Smart Contracts on the Move

Enrique Fynn
Università della Svizzera Italiana
Switzerland

Alysson Bessani
LASIGE, Faculdade de Ciências
Universidade de Lisboa
Portugal

Fernando Pedone
Università della Svizzera Italiana
Interchain Foundation
Switzerland

***Abstract*—Blockchain systems have received much attention and promise to revolutionize many services. Yet, despite their popularity, current blockchain systems exist in isolation, that is, they cannot share information. While interoperability is crucial for blockchain to reach widespread adoption, it is difficult to achieve due to differences among existing blockchain technologies. This paper presents a technique to allow blockchain interoperability. The core idea is to provide a primitive operation to developers so that contracts and objects can switch from one blockchain to another, without breaking consistency and violating key blockchain properties. To validate our ideas, we implemented our protocol in two popular blockchain clients that use the Ethereum virtual machine. We discuss how to build applications using the proposed protocol and show examples of applications based on real use cases that can move across blockchains. To analyze the system performance we use a real trace from one of the most popular Ethereum applications and replay it in a multi-blockchain environment.**

I. Introduction

Blockchain has gained much attention since the introduction of Bitcoin [1], and several blockchain systems have been developed thereafter [2]. Ideally a blockchain is a geographically replicated state machine that tolerates Byzantine failures. Blocks in a blockchain contain transactions, usually cryptographically signed by a user. In a permissionless blockchain system, blocks are produced by miners, each block cryptographically linked to the previous one, forming a chain. To produce a valid block, miners must solve a cryptographic puzzle. The miner whose valid block makes it to the canonical chain receives the block reward. In a permissioned blockchain system, miners can be in a consortium where they behave similarly to members of traditional BFT algorithms, such as PBFT [3], in which case they are named "validators".

As blockchain technology reaches mainstream use, it starts to face issues typical of more mature distributed systems technologies. Two formidable challenges are *scalability* and *interoperability*. Scalability has been early recognized as a major limitation of existing blockchain systems. And several attempts have been made to improve blockchain performance (e.g., [4], [5], [6]). In general, distributed systems scale performance by partitioning (sharding) the application state [7]. If the partitioning is such that most application requests can be executed within a single partition, and the load among partitions is balanced, then performance scales with the number of partitions. Unfortunately, few applications can be optimally partitioned (i.e., all requests fall within a single partition and load is balanced among partitions). As a result, most partitioned systems must handle requests that span multiple partitions. In the particular case of blockchain, it has been shown that even with a nearly perfect partitioning of the data, the existing Ethereum workload would result in a substantial number of cross-partition transactions [8]. The Achilles heel of scalable blockchain systems is their ability to handle cross-partition transactions.

Interoperability has been in the blockchain wishlist for some time. Yet, to date, no general mechanism has been proposed to share information across different blockchains. Inter blockchain communication (IBC) is necessary for multiple blockchains to co-exist in a heterogeneous way. For blockchains to interact with each other, some form of synchronization is required across blockchains. There are two main classes of solutions to handle transactions that involve multiple blockchains: (a) coordinating the blockchains involved in the execution of the transaction, in a scheme akin to atomic commitment [9], [10]; and (b) moving the state required by the transaction to a single target blockchain and then executing the transaction locally at the target blockchain.

Scalability and interoperability are different requirements. However, they can be addressed with a common mechanism: a *move operation* that allows accounts and arbitrary computation (i.e., *smart contracts*) to consistently migrate from one blockchain to another. In brief, the move operation works in two steps. In the first step, it locks a smart contract in the source blockchain. Once locked, the smart contract state cannot be changed in the source blockchain. A second step recreates the smart contract in the target blockchain in a provably correct way.

We have implemented the Move protocol in Ethereum [11] and Burrow [12], two popular blockchains, and evaluated it with different applications. Even though smart contracts with arbitrary code can move across blockchains consistently, we argue that developers should think of smart contracts as first-class objects that can move within blockchains. For example, if a smart contract maintains a set of users in its state, moving it will likely be inefficient because a possibly large state with all users has to move together with the smart contract. If, instead, the smart contract creates a new smart contract per user, then the move can happen more efficiently, at the granularity of individual users.

In summary, the paper makes the following contributions:

- We introduce the move operation, which allows programmable blockchains to interoperate, and provide a

programming model for smart contract developers.
- We extend Solidity, a popular smart contract language, to include operations that help developers to program smart contracts that can move within blockchains. We propose an interface for token smart contracts and evaluate their usage.
- We modify two different blockchain clients and extensively analyze how the system behaves with different applications based on both synthetic and real workloads.

The rest of the paper is structured as follows. Section II presents the background necessary to understand the move operation, detailed in Section III. Section IV explains how to use the move operation to implement interoperability and sharding. In Section V, we describe two different applications that can benefit from the move operation, and from Section VI to VIII we report on the experimental evaluation. Finally, Section IX surveys previous work and in Section X concludes the paper.

II. Background

A blockchain system is a distributed ledger, that is, an append-only log of transactions. *Clients* submit transactions to the blockchain, which are appended to the log and then executed. Geographically distributed *nodes* interconnected through a peer-to-peer overlay network implements the append-only log abstraction. Clients and nodes may be *honest*, in which case they follow their protocol specification, or *malicious*, in which case nothing can be assumed about their behavior. The blockchain system behaves correctly as long as a fraction of the nodes, typically more than two-thirds, are honest.

The append-only log is structured as a linked-list of blocks, each block divided between a header and a body. The header contains, among other information, a cryptographic link to the previous block. The body contains a list of transactions, each transaction cryptographically signed by the client that submitted it. The way the linked list of blocks is built leads to two categories of blockchain systems. In the first category, there are blockchain systems (e.g., Bitcoin [1], Ethereum [11]) that allow the chain of blocks to momentarily fork, that is, multiple blocks may be linked to a block. In the second category, there are blockchain systems that ensure a total order on linked blocks (e.g., Burrow/Tendermint [12]). These systems require nodes to agree on the next block to be appended to the chain, and therefore solve consensus. The techniques proposed in this paper apply to both categories.

Blockchain systems can also be distinguished by the nature of the operations they support. Some blockchain systems limit transactions to distributed asset transfer, while others allow transactions to perform arbitrary computation (e.g., Ethereum, HyperLedger Fabric [13], Cosmos/Tendermint [14]). In this paper, we assume blockchain systems in the second category. In our model, we make a distinction between two types of data objects: accounts which hold state and have a cryptographically derived unique identifier, and smart contracts, or for short contracts. Smart contracts encapsulate executable code, which can hold, read, and modify their own state and call other smart contracts. Clients hold one asymmetric key-pair per account, and every transaction originates from a client, who provides proof of ownership of an account.

Blockchains can grow large in size and complexity. For instance, Ethereum has over three terabytes of log at the moment and it can take several weeks to re-execute all appended transactions. Clients with low storage or computational power can succinctly prove the validity of an arbitrary piece of the state [15] without re-executing the entire log, provided they maintain and verify all the block headers.

