem transaction to the Primary, i.e. the smart contract on the blockchain, which also contains the transfer certificate. Once the smart contract receives such a valid redeem transaction (i.e., a quorum of signatures was reached for the transfer certificate), it checks that the sequence number is not in the Primary redeem log of the sender (in order to prevent reuse), updates this said redeem log, and adds the specified amount in the redeem transaction into the recipient's Primary account.



3.3 Double Spending

A problem that one can think of that can happen in FastPay is double spending, since the acceptance of the transaction happens before the money is sent, i.e., the transaction is final. Double spending is the process of trying to pay for two different things with the "same" money. The buyer basically creates two transactions simultaneously and promises the two sellers that he will pay for the good/service. In the end, he will get the goods/services, and he has created new money out of thin air, which one of the sellers will receive instead of the "real" one, resulting in inflation of the used currency. Usually one solves this problem by using full consensus, i.e., total order broadcast, but as it has been shown by Guerraoui et al. [12] this is not necessary. Double spending in FastPay is in fact not possible because of some underlying properties and countermeasures it uses to exactly prevent such things from happening, without the need for consensus. There are two main points that help prevent double spending. The transaction index and the pending value. Every authority checks, when it receives a transaction, that no previous transfer is pending or the transaction index has already been used, i.e., does not match the expected one by the authority. A correct authority would not send back a signed transfer order if there is a pending transaction or a conflict with the transaction index.

Obviously, one may ask what happens when a Byzantine client sends two different transactions containing the same transaction index but having no pending transactions to the authorities? Since the authorities get the transaction and see that there is no pending transaction and the transaction index matches the expected one, they will sign it and send it back. But this happens to only one transaction, whichever one they received first, because if they receive a second transaction with the same transaction index, it would cause a conflict, and they would not sign the transaction, and there would already be a transfer in their pending value. Together with the fact that the client needs to receive a quorum of signatures for both transactions, it is also impossible to double spend. This is because it may be possible that he does reach a quorum for one transaction, i.e., more than half of the authorities got one transaction, but this is only possible for *one* transaction, the rest of the authorities, which represents the minority, got the second conflicting transaction, which then is not able to form a quorum of signatures. Even when the FastPay protocol tries to finish the left-open transaction, it will not work since the transaction index has already been used by the majority of the authorities, resulting in a lockout for that malicious client. More details on how the FastPay protocol tries to complete unfinished transfers (due to crashes) and the proof that double spending is not possible in FastPay follows later.

Chapter 4

Formalizing FastPay

This section provides the pseudo code for the different actors and scenarios mentioned before in the Functionality section. They are based on the procedure by Baudet et al. [3] and the implementation by Baudet et al. [4]. For simplicity, the (cascading) acknowledgement messages are left out in the pseudo codes, as they provide no meaningful usage to the functionality and understanding of the protocol.

A big difficulty that arose in formalizing FastPay was the way the different modelled entities communicate with each other and trigger the events at the same time. The easiest way to model would have been to assume that every client is an authority, and vice versa. Then the communication would have been simply within the module, but that does not fit the spirit of FastPay. A good approach would have been to model the committee as a module that is implemented by the authorities. This way, the client could simply interact with the committee without knowing how the committee is implemented. The problem here would be that, in order to guarantee that a correct authority receives the information, it would need to send it to $f + 1$ authorities to be sure that at least one correct authority received the information. This means the authority then needs to validate if $f + 1$ authority received the information, which would lead to a lot of overhead and unnecessary long code.

The way it is modelled now is that the client and the authority are separated. But in order for this to work properly, it is necessary to use some modified versions of the Byzantine consistent broadcast called Hybrid-Consistent-Broadcast (module 3). This broadcast is basically the same as the original version, which means it inherits all its properties. The difference is that only one set of processes can use one function, i.e., the broadcast function and the other set of processes can use the other function, the deliver function. The reason for this adjustment is that instead of having only one set of processes, we have two, i.e., a set A consisting of the clients and the smart contract and set B consisting of the authorities Π, where not every set can use the broadcast function and/or receive such broadcasts, i.e., deliver. For example, client y is not supposed to receive a broadcast from client x, only the authorities are supposed to receive such a broadcast, or the authorities are not supposed to broadcast something to the clients (authorities communicate through authenticated perfect links with the clients). The implementation of Hybrid-Consistent-Broadcast is similar to the *Byzantine consistent broadcast* with the addition of those mentioned set specification.

First, we look at the interface of the FastPay client described in module 4. This module consists of a total of five events. There are three request events: the first request event is for broadcasting a new transfer order m from the sender to the authorities; the second event is for collecting signed transfer orders from authorities in order to form a transfer certificate for a certain transfer order, which then gets broadcasted to the authorities; and the third and final request event is for requesting information if a process wants to proceed a transfer order from someone else who may have crashed. The two indication events are i) triggering the smart contract to generate a Primary event in order to transfer funds from the Primary to FastPay (with the given information provided by the account owner) and ii) triggering the smart contract to finish transferring funds from FastPay back to the Primary. The FastPay client is characterized by two properties. The *"correctness"* property guarantees every authority will eventually receive the transfer order, as long as a correct client broadcasts it to at least one correct

Module 3 Interface and properties of Hybrid-Consistent-Broadcast

Name: Hybrid-Consistent-Broadcast, **instance** *hcb*

Events:

Request: $\langle hcb,\ Broadcast \mid B,\ m\rangle$: Broadcasts a message m to all processes in B. Triggered only by the processes in A.

