ROS (part II)

ROBOTICS

POLITECNICO
MILANO 1863

Schedule V. 2.0

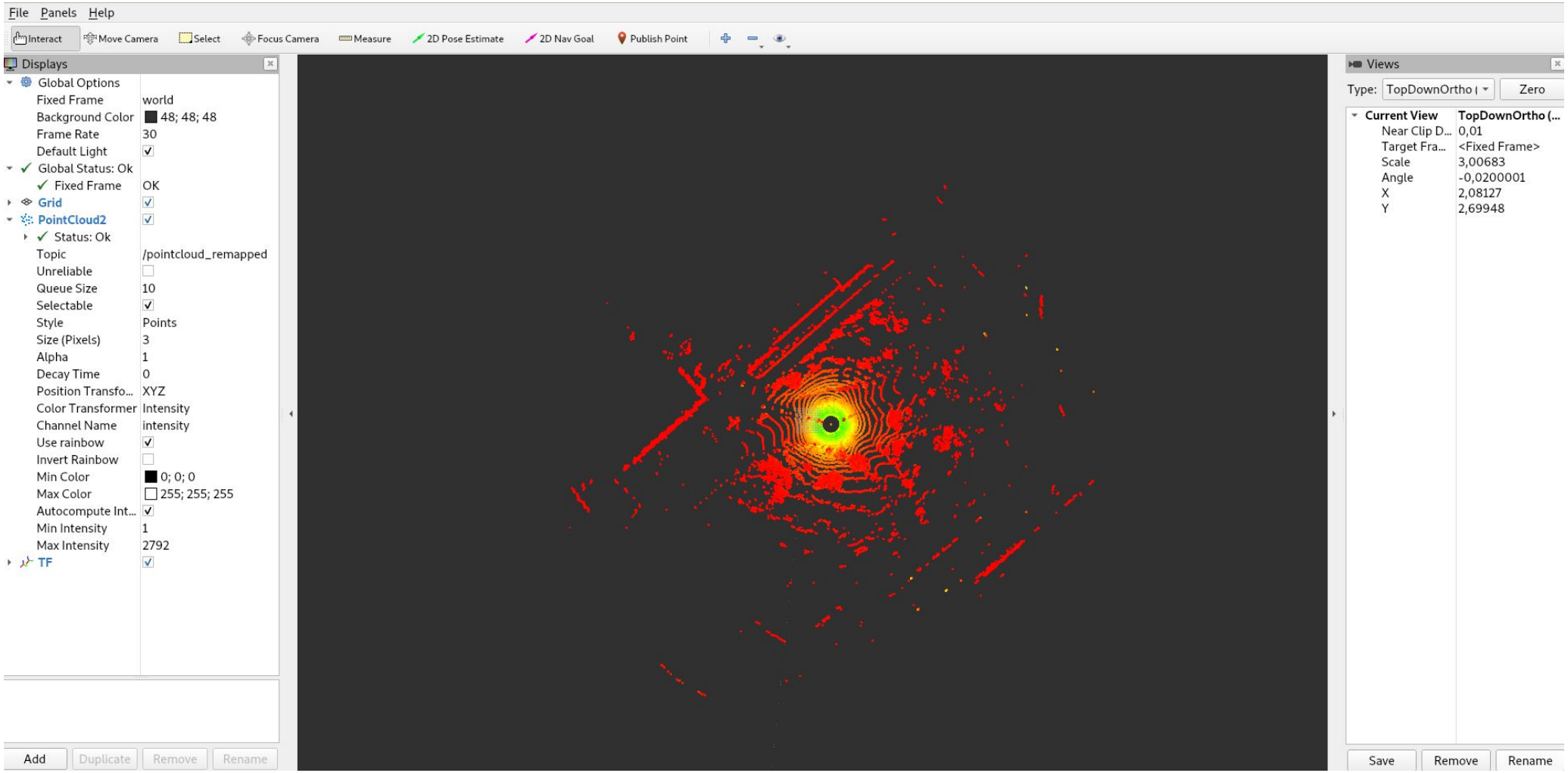
L1	Middleware for robotics and ROS Installation Party	L6	message filter,actionlib, rosbag tools
L2	Ros workspace, publisher/subscriber	L7	ROS on multiple machines, time synchronization, latched pub,async spinner
L3	Publisher, subscriber, launch file , custom messages	L8	Robot Navigation, Stage, Gmapping
L4	Services, parameters, timers	L9	Robot Navigation (Part II), Robot Localization, mapviz
L5	TF, parameters, first project	L10	ROS2, second project

