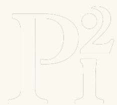


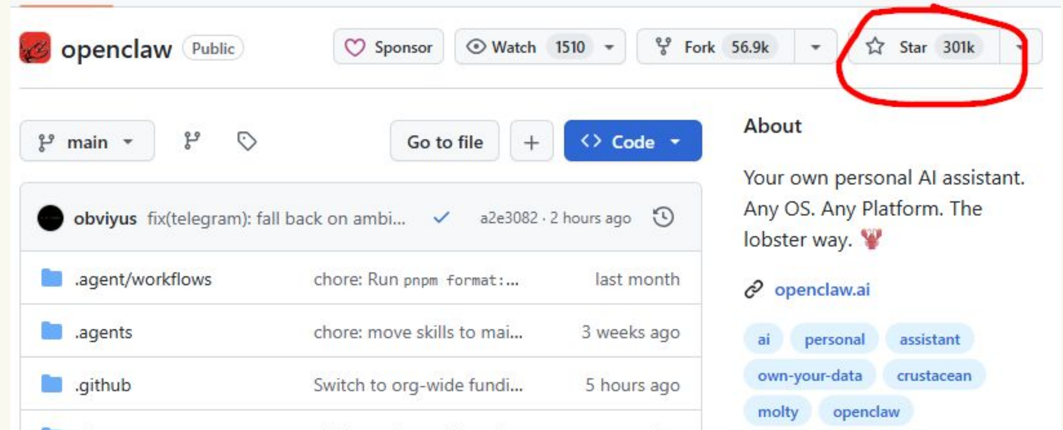
FastSet: Parallel Claim Settlement

Xiaohong Chen, Grigore Rosu
https://fast.xyz

P²i

Autonomous Agents Are Here

- OpenClaw — an open-source personal AI assistant — reached 301k GitHub stars and 56.9k forks within weeks of launch.
- Agents are already writing code, managing infrastructure, and handling complex workflows autonomously.
- This presentation was entirely prepared by an OpenClaw agent — from content research through slide design and formatting. 🤖



P²

MOTIVATION

Billions of Autonomous Agents. Trillions of Microtransactions.

- AI agents purchasing APIs, cloud resources, data at machine speed
- Machine-to-machine payments at sub-cent granularity
- Latency budget: milliseconds, not seconds
- Current payment infrastructure was designed for humans

Can we settle transactions most efficiently?

For a large and useful class of applications — Yes!

This talk: the theory, the protocol, and the proof.

Pi²

The Total Ordering Bottleneck

- Satoshi Nakamoto thought that total ordering is needed to prevent double spending
- Every blockchain forces a single global order on all transactions.
- Bitcoin, Ethereum, Solana — all solve consensus to agree on one canonical sequence.
- Tx_1 , Tx_2 , Tx_3 , Tx_4 → [Global Order] → State
- This creates a **fundamental throughput bottleneck**.

But Do We Need It?

- Asset transfers (payments) have *consensus number* 1. (Guerraoui et al., 2019)
- You don't need consensus to prevent double-spending.
- Two payments by different users can always be processed in either order with the same result.
- This is the theoretical foundation of FastSet.

THE PROBLEM

Pi2

Prior Work and Our Contribution

FastPay (Baudet, 2019)	FastSet (this work)
Payments only	Arbitrary claims ✓
Not programmable	Programmable ✓
Single use case	Auctions, verifiable computing, and more