Indication: $\langle hcb,\ Deliver \mid p,\ m\rangle$: Delivers message m broadcasted by the process $p \in A$. Only occurs at processes in B.

Properties:

HCB1: *Validity*: If a correct process $p \in A$ broadcasts a message m, then every correct process $q \in B$ eventually delivers m.

HCB2: *No duplication*: Every correct process $p \in A$ delivers at most one message.

HCB3: *Integrity*: If some correct process $q \in B$ delivers a message m with sender $p \in A$ and the process p is correct, then m was previously broadcasted by p.

HCB4: *Consistency*: If some correct process $q \in B$ delivers a message m and another correct process in B delivers a message m', then $m = m'$.

authority. The second property *"termination"* ensures that the transfer order will eventually get executed by all authorities when a correct client broadcasts the transfer certificate. The pseudo code 1 belongs to module 4. This pseudo code represents the sender/initiator of a transaction and also the recipient. p represents the initiator of a transaction; q is the recipient; α represents an authority; and Π represents the whole authorities. origSender stands for the original Sender, while currSender stands for the current Sender. The boolean $validEvent(m)$ on line 30 validates if the message m consists of an event (E) and a parameter ($param$) and if such event exists with the provided parameter. Only if this is true can the corresponding event be triggered with the given parameter. If this boolean results in false, then it is possible that a different *al* deliver event will be triggered, where m does not contain an event E, only a *parameter* (see line 42 or inside the function on line 55).

The FastPay Client has two starting events (lines 32 and 45), one for each of the mentioned scenarios in the section 3.2 about the Functionality (except the funding transaction, since this happens at the Primary and is communicated to the smart contract). Both events are nearly the same but differ in the end result since they slightly differ in the information they contain. The transfer order is used for transactions within FastPay, this means only clients and authorities are involved. The Primary transfer order is for transferring funds back to the Primary, so out of FastPay, which additionally uses the smart contract. The transfer order and the Primary transfer order both contain the current sequence number of the sender, the specified amount to transfer, the signature of the sender, and the recipient's address which in the Primary transfer order is the Primary and in the (normal) transfer order is a FastPay account. There are three receiving events, one for the transaction within FastPay, i.e., receiving of the signed transfer orders m_α from the authorities (line 35) where they get validated for correctness with the boolean $valid(m_\alpha)$ on line 36 (i.e., does the m_α correspond to the original transfer order m and comes it in fact from an authority) and then put into a transfer certificate. The other is for receiving the signed primary transfer order from the authorities for shifting funds from FastPay to the Primary (line 48) and the last receiving event is for the recipient. Once he receives a valid transfer certificate, he can then proceed to broadcast the received certificate to the authorities (to finish the transfer, in case the sender has not already done that) and, outside the code, deliver the promised goods/services (line 42).

There is also a section for requesting information, which is split into two events. The first one on line 55 is for triggering the request by the authorities. The *AutheticatedPerfectLinks* are being used for requesting information from authorities, not the *Hybrid-Consistent-Broadcast*, because it is not necessary to be consistent and the way it is implemented in the pseudo code now is basically a best-effort broadcast, which suffices handling the request for information. The second event on line 59 is for receiving the information (which is getting validated correctness) and triggering the next function by the authorities to proceed with the protocol. The requesting order for information from authorities in the first event happens randomly, shuffle is used in the pseudo code, which creates a random uniform distributed

list of authorities. From a security point of view, uniform is not mandatory since it can not do any harm by requesting information in a strict order, and only the correct authorities will have valid information that can be used further. The reason for being randomly uniform is that it can be faster to receive correct information from authorities who have received such information compared to a strict order, where the worst-case scenario would be when only the last authority in that order has the information. If a process receives invalid information from a Byzantine authority, he still has to broadcast this information to all other authorities, which validate the information and decide if it is correct and can be accepted or not. On top of that, the requester also checks if the received information is signed by the original sender, this way, it is not possible for a Byzantine authority to create a new transfer order or a transfer certificate for a certain user. More on the request for information will be discussed later in Section 4.1.

Module 4 FastPay Client

Name: FastPay Client, **instance** *fpc*

Events:

Request: $\langle fpc,\ transfer\_order \mid m\rangle$: Broadcast a transfer order m to the authorities.

$\langle fpc,\ receive\_sign \mid m_\alpha\rangle$: Collecting signed validated transfer orders m_α from each authority α and form with them a transfer certificate which will be broadcasted to all authorities.

$\langle fpc,\ information\_request \mid origSender\_data\rangle$: Requesting information from authorities for an unfinished transfer with given sender data.

Indication: $\langle fpc,\ receive\_fundTrans \mid \text{smart\_contract},\ ft\rangle$: Trigger the smart contract to generate a Primary event using the information in ft.

$\langle fpc,\ receive\_transferCert \mid \text{smart\_contract},\ \text{transfer\_certificate}\rangle$: Trigger the smart contract to finalize the transfer from FastPay to Primary.

