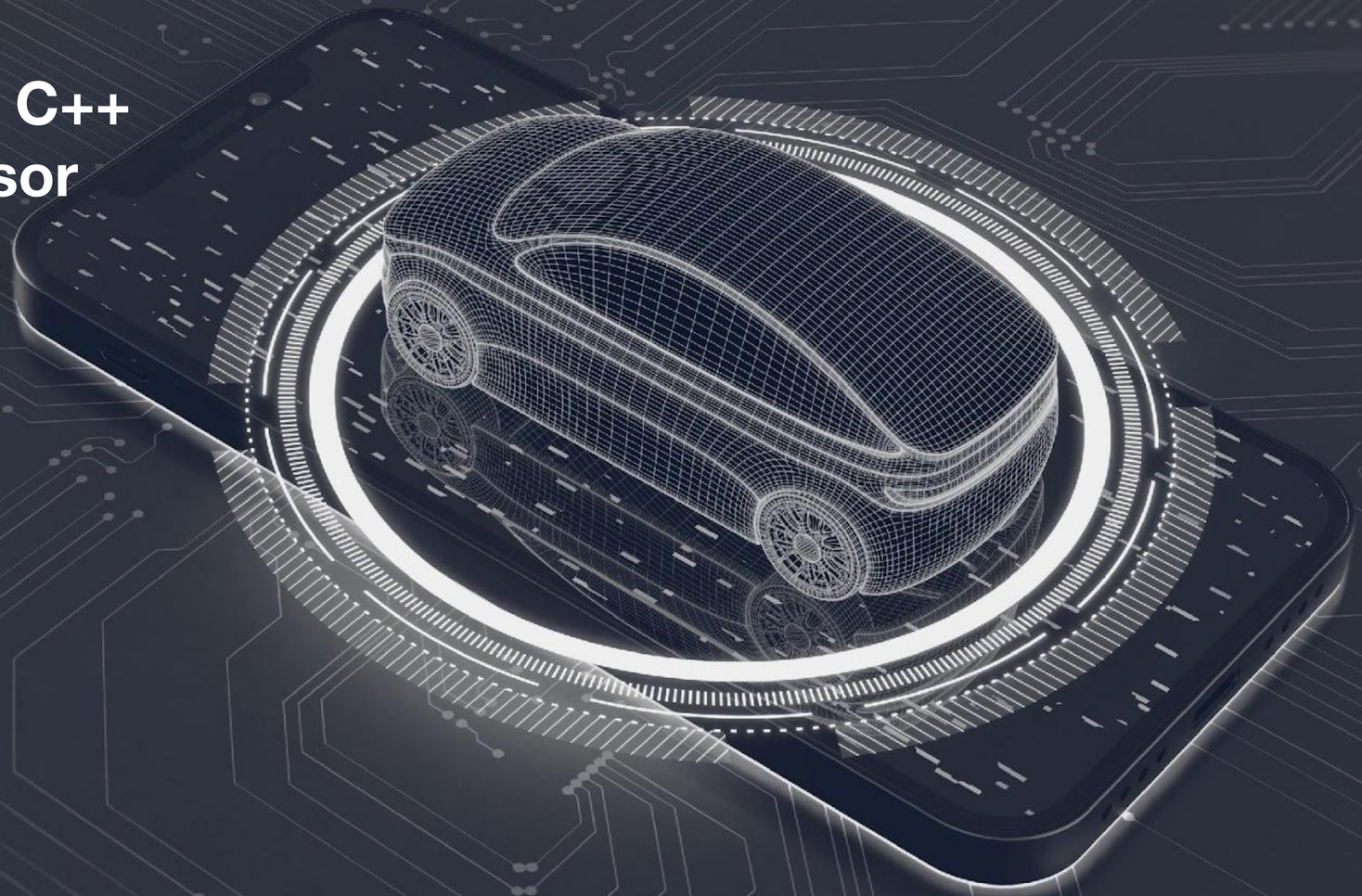
CppCon 2023

Apex.AI

Building bridges: Leveraging C++ and ROS for simulators, sensor data and algorithms



Divya Aggarwal

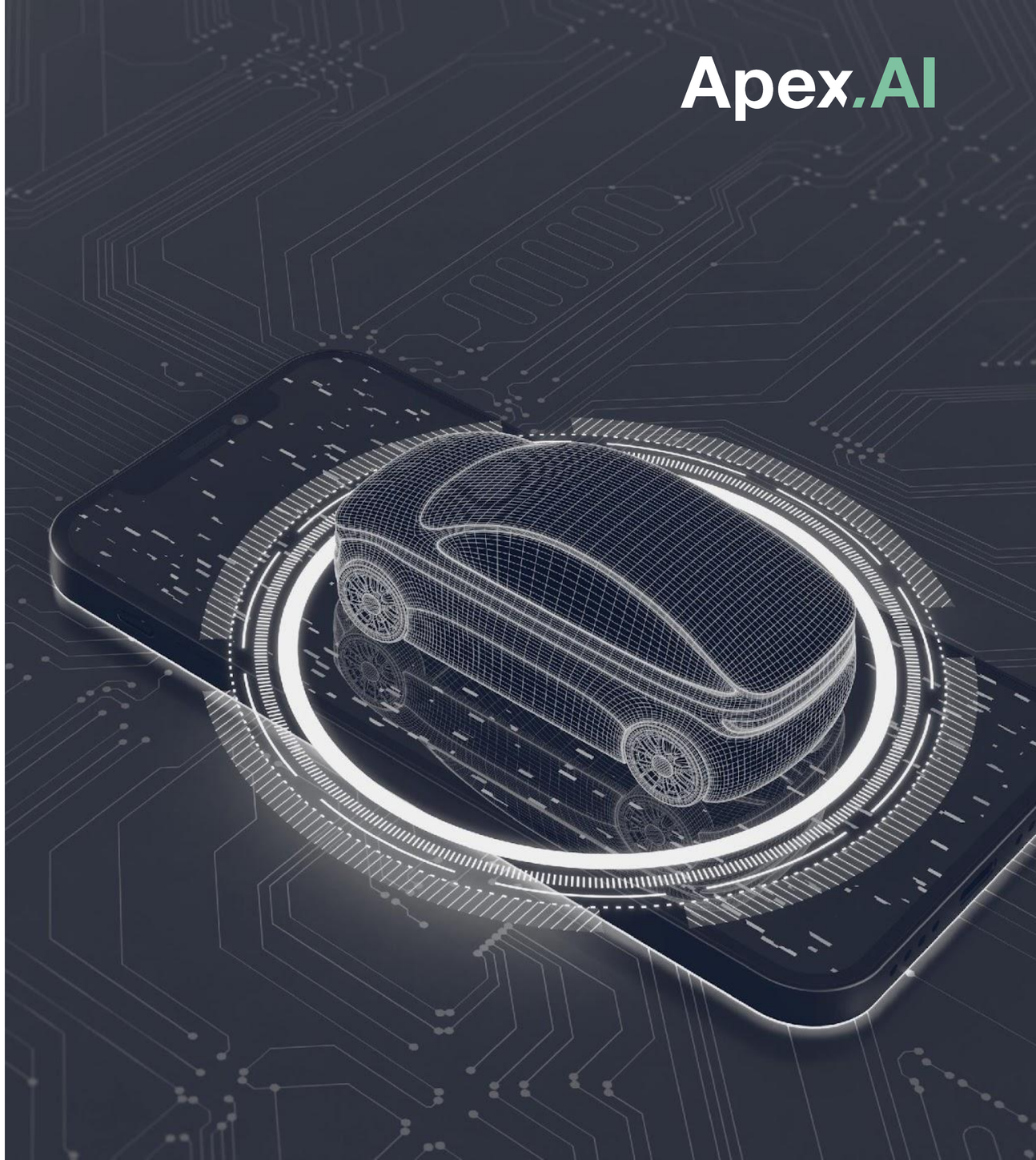
October 5, 2023

Need for Speed? Asphalt? GT?
This is an autonomous driving simulator.