FastSet Protocol: 7 Steps

```
+----------+  1. Claim   +----------+              +-------------------------+
|  Agent   |------------>|          |------------->|   FastSet Validators    |
+----------+             |          |              |                         |
                         |          |              |  +-----+ +-----+ +-----+ |-------------+
+----------+  2. Verify  |          |              |  |     | |     | |     | | 3. Validate |
| Verifier |------------>|  Proxy   |------------->|  +-----+ +-----+ +-----+ |<------------+
|          |             |          |              |  +-----+ +-----+ +-----+ |
|  +-----+ |             |          |  4. Sign     |  |     | |     | |     | |
|  |     | |             |          |<-------------|  +-----+ +-----+ +-----+ |
|  +-----+ |             |          |  5. Confirm  |                          |
|  +-----+ |             |          |------------->|   . . . . . . . . .      |-------------+
|  |     | |             |          |              |                          | 6. Presettle|
|  +-----+ |             |          |              |  +-----+ +-----+ +-----+ |<------------+
|   . . .  |             |          |              |  |     | |     | |     | |
|  +-----+ |             |          |              |  +-----+ +-----+ +-----+ |-------------+
|  |     | |             |          |              |                          | 7. Settle   |
|  +-----+ |             |          |              |                          |<------------+
+----------+             +----------+              +-------------------------+
```

① Claim → ② Verify → ③ Validate → ④ Sign → ⑤ Confirm → ⑥ Presettle → ⑦ Settle

PROTOCOL

Claims, Messages, Certificates

Claim c

Any verifiable statement that can change state.

"I'm Xiaohong Chen"

"Alice pays Bob $10"

"David bids $500 on item #42"

"Riemann hypothesis holds; proof: …"

Message m

$m = \langle c_1\ c_2 \ldots c_{\square}, n \rangle_a$

Batch of claims signed by client a with nonce n, as well as some verifiers (by need)

Certificate $\langle m \rangle_{\#}$

Message with a *quorum* (more than 2/3) of validator signatures.

Irrevocable proof. Settlement is now inevitable.

P2i

Why 2/3 Quorum?

Liveness

Honest nodes alone must be able to form a quorum. Otherwise, bad nodes halt the network by refusing to sign.

Requires: `Good ≥ Q`

Safety (No double spending)

No two conflicting certificates can exist for the same client and nonce. Bad nodes cannot split good nodes into contradictory quorums.

Requires: `Bad + Good/2 < Q`

Proof: $Q > 2N/3$

$\text{Bad} + \text{Good}/2 < \text{Good}$

$\text{Bad} < \text{Good}/2$

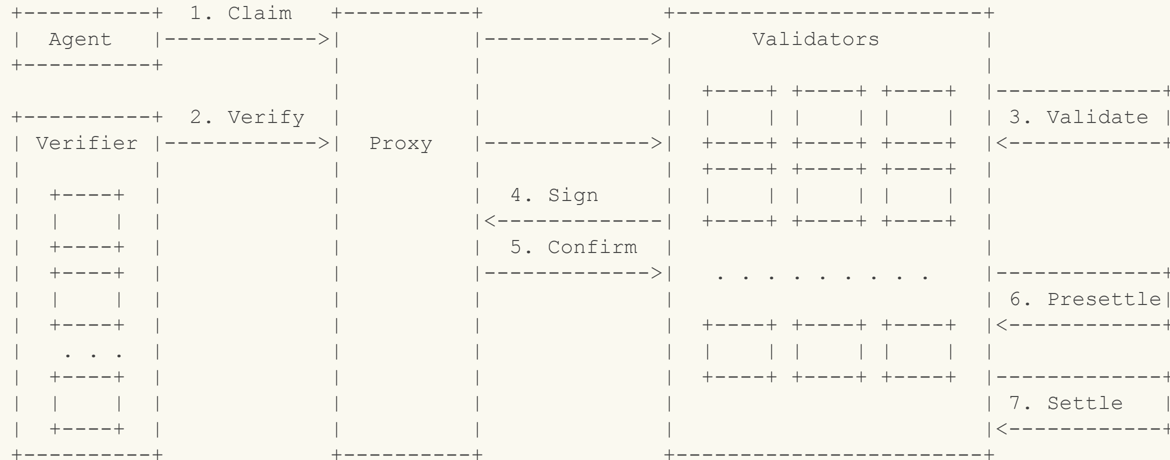
$\text{Good} > 2N/3$, where $N = \text{Good} + \text{Bad}$

