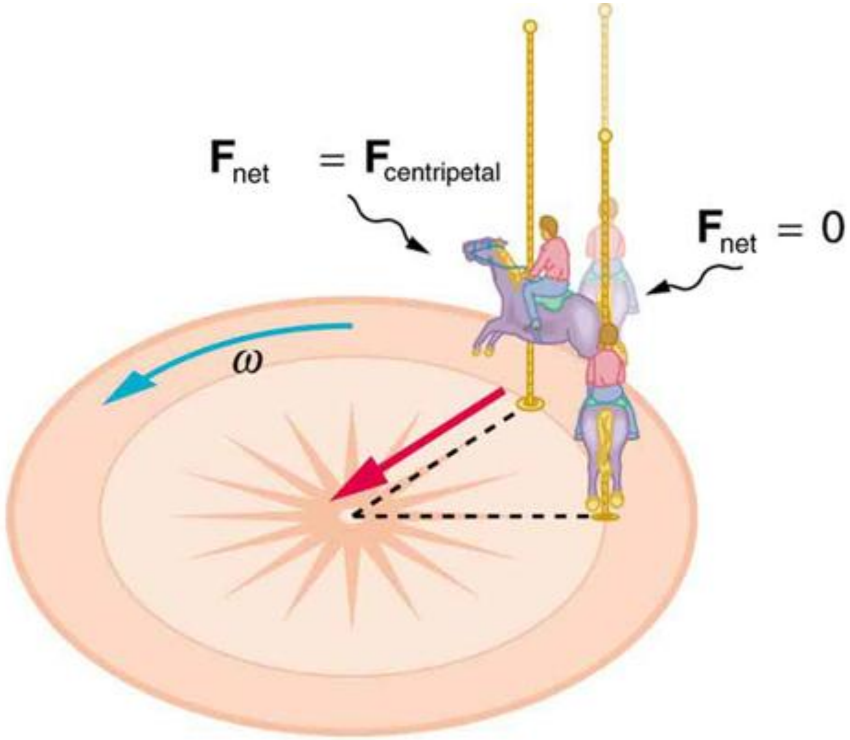
TF

ROBOTICS

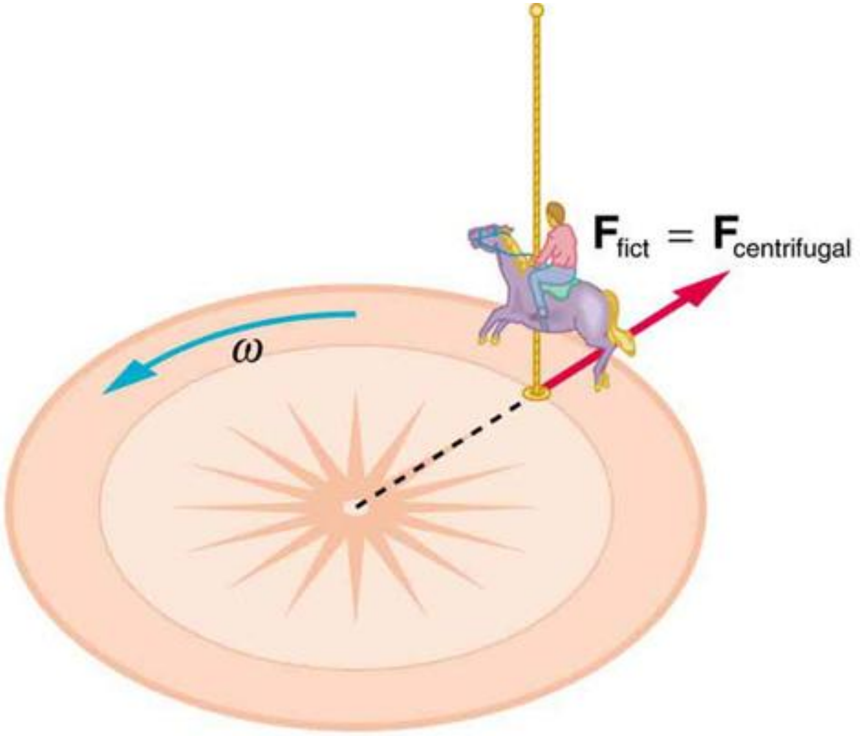
POLITECNICO
MILANO 1863

IN PHYSICS: AN EXAMPLE

