Transactional Cross-Chain Smart Contract Invocations

Ghareeb Falazi[1], Uwe Breitenbücher[2], Frank Leymann[1],
Stefan Schulte[3], and Vladimir Yussupov[1]

[1] Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{falazi, leymann, yussupov}@iaas.uni-stuttgart.de

[2] Reutlingen University, Germany
uwe.breitenbuecher@reutlingen-university.de

[3] Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things,
Institute for Data Engineering, Hamburg University of Technology, Germany
stefan.schulte@tuhh.de

Transactional Cross-Chain Smart Contract Invocations

GHAREEB FALAZI, Institute of Architecture of Application Systems, University of Stuttgart, Germany
UWE BREITENBÜCHER, Reutlingen University, Germany
FRANK LEYMANN, Institute of Architecture of Application Systems, University of Stuttgart, Germany
STEFAN SCHULTE, Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things, Hamburg University of Technology, Germany
VLADIMIR YUSSUPOV, Institute of Architecture of Application Systems, University of Stuttgart, Germany

Blockchains have become increasingly important in recent years and have expanded their applicability to many domains beyond finance and cryptocurrencies. This adoption has particularly increased with the introduction of smart contracts, which are immutable, user-defined programs directly deployed on blockchain networks. However, many scenarios require business transactions to simultaneously access smart contracts on multiple, possibly heterogeneous blockchain networks while ensuring the atomicity and isolation of these transactions, which is not natively supported by current blockchain systems. Therefore, in this work, we introduce the Transactional Cross-Chain Smart Contract Invocation (TCCSCI) approach that supports such distributed business transactions while ensuring their global atomicity and serializability. The approach introduces the concept of Resource Manager Smart Contracts, and 2PC for Blockchains (2PC4BC), a client-driven Atomic Commit Protocol (ACP) specialized for blockchain-based distributed transactions. We validate our approach using a prototypical implementation, evaluate its introduced overhead, and prove its correctness.

CCS Concepts: • **Applied computing** → *Enterprise application integration*; • **Computer systems organization** → **Distributed architectures**; • **Information systems** → Distributed transaction monitors; *Distributed database transactions.*

Additional Key Words and Phrases: blockchain, smart contract, interoperability, cross-chain, cross-ledger, multi chain

ACM Reference Format:
Ghareeb Falazi, Uwe Breitenbücher, Frank Leymann, Stefan Schulte, and Vladimir Yussupov. 2023. Transactional Cross-Chain Smart Contract Invocations. *Distrib. Ledger Technol.* 1, 1 (August 2023), 26 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Blockchain systems have become increasingly important in recent years and have expanded their field of applicability to domains beyond finance and cryptocurrencies, such as health care management [30], supply chain and logistics management [19], the energy sector [32], and others. This trend is influenced by their capability of managing a tamper-resistant and tamper-evident ledger of transactions without the need for a Trusted Third-Party (TTP) [36]. Furthermore, the adoption of smart contracts [34], which are immutable and deterministic user-defined programs

Authors' addresses: Ghareeb Falazi, ghareeb.falazi@iaas.uni-stuttgart.de, Institute of Architecture of Application Systems, University of Stuttgart, Germany; Uwe Breitenbücher, uwe.breitenbuecher@reutlingen-university.de, Reutlingen University, Germany; Frank Leymann, frank.leymann@iaas.uni-stuttgart.de, Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany; Stefan Schulte, stefan.schulte@tuhh.de, Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things, Hamburg University of Technology, Germany; Vladimir Yussupov, vladimir.yussupov@iaas.uni-stuttgart.de, Institute of Architecture of Application Systems, University of Stuttgart, Germany.

deployable on certain blockchain systems, has enabled implementing the logic of sensitive business collaborations that run among mutually-distrustful business partners directly on blockchain networks [8]. However, it is often necessary that a *blockchain-based business process* accesses resources that are distributed across multiple blockchain networks in the same business transaction [6, 8, 13] since there is no "one size fits all" blockchain system and different types of blockchains will continue to coexist [29]. Therefore, enterprises participating in complex use cases, such as supply chains or travel industry scenarios, will likely need to connect to multiple blockchain networks simultaneously and run their *distributed business transactions* across different chains. Moreover, since the business logic is implemented by smart contracts, in enterprise integration scenarios, such distributed business transactions will likely incorporate invocations of multiple smart contracts hosted on different, possibly heterogeneous blockchain networks simultaneously. However, current blockchain systems do not support distributed cross-chain transactions as described above. Thus, it is currently not possible to implement client-side blockchain-based distributed business transactions in a way that ensures their global atomicity and global serializability, which violates their correctness according to the ACID paradigm [3, 4].

Therefore, in this work, we present the *Transactional Cross-Chain Smart Contract Invocation (TCCSCI)* approach, which enables executing distributed business transactions that involve smart contract function invocations distributed over possibly a heterogeneous set of blockchain networks to be executed in an atomic manner and in strict isolation from each other so that their correctness is ensured. We refer to these transactions as TCCSCIs. In our approach, we present the idea of *Resource Manager Smart Contracts (RMSC)*, which is based on the concept of resource managers known from the distributed transactions domain, and the *Two-Phase Commit for Blockchains (2PC4BC)* protocol, which is based on the classical Two-Phase Commit (2PC) protocol [10]. The combination of these two concepts enables to jointly guarantee global atomicity and serializability for TCCSCIs. To summarize, our contributions are as follows:

(1) We present a formal description of the problem of ensuring transactional behavior for cross-chain smart contract invocations based on a formal blockchain system model.
(2) We introduce the TCCSCI approach that solves the stated problem by: (i) proposing the concept of an RMSC that can be used to separate the execution of smart contract functions from the commitment of the changes that occur during the execution, and (ii) introducing the 2PC4BC protocol, which is a client-driven Atomic Commit Protocol (ACP) that utilizes RMSCs to ensure global atomicity and serializability for distributed transactions that involve function invocations of smart contracts hosted on different blockchain networks.
(3) We prove the approach correctness, evaluate its time complexity, and estimate the worse-case execution cost.
(4) We validate the practical feasibility of the TCCSCI approach with a prototypical implementation thereof.

The rest of this paper is structured as follows: In Section 2, we present basic background information regarding blockchain technology and blockchain interoperability. In Section 3, we motivate the importance of TCCSCIs and present the research question this work aims to solve. In Section 4, we introduce a formal blockchain system model that we use as basis for our approach, and formalize the problem statement. In Section 5, we introduce the TCCSCI approach, and in Section 6, we validate it by introducing a prototypical implementation for it. In Section 7, we evaluate the approach and prove its correctness, and in Section 8 we discuss its properties. In Section 9, we discuss the related work, and finally, in Section 10 we provide a conclusion and discuss potential future research directions.

2 BACKGROUND

In this section, we give necessary background information regarding the blockchain technology, the problem of blockchain interoperability, and the concept of resource management in blockchain systems.

2.1 The Blockchain Technology

A *blockchain system* is a distributed system that supports the interaction between parties that are mutually distrustful [36]. By running a *consensus protocol* among the blockchain participants, they can ensure the consistency of a shared state without favoring any one of them over the others and without requiring the honesty of specific participants [5]. To facilitate these properties, a blockchain system maintains the history of all executed *blockchain transactions*, i.e., participant-issued requests to execute operations that change the shared state. Blockchain transactions are usually grouped into *blocks* that are cryptographically linked together to form a data structure known as a *blockchain.* With the help of the consensus protocol, which is used to formulate agreed-upon blocks, the *immutability* and *non-repudiation* of the history of executed blockchain transactions are guaranteed without the need for a *Trusted Third Party (TTP)* [36].

Blockchain systems can be grouped into two major categories: First, there are *permissionless blockchain systems*, such as Bitcoin [21] and Ethereum [34], that allow anyone to participate, and are considered to be more decentralized [36]. However, permissionless blockchain systems usually employ consensus protocols that are relatively slow and temporarily allow multiple transaction histories to exist. Therefore, they are more suitable for peer-to-peer applications that require decentralization. Second, we have *permissioned blockchain systems*, such as Hyperledger Fabric [1]. These systems are characterized by requiring the permission of an authority before an entity can join the system under certain roles [36], which facilitates better control over data privacy. Being permissioned allows these blockchain systems to use Byzantine Fault Tolerant (BFT)-based consensus protocols, which generally have better performance and do not suffer from forking making them better suited to enterprise applications despite relinquishing a certain degree of decentralization.

With the advent of blockchain *smart contracts*, blockchain systems became suited for many more use cases, such as health care management [30], supply chain management [19], and others. A smart contract is an immutable user-defined program directly deployed to the blockchain and manages a portion of the blockchain's shared state [34]. *Smart contract functions* are executed as *units-of-work*, i.e., only the effects of successful executions are persisted, and the partial effects of failed executions are undone. Moreover, the blockchain protocol ensures that they are executed exactly as designed. Hence, they are suitable for implementing the business logic for collaborations among mutually-distrustful partners [8].

2.2 Blockchain Interoperability

While permissionless blockchains are often used for peer-to-peer applications that require decentralization, such as token management, permissioned blockchains are better suited to enterprise applications [7]. Therefore, there is no "one size fits all" blockchain system. In fact, different types of blockchain systems will likely continue to evolve and coexist, since they choose different trade-offs that contradict each other but are suitable for different needs [29]. Therefore, it is often necessary that a given business process accesses resources over multiple *blockchain networks*[1] simultaneously [6, 8, 13]. We refer to the ability of a single business transaction to access resources over different blockchain networks as *blockchain interoperability*. Many blockchain interoperability approaches exist [2], but they all try to solve the same core problem common to blockchain systems, which is that these systems cannot directly access external resources. The reason is that the execution of blockchain transactions must be deterministic, i.e., that all honest peers participating in the execution of these transactions must come up with the same results since, otherwise, consensus cannot be guaranteed [31]. However, accessing external systems is not guaranteed to produce consistent results for all blockchain nodes and over an extended period of time. Therefore, such access is generally prohibited within the frame of blockchain transactions. It is even prohibited that a blockchain transaction of one network accesses

[1]A blockchain network is an instance of a blockchain system.

resources on a different blockchain network. Hence, new data cannot be actively queried by blockchain nodes from external systems, but rather have to be passed to the consensus mechanism as entries within the request messages issued by client applications. Therefore, when a *cross-blockchain application* requires information to be passed from one blockchain network to another, an *off-chain* entity (known as an *oracle*) needs to embed this data into a blockchain transaction execution request and submit it to the consensus mechanism of the target blockchain. Usually a service that acts as a client application to both blockchain networks performs this task [20]. Cross-blockchain applications have various aims. Therefore, many corresponding *blockchain interoperability approaches* have been developed with different kinds of goals, such as Cross-Chain Asset Transfer (CCAT) [14, 15], Cross-Chain Data Migration (CCDM) [9], and Cross-Chain Smart Contract Invocation (CCSCI) [18, 26], which is the main focus of this work.

2.3 Transactional Resource Managers and Blockchain Resource Management Layer

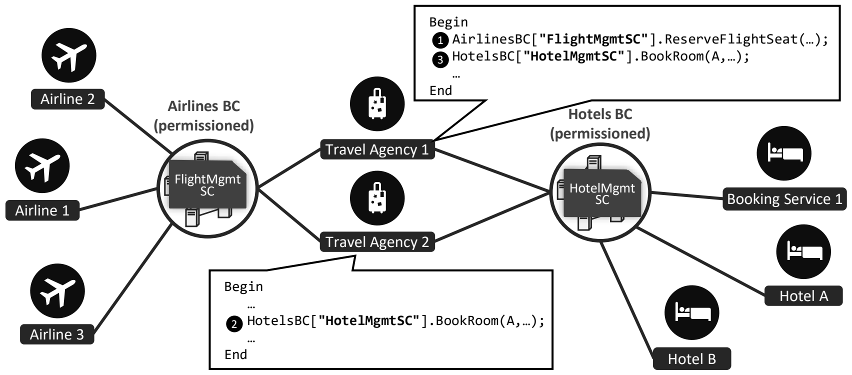
A *transactional resource manager* (or a *resource manager* for short) is a component that manages shared resources in *transaction processing (TP) systems* [4]. Traditionally, these resources could be databases, queues, files, and other shared objects accessible within a transaction. The major responsibility of a resource manager is ensuring the *Atomicity, Consistency, Isolation, and Durability (ACID)* properties for the transactions that access the underlying resource.

A blockchain system also maintains resources. For example, in Ethereum [34], these resources are the *account states*, whereas in Hyperledger Fabric [1], they are the *ledger states*. Hence, although blockchain systems commonly lack a dedicated resource manager component, every blockchain system has mechanisms that ensure that transactions accessing managed resources maintain the ACID properties [7]. For example, in Ethereum the *Ethereum Virtual Machine (EVM)* ensures that aborting a transaction while executing a smart contract function revokes all changes made to the state of the account that hosts the smart contract, whereas the consensus mechanism ensures that transactions are executed serially in a predetermined and agreed upon order. Moreover, in Hyperledger Fabric, the transaction flow utilizes an optimistic locking mechanism that ensures serializability, and a regular database system ensures durability and atomicity. Hence, we refer to the ACID-enforcing mechanisms of a blockchain system as its *Resource Management Layer*. In smart contract-enabled blockchains, this layer sits between smart contracts and the underlying blockchain resources, e.g., ledger states, and exposes the API of a regular resource manager to the smart contract functions.