Blockchain systems typically rely on a Merkle-tree or similar data structure to provide data integrity checks. For example, Bitcoin uses a binary Merkle-tree [15], while Tendermint uses a modified AVL tree [16]. For the sake of simplicity, we call these structures "Merkle-trees". Each block header includes the root of a Merkle-tree (i.e., Merkle-root). Data is encoded on the leaves of the Merkle-tree, and parent nodes are labeled with the cryptographic hash of child nodes grouped together, compacting the structure until a unique Merkle-root is reached. The objective of such data structure is to provide a computationally and spatially cheap way to prove the integrity of leaves of the state without necessarily having all the state.

Merkle-trees allow for a peer to hold only block headers and forego downloading all the blockchain state. Peers can ask for a proved piece of partial state (Merkle-proof) at a specific block height from peers that have the state at the requested block height or happen to have the same Merkle-proof. The information provided by these peers can be checked with the Merkle-root stored in the trusted block header. We denote the unique path of a valid Merkle-proof from object v to Merkle-root m as $\{v\} \mapsto m$. The leafs v and nodes $h \in (\{v\} \mapsto m)$ needed to reconstruct the proof must be given to verify the validity of the proof. The verification of Merkle-proofs can be done optimally in logarithmic time and space on the number of nodes of the tree [15].

Figure 1 illustrates how Merkle-proofs can prune parts of the tree logarithmically. Blocks from b_0 to b_n are shown linked together in the top. From block b_1 we see the Merkle-proof for $\{v\} \mapsto m$. Given a hash function H, m is b_1's Merkle-root composed by h_0 and h_1 hashes. In the figure we see the result of asking for the Merkle-proof of v, anyone can compute the Merkle-root of this proof and accept it only if it is equal to the trusted Merkle-root m stored in b_1. Observe that only v and hatched nodes h_0 and h_3 are needed to verify m.

III. The Move protocol

We propose a new method to move a contract from one blockchain to another. In the following, we state the assumptions needed to support the move operation (Section III-A), introduce the general idea (Section III-B), present the move operation in detail (Section III-C), and discuss extensions to the basic protocol (Section III-F).

A. Assumptions

We make the assumptions listed next in order for blockchains to support the move operation. Although these

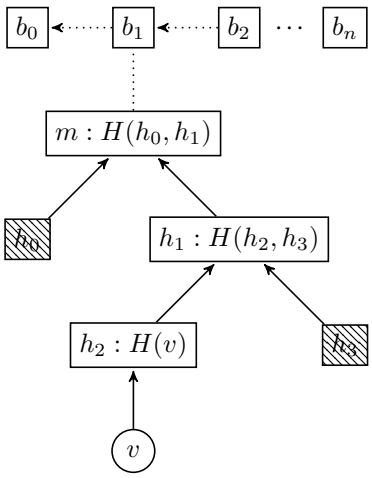

Fig. 1: Merkle-proof example, h_0, $h_3 \in (\{v\} \mapsto m)$.

assumptions are not strictly necessary to move objects across blockchains, they simplify the Move protocol.

In particular, we assume that blockchains must:

(a) support smart contracts (i.e., arbitrary computation);
(b) use the same execution environment (i.e., virtual machine); and
(c) provide a succinct way to prove state variables (e.g., using a Merkle-tree).

Supporting smart contracts that can execute arbitrary computation allows us to investigate more complex and generic use cases for the protocol. When moved to the target blockchain, we assume that smart contracts can execute the same opcode instructions they executed in the source blockchain, i.e., they use the same execution environment. This simplifies how communicating blockchains are capable to understand each other.

Clients can have information about the Merkle-root of any other blockchain by downloading the correspondent block header. Clients can listen to headers from multiple blockchains all at once. Block headers have a constant size of usually hundreds of bytes and are on average a small fraction of block bodies. For example, in Ethereum block headers are around 2% of the block's body. Moreover, blockchains willing to support the Move protocol must agree on certain configured parameters discussed in Section IV-A.

B. Overview

The state of a contract is presumably indivisible and must reside as a whole in a blockchain. Accounts and smart contracts are restrained to live in a single blockchain at a time. Therefore, we must ensure that if a contract moves from one blockchain (source) to another (target), it will no longer be "active" in the source blockchain. When the contract becomes active in the target blockchain, its state must be identical to its state when it became inactive in the source blockchain. This implies that the move operation involving two blockchains must be atomic.

One way to implement an atomic Move operation is to resort to the well-known two-phase commit (2PC) protocol [9], or one of its more resilient variations (e.g., [17]). We refrain from using a 2PC-like Move protocol since it would introduce expensive coordination between the involved blockchains (e.g., members of each blockchain would have to exchange votes in a reliable manner). Instead, we use a two-step approach that divides the move operation into two transactions, *Move1* and *Move2*. + In Move1, the state of a smart contract is "locked" in the source blockchain, after which it is guaranteed not to be changed—although transactions can still read the contents of the locked smart contract. In Move2, the smart contract is reconstructed in the target blockchain, after which it can be safely used. To avoid simple attack vectors, the Move2 transaction is only successful at the target blockchain if it contains a proof that the Move1 transaction was successfully executed at the source blockchain.

This two-step approach reduces coordination between the source and the destination blockchains, but it complicates atomicity. For example, the move of a contract can remain unfinished if the client fails after submitting the Move1 transaction and before it submits the corresponding Move2 transaction. To account for such cases, we allow any client to execute the Move2 transaction, and thereby complete a possibly unfinished Move operation. In the normal case, however, we expect the same client to execute both transactions.

C. The Move protocol in detail

Algorithm 1 details the move operation of contract c from blockchain B_i to B_j. We add a new field to a contract state, referred to as L_c, to L_c represent the blockchain identifier the smart contract c currently resides in. At low level, assigning a new value to L_c in Move1 is implemented with a new EVM opcode, OP_MOVE. The OP_MOVE opcode takes as argument the target blockchain identifier the smart contract is moving to (in this case B_j). When OP_MOVE is executed in c, it changes L_c to B_j, and by consequence blocks the contract state at B_i. Any transactions that try to alter the state of blocked contract c in B_i will be aborted.

Move2 assumes the existence of two boolean functions, V_S and V_P. $V_S(B, m)$ returns true if m is a valid Merkle-root in the blockchain B. $V_P(V \mapsto m)$ returns true if the state V of c is proved by $V \mapsto m$. The smart contract code and other blockchain specific variables (e.g., the amount of currency held by the smart contract) are omitted in the algorithm but still need to be proved by $V \mapsto m$. Notice that before submitting a Move2 transaction for contract c, the client must acquire the proof $V \mapsto m$ at the source blockchain (discussed later).

The Move1 and Move2 transactions allow application developers to execute special routines when a contract is moved. We illustrate the use of this functionality in the next section.

