Actors and Blockchains, Together

Xiaohong Chen[1] and Grigore Rosu[1,2]

[1] Pi Squared Inc., Champaign, IL 61822, USA
xiaohong.chen@pi2.network
[2] University of Illinois, Urbana, IL 61801, USA
grosu@illinois.edu

This paper is dedicated to Gul Agha, leader of the actor model of concurrency, who has taught generations of students that concurrent programming is hard only if you do not have the right model.

Abstract. FastSet is a distributed protocol for decentralized finance and settlement, which is inspired from both actors and blockchains. Account holders cooperate by making claims, which can include payments, holding and transferring assets, accessing and updating shared data, medical records, digital identity, and mathematical theorems, among others. The claims are signed by their owners and are broadcast to a decentralized network of validators, which validate and settle them. Validators replicate the global state of the accounts and need not communicate with each other. In sharp contrast to blockchains, strong consistency is purposely given up as a requirement. Yet, many if not most of the blockchain benefits are preserved, while capitalizing on actor's massive parallelism.

Keywords: Blockchain · Distributed system · Concurrency · Actor model.

1 Introduction

Blockchains are strongly consistent distributed systems in which account holders cooperate by making transactions. Transactions are intents of actions, which need to be given a state to execute in order to materialize. Transactions are signed by their owners and broadcast to a decentralized network of nodes, sometimes called miners or validators. The nodes play dice in order to identify a proposer for the next block (or sequence) of transactions. Nodes replicate the global state and follow a consensus protocol by which the proposed block is processed by all the nodes, thus materializing all the transactions in the block and each node updating its state accordingly. In the end, the transactions submitted to the blockchain are totally ordered, and thus so are the states in between.

Strongly consistent distributed systems [8,6,9,12] "behave like one computer", which is the desirable property for blockchains, but are notorious for higher latency and lower performance and scalability—they require coordination and communication between nodes to maintain consistency. At the other extreme stands the actor model [7,1], aiming at highly parallel computing and massive concurrency. Actors are the basic building blocks of concurrent computation

and communicate only through messages. In response to a message it receives, an actor can modify its own private state, create more actors together with code governing their behavior, and send messages to other actors.

There is no doubt that blockchains play an important role in modern finance. Also, blockchains have demonstrated, in our view irreversibly, that decentralization is not only possible in the world of digital assets, but also very much desirable. Not only decentralization addresses the single point of failure, corruption, and censorship vulnerabilities, but equally importantly, through blockchains, it has lead to the Web3 philosophy and movement: users own their digital assets, from money to diplomas and medical records to pictures and messages, and they and only they can transfer them and decide who has access to what. However, a major overhaul is needed in order to scale blockchains and achieve the level of high performance and low cost required by recent applications. For example, few blockchains can consistently perform more than 1,000 transactions per second (TPS) and it takes seconds, sometimes minutes, to settle a transaction. Metaphorically, because of the total order on transactions that they enforce by their nature, blockchains require the entire universe to squeeze through a narrow pipe. Completely unrelated transactions are required to stay in line and wait to be sequenced in some order that the blockchain must globally choose when forming its next block. This is clearly not scalable, even if all transactions are initiated by humans. The situation is in fact much worse, because AI and AI agents doing transactions on humans' behalf are already here to stay and they require a payment system able to perform millions of TPS, most of which micro-transactions whose cost is expected to be negligible, in the order of fractions of a cent. Blockchains cannot do this. A massively parallel decentralized infrastructure for payments in particular and computing in general is required.

A series of papers before 2008 culminating with Bitcoin [11], have incrementally built a belief that a total order on transactions ought to be required in any distributed/decentralized payment system in order to avoid the infamous double spending attack: an account sending two concurrent transactions attempting to spend the same coin with two different recipients. A total ordering enforced on transaction settlement indeed solves the double spending problem, but is it really necessary? Recent works starting around 2019 [4,5,2] propose a radically different way to approach the problem, a truly concurrent approach where payments can be generated and settled in different orders by different nodes without breaking the semantics of payments. The key insight of these works is that the order in which an account receives payments is irrelevant, and so is the order in which different accounts send payments—provided each account stays valid: no double spending and sufficient balance. That is, as far as the nodes/validators are in agreement on the *set* of locally valid transactions that took place in the system, consensus on a total order is unnecessary. We believe that this apparently simple and innocent observation marks a crucial breakthrough moment in decentralized finance. A moment where the tyranny of sequentiality is abolished and the door is open to innovations that will lead to the next generation of decentralized, yet truly concurrent, safe and scalable infrastructure for digital assets.

Inspired by this recent work on weaker variants of consensus in the context of cryptocurrencies [4,5,2], as well as by the unbounded concurrent computing promise of the actor model [7,1], in this paper we discuss at a high-level FastSet, a general-purpose distributed computing protocol that generalizes the payment system protocol FastPay [2]—we encourage the reader to consult [3] for a detailed presentation of FastSet. FastSet performs nearly embarrassingly parallel settlement of arbitrary verifiable claims, including verifiable computations in arbitrary programming languages. Specifically, FastSet allows a set of *clients* to settle verifiable *claims* consistently, using a set of *validators*. Clients broadcast their claims to all validators. Validators validate the received claims and replicate the system state based on the order in which they receive the claims. Importantly, similarly to FastPay but in sharp contrast to blockchains, the FastSet validators do not need to communicate with each other, nor directly achieve consensus on any values or orders or blocks or computations.

What makes the problem difficult is that claims can have side effects on the validators' states. However, if claims issued by different clients are *weakly independent*, a notion that generalizes the property of commutativity on payments [4,5,2] discussed above to arbitrary state-effectful computations, then the validators' states will remain (strongly eventually) consistent in spite of the different orders in which they receive and process the claims.

Like in blockchains, FastSet accounts have a state (e.g., a balance) and are required to sign any claims they issue. We present the protocol in Section 2, requiring only abstract notions of state, claim, and claim validity and effect. How claims are generated, e.g., randomly by users or programmatically by contracts, is irrelevant for the core protocol and its correctness (Theorem 1). However, to make it practical, implementations of FastSet have to offer specific types of claims and specific ways to generate them. In Section 3 we propose an actor-inspired language for generating claims, which we call the FastSet Language, abbreviate SETL[3], and pronounce "settle". Some accounts are controlled by users, others by SETL programs (or scripts, or contracts). The difference is that the user-controlled accounts are free to issue any claims, including ones that create new accounts and interact with them, while the contract-based accounts can only issue claims as prescribed by their SETL program/script/contract.

