Channels: Horizontal Scaling and Confidentiality on Permissioned Blockchains

Elli Androulaki[1], Christian Cachin[1], Angelo De Caro[1], and Eleftherios Kokoris-Kogias[2](✉)

[1] IBM Research - Zurich, Rüschlikon, Switzerland
{lli,cca,adc}@zurich.ibm.com
[2] EPFL, Lausanne, Switzerland
eleftherios.kokoriskogias@epfl.ch

Abstract. Sharding, or partitioning the system's state so that different subsets of participants handle it, is a proven approach to building distributed systems whose total capacity scales horizontally with the number of participants. Many distributed ledgers have adopted this approach to increase their performance, however, they focus on the permissionless setting that assumes the existence of a strong adversary. In this paper, we deploy channels for permissioned blockchains. Our first contribution is to adapt sharding on asset-management applications for the permissioned setting, while preserving liveness and safety even on transactions spanning across-channels. Our second contribution is to leverage channels as a confidentiality boundary, enabling different organizations and consortia to preserve their privacy within their channels and still be part of a bigger collaborative ecosystem. To make our system concrete we map it on top of Hyperledger Fabric.

1 Introduction

Blockchain technology is making headlines due to its promise of a transparent, verifiable, and tamper-resistant history of transactions that is resilient to faults or influences of any single party [3]. Many organizations [2,4,15,22] either explore the potential of distributed-ledger technology or already embrace it. This, however, is a young technology facing multiple challenges [3,6]. In this paper we look into the challenges of enabling horizontal scaling and providing privacy in the permissioned setting.

First, the scalability of distributed ledgers hinders their mainstream adoption. One class of solutions proposed is sharding [6]. Sharding [20] has been used in order to build *scale-out* systems whose capacity scales horizontally with the number of participants by using the key idea of partitioning the state. Each such state partition can handle transactions parallel to other shards. Recently, several blockchain systems [7,12] proposed sharding mostly in the context of

E. Kokoris-Kogias—Work done at IBM Research - Zurich.

J. Lopez et al. (Eds.): ESORICS 2018, LNCS 11098, pp. 111–131, 2018.
https://doi.org/10.1007/978-3-319-99073-6_6

permissionless blockchains, where some fraction of participating parties might be Byzantine.

A second challenge for distributed ledgers is privacy. A distributed ledger is (by design) a transparent log visible to all the participants. This, however, is a disadvantage when it comes to deploying distributed ledgers among private companies, as they want to keep their data confidential and only selectively disclose them to vetted collaborators. One solution to privacy is to hide the state from all participants by using zero-knowledge proofs [10,13,16]. However, this can pose a problem in a permissioned setting both in terms of performance (especially if the system supports smart contracts) and in terms of business logic (*e.g.,* banks need to see the transactions to balance their books).

In this paper, we look into enabling sharding in the permissioned setting, where the adversarial power can be relaxed. First we deploy channels for horizontal scaling drawing inspiration from the state of the art [7,12], but at the same time navigating the functionality and trust spectrum to create simplified protocols with less complexity and need for coordination. Then, we introduce the idea that, in a permissioned setting, we can leverage the state partition that a channels introduces as a confidentiality boundary. In the second part of the paper, we show how we enable confidential channels while preserving the ability for cross-shard transactions.

Our main contributions are (a) the support for horizontal scaling on permissioned blockchains whith cross-channel transaction semantics, (b) the use of channels as a confidentiality boundary and (c) the formalization of an asset management application on top of blockchain systems.

2 Preliminaries

Blockchain Definitions. In the context of this work, a blockchain is an append-only tamper-evident log maintained by a distributed group of collectively trusted nodes. When these nodes are part of a defined set [1], we call the blockchain *permissioned*. Inside every block there are transactions that may modify the state of the blockchain (they might be invalid [1]). A distributed ledger [23] is a generalization of a blockchain as it can include multiple blockchains that interact with each other, given that sufficient trust between blockchains exists.

We define the following roles for nodes in a blockchain:

1. **Peers** execute and validate transactions. Peers store the blockchain and need to agree on the state.
2. **Orderers** collectively form the ordering service. The ordering service establishes the total order of transactions. Orderers are unaware of the application state, and do not participate in the execution or validation of transactions. Orderers reach consensus [1,5,11,17] on the blocks in order to provide a deterministic input for the blockchain peers to validate transactions.
3. **Oracles** are special nodes that provide information about a specific blockchain to nodes not being peers of that blockchain. Oracles come with

a *validation policy* of the blockchain defining when the announcement of an oracle is trustworthy[1].

4. **(Light) Clients** submit transactions that either read or write the state of a distributed ledger. Clients do not directly subscribe to state updates, but trust some oracles to provide the necessary proofs that a request is valid.

Nodes can implement multiple roles or collapse roles (*e.g.,* miners in Bitcoin [17] are concurrently peers and orderers). In a distributed ledger that supports multiple blockchains that interoperate the peers of one blockchain necessarily implement a client for every other blockchain and trust the oracles to provide proofs of validity for cross-channel transaction. A specific oracle instantiation can be for example that a quorum (*e.g.,* $\frac{2}{3}$) of the peers need to sign any announcement for it to be valid.

Channels: In this paper we extend channels (first introduced in Hyperledger Fabric [1]), an abstraction similar to shards. In prior work [1], a channel is defined as an autonomous blockchain agnostic to the rest of the state of the system. In this work, we redefine a channel as a state partition of the full system that (a) is autonomously managed by a (logically) separate set of peers (but is still aware of the bigger system it belongs) and (b) optionally hides the internal state from the rest of the system.

A channel might communicate with multiple other channels; and there needs to be some level of trust for two channels to transact. Hence, we permit each channel to decide on what comprises an authoritative proof of its own state. This is what we call **validation policy**: clients need to verify this policy in order to believe that something happened in a channel they are transacting with. When channel A wants to transact with channel B, then the peers of A effectively implement a client of channel B (as they do not know the state of B directly). Thus, the peers of A verify that the validation policy of B is satisfied when receiving authoritative statements from channel B.

