Extending Existing Blockchains with Virtualchain

Jude Nelson[†], Muneeb Ali[†‡], Ryan Shea[‡], Michael J. Freedman[†]
[†]Princeton University, [‡]Blockstack Labs

Public blockchains are becoming a ubiquitous network service. However, it's hard to make consensus-breaking changes to production blockchain networks. To overcome this, we created *Virtualchain*, a logical layer for implementing arbitrary fork*-consistent replicated state machines (RSMs) on top of already-running blockchains.

Blockchains provide a **totally-ordered, tamper-resistant journal of state transition events**. New applications can store a log of all state changes in a public blockchain, such as Bitcoin [14], Litecoin [11], or Ethereum [6]. By using the blockchain as a shared ground truth, these applications can then **bootstrap global state** in a secure, decentralized manner, since every application node subscribed to the blockchain can independently construct the same state as all other application nodes.

However, there are two key challenges to using blockchains in this manner. First, a blockchain can fail—it can go offline, or its consensus mechanism can become "centralized" by falling under the *de facto* control of a single entity. To tolerate failures, application journals must be efficiently migrateable across blockchains. Cross-chain migration is already needed by production systems; for example, we migrated a production system from Namecoin [12] (which became centralized) to Bitcoin [17].

The second challenge is that the application's journal can be forked and corrupted by the underlying blockchain. If the blockchain forks, nodes on different forks will write and read different events, forking the journal. The blockchain may drop and re-order transactions when its forks join, causing bootstrapping nodes to construct different state than already-running nodes. Applications must be able to recover from these failures. To do so, we created Virtualchain.

Virtualchain is a logical layer for multiplexing multiple fork*-consistent [10] state transition journals on a blockchain. Application nodes replay their journal to achieve *application-level* consensus at each block b, such that two nodes will agree on a block if and only if the application transactions in that block leave the nodes in an identical state. If their resulting state after executing the operations in b are identical, they generate what we term a *consensus hash* for that block. Consensus hashes enable nodes to independently audit and efficiently query their journals, as well as migrate them between blockchains and detect journal forks.

Virtualchain is implemented in a software library available and ready for production-use. We have used it to implement a global naming and storage system called Blockstack [1], which is one of the largest applications on the Bitcoin blockchain today [13].

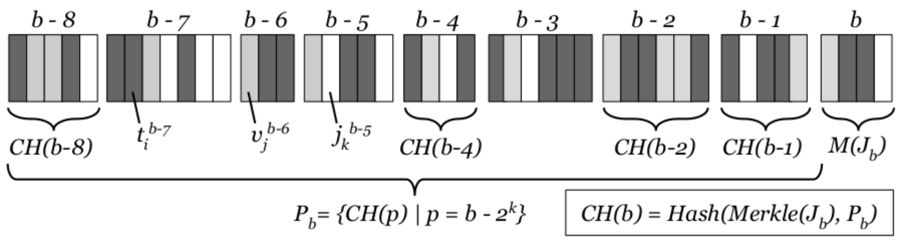

Figure 1: *Overview of how Virtualchain constructs $CH(b)$ from journal transactions. Non-Virtualchain transactions are dark grey (e.g. t_i^{b-7}, meaning "transaction i, packaged in block $b-7$"), journal transactions for different applications are light grey (v_j^{b-6}), and application journal transactions are white (j_k^{b-5}).*

Background. At a high level, blockchains are append-only, totally-ordered replicated logs of transactions [5]. A transaction t is a signed statement that moves tokens owned by a cryptographic keypair. Transactions are causally linked: t_1 happens before t_2 if tokens credited by t_1 are used in t_2. Blockchain peers append transactions by packaging unwritten transactions into a block, and then executing a leader election protocol to determine the next block in the global blockchain.

Most public blockchains use a variant of *Nakamoto consensus* [5], which admits concurrent leaders. Appending conflicting blocks creates *blockchain forks*, which peers resolve using a proof-of-work [9] metric. Transactions in the fork with the most proof-of-work are considered authoritative; conflicting transactions are silently discarded, while non-conflicting transactions are incorporated into subsequent blocks.

Nakamoto consensus gives blockchains the property that longer forks are exponentially rarer if there are no long-lasting network partitions and if most of the compute power is controlled by honest peers [5]. This means that most of the time, transactions are very likely to be durable and linearizable after a constant number of blocks (*confirmations*) have been appended on top of them.

