Anoma Node Architectural Meeting Notes 14-08-2024

The topic of the meeting has been the supervision structure of the Anoma OTP.

The overall design can be observed via the diagram below:

[

Untitled-2024-08-14-1932

1829×1013 151 KB

](https://europe1.discourse-cdn.com/flex013/uploads/anoma1/original/1X/24a12720233c2b236820926bae7210c8c999ad15.jpeg)

The overall philosophy is to safeguard consistency in between the core engines involved with execution and consensus, namely Mempool, Ordering, and Storage.

Terminology:

- one_for_all
- if one child dies, restart all children
- one_for_one
- if a child dies, restart it
- max_restarts: n
- children can die at most n

times otherwise the supervisor crashes

- temporary
- children which do not restart after crash
- transient
- children get restarted only if they crash “abnormally”

The OTP should start with a shoehorn pseudo-supervisor with one_for_all

strategy that will have two children which represent what we currently refer to as Anoma application and the PubSub application.

The PubSub application structure is as follows:

- one_for_all

supervision on top with children: * Registry

- Top Level Event Broker
- Dynamic Supervisor for Filter Agents

– Transient Children

- Registry
- Top Level Event Broker
- Dynamic Supervisor for Filter Agents

– Transient Children

The Registry receives requests to start new filter agents, which it does by asking the Dynamic Supervisor to add new children with specified specs. The above guarantees that if Registry, Top Level Broker quit, or any of the filter agents quit abnormally (i.e. they were not shut down using the Registry functionality) that the entire application get restarted as that might interact with the message reception by the Storage Engine.

The Anoma Node structure is as folows:

- The Node supervisor, which stores all addresses and starts up the core processes. one_for_all

strategy. Children: * Router. Separate actor. If it fails the entire network fails.

- Logger. Important for replayability. If Logger fails, we might lose info for succesfull event replay.

- Supervisor for TX processing. one_for_all

strategy. (Probably also 0 max_restarts

option by default) with children * Storage

- Ordering

- Mempool Supervisor. one_for_all

strategy, max_restarts: 0

with children * Mempool

- Dynamic Supervisor for workers with temporary

children

- Mempool

- Dynamic Supervisor for workers with temporary

children

- Storage

- Ordering

- Mempool Supervisor. one_for_all

strategy, max_restarts: 0

with children * Mempool

- Dynamic Supervisor for workers with temporary

children

- Mempool

- Dynamic Supervisor for workers with temporary

children

- one_for_one

supervisor for remaining Engines. Other engines are user-specified, i.e. only the user changes their state. Hence it is unimportant whether they get restarted with some base state as long as they are accessible for the user.

- Router. Separate actor. If it fails the entire network fails.

- Logger. Important for replayability. If Logger fails, we might lose info for succesfull event replay.

- Supervisor for TX processing. one_for_all

strategy. (Probably also 0 max_restarts

option by default) with children * Storage

- Ordering

- Mempool Supervisor. one_for_all

strategy, max_restarts: 0

with children * Mempool

- Dynamic Supervisor for workers with temporary children

- Mempool
- Dynamic Supervisor for workers with temporary

children

- Storage
- Ordering
- Mempool Supervisor. one_for_all

strategy, max_restarts: 0

with children * Mempool

- Dynamic Supervisor for workers with temporary

children

- Mempool
- Dynamic Supervisor for workers with temporary

children

- one_for_one

supervisor for remaining Engines. Other engines are user-specified, i.e. only the user changes their state. Hence it is unimportant whether they get restarted with some base state as long as they are accessible for the user.

This structure ensures that all self-contained inter-dependent components will have common supervisors that will cause a restart-capability for the entire application in case anything goes wrong. The decision should be backed by having good replayability or snapshotting capability in order to be able to reach a “latest” consistent state of the overall application. Following the Erlang philosophy, we should allow things to crash without too much worry, so this is a sensible strategy.