D. A concrete implementation

We have integrated the Move operation in the Solidity programming language [18]. In our prototype, smart contract developers must implement two functions to allow contracts

Algorithm 1 The operations.

```
1:  procedure MOVE1(c, B_j)          ▷ Move c in B_i to B_j, executed at B_i
2:      moveTo(·)                    ▷ Execute custom function
3:      L_c ← B_j                    ▷ Block contract c in B_i
4:  procedure MOVE2(c, V ↦ m)        ▷ Complete move of c, execute at B_j
5:      if L_c ≠ B_j then            ▷ Is c being moved to the wrong blockchain?
6:          return abort
7:      if V_S(B_i, m) = false then  ▷ Invalid Merkle-root
8:          return abort
9:      if V_P(V ↦ m) = false then   ▷ Invalid proof
10:         return abort
11:     for all v ∈ V do
12:         Call SSTORE(v.key v.value)   ▷ Recreate storage in B_j
13:     return moveFinish(·)         ▷ Execute custom function
```

```
address owner;
uint movedAt;
  function moveTo(uint _blockchainId) public {
    require(owner == msg.sender);
    require(now - movedAt >= 3 days);
  }
  function moveFinish() public {
    movedAt = now;
  }
```

Listing 1: Excerpt of a movable contract in Solidity.

to move, $moveTo(\cdot)$ and $moveFinish(\cdot)$ (see Algorithm 1). This provides the application developer a great deal of flexibility. For example, in Listing 1 we have an excerpt of Solidity code that in few lines ensures that only the contract's owner is allowed to move the contract and the contract must remain at least three days in the target blockchain before moved again.

E. Preventing replay attacks

If a client executes transaction T_{move1} at blockchain B_i to move contract c to B_j, any client can craft a special transaction T_{move2} to be executed in blockchain B_j that reconstructs the state of c in B_j. In T_{move2} the client appends the state of contract c encoded as V in the Merkle-proof $V \mapsto m$. Target blockchain B_j is responsible for verifying that m is accepted by the blockchain as a valid Merkle-root of B_i. Nodes in blockchain B_j verify the correctness of c's state by verifying $V \mapsto m$ and B_i's state root hash. If the proofs are valid, c's state can be safely reconstructed in B_j.

Additional measures should be taken to prevent replay attacks. The attack consists in a (malicious) client crafting a T_{move2} transaction that uses old state of contract c. The replayed transaction would obviously lead to inconsistencies as transactions that followed the first (and thus legitimate) Move2 transaction would be lost. One remedy would be to have nodes store the contract's nonce, a monotonically increasing number that is increased every time the contract is invoked. For instance, in Figure 2 a contract is moved from B_1 to B_2 (transactions T_{move1} and T_{move2}, respectively) and afterwards back to B_1 (transactions $T_{move1'}$ and $T_{move2'}$). It starts with nonce (n) equal to zero and as soon as T_{move2} is executed in B_2 it increments the nonce by one. Afterwards $T_{move1'}(B_1)$ completes in B_2 and changes the nonce to three. When client$_2$ tries to replay transaction T_{move2}, the contract's nonce is one which is less than what was previously seen by B_2 and the transaction aborts.

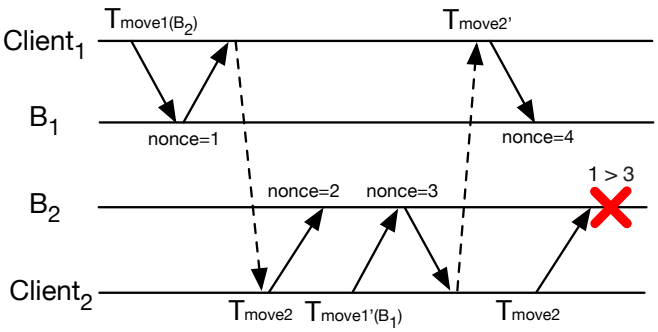

Fig. 2: Preventing replay on stale data.

F. Handling currencies

Most cryptocurrencies rely on an internal currency for generating incentives to maintain the system (e.g., paying miners and transaction fees). In some cases, the currency is the main usage of the system (e.g., Bitcoin). It is important therefore to have a way of transferring this part of the system state from one blockchain to another. As it turns out, we can devise a simple mechanism that uses our protocol to accomplish the feat. It suffices for smart contracts to be allowed to hold currency in their state. We can then implement smart contract "relays" that can transfer currency from blockchains by creating a token in the target blockchain that is provably locked in the source blockchain, similar to Pegged Side Chains [19]. Currencies can be unlocked when contracts return to the original blockchain.

Assume for example that we would like to transfer e units of currency from $client_1$ to $client_2$, from blockchain B_i to blockchain B_j. We assume the existence of contract c in B_i, that when called by client $client_1$ with input B_j, $client_2$ and value associated e creates a contract r that has e units of currency and lets $client_2$ withdraw e from its state (i.e., it executes Move1(B_j) on creation). The $client_2$ can call T_{move2} on r effectively moving it to B_j where the funds would be available as a B_i's token in B_j.

In Figure 3, we can see an example where a client successfully transfers a currency token from blockchain B_1 to a token representation in blockchain B_2. Contract c's function $create$ is called with e units of B_1's associated currency with T_{create}. The transaction creates contract r with e units of currency, seen in the figure as "$", afterwards the same function calls r's $moveTo(B_2)$ changing r's L_c to B_2. The newly created contract has functions to generate tokens which are proved to be backed by e in B_1. The $client_2$ waits for the transaction inclusion in B_1 and sends transaction T_{move2} to B_2, proving that r was moved to B_2. After transaction T_{move2} is included in B_2, $client_2$ calls transaction T_{mint} which executes code in r creating tokens in B_2 that represent the locked coins in B_1.

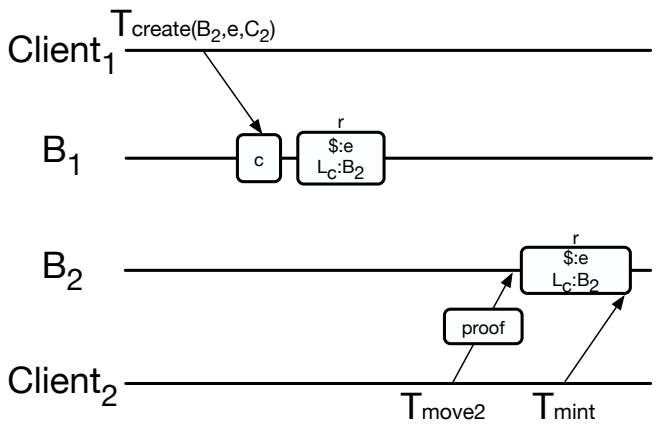

Fig. 3: Move operation example.

G. Additional details

a) Account identifiers: Although each blockchain maintains its own set of accounts, its identifier can be the same if the interacting blockchains use the same rule to derive such identifiers. Consequently, clients could use the same cryptographic keys to use accounts in different blockchains. It becomes essential to incorporate the blockchain's identification to functions that compute contract addresses to ensure a unique system-wide contract identification to avoid collisions in contract identifiers.

b) Finding contracts: L_c can only have two logical states: either in the current blockchain or transferred (executed Move1), and if $moveTo(\cdot)$ and $moveFinish(\cdot)$ are implemented correctly, it can always go back to the first state with Move2. A client who does not know where contract c is located can use L_c to track the contract's location every time it moves.

c) Stale data: Every time a contract is moved it leaves behind stale state on the original blockchain, which could be garbage collected, paying attention to guard against the attack previously described. Designing fee incentives to clean the state is left as future work.

IV. Applications

The Move protocol can be used by blockchains to implement two important blockchain concepts: interoperability and sharding. Interoperability and sharding are related concepts that have increased in importance as blockchain technology matures.

A. Interoperability

Blockchains coexist nowadays with different transaction designs, cryptographic features, and trust assumptions in a heterogeneous environment. Interoperability provides a way to offload transactions from one blockchain to another, unleashing the potential to scale different applications and experiment with different combinations of blockchains.

