Deterministic, asynchronous message driven task execution with ROS

Brian Cairl

- Motivations
  - Preamble on determinism
  - Drawbacks of a timing-dependence in testability
  - Event-driven software in testing

- Asynchronous event-driven software framework
  - Where synchronization meets ROS abstraction
  - High-level implementation details

Overview

Motivations

- With given inputs, can we make any guarantees about software outputs and behavior?

- If we “playback” record sensor data/partial state data, can we get the same outputs as when our software was running live?

- Why do we care?
  - Incident reproproducability
  - Robustness to timing variations

To qualify:

- This talk will address **algorithmic determinism** as a “best effort” attempt at having some level of reproducibility between live scenarios and testing
    - Also, reproducibility between offline test cases
- This will not deal with real-time system determinism

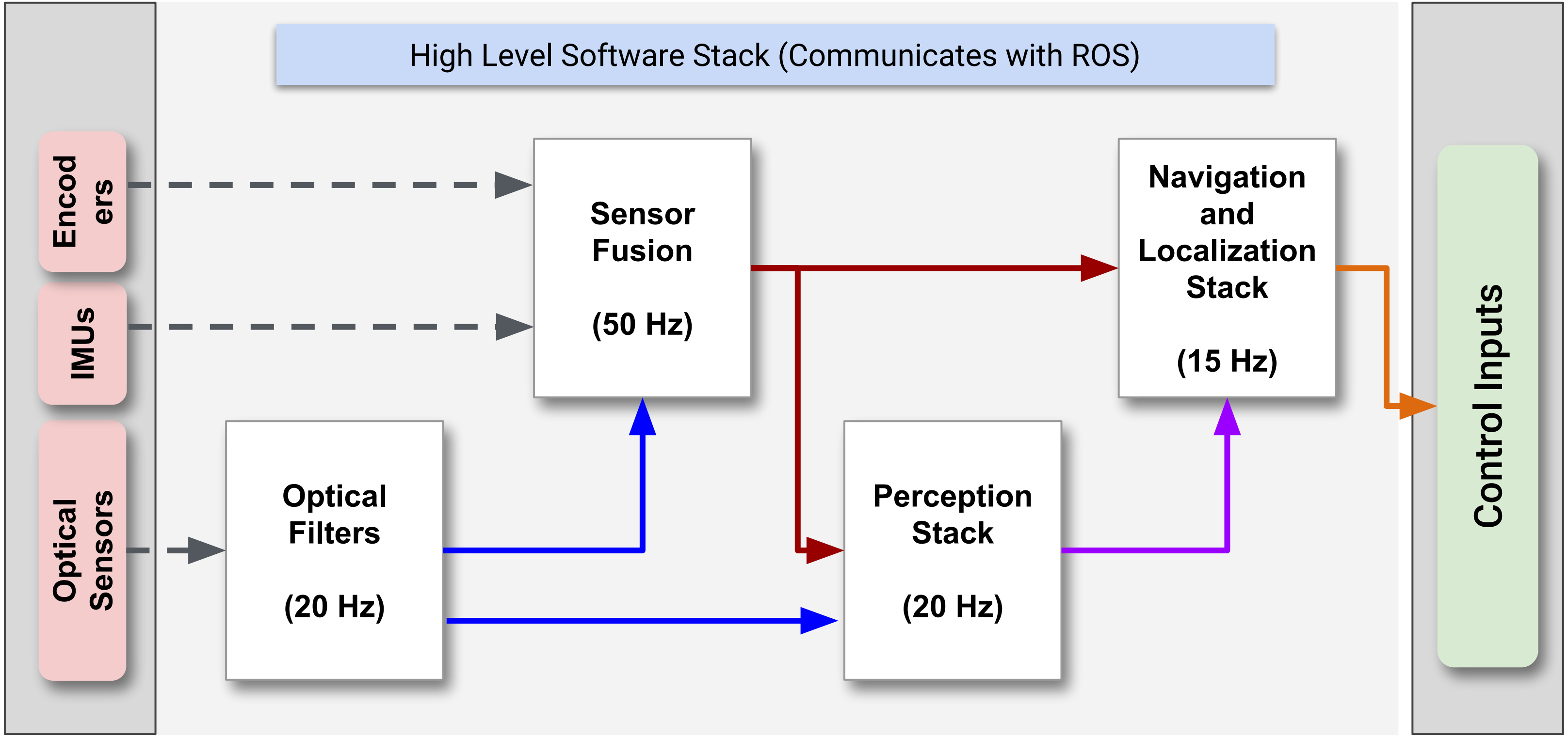
- Typically working with an operating system (e.g. Linux) which is scheduling events and dealing with threads/processing

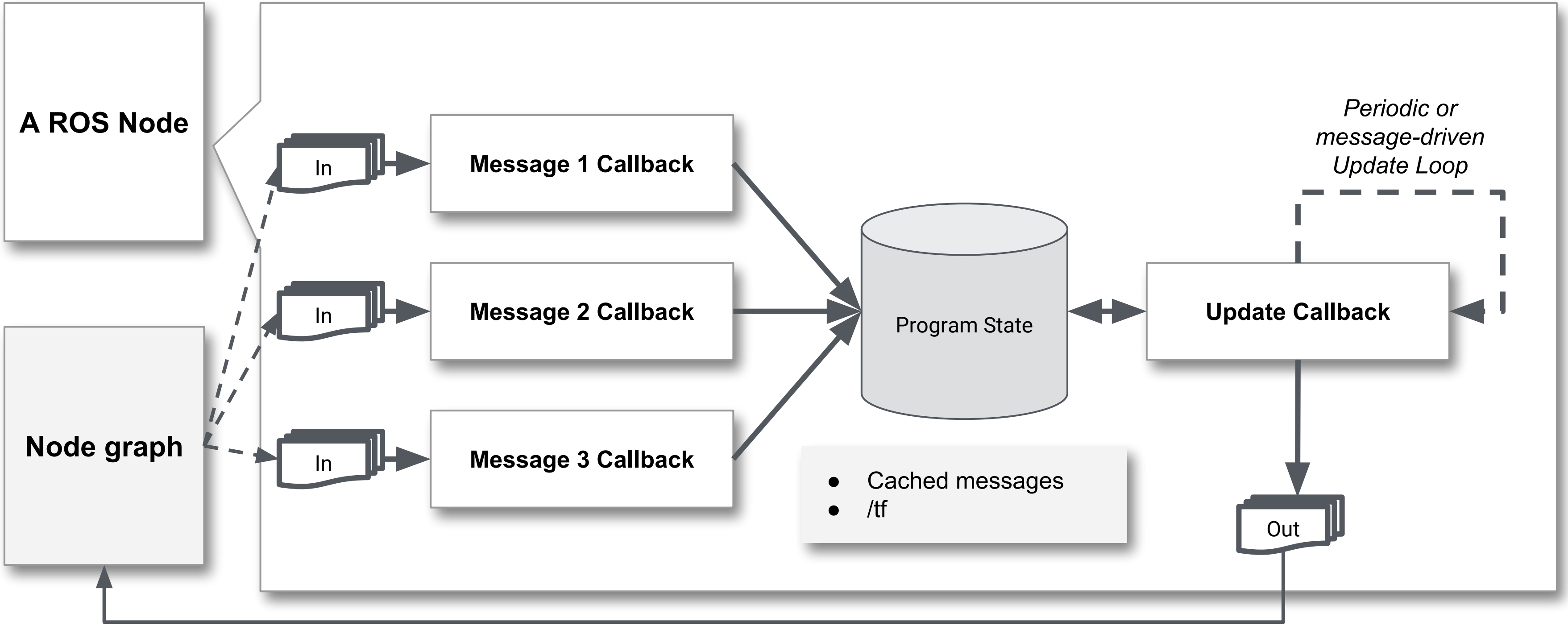
- During runtime:
  - Thread wake up delays
  - Context switching delays
  - Some inherent TCP message transmission and serialization delay
  - Logging, file IO, etc.

- We are usually dealing with:
  - Software which is relatively low-frequency (<200 Hz) and can tolerate some delay (0.1ms - 500ms)
  - a system that is somewhat tolerant to command jitter

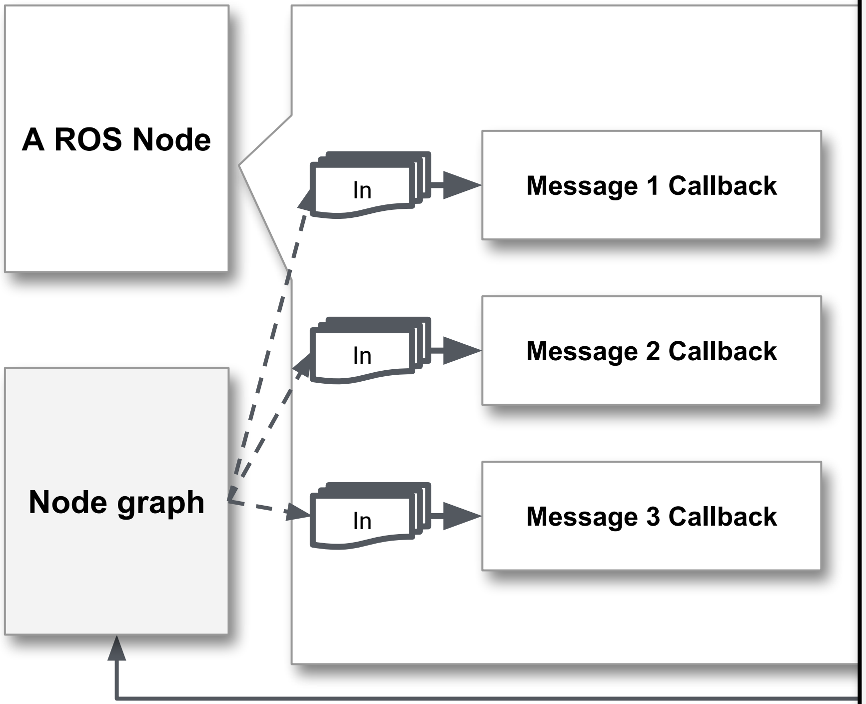
- The host system needs to run fast enough to keep up with incoming data

- Use diagnostic information to figure out whether or not this is (nominally) the case
  - Message output rates
  - Difference between wall time and message stamps


High Level Software Stack (Communicates with ROS)
Encoders
IMUs
Optical Sensors
Sensor Fusion
(50 Hz)
Optical Filters
(20 Hz)
Navigation and Localization Stack
(15 Hz)
Perception Stack
(20 Hz)
Control Inputs



