The VLDB Journal (2022) 31:1291–1309
https://doi.org/10.1007/s00778-021-00686-1

SPECIAL ISSUE PAPER

Cross-chain deals and adversarial commerce

Maurice Herlihy[1] · Barbara Liskov[2] · Liuba Shrira[3]

Received: 29 November 2020 / Revised: 7 May 2021 / Accepted: 27 May 2021 / Published online: 20 August 2021

Abstract
Modern distributed data management systems face a new challenge: how can autonomous, mutually distrusting parties cooperate safely and effectively? Addressing this challenge brings up familiar questions from classical distributed systems: how to combine multiple steps into a single atomic action, how to recover from failures, and how to synchronize concurrent access to data. Nevertheless, each of these issues requires rethinking when participants are autonomous and potentially adversarial. We propose the notion of a *cross-chain deal*, a new way to structure complex distributed computations that manage assets in an adversarial setting. Deals are inspired by classical atomic transactions, but are necessarily different, in important ways, to accommodate the decentralized and untrusting nature of the exchange. We describe novel safety and liveness properties, along with two alternative protocols for implementing cross-chain deals in a system of independent blockchain ledgers. One protocol, based on synchronous communication, is fully decentralized, while the other, based on semi-synchronous communication, requires a globally shared ledger. We also prove that some degree of centralization is required in the semi-synchronous communication model.

1 Introduction

The emerging domain of electronic commerce spanning multiple blockchains is a kind of fun-house mirror of classical distributed computing: familiar features are recognizable, but distorted. For example, atomic transactions are often described in terms of the well-known ACID properties [29]: atomicity, consistency, isolation, and durability. We will see that cross-chain commerce requires structures superficially similar to, but fundamentally different from, atomic transactions. In particular, the notions of correctness for atomic transactions must be rethought.

Here we propose the notion of a *cross-chain deal*, a new computational abstraction for structuring interactions that mirror standard (non-blockchain) commercial practices. Cross-chain deals are inspired by classical atomic transactions and modern cross-chain swaps [11,12,19,30,31,42,43,54,55], but differ, in essential ways, from both.

Here is a simple example. Alice is a ticket broker. She buys tickets at wholesale prices from event organizers and resells them at retail prices to consumers, collecting a modest commission. Alice lives in the future, where tickets are managed on a *ticket blockchain*, a tamper-proof replicated ledger that tracks ticket ownership. Similarly, the coins paid for those tickets live on a distinct *coin blockchain*. Both blockchains support *contracts* (sometimes "smart contracts"), simple programs that control when and how ownership of tickets and coins is transferred. One day Bob, a theater owner, decides to sell two coveted tickets to a hit play for 100 coins. Alice knows that Carol would be willing to pay 101 coins for those tickets, so Alice moves to broker a deal between Bob and Carol. The challenge is to devise a distributed protocol, executed by Alice, Bob, and Carol, communicating through contracts running on various blockchains, to execute a cross-chain deal that transfers the tickets from Bob to Carol, and the coins from Carol to Bob, minus Alice's commission. If all goes as planned, all transfers take place, and if anything goes wrong (someone crashes or tries to cheat), no honest party should end up worse off. For example, if Alice follows the protocol, then even in the presence of failures, she should

Supported by NSF grant 1917990.

✉ Maurice Herlihy
maurice.herlihy@gmail.com

Barbara Liskov
liskov@csail.mit.edu

Liuba Shrira
liuba@brandeis.edu

1 Brown University, Providence, USA

2 MIT CSAIL, Cambridge, USA

3 Brandeis University, Waltham, USA

not end up holding tickets she cannot sell or coins that she must refund. If Carol follows the protocol and pays for the tickets, then she and she alone should receive them.

Cross-chain deals, although somewhat like transactions, are not transactions. In one way they are simpler: transactions are used to carry out complex, possibly distributed, state changes, while deals, by contrast, simply exchange assets among parties. But in a fundamental way they are more complex. A transaction runs on behalf of only one party, which is in charge of what the transaction does and whether it commits. By contrast, a cross-chain commercial deal runs on behalf of multiple parties, each party acting in its own interest. One party's actions may affect the other parties in complex ways, and parties may misbehave in malicious and even irrational ways.

These differences mean that the ACID properties must be reformulated. Classical *atomicity* means that a transaction's effects take place everywhere or nowhere. This notion of atomicity cannot be guaranteed when parties are potentially malicious: the best one can do is to ensure that honest parties cannot end up "worse off" due to the actions of dishonest parties.

Classical *isolation* guarantees that concurrent transactions cannot interfere in destructive ways. Isolation is typically provided by a concurrency control regime such as serializability or snapshot consistency. These regimes are poorly suited to cross-chain commerce, where mutually distrusting parties may require multiple cautious interactions to set up and execute a deal. Instead, non-interference takes the form of protection against *double-spending*: ensuring that one party does not concurrently sell the same asset to multiple counterparties.

Returning to our example, suppose (deviating) Carol erroneously sends 1001 coins to Alice, instead of the 101 she expected. If the deal is consummated, then (compliant) Alice ends up with a commission of 901 coins, an outcome that is neither "all" nor "nothing," even for compliant parties. Of course, such an outcome is unlikely in practice, but such distinctions matter when reasoning about correctness.

Adversarial commerce, defined as economic exchange among mutually untrusted autonomous parties, is here to stay. Moreover, a system architecture composed of autonomous untrusted parties that communicate via shared tamper-proof data stores is the most natural way to organize such a system. Although we will propose protocols based on today's blockchains and contracts, our principal claims do not depend on specific blockchain technologies, or even on blockchains as such. Instead, we focus on computational abstractions central to any systematic approach to adversarial commerce, no matter what technology underlies the shared data stores.

This paper makes the following contributions.

- We propose the *cross-chain deal* as a new computational abstraction for structuring complex distributed exchanges in an adversarial setting. Deals require new notions of correctness and new distributed protocols.
- We propose new safety and liveness properties to replace the classical notions of transactional atomicity.
- We describe two protocols for implementing cross-chain deals: a fully decentralized *timelock* protocol that assumes a synchronous communication model, and a more centralized *certified blockchain* (CBC) protocol for a semi-synchronous model where synchronous communication may be intermittent.
- We present a proof that any protocol that tolerates periods of asynchrony must rely on a centralized blockchain (or similar ledger structure).
- We sketch implementations of the two protocols and use them to analyze the costs of the protocols. Given the immature state of today's blockchain technology, we focus on inherent, abiding, platform-independent trade-offs and costs rather than explicit performance measurements. Our intent is to illustrate the costs associated with the qualitatively different ways in which cross-chain deals can be supported.

The remainder of the paper is organized as follows. We begin in Sect. 2 by providing additional examples of deals and comparing them to a simpler notion: the cross-chain swap Sect. 3 defines deals and our correctness properties. Section 4 provides a brief discussion of blockchains and what we assume about them. We describe how deals work in Sect. 5, explain our two protocols in Sects. 6 and 7, and analyze their costs in Sect. 8. We prove that a centralized resource is required in any protocol that tolerates periods of asynchrony in Sect. 9. We discuss related work in Sect. 10, compare the two approaches in Sect. 11, and conclude in Sect. 12.

2 Deals and swaps

This section provides two more examples motivating the need for on-chain commerce protocols not readily realizable by prior proposals. Then it uses the examples to explain the differences between cross-chain deals and cross-chain swaps.

To simplify the examples, we assume that Alice, Bob, and Carol have all agreed to participate prior to the start of each scenario.

2.1 Cross-chain auctions

Suppose Alice wants to auction some tickets to Bob and Carol. She offers the tickets for sale, and both Bob and Carol make their bids. If neither bid is greater than Alice's reserve price the deal falls through: Alice retains the tickets and Bob and Carol retain their money. Otherwise, the winner gets the tickets, Alice gets the winner's money, and the loser retains his or her money.

Here, too, Alice needs a distributed protocol that causes the appropriate transfer of assets based on the behavior of the participating parties. The exchange of assets should happen atomically: if Alice takes the winner's coins, then the winner gets the tickets and the loser regains their coins.

2.2 Cross-chain flash loans

Blockchains have given rise to novel financial instruments, such as *flash loans* [9], in which a lender provides an unsecured short-term loan to a borrower, subject to the condition that both the loan and its repayment take place in a single deal. This same-deal condition is essential because it allows the lender to avoid the usual risks of lending: the "counterparty risk" that the borrower will default, and the "opportunity cost" of having those assets locked up for a long time.

Today, flash loans are possible only within a single blockchain, but there are uses for cross-chain flash loans. However, to support this possibility there is a need for currency that can be used on more than one blockchain.

One way to accomplish this is through the use of a *stablecoin*, a cryptocurrency issued by an organization that pledges to back each coin by a non-electronic asset or basket of assets. For example, Tether [51] is backed by US dollars, and Libra [6] by a basket of fiat currencies. Parties that use stablecoins trust that the stablecoin organization will not issue unsecured coins. A stablecoin can typically be used on multiple blockchains. For example, the Tether stablecoin is currently supported on seven different blockchains [16].

