Distributed Computing (2023) 36:137–157
https://doi.org/10.1007/s00446-023-00446-0

Cross-chain payment protocols with success guarantees

Rob van Glabbeek[1,2] · **Vincent Gramoli**[3,4] · **Pierre Tholoniat**[3,5]

Received: 26 September 2021 / Accepted: 26 February 2023 / Published online: 8 April 2023

Abstract

In this paper, we consider the problem of cross-chain payment whereby customers of different escrows—implemented by a bank or a blockchain smart contract—successfully transfer digital assets without trusting each other. Prior to this work, cross-chain payment problems did not require this success, or any form of progress. We introduce a new specification formalism called *Asynchronous Networks of Timed Automata* to formalise such protocols. We present the first cross-chain payment protocol that ensures termination in a bounded amount of time and works correctly in the presence of clock drift. We then demonstrate that it is impossible to solve this problem without assuming synchrony, in the sense that each message is guaranteed to arrive within a known amount of time. Yet, we solve an eventually terminating weaker variant of this problem, where success is conditional on the patience of the participants, without assuming synchrony, and in the presence of Byzantine failures. We also discuss the relation with the recently defined cross-chain deals.

Keywords Distributed systems · Cross-chain payment protocols · Fault tolerance · Blockchain · Asynchronous networks of timed automata · Asynchronous communication · Clock drift · Safety and liveness properties

1 Introduction

With the advent of various payment protocols comes the problem of interoperability between them. A simple way for users of different protocols to interact is to do a *cross-chain payment* whereby intermediaries can help customer Alice transfer digital assets to Bob even though Alice and Bob own accounts in different banks or blockchains.

A payment between two customers of the same bank is simple. Alice just informs the bank that she wants to transfer a certain amount from her account to the account of the receiving party Bob; and then the bank carries out this request. Alice and Bob do not need to trust each other but need to trust the bank to not withdraw the money from Alice's account and never deposit it on Bob's. As Bob trusts the bank, he can issue a signed certificate assuring Alice that if the bank says that he has been paid, then Alice is off the hook, and any further disputes about possible non-payment to Bob are between Bob and the bank. To prevent litigation against her, Alice simply needs this statement from Bob as well as a statement that Bob has been paid. As Alice does not trust Bob, this second statement must come from the bank.

To generalise this protocol to payments between customers of different banks, it helps if the two banks have ways to transfer assets to each other, and moreover trust each other. A sound protocol is:

(i) Bob provides Alice with a signed statement that all he requires for her to have satisfied her payment obligation, is a statement from his own bank saying that he has been paid.
(ii) Alice's bank promises Alice that if she transfers money to Bob, she will get a statement from Bob's bank that the transfer has been carried out.
1. Alice orders her bank to initiate the transfer to Bob.
2. Alice's bank withdraws the money from her account, and sends it to Bob's bank.

A 3-page summary of this paper appears as [9].

✉ Rob van Glabbeek
rvg@cs.stanford.edu

Vincent Gramoli
vincent.gramoli@sydney.edu.au

Pierre Tholoniat
pierre@cs.columbia.edu

1 Data61, CSIRO, Sydney, Australia
2 UNSW, Sydney, Australia
3 University of Sydney, Sydney, Australia
4 EPFL, Lausanne, Switzerland
5 École Polytechnique, Paris, France

3a. Bob's bank places the money in Bob's account
3b. and notifies Alice's bank of this.
4. Alice's bank forwards to Alice the statement by Bob's bank saying that Bob has been paid.

This is roughly how payments between customers of different banks happen in the world of banking. Step (ii) is part of general banking agreements, not specific to Alice and Bob. Step (i) is typically left implicit in negotiations between Alice and Bob. All that Alice needs is the combination of steps (i), (ii) and 4 above. Once step (i) and (ii) have been made, she confidently takes step 1, knowing that this will be followed by Step 4. Alice's bank is willing to take step (ii) because it trusts Bob's bank, in the sense that step 2 *will* be followed by Step 3b. In fact, when the two banks trust each other, and have ways to transfer assets to each other, they can abstractly be seen as one bigger bank, and the problem becomes similar to the problem of payments between customers of the same bank.

The problem becomes more interesting when the banks cannot transfer assets to each other and the only trust is the one of customers to their own bank. Typical solutions consist of considering banks as *escrows* and having intermediaries, like Chloe, that play the role of connectors between these escrows. Figure 1 depicts the relations of trust between three customers and two escrows, and where the flow of money is from left to right. Thomas and Schwartz [24] propose two cross-chain payment protocols: (i) the universal protocol requires *synchrony* [6] in that every message between participants is received within a known upper bound and the clock drift between participants is bounded[1]; (ii) the atomic protocol merely requires *partial synchrony* [6], meaning these upperbounds exist but are not known, or that it is known that after a finite but unknown amount of time these upperbounds will come into effect; it coordinates transfers using an ad-hoc group of notaries, selected by the participants, and relies on more than two-third of the notaries to be reliable. Herlihy et al. [14] represent a cross-chain payment as a deal matrix M where $M_{i,j}$ characterises a transfer of some asset from participant i to participant j. They offer a timelock[2] protocol that requires synchrony, and a certified blockchain protocol that requires partial synchrony and a certified blockchain. However, the synchronous solutions of [24] and [14] do not consider clock drift, and for their partially synchronous solutions no success guarantees are established.

Here we introduce a new specification formalism for cross-chain payment protocols, called *Asynchronous Networks of Timed Automata (ANTA)*. ANTA simplify the representation of cross-chain payment to a family of customer automata and a family of escrow automata that describe states from which outgoing transitions are immediately enabled and states from which they are conditionally enabled. These automata allow us to reason formally about the liveness and safety of cross-chain payment protocols. ANTA differ from Alur and Dill's *timed automata* [1], their networks [3] and I/O automata [18] in subtle ways, tuned to the problem at hand. We illustrate ANTA by specifying the universal protocol from [24] and proving that it solves the time-bounded variant of the cross-chain payment problem. Moreover, we fine-tune the protocol to work correctly even in the presence of clock drift.

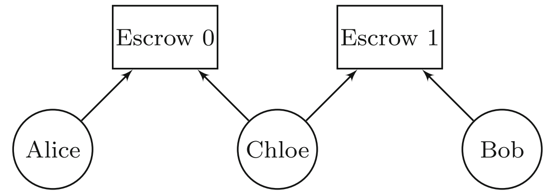


Fig. 1 Trust relations

We also show that there exists no algorithm that can solve the cross-chain payment problem without assuming synchrony, even if we relax the problem statement by merely requiring eventual (instead of time-bounded) termination, and even if all participants either behave correctly or simply crash (rather then displaying Byzantine behaviour). This impossibility result relies on classic indistinguishability arguments from the distributed computing literature and highlights an interesting relation between the cross-chain payment problem and the well-known transaction commit problem [10–12]. Inspired by this earlier work on the transaction commit, we define a weaker variant of the cross-chain payment problem that relaxes the liveness guarantees to be solvable with partial synchrony. This new problem differs from the transaction commit problem and its variants like the non-blocking weak atomic commit problem [11] by tolerating Byzantine failures. It is also different from the problems solved in [24] and [14] in a partially synchronous setting, by requiring some liveness. In particular, a protocol where all participants always abort is not permitted by our problem specification. We propose an algorithm that solves this variant only assuming partial synchrony, and in the presence of Byzantine failures, using the ANTA formalism.

Interestingly, the classical notion of *atomicity*, meaning that the entire transaction goes through, or is rolled back completely, is not appropriate for this kind of protocols. In the words of [14], "This notion of atomicity cannot be guaranteed when parties are potentially malicious: the best one can do is to ensure that honest parties cannot be cheated."

[1] In [24] proper clock drift is not considered; instead clocks may drift up to a bounded amount of time.

[2] https://en.bitcoin.it/wiki/Hashed_Timelock_Contracts.

2 Model and definitions

2.1 Participants and money

We assume n banks or *escrows* $e_0, \ldots, e_{n-1}$ and $n+1$ *customers* $c_0, \ldots, c_n$. These $2n+1$ processes are called *participants*. An escrow is a specific type of process that can handle values for other parties in a predefined manner. Customer c_0 is Alice and c_n is Bob. The customers $c_1, \ldots, c_{n-1}$ are intermediaries in the interaction between Alice and Bob; we call them *connectors*, named Chloe$_i$. Customers c_{i-1} and c_i have accounts at escrow e_{i-1}, and trust this escrow ($i = 1, \ldots, n$). We do not assume any other relations of trust.
We expect that in most applications n=2, meaning that we have a single connector Chloe. It could for instance be that the two escrows e_1 and e_1 are the Bitcoin and Etherium networks. Alice want to pay Bob in Bitcoins, but Bob wants to receive Ethers. Chloe is in the business of facilitating such transactions; she is a customer of both Bitcoin and Etherium. We also treat the case of multiple connectors, as this can be done without much additional complications.

Topology. Not every customer can send value to any other. Here we assume that value can be transferred directly only between customers of the same escrow. Moreover, any transfer between two customers of an escrow can be modelled as two transfers: one from the originating customer to the escrow, and one from the escrow to the receiving customer. Thus, the connections from Fig. 2 describe both the relations of trust and the possible transfers of value. The case $n = 2$ was depicted in Fig. 1.

Note that the total space of customers and escrows can be an arbitrary bipartite graph. We need to consider, however, only those escrows and connectors that lay on the path from Alice to Bob that is chosen for a particular transfer.

Placing value in escrow. Customers can send a specific type of message to ask their escrow to put money aside for them. In particular, two customers may make a deal with an escrow to place value originating from the first customer "in escrow", and, after a predefined period, depending on which conditions are met, either complete the transfer to the second customer, or return the value to the first one. This predefined period is relative to the escrow's local time. The clocks of the $2n + 1$ processes are not necessarily synchronised (see 2.3).

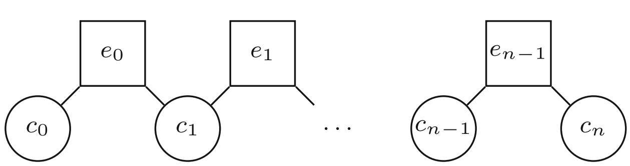


Fig. 2 Customers and escrows

Abstracting the transfer of value. There are many ways of transferring value from one party to another: one could give someone a physical object, such as cash or gold. One could also send a transaction to transfer cryptocurrency or tokens on a distributed ledger, or send a specific message on some banking application. We do not care of how this process is implemented, and we suppose that the participants have already agreed upon the value they expect to be transferred. We use therefore a unified notation: $s(p, \$)$ to say "send a message to trigger the transfer of some previously agreed-upon value to participant p".

Chloe's fee. As Chloe helps out transferring value from Alice to Bob, it is only reasonable that she is paid a small commission. Hence the value transferred from Alice to Chloe might be larger than the value transferred from Chloe to Bob. Additionally, these values may be expressed in different currencies, with possibly fluctuating exchange rates, or they may be objects such as bags of flour that have a quality-dependent value. Deciding which values to transfer may thus be an interesting problem. However, it is entirely orthogonal to the matter discussed in this paper, and hence we shall not consider it any further.

2.2 Communication and computation model

The following model holds for the rest of the paper. When necessary, we will mention explicitly if we have to add some assumptions, such as synchrony of communication.

Communication. We assume that the network does not lose, duplicate, modify or create messages. However, messages can be delayed arbitrarily long: by default we assume asynchronous communication.

The assumption that messages are not duplicated does not restrict generality, for the sender could always equip messages with a unique sequence number, with the understanding that the receiver drops all messages that by inspection of this sequence number can be seen to be duplicates of an already received message.

Likewise, message loss can be mitigated by a retransmit and acknowledgement protocol in which the sender retransmits each outgoing message periodically until it either receives an acknowledgement or interferes from context that the message must have been received. The recipient is then asked to reply to each receipt of a message with an acknowledgement, to stop the retansmissions. The only way this protocol can fail to deliver a message is when the network fails in perpetuity. Our assumption of no message loss thus merely says that the latter will not happen.

Finally, the possibility that the network modifies a message can be reduced to the case that it drops the message by using appropriate encryption, so that a modified message will not decrypt and be discarded.

Authentication. We assume that each customer can sign a message with his unique identifier, thanks to an idealised public key infrastructure. No other process can forge its signature, and any process (including escrows) can verify it.

Certificates. As a consequence, any customer can issue a certificate by signing an appropriate message. For example, Bob can issue a receipt certificate to Alice by signing a RECEIVED message. By combining several signatures, one can define threshold certificates, for instance requiring the signature of a COMMIT message by strictly more than some number of customers. A correct implementation of certificates should take care of preventing replay attacks.

Faults. We do not make any assumption on the behaviour of the processes *a priori*. Later we will define a protocol, and processes will either follow the protocol or deviate from it.

