Google

bittide bootstrap v2

Calin Cascaval
Dec 22, 2022

Recap: Robbie’s bootstrap protocol

Goals & Assumptions

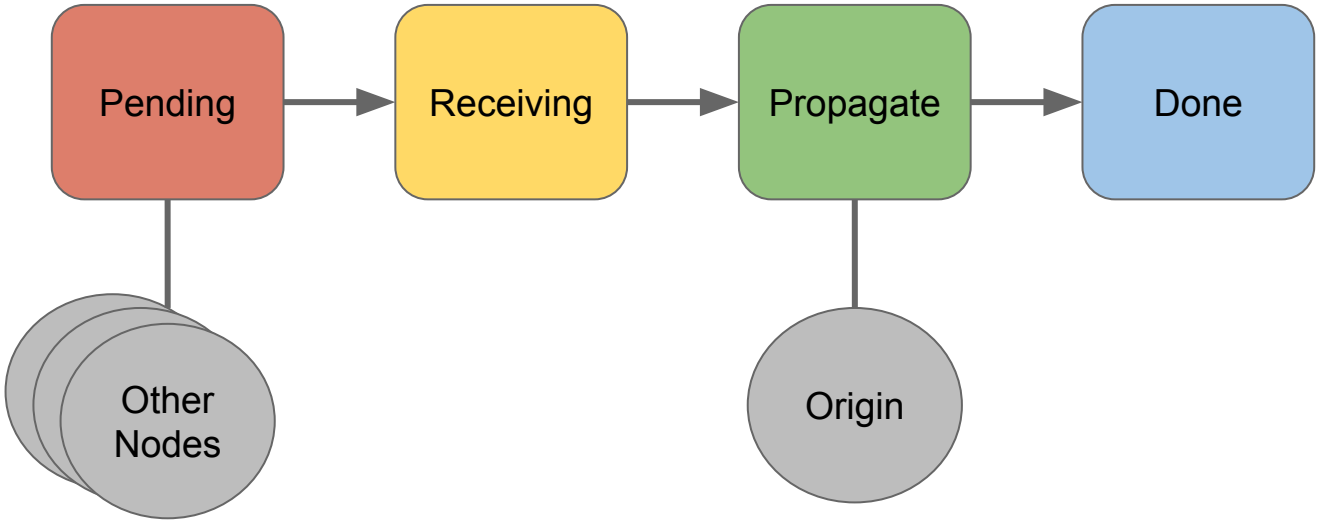
- Design and implement a message passing protocol that Bittide could use to bootstrap itself

Assumptions

- Startup on a single node, all other nodes wait for ‘magic’ start frame
- Network topology unknown to protocol (learn and store)
- Bittide’s underlying scatter gather buffers and frequency control loop are already active
- No existing scheduler

