On Payment Channels in Asynchronous Money Transfer Systems

ODED NAOR, Technion, Israel
IDIT KEIDAR, Technion, Israel

Money transfer is an abstraction that realizes the core of cryptocurrencies. It has been shown that, contrary to common belief, money transfer in the presence of Byzantine faults can be implemented in asynchronous networks and does not require consensus. Nonetheless, existing implementations of money transfer still require a quadratic message complexity per payment, making attempts to scale hard. In common blockchains, such as Bitcoin and Ethereum, this cost is mitigated by payment channels implemented as a second layer on top of the blockchain allowing to make many off-chain payments between two users who share a channel. Such channels only require on-chain transactions for channel opening and closing, while the intermediate payments are done off-chain with constant message complexity. But payment channels in-use today require synchrony, therefore they are inadequate for asynchronous money transfer systems.

In this paper, we provide a series of possibility and impossibility results for payment channels in asynchronous money transfer systems. We first prove a quadratic lower bound on the message complexity of on-chain transfers. Then, we explore two types of payment channels, unidirectional and bidirectional. We define them as shared memory abstractions and prove that in certain cases they can be implemented as a second layer on top of an asynchronous money transfer system whereas in other cases it is impossible.

1 INTRODUCTION

The rise of cryptocurrencies, such as Bitcoin [32], Ethereum [41], Ripple [2], Tendermint [9], and many more, has revolutionized the possibility of using decentralized money systems.

In 2019, Guerraoui et al. [20] defined the abstraction of asset transfer or money transfer capturing the original motivation of Bitcoin. This abstraction is based on a set of known users owning accounts, each account has some initial money, and the users can transfer money between the accounts. It is well-known that deterministic consensus cannot be solved in an asynchronous network [18], meaning that blockchains that rely on consensus require synchrony to work properly. Nonetheless, Guerraoui et al. showed that the asset transfer problem is weaker than consensus [15, 27], i.e., in the Byzantine message-passing model the problem can be solved in an asynchronous network. They provide a concrete implementation of the abstraction in this model using an asynchronous broadcast service.

Yet, in this solution, each payment requires a message complexity of $O(n^2)$, where $n$ is the number of processes in the system. If the number of processes grows, this per-payment quadratic message complexity can pose a real challenge in scaling the asset transfer network. In fact, scalability is one of the major limiting factors of blockchains and consensus protocols, and extensive research was done to reduce the message complexity [11, 17, 26, 33, 42] in various settings.

A promising approach to scale blockchain payments is by using payment channels [28, 35, 37] as a second off-chain layer on top of the blockchain. A payment channel can be opened between two blockchain account owners via an on-chain deposit made to fund the channel, after which the two users can transfer payments on the channel itself off the blockchain. At any time, one of the users can decide to close the channel, after which the current balance in the channel of each of the users is transferred to their on-chain accounts. In this scheme, opening or closing a channel requires a blockchain transaction, which incurs a large message complexity, but all the intermediate payments on the channel require message exchange only between the channel users, freeing the blockchain from these payments and messages.
Payment channels have been actively deployed in central blockchains. For example, Bitcoin’s Lightning Network (LN) [35] is a highly used payment network, with active channels holding a Bitcoin amount equivalent to hundreds of millions of dollars [39].

One of the downsides of using payment channels such as LN is that the payment channels themselves rely on network synchrony. For example, in LN, suppose Alice and Bob have an open payment channel between them and Alice acts maliciously and tries to steal money by closing the channel at a stale state. LN provides a way for Bob to penalize Alice and confiscate all the money in the channel. But to do so, Bob has to detect Alice’s misbehavior on-chain and act within a predetermined time frame, making this method inappropriate with asynchronous users.

In this paper, we explore the possibility of implementing payment channels in asynchronous asset transfer systems. The results of the paper and prior art are summarized in Table 1. We study four abstractions: asset transfer and different types of payment channels.

First, we prove a lower bound on the complexity of asset transfer without channels. We show that no matter what implementation is provided for the asset transfer abstraction, it still requires $\Omega(n^2)$ messages for a single payment to be made by one party and observed by others. This means that the upper bound is tight for the algorithm provided in [20], in which transferring money has $O(n^2)$ message complexity and reading the balance of an account costs $O(1)$. This fundamental result shows that while the synchrony requirement can be relaxed for asset transfer, each payment still requires a rather large number of messages. This means that second-layer solutions such as payment channels are required for scalability.

Next, we consider payment channel abstractions with operations for opening a channel, transferring money in it, and closing the channel. We first consider a bidirectional payment channel as a second-layer atop an asset transfer system, where each side of the channel can make payments to the other. This is similar to LN [35], Teechain [28], Sprites [30], and more payment channel proposals. We prove an impossibility result, showing that synchrony is required to create such bidirectional channels, and therefore a bidirectional channel cannot be implemented on top of an asynchronous asset transfer system.

We next look at unidirectional payment channels, in which only one user, the channel source, can make payments to the other user, the target. We differentiate between two types of unidirectional payment channels: If we allow both the source and the target to close the channel (the source and target close operations, respectively), we again show that synchrony is required. But, if we allow only the target to close the channel, we provide a concrete implementation that
works in an asynchronous network. In this implementation, the opening and closure of the channel require a payment using the asset transfer system, incurring $O(n^2)$ message complexity, whereas every payment on the channel itself requires a single message from the source user to the target user.

Finally, we discuss an extension of payment channels to payment chains, in which payments are made across multiple channels atomically. Like their 2-party counterparts, a chain payment over unidirectional channels with only target close can be implemented using a technique used in LN. As for other channel types, $k$-hop chains are equivalent to $k$-process consensus, i.e., have a consensus number $k$.

To conclude, in this paper we continue the study of asset transfer systems in asynchronous networks and provide concrete possibility and impossibility results for implementing a key scaling solution.

Structure. The rest of the paper is structured as follows: §2 discusses related work; §3 describes the model; §4 details the asset transfer abstraction and proves a lower bound on message complexity; §5 discusses bidirectional payment channels and §6 discusses unidirectional channels; §7 extends the discussion to chain payments; and finally; §8 concludes the paper.

2 RELATED WORK

A few works, some of which predate the blockchain era, identified that a transfer-like system can be implemented under asynchrony [22, 34]. The first definition of the asset transfer abstraction is due to Guerraoui et al. [20]. Subsequently, Auvolat et al. [4] provide a weaker specification for asset transfer and detailed an implementation that uses a broadcast service that guarantees FIFO order between every two processes. Astro [12] implements and empirically evaluates an asynchronous payment system.

