Checking consistency of robot software architectures in ROS
Checking consistency of robot software architectures in ROS

Thomas Witte, Matthias Tichy
28.5.2018

ROS

= Plumbing + Tools + Capabilities + Ecosystem
**roslaunch**

- ROS applications are a network of communicating nodes
- roslaunch can start multiple nodes at once
- xml configuration format
- Can be parametrized and depend on system variables
An example
Most likely result
Debugging...

Strategies:

• Trial and Error, start configuration and manually check data flow
• Guidelines for nodes (e.g. wait for all inputs, fail on configuration errors, …)
• Guidelines for launch configurations
Consistency

- **Inputs and outputs** are created at run-time.
- On **interface changes**, any nodes' launch configuration might **need to be updated**.
- There is **no tool** to statically analyze a launch configuration and **discover these errors**.
This is unsatisfying; can we do better?

**RQ1:** What information can be derived statically from launch files?

**RQ2:** How can we retrieve the missing architectural information without executing the launch file?

**RQ3:** How can we present the data to the user to help understanding launch files?
Tool architecture

RQ1
- resolve includes / substitution args
- XML .launch

RQ2
- analyze nodes
  - sandboxed execution
  - launch-file annotations
  - source code analysis
  - node tree

RQ3
- trace nodes to includes
- check topic connections
- visualize node graph
- create report
RQ1: What information can be derived from launch files?

- Started nodes (package, type, name) as long as they do not depend on roslaunch arguments
- Logical hierarchy (namespaces, groups)
- Configuration (params, args, remappings)
- **No topics, services, message types, ...**
RQ2: How can we retrieve the missing information?

Multiple strategies:

- Analyze source code
- Node interface description (e.g. annotations in the launch configuration)
- Node execution and instrumentation
Lifecycle

1. process arguments
2. read configuration
3. allocate data structures
4. create topic / service connections
5. wait for services or data
6. receive data
7. process data
8. send data
Sandboxed execution

- roslaunch-analyze
- Params, Args
- Topics, Services
- ROS Node
- liberocpp_preload.so
- rospy_preload.py
- advertise, subscribe
- roscpp/rospy
- roscore
- Firejail sandbox
RQ3: How can we present the data to the user?
Evaluation

• Compare recovered node graph with graph at run-time
• ros_tutorials, hector_quadrotor, own projects
• Count correctly recognized topics and services
## Evaluation

<table>
<thead>
<tr>
<th>collision_test_rviz</th>
<th>Topics</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>drone1/quad_node</td>
<td>4/4</td>
<td>2/2</td>
</tr>
<tr>
<td>drone1/quad_script_node</td>
<td>9/9</td>
<td>2/2</td>
</tr>
<tr>
<td>drone1/sim_photo_node</td>
<td>4/4</td>
<td>3/3</td>
</tr>
<tr>
<td>drone1/trajectory_client_node</td>
<td>11/11</td>
<td>2/2</td>
</tr>
<tr>
<td>drone2/quad_node</td>
<td>4/4</td>
<td>2/2</td>
</tr>
<tr>
<td>drone2/quad_script_node</td>
<td>9/9</td>
<td>2/2</td>
</tr>
<tr>
<td>drone2/sim_photo_node</td>
<td>4/4</td>
<td>3/3</td>
</tr>
<tr>
<td>drone2/trajectory_client_node</td>
<td>11/11</td>
<td>2/2</td>
</tr>
<tr>
<td>rviz</td>
<td>8/8</td>
<td>3/3</td>
</tr>
<tr>
<td>trajectory_server_node</td>
<td>7/7</td>
<td>2/2</td>
</tr>
<tr>
<td>hector/indoor_slam_gazebo</td>
<td>Topics</td>
<td>Services</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>gazebo</td>
<td>9/89</td>
<td>30/64</td>
</tr>
<tr>
<td>robot_state_publisher</td>
<td>2/5</td>
<td>2/2</td>
</tr>
<tr>
<td>ground_truth_to_tf</td>
<td>6/6</td>
<td>2/2</td>
</tr>
<tr>
<td>pose_estimation</td>
<td>26/-</td>
<td>2/-</td>
</tr>
<tr>
<td>controller_spawner</td>
<td>0/2</td>
<td>0/2</td>
</tr>
<tr>
<td>estop_relay</td>
<td>3/3</td>
<td>2/2</td>
</tr>
<tr>
<td>pose_action</td>
<td>4/8</td>
<td>2/2</td>
</tr>
<tr>
<td>landing_action</td>
<td>8/15</td>
<td>2/2</td>
</tr>
<tr>
<td>takeoff_action</td>
<td>8/11</td>
<td>2/2</td>
</tr>
<tr>
<td>spawn_robot</td>
<td>2/-</td>
<td>2/-</td>
</tr>
<tr>
<td>hector_mapping</td>
<td>13/13</td>
<td>3/3</td>
</tr>
<tr>
<td>hector_trajectory_server</td>
<td>4/6</td>
<td>2/4</td>
</tr>
<tr>
<td>hector_geotiff_node</td>
<td>3/3</td>
<td>2/2</td>
</tr>
<tr>
<td>rviz</td>
<td>13/13</td>
<td>3/3</td>
</tr>
<tr>
<td>joy</td>
<td>4/4</td>
<td>2/2</td>
</tr>
<tr>
<td>teleop</td>
<td>17/18</td>
<td>2/2</td>
</tr>
</tbody>
</table>
Results

• All strategies are incomplete!
• Node interface must only depend on static configuration
• Inputs and outputs must be configured early
• Blocking waits (e.g. for services) can be intercepted and skipped
Future work

- Improve results of interface recovery (other sources, work around limitations in sandboxed execution)
- Integrate analysis tool into development process (automatically check for interface changes)
- Adapt to new ROS2 launch configuration format
Summary

- Flexible system to enrich static information from launch configurations
- Multiple components to gather data and present them
- Instrumentation and dynamic analysis of unmodified ROS nodes
- Incomplete, but good results for nodes that implement a reasonable lifecycle