Like in the actor model, accounts are regarded as actors that communicate only with the actors they know about, and can create new accounts/actors and then communicate with them. Since validators maintain replicas of the global system state, all the actors/accounts are also replicated on all validators. This should not be regarded as a deviation from the underlying thesis of the actor model, where each actor is meant to be a separate process interacting concurrently with the other actors, but rather as a high-availability high-resilience decentralized implementation of an actor system. Moreover, the actor model is particularly suitable for a language like SETL for two additional reasons: (1) each validator can itself be a high-performance concurrent system, which re-

[3] Not to be confused with the SET Language `https://en.wikipedia.org/wiki/SETL` invented in late 1960s, based on the mathematical theory of sets.

ceives and processes potentially millions of claims per second, so SETL must be a high-performance concurrent programming language; and (2) since each validator receives the claims in different orders, yet all of them (strongly eventually consistently) are expected to replicate the same state, SETL concurrent programs must be easy to reason about, in particular to prove their determinism.

2 The FastSet Protocol

In this section we describe the FastSet protocol, depicted in Figure 1.

2.1 Participants

FastSet involves two types of participants:

Clients. These are account/address holders, who can make *claims*. They can be users, apps (contracts, web2, games), L1s, L2s, micro-chains, AI agents, service providers, execution engines/layers, provers (mathematical, ZK), TEEs, oracles, VRFs, AI compute/inferencers, indexers, history query providers, verifiers (execution, semantics and/or ZK proof based), fact claimers (KYC providers, digital identity providers, academic or medical records, etc.), token issuers (stablecoins, ERCs, NFTs, RWAs, etc.), etc.

Validators. These process claims made by the clients, while at the same time maintaining consistency in the global knowledge about the overall system: balances of all tokens and additional data/state of all clients, global state of the entire protocol, like the set of all the claims that were settled, etc.

All participants possess a key pair consisting of a private signature key and the corresponding public verification key. If *msg* is a message and p is a participant, then $\langle msg \rangle_p$ denotes the message signed by p, whose authenticity can be publicly checked. Like in blockchains, in FastSet the public key of a client a serves as the public account number, or the address of a. Additional layers of privacy can be added if desired, but that is out of the scope of this paper.

2.2 Claims and Weak Independence

We define FastSet parametrically, on top of two important abstractions, *claims* and their *weak independence*, which we discuss next.

A *claim* is any statement that is independently *verifiable*. To help the verifiers, the claim provider may associate additional evidence to the claim, such as a proof or a witness or even an authoritative signature; how claims provide evidence and how verifiers verify a claim is orthogonal to FastSet and is not our concern in this paper. Here are some examples of claims: "I am Joe Smith" (a fact where the signer is important); "Pythagoras theorem" (a fact where the signer is less important); "the price of gold is 100 USD" (an oracle); "the next random number is 17" (a VRF); "I want to buy a ticket to this concert" (an intent); "the result to your query is 42" (an AI service provided); "Python program

`fibonacci` on input 10 evaluates to 55" (verifiers may re-execute, or require a math or ZK proof based on Python formal semantics); "my Angry Birds score is 739" (requires 3rd party or ZK proof); "my next move in this chess game with Alice is Nf3" (modifying a shared storage location sequentially); "I vote YES for that petition" (modifying a shared storage location non-sequentially); "I, Grigore, pay Alice 10 USD" (a payment, modifying two storage locations).

For simplicity and generality, we only assume a global state in FastSet, which will be replicated in each validator. In practice, each client will have their own reserved state space, but that stronger assumption is not needed to prove the correctness of FastSet; all we need is that claims issued by different clients are weakly independent, to be discussed shortly. What is important is that a claim may or may not be valid in a given state (e.g., a payment is invalid when the sender's balance is not enough), and that the processing of a claim can and usually does change the state (e.g., even an otherwise side-effect-free claim may be counted, at a minimum). We write $c{\downarrow_s}$ whenever a claim c is valid in a state s, and in that case $[\![c]\!]s$ denotes the state obtained after processing c in s. We extend these notations to sequences of claims $c_1\, c_2 \ldots c_k$ as expected: $c_1\, c_2 \ldots c_k{\downarrow_s}$ means that each claim in the sequence is valid in the state obtained after processing the previous ones, and $[\![c_1\, c_2 \ldots c_k]\!]s$ is the state obtained after processing the entire sequence. The root of difficulty in FastSet, as well as in concurrent and distributed systems in general, is the fact that claim sequences issued by different interacting clients may arrive to validators interleaved in different ways, which may make their states diverge inconsistently.

Drawing inspiration from concurrency theory (e.g., Mazurkiewicz traces [10]), where two events are independent iff they can be processed in any order with the same result, we say that two claims c and c' are *weakly independent*, written $c \parallel c'$, iff once each of them is independently valid, they can be processed in any order and the final result is the same: $c \parallel c'$ iff for any state s, if $c{\downarrow_s}$ and $c'{\downarrow_s}$ then $c\, c'{\downarrow_s}$, $c'\, c{\downarrow_s}$, and $[\![c\, c']\!]s = [\![c'\, c]\!]s$. Therefore, we weaken the classic notion of independence by requiring it to hold only in those states in which both claims are already valid. This assumption is critical for FastSet because it allows to handle payments and asset transfers as particular claims, and thus generalize FastPay [2]. Here are some examples of weakly independent claims:

- Totally (not weakly) independent claims whose validity has nothing to do with each other, such as: facts which are independently verifiable/true; state queries which are pure, that is, return results but have no side effects; state accesses, including writes/updates, but of disjoint storage locations.
- Commutative operations on a shared location: writing the same value, like in setting a one-way flag to "true"; adding numbers, positive only like in voting or both positive and negative like in reputation systems (e.g., the karma system of Reddit); multiplying numbers, like in aggregating signatures; adding elements to a set, like in signing petitions, bidding in an auction, or submitting (an intent to make) a transaction; adding elements to a canonical data-structure, like to a sorted list (which is re-sorted after each addition).
- Updates of CRDTs, abbreviating Commutative [14,13] or Conflict-free [15] Replicated Data Types, which are data-types used in the context of dis-

tributed systems with the property that concurrent updates on them commute. CRDTs were used mainly in concurrent text editing, but the concept is general and includes many interesting examples, such as integer vectors, sets, maps, and even graphs (with some reasonable restrictions).
– Payments and, more generally, asset transfers from different clients.

Payments are, in particular, a very important use case of FastSet. As critically observed by the authors of FastPay [2], payments have the key property of commutativity on the recipient. Specifically, if Charlie receives payments from both Alice and Bob, then Charlie will end up with the same balance/state no matter in what order the two payments from Alice and Bob are received. And so will Alice and Bob. It can be easily seen that any two payments made by distinct clients are weakly independent. But they are *not* necessarily independent with the standard notion of independence [10]. For example, the claims "Alice pays Bob 1 USD" and "Bob pays Charlie 1 USD" are (weakly independent but) not independent: indeed, consider a state in which Alice has 1 USD and Bob has 0 USD. The same state also demonstrates that two payments made by the same client are not necessarily weakly independent, e.g., "Alice pays Bob 1 USD" and "Alice pays Charlie 1 USD". The weak independence of payments by different clients says that the overall state of the system will be the same independently of the order in which they are processed, provided that the initial state of the system allows for each payment to be independently made.

