Terminology

Some terminology that will help easy reading of this document.

- *rapp* : robot or rocon app launched by the robot's app manager
    - used to be distinct from android app.
- *robot app manager* : application manager design such as that on turtlebot
- *gateways* : the component that enables multimaster between ros systems.
- *flips/pulls* : two mechanisms to share ros connections across multimaster

Problems We Are Trying to Solve

"I just want capability abc. I don't care which software implementation provides that, just so long as it has some specified structure that I can implement and automatically map to my application."

ROS 'standardised' connections can do this, but there is no structure or component yet serving these details to software pre-launching (right now individual software components need to manually check every connection for existence...in runtime). In addition, one things we've noticed with ROS is that while it's relatively easy to get folks to standardise on a type, it is much harder/longer getting them to standardise on a topic location and that is easily broken under slightly different, unforeseen implementations (such as we had with T2's and multiple cmd_vel's). Providing the ability for a robot developer to write capability remapping rules for the particular robot (e.g. xyz is served at /this_place/xyz as part of the specification for capability abc) can be of practical assistance making sure portability (especially from robot to robot) is attained even in the face of some implementation variation and while waiting for a convention to emerge.

"I just want capability abc, I don't care about all the detailed parameterisation details, so long as it provides some expected behaviour."

Navigation stack....only the robot developer navigation guru really understands and has the ability to completely parameterise this stack well. It would be nice to completely remove this from any part of the rapp development cycle.

"I want to optionally be able to specify richer constraints on the capability, e.g. 'head camera' instead of 'camera', 'smooth navigation' instead of 'navigation', but still without the detailed control parameterisation.

Brian's example of android localisation is a good reference for this. Fergs also had some interesting ideas around capabilities.

"My computational capacity is frugal and I need to ensure only the capabilities required for my software application are running ... and to turn them off when my software application finishes."

E.g. Turtlebot 2.

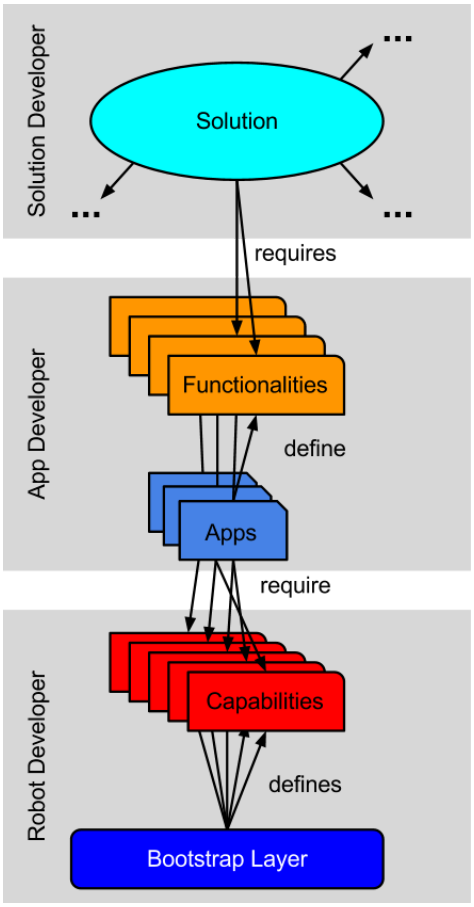
Capabilities

Overview

Capabilities represent an abstraction layer above the bootstrap layer in a robot to simplify and structure the dependencies required by higher level software (e.g. rapps) and inside the robot. They represent an abstraction layer between very robot-specific, detailed knowledge about the robot itself and what it can provide, and software that can utilise that knowledge in a much more consumable and dependant-friendly form. Very strong analogies can be made with Android capabilities (e.g. localisation, camera, internet connection) and Linux's kernel modules.

These capabilities don't always have to be available, nor do they have to be always running, but can be appropriated by higher level software above them.

The picture to the right is where we envision capabilities sitting in a multi-robot rocon environment (everything above the apps is external to the robot).



Goals

- More portable robot apps
- More flexible handling of hardware variation
- Simpler authoring of robot app launchers

Decisions

Important conclusions that came out of the design meeting on 13-04-09.