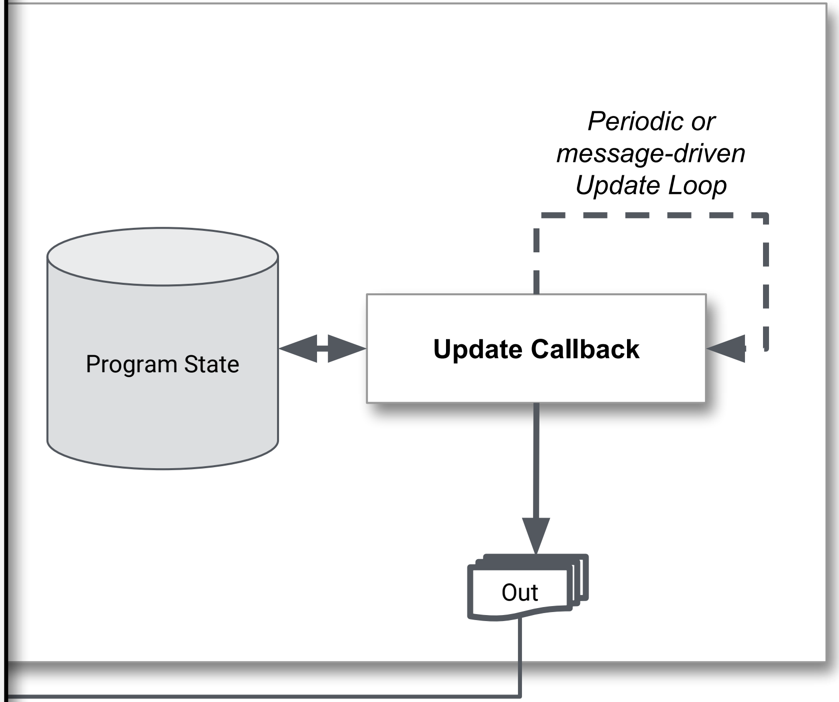
The polling ROS node



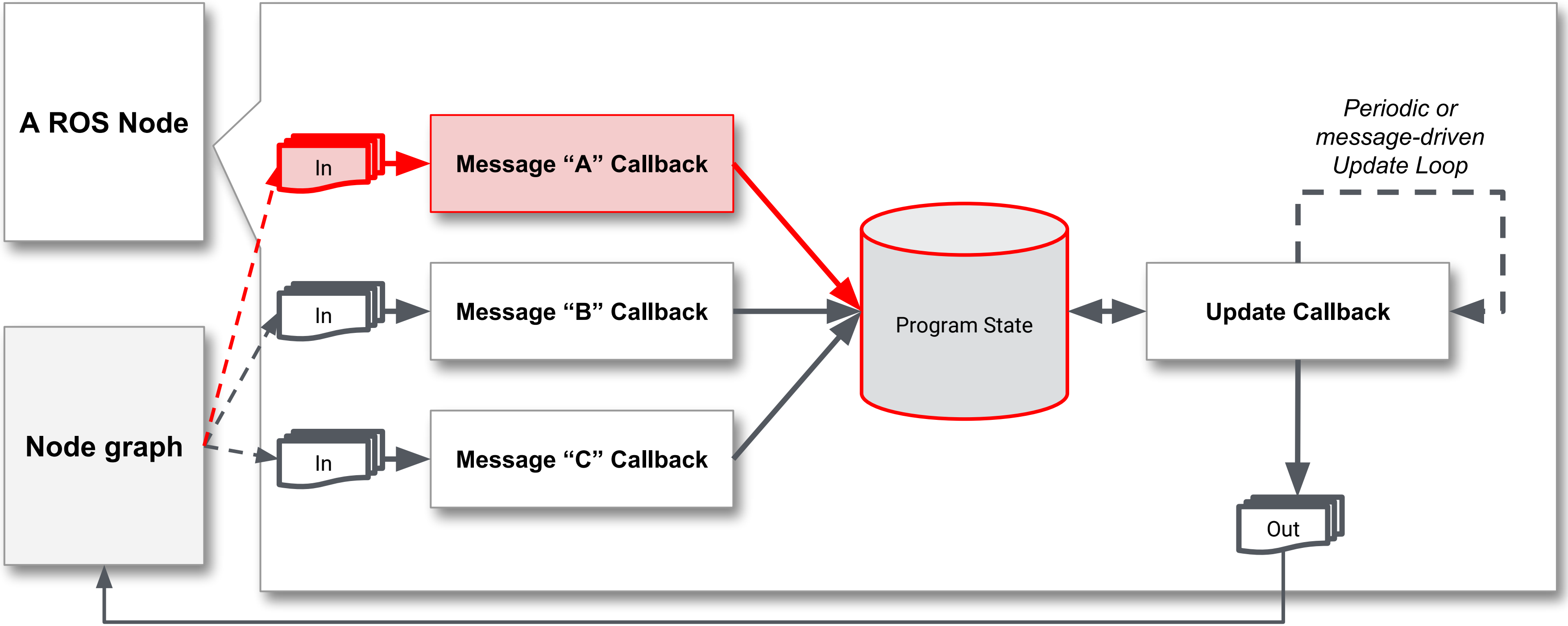

```
class NodeObj
{
Public:
...
  // includes some constructors, init stuff
private:
  ros::Subscriber msg_a_sub;
  MsgA::ConstPtr msg_a;
};

...

{
  // in a method to init things
  sub = nh.subscribe(
    “msg_a”, 10, &NodeObj::callback, this);
}

...

void NodeObj::callback(const MsgA::ConstPtr msg)
{
  this->msg_a = msg;
}
```

The polling ROS node

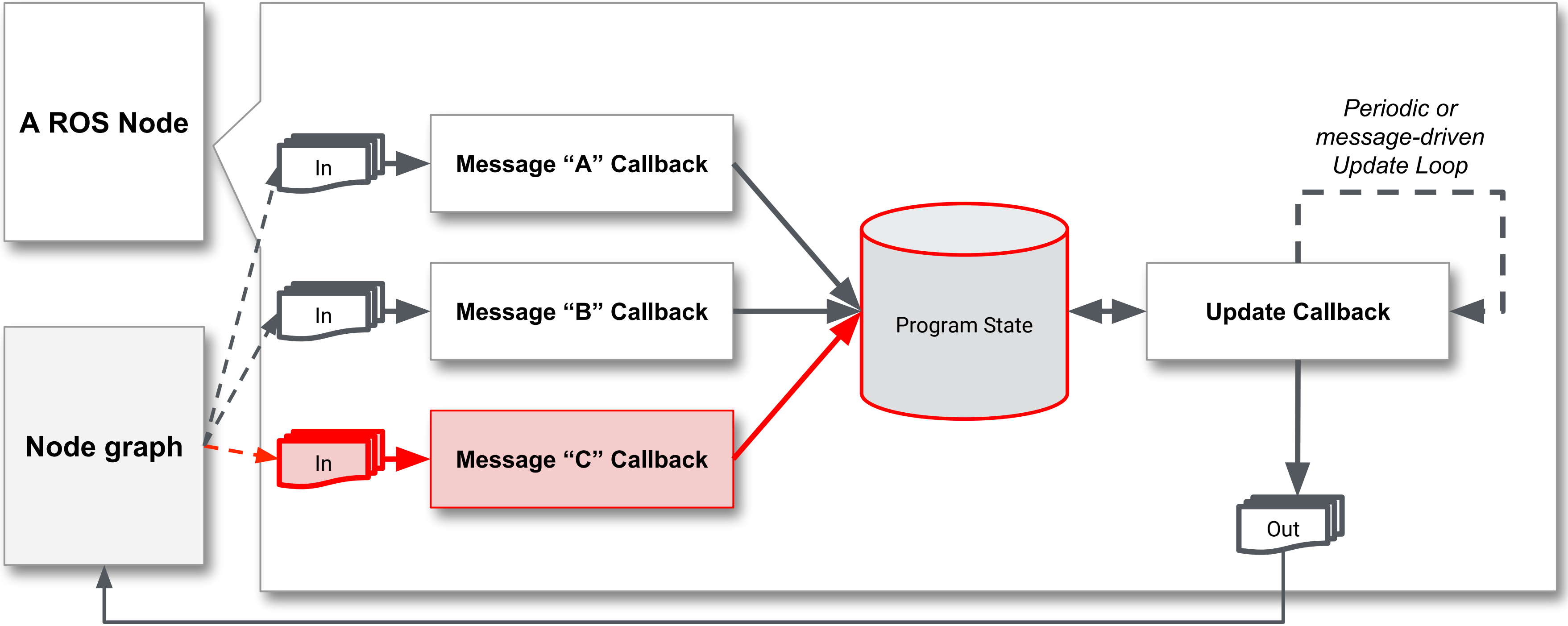
```
class NodeObj
{
...
private:
  ros::Timer updater;
...
  // includes cached messages
};

...

{
  // in a method to init things
  updater = nh.createTimer(
    ros::Duration(.1), &NodeObj::update, this);
}

...

void NodeObj::update(const ros::TimerEvent& evt)
{
  if (this->msg_a && this->msg_b && ... )
  {
    // do a thing
  }
}
```



