ROSPlan Review

KCL 01/07/19

Part 1:
ROS

What is ROS?

A set of software libraries and tools that help you build robot applications.

- A ROS system is distributed into ***nodes***.
- A system is configured by ***parameters***, and then all the nodes are ***launched***.
- Nodes talk by publishing ***messages*** and calling ***services***.

Nodes

/rgbd_camera

/face_recognition

/collision_avoidance

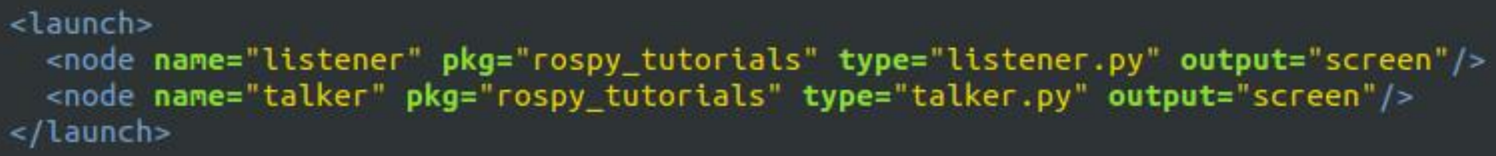
A node is its own separate program.

It is usually one process, but it can be multiple processes.

Nodes have:

- A node type.
- A unique name.
- Launch parameters.

Nodes



```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener.py" output="screen"/>
  <node name="talker" pkg="rospy_tutorials" type="talker.py" output="screen"/>
</launch>
```

```
> roslaunch roslaunch rospy_tutorials talker_listener.launch

> rosnode list

> rosnode info /talker
```

Parameters

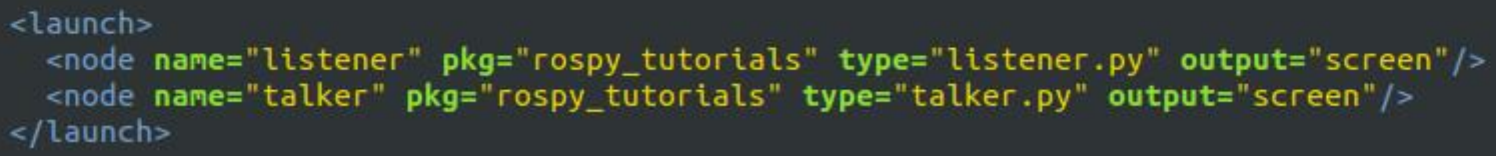
/rgbd_camera
/face_recognition
/collision_avoidance

A parameter server keeps track of parameters, which can be accessed by name.

Parameters are intended to be fairly static values that do not change.

They are well suited for initial configuration.

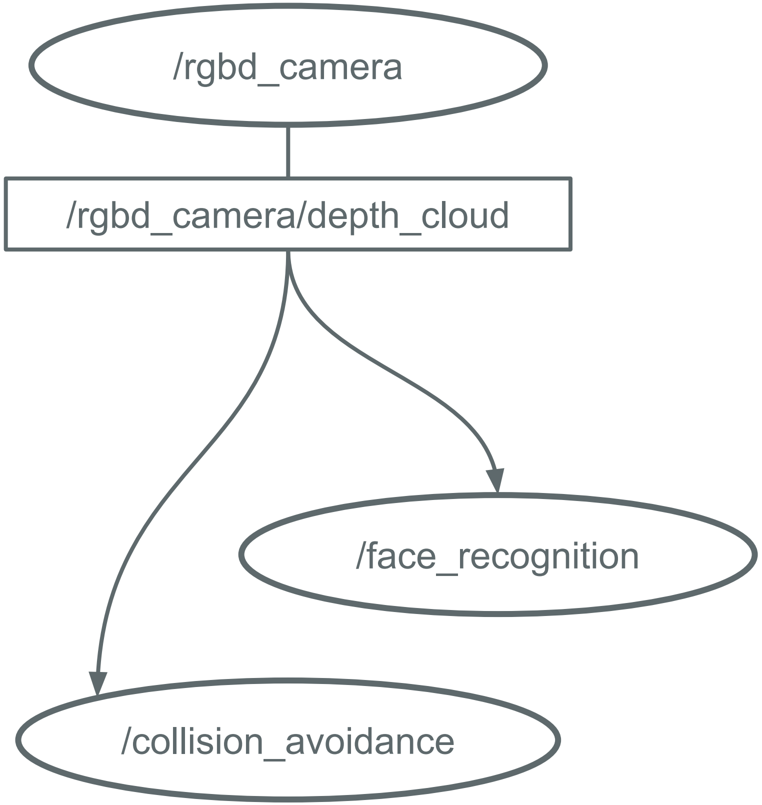
Parameters



```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener.py" output="screen"/>
  <node name="talker" pkg="rospy_tutorials" type="talker.py" output="screen"/>
</launch>
```

```
> rosparam list

> rosparam get /run_id
```

Messages



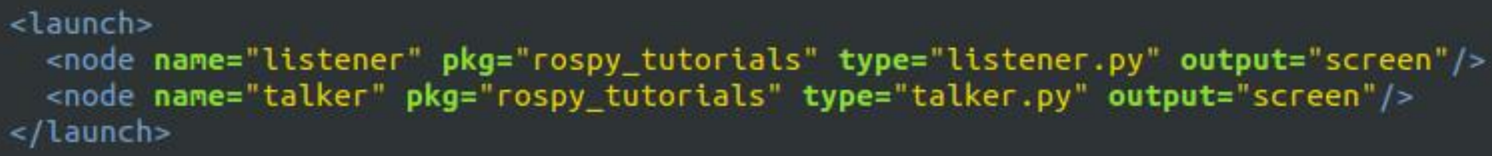
Nodes talk to one another through ROS messages.

A ROS message is published on a topic. Every node subscribed to that topic will recieve the message.

Each Topic has:

- A list of publishers and subscribers.
- A message type.
- A globally unique name.

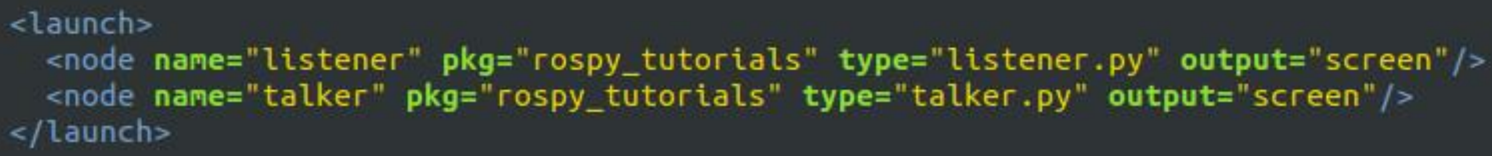
Messages



```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener.py" output="screen"/>
  <node name="talker" pkg="rospy_tutorials" type="talker.py" output="screen"/>
</launch>
```

```
> rostopic list

> rostopic info /chatter

> rostopic echo /chatter -n 5
```

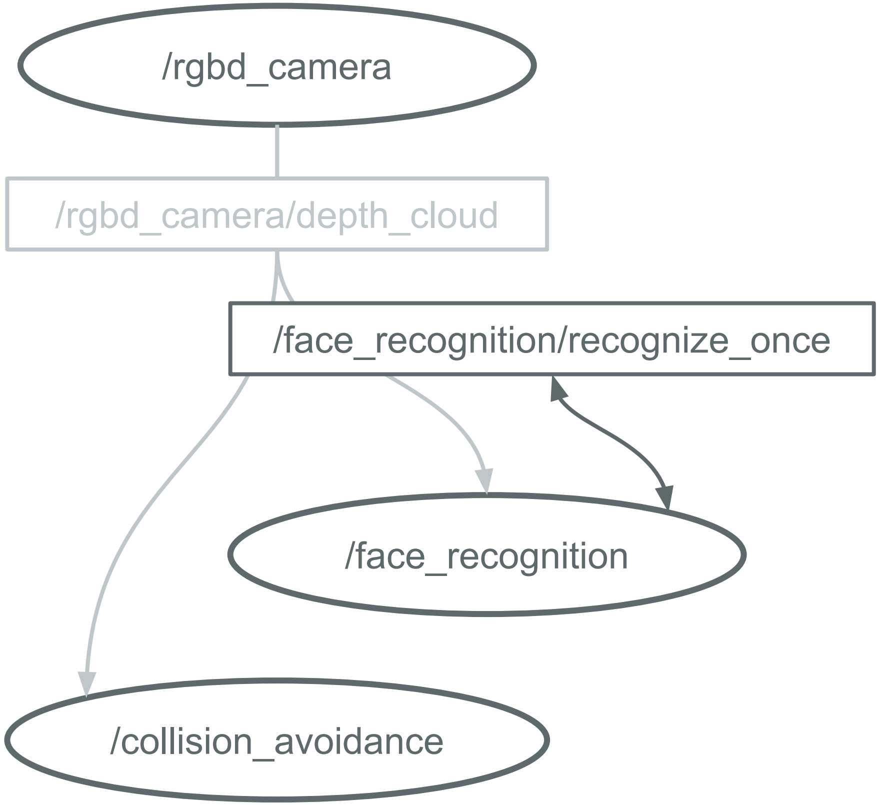
Services



```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener.py" output="screen"/>
  <node name="talker" pkg="rospy_tutorials" type="talker.py" output="screen"/>
</launch>
```

```
> rosservice list

> rosservice info /talker/get_loggers

> rosservice call /talker/get_loggers "{}"
```

Services



Some nodes provide a remote procedure call, called a service.

Services have:

- a globally unique topic name.
- A single service provider.
- A request and response type.

They are different from messages:

- Calling a service is blocking.
- Services have a return value.