For channels to interact, they need to be aware of each other and to be able to communicate. Oracles are responsible for this functionality, as they can gossip authoritative statements (statements supported by the validation policy) to the oracles of the other channels. This functionality needs a bootstrap step where channels and validation policies are discovered, which we do not address in this paper. A global consortium of organizations could publicly announce such information; or consortia represented by channels could communicate off-band. Once a channel is established further evolution can be done without a centralized intermediary, by using skipchains [18].

Threat Model: The peers that have the right to access one channel's state are trusted for confidentiality, meaning that they will not leak the state of the channel on purpose. We relax this assumption later providing forward and backward

[1] *e.g.,* in Bitcoin the oracles will give proofs that have 6 Proofs-of-Work build on top of them.

secrecy in case of compromise. We assume that the ordering service is secure, produces a unique blockchain without forks and the blocks produced are available to the peers of the channels. We further assume that the adversary is computationally bounded and that cryptographic primitives (*e.g.,* hash functions and digital signatures) are secure.

System Goals: We have the following primary goals.

1. **Secure transactions.** Transactions are committed atomically or eventually aborted, both within and across channels.
2. **Scale-out.** The system supports state partitions that can work in parallel, if no dependencies exist.
3. **Confidentiality.** The state of a channel remains internal to the channel peers. The only (if any) state revealed for cross-channel transactions should be necessary to verify that a transaction is valid (e.g. does not create new assets).

3 Asset Management in a Single Channel

3.1 Unspent Transaction-Output Model

In this section, we describe a simple asset-management system on top of the *Unspent Transaction-Output* model (henceforth referred to as UTXO) that utilizes a single, non- confidential channel. In particular, we focus on the UTXO-based data model [17], as it is the most adopted data model in cryptocurrencies, for its simplicity and parallelizability.

Assets in Transactions. In a UTXO system, transactions are the means through which one or more *virtual* assets are managed. More specifically, *mint* transactions signify the introduction of new assets in the system and *spend* transactions signify the change of ownership of an asset that already exists in the system. If an asset is *divisible*, i.e., can be split into two or more assets of measurable value, then a *spend* transaction can signify such a split, indicating the owners of each resulting component of the original asset.

Assets are represented in the transactions by transaction *inputs* and *outputs*. More specifically, in the typical UTXO model, an *input* represents the asset that is to be spent and an *output* represents the new asset that is created in response of the input assets' consumption. We can think of inputs and outputs representing different phases of the state of the same asset, where state includes its ownership (shares). Clearly, an input can be used only once, as after being spent, the original asset is substituted by the output assets, and stops being considered in the system. To ensure the single-spending of any given input, transactions are equipped with information authenticating the transaction creators as the owners of the (parts of the) assets that are referenced by the transaction inputs.

In more technical terms in the standard UTXO model, *input* fields implicitly or explicitly reference *output* fields of other transactions that have not yet been spent. At validation time, verifiers would need to ensure that the outputs referenced by the inputs of the transaction have not been spent; and upon transaction-commitment deem them as spent. To efficiently look up the status of each output at validation time UTXO model is equipped with a pool of *unspent transaction outputs* (UTXO pool).

UTXO Pool. The UTXO pool is the list of transaction outputs that have not yet been *spent*. We say that an output is *spent* if a transaction that references it in its inputs is included in the list of ledger's valid transactions.

To validate a transaction, peers check if (1) the transaction inputs refer to outputs that appear in the UTXO pool as well as (2) that the transaction's creators own these outputs. Other checks take place during the transaction validation, *i.e.,* input-output consistency checks. After these checks are successfully completed, the peers mark the outputs matching the transaction's inputs as spent and add to the pool the freshly created outputs. Hence, the pool consistently includes "unspent" outputs.

Asset or Output Definition. An **asset** is a logical entity that sits behind transaction outputs, implicitly referenced by transaction outputs. As such the terms output and asset can be used interchangeably. An output (the corresponding asset) is described by the following fields:

- *namespace*, the namespace the output belongs to (e.g., a channel);
- *owner*, the owner of the output
- *value*, the value of the asset the output represents (if divisible);
- *type*, the type of the asset the output represents (if multiple types exist).

Depending on the privacy requirements and properties of the ledger they reside, outputs provide this information in the clear (e.g., Bitcoin [17] outputs) or in a concealed form (e.g., ZeroCoin [16], ZeroCash [21]). Privacy-preserving outputs are required to be cryptographically bound to the value of each of the fields describing them, whereas its plaintext information should be available to the owner of the output.

UTXO Operations. We elaborate on the UTXO system functions where we adopt the following notation. For a sequence of values $x_1, \dots, x_i$, we use the notation $[x_i] = (x_1, \dots, x_i)$. By slight abuse of notation, we write $x_1 = [x_1]$. We denote algorithms by sans-serif fonts. Executing an algorithm algo on input y is denoted as $y \leftarrow \mathsf{algo}(x)$, where y can take on the special value $\perp$ to indicate an error.
A UTXO system exposes the following functions:

- $\langle \mathcal{U}, pool \rangle \leftarrow \mathsf{Setup}(\kappa)$ that enables each user to issue one or more identities by using security parameter κ. Henceforth, we denote by sec_{user} the secret information associated to a user with identity *user*. Setup also generates privileged identities, i.e., identities allowed to mint assets to the system, denoted as *adm*. Finally Setup initialises the pool *pool* to $\emptyset$ and returns the set of users in the system $\mathcal{U}$ and *pool*.