Agenda

1. ROS and simulators
2. What is missing?
3. Our proposal

ROS and simulators



Back to basics — ROS

ROS is an open-source **Robot Operating System**

1. A set of software libraries and tools that help you build robot applications that work across a wide variety of robotic platforms

2. Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory and development continued at Willow Garage

3. Since 2013 managed by OSRF (Open Source Robotics Foundation)

4. Production software for robotics applications usually demands low-level access for device drivers and sensors in ROS which is a very suitable use case for C++

The ROS developer community

- **Practically all robotic labs and autonomous driving programs at universities use ROS**
  - There are 10,000+ ROS developers
  - There are thousands of ROS based packages
  - There are complete open-source driving stacks based on ROS
  - There is a rich and open source toolchain

- **Hundreds of companies are using ROS for R&D and production**
  - Lawnmowers, vacuum cleaners, warehouse logistics, …
  - 80% of Tier-1 and OEM automotive companies working on autonomous vehicles are using ROS
  - There are 1000+ vehicles running ROS-based applications today

ROS communication

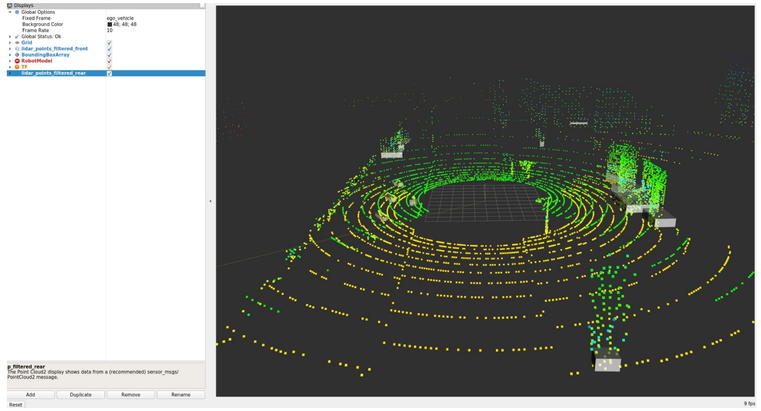
reads: raw sensor's data

ROS driver node

publishes: /sensor topic

subscribes: /sensor topic

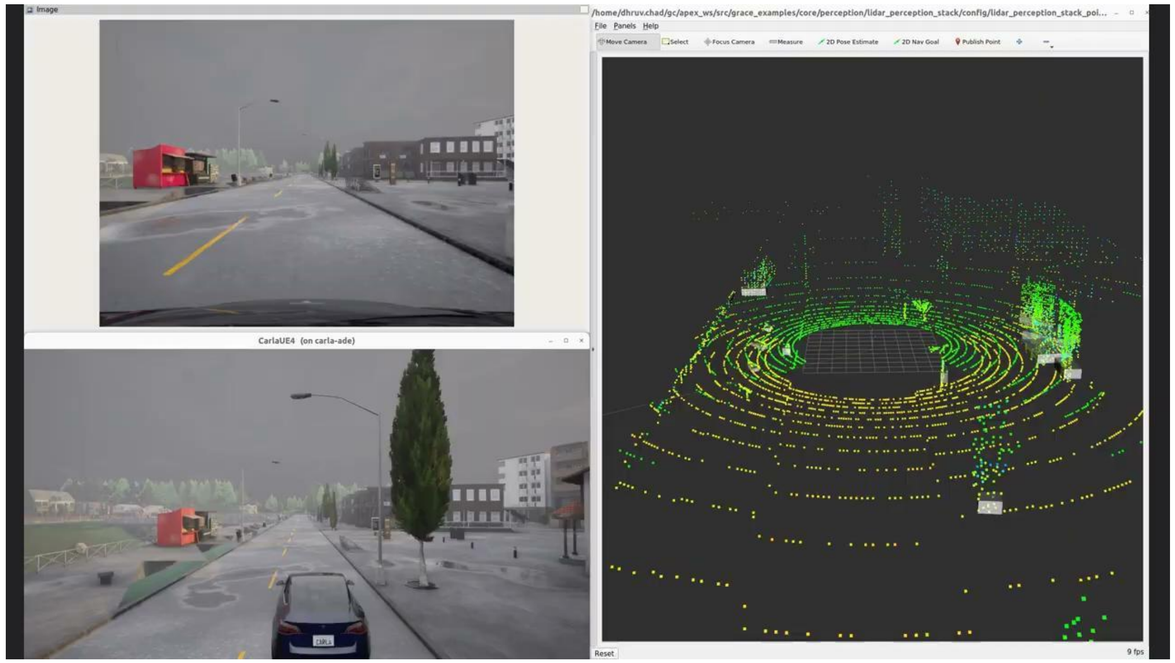
ROS based visualization tool

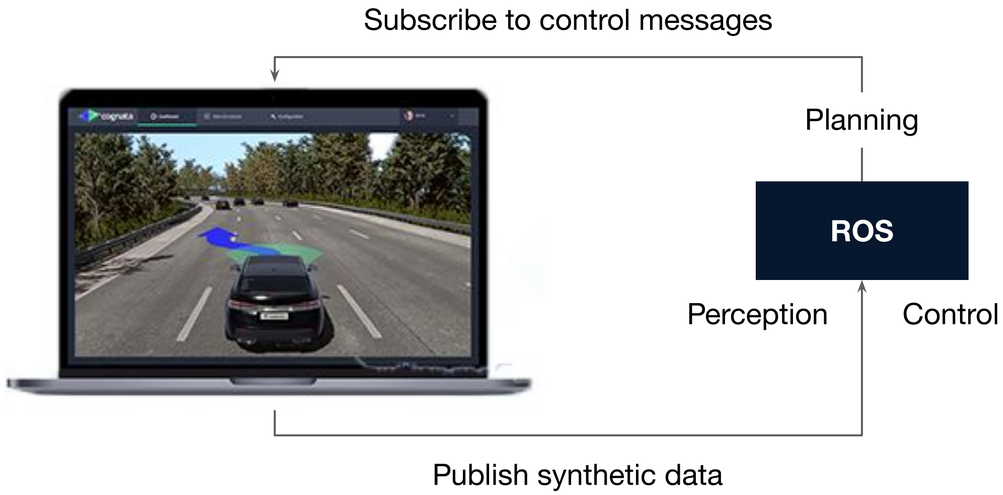


subscribes: /sensor topic

ROS based data processing

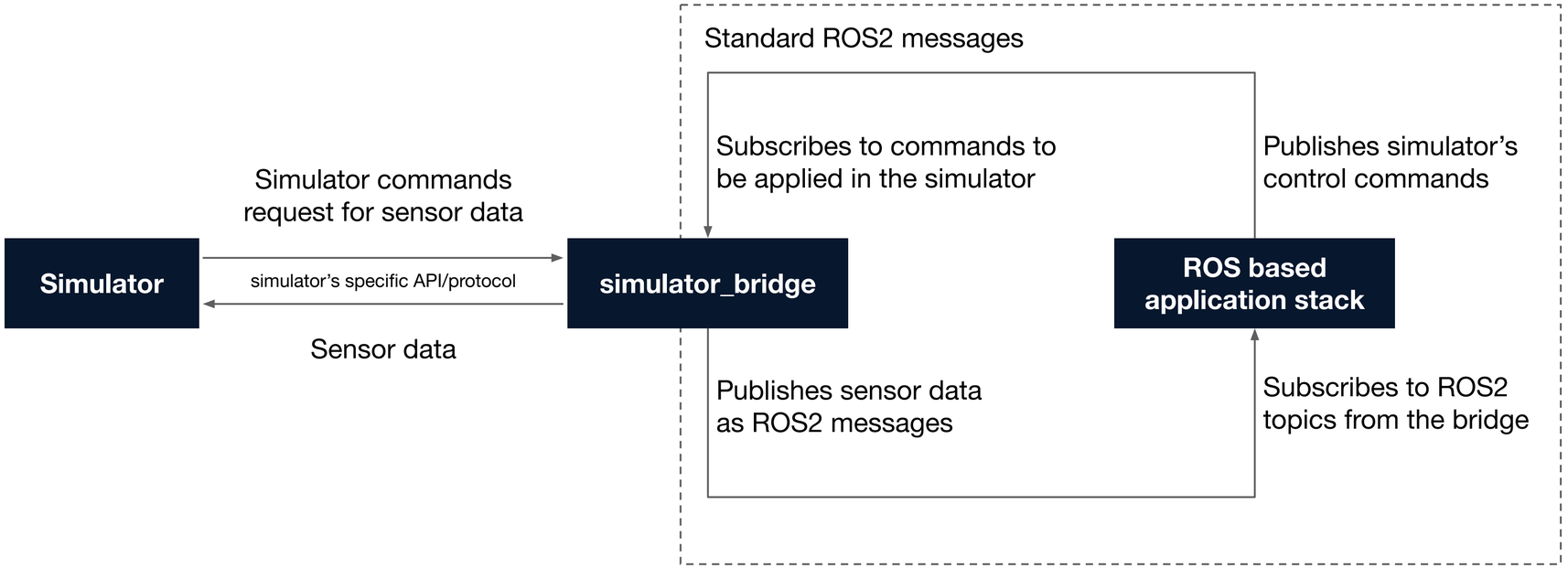