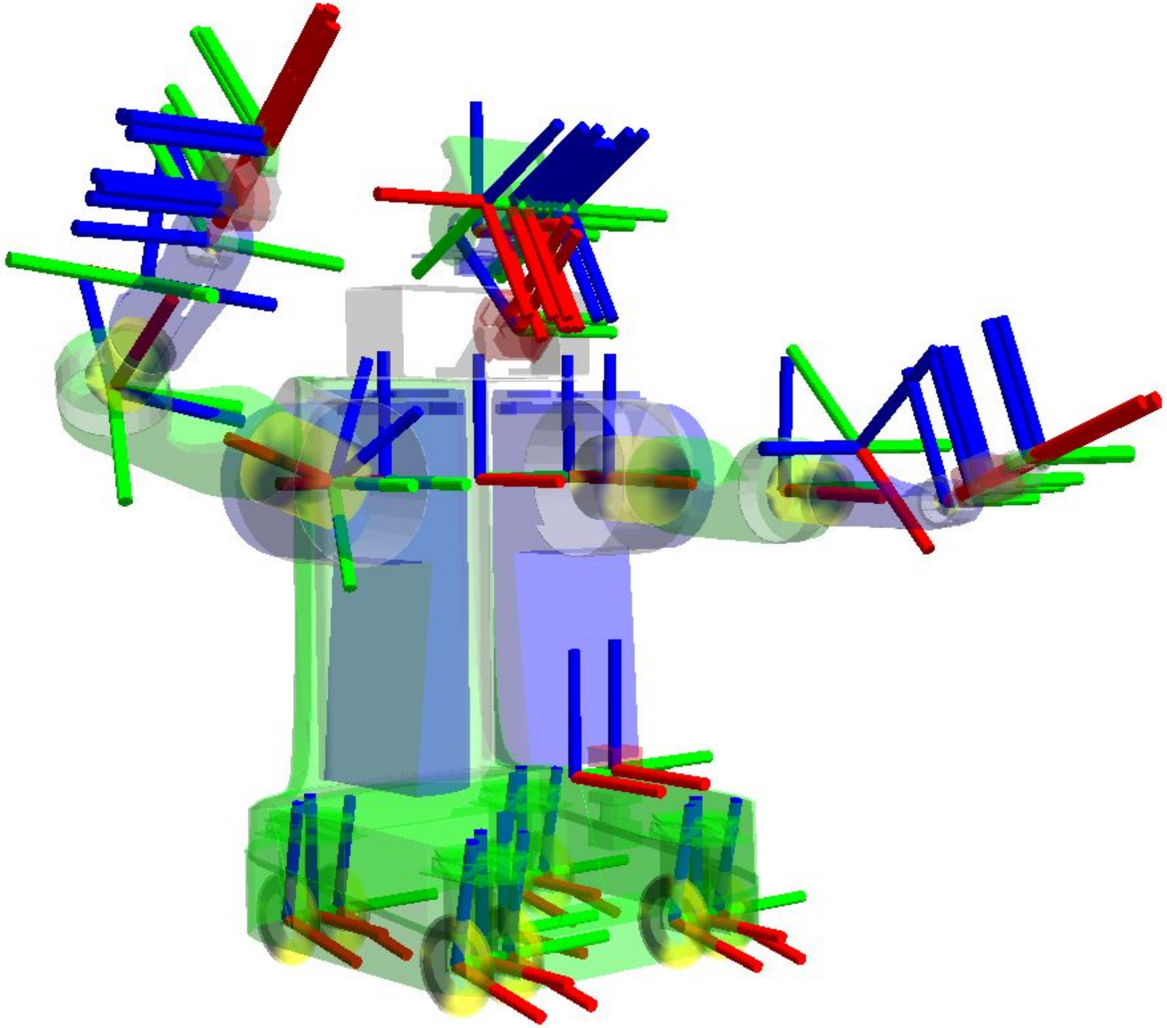
Reference System is everything



VS



IN ROBOTICS



IN ROBOTICS

For manipulators:

- A moving reference frame for each joint
- A base reference frame
- A world reference frame

For autonomous vehicles:

- A fixed reference frame for each sensor
- A base reference frame
- A world reference frame
- A map reference frame

The frames are described in a tree and each frame comes with a transformation between itself and the father/child

The world frame is the most important, but the others are used for

FROM ONE FRAME TO ANOTHER

How is it possible to convert form a frame to another? ***Math***, lot of it.

In a tree of reference frames:

Define a roto-translation between parent and child

Combine multiple roto-translation to go from the root to the

TF: TRANSFORMATION FRAMES

When the full transformation tree is available

Does all the hard work for us!

Interpolation, transformation, tracking

Keep track of all the dynamic transformation for a limited period of time

Decentralized

Provides position of a point in each possible reference frame

TF TREE TOOLS

ROS offers different tools to analyze the transformation tree:

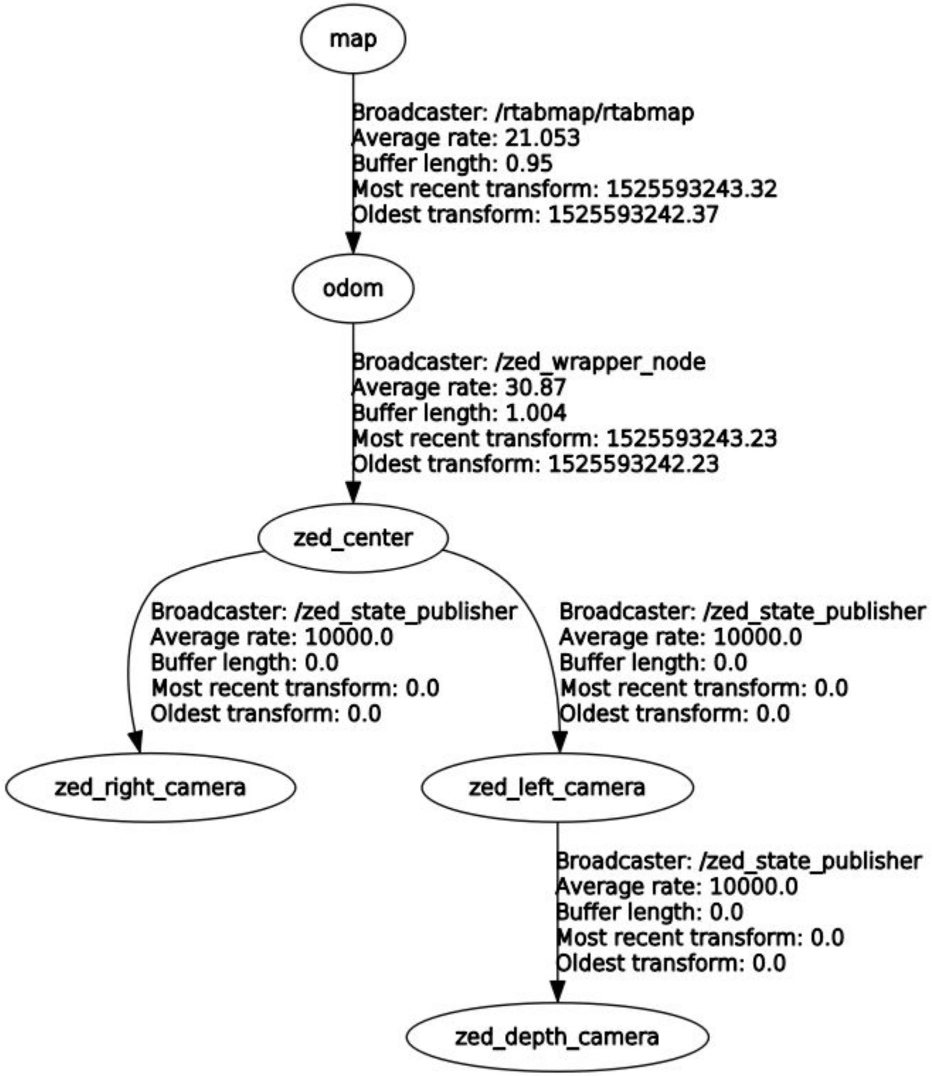
-rosrun rqt_tf_tree rqt_tf_tree

shows the tf tree at the current time

-rosrun tf view_frames

listen for 5 seconds to the /tf topic and create a pdf file with the tf tree

HOW TF_TREE SHOULD LOOK LIKE



WRITE THE TF PUBLISHER

Now that we got an idea regarding how tf works and why it’s useful we can take a look on how to write a tf broadcaster

Usually to do this you need a robot,

we could still us a bag publishing odometry,

but turtlesim is still a good option.

WRITE THE TF PUBLISHER

Subscribe to /turtlesim/pose

convert the pose to a transformation

publish the transformation referred to a world frame

add 4 static transformation for the 4 turtle’s legs

WRITE THE TF PUBLISHER

Create a package called tf_turtlebot inside you catkin environment adding the roscpp, std_msgs and tf dependencies:

```