- $\langle out, sec_{out}\rangle \leftarrow \mathsf{ComputeOutput}(nspace, owner, value, type)$, to obtain an output representing the asset state as reflected in the function's parameters. That is, the algorithm would produce an output that is bound to namespace *nspace*, owned by *owner*, and represents an asset of type *type*, and value *value*. As mentioned before, depending on the nature of the system the result of the function could output two output components, one that is to be posted on the ledger as part of a transaction (*out*) and a private part to be maintained at its owner side (sec_{out}).
- $ain \leftarrow \mathsf{ComputeInput}(out, sec_{out}, pool)$, where, on input an asset pool *pool*, an output *out*, and its respective secrets, the algorithm returns a representation of the asset that can be used as transaction input *ain*. In Bitcoin, an input of an output is a direct reference to the latter, i.e., it is constructed to be the hash of the transaction where the output appeared in the ledger, together with the index of the output. In ZeroCash, an input is constructed as a combination of a serial number and a zero-knowledge proof that the serial corresponds to an unspent output of the ledger.
- $tx \leftarrow \mathsf{CreateTx}([sec_{owner_i}], [ain_i], [out_j])$, that creates a transaction *tx* to request the consummation of inputs $\{ain_k\}_{k=1}^{i}$ into outputs $\{out_k\}_{k=1}^{j}$. The function takes also as input the secrets of the owners of the outputs referenced by the inputs and returns *tx*. Notice that the same function can be used to construct *mint* transactions, where the input gives its place to the freshly introduced assets description.
- $pool' \leftarrow \mathsf{ValidateTx}(nspace, tx, pool)$, that validates transaction inputs w.r.t. pool *pool*, and their consistency with transaction outputs and namespace *nspace*. It subsequently updates the pool with the new outputs and spent inputs and returns its new version $pool'$. Input owner of mint transactions is the admin *adm*.

Properties: Regardless of its implementation, an asset management system should satisfy the properties defined below:

- *Validity.* Let *tx* be a transaction generated from a valid input *ain* according to some pool *pool*, i.e., generated via a successful call to $tx \leftarrow \mathsf{CreateTx}(sec_{owner}, ain, out')$, where $ain \leftarrow \mathsf{ComputeInput}(out, sec_{out}, pool)$, *owner* is the owner of out', and $out' \notin pool$. Validity requires that a call to $pool' \leftarrow \mathsf{ValidateTx}(tx, pool)$ succeeds, i.e. $pool' \neq \perp$, and that $pool' = (pool \setminus \{out\}) \cup \{out'\}$.
- *Termination.* Any call to the functions exposed by the system eventually return the expected return value or $\perp$.
- *Unforgeability.* Let an output $out \in pool$ with corresponding secret sec_{out} and owner secret sec_{owner} that is part of the UTXO pool *pool*; unforgeability requires that it is computationally hard for an attacker without sec_{out} and

sec_{owner} to create a transaction tx such that $\mathsf{ValidateTx}(nspace, tx, pool)$ will not return $\perp$, and that would mark out as spent.

- *Namespace consistency.* Let an output corresponding to a namespace $nspace$ of a user $owner$. Namespace consistency requires that the adversary cannot compute any transaction tx referencing this output, and succeed in $\mathsf{ValidateTx}(nspace', tx, pool)$, where $nspace' \neq nspace$.
- *Balance.* Let a user $owner$ owning a set of unspent outputs $[out_i] \in pool$. Let the collected value of these outputs for each asset type τ be $value_\tau$. Balance property requires that $owner$ cannot spend outputs of value more than $value_\tau$ for any asset type τ, assuming that it is not the recipient of outputs in the meantime, or colludes with other users owning more outputs. Essentially, it cannot construct a set of transactions $[tx_i]$ that are all accepted when sequentially[2] invoking $\mathsf{ValidateTx}(tx, pool)$ with the most recent versions of the pool $pool$, such that $owner$ does not appear as the recipient of assets after the acquisition of $[out_i]$, and the overall spent value of its after that point exceeds for some asset type τ $value_\tau$.

3.2 Protocol

We defined an asset output as, $out = \langle nm, o, t, v\rangle$, where nm is a namespace of the asset, o is the identity of its owner, t the type of the asset, and v its value. In its simplest implementation the UTXO pool would be implemented as the list of available outputs, and inputs would directly reference the outputs in the pool, e.g., using its hash[3]. Clearly a valid transaction for out's spending would require a signature with sec_o.

Asset Management in a Single Channel. We assume two users Alice and Bob, with respective identities$\langle A, sec_A\rangle$ and $\langle B, sec_B\rangle$. There is only one channel ch in the system with a namespace ns_{ch} associated with ch, where both users have permission to access. We also assume that there are system administrators with secrets sec_{adm} allowed to mint assets in the system, and that these administrators are known to everyone.

Asset Management Initialization. This requires the setup of the identities of the system administrators[4]. For simplicity, we assume there is one asset management administrator, $\langle adm, sec_{adm}\rangle$. The pool is initialized to include no assets, i.e., $pool_{ch} \leftarrow \emptyset$.

Asset Import. The administrator creates a transaction tx_{imp}, as:

$$tx_{imp} \leftarrow \langle \emptyset, [out_n], \sigma\rangle,$$

[2] This is a reasonable assumption, given we are referring to transactions appearing on a ledger.

[3] Different approaches would need to be adopted in cases where unlinkabiltiy between outputs and respective inputs is required.

[4] Can be a list of identities, or policies, or mapping between either of the two and types of assets.

where $out_k \leftarrow \mathsf{ComputeOutput}(ns_{ch}, u_k, t_k, v_k)$, (t_i, v_i) the type and value of the output asset out_k, u_k its owner and σ a signature on transaction data using sk_{adm}. Validation of tx_{imp} would result into $pool_{ch} \leftarrow \{pool_{ch} \cup \{[out_n]\}\}$.

Transfer of Asset Ownership. Let $out_A \in pool_{ch}$ be an output owned by Alice, corresponding a description $\langle ns_{ch}, A, t, v\rangle$. For Alice to move ownership of this asset to Bob, it would create a transaction

$$tx_{move} \leftarrow \mathsf{CreateTx}(sec_A; ain_A, out_B),$$

where ain_A is a reference of out_A in $pool_{ch}$, and $out_B \leftarrow \mathsf{ComputeOutput}$ (ns_{ch}, B, t, v), the updated version of the asset, owned by Bob. tx_{move} has the form of $\langle ain_A, out_B, \sigma_A\rangle$ is a signature matching A. At validation of tx_{move}, $pool_{ch}$ is updated to no longer consider out_A as unspent, and include the freshly created output out_B:

$$pool_{ch} \leftarrow (pool_{ch} \setminus \{out_A\}) \cup \{out_B\}\ .$$

Discussion: The protocol introduced above does provide a "secure" (under the security properties described above) asset management application within a single channel. More specifically, the *Validity* property follows directly from correctness of the application where a transaction generated by using a valid input representation will be successfully validated by the peers after it is included in an ordered block. The *Unforgeability* is guaranteed from the requirement of a valid signature corresponding to the owner of the consumed input when calling the ValidateTx function, and *Namespace consistency* is guaranteed as there is only one namespace in this setting. *Termination* follows from the liveness guarantees of the validating peers and the consensus run by orderers. Finally, *Balance* also follows from the serial execution of transactions that will spend the *out* the first time and return $\perp$ for all subsequent calls (there is no *out* in the pool).