Simulators with ROS





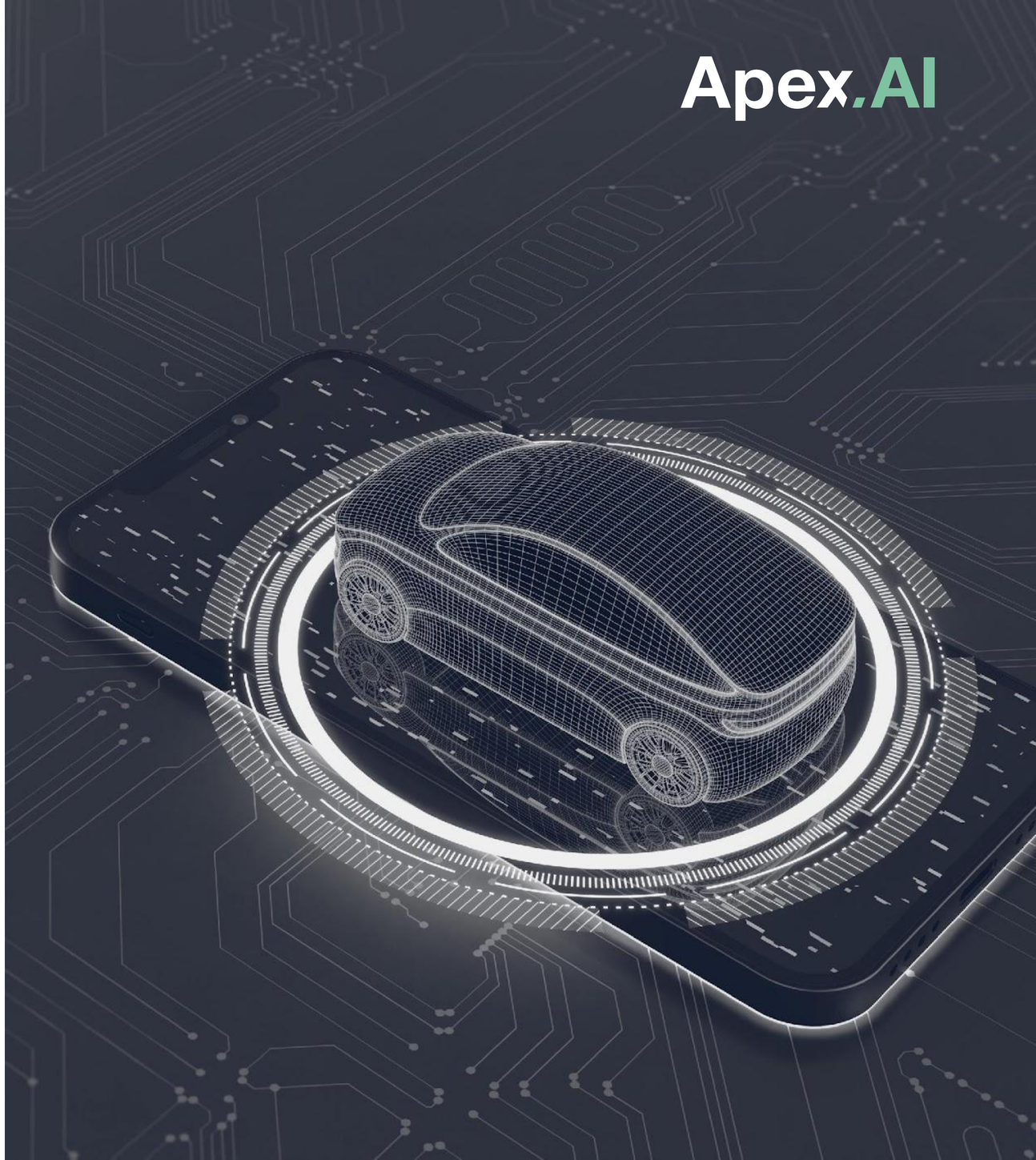
https://www.cognata.com/closed-loop-simulations/

What is a simulator ROS bridge?



- A ROS bridge is a software component that acts as a translator between the simulator and ROS based application stack.
- It is responsible for translating data from virtual sensors in the simulator and control signals from the ROS application.

What is missing?



Determinism

1. For **time determinism**[1] , the results of a calculation are guaranteed to be available before a given deadline.
2. For **data determinism**[1], each time a calculation is started, its results only depend on the input data. For a specific sequence of input data, the results always need to be exactly the same, assuming the same initial internal state.

In this talk, our primary discussion will be towards achieving data determinism via:

- **Deterministic execution**[2] will always run computations in the same order.
- **Deterministic communication**[2] is when, for a message going from process A to process B, communication is guaranteed to be complete always before or always after B executes.

Also, to be data deterministic, any computation should not use system interfaces or other library calls (other than ROS messages) that influence the output of the system. For example, a clock or a random number generator should not influence the output of a computation unless these additional inputs are recorded.

1. AUTOSAR specification
2. 2017 ROSCon talk on "Determinism in robotics software"

Why do we need determinism?

