ROSPlan
ICAPS-2020 online summer school course, Oct. 12, 2020

Oscar Lima Carrion

German Research Center for Artificial Intelligence (DFKI)
Plan-Based Robot Control research department
Osnabrück, Germany
Course structure

1. ROS
2. ROSPlan
3. Q+A
Course structure

1. ROS
2. ROSPlan
3. Q+A
What is ROS?

- ROS = Robot Operating System
- Not really an operating system, but is easier to explain in the business world this way...
- ROS = (A) plumbing + (B) tools + (C) capabilities + (D) ecosystem (Brian Gerkey)
- (A) provides publish-subscribe messaging infrastructure designed to support the quick and easy construction of distributed computing systems.

Source: https://answers.ros.org/question/12230/what-is-ros-exactly-middleware-framework-operating-system/
What is ROS? (A) plumbing + (B) tools + (C) capability + (D) ecosystem

• (B) ROS provides an extensive set of tools for configuring, starting, introspecting, debugging, visualizing, logging, testing, and stopping distributed computing systems.
• (C) ROS provides a broad collection of libraries that implement useful robot functionality, with a focus on mobility, manipulation, and perception.
• (D) ROS is supported and improved by a large community, with a strong focus on integration and documentation. ros.org is a one-stop-shop for finding and learning about the thousands of ROS packages that are available from developers around the world.
Basic concept 1 : Node

- **Node**: a process that performs computation
- Nodes are combined together into a graph and communicate with one another using topics, services, actionlib and the Parameter Server
- Nodes may reside in different machines transparently

![Diagram of ROS components](image)

`/gazebo` – `/gazebo_gui`

`/scan` – `/turtlebot3_drive`

`/cmd_vel` – `/joint_states`

`/robot_state_publisher`
Basic concept 2: Topic

- **Topics**: Named buses (pipes) over which nodes exchange messages.
- Producer – Consumer design pattern
- In general, nodes are not aware of who they are communicating with.
- nodes that are interested in data **subscribe** to the relevant topic.
- nodes that generate data **publish** to the relevant topic.
Properties of topics

• There can be multiple publishers and subscribers to a topic.
• Intended for unidirectional, streaming communication.
• Nodes communicate with each other by publishing messages to topics.
• An active topic can only have a single message type at a time.
Basic concept 3: Messages

• Earlier we saw that nodes can make connections with each other via topics.
• Informally the message can be seen as the envelope you send via post.
• Messages state what kind of information your nodes need to produce in order to communicate together.
Basic concept 4: Service

- Service: is a client/server communication request system.
- There is no feedback while the operation is being performed but only one time at the end.

Typically is a blocking operation, but you can specify a timeout...
Basic concept 5: actionlib

- Similar to services but additionally you can:
  - Get periodic feedback about the progress of the request
  - Temporarily interrupt a task being carried out
  - Cancel the request
Development in ROS

• Mainly C++ and Python are supported.
• Experimental support for multiple other languages:
  – e.g. Java, lisp, nodejs, lua, ruby, R, Go, etc.
• Code is organized in ROS packages that live inside "catkin" workspaces (development folder).
• A catkin workspace is a folder where you modify, build, and install ROS packages.
ROS bash

- Offers a set of shell ROS commands
- `rosbash` enables tab completion on: roslaunch, rosparam, rosnodel, rostopic, rosservice, rosmeg, rossrv, rosbag.
- Most popular include:
  - `roscd pkgname` (cd to pkgname easily)
  - `rosed pkgname filename` (quickly edit a file)
  - `roscat pkgname filename` (quickly visualize a file in terminal)
  - `rosrun pkgname executable` (run executable from anywhere without having to give its full path)
Some important libraries 1/4

• **tf**: Library to keep track of multiple coordinate frames over time.
• Users can transform points, vectors, etc between any two coordinate frames at any desired point in time.
Some important libraries 2/4

- **move_base**

![Diagram of ROS navigation stack setup](image)

**Navigation Stack Setup**

- `move_base`
  - `amcl`
  - `sensor transforms`
  - `odometry source`
  - `move_base_simple/goal` `geometry_msgs/PoseStamped`
  - `internal nav_msgs/Path`
  - `local_planner` `local_costmap`
  - `recovery_behaviors`
  - `global_planner` `global_costmap`
  - `*cmd_vel*` `geometry_msgs/Twist`
  - `map_server`
  - `/map` `nav_msgs/GetMap`
  - `sensor topics`
    - `sensor_msgs/LaserScan`
    - `sensor_msgs/PointCloud`
  - `sensor sources`
  - `provided node`
  - `optional provided node`
  - `platform specific node`
Some important libraries 3/4

ROSPlan (Oscar Lima)
Some important libraries 4/4

• Rviz

3D visualisation tool for ROS
Course structure

1. ROS
2. ROSPlan
3. Q+A
ROSPlan

• "The ROSPlan framework provides a generic method for task planning in a ROS system".
• First step towards integration of AI planning and robotics.
• Uses different technologies to provide with high level robot control.
• Main support: PDDL 2.1
• Experimental support: PPDDL, RDDL, HDDL (HTN new!)
• Plan execution and monitoring via:
  – Simple plan dispatch or Esterel plan dispatch
Architecture
Components