2.3 Synchronous versus asynchronous communication

In the literature five levels of synchrony in communication can be distinguished. As indicated in the table below, terminology is not uniform between the concurrency and the distributed systems communities.

concurrency	*distributed systems*
synchronous communication	rendezvous
I/O automata	I/O automata
asynchronous communication	synchronous communication
	partially synchronous communication
	asynchronous communication

In the concurrency community, communication is called *synchronous* if sending and receiving occur simultaneously, and the sender cannot proceed before receipt of the message is complete. This is the typical paradigm in process algebras such as CCS [20]. Communication is called *asynchronous* if sending occurs strictly before receipt, and the sender can proceed after sending regardless of the state of the recipient(s). An intermediate form is modelled by I/O automata [18]; here sending and receipt is assumed to occur simultaneously, yet the sender proceeds after sending regardless of the state of possible receivers.

In the distributed systems community all communication is by default assumed to be asynchronous in the sense above; synchronous communication as defined above is sometimes called a *rendezvous*. Following [6], communication is called *synchronous* when there is a known upperbound on the time messages can be in transit, and moreover there is a known upperbound on the relative clock drift between parallel processes. It is *partially synchronous* when these upperbounds exist but are not known, or when it is known that after a finite but unknown amount of time these upperbounds will come into effect. If these conditions are not met, communication is deemed *asynchronous*.

The present paper follows the latter terminology; we speak of *fully synchronous* communication when referring to synchronous communication from concurrency theory.

2.4 The syntax and semantics of ANTA

Although strongly inspired by the timed automata of Alur and Dill [1] and their networks [3], our *Asynchronous Networks of Timed Automata (ANTA)* differ from those models in subtle ways, tuned to the problem at hand. The first A refers to asynchronous communication in the concurrency-theoretic sense, and contrasts with the fully synchronous (rendezvous-style) communication assumed in NTA [3]. A crucial difference between ANTA and *Communicating Timed Automata* [16], which also employ asynchronous communication, is that in the latter “all automata move synchronously; time passes at the same pace for all of them” [16].

The formal syntax and semantics of ANTA is given in Appendix A. However, the explanations below suffice for understanding our protocols.

In Fig. 3 our time-bounded cross-chain protocol—essentially the universal protocol from [24]—is depicted as an ANTA. There is one automaton for each participant in the protocol, that is, for each escrow e_i $(i = 0, \ldots, n-1)$ and each customer c_i $(i = 0, \ldots, n)$. Each automaton is equipped with a unique identifier, in this case e_i and c_i. Each automaton has a finite number of *states*, depicted as circles, one of which is marked as the *initial state*, indicated by a short incoming arrow. The states are partitioned into *termination states*, indicated by a double circle, *input states*, coloured white, and *output states*, coloured grey or black. Furthermore there are finitely many *transitions*, indicated as arrows between states. The transitions are partitioned into *input transitions*, labelled $r(id, m)$, *output transitions*, labelled $s(id, m)$, and *time-out transitions*, labelled by arithmetical formulas ψ featuring the variable *now*. Here id must be the identifier of another automaton in the network, and m a message, taken from a set $\mathtt{MSG}$ of allowed messages. Whereas each input and time-out transition has a unique label $r(id, m)$ and ψ, respectively, an output transition may have multiple labels $s(id, m)$. All transitions may have additional labels $u := now$ for some variable u. A termination state has no outgoing transitions, and an output state exactly one, which must be an output transition. An input state may have any number of outgoing input and time-out transitions, and no outgoing output transitions.

Each automaton keeps an internal clock, whose value, a real number, is stored in the variable *now*. The value of *now* increases monotonically as time goes on. All variables maintained by an automaton are local to that automaton, and not accessible by other automata in the network. Each transition is assumed to occur instantaneously, at a particular point in time. In case a transition occurs that is labelled by an assignment $u := now$, the variable u will remember the point in time when the transition took place. Such a variable may be used later in time-out formulas. When (or shortly before, see below) an output transition with label $s(id, m)$ occurs, the automaton sends the message m to the automaton with identifier id. A time-out transition labelled ψ is *enabled* at a time *now* when the formula ψ evaluates to `true`. An input transition labelled $r(id, m)$ is *enabled* only at a time when the automaton receives the message m from the automaton id in the network. Whereas an output transition may be scheduled to occur by the automaton at any time, an input or time-out transition can occur only when enabled.

When an automaton is not performing a transition, it must be in exactly one of its states. It starts at the initial state, where its clock is initialised with an arbitrary value. When the automaton is in an input state, it stays there (possibly forever) until one of its outgoing transitions becomes enabled; in that case that transition will be taken immediately. In case multiple transitions become enabled simultaneously, the choice is non-deterministic. When the automaton reaches a termination state, it halts.

In general, an output state is labelled with a positive *time-out* value $to \in \mathbb{R} \cup \{\infty\}$. It constitutes a strict upperbound on the time the automaton will stay in that state. In case the automaton enters an output state at time *now*, it will take its outgoing transition between times *now* and $now + to$. If its output transition has multiple labels $s(id, m)$, the corresponding transmissions need not occur simultaneously; they can occur in any order between *now* and $now + to$. The output transition is considered to be taken when the last of these actions occurs. In this paper time-out values are indicated by colouring: for the grey states it is the constant ε from Sect. 3.1, and for the black state it is ∞.

2.5 Cross-chain payment protocol

A cross-chain payment protocol prescribes a behaviour for each of the participants in the protocol, the escrows and the customers. Let χ be a certificate signed by Bob saying that Alice's obligation to pay him has been met.

Definition 1 (*Time-bounded cross-chain payment protocol*) A cross-chain payment protocol is a *time-bounded cross-chain payment protocol* if it satisfies the following properties:

- C *Consistency.* For each participant in the protocol it is possible to abide by the protocol.
- T′ *Time-bounded termination.* Each customer that abides by the protocol, and either makes a payment or issues a certificate, terminates within an a priori known period, provided her escrows abide by the protocol.
- ES *Escrow security.* Each escrow that abides by the protocol does not lose money.
- CS *Customer security.*
  - CS1 Upon termination, if Alice and her escrow abide by the protocol, Alice has either got her money back or received the statement χ.
  - CS2 Upon termination, provided Bob and his escrow abide by the protocol, Bob has either received the money or not issued certificate χ.
  - CS3 Upon termination, each connector that abides by the protocol has got her money back, provided her escrows abide by the protocol.
- L *Strong liveness.* If all parties abide by the protocol, Bob is paid eventually.

Requirement C (*consistency* of the protocol) is essential. In the absence of this requirement, any protocol that prescribes an impossible task for each participant would be a correct cross-chain payment protocol (since it trivially meets T, ES, CS and L).

Requirements ES and CS (the *safety* properties) say that if a participant abides by the protocol, nothing really bad can happen to her. These requirements do not assume that any other participant abides by the protocol, and should hold no matter how malicious the other participants turn out to be. The only exception is that the safety properties for a customer (CS) are guaranteed only when the escrow(s) of this customer abide by the protocol.

Property L, saying that the protocol serves its intended purpose, is the only one that is contingent on *all* parties abiding by the protocol.

3 A time-bounded protocol

3.1 Assumptions

Synchrony. The assumption of *synchrony* considered by [6], and called *bounded synchrony* by [24], says 'that there is a fixed upper bound Δ on the time for messages to be delivered (*communication is synchronous*) and a fixed upper bound Φ on the rate at which one processor's clock can run faster than another's (*processors are synchronous*), and that these bounds are known a priori and can be "built into" the protocol.' [6] A consequence of this assumption is that if participant p_1 sends at its local time t_0 a message to participant p_2, and participant p_2 takes t units of its local time to send

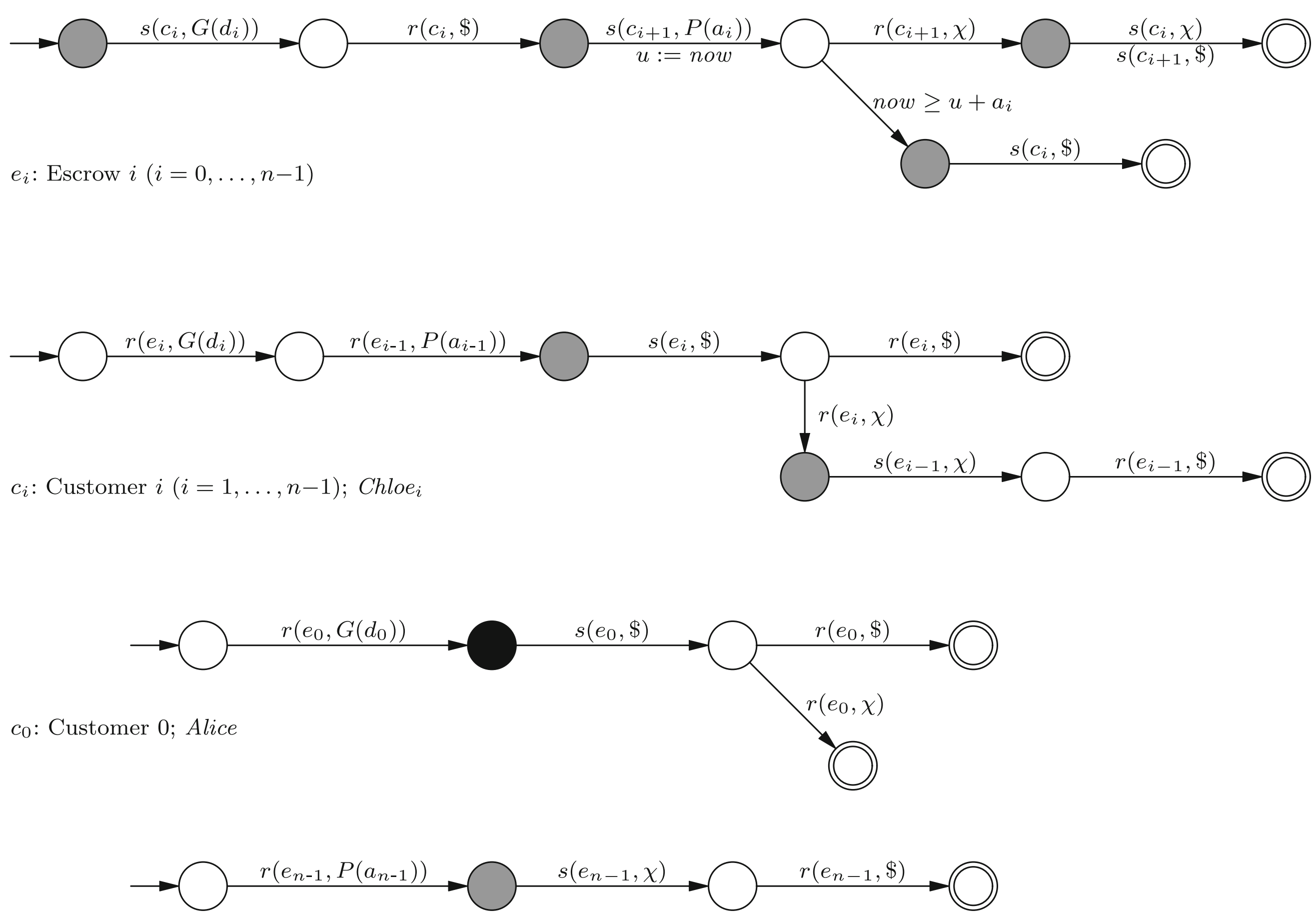


Fig. 3 Automata representing escrows and customers

an answer back to p_2, then p_1 can count on arrival of that reply no later than time $t_0 + \Phi \cdot t + 2 \cdot \Delta$.

Bounded reaction speed. When a participant in the protocol receives a message, it will take some time to calculate the right response and then to transmit that response. Here it will be essential that that amount of time is bounded. So we assume a reaction time $\varepsilon > 0$ such that any message can be answered within time ε.

3.2 A cross-chain payment protocol formalised as an ANTA

In this section we formally model the universal protocol from [24] as an ANTA. Moreover, we replace the timing constants employed in [24] by parameters, and calculate the optimal value of these parameters to ensure correctness of the protocol in the presence of clock drift.

To interpret Fig. 3, all that is left to do is specify the messages that are exchanged between escrows and their customers. We consider 4 kinds of messages. One is the certificate χ, signed by Bob, saying that Alice's obligation to pay him has been met. Another is the value \$ that is transmitted from one participant to another. The remaining messages are promises made by escrow e_i to its customers c_i and c_{i+1}, respectively:

$G(d)$:= 'I guarantee that if I receive \$ from you at my local time w, then I will send you either \$ or χ by my local time $w + d$."

$P(a)$:= "I promise that if I receive χ from you at my local time v, with $v < now + a$, then I will send you \$ by my local time $v + \varepsilon$."