Non-examples of weak independence include claims acquiring the same mutex, or arbitrary accesses of a shared location where at least one is a write. In particular, and this might look surprising at first sight, an exact balance claim is *not* weakly independent with payment claims to that account. For example, Alice's claim "I have 20 USD" is not weakly independent with payment claims made by others to Alice. On the other hand, monotonic balance claims of the form "I have at least 20 USD" are weakly independent with payments made by others. A challenge for FastSet implementations is to determine what constitutes claims on their network and how they want to enforce weak independence.

2.3 Assumptions

In its most basic form, FastSet's clients and validators form a bipartite graph, where each client is connected to each validator but the clients and, respectively, the validators, are not connected to each other. Specifically, the only communication required by the basic protocol is for each client and each validator to be able to send signed messages to each other. In practice, however, some clients will likely communicate with each other outside of the protocol in order to orchestrate and optimize their individual communication with the validators. For example, a user's AI agent may require an approval from the user before buying a ticket; instead of communicating exclusively through claims and waiting for each of them to be settled by FastSet before proceeding to the next step, the user and the agent can both send their claims in parallel and settle the transaction faster and cheaper. Similarly, in practice validators will likely have

mechanisms to communicate with each other in order to synchronize their states faster than waiting for clients to send them the missing certificates. However, such side communication mechanisms are outside the scope of this paper.

At a high-level, FastSet works as follows. At any given moment, any client can broadcast to validators an ordered block of claims in a signed message. Each validator checks the validity of each block of claims received from each client, and if everything checks the validator approves the block by signing the client's message and sending it back to the client without updating its state yet. The client collects approvals from validators and once it reaches quorum it broadcasts a confirmation message to all validators. Once a validator receives the confirmation it proceeds to updating its state. Special care must be paid by the validator to order the state updates to avoid inconsistencies. Eventually, all validators will consistently settle all the valid claims made by all the clients.

The validators do not question, nor check the intent behind or the client-specific semantics of the client's claims. The validators only check the claims' validity and settle them if valid, operations which are expected to be very efficient; indeed, in practice these operations involve no computations other than updating client balances and storage locations with given values. In other words, from validators' perspective the client claims are just simple commands that they must comply with after a few sanity checks. Any computations using programs or smart contracts in various programming languages and VMs happen within the clients, using their resources and not validators'. It is not validators' concern whether client's computations are adequate or correct according to client's intended semantics or specs or terms and conditions or whatever promises they may have made to their clients. All the validators need to know is the block of claims that the client issues as a result of those potentially complex computations. As an analogy, the validators play the role of an operating system (OS) in a computer, whose job is to ensure consistency across the various programs being executed; the programs are the clients of the OS and all they do from OS's perspective is to issue ordered blocks of commands (the claims).

FastSet therefore gives clients ultimate freedom in what claims they can issue. Without any additional verification, it is quite possible that some clients may make mistakes, some of them even maliciously. For example, the client may hold user assets, such as in the case of a bank, or a centralized exchange, or a blockchain. That is, users may send their assets to the client in expectation of some services. Without any additional verification, there is nothing to prevent the client from stealing users' assets. Even without any evil intent, a client's account may be operated by a complex program which may have subtle errors (such as unintended non-determinism) and thus sometimes produce an incorrect sequence of claims. Even without any program at all, a user client like Alice may mistype 11 instead of 1 when sending a payment to Bob, in which case additional verification, like a multi-sig, would have helped.

There is no silver-bullet solution to cover all types of claim verification that clients or applications need or will ever need. To continue to give clients maximum flexibility, yet to allow the honest clients to verifiably prove themselves to

their users, FastSet provides the capability for clients to set up verifiers and a quorum among those in order for its claims to be considered valid by the FastSet validators. Verifiers are themselves clients on the network, whose role is to provide specialized verification services. For example, a verifier can be specialized in verifying program executions in EVM (or Python, or Java, etc.) and clients, e.g., blockchains (or AI agents, or exchanges, etc.), can use it. Such a verifier may re-execute the EVM program to check the result, or may verify a proof produced by an external compute, such as a ZK proof or a TEE proof or even just a signature by some trusted authority (e.g., a centralized exchange which offers services to its KYC-ed clients). It is the client's full responsibility to decide what verifiers to include in their set and what quorum is sufficient.

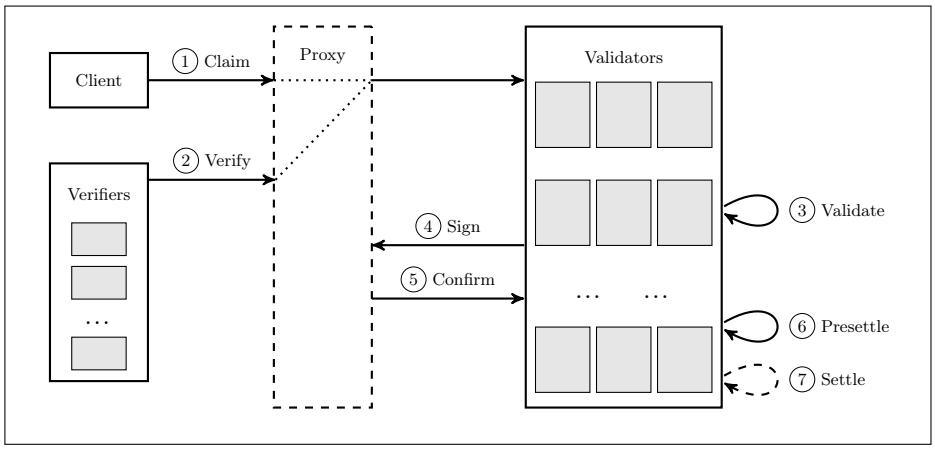
Implementations of FastSet may optionally provide a simple hardwired programming or scripting language to generate sequences of claims. Programs, or smart contracts, or scripts in this language would be executed by the validators on behalf of clients. That is, instead of providing a block of claims to each validator, a client would provide a script that instructs validators how to generate the block of claims on their behalf, possibly using a client-provided input, at the cost of paying some gas fees to the validators. In this case, clients must pay special care to ensure that each validator deterministically generates the same claims in the same order. Section 3 presents a candidate scripting language.