We use these properties to implement fork*-consistent RSMs on top of public blockchains. Application nodes read the blockchain to construct state machine replicas, and submit new transactions to the blockchain to execute state transitions.

Our work differs from prior fork*-consistent systems [10] [8] [7] in three ways. First, we enable open applications: the sets of both application users and application nodes are dynamic and may be empty, since we use an external blockchain to establish the ground truth of the journal and to propagate state transitions. Second, the journal is structured to enable efficient queries on prior state transitions *without* requiring a full blockchain or state machine replica. Third, we utilize a decentralized human-in-the-loop protocol for reconciling fork sets that result from long-lived blockchain forks.

Consensus Hashes. To make progress, nodes scan the blockchain to determine whether or not each transaction represents a valid state transition. Since anyone can write transactions, yet they can be arbitrarily delayed, nodes must be able to select only fresh application state transitions. We achieve this with *consensus hashes*.

A consensus hash is a cryptographic hash that each node calculates at each block. It is derived from the accepted state transitions in the last-processed block, and a geometric series of prior-calculated consensus hashes (Figure 1). Specifically, let $J_b = (j_0^b, j_1^b, ...)$ be the sequence of journal transactions found in block b, let $Merkle(J_b)$ be a function that calculates the Merkle tree root over J_b, and let $Hash(x)$ be a cryptographic hash function. Then, we define $CH(b)$ to be the consensus hash at block b, where $CH(b) = Hash(Merkle(J_b), P_b)$. Block b_0 contains the first journal entry, while P_b is the geometric series of prior consensus hashes starting from b, i.e., the consensus hash for the previous block, two blocks ago, four blocks ago, etc.

Application users include their latest known $CH(b)$ in each transaction they submit, and nodes ignore state transitions with "stale" (too old) or unknown consensus hashes. This way, application nodes ignore forks of their own journal, and application users (or the clients they're using) can tell when to retry lost state transitions. In doing so, consensus hashes preserve the join-at-most-once property of fork*-consistency: a node will accept a state transition with $CH(b)$ only if it has accepted all the prior state-transitions that derived it.

Fast Journal Queries. Not all users will have a copy of the full blockchain on their machine. We use a protocol for fast queries that is useful for creating "lightweight nodes" that do not need blockchain or state replicas. Instead, they can query highly-available but untrusted "full nodes" (which have a full copy of the blockchain) as needed. For example, Blockstack uses this feature to implement its SNV [15] protocol.

For fast queries, application users obtain $CH(b)$ from a trusted node, such as one running on the same host. A user can then use this trusted $CH(b)$ to query previous state transitions from untrusted nodes in a logarithmic amount of time and space. To do so, it iteratively queries and verifies P_b and $Merkle(J_b)$ using $CH(b)$ until it finds $CH(b')$ and $Merkle(J_{b'})$, where b' is the block that contains the state transition to query. Once it has $Merkle(J_{b'})$, it can ask for and verify $J_{b'}$ to get the previous state transitions.

Blockchain Fork Detection and Recovery. If the journal never retroactively forks, the application logic and consensus hashes can preserve the legitimate-request property of fork*-consistency. Retroactive journal forks are highly unlikely, but they *can* occur, since blockchain forks can lose or reorder journal entries. Nodes avoid **short-lived forks** by only accepting sufficiently-confirmed transactions. Applications may increase the number of required confirmations to decrease the likelihood of loss or reordering—for example, Blockstack requires 10 confirmations (in the Bitcoin blockchain).

To detect **long-lived forks**, a node runs multiple processes that subscribe to a geometric series of prior block heights. If a process at a lower height derives a different consensus hash than one from a higher height, then a blockchain fork must have occurred then, and all processes at higher heights have potentially-divergent state. This means all running nodes will be in a separate fork set from bootstrapping nodes.

Reconciling the fork sets requires human intervention, since irreversable actions taken by the application may be based on now-lost state transitions. Fortunately, long-lived forks are rare and severe enough to be widely noticed [2] [3] [4]. This means that when they happen, users and administrators can determine which transactions were affected, and come to concensus about which state-transitions need to be re-sent.

