On Atomicity and Confidentiality Across Blockchains Under Failures

Yuechen Tao, Bo Li, *Fellow, IEEE*, and Baochun Li, *Fellow, IEEE*

***Abstract*—Distributed applications that utilize *heterogeneous* blockchain systems have the potential to be widely deployed. In such applications, users from different blockchains can transact with one another through *cross-chain transactions*. There are two essential features of particular relevance for those applications during cross-chain transactions: the *atomicity* in that either all or none of the blockchains involved confirm a cross-chain transaction, the *confidentiality* in that a blockchain involved in a cross-chain transaction is only accessible for designated users. Existing cross-chain proposals have largely relied on permissioned blockchains to ensure confidentiality. However, we found that failures could occur when reading or writing information during transaction confirmations across permissioned blockchains, namely read/write (r/w) failures, which can lead to the violation of atomicity. In this paper, we propose a novel mechanism, *Unity*, to ensure both atomicity and confidentiality of cross-chain transactions under r/w failures by leveraging permissioned blockchains. When failures occur in reading or writing data, *Unity* classifies the data into two categories based on its status - whether data is the latest version or not, and presents different solutions for atomicity. Specifically, when data is not the latest, we design a four-phase-commit protocol (4pc), in which consensus on confirming or aborting a cross-chain transaction can be achieved. If data is the latest when r/w failures occur, we propose a smart contract based solution (SSC). We examine the effectiveness of *Unity* theoretically and through experiments. With a failure probability of 0.7, *Unity* achieves 98% more atomic cross-chain transactions when compared with the state-of-the-art cross-chain platform, Hyperservice.**

***Index Terms*—Blockchain, protocols, heterogeneous databases, distributed transactions, fault-tolerance, peer-to-peer computing.**

I. Introduction

With its security, provenance, and reliability, blockchain technology has been widely used in real-world systems, especially in Internet-of-things (IoT) scenarios, e.g., electric vehicle charging (EV-charging) and supply chain management [1], [2], [3], [4]. There exist heterogeneous blockchain systems developed by administrations in different scenarios [5], [6]. In that case, a wide range of distributed applications across those heterogeneous blockchain systems may be deployed, where secure collaborations among those administrations can be provided. In those applications, users from different blockchains can transact with each other through *cross-chain* transactions. For example, in the electric vehicle charging scenario, two gas corporations develop two different blockchains, namely chains 1 and 2. Supposing that gas stations A and B belong to those two different gas corporations, and participate in chains 1 and 2, respectively. If gas station A wants to request fuels from gas station B, A spends tokens on chain 1 and B receives tokens on chain 2 through a cross-chain transaction, enabling the collaboration between chains 1 and 2.

Two characteristics of *cross-chain* transactions are notably relevant and crucial for those distributed applications. The first one is transaction *confidentiality*, which has been investigated in recent blockchain research [7], [8], [9]. Specifically, a blockchain system is only accessible to designated users. Supposing that there is a blockchain developed by a commercial foundation. To prevent confidential data from theft by competitors, such a blockchain shall only be accessible for users that are verified by this commercial foundation. In our example, chain 2 developed by gas corporation 2 shall not be accessible for gas station A belonging to gas corporation 1. Second, the transaction *atomicity*, as one of the well-known ACID properties [10], has been universally believed to be essential for resilient distributed databases (RDB) [11], [12], [13]. In blockchain scenarios, atomicity means that either all or none of the involved blockchains confirm a cross-chain transaction, denoted as T, which is critically important for users' property [14]. In our example, atomicity is lost if only chain 1 or 2 confirms T. On one hand, if only chain 1 confirms T, gas station A spends tokens but gas station B cannot receive those tokens. On another hand, if only chain 2 confirms T, gas station B receives tokens, while gas station A does not spend tokens.

Some of existing cross-chain confirmation platforms assume that each blockchain involved in a cross-chain transaction T is permissioned [15], [16], [17], such that *they can ensure confidentiality by design*. Specifically, a blockchain is considered permissioned, if only authenticated users can participate in such a blockchain, i.e., only those authenticated users can open accounts on such a blockchain. In our example, if chains 1 and 2 are permissioned, gas station A cannot participate in chain 2 and gas station B is not able to participate in chain 1 without being authenticated. In that case, the confidential data of gas corporation 2 will not be exposed to gas corporation 1, and vice versa. However, as gas station A only has accounts on chain 1, its tokens can only be traded on chain 1. Therefore, an

Manuscript received 12 March 2022; revised 14 February 2023; accepted 23 February 2023. Date of publication 13 March 2023; date of current version 11 January 2024. Recommended for acceptance by Y. Tong. *(Corresponding author: Bo Li.)*

Yuechen Tao and Bo Li are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Kowloon, Hong Kong (e-mail: ytaoaf@cse.ust.hk; bli@cse.ust.hk).

Baochun Li is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: bli@ece.toronto.edu).

Digital Object Identifier 10.1109/TKDE.2023.3255842

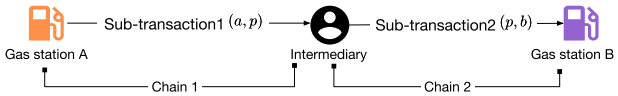


Fig. 1. T across two chains is confidentially accomplished through an intermediary.

important question is, who can send tokens to gas station B on chain 2, and how can those tokens be sent from chain 1 to chain 2? Typically, for a cross-chain transaction T, it is assumed that there exist *intermediaries* [15], [16], [17], serving as the bridge among users from different blockchains. Each intermediary is authenticated to participate in multiple blockchains involved in T. As illustrated in Fig. 1, an intermediary participates in both chains 1 and 2. Gas station A transfers tokens to that intermediary on chain 1, and that intermediary transfers tokens to gas station B on chain 2.

However, existing literature on cross-chain confirmations with confidentiality assurance can penalize atomicity when certain failures take place. Typically, specific information related to the validity of cross-chain transaction T must be read and written across involved blockchains when confirming T. If failures during such a reading or writing process occur, namely read/write (r/w) failures, existing proposals can suffer from atomicity loss. In our example, information related to the validity of T, e.g., the account balance of gas station A, is vital when validating T. It shall be read from chain 1, and be written on chain 2, such that validators on chain 2 know that gas station A has sufficient tokens for T. When either such a reading or writing operation fails, validators on chain 2 will consider T invalid, while validators on chain 1 know that T is valid. As a result, only chain 1 confirms T, leading to the atomicity loss.

It is, however, non-trivial to ensure atomicity under r/w failures across permissioned blockchain systems. Specifically, decentralized blockchains are essentially distinct from RDB, making existing literatures on atomicity assurance [11], [12], [13] not applicable. As discussed in [18], handling failures in reading/writing some data depends on the status of such data in terms of whether it is the latest. In RDB, it is widely assumed that such data is older than the one to be written or read, since read/write failures always occur due to the disaster on a reading/writing server [11], [12], [18]. In contrast, a blockchain system is decentralized without any centralized servers, indicating that such data can be the latest version when read/write failure occurs. As a result, the assumption related to the data status in RDB does not apply in blockchain systems; thus novel mechanisms in handling r/w failures are needed. There are two kinds of data status under r/w failures in blockchain systems. First, the last transaction for modifying data is confirmed after new r/w requests for the latest data arrive. These new r/w operations must fail. In other words, data is updated to the latest version after those r/w failures occur. Second, that last transaction is confirmed, but those new r/w requests for accessing the latest data get lost.

This paper presents *Unity*, a novel cross-chain confirmation protocol that supports transactions atomicity across permissioned and confidential blockchains, consisting of a four-phase-commit protocol (4pc) and several smart contracts (SSC). Our key observation is that, a consensus among blockchains on whether to confirm or abort a cross-chain transaction is a sufficient condition for maintaining atomicity. For the first kind of data status, where data being read/written is not the latest, 4PC coordinates blockchains to ensure the achievement of a consensus. Each phase in 4PC consists of different operations. If data in the older version is read/written across blockchains, miners can move forward to the next phase and conduct corresponding operations. For the second kind of data status, where r/w requests for the latest data are missing, i.e., data cannot be read/written across blockchains, we consider solutions from the perspective of enforcing all r/w operations once the latest data is available. Our innovation design, SSC, composed of several smart contracts, models those operations as transaction logics and records the availability of such data as on-chain conditions. Following the property of smart contracts, everyone can check on-chain conditions to enforce those operations. The difficulty lies in designing transaction logics and conditions, where everyone shall be able to conduct the provenance of such latest data that will be read or written, including but not limited to IoT devices or intermediaries.

To summarize, highlights of *Unity* are three-folds. First, in *Unity*, 4PC achieves a consensus on confirming or aborting a cross-chain transaction among permissioned blockchains under r/w failures with the first kind of data status. Following such a consensus, atomicity could be ensured. Second, SSC in *Unity* models 4PC as several transaction logics and conditions, which can defend against r/w failures with the second data status. Third, we have evaluated *Unity* in terms of atomicity assurance property, the probability of successful cross-chain confirmations, and cross-chain confirmation delay in theory and experiments. To be more specific, we model transaction status and r/w failures via a first-order predicate logic formalism, ACTA [19], based on which we prove the effectiveness of *Unity* on ensuring atomicity. If failures occur with a probability as high as 0.7, even with only 4pc, there are 98% more atomic cross-chain transactions compared to the state-of-the-art cross-chain confirmation platform Hyperservice [16].

The rest of this paper is organized as follows. In Section II, we introduce some preliminaries in confidentiality requirements, transaction confirmations across permissioned blockchains and r/w failures to better motivate our problem. In Section III, we give an architectural description of 4PC and SSC. In Section IV, we present transaction logics and conditions phase-by-phase. In Section V, we theoretically evaluate *Unity* in terms of its effectiveness and the probability of successful cross-chain confirmations. In Section VI, we examine *Unity* through real implementations.

II. Preliminaries

The blockchain technology provides security, provenance, and reliability to distributed systems, where users can safely transact with one another without any centralized authorities. In EV-Charging scenarios, supposing that a commercial corporation develops a blockchain, where designated gas

stations belonging to this commercial corporation can participate. Smart vehicles can request fuel from those gas stations securely by sending tokens to those gas stations on this blockchain. Motivated by such benefits, there have been diverse but isolated blockchain systems designed for different commercial corporations, e.g., TenneT and Bosch [5], [6]. When these corporations collaborate, their blockchains need to be connected. Let us consider the following cross-chain fuel recharging application, where gas stations belonging to different companies, i.e., participating in different blockchains, can buy and sell fuel among one another safely. Such buying and selling operations incur cross-chain transactions. For example, a gas station A on chain 1 running out of fuel buys fuel from gas station B on chain 2. A cross-chain transaction, denoted as T, across chains 1 and 2 appears.

The last section has briefly discussed the importance of ensuring transaction *confidentiality* and *atomicity* across different blockchains. In this section, we give more thorough explanations on existing cross-chain transaction confirmations with those two characteristics and how r/w failures affect such confirmations.