First Robotics Project (some comments)

ROBOTICS

POLITECNICO
MILANO 1863

What we expected



Private namespaces

```
ros::NodeHandle n; // this is a global node handle

ros::NodeHandle nh_private("~"); // this is a private node handle
```

Initialization

```
void callback(const nav_msgs::Odometry::ConstPtr& msg){
  tf::Transform transform;
  transform.setOrigin( tf::Vector3(msg->pose.pose.position.x, msg->pose.pose.position.y,
msg->pose.pose.position.z) );
  tf::Quaternion q(msg->pose.pose.orientation.x, msg->pose.pose.orientation.y, msg->pose.pose.orientation.z,
msg->pose.pose.orientation.w);
  transform.setRotation(q);
  tf::TransformBroadcaster br;
  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), root_frame, child_frame));
}
```

Retrieve parameters

```
void callback(const nav_msgs::Odometry::ConstPtr& msg){

  tf::Transform transform;

  transform.setOrigin( tf::Vector3(msg->pose.pose.position.x, msg->pose.pose.position.y,
msg->pose.pose.position.z) );

  tf::Quaternion q(msg->pose.pose.orientation.x, msg->pose.pose.orientation.y, msg->pose.pose.orientation.z,
msg->pose.pose.orientation.w);

  transform.setRotation(q);

  std::string root_frame;

  n.getParam("root_frame", root_frame);

  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), root_frame, child_frame));

}
```

MESSAGE FILTERS

ROBOTICS

POLITECNICO
MILANO 1863

MESSAGE FILTERS

Useful to synchronize multiple topics

Need topics with header and timestamp

Can synchronize with exact time or approximate time

Camera topics synchronization has a custom version

MESSAGE FILTERS (without policy)

```
message_filters::Subscriber<geometry_msgs::Vector3Stamped> sub1(n, "topic1",
1);
  message_filters::Subscriber<geometry_msgs::Vector3Stamped> sub2(n, "topic2",
1);
  message_filters::TimeSynchronizer<geometry_msgs::Vector3Stamped,
geometry_msgs::Vector3Stamped> sync(sub1, sub2, 10);
  sync.registerCallback(boost::bind(&callback, _1, _2));
```

Create the subscriber

Create the time synchronizer

Bind it with the callback

MESSAGE FILTERS (with policy)

```
typedef
message_filters::sync_policies::ExactTime<geometry_msgs::Vector3Stamped,
geometry_msgs::Vector3Stamped> MySyncPolicy;
```

Create the policy

```
message_filters::Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), sub1, sub2);
  sync.registerCallback(boost::bind(&callback, _1, _2));
```

Bind it with the callback

Create the time synchronizer with the policy

ACTIONLIB

ROBOTICS

POLITECNICO
MILANO 1863

WHAT IS ACTIONLIB

Node A sends a request to node B to perform some task

Service	**Action**
Small execution time	Long execution time
Requesting node can wait	Requesting node cannot wait
No status	Status monitoring
No cancellation	Cancellation

