Enabling Complete Atomicity for Cross-chain Applications Through Layered State Commitments

Yuandi Cai, Ru Cheng, Yifan Zhou, Shijie Zhang, Jiang Xiao, and Hai Jin
National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, China
Email: shijiezhang@hust.edu.cn, jiangxiao@hust.edu.cn

***Abstract*—Cross-chain *Decentralized Applications* (dApps) are increasingly popular for their ability to handle complex tasks across various blockchains, extending beyond simple asset transfers or swaps. However, ensuring all dependent transactions execute correctly together, known as *complete atomicity*, remains a challenge. Existing works provide *financial atomicity*, protecting against monetary loss, but lack the ability to ensure correctness for complex tasks. In this paper, we introduce *Avalon*, a transaction execution framework for cross-chain dApps that guarantees *complete atomicity* for the first time. *Avalon* achieves this by introducing multiple state layers above the native one to cache state transitions, allowing for efficient management of these state transitions. Most notably, for concurrent cross-chain transactions, *Avalon* resolves not only intra-chain conflicts but also addresses potential inconsistencies between blockchains via a novel *state synchronization* protocol, enabling serializable cross-chain execution. We implement Avalon using smart contracts in Cosmos ecosystem and evaluate its commitment performance, demonstrating acceptable latency and gas consumption even under conflict cases.**

***Index Terms*—blockchain interoperability, atomicity, decentralized application**

I. Introduction

The rising popularity of cross-chain *Decentralized Applications* (dApps) [1], [2] is fueling the demand for blockchain interoperability [3], which aims to facilitate interactions across diverse and heterogeneous blockchains. These cross-chain dApps leverage blockchain smart contracts initially to facilitate simple asset transfers within financial contexts [4], and subsequently extend their functionality to encompass complex application logic via state transitions [2]. This expansion broadens the application scenarios to include domains such as supply chains [5], metaverse environments [6], and *Computing Power Networks* (CPNs) [7].

In popular cross-chain dApps, the application logic typically comprises multiple dependent transactions distributed across distinct blockchains. As stated in the pioneer work Hyperservice [2], the successful execution of the entire application logic necessitates these dependent transactions to be executed sequentially so that the subsequent transaction can obtain inputs from the transactions it depends on, ensuring proper state transitions. For instance, consider a cross-chain dApp applied in a CPN scenario, as depicted in Fig. 1. CPN involves multiple clusters each running an underlying blockchain. The computing task submitted to CPN is decomposed into several dependent subtasks, each representing a blockchain transaction (e.g., tx_1, tx_2, and tx_3), and the execution of a transaction relies on the execution result of the transaction preceding it. These dependent subtasks are then allocated to clusters with appropriate computing resources for execution. To maintain this dependency, a transaction in a blockchain is not executed until it receives the execution result of its preceding transaction via cross-chain communication [8].

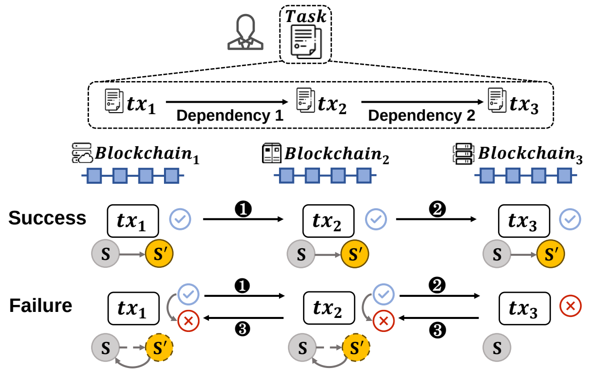


Fig. 1. *Complete atomicity* in cross-chain computing power networks

Atomicity is a critical property in cross-chain transaction execution to ensure the correctness of application logic. Traditionally, atomicity emphasizes the correctness of cross-chain asset transfers or swaps [4], ensuring that these events are either fully completed or all involving parties suffer nearly no financial losses, a concept known as *financial atomicity* [2], [4]. However, for popular cross-chain dApps involving multiple dependent transactions performing state transitions, *financial atomicity* is inadequate and requires an extension. As depicted in Fig. 1, to ensure the correctness of the execution, all dependent transactions must either successfully complete state transitions on their respective blockchains (the successful state transitions $S \rightarrow S'$), or no state transitions occur even if errors arise (maintaining the initial state S). We refer to this property as *complete atomicity*. However, if one of the dependent transactions is aborted midway due to a semantic error or timeouts (e.g., tx_3), transactions preceding the aborted one have already executed on their respective blockchains to complete state transitions, compromising *complete atomicity*. Therefore, the key to achieving *complete atomicity* is ensuring that committed state transitions can be reverted when subsequent transactions fail to execute.

Efforts in blockchain interoperability have centered around cross-chain asset transfers, achieving *financial atomicity* by initiating compensatory transactions to refund spent assets [2]. However, *complete atomicity* poses new challenges to cross-chain executions as complex state transitions across multiple blockchains require efficiently reverting invalid state transitions without compromising the correctness of concurrent transactions. The simple refund approach that compensates transferred assets cannot undo the update effects of complex transitions on the contract states. Another recent approach [9] is to roll back to the checkpointed state [10] established before state transitions and can revert the effects of invalid contract state transitions regardless of how complex the state transitions are. However, the checkpoint-based approach presents two potential drawbacks: i) it requires initiating a new transaction to trigger the rollback of states, which incurs an additional consensus round for the transaction to commit; ii) in the presence of concurrent transactions, rolling back to the checkpoint state compromises the correctness of other concurrent transactions that have read the updated states.

In this paper, we present *Avalon*, a cross-chain execution framework that achieves *complete atomicity* while maintaining seamless integration with existing blockchain infrastructures. *Avalon* operates under the foundational assumption of correct and interoperable underlying blockchains, interconnected via robust cross-chain communication protocols. The core of *Avalon* framework is a layered state commitment structure facilitated by smart contracts. We introduce an initial cache layer, namely the *dirty state layer* for newly generated state transitions, allowing for efficient management of these transitions before their eventual commitment to persistent storage. This design empowers *Avalon* to handle state transitions with agility, enabling seamless reverting or committing operations as required. Upon the successful commitment of all pertinent transactions associated with a given state transition, the corresponding state transitions are seamlessly migrated to the native persistent state layer for permanent storage.

In practical scenarios, processing numerous concurrent transactions across multiple blockchains can lead to conflicts when accessing and modifying identical states cached in the dirty layer. Specifically, assuming concurrent state transitions are organized into an ordered queue in the dirty state layer, committing a state transition directly upon receiving all commitments can lead to potential conflicts. This occurs when a state transition is overwritten by another state transition in the queue that precedes it, thereby compromising *complete atomicity*. To address this issue, we draw inspiration from the concept of *Optimistic Concurrency Control* (OCC) [11] to handle conflicts in each blockchain. Furthermore, to ensure no execution conflicts arise on any blockchain, we introduce a conflict-free *state synchronization* protocol, namely the *prepared state layer* that enables *serializable atomic state transition* across the underlying blockchains. Upon receiving all commitments in the dirty state layer, a vote on whether the transaction is at the top of the dirty state queue is broadcast during the state synchronization phase to notify other blockchains about potential conflicts. If all votes from underlying blockchains confirm a non-conflicting case, the state transition can be safely committed. Cross-chain executions are processed in a serializable order, prioritizing consistency across chains rather than solely focusing on intra-chain transactions.

To illustrate the feasibility and efficiency of *Avalon*, we implement its prototype as cross-chain smart contracts within the Cosmos ecosystem [12]. We evaluate the commitment performance and overhead of *Avalon*, along with its robustness under increased transaction conflict rates. To sum up, we make the following contributions in this paper.

- We elucidate the distinction between *financial atomicity* and *complete atomicity*, as well as the necessity of achieving *complete atomicity* in modern cross-chain dApps.
- We devise an efficient cross-chain execution framework *Avalon*. *Avalon* utilizes the layered state structure to efficiently commit valid state transitions and revert invalid ones to achieve *complete atomicity* even in the case of concurrent transactions.
- We implement *Avalon* within smart contract logic using the Cosmos SDK. The evaluation of *Avalon* demonstrates that it yields acceptable latency and gas consumption in conflict-free and conflict scenarios.

II. Preliminaries

To better understand the requirements to achieve *complete atomicity* of complex cross-chain execution, we present relevant knowledge of blockchain basics and blockchain interoperability schemes. Besides, we extend the classic generic definition of atomicity.