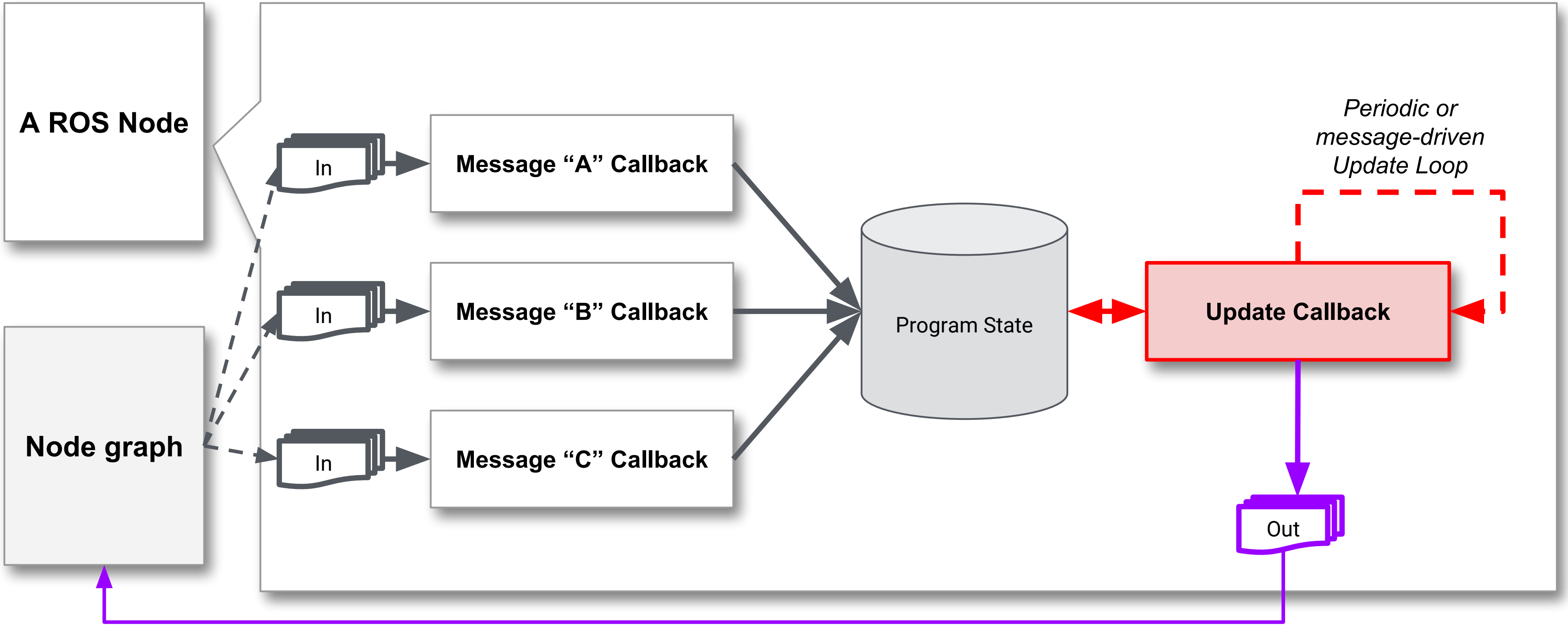
The polling ROS node



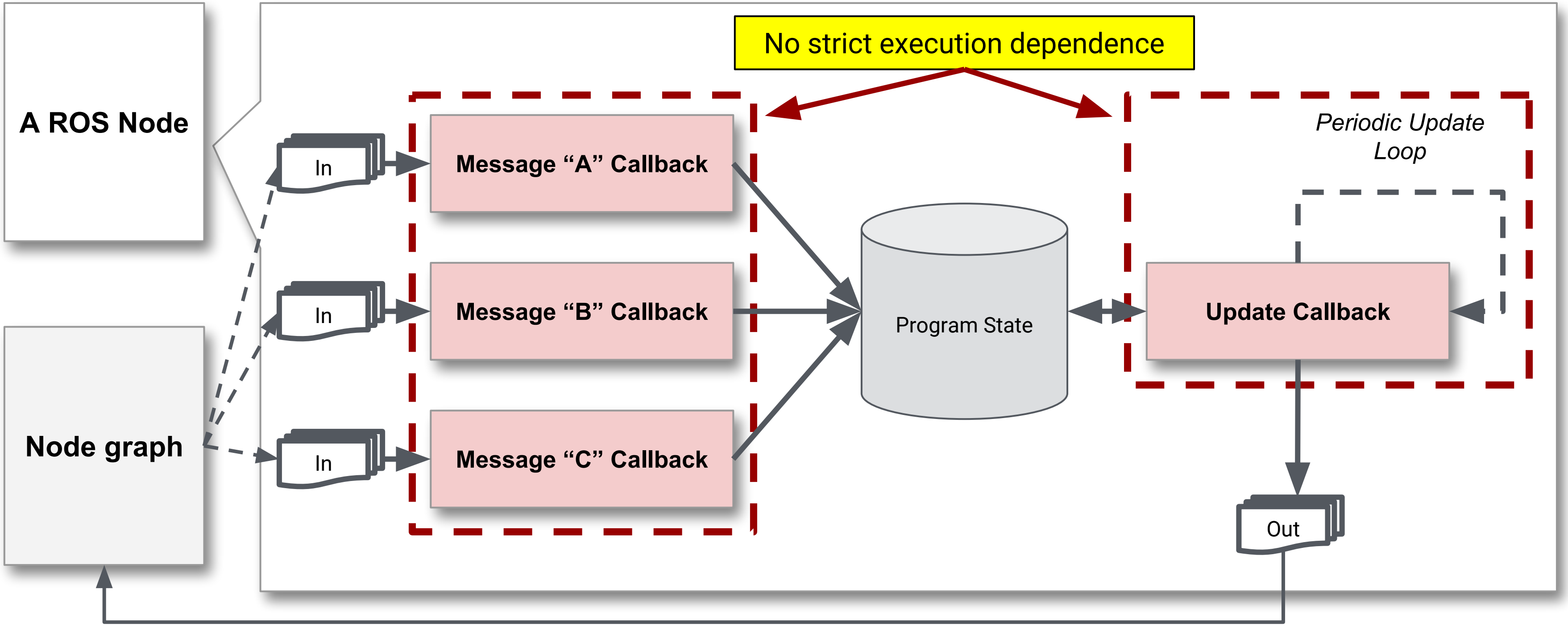
The polling ROS node



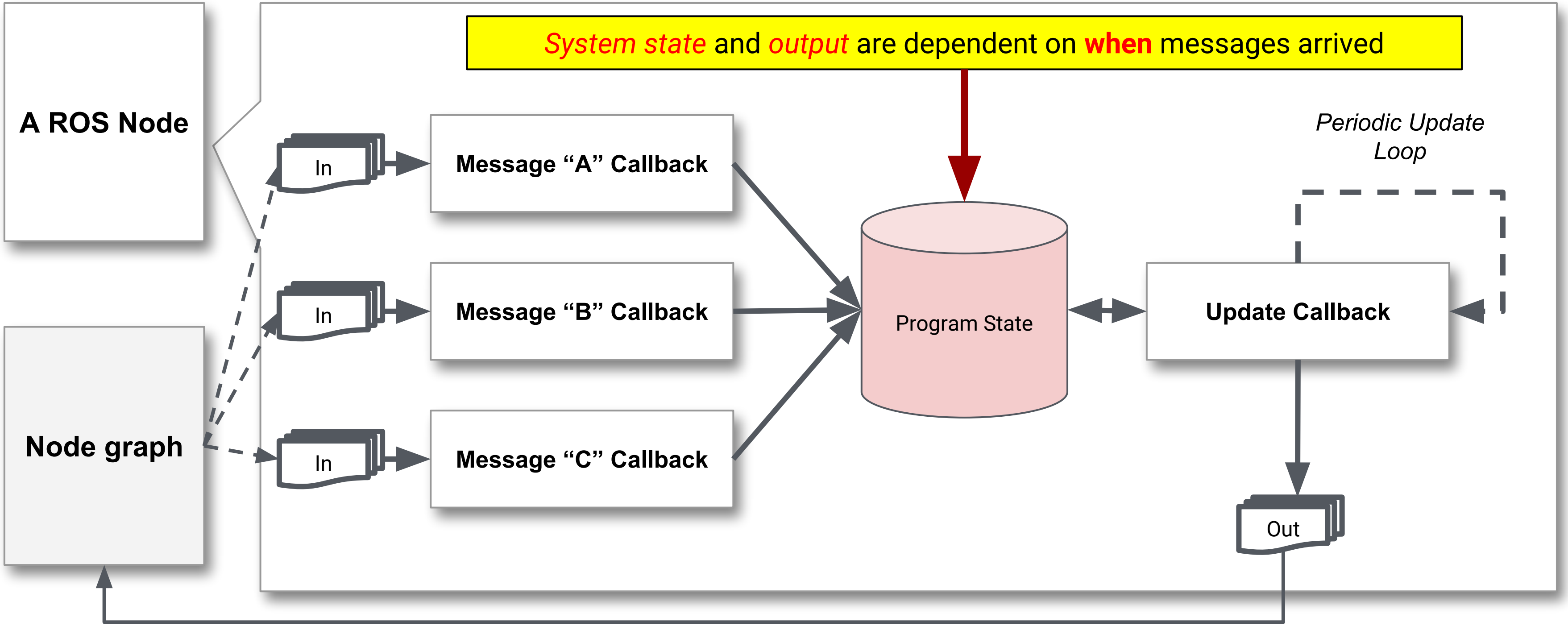
The polling ROS node



The polling ROS node



The polling ROS node

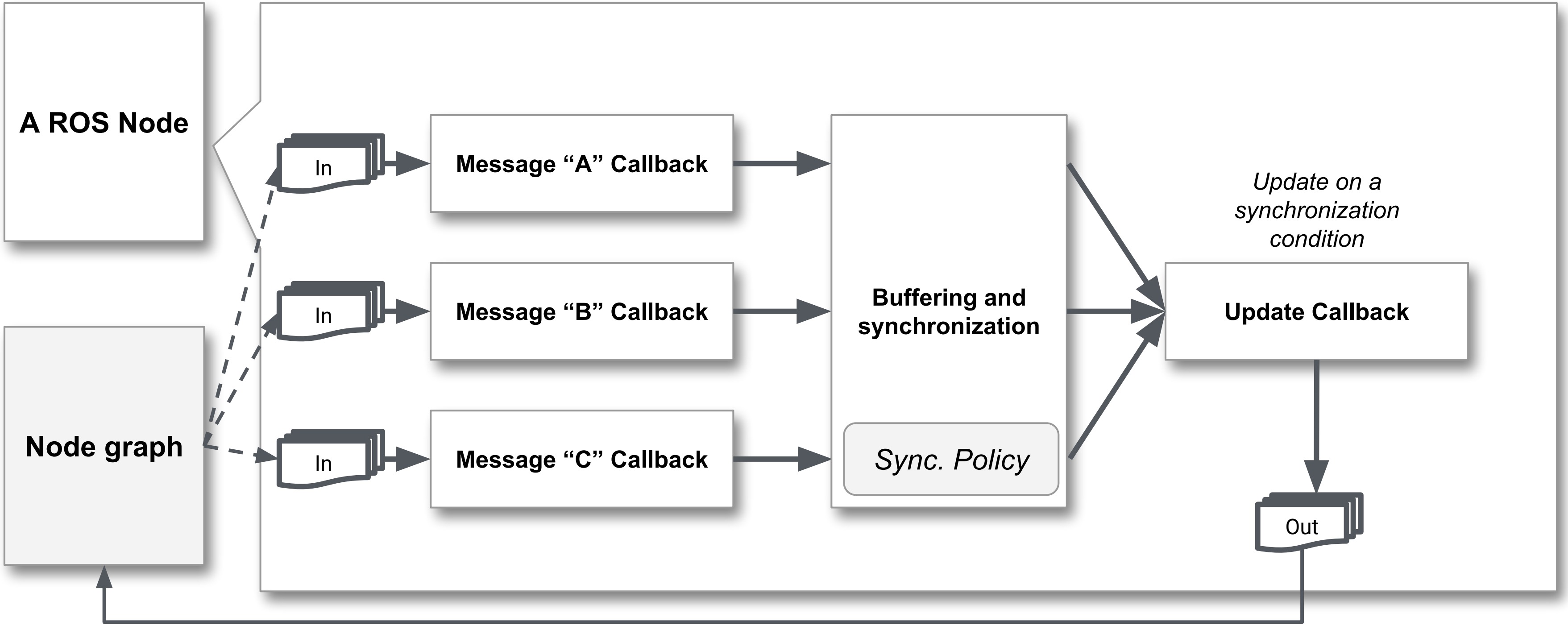


The polling ROS node

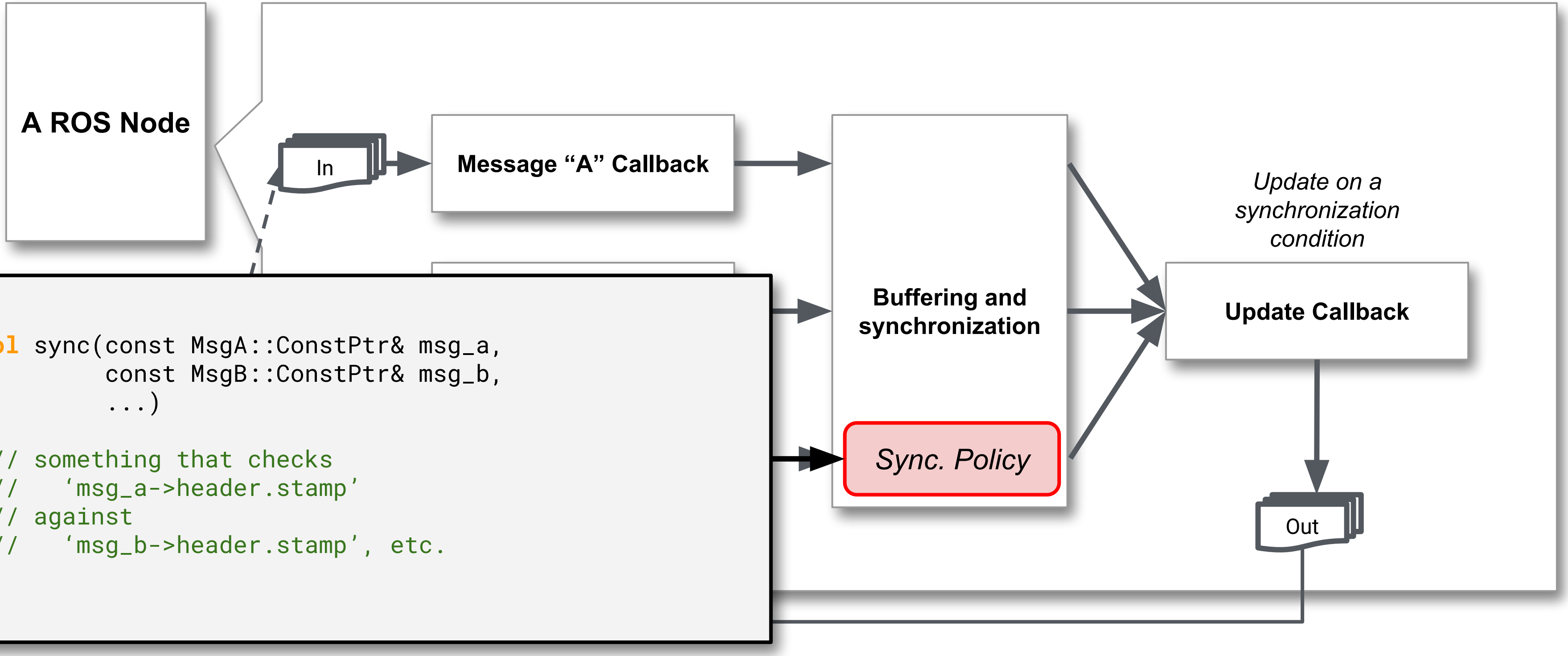
- Can have “zero” delay, since we can output with whatever we have (besides waiting on /tf)

BUT

- Update (output) rate is decoupled from input data