PDDL

Domain

A PDDL domain looks like this:

```
(define (domain <domain name>)

(:types <list of types>)
(:constants <list of constants>)
(:predicates <list of predicates>)
(:functions <list of functions>)
<first operator>
…
<last operator>
)
```

Domain

A PDDL domain looks like this:

```
(define (domain <domain name>)

(:types <list of types>)
(:constants <list of constants>)
(:predicates <list of predicates>)
(:functions <list of functions>)
<first operator>
…
<last operator>
)
```

```
(define (domain survey)
(:types
    waypoint
    robot
)
(:constants
    ugv01 ugv02 - robot
    Inside road - waypoint
)
```

Domain

A PDDL domain looks like this:

(define (domain *<domain name>*)

(:types *<list of types>*)

(:constants *<list of constants>*)

(:predicates *<list of predicates>*)

(:functions *<list of functions>*)

<first operator>

…

<last operator>

)

(:predicates

(robot_at ?v - robot ?wp - waypoint)

(connected ?from ?to - waypoint)

(visited ?wp - waypoint)

)

Domain

A PDDL domain looks like this:

```
(define (domain <domain name>)

(:types <list of types>)
(:constants <list of constants>)
(:predicates <list of predicates>)
(:functions <list of functions>)
<first operator>
…
<last operator>
)
```

```
(:functions
        (distance ?wp1 ?wp2 - waypoint)
)
```

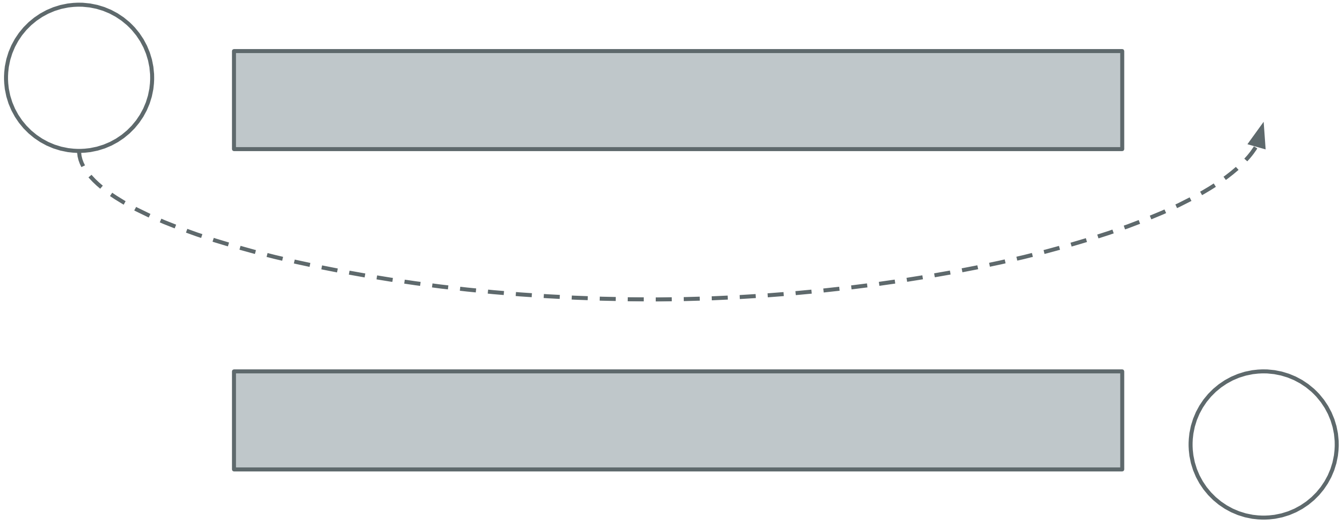
Domain

A PDDL domain looks like this:

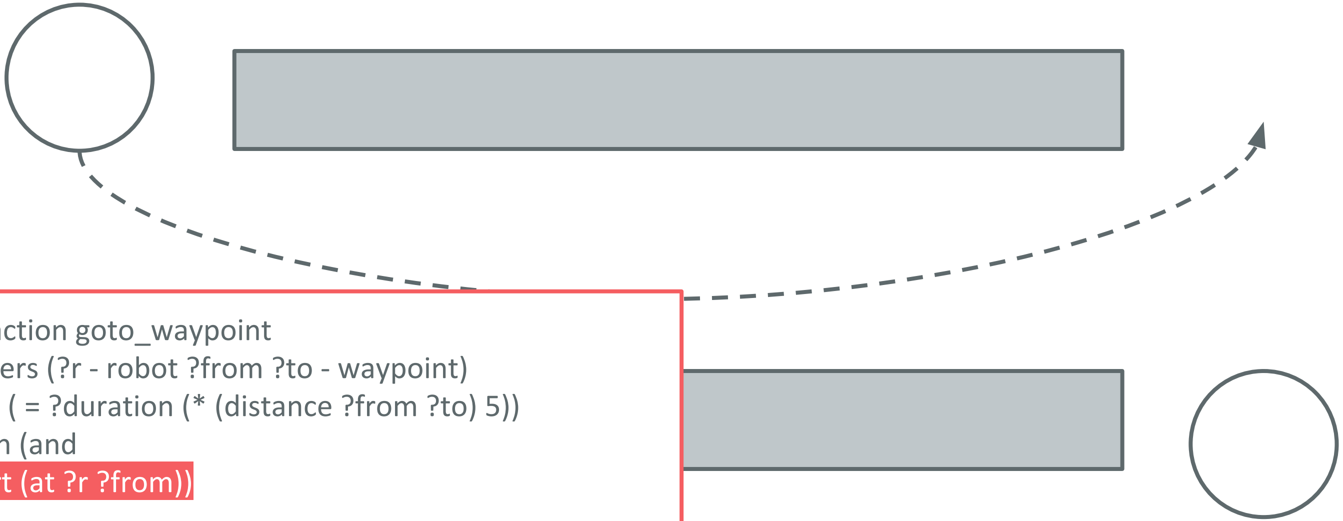
```
(define (domain <domain name>)

(:types <list of types>)
(:constants <list of constants>)
(:predicates <list of predicates>)
(:functions <list of functions>)
<first operator>
...
<last operator>
)
```

```
(:durative-action goto_waypoint
   :parameters (?r - robot ?from ?to - waypoint)
   :duration ( = ?duration (* (distance ?from ?to) 5))
   :condition (and
      (at start (at ?r ?from))
   )
   :effect (and
      (at start (not (at ?r ?from)))
      (at end (at ?r ?to))
   )
)
```

A Robot Plan

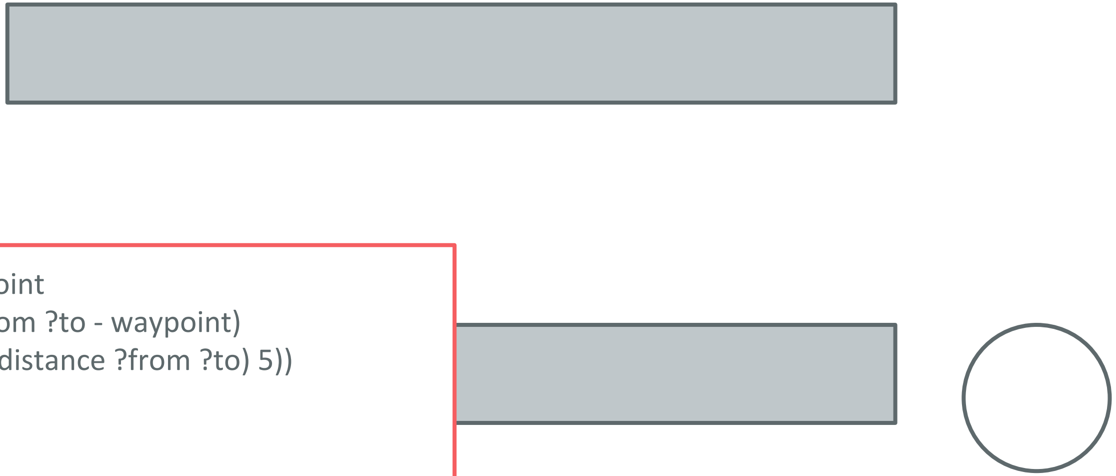


A Robot Plan



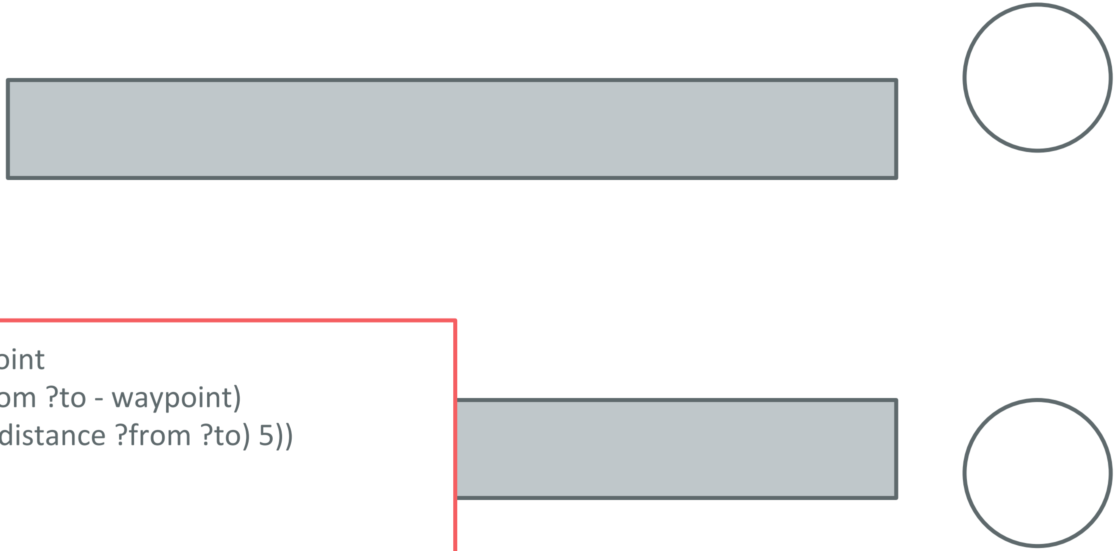
```
(:durative-action goto_waypoint
   :parameters (?r - robot ?from ?to - waypoint)
   :duration ( = ?duration (* (distance ?from ?to) 5))
   :condition (and
      (at start (at ?r ?from))
   )
   :effect (and
      (at start (not (at ?r ?from)))
      (at end (at ?r ?to))
   )
)
```

A Robot Plan



```
(:durative-action goto_waypoint
   :parameters (?r - robot ?from ?to - waypoint)
   :duration ( = ?duration (* (distance ?from ?to) 5))
   :condition (and
      (at start (at ?r ?from))
   )
   :effect (and
      (at start (not (at ?r ?from)))
      (at end (at ?r ?to))
   )
)
```