Suppose Alice discovers the following arbitrage opportunity. She finds two *automated market maker* contracts [10] (AMMs): AMM X will sell Alice 1 token in return for 990 units of the stablecoin, while AMM Y will sell Alice 1000 stablecoins in return for 1 token. (Both AMM prices include their commissions.) Alice needs to find enough capital to make this tiny difference worth exploiting. She would like to arrange a "flash loan" [9] of 990 stablecoins from Bob, convert them to 1000 stablecoins through arbitrage, then repay Bob 993 stablecoins including interest, for a profit of 7 stablecoins. Bob is willing to charge a low interest rate for the flash loan because he is assured that his loan is risk-free: if the deal completes, he collects his interest, but if not, he gets his coins back right away.

The catch is that Bob holds his stablecoins on the blue blockchain, but Alice's arbitrage opportunity is on the red blockchain. Alice and Bob need a distributed protocol that enlists the stablecoin organization to transfer Bob's loan to Alice, and Alice's repayment back to Bob.

2.3 Cross-chain deals versus cross-chain swaps

Today, *cross-chain swaps* are the most common form of cross-chain exchanges. Each party first places an asset into escrow, transferring custody of that asset to a contract, which will eventually either transfer ownership of the asset to a counterparty, or refund the asset to the original owner. Once all parties have escrowed their assets, each party inspects the escrowed assets it is due to receive. When and if all parties approve, the transfers take place. In most of these proposals, transfers are triggered by producing the preimage to a hashed value within a certain time (a *hashed timelock*), but there are also proposals to use blockchain-based two-phase commit protocols [54].

While cross-chain swaps are a special case of cross-chain deals, cross-chain deals are substantially more flexible and powerful. Alice, Bob, and Carol's simple ticket brokering deal cannot be expressed as a classical cross-chain swap because Alice is trading assets she does not own, and therefore cannot escrow. Alice pays Bob with coins she receives from Carol, and sells Carol a ticket she receives from Bob. Alice cannot commit to any transfers at the start of the protocol because she does not yet have custody of any assets. Instead, Alice contributes value by acting as a broker, a middleman relaying assets between Bob and Carol (or more realistically, between a pool of wholesale sellers and a pool of retail buyers).

The use of assets a party does not own also occurs in the cross-chain flash loan, so this exchange cannot be expressed as a cross-chain swap. Here Alice exploits her arbitrage opportunity using stablecoins she does not own at the start of the protocol, and she pays Bob back with assets she does not acquire until the end of the protocol. The auction example cannot be expressed as a cross-chain swap because the auction's outcomes (reserve price exceeded, identity of winner) cannot be determined until all bids have been submitted.

3 Cross-chain deals

Here we define cross-chain deals, what it means to execute them, and what it means for them to be correct.

3.1 Specifying the deal

Each payoff (set of final transfers) for a deal can be expressed as a matrix (or table), where each row and column is labeled with a party, and the entry at row i and column j shows the assets to be transferred from party i to party j. A party's column states what it expects to acquire from the deal (its *incoming* assets), and its row states what it expects to relinquish (its *outgoing* assets). Summarizing outcomes using matrices is convenient when the number of distinct payoffs is

Table 1 Alice brokers a ticket sale between Bob and Carol (Rows represent outgoing transfers, and columns incoming transfers)

	Alice	Bob	Carol
Alice		100 coins	tickets
Bob	Tickets		
Carol	101 coins		

Table 2 Two possible auction outcomes: Bob outbids Carol, and vice versa

	Alice	Bob	Carol
Alice		Tickets	100
Bob	200		
Carol	100		
Alice		200	Tickets
Bob	200		
Carol	300		

small, but can become unwieldy if there are too many distinct outcomes. In that case, a different formalism, such as sets of parameterized constraints, can be used.

For the ticket brokering deal, the payoff is given by the 3×3 matrix in Table 1. Carol expects to transfer 101 coins to Alice in return for tickets transferred from Alice. Similarly, Bob expects to transfer tickets to Alice in return for 100 coins from Alice. Although the table refers only to "tickets," the specific (non-fungible) tickets to be provided would be part of the deal specification, while the specific (fungible) coins would likely be omitted.

The deal where Alice auctions the tickets to Bob and Carol requires two matrices, one for each successful outcome: (1) if Bob outbids Carol, Alice transfers her asset to Bob and refunds Carol's bid, and (2) if Carol outbids Bob, the transfers are symmetric. (A more detailed description of an on-chain auction might also include fees and deposits to penalize malicious behavior by bidders.)

Table 3 displays the payoff matrix for the cross-chain flash loan example. Asset amounts are shown in red or blue to highlight whether the transfer occurs on the red or blue blockchains. As noted, Bob is present on the Blue blockchain, Alice on the Red blockchain, and the Stablecoin organization ("stablecoin.org") executing the cross-chain transfers is present on both.

These three examples have an important property in common: each party can decide for itself whether it wants to accept or reject the proposed final exchange, based on the assets that appear in its own row and column. In the ticket brokering example, Carol can decide whether the tickets are acceptable before paying for them. In the auction example, Alice can decide to accept only bids that exceed her reserve price. In the flash loan example, Bob will choose to make the loan only if Alice is prepared to repay it as part of the deal in which it was lent.

3.2 Correctness

Parties to a deal carry out a *protocol* to complete the deal's transfers. In an environment where parties cannot be guaranteed to follow a protocol, it is impossible to guarantee that all transfers take place as promised by the deal specification. Which kind of partial transfers should be deemed acceptable?

Instead of distinguishing between faulty and non-faulty parties, as in classical models, we distinguish only between *compliant* parties who follow the protocol, and *deviating* parties who do not. Many kinds of fault-tolerant distributed protocols require that some fraction of the parties be compliant. For example, proof-of-work consensus [39] requires a compliant majority, while most Byzantine fault-tolerant (BFT) consensus protocols require a supermajority of more than two-thirds of the participants to be compliant. For cross-chain deals, however, it is prudent to make no assumptions about the number of deviating parties.

This classification of parties as either compliant or deviating is partly inspired by the classification in the *BAR* model [2], which identifies parties as rational, altruistic, or Byzantine, although the two classifications are not directly comparable (c.f. [24]). Our model does not distinguish between rational and altruistic parties. Critically, our classification differs from that of BAR (and other standard models of Byzantine behavior) by not limiting the number of Byzantine parties.

The most fundamental safety property is (informally) that compliant parties should end up "no worse off," even when other parties deviate arbitrarily from the protocol. A party's

Table 3 Alice takes and repays a flash loan from Bob to exploit an arbitrage opportunity

	Alice	Bob	Stablecoin Org.	AMM *X*	AMM *Y*
Alice			993 coins	990 coins	1 token
Bob			990 coins		
Stablecoin Org.	990 coins	993 coins			
AMM *X*	1 token				
AMM *Y*	1000 coins				

payoff for a protocol execution is the sets of incoming and outgoing assets actually transferred. Some payoffs are considered *acceptable*, the rest are not. Some acceptable payoffs are preferable to others, but any acceptable payoff leaves that party "no worse off."

Every party considers the following payoffs acceptable: ALL, where all agreed transfers take place, and NOTHING, where no transfers take place. In addition, we allow a party to consider other payoffs acceptable. For example, a party that expects three incoming transfers and three outgoing transfers may be willing to accept a payoff where it receives only two incoming transfers in return for only two outgoing transfers. Of course, any such choice is deal-dependent.

Payoffs can be ordered by acceptability: if one payoff is acceptable to a party, so is any payoff where that party receives more incoming assets (it gets something extra for nothing), and so is any payoff where that party transfers fewer outgoing assets (it gets something it wants for a discount price). For example, a payoff where a party transfers no outgoing assets but receives some incoming assets is preferable to the NOTHING payoff, and hence is acceptable. Such outcomes, while uncommon in practice, cannot be excluded if parties can act irrationally, or their rationales are unknown.

A cross-chain deal protocol satisfies *safety* if:

Property 1 *For every protocol execution, every compliant party ends up with a payoff it deems acceptable.*

This notion of safety replaces the classical *all-or-nothing* property of atomic transactions, which, as noted, cannot be guaranteed in the presence of deviating parties.

Cross-chain protocols typically rely on some form of *escrow*, where each party transfers ownership of each outgoing asset to an intermediate escrow contract that retains custody of that asset until the outcome of the deal is resolved. The following *weak liveness* property ensures that conforming parties' assets cannot be locked up forever.

Property 2 *No asset belonging to a compliant party is escrowed forever.*

Finally, protocols should satisfy the following *strong liveness* property:

Property 3 *If all parties are compliant and consider their proposed payoffs acceptable, then all transfers eventually happen (all parties' payoffs are* ALL*).*

It well-known [23] that strong liveness is possible only in periods when the communication network is synchronous, ensuring a fixed upper bound on message delivery time. One of the two protocols proposed later depends on an explicit bound on how long it takes to complete all transfers and a party is considered to be deviating if it misses the deadline; the other does not.

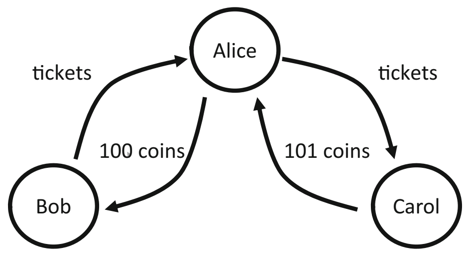

Fig. 1 Alice, Bob, and Carol's ticket brokering deal as a digraph

3.3 Well-formed deals

Not all deals make sense. As an extreme example, suppose Alice and Bob agree that Alice will send 100 coins to Bob, but Bob will do nothing in return. Here, Bob is a "free rider," collecting something for nothing, and if Alice is rational, she would never agree to such a deal. (Of course, Bob should be suspicious if Alice does agree.)

