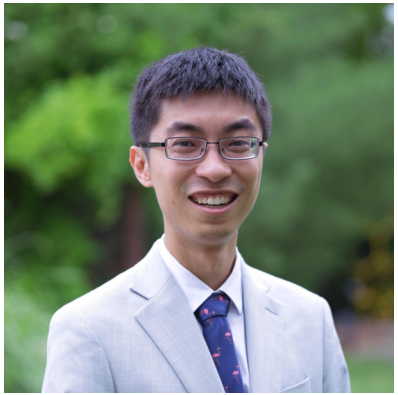
Real-time ROS 2 applications made easy with cactus-rt

Shuhao Wu

Oct 22 2024, ROScon 2024 Odense

Who am I?

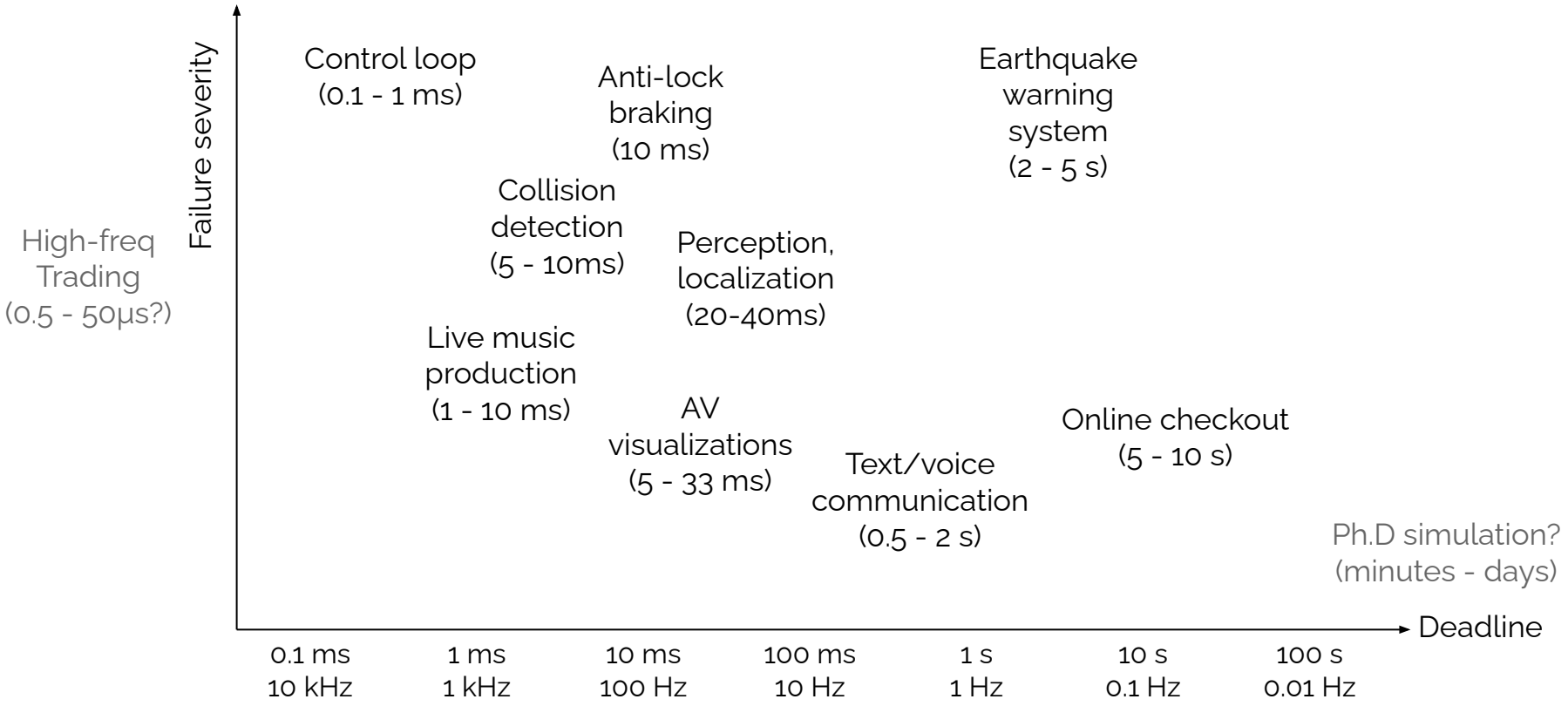
- Shuhao Wu
  - https://shuhaowu.com
- Active in the ROS real-time community
  - Maintain the real-time ROS Raspberry Pi image
  - Hosted real-time workshop last year
- Currently: Senior software engineer @ NVIDIA
  - Cloud and real-time visualizations for autonomous vehicles
- Previously: Staff production engineer @ Shopify
  - Large-scale cloud deployments, distributed systems, databases, performance, etc.
- Masters of mechanical engineering
  - Fluid dynamics, robotics