A Robot Plan



```
(:durative-action goto_waypoint
   :parameters (?r - robot ?from ?to - waypoint)
   :duration ( = ?duration (* (distance ?from ?to) 5))
   :condition (and
      (at start (at ?r ?from))
   )
   :effect (and
      (at start (not (at ?r ?from)))
      (at end (at ?r ?to))
   )
)
```

Problem

A PDDL problem looks like this:

```
(define (problem <problem name>)
(:domain <domain name>)
(:objects <list of objects>)
(:init
  <initial state description>
)
(:goal
  <goal description>
)
)
```

Problem

A PDDL problem looks like this:

```
(define (problem <problem name>)
(:domain <domain name>)
(:objects <list of objects>)
(:init
  <initial state description>
)
(:goal
  <goal description>
)
)
```

```
(define (problem task)
(:domain survey)
(:objects
   wp0 wp1 wp2 wp3 wp4 - waypoint
   auv01 - robot
)
```

Problem

A PDDL problem looks like this:

```
(define (problem <problem name>)
(:domain <domain name>)
(:objects <list of objects>)
(:init
  <initial state description>
)
(:goal
  <goal description>
)
)
```

```
(robot_at auv01 wp0)
(robot_at ugv01 wp0)
(= (distance wp0 wp1) 5.748)
...
```

Problem

A PDDL problem looks like this:

```
(define (problem <problem name>)
(:domain <domain name>)
(:objects <list of objects>)
(:init
  <initial state description>
)
(:goal
  <goal description>
)
)
```

```
(and
   (robot_at ugv01 wp0)
   (robot_at auv01 wp5)
)
```

Storing the State in ROSPlan

Every time planning happens the initial state is the *current state of the world*.

1. ROSPlan must store the current state in a node.
2. The current state must be updated continuously from sensor data.

ROSPlan

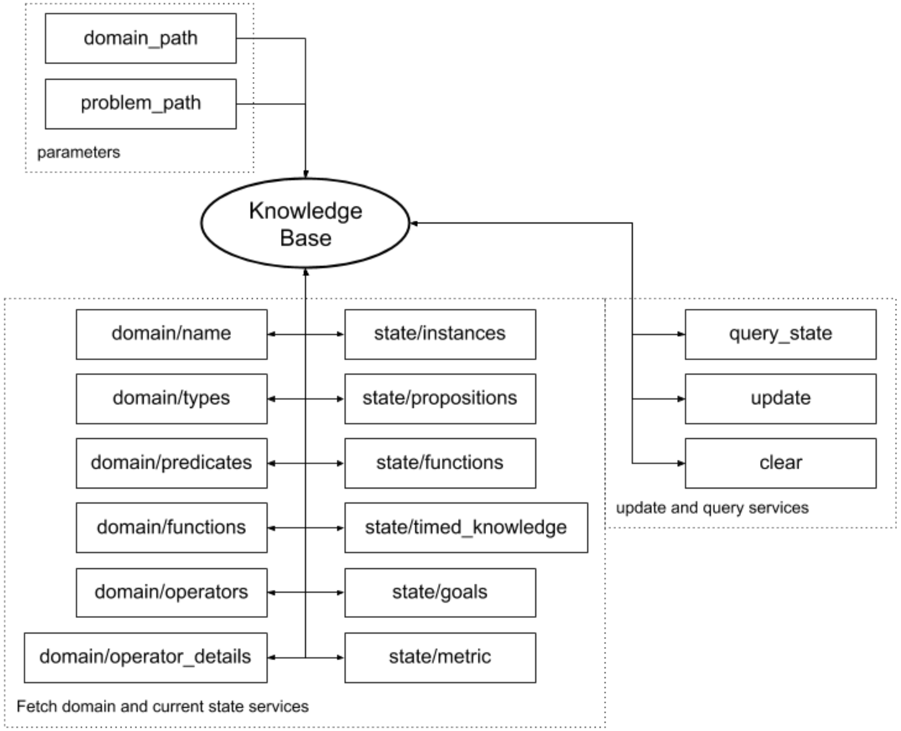
Nodes, Parameters, and Launch

Knowledge Management System

The KMS stores the domain model and the current state.

The node is launched with a domain file parameter, and an optional problem file parameter, which specifies the initial state.

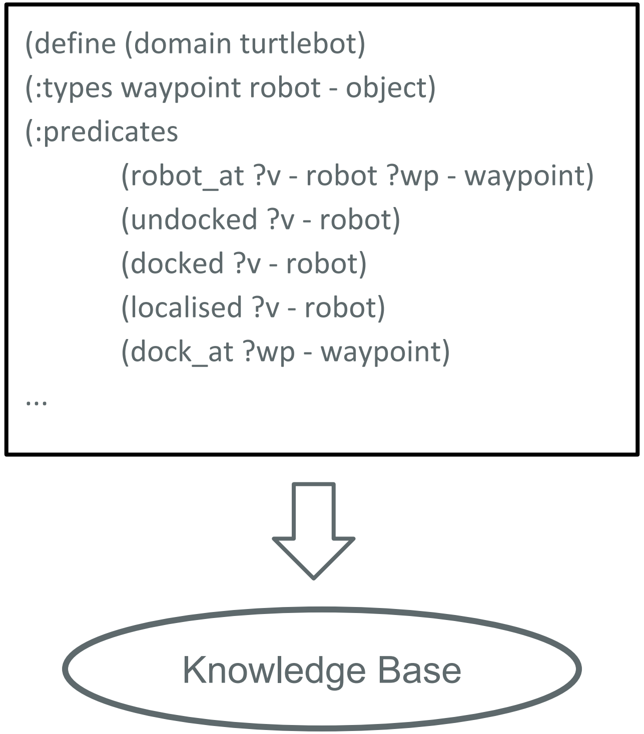
There are many services for fetching domain details, the current state, and performing queries.



Node 1/5

Knowledge Management System

The Knowledge Base node is used to access a domain and state from any node.

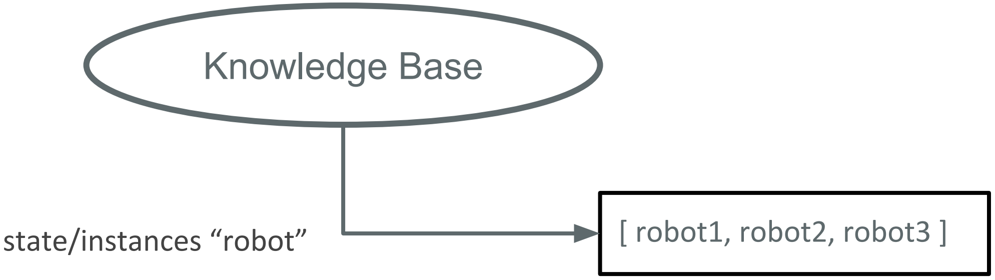


```
(define (domain turtlebot)
(:types waypoint robot - object)
(:predicates
        (robot_at ?v - robot ?wp - waypoint)
        (undocked ?v - robot)
        (docked ?v - robot)
        (localised ?v - robot)
        (dock_at ?wp - waypoint)
...
```

Node 1/5

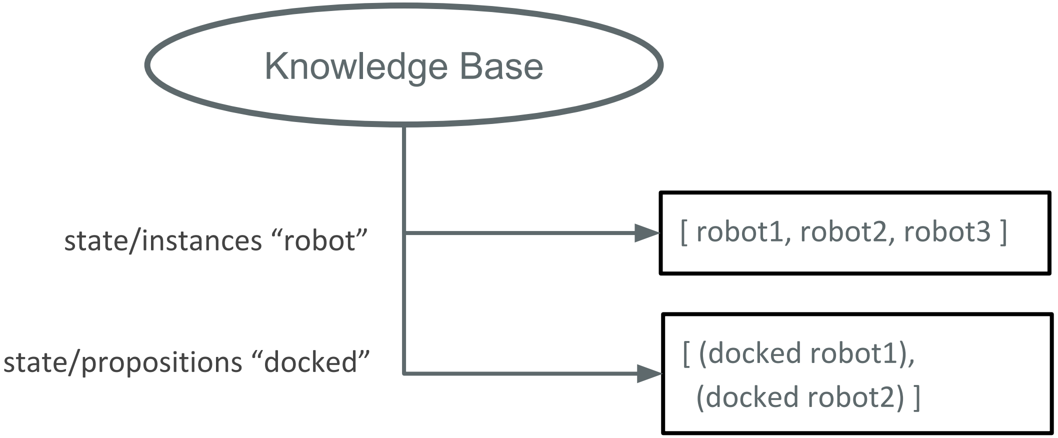
Knowledge Management System

The Knowledge Base node is used to access a domain and state from any node.



Knowledge Management System

The Knowledge Base node is used to access a domain and state from any node.



Knowledge Base
state/instances "robot"
[robot1, robot2, robot3]
state/propositions "docked"
[(docked robot1), (docked robot2)]

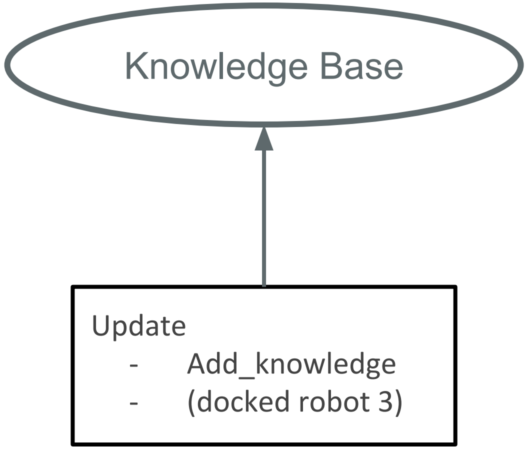
Node 1/5

Knowledge Management System

The Knowledge Base node is used to access a domain and state from any node.