WHAT IS ACTIONLIB

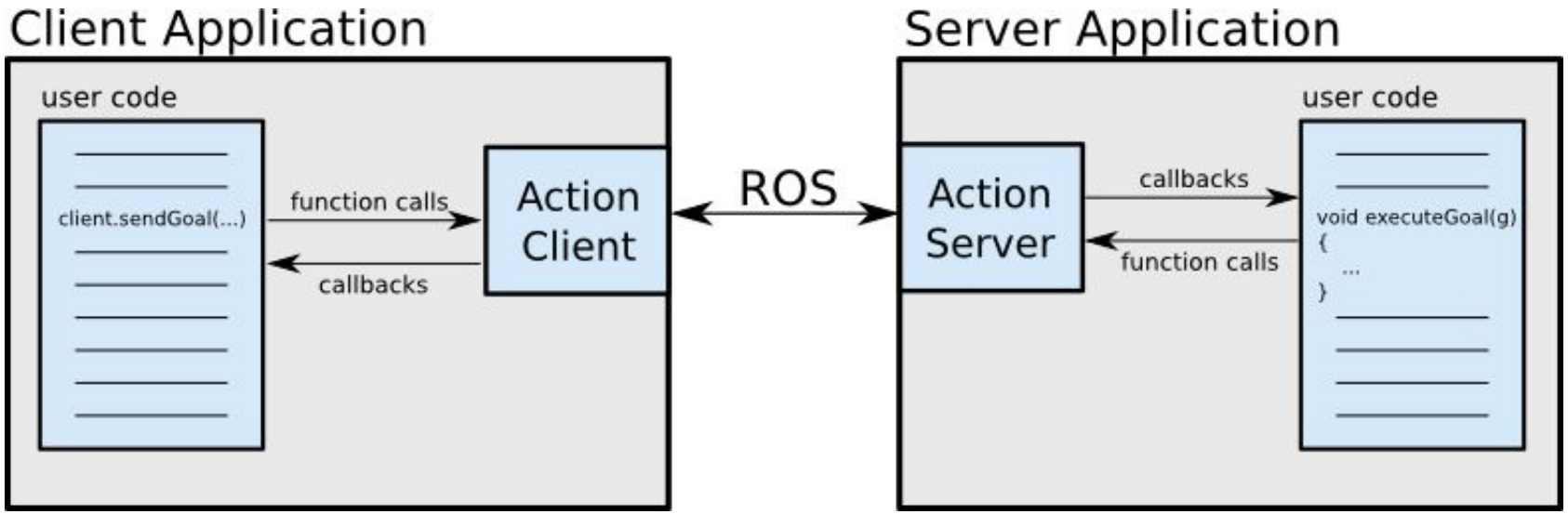
actionlib package is:

- sort of ROS implementation of threads
- based on a client/server paradigm

And provides tools to:

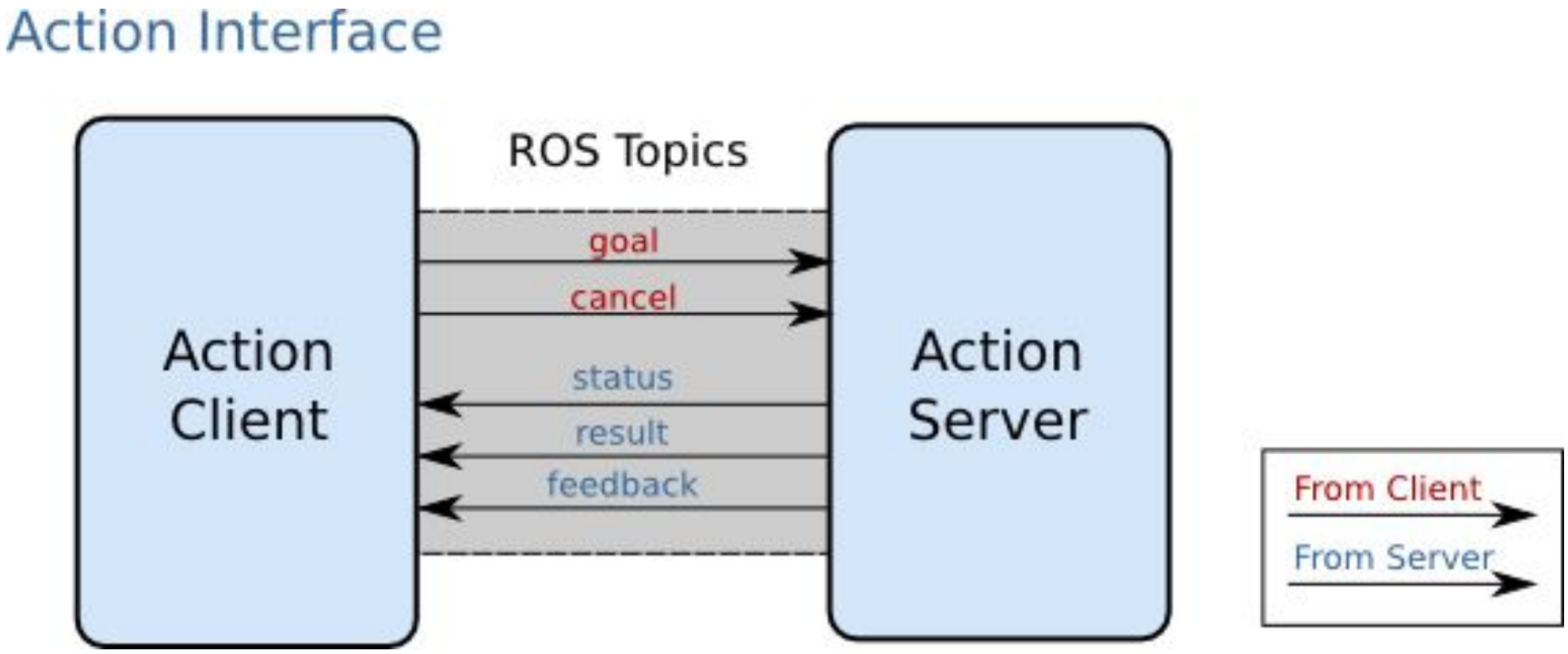
- create servers that execute long-running tasks (that can be preempted).
- create clients that interact with servers