1. Catch early functional bugs before testing the application in the car

2. Run a whole application under constrained resources, where timing can be ignored (allowing more flexibility of where to run the test, instead of always requiring powerful enough hardware to run a large system of nodes)

3. Reproduce a functional (i.e. not related to timing) bug observed in the car

4. Test a scenario provided by a system engineer, e.g. check if a pedestrian is always correctly detected

5. Run an application in a completely event-based approach (SIL). Events can be:
   a. Sending/receiving a message
   b. A timer that expired

6. Test simulation scenarios in the CI

Determinism and challenges with ROS

Deterministic resource usage and runtime is necessary for a safety-critical system

→

- Controlling memory usage
- Controlling task execution

Dynamic memory allocation is not suitable for hard real time systems

→

- Memory usage
  - Allocations during runtime
  - STL constructs with heap usage
    - `std::string`
    - `std::vector`
    - `std::exception`
- Blocking calls, such as `fprintf, fwrite`
- Non-deterministic execution

These changes are needed to bridge this gap to enable real-time behavior

→

- No resource allocation during runtime
- All operations are finite and bounded
- All potentially blocking calls have timeouts
- Execution is deterministic and monitored

Solutions for achieving determinism with ROS

ROS is not actively tested against a real-time operating system (RTOS) on production grade ECUs, thus its real-time suitability is not guaranteed. To the contrary, the ROS implementation does not facilitate deterministic execution, nor expose necessary real-time characteristics required to configure a real-time system.

We can address these shortcomings by:

- Allowing memory allocation only during the initialization phase, and never during the runtime phase
- Providing a memory allocator library for allocation of memory in STL data structures
- Providing a set of APIs to set RTOS-specific properties such as CPU affinity, scheduling priority, and thread distribution
- Providing synchronization primitives that are preemptive, and have scheduling priorities set
- Providing various other constructs to support determinism

Determinism in simulators

- As we understand it, it refers to the ability to produce the same simulation result consistently given the same initial conditions and inputs.
- We can take the example of testing and validation of perception algorithms for autonomous vehicles.
- Perception algorithms process sensor data (from LiDAR, cameras) and perceive the environment around the ego vehicle for detecting pedestrians, traffic lights, etc.
- If we have the same sensors, same positions of ego vehicle and same world objects (pedestrians) etc in the same position and similar appearance across two or more runs of the simulation the expectation will be that the algorithm detects and classifies the pedestrian in both the runs consistently.

Determinism with simulators and algorithms

Simulation run 1:
Camera image: [Image 1]
Detected pedestrians: [Pedestrian A, Pedestrian B]

Simulation run 2:
Camera image: [Image 1]
Detected pedestrians: [Pedestrian A, Pedestrian B]

Simulation run 3:
Camera image: [Image 1]
Detected pedestrians: [Pedestrian A, Pedestrian B]

This determinism can allow repeatability and reproducibility of different AV scenarios in testing.

Determinism in simulators with ROS

1. **Time step sync of simulator:** We can ensure that the simulator itself runs in a controlled fashion. Time step synchronization is one of the common ways to do this. It ensures that all simulation components (sensors, vehicles, control algorithms) advance states based on a agreed upon time step.

2. **Time step sync of ROS based application stack:** We can ensure that the ROS based application stack runs in a deterministic manner with every time step dictated by the ROS bridge

3. **Input consistency:** Inputs to the simulation like vehicle configurations, sensor data and configs, traffic conditions, etc. can be initialized once at the start e.g. defining scenarios and it can always be made sure that the same scenario is run as input across two or more simulator runs.

Determinism in simulators with ROS

Possible configurations based on mode of operation of simulator and ROS based application stack:

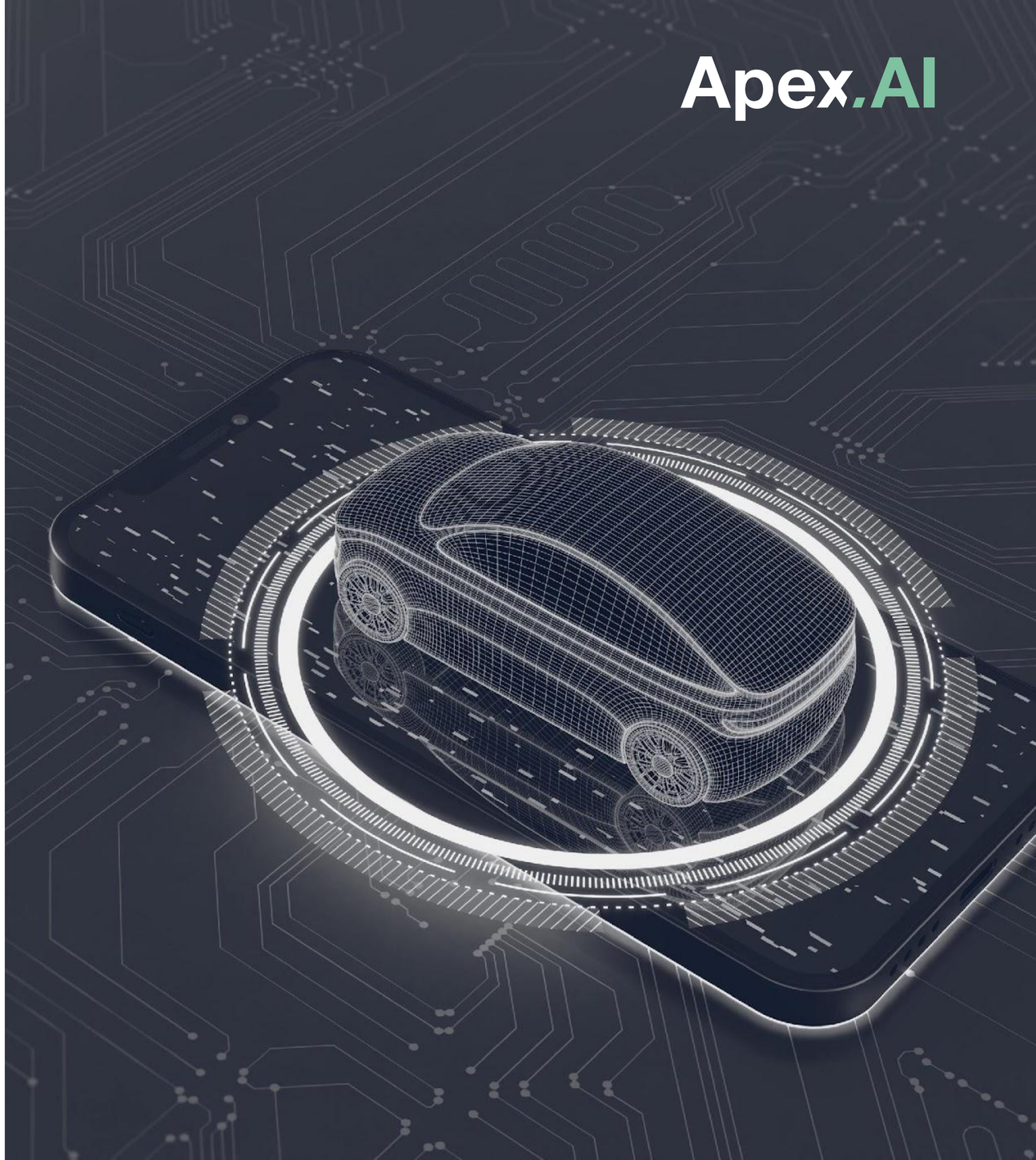
Simulator mode	**ROS application mode**	**Determinism**
Synchronous	As fast as possible	Not fully deterministic, the ROS application will be in free running mode and could have the impact that the order of messages coming from simulator bridge will not be the same sequence which triggers the running ROS application that processes the messages as ROS application is running continuously.
Asynchronous	As fast as possible	Both simulator and ROS application are free running so no determinism will be there between two runs

Determinism in simulators with ROS

Possible configurations based on mode of operation of simulator and ROS based application stack:

Simulator mode	**ROS application mode**	**Determinism**
Synchronous	Controlled by ROS bridge using deterministic execution coordinator feature	Fully deterministic, the ROS application will only run on next message arrival as instructed by the ROS bridge processing simulator input and sending the tick to step simulator forward
Asynchronous	Controlled by ROS bridge using deterministic execution coordinator feature	This mode will not be useful here as simulator is free running so data is always available and we will not gain much by controlling execution of the ROS application

Our proposal



Determinism with Apex.Grace

What is Apex.Grace?