Properties:

FPC1: *Correctness*: If a correct client p broadcasts a transfer order m to at least one correct authority, then every authority will eventually receive the transfer order m.

FPC2: *Termination*: If a correct client p broadcasts a transfer certificate for a transfer order m to at least one correct authority, then the transfer order m will eventually get executed by all authorities.

Next, we look at the interface for the authority, which is shown in module 5. This module has five events, or more precisely, five request events, where the first one is for receiving a synchronization order, which needs to be validated and, if valid, go through the steps of incrementing the recipient's FastPay balance and, if no such account exists, creating a new one in order to deposit the funds. The second one is for receiving, validating and sending back a transfer order. The third is basically the same as the second event, but for a primary transfer order. The fourth one is for doing the actual transfer, i.e., changing the account balances when receiving a valid transfer certificate. The fifth event is for returning information about a requested account. The authority is defined by two properties. The property *"eventual finality"* assures that eventually every authority will record, i.e., complete, a valid transfer order, broadcasted from a correct client. The property *"no double spending"* guarantees, as the name implies, that it is not possible for a Byzantine client to trick the system, i.e., the authorities, by sending two conflicting transfer orders. The pseudo code for the authorities is shown in pseudo code 2. Each authority contains a database of all existing FastPay accounts and their information. Such database contains information like account balances, the expected current sequence numbers of the accounts, the previous recorded index from the smart contract, a variable for a pending transaction, a list for confirmed transactions, and a list synchronized to keep track of the transferred funds from the Primary to the FastPay account. Because of the usage of authenticated perfect links, it is assumed that all authorities will be consistent (although it may be possible for a short duration that this is not the case, but it will eventually get to it at some point, see example 2). The authorities are responsible for validating orders and updating their

Pseudo Code 1 FastPay Client

22: **Implements:**
23: Module FastPay Client, **instance** *fpc*.
24: **Uses:**
25: AuthenticatedPerfectLinks, **instance** al.
26: Hybrid-Consistent-Broadcast, **instance** *hcb*, where: $A =$ Clients and the Smart Contract, $B = \Pi$ (Authorities).

27: **upon event** $\langle fpc,\ Init \rangle$ **do**
28: $current\_sequenceNumb := 0$
29: $signdMessageList := \emptyset$

30: **upon event** $\langle al,\ Deliver\ |\ p,\ m = (E,\ param) \rangle$ **such that** validEvent(m) **do**
31: **trigger** $\langle fpc,\ E\ |\ param \rangle$

32: **upon event** $\langle fpc,\ transfer\_order\ |\ m \rangle$ **do**
33: $current\_sequenceNumb = current\_sequenceNumb + 1$
34: **trigger** $\langle hcb,\ Broadcast\ |\ \Pi,\ (receive\_transferOrder,\ m) \rangle$

35: **upon event** $\langle fpc,\ receive\_sign\ |\ m_\alpha \rangle$ **do**
36: **if** valid(m_α) **then** //validate m_α if it is from authority and correspond to the sent m
37: $signdMessageList = signdMessageList\ \cup\ \{m_\alpha\}$
38: **if** $|signdMessageList| > N - f$ **then**
39: transfer_certificate $= \{m\}\ \cup\ signdMessageList$ //m is the original transfer order
40: **trigger** $\langle al,\ Send\ |\ q,$ transfer_certificate$\rangle$ //as confirmation for recipient
41: **trigger** $\langle hcb,\ Broadcast\ |\ \Pi,\ (receive\_transferCert,$ transfer_certificate$) \rangle$

42: **upon event** $\langle al,\ Deliver\ |$ transfer_certificate$\rangle$ **do** //Only done by the recipient q
43: **if** |transfer_certificate$\backslash\{m\}| > N - f$ **then**
44: **trigger** $\langle hcb,\ Broadcast\ |\ \Pi,\ (receive\_transferCert,$ transfer_certificate$) \rangle$

45: **upon event** $\langle fpc,\ primary\_transferOrder\ |\ po \rangle$ **do** //This is basically the same event as on line 32
46: $current\_sequenceNumb = current\_sequenceNumb + 1$
47: **trigger** $\langle hcb,\ Broadcast\ |\ \Pi,\ (receive\_primaryOrder,\ po) \rangle$

48: **upon event** $\langle fpc,\ receive\_sign\ |\ po_\alpha \rangle$ **do** //This is basically the same event as on line 35
49: $signedMessageList = signedMessageList\ \cup\ \{po_\alpha\}$
50: **if** $|signedMessageList| > N - f$ **then**
51: transfer_certificate $= \{po\}\ \cup\ signedMessageList$
52: **trigger** $\langle hcb,\ Broadcast\ |\ \Pi,\ (receive\_transferCert,$ transfer_certificate$) \rangle$
53: **trigger** $\langle fpc,\ receive\_transferCert\ |$ smart_contract, transfer_certificate$\rangle$
54: //Line 53 is modelled as a redeem Transaction

//The following events are for requesting data
55: **upon event** $\langle fpc,\ information\_request\ |\ origSender\_data \rangle$ **do**
56: randomList = shuffle.Π //shuffle randomly lists the authorities in to the randomList
57: **for each** α **in** randomList **do**
58: **trigger** $\langle al,\ Send\ |\ \alpha,$ origSender_data$\rangle$