- Capabilities should be designed, configured and setup by the robot developer.
- They must to be pre-installed, but not necessarily running on the robot.
- Some richness of capability behaviour needs to be supported.
- Initial work on TurtleBots and simulated robot instances.
- Capabilities should mask as much unnecessary detail from the app developer as possible.

What comprises a capability?

- Pre-installed software
- Pre-requisite interfaces (topics, services etc)
- Customised launchers, remappings and parameterisation
- Capability specification file for the robot

Proposal: Capability specification - the .cap file

For app developers and the app manager a certain set of information needs to be provided.
This includes:

- *Behaviour*: human-friendly description of what the capability provides
- *Interface*: connection-name-type tuples
    - e.g. subscriber - /mobile_base/velocity/commands/velocity - geometry_msgs/Twist
- *"Features"* : exposed semantics, parameters or hierarchies (to be decided) that are configurable or selectable by the capability user for richer behaviour.
    - e.g. camera, head_camera, front_camera; localisation: precise (GPS) vs. rough (WIFI)

Background References

- Android Capabilities
- Mike Ferguson's exploration on github (~1-2 years ago)
- Musings from Ken on the Building Manager Project (2 years ago)

Deliverables - May

Capability Core

A very strong feature for me is to be able to determine, pre-launch, whether a rapp is capable of working. This doesn't just mean launchable and runnable, but whether it is capable of doing it's job (you can launch a talker on most any ros system, but it is only capable of doing its job if both a) listener subscriber is available, and b) it is correctly mapped to the talker's publisher).

Technical Deliverables: capability server, rapp capability checks inside app manager
Visual Deliverables: capability visualisation (rqt plugin?) of available and running capabilities.

Portability

Show the portability side of capabilities. One capability - two different robots, two different navi-stacks (and also two different remappings might be good).

Capability Deliverables: navigation
Visual Deliverables: demonstration on two robots

Demo Capabilities

Capabilities we will need for the demo:

Capability Deliverables: navigation, 3d sensor, ... (todo: add to this list)

Rich Capabilities [optional]

A method to expose some richness of a capability to the higher level. Whether that is via Fergs' semantic tags, exposed abstract parameters for the capability, or even separate capabilities that have an inheritance style relationship (e.g. smooth navigation *is a* navigation capability).

Technical Deliverables: design and implementation method to provide 'rich' capabilities
Visual Deliverables: we don't need it for the cafe scenario, but a simple showable, perhaps with cameras?

Appendix A - Initial Notes [2013-04-09]

Problems We Are Trying to Solve

- *Simplify rapp development*
  - Do not require detailed connection and configuration of underlying parts.
  - Provide more structure to a rapp (idl style)
- *Rapp portability*
  - Move from one robot to another with different hw/sw.
  - Depending only on msgs creates an installation conundrum (below)
  - Allow *swapping* the underlying software (e.g. ros navi <-> yujin navi) across robots
- *Richer dependency information*
  - Message location/type and even software is not enough for apps to make good decisions
  - e.g. ‘head’ camera, ‘cleaning pattern’ navigation
  - And the power of choice to make sure you are using the right dependency
- *Relax dependencies on standardised topics*
  - Rather than expect every robot to have a single conforming set of topics, query a capability manager to find out the topic location/type that is provided by the system. Can then easily remap app topics to whatever is provided by the capabilities.
  - This lets us handle somewhat ‘custom’ robots that should still be able to run the apps more easily and gives the robot developers more freedom.
  - Note, hard to foresee when these ‘custom’ robots are needed (e.g. turtlebot2 wishing to implement a cmd_vel multiplexer to handle reactive, teleop and navigational cmd_vels).
- *Ease app Installation*
  - Decide which underlying software it needs to install by looking up some capability-software mappings which are specific to the robot (set up by the robot developer). Then the app can be left ignorant (more portable) and we benefit from automatic sw installs as they are needed.
- *Turning on and off capabilities as needed*
  - Don’t expect capabilities to be always running
    - It’s a waste of computational power
    - Sometimes not feasible to run them all at the same time (on embedded boards esp.)

Fergs also had some interesting ideas around capabilities as well, especially turning on and off. That fits in line with bringing up and down software on start, stop app.