Confidentiality Requirement: Supposing that there is a distributed system developed by a commercial corporation. To prevent the theft of confidential data by malicious ones, a user is able to join this system only if its identity has been verified successfully by this commercial corporation. When this system incorporates the blockchain technology, i.e., there is a blockchain system developed by this commercial corporation, only designated users can participate in this blockchain. In other words, such a blockchain shall be *permissioned*, and strictly controls the identities of users that have access to it, to ensure its confidentiality [20]. In our example, supposing that competitive companies develop chains 1 and 2. Gas station A can only participate in chain 1, and gas station B can only participate in chain 2.

Actually, it is not easy to enable transaction confirmations across permissioned blockchains. For one thing, it is difficult to send a cross-chain transaction T to all involved users if a user cannot participate in all involved blockchains. Supposing that gas station A in our example is not allowed to open an account on chain 2. In that case, gas station A can only send coins to users on chain 1 instead of gas station B on chain 2. For another, a validator can only check the status of users belonging to the same blockchain. Without the same knowledge on the status of users involved in T, it is difficult for miners from all involved blockchains to make the same decision on confirming or aborting T. Thus, the coordination among blockchains, with which blockchains can jointly validate cross-chain transactions, are necessary for the sake of atomicity.

Existing Cross-Chain Confirmations With Confidentiality: Existing cross-chain confirmation protocols incorporate *intermediaries* (e.g., VESes in Hyperservice) [15], [16], [17], [21], [22], with which cross-chain transactions across permissioned blockchains can be confirmed. Specifically, for cross-chain transaction T, it is assumed that there is an intermediary that is authorized for participating in all involved blockchains. In our example, as shown in Fig. 1, an intermediary, denoted as p, participates in both chains 1 and 2. Accomplishing T is equivalent to confirming a series of sub-transactions: (a, p) and (p, b). (a, p) transfers coins from gas station A to p on chain 1, and (p, b) transfers coins from p to gas station B on chain 2.

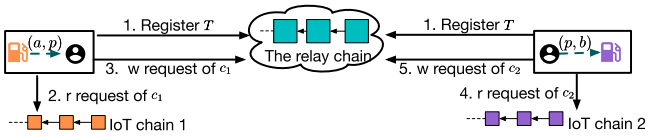


Fig. 2. r/w request: reading or writing request. Existing cross-chain validations via intermediaries.

There is a blockchain (e.g., a relay chain, a witness chain), records the status of (a, p), (p, b) and ensures the honesty of intermediaries, as illustrated in Fig. 2. Unlike traditional database systems, validators in a blockchain *will not* notify others if they consider (a, p) invalid. To prevent the endless waiting if (a, p) is invalid, a predefined *maximum delay*, denoted as M is necessary: if (a, p) has not been confirmed within M, others consider it invalid. In contrast, if (a, p) is confirmed within M, gas station A and the intermediary deliver such confirmation information, noted as c_1, to miners on the relay chain. Upon c_1 is successfully written on the relay chain, this intermediary broadcasts (p, b) on chain 2 to accomplish T.

R/w Requests Across Blockchains: In Fig. 2, the request of reading the latest data, i.e., c_1, is delivered to chain 1 by gas station A and intermediary no later than the expiration of M. Upon successfully reading, they deliver the request of writing the latest data to the relay chain.

Data Status When r/w Failures Occur: We specify two kinds of data status when r/w failures occur using our EV-charging example. First, data is not the latest version when r/w requests arrive. As exposed by [23], blockchains can confirm transactions extremely slowly. In other words, (a, p) is confirmed after M expires. The request to read the latest data, i.e., c_1, must arrive within M before c_1 is available, however. Failures in reading and writing c_1 across chain 1 and the relay chain occur. Second, data is the latest version, but r/w requests are missing. IoT devices and intermediaries are responsible for delivering r/w requests across blockchains. Supposing that (a, p) is confirmed within M. If gas stations and the intermediary fail in delivering r/w requests, c_1 also cannot be written on the relay chain.

Atomicity Loss Under r/w Failures: Under r/w failures, cross-chain transactions in the above validation process suffer from the loss of atomicity regardless of data status. If failures in reading or writing the latest data c_1 across chain 1 and the relay chain occur, that intermediary will not broadcast (p, b) on chain 2. However, in both data status categories, (a, p) will be confirmed eventually, i.e., atomicity is lost.

III. The *Unity* Architecture

Our key observation as follows supplies us with inspiration. Supposing there is a consensus on whether to confirm or abort a cross-chain transaction among blockchains. Then each underlying blockchain can refer to this consensus, based on which it confirms or aborts its sub-transactions. In this way,

either all or none of sub-transactions are confirmed, and atomicity can be ensured. Thus, the objective turns to *achieving a consensus among blockchains under r/w failures without sacrificing confidentiality.*

Our solution, *Unity*, consists of two components, 4PC and SSC. We first consider the cross-chain model built upon intermediaries as discussed in Section II so that the confidentiality issue can be addressed. Atop this model, 4PC, a four-phase-commit-process, can coordinate blockchains to achieve the consensus under r/w failures with the first kind of data status. Besides, specific data must be read or written across blockchains, with which all operations in 4pc can be conducted. To handle the second kind of data status, SSC enforces all operations to take place once specific data is available by modeling 4PC as a series of functions in several smart contracts.

In what follows, we first define cross-chain transactions and their sub-transactions in our cross-chain model built upon intermediaries and make important assumptions to better support *Unity* in Section III-A. We basically describe the four phases of 4PC in Section III-B. Specifically, operations in each phase and data to be read or written across blockchains are exposed in Section III-C.

A. Cross-Chain Model and Assumptions

Definition 1. Cross-chain transactions: A cross-chain transaction $T = \{S, I, R, E\}$ is validated through intermediaries. S is the set of senders, R is the set of receivers, I is the set of intermediaries, and E is the set of sub-transactions.

Definition 2. Sub-transactions: Sub-transactions of T are modeled as follows. Supposing that there is a sender $s \in S$ that transacts with a set of receivers $R_s \subseteq R$ across different underlying blockchains. For each $r \in R_s$, there will be an intermediary $p \in I$ that is correlated with s and r through sub-transactions $\{(s, p), (p, r)\} \subset E$. If all sub-transactions in E are valid, T is valid.

In our example, the cross-chain transaction between the two gas stations A and B is validated with the assistance of an intermediary p. There are two sub-transactions on that two underlying blockchains. The first is $\{(A, p)\}$ on chain 1, and the second is $\{(p, B)\}$ on chain 2. Thus, $\{S = \{A\}, I = \{p\}, R = \{B\}, E = \{(A, p), (p, B)\}\}$.

Unlike existing works, we assume that an intermediary may not directly transact with users through sub-transactions. In our example, miners on chain 1 validate other transactions instead of the sub-transaction (A, p), and p can receive coins sent from gas station A upon those transactions are confirmed. Following the security assumption in Hyperservice [16], underlying blockchains and the relay chain are secure. Specifically, invalid transactions will never be confirmed by the underlying blockchains and the relay chain. As we do not change the relay chain design, all attack models and security analyses in Hyperservice can be applied to *Unity*. Last but not least, we assume that all underlying blockchains support smart contracts.

B. 4pc: Consensus Achievement

We now introduce our four-phase-commit protocol, referred to as 4pc, aiming to achieve a consensus on confirming or aborting a cross-chain transaction T under r/w failures with the first kind of data status. The four phases are conducted as follows:

Phase 1. Registering T on the relay chain and voting on underlying blockchains: First, all users and the intermediary register T on the relay chain. This registration process records the set of sub-transactions E on the relay chain. Upon the successful registration of T, users and intermediaries request miners on their underlying blockchains to vote on the validity of each sub-transaction (u, v). Only if (u, v) is checked as valid, miners vote YES.

Definition 3. The validity of sub-transactions and voting results: For each sub-transaction (u, v), we let $V_{(u,v)}$ represent its validity, and let $R_{(u,v)}$ denote its voting result. $V_{(u,v)} = 1$ means that (u, v) is valid, and $V_{(u,v)} = 0$ means that (u, v) is invalid. Similarly, $R_{(u,v)} = 1$ represents the YES voting result.

Phase 2. Collecting votes and making decisions: Second, the data, i.e., voting results, is read from underlying blockchains and written on the relay chain within a predefined maximum delay, denoted as M. Based on those voting results, miners on the relay chain make a CONFIRMING or ABORTING decision. *If and only if* YES voting results are successfully read from all underlying blockchains and written on the relay chain within M, will the CONFIRMING decision be made.

Definition 4. Decision making: Given all voting results written on the relay chain within M, denoted as $\mathbf{R}(M)$, the decision of confirming or aborting T is:

$$g(\mathbf{R}(M)) = \begin{cases} 1, \text{ if } |\mathbf{R}(M)| = |\{R_{(u,v)}, \forall (u,v) \in E\}| \\ \text{ and } \forall R_{(u,v)} \in \mathbf{R}(M), R_{(u,v)} = 1, \\ 0, \text{ else.} \end{cases}$$

$D_T(M)$ represents for the decision made by the relay chain, i.e., $D_T(M) = g(\mathbf{R}(M))$. $D_T(M) = 1$ or 0 means that a CONFIRMING or ABORTING decision is made.

Noticing that requests for reading voting results must be sent no later than M expires. The reason is as follows. As discussed in Section II, underlying blockchains cannot explicitly notify users if (u, v) is invalid. Therefore, M is necessary to prevent the endless waiting if (u, v) is invalid. To be more specific, if the YES voting result does not exist until M expires, (u, v) is considered invalid.

R/w failures with the first data status category can occur in this phase. As shown in Section II, an underlying blockchain may confirm the validity of (u, v) after M expires. In this way, the YES voting result will not be updated when r/w requests arrive and cannot be successfully read or written across blockchains.

Phase 3. Informing underlying blockchains of this decision: Third, the data, i.e., the decision $D_T(M)$, is read from the relay chain and written on each underlying blockchain. Noticing that r/w failures with the first kind of data status will not take place in this phase. Unlike underlying blockchains where only valid transactions are recorded, the relay chain explicitly makes CONFIRMING or ABORTING decision, i.e., either $D_T(M) = 1$ or $D_T(M) = 0$ will be made eventually. Thus, there is no time limit when sending r/w requests for $D_T(M)$.

Phase 4. Completing or aborting sub-transactions: Forth, all sub-transactions of T are either confirmed or aborted based on the same CONFIRMING or ABORTING decision. Essentially, such

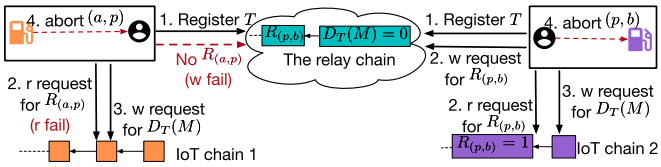


Fig. 3. Four-phase-commit protocol coordinates underlying blockchains and achieves an *aborting* decision.