  - Essentially sampling our inputs
  - Output is dependent on *when* we sampled
- Cannot be run at or faster than real-time and guarantee the same results

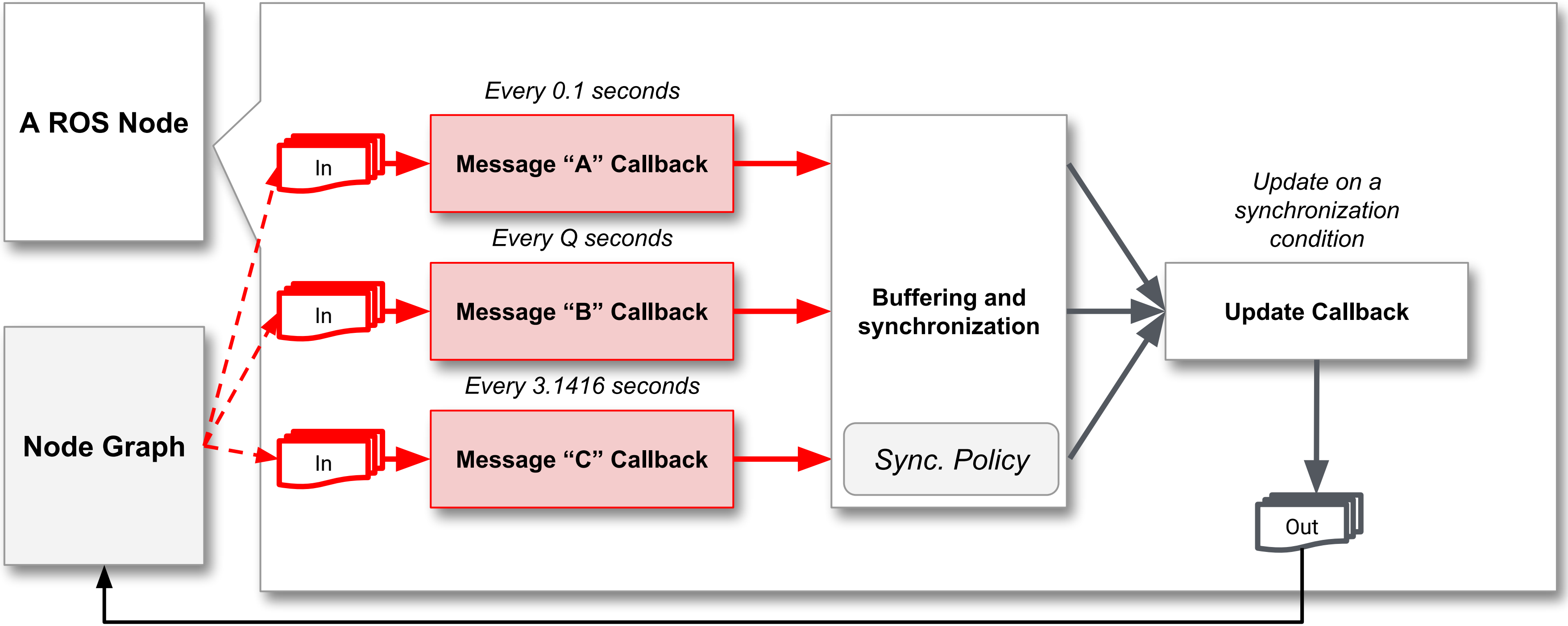


The event-driven ROS node

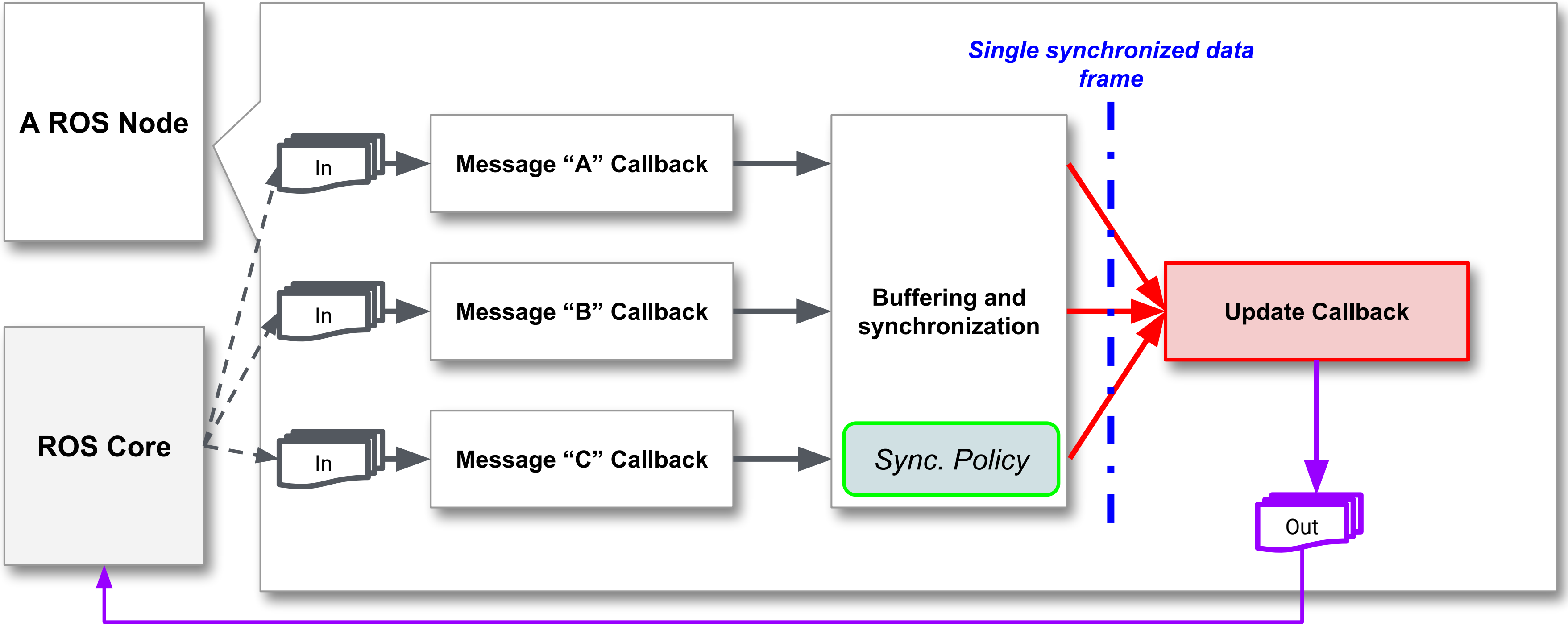



```
bool sync(const MsgA::ConstPtr& msg_a,
          const MsgB::ConstPtr& msg_b,
          ...)
{
  // something that checks
  //   'msg_a->header.stamp'
  // against
  //   'msg_b->header.stamp', etc.
}
```