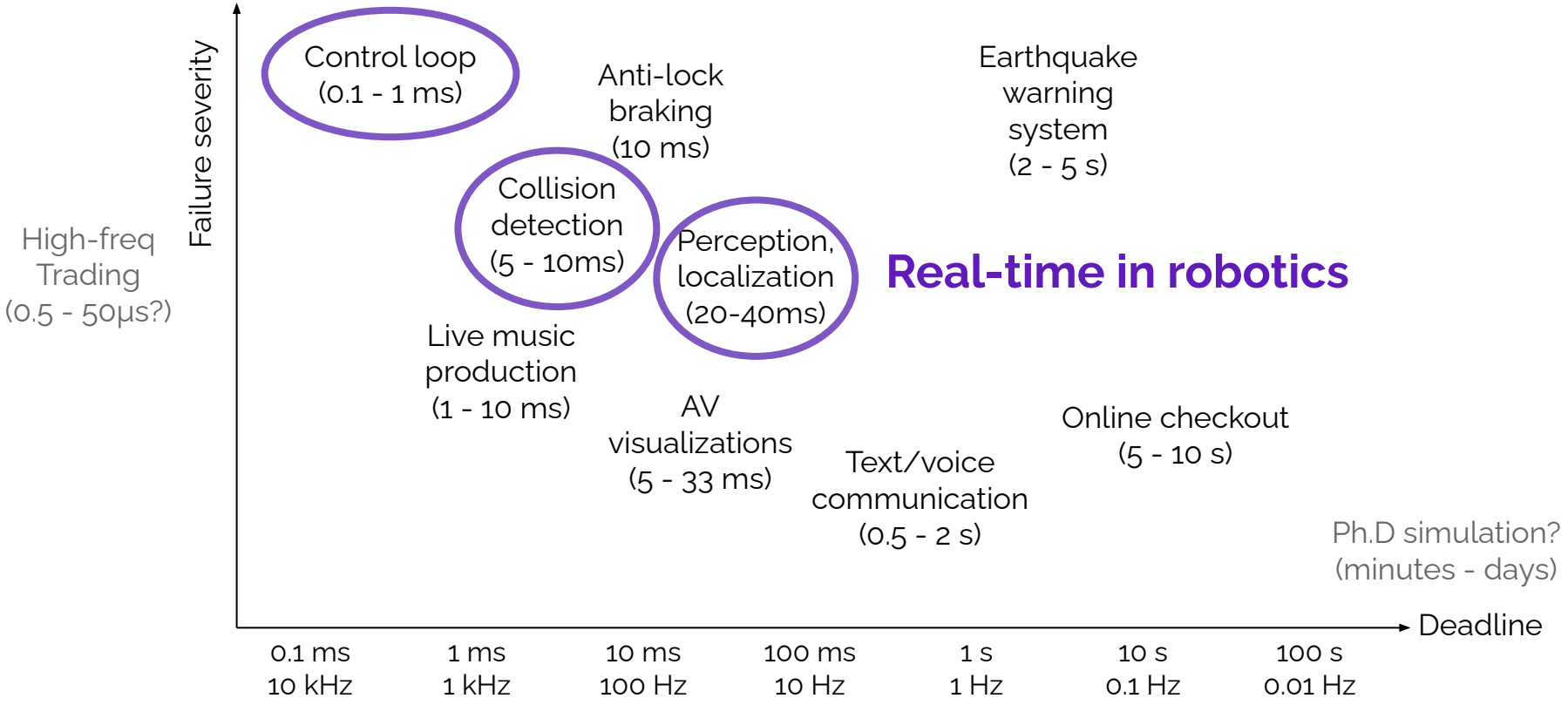
Disclaimer

Opinions are my own, not representative of employer

What is “real-time” programming?



What is “real-time” programming?