It is sometimes convenient to represent a payoff matrix as a *directed graph* ("digraph"), where each vertex represents a party, and each directed arc represents a transfer. Figure 1 shows the ticket-brokering payoff expressed as a digraph. We define a deal to be *well-formed* if every payoff digraph is *strongly connected*: there is a directed path from any node to any other node. If the payoff digraph is not strongly connected, then there are parties that act as *free riders*: they collectively take assets from the others without giving any in return (see Herlihy [31] for a more complete discussion). No rational party would accept a deal with free riders, but even if they did, they would have no incentive to conform to the protocol because they could improve their payoffs by collectively skipping all transfers to free riders.

Recall that for simplicity, we assume that all asset transfers take place on blockchains. One way to extend the model to accommodate off-chain asset transfers (such as mailing a package in response to a cryptocurrency payment), is to add a fictional "external" blockchain to represent off-chain transfers. Of course, measures beyond the scope of this paper would still be needed to ensure the entries recorded on this external blockchain are accurate.

4 System model

For our purposes, a *blockchain* is a publicly readable, tamper-proof distributed ledger (or database) that tracks ownership of *assets* among various *parties* or *contracts*. An asset may be *fungible*, like a sum of money, or *non-fungible*, like a theater ticket. A party can be a person or an organization. We assume multiple independent blockchains, each managing a different kind of asset. We restrict our attention to blockchains that track asset ownership, and to deals that transfer asset ownership from one party to another. We assume all value transfers are explicitly represented on the blockchain. For example, Alice does not send paper tickets to Carol off-chain.

A party can *publish* an entry on a blockchain, and it can *monitor* one or more blockchains, receiving notifications when other parties publish entries. In our model, publishing an entry usually executes a blockchain-resident program called a *contract*. We will use contracts for *escrow*: an asset owner temporarily transfers ownership of an asset to a contract. If certain conditions are met, the contract transfers that asset to a *counterparty*, and otherwise it refunds that asset to the original owner.

A party can publish a new contract on a blockchain, or call a function exported by an existing contract. Contract code and contract state are public, so a party calling a contract knows what code will be executed. Contract code must be deterministic because it is executed by multiple validators who must all observe the same results.

A contract accesses data on the blockchain where it resides, but it *cannot* directly access data from the outside world. We refer to this limitation as *contract myopia*. Contract myopia means that a contract is unable to observe what is happening on other blockchains and it cannot call contracts on other blockchains. Instead a contract on blockchain A can learn of a change to a blockchain B only if some party explicitly informs A of B's change, along with some kind of "proof" that the information reported about B's state is correct.

Parties are given a protocol, which each party may or may not follow. All protocol actions are accomplished by parties invoking functions (methods) provided by smart contracts on blockchains. All states used in protocol descriptions are ultimately implemented as states within contracts. All state transitions are the results of parties calling contract functions. Contract code is passive, public, deterministic, and trusted, while parties are active, autonomous, and mutually untrusting. Parties may or may not act rationally.

We make standard cryptographic assumptions. Each party has a public key and a private key, and any party's public key is known to all. Messages are signed so they cannot be forged, and they include single-use labels ("nonces") so they cannot be replayed.

5 How deals work

This section gives a formal definition of a cross-chain deal in terms of a simple state machine that tracks ownership of assets, and whose transitions represent escrows, transfers, commits, and aborts. The section makes explicit which properties of blockchains and blockchain-like systems are essential to cross-chain deals.

Let $\mathcal{P}$ be a domain of *parties*, and $\mathcal{A}$ a domain of *assets*. (A party may be a person or a contract, and assets are digital tokens representing items of value.) An asset has exactly one *owner* at a time: $Owns(P, a)$ is *true* if P and only P owns a.

A deal tentatively transfers asset ownership from one party to another. We say a tentative transfer *commits* if it becomes permanent, and it *aborts* if it is discarded. A deal *commits* if all its tentative transfers commit, and it *aborts* if all its tentative transfers abort.

While a deal is in progress, its state encompasses two (partial) maps, $C : \mathcal{A} \to \mathcal{P}$ and $A : \mathcal{A} \to \mathcal{P}$, both initially empty. $C(a)$ indicates the eventual owner of asset a if the deal commits at a's blockchain, and $A(a)$ the owner if it aborts at that blockchain. We use $Owns_C(P, a)$ to indicate that P will own a if the deal commits, and $Owns_A(P, a)$ to indicate that P will own a if the deal aborts.

Escrow plays the role of classical concurrency control, ensuring that a single asset cannot be transferred to different parties at the same time. Here is what happens when P places a in escrow during deal D:

Pre: $Owns(P, a)$
Post: $Owns(D, a)$ and $Owns_C(P, a)$ and $Owns_A(P, a)$

The precondition states that P can escrow a only if P owns a. If that precondition is satisfied, the postcondition states that ownership of a is transferred from P to D (via the escrow contract), but P remains the owner of a in both C and A; since no tentative transfer has happened yet, P would regain ownership of a if D were to terminate either way. For example, when Bob escrows his tickets, they temporarily become the property of the contract, but should the deal commit or abort right then, the tickets would revert to Bob.

Next we define what happens when party P tentatively transfers an asset (or assets) a to party Q as part of deal D.

Pre: $Owns(D, a)$ and $Owns_C(P, a)$
Post: $Owns_C(Q, a)$

The precondition requires a to be held in escrow by D, with P the indicated owner should D commit. If the precondition is satisfied, the postcondition states that Q will become the owner of the transferred asset a should D commit (at this point). For example, when Carol transfers 101 coins to Alice, Alice becomes the owner of those coins in C. Alice can then transfer 100 of those coins to Bob, retaining one for herself, all in C.

Assets remain in escrow until the deal terminates. If the deal terminates by committing, the owners of assets in C become the actual owners (displacing D). If it terminates by aborting, the owners of assets in A become the actual owners (again displacing D).

5.1 Phases

A deal is executed in the following phases.

Clearing phase

The participants find one another through a market-clearing service that establishes the proposed transfers, and possibly other deal-specific information. The market clearing service may or may not be centralized, but *it is not a trusted party*, because each party later decides for itself whether to participate, and whether to complete the deal. The precise structure of the market-clearing service is beyond the scope of this paper.

Escrow phase

Parties escrow their outgoing assets. For example, Bob escrows his tickets and Carol her coins.

Transfer phase

The parties perform the sequence of tentative ownership transfers according to the deal. For example, Bob tentatively transfers the tickets to Alice, who subsequently transfers them to Carol.

Validation phase

Once the tentative transfers are complete, each party checks that its incoming assets are properly escrowed (so they cannot be double-spent), that it is still willing to transfer its outgoing assets, and that the overall payoff is acceptable. For the ticket brokering example, Carol checks that the tickets to be transferred are escrowed, that the seats are (at least as good as) the ones agreed upon, and that she is not about to somehow overpay.

In adversarial commerce, it is necessary that each party decide for itself whether the proposed payoffs are acceptable. For example, only Carol can decide whether the tickets she is about to purchase are ones she wants.

Commit phase

The parties vote on whether to make the tentative transfers permanent. If all parties vote to commit in a timely way, the escrowed assets are transferred to their new owners; otherwise they are refunded to their original owners.

5.2 Discussion

Cross-chain deals rely on two critical, intertwined mechanisms. First, the escrow mechanism prevents double-spending by making the escrow contract itself the asset owner. Although contracts can be trusted to faithfully execute their own publicly visible code, care must be taken to ensure weak liveness: assets belonging to compliant parties must not remain escrowed forever in the presence of malicious behavior by counterparties. Second, the commit protocol must be resilient in the presence of malicious misbehavior. A deviating party may be able to steal assets if it can convince some parties that the deal completed successfully, and others that it did not. If a deviating party can prevent (or delay) a decision by the commit protocol, then it can keep assets locked up forever (or a long time).

The principal challenge in implementing cross-chain deal protocols is the design of the integrated escrow management and commit protocol. Just as with classical transaction mechanisms, there are many possible choices and trade-offs. In the remainder of this paper, we describe two cross-chain deal protocols, implemented via contracts, one for a synchronous communication model, and one for a semi-synchronous model, each making different trade-offs concerning decentralization and fault-tolerance. We will prove that these protocols satisfy the correctness conditions introduced earlier.

6 Timelock protocol

We now describe a *timelock* deal protocol where escrowed assets are released if all parties vote to commit in a timely way. Parties do not explicitly vote to abort. Instead, timeouts are used to ensure that escrowed assets are not locked up forever if some party crashes or walks away from the deal.

This protocol assumes a *synchronous* network model where there is a known upper bound Δ on the propagation time for one party's change to the blockchain state to be noticed by the other parties. Protocols such as Bitcoin [39] and Ethereum [21] operate in this model. In practice, synchronous protocols may require countermeasures against denial-of-service attacks, such as choosing timeouts carefully, or establishing "watchtowers" [14], agents paid to watch for and respond to certain events in a timely way.