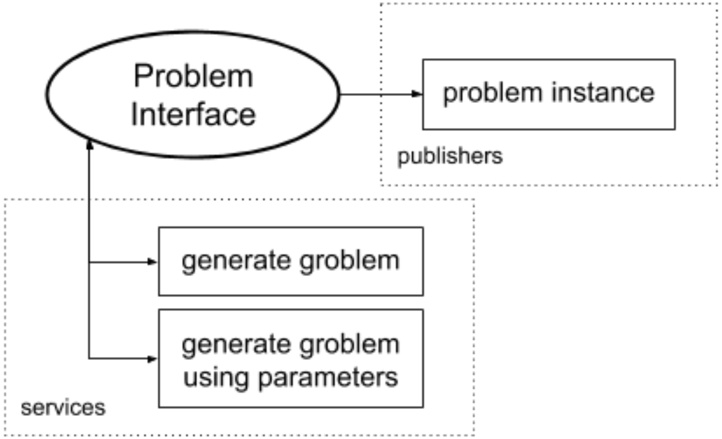
The Knowledge Base can be updated based on sensed data to represent the current state.

This is essential for online planning.



Node 1/5

Problem Interface

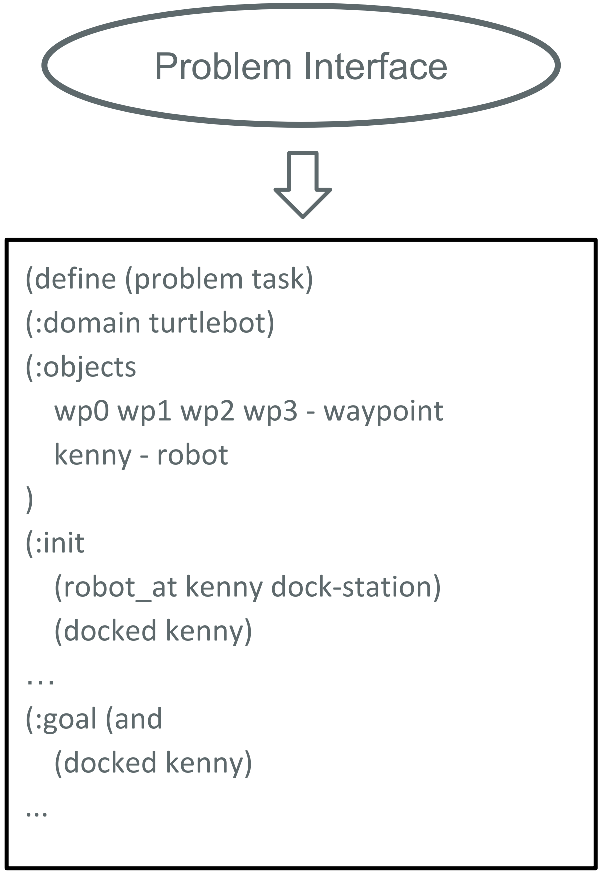


This node is used to generate a problem instance.

It fetches the domain details and current state through service calls to the KMS.

The problem instances can be written to a file and/or published as a ROS message.

Problem Interface



Problem Interface

```
(define (problem task)
(:domain turtlebot)
(:objects
   wp0 wp1 wp2 wp3 - waypoint
   kenny - robot
)
(:init
   (robot_at kenny dock-station)
   (docked kenny)
...
(:goal (and
   (docked kenny)
...
```

This node is used to generate a problem instance.

It fetches the domain details and current state through service calls to the KMS.

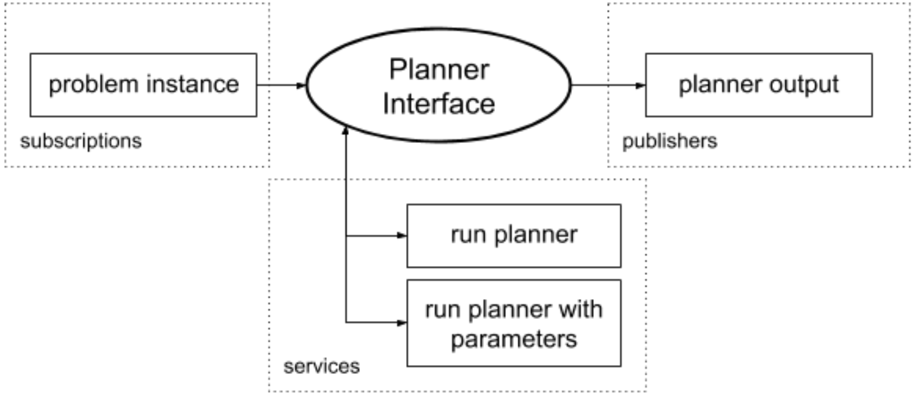
The problem instances can be written to a file and/or published as a ROS message.

Planner Interface

This node is a wrapper for the AI Planner. The *run planner* service returns true if a solution is found by the planner.

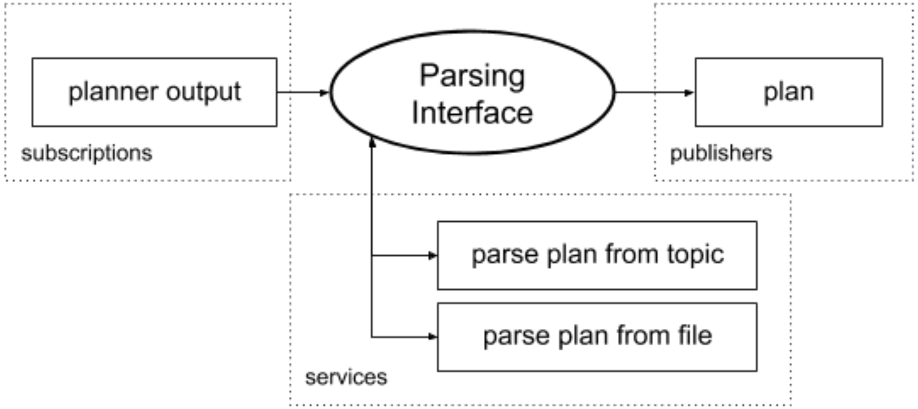
The command line used to run the planner is specified by a launch parameter.

The resulting solution, if one was found, can be written to a file and/or published as a ROS message.



Node 3/5

Parsing Interface



This node is used to convert planner output into a representation for execution.

A ROS message is created for each individual action.

The ROS messages are stored together in one plan message, which also describes when those messages should be published.

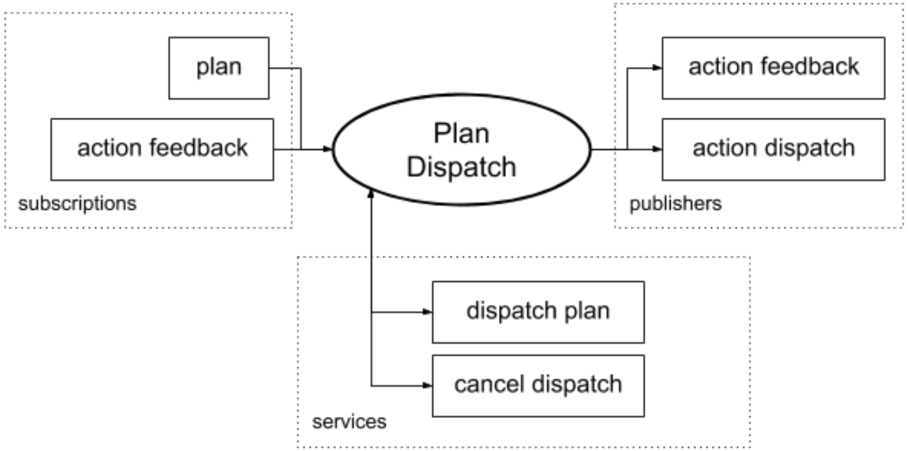
Node 4/5

Plan Dispatch

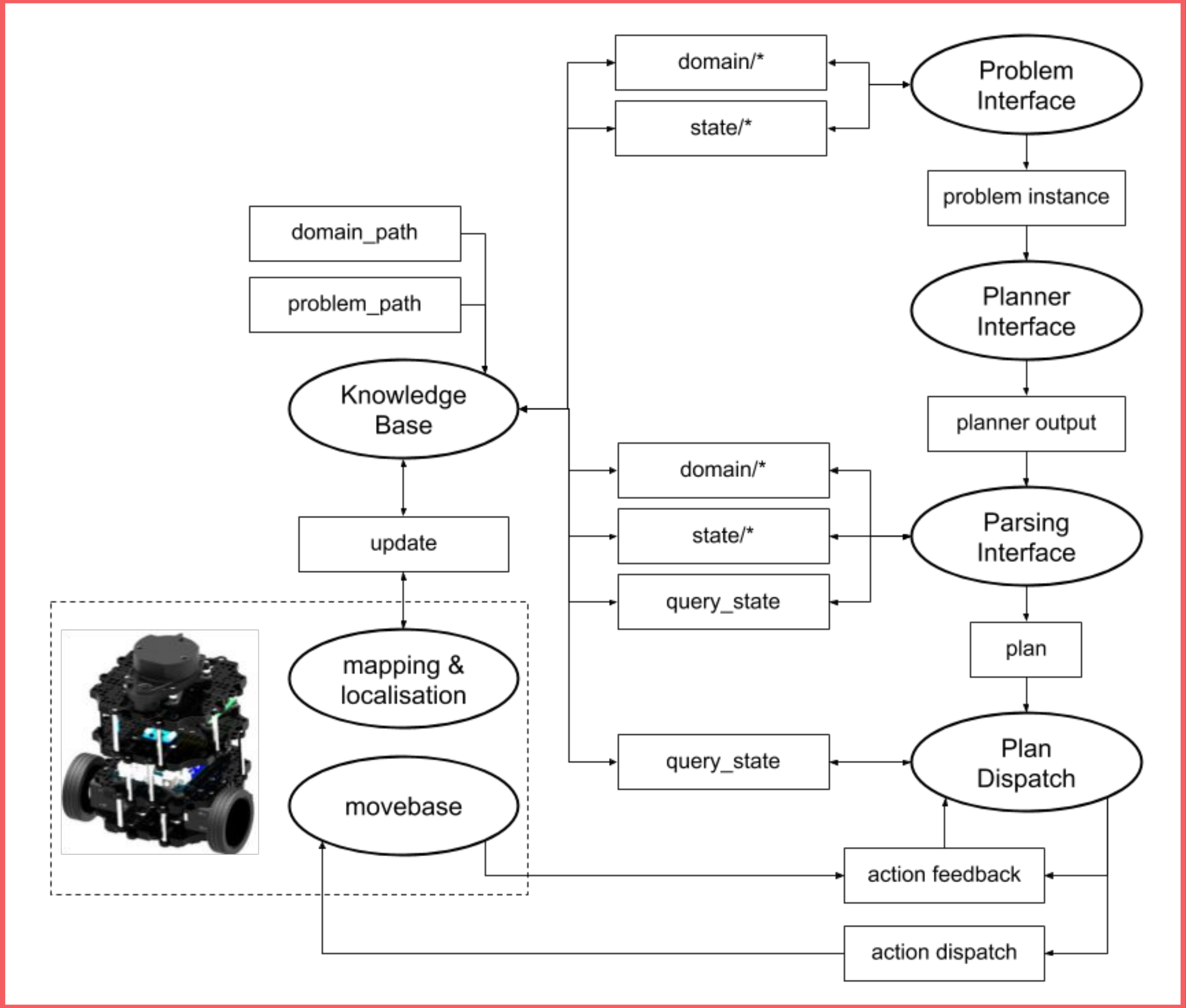
This node is responsible for executing a plan message by publishing the action messages at the right times.