A. State Machine Replication

State Machine Replication (SMR) [13] constructs fault-tolerant services by replicating states among a set of replicas. Each replica receives requests as input and outputs an ever-increasing sequence of requests. Our research focuses on Byzantine SMR tolerating Byzantine faults. Byzantine SMR consists of honest replicas that follow the protocol and Byzantine replicas that deviate arbitrarily and satisfy the following properties.

- *Safety.* If two honest replicas output v and v' at the same sequence number, then $v = v'$.
- *Liveness.* If a client submits a request m, all honest replicas will output m eventually.

A blockchain protocol implements SMR in the form of chained blocks [14], [15]. A typical block contains a set of transactions with application logic, as well as the metadata of consensus and execution. Specifically, once a transaction is included in the block and executed, the state transition it triggers will be recorded in the execution metadata (e.g., structured modified states in each block [16]). Together with the consensus metadata [17], an external operator can efficiently verify the existence of the transaction and the state transition it triggers.

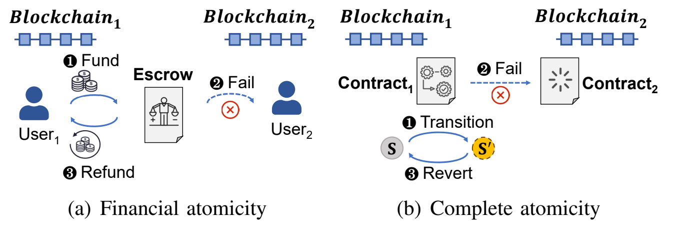


Fig. 2. Comparison of *financial* and *complete atomicity*

B. Cross-chain Communication

Cross-chain communication [18]–[21] delivers authenticated state transitions between any pair of blockchains, crucial for cross-chain dApps. The main component of cross-chain communication is the generation and verification of *finality proof* [22], which indicates the inclusion of a specific transaction in a block. Meanwhile, the communication eventually terminates assuming underlying blockchains are live. Cross-chain communication satisfies the following properties:

- *Authenticity (verifiability of finality proof).* If and only if a transaction tx is finalized on the source blockchain, any replica on the destination blockchain can verify the state transitions corresponding to the committed transaction on the source blockchain.
- *Reliability (delivery of finality proof).* If tx is finalized on the source blockchain, all honest replicas in the destination blockchain will commit the state transition of tx eventually.

Cross-chain communication provides the following interfaces for routing and delivering data across blockchains:

- *Routing:* any process can submit arbitrary data finalized on the source blockchain to the destination blockchain by calling *CCC.route(src, dst, data)*.
- *Delivery:* any process can verify the existence and authenticity of the routed data by calling *CCC.deliver(src, dst, data)*.

C. Atomicity

Classic atomicity [23] ensures either all dependent transactions are executed, or none are applied. If any transaction fails, the system should undo the execution effects of already committed transactions. For cross-chain dApps with complex application logic triggered by smart contracts, the classic atomicity property should encompass a broader range of contract state transitions beyond simple cross-chain asset transfer logic.

Financial atomicity. Prior efforts focus on achieving weaker *financial atomicity* [4], [24], [25] by reverting all token transfers, regardless of state changes. As shown in Fig. 2(a), *financial atomicity* indicates that all involved parties of cross-chain execution suffer nearly no financial loss. However, this results in an incomplete execution trace if some transactions within the entire task have triggered state transitions and are then financially aborted. Cross-chain executions satisfying *financial atomicity* guarantee the following properties:

- *Financial atomicity.* All involved parties experience nearly no financial loss during the execution, regardless of state transitions.
- *Termination.* All involved parties receive the cross-chain execution results eventually.

Complete atomicity. To achieve a more advanced *complete atomicity* property, a blockchain interoperability scheme is expected to trigger no state transitions in the abort case, as depicted in Fig. 2(b). Intuitively, this is impractical since cross-chain communication relies on the inclusion of transactions in consensus to provide verifiability, which inevitably triggers state transitions. We circumvent the state transition on native states through layered state commitments and achieve *complete atomicity* as below:

- *Complete atomicity.* If the application is successfully executed, the state transitions on underlying blockchains are applied. Otherwise, all state transitions are *reverted* or in other words, *aborted*.
- *Termination.* All involved parties receive the cross-chain execution results eventually.

Goal. In this paper, we aim to achieve *complete atomicity* for cross-chain dApp executions and maintain this property even under the existence of concurrent cross-chain executions.

III. System Model

We assume a finite collection of m blockchains denoted as $\Pi_1, \Pi_2, ..., \Pi_m$, each operating as a replicated state machine with a fault-tolerant consensus protocol tolerating Byzantine faults. To model blockchains with certificate-based consensus protocols, we assume the existence of a *Public Key Infrastructure* (PKI) and a secure signature scheme. Note that PKI and cryptography are not strictly required in our design. We assume an adversary can corrupt replicas, coordinate their states, and be either adaptive, strong adaptive, or static according to the underlying consensus. We assume that a network adversary can drop, omit, and reorder messages arbitrarily. Our network timing assumption depends on the underlying consensus protocol and can be synchronous, partial synchronous, or asynchronous [26]. Thanks to our loose assumption on the underlying consensus protocols, a wide range of consensus protocols [14], [15], [27]–[30] can be integrated into our design. The blockchains are heterogeneous, either permissionless or permissioned and each satisfies the formal definition of Byzantine SMR.

A cross-chain application execution $\mathcal{E}$ is indexed by an identification number id, consisting of k dependent transactions, i.e., $\mathcal{E}_{id} = \{tx_i^{id} | \ i = 1, 2, ..., k\}$. Execution does not necessarily cover all underlying m blockchains. For ease of presentation, we assume each transaction tx_i^{id} is submitted to a distinct blockchain Π_i, while our design can be adjusted to the situation where multiple transactions are submitted to one blockchain with slight modifications. We model the transaction dependencies as a *Directed Acyclic Graph* (DAG) denoted as $\mathcal{G} = (V, E)$, where the set of nodes V represents transactions and the set of edges E characterizes the preconditioning requirements among transactions.

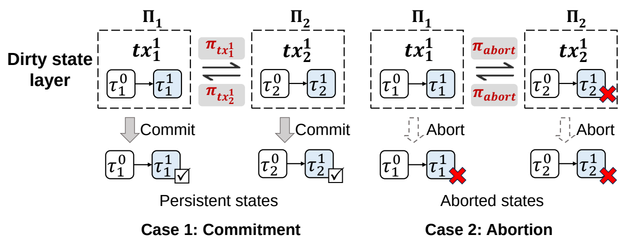


Fig. 3. Overview of *Avalon-Lite*

To model conflicting concurrent executions, we assume the set of concurrent executions accesses an identical state, i.e., some value of a specific smart contract. We use the notation τ_i^j to represent the specific state on blockchain Π_i with its initial state denoted as τ_i^0, where j is a sequence number representing the current state that j previous transactions have modified. Moreover, we define the state transition triggered by tx_i^{id} of an execution $\mathcal{E}_{id}$ as $\tau_i^j \xrightarrow{id} \tau_i^{j+1}$.

To facilitate cross-chain executions with *complete atomicity*, we consider all underlying blockchains to be correct, interoperable, and connected via authenticated network links. These connections allow blockchains to securely exchange the data committed locally within its local consensus with other blockchains. We assume asynchronous network links, ensuring all messages from the source blockchain are eventually delivered. Any secure cross-chain communication scheme can be integrated to construct network links.

IV. Strawman Design

Before diving into the complete design of *Avalon*, we propose a strawman design called *Avalon-Lite* in this section to illustrate the basic idea of achieving *complete atomicity* in cross-chain application execution.

The fundamental challenge in attaining *complete atomicity* in blockchain is to ensure "all-or-nothing" given an append-only ledger. Specifically, consider a cross-chain execution consisting of multiple transactions. In ideal cases, transactions are executed in line according to their dependencies. However, if any transaction aborts, its state transition is not triggered on the corresponding blockchain, while committed transactions on other blockchains are irreversible. Therefore, our core design principle of *Avalon-Lite* is to add a dirty state layer to cache the state transitions before commitment. Dirty states in this layer can be deleted if any transaction is aborted on other blockchains. Only those agreed upon by all participating blockchains can be committed to persistent storage.