$Q > 2N/3$

With $N = 3f+1$:

$Q = 2f+1$ ✓

FastSet Validators API



`v.state` — v's local view of the global network state

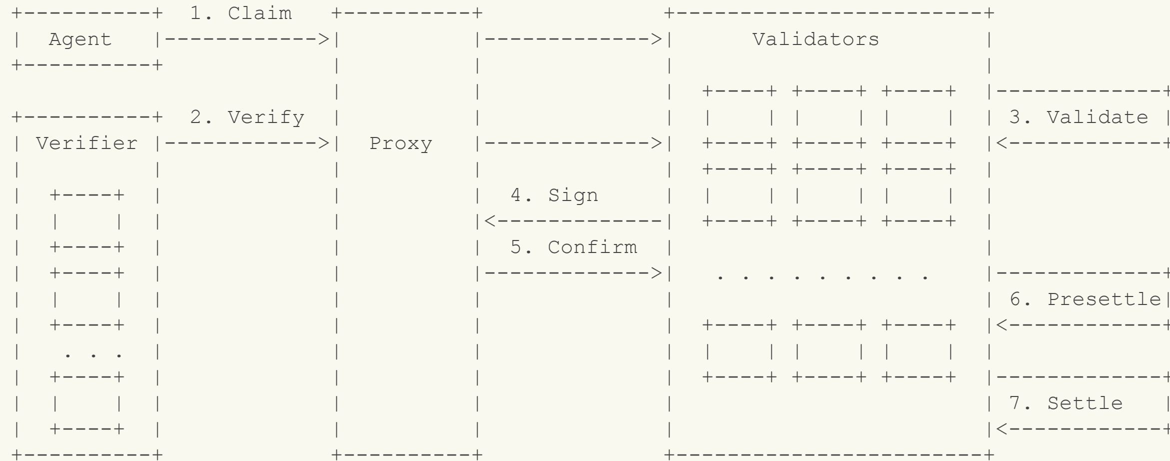
`v.presettled` — certified messages awaiting settlement

`v.settled` — fully settled messages

`v(a).nonce` — next expected nonce for account a

`v(a).pending` — message currently being validated (∅ or {m})

Step 1 — Claim

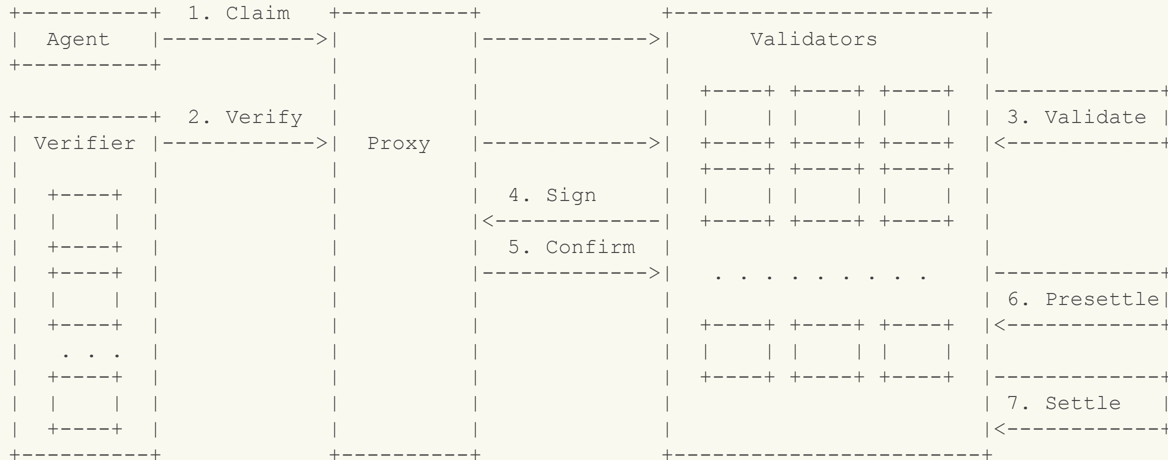


Agent submits a signed message $m = \langle c_1 c_2 \ldots c_n, n \rangle$ to the Proxy.

