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

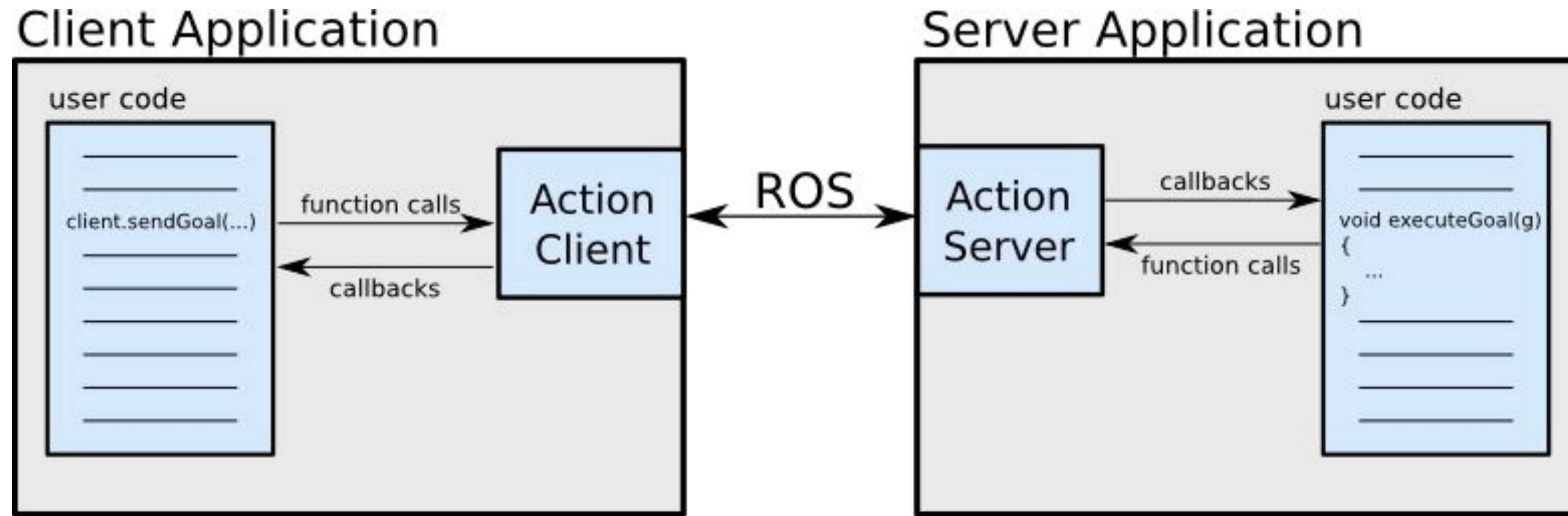
actionlib package is:

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

And provides tools to:

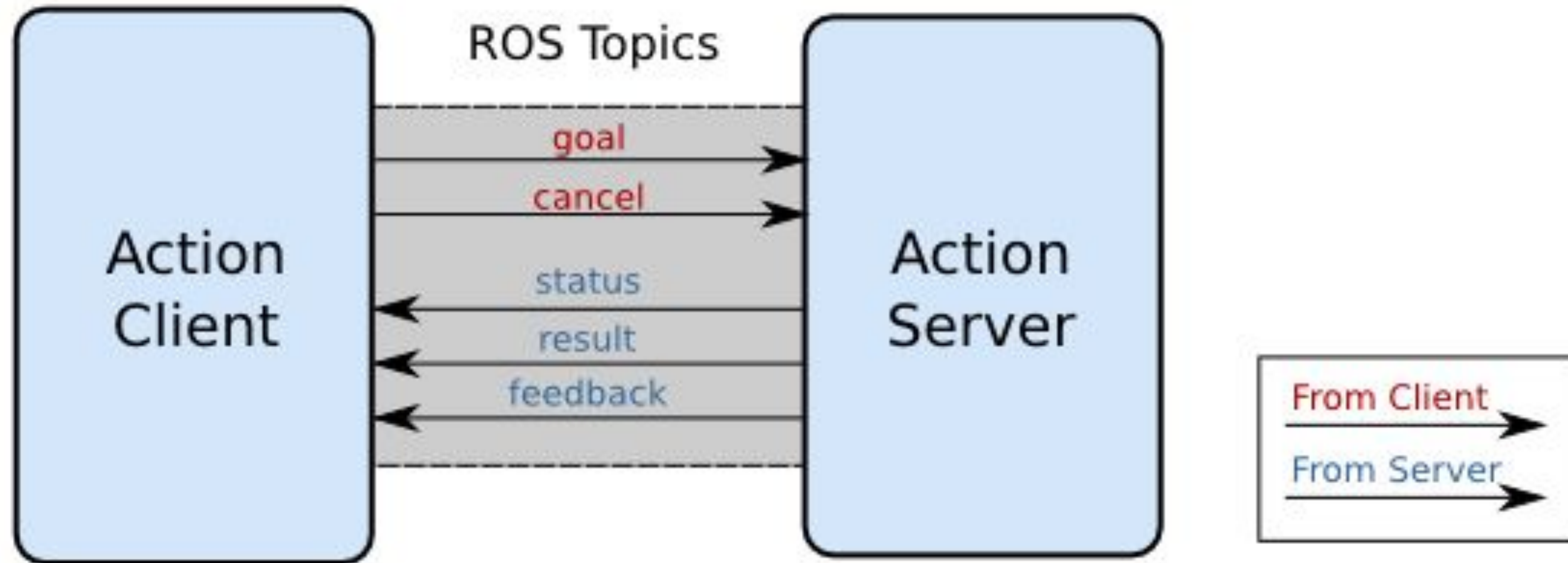
- create servers that execute long-running tasks (that can be preempted).
- create clients that interact with servers