We present an overview of *Avalon-Lite* in Fig. 3, which consists of two key components: cross-chain communication for data exchange and a dirty state layer for state caching. To better demonstrate our idea, we keep the illustration at a high level and defer concrete implementation to Section VII. With cross-chain communication, finality proof π of transactions is exchanged between any pair of blockchains. After all transactions are committed or any transaction is dropped according to the abortion rule, the execution successfully terminates. The execution of all transactions is conducted within the dirty state layer, and after the execution terminates, state transitions are either committed or aborted due to timeout or semantic errors.

Algorithm 1: *Avalon-Lite* protocol (for blockchain Π_i)

```
1  Init:
2      $dirty \leftarrow \tau_i^0$;
3      $committed \leftarrow \tau_i^0$;
4      $txVotes \leftarrow \emptyset$;

5  function execute($tx_i^{id}$, $ID$):
6      $id \leftarrow ID.eid$;
       // transition represents the original contract logic
7      $dirty, \pi_{tx_i^{id}} \leftarrow transition(dirty, tx_i^{id})$;
8      for $dst$ in $ID.chains$ do
9          emit $CCC.route(\Pi_i, dst, (\pi_{tx_i^{id}}, ID))$;

10 upon event $CCC.deliver(src, \Pi_i, (\pi_{tx_{src}^{id}}, ID))$ do:
11     $id \leftarrow ID.eid$;
12     $txVotes[id] \leftarrow txVotes[id] \cup \{src\}$;
13     if $|txVotes[id]| == |ID.chains|$ then
14         $committed \leftarrow dirty$;

15 function abort($ID, \pi_{abort}$):
16     if verify$(ID, \pi_{abort}) == true$ then
17         $id \leftarrow ID.eid$;
18         $txVotes[id] \leftarrow \emptyset$;
19         $dirty \leftarrow \tau_i^0$;
```

We demonstrate the instantiation of *Avalon-Lite* with *CCC* primitive in smart contract logic in Alg. 1. For simplicity, we omit the details of transaction logic and abortion rule, focusing on the design to ensure *complete atomicity*. The protocol begins with an initial state τ_i^0 and an empty map $txVotes$ storing finality proofs. Transactions are confirmed through SMR and executed by calling the *execute* interface, which accepts ID storing the list of participating blockchains $ID.chains$ and the execution index $ID.eid$ for execution $\mathcal{E}_{id}$. After the state transition (line 7), *CCC* is emitted with the finality proof $\pi_{tx_i^{id}}$ to all participating blockchains (lines 8-9). Upon delivering all finality proofs, the state transition is committed (lines 11-14). If the *abort* interface is called with a valid proof, the dirty state is reverted, and $txVotes$ storing commitments of executed transactions is cleared (lines 16-19).

A major concern is that isolated abortion operations on each blockchain can potentially lead to inconsistency, as timeouts triggered according to the local timestamps of replicas may differ, causing some blockchains to commit an execution while others abort. To address this, we borrow the idea in [31] of introducing a synchronized clock that can be constructed by building a secure upper blockchain (e.g., NSB blockchain in [2]) over m underlying blockchains. Each transaction is submitted to its corresponding blockchain with a timestamp acquired through the upper-layer blockchain. A timeout is triggered when the number of blocks generated on the upper blockchain exceeds a pre-determined value. An abortion is verified true if proof shows that not all transactions are published

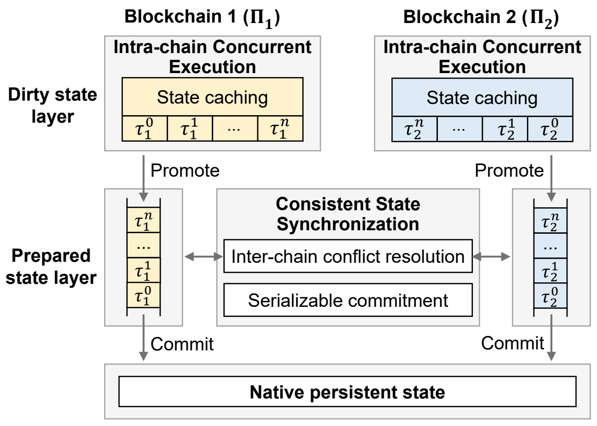


Fig. 4. Architecture of *Avalon*

on the upper blockchain, using the Merkle nil proof stated in [32] to prove the nonexistence of specific transactions.

V. Avalon Design

In this section, we elaborate on an enhanced variant, *Avalon*, based on our strawman design to support concurrent executions across multiple blockchains. The overall architecture of *Avalon* is depicted in Fig. 4.

A. Overview

In real-world scenarios, multiple concurrent transactions can access an identical state. The strawman design fails in such cases due to its lack of concurrency control. Specifically, the state transition of committed states is likely to be rewritten by a subsequently committed transaction. We borrow the idea of OCC to resolve conflicts that may occur in the dirty state layer. Upon receiving all finality proofs of a specific transaction, given a state queue outputted by concurrent executions, the state transition of the transaction is aborted if any state transition previous to it in the queue remains uncommitted.

Although it has been widely studied to ensure the correctness of concurrent execution within a single blockchain [33]–[35], it is non-trivial to enable concurrent execution in a multichain ecosystem. As blockchains exhibit different concurrency levels, the state transition of a specific cross-chain execution may not consistently commit or abort, resulting in a violation of atomicity. As such, our main challenge is to coordinate the state transitions among all underlying blockchains. We propose a state synchronization protocol that requires blockchains to exchange their votes on whether to commit or abort within a single round of communication, namely the *prepared state layer*, as shown in Fig. 4. Moreover, we introduce and fulfill the serializability property in cross-chain concurrent execution with consistent state synchronization. This property focuses on ensuring that concurrent executions across distinct blockchains can be completed in a serializable order rather than merely concurrent executions in a single blockchain.

B. Intra-chain Concurrent Execution

To clarify the problem raised by concurrent executions, we consider a scenario of *Avalon-Lite* handling a list of

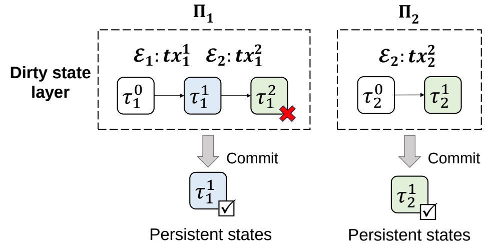


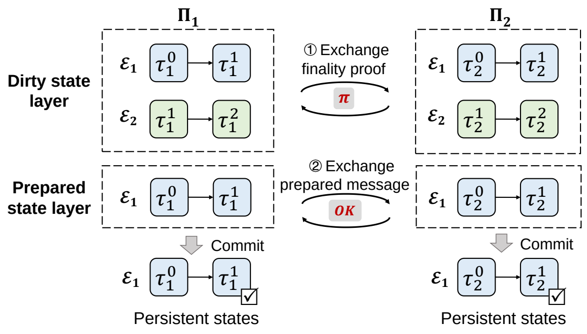
Fig. 5. Inter-chain conflicts

n concurrent uncommitted transactions $tx_i^1, tx_i^2, ..., tx_i^n$ that trigger multiple state transitions in the dirty state layer on Π_i. These uncommitted transactions read the same initial state denoted as τ_i^0, leading to concurrent conflicts, i.e., the successfully committed state transition of a transaction would cause stale reads of other concurrent transactions. To ensure the correctness of execution in the dirty state layer, we inherit the idea of OCC to resolve conflicts. Specifically, we only commit the first state transition and abort all subsequent state transitions in the dirty layer. However, OCC leads to a high abort rate that limits the performance of cross-chain execution. To reduce such aborts, we make an optimization that permits uncommitted states to be visible to all concurrent transactions and commits them according to the order in which they are executed. Note that the transactions are executed and appear in the dirty state layer in the consensus order as SMR guarantees *total order* implicitly in its *safety* property.