To simplify the presentation of FastSet and to uniformly capture more use cases, we assume the existence of a proxy that facilitates the interaction between a client, its verifiers, and the validators. The proxy is just a helper which only affects the liveness, but not the safety of the protocol. All it does is to forward or broadcast messages signed by the various participants, and to blindly aggregate signatures ($\#\{sig_1, sig_2, ...\}$). Should the proxy fail to do its job for whatever reason, anybody can replace it and redo the tasks that were missed, including the client. Proxy can be a client (the sender of the original message containing the claim sequence, or a different client such as a beneficiary or a verifier or a service provider), or a validator, or any combination of these for all or for any of the two services it provides, namely broadcasting client messages and aggregating validator signatures. That is, there could be zero, one, or more proxies for each client or application deployed on FastSet. If zero, then the client initiating the claims can do the job of the proxy, or any of the validators.

An honest validator always follows the FastSet protocol, while a faulty (or Byzantine) validator may deviate arbitrarily. Assume $3f + 1$ validators, with a fixed unknown subset of at most f Byzantine validators. A *quorum* is defined as any subset of $2f + 1$ validators. A protocol message signed by a quorum of validators is *certified* and called a *certificate*.

2.4 FastSet

The validator's goal is to consistently replicate the global state, while at the same time offer services through an API. Each validator v is assumed to maintain the following data at a minimum, and to offer it through its API at request:



Proxy is a helper and only affects liveness, not safety. It can be a client (the sender of the original message containing the claim sequence, or a different client such as a beneficiary or a verifier or a service provider), a validator, or any combination of these for all or for any of the two services it provides: broadcasting messages to validators and signature aggregation ($\#\{sig_1, sig_2, ...\}$).

(1) (2) Client a claims $c_1\, c_2 \,...\, c_k$ by signing message $m \coloneqq \langle c_1\, c_2 \,...\, c_k, n\rangle_a$, where n is its local nonce (incremented after each message). An appropriate set of verifiers $T \coloneqq \{t_1, t_2, ..., t_l\}$ approve m by signing it: $\langle m\rangle_{t_1}$, $\langle m\rangle_{t_2}$, ..., $\langle m\rangle_{t_l}$. Proxy aggregates the verifiers' signatures and broadcasts $\langle m\rangle_{\#T}$ to all validators.

(3) Upon receiving $\langle m\rangle_{\#T}$ with $m = \langle c_1\, c_2 \,...\, c_k, n\rangle_a$, validator v validates m by checking: signature of a is valid; $v(a).nonce = n$; $v(a).pending \subseteq \{m\}$; the verifiers' aggregated signature $\#T$ is valid and reaches the client-specified quorum; and finally, $c_1\, c_2 \,...\, c_k \downarrow_{v.state}$.

(4) Validator v signs m, sends $\langle m\rangle_v$ to proxy, and sets $v(a).pending \coloneqq \{m\}$.

(5) Once a validator quorum is reached on message m, proxy creates certificate $\langle m\rangle_{\#}$ and broadcasts it to all validators.

(6) Upon receiving $\langle m\rangle_{\#}$, each validator v: checks the quorum $\#$; checks $m \notin v.presettled \cup v.settled$; adds m to $v.presettled$. Note that this m in Step 6 can be different from the m in Step 4, even for the same client a.

(7) Settle any message in $v.presettle$ that can be settled: if $m = \langle c_1\, c_2 \,...\, c_k, n\rangle_a$ in $v.presettled$ with $v(a).nonce = n$ and $c_1\, c_2 \,...\, c_k \downarrow_{v.state}$ then $v.state \coloneqq [\![c_1\, c_2 \,...\, c_k]\!]v.state$, increment $v(a).nonce$, reset $v(a).pending \coloneqq \emptyset$, move m to $v.settled$. This step 7 can and should be executed continuously.

Fig. 1. FastSet protocol.

- A state $v.state$, which is its version of the global state that is replicated in each validator. In implementations of FastSet the state may be partitioned in objects and storage locations owned by or associated to different clients, as well as shared objects and locations, but this distinction is not necessary in the presentation of the core protocol, because the weak independence abstraction between claims by different clients suffices. Each validator may have a different state at any given moment due to message delays and to the massive parallelism allowed by FastSet. Yet, we will show that the various state replicas are consistent and they eventually converge.
- Sets $v.settled$ and $v.presettled$ of client submitted messages, which collectively capture all the knowledge that v has about the claims that have achieved validator quorum in the network. The claims in (the messages in) $v.settled$ have already been processed by v, meaning that their effect on $v.state$ has already been applied and v is ready to process the next block of claims by the same client. The claims in (the messages in) $v.presettled$ have not been processed yet by v, but will be processed as v is ready for them. We will show that all messages in $v.presettled$ will be eventually processed and thus moved to $v.settled$.
- For each client/account/address a some data $v(a)$ including: $v(a).nonce$, a sequence number counting the claim settlement requests by a; $v(a).pending$, a temporary placeholder containing at most one (latest valid) message received from a. FastSet implementations may hold more client-specific data. Immediate candidates can be $v(a).verifiers$, a set of verifiers that a uses to verify its claim blocks, especially when it uses the same verifiers for all messages, and related, $v(a).quorum$, a number of verifiers that a considers sufficient in order for its claim blocks to be considered verified.

Figure 1 shows how FastSet works. It is a seven-step protocol.

(1) Client a broadcasts a block of claims $c_1\, c_2 \ldots c_k$, paired with its local nonce n, in a message $\langle c_1\, c_2 \ldots c_k, n\rangle_a$ that it signs, say m. The nonce helps the validators synchronize with the client, making sure that they execute the client's claims in the intended order. An honest client should never submit two different claim blocks with the same nonce, because that is the FastSet equivalent of a double spending attempt. Regardless, the protocol will ensure that the client will only be able to obtain validator quorum on at most one claim block per nonce. A client submitting multiple claim blocks with the same nonce risks to get stuck in a pending status with the validators and never achieve quorum to make progress. We do not treat this situation here, but implementations of FastSet may choose to allow clients to clear their pending status for a fee, to give them an exit situation in case their programs have errors but at the same time to discourage them from attempting to break or slow down the protocol.

(2) Validators will not attempt to validate any received claim block before it accumulates an appropriate number of verifiers which approve it. This information being public, verifiers and clients have multiple ways to synchronize to get this task done. The simplest, but also the least efficient and riskiest for the client, is for the client to just submit its message and assume that its verifiers

listen to some validator and will thus get informed when the client message was settled, then verify it, sign it and send it to the proxy which then aggregates all their signatures and sends them to the validators. A more efficient approach is for the client to inform its verifiers as soon as it sends the original message, so they can start their verification process immediately. An even more efficient approach in some applications could be for the client to send its verifiers its intent to execute a program, also known to the verifiers, so that they can start their verification while the client is still working on generating its claim block, allowing program execution and its verification to take place in parallel.