Interoperability in permissionless systems is challenging mainly because forks can occur since block propagation time is unbounded [20], which invalidates transactions that build on the losing side of the fork. A way for systems proposing interoperability [19], [21], [22] to interact with permissionless blockchains is by introducing a parameter p that specifies the minimum number of blocks that a transaction's block should be behind the blockchain's head for it to be accepted by the other blockchain. The parameter can be configured according to each blockchain involved in the interoperability protocol.

Miners or validators of blockchains willing to interoperate should maintain a light client that validates merkle-roots of other blockchains, proposals for this scheme are discussed in Section IX.

B. Sharding

Sharding enables the blockchain state to be divided into shards responsible for holding a certain portion of the state. Sharding preserves some of the blockchain assumptions, but there are clear trade-offs in security because the members themselves have to be sharded. The way objects are assigned to each shard plays an important role when sharding a blockchain for scalability. For instance, if objects are randomly partitioned into shards, most of the transactions will likely be cross-shard. The rate of cross-shard transactions also increases with the number of shards, and sharding the state while minimizing the number of cross-shard transactions keeping the various shards balanced is a hard problem [8].

To cope with changes in load it becomes essential to incorporate a method to move state from shards, offloading one shard in detriment of another. As shards get congested and fees increase, users are tempted to move their contracts to underused shards.

V. Use cases

We implemented two applications for smart contracts and show how they scale with the number of blockchains:

- *SCoin*: A token smart contract based on a popular Ethereum token interface.
- *ScalableKitties*: A clone of *CryptoKitties*, a popular Ethereum application where virtual cats can migrate and reproduce in different shards.

A. SCoin

ERC20 [23] is a standard interface widely used in Ethereum for token operations including token transfers. *STokenI* and *AccountI*, defined in Listing 2, are interfaces that support all ERC20 operations and allow for contracts to move from one blockchain to another. The main idea of the interface is to use one instance of *AccountI* per user account. Typical ERC20 implementations hold token balances in a map data structure, which multiple blockchains cannot share in our design since we do not allow for contracts to live in two or more blockchains at the same time.

Once created, accounts can freely move from blockchain to blockchain using the *moveTo* and *moveFinish* functions. It

```
contract STokenI {
  function totalSupply() public view returns (uint);
  function newAccount() public payable returns (AccountI, uint);
  function newAccountFor(address _forAddr) public payable returns (AccountI, uint);
  event CreatedAccount(address account, uint salt);
}

contract AccountI {
  function balance() public view returns (uint);
  function allowance(address _spender) public view returns (uint);
  function transfer(AccountInterface _to, uint _tokens) public returns (bool);
  function approve(address _spender, uint _tokens) public returns (bool);
  function transferFrom(AccountInterface _to, uint _tokens) public returns (bool);
  function debit(uint _tokens, bytes _proof) public returns (bool);
  function moveTo(uint _shardId) public;
  function moveFinish() public;
  event Transfer(address _to, uint _tokens);
  event Approval(address _spender, uint _tokens);
}
```

Listing 2: Scalable Token interfaces extending ERC20.

is left to the developer to restrict or even define a policy for moving accounts between blockchains.

To illustrate the *STokenI* interface, we implement *SCoin*, a scalable token contract that implements *STokenI*. *SCoin* creates instances of *SAccount*, which implements *SAccountI*. The implementation of *SCoin* and most functions of *SAccount* are straightforward and application-dependent. We focus next on how to do safe transfers between one *SAccount* to another. To execute a transaction that transfers e tokens from *SAccount* A to B, contract A has to decrease e from its state (called by *transfer* function) and B has to increase e. This is done by calling the *debit* function in B. If contracts A and B are in different blockchains, they have to be first moved to the same blockchain to be able to call each other. Once both contracts are in the same blockchain, how can A know that B is what it claims to be? For instance, one could design a contract B that, when *debit* is called, increases the contract's tokens by an arbitrary amount. A could ask its parent if B was created by the same contract, but A's parent contract might be in a different blockchain. The interface does not specify how contract A can be sure of contract B's origin. It is up to the developer to devise safeguards to prevent incorrect usage. The key idea for *SCoin* is holding a proof in B that it was created by the same contract that created A.

When we create an instance of *SAccount* in *SCoin* it uses a monotonically increasing salt, stored in the instance state. The salt is used to calculate the identifiers of both contracts using the *create2* opcode [24]. With A's salt, B can attest that A was created by the same contract that created B and vice-versa. To execute a transfer from *SAccount* A to B, contract A attests B's origin, decrements its own balance and calls the function $debit(\cdot)$ in B. Contract B agrees to debit its own account only if A passes the same check, and A can safely add B's fund to its own balance. The checks of origin are done with one inexpensive hash operation. In our implementation case, we take advantage of the way contract identifiers are generated in the EVM. A more generic method could be devised using Merkle proofs with the same proposed interfaces.

B. ScalableKitties

ScalableKitties is a clone of *CryptoKitties*, a popular Ethereum smart contract that was created in November 23th 2017 and until the writing of this paper had over four million related transactions. During the apex of its popularity, *CryptoKitties* congested the Ethereum network for several days, accounting for over fifteen percent of all Ethereum transactions [25]. In *CryptoKitties* cats are collectibles that can be bred to generate more cats following a set of rules (e.g., sibling cats cannot mate). Cats were first generated by the contract's owner. Both the initial and bred cats are sold in an auction smart contract. The game was the first mainstream application built on top of Ethereum, and some cats were sold for more than a hundred thousand dollars at the time. The contract is still actively used today.

ScalableKitties's functions map one-to-one to the *CryptoKitties* smart contract but for simplicity we discuss only the functions that are related to the execution of cross-blockchain transactions. Cats are created in two ways: either by having the contract's owner calling a function to generate "promotional" cats or by breeding two cats to generate a third. Breeding is the only operation that can generate cross-blockchain transactions because bred cats can be in different blockchains and need to be moved to the same one. Furthermore, if the owner of cat A wants to breed A with B, it either needs to own both cats or B's owner has to permit B to be sired with A. In the experiments in Section VII-A we replay transactions from the *CryptoKitties* contract on the *ScalableKitties* contract.

VI. Deployment

We modify Hyperledger Burrow [12] and Ethereum [26] to implement the protocol defined in Section III. The resulting systems allow blockchains to communicate with each other (IBC) or implement sharding. To validate our approach, we conducted two types of experiments: we shard Burrow and

analyze how applications can scale performance, and make smart contracts migrate from Ethereum to Burrow and vice-versa to assess the performance and monetary costs of IBC. Both Burrow and Ethereum implement the EVM model, where each opcode executed by the smart contracts has a cost modeled in *gas*, e.g., a sum between two integers costs 3 gas, while creating a new smart contract costs 32000 gas [24].

Burrow uses Tendermint for consensus, which by design introduces the application's Merkle-root from block n in block $n+1$, but in Burrow the state of block n is saved only in block $n+1$, therefore there is a need to wait for two blocks to prove the transaction inclusion required by Move2. For the experiments we set p (defined in Section IV-A) equal to two blocks in Burrow, since clients have no option other to wait for two blocks to get the proof of inclusion of a Move1 transaction. For Ethereum we set p to six blocks.