59: **upon event** $\langle al,\ Deliver\ |\ m \rangle$ **such that** valid(m) **then** //Check the signature of m matches the origSender
60: **trigger** $\langle hcb,\ Broadcast\ |\ \Pi,\ (receive\_transferOrder,\ m) \rangle$

information about accounts. They are also the contact point for requesting information about unfinished transfer orders. If the authorities receive a synchronization order (line 75) they check the synchronization order index and, if it is valid, update the information about that user's account. In the event that there is no such recipient account, a new one will be created (see line 79). If the authorities receive a transfer order m from p, which contains his current sequence number, the specified amount and the address of q, i.e., the recipient, they need to validate it and, if correct, send back a signed version of m. The function $\sigma(...)$ represents the signing function of messages. The boolean $validEvent(m)$ on line 73 is the same as explained in FastPay client. $Validate(m)$ on line 84 is the process of validating if the transfer order is actually signed by the sender ($\sigma(m) =$ signature of p), that there is no currently pending transaction (i.e., $pending^p = \emptyset$), that the specified amount in the order is not negative ($m.amount > 0$), that the sequence number matches the expected next one by the authorities, such that it does not skip any transactions ($m.sequence = current\_sequence^p$) and that the balance of the sender is sufficient ($balance^p > m.amount$) to make the specified payment. If then everything is valid, i.e., results in true, the authority sends back a signed version of m, denoted as m_α. The transfer certificate, which first appears in line 39 of the pseudo code 1, contains the initial transfer order m and at least $N - f$ signed versions from the authorities, i.e., m_α. Once an authority receives such a transfer certificate, it checks if it contains $N - f$ signed versions of m and if this is the case, updates its database (line 88), i.e., records the transfer. If they receive a Primary order transfer certificate, they do exactly the same, with a minor difference. Since this difference is just one line of code, instead of creating/writing the same event again, it is noted with a comment (on line 93). The difference is that the authority does not change the balance of the recipient account since it does not have access to the Primary account, only the smart contract executes this update when it receives the same transfer certificate from the client. Lastly, there is the event for handling information requests. The authority starts by checking if there is a pending order for that certain user, and if there is, it sends this back or if the pending is empty, checks the completed list for that user with the given sequence number and returns the certificate. Why this requesting function is necessary will be discussed later in section 4.1.

Module 5 FastPay Authority

Name: FastPay Authority, **instance** *fpa*

Events:

Request: $\langle fpa,\ receive\_syncOrder \mid \text{synchronization\_order}\rangle$: Validates the funding transaction ft, if no such account exists, creates a new one.

$\langle fpa,\ receive\_transferOrder \mid m\rangle$: Receiving a transfer order m and send back a signed version.

$\langle fpa,\ receive\_primaryOrder \mid po\rangle$: Receiving a primary transfer order po and send back a signed version.

$\langle fpa,\ receive\_transferCert \mid \text{transfer\_certificate}\rangle$: Updates the information/data of the specified accounts in the transfer certificate.

$\langle fpa,\ information\_request \mid \text{currSender, origSender\_data}\rangle$: Returns Information about either an unfinished or finished transfer order with the given origSender data.

Properties:

FPA1: *Eventual Finality*: If a correct client p broadcasts a transfer order m, then every correct authority eventually records m.

FPA2: *No Double Spending*: If a client p broadcasts a transfer order m and another conflicting transfer order m', then every correct authority will only record one and the same transfer order.

The last actor to be looked at is the smart contract. Its interface is represented in module 6. It consists of two request events. The first request event initiates a funding transaction, where funds are shifted from the Primary to the FastPay account. In this funding transaction, there is the information about the address of the recipient FastPay account and the specified amount of value to transfer. This funding transaction comes from the Primary. The other request event is the finalization of the process of shifting funds from a FastPay account back to the Primary. The smart contract has, besides its typical properties like being

Pseudo Code 2 FastPay Authority

Implements:
    Module FastPay Authority, **instance** *fpa*.
Uses:
    AuthenticatedPerfectLinks, **instance** al.
    Hybrid-Consistent-Broadcast, **instance** *hcb*, where: $A =$ Clients and the Smart Contract, $B = \Pi$ (Authorities).

upon event $\langle fpa,\ Init \rangle$
    $prevRecordedIndex = 0$ //Used for the smart contract index
    $balance = \emptyset$
    $current\_sequence = 0$ //Used for the sender indexes
    $pending = \emptyset$
    $confirmed = \emptyset$
    $synchronized = \emptyset$

upon event $\langle hcb,\ Deliver \mid p,\ m = (E,\ param) \rangle$ **such that** validEvent(m) **do**
    trigger $\langle fpa,\ E \mid param \rangle$

upon event $\langle fpa,\ receive\_syncOrder \mid$ synchronization_order$\rangle$
    if synchronization_order.index $= prevRecordedIndex$ **then**
        $prevRecordedIndex = prevRecordedIndex + 1$
        if synchronization_order.account **not exists then**
            create new FastPay account
        $balance^{syr} = balance^{syr} +$ synchronization_order.amount
        $synchronized^{syr} = synchronized^{syr} \cup \{$synchronization_order$\}$
            //syr stands for synchronization_order.recipient