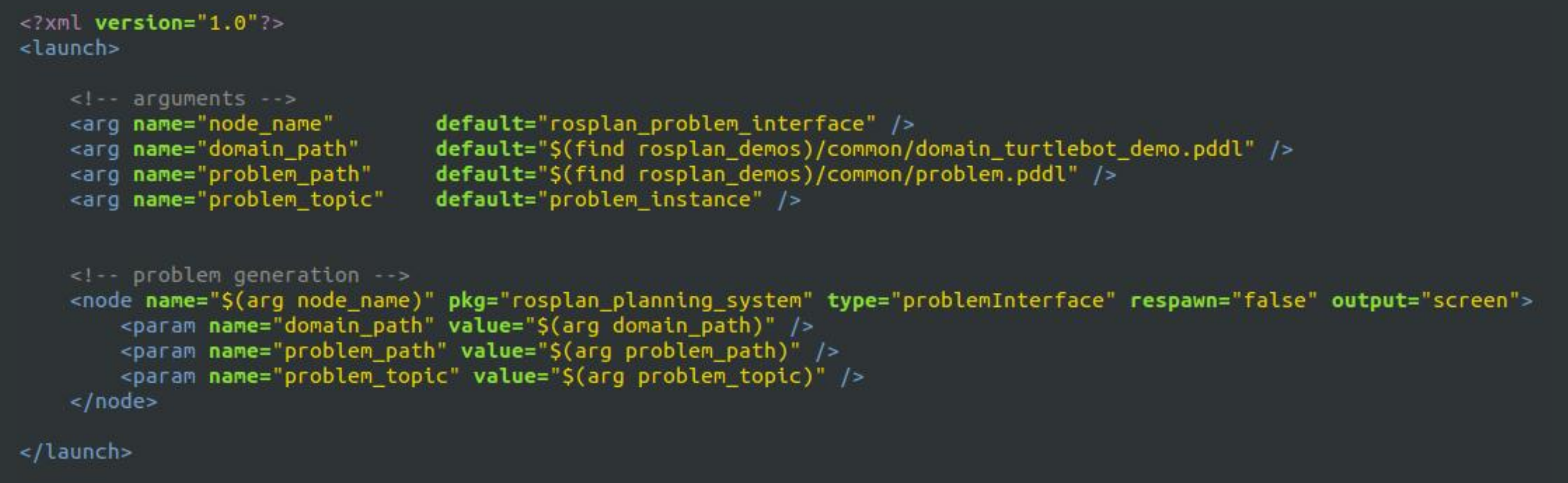
The Plan Dispatch node is closely tied to the Parsing Interface. They must use the same plan representation.

The plan is executed as a service, which returns true if the plan was executed without errors.

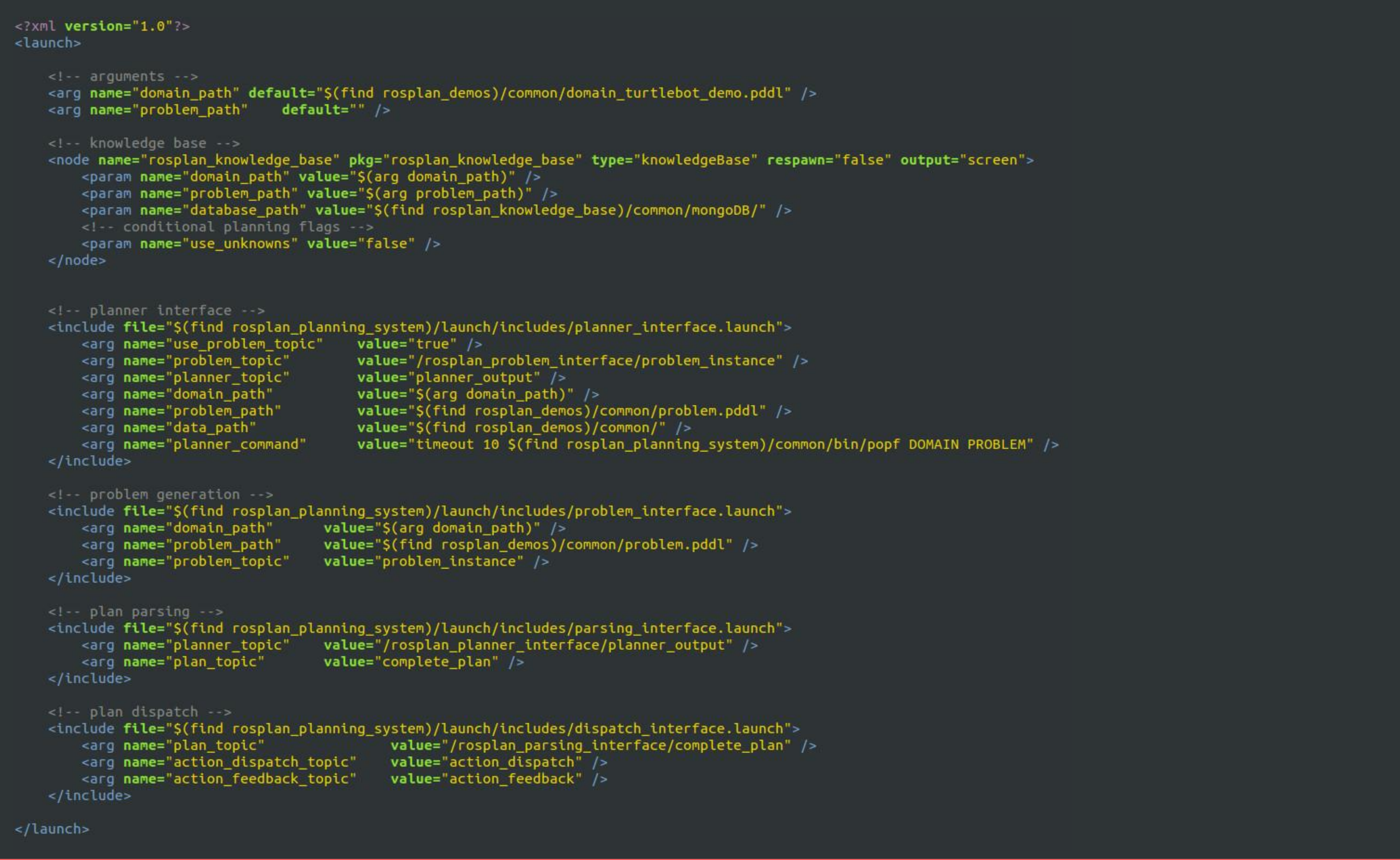


Node 5/5


domain/*
state/*
Problem Interface
problem instance
domain_path
problem_path
Planner Interface
Knowledge Base
planner output
domain/*
update
state/*
Parsing Interface
query_state
plan
mapping & localisation
query_state
Plan Dispatch
movebase
action feedback
action dispatch



```xml
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="node_name"        default="rosplan_problem_interface" />
    <arg name="domain_path"      default="$(find rosplan_demos)/common/domain_turtlebot_demo.pddl" />
    <arg name="problem_path"     default="$(find rosplan_demos)/common/problem.pddl" />
    <arg name="problem_topic"    default="problem_instance" />


    <!-- problem generation -->
    <node name="$(arg node_name)" pkg="rosplan_planning_system" type="problemInterface" respawn="false" output="screen">
        <param name="domain_path" value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <param name="problem_topic" value="$(arg problem_topic)" />
    </node>

</launch>
```



```xml
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path" default="$(find rosplan_demos)/common/domain_turtlebot_demo.pddl" />
    <arg name="problem_path"    default="" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path" value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <param name="database_path" value="$(find rosplan_knowledge_base)/common/mongoDB/" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>


    <!-- planner interface -->
    <include file="$(find rosplan_planning_system)/launch/includes/planner_interface.launch">
        <arg name="use_problem_topic"    value="true" />
        <arg name="problem_topic"        value="/rosplan_problem_interface/problem_instance" />
        <arg name="planner_topic"        value="planner_output" />
        <arg name="domain_path"          value="$(arg domain_path)" />
        <arg name="problem_path"         value="$(find rosplan_demos)/common/problem.pddl" />
        <arg name="data_path"            value="$(find rosplan_demos)/common/" />
        <arg name="planner_command"      value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
    </include>

    <!-- problem generation -->
    <include file="$(find rosplan_planning_system)/launch/includes/problem_interface.launch">
        <arg name="domain_path"      value="$(arg domain_path)" />
        <arg name="problem_path"     value="$(find rosplan_demos)/common/problem.pddl" />
        <arg name="problem_topic"    value="problem_instance" />
    </include>

    <!-- plan parsing -->
    <include file="$(find rosplan_planning_system)/launch/includes/parsing_interface.launch">
        <arg name="planner_topic"    value="/rosplan_planner_interface/planner_output" />
        <arg name="plan_topic"       value="complete_plan" />
    </include>

    <!-- plan dispatch -->
    <include file="$(find rosplan_planning_system)/launch/includes/dispatch_interface.launch">
        <arg name="plan_topic"               value="/rosplan_parsing_interface/complete_plan" />
        <arg name="action_dispatch_topic"    value="action_dispatch" />
        <arg name="action_feedback_topic"    value="action_feedback" />
    </include>

</launch>
```