The automata of Fig. 3 can be informally described as follows: An escrow e_i first sends promise $G(d_i)$ to its (upstream) customer c_i. Here "upstream" refers to the flow of money. The precise values of d_i will be determined later; here they are simply parameters in the design of the protocol. Then it awaits receipt of the money/value from customer c_i. If the money does arrive, the escrow issues promise $P(a_i)$ to its downstream customer c_{i+1} as soon as it can. It remembers the time this promise was issued as u. Then it awaits receipt of the certificate χ from customer c_{i+1}. If the certificate does not arrive by time $u + a_i$, a time-out occurs, and the escrow refunds the money to customer c_i. If the certificate does arrive in time, the escrow reacts by forwarding it to customer c_i, and the money to customer c_{i+1}.

A connector Chloe$_i$ starts by awaiting promises $G(d_i)$ from her downstream escrow e_i, and $P(a_{i-1})$ from her upstream escrow e_{i-1}. Then she proceeds by sending the money to escrow e_i. After sending the money, Chloe$_i$ waits for escrow e_i to send her either the certificate χ or the money back. In the latter case, her work is done; in the former, she forwards the certificate to escrow e_{i-1} and awaits for the money to be sent by escrow e_{i-1}.

The automata for Alice and Bob are both simplifications of the one for Chloe$_i$. Alice awaits promise $G(d_0)$ from her escrow, and then sends the escrow the money. The protocol allows her to wait arbitrarily long before taking that step. Subsequently, she patiently awaits either the return of her money, or certificate χ. Bob awaits promise $P(a_{n-1})$ from his escrow, and then issues certificate χ and sends it to his escrow. He then awaits the money.

3.3 Running the protocol

The protocol consists of two parts. The *set-up* involves the sending and receiving of the promises $G(d_i)$. As these promises are not time-sensitive, they can be exchanged months before the *active part* of the protocol is ran, consisting of all other actions. Here an *action* is an entity $act@p$, with *act* a transition label, and p the identifier of the participant taking that transition. The active part has essentially only one successful run, i.e., when never taking a time-out transition, consisting of the following actions, executed in the following order. Actions separated by commas are executed in either order.

$$
\begin{array}{llcc}
s(e_0,\$)@c_0 & r(c_0,\$)@e_0 & s(c_1,P(a_0))@e_0 & r(e_0,P(a_0))@c_1 \\
s(e_1,\$)@c_1 & r(c_1,\$)@e_1 & s(c_2,P(a_1))@e_1 & r(e_1,P(a_1))@c_2 \\
\dots & & & \\
s(e_k,\$)@c_k & r(c_k,\$)@e_k & s(c_n,P(a_k))@e_k & r(e_k,P(a_k))@c_n \\
s(e_k,\chi)@c_n & r(c_n,\chi)@e_k & \multicolumn{2}{c}{s(c_n,\$),\ s(c_k,\chi)@e_k} \\
 & & r(e_k,\$)@c_n \quad , & r(e_k,\chi)@c_k \\
\dots & & & \\
s(e_1,\chi)@c_2 & r(c_2,\chi)@e_1 & \multicolumn{2}{c}{s(c_2,\$),s(c_1,\chi)@e_1} \\
 & & r(e_1,\$)@c_2 \quad , & r(e_1,\chi)@c_1 \\
s(e_0,\chi)@c_1 & r(c_1,\chi)@e_0 & \multicolumn{2}{c}{s(c_1,\$),s(c_0,\chi)@e_0} \\
 & & r(e_0,\$)@c_1 \quad , & r(e_0,\chi)@c_0
\end{array}
$$

Here $k := n-1$.

An essential feature of this protocol is that the valuable certificate χ passes through the hands of the intermediaries Chloe$_i$ on the way from Bob to Alice. When Chloe$_i$ sends money to her downstream Escrow i (on the way to Bob), she needs a guarantee that she will not lose this money. This guarantee is delivered through a case distinction.

- Her downstream bank promises to either refund her money, or provide the certificate χ in time.
- Her upstream bank promises to pay out if she supplies the certificate χ in time.

Together, this provides a fail-safe guarantee for Chloe$_i$, provided her banks can be trusted.

3.4 Initialisation

There is a scenario where Chloe$_i$ will never send money to escrow i, namely when she receives promise $P(a_{i-1})$ from escrow e_{i-1} before she receives promise $G(d_i)$ from escrow e_i. In that case the receipt of $P(a_{i-1})$ does not trigger a transition, and Chloe$_i$ will remain stuck in her second state. We now modify the protocol in such a way that this cannot occur. This can be done in several ways; it does not matter which of the three modifications we take.

1. One solution is to make Alice wait before starting the active part of the protocol (by leaving the black state) until a point in time when she is sure that all parties Chloe$_i$ already have received promise $G(d_i)$. If we assume that all parties start at the same time, using the reasoning of Sect. 3.1, Alice has to wait at most $\Phi \cdot \varepsilon + \Delta$ before this point has been reached. The only drawback of this solution is that it may be hard to realise that all parties start at the same time.
2. Another approach is to assume that the set-up phase occurred long before Alice actually wants to send money to Bob. It may be part of a general banking agreement. Possibly each escrow always offers promises $G(d)$ for different values of d, and when sending money to escrow i, Chloe$_i$ simply tags it as taking advantage of promise $G(d_i)$. In this approach, the protocol lacks the transitions labelled $s(c_i, G(d_i))$ and $r(e_i, G(d_i))$, with the initial states shifted accordingly. Still, the promise $G(d_i)$ counts as having been made to customer c_i by escrow e_i.
3. An alternative is to introduce a message "We are ready", sent by Chloe$_{n-1}$ to escrow e_{n-2}, and forwarded, via Chloe$_i$ and escrow e_{i-1} all the way to Alice. Each Chloe$_i$ forwards the "We are ready" message only after receiving promise $G(d_i)$ from escrow e_i, so when Alice receives the "We are ready" message she can safely initiate the transfer.

Note that this problem cannot be solved through a diamond shaped automaton for Customer i, in which the messages $G(d_i)$ and $P(a_{i-1})$ can be received in either order. Namely, when Escrow $i-1$ sends the message $P(a_{i-1})$, a time-out u is set. As soon as a period of time a_{i-1} elapses after this event, and no money has been received from customer c_i, the transaction is cancelled. The constant a_i will be chosen in such a way that Chloe$_i$ has just enough time, after receiving message $P(a_{i-1})$, to send the money. In a diamond-shaped graph, Chloe$_i$ must await message $G(d_i)$ before sending the money, and a priori no upperbound on its arrival can be given. So there is no constant a_i that is large enough to give Chloe$_i$ enough time to reply.

3.5 Correctness of the protocol

Now we show that the protocol from Fig. 3 (taking into account the modifications from Sect. 3.4) is correct, in the sense that it satisfies the properties of Definition 1, when making the assumptions of Sect. 3.1. In doing so, we also calculate the values of the parameters d_i and a_i.

Consistency. To check that the protocol is consistent, in the sense that each participant can abide by it, we first of all invoke the assumption of bounded reaction speed, described in Sect. 3.1, and use that the constant ε assumed to exist in Sect. 3.1 is in fact the time-out value associated to most output states. This ensures that it is always possible to send messages in a timely manner. In particular, the protocol prescribes that when an automaton enters a grey state, it will leave this state, by sending one or two messages, within time ε. The assumption *Bounded Reaction Speed* makes sure that this time is sufficient for sending these messages.

The only remaining potential failure of consistency is when the protocol prescribes the transmission of a resource that it is not available. Assuming that the sending of promises and money is not an obstacle (Chloe has been selected, in part, for having this kind of money available), the only issue could be the sending of the certificate signed by Bob. For anyone but Bob this can only be done after receiving it first. However, a simple inspection of the automata of the escrows, Chloe$_i$ and Alice shows that any transition sending the certificate is preceded by a transition receiving it. This establishes requirement C.

Escrow security. That escrows cannot lose money (requirement ES) follows immediately from the observation that an escrow spends the money only after receiving it. This follows from the order of the transitions in the automaton for the escrows.

Honesty. Although not part of Definition 1, we show that an escrow that issues a promise always keeps that promise, when abiding by the protocol. This property (H) will be be used below to establish CS.

To show H, suppose the escrow e_i issues promise $P(a_i)$, and subsequently receives the certificate χ from customer c_{i+i} at a time $v < u + a_i$, where u refers to the time promise $P(a_i, \varepsilon)$ was issued. Then it is too soon for the time-out transition, so the transition labelled $r(c_{i+1}, \chi)$ in the automaton of e_i will be taken, at time v. The automaton shows that $s(c_{i+1}, \chi)$ will occur by time $v + \varepsilon$, thus fulfilling the promise.

To show that an escrow that issues promise G always keeps it, when abiding by the protocol, suppose the escrow e_i receives the money at a time w. Then the transition labelled $r(c_i, \$)$ in the automaton of e_i will be taken, at time w. The automaton shows that either $s(c_i, \$)$ will occur by time $w + \varepsilon + a_i + \varepsilon$, or $s(c_i, \chi)$ will occur by time $w + \varepsilon + a_i + \varepsilon$. Thus, to guarantee that promise G is met, we need to choose d_i and a_i in such a way that

$$d_i \geq a_i + 2\varepsilon \tag{1}$$

for $i = 0, \ldots n-1$. In fact, making the promise as strong as possible yields $d_i := a_i + 2\varepsilon$. When this condition is met, we have established requirement H.

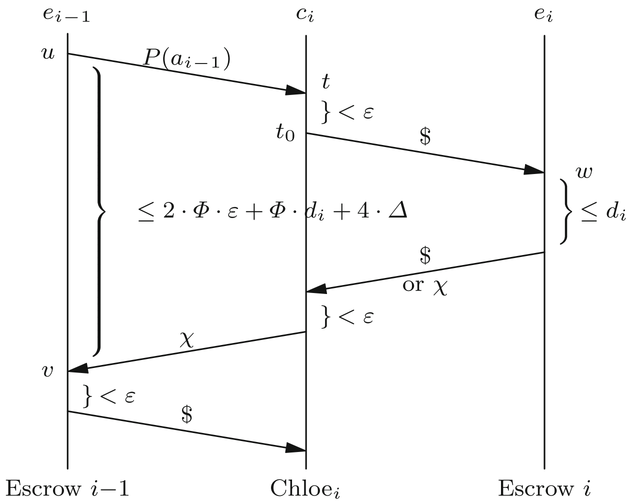
Customer security and time-bounded termination. We will check time-bounded termination (T) together with customer security (CS). To check requirement CS1, suppose that Alice will make the payment $s(e_0, \$)$, at time t. Then earlier she has received promise $G(d_0)$ from escrow e_0. This promise ensures Alice that escrow e_0 will send her either \$ or χ by its local time $w + d_0$, where w is the time Alice's payment is received. Consequently, by the reasoning of Sect. 3.1, using the assumption of bounded synchrony, Alice will receive either certificate χ or her money back by time $t + \Phi \cdot d_0 + 2 \cdot \Delta$.

To check requirement CS2, suppose that Bob issues certificate χ, at time x. Then earlier, at time t, he has received promise $P(a_{n-1})$ from escrow e_{n-1}. Moreover, $x < t + \varepsilon$. For the promise to be meaningful, his certificate needs to arrive at e_{n-1} before time $u + a_{n-1}$, where u refers to the local time at e_{n-1} when the promise was issued. By the reasoning of Sect. 3.1, using the assumption of bounded synchrony, Bob's certificate will arrive at escrow e_{n-1} before time $u + \Phi \cdot \varepsilon + 2 \cdot \Delta$. Hence, we need to choose a_{n-1} in such a way that

$$a_{n-1} \geq \Phi \cdot \varepsilon + 2 \cdot \Delta\ . \tag{2}$$

When this requirement is met, the promise ensures Bob that escrow e_{n-1} will send him the money by its local time $v + \varepsilon$, where v is the time Bob's certificate is received by e_{n-1}. Consequently, Bob will receive payment by time $x + \Phi \cdot \varepsilon + 2 \cdot \Delta$.

To check requirement CS3, suppose that Chloe$_i$ will make the payment $s(e_i, \$)$, at time t_0. Then earlier, she has received promise $G(d_i)$ from escrow e_i and promise $P(a_{i-1})$ from escrow e_{i-1}, the latter at time t. Moreover, $t_0 < t + \varepsilon$. Promise $G(d_i)$ ensures Chloe$_i$ that escrow e_i will send her either \$ or χ by its local time $w + d_i$, where w is the time Chloe$_i$'s payment is received. Consequently, c_i will receive either certificate χ or her money back by time $t_0 + \Phi \cdot d_i + 2 \cdot \Delta$.