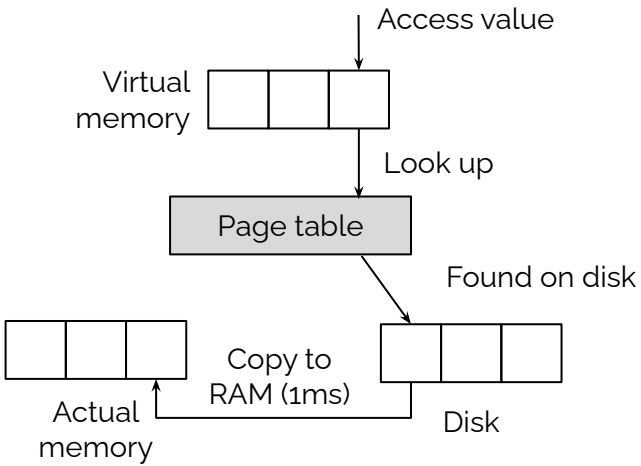


Intro to real-time programming 101

- Real-time OS + scheduler

Intro to real-time programming 101

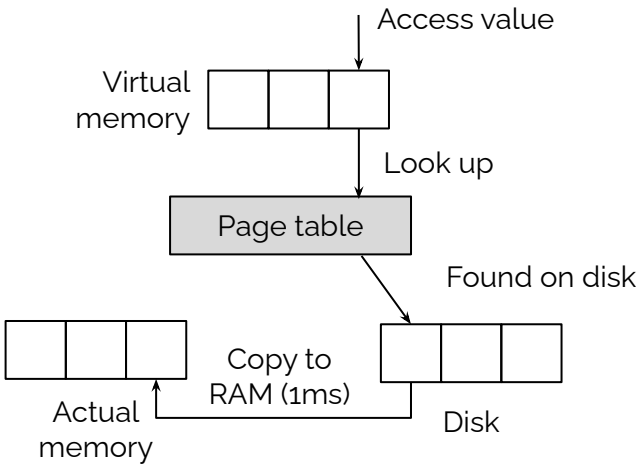
- Real-time OS + scheduler
- Careful memory management
  - No allocation
  - No page faults

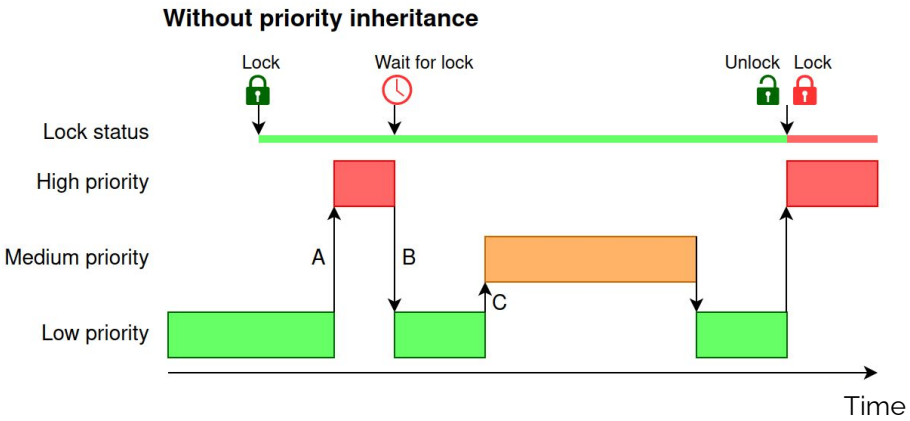


Intro to real-time programming 101

- Real-time OS + scheduler
- Careful memory management
  - No allocation
  - No page faults

- Careful cross-thread communication
  - Priority-inheritance mutex
  - Lockless programming





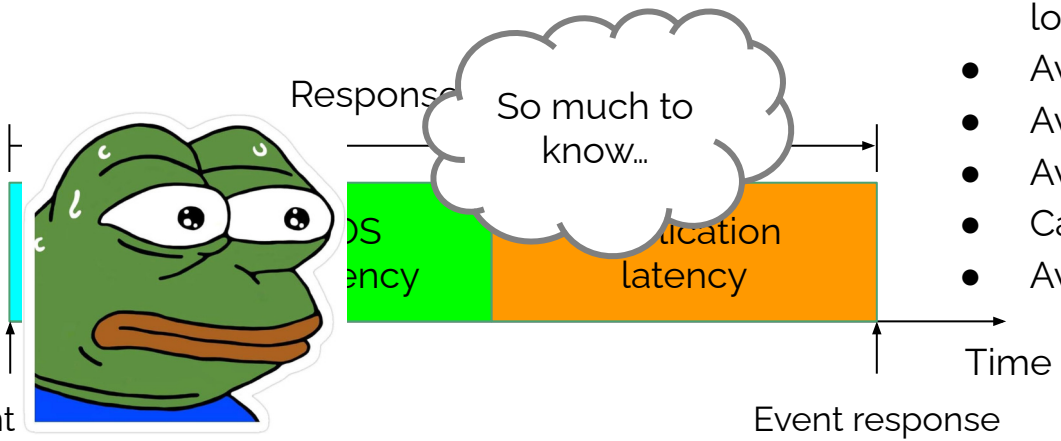
The rest of real-time programming 101

Hardware + OS latency

- Run a real-time OS like Linux + PREEMPT_RT
- Measure + tune hardware and OS latency
- Use a real-time task scheduler like SCHED_FIFO and SCHED_DEADLINE
- Measure and tune IO access latency

Application + library latency

- Lock memory to prevent page faults
- Use priority-inheritance locks or lockless programming for communications
- Use the right clock for looping
- Use asynchronous, allocation-free, and lockless logging
- Avoid dynamically allocate memory
- Avoid unbounded syscalls
- Avoid exceptions
- Careful of amortized O(1) algorithms
- Avoid libraries that does the above