Commands to launch ROSPlan

First, source the workspace:

```
> source devel/setup.bash
```

Then, use the launch file:

```
> roslaunch [package_name] [launch_file_name]
```

For example:

```
> roslaunch rosplan_demos simulated_actions.launch
```

Calling Services and using ROSPlan

Introspection Graph

While ROSPlan is running, first source the workspace:

```
> source devel/setup.bash
```

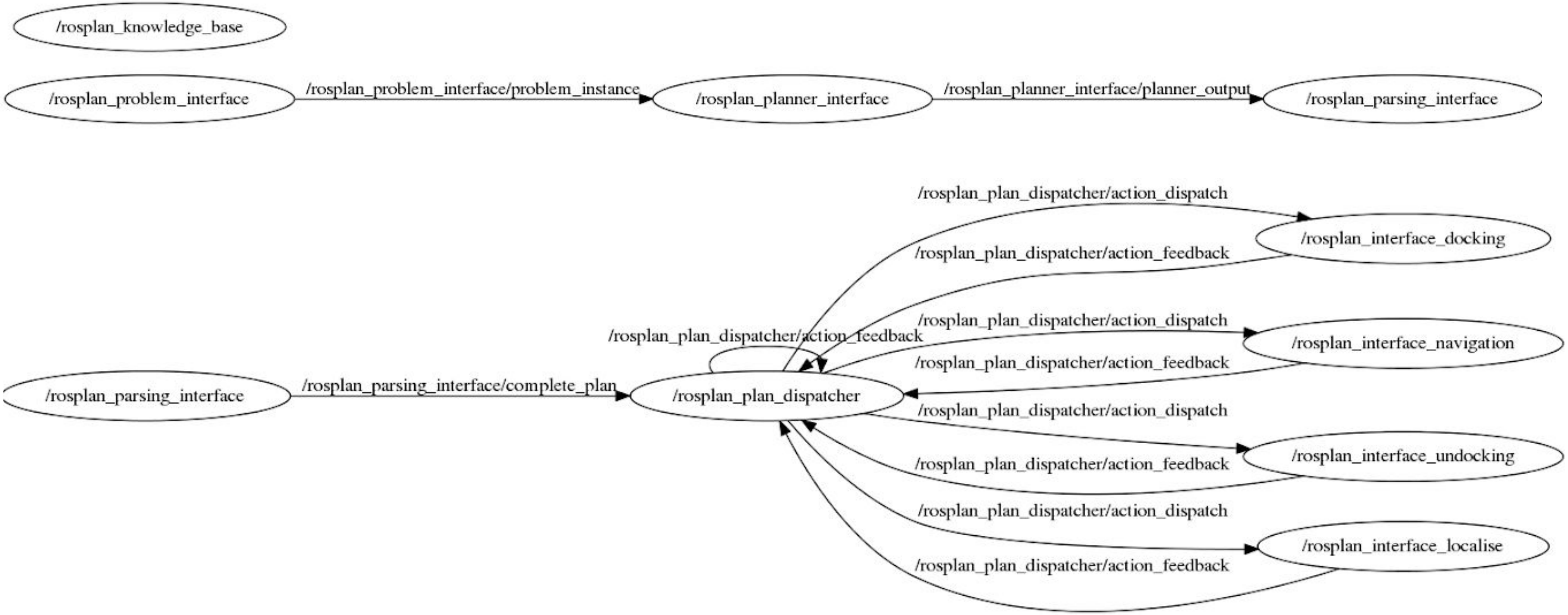
Then, run **RQT:**

```
> rqt
```

In RQT, open the plugin for viewing the node graph:

Plugins →Introspection→Node Graph

Introspection Graph


/rosplan_knowledge_base
/rosplan_problem_interface
/rosplan_problem_interface/problem_instance
/rosplan_planner_interface
/rosplan_planner_interface/planner_output
/rosplan_parsing_interface
/rosplan_plan_dispatcher/action_dispatch
/rosplan_interface_docking
/rosplan_plan_dispatcher/action_feedback
/rosplan_plan_dispatcher/action_dispatch
/rosplan_plan_dispatcher/action_feedback
/rosplan_interface_navigation
/rosplan_plan_dispatcher/action_feedback
/rosplan_parsing_interface
/rosplan_parsing_interface/complete_plan
/rosplan_plan_dispatcher
/rosplan_plan_dispatcher/action_dispatch
/rosplan_interface_undocking
/rosplan_plan_dispatcher/action_feedback
/rosplan_plan_dispatcher/action_dispatch
/rosplan_interface_localise
/rosplan_plan_dispatcher/action_feedback

Calling a service

While ROSPlan is running, first source the workspace:

```
> source devel/setup.bash
```

Then, use the following command to call the *problem generation* service:

```
> rosservice call /rosplan_problem_interface/problem_generation_server
```

(use tab-complete!)

Look at the output on the ROSPlan launch.

Generating a Plan

Use the command:

```
rostopic echo [topic_name] -n 1 -p
```

To view the problem that was published.

Call another service to generate a plan and view it.

These commands might be helpful (and rqt):

- rosnode list
- rosnode info
- rosservice list
- rosservice call [service_name] [request]
- rostopic echo [topic_name] -n 1 -p

Calling a service

Plan, prepare the plan for dispatch, and execute the plan:

```
> rosservice call /rosplan_planner_interface/planning_server

> rostopic echo /rosplan_planner_interface/planner_output -n 1

> rosservice call /rosplan_parsing_interface/parse_plan

> rosservice call /rosplan_plan_dispatcher/dispatch_plan
```

Useful commands

For viewing messages, services, topics, and nodes:

```
> rossrv show [service_type]
> rosmsg show [message_type]
> rosnode list
> rostopic list
> rosservice list
```

For viewing more information on current nodes and topics:

```
> rosnode info [node_name]
> rostopic info [message_topic_name]
> rosservice info [service_topic_name]
```

For viewing the last message published on a topic:

```
> rostopic echo [message_topic_name] -n 1
```

Using Scripts

The services can be called from a script:

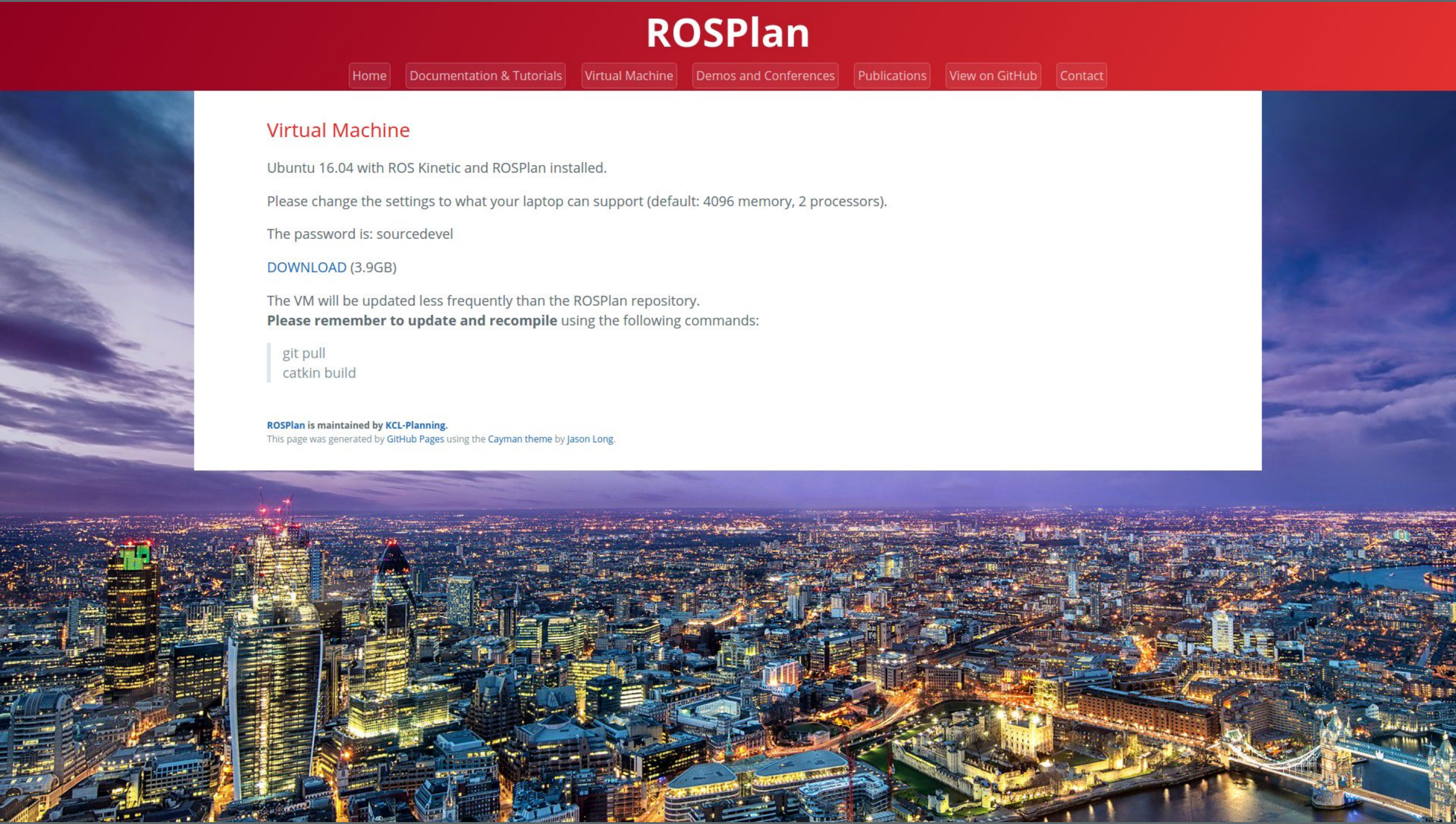
```
#!/bin/bash
rosservice call /rosplan_problem_interface/problem_generation_server;
rosservice call /rosplan_planner_interface/planning_server;
rosservice call /rosplan_parsing_interface/parse_plan;
rosservice call /rosplan_plan_dispatcher/dispatch_plan;
```

(Remember to source the workspace before trying to run the script!)

