Lessons learned building a self-driving car on ROS

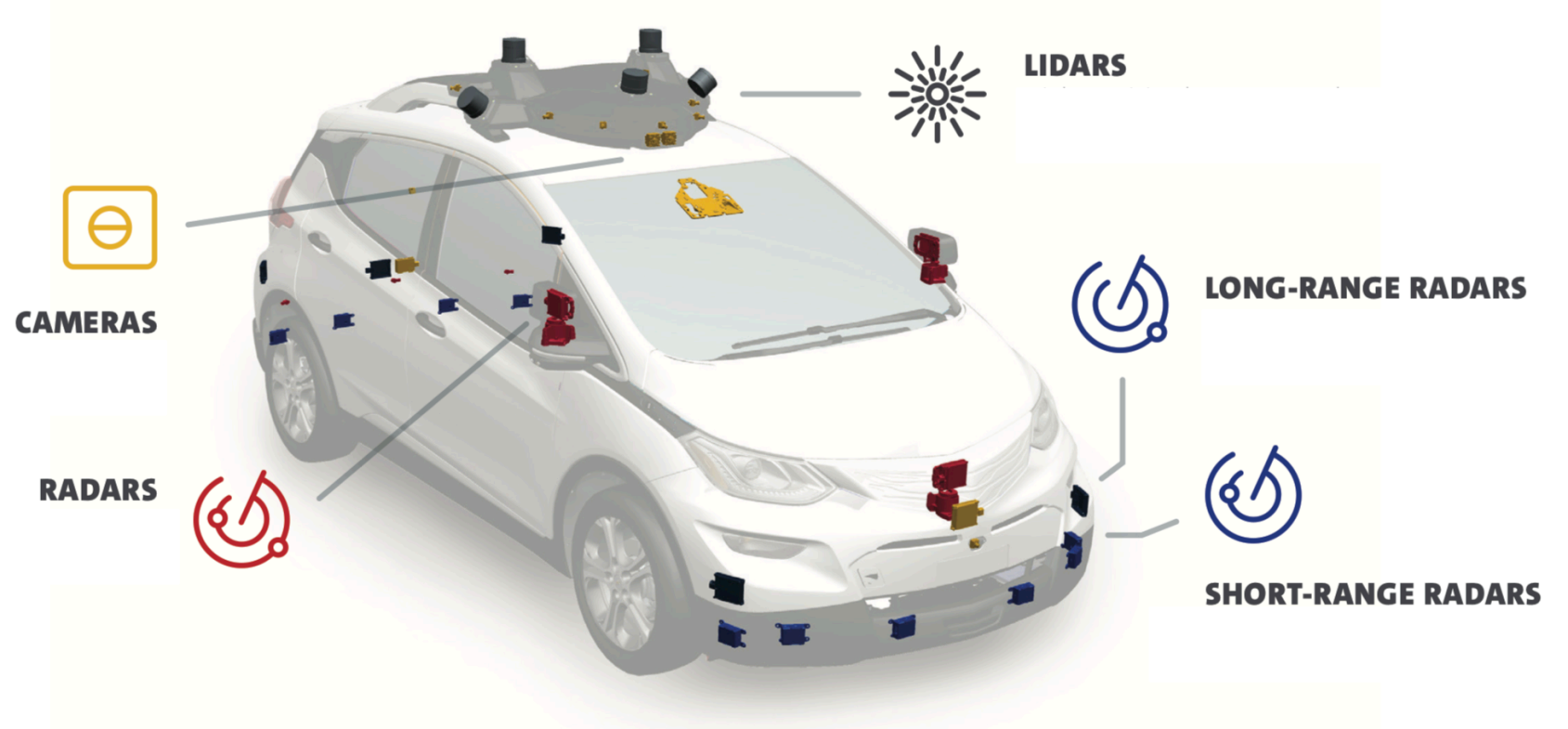
Nicolò Valigi (nicolovaligi.com)
Software Engineer at Cruise Automation

September 29, 2018

The car

A fleet of 100 self driving cars driving around San Francisco and a team of 500 Software Engineers writing code for them.

The sensors



Each car has $\sim$ 10 cameras (fish-eye, wides, ...), an handful of LIDARs and radars, an IMU, wheel sensors, ... Roughly 1GB/s of data coming in from the sensors.

Goals of this talk

The goal of this talk is to share some of the directions where we have pushed ROS to make it scale up to a complex problem.

Three areas of focus:

- Determinism and simulation
- Core framework improvements
- Developer tooling

Simulation and determinism

Why simulation

Testing on real cars is inconvenient and expensive.

We record sensor data as the fleet is driving, and we want to reuse it for tests, simulations, and development of algorithms.

The idea is to spin up a bunch of cloud machines for large-scale simulations of different scenarios.

but...

Where ROS falls short

ROS supports playing back recorded data from a bag, essentially replacing the driver nodes.

The C++ and Python client libraries can also listen to the recorded clock ticks, essentially reproducing the original time stamps.