Real-time programming with ROS 2

- Two types: 1000 Hz real-time vs real-time messaging
  - 1000 Hz real-time: I want a 1000 Hz timer
  - Real-time messaging: I want to process my message within *N* ms end-to-end
- ROS 2 executors are not generally designed for either
- Real-time messaging in ROS requires executor cooperation
  - See ROScon 2023 RT workshop session 3 and 4
- 1000 Hz real-time can be implemented alongside with ROS 2
  - ros2_control adopts this architecture

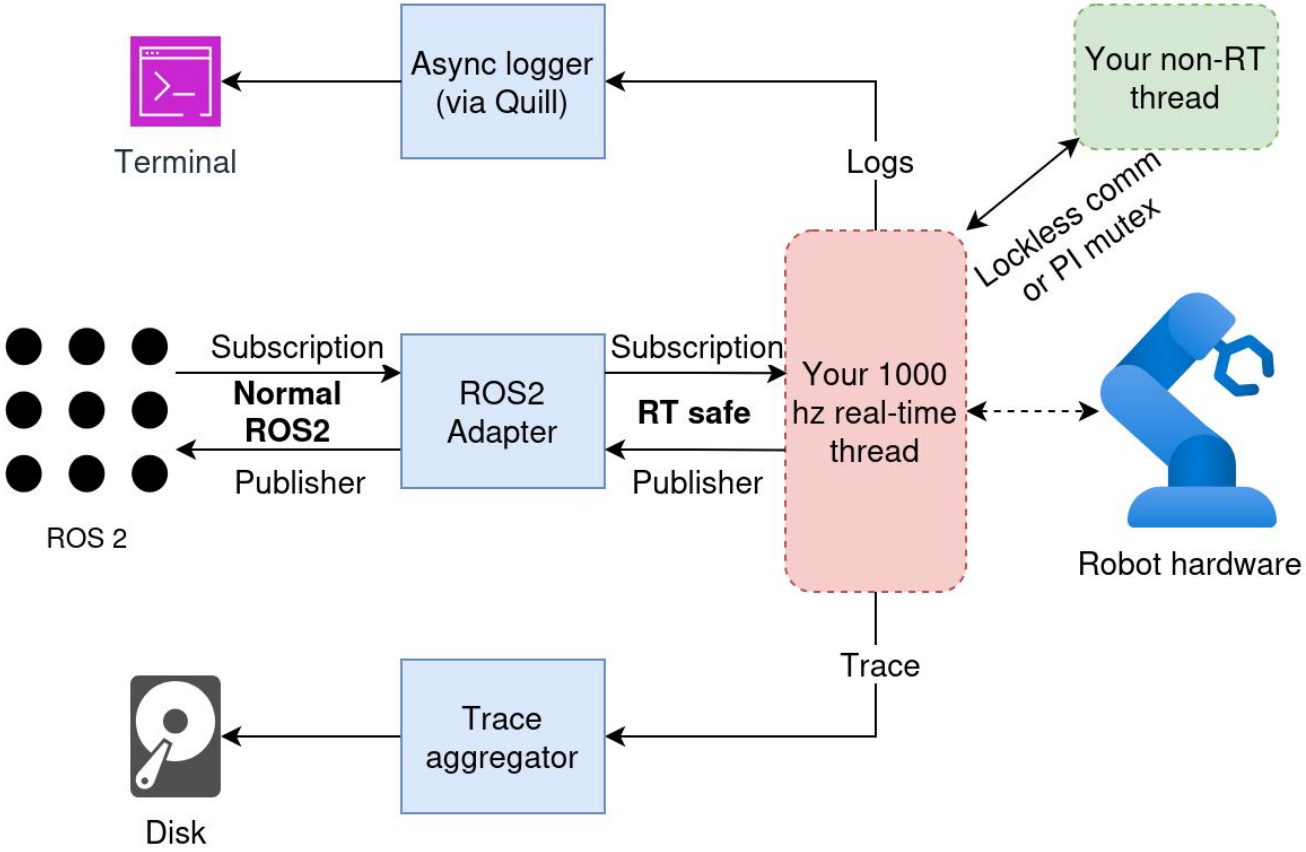
Ideal 1000 Hz ROS 2 dev environment

- “I want to just write the code that executes every 1 ms”
  - Delegate away the interaction with OS APIs to setup real-time
  - Publish to and receive from ROS 2 topics in real-time
  - Communicate between threads in a real-time-safe way
  - Print and log normally
  - Have tracing to verify maximum latency

Ideal 1000 Hz ROS 2 dev environment

- “I want to just write the code that executes every 1 ms”
  - Delegate away the interaction with OS APIs to setup real-time
  - Publish to and receive from ROS 2 topics in real-time
  - Communicate between threads in a real-time-safe way
  - Print and log normally
  - Have tracing to verify maximum latency
- Why can’t we have this in the 2020s?
  - Also where is my flying car?!