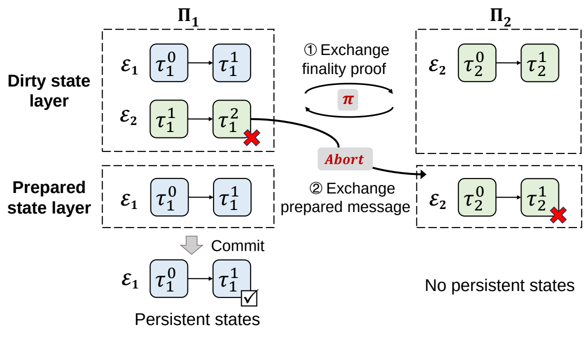
Specifically, we utilize a queue to cache concurrent state transitions in the dirty state layer (i.e., τ_i^0, τ_i^1, ..., τ_i^n). A new state transition is added to the tail of the queue. Given an initial state τ_i^0 and n concurrent transactions, τ_i^j is based on τ_i^{j-1} (for j in $1, ..., n$) and the resulting state transitions are $\tau_i^1, \tau_i^2, ..., \tau_i^n$. If dirty states are committed in the order of their executions, they can all be committed with a trace of $\tau_i^1, \tau_i^2, ..., \tau_i^n$. To maintain this commitment order, we stipulate that one state transition can be successfully committed only if it receives sufficient finality proofs and is located at the top of the dirty state queue. Otherwise, it is aborted, thereby ensuring that this state transition cannot be rewritten by a previous transaction that commits after it.

C. Consistent State Synchronization

Inter-chain conflict resolution. Intra-chain concurrency control ensures transactions are aborted in case there exists any potential conflict in each blockchain. However, inter-chain conflicts still arise since transactions are executed in an isolated manner. Specifically, the state transitions belonging to a specific execution may be committed on some blockchains and aborted on others, violating *complete atomicity*. We present an example of inter-chain conflicts in Fig. 5, where execution $\mathcal{E}_2$ can be committed on Π_2 but is aborted on Π_1. Upon blockchain Π_1 receiving sufficient finality proofs for state transition τ_1^2 of $\mathcal{E}_2$, τ_1^2 is not the top element of the dirty state queue and is thus aborted, maintaining a persistent state τ_1^1 triggered by execution $\mathcal{E}_1$, which is inconsistent with the state τ_2^1 triggered by execution $\mathcal{E}_2$ in Π_2.



(a) Commitment



(b) Abortion

Fig. 6. Inter-chain conflict resolution

To resolve inter-chain conflicts, we introduce an intermediate prepared state layer to coordinate commitments by synchronizing the intra-chain execution results. Specifically, after receiving sufficient finality proofs, rather than directly committing, each blockchain broadcasts a vote on whether the intra-chain execution is correct (i.e., confirmed at the top of the queue) via *CCC*. The state transition can be successfully committed upon receiving sufficient votes for correct execution and is aborted if any vote indicating the intra-chain execution fails is received.

Fig. 6 presents commitment and abortion cases of state synchronization. Executions $\mathcal{E}_1$ and $\mathcal{E}_2$ are modeled as concurrent executions, where $\mathcal{E}_1$ involves Π_1 and some other blockchains not included in this figure while $\mathcal{E}_2$ involves both blockchains. Upon receiving sufficient finality proofs π, a vote on the correct execution of $\mathcal{E}_1$ formalized as an OK message is broadcast to other blockchains. Upon receiving all OK messages from underlying blockchains, the states are committed to persistent storage on each blockchain. For the abortion case illustrated in Fig. 6(b), if $\mathcal{E}_2$ gathers sufficient finality proofs before $\mathcal{E}_1$, a vote indicating incorrect commitment order of $\mathcal{E}_2$ formalized as an $Abort$ message on Π_1 is broadcast to Π_2. Upon receiving an $Abort$ message from any blockchain, the transaction corresponding to the message is aborted, thereby ensuring $\mathcal{E}_2$ is aborted on both blockchains.

We present a detailed description of the state synchronization protocol in Alg. 2. We assume there exist no concurrent uncommitted state transitions in the prepared state layer and defer the discussion of the concurrent case to Alg. 3. To enable

Algorithm 2: Inter-chain conflict resolution

```
1  Init:
2      $dQueue \leftarrow [\tau_i^0]$;
3      $txVotes(dirty), txVotes(prep) \leftarrow \emptyset$;
4      $sMap \leftarrow \emptyset$;
5      $prepared, committed \leftarrow \tau_i^0$;

6  function execute($tx_i^{id}$, $ID$):
7      $basestate \leftarrow dQueue.rear()$;
8      $s \leftarrow transition(basestate, tx_i^{id})$;
9      $dQueue.enqueue(s)$;
10     $sMap[s] \leftarrow ID$;
       // same as lines 8-9 in Alg. 1

11 upon event $CCC.deliver(src, \mathit{\Pi}_i, (\pi_{tx_{src}^{id}}, ID))$ do:
12     $id \leftarrow ID.eid$;
13     $txVotes(dirty)[id] \leftarrow txVotes(dirty)[id] \cup \{src\}$;
14     if $|txVotes(dirty)[id]| == |ID.chains|$ then
15         if $sMap[dQueue.peek()] != ID$ then
               // remove $s$ and subsequent queue elements
16             for $dst$ in $ID.chains$ do
17                 emit $CCC.route(\mathit{\Pi}_i, dst, (Abort, ID))$;
18         else
19             $s \leftarrow dQueue.dequeue()$;
20             $prepared \leftarrow s$;
21             for $dst$ in $ID.chains$ do
22                 emit $CCC.route(\mathit{\Pi}_i, dst, (OK, ID))$;

23 upon event $CCC.deliver(src, \mathit{\Pi}_i, (OK, ID))$ do:
24     $id \leftarrow ID.eid$;
25     $txVotes(prep)[id] \leftarrow txVotes(prep)[id] \cup \{src\}$;
26     if $|txVotes(prep)[id]| == |ID.chains|$ then
27         $committed \leftarrow prepared$;

28 upon event $CCC.deliver(src, \mathit{\Pi}_i, (Abort, ID))$ do:
29     while $dQueue.isEmpty() == false$:
30         $s \leftarrow dQueue.dequeue()$;
31         for $dst$ in $sMap[s].chains$ do
32             emit $CCC.route(\mathit{\Pi}_i, dst, (Abort, sMap[s]))$;
33     $dQueue \leftarrow \emptyset$;
```

reading from uncommitted states, we modify the state queue to provide a *rear* interface that retrieves the tail element. The protocol is set up with a dirty state queue and two empty maps storing commitment votes in the dirty and prepared state layer. The prepared and committed state is initialized as τ_i^0. We maintain a mapping structure $sMap$ that maps state transition in the state queue to the execution $\mathcal{E}_{id}$. A state transition based on the latest updated state is added to the tail of the queue (lines 8-10). After gathering all commitments in the dirty state layer, the state transition is checked to be aborted or promoted to the prepared state layer (lines 13-22). By checking the top element of the dirty state queue is not triggered by $\mathcal{E}_{id}$, the state transition is aborted. All subsequent state transitions in the queue are also removed since they are executed based on the state transition of $\mathcal{E}_{id}$ (lines 15-17). Otherwise, the state transition is deleted from the queue and promoted to the prepared state (lines 19-22). Upon delivering all OK messages for $\mathcal{E}_{id}$, the state transition becomes persistent (lines 25-27). If any $Abort$ message is received, the prepared state is reverted, and the dirty state queue is cleared with corresponding $Abort$ messages sent to other chains (lines 29-33).

Serializable commitment. For ease of presentation, the state synchronization stage discussed above focuses on solely resolving inter-chain conflicts arising from the dirty state layer. Similarly, concurrent state transitions in the prepared state layer also lead to cascading inter-chain conflicts, akin to the case illustrated in Fig. 6(b). A naive solution to completely eliminating inter-chain conflicts is introducing another settlement layer, yet this results in an infinite exchange of messages. To solve this problem, we introduce *serializable commitment* to tackle cascading inter-chain conflicts with slight modifications in the state synchronization stage. Specifically, we stipulate a serial commitment manner in the prepared state layer to ensure no conflicts arise. We also utilize a state queue to organize the caching of concurrent state transitions in the prepared state layer. Upon the correct execution of the intra-chain execution phase, rather than directly broadcasting an OK message, the state transition is pushed to the tail of the prepared state queue. The OK message is broadcast only when the corresponding state transition reaches the top of the prepared state queue, i.e., given concurrent states in the queue, their OK messages are broadcast in a serial order.