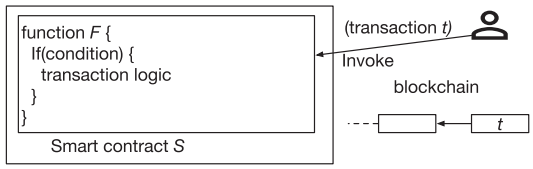


Fig. 4. Transaction logics recorded by a smart contract function is enforced by invoking functions.

a decision is the consensus result among all involved underlying blockchains.

Noticing that, during the voting process on an underlying blockchain in the 1st phase, miners validate some specific transactions that can represent the validity of the corresponding (u, v). The reason is as follows. In blockchain systems, miners directly confirm each transaction that users broadcast if it is valid. Therefore, if a sub-transaction (u, v) is directly sent to miners for validation, it will be confirmed once the YES voting result is made in the 1st phase. Unfortunately, if an ABORTING decision is made, (u, v) shall be aborted in the 4th phase, which is, however, already confirmed in the 1st phase. To prevent this, those specific transactions to be validated in the 1st phase need more careful considerations, based on which we take special care of the voting operation in the 1st phase and the sub-transaction completion operation in the 4th phase, as discussed in Section IV-A.

Fig. 3 illustrates how 4pc ensures atomicity under r/w failures with the first kind of data status in our example. Chains 1 and 2 vote on T_1 and T_2 in the 1st phase, respectively. The data, i.e., these voting results, shall be read from these two chains and written on the relay chain in the 2nd phase. Under r/w failures with the first kind of data status, where chains 1 and 2 obtain such data after the arrival of all r/w requests, the relay chain cannot collect enough YES voting results in the 2nd phase. As a result, an ABORTING decision is made on the relay chain. Following this consensus result, neither T_1 and T_2 can be confirmed in the next two phases.

C. SSC in Phase Changes

There are different operations, e.g., voting and making decisions, in 4 phases. Specific data must be read or written so that miners can move to new phases and conduct their operations. For example, r/w operations on voting results take place before making decisions. Fig. 3 has shown that 4pc handles the first data status category. Next, we cope with the second kind of status, where data is the latest with missing r/w requests, by enforcing all operations once such data is available. The difficulty lies in the provenance and availability check of such data, which is discussed thoroughly in Section IV.

Smart contracts provide hints for this issue. As shown in Fig. 4, a smart contract, written by specific languages such as Solidity and Bitcoin Script, defines *functions*. A function records self-executed transaction logic under certain conditions. For example, a function says that user A will transfer 2 coins to user B if A has more than 5 coins. If this function is invoked successfully, miners will check whether A has more than 5 coins. If yes, they will revise the account balance of these two users on the blockchain accordingly. A user can send an on-chain transaction to invoke this function. In other words, by invoking a function in a smart contract, its transaction logics can be *enforced* to take place once corresponding conditions are met.

By utilizing such a property of smart contracts, operations can be enforced upon data to be read or written is available. We design several smart contracts, denoted as SSC, to help such enforcement. Functions in SSC model operations as transaction logic. Whether such data is ready can be known by checking conditions recorded by functions. When functions are invoked, miners execute the corresponding transaction logics after checking conditions. SSC consists of $\mathrm{SSC_w}$ and $\mathrm{SSC_u}$. $\mathrm{SSC_w}$ is launched on the relay chain, and $\mathrm{SSC_u}$ is on all underlying blockchains. In what follows, we list the set of functions in SSC and the data to be read or written during phase changes.

In the 1st phase, users and intermediaries, I, S, R register $T = \{S, I, R, E\}$ on the relay chain. The function Registration() in SSC_w enforces the registering. To invoke Registration(), an on-chain transaction T_f is broadcast. Then users and intermediaries send voting requests to their underlying blockchains. Function Constructor() in SSC_u enforces the voting operation. To invoke Constructor(), a transaction denoted as T_l, is broadcast.

In the 2nd phase, by checking conditions in function $\mathrm{Collect}_w()$ of $\mathrm{SSC_w}$, miners know whether the data, i.e., voting results, is ready. $\mathrm{Collect}_w()$ writes such data on the relay chain, whose corresponding on-chain transaction is T_{ws}. After that, two functions ValidateYes() and ValidateNo() in SSC_w enforce the decision making operation based on $\mathbf{R}(M)$ and obtain the final decision $D_T(M)$. Two transactions T_c and T_r are broadcast to invoke ValidateYes() and ValidateNo().

In the 3 rd phase, conditions recorded by function $\mathrm{Collect}_u()$ of SSC_u, expose whether the data, i.e., CONFIRMING or ABORTING decision, to be read from the relay chain and written on underlying blockchains is available. If yes, miners on each underlying blockchain acknowledge such a decision. The corresponding on-chain transaction is T_{us}.

In the 4th phase, a sub-transaction is either confirmed or aborted based on that decision. Functions Transfer() and Refund() in SSC_u enforce confirming and aborting operations, respectively. On-chain transactions T_s and T_b are broadcast to invoke Transfer() and Refund().

It is not trivial to design transaction logics and conditions in SSC, requiring data provenance and availability check. For transaction logic design, Section III-B has stated that voting and sub-transaction completion operations in the 1st and 4th phase cannot be trivially designed as validating or confirming

sub-transactions. In other words, their transaction logic needs to be carefully considered. Input parameters of each function are utilized to help the condition checking process. To be more specific, input parameters shall represent data to be read or written, i.e., voting results and decisions. Besides, it is required that everyone can provide such input parameters to functions, not just limited to users and intermediaries. Therefore, it turns out to be non-trivial for designing conditions as the availability and provenance of such input parameters must be considered. Two questions are to be answered: (1) what are the concrete input parameters; (2) how do we prove the availability of data to be read or written using such input parameters?

IV. Transaction Logics and Conditions

We now present conditions and transaction logics recorded by functions in SSC_w and SSC_u. As shown in Section III-C, transaction logics represent operations, and conditions are used to see whether the data to be read or written is available when moving to each phase.

A. Transaction Logics in 4 Phases

Transaction Logics in the 1st Phase: Operations include registering a cross-chain transaction T on the relay chain and voting on underlying blockchains, modeled as transaction logics in the functions Registration() and Constructor(), respectively. In SSC_w, there is a data field DECISION, representing $D_T(M)$, i.e., the decision made on the relay chain. In Registration(), DECISION is set as PENDED, i.e., the relay chain has acknowledged the existence of T and will make a decision on it later. Recall that to invoke Registration(), the function T_f is broadcast. Thus, the confirmation of T_f indicates that the relay chain has acknowledged the existence of T.

We now illustrate how to model the voting operation as transaction logics in the function Constructor(). As shown in Algorithm 2, there is a data field STATUS in SSC_u, representing the status of T. Initially, STATUS is CLOSED. In Constructor() of SSC_u, u transfers the number of coins needed in (u, v) to SSC_u. Once Constructor() is confirmed, the data field STATUS becomes WAITING, and the YES voting result is obtained. As discussed in Section III-C, to call Constructor(), u broadcasts an on-chain transaction T_l.

As explained in Section III-B, miners shall validate special transactions instead of (u, v) during the voting process. In the above function Constructor(), miners validate T_l instead, where u transacts with SSC_u at the amount of funds needed in (u, v). In this circumstance, the confirmation of T_l means that u has transferred coins to SSC_u, i.e., u has sufficient coins to perform the sub-transaction (u, v). In other words, (u, v) is valid. Therefore, the YES voting result shall be generated, i.e., $V_{(u,v)} = 1$ and $R_{(u,v)} = 1$.

Transaction Logics in the 2nd phase: We now elaborate transaction logics of the function $\text{Collect}_w()$ used to read YES voting results from underlying blockchains and write them on the relay chain, as shown in Algorithm 1. Input parameters of $\text{Collect}_w()$ are used to check whether the YES voting result, i.e., the confirmation of T_l, is available. If yes, such input parameters can pass the condition checking. Then miners store such confirmation information on the relay chain. As discussed in Section III-C, how does such input parameters look like and how to store such confirmation information is related to conditions design, which will be introduced in the next subsection.

Transaction logics of $\text{ValidateYes}()$ and $\text{ValidateNo}()$ for making the CONFIRMING or ABORTING decision are included in Algorithm 1. There is a timer counting down the predefined maximum delay M. After this timer stops, in $\text{ValidateYes}()$ and $\text{ValidateNo}()$, miners search for the data representing for confirmations of T_l, i.e., $\mathbf{R}(M)$, on the relay chain. On one hand, in $\text{ValidateYes}()$, if confirmations of T_l from all underlying blockchains are found, i.e., $D_T(M) = g(\mathbf{R}(M)) = 1$, DECISION becomes CONFIRMING, whose value is 1. On another hand, in $\text{ValidateNo}()$, if there lack confirmations of T_l from some underlying blockchains, i.e., $|\mathbf{R}(M)| < |\{R_{(u,v)}, \forall (u, v) \in E\}|$, $D_T(M) = g(\mathbf{R}(M)) = 0$, DECISION is ABORTING, whose value is 0. As T_c is broadcast to call $\text{ValidateYes}()$, the confirmation of T_c means that a CONFIRMING decision is made. Similarly, the confirmation of T_r indicates that an ABORTING decision is made.

Transaction Logics in the 3 rd phase: We now design transaction logics of the function $\text{Collect}_u()$, where a CONFIRMING or ABORTING decision is read from the relay chain and written on underlying blockchains, as illustrated in Algorithm 2. When input parameters pass the condition checking, the confirmation of T_c or T_r is considered to be provided to this underlying blockchain, whose miners revise the data field STATUS to CONFIRMING or ABORTING accordingly. Different with $\text{Collect}_w()$, such confirmation is not directly recorded on underlying blockchains. The reason is related to blockchain structures and will be explained in the next subsection.

Transaction Logics in the 4th Phase: Sub-transactions shall be either confirmed or aborted based on the decision. Noticing that when underlying blockchains vote on the validity of each sub-transaction (u, v) in the 1st phase, the smart contract SSC_u temporarily holds funds that is used in (u, v). Thus, SSC_u must transact with u or v to complete or abort sub-transactions. With a CONFIRMING decision, i.e., $D_T(M) = 1$, SSC_u must transact with v, and (u, v) is completed. In contrast, with an ABORTING decision, i.e., $D_T(M) = 0$ and $V_{(u,v)} = 1$, SSC_u transacts with u. These operations are enforced through invoking functions Transfer() and Refund().

Transaction logics of Transfer() and Refund() are presented in Algorithms 1 and 2. In the 3 rd phase, the data field STATUS in SSC_u has been set as CONFIRMING or ABORTING. On one hand, in Transfer(), if STATUS equals CONFIRMING, coins are deducted from SSC_u and charged to v. On another hand, if STATUS is ABORTING, coins are returned back to u from SSC_u in Refund(). Confirmations of on-chain transactions T_s and T_b corresponding to Transfer() and Refund() represent the completion and abortion of sub-transactions, respectively.