In our brokerage example, Bob places his tickets into escrow, then transfers them to Alice, who transfers them to Carol. All parties examine their incoming assets, and if the resulting payoffs are acceptable, the parties vote to commit at the escrow contract on each asset's blockchain. For example, if Alice, Bob, and Carol all register commit votes on the ticket blockchain, the escrow contract releases the tickets to Carol. All votes are subject to timeouts: if any commit vote fails to appear before the contract's timeout expires, the tickets revert to Bob. (Symmetric conditions apply to Carol's coins.)

Because of the adversarial nature of a deal, each party is motivated to publish its vote on the blockchains controlling its incoming assets (it is eager to be paid), but not on the blockchains controlling its outgoing assets (it is not so eager to pay). To align the protocol with incentives, one party's commit vote may be *forwarded* from one escrow contract to another by a motivated party.

For example, Bob is motivated to publish his commit vote only on the coin blockchain. However, once published, Bob's vote becomes visible to Carol, who is motivated to forward that vote to the ticket blockchain. Carol's position is symmetric: she is motivated to publish her vote only on the ticket blockchain, but Bob is motivated to forward it to the coin blockchain. Alice is motivated to send her vote to both blockchains. (Nevertheless, no harm occurs if a party sends its commit vote directly to any contract.)

To be compliant in this protocol, parties must respond to events in a timely manner. A tricky part of this protocol is how to assign timeouts. A protocol that simply assigns each party a timeout for voting on each asset is incorrect, as shown by the following example.

Suppose the protocol assigns timeouts so that Alice must vote to commit on the ticket blockchain before time A_t, and she must vote to commit on the coin blockchain before time A_c. Let Δ be the worst-case propagation time: the delay between when a party observes a change to one blockchain, and when it makes a subsequent change to another blockchain. Suppose Bob and Carol have both voted to commit on both blockchains. Alice waits until just before A_c to register her vote on the coin blockchain, unlocking Carol's payment to Bob. If Alice now pauses, it may take another Δ for Carol to observe Alice's vote and forward it to the ticket blockchain, implying that $A_t \geq A_c + \Delta$. In another scenario, Alice waits until just before A_t to register her vote on the ticket blockchain, unlocking Bob's ticket transfer to Carol. If Alice now pauses, it may take another Δ for Bob to observe Alice's vote and forward it to the coin blockchain, implying that $A_c \geq A_t + \Delta$, a contradiction.

To resolve this dilemma, each escrow contract's timeout for a party's commit vote depends on the *length of the path* along which that vote was forwarded. For example, if Alice votes directly, her vote will be accepted only if it is received within Δ of the commit protocol's starting time. This vote must be signed by Alice. If Alice forwards a vote from Bob, that vote will be accepted only if it is received within $2 \cdot \Delta$ of the starting time, where the extra Δ reflects the worst-case extra time needed to forward the vote. This vote must be signed first by Bob, then Alice. Finally, if Alice forwards a vote that Bob forwarded from Carol, that vote will be accepted only if it is received within $3 \cdot \Delta$, and so on. This vote must be signed first by Carol, then Bob, then Alice. We refer to this chain of signatures as the vote's *path signature*.

In general, a vote from party X received with path signature p must arrive within time $|p| \cdot \Delta$ of the pre-established commit protocol starting time, where $|p|$ is the number of distinct signatures for that vote.

6.1 Running the protocol

Here is how to execute the phases of a timelock protocol.

Clearing phase

The parties learn the following information from the market-clearing service: the deal identifier D, the list of parties *plist*, a commit phase starting time t_0 used to compute timeouts, and the timeout delay Δ. Most blockchains measure time imprecisely, usually by multiplying the current block height by the average block rate. The choice of t_0 should be far enough in the future to take into account the time needed to perform the deal's tentative transfers, and Δ should be large enough to render irrelevant any imprecision in blockchain timekeeping. Because t_0 and Δ are used only to compute timeouts, their values do not affect normal, deviation-free execution times, where all votes are received in a timely way. If deals take minutes (or hours), then Δ could be measured in hours (or days).

Escrow phase

Each party places its outgoing assets in escrow through an escrow contract

$escrow(D, Dinfo, a)$.

on that asset's blockchain. Here D is the deal identifier and *Dinfo* is the rest of the information about the deal (*plist*, t_0, and Δ); the escrow requests take effect only if the party is the owner of a and a member of *plist*.

Transfer phase

Party P transfers an asset (or assets) a tentatively owned by P to party Q by sending

$transfer(D, a, Q)$.

to the escrow contract on the asset's blockchain. The party must be the owner of a and Q must be in the *plist*.

Validation phase

Each party examines its escrowed incoming assets to see if they represent an acceptable payoff and the deal information provided by the market-clearing service is correct. If so, the party decides to commit.

Commit phase

Each compliant party sends a commit vote to the escrow contract for each incoming asset. (A compliant party is free to altruistically send commit votes to other escrow contracts as well.) A party uses

$commit(D, v, p)$

to vote directly and to forward votes to the deal's escrow contracts, where v is the voter and p is the path signature for v's vote. For example, if Alice is forwarding Bob's vote then v is Bob, and p contains first Bob's signature, and then Alice's signature. (Throughout, we assume that deal identifiers are unique to guard against replay attacks.)

A contract accepts a commit vote only if it arrives in time and is well-formed: all parties in the path signature are unique and in the *plist*, and their signatures are valid and attest to a vote from v. If the commit is accepted, that contract has now accepted a vote from the party.

A contract releases the escrowed asset to the new owner(s) when it accepts a commit vote from every party. If the contract has not accepted a vote from every party by time $t_0 + N \cdot \Delta$, where N is the number of parties participating in the deal, and t_0 the starting time, it will never accept the missing votes, so the contract times out and refunds its escrowed assets to the original owner.

If a deal is well-formed (see Sect. 3.3), compliant parties are required only to forward votes from outgoing assets' contracts to incoming assets' contracts, although nothing prevents them from doing more. This minimal approach is desirable, because parties typically will not want to interact with blockchains they do not otherwise use. If the deal is not well-formed, however, parties may need to forward votes from other contracts to ensure their incoming assets' contracts receive votes in time.

6.2 What could possibly go wrong?

Perhaps counter-intuitively, this protocol does not guarantee that all parties agree on whether the deal committed or aborted. For example, suppose Bob wants to trade b-coins for c-coins, and Carol wants the reverse. Alice brokers their deal, taking 101 b-coins (c-coins) from Bob (Carol), then forwarding 100 coins to each counterparty, keeping a 1-coin commission from each side. This time, Alice happens to hold both kinds of coins. To save time, she transfers 101 of Bob's b-coins into her account, and simultaneously transfers 100 of her own b-coins to Carol, and similarly for Carol (in the opposite direction). Now suppose Alice is infected by a virus, and starts to behave irrationally. After Bob releases his vote, Alice neglects to transfer this vote to Bob's contract (which controls the money for her), but she communicates normally with Carol. Bob's timelock will eventually expire, and he will take back his coins, so for him the deal aborted. Carol, however, will transfer her 101 c-coins to Alice and receive her 100 b-coins, so for her the deal committed normally.

Although this outcome is not "all-or-nothing," it is considered acceptable because all compliant parties end up with acceptable outcomes. But Alice, who irrationally deviated from the protocol, foots the bill, paying Carol without being paid by Bob. We emphasize the unenforceability of classical correctness properties because it may seem counter-intuitive. But if we want to verify that protocols are correct, we must take care to use a realizable notion of correctness.

6.3 Correctness

The correctness of the timelock protocol depends on a combination of cryptographic hashes and the public visibility of data on blockchains. Forwarding a commit vote from one blockchain to another enables subsequent transfers, and the use of cryptographic techniques ensures these votes and transfers cannot be forged.

Theorem 1 *The timelock protocol satisfies safety.*

Proof By construction, transferring a compliant party X's escrowed incoming and outgoing assets is an acceptable payoff for X (since X has voted to commit). Suppose by way of contradiction that X's outgoing asset a is released from escrow and transferred (with commit votes from every party), but the escrow for X's incoming asset b times out and is refunded because of a missing vote from party Z. Suppose Z's commit vote at a's contract arrived with path signature p. The signatures in p cannot include X's, because X is compliant and would have already forwarded Z's vote to b. Z's vote must have arrived at a before time $t_0 + |p| \cdot \Delta$. Since X is compliant, it forwards that vote to b's contract before time $t_0 + (|p| + 1) \cdot \Delta$, where that vote is accepted, a contradiction. □

Theorem 2 *The timelock protocol satisfies weak liveness: no compliant party's outgoing assets are locked up forever.*

Proof Every escrow created by a compliant party has a finite timeout. □

Theorem 3 *The timelock protocol satisfies strong liveness.*

Proof If all parties are compliant, they send commit votes to the escrow contracts for their incoming assets and then forward votes of counterparties as they appear on other contracts. If the deal is well-formed, it is enough to forward votes from incoming to outgoing arcs in the deal digraph. It follows that all commit votes are forwarded to all contracts in time. □

Suppose that Bob acquires Alice and Carol's votes on time, and forwards them to claim the coins, but Alice and Carol are driven offline by a denial-of-service attack before they can forward Bob's vote to the ticket blockchain, so Bob ends up with both the coins and the tickets. Technically, Alice and Carol have deviated from the protocol by not claiming their assets in time. As a countermeasure, Δ should be chosen large enough to make sustained denial-of-service attacks prohibitively expensive.