(3) Once the validator v has been informed that a client's claim block $c_1\, c_2 \ldots c_k$ accumulated the quorum of verifiers, it can proceed to validating it. The obvious and cheap checks are the signatures, the nonce, and the verifier quorum. We have deliberately left open the decision on where the verifiers and the verifier quorum are stored: some implementations of FastSet may store them in the validator, e.g., $v(a).\mathit{verifiers}$ and $v(a).\mathit{quorum}$, in which case the check becomes $|T \cap v(a).\mathit{verifiers}| \geq v(a).\mathit{quorum}$; others may store them as additional items in the message, besides the claim block and nonce; others may include a special claim in the block, say at the end, claiming the verifiers and quorum. In all cases, however, the verifiers and their quorum are part of the contract that the application makes with its users, so it is important that the validators enforce them. The validator also checks that there is no different message sent by the same client with the same nonce that is pending, that is, it checks $v(a).\mathit{pending} \subseteq \{m\}$. In other words, if there is any message pending then it must be the exact same message m. This ensures that the client does not, intentionally or not, attempt to initiate double-spending attacks or non-deterministic computations. Note that messages from validators can be delayed or lost, so the client or its proxy may attempt to submit m multiple times until they receive validator confirmation, which is why $v(a).\mathit{pending}$ is allowed to already contain m. Finally, validator v is ready to check the validity of the actual claim block $c_1\, c_2 \ldots c_k$ in its state $v.\mathit{state}$, that is, $c_1\, c_2 \ldots c_k \downarrow_{v.\mathit{state}}$. Note that the validator does not update its state at this stage. Indeed, the block has not achieved validator (not verifier) quorum yet, which means that the client may still possibly achieve quorum on a different block with other validators.

(4) Validator v now approves the claim block by signing the original client's message and sending it back. The validator also sets $v(a).\mathit{pending}$ to $\{m\}$, so from here on it will accept no other messages from a until m is settled. If the validator received the same message m from a that it approved already, it sends its approval again to account for previous messages having been potentially lost.

(5) When the proxy accumulates validator quorum on message m, it creates a certificate $\langle m \rangle_{\#}$ by aggregating all the received validator confirmations and broadcasts it back to all validators. This certificate is proof that quorum has been achieved and at this moment the state updates in all the validators are imminent, i.e., cannot be stopped anymore once they receive the certificate.

(6) When a validator v receives such a valid certificate $\langle m \rangle_{\#}$, it knows with certainty that a quorum of validators have already validated m, so it is safe to

process and settle m. But that needs to be done only once, to avoid applying the same updates twice or more, so v first checks that m is not already settled, or considered to be settled, i.e., presettled. If that is the case, then v should settle m. However, v may not be ready to settle m, in the sense that the nonce or the claim block of m may not be valid for v. In fact, there is no guarantee that v even participated in the quorum of $\langle m \rangle_{\#}$, or that v is even aware of the existence of the client that originally sent m, because that particular client and v could have been disconnected by now. Taking into account all these situations, v only adds m to $v.\mathit{presettle}$ for now.

(7) The liveness result, 4 in Theorem 1, says that if other validators settled more claims than v, then v should also be able to make progress and settle more claims. That is, if v continuously scans $v.\mathit{presettle}$ it will eventually find messages which are valid and thus can be settled, because FastSet assumes that all messages are eventually delivered. Implementations of FastSet will likely include mechanisms for validators to request updates of missing settled claims from other validators and populate their presettled set with them in order to make faster progress. Note that right before m is settled in this step 7 of the protocol, $v(a).\mathit{pending}$ can be either empty or $\{m\}$. The former happens when v did not get a chance to participate in the quorum on m, so v never checked $c_1\, c_2 \ldots c_k \downarrow_{v.\mathit{state}}$ (e.g., if a had enough balance to make a payment); however, since $v(a).\mathit{nonce} = n$, v is ready to settle m as soon as $c_1\, c_2 \ldots c_k \downarrow_{v.\mathit{state}}$, and it does exactly that. The latter happens when v has already signed m and was waiting for a quorum including other validators as well to be formed. The monotonicity result, 3 in Theorem 1, says that it is safe to settle m as is, that is, checking $c_1\, c_2 \ldots c_k \downarrow_{v.\mathit{state}}$ is theoretically unnecessary. However, we choose to keep this definedness check. As seen in [3], implementations of FastSet may choose to allow applications to disobey the weak independence requirement. Keeping the definedness check will guarantee that the validator's state stays well-defined, the damage being contained to possibly locking the non-conforming accounts.

Theorem 1. *[3] Under the assumptions in Section 2.3, FastSet is correct, i.e.:*

1. *Security* *All certificates generated at Step 5 are consistent, that is, only one per client per nonce. Therefore, FastSet prevents double-spending.*
2. *Determinism* *The order in which a validator receives the messages is irrelevant.*
3. *Monotonicity* *Once a client can settle a message, it will continue to be able to settle it regardless of what other clients do.*
4. *Liveness* *Validators do not get stuck: if one makes progress, then all do.*

3 Examples and Applications

FastSet was defined parametrically in Section 2, on top of abstract notions of claim and state. However, to implement FastSet and build applications, we need a claim language and concrete states. Such a candidate language inspired from

actors is proposed in [3], called SETL, together with a collection of typical Web3 programs: multi-sigs, voting, escrow, verifiable computing, custom assets and payments, auctions, app-chains and blockchains. SETL programs are called *scripts* because they are expected to be small, although some scripts can be non-trivial—due to the massively parallel nature of FastSet and the fact that validators may have different states at the same time. Here we only discuss three such programs at a high level, voting, digital assets and auctions, to illustrate the practicality of FastSet and its relationship to the actor model.

We remind the reader that the most common blockchain functionality, that of a payment system over a native token (e.g., Bitcoin), is supported by default as explained in Section 2.2. SETL provides a primitive `transfer(to,value)` for this; e.g., if `alice` wants to transfer 3 native tokens to `bob`, then she submits and settles the claim $\langle$`transfer(bob,3)`$\rangle_{\texttt{alice}}$. We will explain SETL on the fly, on a by-need basis in order to understand the three examples.

3.1 Voting

Voting is common on blockchains, yet it is an inherently parallel activity in which the order of votes is irrelevant, making FastSet ideal for it. An authority needs to set up and create a vote poll and then allow voters who are verified to vote. Finally, the authority should announce the results.

Let us consider a simple voting scenario, where the authority, say `gov`, who is also responsible for verifying the voters, wants to accumulate `MIN` valid votes to pass some `MOTION`. Then `gov` can create a contract using the SETL script:

```
BASIC_VOTE[MIN]:
    voted : Set;  // users who voted
    constructor {
        voted := empty;
    }
    instance constructor {
        not(instance.owner in voted); // guard: must be true
        verify(contract.owner, 1);    // authority verifies voter
        voted.add(instance.owner);
    }
    passed() { size(voted) >= MIN; }
```

The script above is a code template parametric on `MIN`. It has a contract field `voted`, which is a set that accumulates all the users that voted. The contract constructor block initializes `voted` to empty. The contract can generate two more types of blocks, one which is an instance constructor and the other, `passed()`, which is reserved to the contract owner. These are discussed below.