As discussed in Section III-B, when moving to phase 2 and 3, voting results and the decision must be read and written across blockchains. To be more specific, certain data representing the confirmations of T_l, T_c and T_r shall be input parameters of functions $\text{Collect}_w()$ and $\text{Collect}_u()$ during phase changes.

Algorithm 1: The Smart Contract SSC_w on the Relay Chain ▷Cross-chain Transaction T.

1: **procedure** REGISTRATION(): T_f
2: **Input:** Cross-chain transaction T
3: DECISION $= -1$.
4: **procedure** $\text{COLLECT}_w()$: T_{ws}
5: **Input:** the block header BH, Merkle Proofs p containing T_l.
6: **if** T_l passes the inclusion proof providing p and BH **then**
7: Add BH to the Merkle tree MT whose root is SR_w.
8: **procedure** ValidateYes(): T_c
9: **if** timeElapsed $>= M$ **then**
10: **for** $\forall T_l \in H$ **do**
11: Search the Merkle tree MT whose root is SR_w for T_l.
12: **if** T_l is found in MT **then**
13: DECISION $= 1$.
14: Terminate.
15: **procedure** VALIDATENO(): T_r
16: **if** timeElapsed $>= M$ **then**
17: **for** $\forall T_l \in H$ **do**
18: Search the Merkle tree MT whose root is SR_w for T_l.
19: **if** T_l is not found in MT **then**
20: DECISION $= 0$.

Algorithm 2: The Smart Contract SSC_u on Each Underlying Blockchain ▷Sub-transaction (u, v).

1: **procedure** CONSTRUCTOR(): T_l
2: **if** u have sufficient funds **then**
3: Set the initial balance of this smart contract as X, deduct X coins from the account of u.
4: STATUS $=$ WAITING.
5: **procedure** $\text{COLLECT}_u()$: T_{us}
6: **Input:** the block header BH, Merkle proofs p containing T_c or T_r.
7: **if** T_c or T_r passes the inclusion proof **then**
8: STATUS $=$ CONFIRMING or ABORTING.
9: **procedure** TRANSFER(): T_s
10: **if** STATUS $=$ CONFIRMING **then**
11: Add X coins to v, reduce X coins from SSC_u.
12: **procedure** REFUND(): T_b
13: **if** STATUS $=$ ABORTING **then**
14: Add X coins to u, reduce X coins from SSC_u.

Thus, conditions in these two functions shall be able to check input parameters. Next, we show concrete input parameters and how to prove the existence of such confirmations using input parameters.

B. Conditions for Phase Changes

A user can verify the confirmation of a transaction given a block header and some light-weight information, commonly Merkle proofs in existing Blockchain developments, such as Bitcoin, Ethereum, and Tendermint [24], [25], [26], [27], [28]. Whether that transaction is included in that block can be known by efficiently comparing that block header with the Merkle proof of that transaction via an existing logarithmic complexity algorithm. If that transaction passes such an inclusion proof, its confirmation is naturally verified. In this way, we can verify confirmations of T_l, T_c, and T_r by conducting this inclusion proof using block headers and Merkle proofs.

Thus, input parameters representing confirmations of T_l, T_c, T_r are: headers of blocks that contain T_l, T_c, T_r and their Merkle proofs. In the function, $\text{Collect}_w()$ of SSC_w, a block header from an underlying blockchain is first checked on the inclusion of T_l provided with a Merkle proof. If yes, this block header is added as the rightmost leaf node to a Merkle tree of a block on the relay chain, following the design of Hyperservice. The root of that Merkle tree, denoted as SR_w, is included in the header of that block and updated upon each successful $\text{Collect}_w()$ invocation. For the function $\text{Collect}_u()$ of SSC_u, similar inclusion proof is necessary when conducting the condition check. However, $\text{Collect}_u()$ does not modify the underlying blockchain structure in adding block headers to a Merkle tree. The reason is that an underlying blockchain may not have an additional Merkle tree to store block headers from other blockchains like a relay chain, where such modifications shall not be supported to prevent hard forks.

In Fig. 5, *Unity* handles our cross-chain transaction T between two gas stations. Two sub-transactions in T are $\{(a, p), (p, b)\}$. In the 1st phase, to let miners vote on the validity of (a, p) and (p, b), gas station A and the intermediary broadcast T_l on chains 1 and 2, respectively. Once T_l is confirmed, to move into the 2nd phase, anyone can read such confirmation of T_l from chain 1 and notify the relay chain of writing such confirmation by broadcasting T_{ws}. Besides, anyone can broadcast T_c or T_r so that the relay chain makes the CONFIRMING or ABORTING decision. Similarly, when entering the 3 rd phase, anyone can broadcast T_{us} on its underlying blockchain to record this decision. Finally, this user can broadcast T_s or T_b on its underlying blockchain in the 4th phase, to complete or abort sub-transactions.

Unity ensures atomicity, confidentiality, and fault-tolerance. In the above example, anyone broadcasting transactions is only required to participate in its own underlying blockchain and the relay chain without accessing other underlying blockchains, ensuring confidentiality. Supposing that gas station A and the intermediary have registered T on the relay chain. Everyone can invoke all functions, where voting results and decisions can be read and written across the blockchain, and all operations are enforced. Besides, if the latest data, i.e., the YES voting result on chain 1 is updated after M expires, T_r will be confirmed. As a result, T_b is broadcast so that all sub-transactions are aborted.

V. THEORETICAL ANALYSES ON *UNITY*

We theoretically evaluate *Unity* on its atomicity assurance and the probability of successful cross-chain confirmations.

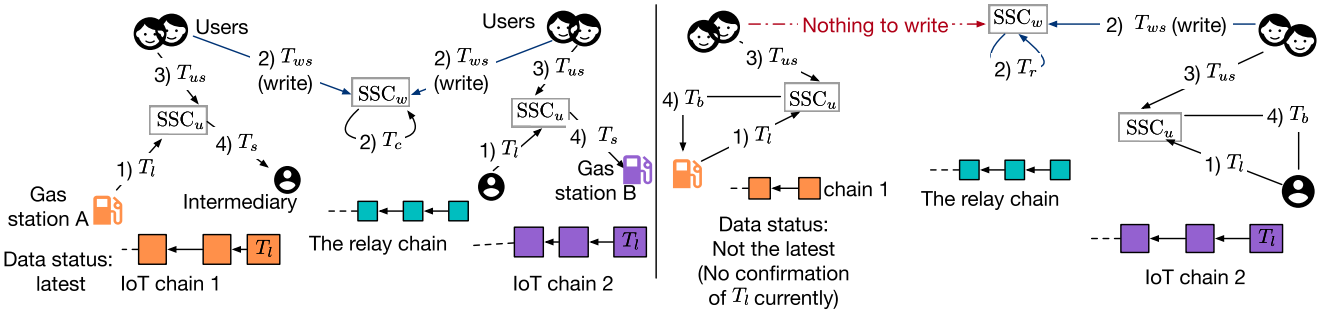


Fig. 5. By confirming T_l, $\mathrm{SSC_u}$ on chains 1 and 2 temporarily hold coins from gas station A and the intermediary during voting on the validity of (a, p) and (p, b). T_c and T_r contribute to enforcing the decision making operation. If T_{ws} and T_{us} are confirmed, r/w operations of voting results and the decision are enforced. T_s and T_b are used to enforce the completion and abortion of sub-transactions, respectively.

A. Atomicity Proof

We use ACTA, which can express the relationship among events that take place in a system, to prove the effectiveness of *Unity* in ensuring atomicity. In *Unity*, events include votings, making CONFIRMING or ABORTING decisions, confirming or aborting sub-transactions, and r/w failures. We first model the relationship among those events when atomicity is ensured. Then we model those events in *Unity* and analyze their correlations, which are equivalent to the relationship in ensuring the atomicity.

Modeling Conditions on Ensuring Atomicity: Recalling Section III, by referring to the consensus on whether to confirm or abort a cross-chain transaction among blockchains, either all or none of underlying blockchains confirm their sub-transactions. There are two conditions for ensuring atomicity. First, such a consensus is achieved. Second, a sub-transaction in an underlying blockchain is not confirmed or aborted until the achievement of such a consensus.

Theorem 1: A cross-chain transaction T is atomic if

$$\forall(u,v), \mathrm{Confirmed}_{(u,v)} \in H \Rightarrow (\mathrm{Confirm}_T \in H) \wedge \neg(\epsilon' \rightarrow \epsilon), \tag{1}$$

$$\text{or } \forall(u,v), \mathrm{Aborted}_{(u,v)} \in H \Rightarrow ((\mathrm{Abort}_T \in H) \wedge \neg(\epsilon' \rightarrow \epsilon)) \vee \exists \epsilon'' \in \mathrm{IE}, (\epsilon''_{(u,v)} \rightarrow \mathrm{Aborted}_{(u,v)})), \tag{2}$$

$\epsilon' \in \{\mathrm{Confirmed}_{(u,v)}, \mathrm{Aborted}_{(u,v)}\}, \epsilon \in \{\mathrm{Confirm}_T, \mathrm{Abort}_T\}$. $\mathrm{IE} = \{\mathrm{WrongTrans}_{(u,v)}\}$. H is the complete history of existing events. ϵ' and ϵ are events belonging to the history H. The predicate $\epsilon' \rightarrow \epsilon$ is true if event ϵ' *precedes* event ϵ in H. $\epsilon' \Rightarrow \epsilon$ explains the dependency among events: ϵ' happens if ϵ is true or satisfied. IE is a set of events, where a sub-transaction (u, v) is aborted locally regardless of the consensus result among blockchains. Specifically, if (u, v) is invalid, denoted as $\mathrm{WrongTrans}_{(u,v)}$, the transaction T_l between u and SSC_u will not be confirmed, i.e., (u, v) is aborted locally. $\mathrm{Confirm}_T$ and Abort_T represent the CONFIRMING or ABORTING consensus of T. $\mathrm{Confirmed}_{(u,v)}$ and $\mathrm{Aborted}_{(u,v)}$ means that the sub-transaction (u, v) has been confirmed or aborted, respectively.

Equation (1) means confirming a sub-transaction (u, v) needs the CONFIRMING decision. Equation (2) states precedent events of aborting (u, v): either an ABORTING decision has been achieved, or (u, v) is locally aborted by underlying blockchains due to events in IE.

Modeling Unity: Our *Unity* is formulated as follows.