WHAT IS ACTIONLIB



The ActionClient and ActionServer communicate via a "ROS Action Protocol", which is built on top of ROS messages

CLIENT-SERVER INTERACTION



CLIENT-SERVER INTERACTION

goal: to send new goals to server

cancel: to send cancel requests to server

status: to notify clients on the current state of every goal in the system.

feedback: to send clients periodic auxiliary information for a goal

result: to send clients one-time auxiliary information upon completion of a goal

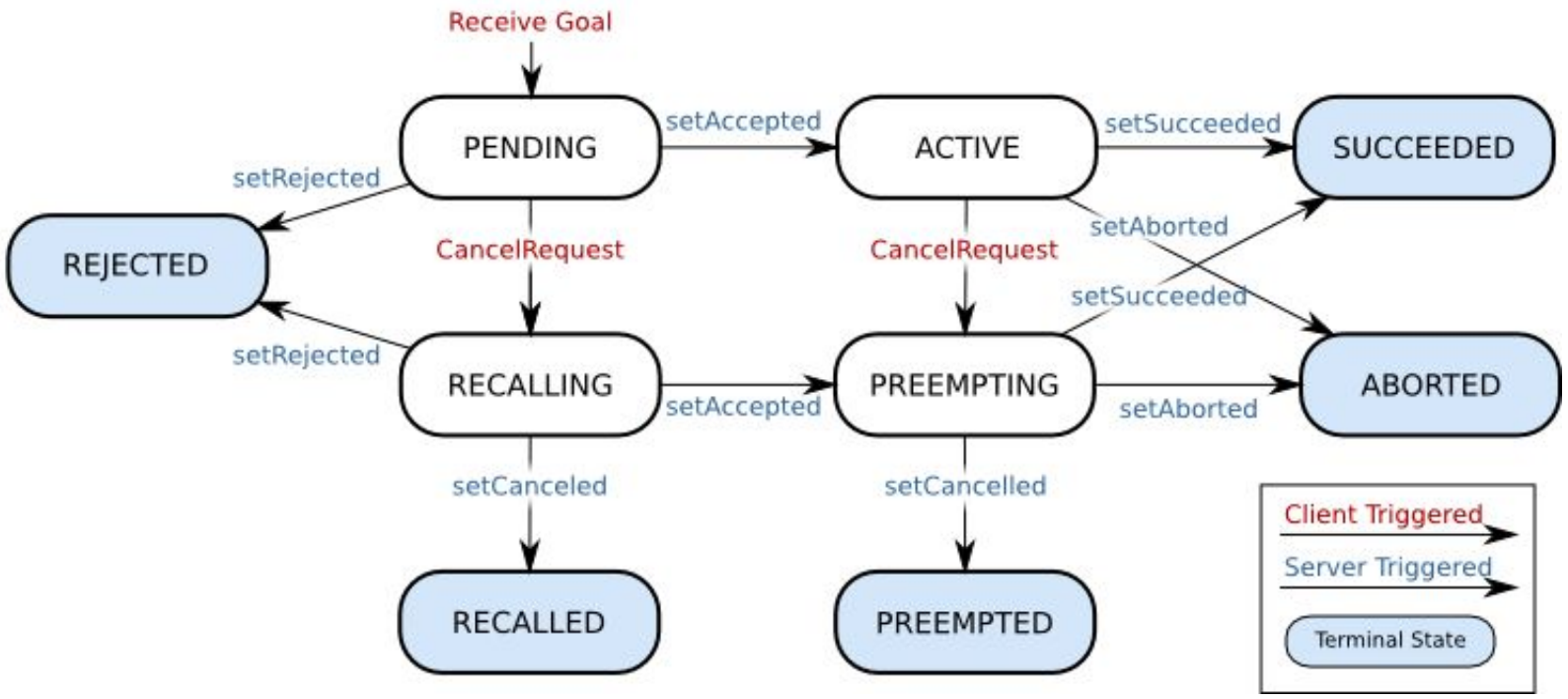
ACTION AND GOAL ID

Action templates are defined by a name and some additional properties through an .action structure defined in ROS

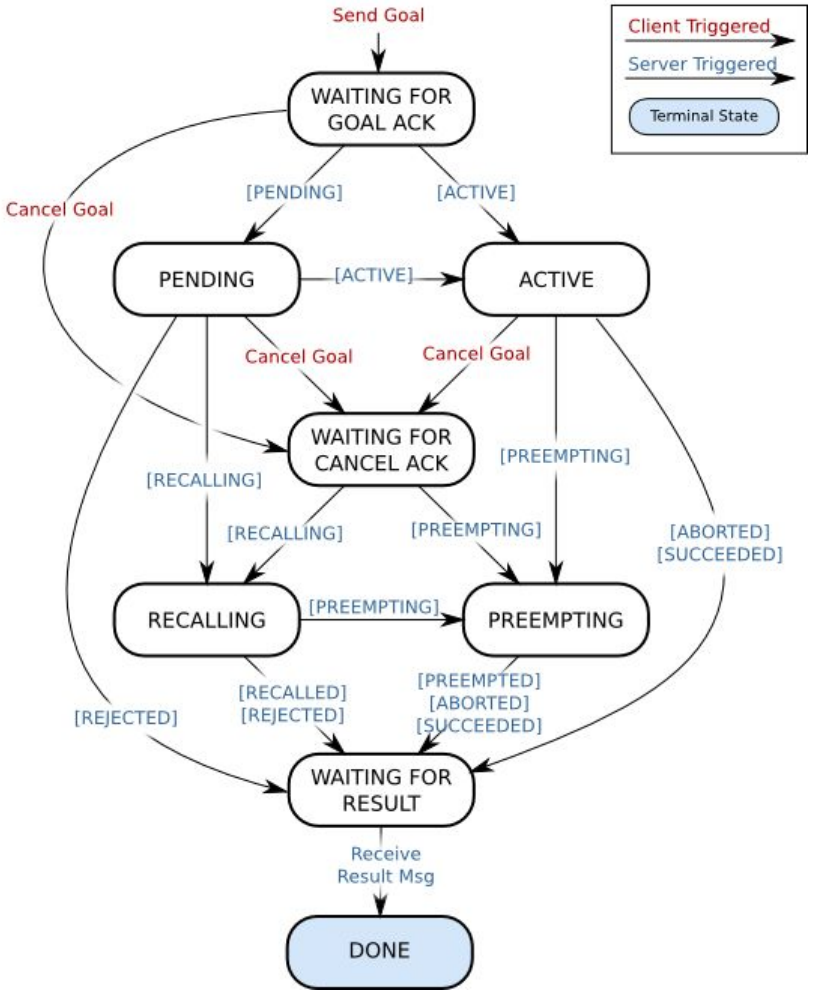
Each ***instance*** of an action has a unique Goal ID

Goal ID provides the action server and the action client with a robust way to monitor the execution of a particular instance of an action.