upon event $\langle fpa,\ receive\_transferOrder \mid m \rangle$ **do**
    if validate(m) **then**
        $pending^p = m$
        $m_\alpha = \sigma(m)$
        trigger $\langle al,\ Send \mid p,\ (receive\_sign,\ m_\alpha) \rangle$

upon event $\langle fpa,\ receive\_transferCert \mid$ transfer_certificate$= \{m, \cdots\} \rangle$ **do** //Modelled as Confirmation Order
    if m.sequence $= current\_sequence^p$ **and** $|$transfer_certificate$\backslash\{m\}| > N - f$ **then**
        $balance^p = balance^p - m$.amount
        $current\_sequence^p = current\_sequence^p + 1$
        $pending^p = pending^p \setminus \{m\}$ //Reset the pending, i.e. set it back to being empty
        $balance^q = balance^q + m$.amount //This is ignored for Primary transfer order certificate
        $confirmed^p = confirmed^p \cup \{$transfer_certificate$\}$

upon event $\langle fpa,\ receive\_primaryOrder \mid po \rangle$ **do**
    if validate(po) **then**
        $pending^p = po$
        $po_\alpha = \sigma(po)$
        trigger $\langle al,\ Send \mid p,\ (receive\_sign,\ po_\alpha) \rangle$

upon event $\langle al,\ Deliver \mid$ currSender, origSender_data$\rangle$
    trigger $\langle fpa,\ information\_request \mid$ currSender, origSender_data$\rangle$

upon event $\langle fpa,\ information\_request \mid$ currSender, origSender_data$\rangle$ **do**
    $\gamma =$ origSender_data.sender
    if $pending^\gamma \neq \emptyset$ **then**
        $m' = pending^\gamma$
        trigger $\langle al,\ Send \mid$ currSender, $m' \rangle$
    else
        if exists transfer_certificate $\in$ confirmed$^\gamma$ **such that** transfer_certificate.sequence = origSender_data.sequence
            trigger $\langle al,\ Send \mid$ currSender, transfer_certificate$\rangle$
        else
            return error message: No information found.

self-verifying, self-enforcing and tamper-proof, two properties related to FastPay. The first property, *"no currency loss"* guarantees, as the name implies, that the transferred currencies will end up on a FastPay account. The second property, *"no currency generation"* ensures that currencies are not created out of thin air, i.e., the balances of both accounts will be changed. The pseudo code for the smart contract can be seen at pseudo code 3. It handles the shifting of funds from Primary to fast pay and vice versa. It is like a starting bridge between the FastPay client and the authorities. Once it receives a funding transaction, it first validates it, i.e., checks if the balance of the sender is sufficient to make the desired transfer, this happens at line 122. If valid, it creates a synchronization order containing the information from the funding transaction and adds his currentSelfIndex to it. After that, it increases its SelfIndex by one (to prevent multiple uses of the same funding transaction), decreases the sender's Primary balance, and informs the authorities of a change of state (line 121). On the other hand, when the smart contract receives a redeem transaction, i.e., a transfer certificate, it checks if it is valid (a quorum of signatures was reached and is not already in the Primary redeem log). If that is the case, it updates the balance for the Primary and updates its Primary redeem log for that user, such that this redeem transaction can not be cashed in multiple times (line 128).

Module 6 FastPay Smart Contract

Name: FastPay Smart Contract, **instance** *fps*

Events:

Request: $\langle fps,\ receive\_fundTrans \mid ft \rangle$: Initiate a transfer from Primary account to FastPay.

$\langle fps,\ receive\_transferCert \mid$ transfer_certificate$\rangle$: Finalize a transfer from FastPay to Primary account.

Properties:

FPS1: *No currency loss*: After receiving a correct funding transaction, there will exist a FastPay account with the specified amount transferred to it.

FPS2: *No currency generation*: After receiving a valid transfer certificate, the sender balance has been subtracted by the specified amount and the recipient will receive this amount.

4.1 How FastPay clients drive the protocol further

In FastPay, mainly the client itself, who wants to make a transaction, is responsible for ensuring that the transaction he sends is correct and that the protocol finishes, since the authorities are only there to validate a transaction for correctness and, if the transaction is correct, change the account balances. But what happens if the client itself, at some point, fails and goes off-line? We observe what happens in transactions within FastPay (3.2.2). Shifting funds from FastPay to the Primary will be similar. Adding funds from the Primary to FastPay does not provide much, since if the sender fails, there is nothing to do (since no one knows that something happened).

A critical point in a transaction within FastPay is the same as just previously mentioned in adding funds from the Primary to FastPay. If the sender fails to broadcast a transfer order, then there is no possibility to recover from that since no one knows that the sender has (tried to) send something. This brings us to the fact that the only action the sender *must* perform, in order for his transaction to eventually proceed, is to form a transfer order with his signature and send it to at least one correct authority. This means that when the sender crashes during sending the transfer order and some authorities have received it but not all, it is possible for others to proceed the protocol by requesting information from the authorities that have received the transfer order. The same goes for if the sender failed between receiving back signed transfer orders from authorities and sending the transfer certificates. Thanks to the signature of

Pseudo Code 3 FastPay Smart Contract

112:**Implements:**
113: Module FastPay smart contract, **instance** *fps*.
114:**Uses:**
115: AuthenticatedPerfectLinks, **instance** al.
116: Hybrid-Consistent-Broadcast, **instance** *hcb*, where: $A =$ Clients and the Smart Contract, $B = \Pi$ (Authorities).