Tendermint is configured to wait for five seconds between each consecutive block, the observed latency being slightly higher than this. Ethereum is configured to wait 15 seconds, which is similar to Ethereum's main public network.

All experiments were conducted using a heterogeneous cluster in a local area network with simulated latencies between nodes based on the values published in [27], where the authors evaluate nodes in 14 regions in four continents on Amazon data centers. We emulate this environment in the cluster and randomly allocate nodes to regions.

VII. Sharding Experiments

We evaluate the Move protocol using the two applications described in Section V. The objective is to show how the capacity to move smart contracts can significantly improve the performance of the applications. In all sharding experiments, one node hosts all clients and maintains one connection per shard to broadcast the client's transactions. We decided to run one validator in each node and 10 validators per shard due to a limitation of our cluster size, comprised of 80 computers, each one with an eight-core Intel Xeon L5420 processor working at 2.5GHz, 8Gb of memory, SATA SSD disks and 1Gbps ethernet card. With this configuration we can run a maximum of 8 shards and we decided to run experiments with 1 (no sharding), 2, 4 and 8 shards. To decide which shard to send the contracts to when needed, we use hash partitioning where the contract's shard is decided by the hash of the contract's identification. Using hash partitioning ensures a good balance among shards but implies probably a higher cross-shard rate the more shards there are. We do not focus our attention to examine different partitioning techniques but we believe greater improvements are possible by using different sharding methods [8].

A. ScalableKitties

In order to produce the data for experiments we scanned all transactions involving the *CryptoKitties* contract deployed [1] in Ethereum since its inception.

[1]At address 0x06012c8cf97bead5deae237070f9587f8e7a266d

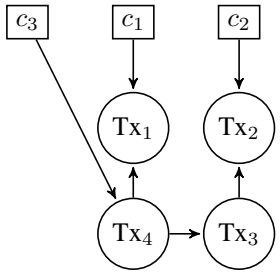

Fig. 4: Dependency graph example

Since every transaction from the original contract must succeed in our implementation, we first construct a dependency DAG (Directed Acyclic Graph) of all the transactions and execute them respecting their dependencies. By doing so we are able to execute some of the transactions in parallel. for instance, consider a client that owns cat c_1 and wants to breed its cat with another user's cat c_2. To execute this transaction, the transaction graph of Figure 4 needs to be respected. First cats c_1 and c_2 have to be created with Tx_1 and Tx_2, respectively, then c_2's owner agrees with the breeding with Tx_3 and finally Tx_4 breeds c_1 and c_2, creating c_3. Vertices c_1, c_2, and c_3 are pointers to transaction vertices that have a dependency on c_1, c_2, and c_3, respectively. Notice that leaf transactions in the DAG can be executed in parallel, for instance, Tx_1 and Tx_2 can be executed in parallel but Tx_4 can only execute when it becomes a leaf, i.e., both Tx_1 and Tx_3 finish.

We replay *CryptoKitties* transactions on *ScalableKitties* in a sharded environment running multiple instances of the Burrow client. We use the dependency DAG described previously to replay transactions to the contract. Transactions that are appended in a block (executed) are removed from the DAG. Subsequent transactions that become leaves in the DAG are submitted until a limit of 250 outstanding transactions is reached. We pre-process the whole DAG in memory, broadcasting the first transactions, updating the DAG, and possibly sending other transactions whose dependencies are satisfied. The process continues until the experiment is over. Increasing the number of shards leads to an increase in the number of cross-shard transactions, that in turn reduce the number of leaves in the DAG since cross-shard transactions depend at least on two transactions: Move1 and Move2.

Figure 5 (left) shows a nearly linear increase in the average number of transactions per second as we increase the number of shards, except when there are eight shards. The reason the throughput with eight shards is lower than expected is that there were not enough ready-to-run transactions in the dependency graph, making the client wait for blocked transactions to finish. This is better visualized in Figure 5 (right), where vertical dashed lines mark the point when each one of the eight shards had less outgoing transactions than established at the beginning of the experiment.

To better understand how varying cross-shard transaction rates can affect performance we conduct experiments in the

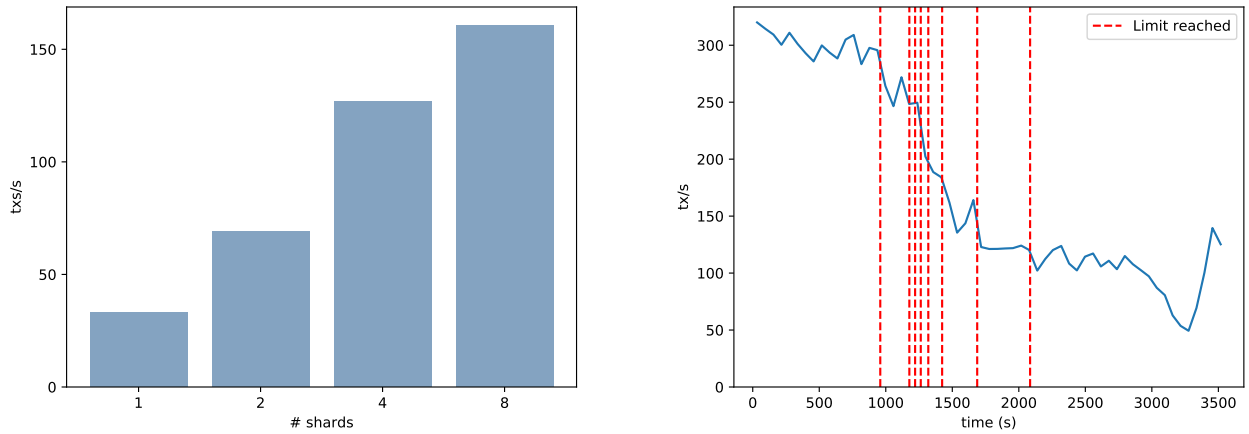

Fig. 5: *ScalableKitties* throughput for 2, 4 and 8 shards (left), and aggregated throughput over time for 8 shards (right).

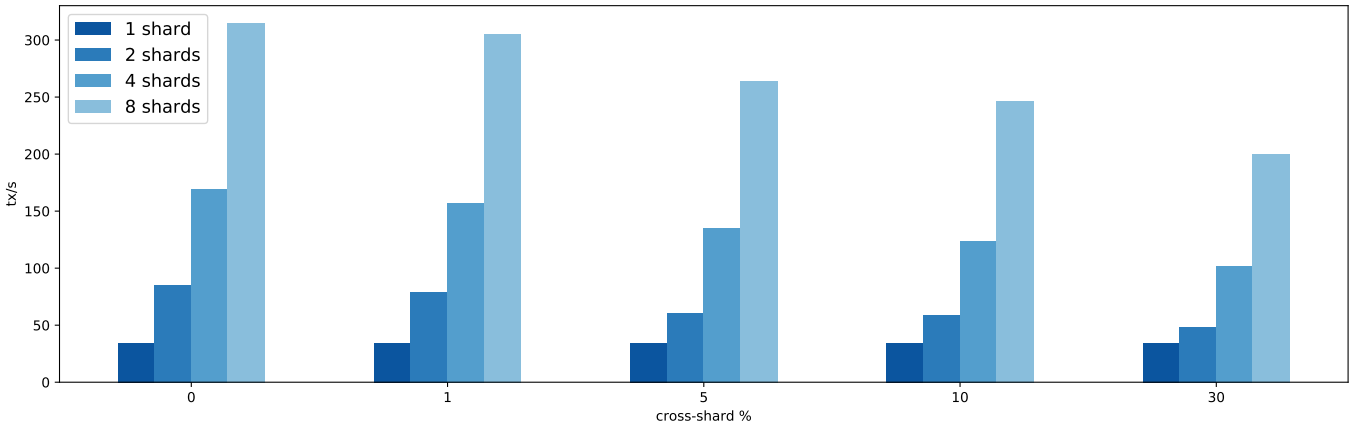

Fig. 6: Performance with varying number of shards and different cross-shard transaction rates.

next section that confirm the performance gains obtained with *ScalableKitties*.