SERVER STATE MACHINE



CLIENT STATE MACHINE



.ACTION EXAMPLE

```
# Define the goal
uint32 dishwasher_id  # Specify the dishwasher id
---
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

SIMPLEACTIONSERVER

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "do_dishes_server");
  ros::NodeHandle n;
  Server server(n, "do_dishes", boost::bind(&exe, _1, &server), false);
  server.start();
  ros::spin();
  return 0;
}
```

SIMPLEACTIONSERVER

```
void exe(const chores::DoDishesGoalConstPtr& goal, Server* as) {
  while(allClean()) {
    doDishes(goal->dishwasher_id)
    if(as->isPreemptRequested() || !ros::ok()) {
      as->setPreempted();
      break;
    }
    as->publishFeedback(currentWork(goal->dishwasher_id))
  }
  if(currentWork(goal->dishwasher_id) == 100)
    as->setSucceeded();
}
```

SIMPLEACTIONCLIENT

```
#include <chores/DoDishesAction.h>
#include <actionlib/client/simple_action_client.h>

typedef actionlib::SimpleActionClient<chores::DoDishesAction> Client;
```

SIMPLEACTIONCLIENT

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "do_dishes_client");
  Client client("do_dishes", true); // true -> don't need ros::spin()
  client.waitForServer();
  chores::DoDishesGoal goal;
  //set goal parameters
  goal.dishwasher_id = pickDishwasher();
```

SIMPLEACTIONCLIENT

```
  client.sendGoal(goal);
  client.waitForResult(ros::Duration(5.0));
  if (client.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    ROS_INFO("Yay! The dishes are now clean");
  std::string state = client.getState().toString();
  ROS_INFO("Current State: %s\n", state.c_str());
  return 0;
}
```

TESTING

Copy the actionlib_tutorial folder inside the src folder of your catkin workspace and compile it

To start the server:

$ rosrun actionlib_tutorials fibonacci_server

The client has some parameters that can be set in the launch file, order and duration; after setting those parameters call:

$ roslaunch actionlib_tutorials launcher.launch

TESTING

You can monitor the server status simply using topics:

$ rostopic list

To get the feedback from the server:

$ rostopic echo /fibonacci/feedback

ROSBAG tools

ROBOTICS

POLITECNICO
MILANO 1863

ROSBAG tools

Perform operation directly on bags

Convert bags, extract data from bags, edit bags

pyhon api/c++ api

Bag to csv

```
import rosbag
import csv
from turtlesim.msg import Pose

bag = rosbag.Bag('/home/airlab/robotics/bags/pose.bag')

f = open('/home/airlab/robotics/bags/pose.csv', 'w')
writer = csv.writer(f)

header = ['timestamp', 'x','y','theta' ]
writer.writerow(header)
```

include for rosbag, csv, msg

Open the bag

Open the log file

Create the csv writer

Write header

Bag to csv

```
for topic, msg, t in bag.read_messages(topics=['/turtle1/pose']):
    writer.writerow ([t,msg.x, msg.y,msg.theta])
bag.close()
f.close()
```

Iterate on msg

Write to csv

Bag editor

```
import rosbag
import csv
from turtlesim.msg import Pose

bag_in = rosbag.Bag('/home/airlab/robotics/bags/pose.bag')
bag_out = rosbag.Bag('/home/airlab/robotics/bags/pose_shifted.bag', 'w')
```

include for rosbag, csv, msg

Open the input bag

Open the output bag

Bag editor

```
for topic, msg, t in bag_in.read_messages(topics=['/turtle1/pose']):
    msg.x +=2
    t.secs += 30
    bag_out.write ('/turtleS/pose',msg, t)
bag_in.close()
bag_out.close()
```

Iterate on msg

Write to out bag