Continuing with the case that she receives χ rather then her money back, she will forward χ to escrow e_{i-1} by time $t_0 + \Phi \cdot d_i + 2 \cdot \Delta + \varepsilon$, which is before $t + \varepsilon + \Phi \cdot d_i + 2 \cdot \Delta + \varepsilon$. Hence it arrives at e_{i-1} by its local time $u + 2 \cdot \Phi \cdot \varepsilon + \Phi \cdot d_i + 4 \cdot \Delta$, where u is the time promise $P(a_{i-1})$ was issued. Here we use that Δ is a valid upperbound on transition times by anyone's clock, and that there is no need to square Φ in $\Phi \cdot d_i$, as also the clock skew between escrows e_i and e_{i-1} is bounded by Φ. This calculation is illustrated by the message sequence diagram above. Since χ needs to arrive at e_{i-1} before time $u + a_{i-1}$ in order for promise $P(a_{i-1})$ to be meaningful, we need to pick

$$a_{i-1} \geq 2 \cdot \Phi \cdot \varepsilon + \Phi \cdot d_i + 4 \cdot \Delta \tag{3}$$

for $i = 1, \ldots, n-1$. When (3) holds, promise $P(a_{i-1})$ ensures Chloe$_i$ that escrow e_{i-1} will send her the money by its local time $v + \varepsilon$, where v is the time the certificate is received by e_{i-1}. Consequently, c_i will receive \$ by time $t_0 + \Phi \cdot d_i + 4 \cdot \Delta + \varepsilon + \Phi \cdot \varepsilon$. Hence, assuming (3), CS3 is guaranteed.

Choosing $=$ for $\geq$ in (1)–(3), we ensure requirement CS by solving these equations. In particular, for $i = 0, \ldots, n-1$,

$$a_i := \Phi^{n-1-i} \cdot (\Phi \cdot \varepsilon + 2 \cdot \Delta) + \sum_{j=i+1}^{n-1} 4 \cdot \Phi^{j-i-1} \cdot (\Phi \cdot \varepsilon + \Delta).$$

For $i = n-1$ this follows by (2). Assume we have it for $i+1$. Then

$$a_{i+1} = \Phi^{n-2-i} \cdot (\Phi \cdot \varepsilon + 2 \cdot \Delta) + \sum_{j=i+2}^{n-1} 4 \cdot \Phi^{j-i-2} \cdot (\Phi \cdot \varepsilon + \Delta).$$

Now, applying (1) and (3), multiply by Φ and add $4 \cdot \Phi \cdot \varepsilon + 4 \cdot \Delta$.

When $\Phi > 1$, our solution for a_i can be simplified by applying the formula for a geometric progression:

$$\sum_{j=i+1}^{n-1} \Phi^{j-i-1} = \sum_{k=0}^{n-i-2} \Phi^k = \frac{\Phi^{n-i-1} - 1}{\Phi - 1}.$$

Liveness. It remains to check property L. Suppose that all parties abide by the protocol. By the reasoning in Sect. 3.4 we may assume that the action $s(e_0, \$)@c_o$ (using the terminology of Sect. 3.3) of Alice sending money to her escrow will not take place until all actions $s(c_i, G(d_i))@e_i$ and $r(e_i, G(d_i))@c_i$ have occurred. In terms of Sect. 3.3 we show that in the remaining active phase of the protocol at least the prefix of the displayed sequence of actions until and including $s(c_n, \$)@e_{n-1}$ will take place, in that order, and not interleaved with any other actions. When this happens, Bob must be in his third state, and the required action $r(e_{n-1}, \$)@c_n$ will follow surely.

Towards a contradiction, let the initial behaviour of the active part of the protocol be a strict prefix of this sequence, where a is the first action in the sequence that does not occur as scheduled. A simple case analysis shows that when a is scheduled, in fact no action other than a is possible.

The action a cannot be of the form $s(e_i, \$)@c_i$, because when this action is due, customer c_i is in a state where this action must be taken within time ε. An exception is the case $n = 0$, but also here the action must be taken in within a finite amount of time (or there is nothing to verify).

The action a cannot be $r(c_i, \$)@e_i$ either, because each message sent must arrive eventually, and the receiving party e_i is in its second state, and thus able to perform the receive action.

Similarly, a cannot be $s(c_{i+1}, P(a_i))@e_i$, as here the sender e_i must be in its third state.

Since all actions $r(e_i, G(d_i))@c_i$ have already occurred, a cannot be of the form $r(e_{i-1}, P(a_{i-1}))@c_i$.

The case $a = s(e_{n-1}, \chi)@c_n$ can be excluded, as here Bob must be in his second state.

If $a = r(c_n, \chi)@e_{n-1}$, then the recipient e_{n-1} must be in its fourth state and, due to the careful choice of a_{n_1} (see (2)), the time-out transition cannot intervene. So that choice of a is excluded too.

The argument against $a = s(c_n, \$)@e_{n-1}$ is trivial.

4 Impossibility under partial synchrony of communications

When communications can experience arbitrarily long delays, it is not possible to expect from a protocol to terminate in an a priori known amount of time. If we want to perform cross-chain payments in a partially synchronous setting, it is therefore necessary to define a different class of cross-chain payment protocols. The obvious idea is to remove any time bound in the definition. But as we will show, this is not enough to make the problem solvable.

Definition 2 (*Eventually terminating cross-chain payment protocol with strong liveness guarantees*) A cross-chain payment protocol is an *eventually terminating cross-chain payment protocol with strong liveness guarantees* if it satisfies all the properties of Definition 1 except Property T, which is replaced by:

T′ *Eventual termination.* Each customer that abides by the protocol, and either makes a payment or issues a certificate, terminates *eventually*, provided her escrows abide by the protocol.

Theorem 1 *If communications are partially synchronous, then there is no eventually terminating cross-chain payment protocol with strong liveness guarantees. This even holds if we only allow the participants to either follow the protocol or crash.*

Proof Assume an eventually terminating cross-chain payment protocol with strong liveness guarantees.

Consider a run r in which all participants abide by the protocol. It exists by property C. By property L Bob will be paid in this run, and by property T′ all customers terminate. Since by properties ES and CS no participant makes a loss, Alice will not get her money back. Hence by property CS1 Alice will end up with the certificate χ. Let c_i be the last customer that holds the certificate before it reaches Alice. This must be either Bob or one of the connectors. Let s be the state in which c_i is about to send on χ.

The protocol may not prescribe that c_i has already received the money in state s. For then customer c_i could decide to keep the money as well as the certificate, while all other participants keep abiding by the protocol, which would violate properties T′, ES or CS.

Now consider the following two runs of the system, that are the same until state s. In run r_1 customer c_i never lets go of the certificate, nor sends out any other message past state s, while all other participants abide by the protocol; in run r_2 all participants abide by the protocol, but c_i's message with the certificate, and all subsequent messages from c_i, experience an extreme delay.

First assume that c_i is in fact Bob. By property T′ run r_1 reaches a state s' in which all customers other than Bob are terminated. Using properties ES and CS, in this state Alice ends up without the certificate, and thus with the money, and all customers Chloe$_j$ play even. It follows that Bob never receives his money. Yet for all participants other than Bob, runs r_1 and r_2 are indistinguishable, so r_2 will reach a similar state. This violates property CS2.

Now assume customer c_i is not Bob. So Bob has already issued the certificate. By property T′ run r_1 reaches a state s' in which all customers other than c_i are terminated. Using properties ES and CS, in this state Alice as well as Bob end up with the money, and all customers Chloe$_j$ with $j \neq i$ play even. It follows that Chloe$_i$ loses her money. Yet for all participants other than Chloe$_i$, runs r_1 and r_2 are indistinguishable, and the delayed certificate sent by Chloe$_i$ may arrive only after the system has reached state s'. This violates property CS3. □

5 Solution to a variant under partial synchrony

5.1 Eventually terminating cross-chain payment protocol with weak liveness guarantees

Weak liveness guarantees. As it is impossible to design an eventually terminating cross-chain payment protocol with strong liveness guarantees under partial synchrony, we define a variant with *weak liveness guarantees*, and show it implementable.

In view of the impossibility proof given above, the *strong liveness* condition (L) is too strong. We replace it by a (realistic and still desirable) property called *weak liveness* such that the problem becomes solvable. A similar situation exists in the atomic commit problem literature, for instance *weak non-triviality* defined by Guerraoui [11] or condition $AC4$ for atomic commit defined by Hadzilacos in [12]:

> If all existing failures are repaired and no new failures occur for a sufficiently long period of time, then all processes will reach a decision.

Abort certificate. In the synchronous solution, we used timelocks to ensure that the money will not get stuck forever in escrow. This solution is no longer pertinent under partial synchrony. We need to replace them by a safe way to unlock the funds stored in an escrow.

We therefore modify the definition of the certificate χ. Instead of having a single certificate simply signed by Bob, we have two more general certificates called *commit certificate* χ_c and *abort certificate* χ_a, that can never exist simultaneously.

New definition of the problem. When we take into account the two previous tweaks, we reach the following definition. We highlight in italics the difference with our previous definitions of cross-chain payment protocols. In particular, we replace "Bob will not issue χ" by "Bob will receive χ_a", and "Alice will receive χ" by "Alice will receive χ_c".

Definition 3 (*Eventually terminating cross-chain payment protocol with weak liveness guarantees*) A cross-chain payment protocol is an *eventually terminating cross-chain payment protocol with weak liveness guarantees* if it satisfies the following properties:

C *Consistency.* For each participant it is possible to abide by the protocol.
CC *Certificate consistency. An abort and a commit certificate cannot be issued both.*
T′ *Eventual termination.* Each customer that abides by the protocol terminates eventually, provided her escrows abide by the protocol.
ES *Escrow security.* Each escrow that abides by the protocol does not lose money.
CS′ *Customer security.*

CS1′ Upon termination, if Alice and her escrow abide by the protocol, Alice has either got her money back or received the *commit certificate* χ_c.
CS2′ Upon termination, if Bob and his escrow abide by the protocol, Bob has either received the money *or the abort certificate* χ_a.
CS3′ Upon termination, each connector that abides by the protocol has got her money back, provided her escrows abide by the protocol.

L′ *Weak liveness.* If all parties abide by the protocol, *and if the customers wait sufficiently long before and after sending money*, then Bob is eventually paid.

5.2 Transaction manager abstraction

The previous impossibility result shows that when there is no synchrony, it is hard for processes to agree on a uniform commitment decision (abort or commit). We are going to leverage the existing solutions to the classical consensus problem, and embed them in an abstraction called "transaction manager", defined as follows:

Definition 4 (*Transaction manager*) A *transaction manager*, called TM, is a process that can receive binary values from the set {COMMIT, ABORT} from a customer, and that can send certificates drawn from $\{\chi_c, \chi_a\}$ to customers. It must satisfy:

– *TM-Consistency*. TM does not issue two different certificates.
– *TM-Termination*. If a customer proposes to ABORT or if Bob proposes to COMMIT, then TM sends eventually a certificate χ_c or χ_a to every customer.
If a customer proposes ABORT or COMMIT after TM has issued a certificate, TM will send a copy of that certificate to that customer.
– *TM-Commit-Validity.* TM can issue χ_c only if Bob proposed COMMIT.
– *TM-Abort-Validity.* TM can issue χ_a only if some customer proposed ABORT.

There are several ways of implementing a transaction manager.

– *Centralised transaction manager.* The transaction manager can be a centralised actor trusted by every customer.
– *Distributed transaction manager.* The transaction manager could be a collection of k parties ("validators") appointed by the participants in the protocol. These validators could run the consensus algorithm for partial synchrony from Dwork, Lynch and Stockmeyer [6], or any equivalent algorithm. This works when less than one third of these validators are unreliable. In this case, "sending a message to the TM" means sending it to each of these validators, and "the TM sending a decision" means strictly more than one third of the validators sending the jointly taken decision.
– *External decentralised transaction manager.* The transaction manager can be a decentralised data structure. For example, a smart contract running on a permissionless blockchain shared by every customer can be programmed to be a transaction manager.

In the following, we assume that we have such a transaction manager. Even if it is run by the customers, we do not specify the messages exchanged to run it: the transaction manager is a black-box embedding a consensus algorithm that is running off-protocol. Section 5.5 provides an example implementation.

5.3 A protocol responding to the problem

Description of the protocol. In the version of the protocol depicted in Figs. 4, 5, 6, 7, Chloe$_i$ awaits the two promises of her escrows, just like in the protocol of Fig. 3, and then sends the money to her downstream escrow. Subsequently she awaits an abort or commit certificate from the transaction manager. If she gets a commit certificate, she cashes it in at her upstream escrow to obtain the money. If she gets an abort certificate instead, she cashes it in at her downstream escrow for a refund of the money she paid earlier. In case she loses patience before she gets the second promise, which happens at a time T_i specific for Chloe$_i$, she simply quits. In case she loses patience after she has invested the money but before she gets any certificate, the time-out transition occurs, and she sends an abort proposal to the transaction manager. The latter replies on this with either an abort or a commit certificate, and she cashes those in as above.