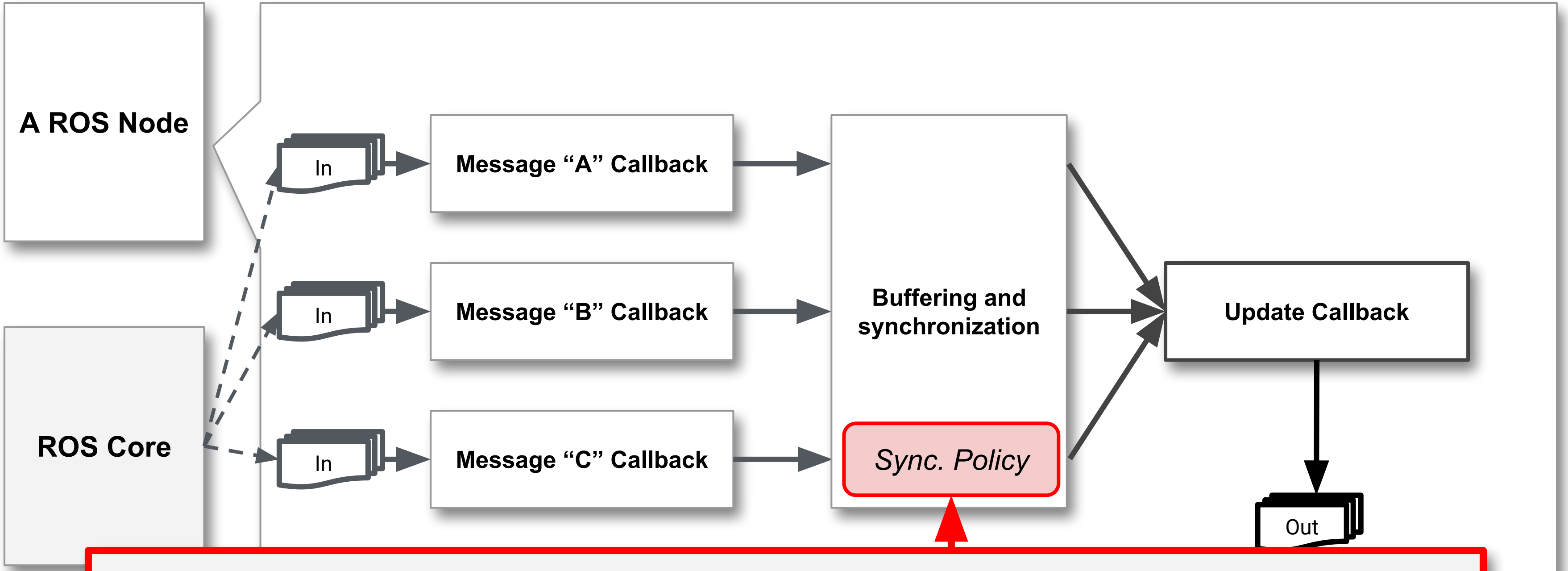
The event-driven ROS node



The event-driven ROS node



The event-driven ROS node



http://wiki.ros.org/message_filters

The event-driven ROS node

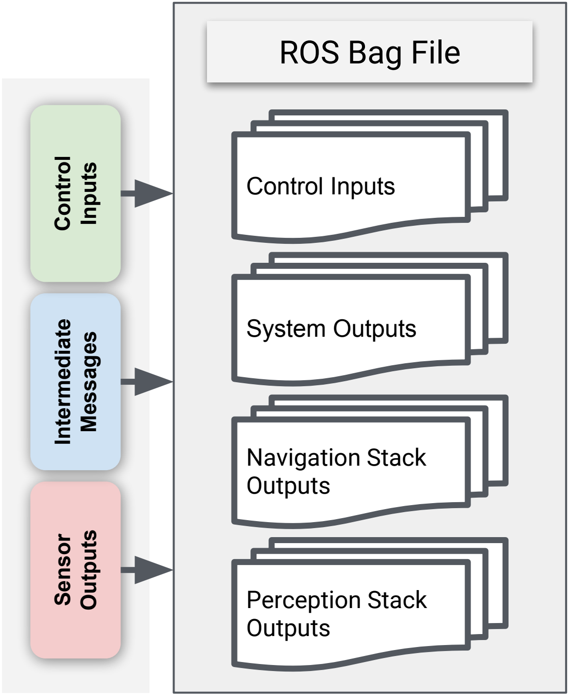
- Can be robust against message interleaving at runtime, at the expense of delay
- Delay is passed on from one node to dependent nodes, but delay can be calculated beforehand
- Running with the same input data will always produce the same outputs

- Removes ambiguity about software brittleness under different timing/system load conditions
  - Repeatable functional tests

- Can run faster than real-time
  - Important for simulation where randomized system configurations/inputs can be tested quickly

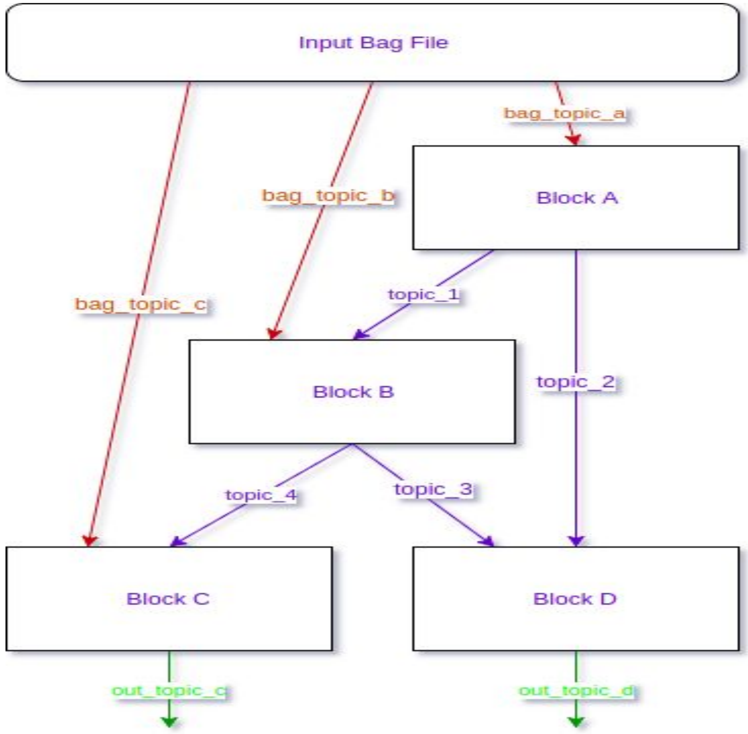
- We can test with real data and be reasonably confident that we can reproduce errors with said data

- Recorded data could represent conditions that uncovered an edge case that caused an incident, e.g.:
  - Robot stuck behind an obstacle
  - Robot didn’t track an important object of interest
- To guarantee that an edge case can be circumvented, software determinism is key to guarantee repeatability



With an event-driven system:

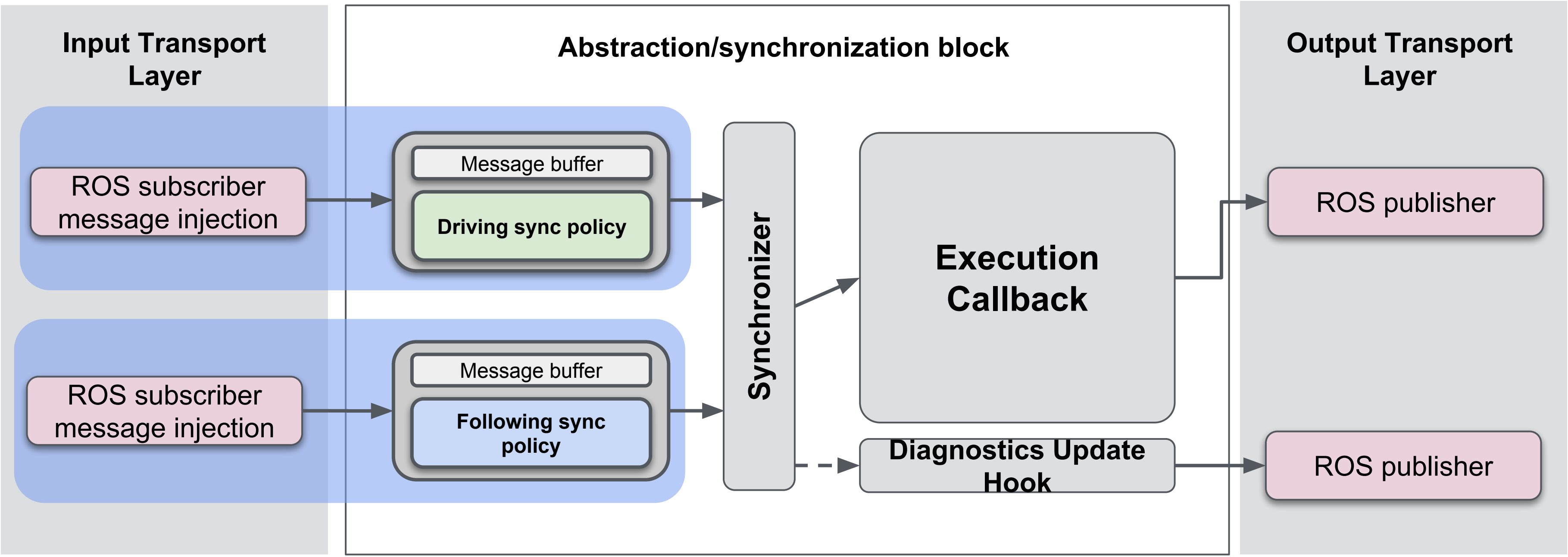
- We don't need a ROS core and we don't need write `ros_test` cases
- Make test cases from bag files (see rosbag API)
- This requires a some extra architectural considerations



Flow

An asynchronous, event-driven framework

- Maintain overarching ROS node-based structure
- Decouple execution portion and communication portion of the code
- Make execution event-driven (deterministic)
- Support intra/extra node communication
    - Support message injection/production without a ROS core
- Allow multiple execution units (blocks) to run in the same program, similar to nodelets



Flow Framework: *Desired unit structure*

Input Transport Layer

ROS subscriber message injection

ROS subscriber message injection

- Only responsible for pumping messages in and moving messages out from our abstractions layer
- Not really dependent on ROS
  - In the case of ROS subscribers, we can inject/received messages with ROS callback queue from single thread
  - Replace with ROS2 subs/pubs
  - Directly inject messages from a bag