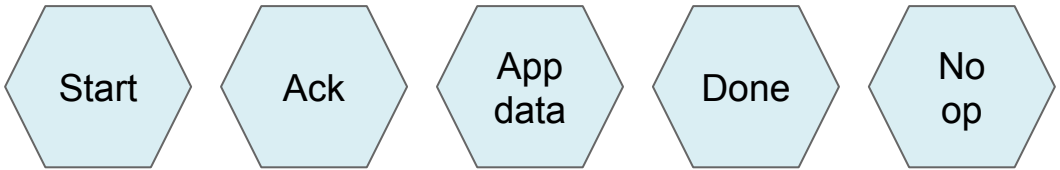
Protocol

- Each node keeps track of its state, which progresses through 4 stages

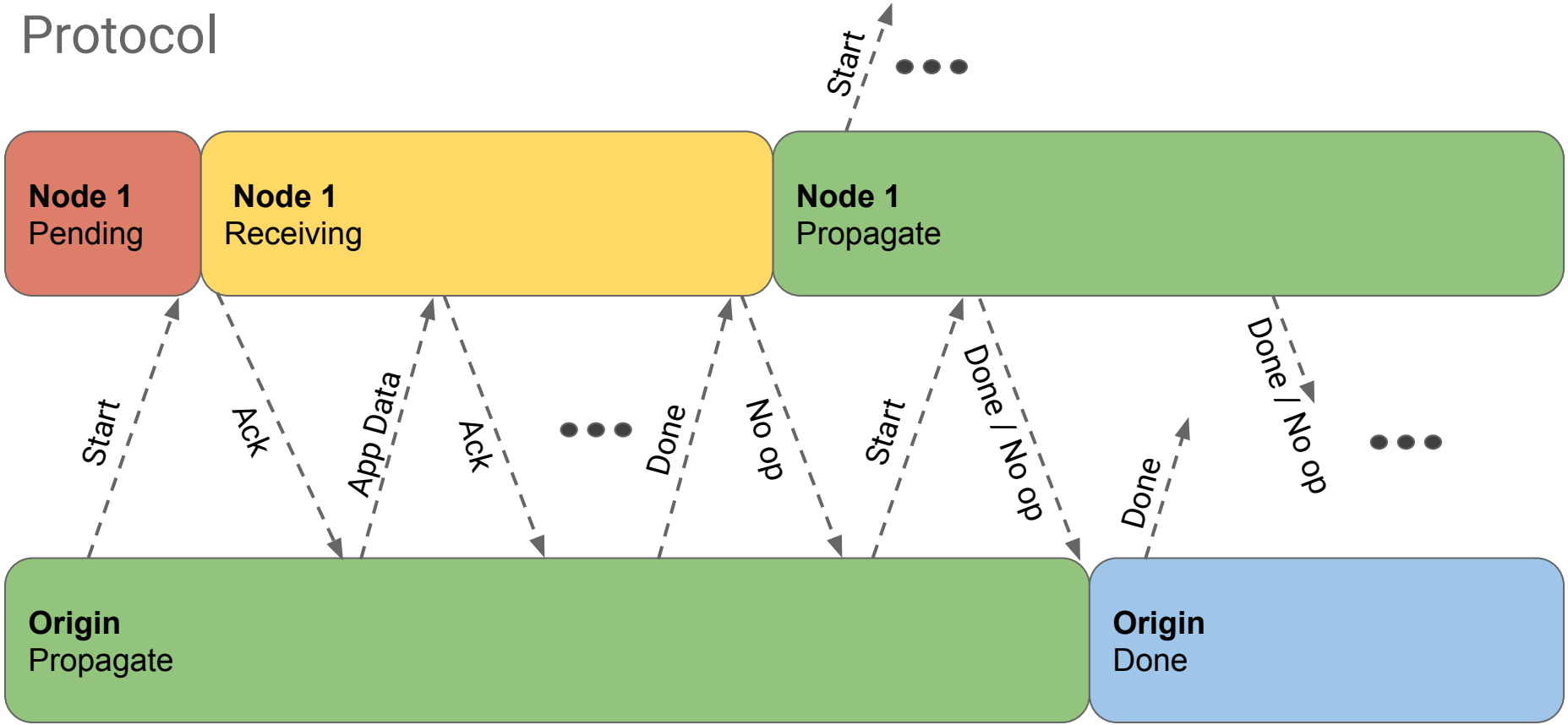


Protocol

- At every Bittide cycle, every node receives a “packet” and produces a “packet” for each outgoing link
- A packet consists of
  - A unique source id for the node (maybe the MAC address in real life)
  - An bitmap adjacency matrix (more on this soon)
  - A “message”
- The type of message sent out depends on the incoming message type and the state of the node
- Messages can have the following types:



Protocol


Start
Node 1
Pending
Node 1
Receiving
Node 1
Propagate
Start
Ack
App Data
Ack
Done
do on
Start
Done / No op
Done
Done / No op
Origin
Propagate
Origin
Done

why a v2?