3 MOTIVATION AND RESEARCH QUESTION

CCSCIs, which are the focus of this work, allow the invocation of smart contract functions of two or more blockchain networks to be composed or nested together. Moreover, ensuring the *atomicity*, i.e., all-or-nothing behavior, of CCSCI executions is crucial in many enterprise application scenarios. For example, consider the exemplary use case depicted in Figure 1. It shows two permissioned blockchain networks that act as marketplaces for airline and hotel room reservations[2]. In the first network, different airlines offer flight seats for booking, while in the second network, different hotels and property rental services offer rooms for booking. The business logic for each of these blockchain services, or *Decentralized Applications (DApps)*, is encoded in the form of smart contracts that are deployed directly on the corresponding blockchain networks. Obviously, travel agencies are interested in both of these DApps, so they are connected to the two networks. Furthermore, in order to interact with these DApps, they invoke the smart contracts functions that encode the desired functionality, e.g., booking a hotel room or a flight seat, by submitting blockchain

[2]Note that we are not restricted to permissioned blockchains. For instance, conditional payments may be performed using a smart contract in a permissionless blockchain network.



```
Begin
❶ AirlinesBC["FlightMgmtSC"].ReserveFlightSeat(…);
❸ HotelsBC["HotelMgmtSC"].BookRoom(A,…);
…
End
```

```
Begin
  …
❷ HotelsBC["HotelMgmtSC"].BookRoom(A,…);
  …
End
```

Fig. 1. A scenario that demonstrates the need for atomicity in the execution of CCSCIs. The business transaction executed by Travel Agency 1 is not executed atomically because of a conflicting concurrent execution of another transaction executed by Travel Agency 2.

transactions to the underlying blockchain networks. If these transactions are valid, the consensus mechanisms will accept them, which results in the execution of the requested smart contract functions.

Consequently, travel agencies can provide more sophisticated services, e.g., planning a round-trip flight combined with a hotel stay. Let us assume *Travel Agency 1*, which represents the enterprise application of some travel agency, is currently executing such a *business transaction* as shown in Figure 1: it starts by ❶ invoking the flight seat reservation function of the flight management smart contract hosted on the Airlines Blockchain (we denote this as `AirlinesBC["FlightMgmtSC"].ReserveFlightSeats(...);`). Assuming this invocation succeeds, the business transaction moves to executing a second function in which a room in *Hotel A* is reserved. However, before this happens, another travel agency, *Travel Agency 2* ❷ tries to book a room in *Hotel A* by executing the same function as part of a different business transaction. Unfortunately, this results in booking the last room available in *Hotel A*. Therefore, when the business transaction of *Travel Agency 1* tries to book a room in this hotel (step ❸), the operation fails. In this case, we ended up in a business transaction that is *non-atomic*, i.e., it has its intended effects persisted in only one of the underlying blockchain networks but not the other (only the flight was booked but not the hotel room). Clearly, this is an undesirable state. Therefore, it is very crucial to ensure an all-or-nothing behavior in such business transactions, i.e., to ensure their *global atomicity*, which requires that a distributed transaction either entirely commits or entirely aborts at *all* nodes [17]. Another problem we notice in this example is that parallel business transactions were not isolated, i.e., they were allowed to manipulate shared data at the same time, which can result in unintended results. Hence, it is also important to ensure *global serializability*, which requires that the execution of a set of interleaved distributed transactions running concurrently is equivalent to some serial execution of these transactions [17].

Many more use cases, such as trade finance [26], and supply-chain management [8], involve business transactions that include invocations of smart contracts functions hosted on different blockchain networks, which we referred to as CCSCIs. However, as we will show in Section 9, no existing approach offers a practical solution for executing these business transactions while ensuring their global atomicity and global serializability, which is often a requirement for their correctness. Therefore, the research question for this work is:

Research Question: *How can global atomicity and global serializability be guaranteed in business transactions that involve the invocation of smart contract functions located in different, possibly heterogeneous blockchain networks?*

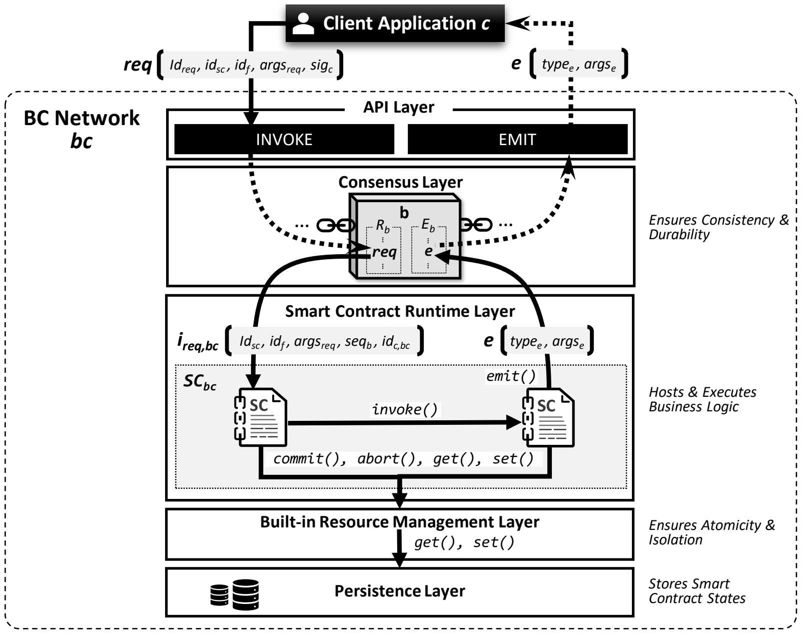

Client Application c
req {Id_req, id_sc, id_f, args_req, sig_c}
e {type_e, args_e}
BC Network bc
API Layer
INVOKE
EMIT
Consensus Layer
b
R_b
E_b
req
e
Ensures Consistency & Durability
Smart Contract Runtime Layer
i_req,bc {Id_sc, id_f, args_req, seq_b, id_c,bc}
e {type_e, args_e}
SC_bc
emit()
SC
invoke()
commit(), abort(), get(), set()
Hosts & Executes Business Logic
Built-in Resource Management Layer
get(), set()
Ensures Atomicity & Isolation
Persistence Layer
Stores Smart Contract States

Fig. 2. An architectural representation of the blockchain system model we use in this work.

4 FORMAL FOUNDATIONS

In this section, we introduce a formal system model for blockchain systems that we use as basis for our approach. This model only contains concepts that are common in various blockchain systems and relevant for our approach. The idea is to present our approach in a way that is independent of any concrete blockchain system. Moreover, we formalize our research question based on this system model and describe the assumptions on which our approach is based.

4.1 System Model

This section presents the formal system model, which we use as basis for introducing our approach in Section 5. An architectural representation of the system model is shown in Figure 2. The model describes the set of entities involved in any individual blockchain network *bc* that takes part in our approach. In addition, it describes the relationships between these entities and the requirements they must fulfill. The model's aim is not to capture as many details related to blockchain systems as possible, but capture only those details that are directly relevant for our approach.

- A *client application* c is a blockchain-external, or *off-chain*, program that communicates with the blockchain smart contracts of some blockchain network bc. Every client application c is associated with an identity $id_{c,bc}$ that uniquely identifies it within bc. It also has an asymmetric cryptography key pair (pu_c, pr_c) used for signing request messages and submitting them to the API Layer of bc. There is a unique mapping between pu_c and $id_{c,bc}$.
- The blockchain network bc hosts a set of *smart contracts* SC_{bc}. Each smart contract $sc \in SC_{bc}$ is defined as follows: $sc := (id_{sc}, F_{sc})$. Here, id_{sc} uniquely identifies the smart contract sc within bc and F_{sc} is the set of *smart contract functions* contained in sc. A smart contract function $f \in F_{sc}$ is a deterministic program that has an identifier id_f that is unique within sc. We assume the following set of basic smart contract operations must be supported by the

smart contract runtime layer of the blockchain network bc: (i) `commit` is used to signal the successful end of a given invocation of a smart contract function f, (ii) `abort` is used to signal an error in a given invocation of f, (iii) `set` is used to change the value of a *data item* stored in the *persistence layer*, which stores the states of all smart contracts hosted on the blockchain network bc, (iv) `get` is used to retrieve the value of a data item stored in the persistence layer, (v) `invoke` is used to invoke another smart contract function within the same blockchain network bc, and (vi) `emit` is used to send a *smart contract event* to client applications.

- A client application c invokes a function $f \in F_{sc}$ of a smart contract sc by submitting a *smart contract function invocation request* (aka a "*blockchain transaction*") $req := (id_{req}, id_{sc}, id_f, args_{req}, sig_c) \in REQ$ to the `INVOKE` method of the *API layer* of the blockchain network bc. Here, REQ is the set of all possible smart contract function invocation requests of all blockchain networks. Moreover, in every request $req \in REQ$, three identifiers unique within the blockchain network bc to which the request is sent are provided: (i) id_{req} represents the unique identifier of the request, (ii) id_{sc} represents the identifier of the smart contract that contains the function to be invoked f, and (iii) id_f represents the identifier of f. Furthermore, $args_{req}$, is a sequence of arguments that will be passed to f. Finally, req contains a client-generated cryptographic signature sig_c that allows verifying its integrity and associates it with the client application c. We assume that all submitted requests $req \in REQ$ will eventually be delivered to the API layer (which is usually HTTP-based) after an arbitrary but finite delay.
- The *consensus layer* of the blockchain network verifies the validity of each request $req \in REQ$ and includes verified requests into the next block $b := (seq_b, R_b, \prec_{R_b}, E_b, \prec_{E_b})$ being constructed. Here, seq_b represents the sequence number of b, while $R_b \subset REQ$ is the set of requests included within b (such that $req \in R_b$). R_b is strictly and totally ordered using the relation $\prec_{R_b}$: $R_b \times R_b$, which reflects the order in which the requests in b will be executed. E_b represents the set of smart contract events that are emitted during the execution of the requests $req_1, ...req_{|R_b|} \in R_b$. E_b is strictly and totally ordered using the relation $\prec_{E_b}$: $E_b \times E_b$ which represents the order in which the events are emitted. E_b starts as an empty set, then it gets gradually filled with the events that are emitted during the execution of the smart contract functions invoked by the requests $req_1, ...req_{|R_b|} \in R_b$ as we will discuss next.
  We assume that in order for a request $req \in REQ$ to be considered valid, it must be unique and that any request successfully validated by the consensus layer will eventually be included into a block. Furthermore, we assume that the consensus layer only associates monotonically increasing sequence numbers to the generated blocks. Therefore, seq_b uniquely identifies the block b within the blockchain network that received the request.
  After the consensus layer adds "enough"[3] validated requests $req \in REQ$ to R_b, it triggers the invocation of all requests in it sequentially according to $\prec_{R_b}$. For each request $req = (id_{req}, id_{sc}, id_f, args_{req}, sig_c) \in R_b$, the function identified by id_f inside the smart contract identified by id_{sc} is invoked using the arguments $args_{req}$ passed by the client application in combination with *context* information, i.e., (i) the *sequence number* seq_b of the block containing the request, which enables f to identify the point in time of which the invocation takes place, and (ii) $id_{c,bc}$, the identity of the client application that submitted the request[4]. Therefore, when a request $req \in R_b$ that is included into the block b is triggered, the resulting *smart contract function invocation* is defined as $i_{req,b} := (id_{sc}, id_f, args_{req}, seq_b, id_{c,bc}) \in I_{bc}$, where I_{bc} represents the set of all invocations that result from valid smart contract function invocation requests included into the blockchain structure of the blockchain network bc.
- The *smart contract runtime layer* is responsible for performing all smart contract function invocations in I_{bc}. During an invocation $i_{req,b} \in I_{bc}$, the basic smart contract operations `set`, `get`, `commit`, and `abort` are forwarded to the *built-in*

[3]This is determined on a technology-specific basis. For example, Ethereum miners try to fill the block up to a known threshold, i.e., the block's *gasLimit* [34].
[4]The consensus layer infers $id_{c,bc}$ from sig_c.

resource management layer that ensures the atomicity and isolation of invocations within the blockchain network bc, and communicates with the underlying persistence layer by reading from it and writing to it. Moreover, the basic smart contract operation `emit` stores a smart contract event $e := (type_e, args_e)$ in the set E_b within the block b. Here, $type_e$ represents a custom-defined event type, and $args_e$ represents a set of data items that will be reported back to the client application c. Later, when the block b is added to the blockchain structure of the blockchain network bc, the client application receives an asynchronous message from the `EMIT` method of the API Layer of bc containing the smart contract event e. Since e is permanently stored into the blockchain data structure, even if the client application c is unavailable when the block b is added to the blockchain, it still can receive the smart contract event e when it becomes available again, and it receives it in the correct order. Furthermore, the basic smart contract operation `invoke` executes a different smart contract function as a *sub-invocation* of the current invocation. This means the following: (i) `set` and `get` operations performed by the sub-invocation are considered to be part of the parent invocation when it comes to atomicity and isolation, (ii) a `commit` operation performed by the sub-invocation is allowed to return values to the parent invocation, but does not guarantee that the state changes incurred will be persisted, and (iii) an `abort` operation performed by the sub-invocation reverts all state changes that took place during its execution and reports the failure to the parent invocation, which handles it according to its business logic, i.e., it does not necessarily abort, too. Finally, we assume all invocations end either with the execution of a `commit` operation or the execution of an `abort` operation. In addition to the logic of f, the built-in resource management layer may also trigger the `abort` operation if it determines that the invocation $i_{req,b}$ cannot finish in a serializable manner. Similarly, the smart contract runtime layer may also abort if it detects that $i_{req,b}$ is infinite or too costly. The execution of this operation is handled by the built-in resource management layer, which also uses the `emit` operation to asynchronously report aborted invocations to the client application. Regardless of whether an invocation $i_{req,b}$ ends with a `commit` or an `abort`, *it is always guaranteed that it is atomic and serializable within the blockchain network.* In fact, we assume that smart contract function invocations fulfill the ACID properties [4].