$ catkin_create_pkg tf_turtlebot std_msgs  roscpp tf
```

now cd to the package src folder and create the file tf_publisher

```
$ gedit tf_publisher.cpp
```

WRITE THE TF PUBLISHER

First we write some standard include:

```
#include "ros/ros.h"
#include "turtlesim/Pose.h"
#include <tf/transform_broadcaster.h>
```

WRITE THE TF PUBLISHER

Then we write the main function:

```
int main(int argc, char **argv)
{
 ros::init(argc, argv, "subscribe_and_publish");
 tf_sub_pub my_tf_sub_bub;
 ros::spin();
 return 0;
}
```

Notice that we still have to initialize ros, but we are not creating the node handle here, instead we instantiate an object of class tf_sub_pub

WRITE THE TF PUBLISHER

Now we have to create our class:

```
class tf_sub_pub
{
    public:
    tf_sub_pub(){
    }
    private:

};
```

WRITE THE TF PUBLISHER

First we declare as private the node handle:

```
ros::NodeHandle n;
```

Then we create the subscriber and the tf broadcaster:

```
tf::TransformBroadcaster br;
ros::Subscriber sub;
```

WRITE THE TF PUBLISHER

Now we can call the subscribe function inside the class constructor:

```
sub = n.subscribe("/turtle1/pose", 1000, &tf_sub_pub::callback, this);
```

Then we write the callback function:

```
void callback(const turtlesim::Pose::ConstPtr& msg){
}
```

WRITE THE TF PUBLISHER

Inside the callback we create a transform object:

```
tf::Transform transform;
```

and populate it using the data from the message (we are in a 2D environment):

```
transform.setOrigin( tf::Vector3(msg->x, msg->y, 0) );
tf::Quaternion q;
q.setRPY(0, 0, msg->theta);
transform.setRotation(q);
```

WRITE THE TF PUBLISHER

Last we publish the transformation using the broadcaster; the stampedtransform function allow us to create a stamped transformation adding the timestamp, our custom transformation, the root frame and the child frame:

```
br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", "turtle"));
```