Definition

Capabilities define what the robot is able to do when it is only running the bootstrap layer (no apps have been started yet). The capabilities themselves are defined by the current hardware configuration of the robot and

the control software running it (bootstrap layer).

Each app in turn requires a set of capabilities to be able to run. Hence, the capabilities of a robot define which apps can be run on it.

It is the *robot developer*'s task to define the capabilities of his robot, since he knows all the details of it.

Turtlebot examples:

- mobile_base: allows moving the robot
- RGB camera: allows receiving a video stream
- bumpers: allows reacting to bumper events

Bit of a big picture to the right ------>

Standardization

In order to make apps usable on multiple robots, capabilities need to be standardised. More specific, an app developer needs to know, what a specific capability provides and how it can be accessed. In the ROS world the available standard interface types are topics, services and actions. Hence, a standardised capability needs to define its interface, which is constant among all robots. In this way apps can define dependencies on capabilities, which are checked, when requested to be installed/remapped/configured and run on a specific robot.

Example:

- capability: mobile_base
    - provides
        - moving the robot around
    - Interface
        - cmd_vel
            - description: listens to velocity commands
            - XXX: subscriber
            - type: geometry_msgs/Twist
        - odom
            - description: provides odometry information
            - XXX: topic
            - type: nav_msgs/Odometry
        - motor_power:
            - description: allows enabling and disabling the motors
            - XXX: service
            - type: mobile_base_msgs/MotorPowerSrv

Example: handling implementation variance.

- Diff Mobile Base (“is a” Mobile Base)
- Omni Mobile Base (“is a” Mobile Base)

Handling this variation is important - sometimes an app doesn’t care about the exact mobile base type, sometimes it can only use one or the other, sometimes it changes its behaviour depending (i.e. just needs the information).

We can possibly implement this in various ways - ‘is a’ as indicated above, or by tags or semantics (such as in fergs initial implementation). Alternatively, the app can implement some logic - e.g. depends on omni OR diff.

Capability Execution

We need to be able start and stop capabilities. Current thinking is to have the robot app manager start capabilities when starting a rapp, stop them when stopping the rapp.

Capability Configuration

Capability specifications need to be configured by the robot developer since he best knows the robot. For this, it needs:

- list of preferred software that needs to be installed to provide that capability
- matching parameterisation (configuration) defaults for that software.
- suitable launchers to execute that software

As described before, a standardised capability will have a standardised interface. However, those interfaces may be implemented in varying ways on each robot. For example, mobile_base on one robot may listen to velocity commands on */cmd_vel*, while on another robot that is */mobile_base/commands/velocity*.

Allowing robot developers to adjust capabilities, while still following the standard, gives them more freedom to adapt them to their needs and style. Hence, capabilities have a fixed and flexible configuration part. Fixed parts are those defined by the standard, such as message types, and those the robot developers doesn’t want an app (developer) to change, such as safety-related parameterisation of a capability. Flexible parts include the exact topic names a capability is using and parameters, which an app may change, such as the frequency of publishing specific information, navigation parameters, 3d sensor modules to run etc.

What and how do we expose limited parameterisation of capabilities to apps?

Actions

Probably should start to address small things - immediate and practical problems.

This is only a rough tentative list subject to the whims of the google hangout to come.

Goals

- Provide information for rapps to remap internal topics to the required locations (e.g. turtlebot2 cmd_vel)
- Start the required launchers beneath a rapp instead of the rapp including all the launchers themselves.
- Install the required debs via the app manager without having the rapp depend on those debs.
- Activate and deactivate capabilities as a rapp starts/stops.
- ....

Decisions

- Turtlebot capability specification
    - Map capabilities to packages, launchers and configurations and rules to start them.
- Capability Server
    - Execute launchers with remapping and param configuration upon request to start capability.
    - Provide an interface with which an app can find topics provided by the capability.
- Rapps (.rapp specifications)
    - List capability dependencies in their specification
- Robot App Manager
    - Request rapp capability
    - Remap rapp topics to capability topics and then launch rapp
    - Call capability server to start the requested capability

Maybe best place to start is a dummy talker-listener capability tutorial and then perhaps turtlebot apps/capabilities (e.g. turtlebot_navigation/navigation and follower/3d sensor)?

What is There

rocon_app_platform : upgrading the old turtlebot app manger

- has start app,