Solving Byzantine consensus in an asynchronous network is possible by using randomization to circumvent the FLP [18] result. Earlier protocols such as [6, 36] have exponential message complexity. Later protocols such as [1, 21, 26, 31] improve the message complexity in various settings, but they still are not deterministic, and therefore their performance can only be measured in the expected case.

There is also extensive research done on scaling cryptocurrencies using payment channels, including the Lightning Network [35], Teechain [28], Bolt [19], Sprites [30], Perun [16], Duplex Micropayment Channels [13], Raiden [37], and more. In Ethereum [41], there are multiple second-layer networks like Arbitrum [25], StarkNet [7], and Optimism [40]. They all rely on underlying synchronous blockchains to operate.

To the best of our knowledge, we are the first to show a quadratic lower bound for payments in asset transfer systems, as well as to explore second-layer payment channels as a scaling solution for money transfer in the asynchronous setting.

3 MODEL AND PRELIMINARIES

We study a message-passing distributed system that consists of a set $\Pi = \{p_1, \ldots, p_n\}$ of $n$ processes. The processes can interact among themselves by sending messages. An adversary can corrupt up to $f < n/3$ processes, where $f \in \Theta(n)$.

If not mentioned explicitly, we assume a corrupt process is Byzantine, i.e., it can deviate from the prescribed algorithm and act arbitrarily. Any non-corrupt process is correct. A crash-fail fault is when a process stops participating in the algorithm. Every two processes share an asynchronous reliable link between them, such that if one correct process sends a message to another correct process, it eventually arrives, and the target can ascertain its source.
We assume the existence of a Public Key Infrastructure (PKI), whereby processes that know a private key can use it to sign messages such that all other processes can verify the signature. The adversary cannot forge a signature if the private key is owned by a correct process. We assume each private key is owned by one process. We further assume multisignatures [5], whereby in order to produce a valid signature matching a constant-sized public key, more than one private key is used to sign the message. Note that signing can be sequential.

We study algorithms in the message-passing model that implement abstractions that are defined as shared-memory objects. A shared memory object has a set of operations, and processes access the object via these operations. Each operation starts with an invocation event by a process and ends with a subsequent response event. Invocations and responses are discrete events.

An implementation or an algorithm \( \pi \) of a shared-memory object abstraction is a distributed protocol that defines the behaviors of processes as deterministic state machines, where state transitions are associated with actions: sending or receiving messages, and operation invocations or responses. A global state of the system is a mapping to states from systems components, i.e., processes and links. An initial global state is when all processes are in initial states and there are no messages on the links between the processes. A run, or an execution of an implementation is an alternating series of global states and actions, beginning with some initial global state, such that state transitions occur according to \( \pi \).

We assume that the first action of each process is an invocation of an operation and that it does not invoke another operation before receiving a response for its last invoked operation.

Each execution creates a history \( H \) that consists of a sequence of matching invocations and responses, each with the assigned process that invoked the operation and the matching responses. A history defines a partial ordering: operation \( \text{op}_1 \) precedes \( \text{op}_2 \) in history \( H \), labeled \( \text{op}_1 \prec_H \text{op}_2 \), if \( \text{op}_1 \)'s response event happens before \( \text{op}_2 \)'s invocation event in \( H \). History \( H \) is sequential if each invocation, except perhaps the last, is immediately followed by a matching response. An operation is pending in history \( H \) if it has an invocation event in \( H \) but does not have a matching response. A history \( H' \) is a completion of history \( H \) if it is identical to \( H \) except removing zero or more pending operations in \( H \) and by adding matching responses for the remaining ones. A shared-memory abstraction is usually defined in terms of a sequential specification. A legal sequential history is a sequential history that preserves the sequential specification.

The correctness criteria we consider is Byzantine sequential consistency (BSC). This allows to extend sequential consistency [3] to runs with Byzantine processes and not only crash-fail, and is an adaptation of the definition of Byzantine linearizability [10].

First, we formally define a sequentially consistent history for runs with crash-fail faults:

Definition 3.1 (sequentially consistent history). Let \( E \) be an execution of an algorithm and \( H \) its matching history. Then \( H \) is sequentially consistent if there exists a completion \( \tilde{H} \) of \( H \) and a legal sequential history \( S \) such that: (i) for every process \( p \), \( S_{\mid p} = \tilde{H}_{\mid p} \), and (ii) for any process \( p \) and operations \( \text{op}_1, \text{op}_2 \) s.t. \( \text{op}_1 \prec_{\tilde{H}} \text{op}_2 \), then \( \text{op}_1 \prec_S \text{op}_2 \).

An algorithm is sequentially consistent if all its histories are sequentially consistent.

For Byzantine sequential consistency (BSC), let \( H_{\mid c} \) denote the sub-history of \( H \) with all of the operations of correct processes. We say a history is BSC if \( H_{\mid c} \) can be augmented with operations of Byzantine processes such that the completed history is sequentially consistent. Formally:

Definition 3.2 (Byzantine sequential consistency (BSC)). A history \( H \) is BSC if there exists a history \( H' \) such that \( H'_{\mid c} = H_{\mid c} \), and \( H' \) is sequentially consistent.
Similar to sequential consistency, an algorithm is BSC if all its histories are BSC. We choose BSC as the correctness criteria and not Byzantine linearizability as it simplifies the implementations we provide below. We explain in §6.3.2 how to change the implementation we provide to satisfy Byzantine linearizability. The difference of sequential consistency from linearizability [23] is that linearizability also preserves real-time order.

4 ASSET TRANSFER

4.1 Asset transfer abstraction

Let $A$ be an asset transfer abstraction, which is based on the one defined in [20]. $A$ holds a set of accounts. Each account $a \in A$ is defined by some public key, and there is a mapping $\text{owner}(a) : A \mapsto \Pi$, that matches for each account $a$ the process that can produce a signature corresponding to the public key associated with $a$. In case of an account $b$ associated with a multisignature, $\text{owner}(b)$ is the set of processes whose private keys can produce a signature matching $b$’s public key. The state of each account $a$ is of the form $A(a) \in \mathbb{R}_{\geq 0}$ and represents the balance of the account $a$. Each account initially holds its initial balance.