117:**upon event** $\langle fps,\ Init \rangle$
118: $currentSelfIndex = 0$
119: $total\_balance = 0$
120: $PrimaryRedeemLog = \emptyset$

121:**upon event** $\langle fps,\ receive\_fundTrans \mid ft \rangle$ **do**
122: **if** validate(ft) **then**
123: synchronization_order $= ft$
124: synchronization_order.index $= currentSelfIndex$
125: $total\_balance^{sender} = total\_balance^{sender} - ft.amount$
126: $currentSelfIndex = currentSelfIndex + 1$
127: **trigger** $\langle hcb,\ Broadcast \mid \Pi,\ (receive\_syncOrder,$ synchronization_order$) \rangle$

128:**upon event** $\langle fps,\ receive\_transferCert \mid$ transfer_certificate $= \{m, \cdots\} \rangle$
129: **if** |transfer_certificate$\backslash\{m\}| > N - f$ **and** transfer_certificate $\notin PrimaryRedeemLog_p$ **then**
130: $PrimaryRedeemLog_p = PrimaryRedeemLog_p \cup \{$transfer_certificate$\}$
131: $total\_balance^{tcr} = total\_balance^{tcr} +$ m.amount
132: //tcr stands for transfer_certificate.recipient

the original sender, it is not possible for a malicious client to create a new transfer order under that user's name. Additionally, FastPay prevents double spending (see property FPA2), which makes it robust in a real-world scenario.

As mentioned by Baudet et al. [3], any party in possession of a signed transfer order may attempt to make a payment progress concurrently. And as long as the sender is correct, the protocol will conclude (and if not, it may only lock the account of the faulty sender). In fact, even the authorities may also attempt to complete the protocol once they receive a valid transfer order. This results in the fact that anyone may request a transfer order that is partially confirmed by any authority and tries to form a certificate, which it then submits as a confirmation order in order to complete the protocol. Baudet et al. went no further in specifying who may complete the transaction, when, and under what circumstances. The most intuitive actor in question, who would proceed with the transaction, is the recipient, since he is the one that wants to receive the funds and is, besides the authorities, one of the first to know about the transfer order. If everyone else could proceed the transaction, it would pose further difficulties. Like, how do the others know when a transfer order is not completed due to the sender's crash, i.e., knowing the sender's status? Or what would be the incentive to complete such a transfer order? How could you prevent others from completing the transaction if the sender has not crashed when there is a reward for completing a transaction? And many more open questions. Answering those questions is out of scope for this bachelor's thesis, so we will go with the assumption that the recipient will be the one who proceeds to complete the transaction, in case the sender crashes during the process.

A FastPay client can request information about a certain user from the authorities (see pseudo code 1 line 55) in order to get a partially validated transaction order and transform it into a confirmation order or directly get a transfer certificate from an authority. A client requests information in a random order from each authority (since it is not known who has some information). Each authority then checks if there is a transaction in the pending state (see pseudo code 2 line 104) for that requested client. If that is the case, the authorities then send back the transaction order (which is signed by the original sender). If the client receives a transfer order and has validated it as correct, it then proceeds with the protocol with the other

authorities that did not receive this order until it can form a transfer certificate containing $N - f$ valid signed messages from authorities and complete the transaction. The client who requests information can not know if the information he receives is 100% valid. Because in order for a client to be sure that the transfer order he received from an authority is correct, it would need to receive a quorum of that transfer order, but this would mean that already $N - f$ authorities have received and accepted the transfer order, and then the protocol would proceed anyway. So the client assumes that the information it receives from authorities is correct and valid, i.e., an authority accepts only valid transfer orders, which means they also send back only valid information. But if there is a Byzantine authority that sends back invalid information for a request, it does not harm the protocol or the client that tries to proceed the protocol, because if it receives an invalid transfer order from a Byzantine authority, the protocol will not proceed since the client has to broadcast the received transfer order to all other authorities in order to receive their signed version in order to form a certificate, which it can only do once it receives a quorum of signed transfer orders. This means it is not possible to form such certificates, because correct authorities will not accept the invalid transfer order, and only the original sender may then be in a blocked stage. The client who tried to proceed with the protocol is only "blocked" from proceeding with the protocol for that certain user, not for himself or others.

Another recoverable state would be if a sender nearly completed the whole process but crashed during the sending of the transfer certificate to all authorities (and maybe to the recipient itself). This means some authorities may have received the transfer certificate and completed the transfer on their side, but some authorities did not. If the sender sent the transfer certificate to the recipient first (before he sent them to the authorities) as a confirmation that the transfer will conclude and then crashed, the recipient can use this transfer certificate and broadcast it to all authorities and bring them all to the complete/final state for this transfer order. If the recipient did not receive a transfer certificate but some authorities did, then the recipient can use the same request function as before and request the transfer certificate from some authorities who completed the transfer (since the authorities store the completed transfer certificates in the *confirmed* list) with the information of the original sender. The authorities would skip the if statement in the pseudo code 2 on line 104 and proceed further to look up the sequence number and sender name in their confirmed list (line 108). Once the requester receives back such a transfer certificate, he can then send it to all authorities again and bring them all to the complete/final state for this transfer order. The client can skip the part about collecting a quorum of signatures this way, since the certificate was already formed and accepted by an authority. For a Byzantine authority to fake such a certificate is nearly impossible, so this scenario would not be a big threat and just in case, the requester checks anyway if the original sender signature matches the expected one from the received information.