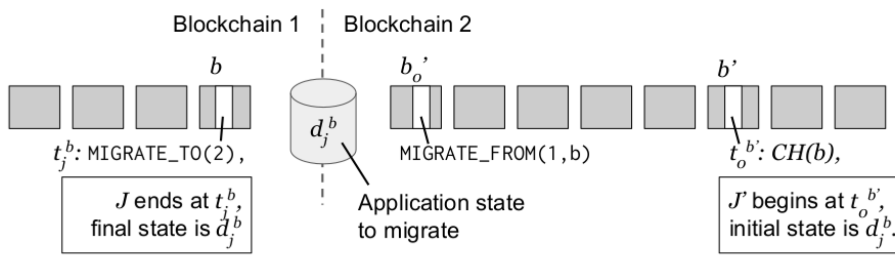
Blockchain 1
Blockchain 2
b
b_o'
b'
d_j^b
t_j^b: MIGRATE_TO(2),
MIGRATE_FROM(1,b)
$t_o^{b'}$: CH(b),
J ends at t_j^b, final state is d_j^b
Application state to migrate
J' begins at $t_o^{b'}$, initial state is d_j^b.

Figure 2: *Migrating from blockchain 1 to 2 at block b is accomplished by using J's final consensus hash from its final transaction t_j^b on blockchain 1 to bootstrap trust in the application's state at j, d_j^b, on blockchain 2.*

To merge the fork sets, the administrators replicate a running node's journal to highly-available file servers (e.g. cloud storage), and send a specially-crafted `CHECKPOINT` transaction that tells nodes the consensus hash of the authoritative journal and where to find the replicas. When a node processes the `CHECKPOINT`, it fetches and validates the off-chain journal and starts building off of it, ignoring all previous transactions.

Accepting a `CHECKPOINT` implies trust in the set of administrators. As such, Virtualchain allows users to whitelist the principals who may send `CEHCKPOINT`s, and users can independently verify their actions by monitoring the blockchains.

Efficient Cross-chain Migration. Migrating from one blockchain to another is similar to repairing journal forks. Doing so requires announcing a future block until which the current blockchain will be valid (no new transactions will be accepted on the current blockchain after that block), and then executing a two-step commit to bind the existing journal to the new blockchain. (Figure 2).

To begin, the application administrator(s) announces a future block after which the current blockchain will no longer be used and sends special "migrate" transactions to both the current and the new blockchain (to announce the migration process). The administrator(s) (a) acquires a lock on the new blockchain, (b) writes the current application state (excluding historic state transitions) to the new blockchain, and (c) releases the lock on the new blockchain and opens up the new blockchain to new transactions. Virtualchain verifies that the migrate transactions are signed by the same principal and verifies that the last-known state on the old blockchain is consistent with the consensus hash announced on the new blockchain. This enables seemless cross-chain migration.

Conclusion. We have presented an overview of Virtualchain, a logical layer for multiplexing an existing blockchain to host multiple fork*-consistent replicated state machines. By using the blockchain to store each application's state-transition journal, and by handling application consensus off-chain, Virtualchain enables applications to use any blockchain for consensus and migrate state between them. Virtualchain is already used in a production system with 55,000 users [1] and is released as open source [16].

References

[1] M. Ali, J. Nelson, R. Shea, and M. Freedman. Blockstack: A global naming and storage system secured by blockchains. In *Proc. USENIX Annual Technical Conference (ATC '16)*, June 2016.

[2] Bitcoin Improvement Proposal 50.

[3] Bitcoin Improvement Proposal 66.

[4] List of Bitcoin CVEs.

[5] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Proc. IEEE Symposium on Security and Privacy*, May 2015.

[6] Ethereum. `http://gavwood.com/Paper.pdf`.

[7] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with Frientegrity: Privacy and integrity with an untrusted provider. In *Proc. 21st USENIX Security Symposium*, August 2012.

[8] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[9] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks*, pages 258–272. Springer, 1999.

[10] J. Li and D. Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proc. 4th USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '07)*, February, 2007.

[11] Litecoin. `https://litecoin.org`.

[12] Namecoin. `https://namecoin.info`.

[13] Statistics of usage for Bitcoin OP_RETURN. `http://opreturn.org`.

[14] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Tech report, 2009. `https://bitcoin.org/bitcoin.pdf`.

[15] Simplified name verification protocol. `http://blockstack.org/docs/light-clients`.

[16] Virtualchain source code release v0.0.13, 2016. `http://github.com/blockstack/virtualchain`.

[17] Why Onename is migrating to the Bitcoin blockchain. `http://blog.onename.com/namecoin-to-bitcoin`.