The message contains a batch of claims and a nonce n.

The nonce is incremented after each message to prevent replay.

Step 2 — Verify

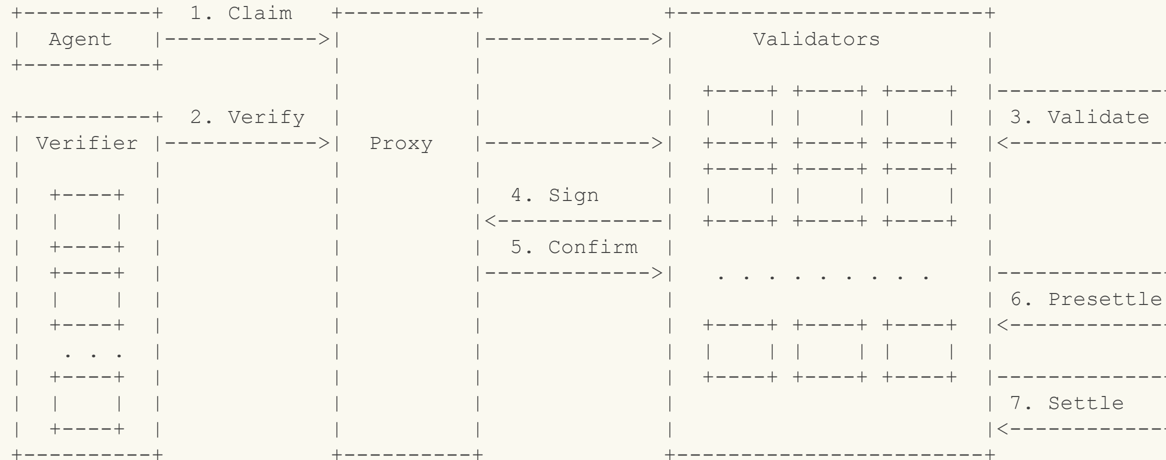


Verifiers check the claims and co-sign the message.

Proxy aggregates verifier signatures and add them to m.

The verified message is then broadcast to all validators.

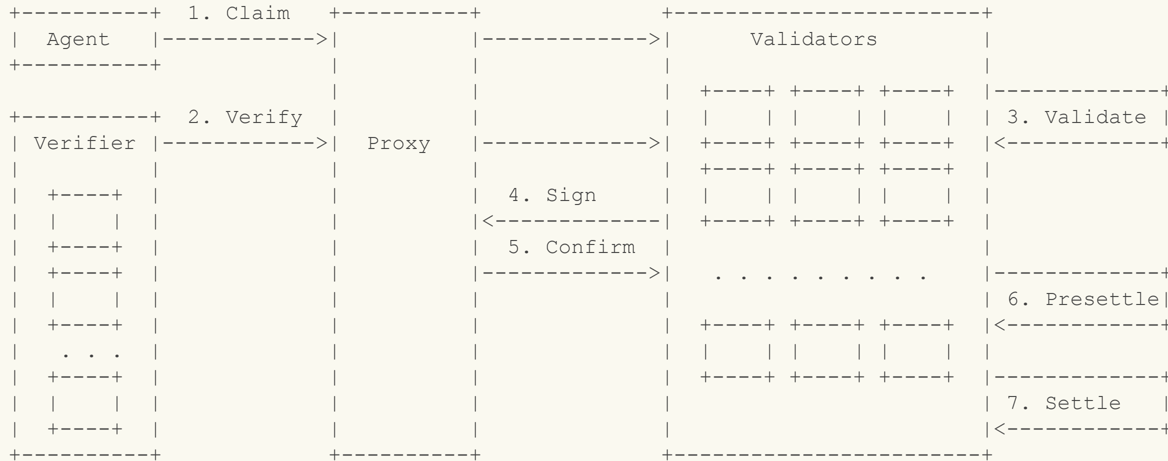
Step 3 — Validate



Upon receiving $m = \langle c_1 c_2 \ldots c_{\square}, n\rangle$, each validator independently checks:

- All signatures valid and verifier quorum reached
- Correct nonce: $v(a).nonce = n$
- No conflicting pending message: $v(a).pending \subseteq \{m\}$
- All claims valid in current state: $c_1 c_2 \ldots c_{\square} \downarrow v.state$

Step 4 — Sign

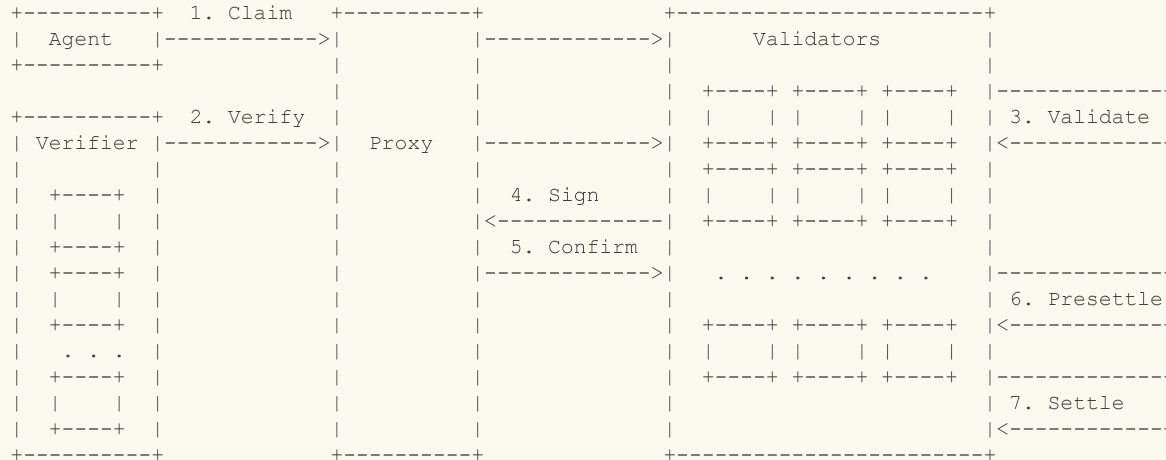


If valid, validator signs m and sets `v(a).pending := {m}`.

No state update yet — the validator is locking m, but not settling.

The signed message is returned to the Proxy.

Step 5 — Confirm



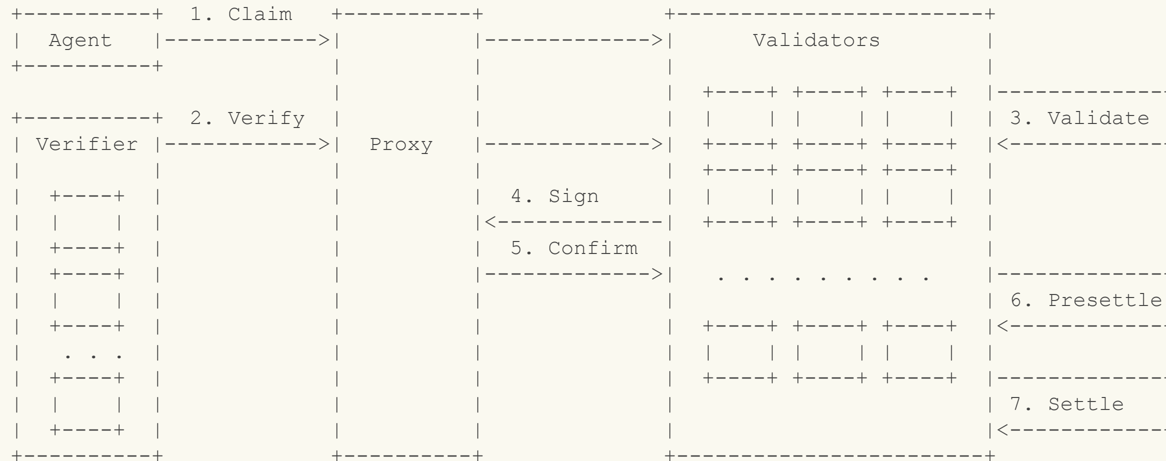
Proxy collects validator signatures.