The protocol can be extended to naively scale-out. We can create more than one channel (each with its own namespace), where each one has a separate set of peers and each channel is unaware of the existence of other channels. Although each channel can have its own ordering service, it has been shown in 1 [1], that the ordering service does not constitute a bottleneck. Hence, we assume that channels share the ordering service.

The naive approach has two shortcomings. First, assets cannot be transferred between channels, meaning that value is "locked" within a channel and is not free to flow wherever its owner wants. Second, the state of each channel is public as all transactions are communicated in plaintext to the orderers who act as a global passive adversary.

We deal with these problems by introducing (i) a step-wise approach on enabling cross-channel transactions depending on the functionality required and the underlying trust model (See, Sect. 4), and (ii) the notion of confidential channels (see Sect. 5). Further, for confidential channels to work we adapt our algorithms to provide confidentiality while multiple confidential channels transact atomically.

4 Atomic Cross-Channel Transactions

In this section, we describe how we implement cross-channel transactions in permissioned blockchains (that enable the scale-out property as shown in prior work [12]). We introduce multiple protocols based on the functionality required and on the trust assumptions (that can be relaxed in a permissioned setting). First, in Sect. 4.1, we introduce a narrow functionality of 1-input-1-output transactions where Alice simply transfers an asset to Bob. Second, in Sect. 4.2, we extend this functionality to arbitrary transactions but assume the existence of a trusted channel among the participants. Finally, in Sect. 4.3, we lift this assumption and describe a protocol inspired by two-phase commit [24]. These protocols do not make timing assumptions but assume the correctness of the channels to guarantee fairness, unlike work in atomic cross-chain swaps [8].

Preliminaries. We assume two users Alice (u_a), and Bob (u_b). We further assume that each channel has a validation policy and a set of oracles (as defined in Sect. 2). We assume that each channel is aware of the policies and the oracles that are authoritative over the asset-management systems in each of the rest of the channels.

Communication of Pools Content Across Channels. On a regular basis, each channel advertises its pool content to the rest of the channels. More specifically, the oracles of the asset management system in each channel are responsible to regularly advertise a commitment of the content of the channel's pool to the rest of the channels. Such commitments can be the full list of assets in the pool or, for efficiency reasons, the Merkle root of deterministically ordered list of asset outputs created on that channel.

For the purpose of this simplistic example, we assume that for each channel ch_i, a commitment (*e.g.,* Merkle root) of its pool content is advertised to all the other channels. That is, each channel ch_i maintains a table with the following type of entries: $\langle ch_j, cmt_j \rangle, j \neq i$, where cmt_j the commitment corresponding to the pool of channel with identifier ch_j. We will refer to this pool by $pool_j$.

4.1 Asset Transfer Across Channels

Let out_A be an output included in the unspent output pool of ch_1, $pool_1$, corresponding to

$$out_A \leftarrow \mathsf{ComputeOutput}(ch_1, u_a, t, v)$$

i.e., an asset owned by Alice, active on ch_1. For Alice to move ownership of this asset to Bob and in channel with identifier ch_2, she would first create a new asset for Bob in ch_2 as

$$out_B \leftarrow \mathsf{ComputeOutput}(ch_2, u_b, t, v)$$

she would then create a transaction

$$tx_{move} \leftarrow \mathsf{CreateTx}(sec_A; ain_A, out_B),$$

where ain_A is a reference of out_A in $pool_1$. Finally, sec_A is a signature matching pk_A, and ownership transfer data.

At validation of tx_{move}, it is first ensured that $out_A \in pool_1$, and that $out_A.namespace = ch_1$. out_A is then removed from $pool_1$ and out_B is added to it, i.e.,

$$pool_1 \leftarrow (pool_1 \setminus \{out_A\}) \cup \{out_B\} .$$

Bob waits till the commitment of the current content of $pool_1$ is announced. Let us call the latter $view_1$. Then Bob can generate a transaction "virtually" spending the asset from $pool_1$ and generating an asset in $pool_2$. The full transaction will happen in ch_2 as the spend asset's namespace is ch_2. More specifically, Bob creates an input representation

$$\{ain_B\} \leftarrow \mathsf{ComputeInput}(out_B; sec_B, \pi_B)$$

of the asset out_B that Alice generated for him. Notice that instead of the pool, Bob needs to provide π_B, we explain below why this is needed to guarantee the balance property. Finally, Bob generates a transaction using ain_B.

To be ensured that the out_B is a valid asset, Bob needs to be provided with a proof, say π_B, that an output matching its public key and ch_2 has entered $pool_1$, matching $view_1$. For example, if $view_1$ is the root of the Merkle tree of outputs in $pool_1$, π_B could be the sibling path of out_B in that tree with out_B. This proof can be communicated from the oracles of ch_1 to the oracles of ch_2 or be directly pulled by Bob and introduced to ch_2. Finally, in order to prevent Bob from using the same proof twice (*i.e.,* perform a replay attack) $pool_2$ need to be enhanced with a set of spent cross-transaction outputs (ScTXOs) that keep track of all the output representations out_X that have been already redeemed in another tx_{cross}. The out_B is extracted from π_B.

Validity property holds by extending the asset-management protocol of every channel to only accept transactions that spend assets that are part of channel's name-space. *Unforgeability* holds as before, due to the requirement for Alice and Bob to sign their respective transactions. *Namespace Consistency* holds as before, as validators of each channel only validate consistent transactions; and *Termination* holds because of the liveness guarantees of ch_1 and ch_2 and the assumption that the gossiped commitments will eventually arrive at all the channels. Finally, the *Balance* property holds as Alice can only spent her asset once in ch_1, which will generate a new asset not controlled by Alice anymore. Similarly, Bob can only use his proof once as out_B will be added in the ScTXO list of $pool_2$ afterwards.