- When the smart contract runtime layer finishes all the invocations triggered by the requests $req_1, ...req_{|R_b|} \in R_b$, it permanently adds the block b to the blockchain structure of the blockchain network bc.

4.2 The Transactional CCSCI Problem

Having formally defined the system model that describes the functionality of a blockchain network, we now use it to formally define the problem that we want to solve, which is based on the *Two-Phase Commit (2PC) Problem Definition* [3, 11, 12]: A client application c executes a transactional program that invokes smart contract functions hosted on two or more blockchain networks $bc_1, bc_2, ..., bc_n$ $(n > 1)$, which we assume never crash. We call this execution a *Transactional CCSCI (TCCSCI).* In fact a TCCSCI is a distributed transaction [4] that invokes smart contracts distributed across multiple blockchains. Therefore, we use dtx to denote it, and we formally define it as a totally and strictly ordered set of operations that the runtime environment of the client application c executes, i.e., $dtx := (\Omega_{dtx}, \prec_{dtx})$, in which $\Omega_{dtx} := \{\omega_1, ..., \omega_m\}(m > 3)$, and the relation $\prec_{dtx}$: $\Omega_{dtx} \times \Omega_{dtx}$ defines the strict and total order of the operations. Here, ω_1, the first operation executed in dtx according to $\prec_{dtx}$, is always `start_dtx`, which marks the beginning of dtx, and ω_m, the last operation executed in dtx according to $\prec_{dtx}$, is either `commit_dtx` or `abort_dtx`, which mark the successful or failed end of the distributed transaction dtx, respectively[5]. The concrete implementation of these three operations according to our approach will be described in Section 5. Each of the operations $\omega_2, ..., \omega_{m-1} \in \Omega_{dtx}$ is a

[5]Note that `commit_dtx` and `abort_dtx` are different from `commit` and `abort` introduced in Section 4.1. The former pair are operations executed by the runtime environment of the client application, and the latter pair are operations executed by the smart contract runtime layer of a blockchain network.

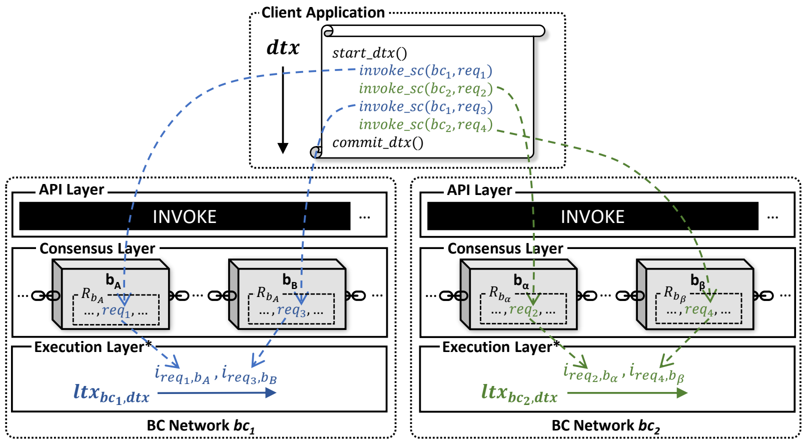


Fig. 3. An example that shows how a distributed transaction dtx that contains smart contract function invocations on bc_1 and bc_2 results in the local transactions $ltx_{bc_1,dtx}$ in bc_1 and $ltx_{bc_2,dtx}$ in bc_2. For brevity, *Execution Layer** refers to all the layers below the Consensus Layer, i.e., Smart Contract Runtime Layer, Built-in Resource Management Layer, and Persistence Layer (see Figure 2).

request submission operation defined as $invoke\_sc := (id_{bc},\ req)$, which represents submitting the request $req = (id_{req},\ id_{sc},\ id_f,\ args_{req},\ sig_c) \in REQ$ to the `INVOKE` method of the API Layer of the blockchain network identified by id_{bc} in order to invoke the function identified by id_f of the smart contract identified by id_{sc}.

Furthermore, we define a *local transaction* $ltx_{bc,dtx} := (I_{bc,dtx}, \prec_{I_{bc}})$ to be the strictly and totally ordered set of smart contract function invocations that result from the execution of a TCCSCI dtx in the blockchain network bc:

$$I_{bc,dtx} := \{i_{req,b} \in I_{bc} | b \in Blocks_{bc} \wedge \exists o \in \Omega_{dtx} : o = invoke\_sc(id_{bc}, req)\} \qquad (1)$$

Formula 1 indicates that the set $I_{bc,dtx}$ contains all invocations in the blockchain network bc that result from the `invoke_sc` operations executed within dtx. Here, $Blocks_{bc}$ denotes the set of blocks of the blockchain network bc.

$$i_{req_A,b_1} \prec_{I_{bc}} i_{req_B,b_2} \iff (seq_{b_1} < seq_{b_2} \vee (b_1 = b_2 \wedge req_A \prec_{R_{b_1}} req_B)) \qquad (2)$$

Formula 2 indicates that the relation $\prec_{I_{bc}}: I_{bc} \times I_{bc}$ orders any two invocations $i_{req_A,b_1}, i_{req_B,b_2} \in I_{bc}$ according to the sequence numbers of the blocks b_1 and b_2 that include the smart contract invocation requests that triggered them. If these requests are in the same block b, i.e., if $b_1 = b_2 = b$, the invocations are ordered using the relative order of the corresponding requests within R_b (see Section 4.1). The relationship between dtx and $ltx_{bc,dtx}$ is demonstrated in Figure 3 using an example that shows a TCCSCI with four `invoke_sc` operations that invoke smart contract functions on two different blockchain networks.

Furthermore, we assume an off-chain transaction manager component TM coordinates a commitment protocol P with the purpose of guaranteeing the global atomicity of dtx. P is triggered when the client application requests the execution of the `commit_dtx` operation. We assume that TM may suffer from crashes at any time and recovers after some arbitrary amount of time. Information stored by the blockchain persistence layer is never lost. Information logged on the disk TM survives later crashes. Each blockchain network bc unilaterally evaluates a *vote* that can be either "yes" or "no", which indicates whether it can commit the local transaction $ltx_{bc,dtx}$, which results from the TCCSCI dtx.

The problem is to have the blockchain networks collectively *guarantee the global serializability* of the TCCSCI dtx and *decide* on either *commit* or *abort* considering the following requirements that guarantee the *global atomicity* of dtx:

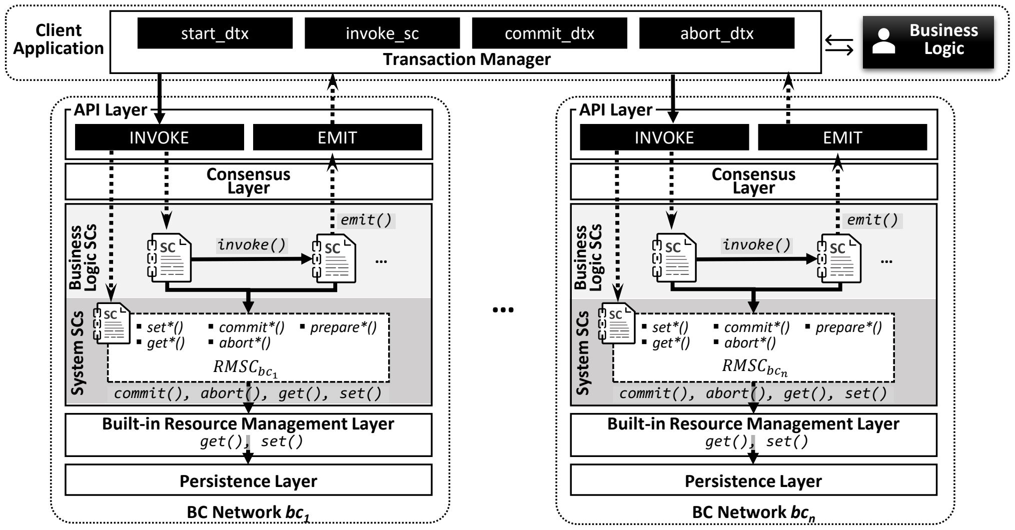


Fig. 4. An overview of the TCCSCI approach that guarantees global atomicity and global serializability of TCCSCIs.

- *Agreement:* no two blockchain networks decide differently.
- *Termination:* if all TM failures are repaired and it does not crash for long enough, then all blockchain networks decide.
- *Abort-Validity:* abort is the only possible decision if some blockchain networks vote "no".
- *Commit-Validity:* commit is the only possible decision if TM is operative and all blockchain networks vote "yes".

5 THE TRANSACTIONAL CROSS-CHAIN SMART CONTRACT INVOCATION APPROACH

In this section, we present the *TCCSCI approach*, which guarantees global serializability and atomicity of business transactions that involve the invocation of smart contract functions located in different, possibly heterogeneous blockchain networks. We aim to fulfill the following requirements that ensure a relatively low adoption barrier: (i) No dependence on TTPs, i.e., the approach must not require that the client application or blockchain networks have to trust third-parties. (ii) Achieve *arbitrary smart contract composition*, i.e., the approach must allow the client application to arbitrarily specify which smart contract functions are invoked on which blockchain network and in which order, and it must allow the client application to pass data resulting from one invocation to another. (iii) The approach must *allow heterogeneity*, i.e., it must be applicable to both permissioned and permissionless blockchains, and to any blockchain system within these two categories that adheres to the system model presented in Section 4.1. (iv) The approach must not require *changing existing blockchain protocols*. We evaluate the fulfillment of these requirements in Section 8.

5.1 Conceptual Overview of the TCCSCI Approach

The overall idea of the TCCSCI approach is to use smart contracts that act as resource managers to ensure the global serializability of TCCSCIs, and to implement a variant of the 2PC protocol to ensure their global atomicity. Specifically, blockchain systems are designed to ensure the atomicity and isolation of the individual smart contract function invocations they run. However, in the context of ensuring transactional behavior for CCSCIs, we might need to roll back the effects of an already-finished invocation as demonstrated in Section 3. Therefore, our approach needs to find a way to separate the execution of a smart contract function invocation from its commitment, i.e., have the ability to provisionally execute an invocation and later decide whether to commit the incurred changes, i.e., "install" them durably in the persistence layer, or to revoke them. Traditionally, such a feature is provided by the concept of a resource manager [4]. However, on the one hand, built-in blockchain resource management layers are not designed for TCCSCIs, i.e., they cannot ensure their atomicity and serializability. On the other hand, we do not want to introduce changes to them, since this means requiring to change existing blockchain protocols, which limits the adoptability of the approach.

Therefore, we propose to implement the functionality of a resource manager that supports TCCSCIs directly in the smart contract layer in the form of a *Resource Manager Smart Contract* as shown in Figure 4. We assume that for each blockchain network $bc_1, ..., bc_n$, we have a dedicated RMSC, $RMSC_{bc_1}, ..., RMSC_{bc_n}$. Please note how RMSCs expose a set of functions similar, except for `prepare*`, to the basic operations exposed by the built-in resource management layer (to differentiate between the two, we add the symbol "*" to the end of RMSC function names). Given a blockchain network bc, $RMSC_{bc}$ acts as a layer on top of the built-in resource management layer of bc. Therefore, when a user-defined smart contract sc in bc needs to participate in a TCCSCI, it must direct all its read and write requests to $RMSC_{bc}$ by invoking the functions `set*` and `get*` using the `invoke` basic operation instead of directly calling the basic operations `set` and `get` of the built-in resource management layer. As discussed in Section 5.2, `set*` and `get*` are implemented in a way that (i) ensures serializability by implementing *Strict Two-Phase Locking (S2PL)* [3], and (ii) facilitates recovery, i.e., recover the previous values of changed data items if a TCCSCI is aborted (by maintaining *previous images*).

To support the global atomicity of TCCSCIs, the TCCSCI approach introduces a transaction manager component as part of the client application. We denote this component TM. The purpose of TM is to coordinate a specialized ACP derived from the traditional 2PC protocol. As we will see in Section 5.3, given a TCCSCI dtx, this protocol, which we refer to as the *2PC4BC protocol*, will ensure that all the corresponding local transactions are either committed or aborted. To this end, when the business logic of the client application is successfully executed until the end, i.e., when the operation $commit\_dtx$ is executed, TM will communicate with the RMSCs of the involved blockchain networks to run the 2PC4BC protocol by: (i) invoking the `prepare*` function provided by the smart contracts $RMSC_{bc_1}, ..., RMSC_{bc_n}$ of all participating blockchain networks $bc_1, ..., bc_n$, which requests them to vote on whether they are willing to commit their local transactions $ltx_{bc_1,dtx}, ..., ltx_{bc_n,dtx}$. (ii) If all vote "yes", requesting them to perform the commit by invoking the `commit*` function; otherwise, requesting them to abort by invoking the `abort*` function. The `commit*` function of a given $RMSC_{bc}$ ensures that the effects of the local transaction $ltx_{bc,dtx}$ are "installed" in the persistence layer of bc, whereas the `abort*` function restores the previous images of the data items manipulated in $ltx_{bc,dtx}$, thus effectively revoking it. As a result, the global atomicity of dtx is guaranteed. In the next two sections, we will give more details on the RMSCs and the 2PC4BC protocol.