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

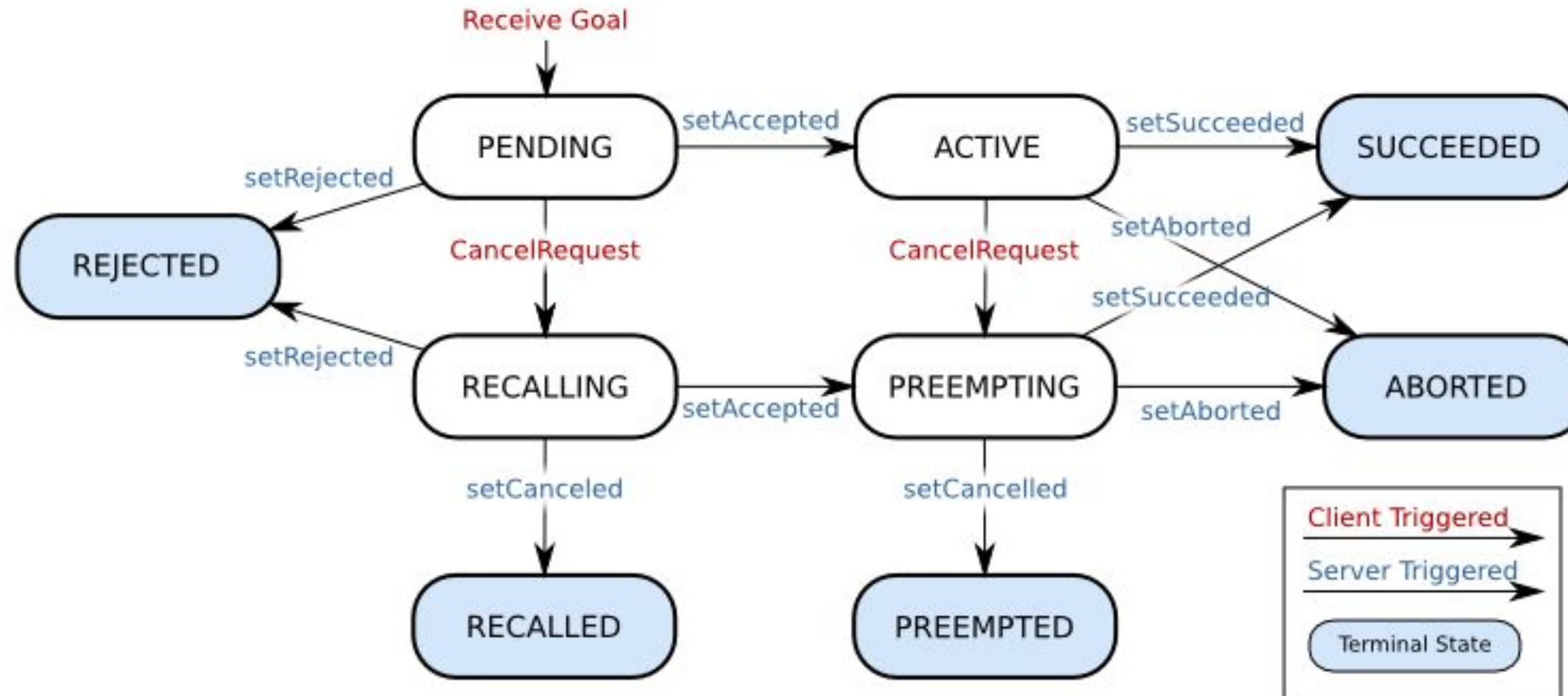
ACTION AND GOAL ID

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

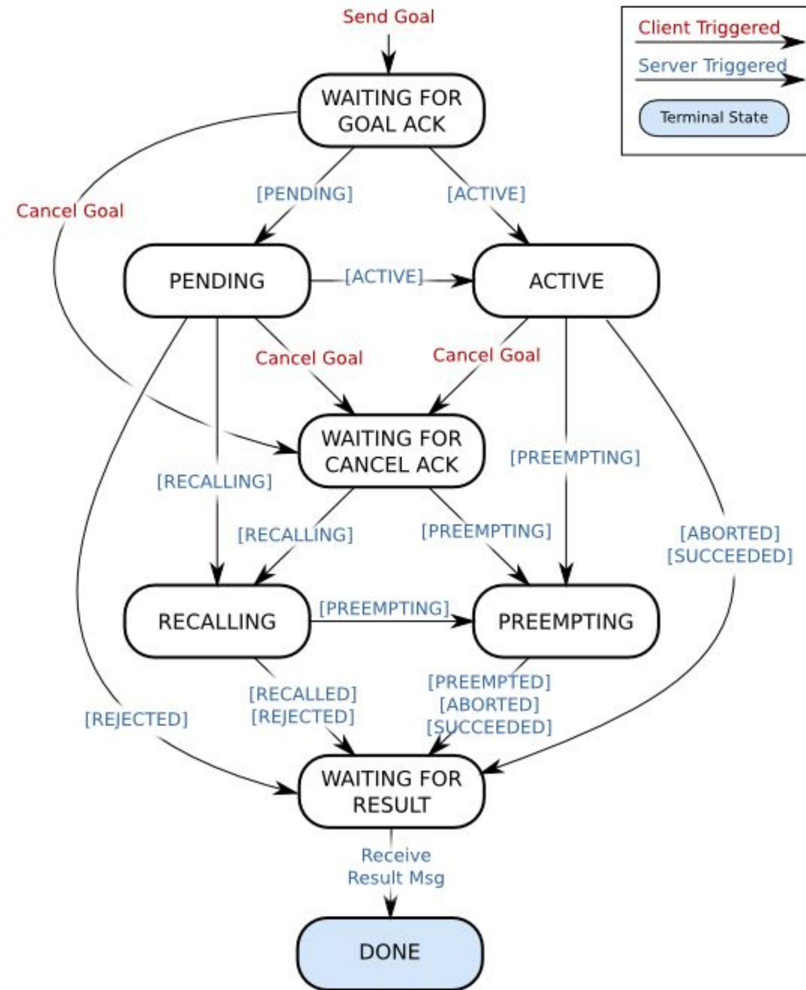
Each ***instance*** of an action has a unique Goal ID