- Apex.Grace is the reliable and real-time SDK for autonomous mobility.
- Apex.Grace provides first class C and C++ APIs for application developers, which are compatible with ROS 2.
- Apex.Grace abstracts complexity through simple to use APIs.
- Apex.Grace Cert is in addition certified to ISO 26262, ASIL as a SEooC.

Specifically for this talk, the focus will be on major C++ based Apex.Grace constructs related to determinism:

1. Executor management framework
2. Fixed-order execution coordinator based on executor

Determinism with Apex.Grace

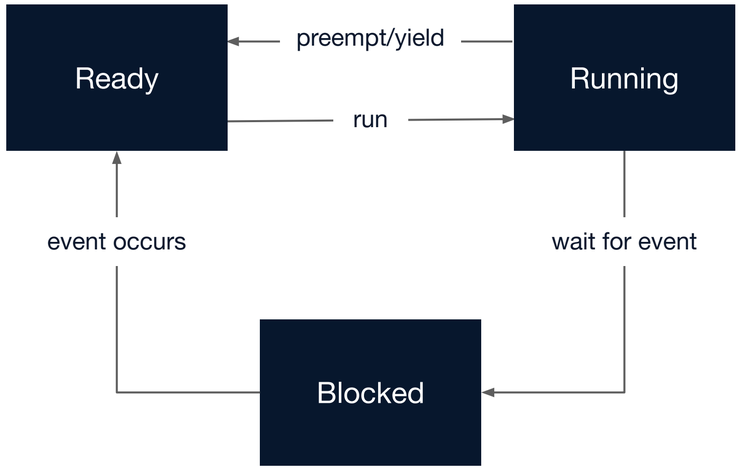
Major components:

1. **Executor management framework**
2. Fixed-order execution coordinator based on executor

Defining an executor

Scheduler

- Typically a part of the operating system
- Assigns the work to the CPU
- Decides, according to thread priorities and scheduler configuration, which of the ready threads run and which are queued up
- Main focus is on transitions between Ready and Running
- In Apex.Grace, the scheduler of the underlying operating system is configured and used



Executor

- Typically part of a framework like ROS 2 or Apex.Grace
- Defines the work to do for the scheduler
- Controls which events unblock a thread, and which tasks the thread shall execute once getting unblocked. After all tasks were executed, the thread will be blocked again
- Main focus is on transitions between Blocked and Ready, Running, and Blocked
- The executor reacts on events like timers, incoming messages, and interrupts

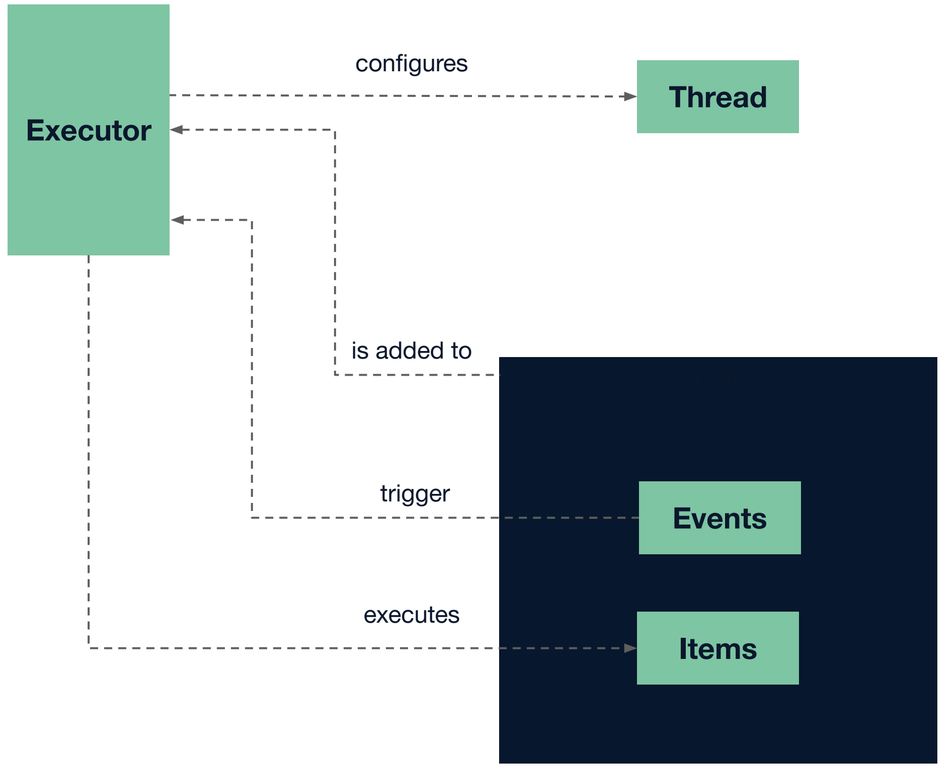
Entities in the executor context

Executable item
- Implements run-to-completion functionality
- Provides an entry point for execution (execute_impl())

Triggering event
- Indicates that there is work to be done
- Can be timers, subscribers, clients, services, or file descriptors

Execution task combines items and their triggering events
- Can have a single item, a chain of items, or a graph of items
- Can have one or many triggering events (like on message,
- or on timer)

Executor executes the items of a task on triggering events
- Controls the execution of one or many tasks
- Uses one or many threads to execute the tasks



Look and feel of the Apex.Grace executor

```
// Simple node with a single entry point
class my_simple_node : public apex::executor::apex_node_base  {

  void execute_impl() override { // Process the messages } ①

  subscription_list get_triggering_subscriptions_impl() override {return this->triggering_sub;} ②
};

// Executor setup (main)
auto node = std::make_shared<my_simple_node>("my_node");
auto executor = apex::executor::executor_factory::create(); ③
executor->add(node); ④
executor->run(); ⑤
```

1. Logic to process messages. Called by the executor when a triggering event happens
2. Defines that the subscriber with the name *triggering_sub* is triggering
3. Creates an executor
4. Interface to add a data-driven executable item (*my_simple_node* is an executable item!)
5. Starts the executor which will now react on triggering events (messages received on the topic of *triggering_sub*)

Example of an Apex.Grace node implementation

```
void execute_impl() override {

        auto non_triggering_msgs {non_triggering_sub->take()}; ①
        auto triggering_msgs {triggering_sub->take()};②

        if (!non_triggering_msgs.empty() && !triggering_msgs.empty()) {
                auto loaned_message{m_publisher->borrow_loaned_message()};

                my_algo(triggering_msgs.back(), non_triggerings_msgs.back(), loaned_message); ③

      pub.publish(std::move(loaned_message)); ④
   }
}
```