5.2 Resource Manager Smart Contract

As mentioned earlier, the main purposes of an RMSC is to (i) enable business logic smart contracts to separate their execution from the commitment of the incurred changes, which allows for rolling back executed but uncommitted function invocations if deemed necessary to ensure the global atomicity of a TCCSCI, and (ii) ensure that the access to data items is controlled in such a way that prevents unintentional interactions between concurrent TCCSCIs, which ensures global serializability. An example of unintentional interactions between concurrent TCCSCIs is accessing uncommitted changes that might still be revoked later. Every blockchain network bc involved in the execution of a TCCSCI requires an $RMSC_{bc}$ deployed on it. This $RMSC_{bc}$ will then be used to support the execution of the local transactions $ltx_{bc,dtx}$ of all TCCSCIs. Every $RMSC_{bc}$ exposes five public functions, `set*`, `get*`, `commit*`, `abort*`, and `prepare*`. We will postpone the discussion of `commit*`, `abort*`, and `prepare*` to the next section since they are more

Listing 1 Pseudocode for $RMSC_{bc}$: `set*` and `get*` functions along with other helping functions.

```
Context: $seq_b$, $id_{c,bc}$
Data: vars[], txs[], $TimeoutDuration$
external function set*($vName$, $value$, $id_{dtx}$):
    require(ensureStarted($id_{dtx}$) = $true$)
    require(txs[$id_{dtx}$].$owner$ = $id_{c,bc}$)
    [$successful$, $newLock$] ← acquireLock($vName$, $id_{dtx}$,
      $WRITE$)
    if $successful$ = $false$ then // Cannot acquire the lock!
        doAbort_ltx($id_{dtx}$)
    else
        if $newLock$ = $true$ then // 1$^{st}$ write to variable?
            vars[$vName$].$prevImage$ ← vars[$vName$].$value$
        vars[$vName$].$value$ ← $value$
external function get*($vName$, $id_{dtx}$):
    require(ensureStarted($id_{dtx}$) = $true$)
    require(txs[$id_{dtx}$].$owner$ = $id_{c,bc}$)
    [$successful$, _] ← acquireLock($vName$, $id_{dtx}$, $READ$)
    if $successful$ = $false$ then // Cannot set the lock!
        doAbort_ltx($id_{dtx}$)
    else
        return vars[$vName$].$value$
internal function ensureStarted($id_{dtx}$):
    if txs[$id_{dtx}$] = $\perp$ then // We have to initialize $dtx$
        txs[$id_{dtx}$].$owner$ ← $id_{c,bc}$
        txs[$id_{dtx}$].$state$ ← $STARTED$
        txs[$id_{dtx}$].$timeout$ ← $seq_b + TimeoutDuration$
        return $true$
    return txs[$id_{dtx}$].$state$ = $STARTED$
internal function acquireLock($vName$, $id_{dtx}$, $lType$):
    $v$ ← vars[$vName$]
    $tx$ ← txs[$id_{dtx}$]
    if ($lType$ = $WRITE$ ∧ $tx$ has write lock over $v$) ∨ ($lType$ =
      $READ$ ∧ $tx$ has any lock over $v$) then // No need to lock
        [$successful$, $newLock$] ← [true, false]
    else if ($lType$ = $READ$ ∧ $v$ is not write-locked) ∨ ($lType$ =
      $WRITE$ ∧ no other transaction has locks over $v$) then
        lock($vName$, $id_{dtx}$, $lType$)
        [$successful$, $newLock$] ← [true, true]
    else // There are conflicting locks!
        $lHolders \leftarrow v.rlHolders \cup \{v.wlHolder\}$
        if $\forall h \in lHolders$ (txs[$h$].$timeout \leq seq_b \wedge$
          txs[$h$].$state$ = $STARTED$) then
            foreach $h \in lHolders$ do
                doAbort_ltx($h$)
            lock($vName$, $id_{dtx}$, $lType$)
            [$successful$, $newLock$] ← [true, true]
        else // We cannot set the lock
            [$successful$, $newLock$] ← [false, false]
    return [$successful$, $newLock$]
internal function lock($vName$, $id_{dtx}$, $lType$):
    if $lType$ = $WRITE$ then
        vars[$vName$].$wlHolder$ ← $id_{dtx}$
        vars[$vName$].$rlHolders$ ← $\emptyset$
    else
        vars[$vName$].$rlHolders$ ←
          vars[$vName$].$rlHolders \cup \{id_{dtx}\}$
```

relevant for the proposed 2PC4BC protocol, and focus instead on `set*` and `get*` that represent the interface exposed to user-defined smart contracts willing to participate in TCCSCIs.

5.2.1 Pseudocode Syntax. The pseudocode of the `set*` and `get*` functions and other required helping functions are shown in Listing 1. Throughout the pseudocode, the basic smart contract operations `set` and `get` exposed by the built-in resource management layer are implicitly executed whenever we assign a value to or read a value from the data items stored in the persistence layer. The data items that the smart contract $RMSC_{bc}$ accesses are declared in the `Data` block of the pseudocode. These data items are only accessible via $RMSC_{bc}$ functions, i.e., no other smart contract can directly read or manipulate them. Moreover, the basic operation `abort` is implicitly executed whenever the pseudocode command `require(<condition>)` is invoked and the `<condition>` evaluates to `false`, while the basic operation `commit` is implicitly triggered when any function execution finishes without incurring an `abort` operation. Finally, the context information provided by the consensus layer, i.e., the sequence number seq_b of the block that contains the current smart contract function invocation request and the client application identity $id_{c,bc}$ within the local blockchain network bc are made available to all functions of $RMSC_{bc}$ by declaring them in the `Context` pseudocode block. Please note that having exclusive access to certain data items stored in the persistence layer and having access to the context information are common capabilities for all smart contracts and are not restricted to RMSCs.

5.2.2 Needed Data Items and Their Structures. The `Data` pesudocode block shows that $RMSC_{bc}$ has access to three data items, $vars$, txs, and $TimeoutDuration$: (i) $vars$ is a mapping data item that hosts a set of variables. The purpose of these variables is to represent the data items that user-defined smart contracts want to have access to during TCCSCI executions. User-defined smart contracts can access these variables only via the `set*` and `get*` functions. This allows

$RMSC_{bc}$ to embed the locking mechanism needed to ensure global serializability into them. Each variable $v \in vars$ is accessible inside $RMSC_{bc}$ using the notation $vars[name]$ and is defined as follows: $v := (name, value, prevImage, wlHolder, rlHolders)$. Here, $name$ represents the name of the variable, $value$ is its current value, $prevImage$ is the value before the first successful `set*` operation that accesses v in the current local transaction, $wlHolder$ is the identifier of the local transaction that holds a write lock over v (or null $\perp$), and $rlHolders$ is the set of identifiers of the local transactions that hold read locks over v (if any). A new variable is added to the $vars$ mapping whenever a user-defined smart contract tries to access a variable with an unknown $name$ using the `set*` or `get*` functions. (ii) txs is a mapping data item that represents all existing local transactions of the blockchain network bc. Each $tx \in txs$ is accessible inside $RMSC_{bc}$ using the notation $txs[id_{dtx}]$ and contains the necessary information to enable the corresponding local transaction to participate in the 2PC4BC protocol. Hence, it is defined as follows: $tx := (id_{dtx}, owner, state, timeout)$. Here, id_{dtx} represents the identifier of the TCCSCI. We assume that this identifier is unique within all blockchain networks involved in the TCCSCI dtx. Therefore, it is used also as an identifier for every local transaction that originates from dtx. The function `start_tx` in TM (as shown later) is responsible for generating id_{dtx} when dtx starts. Furthermore, $owner$ represents the identifier $id_{c,bc}$ of the client application c that executes the TCCSCI, $state \in \{STARTED, PREPARED, COMMITTED, ABORTED\}$ represents the current state of the local transaction $ltx_{bc,dtx}$, and $timeout$ represents the latest block sequence number in which the local transaction is allowed to maintain locks if it is still in the $STARTED$ state. (iii) $TimeoutDuration$ is a data item representing the duration of time, measured in the number of generated blocks, in which any local transaction in the blockchain network bc is allowed to remain in the $STARTED$ state.

5.2.3 The $RMSC_{bc}$ Functions. The function `set*` is marked with the keyword `external`, which means it can be invoked from outside the smart contract $RMSC_{bc}$. In fact, it needs to be invoked by any user-defined smart contract in the same blockchain network willing to update the value of a specific variable in the context of some local transaction $ltx_{bc,dtx}$.

With the help of the private, i.e., `internal`, function `ensureStarted`, `set*` ensures that the local transaction $ltx_{bc,dtx}$ either is already in the $STARTED$ state, or does not yet exist, in which case a new entry for it in the txs mapping is created and initialized. Next, the function `set*` ensures that the client application from which the current invocation request originates has the same identity $id_{c,bc}$ as the owner of the local transaction $ltx_{bc,dtx}$. This is necessary to ensure that no other client application is allowed to interfere with the execution of the TCCSCI on the current blockchain network bc. Next, the function tries to acquire a write lock over the variable to be set with the help of the `acquireLock` internal function. If this was unsuccessful (see line 5), the local transaction is immediately aborted using the private function `doAbort_ltx`, which we will explain in Section 5.3. Otherwise, the execution moves on to check if this is the first time the current local transaction changes the value of the variable, in which case it stores the previous value in the field $prevImage$. Finally, it changes the current value of the variable by assigning the new value to it.

The public function `get*` is invoked by any user-defined smart contract in the same blockchain network bc willing to read the value of a specific data item in the context of the local transaction identified with id_{dtx}. The function first performs checks similar to `set*`, then it tries to acquire a read lock over the specified variable. If this is unsuccessful (see line 15), the TCCSCI is aborted. Otherwise, the value of the variable is returned to the invoking smart contract function.

Both `set*` and `get*` utilize the private function `acquireLock`, which tries to set a lock of the type specified by the argument $lType \in \{WRITE, READ\}$ on a specific variable v. To this end, it follows a set of well-known rules [3] to find out if there are conflicting locks already set over the variable v. In short, if a given local transaction is willing to have a write lock over v, any existing locks from concurrent local transactions are considered to conflict with it, whereas if it is willing to have a read lock over v, then any existing write lock from a concurrent local transaction is considered to

Listing 2 Pseudocode for the $RMSC_{bc}$ functions relevant for the 2PC4BC protocol.

```
Context: $seq_b$, $id_{c,bc}$
Data: vars[], txs[], $TimeoutDuration$
external function prepare*($id_{dtx}$):
    require(txs[$id_{dtx}$].$owner = id_{c,bc}$)
    require(txs[$id_{dtx}$].$state \in \{STARTED, ABORTED\}$)
    if txs[$id_{dtx}$].$state = STARTED$ then
        txs[$id_{dtx}$].$state \leftarrow PREPARED$
        emit("VOTE_EVENT", $id_{dtx}$, "YES")
    else
        emit("VOTE_EVENT", $id_{dtx}$, "NO")
external function commit*($id_{dtx}$):
    require(txs[$id_{dtx}$].$owner = id_{c,bc}$)
    require(txs[$id_{dtx}$].$state = PREPARED$)
    txs[$id_{dtx}$].$state \leftarrow COMMITTED$
    releaseLocks($id_{dtx}$)
external function abort*($id_{dtx}$):
    require(txs[$id_{dtx}$].$owner = id_{c,bc}$)
    require(txs[$id_{dtx}$].$state \neq COMMITTED$)
    if txs[$id_{dtx}$].$state \neq ABORTED$ then
        doAbort_ltx($id_{dtx}$)
internal function doAbort_ltx($id_{dtx}$):
    foreach $v \in vars$ do
        if $v.wlHolder = id_{dtx}$ then
            $v.value \leftarrow v.beforeImage$
    txs[$id_{dtx}$].$state \leftarrow ABORTED$
    releaseLocks($id_{dtx}$)
internal function releaseLocks($id_{dtx}$):
    foreach $v \in vars$ do
        if $v.wlHolder = id_{dtx}$ then
            $v.wlHolder \leftarrow \perp$
        else if $id_{dtx} \in v.rlHolders$ then
            $v.rlHolders \leftarrow v.rlHolders \setminus \{id_{dtx}\}$
```

conflict with it. If no conflicting locks are detected, `acquireLock` locks the variable (with the help of the `lock` function), but if conflicting locks are detected, it checks if all the corresponding local transactions have timed-out while still being in the *STARTED* state (see line 36). If that is the case, all conflicting local transactions are aborted, and the required lock is set. This ensures that local transactions do not hold locks forever, which would, otherwise, indefinitely "starve" other local transactions willing to access the locked variables. Obviously, if the current local transaction already has a suitable lock on v, then `acquireLock` will do nothing.