Run the script normally from a terminal, or using the *rosrun* command:

```
> ./plan_and_execute.bash
> rosrun rosplan_demos plan_and_execute.bash
```

Hands On



ROSPlan

Virtual Machine

Ubuntu 16.04 with ROS Kinetic and ROSPlan installed.

Please change the settings to what your laptop can support (default: 4096 memory, 2 processors).

The password is: sourcedevel

DOWNLOAD (3.9GB)

The VM will be updated less frequently than the ROSPlan repository.
Please remember to update and recompile using the following commands:

```
git pull
catkin build
```

ROSPlan is maintained by KCL-Planning.
This page was generated by GitHub Pages using the Cayman theme by Jason Long.

Set-up for this Week

Install ROSPlan via the instructions online.

```
> git clone https://github.com/KCL-Planning/rosplan
> git clone https://github.com/KCL-Planning/rosplan_demos
> git clone https://github.com/KCL-Planning/rosplan_interface_strategic
```

Rebuild everything and launch the nodes (from the workspace):

```
> catkin build
> source devel/setup.bash
> roslaunch rosplan_demos simulated_actions.launch
```

Set-up for this Week

```
> rosservice call /rosplan_problem_interface/problem_generation_server
```

Set-up for this Week

```
> rostopic echo /rosplan_problem_interface/problem_instance -p -n 1
```

Set-up for this Week

```
> rostopic echo /rosplan_problem_interface/problem_instance -p -n 1
```

```
(define (problem task)
(:domain turtlebot)
(:objects
    wp0 wp1 wp2 wp3 wp4 - waypoint
    kenny - robot
)
(:init
    (robot_at kenny wp0)
    (docked kenny)
    (dock_at wp0)
)
(:goal (and
    (visited wp0)
    (visited wp1)
    (visited wp2)
    (visited wp3)
    (visited wp4)
    (docked kenny)
)))
```

```
(define (domain turtlebot_demo)

(:requirements :strips :typing :fluents :disjunctive-preconditions :durative-actions)

(:types
        waypoint
        robot
)

(:predicates
        (robot_at ?v - robot ?wp - waypoint)
        (connected ?from ?to - waypoint)
        (visited ?wp - waypoint)
)

(:functions
        (distance ?wp1 ?wp2 - waypoint)
)

;; Move between any two waypoints, avoiding terrain
(:durative-action goto_waypoint
        :parameters (?v - robot ?from ?to - waypoint)
        :duration ( = ?duration 10)
        :condition (and
                (at start (robot_at ?v ?from)))
        :effect (and
                (at end (visited ?to))
                (at start (not (robot_at ?v ?from)))
                (at end (robot_at ?v ?to)))
)
)
```

```
(define (domain turtlebot)

(:requirements :strips :typing :fluents :durative-actions :number-fluents)

(:types
        waypoint robot - object
        printer - waypoint
)

(:functions
        (bailout_distance ?a - waypoint)
        (distance ?a ?b - waypoint)
        (papers_delivered ?r - robot ?w - waypoint)
)

(:predicates

        (bailout_available)
        (bailout_location ?to - waypoint)

        (robot_at ?v - robot ?wp - waypoint)
        (undocked ?v - robot)
        (docked ?v - robot)
        (localised ?v - robot)
        (dock_at ?wp - waypoint)

        ;; Printing
        (carrying_papers ?r - robot)
        (nocarrying_papers ?r - robot)
        (asked_load ?r - robot)
        (asked_unload ?r - robot)
        (delivery_destination ?w - waypoint)
)

;; Bailout move
(:durative-action bailout
        :parameters (?v - robot ?to - waypoint)
        :duration ( = ?duration (+ 100 (* 5 (bailout_distance ?to))))
        :condition (and
                (at start (bailout_location ?to))
                (at start (bailout_available))
                (at start (localised ?v))
                (over all (undocked ?v))
                )
        :effect (and
                (at start (not (bailout_available)))
                (at end (not (asked_load ?v)))
                (at end (not (asked_unload ?v)))
                (at end (robot_at ?v ?to))
                )
)
```

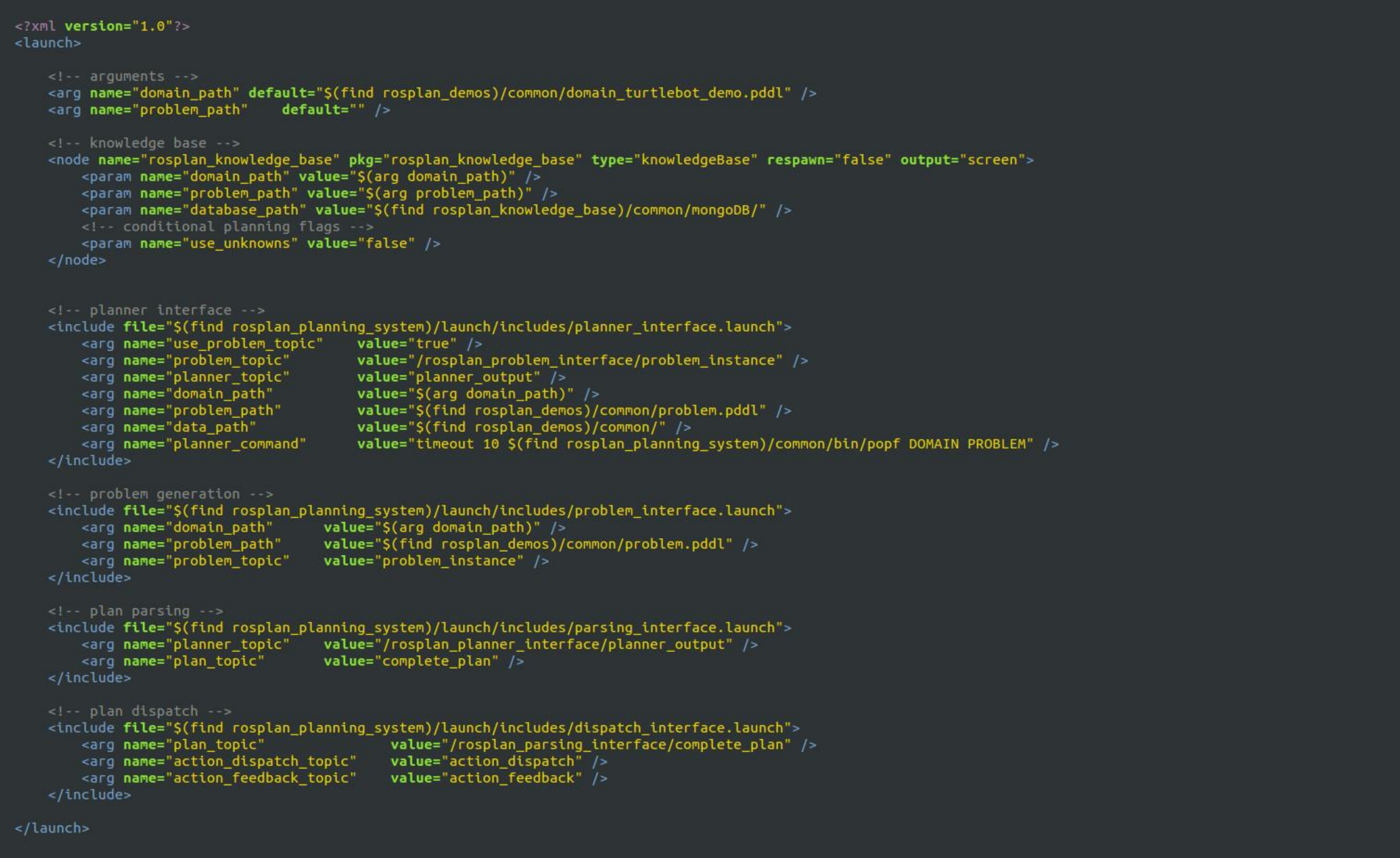
```
;; Move to any waypoint, avoiding terrain
(:durative-action goto_waypoint
        :parameters (?v - robot ?from ?to - waypoint)
        :duration ( = ?duration (* 5 (distance ?from ?to)))
        :condition (and
                (at start (robot_at ?v ?from))
                (at start (localised ?v))
                (over all (undocked ?v))
                )
        :effect (and
                (at start (not (robot_at ?v ?from)))
                (at end (not (asked_load ?v)))
                (at end (not (asked_unload ?v)))
                (at end (robot_at ?v ?to))
                )
)

;; Localise
(:durative-action localise
        :parameters (?v - robot)
        :duration ( = ?duration 60)
        :condition (over all (undocked ?v))
        :effect (at end (localised ?v))
)


;; Dock to charge
(:durative-action dock
        :parameters (?v - robot ?wp - waypoint)
        :duration ( = ?duration 30)
        :condition (and
                (over all (dock_at ?wp))
                (at start (robot_at ?v ?wp))
                (at start (undocked ?v)))
        :effect (and
                (at end (docked ?v))
                (at start (not (undocked ?v))))
)

(:durative-action undock
        :parameters (?v - robot ?wp - waypoint)
        :duration ( = ?duration 10)
        :condition (and
                (over all (dock_at ?wp))
                (at start (docked ?v)))
        :effect (and
                (at start (not (docked ?v)))
                (at end (undocked ?v)))
)
```

```
(:durative-action ask_load
        :parameters (?r - robot ?p - printer)
        :duration ( = ?duration 5)
        :condition (and
                (over all (nocarrying_papers ?r))
                (over all (robot_at ?r ?p))
                )
        :effect (and
                (at end (asked_load ?r))
                )
)

(:durative-action ask_unload
        :parameters (?r - robot ?w - waypoint)
        :duration ( = ?duration 5)
        :condition (and
                (over all (carrying_papers ?r))
                (over all (robot_at ?r ?w))
                (over all (delivery_destination ?w))
                )
        :effect (and
                (at end (asked_unload ?r))
                )
)

(:durative-action wait_load
        :parameters (?r - robot ?p - printer)
        :duration ( = ?duration 15)
        :condition (and
                (at start (asked_load ?r))
                (at start (nocarrying_papers ?r))
                (over all (robot_at ?r ?p))
                )
        :effect (and
                (at end (carrying_papers ?r))
                (at end (not (nocarrying_papers ?r)))
                )
)

(:durative-action wait_unload
        :parameters (?r - robot ?w - waypoint)
        :duration ( = ?duration 15)
        :condition (and
                (at start (asked_unload ?r))
                (at start (carrying_papers ?r))
                (at start (delivery_destination ?w))
                (over all (robot_at ?r ?w))
                )
        :effect (and
                (at start (not (carrying_papers ?r)))
                (at end (nocarrying_papers ?r))
                (at end (increase (papers_delivered ?r ?w) 1))
                )
)

)
```