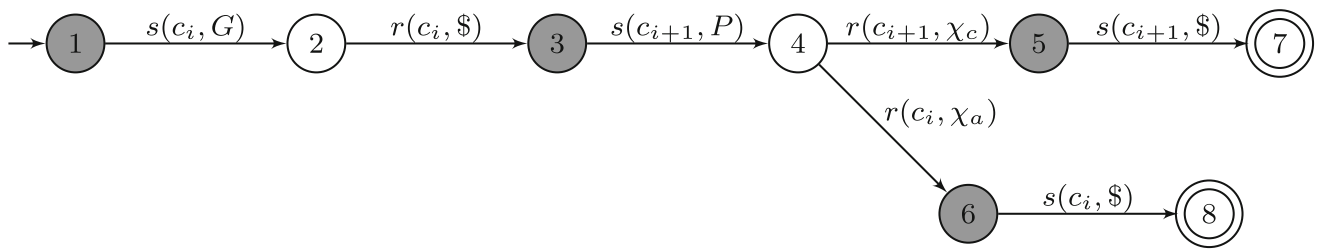


Fig. 4 Automaton for escrow e_i

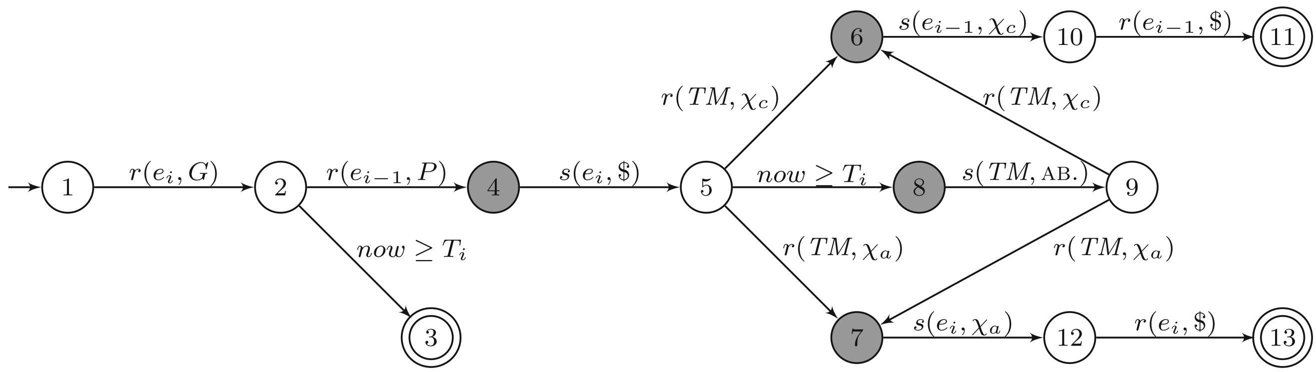


Fig. 5 Automaton for customer c_i, $i \in \{1..n-1\}$

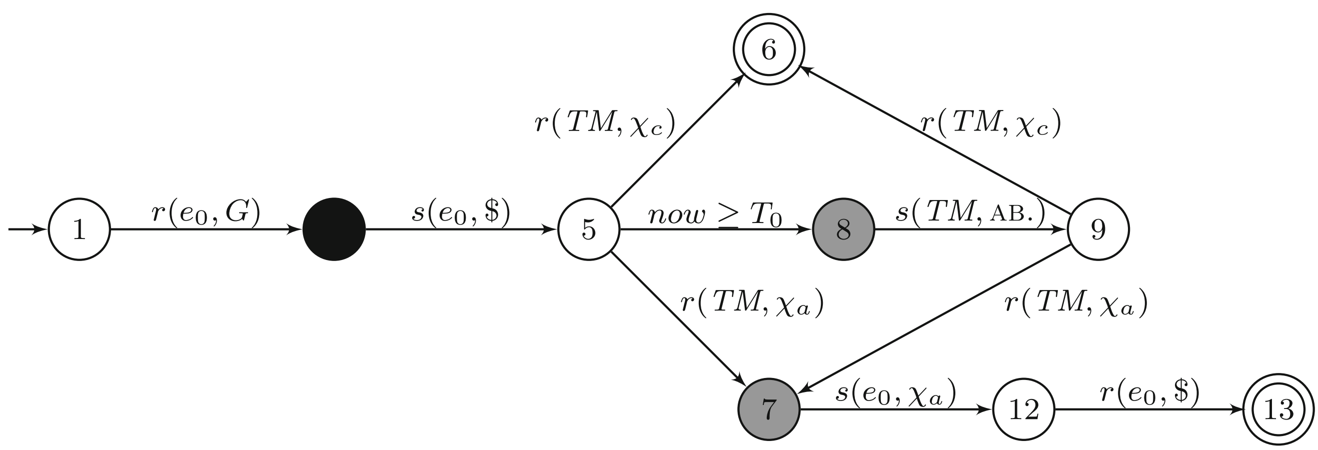


Fig. 6 Automaton for customer c_0 (Alice)

So the tags G and P can be understood as promises that say:

G: "I guarantee that if I receive \$ from you, then if you send me χ_a I will send you \$".
P: "I promise that if you send me χ_c I will send \$ to you".

The automaton for Alice is just a simplified version of the one for Chloe, and the one for the escrows is trivial. For Bob, the important modification is that he alerts the TM with a COMMIT message when the protocol is ready for this. Moreover, in case Bob loses patience before receiving any promise, he sends an ABORT message to the TM, so that he receives the abort certificate in response.

5.4 Proof of correctness

Let us call P the protocol defined by the above ANTA. Section 3.4 (Initialisation) applies to P as well, and we assume that the appropriate modifications are made.

Theorem 2 *Protocol P is an eventually terminating cross-chain payment protocol with weak liveness guarantees.*

Proof The properties to prove are close to the time-bounded cross-chain payment protocol, and the proof is similar. We

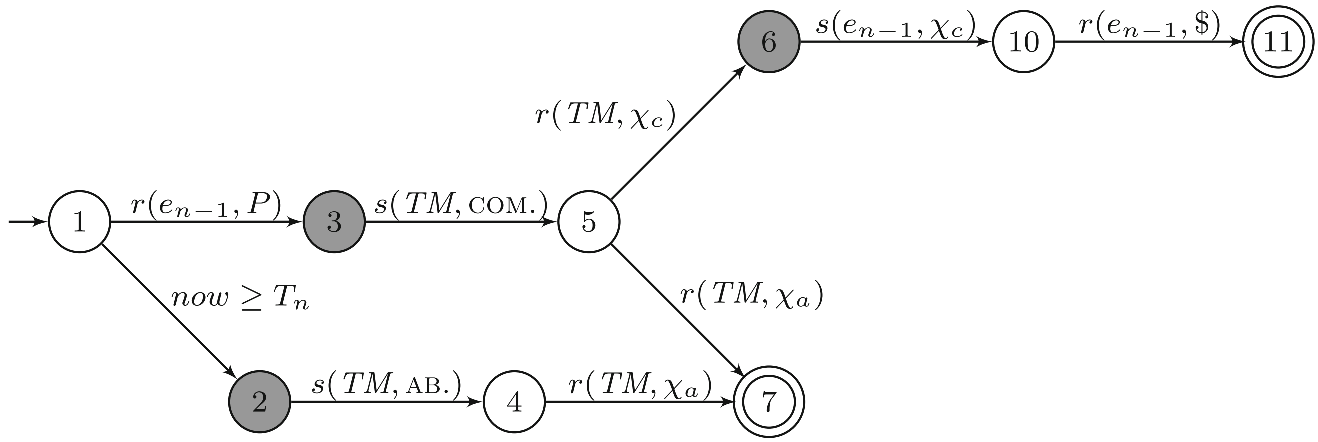


Fig. 7 Automaton for customer c_n (Bob)

split the proof in the following lemmas: Lemma 1, 2, 3, 4, 5 and 6. ◻

Lemma 1 (Consistency) *For each participant in P it is possible to abide by P.*

Proof We have to ensure that each participant will be able to follow the transitions after a grey or black state. The only way this would not be possible would be when the protocol asks to transmit a resource that is not available. It is always possible to send tags and money, but we need to verify that any sending of a certificate is preceded by the receipt of this certificate—except for the issuer of the certificate. Such a property is clear after inspection of the automata of escrows and customers. ◻

Lemma 2 (Certificate consistency) *An abort and a commit certificate can never be issued both.*

Proof This is an immediate consequence of Property *TM-Consistency* of Definition 4. ◻

Lemma 3 (Eventual termination) *Each customer that abides by P will terminate eventually, provided her escrows abide by the protocol.*

Proof Thanks to Lemma 1, in order to show that a terminating state will be reached, it is sufficient to prove that every input state will be left eventually. We thus establish Lemma 3 by showing that Chloe$_i$ and Alice cannot get stuck in states 1, 2, 5, 9, 10 and 12, and Bob cannot get stuck in states 1, 4, 5 and 10.

Since her downstream escrow will surely leave state 1, and thus send the message G, it follows that Chloe$_i$, and Alice, will receive this message, and thereby leave state 1.

Chloe$_i$ will leave state 2 at time T_i at the latest, or immediately when reaching this state after time T_i. By the same reasoning, Bob will not get stuck in state 1.

Chloe$_i$ and Alice will leave state 5 at time T_i at the latest, or immediately when reaching this state after time T_i. She will send an ABORT message to TM to reach state 9, and by the property *TM-Termination* of Definition 4, TM will eventually reply to her with χ_a or χ_a. Hence she will leave state 9. By the same reasoning, Bob will not get stuck in state 5.

To reach state 4, Bob sends an ABORT message to TM. By the properties *TM-Termination* and *TM-Commit-Validity* of Definition 4, TM will eventually reply to him with χ_a. Hence he will leave state 4.

Now assume Customer c_{i+1} (Chloe or Bob) reaches state 10. Then Customer c_{i+1} has already received the promise P from Escrow i, and thus Escrow i must have send this promise, thereby reaching state 4. To reach state 10, Customer c_{i+1} sends certificate χ_c to Escrow i. By Lemma 2, the TM never issues certificate χ_a, so Customer c_i cannot send it to Escrow i. It follows that Escrow i will reach state 5, and send the money to Customer c_{i+1}. Hence Customer c_{i+1} will leave state 10 and reach the terminating state 11.

Finally assume Customer c_i (Alice or Chloe) reaches state 12. Then Customer c_i has already received promise G from Escrow i, and thus Escrow i must have send this promise, thereby reaching state 2. After receiving promise G, Customer c_i has send the money to Escrow i, so Escrow i will have reached state 3, and hence also state 4. To reach state 12, Customer c_i sends certificate χ_a to Escrow i. By Lemma 2, the TM never issues certificate χ_c, so Customer c_{i+1} cannot send it to Escrow i. It follows that Escrow i will reach state 6, and send the money to Customer c_i. Hence Customer c_i will leave state 12 and reach the terminating state 13. ◻

Lemma 4 (Escrow-security) *Any escrow that abides by P will not lose money.*

Proof The result is immediate: any transition where an escrow sends money has been preceded by a transition where it receives the money. An escrow's balance cannot become negative if it follows the protocol. □

Lemma 5 (Customer-security)

1. *Upon termination, if Alice and her escrow abide by P, Alice has either got her money back or received the commit certificate χ_c.*
2. *Upon termination, if Bob and his escrow abide by P, Bob has either received the money or the abort certificate χ_a.*
3. *Upon termination, each connector that abides by P has got her money back, provided her escrows abide by P.*

Proof

1. If Alice terminates in state 13, she has got her money back in the last transition. If she terminates in state 6, she has got the certificate χ_c in the last transition.
2. The result is similar for Bob: if he terminates in state 7, he has received χ_a. Otherwise, he terminates in state 11 and has been paid correctly.
3. To reach termination, Chloe$_i$, $i \in \{1..n-1\}$ has either never spend the money (termination state 3), or received \$ either from e_{i-1} (termination state 11) or from e_i (termination state 13). In both these cases, she has got her money back. □

Lemma 6 (Weak liveness) *If all participants abide by P, and if the customers wait sufficiently long before and after sending money, then Bob will be paid.*

Proof Let us suppose that all the participants abide by P. Suppose that for all $i \in [0, n-1]$, T_i is large enough for Alice, Bob and every connector to never take any time-out transition. Instead, Bob will be the first customer to call the transaction manager in his transition from state 3 to 5.