5.3 Two-Phase Commit for Blockchains Protocol

We now present the 2PC4BC protocol, which aims to achieve global atomicity in the CCSCIs (see Section 4.2). The 2PC4BC protocol is based on the original 2PC protocol [10] and, therefore, defines the same two protocol roles: (i) *participants*, which are the blockchain networks that want to reach an agreement regarding the commitment of a TCCSCI dtx. In fact, since we aim to avoid changing existing blockchain protocols, the participant's logic is not implemented as part of the blockchain peer node itself, but rather embedded into the RMSC of the corresponding blockchain network. The second role defined is (ii) the *coordinator*, which is responsible for accumulating the votes from all participants, and accordingly coming up with a decision to commit or abort as defined by the original 2PC protocol. This role is played by the TM component of the client application (see Figure 4). During protocol execution, the coordinator sends messages to the participants using smart contract function invocation requests, and the participants respond using smart contract events (see Section 4.1). In fact, participants never exchange messages among each other nor initiate communication with the coordinator. The protocol supports the four requirements defined in the problem statement: Agreement, Termination, Abort-Validity, and Commit-Validity as shown later in Section 7.1.

We start by introducing the pseudocode for the roles that take part in the 2PC4BC protocol, and then we describe the protocol itself in detail. First, Listing 2 shows the pseudocode of the functions of the RMSC relevant for the protocol. Specifically, the functions `prepare*`, `commit*`, and `abort*` are public and invoked by TM as part of the protocol's interactions. These functions ensure the authenticity and validity of the received requests, i.e., that the current state permits executing them, then change the state accordingly. Moreover, they perform additional functionality such as voting (`prepare*`) and releasing locks (`commit*` and `abort*`).

Second, Listing 3 shows the pseudocode of the TM component. We assume the component uses the mapping $SDK[]$ declared in the `Input` block to have access to the set of Software Development Kits (SDKs) that allow it to communicate with the underlying blockchain networks via: (i) the `invoke` operation, which allows invoking a smart contract function, and (ii) the `listenTo` operation, which allows waiting for a smart contract event of a specific type to be emitted and delivers it to the component. These operations correspond to the functions `INVOKE` and `EMIT` of the API layer of the blockchain system model we presented in Section 4.1. When these two operations are invoked in the code, we use the keyword `async` to emphasize that their execution is asynchronous, i.e., TM does not wait for the operation to complete, but rather moves immediately to the next operation, since it may take up to a few minutes depending on the corresponding blockchain network. Furthermore, we assume the TM component has access to the addresses of all relevant RMSCs via the mapping $RMSC[]$ and to the identifiers of all RMSC functions that take part in the 2PC4BC protocol, i.e., `prepare*`, `commit*`, and `abort*`, via the constants $id_{prepare}, id_{commit}, id_{abort}$ respectively. This information is also provided in the `Input` block at the top of Listing 3. Moreover, the component stores information about existing TCCSCIs in the mapping $txs[]$ that is maintained in a local, non-volatile storage. All changes to this set are written synchronously and atomically to the local storage so that they survive any subsequent crashes. The $txs[]$ mapping is declared in the `Data` block in Listing 3. For each $tx \in txs$, the TM component maintains the following information, which represents one TCCSCI, $tx := (id_{dtx},\ bcs,\ state,\ verdict,\ yes)$. Here, id_{dtx} is the unique identifier of the TCCSCI, $bcs \subseteq SDK$ represents the SDKs of the blockchain networks that take part in the TCCSCI, $state \in \{\ AWAITING\_REQUESTS, AWAITING\_VOTES, ABORTED, COMMITTED\ \}$ is the current 2PC4BC protocol state while processing the TCCSCI, $verdict \in \{\ COMMIT, ABORT, \perp\ \}$ is the decision reached by TM on whether to commit or abort the TCCSCI, and yes is a counter for "yes" votes. The pseudocode is divided into handlers marked with the keyword `handle`. Each handler represents the logic for handling a specific event. Four handlers, i.e., (i) `start_dtx()`, (ii) `invoke_sc(`$id_{dtx}, id_{bc}, id_{sc}, id_f, args$`)`, (iii) `commit_dtx(`$id_{dtx}$`)`, and (iv) `abort_dtx(`$id_{dtx}$`)`, handle request messages submitted by the business logic component of the client application. Furthermore, `bc.emit("VOTE_EVENT", e)` handles the voting events received from participant blockchain networks, and `bc.emit("ERROR_EVENT", e)` handles all error events reported by the involved resource manager and user-defined smart contracts. Finally, the `crashRecovery()` is triggered whenever TM recovers back from a crash, and it has the goal of handling recovery. In addition to these handlers, the pseudocode introduces the logic of two helper functions `doAbort_dtx` and `doCommit_dtx` that are only used internally. In the following, we explain the 2PC4BC protocol in two steps: we start by explaining the normal execution, and then we explain the various types of possible failures that may occur and how the protocol handles them.

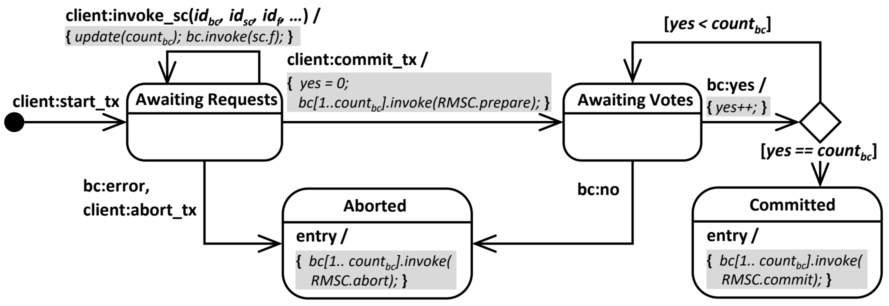
5.3.1 Normal Execution. Figure 5 presents state machine diagrams for the coordinator and the participants as they execute a TCCSCI, which includes running the 2PC4BC protocol at the end to achieve agreement over its commitment. The business logic component of the client application marks the beginning of the TCCSCI by submitting a `start_dtx` request to TM, which causes it to move to the "Awaiting Requests" state. While in this state, TM receives smart contract function invocation requests from the business logic component of the client application (see line 5 of Listing 3), which it submits to the corresponding blockchain networks. At this stage, the RMSCs of these blockchain networks support the execution of the invoked smart contracts (as explained in Section 5.2) and are, therefore, in the "Started" state. At some point, the business logic decides to commit the TCCSCI, so it sends a `commit_dtx` request to TM, which triggers the start of the 2PC4BC protocol. As the name suggests, the protocol has two phases. In *Phase 1*, the coordinator sends a voting request to all participating blockchain networks by invoking the `prepare*` function of their RMSCs, and moves to the "Awaiting Votes" state. When a participant receives the request while being in the "Started" state, it knows it will

Listing 3 Pseudocode for the TM component functions that manage the execution of the 2PC4BC protocol.

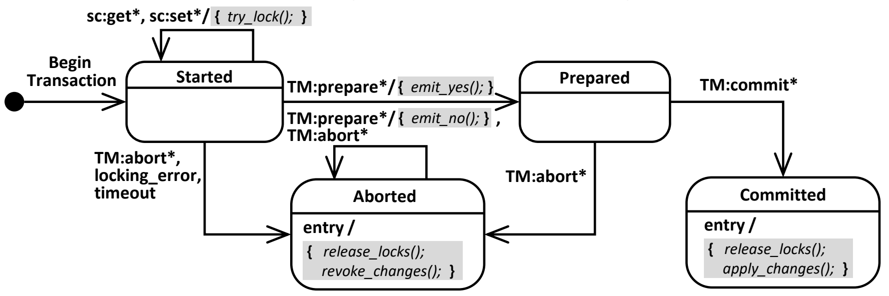
```
   Input: SDK[], RMSC[], $id_{prepare}$, $id_{commit}$, $id_{abort}$
   Data: txs[]
1  handle start_dtx():
2      $id_{dtx} \leftarrow$ generate()
3      txs[$id_{dtx}$].$state \leftarrow AWAITING\_REQUESTS$
4      return $id_{dtx}$
5  handle invoke_sc($id_{dtx}$, $id_{bc}$, $id_{sc}$, $id_f$, $args$):
6      if txs[$id_{dtx}$].$state$=$AWAITING\_REQUESTS$ then
7          $bc \leftarrow$SDK[$id_{bc}$]
8          if $bc \notin$ txs[$id_{dtx}$].$bcs$ then
9              async $bc$.listenTo("ERROR_EVENT")
10             txs[$id_{dtx}$].$bcs \leftarrow$ txs[$id_{dtx}$].$bcs \cup \{bc\}$
11         async $bc$.invoke($id_{sc}$, $id_f$, $args$)
12 handle commit_dtx($id_{dtx}$):
13     if txs[$id_{dtx}$].$state$=$AWAITING\_REQUESTS$ then
14         txs[$id_{dtx}$].$state \leftarrow AWAITING\_VOTES$
15         txs[$id_{dtx}$].$yes \leftarrow 0$
16         foreach $bc \in$ txs[$id_{dtx}$].$bcs$ do
17             async $bc$.listenTo("VOTE_EVENT")
18             async $bc$.invoke(RMSC[$bc$], $id_{prepare}$, $\{id_{dtx}\}$)
19 handle abort_dtx($id_{dtx}$):
20     if txs[$id_{dtx}$].$state$=$AWAITING\_REQUESTS$ then
21         doAbort_dtx($id_{dtx}$)
22 handle bc.emit("ERROR_EVENT", $e$):
23     doAbort_dtx($e.args[1]$)
24 handle bc.emit("VOTE_EVENT", $e$):
25     $id_{dtx} \leftarrow e.args[1]$
26     if txs[$id_{dtx}$].$state$=$AWAITING\_VOTES$ then
27         if $e.args[2]$ = "NO" then
28             doAbort_dtx($id_{dtx}$)
29         else if $++yes = |$txs[$id_{dtx}$].$bcs|$ then
30             doCommit_dtx($id_{dtx}$)
31 handle crashRecovery():
32     foreach $tx \in txs$ do
33         $id_{dtx} \leftarrow tx.id$
34         if $tx.state = AWAITING\_VOTES$ then
35             if $tx.verdict \in \{\perp, ABORT\}$ then
36                 doAbort_dtx($id_{dtx}$)
37             else
38                 doCommit_dtx($id_{dtx}$)
39 function doAbort_dtx($id_{dtx}$):
40     txs[$id_{dtx}$].$verdict \leftarrow ABORT$
41     foreach $bc \in$ txs[$id_{dtx}$].$bcs$ do
42         async $bc$.invoke(RMSC[$bc$], $id_{abort}$, $\{id_{dtx}\}$)
43     txs[$id_{dtx}$].$state \leftarrow ABORTED$
44 function doCommit_dtx($id_{dtx}$):
45     txs[$id_{dtx}$].$verdict \leftarrow COMMIT$
46     foreach $bc \in$ txs[$id_{dtx}$].$bcs$ do
47         async $bc$.invoke(RMSC[$bc$], $id_{commit}$, $\{id_{dtx}\}$)
48     txs[$id_{dtx}$].$state \leftarrow COMMITTED$
```

be able to commit it afterward since TCCSCI already has all the required locks. Therefore, it responds with a "yes" vote, which is transferred back to the coordinator via a smart contract event, and moves to the "Prepared" state. In *Phase 2*, the coordinator collects participants' votes, and when it detects that all participants have voted "yes", it decides to commit the TCCSCI, moves to the "Committed" state, and informs the participants about the decision by invoking the `commit*` function of their RMSCs. As a result, the participants also move to the "Committed" state, permanently apply the local changes of the TCCSCI, and release all locks obtained during its execution. At this stage, the protocol is over.

5.3.2 Failure Handling. Multiple types of failure may affect the execution of the 2PC4BC protocol. First, during the execution of a TCCSCI dtx, an invoked smart contract that represents some service may fail, e.g., due to a violation of its business logic or a bug in its implementation. In this case, the handler `bc.emit("ERROR_EVENT", e)` of the TM component (see line 22 in Listing 3) is triggered and the TCCSCI is aborted. A viable alternative is to forward such errors to the business logic of the client application and let it decide whether to abort the TCCSCI or handle the error differently, e.g., by trying a different smart contract that provides a similar business service. Second, even if all smart contract invocations were successful, the client application might still decide to abort before triggering `commit_dtx` due to its own business logic. In this case, it sends an `abort_dtx` request to the TM component (see line 19 of Listing 3), which aborts the TCCSCI. Third, an invocation of a smart contract function hosted in one of the participant blockchain networks bc might fail because it was not able to obtain a lock over an accessed data item (see Section 5.2). In this case, $RMSC_{bc}$ will unilaterally abort the local transaction $ltx_{bc,dtx}$ using the function `doAbort_ltx` (see Listing 2), causing it to move to the "Aborted" state (see Figure 5b). Aborting $ltx_{bc,dtx}$ entails restoring the previous images for all data items changed during the execution, and releasing all obtained locks. Later, when the 2PC4BC protocol is triggered, $RMSC_{bc}$ will vote "no" in *Phase 1*, and the whole TCCSCI will be aborted, i.e., all relevant local transactions will also be aborted.


client:invoke_sc(id_bc, id_sc, id_f, ...) /
{ update(count_bc); bc.invoke(sc.f); }
client:commit_tx /
{ yes = 0; bc[1..count_bc].invoke(RMSC.prepare); }
[yes < count_bc]
client:start_tx
Awaiting Requests
Awaiting Votes
bc:yes /
{ yes++; }
[yes == count_bc]
bc:error, client:abort_tx
bc:no
Aborted
entry /
{ bc[1.. count_bc].invoke(RMSC.abort); }
Committed
entry /
{ bc[1.. count_bc].invoke(RMSC.commit); }

(a) The state machine diagram for the Transaction Manager component.


sc:get*, sc:set*/ { try_lock(); }
Begin Transaction
Started
TM:prepare*/ { emit_yes(); }
Prepared
TM:commit*
TM:prepare*/ { emit_no(); } , TM:abort*
TM:abort*, locking_error, timeout
Aborted
entry /
{ release_locks(); revoke_changes(); }
TM:abort*
Committed
entry /
{ release_locks(); apply_changes(); }

(b) The state machine diagram for a Resource Manager Smart Contract.