Simulating time is enough for small systems, but breaks down soon:

- on slower hardware, you either slow down the replay or occasionaly miss frames.
- *determinism* goes out of the window (see Ingo Lütkebohle's great talk at ROSCon 2017).

What we did at Cruise

Simulation needs to be a first-class concept of any Robotics framework. On top of ROS, we added systems to achieve deterministic lock-step simulation from recorded data.

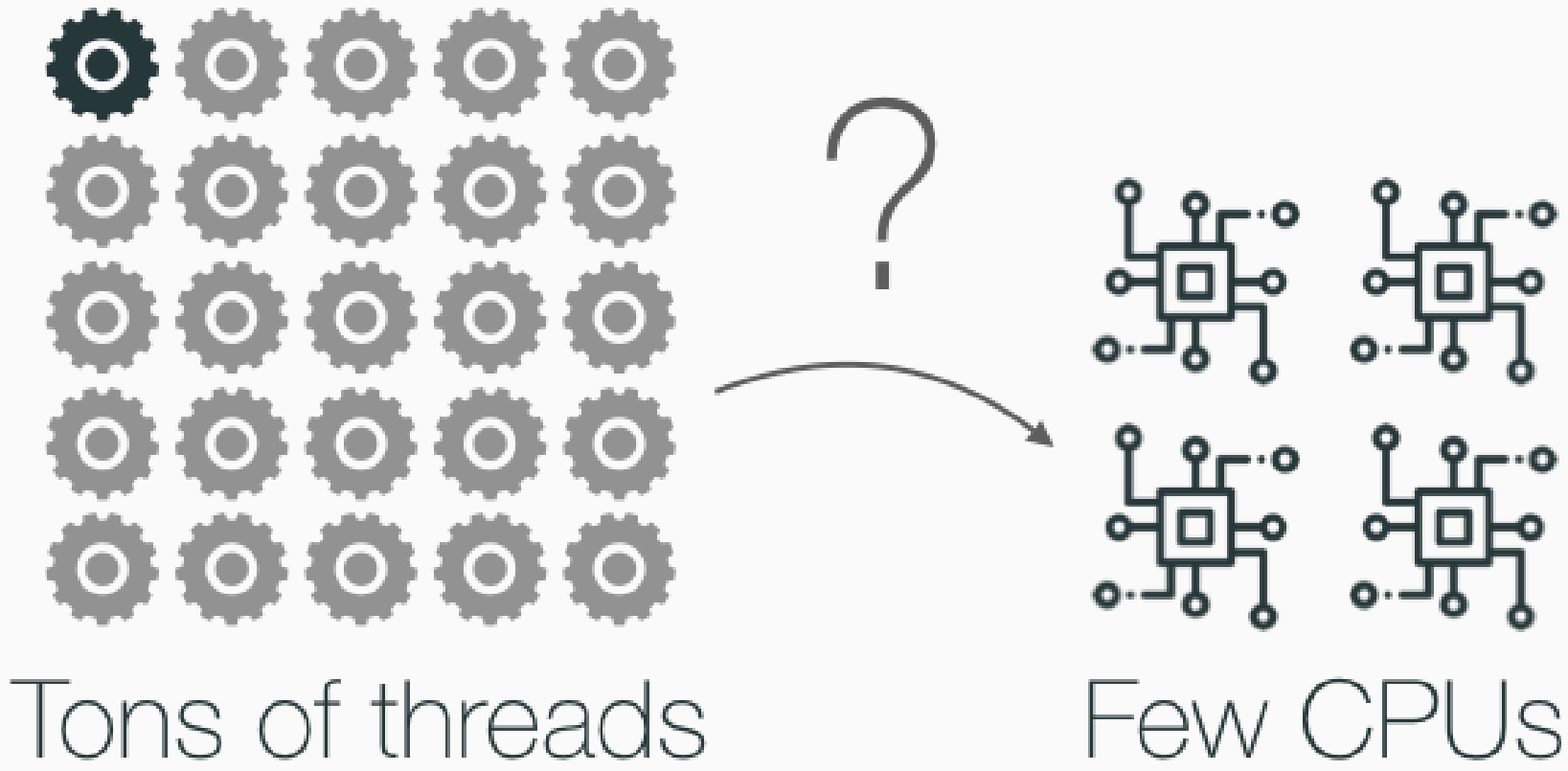
Three steps to achieve this:

1. remove flexibility in the pub-sub model $\rightarrow$ precise computation graph
2. schedule computations deterministically (by controlling data flow)
3. send acknowledgements to keep track of computations inside the nodes.

Core framework improvements

Interaction with the OS scheduling

ROS encourages a proliferation of nodes and nodelets. Under load, this becomes an OS scheduling nightmare, as you have many more processes/threads than CPU cores.