Robbie's bootstrap protocol (doc, slides) has a number of limitations:

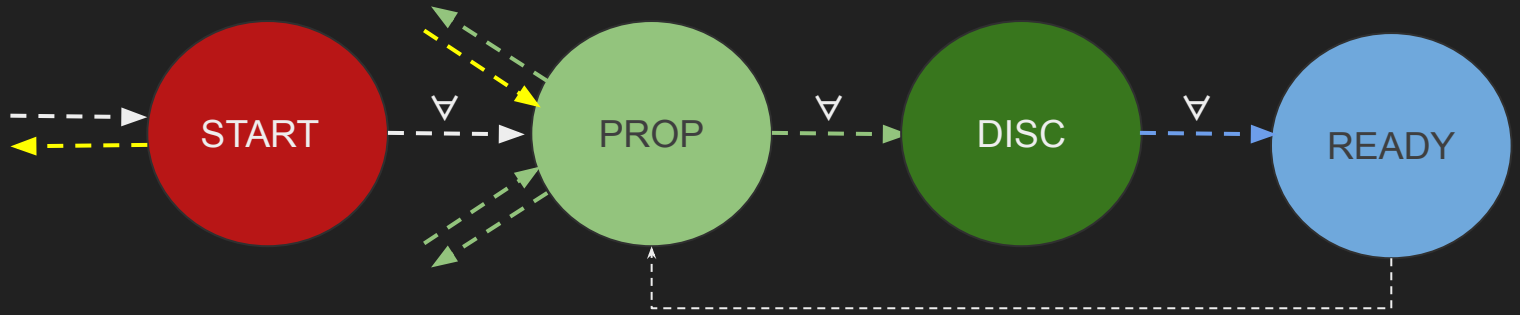
- parents infer the state of the children based on the messages they send and message response types, rather than children reporting their state; this creates issues for children with multiple parents.
- nodes respond with status messages before processing all the inputs for a cycle, resulting in different responses to neighbors depending.
- topology discovery is interleaved with application loading; thus it is not possible to load multiple applications without an additional protocol.
- there is an (outside selection) of the origin node and it's handled differently.
- there is no failure tolerance or recovery, as there is a "final" done state; only the origin has the correct state of the topology and there is no dissemination of that state to all nodes.

bootstrap v2

guidelines

1. Maintain all the original assumptions except having a special origin node that starts the protocol.
2. Separate topology discovery and application loading in two separate protocols.
    - topology discovery will be running continuously, therefore handling node additions and failures; there are only intermediate “done” states when the system is stable and notifications of failures and topology changes will be propagated to the scheduler.
    - the topology is disseminated to all the nodes and the external scheduler.
    - with the topology available, nodes will compute one (or more) spanning trees locally; the spanning tree is computed using a deterministic algorithm ordering nodes based on their ids. Multiple spanning trees are used to guarantee reachability in case of failures (algos available in the literature).
    - the system uses the spanning tree for loading applications; the scheduler decides which nodes receive which application.

node states



topology discovery (propagation)

Every node starts in a “promiscuous” mode, broadcasting their status (START) to all neighbors

When a node receives acknowledgments for the START from every neighbor it transitions to propagating its topology view (PROPAGATE).

When a node has all its neighbors as PROPAGATE:

- if it is a terminal node (a single connection to a parent), it transitions to DISCOVERED
- otherwise iterates and updates its topology until:
  - there are no more changes coming from neighbors;
  - or hears DISCOVERED from all its neighbors;
  - at that point it transitions to DISCOVERED.

topology discovery (dissemination)

when a node reached DISCOVERED it computes the spanning tree over the current view of the topology to identify its immediate children, to whom it will listen for the READY message

Computing the spanning tree will “elect” the origin as the node with the lowest id. When the origin hears from all its neighbors that they’ve been DISCOVERED it concludes that it has the complete topology and it starts propagating it to its children in the spanning tree with a READY message.

Nodes transition to READY:

- if terminal, when their parents propagate the READY message; it then acknowledges the READY.
- otherwise when they receive acknowledgements for the READY message from all their children in the spanning tree.

topology discovery (guarantees)

Propagation guarantee: if there is a node that has not yet been discovered, its neighbors will notice a change, and thus will continue to PROPAGATE, until there are no other discovered nodes.

Termination guarantee: a node will always compute a correct subtree of the spanning tree, since it heard from all its neighbors

topology changes: addition, removal, failures

Assumption: nodes continuously acknowledge the READY state to their neighbors. When a node has a neighbor in PROPAGATE, it transitions to PROPAGATE.

Adding a node to the topology:

- neighbors in READY state receive a START message, transition to PROPAGATE and follow the protocol to reach READY again.

Failed link:

- if a node does not hear from a neighbor, it removes the connection in its topology and transitions to PROPAGATE. Then follow the protocol to reach READY again.