Fig. 5. The proposed 2PC4BC protocol expressed as two state machine diagrams for the TM component and the RMSCs.

Another type of possible failures is a TM crash. We assumed in the problem statement (see Section 4.2) that TM may crash at any time, and that it will eventually be operational again after an arbitrary period of time. After it recovers, it only "remembers" what was stored in its stable storage prior to the crash, i.e., the contents of the txs mapping (see Listing 3), and it executes the handler `crashRecovery()` (see line 31 of Listing 3) that has the goal of ensuring the correctness of the protocol execution despite the crash failure. To see how, we differentiate between three points in the lifecycle of the protocol in which TM may crash: (i) If the crash happens after the TCCSCI is started, but before `commit_dtx` is triggered, i.e., TM is in the "Awaiting Requests" state, and the involved RMSCs are in the "Started" state, then it is safe for any RMSC to unilaterally abort the local transaction since it has not voted yet. This is useful to ensure that the data items locked by the TCCSCI do not become inaccessible for other TCCSCIs for a long period of time. To trigger such an abort, each RMSC has a timer based on the current block sequence of the blockchain network for all local transactions in the "Started" phase. Any access request to one of the data items locked by the local transaction after it has timed-out in the blockchain network bc will cause it to be aborted (see line 36 of Listing 1). As discussed earlier, each RMSC that aborts a local transaction of a TCCSCI will vote "no" if the client application attempts to commit it. Therefore, the TM component has no special logic to handle this situation after it recovers from the crash. (ii) If the crash happens after `commit_dtx` is triggered but before a verdict was reached, TM will detect the following condition in its stable storage after it recovers: $txs[id_{dtx}].state = AWAITING\_VOTES \wedge txs[id_{dtx}].verdict = \perp$. Since TM has not reached a verdict yet, it can safely abort the TCCSCI (see lines 34–36 of Listing 3). (iii) If the crash happens after a verdict is reached but before sending it to all participants, TM will detect the following condition in its stable storage after it recovers: $txs[id_{dtx}].state = AWAITING\_VOTES \wedge txs[id_{dtx}].verdict \in \{ABORT, COMMIT\}$. In this case, TM retransmits the verdict to all involved RMSCs (see lines 34–38 of Listing 3).

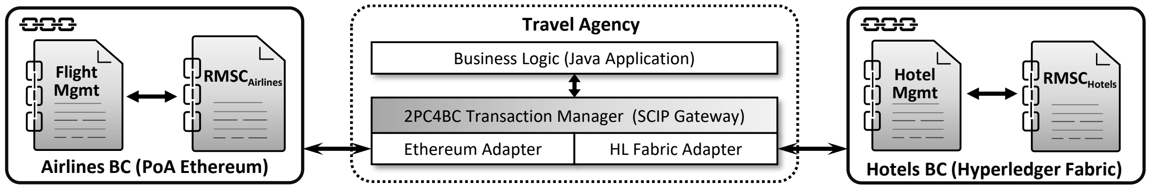

Flight Mgmt
RMSC Airlines
Airlines BC (PoA Ethereum)
Travel Agency
Business Logic (Java Application)
2PC4BC Transaction Manager (SCIP Gateway)
Ethereum Adapter
HL Fabric Adapter
Hotel Mgmt
RMSC Hotels
Hotels BC (Hyperledger Fabric)

Fig. 6. A prototypical implementation of the travel agency scenario demonstrating the practical feasibility of the TCCSCI approach.

6 PROTOTYPICAL IMPLEMENTATION

To prove the practical feasibility of our approach, we have used it to prototypically implemented the motivational scenario presented in Section 3. To highlight the heterogeneity capabilities of the approach, we assume that *Airlines BC* is an Ethereum-based permissioned blockchain network that uses Proof-of-Authority (PoA) [28], while *Hotels BC* is a Hyperledger Fabric-based permissioned blockchain network that uses Raft [16]. Figure 6 shows an overview of the prototype. Components shown in gray are newly developed or significantly adapted to support our approach. The other components are re-used. Specifically, for *Airlines BC*, we implemented $RMSC_{Eth}$, a generic Ethereum RMSC using the *Solidity* language, which makes it also compatible with any other blockchain system that uses the EVM [34]. Moreover, we implemented a simple $FlightManagement$ smart contract that utilizes $RMSC_{Eth}$ to enable it to participate in TCCSCIs. Similarly, for *Hotels BC*, we implemented $RMSC_{HLF}$, a generic Hyperledger Fabric RMSC using JavaScript, which can be used on any Hyperledger Fabric blockchain network. Furthermore, we implemented a simple $HotelManagement$ smart contract that utilizes $RMSC_{HLF}$ to enable it to participate in TCCSCIs.

To implement the TM component, we extended the *SCIP Gateway*, which is an implementation of the Smart Contract Invocation Protocol (SCIP) [6, 8]. SCIP aims to build an abstraction layer on top of heterogeneous blockchain networks by providing client applications with a set of technology-agnostic methods to invoke smart contract functions and to perform other operations common in most blockchain technologies. The *SCIP Gateway* is a software component that communicates with client applications using SCIP and with blockchain networks using their own protocols. To this end, it utilizes technology-specific adapters. Currently, adapters for five different smart contract-enabled blockchain technologies are available. In this work, we have extended the *SCIP Gateway* with the functionality of a 2PC4BC TM.

With the help of the 2PC4BC TM, client applications, such as the application that hosts the business logic of the travel agency, can execute TCCSCIs that can span a heterogeneous set of blockchain networks while only having to use a technology-agnostic API. The motivational scenario demonstrated in Figure 1 shows that if both travel agencies interact with the two blockchain networks using TCCSCIs according to the TCCSCI approach, then the TCCSCI of *Travel Agency 1* will be aborted since the smart contract invocation in step ❷ fails. This will lead to reverting the effects of the invocation in step ❶ (with the help of the introduced $RMSC_{Eth}$), thus guaranteeing the global atomicity of the TCCSCI. Note that *Travel Agency 1* and *Travel Agency 2* do not have to use the same 2PC4BC transaction manager to participate in globally serializable and atomic TCCSCIs. Instead, each of them can use its own instance thereof highlighting the fact that 2PC4BC TMs are not TTPs. The entire prototype is publicly accessible on GitHub[6].

7 EVALUATION

In this section, we prove the approach's correctness, discuss security implications, and calculate time and cost overhead.

[6]Ethereum smart contracts: https://github.com/TIHBS/EthereumResourceManager, Fabric Chaincode: https://github.com/TIHBS/fabric-resource-manager, SCIP Gateway: https://github.com/TIHBS/BlockchainAccessLayer

7.1 Correctness of the TCCSCI Approach

We now show that the TCCSCI approach solves the problem stated in Section 4.2. First, we show that the approach guarantees global serializability, i.e., that the execution of concurrent TCCSCIs is equivalent to some serial execution thereof. The RMSC we introduce in our approach (see Section 5.2) implements the S2PL algorithm [3]. To see why, consider $RMSC_{bc}$, the RMSC of the blockchain network bc, and note that accessing shared data items hosted on bc's persistence layer is protected via locks in a way that prevents conflicting operations from occurring (see the function `acquireLock` in Listing 1). Furthermore, the locks obtained during the execution of the local transaction $ltx_{bc,dtx}$ of a TCCSCI dtx, are only released after $ltx_{bc,dtx}$ finishes with either a commit or an abort (in Listing 2, see how the function `releaseLocks` is only invoked at the end of a commit or an abort operation). Hence, $RMSC_{bc}$ implements the strict version of the 2PL algorithm, i.e., S2PL. It is shown in the literature that if S2PL is used at each site in a distributed transaction, then global serializability is guaranteed [3, pp. 77–78]. Since we assume an RMSC is used for each blockchain network involved in the TCCSCI approach, global serializability is indeed guaranteed.

Second, we show that the approach guarantees global atomicity, i.e., that the 2PC4BC protocol fulfills the four requirements mentioned in Section 4.2: (i) 2PC4BC guarantees *Agreement.* To see why, notice that $RMSC_{bc}$ decides to commit the local transaction $ltx_{bc,dtx}$ of a given TCCSCI dtx *if and only if* it receives a `commit*` request from TM. In turn, when TM sends a `commit*` request, it sends it to all involved RMSCs (see the function `doCommit_dtx` of Listing 3). Therefore, if one RMSC decides to commit, then we are sure all other RMSCs also decide to commit. Since the only options for the verdict are to commit or to abort, the previous argument proves that 2PC4BC achieves *Agreement.* (ii) 2PC4BC guarantees *Termination.* To see why, remember that we assumed blockchain networks never fail, and that TM eventually recovers from potential failures and "remembers" the contents of its stable storage. Furthermore, we assumed that the requests sent from TM to the involved blockchain networks will eventually be delivered and accepted, i.e., permanently become part of the corresponding blockchain histories, and vice versa, all communication sent from RMSCs to TM will be delivered and processed, since they are embedded into smart contract events which are stored permanently in the blockchain data structure. Thus, even if TM is not operational when a message is sent to it from an RMSC, it will be able to receive it and process it when it eventually becomes operational again. Besides, looking at the state machine diagram of TM (see Figure 5a), we see that after the TCCSCI is ended, TM *will eventually end up in either the "Comitted" or the "Aborted" states,* and in both cases it will send a verdict to all involved RMSCs. It cannot remain indefinitely in the "Awaiting Votes" state, since when an RMSC receives a `prepare*` request, it will cast a vote that will eventually reach TM (see the function `prepare*` in Listing 2). In addition, by looking at the state machine diagram of an RMSC (see Figure 5b), we see that all transitions triggered by receiving a verdict from TM lead to either the "Committed" or the "Aborted" states. Hence, considering our assumptions, 2PC4BC will eventually terminate. (iii) 2PC4BC guarantees *Abort-Validity.* It is easy to see that, even if a single RMSC votes "no", TM will decide on "Abort" as the verdict (see the `bc.commit("VOTE_EVENT", e)` handler of Listing 3). Therefore, *Abort-Validity* is ensured. (iv) 2PC4BC guarantees *Commit-Validity.* It is clear that, if TM receives "yes" votes from all participants, it will decide on "Commit" as the verdict (see the `bc.commit("VOTE_EVENT", e)` handler of Listing 3). Thus, *Commit-Validity* is ensured.

7.2 Security Implications

A common point of reference in discussing security is the CIA triad [27]. CIA stands for Confidentiality, Integrity, and Availability. First, the presented approach does not negatively affect confidentiality because (i) the logic of the control flow of any TCCSCI is only known to the client application and is not published to any involved blockchain network and

(ii) smart contract function invocation requests are submitted to the the corresponding blockchain networks directly by the client application without the involvement of TTPs. Of course, if the blockchain networks themselves are public, the data stored in them is not confidential, but this is an artifact of public blockchains and not of our approach. Second, the approach does not negatively affect integrity because all messages sent by the client application as part of a TCCSCI are cryptographically signed blockchain transactions. Additionally, the functions of the RMSC include cryptographic checks that ensure only the request messages submitted by the client application that owns the TCCSCI are processed. Third, the 2PC4BC protocol blocks if the TM component crashes during the "Awaiting Votes" state until the failure is fixed, because during this state, participants cannot unilaterally decide to abort, since this might violate the *Agreement* requirement. This negatively affects availability, since all locked variables are inaccessible to other TCCSCIs during blocking. This undesirable feature is inherent to 2PC [3, 12]. In other types of distributed systems, this can be alleviated by using a *cooperative termination protocol* or a different ACP, such as Three-Phase Commit (3PC) [3]. However, these options require direct communication between participants, which is not possible when the participants are blockchain networks without the introduction of TTPs that forward messages between them. A feasible way to tackle this problem is employing fault-tolerance techniques [22] for the TM component.

7.3 Approach Overhead

First, we discuss the *time complexity* of the approach. Let's define a *request round* to be a set of smart contract function invocation requests that are simultaneously submitted by the client application. We call the duration of one round a *round duration*, which is equal to the duration of its "slowest" request determined by the blockchain network that takes the longest to store a submitted request into its immutable history. We define *time complexity* to be the sum of all round durations involved in a crash-free TCCSCI in the worst case. During a TCCSCI, the client application submits β rounds of invocation requests targeted at user-defined smart contracts. Furthermore, in the 2PC4BC protocol, exactly two request rounds are sent by TM, one to the `prepare*` function of all involved RMSCs, and one either to the `commit*` or the `abort*` RMSC functions. Therefore, the total number of message rounds involved in the execution of a TCCSCI is: $\beta + 2$. A *trivial approach* that submits the invocation requests specified in the TCCSCI without guaranteeing correctness has β request rounds. Let δ denote the round duration assuming the "slowest" blockchain network is involved. Then, our approach incurs a *time overhead* of 2δ regardless of how many blockchain networks are involved.