B. SCoin

We now present results for the SCoin application defined in Section V-A. In these experiments, we benchmark how well the protocol can perform with a single application with varying number of transactions that require cross-shard communication (i.e., tokens transferred between different partitions). We try to measure latency and throughput tradeoffs with a varying number of cross-shard transactions.

Each client in the experiment tries to execute *transfer* transactions in a closed-loop. If the transaction is cross-shard, i.e., if a client is trying to transfer its token to an account that resides in a different shard, the client first move its account to the corresponding shard and then executes the *transfer* transaction afterward in the destination shard. Similarly to the previous experiment, we tune the number of clients in the system to avoid a significant degradation of the average latency for each client, thus caping each shard to 250 clients. We experiment with different cross-shard transaction rates for different shard numbers.

Figure 6 shows the aggregate throughput of the system for a varying number of shards and different percentages of cross-shard transactions. The one shard experiment is shown in every cross-shard rate experiment as a reference. We can see how performance degrades when increasing the number of cross-shard transaction rates, but the throughput grows linearly at different rates of cross-shard. For comparison, the previous experiment had on average 5.86%, 7.93%, 7.85% cross-blockchain transaction rate for 2, 4 and 8 shards, respectively. Any system that reserves part of its allocated resources to process cross-shard transactions will have similar behavior, due to cross-shard transactions occupying the resources that otherwise would be used by single-shard transactions.

The average latency for clients does not change significantly when increasing the number of shards, remaining at around 7 seconds for single-shard transactions and 34 seconds for cross-shard transactions. Cross-shard transactions demand two transactions for each move operation, plus waiting for two

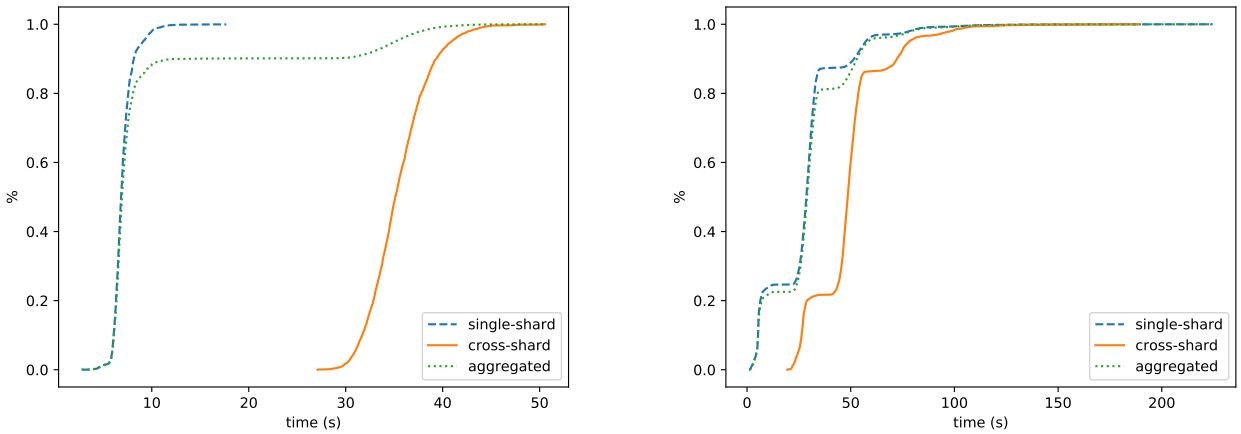

Fig. 7: Latency CDF for 4 shards experiment with 10% cross-shard transactions.

blocks to prove the contract's state and one final transaction to complete the operation, confirming the expected latency of waiting for five blocks per cross-shard transaction. In Figure 7 (right) we can see the cumulative distribution function (CDF) for clients' observed latencies in a scenario with four shards and 10% cross-shard transactions rate. The aggregated latency shows both single and cross-shard transactions and we can see that, as expected, around 10% of the transactions takes more than 30 seconds to complete. Differently from other sharded systems (e.g., [28]) the protocol does not suffer from a *convoy effect* [29], that is, cross-shard transactions do not delay single-shard transactions.

1) Retries: In the previous experiments, to better control the rate of cross-shard transactions, clients submit transactions only if they know the contracts are not going to be moved when the transaction is executed. To better model transactions in the presence of conflicts we experiment with the *SCoin* contract without any help from external sources. Two situations can make clients retry transactions: when performing a single-shard transaction and the interacted contract is moved to another shard or when performing a cross-shard transaction and the called contract is moved to another shard. In the experiment we make clients wait a random time corresponding to the creation of 0 to 10 blocks if the transaction fails for any of these reasons, this is done to prevent contracts moving back and forth in an endless cycle.

Figure 7 (right) shows the ideal latency where no conflicts exist, and Figure 7 (left) shows the latency when conflicts can happen. A clear increase in latency is observed when comparing both figures, but when retries can happen the number of times the same transaction fails and has to retry is highly skewed, for instance, 66% of the transactions that retry, do it just once, and only 1% of these transactions are retried more than three times.

VIII. IBC EXPERIMENTS

This section presents some experiments for inter-blockchain communication (IBC) considering Burrow/Tendermint and Ethereum. These experiments aim to measure the time and gas (which translates to cryptocurrency costs) consumed for operations with five different applications:

- *SCoin*: Transfer one token from one blockchain to another and transfer the virtual currency to another account in the target blockchain.
- *ScalableKitties*: Transfer one virtual cat from one blockchain to another, breeds the cat with an existing cat in the target blockchain and gives birth to another cat.
- State 1, State 10, State 100: Transfer the state containing respectively 1, 10 and 100 32-byte state variables from one blockchain to another.

These operations require moving their corresponding smart contract from the source to the target blockchain, an operation requiring two transactions (Move1 and Move2) and the wait for p blocks in-between transactions. After that, further transactions might need to be executed in the target blockchain. In our examples: *SCoin* requires one further operation to transfer the token to a contract in the target blockchain, while *ScalableKitties* requires two transactions *breed* and *giveBirth* to mate and produce a new virtual cat, respectively. The state transfer experiments do not require any further transactions for completion.

Figure 8 presents the time required to perform an operation from Ethereum to Burrow and vice-versa. Unsurprisingly, the time to perform a single transaction is bound to the latency between consecutive blocks in each blockchain. To execute Move2 from Ethereum to Burrow one is required to wait for 6 Ethereum blocks that translates to approximately 90 seconds and ends up dominating the overall time for every operation.