4.2 Cross-Channel Trade with a Trusted Channel

The approach described above works for cases where Alice is altruistic and wants to transfer an asset to Bob. However, more complicated protocols (e.g fair exchange) are not supported, as they need atomicity and abort procedures in place. For example, if Alice and Bob want to exchange an asset, Alice should be able to abort the protocol if Bob decides to not cooperate. With the current protocol this is not possible as Alice assumes that Bob wants the protocol to finish and has nothing to win by misbehaving.

A simple approach to circumvent this problem is to assume a commonly trusted channel ch_t from all actors. This channel can either be an agreed upon "fair" channel or any of the channels of the participants, as long as all participants are able to access the channel and create/spend assets on/from it. The protocol uses the functionality of the asset transfer protocol described above (Sect. 4.1) to implement the Deposit and Withdraw subprotocols. In total, it exposes three functions and enables a cross-channel transaction with multiple inputs and outputs:

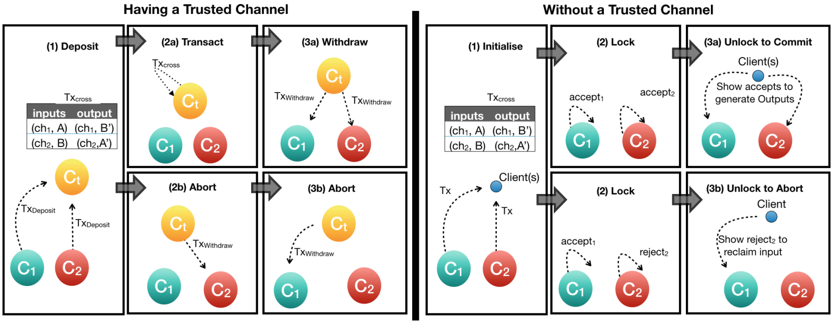
1. **Deposit:** All parties that contribute inputs transfer the assets to ch_t but maintain control over them by assigning the new asset in ch_t on their respective public keys.
2. **Transact:** When all input assets are created in ch_t, a tx_{cross} is generated and signed by all ain owners. This tx_{cross} has the full logic of the trade. For example, in the fair exchange it will have two inputs and two outputs. This tx_{cross} is validated as an atomic state update in ch_t.
3. **Withdraw:** Once the transaction is validated, each party that manages an output transfers their newly minted assets from ch_t to their respective channels ch_{o_i}.

Any input party can decide to abort the protocol by transferring back the input asset to their channel, as they always remain in control of the asset.

The protocol builds on top of the asset-transfer protocol and inherits its security properties to the extent of the Deposit and Withdraw sub-protocols. Furthermore, the trusted channel is only trusted to provide the necessary liveness for assets to be moved across channels, but it cannot double-spent any asset as they still remain under the control of their rightful owners (bound to the owner's public key). As a result, the asset-trade protocol satisfies the asset-management security requirements because it can be implemented by combining the protocol of Sect. 4.1 for the "Transact" function inside ch_t and the asset-transfer protocol of Sect. 4.2 for "Withdraw" and"Deposit" (Fig. 1).

4.3 Cross-Channel Trade Without a Trusted Channel

A mutually trusted channel (as assumed above), where every party is permitted to generate and spend assets, might not always exist; in this section, we describe a protocol that lifts this assumption. The protocol is inspired by the Atomix protocol [12], but addresses implementation details that are ignored in Atomix, such as how to represent and communicate proofs, and it is more specialized to our asset management model.



inputs	output
(ch1, A)	(ch1, B')
(ch2, B)	(ch2,A')

inputs	output
(ch1, A)	(ch1, B')
(ch2, B)	(ch2,A')

Fig. 1. Cross-channel transaction architecture overview with (4.2) and without (4.3) a trusted channel

1. **Initialize.** The transacting parties create a tx_{cross} whose inputs spend assets of some input channels (ICs) and whose outputs create new assets in some output channels (OCs). More concretely.
   If Alice wants to exchange out_A from ch_1 with Bob's out_B from ch_2. Alice and Bob work together to generate the tx_{cross} as

   $$tx_{cross} \leftarrow \mathsf{CreateTx}([sec_A, sec_B]; [ain_A, ain_B]; [out_A, out_B])$$

   where ain_A, ain_B are the input representations that show the assets to exist in the respective pools.
2. **Lock.** All input channels internally spend the assets they manage and generate a new asset bound to the transaction (we call it the "locked" asset), by using a collision resistant Hash function to derive the name-space of the new asset, as $H(tx_{cross})$[5]. The locked asset's value is either equal to the sum of the assets previously spent for that channel or 0, depending on whether the tx_{cross} is valid according to the current state. In both cases there is a new asset added in $pool_i$. Or in our example:
   Alice submits tx_{cross} to ch_2, which generates the "locked" asset for tx_{cross}. Alice then receives π_B, which shows that out_B is locked for tx_{cross} and is represented by $out_{B'}$, which is the locked asset that is generated specifically for tx_{cross} and is locked for Alice but not spendable by Alice. Specifically,

   $$asset_{2'} = \langle H(tx_{cross}), t, v \rangle,$$

   where v is either equal to the value of $asset_2$ or 0, depending on whether $asset_2$ was already spent. Same process happens for Bob. Notice that the namespace of the asset change to $H(tx_{cross})$ indicates that this asset can only be used as proof of existence and not spent again in ch_2.
3. **Unlock.** Depending on the outcome of the lock phase, the clients are able to either commit or abort their transaction.

[5] The transaction's hash is an identifier for a virtual channel created only for this transaction.

(a) **Unlock to Commit.** If all ICs accepted the transaction (generated locked assets with non-zero values), then the respective transaction can be committed.
Each holder of an output creates an unlock-to-commit transaction for his channel; it consists of the lock transaction and an oracle-generated proof for each input asset (e.g. against the gossiped MTR). Or in our example: Alice (and Bob respectively) collects $\pi_{A'}$ and $\pi_{B'}$ which correspond to the proofs of existence of $out_{A'}, out_{B'}$ and submits in ch_1 an unlock-to-commit transaction:

$$tx_{uc} \leftarrow \mathsf{CreateTx}([\pi_{A'}, \pi_{B'}]; [ain_{1'}, ain_{2'}]; [out_{A''}];)$$

The transaction is validated in ch_1 creating a new asset ($out_{A''}$), similar to the one Bob spent at ch_2, as indicated by tx_{cross}.