When quorum ($2f+1$) is reached → certificate $\langle m \rangle_{\#}$ is formed.

The certificate is irrevocable proof — **settlement is now inevitable**.

Proxy sends the certificate to all validators.

Step 6 — Presettle

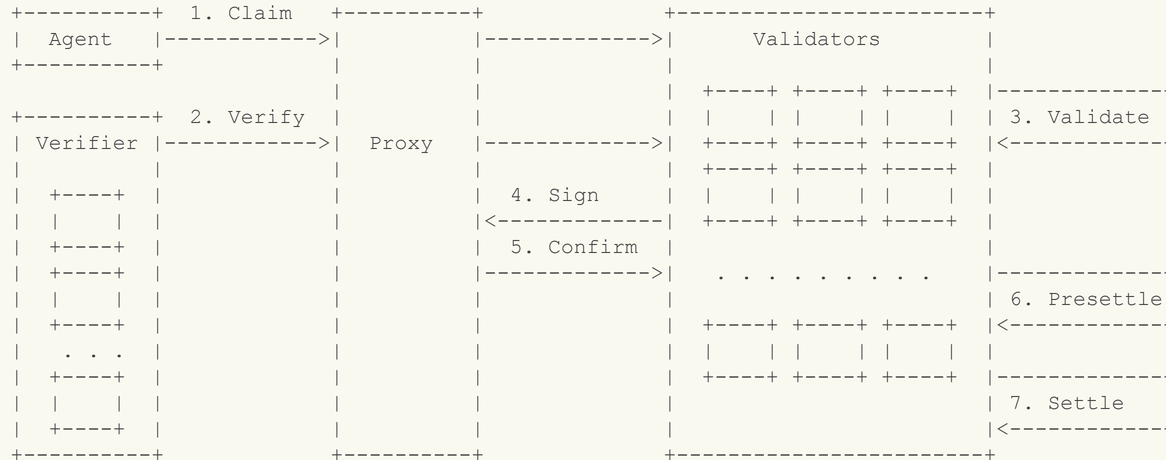


Upon receiving $\langle m \rangle_\#$, validators add m to their presettled set: `v.presettled = v.presettled ∪ {m}`

The message may not be ready to settle yet (e.g., claims not valid, nonce not right, etc.).

Pre-settlement guarantees the message WILL be settled.

Step 7 — Settle



Validators continuously scan the presettled set.

When a pre-settled message becomes valid, it gets settled: $\texttt{v.state} := [\![c_1 \ldots c_n]\!](\texttt{v.state})$

Increment nonce. Move message from `v.presettled` to `v.settled`.

Settlement is now complete.

P²

Why No Global Ordering?

- Validators don’t talk to each other. Clients talk to validators 1-1.
- Each account has its own nonce sequence.

Alice: nonce 1 → 2 → 3 ... Bob: nonce 1 → 2 → 3 ... Carol: nonce 1 → 2 ...

- Same-client messages: ordered by nonce (sequential).
- Different-client messages: no ordering needed (parallel). ✓

Correctness of FastSet Protocol

- Security: One certificate per client per nonce → no double-spending.
- Determinism: Message arrival order is irrelevant to final state.
- Monotonicity: Once settleable, stays settleable regardless of others' actions.
- Liveness: If one validator makes progress, all do.