Removing a node:

- similar to single link failure, for all the links connecting the node.

application loading

guidelines

applications are loaded onto slices (spatio-temporal partitions)

- app binaries are specific to each node – they contain the schedule for compute and comms
  - this argues for a direct connection between the global scheduler and the node; however,
  - the global scheduler and its path to the node becomes a single point of failure.

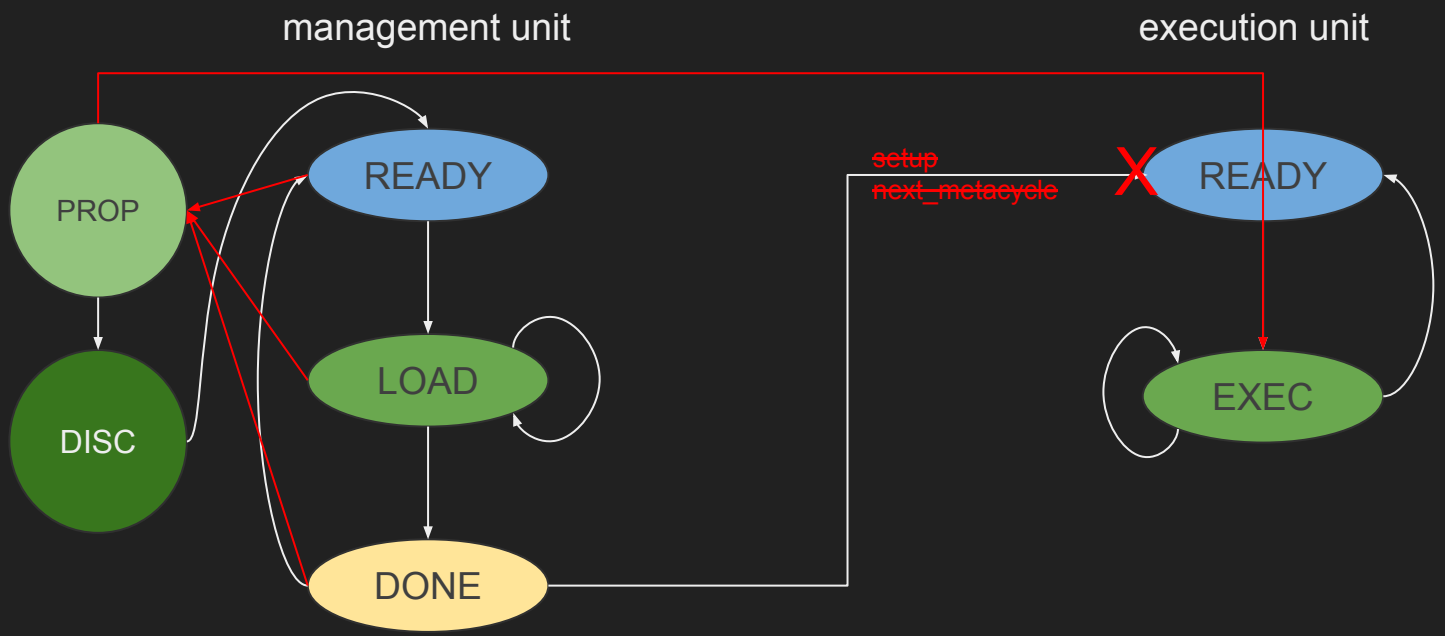
a node may belong to multiple slices

- need to store multiple app binaries
- transitions between slices are handled by the schedule

  => application loading will happen while executing, so we now distinguish between the management unit and the execution unit.

app loading protocol

app loading: failures



app loading: failures

on node and link failures:

- the management unit follows the bootstrap protocol (transition to PROPAGATE)
  - there is no signaling to the execution unit that a “new” application has been loaded
- the execution unit is notified and it is left to the application to decide whether to continue executing (potentially error recovery code) or stopping

implementation

aegir implementation

bootstrap protocol implemented as an aegir application

- validates the protocol
- does not test failures
  - Chris is working on adding support for adding and removing nodes in platform
- protocol will be moved into platform to properly simulate hardware behavior

application loading: not implemented

- requires modeling management units
- the protocol is trivial; the failure handling library is not