Using the notation of Sect. 3.3, the active part of the protocol—after the exchange of the tags G—must start with the following sequence of actions, executed in this order:

$$
\begin{array}{llll}
s(e_0,\$)@c_0 & r(c_0,\$)@e_0 & s(c_1,P)@e_0 & r(e_0,P)@c_1 \\
s(e_1,\$)@c_1 & r(c_1,\$)@e_1 & s(c_2,P)@e_1 & r(e_1,P)@c_2 \\
\ldots & & & \\
s(e_k,\$)@c_k & r(c_k,\$)@e_k & s(c_n,P)@e_k & r(e_k,P)@c_n \\
 & & & s(TM,\text{COM.})@c_n
\end{array}
$$

By the *TM-Abort-Validity* property of Definition 4, since the only proposal was COMMIT, TM will issue the certificate χ_c. In particular, Bob will give it to e_{n-1} and receive the payment in exchange. □

Interestingly, nothing in the proof depends in any way on the assumption of partially synchronous communication. The only place where this is needed is for the implementation of the transaction manager TM—see Sect. 5.5. In case one is content with a centralised transaction manager as described in Sect. 5.2, our protocol works correctly also when assuming communication to be asynchronous.

5.5 Implementation of a decentralised transaction manager

As an example, in this section we provide an explicit implementation of a transaction manager. It is a wrapper, expressed in pseudo-code, around a binary Byzantine consensus algorithm, such as the one from [6], which is treated as a black box.

Suppose that we have *m validators*, which are agents running a consensus algorithm. The validators communicate which each other by exchanging messages. We suppose that a certain number f of validators can be *faulty*, in the sense that we allow arbitrary (Byzantine) behaviour. All other validators are assumed to abide by the protocol defining a consensus algorithm.

Definition 5 (*Binary Byzantine Consensus*) A binary Byzantine consensus (BBC) algorithm is an algorithm in which every validator can *propose* a binary value (i.e. in {0, 1}) and *decide* a binary value. Assuming that every non-faulty validator proposes a binary value, the following properties must be ensured:

- *BBC-Termination.* Each non-faulty process eventually decides on a binary value.
- *BBC-Agreement.* No two non-faulty processes decide on different binary values.
- *BBC-Validity.* If all non-faulty processes propose the same value, no other value can be decided by a non-faulty process.

Theorem 3 ([6]) *Assuming partially synchronous communication, a binary Byzantine consensus algorithm exists when $f < m/3$.*

Using such an algorithm as a black box, we now implement a TM. Our validators can either be customers, like Alice, Chloe and Bob, or external parties. If the set of validators is included in or equal to the set of customers, we can talk of an *internal decentralised transaction manager*. Our TM implementation is only valid when assuming partially synchronous communication, and $f < m/3$. Each customer can communicate with every validator.

Reliable broadcast call. To call the transaction manager, a customer reliably broadcasts a message to all validators. Here a *reliable broadcast* is a protocol described by Bracha in [4]. It is guaranteed to terminate, even in a setting with asynchronous communication, provided less than one-third of all the broadcast recipients is faulty—the rest abiding by the protocol. It guarantees that if the sender abides by the protocol, all recipients will receive the message sent. Moreover, even if the sender is faulty, either all correct recipients agree on the same value sent, or none of them accepts any value as having been sent [4].

If a faulty customer sends a call with different values to different validators, or sends something to some validators and nothing to others, the reliable broadcast primitive will filter these messages out. In particular, if a validator receives a call from a customer then eventually every validator will receive this call from this customer.

Certificate implementation. With $\sigma_k(v)$ we denote the value $v \in \{0, 1\}$ cryptographically signed by validator k. Such a signed message models the decision ABORT (if $v = 0$) or COMMIT ($v = 1$) taken by validator k. Since up to f validators may be unreliable, a valid certificate is a message that contains (for instance as attachments) more than f copies of the same decision, taken by different validators k. We model such certificates as sets. Hence:

- χ_c is any set of at least $f+1$ messages $\sigma_k(1)$ signed by at least $f+1$ different validators k.
- χ_a is any set of at least $f+1$ messages $\sigma_k(0)$ signed by at least $f+1$ different validators k.

Such a certificate is verifiable non-interactively by a third party such as any customer or escrow. Asking for $f+1$ signatures ensures that at least one correct validator has issued this certificate, and in particular will guarantee property CC (certificate consistency). Of course a real implementation will not rely on simple signatures of the string "0" or "1" because of the possibility of replay attacks.

TM implementation. When our protocol prescribes the action $s(TM, \text{AB.})$, resp. $s(TM, \text{COM.})$, this is implemented as reliably broadcasting ABORT, resp. COMMIT, to all validators. The transition $r(TM, \chi)$ denotes the receipt of certificate χ from one of the validators. The behaviour of TM is described as Algorithm 1.

Algorithm 1 *BFT-TM* algorithm for validator v_k, $k \in [1..m]$.

1: When validator k has not proposed, nor decided, a value so far:
2:     when receive ABORT from a customer c_i, $i \in [0..n]$:
3:         propose(0) to BBC
4:     when receive COMMIT from c_n:
5:         propose(1) to BBC

6: When validator k decides value v (by running BBC):
7:     broadcast($\sigma_i(v)$) to all validators
8:     await receipt of $\sigma_j(v)$ for all j in a certain $J \subseteq [1..m]$ such that $|J| > f$
9:     $\chi := \{\sigma_j(v),\ j \in J\}$
10:     broadcast(χ) to all customers

11: When validator k has decided a value v:
12:     when receive ABORT or COMMIT from cust. c_i, $i \in [0..n]$:
13:         send χ to customer c_i.

Theorem 4 (BFT-TM correctness) *The BFT-TM algorithm implements a TM as defined in Definition 4.*

Proof The correctness properties of a transaction manager derive almost immediately from the correctness properties of a binary Byzantine consensus algorithm.

1. *TM-Termination.* Let us suppose that a customer sends an abort proposal to TM, i.e. to each and every validator, or that Bob proposes to commit. Then every correct validator starts participating in BBC with the initially proposed binary value 0 or 1, respectively. By *BBC-Termination*, every correct validator eventually passes line 6 of the BFT-TM algorithm. By assumption on the number of correct validators, every validator eventually receives at least $f+1$ signatures on this value, thereby forming a certificate that is sent to every customer.
   If a customer proposes ABORT or COMMIT after TM has issued a certificate, each correct validator will send a copy of that certificate to that customer by lines 11–13 of the algorithm.
2. *TM-Consistency.* By contradiction, let us suppose that two customers receive different certificates. As a certificate contains at least $f+1$ signatures, a correct validator has broadcast $\sigma_i(0)$ and another correct validator has broadcast $\sigma_j(1)$. Line 6 of BFT-TM shows that the value signed and broadcast has been decided by BBC. This is a contradiction with *BBC-Agreement*.
3. *TM-Commit-Validity.* TM-Commit-Validity says that if Bob does not propose commit, then TM cannot issue χ_c. If Bob does not propose commit, then no correct validator will ever propose 1 because of lines $1-5$ of BFT-TM. Now *BBC-Validity* implies that no correct validator can decide 1, and consequently the commit certificate cannot be issued by TM.
4. *TM-Abort-Validity.* TM-Abort-Validity says that if no customer proposes to abort, then TM cannot issue χ_a. If no customer proposes to abort, then no correct validator will ever propose 0 because of lines $1-5$ of BFT-TM. The final argument is the same as above. □

6 Related work

6.1 The interledger protocols

In [24] two protocols are presented for payments across payment systems. Here a *payment system* is thought of as an independent bank, where people can have accounts. The intended application is in digital payment systems, such as Bitcoin [21]. In [24] the payment systems, or rather the offered functionality, are loosely referred to as *ledgers*, and payments between customers of different ledgers as *interledger payments*. Following popular terminology, we here speak of *escrows* and *cross-chain payments*.

The protocols from [24] generalise to the situation of a longer zigzag than the one presented in Fig. 1, involving n escrows and $n-1$ intermediaries Chloe$_i$. As mentioned, the correctness proof in [24] for the universal protocol requires the assumption of synchrony from Dwork et al. [6]. Here we point out that this assumption is necessary. To make such a statement, we need to define when we consider a cross-chain payment protocol correct. The correctness proof by Thomas and Schwartz [24] consists of reasonable properties that are shown to hold for the chosen protocol. These properties are stated in terms of that specific protocol, and the reader can infer that they sum up to a correctness argument. But no formal statement occurs of what it means for a general cross-chain payment protocol to be correct, and this is what we need to make any negative statement about it.

6.2 Cross-chain swaps

A cross-chain swap is a deal where a transaction from Alice to Bob in one blockchain is matched by a transaction from Bob to Alice in another. Herlihy proposes atomic cross-chain swaps in a synchronous environment [13]. Zakhary et al. [26] adapt this to work even with asynchronous communication. Ron van der Meyden [19] verifies a cross-chain swap protocol by modelling a timelock predicate as a Boolean variable indicating whether the asset is transferred. This approach also requires synchrony. In XClaim [28] Zamyatin et al. propose a solution to swap blockchain-backed assets. Their protocol assumes that adversaries are behaving rationally, and requires synchrony. Atomic swaps cannot be used to solve the cross-chain payment problem—compare Sect. 6.5.

6.3 Other cross-chain technologies

In the lightning network, Poon and Dryja [22] can relay payments outside the blockchain or "offchain" through connected intermediaries, but they require synchrony and do not consider clock drift. Avarikioti et al. [2] propose an off-chain payment protocol that is safe under asynchrony, between two parties only. Gazi et al. [8,15] propose a rigorous formalisation of ledger and cross-chain transfers, and focus on proof-of-stake and proof-of-work blockchains. However, their results are not extensible to partial synchrony and do not consider clock drift.

Zamyatin et al. [27] define a similar cross-chain communication problem, involving the execution of two well-formed transactions on distinct ledgers before respective time bounds t and t'. They show that the problem is unsolvable in the asynchronous setting without a trusted third party, by reduction from the fair exchange impossibility result, itself derived from the impossibility of consensus in an asynchronous setting with one crash failure. Our result directly implies that also this type of cross-chain communication is unsolvable in the asynchronous setting. The difference is that we show that it is not even possible when partial synchrony is assumed.

Lind et al. [17] relax the synchrony assumption but require a trusted execution environment (TEE). Such a solution cannot be used to solve our problem as it would require trusting a third-party, often represented as the manufacturer of this TEE.

Other approaches [14,23,25] rely on a separate blockchain that plays the same role as our transaction manager (cf. Sect. 5.2). However, [25] and [23] do not aim at ensuring liveness, and [14] aims at ensuring liveness only in periods where communication proceeds synchronously. Wood [25] proposes a multi-chain solution that aims at combining heterogeneous blockchains together without trust. As far as we know, it has not been proved that the protocol terminates. Ranchal-Pedrosa and Gramoli [23] relax the synchrony assumption using an alternative 'child' blockchain to the so-called 'parent' blockchain in order to execute a series of transfers outside the parent blockchain. This protocol does not guarantee that the intermediary transfers on the child blockchain eventually take effect. Herlihy et al. [14] model cross-chain deals as a matrix M where $M_{i,j}$ characterises a transfer between participants i and j, and offer a timelock-based solution under the synchrony assumption, without clock drift, and a certified blockchain protocol that requires partial synchrony. As remarked in [14], a strong liveness guarantee is not feasible when merely assuming partial synchrony. In this context the targeted cross-chain deal problem admits solutions where all correct processes simply abort. Our corresponding problem differs by requiring a weaker liveness guarantee in that it formulates conditions under which a successful transfer is ensured. We present a more detailed comparison between our work and that of [14] in Sect. 6.5.

6.4 Crash fault tolerant solutions

The transaction commit problem is a classical problem from the database literature, tackled for instance by Gray and Lamport [10], Guerraoui [11] and Hadzilacos [12]. It consists of ensuring that either all the processes commit a given transaction or all the processes abort this transaction.

The Non-Blocking Atomic Commitment problem has been formally defined by Guerraoui, for asynchronous systems with unreliable failure detectors (encapsulating partial synchrony) in a crash (fail-stop) model [11] but not in a Byzantine model. Guerraoui proved an impossibility result when the problem requires that "If all participants vote *yes*, and there is no failure, then every correct participant eventually decides *commit*". He then proposes a weaker variant of the problem where the above requirement is replaced by "If all participants vote *yes*, and no participant is ever suspected, then every correct participant eventually decides *commit*". The two problems we consider in this paper present a similar distinction; however, our problems target Byzantine fault tolerance, not crash fault tolerance.

Anta et al. [7] propose a general and rigorous definition of a ledger in which they consider the atomic append problem in an asynchronous model. Similarly, their model considers only crash failures but was later generalised to the Byzantine fault tolerant setting [5] by limiting the number of Byzantine clients to $\lceil n/3 \rceil - 1$.