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

CLIENT

```
#include <ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include <actionlib_tutorials/FibonacciAction.h>
```

Some standard include, plus include the header file for our custom actionlib like we did with custom messages and services

CLIENT

```
nt main (int argc, char **argv)
{
  ros::init(argc, argv, "test_fibonacci");
  actionlib::SimpleActionClient<actionlib_tutorials::FibonacciAction>
ac("fibonacci", true);
```

Create the client, automatically create a new thread to handle it

```
  ROS_INFO("Waiting for action server to start.");


  ac.waitForServer();
```

wait for the Fibonacci server to be up

CLIENT

```
ROS_INFO("Action server started, sending goal.");
 // send a goal to the action
 actionlib_tutorials::FibonacciGoal goal;
 int order =10;
 double duration =1.0;
 ros::param::get("order",order);
 ros::param::get("duration",duration);
 goal.order = order;
 ac.sendGoal(goal);
```

Create a goal for the server

Check if we have a parameter with order or duration value

Assign to the goal the value and send it

CLIENT

```
bool finished_before_timeout = ac.waitForResult(ros::Duration(duration));

  if (finished_before_timeout)
  {
    actionlib::SimpleClientGoalState state = ac.getState();
    ROS_INFO("Action finished: %s",state.toString().c_str());

  }
  else
    ROS_INFO("Action did not finish before the time out.");

  return 0;
}
```

Wait for the goal to be completed

Check if the goal was completed in time

CLIENT

```
#include <ros/ros.h>

#include <actionlib/server/simple_action_server.h>

#include <actionlib_tutorials/FibonacciAction.h>
```

Some standard include, plus include the header file for our custom actionlib like we did with custom messages and services

CLIENT

```
int main(int argc, char** argv)
{
  ros::init(argc, argv, "fibonacci");

  FibonacciAction fibonacci("fibonacci");
  ros::spin();

  return 0;
}
```

init the node

Create the action and spin

CLIENT

```
ros::NodeHandle nh_;
  actionlib::SimpleActionServer<actionlib_tutorials::FibonacciAction> as_;

  std::string action_name_;
  actionlib_tutorials::FibonacciFeedback feedback_;
  actionlib_tutorials::FibonacciResult result_;
```

Create the Node handle

create the server

Some messages for the server

CLIENT

```
FibonacciAction(std::string name) :
   as_(nh_, name, boost::bind(&FibonacciAction::executeCB, this, _1), false),
   action_name_(name)
  {
   as_.start();
  }

  ~FibonacciAction(void)
  {
  }
```

create the server

Start the server

CLIENT

```
void executeCB(const actionlib_tutorials::FibonacciGoalConstPtr &goal)
  {
    ros::Rate r(1);
    bool success = true;

    feedback_.sequence.clear();
    feedback_.sequence.push_back(0);
    feedback_.sequence.push_back(1);

    ROS_INFO("%s: Executing, creating fibonacci sequence of order %i with seeds
%i, %i", action_name_.c_str(), goal->order, feedback_.sequence[0],
feedback_.sequence[1]);
```

← Simulate intense compute time

← Init the feedback

CLIENT

```
for(int i=1; i<=goal->order; i++){
    if (as_.isPreemptRequested() || !ros::ok()) {
        ROS_INFO("%s: Preempted", action_name_.c_str());
        as_.setPreempted();
        success = false;
        break;
    }
    feedback_.sequence.push_back(feedback_.sequence[i] +
feedback_.sequence[i-1]);
    as_.publishFeedback(feedback_);
    r.sleep();
}
```

Check if the server need to stop

Create and publish a feedback

CLIENT

```
if(success)
    {
      result_.sequence = feedback_.sequence;
      ROS_INFO("%s: Succeeded", action_name_.c_str());
      // set the action state to succeeded
      as_.setSucceeded(result_);
    }
```

If the goal was completed correctly

Set the result

Set succeeded

TESTING

You can monitor the server status simply using topics:

$ rostopic list

To get the feedback from the server:

$ rostopic echo /fibonacci/feedback

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