1. Take available messages on the non-triggering topic
2. Take available messages on the triggering topic
3. Process latest messages from subscribers and fill the loaned message
4. Publish output message

Determinism with Apex.Grace

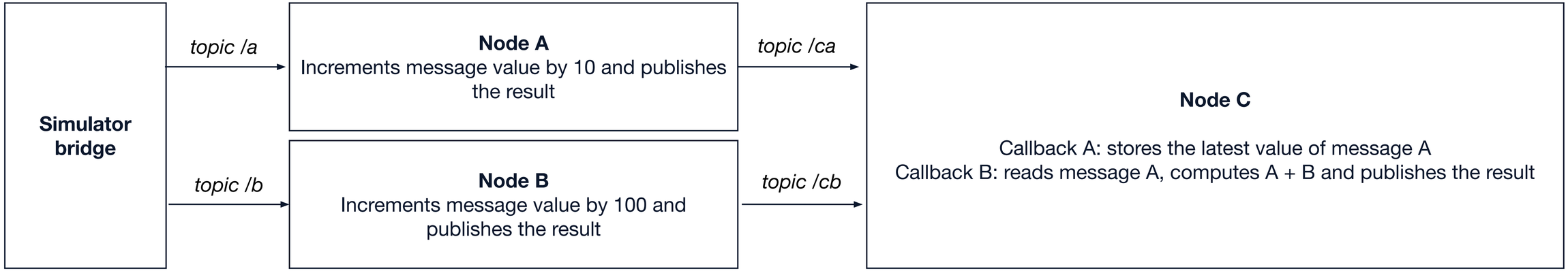
Major components:

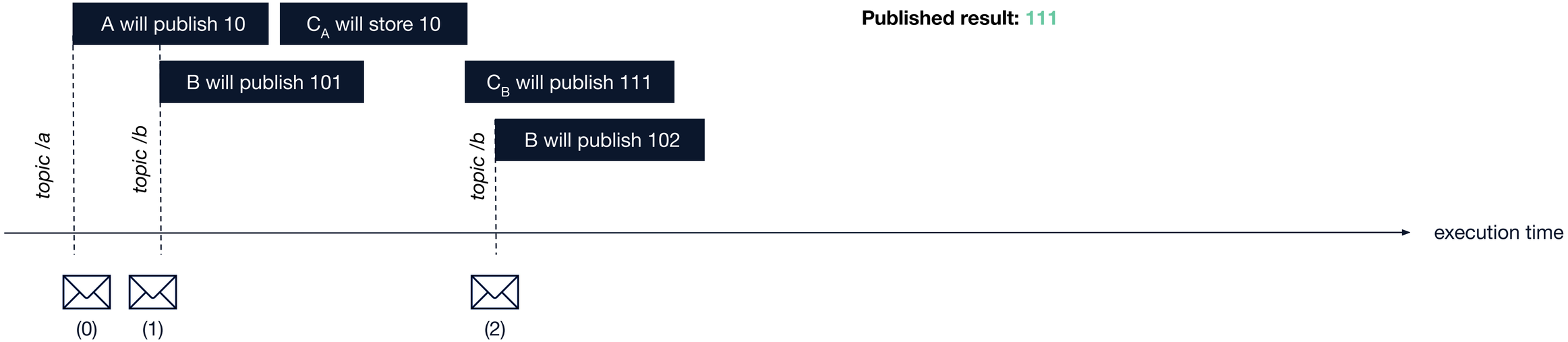
1. Executor management framework
2. **Fixed-order execution coordinator based on executor**

Different types of execution of the ROS application stack

	Uncontrolled-order execution	**Fixed-order execution**
Behavior	Normal ROS execution with live, simulated or recorded data via ROS messages	Deterministically stimulates a system of nodes that can be both data- and time-driven from the simulator. The execution order is always the same
Limitations	Execution order can vary which can be a source of flakiness in integration tests	**Execution order may not be the same as the order in the live system.** The nodes must conform to the definition of a reproducible node to ensure it is possible to produce the same results
Use case	Run a rosbag (recorded data) to observe a behavior, test sequential nodes	Reproducible test of the functional behavior of an application in a given execution order

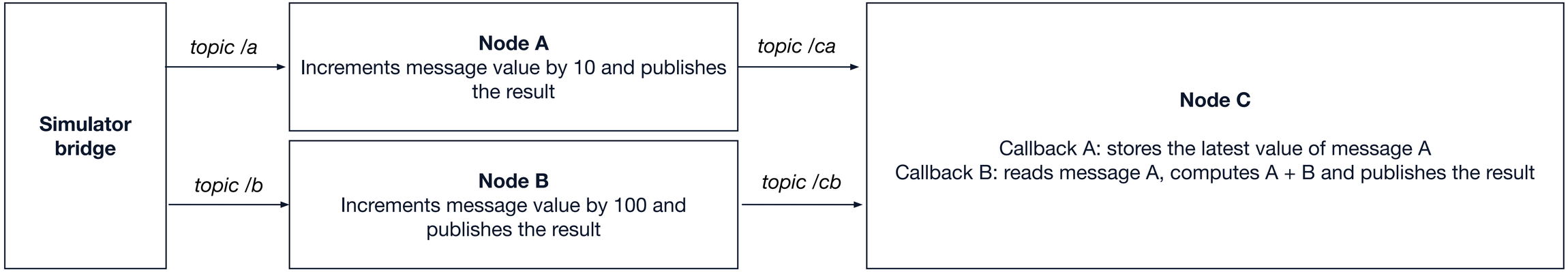
Uncontrolled-order execution (first run)

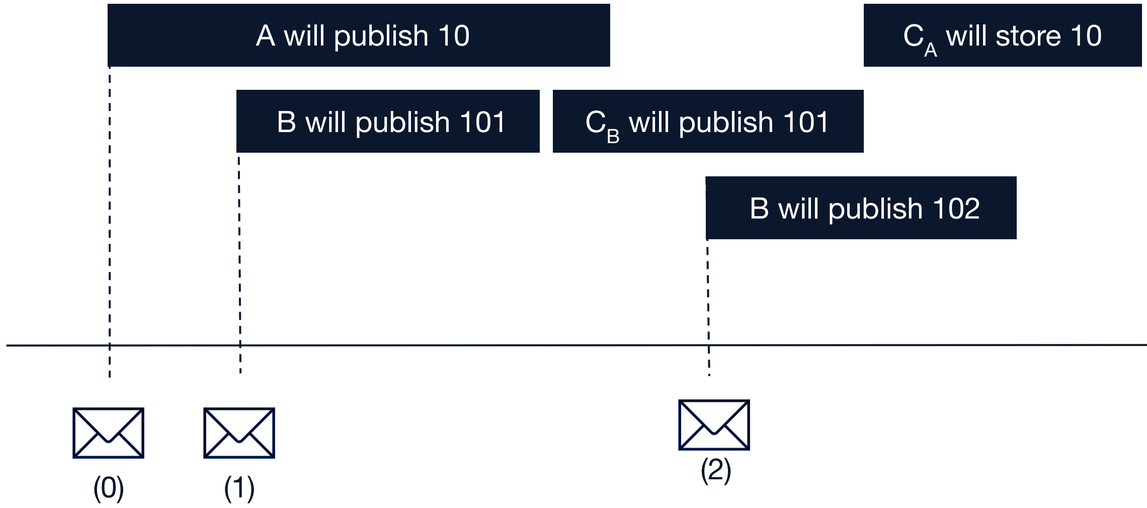




Published result: 111

Uncontrolled-order execution (second run)