Finally, we discuss the *cost overhead* of executing a TCCSCI using our approach. Certain blockchain systems, mostly permissionless, require that the execution of smart contract functions is associated with *execution fees* that correspond to how costly the execution is to the blockchain nodes, in addition to fees that are meant to incentivize nodes to process the invocation request quickly. We refer to the sum of all fees associated with a smart contract function invocation as the *invocation cost*. We try to estimate an upper bound of the cost overhead of our approach. Therefore, we focus on Ethereum Mainnet (*Ethereum* for short). Ethereum calculates the execution fees on an operation basis. Let us define $cost^{op}(o, ctxt)$ to be the cost of executing the basic smart contract operation o within the context $ctxt$. We define the *context* of a basic operation or a function to be all the data items that are accessible to them at the time of execution, which includes, e.g., the passed arguments and the accessible data items stored in the persistence layer. Similarly, we define $cost^{fu}(f, ctxt)$ to be the cost of invoking and executing the smart contract function f within the context $ctxt$ if the invocation originates from another smart contract function in Ethereum, while we define $cost^{req}(f, ctxt)$ to be the cost of invoking and executing the smart contract function f within the context $ctxt$ if the invocation originates from a client application, which includes the additional fees paid for submitting a request to Ethereum.

When using the approach, user-defined smart contracts have to invoke the `set*` and `get*` functions of the corresponding RMSC instead of calling the basic operations `set` and `get`. In fact, this is the only requirement imposed by the TCCSCI approach on user-defined smart contracts. Hence, we define $\kappa^{set*}(ctxt) := cost^{fu}(set*, ctxt) - cost^{op}(set, ctxt)$ to be the overhead of using our approach for setting a value in a given context, which is the difference between the cost of an invocation of the `set*` function and the cost of a call to the `set` basic operation using the same context. We define $\kappa^{get*}(ctxt)$ in a similar fashion. By looking at Listing 1, we note that $\forall ctxt : \kappa^{set*}(ctxt) > \kappa^{get*}(ctxt)$. Since we aim to find an upper bound for the cost overhead and to simplify our calculations, we assume that all data access operations are `set*` operations of different variables. Furthermore, by looking at the code of the function `set*`, we notice that it executes more basic operations (and incur more cost) if it is the first invocation in a given local transaction $ltx_{bc,dtx}$. Hence, let's define κ_1^{set*} to be the value of $\kappa^{set*}(ctxt)$ when the context indicates that this is the first data access in $ltx_{bc,dtx}$, and let's define $\kappa_{>1}^{set*}$ to be the value of $\kappa^{set*}(ctxt)$ when the context indicates that this is not the first invocation in $ltx_{bc,dtx}$. Let n denote the number of blockchain networks involved in the TCCSCI dtx. Consequently, we have a total of n occurrences of κ_1^{set*}. Furthermore, let us denote α to be the total number of times data items were accessed across all the smart contract functions invoked in dtx. Hence, an upper bound for the cost overhead of invoking the smart contract functions of a TCCSCI is: $(\alpha - n)\kappa_{>1}^{set*} + n\kappa_1^{set*}$.

We now estimate an upper bound for the cost overhead of the 2PC4BC protocol. The protocol includes sending n `prepare*` and n `commit*` or `abort*` requests. By looking at Listing 2, we notice that $\forall ctxt : cost^{req}(abort*, ctxt) > cost^{req}(commit*, ctxt)$. Therefore, we assume the TCCSCI ends with an “Abort” verdict. Furthermore, notice that the cost of an `abort*` linearly increases with the number of variables that were `set*` during the local transaction $ltx_{bc,dtx}$, since it needs to unlock each of them and to restore their previous values. We define $\kappa_k^{abort*} := cost^{req}(abort*, ctxt)$ to be the cost for invoking the function `abort*` by the TM component when $ctxt$ indicates that k different variables were accessed during $ltx_{bc,dtx}$. Hence, $\kappa_1^{abort*} - \kappa_0^{abort*}$ represents the cost associated with every accessed variable when the TCCSCI dtx is being aborted. Moreover, we define $\kappa_{yes}^{prepare*} := cost^{req}(prepare*, ctxt)$ to be the cost for invoking the function `prepare*` by TM when $ctxt$ indicates a “yes” vote (because this corresponds to more cost). Hence, the upper bound for the cost overhead of the 2PC4BC protocol is: $\alpha(\kappa_1^{abort*} - \kappa_0^{abort*}) + n(\kappa_{yes}^{prepare*} + \kappa_0^{abort*})$. This gives the following estimate for the upper bound of the cost overhead of the whole approach:

$$\alpha(\kappa_{>1}^{set*} + \kappa_1^{abort*} - \kappa_0^{abort*}) + n(\kappa_1^{set*} + \kappa_{yes}^{prepare*} + \kappa_0^{abort*} - \kappa_{>1}^{set*}) \quad (3)$$

We now empirically estimate the values, $\kappa_{>1}^{set*}$, κ_1^{set*}, κ_0^{abort*}, κ_1^{abort*}, and $\kappa_{yes}^{prepare*}$. To this end, we use $RMSC_{Eth}$, which was introduced in Section 6 for the Ethereum blockchain, and we invoke the corresponding functions either directly from a client application (for `prepare*`, `commit*`, and `abort*`) or from another smart contract (for `set*` and `get*`).

The environment used to conduct these experiments has the following setup: To estimate the gas consumed, we use Remix IDE[7] (version 0.33.0), which has an integrated Ethereum simulator. More specifically, we set the compiler version to 0.8.18 and the EVM version to “Constantinople”. Furthermore, we run the executions on a machine with 40GB of RAM and a quad-core Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz of base frequency.

Gas is the unit that measures the amount of computational effort needed to execute operations on the Ethereum network [34]. Table 1 shows a summary of the experiments we conducted and their outcomes. By applying the resulting values to Formula 3, we get the following estimation of the upper bound for the cost overhead of using the TCCSCI approach measured in the amount of consumed gas: $32\,922\alpha + 159\,219n$.

[7]https://remix.ethereum.org/

Table 1. The gas consumption of different operations in different steps of the TCCSCI approach.

Context	RMSC Execution Cost (Gas)			2PC4BC Execution Cost (Gas)		
	$cost^{op}(set, ctxt)$	$cost^{fu}(set*, ctxt)$	$\kappa^{set*}(ctxt)$	$\kappa_{yes}^{prepare*}$	κ_0^{abort*}	κ_1^{abort*}
First Call	22 710	157 598	134 888	10 880	15 603	46 373
Next Calls	22 710	24 862	2 152	-	-	-

At the time of writing, the average gas price, which is the price the client application is willing to pay per unit of gas, is 33 gwei (or 33×10^{-9} ethers), and the price of 1 ether in USD is $1 624.08. This means that, in the worst case, for each data access within a given TCCSCI, the approach incurs an overhead of ~$1.76, and for each blockchain network that participates in the TCCSCI, the approach incurs an overhead of ~$8.53.

8 DISCUSSION

In this section, we discuss the characteristics, trade-offs, and limitations of the TCCSCI approach. When we introduced the TCCSCI approach, we determined a set of requirements that would ease its adoption (see Section 5). We now discuss how well the approach fulfills these requirements. First, the approach does not depend on a TTP. The only new components introduced by the approach are the RMSCs and TM. RMSCs are regular smart contracts hosted on the participating blockchain networks, and TM is a component of the client application. Therefore, the TCCSCI approach does not introduce any TTP. Second, the approach allows arbitrary composition of user-defined smart contracts during TCCSCIs as long as they use the RMSCs of the blockchain networks they are hosted on to access the required data items. Specifically, the business logic of the client application is free to invoke smart contract functions of different blockchain networks and pass data between them within the scope of a TCCSCI as long as it uses a TM component that follows the 2PC4BC protocol to ensure global atomicity. Finally, the approach allows heterogeneity since it supports any smart contract-enabled blockchain technology that fits into the system model introduced in Section 4.1 regardless of its type (permissioned or permissionless) and the used consensus mechanism, and without changing its protocol.

Despite the approach's desirable correctness and adoptability characteristics, it still suffers from certain drawbacks. First, the approach might suffer from a relatively high abortion rate in certain situations. To see why, notice that RMSC's implementation aborts the TCCSCI if it cannot immediately obtain a lock on a data item it needs to access. In ordinary resource managers, a *scheduling algorithm* is employed that might decide to temporarily block (or halt) a transaction that cannot obtain a lock in the hope that it can later obtain it when conflicting transactions commit or abort [3]. However, the execution of a smart contract function cannot be halted and resumed later, since it has to take place during a single block. So if we decide to halt (and not abort) a TCCSCI when it cannot obtain a lock, the client application will have to re-submit the request to execute the failed smart contract invocation repeatedly until it is able to obtain the lock, but how often should it try that and in which intervals? Answering these questions effectively shifts the scheduling responsibility from the resource manager to the client application, thus over-complicating it. Note that our scheduling strategy causes relatively many abortions only if we have a high contention rate between concurrent TCCSCIs over the same data items. Furthermore, a positive feature that we get from our scheduling strategy is that *we avoid deadlocks altogether* [4] because it forces TCCSCIs that access conflicting data items to be executed serially. Second, our approach requires rewriting existing user-defined smart contracts to make them use the RMSCs of their blockchain networks whenever they access the data items they need. The only possible alternative to this (if we still want to guarantee correctness) is to introduce changes to the built-in resource management layer of all used blockchain networks so that they support distributed transactions. However, we deem this option to be impractical since it requires changing existing blockchain protocols, which would greatly impact the approach's adoptability.

Finally, the TCCSCI approach has good scalability in relation to *n*, the number of blockchain networks involved in the same TCCSCI, since (i) the time overhead of the approach is independent of *n* and (ii) the cost overhead increases linearly with *n* (see Section 7). Furthermore, the TM component has to support communicating with the corresponding blockchain types. Our prototype makes this feasible using pluggable blockchain adapters. As mentioned in Section 6, the prototype currently includes five pluggable adapters for different blockchain types including Hyperledger Fabric and any blockchain that exposes the Ethereum JSON-RPC API. Having adapters for widely used blockchain types and having the ability to easily add new adapters increase the practicality of the proposed prototype. Nonetheless, experiments outside the lab environment are still needed to evaluate the prototype's real-world feasibility.

9 RELATED WORK

Several existing approaches aim to support CCSCIs. The first category of approaches uses an ACP to achieve global atomicity. For example, Wang et al. [33] present a CCSCI approach in which multiple Hyperledger Fabric blockchain networks with a modified implementation collaborate to achieve an atomic CCSCI. To this end, an off-chain *Scheduler* component receives smart contract invocation requests from a client application and forwards them to the corresponding networks. It also coordinates the execution of an ACP directly with the nodes of the underlying networks, which are modified for this purpose. However, during the CCSCI execution, no locks are obtained, and thus unintentional interactions between parallel CCSCIs are possible. Therefore, structures that record data dependencies between CCSCIs are maintained. If the ACP decides to abort a CCSCI, all dependent CCSCIs are also aborted and if they are already committed, they are rolled back, which violates their durability and, consequently, global serializability [3, chapter 2]. Another example is the approach proposed by Xiao et al. [35] which introduces a custom blockchain network that acts as a 2PC transaction manager, and off-chain relayer components that participate in the ACP on behalf of the actual blockchain networks. These relayers are TTPs that the blockchains networks need to trust in order to participate in CCSCIs. Furthermore, the approach does not involve locking mechanisms, allowing parallel CCSCIs to erroneously interact. The approach proposed by Robinson and Ramesh [25] stands out from the former approaches by employing a locking-based concurrency control mechanism that guarantees global serializability. In this approach, similar to our approach, a dedicated smart contract in each blockchain network performs locking and recovery handling, and the client application handles 2PC. However, a major difference is that smart contracts across blockchains are able to synchronously invoke each other. To this end, the client application has to simulate the whole smart contract *invocation tree* of the CCSCI before it starts, and has to include the outcomes in a request message submitted to a special Cross-Chain Communication Smart Contract (CCCSC) on one of the involved blockchain networks. To perform the simulation, the client application needs to be able to access the current state of all involved blockchains, and to locally simulate the execution of the involved smart contracts. During execution, the CCCSC and the client application run a special protocol, which guarantees that whenever a smart contract wants to invoke a remote smart contract, the results of the invocation are already stored on a local smart contract, which can be simply queried instead of performing the actual invocation. This is made possible with the knowledge about the invocation tree and the results of the simulation. If during runtime, the invocation results deviate from the simulation, or if any lock is not obtainable, the whole CCSCI is aborted. Obviously, this approach requires a complex client application capable of performing the aforementioned simulations and to coordinate both commitment and execution. Moreover, aborts are more likely than our approach.

Table 2. Comparison of CCSCI approaches. GA: Global Atomicity, GS: Global Serializability, Pd: Permissioned, Ps: Permissionless.

Approach	Category	TTP Needed?	Arbitrary Composition?	Guarantees	Blockch. Types	Modifications Needed?	Simulation Needed?
Wang et al. [33]	2PC	Yes	Yes	GA	Pd	Yes	No
Xiao et al. [35]	2PC	Yes	No	GA	Ps & Pd	No	No
Robinson and Ramesh [25]	2PC	No	Yes	GA & GS	Ps & Pd	No	Yes
Nissl et al. [23]	RPC	Yes	No	None	Ps & Pd	No	No
LayerZero [38]	RPC	Yes	No	None	Ps & Pd	No	No
TokenBridge/AMB [24]	RPC	Yes	No	None	Ps & Pd	No	No
HyperService [18]	Workflow	Yes	Yes	None*	Ps & Pd	No	No
TCCSCI	2PC	No	Yes	GA & GS	Ps & Pd	No	No

*Only financial atomicity is guaranteed, which we do not consider as one of the correctness criteria.