To create a contract, `gov` has to sign a message effectively claiming a contract with the code template above instantiated with some value, say 1000. Let `MOTION` be the unique identifier (i.e., a hash or number) associated to this claim, that is:

$$\texttt{MOTION} = \langle \texttt{contract(BASIC\_VOTE[1000])} \rangle_{\texttt{gov}}$$

All a voter has to do in order to vote is create an instance of this contract:

$\langle$`instance(MOTION)`$\rangle_{\texttt{alice}}$
$\langle$`instance(MOTION)`$\rangle_{\texttt{bob}}$
$\langle$`instance(MOTION)`$\rangle_{\texttt{charlie}}$

As the code shows, when an instance is created, the `instance.owner` is verified by `gov` and added to the set of `voters`. In SETL, `verify` takes a set of verifiers (first argument) and a quorum (second argument) to sign the block as part of Step 2 of the FastSet protocol (Figure 1). The guard in the instance constructor ensures that no other instance of the same contract can be created by the same voter, that is, each voter can only vote once. Indeed, the semantics of guards in SETL is "ignore if true, invalidate the block if false".

The contract owner can announce the result whenever `MIN` is reached:

$\langle$`MOTION.passed()`$\rangle_{\texttt{gov}}$

The toy voting script above was overly simplified on purpose, to highlight the concurrent programming capability of FastSet and SETL, and their relationship to the actor model. In practice, voting contracts will likely separate vote registration from voting itself, allow for several voting options, have weighted votes, a deadline, etc. A contract creation by `contract.owner` effectively creates an actor, whose state is the set `voted` initialized to empty. By the semantics of SETL, only the contract owner can send `passed()` messages to the contract actor. Indeed, in SETL, `instance` fields form the state of the contract instance actors, and these instance actors can only initiate `instance` blocks according to the contract. The fields which do not have an `instance` modifier form the state of the contract actor, and the blocks which do not have an `instance` modifier can only be generated by the contract owner. In SETL, anybody can create instances of the contract. In our case here the instances have no state, because there are no instance fields, but they all execute the instance constructor at creation. Effectively, each instance is an actor controlled by the `instance.owner`. Instance actors communicate with the contract actor by means of the `voted` contract field—its monotonic updates can and should be regarded as messages sent by the instances to the contract. Faithfully to the actor model, actors only communicate with actors they have knowledge about.

A clarification is needed with regard to the fact that owners of contracts and of instances, respectively, can sign on their behalf. In Section 2, FastSet stipulates that each account a can only submit claim blocks which are signed by a. In SETL, on the other hand, each contract creation and each instance of it becomes an actor, which has its own state and address. However, it would be rather inconvenient and likely insecure and unmanageable by users to allow each actor to own its own private key to sign all its claim blocks. Instead, SETL allows the contract-based actors to only be controlled by their owners, technically just other accounts that have a private key and thus can sign messages. That is, the unique owner of the created contract, or the unique owner of some instance of it, is the only entity that can sign blocks on behalf of the address corresponding to

the associated contract or instance actor. This is not a modification of the FastSet protocol, but rather a particular implementation of it. Indeed, a single private key can be used to generate infinitely many public keys through various methods, like hierarchical deterministic (HD) wallets or different cryptographic algorithms. These different public keys are deterministically derived from the same private key and are used as contract-based actor addresses in SETL, although there is only one initial pairing between a private key and its primary public key.

We would like to take the opportunity to further discuss this simple voting contract below, because it touches upon a few important aspects of FastSet that make it different from blockchains.

First, unless `gov` maintains a list of voters they verified off-set, it is impossible to precisely know the number of votes. Indeed, even if `alice` reaches validator quorum on $\langle$`instance(MOTION)`$\rangle_{\texttt{alice}}$, she or her proxy may have delayed sending the certificate back to validators, so she might have never been added to the set of `voters` on some validators. Even if that certificate has been sent, some validators may be delayed in their execution of the Step 7 of FastSet and have not updated their state yet. The crucial guarantee, however, is that no validator will settle $\langle$`instance(MOTION)`$\rangle_{\texttt{alice}}$ twice. If voters need to prove that they indeed voted, the contract can be modified by adding a new block

```
instance voted() { instance.owner in voted; }
```

and now `alice` can settle $\langle$`motion_alice.voted()`$\rangle_{\texttt{alice}}$ and use its certificate as proof—here `motion_alice` is the instance $\langle$`instance(MOTION)`$\rangle_{\texttt{alice}}$.

Similarly and for the same reasons, it is impossible for `alice` to know with certainty whether her vote was actually counted, that is, that it contributed to claim $\langle$`MOTION.passed()`$\rangle_{\texttt{gov}}$ being settled. This is the case even if `alice` settled a claim $\langle$`motion_alice.voted()`$\rangle_{\texttt{alice}}$ like discussed above chronologically before $\langle$`MOTION.passed()`$\rangle_{\texttt{gov}}$ on some validator(s) that she uses to observe the voting process. Since chronological order is useful in cases like above, and for other reasons, [3] extends FastSet and SETL with claim timestamping: claim issuers timestamp each claim block, and validators check and approve the timestamp.

Finally, it is insightful to note that although it may appear that a (malicious or not) voter can attempt to create two different instances and submit them concurrently to the validators in the hope that they both get settled, that is not possible thanks to the fact that each account has a nonce and at most one message is allowed in a validator's pending for any given account (steps 3 and 4 in Figure 1). A voter attempting to create two concurrent (same nonce) instances would therefore risk to get its account stuck. There is nothing to prevent voters from attempting to create two instances in sequence, each issued with a different nonce, but in that case they are processed in order and the second one will fail on all validators as soon as they processed the first one, as intended.

3.2 Non-Native Tokens and Digital Assets

Below is a simple contract template that mints a token only at contract creation time and the total supply is assigned to the contract owner. We expect

token contracts to be more involved, with flexible minting and verifiers for it (e.g., `gov`, `bank`, etc.), possibly with approvals and transfer-from capabilities like in Ethereum ERC20 tokens. Our goal here is to keep it simple in order to demonstrate that tokens can be very efficiently supported by FastSet.

```
TOKEN[NAME,SUPPLY]:
    name    : String;
    balance : Address -> Int;
    constructor {
        name := NAME;
        balance[contract.owner] := SUPPLY;
    }
    instance transfer_token(to,v) {
        v > 0;  // guard: must be true
        balance[instance.owner] >= v;
        balance[instance.owner] -= v;
        balance[to] += v;
    }
```

Here, we name the non-native token transfer function `transfer_token`, so that it is not confused with the native transaction function `transfer`.