$A$ has two operations: The first, $A.\text{read}(a)$, returns the balance of account $a$, i.e., it returns $A(a)$ and can be called by any process. The second is $A.\text{transfer}(a, [(b_1, \text{amt}_1), \ldots, (b_k, \text{amt}_k)])$, which, for every $1 \leq i \leq k$, transfers from account $a$’s balance $\text{amt}_i$ and deposits it in $b_i$. This call succeeds, and returns success if it is called by $\text{owner}(a)$ and if the account has enough balance to make the transfer, i.e., $A(a) \geq \sum_{i=1}^{k} \text{amt}_i$. Otherwise, it returns fail and does nothing.

The $\text{transfer}$ operation is an extension of the one defined in [20] in that it allows to transfer money from one account to multiple accounts, whereas the original work only allows transferring money from one account to another each time.

In this paper, we consider implementations of the asset transfer abstraction that are Byzantine sequentially consistent. In particular, this means that for every transfer operation, there exists a time $t$ such that if a correct process invokes the read operation after $t$, then it observes the changes made by the transfer. We note that in [20], the correctness criteria provided for the asset transfer implementation in the message-passing model is stronger than BSC and weaker than Byzantine linearizability. Informally, successful transfers by correct processes preserve real-time order, but reads can see a “stale” state of an account.

4.2 Message complexity of asset transfer

4.2.1 Lower bound. The asset transfer implementation we discuss in §4.2.2 below has a message complexity of $O(n^2)$ for the $\text{transfer}$ operation, and $O(1)$ message complexity for the $\text{read}$ operation.

We show that if $f \in \Theta(n)$ then this is also the lower bound in runs in which money is transferred to some account $b$, and multiple processes read the balance of $b$. The proof follows the technique used in Dolev-Reischuk’s lower bound for Byzantine Broadcast [14]. Formally we prove the following theorem:

**Theorem 4.1.** Consider an algorithm that implements the asset transfer abstraction. Then there exists a run with a single transfer invocation and multiple read invocations in which correct processes send at least $(f/2)^2$ messages.

**Proof.** Let $\pi$ be an algorithm that implements asset transfer, and assume by contradiction that in all its runs with a single transfer correct processes send less than $(f/2)^2$ messages. We look at all executions of $\pi$ with two accounts $a, b \in A$ s.t. $\text{owner}(a) = p$ for some process $p \in \Pi$, and that initially $A(a) = 1, A(b) = 0$.

Consider first an execution $\sigma_0$ in which the adversary, denoted $\text{adv}_0$, corrupts a set $V$ of processes, not including $p$, such that $|V| = \lceil f/2 \rceil$. Denote the set of remaining correct processes as $U$. In $\sigma_0$, process $p$ calls $A.\text{transfer}(a, [(b, 1)])$. 


By the correctness definition of $A$, and since $p$ is correct, there exists a time $t_0$ during the run after which any correct process that invokes $A.read(b)$ returns 1.

The adversary $adv_0$ causes the corrupt processes in $V$ to simulate the behavior of correct processes that call $A.read(b)$ after $t_0$, and follow the algorithm except for the following changes: they ignore the first $f/2$ messages they receive from processes in $U$, and they do not send any message to other processes in $V$. Note that while $t_0$ is not known to the processes, we construct the runs from the perspective of a global observer and may invoke read after $t_0$.

Because correct processes send, in total, at most $(f/2)^2$ messages and corrupt processes do not send messages to other processes in $V$, then the processes in $V$ together receive at most $(f/2)^2$ messages. Thus, by the pigeonhole principle, there exists at least one process $q \in V$ that receives at most $f/2$ messages. Denote the set of processes that send $q$ messages as $U'$, and denote $U'' = U \setminus U'$. Note that $U'$ may include process $p$, and that $|U'| < f/2$.

Next, we construct a run $\sigma_1$ with an adversary $adv_1$ that are the same as $\sigma_0$ and $adv_0$, respectively, except for the following changes: $adv_1$ corrupts all the processes in $V \setminus \{q\}$, and all the processes in $U'$. Since $|U'| < f/2$ and $|V| \leq \lceil f/2 \rceil$, the adversary $adv_1$ corrupts at most $f$ processes in $\sigma_1$. $adv_1$ prevents the corrupt processes from sending any message to $q$, but causes them to behave correctly to all other correct processes in $U''$.

By definition, the behavior of the corrupt processes in $\sigma_1$, i.e., the processes in $U' \cup (V \setminus \{q\})$, towards the correct processes in $U''$ is the same as in $\sigma_0$. Since process $q$ simulates a correct process that ignores the first $f/2$ messages in $\sigma_0$, its behavior towards the processes in $U$ is identical in both runs as well. Thus, runs $\sigma_0$ and $\sigma_1$ are indistinguishable for the correct processes in $U''$, ensuring that they behave the same. Since process $q$ acts in $\sigma_0$ like a correct process that does not receive any message, both runs are indistinguishable to it as well.

Nonetheless, process $q$ still has to return a value for its $A.read(b)$ call. Denote the time when the call returns as $t_1$. If it returns a value different than 1, then we conclude the proof, since it is a violation of the read call specification. Otherwise, we construct a run $\sigma_2$ with an adversary $adv_2$ that are the same as $\sigma_1$ and $adv_1$, respectively, except that there is no transfer invocation and all messages to $q$ are delayed until after $t_1$. For process $q$, runs $\sigma_1$ and $\sigma_2$ are indistinguishable until $t_1$, therefore it returns 1 for the $A.read(b)$ call, violating the read call specification which should return 0, concluding the proof.

\[\square\]

We proved that the lower bound for the message complexity of an asset transfer object is $\Omega(n^2)$, assuming $f \in \Theta(n)$.

### 4.2.2 Upper bound.
In [20], an implementation in the message-passing model for the asset transfer abstraction is provided. It uses a broadcast service defined in [29] that tolerates up to $f < n/3$ Byzantine failures. This broadcast has all the guarantees of reliable broadcast [8] (integrity, agreement, and validity), and also preserves source order, i.e., any two correct processes $p_1$ and $p_2$ that deliver messages $m$ and $m'$ broadcast from the same process $p_3$, do so in the same order. The read operation is locally computed, and the transfer operation consists of a broadcast of a single message.

There are several protocols that can be used to implement such source-order broadcast service, including a protocol in [29]. Bracha’s reliable broadcast [8] can also be used to implement such a service if each correct process adds a sequence number to each message it broadcasts, and each correct process delivers messages from the same process in the order of the sequence numbers. These protocols have a message complexity of $O(n^2)$ per broadcast, proving that the lower bound message complexity we prove above is tight.