A good way to analyze the impact of each operation in the system's performance is to measure the gas consumed by each operation, the gas cost is expected to grow linearly with

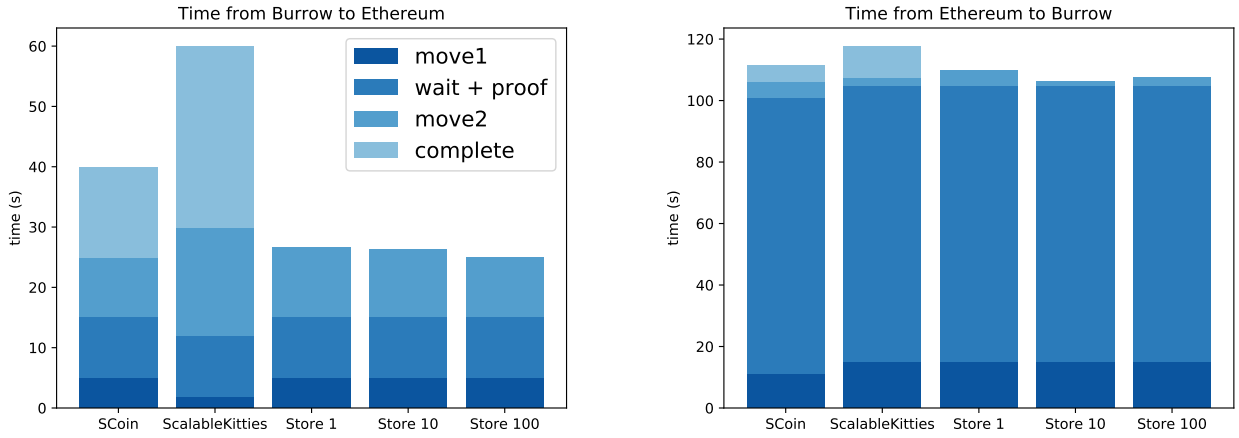

Fig. 8: Latency for five different inter-blockchain applications.

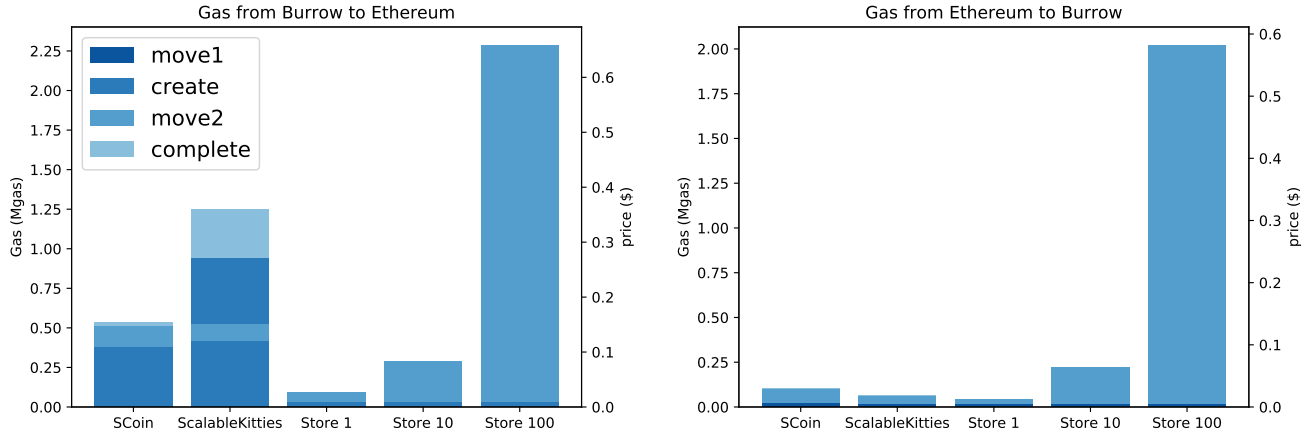

Fig. 9: Gas and monetary costs for five different inter-blockchain applications.

the size of the transferred smart contract state, since creating or modifying state variables are expensive operations in the EVM model. Figure 9 shows in the left y-axis the amount of gas paid by each transaction. To better understand these costs, in the right y-axis we present the same costs in US dollars, considering the current average value of one gas as two Gwei (2×10^{-9} Eth) and one Eth as $144 (the price in the middle of December of 2019).

The way Burrow and Ethereum calculate gas prices are different, besides the actual values per operation being different, some operations pay no gas, e.g., in Ethereum, if one smart contract or transaction creates a new smart contract it pays an amount of gas per byte of the contract code that is being created, while in Burrow no gas is paid per byte of code. In both systems the code is immutable and stored in the state Merkle-tree with the key based on the code's hash.

In Figure 9, the vertical hatched bars represent the gas paid by the creation of new contracts in Ethereum, every recreated contract pays a constant gas based on the size of the moved code. In the *ScalableKitties* application the *giveBirth* function creates a new contract thus it pays for the gas again, for *SCoin* and *ScalableKitties* the gas paid for the code creation corresponds to around 70% of the total gas cost. We note that it is possible to reduce significantly the Ethereum contract creation costs if the contract code is already in the blockchain.

IX. Related work

Scaling blockchains is a hot topic for both industry and academia, and many proposals have appeared in the last years (e.g., [30], [31]). In this paper, we showed that scalability can be achieved with a novel IBC protocol and sharding. A different approach to scaling blockchains is to shift computation from the blockchain (on-chain) to the outside (off-chain). For example, in the method proposed by TrueBit [32] most of the computation is done off-chain. Cheating participants can be caught by using an on-chain game in which anyone can prove they misbehaved in logarithmic time. Another example of a scaling solution is the lightning network [33], which works

on top of Bitcoin, and its Ethereum counterpart, Raiden [34]. Users of such systems create off-chain channels between them in order to minimize on-chain transactions. The efficiency of such systems is highly application-dependent.

Pegged Sidechains [19] focus on transferring assets between proof-of-work blockchains. The idea is to lock assets in one blockchain and recreate them in another blockchain by providing a proof of such locking in the original blockchain. The Move protocol generalizes Pegged Sidechains in several aspects. First, it allows to transfer any state across blockchain systems, not only assets; second, it applies to both proof-of-stake and proof-of-work approaches; third, the Move protocol shifts control to the application developer, who can develop scalable applications with their own logic (e.g., introducing load balancing mechanisms).

In HyperService [10], the authors create a new programming language and system to make blockchain applications interoperable. HyperService orchestrates the execution of applications that span multiple blockchains. We take a different approach: a smart contract is executed in one blockchain, after the smart contract dependencies are moved to the same blockchain.

In [35], atomic token swaps between multiple blockchains are studied and proposed as a protocol that, similar to ours, is done in two phases. As noted by the author, "atomic cross-chain swap is an atomic cross-chain transaction, but not vice-versa". Our protocol could be used to implement atomic swaps in a similar way as shown in III-F, although a more efficient solution for performing token swaps with more than two blockchains, combining our protocol with the techniques proposed in [35] would be interesting future work. In [36] the authors propose a solution for permissioned ledgers where blockchains implement a common "relay" mechanism for cross-blockchain transactions with smart contracts. The protocol provides a great deal of flexibility but it requires smart contract developers to have a deep understanding of the underlying cross-chain protocol. Without allowing contract's state to migrate, blockchain systems risk having their performance limited by cross-blockchain transaction performance.