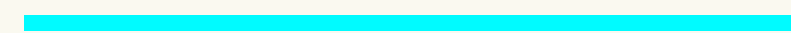
FastSet: Parallel Claim Settlement

Xiaohong Chen[1], Grigore Rosu[1,2]
Pi Squared Inc. (http://pi2.network)[1]
University of Illinois Urbana-Champaign[2]

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

Weak Independence



When can two claims be processed in parallel?

The Problem with Classical Independence

- Mazurkiewicz (1987): c $/\!/$ c' iff they commute in all states. Too strong!
- "Alice pays Bob 10" and "Bob pays Carol 10" — one payment can enable the other.
- NOT classically independent.
- But if both are individually feasible (Alice ≥ 10 AND Bob ≥ 10), the result is identical regardless of order.
- Classical independence rejects this. Weak independence accepts it.

P^2_i

Weak Independence

- $c \mathbin{/\!/} c'$ iff for any state s, if $c \downarrow s$ and $c' \downarrow s$:
- $c\, c' \downarrow s$ and $c'\, c \downarrow s$ and $[\![c\, c']\!](s) = [\![c'\, c]\!](s)$
- Only requires commutativity when BOTH claims are already valid.
- Classical: commute in all states. Weak: commute when both valid. **Strictly weaker → captures more.**

P²

What's Weakly Independent, What's Not

✓ Weakly Independent

Payments by different clients, including chained payments:

"Alice pays Bob" then "Bob pays Carol"

Bids by different bidders in an auction

Independent computation results submitted by different clients

✗ NOT Weakly Independent

Two payments by same client:

"Alice pays Bob 1" and "Alice pays Carol 1" (if Alice has only 1)

Same-client claims ordered by nonce — sequential, never concurrent

Applications

SETL: a contract language for FastSet

Payments · Auctions · Verifiable Computing

SETL — The Contract Language

SETL is a simple contract language for writing FastSet applications.

FastSet claims = SETL scripts

We demonstrate three contracts: **TOKEN**, **AUCTION**, and **VERIFIED_COMPUTE**.

Payments

Pi^2

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

Concrete Walkthrough

Alice calls `transfer_token(Bob, 10)`
Claim = ⟨`transfer_token(Bob, 10), nonce=5`⟩a□ice

Validators check:
sig ✓ nonce=5 ✓ `balance[Alice]≥10`✓

Quorum → certificate → settle:
`balance[Alice] -= 10`
`balance[Bob] += 10`
nonce → 6

Done.
< 100ms. No block, no mempool.

Carol calls `transfer_token(Dave, 20)` simultaneously.
Same 7 steps, independently. No waiting.
Both settle in parallel.

Payments

Pi^2

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

What About Conflicts?

Same sender, two transfers?

Alice: `transfer(Bob,10)` nonce=5
Alice: `transfer(Carol,20)` nonce=6
Nonce 6 waits for nonce 5. Serialized with herself.
Everyone else proceeds in parallel.

Both pay Bob?

Alice: `balance[Bob] += 10`
Carol: `balance[Bob] += 20`
Both additions. `100+10+20 = 100+20+10`.
Commutes ⟶ parallel. ✓

The rules:

Same sender ⟶ ordered by nonce.
Same recipient ⟶ no conflict.
Different senders ⟶ never wait.

Auctions

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

Alice bids 50, Bob bids 80, Carol bids 100.
Each bid is a claim — goes through the 7 steps independently.
Bob outbids Alice: refunds Alice 50, locks Bob's extra 30.
Carol outbids Bob: refunds Bob 80, locks Carol's extra 20.
All three bids can arrive concurrently.

Are concurrent bids weakly independent?
Only one touches `highestBid` at a time (per-sender nonce).
The `if`-guard makes losing bids no-ops — no state change.
Winning bid updates `highestBid` + refunds previous winner.
Result: only the highest bid "wins." Others bounce harmlessly.

After `stopTime`:
`end()` transfers the item to winner, payment to auctioneer.
No global ordering needed. No front-running.
Time assertion replaces `block.timestamp` — progresses universally.