Note that our definition for the transfer call of the asset transfer abstraction allows transferring in each invocation money from one account to multiple accounts, while in [20] the transfer call allows a transfer to a single account for each invocation. The implementation in [20] can easily be adjusted to support this change by including in each broadcast message the multiple accounts to which money is transferred.
We define a bidirectional payment channel abstraction as a shared memory object. We seek to analyze if payment channels that are used in common blockchains as a second layer can also be used in a closed form. Specification 1.

5 BIDIRECTIONAL PAYMENT CHANNEL

We seek to analyze if payment channels that are used in common blockchains as a second layer can also be used in a similar manner on top of an asynchronous asset transfer system. We discuss bidirectional payment channels, in which a channel is opened between two processes by making a transfer on the asset transfer system. After the channel is opened, both processes can make bidirectional payments on the same channel. Either process can close the channel at any time, after which their accounts in the asset transfer system reflect the state of the channel. This abstraction is similar to payment channels in the Lightning Network [35] in Bitcoin [32] and Raiden [37] in Ethereum [41].

First, we formally define this abstraction, and then provide an impossibility result, proving it cannot be implemented in asynchronous networks.

5.1 Definition

We define a bidirectional payment channel abstraction as a shared memory object $BC$. The formal definition is in Specification 1. $BC$ is defined based on the existence of an asset transfer object $A$. A payment channel in $BC$ is of the form $(a, b)$, where $a, b$ are accounts in $A$.

The state of a payment channel $BC(a, b)$ is $\{\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}\} \cup \{\bot\}$. The channel $BC(a, b)$ can be open, and then $BC(a, b) = (bal_a, bal_b)$, which represents the balances $bal_a, bal_b$ of accounts $a, b$ in the channel $(a, b)$, respectively. If the channel is closed, then its state is $BC(a, b) = \bot$.

The set of operations is the following:

### Specification 1: Bidirectional payment channel abstraction. Operations for process $p$.

| Line | Description |
|------|-------------|
| 1    | Procedure $BC.transfer((a, b), amt)$: |
| 2    | if $BC(a, b) = \bot$ then |
| 3    | return |
| 4    | $(bal_a, bal_b) \leftarrow BC(a, b)$ |
| 5    | if $owner(a) = p \land bal_a \geq amt$ then |
| 6    | execute_payment($a, b, -amt, amt$) |
| 7    | if $owner(b) = p$ and $bal_b \geq amt$ then |
| 8    | execute_payment($a, b, amt, -amt$) |
| 9    | Procedure $BC.close((a, b), bal)$: |
| 10   | if $R(a, b) = \bot$ then |
| 11   | return fail |
| 12   | $(curr_bal_a, curr_bal_b) \leftarrow BC(a, b)$ |
| 13   | other_bal = curr_bal_a + curr_bal_b |
| 14   | if $owner(a) = p$ then |
| 15   | if $bal > curr_bal_a$ then |
| 16   | return fail |
| 17   | return execute_close($a, b$, bal, other_bal) |
| 18   | if $owner(b) = p$ then |
| 19   | if $bal > curr_bal_b$ then |
| 20   | return fail |
| 21   | return execute_close($a, b$, other_bal, bal) |
| 22   | return fail |
| 23   | Function $execute_payment((a, b), amt_a, amt_b)$: |
| 24   | $(bal_a, bal_b) \leftarrow BC(a, b)$ |
| 25   | new_bal_a $\leftarrow bal_a + amt_a$ |
| 26   | new_bal_b $\leftarrow bal_b - amt_b$ |
| 27   | $BC(a, b) \leftarrow (new_bal_a, new_bal_b)$ |
| 28   | Function $execute_close((a, b), amt_a, amt_b)$: |
| 29   | $A(a) \leftarrow A(a) + amt_a$ |
| 30   | $A(b) \leftarrow A(b) + amt_b$ |
| 31   | $BC(a, b) \leftarrow \bot$ |
| 32   | return success |
We show that implementing a sequentially consistent bidirectional payment channel in the message-passing model.

The returned value for any process making the call has to be an input of the call from one of the processes, and it is an impossibility result and crash-fail faults are weaker than Byzantine faults, it also applies to runs with Byzantine processes as well.

5.2 Impossibility result of a bidirectional payment channel object

We show that implementing a sequentially consistent bidirectional payment channel in the message-passing model requires synchrony. To this end, we solve wait-free consensus among 2 processes with shared registers and an instance of BC. The consensus abstraction has one call, propose(v), which is called with some proposal v, and returns a value. The returned value for any process making the call has to be an input of the call from one of the processes, and it has to be the same value for all invocations, regardless of the caller process. We assume in this proof crash-fail faults, i.e., a process corrupted by the adversary stops participating in the protocol, but does not deviate from it. Since this is an impossibility result and crash-fail faults are weaker than Byzantine faults, it also applies to runs with Byzantine processes as well.

**Lemma 5.1.** Consensus has a wait-free implementation for 2 processes in the read-write shared memory model with an instance of a bidirectional payment channel shared-memory object and shared registers.
The algorithm for solving consensus among two processes using a bidirectional payment channel object is detailed in Alg. 2. We assume that there are two processes $p_1, p_2$ with ownership of accounts $a, b$, respectively, and an open payment channel $(a, b)$ at the beginning of the run with balances $BC(a, b) = (1, 1)$.

Before either of the processes performs an operation on the payment channel, they write their proposal $o$ in a shared register (Lines 2 and 7). Then, $p_1$ attempts to make a payment on the channel and then close it, and $p_2$ tries to close the channel without making or accepting any payment.

Because $BC$ is sequentially consistent, the algorithm ensures that eventually after the channel is closed either the payment from $a$ to $b$ on the channel succeeds or not, and the balance in $b$’s account reflects it, i.e., there exists a time $t$ after which invoking $B.read(b)$ returns either 1 or 2.

If the read call in Line 11 returns 2, then the channel was closed by $p_1$ after it successfully made the payment on the channel. Since before $p_1$ makes the payment on the channel it writes its proposal to register $R_1$, then its value is already written by the time the channel is closed, and it is returned by the propose call. If the return value of the read call is 1, then process $p_2$ closed the channel. Since $p_2$ closes the channel after it writes its proposal in $R_2$, then its proposed value is returned.

In either case, when the channel is closed, there is already a proposal written in either $R_1$ or $R_2$, i.e., the returned value is an input to the propose call by either process, and both processes return the same value. □

Based on the above theorem and FLP [18], we get the following result:

**Theorem 5.1.** There does not exist an implementation of the bidirectional payment channel abstraction in the asynchronous message-passing model.