Alg. 3 outlines the design for serially emitting OK messages to other blockchains to avoid cascading conflicts. We set up the prepared state with a $pQueue$ to store all concurrent state transitions. After a state transition is promoted to the prepared state layer, an OK message is routed to other blockchains if the state is at the peak of the $pQueue$ (lines 3-6). Upon delivering all OK messages for $\mathcal{E}_{id}$ from other blockchains, the corresponding state is committed to the persistent storage by calling the $dequeue$ interface (line 10). After commitment, the OK message for the next element is routed to other blockchains (lines 11-15). If an $Abort$ message is received, all elements in both queues are aborted since those state transitions are based on the aborted state (line 16).

Another problem incurred by serial execution in the prepared state layer is deadlock. Considering two concurrent executions denoted as $\mathcal{E}_1$ and $\mathcal{E}_2$ executed in a different order in distinct blockchains, some blockchains first lock on $\mathcal{E}_1$ while others first lock on $\mathcal{E}_2$, leading to infinite waiting since no $Abort$ message will be emitted. To resolve the deadlock, blockchains exchange the information of the top elements in their prepared state queue. Upon receiving an OK message, blockchains first check whether its corresponding state transition exists in the prepared state queue but is not the top element. If so, a tentative abortion message *t-Abort* with the top element is sent back to the source of the OK message. If the element in the *t-Abort* message is also not the top element in the source blockchain, indicating a deadlock, $Abort$ messages are broadcast by the source chain to abort both transactions.

Algorithm 3: Serializable commitment

```
1  Init:
2      $pQueue \leftarrow [\tau_i^0]$;
       // same as lines 2-5 in Alg. 2

       //same as lines 6-19 in Alg. 2
3      $pQueue.enqueue(s)$;
4      if $pQueue.peek() == s$ then
5          for $dst\ in\ ID.chains$ do
6              emit $CCC.route(\Pi_i, dst, (OK, ID))$;

7  upon event $CCC.deliver(src, \Pi_i, (OK, ID))$do:
8      $id \leftarrow ID.eid$;
9      $txVotes(prep)[id] \leftarrow txVotes(prep)[id] \cup \{src\}$;
10     if $|txVotes(prep)[id]| == |ID.chains|$ then
11         $committed \leftarrow pQueue.dequeue()$;
12         if $pQueue.isEmpty() == false$ then
13             $ID \leftarrow sMap[pQueue.peek()]$;
14             for $dst\ in\ ID.chains$ do
15                 emit $CCC.route(\Pi_i, dst, (OK, ID))$;

16 upon event $CCC.deliver(src, \Pi_i, (Abort, ID))$do:
       // abort all elements in $dQueue$ and $pQueue$ as
         lines 29-33 in Alg. 2
```

Avalon achieves the serializability property in terms of concurrent cross-chain executions, referred to as *serializable atomic state transition*. Informally, given multiple concurrent executions, the property ensures all participating blockchains commit the state transitions in an identical serializable order. We expect to formalize this property in Section VI to enrich the research of classic serializability [36], [37] that mainly focuses on a single replicated state machine.

VI. Correctness Analysis

Correctness analysis covers two main aspects: *complete atomicity* and termination. To better demonstrate atomicity, we define *atomic state transition* for any pair of states.

- *Atomic state transition.* Given any pair of states τ_i^j and $\tau_{i'}^{j'}$ accessed by execution $\mathcal{E}_{id}$, if either both state transition $\tau_i^j \xrightarrow{id} \tau_i^{j+1}$ and $\tau_{i'}^{j'} \xrightarrow{id} \tau_{i'}^{j'+1}$ are eventually applied or neither of them is applied, atomic state transition is triggered by $\mathcal{E}_{id}$.

The above definition closely resembles our informal definition of *complete atomicity* in Section II-C. Moreover, with the above definition that focuses on one single execution, we can extend it to *serializable atomic state transition* where multiple concurrent executions exist.

- *Serializable atomic state transition.* Given any pair of states τ_i^j and $\tau_{i'}^{j'}$ accessed by concurrent executions $\mathcal{E} \leftarrow \{\mathcal{E}_a, \mathcal{E}_{a+1}, ..., \mathcal{E}_{a'}\}$, if all concurrent executions eventually trigger atomic state transition in identical order as if they are executed sequentially, serializable atomic state transition is triggered by $\mathcal{E}$.

The *complete atomicity* property states that with a finite input set of executions, all participating blockchains trigger serializable atomic state transition. Informally, a subset $\mathcal{E}' \subset \mathcal{E}$ of concurrent executions modify the states in identical order and the rest $\mathcal{E} \cap (\neg\mathcal{E}')$ is aborted. The termination property ensures all possible state transitions end within a predefined finite time. To ensure the above properties, we assume each blockchain is a correct replicated state machine. The *CCC* primitive enables interoperability among blockchains and satisfies authenticity and reliability. We defer detailed proofs to Section A of the Appendix.

Theorem 1 (Complete atomicity). Given a finite set of concurrent executions $\mathcal{E} \leftarrow \{\mathcal{E}_a, \mathcal{E}_{a+1}, ..., \mathcal{E}_{a'}\}$ on m distinct blockchains $\mathit{\Pi}_1, \mathit{\Pi}_1, ..., \mathit{\Pi}_m$ input in arbitrary orders, for the states of any pair of blockchains, $\mathcal{E}$ eventually triggers serializable atomic state transition.

Theorem 2 (Termination). Given a finite set of concurrent executions $\mathcal{E} \leftarrow \{\mathcal{E}_a, \mathcal{E}_{a+1}, ..., \mathcal{E}_{a'}\}$ on m distinct blockchains $\mathit{\Pi}_1, \mathit{\Pi}_2, ..., \mathit{\Pi}_m$ input in arbitrary orders, all transactions in $\mathcal{E}$ are committed or aborted within a finite time period.

VII. Implementation

We implement *Avalon* within Cosmos ecosystem [12] that instantiates multiple interoperable blockchains natively. The *Avalon* protocol is implemented within smart contract logic written in Rust with CosmWasm [38]. We use IBC [21] of the Cosmos-SDK to enable cross-chain communication.

Before diving into the details, we first briefly discuss the feasibility of facilitating *Avalon* with Cosmos. Cosmos-SDK enables developers to build interoperable blockchains without implementing basic blockchain functionalities like a fault-tolerant consensus protocol or transaction execution logic, which motivates us to deploy cross-chain execution logic upon multiple Cosmos blockchains (i.e., zones). Smart contract logic is implemented with the CosmWasm library that allows developers to define arbitrary functions. The IBC module consists of a network transport layer maintained by IBC clients that enables authenticated connections between blockchains to exchange data and an application layer that defines the sending and receiving logic on the server and receiver sides.

We deploy m interoperable blockchains in the ecosystem to instantiate *Avalon*. On each blockchain, the protocol consists of an *Avalon* smart contract as the proxy module and dApp contracts as the execution module, as shown in Fig. 7. The proxy module implements the *Avalon* protocol, while the execution module handles application-specific logic for cross-chain execution. We consider m transactions, each executed on a different blockchain. Note that *Avalon* can hold any number of transactions and an execution does not necessarily cover all blockchains. During execution, each transaction is submitted to the proxy module of each underlying blockchain. The proxy module first calls the application contract for execution and triggers state transitions in the dirty state layer. After that,

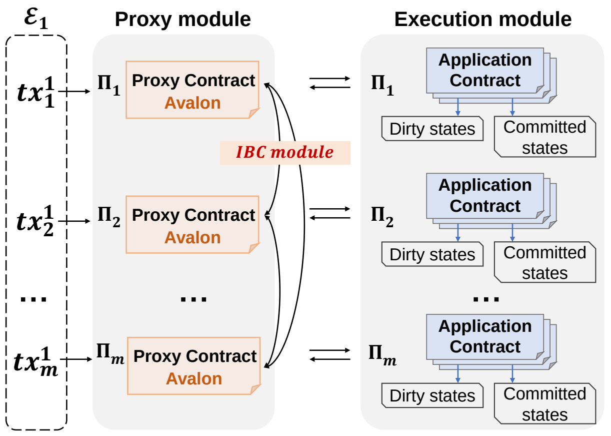


Fig. 7. Implementation of *Avalon*

the state transition is relayed to other blockchains along with its proof. If a state transition is committed, the application contract is called by the *Avalon* contract to modify its state.

VIII. Evaluation

In this section, we evaluate the prototype of *Avalon* in terms of its performance scalability and its resilience to conflicts. Our experimental results answer the following questions:

(1) How is the system scalability in terms of the number of underlying blockchains?
(2) How is the gas cost and latency performance of *Avalon*?
(3) What is the impact of the increased concurrency level on the performance of the system?