THE CODE

```
#include "ros/ros.h"
#include "turtlesim/Pose.h"
#include <tf/transform_broadcaster.h>
class tf_sub_pub
{
public:
        tf_sub_pub(){
        sub = n.subscribe("/turtle1/pose", 1000, &tf_sub_pub::callback, this);
}
void callback(const turtlesim::Pose::ConstPtr& msg){
tf::Transform transform;
transform.setOrigin( tf::Vector3(msg->x, msg->y, 0) );
tf::Quaternion q;
q.setRPY(0, 0, msg->theta);
transform.setRotation(q);
br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", "turtle"));
}
private:
ros::NodeHandle n;
tf::TransformBroadcaster br;
ros::Subscriber sub;
};
int main(int argc, char **argv)
{
 ros::init(argc, argv, "subscribe_and_publish");
 tf_sub_pub my_tf_sub_bub;
 ros::spin();
 return 0;
}
```

WRITE THE TF PUBLISHER

Now as usual we have to add this new file to the CMakeLists file. We specified the dependencies during the package creation, so we only need to add the lines:

```
add_executable(tf_turtlebot
 src/tf_publisher.cpp
)
add_dependencies(tf_turtlebot ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(tf_turtlebot
 ${catkin_LIBRARIES}
)
```

TESTING

Now we can cd to the root of the environment and compile everything

Before adding the legs transformation we can test our code:

run turtlesim, turtlesim_teleop and our node, then open rviz to visualize the tf

$ roscore

$ rosrun turtlesim turtlesim_node

$ rosrun turtlesim turtle_teleop_key

$ rosrun tf_turtlebot tf_turtlebot

$ rviz

ADD STATIC TF

After properly testing our code we can add the other tf.

But the legs tf are fixed from the turtlebot body, so we don’t need to write a tf broadcaster like we did, we can simply run them using the static transform node

We don't't’ want to manually start four tf in four different terminals, so we will create a launch file:

create a folder launch and a file called launch.launch

ADD STATIC TF

The launch file will have as usual the <launch> tags and the node we previously wrote:

```
<launch>
<node pkg="tf_turtlebot" type = "tf_turtlebot" name = "tf_turtlebot"/>
</launch>
```

We can also add the two turtlesim node:

```
<node pkg="turtlesim" type = "turtlesim_node" name = "turtlesim_node"/>
<node pkg="turtlesim" type = "turtle_teleop_key" name = "turtle_teleop_key"/>
```

ADD STATIC TF

Now we will add the four static tf specifying in the args field the position (x,y,z) and the rotation as a quaternion (qx,qy,qz,qw) then the root frame, the cild frame and the update rate:

```
<node pkg="tf" type="static_transform_publisher" name="back_right" args="0.3 -0.3 0 0 0 0 1 turtle FRleg 100" />
<node pkg="tf" type="static_transform_publisher" name="front_right" args="0.3 0.3 0 0 0 0 1 turtle FLleg 100" />
<node pkg="tf" type="static_transform_publisher" name="front_left" args="-0.3 0.3 0 0 0 0 1 turtle BLleg 100" />
<node pkg="tf" type="static_transform_publisher" name="back_left" args="-0.3 -0.3 0 0 0 0 1 turtle BRleg 100" />
```

Now we will only need to call the launch file to start all the nodes:

$ roslaunch tf_turtlebot launch.launch

ADD STATIC TF

Now run rqt_tf_tree to show the tf tree and rviz for the visual rappresentation of the turtle position

If you want to see the published tf you can use rostopic echo, but also:

$ rosrun tf tf_echo father child

$ rosrun tf tf_echo \world \FRleg

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

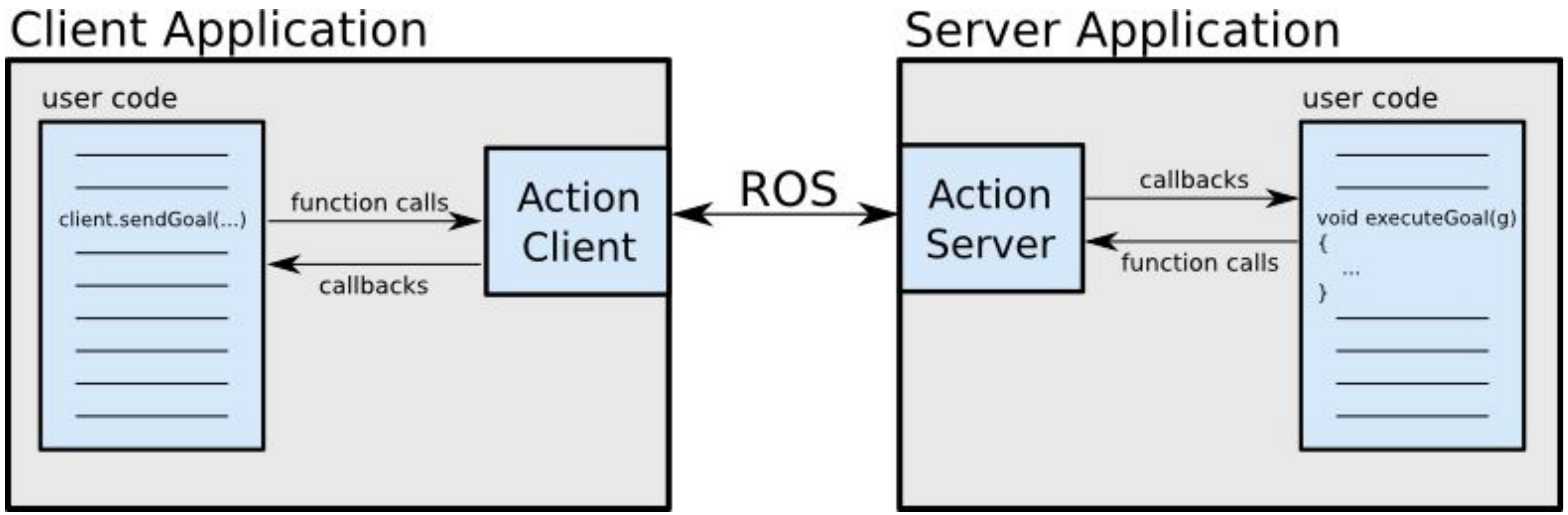
actionlib package is:

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

And provides tools to:

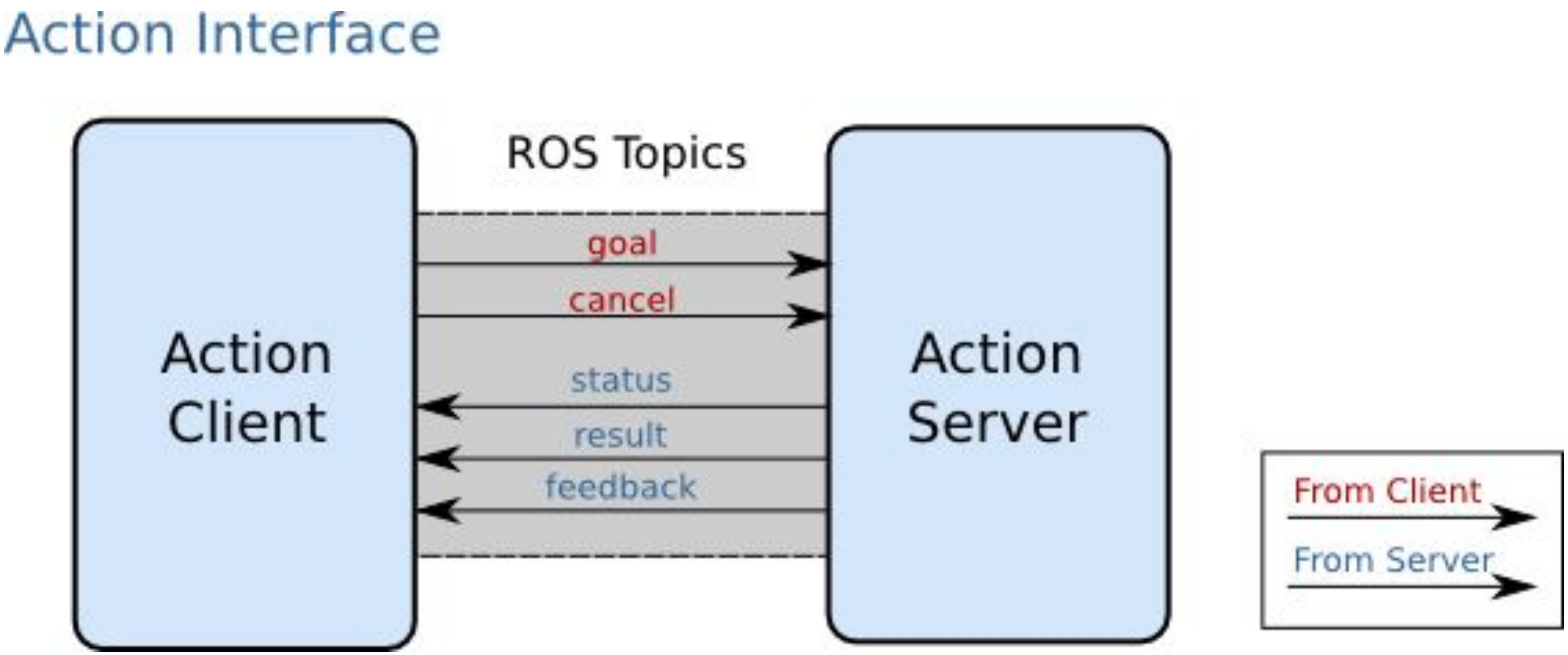
- create servers that execute long-running tasks (that can be preempted).
- create clients that interact with servers