### 6 UNIDIRECTIONAL PAYMENT CHANNEL

After proving that a bidirectional payment channel cannot be implemented in asynchronous networks, we explore another type of payment channels, unidirectional. The main difference from bidirectional channels is that unidirectional channels are asymmetric. There is only one user, the source, that can open and transfer money in the channel. We show in which cases unidirectional payment channels can be implemented in an asynchronous message-passing network and in which cases they cannot. This type of payment channel was also proposed for the Bitcoin network [38] a few years before bidirectional channels such as LN were proposed. We begin by formally defining the unidirectional payment channel abstraction.

#### 6.1 Definition

We define a unidirectional payment channel abstraction as a shared memory object $B$. The formal definition is in Specification 3. $B$ is defined based on the existence of an asset transfer object $A$. A payment channel in $B$ is of the form $(a, b)$ where $a, b$ are accounts in $A$.

The state of a payment channel $B(a, b)$ is $\{\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}\} \cup \{\bot\}$. Intuitively, a unidirectional payment channel $B(a, b)$ can either be open, and then $B(a, b) = (bal_a, bal_b)$, or closed, and then $B(a, b) = \bot$. The initial state is that all the unidirectional payment channels are closed, e.g., for payment channel $(a, b)$, the state is $B(a, b) = \bot$ at the beginning of the run.

The set of operations is the following:
Specification 3: Unidirectional payment channel abstraction. Operations for process $p$.

Shared Objects:
- $A$: asset transfer object, with initial accounts
- $B$: unidirectional payment channel object

Procedure $B$.open($a$, $b$, $amt$):
1. if $p \notin \text{owner}(a) \lor B(a, b) = \bot \lor A(a) < amt$ then return fail
2. $A(a) \leftarrow A(a) − amt$
3. $B(a, b) \leftarrow (amt, 0)$
4. return success

Procedure $B$.transfer($a$, $b$, $amt$):
1. if $p \notin \text{owner}(a) \lor B(a, b) = \bot$ then return fail
2. $(\text{new}_a, \text{new}_b) \leftarrow B(a, b)$
3. if $\text{new}_a < \text{amt}$ then return fail
4. $\text{new}_a \leftarrow \text{bal}_a − \text{amt}$
5. $\text{new}_b \leftarrow \text{bal}_b + \text{amt}$
6. $B(a, b) \leftarrow (\text{new}_a, \text{new}_b)$
7. return success

Procedure $B$.target_close($a$, $b$, $bal_a$):
1. if $p \notin \text{owner}(b) \lor B(a, b) = \bot$ then return fail
2. $(\text{curr}_a, \text{curr}_b) \leftarrow B(a, b)$
3. if $\text{bal}_b > \text{curr}_b$ then return fail
4. $\text{bal}_b \leftarrow \text{curr}_b + \text{bal}_b − \text{bal}_a$
5. return execute_close($a$, $b$, $\text{bal}_a$)

Function execute_close($a$, $b$, $\text{bal}_a$):
1. $A(a) \leftarrow A(a) + \text{bal}_a$
2. $A(b) \leftarrow A(b) + \text{bal}_b$
3. $B(a, b) \leftarrow \bot$
4. return success

- open($a$, $b$, $amt$). A process that owns account $a$ can open a unidirectional payment channel with any other account $b$ with amount $amt$, as long as it has enough balance in $A$ and that it does not already have an open payment channel with $b$. The call returns success if the channel is opened successfully, and fail otherwise.
- transfer($a$, $b$, $amt$). A payment in the payment channel ($a$, $b$) is possible if the channel is open, if the caller of the operation $a$ is owner($a$) and $a$ has enough balance in the channel to make the payment. This call does not return a response.
- source_close($a$, $b$, $bal_a$). A source closing of a payment channel ($a$, $b$) can be called by owner($a$). The call succeeds if the process that invokes the call does not try to close it with balance $bal_a$ that exceeds the amount it has in the channel. After the call ends, the balances in the channel are transferred to accounts $a$, $b$ in the asset transfer system. The call returns success if the channel is closed successfully, and fail otherwise.
- target_close($a$, $b$, $bal_b$). A target closure of a payment channel ($a$, $b$) is symmetrical to the source_close call, but is invoked by owner($b$).

We differentiate between two types of unidirectional payment channels, depending on whether the source close call is included in the allowed set of operations of the shared object or not. Note that without source close, the source depends on the target to close the channel to receive its deposit back after the channel is opened. However, for the target to receive its balance from the channel in the asset transfer system, it has to eventually close the channel. When the target closes the channel, the source also receives its respective balance.
6.2 Impossibility of unidirectional payment channel with source close

We begin by showing that a unidirectional payment channel that has the source close operation has a consensus number of at least 2, and therefore cannot be implemented in an asynchronous message passing network. Formally, we prove:

**Lemma 6.1.** Consensus has a wait-free implementation for 2 processes in the read-write shared memory model with an instance of a unidirectional payment channel with source close shared memory object and shared registers.

**Proof.** In Algorithm 2, only process \( p_1 \) transfers money to \( p_2 \) using the channel, and they both attempt to close the channel: \( p_1 \) after the payment is made, and \( p_2 \) without accepting the payment. Thus, changing the close call in Line 4 to \( B\.source\_close \) and the call in Line 8 to \( B\.target\_close \) yields a consensus algorithm among 2 processes using a unidirectional payment channel with source and target close operations. \( \square \)

Based on the above theorem and FLP [18], we get the following result:

**Theorem 6.1.** There does not exist an implementation of the unidirectional payment channel abstraction with source close in the asynchronous message-passing model.

6.3 Implementation of unidirectional payment channel without source close

We now observe the unidirectional payment channel abstraction without the source close operation. We first prove a lower bound on the message complexity of an implementation, and then prove that this lower bound is tight by providing an implementation for the abstraction in the asynchronous message-passing model.

6.3.1 Lower bound. We prove that any algorithm that implements the unidirectional payment channel specification incurs a combined message complexity for open, transfer, and close of \( \Omega(n^2) \). To this end, we prove the following lemma:

**Lemma 6.2.** Consider an algorithm that implements the unidirectional payment channel abstraction \( B \), and an asset transfer \( A \). Then there exists a run with \( B\.open, B\.transfer, B\.target\_close, \) and \( A\.read \) calls, in which correct processes send at least \((f/2)^2 \) messages.