Experimental setup. We evaluate *Avalon* through experiments on AWS. We deploy the testbed on the ECS.m5d.xlarge instance with 4vCPUs and 16 GB memory. Concretely, we deploy a varying number of Cosmos blockchains using the wasmd[1] library which is implemented with Tendermint [39] for consensus and Leveldb[2] for persistent storage. Each pair of blockchains is connected via a relayer instantiated with the relayer[3] library. As for the experimental settings, we set the number of blockchains as 3, 6, and 9 to evaluate system scalability. Each blockchain is equipped with a client that continuously submits transactions to initiate executions.

Metrics and benchmark. We evaluate the performance of *Avalon* in terms of latency, gas cost with and without conflicts, and abortion rate under conflicts. To remove noise from the execution dependency graph, we set the dependency in a simple sequential manner where an execution result is only routed to one subsequent blockchain until the blockchain with the largest index number is reached. Our benchmark without conflicts consists of a single execution initiated by sending a transaction to the head of the dependency. As for the benchmark with conflicts, we increase the input concurrency level by steadily shortening the input interval of transactions. Moreover, our interest also includes the abortion rate of the system when the orders of concurrent execution input to the

[1]https://github.com/CosmWasm/wasmd
[2]https://github.com/google/leveldb
[3]https://github.com/cosmos/relayer

TABLE I
PERFORMANCE OF *Avalon* WITHOUT CONFLICTS

m	# of IBC	Gas cost	Latency
3	28	5.57M	15.1s
6	130	26.23M	34.7s
9	304	59.67M	59.2s

underlying blockchains are arbitrary and inconsistent. Inspired by a recent work [40], we inject delays equally distributed from 0 to 5 seconds of the client submitting transactions to generate inconsistent input orders. The latency of *Avalon* indicates the time elapsed from the execution of the first transaction in the dirty state layer to the commitment of all transactions involved. To reduce the impact of experimental errors, each set of experiments is repeated three times. To illustrate the average performance and the deviation of multiple runs, we equip each point in the figure with an error bar.

A. System Performance Without Conflicts

In this experiment, we evaluate the performance of *Avalon* in terms of gas cost, number of IBC messages, and latency without conflicts. As illustrated in Table I, the gas cost and the number of IBC messages of the prototype exhibit quadratic complexity with respect to the number of blockchains, which corresponds to our design. The gas cost of the protocol mainly originates from the exchange of IBC messages since they demonstrate a similar trend as the number of blockchains increases. Based on the gas price [41] of the Cosmos blockchain at the time of writing, the prototype can commit a message with 60M gas usage that approximately costs \$2.8 USD.

The protocol delivers a latency of tens of seconds, which stems from two rounds to exchange the dirty and prepared votes among all participating blockchains. Specifically, during each round, the IBC clients spend one round trip to relay messages and several additional round trips to update their status and broadcast acknowledgments, with each round trip normally taking 2 seconds. Moreover, the latency scales almost linearly as the number of blockchains increases. This linearity is due to the fact that relayers take a longer time to update their status and broadcast acknowledgments under a more extensive workload, resulting in increased latency. Notably, if equipped with a more efficient relayer, which is of independent interest to our approach, *Avalon* is expected to deliver sublinear latency as the number of rounds needed for commitment is constant according to the protocol design.

B. Performance With Conflicts

In this set of experiments, we evaluate performance with conflicts. Our experimental metrics include the latency, gas cost, and abortion rate as concurrency and disorder arise. To model concurrency, we consider a set of concurrent executions with clients continuously submitting transactions. We vary the time interval of submitting a transaction to model different concurrency levels. To model disorders, we inject delays to the clients submitting transactions. Note that the maximum delay is larger than two round trips of the IBC clients and is sufficient to simulate input disorders. For latency, we focus on executions that are successfully committed. Therefore, disorders have little impact on latency performance as disorders are expected only to introduce concurrency chaos but not impact any single execution. We evaluate the gas cost and abortion rate with and without disorders.

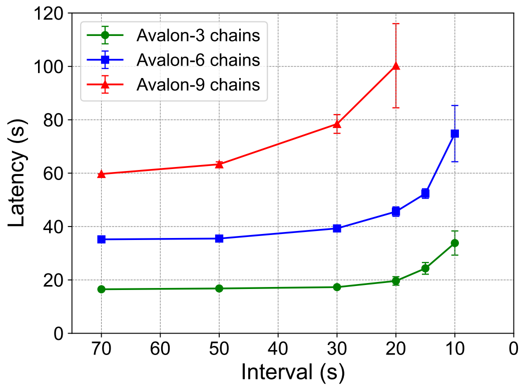


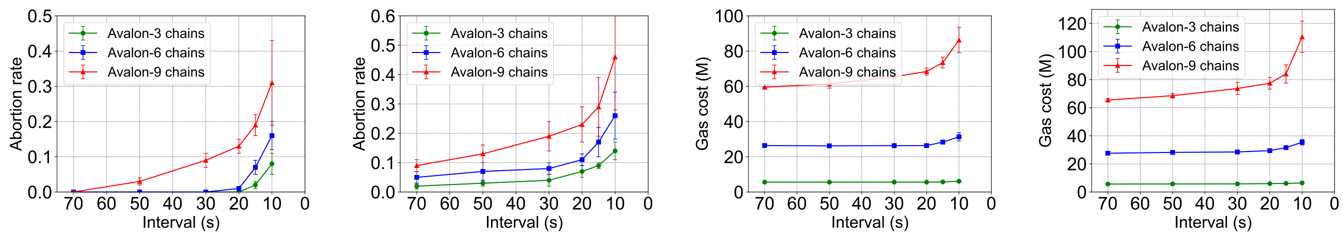
Fig. 8. Latency under conflicts

Fig. 8 demonstrates the latency of *Avalon* under various concurrency levels. When the time interval is large enough to cover multiple IBC round trips, indicating a low concurrency level, transactions hardly conflict with each other, and the latency is almost identical to that without conflicts. Under a high concurrency level, the latency rises significantly. We investigate that this is due to the weak ability of the relayers to handle concurrent transactions. Specifically, a relayer can only update its state after receiving an acknowledgment message corresponding to a previously relayed message. By progressively reducing the time interval, the latency eventually saturates, indicating the system has reached its peak load.

Fig. 9 depicts the system performance in terms of gas cost per successful execution and the abortion rate of executions. We vary the levels of concurrency and disorders and evaluate the robustness of the system. Similar to the latency performance, if the time interval is large enough to cover several IBC round trips and no delay injection is applied to the clients, the system delivers comparable performance to that without conflicts. With the level of either concurrency or disorders increases, the gas cost per successful execution and abortion rate increase but the system still delivers acceptable performance under both high concurrency and disorders, with at most 50% degradation.

IX. RELATED WORK

Blockchain interoperability technology [3] plays a crucial role in breaking down barriers between distinct blockchains and facilitates seamless integration. Current blockchain interoperability efforts focus on the following two aspects: i) cross-chain bridge, which enables arbitrary data to be efficiently transmitted across blockchains, and ii) atomicity of cross-chain transaction execution, which ensures the correctness and completeness of the entire cross-chain dApp logic. Below, we



(a) Abortion rate without disorders (b) Abortion rate with disorders (c) Gas cost without disorders (d) Gas cost with disorders

Fig. 9. Abortion rate and gas cost under conflicts

TABLE II
COMPARISON OF *Avalon* WITH STATE-OF-THE-ART PROTOCOLS

System	Financial atomicity	Complete atomicity	Conflict resolution	Serializability	Security
He-HTLC [24]	✓	×	N/A	N/A	✓
HyperService [2]	✓	×	N/A	N/A	✓
Lu et al. [9]	✓	△	Serial locks	✓	DoS
***Avalon* (our work)**	✓	✓	OCC + state sync	✓	✓

"✓": fully meet this property; "×": cannot meet this property; "△": meet this property at the cost of security

illustrate relevant works on these two aspects and detail the comparison of *Avalon* with approaches on atomicity.

A. Cross-chain Bridges

Cross-chain bridges can be classified into two categories based on their trust assumptions: committee-based bridges and client-based bridges. Committee-based bridges [19], [20], [42] assume a static committee eligible to vote for the transmitted data, and data with a majority of votes are considered valid. Client-based bridges [21], [25], [43], [44] eliminate the trusted committee and take the approach of seamlessly verifying the transmitted data with consensus metadata forwarded by cross-chain clients. A recent work [18] leverages the security of light clients and the simplicity of intermediate committees via an ultra-light node mode.