Output Transport Layer

ROS publisher

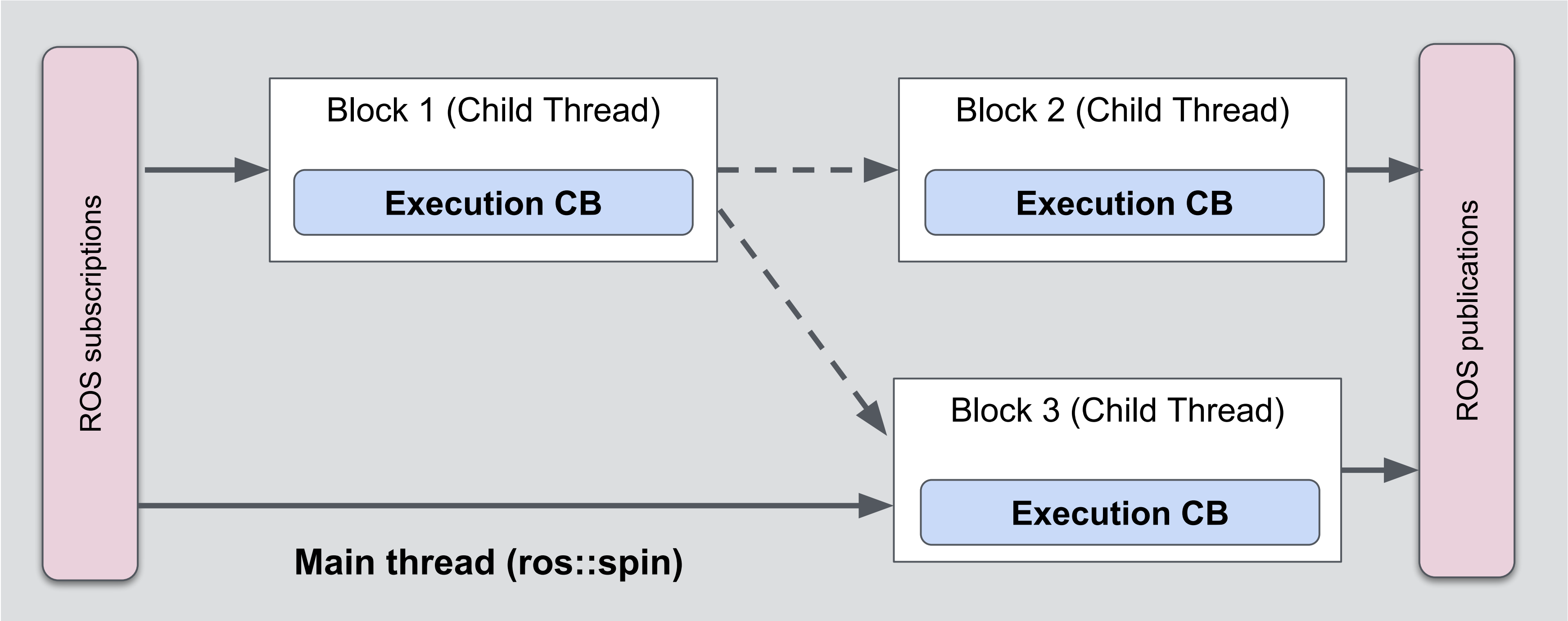
ROS publisher

Flow Framework: *Desired unit structure*

- Nodes can contain 1 or more blocks

- Blocks run in parallel, each in a separate thread

- Blocks are connected through input and output channels
  - Can interface with ROS or another transport layer
  - Can interface with other blocks
  - Blocks pass messages (or any data type, if intraprocess)
  - Input channels govern synchronization behavior

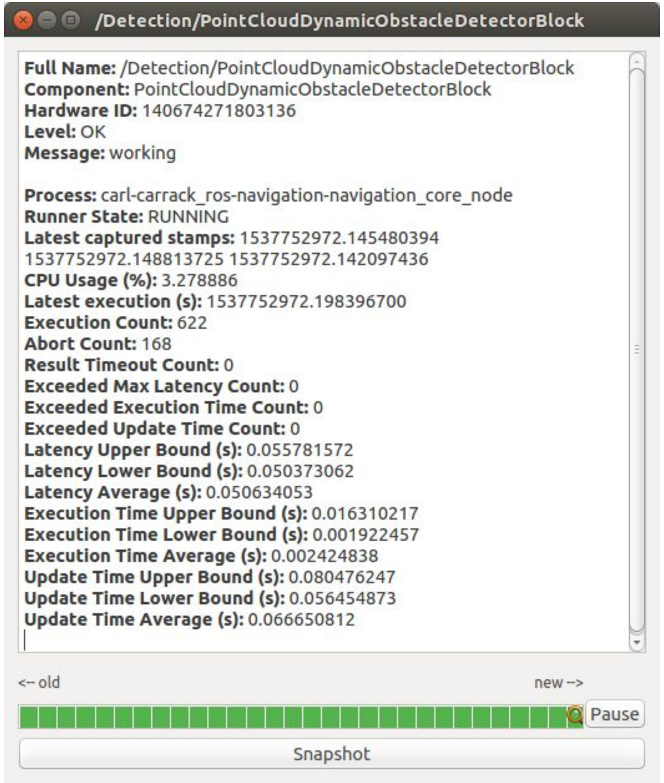

ROS subscriptions
Block 1 (Child Thread)
Execution CB
Block 2 (Child Thread)
Execution CB
Block 3 (Child Thread)
Execution CB
ROS publications
Main thread (ros::spin)

Blocks obfuscate parallel design

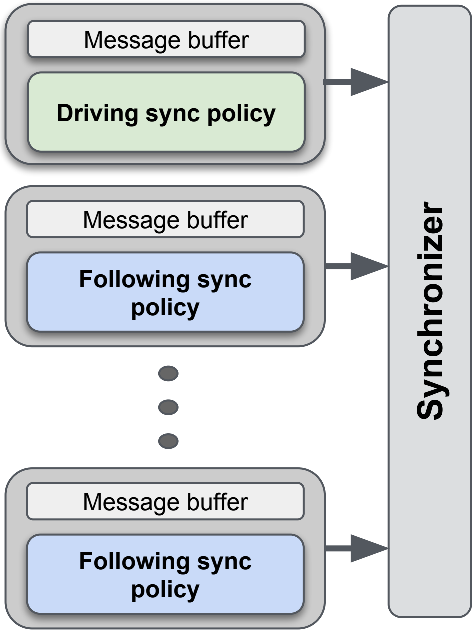
- Thread execution is driven by incoming data
- Thread will sleep when not executing
- Thread safety is enforced by the wrapping structure
- System design comes down to what the block will execute, and how its connected to other things
- The connection methods are *interchangeable*