APPLICATIONS

Verified Computing

```
VERIFIED_COMPUTE[LANG, VERIFIERS, QUORUM]:
  instance compute(pgm, input, s,
                   claims, s', proof) {
    verify(VERIFIERS, QUORUM);
    claim("compute", LANG, pgm,
          input, s, claims, s', proof);
  }

FINALIZE_SEQUENTIALLY[GENESIS_STATE]:
  state : Bytes;
  constructor {
    state := GENESIS_STATE;
  }
  step(pgm, input, s, claims, s', proof) {
    state == s;
    isSettled(
      claim("compute", EVM, pgm,
            input, s, claims, s', proof));
    claims;
    state := s';
  }
```

Two contracts, two roles.

`VERIFIED_COMPUTE`: anyone submits a computation claim. A quorum of verifiers checks the proof. That's it — settled. No re-execution. The proof is the execution.

`FINALIZE_SEQUENTIALLY`: an app that needs ordered state.
`state == s` ensures steps apply in order.
`isSettled()` checks the computation was already verified.
`claims` applies any side-effects (e.g., token transfers).
`state := s'` advances the app state.

Why two contracts?

Verification is embarrassingly parallel — different programs, different inputs, different provers. All weakly independent. Finalization is sequential by design — state transitions must apply in order. One contract per concern.

This is how you rebuild blockchains on FastSet.
Run EVM off-chain. Prove it. Settle the result.
FastSet doesn't execute — it settles.

FastSet: Parallel Claim Settlement

Total ordering is unnecessary for asset transfers

Weak independence captures exactly which operations can be parallel

Zero validator-to-validator communication (7 steps, BFT)

Not just payments — programmable settlement

Sub-100ms finality · 250,000+ TPS · Near-zero fees · Live today

Chen, Rosu — fast.xyz

FastSet

CRDTs

A general framework where weak independence follows automatically from the data types used.

FastSet CRDTs

- A FastSet CRDT is a triple (D, H, G) where:
- D = data domain (e.g., $\mathbb{N}$, $\mathscr{P}$(Claim))
- H = host operations (submitted by cell owner)
- G = guest operations (submitted by non-owners)
- Rule: $\forall\ f \in H \cup G,\ g \in G$: if $f \downarrow v$ and $g \downarrow v$ then $fg \downarrow v$, $gf \downarrow v$, $[[fg]](v) = [[gf]](v)$
- Any operation must commute with any guest operation, when both are defined.
- Classic CRDTs = special case where H = $\varnothing$. FastSet CRDTs are strictly more general.

P²

Example — Balance CRDT (D = ℕ)

Operations

H (host): dec_k

Defined when $b \geq k$

Effect: $b \to b - k$

Owner only.

G (guest): inc_k

Always defined

Effect: $b \to b + k$

Anyone can call.

Payment Example

"Alice pays Bob k":

Alice.balance: dec_k (host)

Bob.balance: inc_k (guest)

dec + dec: ordered by nonce (both host — never concurrent).

inc + inc: always commute (both guest). ✓

Payments by different senders are weakly independent.

Pi^2

Example — Settlement CRDT

- D = $\mathscr{P}$(Claim)
- G: claim(c) — always defined — S $\rightarrow$ S $\cup$ {c}
- G: isSettled(c) — defined when c $\in$ S — identity
- H = $\varnothing$ $\rightarrow$ classic CRDT (all ops are guest, owned by no one)
- Classic CRDTs are a special case of FastSet CRDTs.
- Connects FastSet CRDTs to the well-studied CRDT literature.

Soundness and Completeness

Completeness ✓

Weakly independent claims can always be decomposed into FastSet CRDT operations.

The two notions — weak independence of claims and commutativity of CRDT guest operations — are equivalent.

Soundness ✓

Claims built from FastSet CRDT operations are automatically weakly independent.

If your data types are FastSet CRDTs, correctness follows — no case-by-case proofs needed.