Several works [2], [9], [45], [46] have attempted to employ cross-chain bridges to construct a multi-chain world, of which two very recent works [45], [46] secure the states of one blockchain on multiple blockchains through replication. To sum up, cross-chain bridges provide the basis for trusted cross-chain communication in blockchain interoperability, which is orthogonal to our work.

B. Atomicity

Table II compares *Avalon* with relevant works on atomicity. Prior efforts [2], [24], [47]–[50] ensure *financial atomicity* with a refund mechanism. For example, HyperService [2] constructs an insurance contract containing sufficient funds to refund the coins spent by executed transactions financially. However, these works fail to fulfill *complete atomicity* as they cannot roll back contract state updates. In terms of concurrency control that covers the aspects of conflict resolution and execution serializability, He-HTLC [24] focuses on secure pairwise token transfers, and Hyperservice assumes the execution of a single set of dependent transactions. Both works are proven secure in their respective models.

The closest work to ours recently proposed by Lu et al. [9] attempts to achieve *complete atomicity* with cached checkpoints to revert invalid state transitions, adding one additional round of consensus to perform state reverting. Besides, it relies on a two-phase commit protocol orchestrated by a coordinator to avoid conflicts, which requires locking states on the respective blockchains before state commitment. However, the overhead of serial locks is unclear, and the system can only process one transaction at a time, performing poorly with concurrent transactions. In contrast, our serial paradigm within the prepared state layer limits concurrency but allows concurrent transactions to be committed.

In contrast, *Avalon* fulfills *complete atomicity* with efficient state commitment based on a layered state structure, removing invalid state transitions on the top state layers before commitment. Besides, instead of using the pessimistic lock, *Avalon* follows the spirit of OCC and devises a state synchronization protocol to efficiently coordinate executions on underlying blockchains for execution serializability.

X. CONCLUSION

In this paper, we address the limitations of classic *financial atomicity* in cross-chain dApps by introducing the concept of *complete atomicity*. Our proposed *Avalon*, is a transaction execution framework tailored for cross-chain interoperability, ensuring no state transitions occur in case of failed executions. Leveraging a multi-tier state layer and a state synchronization protocol, *Avalon* guarantees *complete atomicity* and correctness in concurrent cross-chain executions. Our formalization and proof of *Avalon's* layered commitments provide robust assurances of its reliability. *Avalon's* adaptability to different consensus protocols further enhances its practicality. Evaluation results underscore *Avalon's* practical viability and efficacy in handling complex cross-chain scenarios.

ACKNOWLEDGMENT

This work was supported by National Key Research and Development Program of China under Grant No. 2021YFB2700700. Shijie Zhang and Jiang Xiao are the corresponding authors.

References

[1] K. Wu, Y. Ma, G. Huang, and X. Liu, "A first look at blockchain-based decentralized applications," *Software: Practice and Experience*, vol. 51, no. 10, pp. 2033–2050, 2021.

[2] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, and Y.-C. Hu, "Hyperservice: Interoperability and programmability across heterogeneous blockchains," in *Proceedings of the 26th ACM SIGSAC conference on computer and communications security (CCS'19)*. ACM, 2019, pp. 549–566.

[3] K. Ren, N.-M. Ho, D. Loghin, T.-T. Nguyen, B. C. Ooi, Q.-T. Ta, and F. Zhu, "Interoperability in blockchain: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12 750–12 769, 2023.

[4] M. Herlihy, "Atomic cross-chain swaps," in *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC'18)*. ACM, 2018, pp. 245–254.

[5] R. Han, J. Xiao, X. Dai, S. Zhang, Y. Sun, B. Li, and H. Jin, "Vassago: Efficient and authenticated provenance query on multiple blockchains," in *Proceedings of the 40th International Symposium on Reliable Distributed Systems (SRDS'21)*. IEEE, 2021, pp. 132–142.

[6] Y. Wang, Z. Su, N. Zhang, R. Xing, D. Liu, T. H. Luan, and X. Shen, "A survey on metaverse: Fundamentals, security, and privacy," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 319–352, 2022.

[7] X. Tang, C. Cao, Y. Wang, S. Zhang, Y. Liu, M. Li, and T. He, "Computing power network: The architecture of convergence of computing and networking towards 6g requirement," *China communications*, vol. 18, no. 2, pp. 175–185, 2021.

[8] P. Han, Z. Yan, W. Ding, S. Fei, and Z. Wan, "A survey on cross-chain technologies," *Distributed Ledger Technologies: Research and Practice*, vol. 2, no. 2, pp. 1–30, 2023.

[9] H. Lu, A. Jajoo, and K. S. Namjoshi, "Atomicity and abstraction for cross-blockchain interactions," *arXiv preprint arXiv:2403.07248*, 2024.

[10] D. Karakostas and A. Kiayias, "Securing proof-of-work ledgers via checkpointing," in *Proceedings of the 4th IEEE International Conference on Blockchain and Cryptocurrency (ICBC'21)*. IEEE, 2021, pp. 1–5.

[11] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, 1981.

[12] J. Kwon and E. Buchman, "Cosmos whitepaper," *A Network of Distributed Ledgers*, vol. 27, pp. 1–32, 2019.

[13] L. Lamport and M. Massa, "Cheap paxos," in *Proceedings of the 5th International Conference on Dependable Systems and Networks (DSN'04)*. IEEE, 2004, pp. 307–314.

[14] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[15] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*. ACM, 2019, pp. 347–356.

[16] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[17] J. Xu, C. Wang, and X. Jia, "A survey of blockchain consensus protocols," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–35, 2023.

[18] "layerzero bridge," Layerzero. [Online]. Available: https://layerzero.network/

[19] "Polynetwork bridge," Polynetwork. [Online]. Available: https://poly.network/

[20] "Wormhole bridge," Wormhole. [Online]. Available: https://wormhole.com/

[21] "Cosmos ibc," Cosmos. [Online]. Available: https://github.com/cosmos/ibc

[22] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[23] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: Past, present, and future trends," *ACM Computing Surveys*, vol. 54, no. 8, pp. 1–41, 2021.

[24] S. Wadhwa, J. Stöter, F. Zhang, and K. Nayak, "He-htlc: Revisiting incentives in htlc," in *Proceedings of the 30th Network and Distributed System Security Symposium (NDSS'23)*. ISOC, 2023.

[25] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, "zkbridge: Trustless cross-chain bridges made practical," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*. ACM, 2022, pp. 3003–3017.

[26] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[27] D. Malkhi and K. Nayak, "Hotstuff-2: Optimal two-phase responsive bft," *Cryptology ePrint Archive*, 2023.

[28] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*. ACM, 2022, pp. 2705–2718.

[29] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*. ACM, 2019, pp. 337–346.

[30] D. Malkhi, A. Momose, and L. Ren, "Towards practical sleepy bft," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*. ACM, 2023, pp. 490–503.

[31] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 933–940, 1987.

[32] J.-Y. Kim, J. Lee, Y. Koo, S. Park, and S.-M. Moon, "Ethanos: efficient bootstrapping for full nodes on account-based blockchain," in *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*. ACM, 2021, pp. 99–113.

[33] J. Xiao, S. Zhang, Z. Zhang, B. Li, X. Dai, and H. Jin, "Nezha: Exploiting concurrency for transaction processing in dag-based blockchains," in *Proceedings of the IEEE 42nd International Conference on Distributed Computing Systems (ICDCS'22)*. IEEE, 2022, pp. 269–279.

[34] P. Garamvölgyi, Y. Liu, D. Zhou, F. Long, and M. Wu, "Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts," in *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. ACM/IEEE, 2022, pp. 2315–2326.

[35] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, "Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'23)*. ACM, 2023, pp. 232–244.

[36] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM*, vol. 26, no. 4, pp. 631–653, 1979.

[37] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev, "Serializability for eventual consistency: criterion, analysis, and applications," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*. ACM, 2017, pp. 458–472.

[38] "Cosmwasm," Cosmos. [Online]. Available: https://github.com/CosmWasm/cosmwasm

[39] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.

[40] X. Dai, B. Zhang, H. Jin, and L. Ren, "Parbft: Faster asynchronous bft consensus with a parallel optimistic path," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*. ACM, 2023, pp. 504–518.