**Proof.** Consider all runs with accounts \( a, b \) that are owned by a single process, i.e., \( owner(a) = owner(b) = p \), with initial balances \( A(a) = 1, A(b) = 0 \). A transfer call \( A\.transfer(a, ((b, 1))) \) by \( p \) can be implemented using a unidirectional channel \( (a, b) \) by having process \( p \) invoke \( B\.open((a, b), 1), B\.transfer((a, b), 1), \) and lastly, \( B\.target\_close((a, b), 1) \). Then, the proof proceeds in the same manner as the proof of Theorem 4.1. \( \square \)

6.3.2 Upper bound. We provide an algorithm in the asynchronous message-passing model that implements a unidirectional payment channel without source close. The algorithm assumes an asset transfer system \( A \), implemented as in [20], and discussed in §4.2.2. The algorithm is detailed in Algorithm 4. We denote an account name with a string \( c_1c_2 \cdots c_k \in A \) to refer to an account with a public key that is a \( k \)-of-\( k \) multisignature of \( \{owner(c_1), \ldots, owner(c_k)\} \).

For example, to sign an invocation of the asset transfer object \( A \) of account \( ab \), like transferring money from \( ab \) to another account, both \( owner(a) \) and \( owner(b) \) need to sign the message with their respective private keys before the call can be invoked. The transfer call with an appropriate multisignature can be invoked by any process, in particular, the last process to sign the invocation and complete the signature. When the algorithm mentions that a process creates an \( A\.transfer \) invocation, e.g., in lines 5 and 22, it does not mean the process invokes the transfer operation, but
Algorithm 4: Unidirectional payment channel without source\_close implementation in the asynchronous message-passing model. Operations for process $p$.

**Shared Objects:**
- $A$ - asset transfer object

**Local variables:**
- $source[]$ - a dictionary with the balances of all channels that $p$ is the source, initially $\bot$ for all channels
- $target[]$ - a dictionary with asset transfer multisig invocations for all channels $p$ is the target, initially $\bot$ for all channels

```plaintext
// This call can be invoked by $owner(a)$
Procedure open($a, b, amt$) :                          // This message is received by $owner(b)$
  if owner($a$) $\neq$ $p \lor source[ab] = \bot$ then
    return fail
  invoke $A$.transfer($a, \{(ab, amt)\}$) // $ab$ is a multisig account
  create an $A$.transfer($ab, \{(a, (amt), (b, 0))\}$) invocation $tx$
  add $p$'s signature to $tx$  // $tx$ is still not a valid transaction
  send ('open', $tx$, $amt$) to $owner(b)$
  source$[ab] \leftarrow (amt, 0)$
  return success

// This call can be invoked by $owner(a)$
Procedure transfer($a, b, amt$) :
  if owner($a$) $\neq$ $p \lor source[ab] = \bot$ then
    return
  ($bal_a, bal_b$) $\leftarrow source[ab]$
  if $bal_a = \bot$ then
    return
  ($new\_bal_a, new\_bal_b$) $\leftarrow (bal_a - amt, bal_b + amt)$
  create an $A$.transfer($ab, \{(a, new\_bal_a), (b, new\_bal_b)\}$)
  invocation $tx$
  add $p$'s signature to $tx$
  send ('transfer', $tx$, $amt$) to $owner(b)$
  source$[ab] \leftarrow (new\_bal_a, new\_bal_b)$

// This call can be invoked by $owner(b)$
Procedure target\_close($a, b, bal_a$) :
  if owner($b$) $\neq$ $p \lor target[ab] = \bot$ then
    return fail
  get $A$.transfer($ab, \{(a, curr\_bal_a), (b, curr\_bal_b)\}$) transaction $tx$
  from $target[ab]$
  if $bal_a = curr\_bal_a$ then
    return fail
  add $p$'s signature to $tx$  // this completes the multisig
  invoke $tx$  // invoke $A$ with closing transaction
  target$[ab] \leftarrow \bot$
  send ('close', $tx$) to $owner(a)$
  return success

Function validate\_tx($tx, a$):
  return $tx$ is a valid invocation of $A$ and it contains $owner(a)$'s signature

```

rather that it adds its signature to a multisignature message allowing an invocation of $A$'s operation. Any invocation of $A$.transfer call is mentioned explicitly (Lines 4, 41). We further assume FIFO order on messages sent between every two processes. This can be easily implemented with sequence numbers.
On Payment Channels in Asynchronous Money Transfer Systems

We explain below the implementation details of the algorithm for each of the abstraction’s operations.

**Open.** The open procedure of a channel \((a, b)\) (Line 1) requires \(p_1 = \text{owner}(a)\) to make an initial deposit by invoking the transfer method of \(A\) from account \(a\) to a multisignature account \(ab\) (Line 4). After the transfer is completed, \(p_1\) creates a transaction \(tx\) that transfers the deposit back to its account and 0 to \(p_2 = \text{owner}(b)\) and sends it to \(p_2\) (Line 6). Note that at this stage, process \(p_1\) cannot invoke \(A\) with \(tx\) since it requires a multisignature, but when \(p_2\) receives it, it has the ability to add its signature as well and then invoke \(A\) with the \(tx\).

When \(p_2\) receives \(tx\) this transaction message it also awaits for the balance in account \(ab\) to reflect the deposit (Line 10) to ensure the money was deposited in account \(ab\) using the asset transfer system, after which it considers the account as open.

**Transfer.** When \(p_1\) wants to transfer money in an open channel \((a, b)\) (Line 15) it creates a transaction \(tx\) which is an \(A\).transfer invocation transferring money from the multisignature account \(ab\) to accounts \(a\) and \(b\) with the last balance of the channel after the payment. E.g., if the balance of the channel is \((10, 1)\), and \(p_1\) wants to make a payment of 1 on the channel, it creates transaction \(tx\) required to invoke \(A\).transfer\((ab, [(a, 9), (b, 2)])\), which transfers 9 money units to \(p_1\) and 2 to \(p_2\). Then, \(p_1\) adds its signature to \(tx\) (Line 23), and sends it to \(p_2\) which stores it. Note that \(p_1\) cannot invoke \(A\) with \(tx\) since it is still missing \(p_2\)’s signature. Thus, making a payment on the channel simply requires one message from the source user to the target user containing \(tx\), and multiple payments can be done on the channel without invoking \(A\)’s transfer call.