Definition 5; Events in *Unity* are related to each other, and their relationship can be mathematically expressed.

$$\forall(u,v), \mathrm{Lock}_{(u,v)} \in H \Rightarrow \mathrm{WrongTrans}_{(u,v)} \notin H, \tag{3}$$

$$\forall(u,v), \mathrm{VoteYes}_{(u,v)} \in H \Rightarrow (\mathrm{Lock}_{(u,v)} \rightarrow \mathrm{Expired}_{(u,v)}) \wedge (\mathrm{Request}_{(u,v)} \in H) \tag{4}$$

$$\forall(u,v), \mathrm{RFail}_{(u,v)} \in H \Rightarrow \mathrm{Expired}_{(u,v)} \rightarrow \mathrm{Lock}_{(u,v)}$$

$$\forall(u,v), \mathrm{WFail}_{(u,v)} \in H \Rightarrow (\mathrm{Expired}_{(u,v)} \rightarrow \mathrm{Lock}_{(u,v)}) \vee (\mathrm{Request}_{(u,v)} \notin H) \tag{5, 6}$$

$$\mathrm{Confirm}_T \in H \Rightarrow \forall(u,v), (\mathrm{VoteYes}_{(u,v)} \in H) \tag{7}$$

$$\mathrm{Abort}_T \in H \Rightarrow \exists(u,v), \mathrm{RFail}_{(u,v)} \in H \vee \mathrm{WFail}_{(u,v)} \in H \vee \mathrm{Lock}_{(u,v)} \notin H \tag{8}$$

$$\forall(u,v), \mathrm{Confirmed}_{(u,v)} \in H \Rightarrow \mathrm{Confirm}_T \rightarrow \mathrm{Confirmed}_{(u,v)} \tag{9}$$

$$\forall(u,v), \mathrm{Aborted}_{(u,v)} \in H \Rightarrow (\mathrm{VoteYes}_{(u,v)} \in H \vee \mathrm{RFail}_{(u,v)} \in H \vee \mathrm{WFail}_{(u,v)} \in H \Rightarrow (\mathrm{Abort}_T \rightarrow \mathrm{Aborted}_{(u,v)})) \vee (\mathrm{WrongTrans}_{(u,v)} \rightarrow \mathrm{Aborted}_{(u,v)}) \tag{10}$$

Specifically, users and intermediaries register T on the relay chain, denoted as $\mathrm{Register}_T$. If (u, v) is checked as valid, coins needed for completing T is locked in a smart contract, denoted as $\mathrm{Lock}_{(u,v)}$. The expiration of the predefined maximum delay is $\mathrm{Expired}_{(u,v)}$. $\mathrm{Request}_{(u,v)}$ provides the voting result. $\mathrm{RFail}_{(u,v)}$ means that the voting result of (u, v) cannot be read. $\mathrm{WFail}_{(u,v)}$ means that the voting result of (u, v) cannot be written on the relay chain. $\mathrm{VoteYes}_{(u,v)}$ indicates that a YES voting result is obtained. Decisions on T are $\mathrm{Confirm}_T$ or Abort_T. Finally, (u, v) is either completed, or otherwise, denoted as $\mathrm{Confirmed}_{(u,v)}$ and $\mathrm{Aborted}_{(u,v)}$.

In (3), validation requests are successfully sent to all underlying blockchains, and T has been registered on the relay chain. (4) indicates that the YES voting result of (u, v) is successfully

obtained on an underlying blockchain *if* T_l corresponding to (u, v) is confirmed. Equation (5) means that if the voting result of (u, v) is updated after the arrival of r/w requests, such voting result cannot be read. Equation (6) means that if the voting result is not updated on time or r/w requests do not arrive, it cannot be written on the relay chain. In (7) and (8), T is considered valid only when YES voting results of all sub-transactions have been read from underlying blockchains and written on the relay chain. Equation (9) means that (u, v) is confirmed *if and only if* a decision on confirming T has been made before. Equation (10) states that in case an ABORTING decision is made, (u, v) is aborted only after this decision is received, or it has been aborted locally due to unsuccessful T_l, otherwise.

Analyzing Unity: We first analyze conditions for completing or aborting a sub-transaction in *Unity*.

Lemma 1: A valid sub-transaction cannot be completed or aborted until a decision is made, Mathematically speaking:

$\forall(u,v), \text{VoteYes}_{(u,v)} \in H \vee \text{RFail}_{(u,v)} \in H \vee \text{WFail}_{(u,v)} \in H \Rightarrow \neg(\epsilon' \rightarrow \epsilon)$.

Proof: Combining (5) and (6), we have $\text{RFail}_{(u,v)} \in H \vee \text{WFail}_{(u,v)} \in H \Rightarrow (\text{Expired}_{(u,v)} \rightarrow \text{Lock}_{(u,v)}) \vee (\text{Request}_{(u,v)} \notin H)$.

1. $\forall(u,v), (\text{VoteYes}_{(u,v)} \in H \Rightarrow \neg(\text{Confirmed}_{(u,v)} \rightarrow \text{Confirm}_T))$.

Assuming $\text{VoteYes}_{(u,v)} \in H$ and $\text{Confirmed}_{(u,v)} \rightarrow \text{Confirm}_T$. Thus, $\text{Confirmed}_{(u,v)} \in H$. With (9), $\text{Confirm}_T \rightarrow \text{Confirmed}_{(u,v)}$, which contradicts the assumption $\text{Confirmed}_{(u,v)} \rightarrow \text{Confirm}_T$.

2. $\forall(u,v), \text{RFail}_{(u,v)} \in H \vee \text{WFail}_{(u,v)} \in H \Rightarrow (\neg(\text{Confirmed}_{(u,v)} \rightarrow \text{Confirm}_T))$.

Assuming $\text{RFail}_{(u,v)} \in H \vee \text{WFail}_{(u,v)} \in H$ and $\text{Confirmed}_{(u,v)} \rightarrow \text{Confirm}_T$. Thus, $\text{Confirm}_T \in H$. Combining (7), $\forall(u,v), (\text{VoteYes}_{(u,v)} \in H)$, contradicting the assumption $\text{RFail}_{(u,v)} \in H \vee \text{WFail}_{(u,v)} \in H$.

3. $\forall(u,v), (\text{VoteYes}_{(u,v)} \in H \Rightarrow \neg(\text{Confirmed}_{(u,v)} \rightarrow \text{Abort}_T))$.

Assuming that $\text{VoteYes}_{(u,v)} \in H$ and $\text{Confirmed}_{(u,v)} \rightarrow \text{Abort}_T$. Thus, $\text{Confirmed}_{(u,v)} \in H$. Combining (9), we have: $\text{Confirm}_T \in H$, contradicting $\text{Abort}_T \in H$ in the assumption.

4. $\forall(u,v), \text{RFail}_{(u,v)} \in H \vee \text{WFail}_{(u,v)} \in H \Rightarrow \neg(\text{Confirmed}_{(u,v)} \rightarrow \text{Abort}_T)$.

Assuming that $\text{RFail}_{(u,v)}$ or $\text{WFail}_{(u,v)} \in H$, $\text{Confirmed}_{(u,v)} \rightarrow \text{Abort}_T$. Thus, $\text{Confirmed}_{(u,v)} \in H$. With (9) and (7), $\text{Confirm}_T \in H$, $\forall(u,v), \text{VoteYes}_{(u,v)} \in H$, while $\text{RFail}_{(u,v)}$ or $\text{WFail}_{(u,v)} \in H$.

5. $\forall(u,v), (\text{VoteYes}_{(u,v)} \in H \Rightarrow \neg(\text{Aborted}_{(u,v)} \rightarrow \text{Confirm}_T))$.

Assuming $\text{VoteYes}_{(u,v)} \in H$ and $\text{Aborted}_{(u,v)} \rightarrow \text{Confirm}_T$. Combining $\text{VoteYes}_{(u,v)} \in H$ and (10): $\text{Abort}_T \rightarrow \text{Aborted}_{(u,v)}$. With the assumption: $\text{Confirm}_T \in H$, $\text{Abort}_T \in H$, contradicting with each other.

6. $\forall(u,v), (\text{VoteYes}_{(u,v)} \in H \Rightarrow \neg(\text{Aborted}_{(u,v)} \rightarrow \text{Abort}_T))$.

Assuming that $\text{VoteYes}_{(u,v)} \in H$ and $\text{Aborted}_{(u,v)} \rightarrow \text{Abort}_T$. As $\text{VoteYes}_{(u,v)} \in H$, combining (10), we have $\text{Abort}_T \rightarrow \text{Aborted}_{(u,v)}$, which contradicts the assumption $\text{Aborted}_{(u,v)} \rightarrow \text{Abort}_T$.

Proving $\forall(u,v), \text{RFail}_{(u,v)} \in H \vee \text{WFail}_{(u,v)} \in H \Rightarrow \neg(\text{Aborted}_{(u,v)} \rightarrow \text{Abort}_T)$, $\forall(u,v), \text{RFail}_{(u,v)} \in H \vee \text{WFail}_{(u,v)} \in H \Rightarrow \neg(\text{Aborted}_{(u,v)} \rightarrow \text{Confirm}_T)$ are similar to 5 and 6.

Theorem 2: Combing Lemma 1, (9) and (10), the confirmation or abortion of a cross-chain transaction using *Unity* is:

$$\begin{aligned}
&\forall(u,v), \text{Confirmed}_{(u,v)} \in H \Rightarrow \text{Confirm}_T \in H \wedge \neg(\epsilon' \rightarrow \epsilon),\\
&\forall(u,v), (\text{Aborted}_{(u,v)} \in H \Rightarrow (\text{Abort}_T \in H \wedge \neg(\epsilon' \rightarrow \epsilon))\\
&\vee \exists \epsilon'' \in \text{IE}, (\epsilon'' \rightarrow \text{Aborted}_{(u,v)})),
\end{aligned}$$

matching conditions in Theorem 1, i.e., *Unity* ensures atomicity.

Proof: $\forall(u,v), \text{Confirmed}_{(u,v)} \in H$

$$\begin{aligned}
&\Rightarrow \text{Confirm}_T \rightarrow \text{Confirmed}_{(u,v)}\\
&\Rightarrow (\text{Confirm}_T \in H) \wedge (\text{Confirm}_T \rightarrow \text{Confirmed}_{(u,v)})\\
&\Rightarrow (\text{VoteYes}_{(u,v)} \in H) \wedge (\text{Confirm}_T \rightarrow \text{Confirmed}_{(u,v)})\\
&\Rightarrow \text{Confirm}_T \in H \wedge \neg(\epsilon' \rightarrow \epsilon)\\
&\forall(u,v), \text{Aborted}_{(u,v)} \in H \Rightarrow\\
&(\text{VoteYes}_{(u,v)} \in H \vee \text{RFail}_{(u,v)} \in H\\
&\vee \text{WFail}_{(u,v)} \in H \Rightarrow (\text{Abort}_T \rightarrow \text{Aborted}_{(u,v)})) \vee\\
&(\text{Lock}_{(u,v)} \notin H \rightarrow \text{Aborted}_{(u,v)}) \Rightarrow\\
&(\neg(\epsilon' \rightarrow \epsilon) \wedge \text{Abort}_T \in H) \vee \text{WrongTrans}_{(u,v)}\\
&\rightarrow \text{Aborted}_{(u,v)} \Rightarrow (\neg(\epsilon' \rightarrow \epsilon) \wedge \text{Abort}_T \in H)\\
&\vee \exists \epsilon'' \in \text{IE}, (\epsilon'' \rightarrow \text{Aborted}_{(u,v)}).
\end{aligned} \tag{11}$$

B. *Successful Rate in* Unity

We consider concrete circumstances where r/w failures with two kinds of data status can occur. First, the voting results are not updated within M because the confirmation speed in underlying blockchains is extremely low. Denote the probability of such a low confirmation speed as p_{vb}. Second, voting results are updated but users and intermediaries fail to deliver the corresponding r/w requests. p_u and p_i are probabilities of such failures rising from a user u and intermediary i, respectively.