[41] "Today's cryptocurrency prices by market cap," CoinMarketCap. [Online]. Available: https://coinmarketcap.com/

[42] "Ccip," Chainlink. [Online]. Available: https://chain.link/cross-chain

[43] B. C. Ghosh, T. Bhartia, S. K. Addya, and S. Chakraborty, "Leveraging public-private blockchain interoperability for closed consortium interfacing," in *Proceedings of the 40th IEEE International Conference on Computer Communications (INFOCOM'21)*. IEEE, 2021, pp. 1–10.

[44] "Near rainbow bridge," NEAR Blockchain. [Online]. Available: https://near.org/bridge

[45] P. Sheng, X. Wang, S. Kannan, K. Nayak, and P. Viswanath, "Trustboost: Boosting trust among interoperable blockchains," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*. ACM, 2023, pp. 1571–1584.

[46] E. N. Tas, R. Han, D. Tse, and M. Yu, "Interchain timestamping for mesh security," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*. ACM, 2023, pp. 1585–1599.

[47] I. Tsabary, M. Yechieli, A. Manuskin, and I. Eyal, "Mad-htlc: because htlc is crazy-cheap to attack," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (SP'21)*. IEEE, 2021, pp. 1230–1248.

[48] S. A. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP'22)*. IEEE, 2022, pp. 1299–1316.

[49] Y. Xue, D. Jin, and M. Herlihy, "Fault-tolerant and expressive cross-chain swaps," in *Proceedings of the 24th International Conference on Distributed Computing and Networking (ICDCN'23)*. ACM, 2023, pp. 28–37.

[50] M. Herlihy, B. Liskov, and L. Shrira, "Cross-chain deals and adversarial commerce," *The VLDB journal*, vol. 31, no. 6, pp. 1291–1309, 2022.

Appendix

A. Proof of Theorems in Section VI

Theorem 1 (Complete atomicity). Given a finite set of concurrent executions $\mathcal{E} \leftarrow \{\mathcal{E}_a, \mathcal{E}_{a+1}, ..., \mathcal{E}_{a'}\}$ on m distinct blockchains $\varPi_1, \varPi_2, ..., \varPi_m$, for the states of any pair of blockchains, $\mathcal{E}$ eventually triggers serializable atomic state transition.

Proof. We model the case where $\mathcal{E}$ contains one single execution $\mathcal{E}_k$ as Case 1 and the case where $\mathcal{E}$ contains multiple concurrent executions as Case 2. For Case 1 as a special case, we prove $\mathcal{E}_a$ triggers atomic state transition. For Case 2, we prove $\mathcal{E}$ triggers serializable atomic state transition.

Case 1: We prove atomic state transition in this case by contradiction. Let τ_i^j and $\tau_{i'}^{j'}$ denote the state of any pair of blockchains $\varPi_i$ and $\varPi_{i'}$ accessed by $\mathcal{E}_a$ but eventually only one state is modified. Without loss of generality, we consider the state transition $\tau_i^j \xrightarrow{a} \tau_i^{j+1}$ is applied and the state transition $\tau_{i'}^{j'} \xrightarrow{a} \tau_{i'}^{j'+1}$ is aborted.

As the state is eventually committed and modified as τ_i^{j+1} permanently in blockchain $\varPi_i$ by $\mathcal{E}_a$, sufficient dirty state votes and prepare state votes must have been received by replicas in $\varPi_i$ via the *CCC* primitive. Since the state transition $\tau_{i'}^{j'} \xrightarrow{a} \tau_{i'}^{j'+1}$ is aborted, at least one abortion message has been broadcast via the *CCC* primitive either from the dirty state layer or the prepared state layer. Combining both facts, at least one corrupted blockchain has equivocated in either of the two layers, conflicting with our assumption that all underlying blockchains are correctly coordinated by SMR. Specifically, equivocation in the dirty state layer indicates both a Merkle inclusion proof and a Merkle nil proof have been generated for the transaction on the corrupted blockchain, and equivocation in the prepared state layer indicates both an OK message and an $Abort$ message are broadcast by a corrupted blockchain.

Case 2: We further divide this case into two contradictory subcases and prove serializable atomic state transition each by contradiction.

The first subcase depicts that the underlying blockchains commit an inconsistent set of executions. Consider the states τ_i^j and $\tau_{i'}^{j'}$ of any pair of blockchains $\varPi_i$ and $\varPi_{i'}$ accessed by $\mathcal{E} \leftarrow \{\mathcal{E}_a, \mathcal{E}_{a+1}, ..., \mathcal{E}_{a'}\}$.

Let $\mathcal{E}^i$ and $\mathcal{E}^{i'}$ denote the set of executions that modify the blockchain states of blockchain $\varPi_i$ and $\varPi_{i'}$, respectively. If the equation $\mathcal{E}^i = \mathcal{E}^{i'}$ does not meet, without loss of generality, we consider an element $\mathcal{E}_b$ with $\mathcal{E}_b \in \mathcal{E}^i$ and $\mathcal{E}_b \notin \mathcal{E}^{i'}$. Similar to proofs in Case 1, the commitment of execution $\mathcal{E}_b$ on blockchain $\varPi_i$ and abortion on blockchain $\varPi_{i'}$ indicate equivocation of at least one blockchain, a contradiction.

The second subcase depicts that the underlying blockchains commit a consistent set of executions but not in identical order. Consider the states τ_i^j and $\tau_{i'}^{j'}$ of any pair of blockchains $\varPi_i$ and $\varPi_{i'}$ accessed by $\mathcal{E} \leftarrow \{\mathcal{E}_a, \mathcal{E}_{a+1}, ..., \mathcal{E}_{a'}\}$. Let $\mathcal{E}^e$ denote the set of executions that truly modify the blockchain states on $\varPi_i$ and $\varPi_{i'}$. Consider there exists two executions $\mathcal{E}_a$ and $\mathcal{E}_b$ in $\mathcal{E}^e$. Without loss of generality, $\mathcal{E}_a$ is committed before $\mathcal{E}_b$ in $\varPi_i$ and the executions are committed in a reverse order in $\varPi_{i'}$. The successful commitment of execution $\mathcal{E}_a$ in $\varPi_i$ indicates that sufficient prepared votes are received by $\varPi_i$ via *CCC*, including the vote from $\varPi_{i'}$, indicating $\varPi_{i'}$ has broadcast a prepared vote for $\mathcal{E}_a$ even though $\mathcal{E}_a$ is surely not the top element in the prepared state queue. According to our sequential commitment paradigm of the prepared state layer, $\varPi_{i'}$ has deviated from the protocol, contradicting our assumption that all underlying blockchains are secure. □

Theorem 2 (Termination). Given a finite set of concurrent executions $\mathcal{E} \leftarrow \{\mathcal{E}_a, \mathcal{E}_{a+1}, ..., \mathcal{E}_{a'}\}$ on m distinct blockchains $\varPi_1, \varPi_2, ..., \varPi_m$ input in arbitrary orders, all transactions in $\mathcal{E}$ are committed or aborted within a finite time period.

Proof. The proof for termination is straightforward and relies on the correctness of the underlying blockchains and the reliability of the *CCC* primitive. The key insight lies in that any transaction that stays in either the dirty state layer or the prepared state layer will be promoted to the next layer or aborted within a finite time.

Considering transactions in the dirty state layer, the termination of transactions is ensured by a predefined timeout parameter. After execution, the finality proof is broadcast and forwarded to all blockchains eventually since the *CCC* primitive satisfies reliability. If all involved transactions are committed within the predefined time, the execution terminates and promotes all state transitions to the prepared state layer. If any transaction is not published on the blockchain, a motivated counterparty can trigger the abortion phase by calling the abort interface to all underlying blockchains.

For a transaction belonging to execution $\mathcal{E}_{id}$, if it has already been promoted to the prepared state layer, all underlying blockchains must have successfully executed their corresponding transactions in the dirty state layer. Given the reliability of *CCC*, finality proofs of the transactions are eventually delivered and broadcast and underlying blockchains will generate an OK or $Abort$ message according to the rule of the dirty state layer. Moreover, the OK or $Abort$ message will eventually be delivered and processed by the commitment policy in the prepared state layer. Within a finite time, sufficient OK messages to commit the execution or any $Abort$ message will be delivered, thus guaranteeing the termination property.

□