- **Knowledge Base (KB)**: stores the (symbolic) planning model (domain and state); Communication via services
- **Problem interface**: query state from KB and create a problem instance
- **Planner interface**: wrapper around the AI planner, write its output to a topic
- **Parsing interface**: Convert planner output into a representation suitable for execution
- **Plan dispatch**: Execution and monitoring layer
Debugging tools

- **KB rqt gui**: KB visualisation/manipulation
- **Esterel rqt gui**: Plan visualisation and realtime execution progress
ROSPlan KB gui

- Command: `rqt --standalone rosplan_rqtdispatcher.ROSPlanDispatcher`
Action interface (RPActionInterface)

- Provides a base class implementation to ease the process of robot action creation
- Available in C++ and Python
- Steps:
  - 1) Subscribe to the plan
  - 2) If action is not relevant, exit
  - 3) Send actionlib feedback telling the action is enabled
  - 4) Provide virtual function for concrete implementation
  - 5) Upon action success, update KB according to the model
  - 4) Send actionlib result (action achieved or failed)
Simulated Actions *(RPSimulatedActionInterface)*

- Syntethic simulator of actions
- Made as a replacement of a physics-based robot simulator
- Make multiple mockup actions, useful for testing/debugging
- Parameters:
  - `action_duration`
  - `action_duration_stddev`
  - `action_probability`
Esterel plan dispatch

• Realtime graph-based algorithm for plan execution - monitoring
• Support for concurrent actions
• Preconditions are checked before sending action for execution

• In a nutshell: Converts a temporal PDDL planner output into a graph, which edges represent ordering constraints.
Semantics behind edges

- Conditional edge encapsulates 1 or more casual links
- All edges specify ordering constraints: source node effects need to happen before sink node gets signal
- Node cannot fire unless it has received all incoming edges / signals
- Examples:

```
A1 START
  ↓                ↓
  [0, inf]         [0, inf]
  ↓       ↓       ↓       ↓
A2 END    A2 END  A2 END  A2 END
  ↓       ↓       ↓       ↓
  [0, inf]  [3,3]  [15,15] [10.5]
  ↓       ↓       ↓       ↓
A1 END    A2 START A2 START A2 START
  ↓                ↓
  [1,1]            [15,15]
  ↓                ↓
  [0.5]            [10.10]
  ↓                ↓
A1 END
```

Example: "ROSPlan" (Oscar Lima)
Esterel plan dispatch explanation 1/4

step 1: PreparePlan() output

planner output example:

```plaintext
10.001: (goto_waypoint kenny wp0 wp1) [10.000]
20.002: (goto_waypoint kenny wp1 wp2) [10.000]
```

Node associated action (not displayed)

- **node_id: 0**
  - name: plan_start
  - node_type: PLAN_START
  - edges_in = []
  - edges_out = []

- **node_id: 1**
  - name: go_to_waypoint_start
  - node_type: ACTION_START
  - edges_in = []
  - edges_out = []

- **node_id: 3**
  - name: go_to_waypoint_start
  - node_type: ACTION_START
  - edges_in = []
  - edges_out = []

- **node_id: 2**
  - name: go_to_waypoint_end
  - node_type: ACTION_END
  - edges_in = []
  - edges_out = []

- **node_id: 4**
  - name: go_to_waypoint_end
  - node_type: ACTION_END
  - edges_in = []
  - edges_out = []
Esterel plan dispatch explanation 2/4

- **Step 2**: Order nodes by dispatch time
  - INITIAL STATE
  - A1 START
  - A1 END
  - A2 START
  - A2 END

- **Step 3**: Create start-end action edges
  - INITIAL STATE
  - A1 START
  - A1 END
  - A2 START
  - A2 END

- **Step 4**: Create conditional support edges
  - INITIAL STATE
  - A1 START
  - A1 END
  - A2 START
  - A2 END
Esterel plan dispatch explanation 3/4

• Interference edge

• a and b interfere if:

\[
\text{eff}^+ a \cap \text{eff}^- b \neq \emptyset
\]
\[
\text{pre}^+ a \cap \text{eff}^- b \neq \emptyset
\]
\[
\text{pre}^- a \cap \text{eff}^+ b \neq \emptyset
\]
\[
\text{eff}^n a \cap \text{eff}^n b \neq \emptyset
\]

• ROSPlan currently checks the ones marked with *
Esterel plan dispatch explanation 4/4
ROSPlan demos

- Example launch file on how to launch a turtlebot3 robot in *stage* simulator

- **Exploration** demo in stage
ROSPlan demos

- Turtlebot exploration demo in Gazebo
ROSPlan turtlebot demo

Course structure

1. ROS
2. ROSPlan
3. Q+A