**Close.** When \(p_2\) wants to close the channel \((a, b)\) (Line 34), it takes the last transaction of account \(ab\) it received from \(p_1\) and adds its signature to it (Line 41). \(p_2\)’s signature completes the multisignature, making it a valid transaction, and allowing \(p_2\) to use it to invoke \(A\)’s transfer operation (Line 41). Process \(p_2\) then sends \(tx\) to \(p_1\), which ensures the transaction is also delivered via the broadcast of \(A\) by checking that balance in account \(ab\) is 0, after which it considers the channel closed.

Thus, opening and closing of the channel requires invoking an \(A\).transfer operation, which incurs \(O(n^2)\) messages because of the broadcast, but transferring money on the channel itself requires only one message per transfer.

**Correctness.** We prove below that the implementation (Algorithm 4) is Byzantine sequentially consistent (BSC) in respect to the sequential specification (Specification 3). The algorithm can be changed to be Byzantine linearizable by having each message answered with an ack message. E.g., after the open message is received in Line 16, the process sends an ack message back to the original sender. The linearization point then is when the ack message is received. This change requires sending more messages as part of the algorithm and also extends the latency, but this change does not affect the overall message complexity. This is also the reason why we chose BSC as the correctness criteria.

We examine an execution \(E\) of the algorithm. Let \(H\) be \(E\)’s matching history. Let \(\tilde{H}\) be a completion of \(H\) by removing any pending open and close calls that did not reach \(A\)’s transfer call invocation (Lines 4 and 41, respectively). We define \(H’\) as an augmentation of \(\tilde{H}\) as follows: For any correct process \(q = \text{owner}(b)\) that invokes \(B.target\_close((a, b), bal_b)\) s.t. process \(p = \text{owner}(a)\) is a Byzantine process, we add before the target close call the following two Byzantine invocations to \(H’\) by \(p\):

- \(B.open((a, b), bal_a)\) with an account \(a\) s.t. \(A(a) \geq bal_a\). Since account \(a\) has enough money to open the channel, the open call succeeds.
- \(B.transfer((a, b), bal_a)\) which is invoked by process \(q\) immediately after the previous open call returns.

These two calls ensure that when \(q\) invokes the close operation, it succeeds.

Next, we construct a linearization \(E’\) of \(H’\) by defining the following linearization points:
• Any open or close call that fails is linearized immediately after its invocation.
• A transfer call that returns because of the if statements (Lines 16, 19) is linearized immediately after its invocation.
• Any successful \( \text{open}((a, b), \text{amt}) \) s.t. \( q = \text{owner}(b) \) is a correct process, then it linearizes after \( q \) reaches Line 10. If \( q \) is Byzantine then the open call linearizes when it ends.
• Any \( \text{transfer}((a, b), \text{amt}) \) that reaches Line 24 s.t. \( q = \text{owner}(b) \) is a correct process, then it linearizes after \( q \) reaches Line 28. If \( q \) is Byzantine then the transfer call linearizes when it ends.
• Any successful \( \text{target\_close}((a, b), \text{bal}_b) \) s.t. \( p = \text{owner}(a) \) is a correct process, then it linearizes after \( p \) reaches Line 46. If \( p \) is Byzantine then the close call linearizes when it ends.

We now prove that \( E' \), the linearization of \( H' \), satisfies the sequential specification of a unidirectional payment channel without source close. We assume that for channel \((a, b)\), \( p = \text{owner}(a), q = \text{owner}(b) \).

**Lemma 6.3.** A B.open call for channel \((a, b)\) succeeds only if the channel is closed when the call is invoked.

**Proof.** Immediate from the algorithm. A channel \((a, b)\) opening fails if \( \text{source}[ab] = \bot \) (Line 2). This is the case for all channels at the beginning of the run, or if the channel was previously closed successfully (Line 48).

**Lemma 6.4.** For any B.transfer call for channel \((a, b)\) there is a preceding open call for the channel in \( H' \).

**Proof.** If the transfer call in \( H' \) is invoked by a correct process, then from the algorithm \( \text{source}[ab] \neq \bot \). This is only possible by the algorithm if \( p \) invokes an open call for the channel before the transfer invocation. Therefore, if \( q \) is correct, then the open and transfer calls linearize when \( q \) receives the messages for the corresponding calls (Lines 10, 24). Since we assume FIFO order on the links between any two processes, then the open call linearizes before the transfer call. If \( q \) is Byzantine, then the open and transfer calls linearize immediately after they successfully return.

If \( p \) is Byzantine, then the transfer invocation is in \( H' \) because there is some close invocation by a correct process \( q \). Before that, there is also a matching open call by \( p \). The open call linearizes before the transfer call.

**Lemma 6.5.** For any successful B[target\_close](a, b, balb) call in \( H' \) there is a preceding B.open((a, b), amt) call for channel \((a, b)\) s.t. \( \text{amt} \geq \text{bal}_b \), followed by a B.transfer call that changes the state of the channel to \((\text{bal}_a, \text{bal}_b)\) for \( \text{bal}_a = \text{amt} - \text{bal}_b \).

**Proof.** If the close call ends successfully, then \( q \) has in target[ab] a valid transaction A.transfer(ab, [(a, bal_a), (b, bal_b)]), otherwise, the call fails. Therefore, if \( p \) is a correct process, it opens the channel \((a, b)\) with some deposit \( \text{amt} \), and transfers in the channel s.t. the balances in the channel change to \((\text{bal}_a, \text{bal}_b)\). Both the open and transfer are linearized before the close invocation, otherwise, the close call fails. If \( p \) is Byzantine, then we add the matching open and transfer invocations to \( H' \) which are linearized before the close invocation.

Finally, we prove resiliency to Byzantine processes.

**Lemma 6.6.** In an infinite execution of Algorithm 4 every call invoked by a correct process eventually returns.

**Proof.** In all cases where an open or close calls returns fail it does so immediately, since it is done prior to any invocation of A. Transfer calls that return due to the if statements (Lines 16, 19) also return immediately.

For the open call, the if condition in the channel open call (Line 2) ensures that the process that invokes the call owns account \( a \) and that it has enough balance to open the channel. We also assume that a correct process does not
invoke a new call before a previous call has a response event. Therefore, the conditions checked during the if statement hold when A’s transfer call is invoked, and by the asset transfer specification the call succeeds.

A transfer call that does not invoke any of A’s calls, nor does it wait for a reply after it sends the transaction in Line 24. Therefore, this call also returns immediately.