It should be mentioned that it is necessary that the protocol has to be finished once the authorities have received a valid transfer order (at least for the correct clients), or else a certain client would be permanently in a blocked state since the authorities do not allow a new transfer order into their pending state. But this is also not bad, since it is desired not to have any Byzantine processes (or even Byzantine authorities) in the system to be able to do any more transfers (once a traitor, forever a traitor).

4.2 Proofs

In order for FastPay to work properly and not run into any problems, it is necessary that it fulfils some properties, or more precisely, that the involved parties hold such properties. The FastPay client has two properties, which provide the guarantee of the protocol initialization of a transfer order and initialization for completion.

Theorem 2. *The pseudo code 1 fulfils the property FPC1.*

Proof. If a correct client broadcasts a transfer order via Hybrid-Consistent-Broadcast (inheriting all their properties) to *at least one correct* authority, then due to the nature of the protocol, every authority will at some point receive this transfer order. If the sender does not crash, then eventually every authority will receive the broadcast (HCB1). If the sender crashes but at least one correct authority receives the transfer

order, then any other client can request this information from that authority and proceed in the protocol for that certain user who was not able to complete the transfer. This means every authority will at some point receive the transfer order m. Now let's assume that the client is not correct and broadcasts a faulty transfer order, then no correct authority would accept it (which is a correct behaviour and results in no problems). If the client is not correct and also an authority, which accepts the faulty transfer order, then this would also not cause any further problems, since the faulty transfer order would need to be accepted by $N - f$ authorities which is not possible due to Byzantine fault tolerance. □

Theorem 3. *The pseudo code 1 fulfils the property FPC2.*

Proof. After the client (either the original sender or another process who finishes this transfer) has gathered enough signed transfer orders from authorities (reaching a quorum of signatures), it can then form a transfer certificate. Once broadcasted to at least one correct authority, eventually every authority will receive such a transfer certificate, i.e., eventually execute that transfer order. Again, this is because even when the client crashes during broadcasting the message but manages to broadcast it to at least one correct authority, every other client can finish the protocol by requesting that transfer certificate from that correct authority (in case the sender has crashed). If the client is not correct, meaning it sent a faulty transfer certificate, no correct authority would accept it, resulting in no problems. Assuming the client is not correct and also a Byzantine authority accepts this faulty transfer certificate (or the Byzantine authority creates such a transfer certificate by themselves), it would not possess further problems even when a correct client requests this information in order to finish the transfer order, since correct authorities would not accept this transfer certificate due to various reasons, like not matching with their pending transfer order, not having a quorum of signatures, or they do not even have a pending transfer order. □

Corollary 1. *The pseudo code 1 for the FastPay Client implements module 4.*

The authorities are responsible for ensuring that (valid) transfer orders get eventually executed and finished, and they are also responsible for preventing illegal actions from happening, like double spending or accepting faulty transfer orders.

Theorem 4. *The pseudo code 2 fulfils the property FPA1.*

Proof. The client p broadcasts the transfer order m using Hybrid-Consistent- Broadcast, meaning that thanks to the property HCB1, eventually every authority will receive m. After an authority has done its task, it sends it back to the process via the AuthenticatedPerfectLinks, which has the property AL1, so the client p will eventually receive the signed m from the authority. Once the client p has eventually received at least $N - f$ such signed m's from the authorities, it then creates a transfer certificate and broadcasts it to the authorities again using Hybrid-Consistent-Broadcast, which means every authority will eventually receive the transfer certificate because of the validity property. Once an authority eventually receives such a transfer certificate, it does its task and records the transfer order m as completed. □

Theorem 5. *The pseudo code 2 fulfils the property FPA2.*

Proof. This is a proof by contradiction. Assume the sender has sent two conflicting transfer orders and somehow still collected a quorum of signatures for both conflicting transfer orders. This means he can double spend. This would mean that two different quorums of $N - f$ authorities have signed that the transfer orders are valid. Since two Byzantine quorums will always overlap in at least one correct process (i.e., authority) (see 2.2), implies that a correct process (i.e., authority) has signed both transactions as valid. This results in a contradiction since a **correct** process (i.e., authority) could not do such a thing as sign two transfer orders for the same transfer index/pending transfer order. □

Corollary 2. *The pseudo code 2 for the authorities implements module 5.*

The smart contract, i.e., the Primary is the connection between the initial currency (stored as information on the blockchain) and FastPay as a side-chain. Thus, it is important that the transfer does not violate any properties, like FPS1 and FPS2.

