The next (big) step for the ROS middleware

May 12, 2013
Dirk Thomas
ROSCon 2013

Open Source Robotics Foundation

Why "ROS 2.0"?

ROS is great and successful...

... but there is still room for improvement.

- Implemented "everything" from scratch
- Unreliable transport with silent failures
- Non-deterministic start-up behavior
- Separate API for nodes/nodelets
- Limited support for multi-robot
- Limited introspection capability

And much more...

Goals

- Reuse existing libraries / less maintenance
- Modular interfaces / separation of concerns
- Better extensibility
- Enable introspection / debugging / dynamic system response
- Reconnectivity on network changes
- "Verifiable" configuration
- Support demands of "new" domains better:

  - Multi-robot
  - Embedded systems
  - Web interfaces
  - Usable for products

Why "ROS 2.0" instead of 1.X+1?

Why decide between writing a node or nodelet at programming time?

- Should have the same API, decision at start time
- Which one to change (breaking the other)?

  - if possible, provide backward compatible API relaying to new API

Why do we need two wire protocols?

- XMLRPC to talk to master and negotiate with nodes
- TCPROS for all other connections

Why "ROS 2.0" instead of 1.X+1?

Want a deterministic start-up behavior?

- Currently, nodes implementing a custom main function will not comply

Why do we rely on a single master?

- Could it be fully distributed or have multiple masters?

ROS 2.0

ROS 2.0 is a codename describing the numerous efforts.

A lot of these require significant API changes – deploying them iteratively in current ROS would be a significant (if not impossible) overhead.

A separated API will:

- Keep the existing API much more stable
- Enable a cleaner design/development process
- Still allow writing a backward compatibility layer on-top

Approach

- Design abstract protocols and interfaces between layers
- Reuse off-the-shelf libraries; make them pluggable
  - msgpack / protobuf
  - zeromq / amqp
  - zeroconf / avahi
- Breaking API if necessary
  (instead of incremental changes)
  - but communication is possible with "ROS 1.0" ecosystems
  - consider scripts to make a potential upgrade path easier

A lot of affected subsystems

- Modularity points

  - exchangeable msg spec / serialization / transport / compression
- Required capabilities of core systems to support existing concepts

  - topics / services / actions
  - parameters / dynamic reconfigure
  - nodelets
- Network level communication

  - Discovery and negotiation
  - Topology
- Configuration space

  - build time vs. deploy time vs. run time

Up to now

- Bottom-up approach
  - Buildsystem in Groovy was the first step
  - Build infrastructure in Hydro the second
- Prototyping
  - Component-based life cycle, dyn. start/reconfigure/stop of comp.
  - Introspectable components including parameters, callbacks, etc.
    (see DARC - distributed asynchronous reactive components)
- Working on the process to
  - Collect use cases (will be classified:
    must-have, nice-to-have, not-being-implemented, out-of-scope)
  - Extract requirements
  - Derive design decisions

Next Steps

Continue and announce the process and open it to the community to collect more use cases and derive design requirements from them.

Based on that:

Derive high-level system architecture

- Write design documents to make decisions comprehensible
- The design documentation should exhibit *traceability*
    - Users should be able to trace design requirements back to design decisions and back to the use cases which drove those design decisions

Outlook

Stage 1

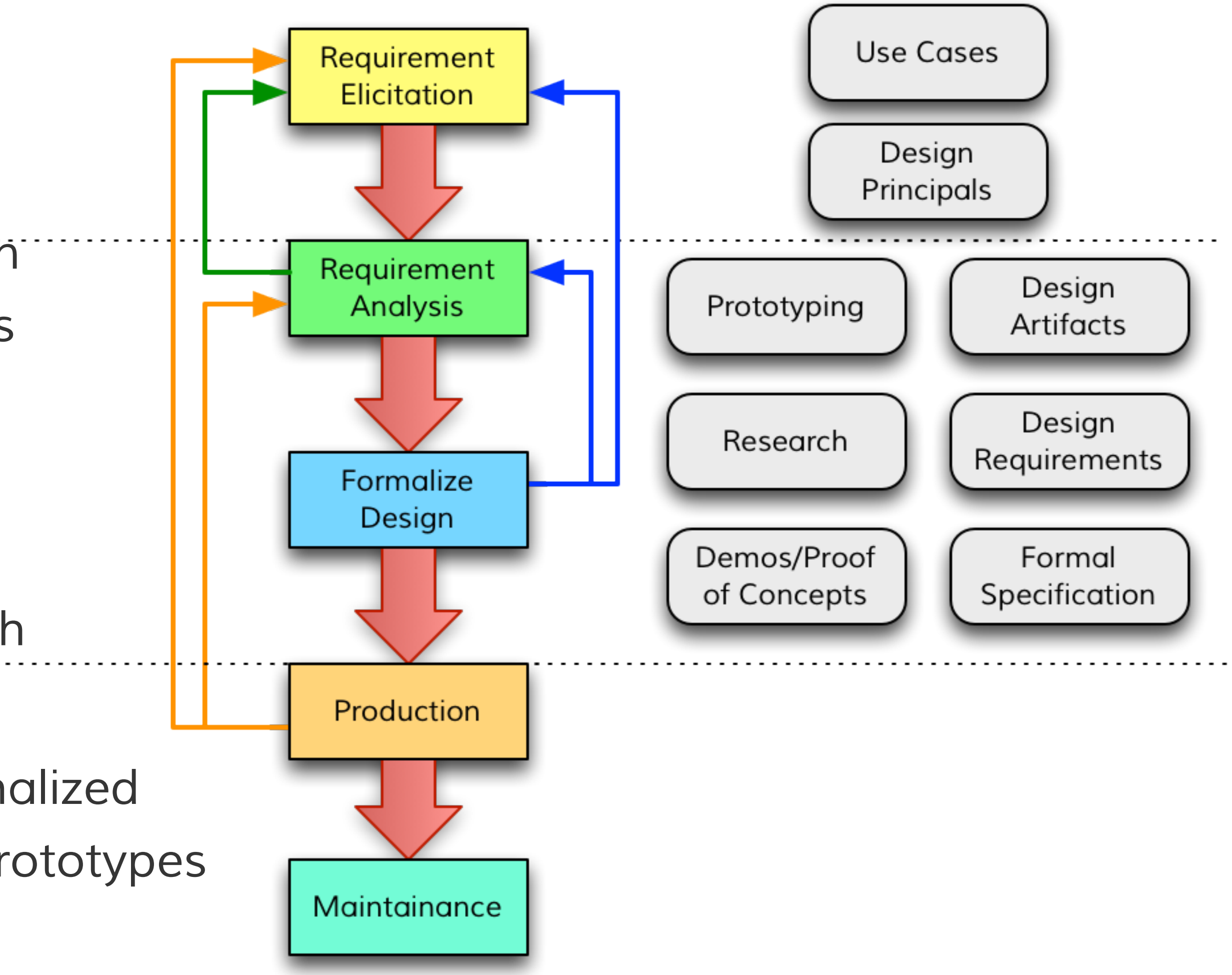
- Use cases Elicitation
- Derive requirements

Stage 2

- Conceptual design
- Whiteboarding
- Technology research

Stage 3

- System design formalized
- Backed by partial prototypes



Alpha release in the first half of 2014.

Stable release in the second half of 2014 (side-by-side with "ROS 1.0").

Questions?
Feedback?

or even better...

Use Cases!

Requirements!