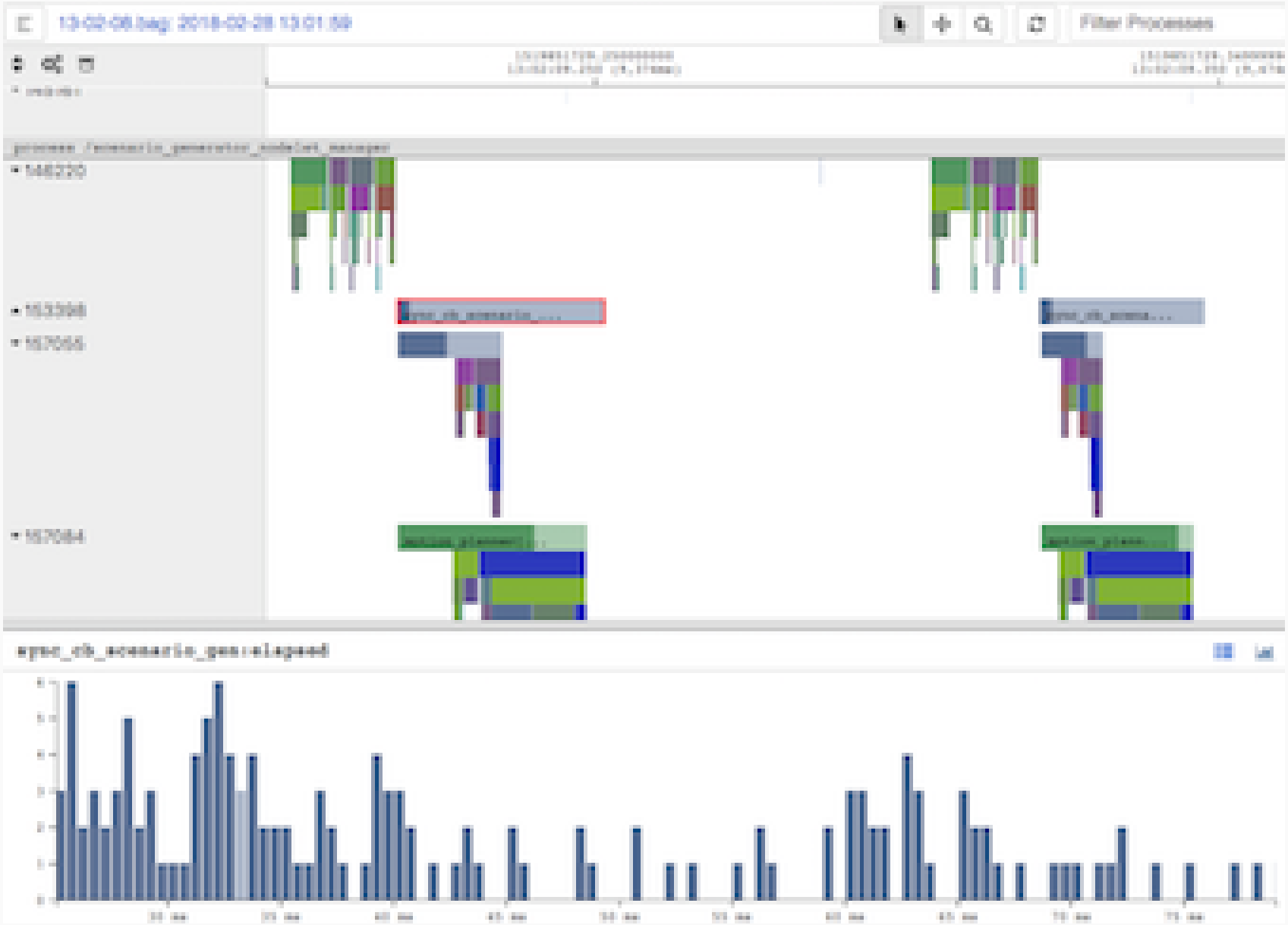
Runtime profiling

The framework doesn't help you keep track of the runtime of different components. In the typical Robotics stack, this means that downstream components are late and you don't know why.

What we did at Cruise

- built some tooling around the Chrome trace viewer for visualizing *flame graphs* throughout the system.
- pass around contextual information that can be used to trace metrics about a thread of computation (e.g. computer vision pipeline).

Profile viewer



Developer tooling

Rosbag, serialization, and data

Bags are good for logging but:

- the file format needs lots of file seeks $\rightarrow$ unsuitable for "big-data" style tooling
- ROS serialization is not forwards compatible at all: just adding a new field breaks deserialization on the old code.

What we did at Cruise

- built an extensive data pipeline to convert bags to slice and dice bags, supporting big-data tools for batch requests.

Webviz vs ROS-style tooling

RQt, Rviz, and friends are very flexibile, but:

- need a Linux workstation with ROS
- have a steep learning curve for both use and customization

What we did at Cruise

- created a Web-based frontend that emulates most of the RQt set of features, and can easily be extended in Javascript.

Webviz

Build system

Catkin's modularity means that it needs to do extra work for each build, making it slow.

What we did at Cruise

- migrated to *Bazel*, the open source version of Google's build system
- sped up workstation builds and added a shared cache for CI

Thank you