- Diagnostic hooks can be attached to each block, as part of the block design

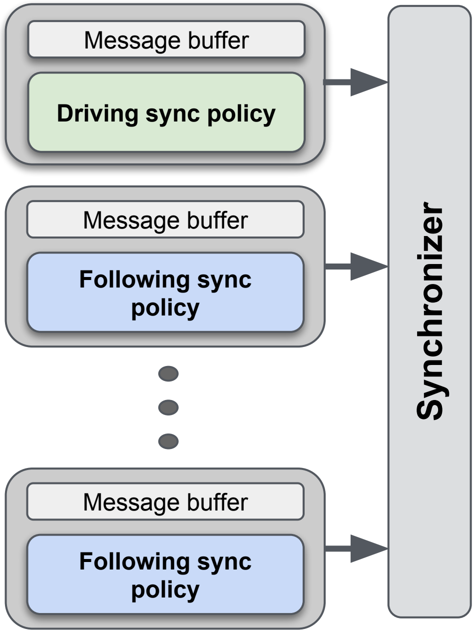
- Enables per-task execution monitoring



diagnostic_msgs

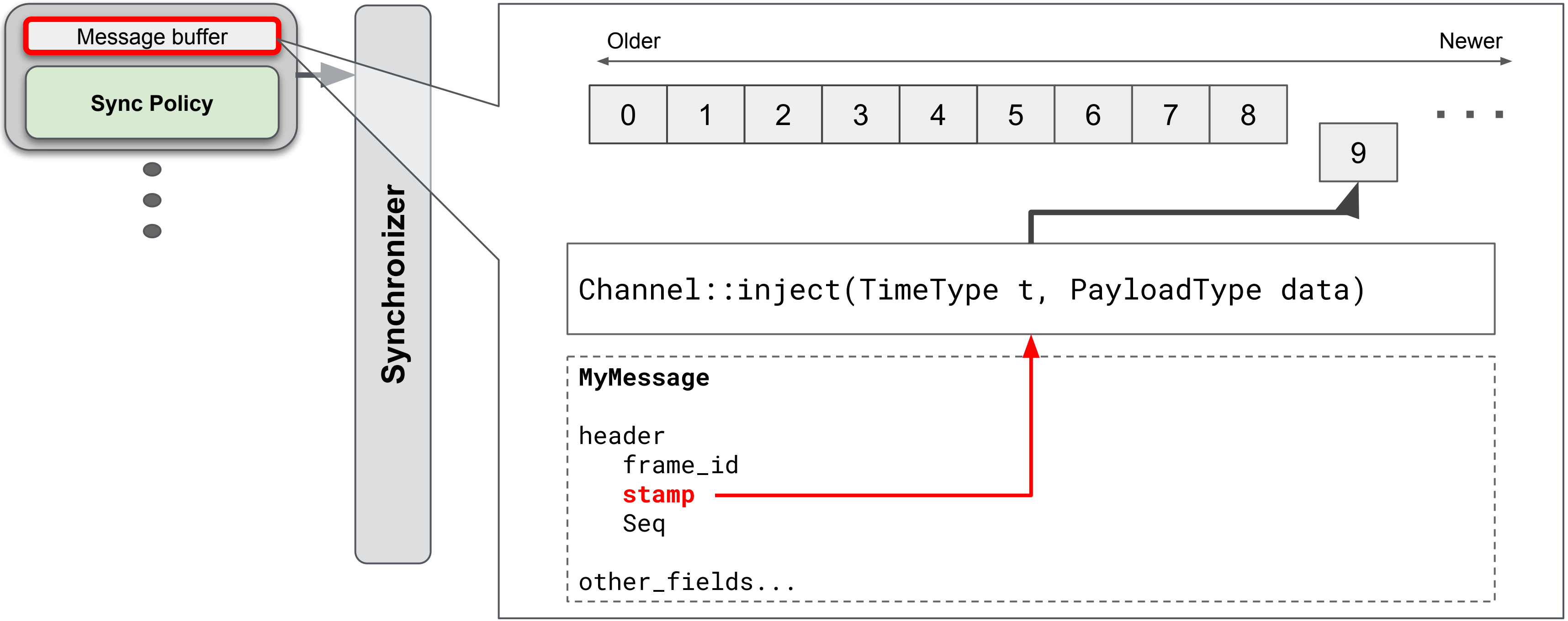


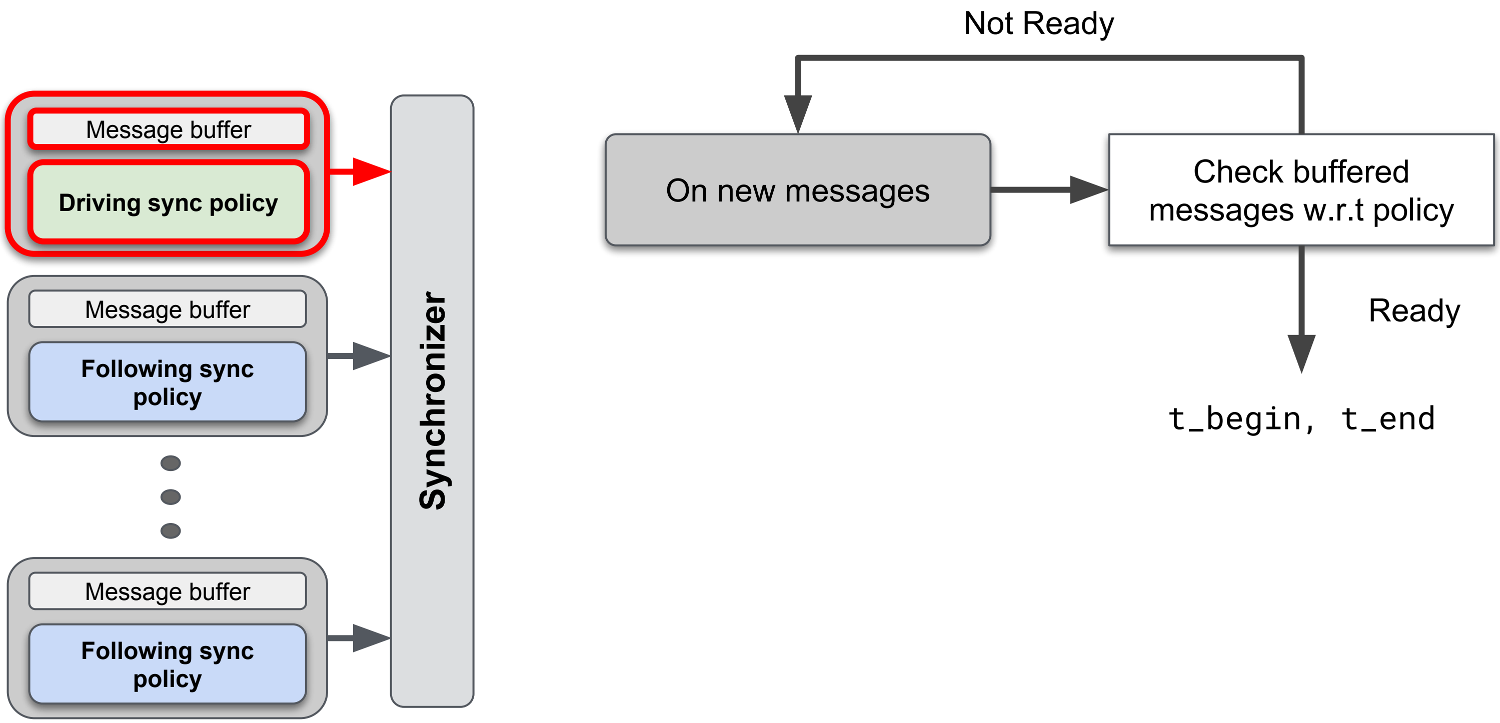
- First input “drives” synchronization
- Additional inputs are synchronized based on a time range from driving input
- Each sync. policy knows how to deal with discarding irrelevant data or skipping frames
- Synchronizer outputs a data frame with messages for each input channel

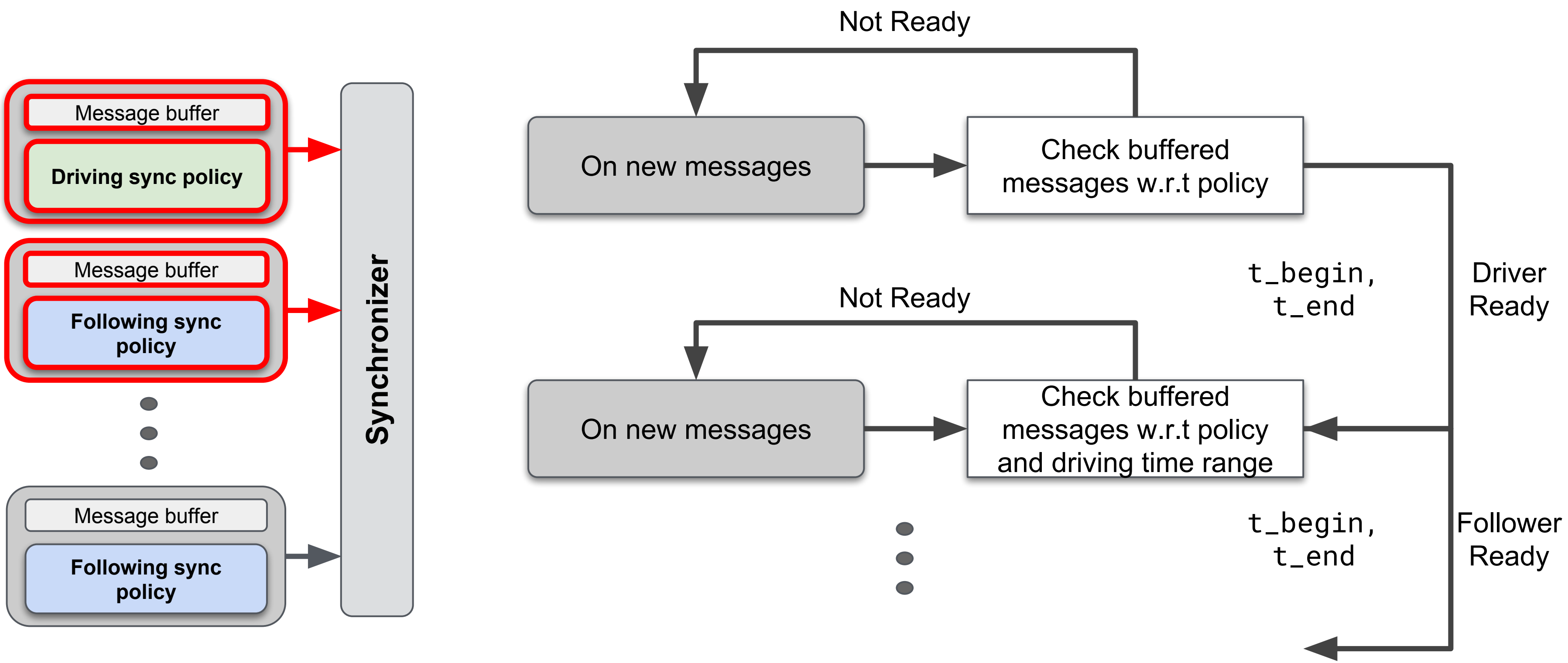


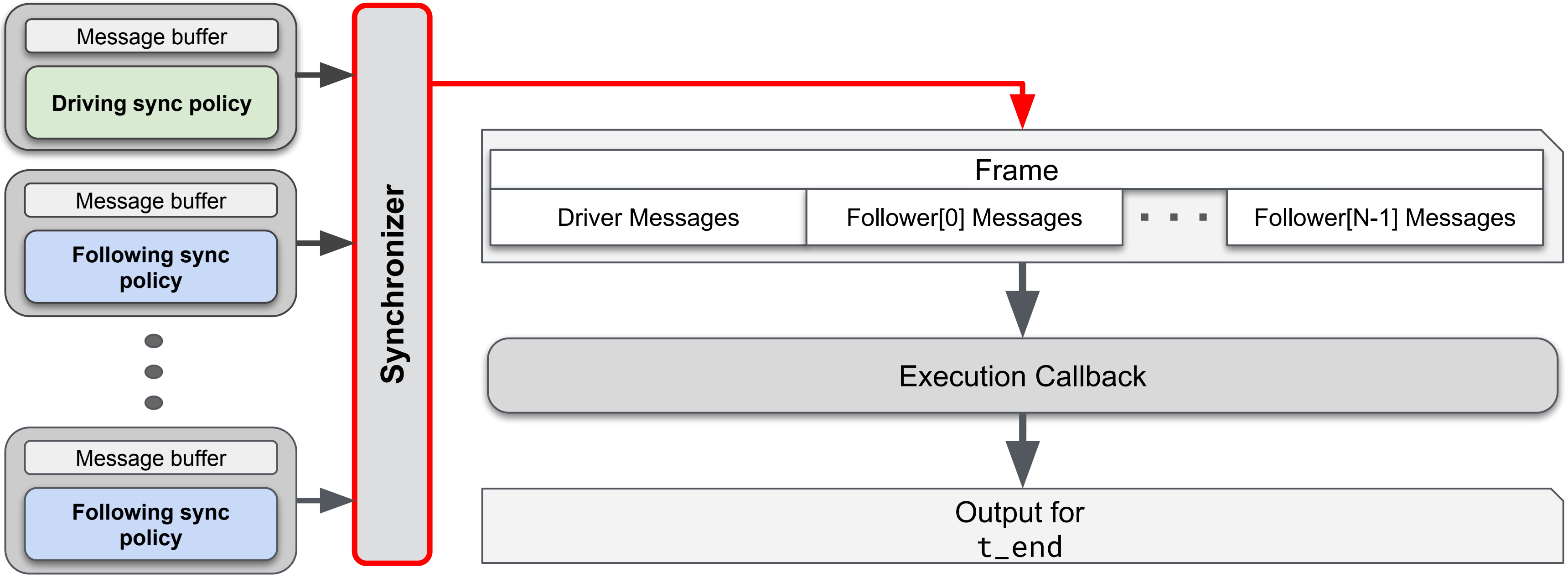
- Synchronization policies are part of the channel, which determine overall synchronization behavior

- There are a few extra directives that each policy can emit to skip or abort on a synchronization attempt


Message buffer
Sync Policy
Synchronizer
Older
Newer
0
1
2
3
4
5
6
7
8
9
Channel::inject(TimeType t, PayloadType data)
MyMessage
header
frame_id
stamp
Seq
other_fields...

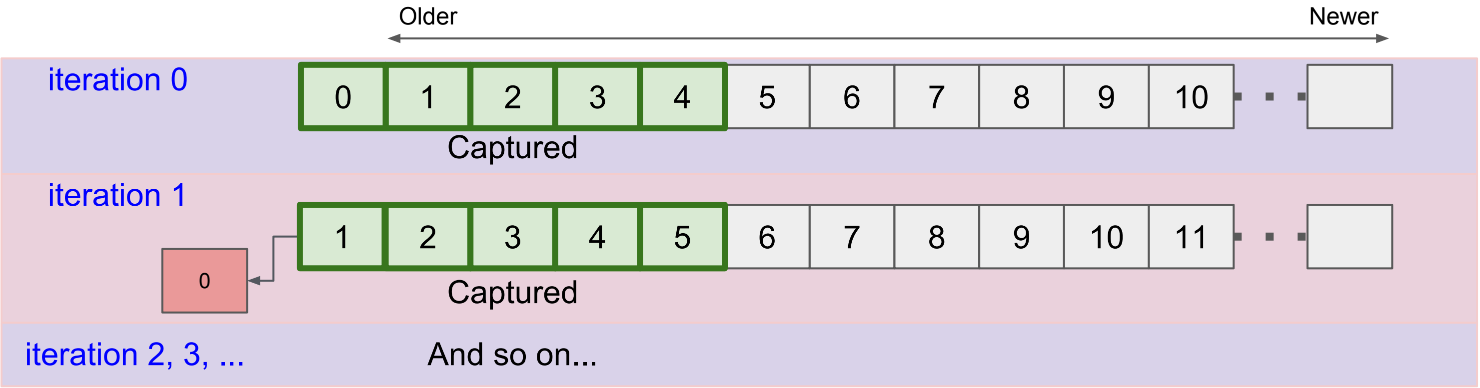

Message buffer
Driving sync policy
Message buffer
Following sync policy
Message buffer
Following sync policy
Synchronizer
Not Ready
On new messages
Check buffered messages w.r.t policy
Ready
t_begin, t_end


Message buffer
Driving sync policy
Message buffer
Following sync policy
Message buffer
Following sync policy
Synchronizer
Not Ready
On new messages
Check buffered messages w.r.t policy
t_begin, t_end
Driver Ready
Not Ready
On new messages
Check buffered messages w.r.t policy and driving time range
t_begin, t_end
Follower Ready