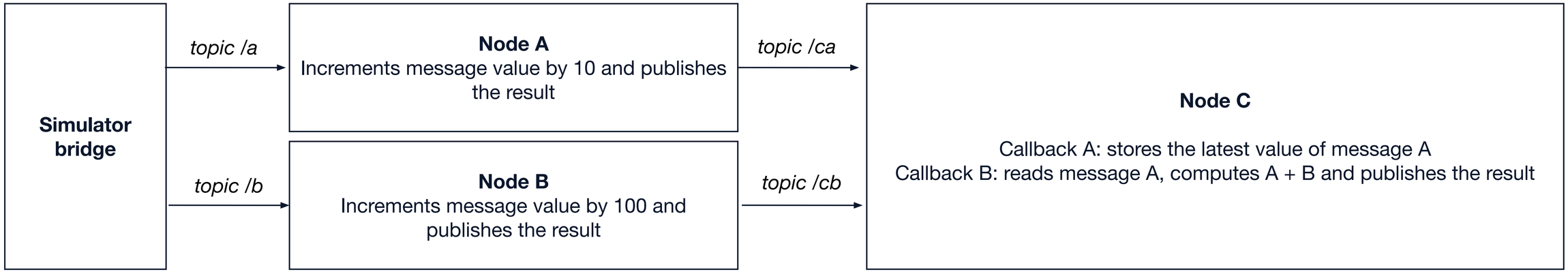


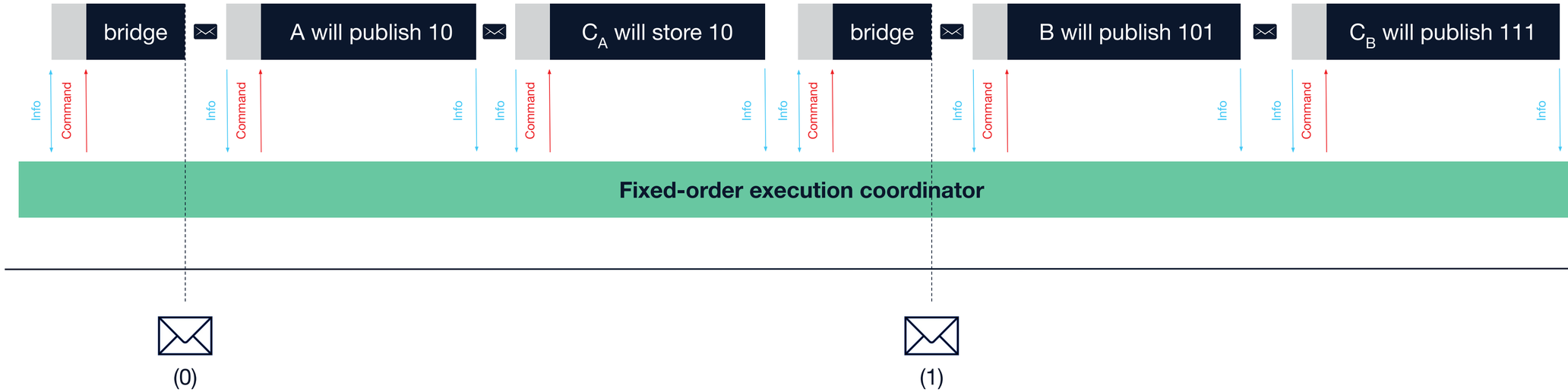


Published result: 101

- Message A not available when callback B ran
- Order of execution changed and influenced the output of the system: non deterministic => flaky test!

Fixed-order execution





Published result: 111

- Callbacks are executed in a specific order independently of execution time

Scheduler with fixed-order execution

In a concurrent system, it is not trivial to control the execution order of the threads within a single process. Execution of the same application can take varied times across multiple runs. To improve determinism, there can be two possibilities:

1. "Configure the scheduler or write your own" to make sure execution order does not change. This is not a friendly approach for cross platform runs.

2. "Write your own execution management framework" that is not dependent on how threads are scheduled.

Our proposal works with the second approach, where we have implemented our own fixed-order execution framework in C++. There is a coordinator entity that controls the execution order and makes sure that only a single “user” thread will be in executing state at any given point in time.

Fixed-order execution

To execute an application deterministically, all executors are replaced with "fixed-order executor", which coordinate among each other through a central execution coordinator.

- A timer queue proxy and messenger objects should be created to enable the communication with the fixed-order execution coordinator.
- The messenger object installs a hook for Publisher<T> from rclcpp in its constructor and removes the hook in its destructor. This hook messages the coordinator when anything is published and is crucial for the correct functioning of the fixed-order execution algorithm.
- A fixed-order executor provides the exact same API as the live executor, so all that is needed is to replace the call to the executor factory:

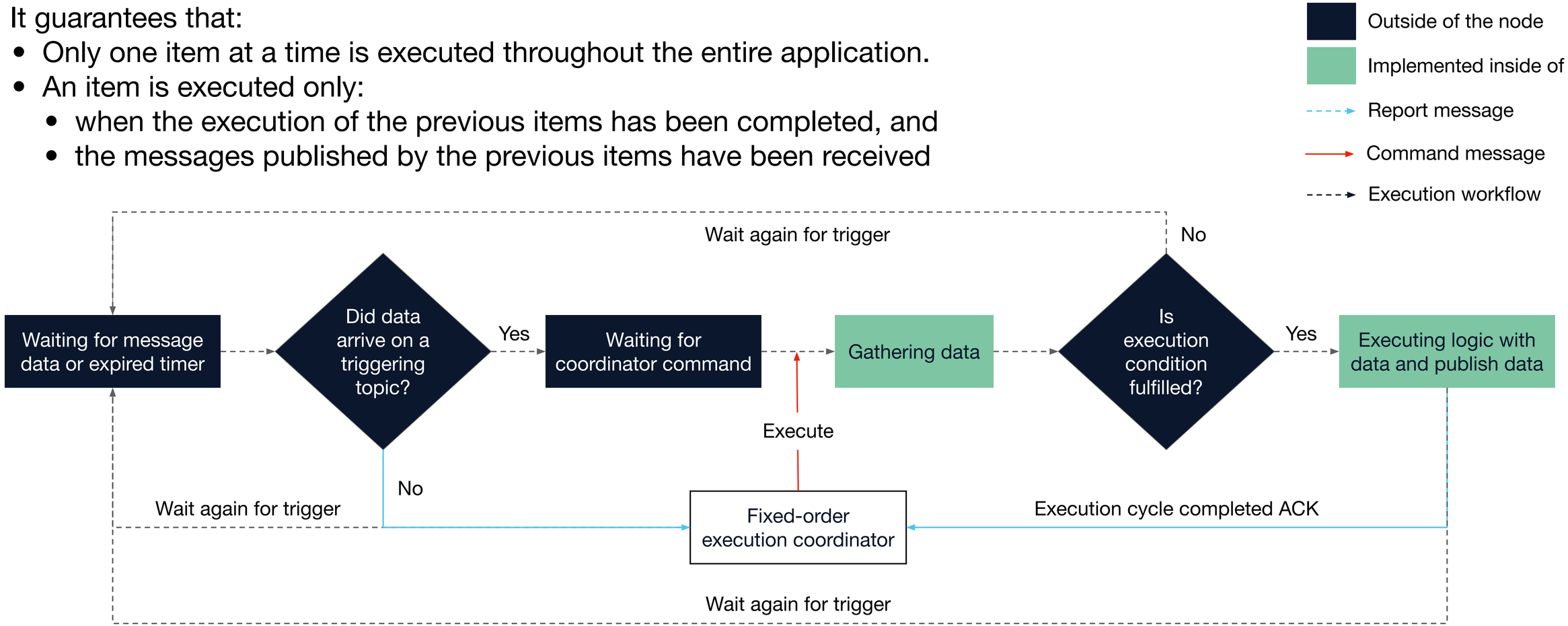
```
//Fixed order Executor setup (main)
   apex::determinism::messenger messenger (“apex_coordinator_messenger”);
   apex::determinism::timer_queue_proxy timer_queue_proxy(“apex_determinism_timer_queue”,5s);
   const auto exec = apex::executor::executor_factory::create_for_fixed_order_execution(messenger,
timer_queue_proxy);
```