6.5 Cross-chain deals versus cross-chain payments

In Herlihy et al. [14], a *cross-chain deal* is given by a matrix M where $M_{i,j}$ is listing an asset to be transferred from party i to party j. It can also be represented as a directed graph, where each vertex represents a party, and each arc a transfer; there is an arc from i to j labelled v iff $M_{i,j} = v$ and $v \neq 0$.

They present two protocols for implementing such a deal, while aiming to ensure:

- *Safety.* In each protocol execution, every compliant party ends up with an acceptable payoff.
- *Termination.*[3] No asset belonging to a compliant party is escrowed forever.
- *Strong liveness.* If all parties are compliant and willing to accept their proposed payoffs, then all transfers happen.

Here a payoff is *acceptable* to a party i in the deal if party i either receives all assets $M_{j,i}$ while giving all assets $M_{i,j}$, or if party i loses nothing at all; moreover, any outcome where she loses less and/or gains more than an acceptable outcome is also acceptable.

Each entry $M_{i,j}$ contains a type of asset and a magnitude—for instance "5 bitcoins". For each type of asset a separate blockchain is assumed to act as an escrow. The programming of these blockchains is assumed to be open source, so that all parties can convince themselves that all escrows abide by the protocol. With this in mind, their *Termination* requirement corresponds with our *Eventual termination* of Definition 3, while *Safety* is the counterpart of our *Customer security*. Our requirement of *Escrow security* is left implicit in [14]; since blockchains do not possess any assets to start with, they surely cannot lose them. Finally, their *Strong liveness* property is the counterpart of ours.

Herlihy et al. [14] offer a timelock commit protocol that requires synchrony, and assures all three of the above correctness properties. They also offer a certified blockchain protocol that requires partial synchrony and a certified blockchain, and ensures *Safety* and *Termination*; in a partially synchronous environment no protocol can offer *Strong liveness*. For both protocols the correctness is proven for so-called *well-formed* cross-chain deals: those whose associated directed graph is strongly connected.

The cross-chain payment cannot be seen as a special kind of cross-chain deal. In first approximation, a cross-chain payment looks like a non-well-formed deal of the form

$$\begin{bmatrix} 0 & \$ & & & & \\ & 0 & \$ & & (0) & \\ & & 0 & \ddots & & \\ & & & \ddots & \$ & \\ & (0) & & & 0 & \$ \\ & & & & & 0 \end{bmatrix} \cong c_0 \xrightarrow{\$} c_1 \xrightarrow{\$} \dots \xrightarrow{\$} c_n.$$

However, this representation abstracts from the certificate χ that plays an essential role in the statement of the time-bounded cross-chain payment problem. Factoring in χ, an alternative representation would be

$$\begin{bmatrix} 0 & \$ & & & & \\ & 0 & \$ & & (0) & \\ & & 0 & \ddots & & \\ & & & \ddots & \$ & \\ & (0) & & & 0 & \$ \\ \chi & & & & & 0 \end{bmatrix}$$

or

$$\begin{bmatrix} 0 & \$ & & & & \\ \chi & 0 & \$ & & (0) & \\ & \chi & 0 & \ddots & & \\ & & & \ddots & \$ & \\ & (0) & & \chi & 0 & \$ \\ & & & & \chi & 0 \end{bmatrix}.$$

However, these solutions presume a shared blockchain between Alice and Bob for the transfer of the certificate;

[3] In [14], this property is called "weak liveness". We rename it here, to avoid confusion with our own weak liveness property, which is of a very different nature.

this runs counter to the problem description of cross-chain payments.

Conversely, neither is there a reduction from the cross-chain deal problem to the cross-chain payment problem. A deal presented by a cyclic graph can be represented as a cross-chain payment where Alice and Bob are identified. For instance, an atomic swap between two customers A and C can be expressed as a cross-chain payment with three customers:

$$\begin{bmatrix} 0 & a \\ b & 0 \end{bmatrix} \quad \cong \quad A \xrightarrow{a} C \xrightarrow{b} B = A.$$

However, this idea does not generalise to well-formed cross-chain deals in general. Since every strongly connected graph can be represented as a single cycle with repeated elements, there is an obvious candidate reduction of such deals to cross-chain payments, simply by identifying suitable intermediaries Chloe_i and Chloe_j. However, this reduction does not preserve the safety property of cross-chain deals; for when the deal goes through for Chloe_j but is aborted for Chloe_i, the resulting outcome is not (necessarily) acceptable for the unified participant $\text{Chloe}_{\{i,j\}}$.

7 Conclusion

We formalised the problem of cross-chain payment with success guarantees. We show that there is no solution to the existing variant of this problem without assuming synchrony, and offer a synchronous solution—one that works even in the presence of clock drift. We then relax the liveness guarantee of this problem in order to propose a solution that works in a partially synchronous setting. This new problem differs from existing ones in that it prevents all participants from always aborting, hence guaranteeing success when possible. Besides the new problem statements and our impossibility result, an interesting aspect of our work is to relate recent blockchain problems, like interledger payments, to the classic problem of transaction commit, and to offer Byzantine fault tolerant solutions to these.

Acknowledgements This research is supported under the Australian Research Council Discovery Projects funding scheme (number 180104030) entitled "Taipan: A Blockchain with Democratic Consensus and Validated Contracts" as well as Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled "The Red Belly Blockchain: A Scalable Blockchain for Internet of Things".

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

A The syntax and semantics of ANTA

Let $\mathcal{U}$ be a fixed finite set of *clock variables*. An *arithmetical expression* is a term build from clock variables $u \in \mathcal{U}$ and the constants 0 and 1 by means of the binary operators addition, subtraction, multiplication and division. A *time-out expression* has the form $\mathit{now} \geq \varphi$, with φ an arithmetical expression.

Let $\mathcal{D}$ be a finite set of *automaton identifiers*, and $\mathtt{MSG}$ a fixed set of *messages*. The messages are not defined here; they are a parameter of the ANTA formalism, to be chosen for each application. An *input expression* has the form $r(id, m)$, with $id \in \mathcal{D}$ and $m \in \mathtt{MSG}$. It models the receipt of message m from automaton id. Likewise, an *output expression* has the form $s(id, m)$. It models the sending of m to id. Let E_U, E_I and E_O be the sets of time-out, input and output expressions.

A $\mathcal{D}$*-automaton* is a tuple $(I, O, F, i, T_U, T_I, T_O, \mathcal{V})$ with

- I, O and F disjoint sets of *input*, *output* and *final states*,
- $i \in S := I \cup O \cup F$, the *initial state*,
- $T_U \subseteq I \times E_U \times \mathscr{P}(\mathcal{U}) \times S$, the *time-out transitions*,
- $T_I \subseteq I \times E_I \times \mathscr{P}(\mathcal{U}) \times S$, the *input transitions*,
- $T_O : O \to \mathbb{R}^\infty \times \mathscr{P}^+_{\text{fin}}(E_O) \times \mathscr{P}(\mathcal{U}) \times S$, the *output transitions*,
- and $\mathcal{V} \subseteq \mathcal{U}$ a set of *customer-initialised* clock variables.

The sets I, O, F, T_U and T_I are required to be finite. States are depicted as circles. Final (or *termination*) states are double circles, and output states are shaded. The initial state is marked by a short incoming arrow. A time-out or input transition (s, e, U, s') is depicted as an arrow from $s \in I$ to $s' \in S$, labelled with the expression $e \in E_U \cup E_I$ and with the assignments $u := \mathit{now}$ for $u \in U \subseteq \mathcal{U}$. An output transition $(s, (\mathit{to}, \mathcal{E}, U, s'))$ is depicted as an arrow from $s \in O$ to $s' \in S$, labelled with the finite nonempty set of output expressions $\mathcal{E} \in \mathscr{P}^+_{\text{fin}}(E_O)$ and with the assignments $u{:=}\mathit{now}$ for $u \in U$; moreover, the output state s is labelled with the time-out value $\mathit{to} \in \mathbb{R}^\infty := \mathbb{R} \cup \{\infty\}$.

An *Asynchronous Networks of Timed Automata (ANTA)* is a function $\mathcal{A}$ from a finite set $\mathcal{D}$ of automaton identifiers to the class of $\mathcal{D}$-automata. For each $d \in \mathcal{D}$, let $\mathcal{A}(d)$ be the tuple $(I^d, O^d, F^d, i^d, T^d_U, T^d_I, T^d_O, \mathcal{V}^d)$.

Semantics of individual $\mathcal{D}$-automata

A *valuation* $\xi : \mathcal{U} \rightharpoonup \mathbb{R}$ is a partial function that associates real numbers to some of the clock variables. If $U \subseteq \mathcal{U}$ is a set of clock variables that are set at time *now*, then $\xi[U]^{now}$ denotes the valuation with $\mathrm{dom}(\xi[U]^{now}) = \mathrm{dom}(\xi) \cup U$, defined by $\xi[U]^{now}(u) = now$ for $u \in U$, and $\xi[U]^{now}(u) = \xi(u)$ for $u \in \mathrm{dom}(\xi) \backslash U$.

The *evaluation* $[\![\varphi]\!](\xi) \in \mathbb{R}^{\infty}$ of an arithmetical expression φ under a valuation ξ is the real number obtained by applying the arithmetical operators of φ after filling in the values $\xi(u) \in \mathbb{R}$ for the clock variables $u \in \mathcal{U}$ occurring in φ. In case φ contains clock variables that are not in the domain of ξ, or in case of division by 0, $[\![\varphi]\!](\xi) := \infty$.

A *configuration* of a $\mathcal{D}$-automaton $\mathcal{A}(d)$ fully describes the state of $\mathcal{A}(d)$ at some point. It is a tuple (ξ, now, s, pt, dl) with $\xi : \mathcal{U} \rightharpoonup \mathbb{R}$ a *valuation*, $now \in \mathbb{R}$ the local time at $\mathcal{A}(d)$ in this configuration, $s \in S$ the current state of $\mathcal{A}(d)$, $pt \in \mathscr{P}^{+}_{\mathrm{fin}}(E_O)$ a set of pending transmissions, and $dl \in \mathbb{R}^{\infty}$ a deadline by which the automaton must have left that state.

Given a triple (ξ, now, s), we define a corresponding set of pending transmissions $pt := Pt(\xi, now, s)$ as follows. If $s \in I \cup F$ then $Pt(\xi, now, s) := \emptyset$, and if $s \in O$ with $T_O(s) = (to, \mathcal{E}, U, s')$ then $Pt(\xi, now, s) := \mathcal{E} \in \mathscr{P}^{+}_{\mathrm{fin}}(E_O)$.

Given a triple (ξ, now, s), we define a corresponding deadline $dl := Dl(\xi, now, s)$ as follows. In case $s \in F$ we take $Dl(\xi, now, s) := \infty$. This says that once an automaton enters a final state, it may (and will) stay there forever. In case $s \in O$ and $T_O(s) = (to, \mathcal{E}, U, s')$ then $Dl(\xi, now, s) := now + to$. This says that when an automaton enters an output state, it will stay there less than the value *to* that labels this state. Finally, if $s \in I$ then $Dl(\xi, now, s)$ is defined as the minimum of all values $[\![\varphi]\!](\xi)$, for time-out transitions $(s, now \geq \varphi, U, s')$ leaving state s. Here the minimum of the empty set is ∞. This says that an automaton will not linger in an input state when one of its time-out transitions is enabled. Here time-out transitions with undefined parts are not enabled.

The behaviour of $\mathcal{A}(d)$ is described by defining its possible initial configurations as well as a transition relation between configurations, which tells how this automaton can evolve.

A configuration (ξ, now, s, pt, dl) of a $\mathcal{D}$-automaton $\mathcal{A}(d) = (I, O, F, i, T_U, T_I, T_O, \mathcal{V})$ is *initial* iff $\mathrm{dom}(\xi) = \mathcal{V}$, that is, ξ associates values only to customer-initialised clock variables, $s = i$, $pt = Pt(\xi, now, s)$ and $dl = Dl(\xi, now, s)$. An initial state, that is, the initial values *now* and $\xi(v)$ for $v \in \mathcal{V}$, is meant to be chosen by the party that is represented by the automaton. In the case of the automaton of Fig. 5 for instance, we have $\mathcal{V} = \{T_i\}$, and the value T_i (relative to *now*) is chosen by Chloe$_i$. In case $\mathcal{V} = \emptyset$, as in Fig. 3, the initial value of *now* is irrelevant, so one could just as well take $now := 0$.

$$\frac{s \in I \wedge now < now' \leq dl \wedge z = now' - now}{(\xi, now, s, pt, dl) \xrightarrow{z} (\xi, now', s, pt, dl)}$$

$$\frac{s \in O \cup F \wedge now < now' < dl \wedge z = now' - now}{(\xi, now, s, pt, dl) \xrightarrow{z} (\xi, now', s, pt, dl)}$$

$$\frac{s \in I \wedge (s, now \geq \varphi, U, s') \in T_U \wedge now \geq [\![\varphi]\!](\xi)}{(\xi, now, s, pt, dl) \xrightarrow{\bullet} (\xi[U]^{now}, now, s', pt', dl')}$$

$$\frac{s \in I \wedge (s, r(id,m), U, s') \in T_I}{(\xi, now, s, pt, dl) \xrightarrow{r(id,m)} (\xi[U]^{now}, now, s', pt', dl')}$$

$$\frac{(s, r(id,m), U, s') \notin T_I \text{ for all } U \text{ and } s'}{(\xi, now, s, pt, dl) \xrightarrow{r(id,m)} (\xi, now, s, pt, dl)}$$

$$\frac{s \in O \wedge s(id,m) \in pt \wedge pt^- := pt \backslash \{s(id,m)\} \neq \emptyset}{(\xi, now, s, pt, dl) \xrightarrow{s(id,m)} (\xi, now, s, pt^-, dl)}$$

$$\frac{s \in O \wedge pt = \{s(id,m)\}}{(\xi, now, s, pt, dl) \xrightarrow{s(id,m)} (\xi[U]^{now}, now, s', pt', dl')}$$

Here $pt' := Pt(\xi[U]^{now}, now, s')$, $dl' := Dl(\xi[U]^{now}, now, s')$.