PolkaDot [21] aims to create a decentralized *federation* of blockchains by allowing other blockchains (called parachains) to exist. Existing blockchains can be interfaced by *relay-chains*, but no details are given on how the validation happens on existing blockchains. Similar to our work, blockchains in PolkaDot need well-defined parameters so they can interoperate, e.g., the number of blocks to wait to accept the transaction as being final. Cosmos [22] also aims to provide IBC by allowing multiple blockchains, called *zones*, to communicate with each other. All Cosmos zones run the Tendermint algorithm for consensus. One zone, called *Cosmos hub* acts as a central communication interface for all other zones. Interledger [37] is a proposed protocol for IBC that tackles payments. To achieve safety and liveness, transactions are either escrowed by notaries that run PBFT [3] or use incentives on rational actors . The Interledger approach does not integrate with existing blockchains and is constrained to simple token transfers.

In Omniledger [5], the authors developed a protocol that can process transactions across shards, called Atomix, built on top of Bitcoin's UTXO model. In Atomix, clients can lock their input and are left with the burden of unlocking them in case the transaction aborts. The authors briefly discuss applying Atomix to smart contracts and suggest that Atomix is suitable for scenarios where clients execute simple operations and transactions do not conflict. Our approach exposes IBC primitives to developers to give them more freedom to programmatically condition cross-blockchain operations. Similar to Omniledger, Elastico [38] focus on Bitcoin's UTXO model and there is no need for cross-shard transactions because outputs are assumed to be disjointedly distributed to shards.

Chainspace [39] builds on top of BFT-SMaRt [40], an open source BFT consensus implementation written in Java. One of the subsystems of Chainspace is S-BAC, a two-phase commit protocol that deals with cross-shard transactions. In our model, we expose sharding primitives that let developers programmatically control the shard designation of smart contracts, doing so simplifies the protocol and allow for a more organic distribution of objects. Differently from S-BAC, our protocol does not implement aborts and it is the developer's responsibility to avoid contracts stuck on the first phase of the protocol. Chainspace provides helpful insights for further works on sharding smart contracts that could be also applied in the presented protocol, e.g., having shards checkpoints can be beneficial to reduce the bandwidth and storage cost of moving a contract. Similarly, the efficient shard state transfer protocol described in [41] can alleviate this costs

A protocol with similar goals as ours has been proposed in the Ethereum research forum concurrently with the development of this work [42]. It describes a similar operation to move the state from one shard to another, called *yanking*. Other threads in the same forum further extend the proposed idea, but until the writing of this paper, it remains a work in progress tailored specifically for the Ethereum environment, and constrained by its own design choices.

X. Conclusions

In this paper, we present a practical protocol that can be used to develop smart contracts that can move between different blockchains. Our protocol enables smart contract developers to create blockchain applications that interoperate and scale in an ecosystem of multiple blockchains. We developed two applications for our protocol and extensively evaluated their performance and tradeoffs. The simplicity of our protocol opens up possibilities for further improvements, such as decentralized load balancing smart contracts for sharded blockchains.

Acknowledgments

We wish to thank our shepherd, Pascal Felber, and the anonymous reviewers for helping us improve the paper. This work was partially supported by FCT through project ThreatAdapt (FCT-FNR/0002/2018), the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020), and the Swiss National Science Foundation (project number 175717).

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] C. Cachin and M. Vukolić, "Blockchains consensus protocols in the wild," *arXiv preprint arXiv:1707.01873*, 2017.

[3] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[4] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*. Springer, 2017, pp. 357–388.

[5] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger." *IACR Cryptology ePrint Archive*, vol. 2017, p. 406, 2017.

[6] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 931–948.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[8] E. Fynn and F. Pedone, "Challenges and pitfalls of partitioning blockchains," in *BCRB 18: DSN Workshop on Byzantine Consensus and Resilient Blockchains*, 2018.

[9] J. N. Gray, "Notes on data base operating systems," in *Operating Systems*. Springer, 1978, pp. 393–481.

[10] 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 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 549–566.

[11] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[12] "Hyperledger burrow client," https://github.com/hyperledger/burrow.

[13] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 30.

[14] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1807.04938

[15] M. Szydlo, "Merkle tree traversal in log space and time," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2004, pp. 541–554.

[16] "Iavl+ implementation," https://github.com/tendermint/iavl.

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

[18] "Solidity language," http://solidity.readthedocs.io/en/latest/index.html.

[19] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," *https://www.blockstream.com/sidechains.pdf*, 2014.

[20] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *IEEE P2P 2013 Proceedings*. IEEE, 2013, pp. 1–10.

[21] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," *https://pdfs.semanticscholar.org/f76f/652385edc7f49563f77c12bbf28a990039cf.pdf*, 2016.

[22] J. Kwon and E. Buchman, "Cosmos whitepaper," *https://cosmos.network/resources/whitepaper*, 2019.

[23] V. Fabian and B. Vitalik. (2015) Erc20 token standard. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[24] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014, https://files.gitter.im/ethereum/yellowpaper/VIyt/Paper.pdf.

[25] Q. T. Zhong and Z. Cole, "Analyzing the effects of network latency on blockchain performance and security using the whiteblock testing platform," 2019, https://www.whiteblock.io/wp-content/uploads/2019/07/analyzing-effects-network.pdf.

[26] "Official go implementation of the ethereum protocol," https://github.com/ethereum/go-ethereum.

[27] T. Crain, C. Natoli, and V. Gramoli, "Evaluating the red belly blockchain," *arXiv preprint arXiv:1812.11747*, 2018.

[28] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *2010 29th IEEE Symposium on Reliable Distributed Systems*. IEEE, 2010, pp. 214–224.

[29] T. Ahmed-Nacer, P. Sutra, and D. Conan, "The convoy effect in atomic multicast," in *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*. IEEE, 2016, pp. 67–72.

[30] P. Gazi, A. Kiayias, and D. Zindros, "Proof-of-stake sidechains." *IACR Cryptology ePrint Archive*, vol. 2018, p. 1239, 2018.

[31] G. Wang, Z. J. Shi, M. Nixon, and S. Han, "Sok: Sharding on blockchain," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 41–61.

[32] J. Teutsch and C. Reitwießner, "A scalable verification solution for blockchains," https://people.cs.uchicago.edu/teutsch/papers/truebit.pdf, 2017.

[33] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016.

[34] "cheap, scalable token transfers for ethereum," https://raiden.network.

[35] M. Herlihy, "Atomic cross-chain swaps," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, 2018, pp. 245–254.

[36] E. Abebe, D. Behl, C. Govindarajan, Y. Hu, D. Karunamoorthy, P. Novotny, V. Pandit, V. Ramakrishna, and C. Vecchiola, "Enabling enterprise blockchain interoperability with trusted data transfer (industry track)," *arXiv preprint arXiv:1911.01064*, 2019.

[37] S. Thomas and E. Schwartz, "A protocol for interledger payments," *https://interledger.org/interledger.pdf*, 2015.

[38] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 17–30.

[39] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A Sharded Smart Contracts Platform," in *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[40] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 355–362.

[41] A. Nogueira, A. Casimiro, and A. Bessani, "Elastic state machine replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, Sep. 2017.

[42] V. Buterin, "Cross-shard contract yanking," https://ethresear.ch/t/cross-shard-contract-yanking/1450, 2018.