Theorem 6. *The pseudo code 3 fulfils the property FPS1.*

Proof. This proof is straight-forward. Once the smart contract receives a correct funding transaction and has changed the balance of the sender, it sends a synchronization order to the authorities to inform them of the change of state. The authorities will then update the balance of the recipient account on their side (i.e., on the FastPay account), and if there is no such account, immediately create one and transfer the funds to it. This results in that the currency will never be lost, it will always land on a FastPay account. □

Theorem 7. *The pseudo code 3 fulfils the property FPS2.*

Proof. This follows immediately from the pseudo codes. When a transfer certificate is received by the smart contract, it means that the authorities will also eventually receive the transfer certificate. When the smart contract validates the transfer certificate as valid, so will the authorities, which means the sender's balance has (eventually) already been adjusted on the FastPay account side, which leaves the smart contract to add the amount to the Primary balance, i.e., no currency was created out of thin air. □

Corollary 3. *The pseudo code 3 for the smart contract implements module 6.*

Chapter 5

Conclusion

FastPay shines in its performance, thanks to the results of Guerraoui et al. [12], that it is not mandatory to use full consensus in order to prevent attacks like double spending. With its committee-based system, FastPay provides high security, and thanks to the Byzantine consistent broadcast, it allows for faster communications compared to consensus, resulting for FastPay to have an overall low latency. Also, due to the committee-focused approach, it opens up for easy scalability and matching the needed throughput, compared to the permissioned-based blockchain, which is limited due to its complexity and lack of scalability (due to the full Byzantine consensus protocol). FastPay also tolerates up to one third of authorities to crash or be Byzantine without losing safety or performance. Thanks to the way the protocol is built up, it even allows the sender to crash at certain points, and the transfer will still be executed without any problems of correctness or being tampered with, as it has been shown. Modelling FastPay in pseudo code has shown, depending on how one wants to model it, that it can bring up some difficulties or even lead to efficiency loss. Hence, the practical solution of cross-trigger was introduced in order to make it easier to understand the pseudo code, which is modelled in two processes (clients and authorities), and to make it less code-heavy.

The idea behind FastPay, i.e., not needing to use consensus and using a committee-based approach, motivates moving away from classical centralized solutions like those common banks use or the use of other slower protocols. Since FastPay can be used as an ad-hoc (i.e. as a side-chain), it does not need an overall reconstruction of a fundamental system, which allows for easy access and usage, and it allows for a transition time to convert a whole system to it without impacting its current protocol in use.

Overall, FastPay fits perfectly in the environment of the modern world, where technology is always evolving and the need for better and faster usability and scalability grows. It took something that was meant to be the standard and improved it, making it faster, more scalable and less complex. There exists, in fact, an already improved version of FastPay made by the same group called *Zef*, a low-latency, scalable, private payment protocol (c.f. Baudet et al. [5]). Zef extends FastPay with digital coins that are both opaque and unlinkable i.e. anonymous. This may be another interesting topic to look at in more detail, especially how it evolved from the base of FastPay, i.e., what changed and if they made any improvements to the base of FastPay.

Bibliography

[1] Z. Avarikioti, E. Kokoris-Kogias, R. Wattenhofer, and D. Zindros, "Brick: Asynchronous incentive-compatible payment channels," in *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II* (N. Borisov and C. Díaz, eds.), vol. 12675 of *Lecture Notes in Computer Science*, pp. 209–230, Springer, 2021.

[2] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Sok: Consensus in the age of blockchains," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*, pp. 183–198, ACM, 2019.

[3] M. Baudet, G. Danezis, and A. Sonnino, "Fastpay: High-performance byzantine fault tolerant settlement," in *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*, pp. 163–177, ACM, 2020.

[4] M. Baudet, F. Garillot, and M. Kelkar, "Fastpay." `https://github.com/novifinancial/fastpay`, 2021. Commit used: db39bb0.

[5] M. Baudet, A. Sonnino, M. Kelkar, and G. Danezis, "Zef: Low-latency, scalable, private payments," in *Proceedings of the 22nd Workshop on Privacy in the Electronic Society, WPES 2023, Copenhagen, Denmark, 26 November 2023* (B. P. Knijnenburg and P. Papadimitratos, eds.), pp. 1–16, ACM, 2023.

[6] M. L. Bech and B. Hobijn, "Technology Diffusion within Central Banking: The Case of Real-Time Gross Settlement," *International Journal of Central Banking*, vol. 3, pp. 147–181, September 2007.

[7] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[8] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[9] D. Collins, R. Guerraoui, J. Komatovic, P. Kuznetsov, M. Monti, M. Pavlovic, Y. Pignolet, D. Seredinschi, A. Tonkikh, and A. Xygkis, "Online payments by merely broadcasting messages," in *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pp. 26–38, IEEE, 2020.

[10] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer, "On scaling decentralized blockchains - (A position paper)," in *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers* (J. Clark, S. Meiklejohn, P. Y. A. Ryan, D. S. Wallach, M. Brenner, and K. Rohloff, eds.), vol. 9604 of *Lecture Notes in Computer Science*, pp. 106–125, Springer, 2016.

[11] G. Danezis and S. Meiklejohn, "Centrally banked cryptocurrencies," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, The Internet Society, 2016.

[12] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi, "The consensus number of a cryptocurrency," *Distributed Comput.*, vol. 35, no. 1, pp. 1–15, 2022.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Strengelbach 13.12.23
Ort/Datum

N. Maggio
Unterschrift