cactus-rt: A real-time framework on Linux



A simple 1000 Hz thread and app

```cpp
class RT1000 : public cactus_rt::CyclicThread {
 public:
  RT1000() : CyclicThread("RT1000", MakeConfig()) {
    Logger()->set_log_level(quill::LogLevel::Debug);
  }

 protected:
  LoopControl Loop(int64_t /*elapsed_ns*/) noexcept final {
    // Code here runs every 1ms
    return LoopControl::Continue;
  }

 private:
  static cactus_rt::CyclicThreadConfig MakeConfig() {
    cactus_rt::CyclicThreadConfig config;
    config.period_ns = 1'000'000;  // 1ms period, 1000 Hz
    config.SetFifoScheduler(80);   // SCHED_FIFO, rtprio = 80
    return config;
  }
};
```

```cpp
int main() {
  cactus_rt::App app;

  auto thread = app.CreateThread<RT1000>();

  app.Start();
  app.Join();

  return 0;
}
```

Full example source code

Asynchronous logging

- Logging is critical to debug logic issues
- Normal loggers (std::println, spdlog, rclcpp log) not safe for real-time
- Real-time-safe logger must:
  - Not call `write` synchronously
  - Not use locks
  - Not format string in real-time thread
- cactus-rt integrates with Quill
  - Automatically starts background logging thread
  - Each `Thread`/`CyclicThread` has its own logger via `Logger()`

```
LoopControl Loop(int64_t elapsed_ns) noexcept final {
  LOG_DEBUG(Logger(), "This logs every iteration. elapsed={}ns", elapsed_ns);
  LOG_DEBUG_LIMIT(1s, Logger(), "This logs every 1s");
  return LoopControl::Continue;
}
```

A simple 1000 Hz thread interacting with ROS

- First create the ROS 2 pub/sub via the `ros2_adapter_`
  - In `Thread::InitializeForRos2`

```
void InitializeForRos2(cactus_rt::ros2::Ros2Adapter& ros2_adapter) final {
  cmd_vel_sub_ = ros2_adapter.CreateSubscriptionForLatestValue<Velocity2D, geometry_msgs::msg::Twist>(
    "/cmd_vel", rclcpp::QoS(10)
  );

  feedback_pub_ = ros2_adapter.CreatePublisher<Velocity2D, geometry_msgs::msg::Twist>(
    "/cmd_vel_achieved", rclcpp::QoS(10)
  );
}
```

Full example source code

A simple 1000 Hz thread interacting with ROS

- Read subscription or write to publisher in `Loop` function
  - `StampedValue<T>` is two fields
    - `.id` is the sequence number
    - `.value` is `T`
- `T` is not a ROS type, but a real-time-safe type
    - Necessary as some ROS types allocates a `std::vector` (e.g. `JointTrajectory`)

```
struct Velocity2D {
  double vx;
  double vy;
  double w;
};
```

```
LoopControl Loop(int64_t /*elapsed_ns*/) noexcept final {
  StampedValue<Velocity2D> msg = cmd_vel_sub_->ReadLatest();

  Velocity2D achieved_vel = Drive(msg.value.vx, msg.value.vy, msg.value.w);
  feedback_pub_->Publish(achieved_vel);

  LOG_DEBUG(Logger(), "Received id = {}; vx, vy, w = {}, {}, {}; Achieved vx,

  return LoopControl::Continue;
}
```

Full example source code

A simple 1000 Hz thread interacting with ROS

- Type conversion via ROS 2 type adapter (REP 2007)
- Possible to use ROS 2 type directly, but needs an extra template argument

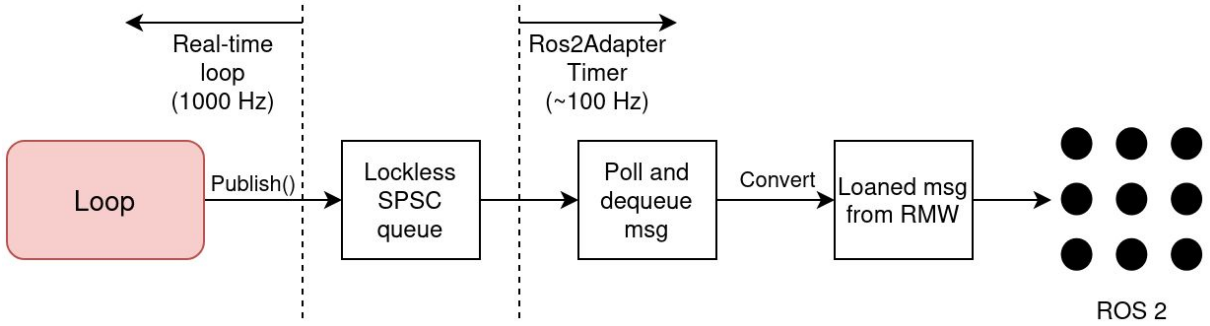
```cpp
template <>
struct rclcpp::TypeAdapter<Velocity2D, geometry_msgs::msg::Twist> {
  using is_specialized = std::true_type;
  using custom_type = Velocity2D;
  using ros_message_type = geometry_msgs::msg::Twist;

  static void convert_to_ros_message(const custom_type& source, ros_message_type& destination) {
    destination.linear.x = source.vx;
    destination.linear.y = source.vy;
    destination.angular.z = source.w;
  }

  static void convert_to_custom(const ros_message_type& source, custom_type& destination) {
    destination.vx = source.linear.x;
    destination.vy = source.linear.y;
    destination.w = source.angular.z;
  }
};
```