One can now create a `TOKEN` contract and make a first transfer as follows:

$$\texttt{USDC} = \langle \texttt{contract(TOKEN["USDC",1000000])} \rangle_{\texttt{circle}}$$
$$\texttt{circle\_wallet} = \langle \texttt{instance(USDC)} \rangle_{\texttt{circle}}$$
$$\texttt{alice\_USDC} = \langle \texttt{instance(USDC)} \rangle_{\texttt{alice}}$$
$$\langle \texttt{circle\_wallet.transfer\_token(alice,1000)} \rangle_{\texttt{circle}}$$

Note that the contract owner, `circle`, also needed to create an instance of the contract in order to gain access to claim blocks reserved for instances. The recipient of the transfer, `alice`, also happened to create an instance wallet, although that was not needed in order to receive transfers. Indeed, even if `bob` does not have an instance of `USDC`, the following is still possible:

$$\langle \texttt{alice\_USDC.transfer\_token(bob,100)} \rangle_{\texttt{alice}}$$

All the non-native tokens and assets are stored in the contract, in its `balance` field. Whenever `bob` wants to get access to the 100 tokens that `alice` transferred to him, he can simply create an instance:

$$\texttt{bob\_USDC} = \langle \texttt{instance(USDC)} \rangle_{\texttt{bob}}$$

and see the 100 tokens in his balance.

A successful token can result in many transfers made by many users in parallel, especially if used for payments. For example, VISA averages 20,000 transactions per second. With the advancement of AI and AI agents, it is expected that the demand for higher rates for payments and micro-payments will significantly grow, perhaps to the order of millions of transfers per second. FastSet is ready

for the challenge. It can settle a theoretically unlimited number of claims per second, including transfer claims like above. Indeed, there is nothing to prevent two transfers made by different accounts to proceed concurrently—we assume that location updates are atomic, which can be ensured using conventional synchronization mechanisms (e.g., software locks or hardware transactions).

But is the result deterministic, regardless of how the transfers are interleaved and settled by each of the validators? Theorem 1 ensures the correctness of the protocol, including its validator determinism, whenever the weak independence property holds. Because digital asset transfers in general and payments in particular represent an important use case for FastSet, below we prove the weak independence property of the `TOKEN` contract above. Suppose that two different accounts, `a1` and `a2`, can issue claim blocks

$\langle$`transfer_token(b1,v1)`$\rangle_{\texttt{a1}}$
$\langle$`transfer_token(b2,v2)`$\rangle_{\texttt{a2}}$

That implies `v1 > 0` and `balance[a1] >= v1`, and respectively, `v2 > 0` and `balance[a2] >= v2`. We have to prove that the balances of `a1`, `a2`, `b1`, and `b2` are, respectively, the same no matter in which order the two transfer claims are processed. We analyze several cases, depending on whether `b1` and `b2` are equal or not, or if any of them or both are equal to any of `a1` or `a2`. The most common case is that `b1` and `b2` are distinct and also distinct from `a1` and `a2`. Then in both cases the balances are clearly the same: `a1-v1`, `a2-v2`, `b1+v1`, `b2+v2`. If `b1==b2==b` and are different from `a1` and `a2`, then the balances are also the same: `a1-v1`, `a2-v2`, and `b+v1+v2`. If `b1==a2` and `b2` is distinct, then the balances are also the same: `a1-v1`, `a2-v2+v1`, and `b2+v2`. If `b1==a2` and `b2==a1`, then the balances are the same, too: `a1-v1+v2`, `a2-v2+v1`. Finally, if `b1==b2==a2` then the balances are: `a1-v1`, `a2+v1`. The remaining cases are similar. Consequently, the `TOKEN` contract satisfies the weak independence requirement that guarantees its determinism no matter how the transfer claim blocks by different accounts are permuted. Because the (atomic) addition and subtraction operations on integers are commutative, an even stronger property holds: the individual claims in different blocks can also be further interleaved. This gives validators the freedom to maximize the parallelism and thus the throughput of token transfers.

3.3 Auctions

Auctions tend to be complex contracts, where a set of bidders submit their bids for an item. The auctioneer keeps the highest bid and the highest bidder receives the item, while the other bidders redeem their funds that were outbidden. There are many variations of auction contracts, which, to our knowledge, fall in two broad categories when it gets to how the bidders redeem their funds: either they do it themselves, usually by calling a withdraw function when permitted, or the auctioneer sends each of them their bids back when the auction ends. None of these approaches is possible within SETL, due to its deliberately restricted nature. Indeed, there is no way for an account to withdraw any funds from another

account, at least not with the current implementations of the native `transfer` and the contract `transfer_token` block in Section 3.2: the other account has to send the funds explicitly. Also, SETL currently has no mechanism to iterate through all the keys of a map (this might be needed, eventually).

We here present a different type of auction contract, which takes full advantage of the parallel nature of FastSet. The key insight is that the highest bidder pays the previously highest bidder back, and sends the difference to the contract. This way, the contract only locks the auctioneer's item and the highest bid, so there is no need to send any funds back to the outbidden participants.

```
AUCTION[ITEM,BIDDING_TIME]:
    stopBiddingTime : Int;
    highestBidder : Address;
    highestBid : Int;

    constructor {
        ITEM.transfer_token(contract, 1);
        stopBiddingTime := time + BIDDING_TIME;
        highestBidder := contract.owner;
        highestBid := 0;
    }
    instance bid(amount) {
        time <= stopBiddingTime;
        if (amount > highestBid) {
            transfer(highestBidder, highestBid);
            transfer(contract, amount - highestBid);
            highestBidder := instance;
            highestBid := amount;
        }
    }
    instance withdraw(amount) {
        transfer(instance.owner, amount);
    }
    end() {
        time > stopBiddingTime;
        ITEM.transfer_token(highestBidder.owner,1);
        transfer(contract.owner, highestBid);
    }
```

The auctioneer creates the contract above, at the same time sending it the `ITEM` for bidding, as well as a `BIDDING_TIME` for how long bidding is allowed, e.g., `AliceTicket` $= \langle$`contract(AUCTION[ticket_item,1000])`$\rangle_{\texttt{alice}}$. The SETL `time` construct is instantiated by the client with its current timestamp, and is verified by validators that it is within adequate network delays [3]. For simplicity we assume only one item, which is a token as in Section 3.2. This item is locked in the contract until the auction ends. The contract initializes the

`highestBidder` as the contract owner, `alice` in our case, so the contract owner can redeem the item in case nobody bids on it during the specified period.