A target_close call that returns success invokes A with a transfer call that transfers money from account ab (Line 41). To reach this line, the process has to first check if the channel is open, and it has the matching transaction in target[ab] during the if statement of the call. Therefore, invoking A will eventually succeed by the asset transfer specification, and the target_close call returns successfully.

Thus, we can conclude the above lemmas with the following result:

**Theorem 6.2.** Algorithm 4 implements a Byzantine sequentially consistent unidirectional payment channel without source close abstraction (Specification 3).

### 7 CHAIN PAYMENTS

We can extend the discussion of payment channels to chain payments. A chain payment system allows making payments off-chain between users who do not share a direct payment channel between them, but do share a route through intermediate users. For example, suppose Alice wants to make a payment to Bob, but she does not share a direct payment channel with him. Rather, she has a channel with Charlie, and Charlie has a channel with Bob. A chain payment allows using the route from Alice to Bob via Charlie to make the payment on all channels atomically. Chain payments are used extensively as part of the Lightning Network [35] on Bitcoin.

The intuitive way of defining a chain payment object is with a single operation that makes a payment through a chain ((a1, a2), (a2, a3), . . . , (a_k−1, a_k)), and the outcome of the payment affects all channels of the chain in an atomic manner. For example, suppose the balances of the above channels of the chain are (bal1, bal2), (bal2, bal3), . . . , (bal_k−1, bal_k), respectively, and a payment of amt is made via the chain. Then, after the linearization point, the balances of the channels are (bal1 − amt, bal2 + amt), (bal2 − amt, bal3 + amt), . . . , (bal_k−1 − amt, bal_k + amt), respectively. In this case, assuming that the channels are bidirectional or unidirectional with source close, it can be proven that the consensus number of such chain payment object is k, meaning this object can be used to solve consensus between k processes, in a similar manner to the 2-consensus we prove in this paper for these channel types.

We also note that even if the channels of the chain are unidirectional without source close, implementing a chain payment is not intuitive and straightforward in asynchronous networks. A possible solution is to adopt the use of Hash Time-Locked Contracts (HTLCs) [13, 24] which are in use in the Lightning Network for chain payments. An HTLC is a special conditioned payment between two users Alice and Bob. It asserts that Bob receives the payment if he can present a hash pre-image of a specific value before a timeout, otherwise Alice gets the payment. If we equip the asset transfer system with a Hash Time Contract operation (without the timeout component), we can implement chain payments in asynchronous networks based on the unidirectional channels without source close we presented in Algorithm 4. We leave as an open research question other possible asynchronous implementations of chain payments for future work.

### 8 CONCLUSION

In this paper we presented the possibility of using payment channels in asynchronous asset transfer systems as a scaling solution. We first proved that without such channels, a payment in an asset transfer system requires a quadratic
message complexity per payment. We then proved that bidirectional payment channels cannot be implemented in
asynchrony, and moreover, not all unidirectional payment channels can.

REFERENCES

[1] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically optimal validated asynchronous byzantine agreement. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. 337–346.

[2] Frederik Armknecht, Ghassan O Karam, Avikarsha Mandal, Franck Youssef, and Erik Zenner. 2015. Ripple: Overview and outlook. In International Conference on Trust and Trustworthy Computing. Springer, 163–180.

[3] Hagit Attiya and Jennifer L Welch. 1994. Sequential consistency versus linearizability. ACM Transactions on Computer Systems (TOCS) 12, 2 (1994), 91–122.

[4] Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. 2020. Money Transfer Made Simple: a Specification, a Generic Algorithm, and its Proof. Bull. EATCS 152 (2020). http://eatcs.org/index.php/eatcs/article/view/629

[5] Mihir Bellare and Gregory Neven. 2007. Identity-based multi-signatures from RSA. In Cryptographers’ Track at the RSA Conference. Springer, 145–162.

[6] Shai Ben-David, Allan Borodin, Richard Karp, Gabor Tardos, and Avi Wigderson. 1994. On the power of randomization in on-line algorithms. Algorithmica 11, 1 (1994), 2–14.

[7] El Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. IACR Cryptol. ePrint Arch. 2018 (2018), 46.

[8] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. Information and Computation 75, 2 (1987), 130–143.

[9] Ethan Buchman. 2017. Tendermint: Byzantine fault tolerance in the age of blockchains (2016).

[10] Shir Cohen and Idit Keidar. 2021. Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer. In 35th International Symposium on Distributed Computing (DISC 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209), Seth Gilbert (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:18. https://doi.org/10.4230/LIPIcs.DISC.2021.18

[11] Shir Cohen, Idit Keidar, and Alexander Spiegelman. 2020. Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP. In 34th International Symposium on Distributed Computing (DISC 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[12] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Matteo Monti, Athanasios Sygkis, Matej Pavlovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. 2020. Ordine Payments by Merely Broadcasting Messages (Extended Version). arXiv preprint arXiv:2004.13184 (2020).

[13] Christian Decker and Roger Wattenhofer. 2015. A fast and scalable payment network with bitcoin duplex micropayment channels. In Symposium on Self-Stabilizing Systems. Springer, 3–18.

[14] Danny Dolev and Ruudigier Reischuk. 1982. Bounds on Information Exchange for Byzantine Agreement. In Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC ’82). Association for Computing Machinery, New York, NY, USA, 132–140.

[15] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. Journal of the ACM (JACM) 35, 2 (1988), 288–323.

[16] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. 2017. PERUN: Virtual Payment Channels over Cryptographic Currencies. IACR Cryptol. ePrint Arch. 2017 (2017), 635.

[17] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. 2016. Bitcoin-ng: A scalable blockchain protocol. In 13th {USENIX} symposium on networked systems design and implementation (NSDI ’16), 45–59.

[18] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) 32, 2 (1985), 374–382.

[19] Matthew Green and Ian Miers. 2017. Bolt: Anonymous payment channels for decentralized currencies. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 473–489.

[20] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2019. The consensus number of a cryptocurrency. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. 307–316.

[21] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster asynchronous BFT protocols. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. 803–818.

[22] Saurabh Gupta. 2016. An asymptotically optimal validated asynchronous byzantine agreement algorithm. In Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing. 337–346.

[23] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) 12, 3 (1990), 463–492.

[24] HTLC. [n. d.]. Hash Time Locked Contracts. https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts

[25] Harry Kalodner, Steven Goldfeder, Xiaochi Chen, S Matthew Weinberg, and Edward W Felten. 2018. Arbitrum: Scalable, private smart contracts. In 27th {USENIX} Security Symposium ({USENIX} Security 18). 1353–1370.