Lemma 2: The probability P in successfully confirming T is modeled as a multivariable function, with respect to p_{vb}, p_u and p_i.

$$P = P(\vec{p}_u, \vec{p}_i, \vec{p}_{vb}) = \prod\nolimits_u (1 - p_u) \prod\nolimits_i (1 - p_i) \prod\nolimits_b (1 - p_{vb}). \tag{12}$$

Proof: A valid cross-chain transaction T in *Unity* can be confirmed under the following two independent conditions. First, users and intermediaries can register this cross-chain transaction on the relay chain. P_1 represents the probability that this condition is satisfied, and $P_1 = \prod_u (1 - p_u) \prod_i (1 - p_i)$. Second, all underlying blockchains do not experience low confirmation speed when voting on the correctness of sub-transactions. The probability that this condition is satisfied is

$P_2 = \prod_u (1-p_u)\prod_i (1-p_i)$. Thus, we have: $P = P_1 P_2 = \prod_u (1-p_u)\prod_i (1-p_i)\prod_b (1-p_{vb})$.

Theorem 3: In *Unity*, under the r/w failures with the second kind of the data status, a cross-chain transaction is more likely to be terminated.

Proof: We first analyze the impact of r/w failures under the second kind of data status, denoted as $P^{|U|+|I|}$, which can be formulated as: $\dfrac{\partial^{|U|+|I|}(P)}{\cdots \partial p_u \partial p_i \cdots} = -\prod_b (1-p_{vb})$, where $|U|$ and $|I|$ represent the numbers of users and intermediaries, respectively.

Similarly, we have the effect of r/w failures with the first kind of data status, denoted as $P^{|B|}$:

$$P^{(|B|)} = \frac{\partial^{|B|}(P)}{\cdots \partial p_{vb} \cdots} = -\prod\nolimits_u (1-p_u)\prod\nolimits_i (1-p_i). \quad (13)$$

$|B|$ is the number of underlying blockchains. It is sufficient to show that $P^{|B|} > P^{|U|+|I|}$ given the same probabilities in p_{vb}, p_u and p_i:

$$\begin{aligned}
&\text{If } \forall(u,i,b),\ p_u = p_i = p_{vb} = p,\\
&\text{then: } P^{|B|} > P^{|U|+|I|}. \text{ We have:}\\
&P^{(|B|)} = -\prod\nolimits_u (1-p_u)\prod\nolimits_i (1-p_u)\\
&= -\prod\nolimits_u (1-p_u)\prod\nolimits_i (1-p_i)\\
&= -(1-p)^{|U|+|I|}\\
&P^{|U|+|I|} = -\prod\nolimits_b (1-p_{vb}) = -(1-p)^{|B|}\\
&< -(1-p)^{|U|+|I|} = P^{|B|}.
\end{aligned}$$

VI. Evaluation

In this section, we compare the performance of *Unity* with the state-of-the-art cross-chain confirmation platform, Hyperservice, on the atomicity assurance property, the successful confirmation probability, and delay. In Section VI-A, we describe implementation details, our cross-chain transaction settings, and simulations of failures. In Section VI-B, we test the cross-chain atomicity assurance. In Section VI-C, we examine the probability of successfully confirming a cross-chain transaction under failures. In Section VI-D, we conduct analyses on the delay for confirming a cross-chain transaction with regard to the average time of confirming transactions on underlying blockchains and the relay chain.

A. Implementation

Blockchain and Transaction Settings: Underlying blockchains are implemented as Ethereum blockchains, which have already existed in real-world scenarios [6]. The relay chain incorporates the NSB blockchain developed by Hyperservice. Its consensus protocol is Proof-of-Action, a variant of the Proof-of-X (PoX) protocol. Each block on the NSB blockchain contains a Merkle tree whose leaf nodes record headers from Ethereums.

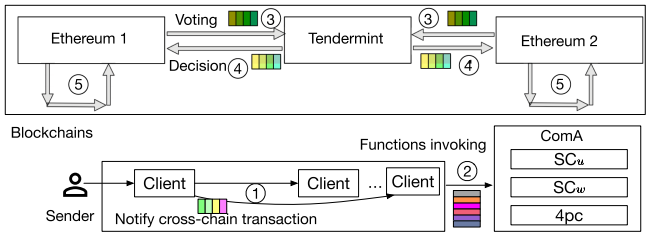


Fig. 6. Confirming a cross-chain transaction across two Ethereums in *Unity*.

An account on each Ethereum is assumed to be owned by a user, which can only participate in one Ethereum. Thus, a user cannot access data from another Ethereum. In these circumstances, confidentiality is ensured. An intermediary owns three accounts on the relay chain and two Ethereums, denoted as p_r, p_1, and, p_2, respectively. Following the convention in Hyperservice [16], we use two kinds of cross-chain transactions. First, each user only owns one account on its Ethereum and transfers coins to another. The second is a mutual payment operation, where each user sends and receives ethers with one another. For example, two users A and B own $2m$ accounts on Ethereum 1 and 2, denoted as $A_1, \ldots A_2 m, B_1, \ldots, B_2 m$. A_i sends 1 ether to B_i, and receives 2 ethers from B_i. Following Section III-A, $S = \{A_1, \ldots A_m, B_{m+1}, \ldots, B_2 m\}, I = \{p_r, p_1, p_2\}, R = \{A_{m_1}, \ldots A_2 m, B_1, \ldots, B_m\}, E = \{(A_i, p_1), (p_2, B_i) | i \in [1, m]\} \cup \{(B_i, p_2), (p_1, A_i) | i \in [m+1, 2m]\}$. We let $m = 1$, and sub-transactions are: $(A_1, p_1), (p_2, B_1), (B_2, p_2), (p_1, A_2)$.

Smart Contracts $\mathbf{SSC}_w$ *and* $\mathbf{SSC}_u$*:* A function in SSC$_w$ on the relay chain is implemented as a built-in `Tendermint ABCI application` written with `Go` language. The timer in ValidateYes() and ValidateNo() is implemented via the built-in `timer` package of `Go`. Users interact with those functions by implementing two interfaces `CheckTx()`, `Commit()` provided by `Tendermint`, by calling which, transactions T_f, T_{ws}, T_c, and T_r can be checked and broadcast. Different with SSC$_w$ on the relay chain, the smart contract SSC$_u$ on Ethereums is written as a class using `Solidity` v0.8. Users interact with Ethereums through the `Web3.js` library providing multiple interfaces on invoking functions in SSC$_u$. Specifically, SSC$_u$ is first compiled to bytecodes using the interface `solc`, so that transaction logics can be recognized by the Ethereum Virtual Machine (`EVM`) modules of miners. In particular, parameters provided to functions in SSC$_u$ are encoded following the `Ethereum ABI` encoding standard, in the form of fulfilling `data` fields of corresponding functions.

Workflow: To deploy *Unity*, we just need to deploy $\mathbf{SSC}_w$ and $\mathbf{SSC}_u$ on the relay chain and underlying blockchains, respectively. The workflow of *Unity* when confirming cross-chain transactions among two Ethereums is illustrated in Fig. 6. Each user has a cross-chain transaction processing client that automatically warps all functions invoking operations to on-chain transactions and interacts with one another. A sender first calls clients of users and intermediaries (①) to register T on the relay chain and send validation requests to underlying blockchains by invoking Register() and Constructor() (②). After that, if

users and intermediaries go offline, functions can be invoked by anyone so that all operations are enforced ((③)(④)(⑤)).

Failures Simulation: We use two specific errors to simulate r/w failures under two kinds of data status. First, a blockchain has not updated the data to be read or written when requests arrive. We let miners in Ethereums put off the calling of a mining interface `miner.start()` until the predefined maximum delay expires, namely low confirmation speed. Second, data to be read or written is the latest resulting from delivery errors when users or intermediaries deliver r/w requests. We just let confirmation processing clients of users or intermediaries terminate.

Performance Metrics: First, if the number of cross-chain transactions losing atomicity, denoted as E, does not equal 0, atomicity is lost. Second, a ratio, p_u, between the number of unsuccessful and total cross-chain transactions, quantifies the probability that a cross-chain transaction is unsuccessful, i.e., either aborted or losing atomicity. Third, the elapsed time, D, from submitting a cross-chain transaction until it is confirmed, exposes confirmation latency.

Benchmarks: We evaluate our 4pc and SSC against the state-of-art cross-chain transaction confirmation platform Hyperservice on the above performance metrics. Though [14], [29] serves as an atomic assurance protocol for swapping coins across blockchains, they do not incorporate intermediaries. To be more specific, each user must participate in all blockchains during cross-chain confirmations. Therefore, [14], [29] cannot become a comparison benchmark for cross-chain atomicity with the confidentiality requirement.

Results are summarized as follows.

- 4pc prevents the loss of atomicity from r/w failures with the first data status category. SSC handles the second category.
- If the failure probability is higher than 0.7, only 4pc is enough to ensure atomicity.
- If r/w failures in the first kind of data status occur at a probability higher than 0.5, a cross-chain transaction is aborted in *Unity* at a probability near 1.
- In *Unity*, the delay for confirming a cross-chain transaction is more sensitive to the relay chain's latency than underlying blockchains.
- The confirmation delay of 4pc is near Hyperservice.

B. Evaluating the Atomicity

We first prove that the cross-chain atomicity under different failures can be ensured using *Unity*. To move forward, we evaluate 4pc on the atomicity assurance.

Atomicity in Unity: For each transaction type, we send 10 cross-chain transactions into the network. Besides, we randomly select an Ethereum blockchain and a user, either a user or an intermediary, experiencing low confirmation speed and delivery errors, respectively. In Table I, numbers of cross-chain transactions losing their atomicity, E, are recorded. In Table I, all cross-chain transactions are atomic under the two specific errors. This result indicates that *Unity* ensures atomic cross-chain transactions under r/w failures with two data status categories.

Atomicity in 4pc: We compute numbers of cross-chain transactions losing atomicity, E, under different numbers of failures in 4pc, compared with Hyperservice. Cross-chain transactions are asset movement operations. Different from the previous experiment, errors can occur at the same time when confirming a cross-chain transaction. Results are shown in Fig. 7(a), (b).