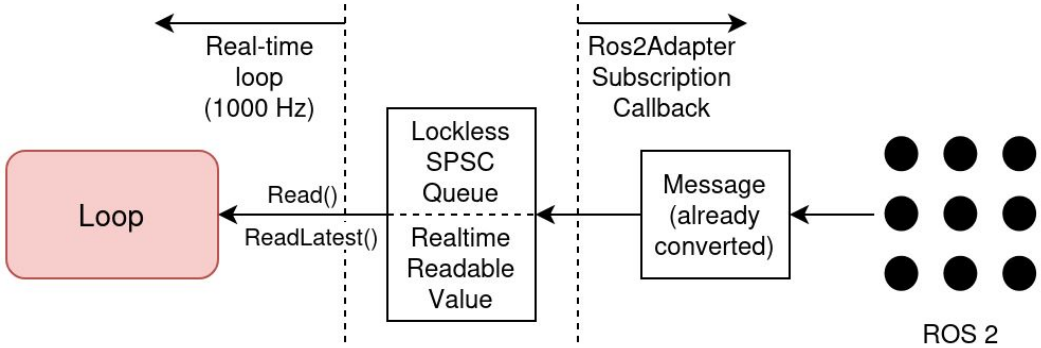
Full example source code

How it's made: publisher



- Using [`moodycamel::readerwriterqueue`]
- Timer polls for new messages and forwards to ROS (with type conversion)
- Data published are batched and slightly delayed due to async architecture

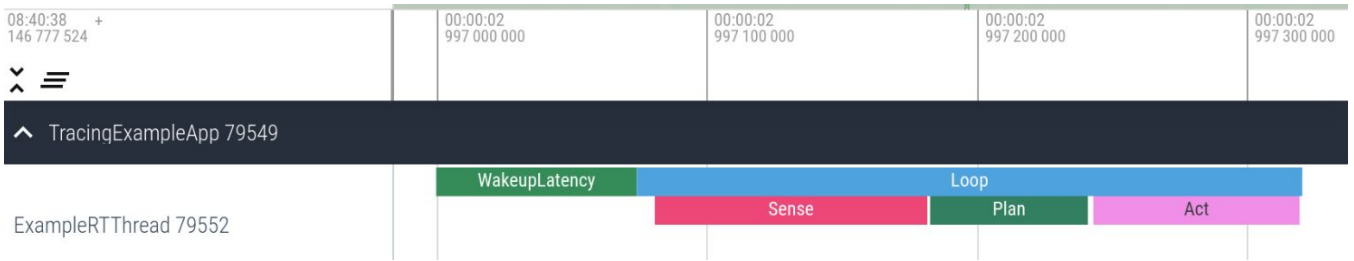
How it's made: subscriber



- Two variants: all values and latest value
- Detect dropped messages via sequence numbers
- Subscription data not received via callbacks, but via explicit polling

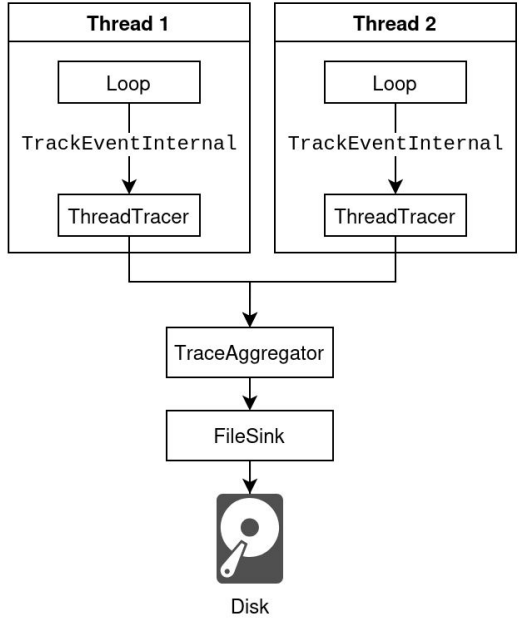
Validating timing via RT-safe tracing

- Tracing is critical to debug timing issues
  - Timing issues critical to real-time software
- Most tracing systems are not real-time safe
  - Locks, memory allocation (resizable buffers), etc
- cactus-rt provides a built-in real-time-safe tracing system