(b) **Unlock to Abort.** If, however, at least one IC rejects the transaction (due to a double-spent), then the transaction cannot be committed and has to abort. In order to reclaim the funds locked in the previous phase, the client must request the involved ICs that already spent their inputs, to re-issue these inputs. Alice can initiate this procedure by providing the proof that the transaction has failed in ch_2. Or in our case if Bob's asset validation failed, then there is an asset $out_{B'}$ with zero value and Alice received from ch_2 the respective proof $\pi'_{B'}$. Alice will then generate an unlock-to-abort transaction:

$$tx_{ua} \leftarrow \mathsf{CreateTx}([\pi_{B'}], [ain_{2'}]; [out_{A''}])$$

which will generate a new asset $out_{A''}$ that is identical to out_A and remains under the control of Alice.

Security Arguments: Under our assumptions, channels are collectively honest and do not fail hence propagate correct commitments of their pool (commitments valid against the validation policy).

Validity and *Namespace Consistency* hold because every channel manages its own namespace and faithfully executes transactions. *Unforgeability* holds as before, due to the requirement for Alice and Bob to sign their respective transactions and the tx_{cross}.

Termination holds if every tx_{cross} eventually commits or aborts, meaning that either a transaction will be fully committed or the locked funds can be reclaimed. Based on the fact that all channels always process all transactions, each IC eventually generates either a commit-asset or an abort-asset. Consequently, if a client has the required number of proofs (one per input), then the client either holds all commit-assets (allowing the transaction to be committed) or at least one abort-asset (forcing the transaction to abort), but as channels do not fail, the client will eventually hold enough proof. Termination is bound to the assumption that some client will be willing to initiate the unlock step, otherwise his assets will remain unspendable. We argue that failure to do such only results in harm of the asset-holder and does not interfere with the correctness of the asset-management application.

Finally, *Balance* holds as cross-channel transactions are atomic and are assigned to specific channels who are solely responsible for the assets they control (as described by validity) and generate exactly one asset. Specifically, if all input channels issue an asset with value, then every output channel unlocks to commit; if even one input channel issues an asset with zero value, then all input channels unlock to abort; and if even one input shard issues an asset with zero value, then no output channel unlocks to commit. As a result, the assigned channels do not process a transaction twice and no channel attempts to unlock without a valid proof.

5 Using Channels for Confidentiality

So far we have focused on enabling transactions between channels that guarantee fairness among participants. This means that no honest participant will be worse off by participating in one of the protocols. Here, we focus on providing confidentiality among the peers of a channel, assuming that the orderers upon which the channel relies for maintaining the blockchain are not fully trusted hence might leak data.

Strawman Solution. We start with a simple solution that can be implemented with vanilla channels [1]. We define a random key k and a symmetric encryption algorithm that is sent in a private message to every participating peer. All transactions and endorsements are encrypted under k then sent for ordering, hence the confidentiality of the channel is protected by the unpredictability of the symmetric encryption algorithm.

This strawman protocol provides the confidentiality we expect from a channel, but its security is static. Even though peers are trusted for confidentiality, all it takes for an adversary to compromise the full past and future confidential transactions of the system is to compromise a single peer and recover k. Afterwards the adversary can collude with a Byzantine order to use the channels blockchain as a log of the past and decrypt every transactions, as well as keep receiving future transactions from the colluding orderer.

5.1 Deploying Group Key Agreement

To work around the attack, we first need to minimize the attack surface. To achieve this we need to think of the peers of a channel, as participants of a confidential communication channel and provide similar guarantees. Specifically, we guarantee the following properties.

1. **Forward Secrecy:** A passive adversary that knows one or more old encryption keys k_i, cannot discover any future encryption key k_j where $i < j$
2. **Backward Secrecy:** A passive adversary that knows one or more encryption keys k_i, cannot discover any previous encryption key k_j where $j < i$

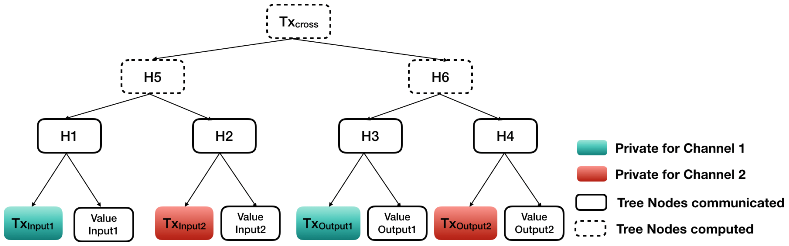


Fig. 2. Privacy preserving cross-channel transaction structure

3. **Group Key Secrecy:** It is computationally infeasible for an adversary to guess any group key k_i
4. **Key Agreement:** For an epoch i all group members agree on the epoch key k_i.

There are two types of group key agreement we look into:

Centralized Group-Key Distribution: In these systems, there is a dedicated server that sends the symmetric key to all the participants. The centralized nature of the key creation is scalable, but might not be acceptable even in a permissioned setting where different organizations participating in a channel are mutually suspicious.

Contributory Group-Key Management: In these systems, each group member contributes a share to the common group key, which is then computed by each member autonomously. These protocols are a natural fit to decentralized systems such as distributed ledgers, but they scale poorly.

We use the existence of the validation policy as an indication of the trusted entities of the channel (*i.e.,* the oracles) and create a more suitable protocol to the permissioned setting. Another approach could be to introduce a key-management policy that defines the key-generation and update rules but, for simplicity, we merge it with the validation policy that the peers trust anyway. We start with a scalable contributory group-key agreement protocol [9], namely the Tree-Based Group Diffie-Hellman system. However, instead of deploying it among the peers as contributors (which would require running view-synchronization protocols among them), we deploy it among the smaller set of oracles of the channel. The oracles generate symmetric keys in a decentralized way, and the peers simply contact their favorite oracle to receive the latest key. If an oracle replies to a peer with an invalid key, the peer can detect it because he can no longer decrypt the data, hence he can (a) provably blame the oracle and (b) request the key from another oracle.