Message buffer
Driving sync policy
Message buffer
Following sync policy
Message buffer
Following sync policy
Synchronizer
Frame
Driver Messages
Follower[0] Messages
Follower[N-1] Messages
Execution Callback
Output for
t_end

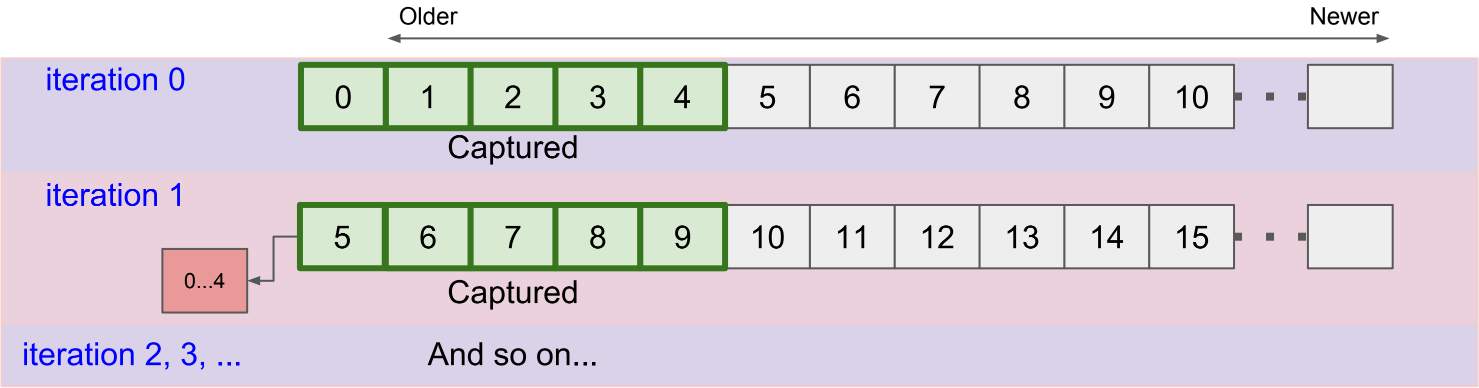
Next-N (sliding window)

- Return latest message, and N-1 messages before, ordered in time
- Synchronize on time range between (N-1)th message stamp and latest message stamp
- Discard oldest message



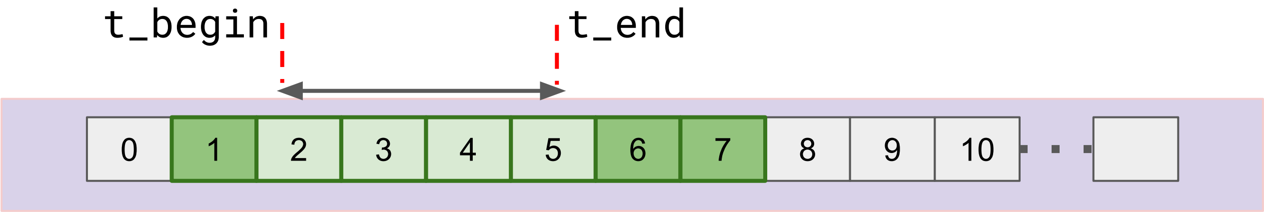
Next-N (without replacement)

- Return latest message, and N-1 messages before, ordered in time
- Synchronize on time range between (N-1)th message stamp and latest message stamp
- Discard all captured messages



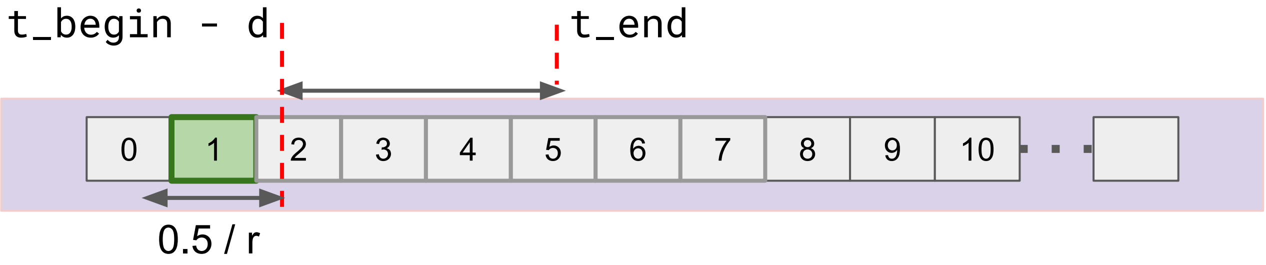
N-Before, M-after

- Return ***N*** messages before the earliest driving stamp, and ***M*** messages after the latest stamps
- Invalidate frame if ***N*** before cannot be grabbed from the buffer
- Wait for data if ***M*** after cannot be cannot be captured



Closest Before

- Assumes an input rate, ***r,*** and a period of delay, ***d***
- Return closests message before earliest driving time stamp that fallse within (0.5/**r**) s of this stamp minus delay period
- Wait if there are only messages earlier than (0.5/**r**) s
- Discard frame if there are only messages after the earliest driving stamp



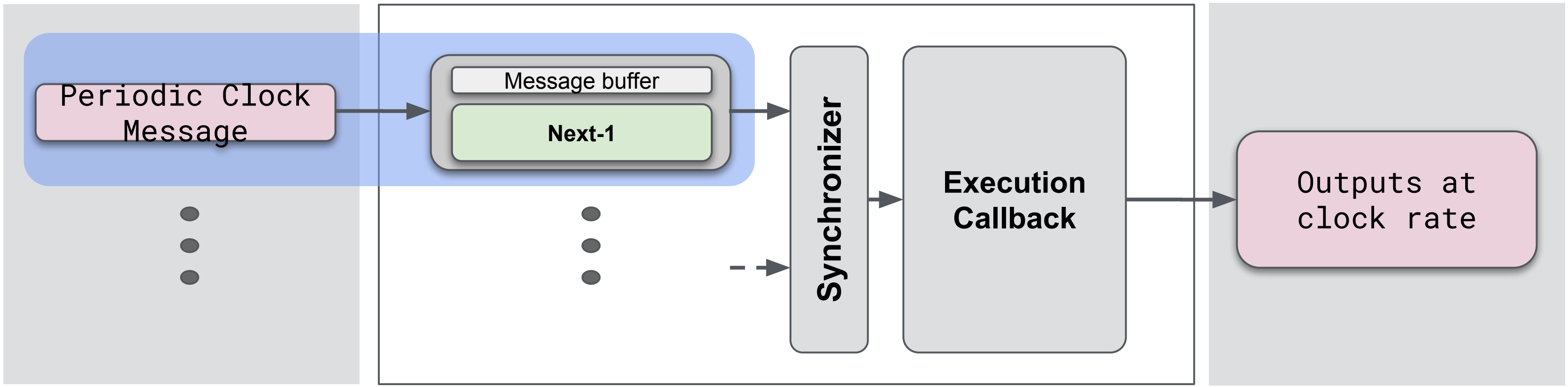
Latched

- Return latest message that occured before the earliest driving stamp
- If such a messages does not exist, invalidate all frames until earliest driving stamp is older than latched stamp

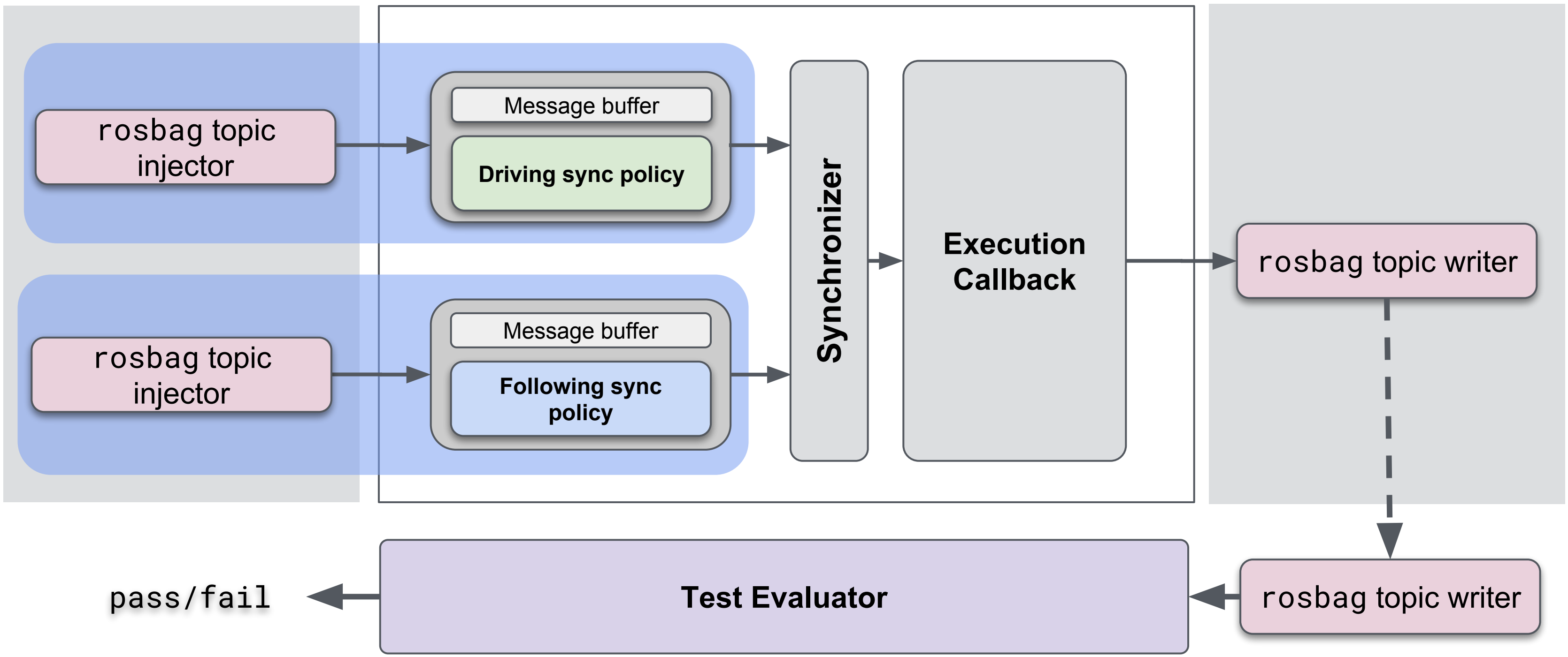
Activation

- Same as latched, but returns message only when input data satisfies a particular condition
- Used to dump frames and effectively deactivate a block

- Using the described input policies, we can "fake" output-driven events by attaching driving inputs to periodic clock message publishers



Execution portion of our code remains unchanged between test and live software



- Deterministic software is critical in testing and reproducing issues
- If software is deterministic, you can have higher confidence in edge-case avoidance when testing against data from incidents
    - A good way to perform this testing is with rosbag file data
- Flow is a framework built on event-driven execution that with ROS agnostic message passing in mind
    - In the process to become an open source framework