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

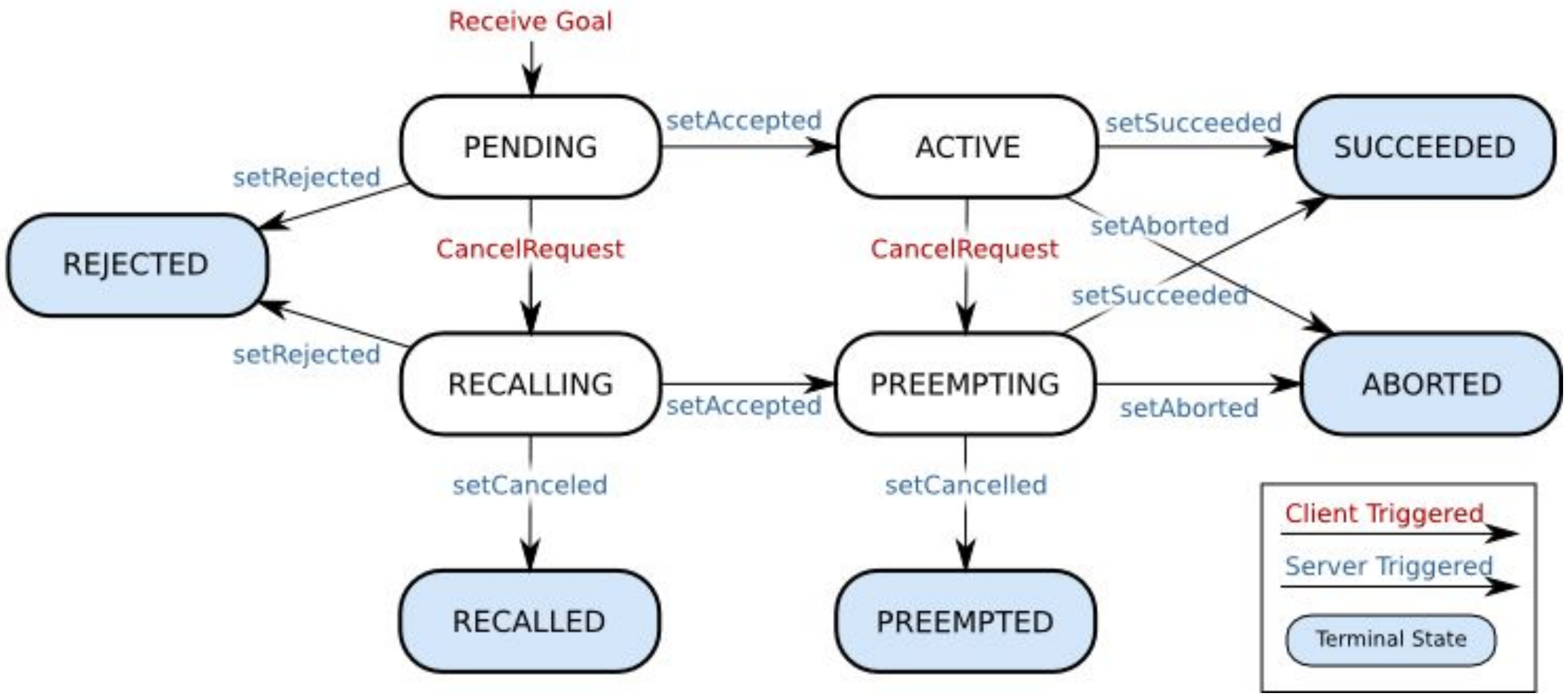
ACTION AND GOAL ID

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

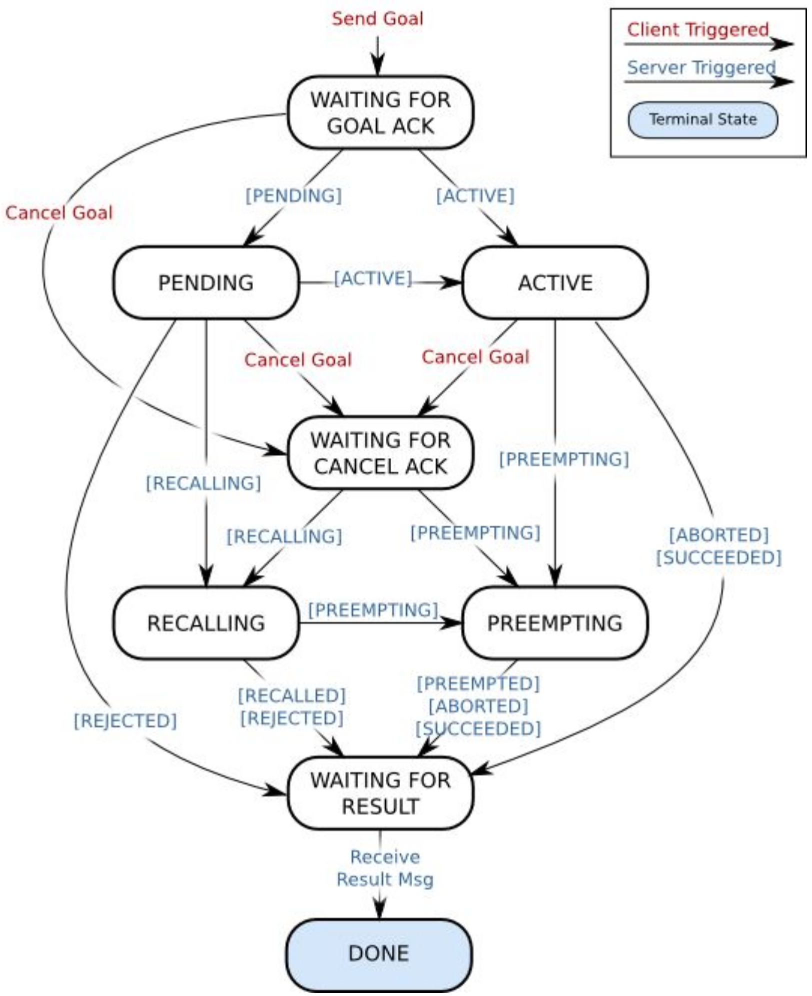
Each ***instance*** of an action has a unique Goal ID

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_;
```

Create the Node handle

```
  actionlib::SimpleActionServer<actionlib_tutorials::FibonacciAction> as_;
```

create the server

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

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