Fixed-order execution coordinator

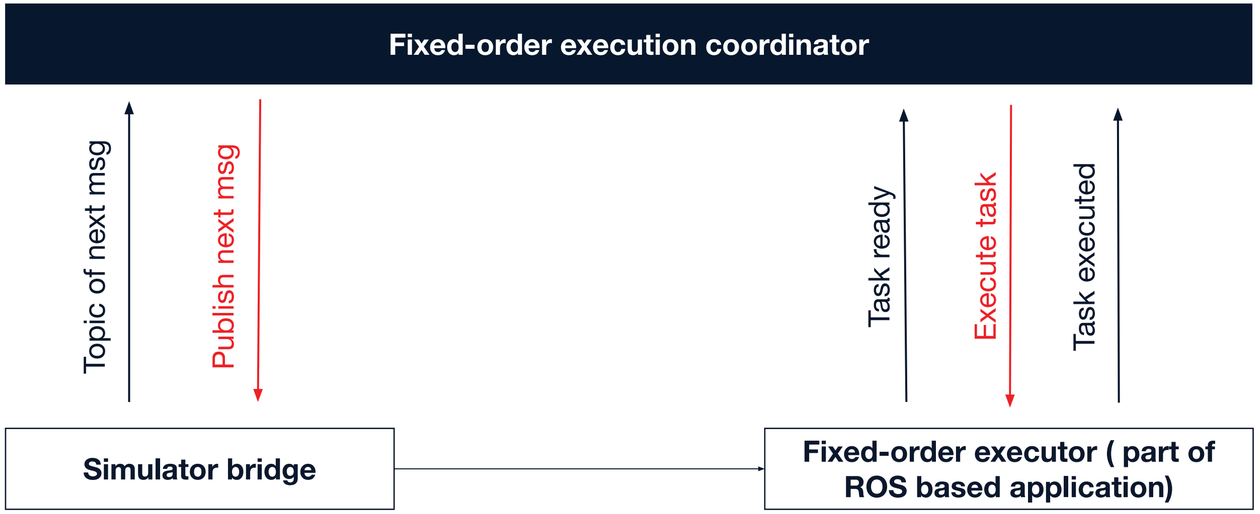
When an application is run with the fixed-order executor, a central fixed-order execution coordinator process controls the order of execution in the system's executors. There is only one system-wide coordinator no matter how many processes are used by the application.

It guarantees that:

- Only one item at a time is executed throughout the entire application.
- An item is executed only:
  - when the execution of the previous items has been completed, and
  - the messages published by the previous items have been received

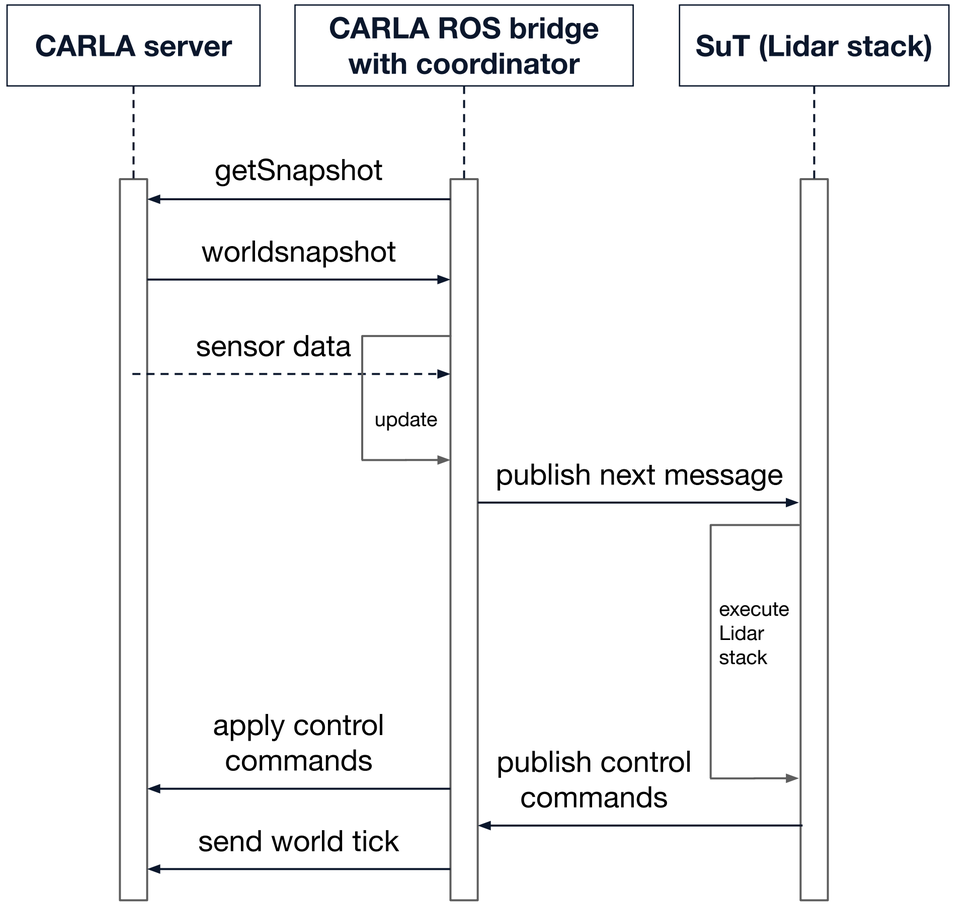


Fixed-order execution with simulator



A case study — Integration with CARLA simulator

The diagram below illustrates the approach for adding the **fixed-order execution coordinator** to the CARLA C++ ROS bridge to enable deterministic execution when the simulator is configured to run in **synchronous mode**:




```
while (true) {
  get_snapshot();
  for (sensor : all_sensors) {
    sensor.update(); // storing all messages that would be sent in a
vector instead of sending directly
  }
  for (stored_message: messages) {
deterministic_execution_coordinator.send_message(stored_message);
  }
  check_that_control_command_arrived();
 apply_control_commands();
  next_simulation_tick();
}
```

Questions?

Apex.AI

Thank you!

divya.aggarwal@apex.ai

CARLA License

MIT License

Copyright (c) 2017 Computer Vision Center (CVC) at the Universitat Autonoma de Barcelona (UAB).

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Referenced from : https://github.com/carla-simulator/carla/blob/master/LICENSE