Note the use of the conditional statement in the `instance bid(amount)` block. Unlike guards, which are invalid when false, conditional statements control the execution flow but are always valid. However, if the executing branch of the conditional encounters an invalid claim, then the entire block is invalid; this would be the case, for example, if the instance bids a larger `amount` than what it holds. Bidders create instances of the contract, send funds to their instances in order to bid them, initiate `bid` blocks through their instances, and finally withdraw from their instances whatever funds were not used for bidding, e.g.,

`auction3` = $\langle$`instance(AliceTicket)`$\rangle_{\texttt{bob}}$
$\langle$`transfer(auction3,100)`$\rangle_{\texttt{bob}}$ — `bob` sends 100 to its instance
$\langle$`auction3.bid(25)`$\rangle_{\texttt{bob}}$ — `bob` sends 25 to `AliceTicket` (not `alice`)
`tk_alice` = $\langle$`instance(AliceTicket)`$\rangle_{\texttt{charlie}}$
$\langle$`transfer(tk_alice,50)`$\rangle_{\texttt{charlie}}$ — `charlie` sends 50 to its instance
$\langle$`tk_alice.bid(30)`$\rangle_{\texttt{charlie}}$ — sends 25 to `auction3` and 5 to `AliceTicket`
$\langle$`auction3.bid(40)`$\rangle_{\texttt{bob}}$ — sends 30 to `tk_alice` and 10 to `AliceTicket`

Bidding stops after the allowed time, and then the auctioneer (the contract owner) issues an `end()` block which sends the item to the highest bidder and the paid amount from the contract to the contract owner. For example,

$\langle$`AliceTicket.end()`$\rangle_{\texttt{alice}}$ — sends 40 to `alice` and ticket to `bob`

At any moment during or after the auction ends, the bidders can withdraw any available funds from their instances, e.g.,

$\langle$`tk_alice.withdraw(50)`$\rangle_{\texttt{charlie}}$ — `charlie` withdraws its funds
$\langle$`auction3.withdraw(60)`$\rangle_{\texttt{bob}}$ — `bob` withdraws everything left

The weak independence assumption that guarantees the correctness of FastSet is still obeyed by the contract above, but it is less obvious than in the previous examples. Before we show that weak independence holds, let us first illustrate the power, but also the complexity of concurrency, in order to appreciate the critical role that weak independence plays. The potential problem is that the shared field `highestBid` can be both written and read by different instances. Consequently, multiple bidders may bid concurrently and, although their individual bids get quorum, they may potentially not be able to settle. For example, in the scenario above suppose that both `bob` and `charlie` send their first bids at the same time and the validators receive their bid claims at the same time. Both claims are valid in Step 3 of the FastSet protocol (Figure 1) regardless of which is processed first, because at that step the validator states are not modified. All validators therefore sign both claims and quorum is achieved for both.

Suppose now that half of the validators receive/process the two bids in one sequence in their Step 7, say `bob` first and `charlie` second, while the other half in the other sequence, `charlie` first and `bob` second. In both cases the contract will hold 30 tokens, `charlie` will be the highest bidder, `bob`'s instance balance of

100 tokens is unaffected, and `charlie`'s instance balance is 20 tokens. However, in the second case, `bob`'s bid never took place, in the sense that the body of the conditional statement was not executed. But, importantly, `bob`'s bid block was still valid, so in the end all validators are in the same state and with the same claims settled. Since `bob`'s instance balance was not affected on any of the validators, his next $\langle \texttt{auction3.bid(40)} \rangle_{\texttt{bob}}$ correctly outbids `charlie` on all validators. We leave it as an exercise to the curious reader to notice that weak independence would be violated if we replaced the conditional statement with a guard that invalidated the block when the amount was not higher than the highest bid. In [3] we discuss this case in more depth and propose an extension of FastSet that would work with such examples, where weak independence is allowed to be temporarily broken. We also leave it as an exercise to the reader to notice that the weak independence assumption would also be violated if the previously highest bid would be returned directly to the owner of the instance, instead of to the instance itself. We discuss this case in more depth in [3] as well.

Let us now prove that the `AUCTION` contract obeys weak independence. The interesting case is when two independently highest bids are submitted at the same time. Regardless of the order in which they are processed in a given state, once both are processed the state is the same: the currently highest bidder's instance is paid back, the winner of the two bids becomes the highest bidder, and the state of the loser of the two bids stays unchanged.

4 Conclusion

This paper discussed FastSet, an actor-inspired replica-based distributed protocol for embarrassingly parallel settlement of claims. A *claim* is any statement that comes with a proof, such as a payment or more generally a digital asset transfer, a realized blockchain transaction or more transactions that form a block, an execution of a program in a programming language or virtual machine, an AI model inference or fine tuning, a TEE execution, a vote, an auction bid, among many others. A *proof* is anything that is verifiable and is accepted by the application or user/account that uses the claim: a signature (simple, aggregated, TEE, etc), a cryptographic/ZK proof, a mathematical proof, a formal semantics derivation, a program (re)execution, and so on. The verification of all proofs which are not signatures is deferred to special service providers, called *verifiers* which are account holders like any other applications/users. The replicas, called *validators*, only validate signatures, regarded as the simplest and fastest way to check proofs, and settle the claims. Each claim can be verified, validated, and *settled optimally*: independently and in parallel with any other claim. The validators need not communicate with each other, so there is no consensus in the strict sense of the word as used in blockchains. Specifically, FastSet is not strongly consistent [8], but it is strongly eventually consistent [15].

FastSet should not be regarded as the foundation for the next blockchain. It should be regarded as the Web3 infrastructure on which the next wave of blockchains and decentralized and verifiable computing applications will be built.

References

1. Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
2. Mathieu Baudet, George Danezis, and Alberto Sonnino. FastPay: High-performance byzantine fault tolerant settlement. In *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*, pages 163–177. ACM, 2020.
3. Xiaohong Chen and Grigore Rosu. Fastset: Parallel claim settlement, 2025.
4. Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency (extended version), 2019.
5. Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. *Distributed Computing*, 35(1):1–15, 2022.
6. Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
7. Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973.
8. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
9. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
10. Antoni Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets 1986, Part II: Proceedings of an Advanced Course, Bad Honnef, 8.–19. September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1987.
11. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Nov 2008. Accessed: 2025-06-15.
12. Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–320, Philadelphia, PA, June 2014. USENIX Association.
13. Nuno Preguiça, Marc Shapiro, and Jose Legatheaux Martins. Designing a commutative replicated data type for cooperative editing systems. Research Report TR-02-2008 DI-FCT-UNL, Universidade Nova de Lisboa, Dep. Informática, FCT, 2008.
14. Marc Shapiro and Nuno Preguiça. Designing a commutative replicated data type. Research Report RR-6320, Institut National de Recherche en Informatique et en Automatique (INRIA), October 2007.
15. Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011.