Vulnerability to extended denial-of-service attacks is not unique to the timelock protocol. For example, both the original hashed-timelock protocol [42] and the Lightning payment network [45] suffer from the same vulnerability: a party that goes off-line at the wrong time may miss a deadline and lose its escrowed assets. Such vulnerabilities can be alleviated by the use of *watchtowers* [14], robust on-line services that monitor escrow contracts on behalf of off-line parties.

7 CBC protocol

Now we describe a protocol that operates in the more demanding semi-synchronous communication model [20], where there is initially no bound on propagation time, but the system eventually reaches a *global stabilization time* (GST) after which the system becomes synchronous. (In practice, the synchronous periods need only last "long enough" to stabilize the protocol.) Consensus protocols such as Algorand [27], Libra [6], and Hot-Stuff [1] operate in this model. Semi-synchronous protocols avoid explicit timeouts, but as shown in Sect. 9, they necessarily require a greater degree of centralization than synchronous protocols.

Unlike in the classical two-phase commit protocol [8], there is no coordinator; instead we use a special blockchain, the *certified blockchain*, or *CBC*, as a kind of shared whiteboard. The CBC might be a stand-alone blockchain, or one already used in the deal. Since we cannot use timed escrow to ensure weak liveness, we allow parties to vote to abort if presented with unacceptable payoffs, or if too much time has passed.

The critical property of the CBC is that it *orders* events. After all assets have been escrowed, each party to a deal sends the CBC a vote whether to commit or abort that deal. Parties can vote more than once, and they can vote for different outcomes. The deal *commits* if the CBC ordering implies that every party voted to commit the deal before any party voted to abort. The deal *aborts* if the CBC ordering implies that some party voted to abort before every party voted to commit. The CBC resolves voting race conditions: if a deviating Alice simultaneously votes to commit and to abort, the CBC decides the order in which those votes take effect.

A party can rescind an earlier commit vote by voting to abort (for example, if the deal is taking too long to complete). To ensure strong liveness, a compliant party that has voted to commit must wait long enough to give the other parties a chance to vote before it changes its mind and votes to abort.

A party can extract a *proof* from the CBC that particular votes were recorded in a particular order. A party claiming an asset (or a refund) presents a proof of commit (or abort) to the contract managing that asset. The contract checks the proof's validity and carries out the requested transfers if the proof is valid. A *proof of commit* proves that every party voted to commit the deal before any party voted to abort, while a *proof of abort* proves that some party voted to abort before every party voted to commit.

7.1 Running the protocol

Here is how to execute the phases of a CBC protocol.

Clearing phase

The parties consult the market-clearing service to learn a unique identifier D and a list of participating parties *plist* (this protocol does not require the t_0 starting time or Δ). One party records the start of the deal on the CBC by publishing an entry:

$$\mathit{startDeal}(D, \mathit{plist}).$$

The calling party must appear in *plist*. If more than one *startDeal* for D is recorded on the CBC, the earliest is considered definitive.

Escrow phase

Each party places its outgoing assets in escrow:

$$\mathit{escrow}(D, \mathit{plist}, h, a, e)$$

Here, h is a hash that identifies a particular *startDeal* entry on the CBC that started the deal; it is needed in case there is more than one such entry on the CBC. The e argument indicates data structures that vary depending on the algorithm used to implement the CBC, as discussed in Sect. 7.4. As in the timelock protocol, the sender must be the owner of asset a and a member of *plist*.

Transfer phase

Party P transfers an asset (or assets) a tentatively owned by P to party Q by sending

$\mathit{transfer}(D, a, Q)$.

to the escrow contract on the asset's blockchain. P must be the owner of a and Q must be in the *plist*.

Validation phase

As before, each party checks that its proposed payoff is acceptable and that assets are properly escrowed with the correct *plist* and h.

Commit phase

The commit phase is split into two sub-phases: first, each party votes, and second, when the voting is complete, each party uses a record of that vote to unlock escrowed assets.

Each party P publishes either a commit or abort vote for D on the CBC:

$\mathit{commitDeal}(D, h)$ or $\mathit{abortDeal}(D, h)$

where D is the deal identifier, and h is the *startDeal*. As usual, each voter must be in the start-of-deal *plist*.

Each party P monitors the CBC to find out when the outcome of the deal is known. Then the party transfers this information to the appropriate contract(s). If the deal commits, a party sends

$\mathit{commit}(D, h, pr)$

to its incoming assets, while if the deal aborts, the party sends

$\mathit{abort}(D, h, pr)$

to its outgoing assets. In either case P must be in the *plist*. Here pr is a proof that the requested action is correct; we discuss these proofs further in Sect. 7.4.

7.2 What could possibly go wrong?

When things go wrong, the CBC protocol permits fewer outcomes than the timelock protocol, because all compliant parties agree on whether the deal committed or aborted. For example, in the timelock protocol, parties must respect timeouts to remain compliant: a persistent denial-of-service attack may cause a party to miss a timeout, deviate from the protocol, and lose an escrowed asset. In the CBC protocol, a persistent denial-of-service attack may cause the deal to fall through by suppressing a party's vote to commit, but any affected party remains compliant, and ends up no worse off.

7.3 Correctness

The CBC protocol establishes a public ordering on parties' votes to commit or abort. Safety is satisfied because compliant parties agree on whether a deal commits or aborts. Weak liveness is satisfied because any compliant party whose assets are locked up for too long will eventually vote to abort. Strong liveness is satisfied in periods when the network is synchronous because every party votes to commit before any party votes to abort.

7.4 Cross-chain proofs

Here we briefly turn our attention from the communication and failure model of the deal protocol to the communication and failure model of possible CBC implementations.

It is easy for (active) parties to ascertain whether a deal committed or aborted; it is not so easy for myopic (passive) contracts, which cannot directly observe other blockchains, to do so.

A deal's *decisive vote* is the one that determines whether the deal commits or aborts. A straightforward approach is to present each contract with a subsequence of the CBC's blocks, starting with the deal's first *startDeal* record, and ending with its decisive vote. But how can the contract tell whether the blocks presented are really on the CBC? The answer depends partly on the kind of algorithm underlying the CBC blockchain.

7.5 Byzantine fault-tolerant consensus

Let us assume the CBC relies on *Byzantine fault-tolerant* (BFT) consensus [1,4,13,47]. BFT protocols guarantee safety even when communication is asynchronous, and they ensure liveness when communication becomes synchronous after the GST.

Blocks are approved by a known set of $3f+1$ *validators*[1] of which at most f can deviate from the protocol. (The details of how validators reach consensus on new blocks are not important here.) To support long-term fault tolerance, the blockchain is periodically *reconfigured* by having at least $2f+1$ current validators elect a new set of validators. For ease of exposition, assume each block contains the next block's group of validators and their keys.

Each block in a BFT blockchain is vouched for by a certificate containing at least $f+1$ validator signatures of that block's hash. (Any $f+1$ signatures are enough because at

[1] In proof-of-stake blockchains such as Algorand [27], the number of validators may vary.

least one of them must come from an honest validator.) This sequence of blocks and their certificates can be used as a proof. Parties must identify correct validators when putting assets in escrow, and they must make sure that their incoming escrow contracts were passed the right validators and the right *h* before voting to commit.

Checking the proof as just described is a lot of work; the proof is likely to be spread over many blocks, each containing a large number of entries. Furthermore, we cannot shorten the proof by omitting irrelevant block entries, because then a malicious party might fool a contract into making a wrong decision. But there are many ways to make BFT proofs more efficient.

A straightforward optimization is to take advantage of the fact that the CBC has validators. This allows the parties to request certificates from the CBC. Such a certificate would vouch for the current state of the deal (active, committed, aborted). This certificate alone constitutes a proof provided the original validators are still active; otherwise the party must also provide the chain of validators across each reconfiguration.

7.6 Proof-of-work (nakamoto) consensus

Existing proof-of-work blockchains (such as Bitcoin [39] or Ethereum [21]) require a synchronous model of communication. In this model, proofs of commit or abort generated by a CBC implemented using proof-of-work consensus are possible, but care is needed because such blockchains lack *finality*: any such proof might be contradicted by a later proof, although forging a later, contradictory proof becomes more expensive to the adversary the longer it waits. (Kiayias *et al.* [33,34] propose changes to standard proof-of-work protocols that would make such "proofs-of-proof-of-work" more compact.)

Here is a scenario where Alice can construct a fake "proof of abort" for a proof-of-work CBC. As soon as the deal execution starts, Alice (perhaps aided by partners in crime) privately mines a block that contains an *abort* vote from Alice. When her part of the deal is complete, however, Alice publicly sends a *commit* vote to the CBC. If, by the time all parties have voted *commit*, Alice was able to mine a private *abort* block, then Alice can use that fake proof of abort to halt outgoing transfers of her assets, while using the legitimate proof of commit to trigger incoming transfers.

In the spirit of proof-of-work, such an attack can be made more expensive by requiring a proof of commit or abort to include some number of *confirmation* blocks beyond the one containing the decisive vote, forcing Alice to outperform the rest of the CBC's miners for an extended duration. To deter rational cheaters, the number of confirmations required should vary depending on the value of the deal, implying that high-value deals would take longer to resolve than lower-value deals.