More specifically we only deploy the group-join and group-leave protocols of [9], because we do not want to allow for splitting of the network, which might cause forks on the blockchain. We also deploy a group-key refresh protocol that is similar to group-leave, but no oracle is actually leaving.

5.2 Enabling Cross-Shard Transactions Among Confidential Channels

In the protocols we mentioned in Sect. 4, every party has full visibility on the inputs and outputs and is able to link the transfer of coins. However, this might not be desirable. In this section, we describe a way to preserve privacy during cross-channel transactions within each asset's channel.

For example, we can assume the existence of two banks, each with its own channel. It would be desirable to not expose intra-channel transactions or account information when two banks perform an interbank asset-transfer. More concretely, we assume that Alice and Bob want to perform a fair exchange. They have already exchanged the type of assets and the values they expect to receive. The protocol can be extended to store any kind of ZK-Proofs the underlying system supports, as long as the transaction can be publicly verified based on the proofs.

To provide the obfuscation functionality, we use Merkle trees. More specifically, we represent a cross-shard transaction as a Merkle tree (see Fig. 2), where the left branch has all the inputs lexicographically ordered and the right branch has all the outputs. Each input/output is represented as a tree node with two leaves: a private leaf with all the information available for the channel and a public leaf with the necessary information for third party verification of the transaction's validity.

The protocol for Alice works as follows:

Transaction Generation:

1. Input Merkle-Node Generation: Alice generates an input as before and a separate Merkle leaf that only has the type of the asset and the value. These two leaves are then hashed together to generate their input Merkle node.
2. Output Merkle-Node Generation: Similarly, Alice generates an Output Merkle node, that consists of the actual output (including the output address) on the private leaf and only the type and value of the asset expected to be credited on the public.
3. Transaction Generation: Alice and Bob exchange their public Input and Output Merkle-tree nodes and autonomously generate the full Merkle tree of the transaction.

Transaction Validation:

1. Signature Creation: Alice signs the MTR of the tx_{cross}, together with a bitmap of which leaves she has seen and accepts. Then she receives a similar signature from Bob and verifies it. Then Alice hashes both signatures and attaches them to the full transaction. This is the tx_{cross} that she submits in her channel for validation. Furthermore, she provides her full signature, which is logged in the channel's confidential chain but does not appear in the pool; in the pool the generated asset is $H(tx_{cross})$.
2. Validation: Each channel validates the signed transaction (from all inputs inside the channel's state) making sure that the transaction is semantically

correct (*e.g.,* does not create new assets). They also check that the publicly exposed leaf of every input is well generated (e.g. value and type much). Then they generate the new asset ($H(tx_{cross})$ as before) that is used to provide proof-of-commitment/abortion. The rest of the protocol (e.g. Unlock phase) is the same as Sect. 4.3.

Security and Privacy Arguments. The atomicity of the protocol is already detailed above. Privacy is achieved, because the source and destination addresses (accounts) are never exposed outside the shard and the signatures that authenticate the inputs inside the channel are only exposed within the channel. We also describe the security of the system outside the atomic commit protocol. More specifically,

1. Every tx_{cross} is publicly verifiable to make sure that the net-flow is zero, either by exposing the input and output values or by correctly generating ZK-proofs.
2. The correspondence of the public and private leaf of a transaction is fully validated by the input and/or output channel, making sure that its state remains correct.
3. The hash of the tx_{cross} is added in the pool to represent the asset. Given the collision resistance of a hash function, this signals to all other channels that the private leaves correspond to the transaction have been seen, validated and accepted.

The scheme can be further enhanced to hide the values using Pedersen commitments [19] and range-proofs similar to confidential transactions [14]. In such an implementation the Pedersen commitments should also be opened on the private leaf for the consistency checks to be correctly done.

6 Case Study: Cross-Shard Transactions on Hyperledger Fabric

In order to implement the cross-channel support on Fabric v1.1, we start with the current implementation of FabCoin [1] that implements an asset-management protocol similar to the one introduced in Sect. 3.2.

Channel-Based Implementation. As described by Androulaki et al. [1], a Fabric network can support multiple blockchains connected to the same ordering service. Each such blockchain is called a *channel.* Each channel has its own configuration that includes all the functioning metadata, such as defining the membership service providers that authenticate the peers, how to reach the ordering service, and the rules to update the configuration itself. The genesis block of a channel contains the initial configuration. The configuration can be updated by submitting a *reconfiguration transaction.* If this transaction is valid with the respect to the rules described by the current configuration, then it gets committed in a block containing only the reconfiguration transaction, and the chances are applied.

In this work, we extend the channel configuration to include the metadata to support cross-channel transactions. Specifically, the configuration lists the channels with which interaction is allowed; we call them *friend channels*. Each entry also has a *state-update validation policy*, to validate the channel's state-updates, the identities of the oracles of that channel, that will advertise state-update transactions, and the current commitment to the state of that channel. The configuration block is also used as a *lock-step* that signals the view-synchrony needed for the oracles to produce the symmetric-key of the channel. If an oracle misbehaves, then a new configuration block will be issued to ban it.

Finally, we introduce a new entity called *timestamper* (inspired by recent work in software updates [18]) to defend against freeze attacks where the adversary presents a stale configuration block that has an obsolete validation policy, making the network accepting an incorrect state update. The last valid configuration is signed by the timestampers every'*interval*, defined in the configuration, and (assuming loosely synchronised clocks) guarantees the freshness of state updates[6].

Extending FabCoin to Scale-out. In FabCoin [1] each asset is represented by its current output state that is a tuple of the form $(txid.j, (value, owner, type))$. This representation denotes the asset created as the j-th output of a transaction with identifier *txid* that has *value* units of asset *type*. The output is owned by the public key denoted as *owner*.

To support cross-channel transactions, we extend FabCoin transactions by adding one more field, called *namespace*, that defines the channel that manages the asset (*i.e.,* $(txid.j, (namespace, value, owner, type))$.

Periodically, every channel generates a *state commitment* to its state, this can be done by one or more channel's oracles. This state commitment consists of two components: (i) the root of the Merkle tree built on top of the UTXO pool, (ii) the hash of the current configuration block with the latest timestamp, which is necessary to avoid freeze attacks.