TABLE I
NUMBER OF CROSS-CHAIN TRANSACTIONS *LOSING ATOMICITY* UNDER DIFFERENT KINDS OF ERRORS

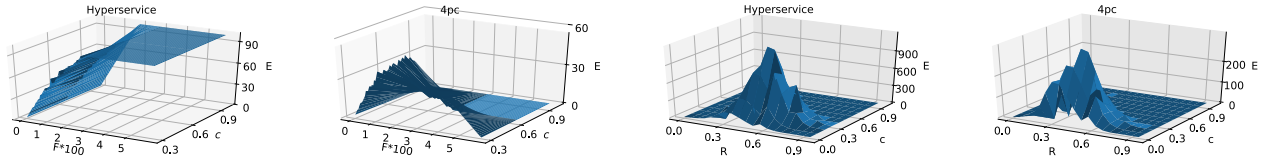
	#delivery errors = 10			#low confirmation speed = 10		
	Hyperservice	4pc	*Unity*	Hyperservice	4pc	*Unity*
Asset movements	4	5	0	2	0	0
Mutual payment	5	3	0	7	0	0

- In *Unity*, the delay for confirming a cross-chain transaction is more sensitive to the relay chain's latency than underlying blockchains.
- The confirmation delay of 4pc is near Hyperservice.

Parameter settings are illustrated as follows. We have 100 cross-chain transactions. A cross-chain transaction experiences 3 kinds of errors at most. Delivery errors do not take place in the 1st phase. The maximum number of errors is 300. The x-axis is the actual number of errors, denoted as F. The y-axis is the percentage of low confirmation speed, denoted as c. The z-axis is the number of cross-chain transactions that lose atomicity, denoted as E.

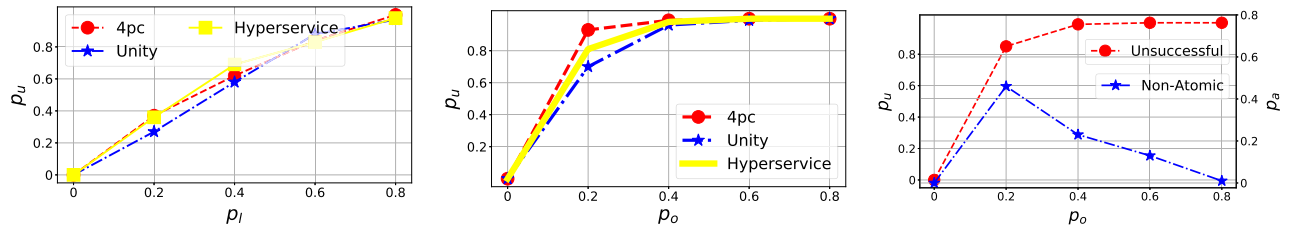
There are two findings in Fig. 7(a), (b). First, in Fig. 7(b), almost all cross-chain transactions are atomic in 4pc if $F > 200$, no matter how many delivery errors there are. The reason is as follows. If low confirmation speed and delivery errors take place when validating the same cross-chain transaction, this cross-chain transaction will be aborted by 4pc eventually. With more errors, there will be more cross-chain transactions experiencing low confirmation speed, and more cross-chain transactions can maintain their atomicity by being aborted in 4PC. Second, in Fig. 7(a), the number of cross-chain transactions losing atomicity in Hyperservice keeps increasing with more errors. The reason is that Hyperservice cannot defend against both kinds of errors. More errors in this circumstance result in more cross-chain transactions losing their atomicity.

To further verify these two findings, we expanded the evaluation scale by simulating confirmation results in Hyperservice and 4pc. Specifically, there are at most 6542 errors that can occur at any time, including the 1st phase. We iterate all probable circumstances that errors can occur when validating a cross-chain transaction in Hyperservice and 4pc. The x-axis, R represents the probability that an error occurs, calculated by the ratio between the actual number of errors and the maximum one. Results are in Fig. 7(c), (d). In Fig. 7(d), when R is larger than 0.7, almost all cross-chain transactions are atomic in 4pc. Another interesting finding is that in Hyperservice, as illustrated by Fig. 7(c), when $R > 0.98$ and $c > 0.6$, the number of cross-chain transactions that lose atomicity is also near 0. The reason is that delivery errors can occur at the very beginning before a cross-chain transaction is sent to miners. In these circumstances, validation on this cross-chain transaction does not start, and none of the sub-transactions are confirmed.



(a) More cross-chain transactions lose atomicity with higher F atomicity.
(b) When $F > 200$, almost all cross-chain transactions are atomic in 4pc.
(c) Almost all transactions are atomicatomic if $R > 98\%$ in Hyperser-ity regardless of error types. vicce.
(d) If $R > 0.7$, 4pc ensures atomic-

Fig. 7. F represents how many times failures occur. c: the probability that validation status are not read successfully on time due to low confirmation speed. R: the probability that failures occur. E: Total number of cross-chain transactions losing atomicity.



(a) p_l: probability of low confirmatino speed. All most all cross-chain transactions fail after $p_l > 0.8$.
(b) p_o: probability of delivery errors. All most all cross-chain transactions fail after $p_o > 0.5$.
(c) p_a: probability of atomicity loss.

Fig. 8. Probabilities of unsuccessful cross-chain transaction transactions (p_u) as to error probabilities (p_l, p_o). (a), (b): Low confirmation speed less substantially affect cross-chain confirmations.

Insight: In the event that failures come about at a probability higher than 0.7, 4pc can ensure atomicity.

C. Evaluating the Unsuccessful Probability

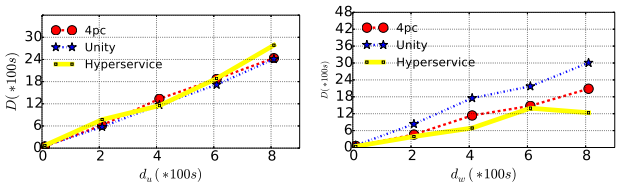
This evaluation is in two moves. We first examine how low confirmation speed and delivery errors affect probabilities of unsuccessful confirmations in 4pc, Hyperservice, and *Unity*. We distinguish unsuccessful cross-chain transactions into two categories: aborted and losing atomicity. Then, the relationship between numbers of cross-chain transactions in these two categories using 4pc is learned.

For each p_o varying from $[0, 1]$ representing the probability of delivery errors, we use a uniform distributed probability function to generate a variable v. If $v < p_o$, we simulate delivery errors for a user when validating a cross-chain transaction and check whether it is successfully confirmed later. This operation is repeated 100 times. We record the number of unsuccessful times and compute the ratio between such a number and 100. The same process is conducted for the probability at low confirmation speed p_l. In Fig. 8, x-axes are probabilities in low confirmation speed and delivery errors, denoted as p_o, p_l, respectively. The y-axis is the ratio above, denoted as p_u.

We now explain four findings from Fig. 8. First, in Fig. 8(b), when the probability of delivery errors for each user and intermediary is higher than 0.5, almost all cross-chain transactions fail. Second, in Fig. 8(b), where only delivery errors occur, *Unity* outperforms Hyperservice and 4pc. This result is easy to explain as SSC in *Unity* eliminates the need for users and intermediaries to deliver r/w requests. Third, in Fig. 8(b), more unsuccessful cross-chain transactions appear in 4pc than Hyperservice if only delivery errors appear. The reason is that users and intermediaries in 4pc need to more frequently deliver requests of reading or writing voting results and decisions across blockchains. Forth, in Fig. 8(a), cross-chain transactions have an almost equal probability of being confirmed in these three proposals given the context that only low confirmation speed exists. This is because none of the three protocols can ensure the confirmation of cross-chain transactions in the presence of r/w failures with the first kind of data status.

Insights: By diving into these four findings, we obtain two insights as follows. Based on the first finding, r/w failures with the first kind of data status have less significant and negative effects on the probability of successfully confirming a cross-chain transaction, which verifies the result in Theorem 3. Based on the second finding, under delivery errors, a cross-chain transaction is more probable to be confirmed in *Unity*, compared to Hyperservice.

Then we verify the analysis in Fig. 7(d) that with a large number of delivery errors, a cross-chain transaction is highly probable to be aborted, rather than losing atomicity, in 4pc. Unsuccessful cross-chain transactions are separated into two categories: aborted and losing atomicity. We record the numbers of these two categories of unsuccessful cross-chain transactions in 4pc under different probabilities in delivery errors. The result is shown in Fig. 8(c). The x-axis is the probability of delivery errors. The y-axes are the percentages of aborted cross-chain transactions and those losing their atomicity. Results show that if the probability that a user or intermediary experiences delivery errors is larger than 0.6, 4pc can ensure atomicity at a probability higher than 98% by aborting cross-chain transactions.



(a) Cross-chain confirmation delay (D) and Ethereum delay (d_u). (b) Cross-chain confirmation delay (D) and the relay chain (d_w).

Fig. 9. Relay chain has more significant impact on the cross-chain confirmation latency D.

D. Evaluating the Delay

We assessed the latency of confirming a cross-chain transaction, D, with respect to delays in underlying blockchains and the relay chain in this experiment. We let d_u and d_w represent the delay in the underlying blockchain and the relay chain, respectively. By changing the mining difficulty in Ethereum, the delay for confirming a sub-transaction in an underlying blockchain, d_u, varies from [10 s–800 s]. By setting additional sleeping commands in the `checkTx()` interface, the delay on the relay chain, d_w, varies from [10 s–800 s]. For each d_u, we repeatedly send an asset movement cross-chain transaction into the network for 100 times, and record the average latency for confirming one cross-chain transaction in Fig. 9(a). The $x-$axis is d_u, and the y-axis is the average latency, D. Similarly, D with respect to d_w is in Fig. 9(b).

Insights: Fig. 9 gives us three insights. First, in Fig. 9(a), (b), the confirmation delay of the relay chain has a more significant impact on D in *Unity*. The reason is that the relay chain collects voting results from all underlying blockchains, requiring at least 4 validations. In contrast, there are only three validations on each underlying blockchain, fewer than the relay chain. Second, in Fig. 9(a), (b), D in 4pc is about the same as that in Hyperservice, while *Unity* takes much longer time to confirm a cross-chain transaction. The reason is that 4pc does not incorporate smart contracts on the relay chain in writing voting results, incurring much fewer validations. Third, combing the insight in Section VI-B, 4pc achieves a balance between atomicity and efficiency: with masses of failures, 4pc ensures atomicity without a sharp increase in D.

VII. Related Work

All involved blockchains must jointly validate a cross-chain transaction. Typically, a cross-chain transaction is split into multiple sub-transactions, each being validated on an underlying blockchain [14], [29], [30], [31], [32], [33]. To issue all sub-transactions, a user is required to participate in all underlying blockchains in these works. However, in distributed applications with the confidentiality requirement, it is not easy for a user to participate in all blockchains. Therefore, these works do not adapt to distributed applications with confidentiality requirements.