To summarize, while it is technically possible to produce commit or abort proofs from a proof-of-work CBC, the result is likely to be slow and complex. In the same way a proof-of-work blockchain can fork, a "proof-of-proof-of-work" [33] can be contradicted by a later "proof-of-proof-of-work." Similarly, to make the production of contradictory proofs expensive, the proof's difficulty must be adjusted to match the value of the assets transferred by the deal. By contrast, a BFT certificate of commit or abort is final, and independent of the value of the deal's assets.

8 Cost analysis

This section analyzes the costs associated with each of the protocols. The code shown here is written in pseudocode based on the Solidity programming language [46]. It not intended to be a detailed implementation; it is only intended to illustrate how such implementations might be organized. (For readability we have taken minor liberties with the Solidity language's syntax and semantics.)

To compare their implementation costs, we use a cost model inspired by the Ethereum [52] blockchain, currently the best-developed platform. The costs of non-PoW blockchains are likely to be similar.

To make denial-of-service attacks prohibitively expensive, virtual machines that execute contracts typically charge for each instruction executed. In Ethereum [52], this charge is expressed in terms of *gas* units, whose value (in fractions of ether, the blockchain's native currency) varies according to demand. For example, the gas cost of simple arithmetic operations or accesses to short-lived memory is in single digits, and control flow or read operations from long-lived storage is in double or triple digits. In general, gas costs are dominated by two kinds of operations: writing to long-lived storage is (usually) 5000 gas, and each signature verification is 3000 gas.

Table 4 summarizes gas costs for a deal with n parties, m assets, and $t \leq n$ transfers.

8.1 Gas costs

To illustrate our gas cost analysis, Fig. 2 shows a fragment of a pseudocode implementation of a generic **`EscrowManager`** contract for a fungible asset. (This contract follows the popular ERC20-standard [25] for fungible tokens.) The heart of the **`EscrowManager`** contract is a pair of mappings: **`escrow`** records how many tokens each party has escrowed (Line 3), and **`onCommit`** records how many tokens each party would receive if the deal commits (Line 4). For clarity, some error and sanity checking has been omitted.

Table 4 Gas costs

Protocol	Escrow	Transfer and Validation	Commit or Abort
Timelock	$O(m)$ writes	$O(t)$ writes	$O(mn^2)$ sig. ver. + $O(m)$ writes
CBC	$O(m)$ writes	$O(t)$ writes	$O(m(f+1))$ sig. ver. + $O(m)$ writes

Fig. 2 Pseudocode code for escrow an transfer (some details omitted)

```
 1  contract EscrowManager {
 2      ERC20Interface asset;                  // contract holding assets
 3      mapping(address => uint) escrow;    // escrowed assets
 4      mapping(address => uint) onCommit;   // result of tentative transfers
 5      ...
 6      // transfer into escrow account
 7      function escrow (uint amount) public {
 8          require (asset.transferFrom( msg.sender, this , amount));
 9          escrow[ msg.sender] = escrow[ msg.sender] + amount;
10          onCommit[ msg.sender] = onCommit[ msg.sender] + amount;
11      }
12      // tentative transfer
13      function transfer (address to, uint amount) public {
14          require (onCommit[ msg.sender] >= amount);
15          onCommit[ msg.sender] = onCommit[ msg.sender] - amount;
16          onCommit[to] = onCommit[to] + amount;
17      }
18      ...
19  }
```

Escrow phase

Each party calls the *escrow* function to escrow some number of tokens. This function incurs 2 storage writes (in a function call) to transfer the token from the sender to the escrow contract (Line 8), and 1 storage write each to update the **`escrow`** (Line 9) and the **`onCommit`** (Line 10) maps, for a total of 4 storage writes. Globally, the escrow phase incurs $O(m)$ gas costs.

Transfer phase

Each party calls the **`transfer`** function to transfer some number of escrowed tokens to another party. This function incurs 1 storage write to decrement the sender's tentative **`onCommit`** balance (Line 15), and another to increment the recipient's balance (Line 16). Globally, the transfer phase incurs $O(t)$ gas costs.

Validation phase

Each party monitors its incoming and outgoing escrow contracts to ensure it is satisfied with the assets it is due to acquire and relinquish. This computation takes place entirely at the parties, and incurs no gas cost.

Timelock protocol commit phase

Timelock escrow contracts verify commit path signatures. Note that signatures are generated by parties, not by contracts, so while signature generation incurs computation costs at parties, it incurs no gas costs at contracts.

Figure 3 shows a pseudocode fragment for a timelock escrow contract. The contract records the set of parties participating in the deal (Line 2) and which ones have voted (Line 2). The *commit* function takes as arguments the voter, the set of signers, and their signatures. It checks that the deal has not timed out (Line 6), that the voter is legitimate (Line 7), that the vote has not already been recorded (Line 8), and that there are no duplicate signers (Line 9). The expensive steps are verifying each of the signatures (Line 11), and recording the voter (Line 13) in long-lived storage.

Each escrow contract verifies a vote from each of n parties, and each party's vote could have been signed by up to $n-1$ others, yielding a worst-case per-contract bound of $O(n^2)$ signature verifications, plus a constant number of storage writes for other bookkeeping. Since there are m contracts, the timelock commit protocol incurs an $O(mn^2)$ global gas cost. In the best case, a deal can abort with no signature verifications, but in the worst case, aborting can cost almost as much as committing.

Fig. 3 Pseudocode fragment for timelock contract voting (some details omitted)

```
contract TimelockManager is EscrowManager{
    address [] parties ;            // participating parties
    address [] voted;               // which parties have voted
    ...
    function commit (address voter, address [] signers , bytes32 [] sigs ) public {
        require (now < start + (path.length() * DELTA)); // not timed out
        require ( parties . contains(voter ));               // legit voters only
        require (!voted.contains(voter ));                   // no duplicate votes
        require (checkUnique(signers ));                     // no duplicate signers
        for (int i = 0 ; i < signers . length; i++) {
            require (checkSig(voter , signers [ i ], sigs [ i ])); // expensive
        }
        voted.push(voter );                                  // remember who voted
    }
    ...
}
```

Fig. 4 Pseudocode fragment for CBC proof-checking (some details omitted)

```
1  contract CBCManager is EscrowManager{
2      address [] validators ;          // CBC validators
3      ...
4      // check commit proof is valid
5      function commit (address[] signers , bytes32 [] sigs ) public {
6          require (checkUnique(signers ));           // no duplicate signers
7          require ( validators . contains( signers )); // only validators voted
8          require ( signers . length >= f+1);        // enough validators voted
9          for (int i = 0 ; i < f+1; i++) {
10             require (checkSig( signers [ i ], sigs [ i ])); // expensive
11         }
12         outcome = COMMITTED;                       // remember we committed
13     }
14     ...
15 }
```

CBC protocol commit phase

Escrow contracts check proofs from the CBC by verifying that they are correctly signed by enough validators. Figure 4 shows a pseudocode fragment for an escrow contract for a CBC using an underlying BFT consensus protocol that tolerates f Byzantine validators. We assume the optimization where parties request status certificates from the CBC; for brevity, we assume there have been no reconfigurations.

The contract keeps track of the CBC's current set of validators (Line 2); it also knows their public keys. The *commit* function takes as arguments the set of signers and their signatures. It checks that there are no duplicate validators (Line 6), that all signers are validators (Line 7), and that there are enough votes (Line 8). The expensive step is verifying each of the validator signatures (Line 10); there will also be a constant number of storage writes to record the outcome and to update the escrow and ownership mappings.

Each contract verifies $f+1$ signatures, or $(k+1)(f+1)$ if the set of validators has changed k times. The global gas cost is $O(m(f+1))$ signature verifications plus a constant number of storage writes to update the escrow mappings.

8.2 Time costs

We analyze each commit protocol's timing delays when the network is synchronous, with bound Δ on the time needed both to change a blockchain state, and to have that change observed by any interested party. The results are summarized in Table 5; here we assume each asset is transferred k times.

For both protocols, if all parties are conforming, the escrow phase takes time at most Δ, since every party updates its outgoing assets' escrow contracts in parallel. Similarly, the transfer phase takes time at most $k \cdot \Delta$. It may be possible to execute transfers concurrently, in which case this phase takes time at most Δ. At the end of the transfer phase, validation is local and immediate.

Timelock protocol commit phase

If each party sends its commit vote only to the blockchains managing its incoming assets, then the worst-case duration of the commit phase is proportional to the longest sequence of transfers, which is bounded by $n\Delta$.

Table 5 Delays for synchronous communication

Protocol	Escrow	Transfer and Validation	Commit	Abort
Timelock	Δ	$k\Delta$ or Δ	$O(n)\Delta$	$O(n)\Delta$
CBC	Δ	$k\Delta$ or Δ	$O(1)\Delta$	Per-party timeout

CBC protocol commit phase

All conforming parties send their votes to the CBC in parallel, and these votes are available very quickly when the CBC is implemented using a BFT protocol such as from the PBFT family [1,13]. It requires at most another Δ for the escrow contracts to transfer or refund their assets.

9 Centralization

A commit protocol is *decentralized* if there is no single blockchain that must be accessed by all parties in any execution. A single blockchain shared by all parties in a protocol is not a safety hazard, since blockchains are tamper-proof, but the common blockchain could be the target of denial-of-service attacks, causing assets to be temporarily locked up. Moreover, parties must trust the common blockchain's validators not to engage in *censorship*, where validators selectively choose to ignore certain deals, causing them to abort when they could otherwise have committed.

