IGVC Qt → ROS Transition

2015 Competitive Season

The following document aims to present the motivations for switching from the single-process, multi-threaded, signals-based event system of 2013 and 2014 to the Robot Operating System (ROS - www.ros.org) framework as the foundation for the RoboJackets' IGVC code base.

Current, Qt-Based System Overview

Our current architecture relies heavily on Qt Signals & Slots (doc.qt.io/qt-5/signalsandslots.html). This system allows inheritors of QObject to broadcast messages to any other QObject that choses to receive those messages. It is, at its core, a delegation framework of the style frequently used to grant slightly more advanced callback functionality to GUI elements.

Under this paradigm, our various modules take the form of single classes, some of which run their own threaded loops for device query. This allows for the degree of parallelism we desire to prevent any one sensor from dramatically slowing down any other sensor.

Unfortunately, though this system has served us well up to this point, new feature additions are proving difficult and beginning to muddy the once clean communication framework. The addition of the *Coordinator* system and desire for flexible data-logging semantics have been especially difficult in the Qt Signals & Slots framework.

ROS Fundamentals

ROS is not an "operating system," as its misleading name may suggest. It is, instead, a messaging framework. The goal of the ROS community is to maintain a framework that facilitates collaboration on domain-agnostic robotics software. One ideal result from the existence of ROS would be that much of the "boiler-plate" code necessary to get any robotics project started would already be taken care of for the developer by the community library, encouraging rapid progress in robotic innovation. In practice, hardware platforms vary tremendously and even the most basic functionality is still considered active research. This means that the real benefit of ROS is its messaging framework and the features built directly on this framework, such as data-logging, visualization, and distributed systems.

Comparison

Qt

- Simple message passing API
- Single-process design keeps code compact and local
- Extensive framework results in smaller dependency graph
- Consistent API throughout code base
- Relatively inflexible
- Lack of any features beyond simple messaging
- Code for non-domain features claiming first-class status among domain-specific code
- Dependant on the *qmake* build system to generate necessary meta-objects
- Simple, single project file

ROS

- More complex messaging API
- Multi-process design scatters code both in architecture and file system
- Deep dependency graph with many *bridge* packages needed
- API varies based on immediate application (ie, ROS code v. GUI code)
- Healthy library of messaging extensions (ie, data logging, visualisation)
- Non-domain code relatively hidden from application developers
- Dependent on the catkin *build* system
- Relatively complicated, nested build files via CMakeLists

Specific Grievances with Qt Signals & Slots

Age

Though the Qt developers and community is active and growing, the framework predates even the C++ Standard Library. In the name of backwards compatibility, Qt APIs are slow to change and are rarely compatible with the latest and greatest in C++ syntax and technology. In addition, Qt prefers to deal in its own implementations of standard objects (ie, QString and Qt collections), necessitating a great deal of code to convert between Qt objects and Standard Library objects.

Related to this problem is Qt's poor support of templates.Having only been introduced to C++ two years before Qt's birth, templates were still young and out of the norm in C++ development during the important forming years of the framework. This means that there are types like *QStringList* instead of a generic *QList*<*QString*>. Signals & Slots cannot be templated, a frustrating limitation when approaching problems such as generic data logging.

Queuing

Message queuing is not controllable in Qt Signals & Slots. No matter how long a slot takes to process a signal, it will have to process *every* signal emitted to its slot. This pales in comparison to ROS's fine-toothed control over how many messages are allowed to queue up at any given subscriber before old messages are erased in favor of incoming data. This becomes important when sensor nodes are pumping data in at a faster rate than intelligence nodes can process them.

Coupling

Event-based architectures work best when nodes are completely decoupled, needing only to know when data of a certain pattern is available without caring what the data's source is. Of course, this is only possible to an extent. Qt Signals & Slots, unfortunately, do not go as far as they could to facilitate this decoupling. While there is certainly a large degree of separation and flexibility in managing the signal graph, at some point, developer-facing code must manually couple one signal to one slot, one pair at a time. There are cases where even this level of coupling can become onerous. When a system benefits from dynamic configurations of nodes and data connections, as IGVC does, implementing such a system becomes a complicated developer-facing API, as evident in the recently added *Coordinator* system in the IGVC code base.

Device Drivers

As a final note, it is worth mentioning the key benefit of the large library of community packages available in ROS. Most, if not all, sensors and actuators will already have ROS-enabled drivers written for them. This means application developers will spend less time debugging hardware access code and more time focusing on intelligent parsing of the data. Below follows a table of devices currently used on the IGVC platform and their ROS support status

Device	Status	ROS Package
PointGrey Bumblebee 2	Official Package	camera1394
SICK TiM 551	Official Package	sick_tim
Ardupilot	3rd Party Package	ardupilotmega-ros
Outback A321 GPS	Partial Official Support	gps_common