Tracing with cactus-rt

- RAII-based API to mark beginning and end of a span
- Internally writes trace event to a lockless queue
  - Overhead: 0.02 - 2 µs per call on the RT thread
- Background trace aggregator writes out Perfetto-compatible data



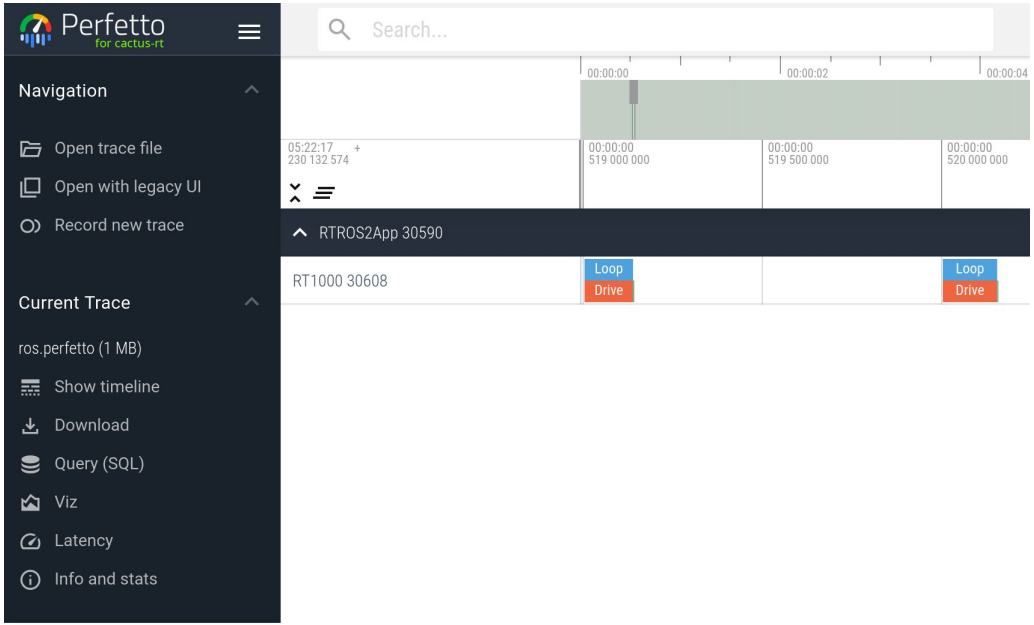

```
LoopControl Loop(int64_t /*elapsed_ns*/) noexcept final {
  StampedValue<Velocity2D> msg = cmd_vel_sub_->ReadLatest();

  Velocity2D achieved_vel;
  {
    auto span = Tracer().WithSpan("Drive");
    achieved_vel = Drive(msg.value.vx, msg.value.vy, msg.value.w);
  }

  {
    auto span = Tracer().WithSpan("Publish");
    feedback_pub_->Publish(achieved_vel);
  }
```

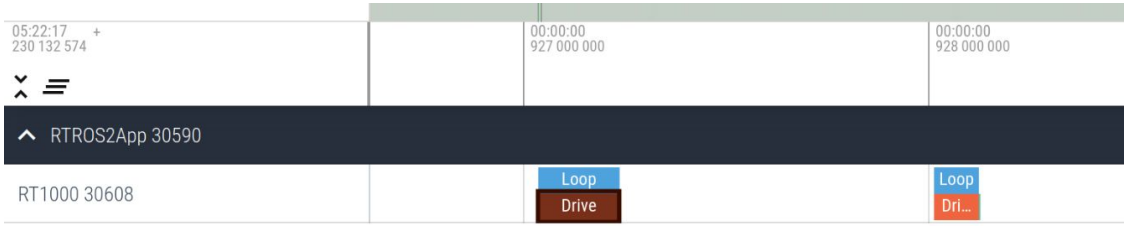
Visualizing traces

- Visualize in Perfetto's UI:
  - Upstream visualizer: https://ui.perfetto.dev
  - cactus-rt-specific fork https://cactusdynamics.github.io/perfetto/



Visualizing traces

- Visualize in Perfetto's UI:
  - **Timeline view**
  - **Table view with sorting**



Current Selection | Table slice (4488)

Table slice — Showing rows 1-100 of 4488 — Show debug track | Copy SQL query

name = 'Drive'