The timelock protocol of Sect. 6 is decentralized in this sense because for well-formed deals each party interacts only with blockchains on its incoming and outgoing arcs. A compliant party first sends votes to the escrow contracts on its incoming arcs, then it monitors the blockchains on its outgoing arcs, and forwards new votes to its incoming arcs. In particular, there is no single blockchain that must be accessed by all compliant parties.

The CBC protocol is not decentralized, because the CBC itself acts as a common whiteboard shared by all parties.

In this section we show that some degree of centralization in the CBC protocol is inevitable: any protocol that tolerates periods of asynchrony, and in which every compliant party agrees on whether the deal committed or aborted, must include a single blockchain accessed by every party in some execution.

Consider a model in which deal protocols take place exclusively through operations on shared blockchains. Each blockchain provides two operations: a party can *read* a blockchain's current state, or it can *publish* an entry on that blockchain. A blockchain's state is just a (totally ordered) sequence of published entries that is interpreted by contracts.

By way of contradiction, let us assume we have a protocol that guarantees (1) all compliant parties agree whether the deal committed or aborted, and (2) all compliant parties decide after taking a finite number of steps.

A deal's *state* consists of the states of the compliant parties and the set of blockchain states. A deal state is *bivalent* if the deal's final status (committed vs aborted) is not yet fixed: there is some execution starting from that state in which the compliant parties all enter a final *committed* state, and one in which they all enter a final *aborted* state. A deal state is *univalent* if the deal's fate is fixed: the deal has the same fate in every execution starting from that state. A univalent state is *commit-valent* if the deal's fate is to commit, and *abort-valent* otherwise.

A deal's set of possible states forms a tree, where each tree node represents a deal state, each edge represents a step where a party reads or publishes on a blockchain, and each leaf node represents a final state where the deal has either committed or aborted. A bivalent state is a node whose descendants in the tree include leaves where the deal commits and leaves where it aborts, while a univalent state is a node whose descendant leaves assign the same fate to the deal.

It is enough to prove our claims for executions where deviating parties behave correctly up to a point and then simply halt. We place no limits on the number of deviating parties, but we do assume there are at least two conforming parties. To rule out uninteresting cases, a deal is *non-trivial* if there is some execution in which the compliant parties abort.

First, we observe that an initial bivalent state exists, meaning that the fate of the deal cannot be predetermined.

Lemma 1 *Every non-trivial deal protocol has a bivalent initial state.*

Proof If all parties are compliant, then the deal completes by strong liveness. Nevertheless, by non-triviality, there is an execution in which the deal aborts. □

A deal state is *critical* if:

- It is bivalent, and
- if any compliant party takes a step, the deal state becomes univalent.

Lemma 2 *Every non-trivial deal has a critical state.*

Proof Suppose not. By Lemma 1, the deal has a bivalent initial state. Start the protocol in this state. As long as some party can take a step without making the deal state univalent, take that step. By hypothesis, the deal cannot run forever without committing or aborting, so the deal must eventually enter a critical state. □

Theorem 4 *There is some execution in which every compliant party reads or publishes on the same blockchain.*

Proof By Lemma 2, there is an execution where the deal reaches a critical state s. We claim that in that critical state, all compliant parties must be about to publish on the same blockchain. We can assume the system is in the asynchronous phase before the GST, so no party can wait for another to take a step (because the other may be deviating and halted). In particular, any party that executes the remainder of the protocol by itself (running *solo*) must commit or abort after a finite number of steps.

We can divide the compliant parties into two sets: each party in A is about to carry the protocol to an abort-valent state, and each party in C is about to carry the protocol to a commit-valent state.

Suppose a is about to read from a blockchain, while c is about to read or publish to the same or different blockchain. Consider two possible execution scenarios. In the first, a takes the first step, driving the deal to an abort-valent state s. Party a then pauses, and c runs solo and aborts after a finite number of steps. In the second scenario, c moves first, driving the protocol to a commit-valent state s'. Party c then runs solo starting in s' and eventually commits. But s and s' are indistinguishable to c (the read a performed could only change its local state which is not visible to c), which means that c either commits in both or aborts in both executions, a contradiction. (The symmetric argument holds if c reads and a publishes.)

Suppose instead both parties are about to publish to different blockchains. Party a is about to publish on b_0 and c on b_1. Here are two execution scenarios. In the first, a publishes on b_0 and then c publishes on b_1, so the resulting protocol state is abort-valent because a went first. In the second, c publishes on b_1 and then a publishes on b_0, so the resulting protocol state is commit-valent because c went first. But both scenarios lead to indistinguishable protocol states: neither a nor c can tell which party acted first, a contradiction.

The only remaining possibility is that both a and c are about to publish on the same blockchain. Since this argument works for any $a \in A$ and $c \in C$, it follows that all compliant parties are about to publish on the same blockchain. □

10 Related work

Today, the most common way to trade electronic assets is to use a trusted third party, sometimes called a *crypto exchange*, such as Coinbase or Binance. Such exchanges are typically unregulated, and provide no guarantees of any kind. Crypto exchanges have been known to lose substantial sums to hackers [7], and even to vanish along with their customers' deposits [49].

The need for safety in adversarial commerce has inspired an outburst of interest in *cross-chain swaps* [11,12,19,31, 42,43,54,55], where each party transfers an asset to another party and halts. Cross-chain swaps are attractive because they reduce or eliminate the use of exchanges, some of which have proved to be untrustworthy [49,50]. However, we have seen that existing cross-chain swap proposals have limited expressive power: they cannot support indirect transfers, as mediated by a broker, nor can they support conditional exchanges such as auctions, where the seller exchanges assets only with the highest bidder. In general, cross-chain swaps do not support the kinds of complex "business logic" required by many kinds of modern financial deals.

To our knowledge, the only cross-chain swap protocols used in practice are *hashed timelocked contracts* [11,12, 19,42,43]. Herlihy [31] generalizes prior two-party cross-chain swap protocols to a protocol for multi-party swaps on arbitrary strongly connected directed graph. Herlihy also observes that the classical "all-or-nothing" correctness property is ill-suited to cross-chain swaps, and proposes an alternative correctness property which is more specialized than the one presented here because it is formulated explicitly in terms of direct swaps, not the more general structures permitted by cross-chain deals. For example, Herlihy assumed that any swap outcome where a party receives only partial inputs and partial outputs is unacceptable, but the notions of correctness introduced here allow parties to specify whether some such partial deal outcomes are acceptable.

The timelock commit protocol presented here has a simpler structure than the one proposed by Herlihy. That protocol used secrets held by a carefully chosen subset of parties. Our protocol replaces secrets with votes performed by everyone, so it is possible to treat all parties uniformly, and there is no need for a careful contract deployment phase. Our protocol also clarifies when parties review the transactions' final outcomes. Both commit protocols use timeout mechanisms based on path signatures.

Zakhary *et al.* [54] propose a cross-chain swap protocol for proof-of-work blockchains using a *witness blockchain* as a central coordinator. A contract on the witness chain is given a master plan for the transaction. Each participant sends this contract a proof that it has completed its part, and the contract then decides the transaction outcome. The witness chain proof of the decided transaction's outcome is sent to each participant's blockchain. Each such cross-chain proof effectively requires one blockchain to partially simulate the other, implying that the witness chain's contracts must be aware of each participant chain's internal structure, and vice versa. Moreover, as discussed earlier, proof-of-work chains must produce proofs proportional in size and computational complexity to the value of the assets traded. By contrast, in the

CBC protocol for cross-chain deals, participants' contracts must understand proofs produced by the CBC, but the CBC can treat participants as black boxes. The CBC simply tallies and orders votes, so it does not need to know anything about the deal's master plan.

Off-chain payment networks [18,28,30,40,45] and state channels [17] use hashed timelock contracts to circumvent the scalability limits of existing blockchains. They conduct repeated off-chain transactions, finalizing their net transactions in a single on-chain transaction. The use of hashed timelock contracts ensures that parties cannot be cheated if one party tries to settle an incorrect final state. Lind *et al.* [37] propose using (hardware) trusted execution environments to ease synchrony requirements. It remains to be seen whether cross-chain deals can be implemented on off-chain networks. Arwen [30] supports multiple off-chain atomic swaps between parties and exchanges, but their protocol is specialized to currency trading and does not seem to support non-fungible assets. Komodo [43] supports off-chain cross-platform payments.

Sharded blockchains [3,35] address scalability limits of blockchains by partitioning the state into multiple shards so that transactions on different shards can proceed in parallel, and support multi-step atomic transactions spanning multiple shards. An atomic transaction that spans multiple shards is executed at the client in Chainspace [3], or at the server in Omniledger [35]. In these systems a transaction represents a single trusted party and there is no support for transactions involving untrusted parties.

Chainspace [3] allows transactions to specify immutable proof contracts to be executed at the server. The proofs are used to validate client execution traces resembling optimistic concurrency control. Channels [5], an extension of Omniledger Atomix protocols, uses proofs in a two-phase protocol similar to our CBC, for atomic untrusted cross-shard single-step multi-party UTXO [32] transfers, but does not support multi-step deals or non-fungible assets.

The BAR (byzantine, altruistic and rational) computation model [2,15] supports cooperative services spanning autonomous administrative domains that are resilient to Byzantine and rational manipulations. Like Byzantine fault-tolerant systems, BAR-tolerant systems assume a bounded number of Byzantine faults, and as such do not fit the adversarial deal model, where any number of parties may be Byzantine.