Fig. 8 Transitions between automaton configurations

The transition relation between configuration is defined in Fig. 8. Transitions are labelled either with a positive real number $z \in \mathbb{R}^{+}$, to indicate passage of time, or with the special symbol $\bullet$, to indicate an (instantaneous) time-out-transition, or with an input or output expression, indicating the receipt or transmission of a message. The latter two transitions are also instantaneous, in the sense that only the end of a durational receipt or transmission activity is modelled. The first two rules state that an automaton can idle in a state as long as the deadline pertaining to that state is not reached. For output state this deadline is strict (hence "$<$"), whereas for input states it is not. The fourth and fifth rules say that a message m from automaton id can arrive at any time; this is not under the control of the receiving automaton. However, the receiving automaton will perform a transition in response to this incoming message only if it is in a state s with an input transition labelled $r(id, m)$. In all other cases the incoming message is ignored.

Asynchronous semantics of ANTA

Given an ANTA $\mathcal{A}$ with domain $\mathcal{D}$, let $\mathcal{C}_d$ for $d \in \mathcal{D}$ denote the set of configurations of the automaton $\mathcal{A}(d)$, and let $\mathcal{C} := \prod_{d \in \mathcal{D}} \mathcal{C}_d$. A *configuration* of $\mathcal{A}$ is a pair $(\vec{C}, P)$ of a $\mathcal{D}$-tuple $\vec{C} \in \mathcal{C}$ of configurations of the automata in the network, and a set P of pending messages. Here a *pending message* is a tuple $(\vec{t}, sd, id, m) \in \mathbb{R}^{\mathcal{D}} \times \mathcal{D} \times \mathcal{D} \times \mathtt{MSG}$, with $m \in \mathtt{MSG}$ the content of the message, *sd* and *id* its sender and destination,

$$\frac{C_d \xrightarrow{t^d} C'_d \quad \text{for each } d \in \mathcal{D}}{(\vec{C}, P) \longrightarrow (\vec{C}', P)}$$

$$\frac{C_{id} \xrightarrow{\bullet} C'_{id} \wedge C_d = C'_d \quad \text{for each } d \in \mathcal{D}\backslash id}{(\vec{C}, P) \xrightarrow{\bullet @id} (\vec{C}', P)}$$

$$\frac{\begin{array}{c} C_{id} \xrightarrow{r(sd,m)} C'_{id} \wedge C_d = C'_d \quad \text{for each } d \in \mathcal{D}\backslash id \\ \wedge\, P = P' \uplus \{(\vec{t}, sd, id, m)\} \end{array}}{(\vec{C}, P) \xrightarrow{r(sd,m)@id} (\vec{C}', P')}$$

$$\frac{\begin{array}{c} C_{sd} \xrightarrow{s(id,m)} C'_{sd} \wedge C_d = C'_d \quad \text{for each } d \in \mathcal{D}\backslash sd \\ \wedge\, P' = P \uplus \{(now(\vec{C}), sd, id, m)\} \end{array}}{(\vec{C}, P) \xrightarrow{s(id,m)@sd} (\vec{C}', P')}$$

Fig. 9 Transitions between ANTA configurations

and $\vec{t} = (t_d)_{d\in\mathcal{D}}$ the time the message was sent, seen as vector of reals according to the local clock of each automaton $d \in \mathcal{D}$.

Given a configuration $(\vec{C}, P)$, one has $\vec{C} = (C_d)_{d\in\mathcal{D}}$, where $C_d = (\xi^d, now^d, s^d, pt^d, dl^d)$ is a configuration of $d \in \mathcal{D}$; let $now(\vec{C}) \in \mathbb{R}^{\mathcal{D}}$ denote the $\mathcal{D}$-tuple $(now^d)_{d\in\mathcal{D}}$.

The behaviour of an ANTA $\mathcal{A}$ is described by its initial configurations and a transition relation between configurations. A configuration $(\vec{C}, P)$ is *initial* if $P = \emptyset$ and C_d is initial for each $d \in \mathcal{D}$.

Fig. 9 defines the transition relation for the asynchronous semantics of ANTA, as employed in Sect. 5.3. The first rule says that the network may evolve simply by time passing in each automaton. The amounts of time t^d according to the local clocks of each automaton $d \in \mathcal{D}$ need not be related in any way. The remaining rules allow the network to evolve by one automaton performing a time-out, receive or send transition, which takes no time at all, and the others remaining unchanged. In the case of receive or send transitions, the corresponding message is taken from or added to the set of pending messages. Note that the field $\vec{t}$ of a pending message is not used at all, and could just as well have been omitted. It is there merely to enable a synchronous semantics of ANTA, obtained from the one above by eliminating certain ANTA configurations and transitions.

Synchronous semantics of ANTA

To make ANTA compatible with the assumption of asynchrony from Dwork et al. [6] we need to exclude configurations in which a message is pending longer than the upperbound Δ of Sect. 3.1, as measured by the clock of any automaton in the network, and ensure a fixed upper bound Ψ on the rate at which one automaton's clock can run faster than another's.

Definition 6 A configuration $(\vec{C}, P)$ is *valid* if P contains no pending message $(\vec{t}, sd, id, m)$ such that $now^d > t_d + \Delta$ for some $d \in \mathcal{D}$.

A vector $\vec{z} \in (\mathbb{R}^+)^{\mathcal{D}}$ of durations is *compatible* if $\frac{z_d}{z_e} \leq \Psi$ for all $d, e \in \mathcal{D}$.

The the synchronous semantics of ANTA, based on given constants Δ and Ψ, differs from the asynchronous semantics only in the first rule of Fig. 9, which obtains the extra requirements that $(\vec{C}', P)$ is valid and $\vec{z}$ compatible. This is the semantics employed in Sect. 3.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
2. Avarikioti, G., Kogias, E. K., Wattenhofer, R., Zindros, D.: Brick: asynchronous payment channels (2020). arXiv:1905.11360
3. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Wang, Y.: UPPAAL—a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, October 1995, LNCS, vol. 1066, pp. 232–243. Springer (1996). https://doi.org/10.1007/BFb0020949
4. Bracha, G.: Asynchronous Byzantine agreement protocols. Inf. Comput. **75**(2), 130–143 (1987). https://doi.org/10.1016/0890-5401(87)90054-X
5. Cholvi, V., Anta, A. F., Georgiou, C., Nicolaou, N., Raynal, M.: Atomic appends in asynchronous Byzantine distributed ledgers. In: Proceedings of the 16th European Dependable Computing Conference (EDCC), pp. 77–84 (2020)
6. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988). https://doi.org/10.1145/42282.42283
7. Fernández Anta, A., Georgiou, C., Nicolaou, N.: Atomic appends: selling cars and coordinating armies with multiple distributed ledgers (2018). arXiv:1812.08446
8. Gazi, P., Kiayias, A., Zindros, D.: Proof-of-stake sidechains. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, pp. 139–156. IEEE (2019). https://doi.org/10.1109/SP.2019.00040
9. van Glabbeek, R., Gramoli, V., Tholoniat, P.: Feasibility of cross-chain payment with success guarantees. In: Proceedings 32nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2020, pp. 579–581. ACM (2020). https://doi.org/10.1145/3350755.3400264
10. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Datab. Syst. **31**(1), 133–160 (2006). https://doi.org/10.1145/1132863.1132867
11. Guerraoui, R.: Revisiting the relationship between non-blocking atomic commitment and consensus. In: Hélary, J.M., Raynal, M. (eds.) Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG), LNCS, vol. 972, pp. 87–100. Springer (1995). https://doi.org/10.1007/BFb0022140
12. Hadzilacos, V.: On the relationship between the atomic commitment and consensus problems. In: Proceedings of the Asilomar Workshop on Fault-Tolerant Distributed Computing, LNCS, pp. 201–208. Springer (1990). https://doi.org/10.1007/BFb0042336
13. Herlihy, M.: Atomic cross-chain swaps. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC, pp. 245–254 (2018). https://doi.org/10.1145/3212734.3212736

14. Herlihy, M., Liskov, B., Shrira, L.: Cross-chain deals and adversarial commerce. Proc. VLDB Endow. **13**(2), 100–113 (2019). https://doi.org/10.14778/3364324.3364326. arXiv:1905.09743v5
15. Kiayias, A., Zindros, D.: Proof-of-work sidechains. In: Bracciali, A., Clark, J., Pintore, F., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security, Revised Selected Papers. Financial Cryptography Workshops 2019, LNCS, pp. 21–34. Springer (2020). https://doi.org/10.1007/978-3-030-43725-1_3
16. Krcál, P., Yi, W.: Communicating timed automata: the more synchronous, the more difficult to verify. In: Ball, T., Jones, R.B. (eds.) Proceedings of 18th International Conference on Computer Aided Verification, CAV'06, LNCS, vol. 4144, pp. 249–262. Springer (2006). https://doi.org/10.1007/11817963_24
17. Lind, J., Naor, O., Eyal, I., Kelbert, F., Pietzuch, P., Sirer, E.: Teechain: reducing storage costs on the blockchain with offline payment channels. In: Proceedings of the 11th ACM International Systems and Storage Conference, p. 125, ACM (2018). https://doi.org/10.1145/3211890.3211904
18. Lynch, N.: Input/output automata: basic, timed, hybrid, probabilistic, dynamic. In: Amadio, R.M., Lugiez, D. (eds.) Proceedings of the International Conference on Concurrency Theory (CONCUR), LNCS, vol. 2761, pp. 191–192 (2003). https://doi.org/10.1007/978-3-540-45187-7_12
19. van der Meyden, R.: On the specification and verification of atomic swap smart contracts (extended abstract). In: Proceedings of the IEEE International Conference on Blockchain and Cryptocurrency, ICBC, pp. 176–179 (2019). https://doi.org/10.1109/BLOC.2019.8751250
20. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
21. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
22. Poon, J., Dryja, T.: The bitcoin lightning network: scalable off-chain instant payments, Draft Version 0.5.9.2 (2016). https://lightning.network/lightning-network-paper.pdf
23. Ranchal-Pedrosa, A., Gramoli, V.: Platypus: offchain protocol without synchrony. In: Proceeding of the 18th IEEE International Symposium on Network Computing and Applications, NCA (2019). arXiv:1907.03730
24. Thomas, S., Schwartz, E.: A protocol for interledger payments. Whitepaper (2015). https://interledger.org/interledger.pdf
25. Wood, Gavin: Polkadot: Vision for a heterogeneous multi-chain framework. White Paper (2016). https://polkadot.network/PolkaDotPaper.pdf
26. Zakhary, V., Agrawal, D., El Abbadi, A.: Atomic commitment across blockchains. Proc. VLDB Endow. 13(9), 1319–1331 (2020). https://doi.org/10.14778/3397230.3397231
27. Zamyatin, A., Al-Bassam, M., Zindros, D., Kokoris-Kogias, E., Moreno-Sanchez, P., Kiayias, A., Knottenbelt, W.J.: SoK: communication across distributed ledgers. In: Borisov, N., Diaz, C. (eds.) Proceedings of the 25th International Conference on Financial Cryptography and Data Security (FC'21), LNCS, vol. 12675, pp. 3–36. Springer (2021). https://doi.org/10.1007/978-3-662-64331-0_1
28. Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., Knottenbelt, W.J.: XCLAIM: trustless, interoperable, cryptocurrency-backed assets. In: 2019 IEEE Symposium on Security and Privacy, SP'19, San Francisco, USA, pp. 193–210 (2019). https://doi.org/10.1109/SP.2019.00085