Furthermore, a group of approaches do not try to coordinate the commitment of CCSCIs. For example, Nissl et al. [23] introduce an approach that implements a mechanism similar to Remote Procedure Calls (RPCs), in which a smart contract on a source blockchain invokes a smart contract on a target blockchain by invoking a local cross-chain smart contract passing the invocation details to it. This smart contract then emits an event with these details, which is picked up by a distributed network of TTPs that forward it to the target smart contract. The approach also introduces a mechanism that allows the target smart contract to verify that the invocation originated from the source blockchain with the help of additional TTPs. Many other approaches follow a similar mechanism, such as LayerZero [38] and TokenBridge/AMB [24]. Another example is HyperService [18], which allows defining CCSCIs using HSL, a smart contract composition language. A compiled HSL program gets deployed onto a middleware layer consisting of a set of TTP nodes and a dedicated permissioned blockchain network called NSB. Afterwards, the client application collaborates with the TTPs and the underlying blockchain networks according to a custom cryptographic execution protocol to execute the CCSCI. A special smart contract on NSB ensures that honest participants receive almost no financial loss even if the other participants deviate from the protocol . Hence, HyperService achieves *financial atomicity*, but fails to achieve global atomicity and serializability. Table 2 summarizes the properties of the aforementioned approaches including to which category they belong, whether a TTP is needed, whether arbitrary composition of smart contract function invocations is supported, which CCSCI correctness guarantees are ensured, which blockchain types are supported, whether modifications to the standard blockchain protocols are needed, and whether the client application has to locally simulate the CCSCI execution prior to starting it. The last row shows the TCCSCI approach in comparison.

The table shows that only the approach of Robinson and Ramesh [25] has properties close to ours. However, their approach results in a higher rate of aborts, and requires the client application to simulate the CCSCI and to participate in a sophisticated protocol during execution. Furthermore, their approach has higher cost overhead due to the higher protocol complexity and larger sizes of request messages submitted to the involved blockchain networks. For example, using their approach to read an integer from one blockchain network and write it to another blockchain network incurs a cost overhead of 1 013 784 units of gas according to the evaluation presented in [25], while executing the same scenario using our approach incurs a cost overhead of only $32\,922(2) + 159\,219(2) = 384\,282$ units of gas in the worst case. This corresponds to a cost saving of at least 62.1 % when our approach is used. Finally, their approach incurs a time overhead of $(1 + w)\delta$, where w represents the number of involved blockchain networks that execute state-changing smart contract functions as part of the CCSCI, whereas our approach has a constant time overhead of 2δ, which makes it equally or more efficient in terms of time overhead in all scenarios involving any state-changing operations.

10 CONCLUSION

In this work, we have introduced and formalized the problem of ensuring transactional behavior for Cross-Chain Smart Contract Invocations (CCSCIs), which are business transactions that invoke smart contracts distributed across multiple blockchains, and presented the Transactional CCSCI (TCCSCI) approach, which solves it. Moreover, we have proved the correctness of the approach and discussed a prototypical implementation thereof. Finally, we evaluated its overhead in terms of cost and time, discussed its security implications and trade-offs, and compared it to existing approaches.

Our approach answers the question of *how to guarantee global atomicity and global serializability in business transactions that involve the invocation of smart contract functions located in different, possibly heterogeneous blockchain networks* by introducing on-chain Resource Manager Smart Contracts (RMSCs) that implement locking over shared variables and guarantee the isolation of parallel CCSCIs, and 2PC4BC, a variant of 2PC that ensures the global atomicity of CCSCIs. The approach is technology-agnostic. Therefore, it supports heterogeneous blockchain networks. To the best of our knowledge, the TCCSCI approach is the first TTP-free approach that provides global serializability and atomicity, allows arbitrary composition of heterogeneous smart contract functions, and supports standard blockchain protocols while not requiring complex client-side simulations. Actually, our approach also solves the problem of Transactional Cross-Block Smart Contract Invocations [37], which aims to achieve atomicity and isolation for business transactions involving smart contract function invocations over multiple blocks on the same blockchain network.

A limitation of the approach is that the 2PC4BC protocol temporarily blocks in a specific failure situation thus affecting availability. Another limitation is the relatively high abortion rate in case of high contention over shared data. Finally, the approach incurs a constant time overhead regardless of the number of involved blockchain networks, which is a drawback only in rather small use cases. An open question is how the approach can be generalized to include non-blockchain resource managers such as database management systems and messaging middlewares, which we plan to tackle in future work. Furthermore, we plan to introduce a smart contract re-writing tool that allows developers to effortlessly transform regular smart contracts to be TCCSCI-compatible by integrating them with the local RMSCs.

ACKNOWLEDGMENTS

In memory of Prof. Dr. Florian Daniel†, who started this work together with us in its early days.

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development as well as the Christian Doppler Research Association for the Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things is gratefully acknowledged.

REFERENCES

[1] Elli Androulaki, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Artem Barger, Sharon Weed Cocco, Jason Yellick, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, and Gennady Laventman. 2018. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *EuroSys '18*. ACM Press, New York, USA, 1–15. https://doi.org/10.1145/3190508.3190538

[2] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. 2022. A Survey on Blockchain Interoperability: Past, Present, and Future Trends. *Comput. Surveys* 54, 8 (2022), 1–41. https://doi.org/10.1145/3471140

[3] Philip Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., USA.

[4] Philip Bernstein and Eric Newcomer. 2009. *Principles of Transaction Processing.* Morgan Kaufmann Publishers, Burlington, MA 01803, USA.

[5] Christian Cachin and Marko Vukolic. 2017. Blockchain consensus protocols in the wild (keynote talk). In *DISC 2017*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:16. https://doi.org/10.4230/LIPIcs.DISC.2017.1

[6] Ghareeb Falazi, Uwe Breitenbücher, Florian Daniel, Andrea Lamparelli, Frank Leymann, and Vladimir Yussupov. 2020. Smart Contract Invocation Protocol (SCIP): A Protocol for the Uniform Integration of Heterogeneous Blockchain Smart Contracts. In *CAiSE'20*, Vol. 12127. Springer International

Publishing, 134–149. https://doi.org/10.1007/978-3-030-49435-3_9

[7] Ghareeb Falazi, Vikas Khinchi, Uwe Breitenbücher, and Frank Leymann. 2019. Transactional properties of permissioned blockchains. *SICS Software-Intensive Cyber-Physical Systems* 35, 1 (2019), 49–61. https://doi.org/10.1007/s00450-019-00411-y

[8] Ghareeb Falazi, Andrea Lamparelli, Uwe Breitenbücher, Florian Daniel, and Frank Leymann. 2020. Unified Integration of Smart Contracts Through Service Orientation. *IEEE Software* 37, 5 (2020). https://doi.org/10.1109/MS.2020.2994040

[9] Enrique Fynn, Alysson Bessani, and Fernando Pedone. 2020. Smart Contracts on the Move. In *DSN'20*. IEEE, 233–244.

[10] Jim Gray. 1978. Issues And Results In The Design Of Operating Systems. In *Notes on Data Base Operating Systems*. Springer Berlin Heidelberg, Chapter 3, 393–481. https://doi.org/10.1007/3-540-08755-9_9

[11] Jim Gray. 1990. A comparison of the Byzantine Agreement problem and the Transaction Commit Problem. In *Fault-Tolerant Distributed Computing*, Barbara Simons and Alfred Spector (Eds.). Springer New York, New York, NY, 10–17.

[12] Rachid Guerraoui. 2002. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing* 15, 1 (2002), 17–25. https://doi.org/10.1007/s446-002-8027-4

[13] Thomas Hardjon, Alexander Lipton, and Alex Pentland. 2021. Interoperability of Distributed Systems. In *Building the New Economy: Data as Capital*. MIT Connection Science & Engineering, Chapter 12, 321–364.

[14] Maurice Herlihy. 2018. Atomic Cross-Chain Swaps. In *PODC'18*. ACM, 245–254. https://doi.org/10.1145/3212734.3212736

[15] Adrian Hope-Bailie and Stefan Thomas. 2016. Interledger. In *WWW '16 Companion*. ACM Press, New York, USA, 281–282.

[16] Hyperledger. 2022. *The Ordering Service - Hyperledger Fabric Main Documentation*. Linux Foundation. Retrieved March 28, 2023 from https://hyperledger-fabric.readthedocs.io/en/release-2.5/orderer/ordering_service.html

[17] Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Gustavo Alonso. 2010. Database Replication: A Tutorial. In *Replication: Theory and Practice*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter 9, 219–252. https://doi.org/10.1007/978-3-642-11294-2_12

[18] Zhuotao Liu, Yangxi Xiang, Jian Shi, Peng Gao, Haoyu Wang, Xusheng Xiao, Bihan Wen, and Yih-Chun Hu. 2019. HyperService: Interoperability and Programmability Across Heterogeneous Blockchains. In *CCS '19*. ACM, London United Kingdom, 549–566. https://doi.org/10.1145/3319535.3355503

[19] Matteo Montecchi, Kirk Plangger, and Michael Etter. 2019. It's real, trust me! Establishing supply chain provenance using blockchain. *Business Horizons* 62, 3 (2019), 283–293. https://doi.org/10.1016/j.bushor.2019.01.008

[20] Roman Mühlberger, Stefan Bachhofner, Eduardo Castelló Ferrer, Claudio Di Ciccio, Ingo Weber, Maximilian Wöhrer, and Uwe Zdun. 2020. Foundational Oracle Patterns: Connecting Blockchain to the Off-Chain World. In *BPM 2020: Blockchain and RPA Forum*. Springer International Publishing, Cham, 35–51. https://doi.org/10.1007/978-3-030-58779-6_3

[21] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system.* Technical Report. https://bitcoin.org/bitcoin.pdf

[22] Mehdi Nazari Cheraghlou, Ahmad Khadem-Zadeh, and Majid Haghparast. 2016. A survey of fault tolerance architecture in cloud computing. *Journal of Network and Computer Applications* 61 (2016), 81–92.

[23] Markus Nissl, Emanuel Sallinger, Stefan Schulte, and Michael Borkowski. 2021. Towards Cross-Blockchain Smart Contracts. In *DAPPS'21*. IEEE, United Kingdom, 85–94. https://doi.org/10.1109/DAPPS52256.2021.00015

[24] OmniBridge. 2022. *TokenBridge/Arbitrary Message Bridge*. Gnosis. Retrieved April 06, 2023 from https://docs.tokenbridge.net/amb-bridge/

[25] Peter Robinson and Raghavendra Ramesh. 2021. General Purpose Atomic Crosschain Transactions. In *BRAINS'21*. 61–68.

[26] Peter Robinson, Raghavendra Ramesh, and Sandra Johnson. 2022. Atomic Crosschain Transactions for Ethereum Private Sidechains. *Blockchain: Research and Applications* 3, 1 (2022), 100030. https://doi.org/10.1016/j.bcra.2021.100030

[27] Spyridon Samonas and David Coss. 2014. The CIA strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security* 10, 3 (2014).

[28] Péter Szilagyi. 2017. *Clique PoA protocol & Rinkeby PoA testnet.* Ethereum Foundation. https://github.com/ethereum/EIPs/issues/225

[29] Paolo Tasca and Claudio Tessone. 2019. A Taxonomy of Blockchain Technologies: Principles of Identification and Classification. *Ledger* 4 (2019), 1–39. https://doi.org/10.5195/ledger.2019.140

[30] Shreshth Tuli, Shikhar Tuli, Gurleen Wander, Praneet Wander, Sukhpal Singh Gill, Schahram Dustdar, Rizos Sakellariou, and Omer Rana. 2020. Next generation technologies for smart healthcare: challenges, vision, model, trends and future directions. *Internet Technology Letters* 3, 2 (2020), e145.

[31] Marko Vukolić. 2017. Rethinking Permissioned Blockchains. In *BCC '17*. ACM, New York, NY, USA, 3–7. https://doi.org/10.1145/3055518.3055526

[32] Qiang Wang, Rongrong Li, and Lina Zhan. 2021. Blockchain technology in the energy sector: From basic research to real world applications. *Computer Science Review* 39 (2021), 100362. https://doi.org/10.1016/j.cosrev.2021.100362

[33] Wenqi Wang, Zhiwei Zhang, Guoren Wang, and Ye Yuan. 2022. Efficient Cross-Chain Transaction Processing on Blockchains. *Applied Sciences* 12, 9 (2022), 4434. https://doi.org/10.3390/app12094434

[34] Gavin Wood. 2022. *Ethereum: a secure decentralised generalised transaction ledger - BERLIN VERSION.* Technical Report. Ethereum Foundation. https://ethereum.github.io/yellowpaper/paper.pdf

[35] Xingtang Xiao, Zhuo Yu, Ke Xie, Shaoyong Guo, Ao Xiong, and Yong Yan. 2020. A Multi-blockchain Architecture Supporting Cross-Blockchain Communication. In *Artificial Intelligence and Security*, Vol. 1253. Springer Singapore, 592–603. https://doi.org/10.1007/978-981-15-8086-4_56

[36] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. 2018. *Blockchain technology overview.* Technical Report. National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.IR.8202

[37] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2019. Transactional Smart Contracts in Blockchain Systems. arXiv:1909.06494

[38] Ryan Zarick, Bryan Pellegrino, and Caleb Banister. 2021. LayerZero: Trustless Omnichain Interoperability Protocol. arXiv:2110.13871