The CBC somewhat resembles an *oracle* [44], a trusted data feed that reports physical-world occurrences to contracts.

The *fair exchange* problem [26,38] is a precursor to the atomic cross-chain swap problem. Alice has a digital asset Bob wants, and vice versa, and at the end of the protocol, either Alice and Bob have exchanged assets, or they both keep their assets. In the absence of blockchains, trusted, or semi-trusted third parties are required, but the roles of those trusted parties can be minimized in clever ways.

The auction example presented earlier is intentionally simplistic, serving to illustrate how a deal might have contingent outcomes. Modern auctions come in many different formats, including English, Vikray, Dutch, and others [36,41]. (A comprehensive and accessible survey of modern auction theory appears in the citation for the 2020 Nobel Prize for Economics [48].) The authors are convinced that the application of cross-chain deals to modern distributed auctions merits further study.

The proof of Theorem 4 uses techniques first proposed by Fischer, Lynch, and Paterson [23].

11 Discussion

Deals can be enhanced to provide incentives for good behavior. For example, a party might escrow a small deposit that is lost if that party is the first to cause the deal to fail. Designing and implementing such incentives is an area of ongoing research [22,53].

In the timelock commit protocol, if Δ is too small, parties may be vulnerable to an extended denial-of-service attack, which can cause them to lose their incoming assets. There is a similar threat to the CBC commit protocol, where the CBC itself might be the target of a denial of service attack, but an attack on the CBC causes the deal's assets to be locked up for the duration of the attack, not lost forever.

A more subtle issue concerning the CBC commit protocol is that the parties must trust the CBC not to engage in *censorship*, where CBC validators selectively choose to ignore or postpone certain deals, causing them to abort when they could otherwise have committed.

The main difference in performance in the timelock and CBC protocols is the number of signatures that must be verified to complete the protocol, so we need to consider n parties vs. $f+1$ validators. Many deals will likely have only a few participants, and in this case it will be more expensive to commit a CBC deal ($O(m(f+1))$) than a timelock deal ($O(mn^2)$); in a deal with many participants the reverse may be true. Even if the CBC protocol is more expensive, one gets what one pays for: the CBC protocol works in a more demanding timing model.

12 Conclusions

Today's distributed data management systems face a new and daunting challenge: enabling commerce among autonomous parties who do not know or trust one another, a model we have called *adversarial commerce*. Prior approaches, such as atomic transactions or cross-chain swaps, are inappropriate for the trust model or have limited expressive power.

In adversarial commerce, each party in a deal decides for itself whether to participate in a particular interaction. Parties agree to follow a common protocol, an agreement that can be monitored, but not enforced. Correctness is local and selfish: all parties that follow the protocol end up "no worse off" than when they started, even in the presence of faulty or malicious behavior by an arbitrary number of other parties. Examples of adversarial commerce include securities trading, digital asset management, the Internet of Things, supply chain management, and, of course, cryptocurrencies.

It is easy to confuse cross-chain deals with atomic transactions, and with cross-chain swaps, since they address similar needs. We hope this paper has clarified the critical distinctions between them, and explained why cross-chain deals are needed to address the demands of adversarial commerce.

References

1. Abraham, I., Gueta, G.,Malkhi, D: Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, abs/1803.05069 (2018)
2. Aiyer, A. S., Alvisi, L., Clement, A., Dahlin, M., Martin, J.-P., Porth, C.: BAR fault tolerance for cooperative services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 45–58, New York, NY, USA, ACM (2005)
3. Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., Danezis, G. : Chainspace: A sharded smart contracts platform. *CoRR*, abs/1708.03778 (2017)
4. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman,G., Manevich, Y., Muralidharan, S., Murthy, C. Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W., Yellick, J: Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, ACM (2018)
5. Androulaki, E., Cachin, C., Caro, A. D, Kokoris-Kogias, E.: Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *ESORICS* (2018)
6. Association, L.: An introduction to libra (2019)
7. Barrett, B.: Hack brief: hackers stole $40 million from binance cryptocurrency exchange (2019)
8. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (1986)
9. Binance Academy. What are flash loans in DeFi? (2020)
10. Binance Academy. What is an automated market maker (AMM)? (2020)
11. bitcoinwiki. Atomic cross-chain trading
12. Bowe, S., Hopwood, D. : Hashed time-locked contract transactions
13. Castro, M., Liskov, B. : Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association. Number of pages: 14 Place: New Orleans, Louisiana, USA tex.acmid: 296824
14. Chester, J.: Your guide on bitcoin's lightning network: The opportunities and the issues (2018)
15. Clement, A., Li, H., Napper, J., Martin, J.P.M., Alvisi, L., Dahlin, M.: BAR primer. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN), DCC Symposium*, Place, Anchorage, AK (2008)
16. Coindesk. Tether stablecoin launches on its seventh blockchain (2020)
17. Coleman, J., Horne, L. , Xuanji, L.: Counterfactual: generalized state channels (2018)
18. Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Pelc, A., Schwarzmann, A.A. (eds.) Stabilization. Safety, and Security of Distributed Systems, pp. 3–18. Springer, Cham (2015)
19. DeCred. Decred cross-chain atomic swapping
20. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
21. Ethereum
22. Evans, A.: A crash course in mechanism design for cryptoeconomic applications (2017)
23. Fischer, M. J., Lynch, N. A., and Paterson, M. S.: Impossibility of distributed consensus with one faulty process. J. ACM, 32(2), 374–382, 1985. Number of pages: 9 Publisher: ACM tex.acmid: 214121 tex.address: New York, NY, USA tex.issue_date (1985)
24. Ford, B., Böhme, R.: Rationality is Self-Defeating in Permissionless Systems. arXiv:1910.08820 [cs], arXiv:1910.08820 (2019)
25. Foundation, E.: ERC20 token standard (2019)
26. Franklin, M.K. and Tsudik, G.: Secure group barter: multi-party fair exchange with semi-trusted neutral parties. In *Financial Cryptogrphy* (1998)
27. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, ACM (2017)
28. Green, M., Miers, I., Bolt: Anonymous payment channels for decentralized currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017)
29. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4), 287–317 (1983)
30. Heilman, E., Lipmann, S., Goldberg, S.: The arwen trading protocols (2019)
31. Herlihy, M.: Atomic cross-chain swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 245–254, New York, NY, USA. ACM. Number of pages: 10 Place: Egham, United Kingdom tex.acmid: 3212736 (2018)
32. Investopedia. UTXO (2019)
33. Kiayias, A., Lamprou, N., Stouka, A.-P.: Proofs of proofs of work with sublinear complexity. In *International Conference on Financial Cryptography and Data Security* (2016)
34. Kiayias, A., Miller, A., Zindros, D.: Non-interactive Proofs of Proof-of-Work. In: Bonneau, J., Heninger, N. (eds.) Financial Cryptography and Data Security. Lecture Notes in Computer Science, pp. 505–522. Springer International Publishing, Cham (2020)

35. Kokoris Kogias, E., Jovanovic, P.S., Gasser, L., Gailly, N., Syta, E., Ford, B.A., OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, page 16 (2018)
36. Krishna, V.: *Auction theory*. Academic Press/Elsevier, Burlington, MA, 2nd ed edition, OCLC: ocn326688263 (2010)
37. Lind, Y., Eyal, I., Kelbert, F., Naor, O., Pietzuch, P.R., Sirer, E.G., Teechain: Scalable blockchain payments using trusted execution environments. *CoRR*, abs/1707.05454 (2017)
38. Micali, S.: Simple and fast optimistic protocols for fair electronic exchange. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 12–19, New York, NY, USA. ACM. Number of pages: 8 Place: Boston, Massachusetts tex.acmid: 872038 (2003)
39. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009)
40. Network, R.: What is the raiden network?
41. Nisan, N.: (Eds) *Algorithmic Game Theory*. Cambridge University Press, Cambridge ; New York. OCLC: ocn122526907 (2007)
42. Nolan, T.: Atomic swaps using cut and choose (2016)
43. Organization, T.K.: The BarterDEX whitepaper: a decentralized, open-source cryptocurrency exchange, powered by atomic-swap technology
44. Peterson, J., Krug, J., Zoltu, M., Williams, A.K., Alexander, S., Augur: a decentralized oracle and prediction market platform (2018)
45. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016)
46. Solidity documentation
47. Tendermint (2015)
48. The Committee for the Prize in Economic Sciences in Memory of Alfred Nobel. Improvements to Auction Theory and Inventions of New Auction Formats (2020)
49. Wikipedia. Mt. Gox (2019)
50. Wikipedia. Quadriga fintech solutions (2019)
51. Wikipedia. Tether (cryptocurrency) (2020)
52. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)
53. Xue, Y., Herlihy, M.: Hedging against sore loser attacks in cross-chain transactions. In *ACM Symposium on Principles of Distributed Computing* (2021)
54. Zakhary, V., Agrawal, D., El Abbadi, A.: Atomic commitment across blockchains. *CoRR* (2019). ArXiv:1905.02847. tex.bibsource: dblp computer science bibliography, https://dblp.org tex.biburl: https://dblp.org/rec/bib/journals/corr/abs-1905-02847 tex.timestamp (2019)
55. Zyskind, G., Kisagun, C., FromKnecht, C.: Enigma Catalyst: a machine-based investing platform and infrastructure for crypto-assets