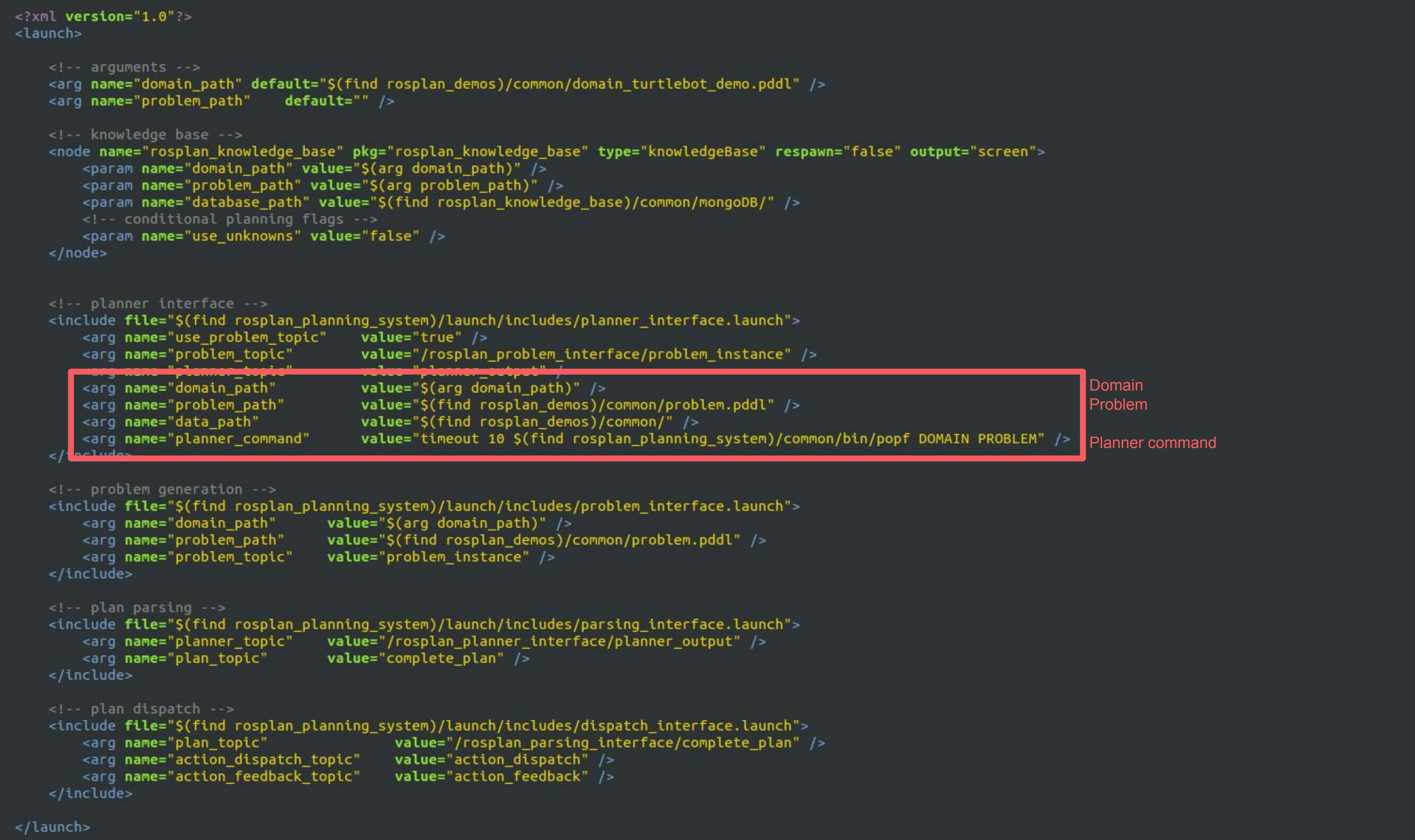
Using the planner

On the virtual box, and in ROSPlan is the planner POPF:

```
> rosrun rosplan_planning_system popf
```



```
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path" default="$(find rosplan_demos)/common/domain_turtlebot_demo.pddl" />
    <arg name="problem_path"    default="" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path" value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <param name="database_path" value="$(find rosplan_knowledge_base)/common/mongoDB/" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>


    <!-- planner interface -->
    <include file="$(find rosplan_planning_system)/launch/includes/planner_interface.launch">
        <arg name="use_problem_topic"    value="true" />
        <arg name="problem_topic"        value="/rosplan_problem_interface/problem_instance" />
        <arg name="planner_topic"        value="planner_output" />
        <arg name="domain_path"          value="$(arg domain_path)" />
        <arg name="problem_path"         value="$(find rosplan_demos)/common/problem.pddl" />
        <arg name="data_path"            value="$(find rosplan_demos)/common/" />
        <arg name="planner_command"      value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
    </include>

    <!-- problem generation -->
    <include file="$(find rosplan_planning_system)/launch/includes/problem_interface.launch">
        <arg name="domain_path"      value="$(arg domain_path)" />
        <arg name="problem_path"     value="$(find rosplan_demos)/common/problem.pddl" />
        <arg name="problem_topic"    value="problem_instance" />
    </include>

    <!-- plan parsing -->
    <include file="$(find rosplan_planning_system)/launch/includes/parsing_interface.launch">
        <arg name="planner_topic"    value="/rosplan_planner_interface/planner_output" />
        <arg name="plan_topic"       value="complete_plan" />
    </include>

    <!-- plan dispatch -->
    <include file="$(find rosplan_planning_system)/launch/includes/dispatch_interface.launch">
        <arg name="plan_topic"               value="/rosplan_parsing_interface/complete_plan" />
        <arg name="action_dispatch_topic"    value="action_dispatch" />
        <arg name="action_feedback_topic"    value="action_feedback" />
    </include>

</launch>
```



```
<?xml version="1.0"?>
<launch>

    <!-- arguments -->
    <arg name="domain_path" default="$(find rosplan_demos)/common/domain_turtlebot_demo.pddl" />
    <arg name="problem_path"    default="" />

    <!-- knowledge base -->
    <node name="rosplan_knowledge_base" pkg="rosplan_knowledge_base" type="knowledgeBase" respawn="false" output="screen">
        <param name="domain_path" value="$(arg domain_path)" />
        <param name="problem_path" value="$(arg problem_path)" />
        <param name="database_path" value="$(find rosplan_knowledge_base)/common/mongoDB/" />
        <!-- conditional planning flags -->
        <param name="use_unknowns" value="false" />
    </node>


    <!-- planner interface -->
    <include file="$(find rosplan_planning_system)/launch/includes/planner_interface.launch">
        <arg name="use_problem_topic"    value="true" />
        <arg name="problem_topic"        value="/rosplan_problem_interface/problem_instance" />
        <arg name="planner_topic"        value="planner_output" />
        <arg name="domain_path"          value="$(arg domain_path)" />
        <arg name="problem_path"         value="$(find rosplan_demos)/common/problem.pddl" />
        <arg name="data_path"            value="$(find rosplan_demos)/common/" />
        <arg name="planner_command"      value="timeout 10 $(find rosplan_planning_system)/common/bin/popf DOMAIN PROBLEM" />
    </include>

    <!-- problem generation -->
    <include file="$(find rosplan_planning_system)/launch/includes/problem_interface.launch">
        <arg name="domain_path"     value="$(arg domain_path)" />
        <arg name="problem_path"    value="$(find rosplan_demos)/common/problem.pddl" />
        <arg name="problem_topic"    value="problem_instance" />
    </include>

    <!-- plan parsing -->
    <include file="$(find rosplan_planning_system)/launch/includes/parsing_interface.launch">
        <arg name="planner_topic"    value="/rosplan_planner_interface/planner_output" />
        <arg name="plan_topic"       value="complete_plan" />
    </include>

    <!-- plan dispatch -->
    <include file="$(find rosplan_planning_system)/launch/includes/dispatch_interface.launch">
        <arg name="plan_topic"               value="/rosplan_parsing_interface/complete_plan" />
        <arg name="action_dispatch_topic"    value="action_dispatch" />
        <arg name="action_feedback_topic"    value="action_feedback" />
    </include>

</launch>
```

Domain
Problem

Planner command

Calling the Services

Plan, prepare the plan for dispatch, and execute the plan:

```
> rosservice call /rosplan_planner_interface/planning_server

> rostopic echo /rosplan_planner_interface/planner_output -n 1 -p

> rosservice call /rosplan_parsing_interface/parse_plan

> rosservice call /rosplan_plan_dispatcher/dispatch_plan
```

In a third terminal:

```
> rqt --standalone rosplan_rqt.esterel_plan_viewer.ROSPlanEsterelPlanViewer
```

Part 5:
Turtlebot Simulation

Gazebo and rviz

Gazebo is a simulation environment for robotics in ROS.

Gazebo simulates the physics of the robot and the world, and displays a 3D model of everything.

rviz is a visualisation program for ROS systems.

rviz displays data that is published on ROS topics, such as sensor data and camera images.

rviz can be used with the real robot as well as the gazebo simulation.

Stage and rviz

Stage is another simulation environment for robotics in ROS.

It is less demanding that Gazebo.

Launching the simulation

A launch file for ROSPlan and a simulated environment already exists:

```
> roslaunch rosplan_demos turtlebot_redone.launch
```

And a script to call the services:

```
> rosrun rosplan_demos turtlebot_explore.bash
```

Exercise

Modify the demo domain and problem files:

1. The robot must perform a visual inspection at each waypoint. The inspection takes 60 seconds.
2. The robot must recharge after every 2 inspections.

Modify the example launch file to include simulated action nodes for any new operators you have added to the domain.

Run the modified demo.