Table 1. Atomic commit protocol on fabric channels

Protocol	Atomicity	Trust assumption	Generality of transactions	Privacy
Asset transfer (Sect. 4.1)	Yes	Nothing extra	1-input-1-ouptut	No
Trusted channel (Sect. 4.2)	Yes	Trusted intermediary channel	N-input-M-output	No
Atomic commit (Sect. 4.3)	Yes	Nothing extra	N-input-M-output	No
Obfuscated transaction AC (Sect. 5.2)	Yes	Nothing extra	N-input-M-output	Yes

[6] unless both the timestamp role and the validation policy are compromised.

Then, the oracles of that channel announce the new state commitment to the friend channels by submitting specific transactions targeting each of these friend channels. The transaction is committed if (i) the hashed configuration block is equal to the last seen configuration block, (ii) the timestamp is not "too" stale (for some time value that is defined per channel) and (iii) the transaction verifies against the state-updates validation policy. If those conditions hold, then the channel's configuration is updated with the new state commitment. If the first condition does not hold, then the channel is stale regarding the external channel it transacts with and needs to update its view.

Using the above state update mechanism, Alice and Bob can now produce verifiable proofs that certain outputs belong to the UTXO pool of a certain channel; these proofs are communicated to the interested parties differently, depending on the protocol. On the simple asset-transfer case (Sect. 4.1), we assume that Alice is altruistic (as she donates an asset to Bob) and request the proofs from her channel that is then communicated off-band to Bob. On the asset trade with trusted channels (Sect. 4.2) Alice and Bob can independently produce the proofs from their channels or the trusted channel as they have visibility and access rights. Finally on the asset trade of Sect. 4.3, Alice and Bob use the signed cross-channel transaction as proof-of-access right to the channels of the input assets in order to obtain the proofs. This is permitted because the tx_{cross} is signed by some party that has access rights to the channel and the channels peers can directly retrieve the proofs as the asset's ID is derived from $H(tx_{cross})$.

7 Conclusion

In this paper, we have redefined channels, provided an implementation guideline on Fabric [1] and formalized an asset management system. A channel is the same as a shard that has been already defined in previous work [7,12]. Our first contribution is to explore the design space of sharding on permissioned blockchains where different trust assumptions can be made. We have introduced three different protocols that achieve different properties as described in Table 1. Afterwards we have introduced the idea that a channel in a permissioned distributed ledger can be used as a confidentiality boundary and describe how to achieve this. Finally, we have merged the contributions to achieve a confidentiality preserving, scale-out asset management system, by introducing obfuscated transaction trees.

Acknowledgments. We thank Marko Vukolić and Björn Tackmann for their valuable suggestions and discussions on earlier versions of this work. This work has been supported in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780477 PRIViLEDGE.

References

1. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth European conference on Computer systems, EuroSys 2018. ACM, New York (2018). https://arxiv.org/abs/1801.10228
2. Bishop, G.: Illinois begins pilot project to put birth certificates on digital ledger technology, September 2017. https://www.ilnews.org/news/statewide/illinois-begins-pilot-project-to-put-birth-certificates-on-digital/article_1005eca0-98c7-11e7-b466-170ecac25737.html
3. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: SoK: research perspectives and challenges for bitcoin and cryptocurrencies. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 104–121. IEEE (2015). http://ieeexplore.ieee.org/abstract/document/7163021/
4. Browne, R.: IBM partners with nestle, unilever and other food giants to trace food contamination with blockchain, September 2017. https://www.cnbc.com/2017/08/22/ibm-nestle-unilever-walmart-blockchain-food-contamination.html
5. Cachin, C., Vukolic, M.: Blockchain consensus protocols in the wild. CoRR, abs/1707.01873 (2017). https://arxiv.org/abs/1707.01873
6. Croman, K., et al.: On scaling decentralized blockchains (a position paper). In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 106–125. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_8. http://fc16.ifca.ai/bitcoin/papers/CDE+16.pdf
7. Danezis, G., Meiklejohn, S.: Centrally banked cryptocurrencies. In: 23rd Annual Network and Distributed System Security Symposium (NDSS), February 2016. https://eprint.iacr.org/2015/502.pdf
8. Herlihy, M.: Atomic cross-chain swaps. arXiv preprint arXiv:1801.09515 (2018)
9. Kim, Y., Perrig, A., Tsudik, G.: Tree-based group key agreement. ACM Trans. Inf. Syst. Secur. (TISSEC) **7**(1), 60–96 (2004)
10. Kokoris-Kogias, E., et al.: Hidden in plain sight: storing and managing secrets on a public ledger. Cryptology ePrint Archive, Report 2018/209 (2018). https://eprint.iacr.org/2018/209
11. Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing bitcoin security and performance with strong consistency via collective signing. In: Proceedings of the 25th USENIX Conference on Security Symposium (2016). http://arxiv.org/abs/1602.06997
12. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: OmniLedger: a secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 19–34. IEEE (2018)
13. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. Technical report, Cryptology ePrint Archive, Report 2015/675 (2015). http://eprint.iacr.org
14. Maxwell, G.: Confidential transactions (2015). http://people.xiph.org/~greg/confidentialvalues.txt
15. Melendez, S.: Fast, Secure Blockchain Tech from an Unexpected Source Microsoft, September 2017. https://www.fastcompany.com/40461634/
16. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: anonymous distributed e-cash from bitcoin. In: 34th IEEE Symposium on Security and Privacy (S&P), May 2013

17. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). https://bitcoin.org/bitcoin.pdf
18. Nikitin, K., et al.: CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 1271–1287 (2017)
19. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 129–140. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_9
20. Roy, R.: Shard âĂŞ a database design, July 2008. http://technoroy.blogspot.ch/2008/07/shard-database-design.html
21. Sasson, E., et al.: Zerocash: decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 459–474. IEEE (2014)
22. Simonsen, S.: 5 Reasons the UN is jumping on the blockchain bandwagon, September 2017. https://singularityhub.com/2017/09/03/the-united-nations-and-the-ethereum-blockchain/
23. Swanson, T.: Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. Report, April 2015
24. Wikipedia: Atomic commit, February 2018. https://en.wikipedia.org/wiki/Atomic_commit