Some works incorporate intermediaries to protect confidentiality [15], [16], [17], [21], [22], [27], [32]. An intermediary is required to have access for multiple blockchains. To complete a cross-chain transaction, users from different blockchains can transact with that intermediary, eliminating the need for participating in one another's blockchain. Specifically, the validation status must be read from a blockchain and written on other blockchains, such that validators on all blockchains can keep the same knowledge on the transaction status. Unfortunately, such read/write operations can fail in asynchronous distributed scenarios, as discussed by existing works [14], [33]. Under read/write failures, validators on different blockchains have different knowledge on the transaction status, and must make different decisions on that cross-chain transaction, resulting in the atomicity loss eventually.

Generally, the two-phase-commit protocol [14], [33], [34] ensures atomicity, whose variants are used in diverse scenarios, including but not limited to blockchains. Ins [14], [33], there is a witness blockchain helping coordinate the two-phase-commit-process among involved blockchains. To be more specific, each user participates in all involved blockchains, transmits the validation status at appropriate points across blockchains. However, if a user loses in delivering such validation status, atomicity is lost. Besides, as each user can access data of all involved blockchains, the confidentiality requirement is not satisfied. Other cross-chain transaction proposals tried to let users return funds back if failures take place [16]. However, they assume that all users and intermediaries are honest enough to sign the funds reversion transaction. In *Unity*, such funds are locked by smart contracts and will be enforced to be returned.

There also exist research works on enabling cross-shard transactions in a blockchain system [35], [36], [37], [38], [39]. Similar to cross-chain transactions, a cross-shard transaction is split into several sub-transactions and is validated by different sub-communities (i.e., shards) jointly. However, those proposals cannot be directly applied in cross-chain scenarios. For one thing, a user can freely join any shards by broadcasting transactions, sacrificing confidentiality. For another, users and miners in different shards can communicate directly as they are managed by the same blockchain system. However, users and miners from heterogeneous blockchain systems typically obey different consensus protocols, making such communications impractical.

VIII. Discussion

Incentive Mechanisms: All users on an underlying blockchain shall be encouraged to invoke $\text{Collect}_w()$ and $\text{Collect}_u()$, whose input parameters represent the data to be read and write across blockchains during phase changes. Such an incentive mechanism could be a plug-in unit of our proposal. Fortunately, existing works, such as BTCRelay [28], have provided multiple solutions to this problem. These methods can be directly applied to our proposal by adding new functions in SSC_w and SSC_u, where users obtain rewards once their block headers and Merkle proofs have passed the inclusion proof.

Selection of Intermediaries: Considering the scenario where commercial corporations cooperate with one another, where special roles can access part of the data in corporations. Those roles must participate in different blockchains managed by those corporations and can take on the responsibility of intermediaries.

Therefore, checking whether somebody can become an intermediary is equivalent to checking the role of this user, which is a popular topic and may become the research dominance in the security and privacy community, going parallel with our focus.

IX. Conclusion

This paper proposes *Unity*, which protects the transaction atomicity under r/w failures across permissioned blockchains, so that the confidentiality is also ensured by design. We capture two data status categories under r/w failures and propose separate solutions for ensuring atomicity. For the first kind of data status, where data is not the latest version when r/w failures occur, *Unity* first presents 4pc, a four-phase-commit protocol, to ensure that a cross-chain transaction is either confirmed or aborted. For the second kind of data status, where data is the latest while it cannot be read/written across blockchains, *Unity* incorporates several smart contracts, denoted as SSC, whose highlight is to enforce all operations during cross-chain confirmation once such latest data is available. Theoretical and experimental results investigated *Unity* and provided several important insights. First, *Unity* ensures atomicity under r/w failures. Second, if failures occur frequently, 4pc ensures atomicity without prolonging the confirmation latency, compared with the state-of-the-art cross-chain confirmation platform, Hyperservice. Actually, SSC requires more validations on the relay chain, extending the confirmation delay. Our future work may consider how to reduce such delays for a better user experience.

References

[1] K. Karlsson et al., "Vegvisir: A partition-tolerant blockchain for the Internet-of-Things," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1150–1158.

[2] Z. Yang, K. Yang, L. Lei, K. Zheng, and V. C. Leung, "Blockchain-based decentralized trust management in vehicular networks," *IEEE Internet of Things J.*, vol. 6, no. 2, pp. 1495–1505, Apr. 2019.

[3] A. S. Sani et al., "Xyreum: A high-performance and scalable blockchain for IIoT security and privacy," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1920–1930.

[4] J. Huang, L. Kong, G. Chen, L. Cheng, K. Wu, and X. Liu, "B-IoT: Blockchain driven Internet of Things with credit-based consensus mechanism," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1348–1357.

[5] B. Team, "Cars that negotiate with charging stations," 2021. [Online]. Available: https://www.bosch.com/stories/dlt-cars-that-negotiate-with-charging-stations/

[6] E. Team, "50,000 electric car charging stations in Europe will accept cryptocurrency payments," 2021. [Online]. Available: https://voi.id/en/technology/61952/50000-electric-car-charging-stations-in-europe-will-accept-cryptocurrency-payments

[7] M. J. Amiri, D. Agrawal, and A. E. Abbadi, "CAPER: A cross-application permissioned blockchain," in *Proc. VLDB Endowment*, vol. 12, no. 11, pp. 1385–1398, 2019.

[8] E. Kokoris-Kogias, E. C. Alp, L. Gasser, P. Jovanovic, E. Syta, and B. Ford, "CALYPSO: Private data management for decentralized ledgers," in *Proc. VLDB Endowment*, vol. 14, pp. 586–599, 2020.

[9] E. Androulaki, C. Cachin, A. De Caro, and E. K. Kogias, "Channels: Horizontal scaling and confidentiality on permissioned blockchains with application on hyperledger fabric," EPFL, 2018.

[10] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, 1983.

[11] P. A. Bernstein and N. Goodman, "The failure and recovery problem for replicated databases," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 1983, pp. 114–122.

[12] T. Pelkonen et al., "Gorilla: A fast, scalable, in-memory time series database," in *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1816–1827, 2015.

[13] C. Mohan, R. Strong, and S. Finkelstein, "Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 1983, pp. 89–103.

[14] V. Zakhary, D. Agrawal, and A. El Abbadi, "Atomic commitment across blockchains," in *Proc. VLDB Endowment*, vol. 13, no. 9, pp. 1319–1331, 2020.

[15] S. D. Lerner, "Rootstock: Bitcoin powered smart contracts," 2015. [Online]. Available: https://docs.rsk.co/RSK_White_Paper-Overview.pdf

[16] Z. Liu et al., "HyperService: Interoperability and programmability across heterogeneous blockchains," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 549–566.

[17] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," *White Paper*, 2016.

[18] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres, "Ensuring relaxed atomicity for flexible transactions in multidatabase systems," *ACM SIGMOD Rec.*, vol. 23, no. 2, pp. 67–78, 1994.

[19] P. K. Chrysanthis and K. Ramamritham, "Synthesis of extended transaction models using ACTA," *ACM Trans. Database Syst.*, vol. 19, no. 3, pp. 450–491, 1994.

[20] A. Banafa, "IoT and blockchain convergence: Benefits and challenges," *IEEE Internet of Things*, 2017.

[21] POA bridge, 2018. [Online]. Available: https://github.com/poanetwork/poa-bridge

[22] Comos, 2019. [Online]. Available: https://cosmos.network

[23] M. Ali, J. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A global naming and storage system secured by blockchains," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 181–194.

[24] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, "FlyClient: Super-light clients for cryptocurrencies," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 928–946.

[25] A. Kiayias and D. Zindros, "Proof-of-work sidechains," in *Proc. Springer Int. Conf. Financial Cryptogr. Data Secur.*, 2019, pp. 21–34.

[26] P. Gaži, A. Kiayias, and D. Zindros, "Proof-of-stake sidechains," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 139–156.

[27] V. Buterin, "Chain interoperability," *R3 Res. Paper*, 2016.

[28] BTC Rrelay, 2019. [Online]. Available: http://btcrelay.org/

[29] M. Herlihy, "Atomic cross-chain swaps," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2018, pp. 245–254.

[30] Atomic cross-chain trading, 2018. [Online]. Available: https://en.bitcoin.it/wiki/Atomic_cross-chain_trading

[31] Alt chains and atomic transfers, 2013. [Online]. Available: https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949

[32] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. Knottenbelt, "XCLAIM: Trustless, interoperable, cryptocurrency-backed assets," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 193–210.

[33] S. A. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 1299–1316.

[34] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, 2006.

[35] Z. Hong, S. Guo, P. Li, and W. Chen, "Pyramid: A layered blockchain sharding system," in *Proc. IEEE Int. Conf. Commun.*, 2021, pp. 1–10.

[36] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[37] Y. Tao, B. Li, J. Jiang, H. C. Ng, C. Wang, and B. Li, "On sharding open blockchains with smart contracts," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1357–1368.

[38] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A secure, scale-out, decentralized ledger via sharding," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 583–598.

[39] M. Zamani, M. Movahedi, and M. Raykova, "RapidChain: Scaling blockchain via full sharding," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 931–948.

Yuechen Tao received the BEngr degree from the Department of Computer Science, Shandong University, China, in 2016. Since 2016, she has been with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, where she is currently working toward the PhD degree. Her current research interests include performance analyses and improvements in blockchains, atomicity assurance of cross-chain transactions.

Bo Li (Fellow, IEEE) received the BEng (summa cum laude) degree in computer science from Tsinghua University, Beijing, China, and the PhD degree in electrical and computer engineering from the University of Massachusetts at Amherst. He is a chair professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He held a Cheung Kong visiting chair professor with Shanghai Jiao Tong University between 2010 and 2016. He was an adjunct researcher with the Microsoft Research Asia (MSRA) (1999–2006) and with the Microsoft Advanced Technology Center (2007–2009). He made pioneering contributions in multimedia communications and the Internet video broadcast, in particular the Coolstreaming system, which was credited as first large-scale Peer-to-Peer live video streaming system in the world. It attracted significant attention from both industry with substantial VC investment, and academia in receiving the Test-of-Time Best Paper Award from IEEE INFOCOM (2015). He received 6 Best Paper Awards from IEEE including INFOCOM (2021). He has been an editor or a guest editor for more than a two dozen of IEEE and ACM journals and magazines. He was the Co-TPC chair for IEEE INFOCOM 2004.

Baochun Li (Fellow, IEEE) received the BEngr degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995, and the MS and PhD degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000, respectively. Since 2000, he has been with the Department of Electrical and Computer Engineering, University of Toronto, where he is currently a professor. He holds the Bell Canada endowed chair in computer engineering since August 2005. His research interests include cloud computing, distributed systems, datacenter networking, and wireless systems.