ID	Timestamp	Duration	Thread duration	Category	Name	Thread name	tid	Process name
2782	00:00:00.927 035 507	200us 976ns	NULL	NULL	Drive	RT1000	30608	RTROS2App
8377	00:00:02.792 019 707	199us 800ns	NULL	NULL	Drive	RT1000	30608	RTROS2App
1834	00:00:00.611 005 084	199us 560ns	NULL	NULL	Drive	RT1000	30608	RTROS2App
6883	00:00:02.294 009 898	199us 461ns	NULL	NULL	Drive	RT1000	30608	RTROS2App
10642	00:00:03.547 003 001	199us 352ns	NULL	NULL	Drive	RT1000	30608	RTROS2App
5788	00:00:01.929 007 729	199us 304ns	NULL	NULL	Drive	RT1000	30608	RTROS2App

Visualizing traces

- Perfetto's visualizer is one of the best-in-class
  - Timeline view
  - Table view with sorting
  - **SQL processing**

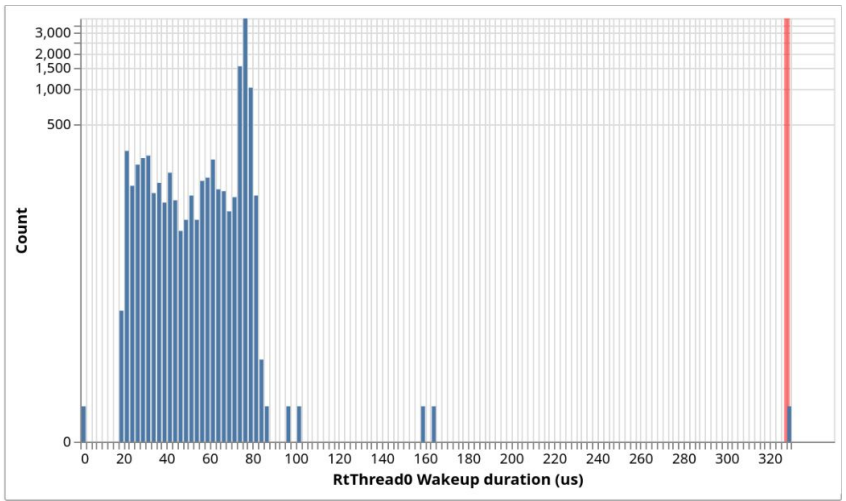
```
1  SELECT IMPORT("experimental.slices");
2  select id, dur / 1000 AS dur_ms, name, category, thread_name
3  from experimental_slice_with_thread_and_process_info
4  where
5    name = "Wakeup" AND dur > 1000
6  order by dur desc limit 10
```

Query result (10 rows) - 38ms SELECT IMPORT("exp… Copy query Copy result (.tsv) Close

id	dur_ms	name	category	thread_name
31308	1133	Wakeup	cactusrt	RtThread1
31315	400	Wakeup	cactusrt	RtThread1
31582	328	Wakeup	cactusrt	RtThread0
28287	164	Wakeup	cactusrt	RtThread0
22329	158	Wakeup	cactusrt	RtThread0
15816	122	Wakeup	cactusrt	RtThread1
31730	120	Wakeup	cactusrt	RtThread1
19320	120	Wakeup	cactusrt	RtThread1
19566	118	Wakeup	cactusrt	RtThread1
25458	118	Wakeup	cactusrt	RtThread1

Visualizing traces

- Perfetto's visualizer is one of the best-in-class
  - Timeline view
  - Table view with sorting
  - SQL processing
  - **Custom vega-lite visualization**
  - **cactus-rt-specific fork: pre-baked vega-lite histogram**



Other features

- Priority inheritance mutex: `cactus_rt::mutex`
- SPSC atomic data structures to pass data to/from real-time thread
  - `RealtimeReadableValue` and `RealtimeWritableValue`
    - Originally designed by Dave Rowland & Fabian Renn-Giles
    - Formally verified with TLA+ and implemented in cactus-rt
- Xorshift constant-time random number generator
  - Not fully uniform random, but guarantees constant time
- Possible future work:
  - Built-in RT memory allocation tracer
  - BPF-based syscall tracer
  - Additional atomic data structures (SPSC ring buffer, RCU, exchange buffer, others)
  - Gradient-descent-based maximum time explorer

cactus-rt is a RT programming framework

- cactus-rt makes building 1000 Hz real-time ROS 2 applications easy
  - Open source with MPLv2
- Comes loaded with bells and whistles for RT
- It ain't a flying car, but you can program one with it?

Star it on GitHub! https://github.com/cactusdynamics/cactus-rt

Questions?

Basic steps of creating a RT ROS 2 application

1. Create `App` and `CyclicThread`
   - Implement `Loop` function
2. Use `InitializeForRos2` to register publishers and subscribers
   - Create type adapter for real-time and ROS types
3. Read and publish messages to ROS 2 via real-time types
4. Log with Quill
5. Trace and visualize with Perfetto