Blockchains Synchronisation

Research Offsite 2023

Often Neglected

The Literature

- Not a scientific problem
- Never described or discussed
- Often not even implemented

Our Experience

- Communication complexity can be high
- One of the components causing the most pain
- Key performance bottleneck

The Result

Multiple ad-hoc components

Narwhal synchroniser (for consensus)

Checkpoint synchroniser (for epoch change & Sui)

Others: Gossip, full nodes stream, state snapshots

Desired Features

- Native support for reconfiguration (when should it stop?)
- Does not get in the way of disk pruning
- Efficient caching layer (better than relying on the db layer)
- Native support for stake and app-level DoS protections
- Co-designed with the common data dissemination method (e.g., subscriber model)

Two Different Purposes

Live Sync

- Required to commit (liveness)
- Can only leverage statistics and partial Dag
- Internal component tied to consensus
- **Needs low latency**
- Harder to build

Historic Sync

- Allow slow nodes to catch up
- Can leverage the commit sequence
- External component
- **Needs high throughput**
- Easier to build

Observations

- Task is often parallelizable
- No need to re-verify all signatures
- The Dag gives plenty of info about the reliability of peers

Historic Sync

Easier to build

Periodically disseminate proofs of latest commits (implicitly or explicitly)

Identify what needs to be synched

Request chunks of committed sequence in parallel

Easily verify chunk K using chunk K+1

Historic Sync

Harmful if done wrong

No point of low latency if clients perceive high latency

Dedicated testbed to benchmark the historic sync

Live Sync

Harder to build

Step 1: minimum for liveness

```
Sample random peer i
RequestBlocks(i, [references])
ReplyBlocks(i, [blocks])
```

Live Sync

Harder to build

Step 2: performance under network partitions / censorship

```
Sample random peer i

RequestStream(I, all-from-peer-j)

Stream: [block(j)]
```

Periodically re-try connection with peer i

Live Sync

Harder to build

Step 3: smarter peer selection

- The Dag often tells which peers holds a specific missing block
- Locally keep scores for each peer (fast network, authored many Dag vertices, etc)
- Bias the peer sampling of streams with these scores

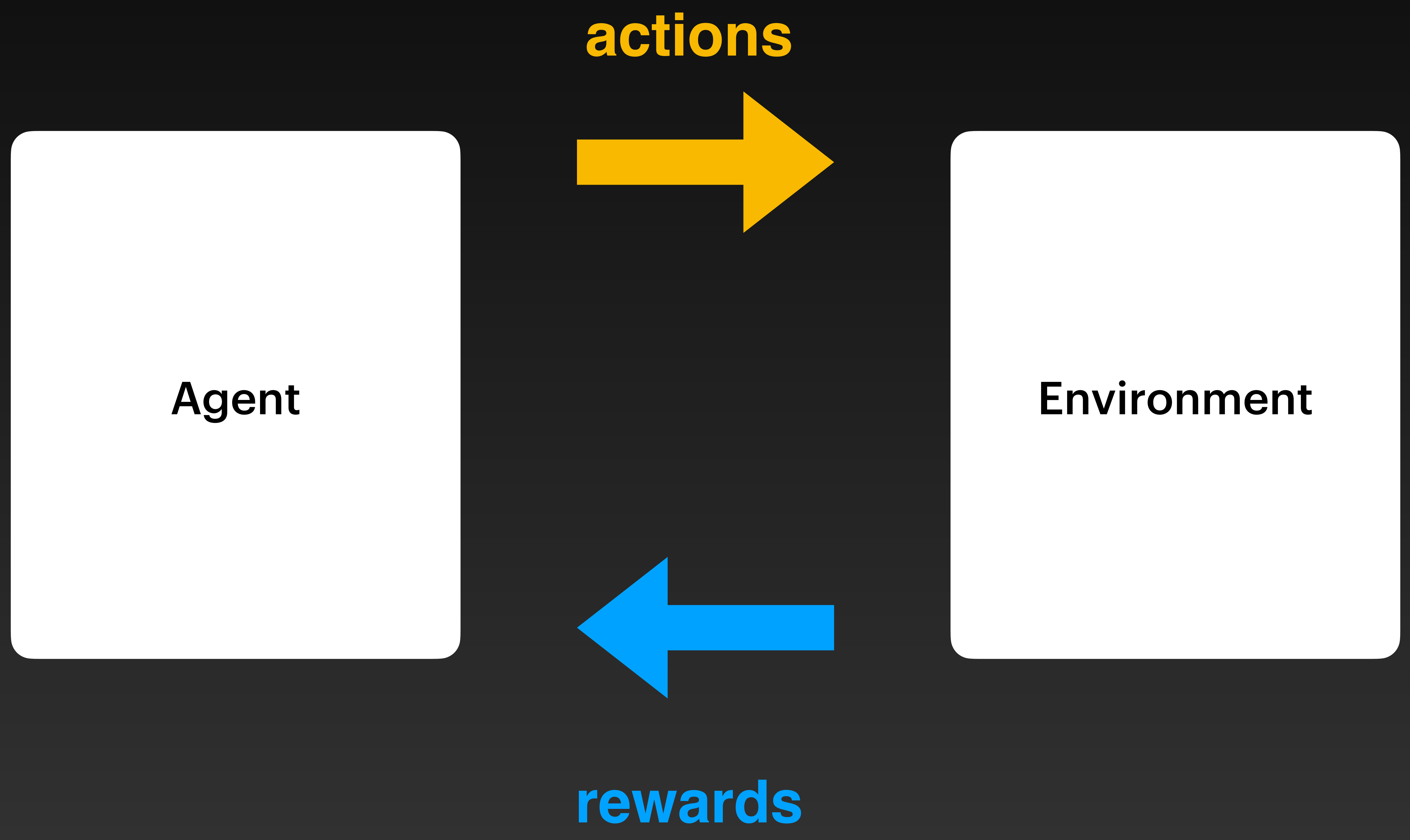
Live Sync

Harder to build

Step 4: automated sync policy (blue sky)

- A RL agent explores and learns the best sync policy
- SARSA: simple, state-of-the-art, cautious, and adapted to continuous problems

SARSA Sync



SARSA Sync

Start simple: History sync

- Pre-populate a dag
- Connect the peers to each other (various latencies)
- Sync as fast as possible while training the agent

SARSA Sync

State

- Set of missing block references: (author, round, digest)
- Network connection strength
- The Dag (who committed what)
- Pending state: the number of blocks that could be processed upon getting a missing one

SARSA Sync

Actions

```
RequestBlocks(i, [references])

RequestBlocks(i, all-from-peer-j)

StopStream(i, all-from-peer-j)

No-op
```

And combination of the above

SARSA Sync

Reward

`Download throughput`

SARSA Sync

Multi-Agent SARSA

- The Dag acts as communication medium (even in an BFT way)