**Laurent LEQUIEVRE – Ingénieur CNRS - Institut Pascal UMR6602**

**Equipe Maccs http://ip.univ-bpclermont.fr/index.php/fr/maccs**

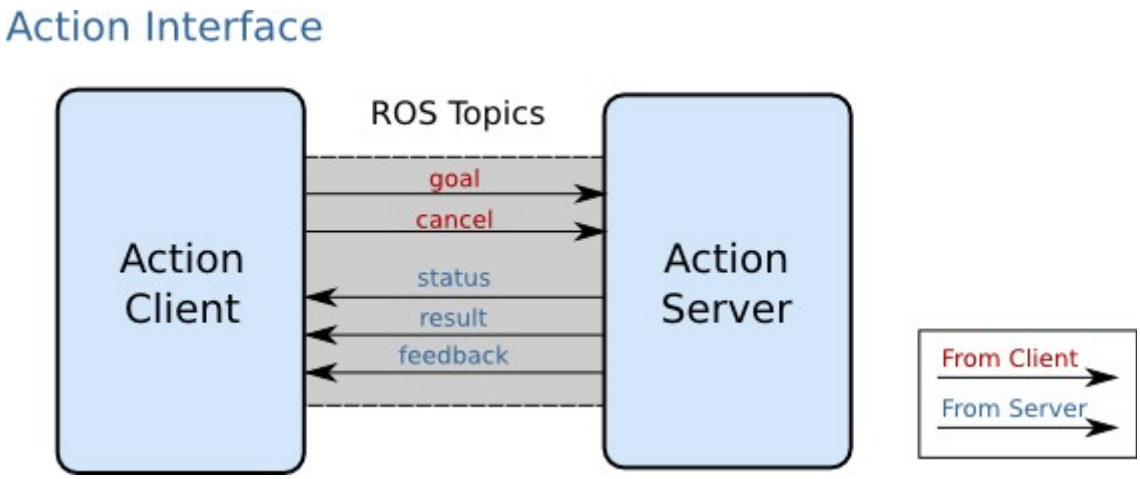
**Juan Antonio CORRALES RAMON - Youcef MEZOUAR**

# Notions de ROS actionlib

La notion de 'service' est très utile lorsque le travail demandé ne prend pas trop de temps, car cela bloque le client jusqu'à la réponse du 'service'.

Lorsque le temps de traitement d'une requête est long, il faut mieux utiliser une 'action'. Le paquet ROS qui définit cette notion est nommé 'actionlib' (http://wiki.ros.org/actionlib).

Sur le même principe que le 'service', un client peut formuler une requête (un 'goal') qui sera exécutée sur un serveur, que l'on peut à tout moment, soit annuler ('cancel'), soit vérifier sa progression ('feedback'). Le client qui a demandé cette requête n'est pas bloqué pendant son exécution sur le serveur.
La requête demandée pouvant être annulée à tout moment, le système a une capacité de préemption.




ROS Messages

- **goal** - Used to send new goals to servers.
- **cancel** - Used to send cancel requests to servers.
- **status** - Used to notify clients on the current state of every goal in the system.
- **feedback** - Used to send clients periodic auxiliary information for a goal.
- **result** - Used to send clients one-time auxiliary information upon completion of a goal.

Nous allons créer un Client et un Serveur. Le Client va soumettre une requête qui calcule la suite de Fibonacci d'ordre n (qui sera le 'goal'). Le calcul se fera sur le Serveur. Une fois le calcul réalisé, le Client recevra le résultat et l'affichera.
De même le Client récupérera la progression ('feedback') du calcul intermédiaire de la suite. Enfin, pendant le calcul de la suite, le client effectuera un autre traitement (une simple boucle d'affichage), montrant qu'il n'est pas bloqué.

1- Créer un nouveau paquet nommé 'my_first_action' dans votre 'workspace'. Ce nouveau paquet sera dépendant de 'roscpp' mais aussi de 'actionlib' et 'actionlib_msgs'.
http://wiki.ros.org/actionlib
http://wiki.ros.org/actionlib_msgs

2- Créer un fichier nommé 'Fibonacci.action' dans un nouveau dossier 'action' à la racine de votre paquet.

Ce fichier contiendra :

```
#goal definition
int32 order
---
#result definition
int32[] sequence
---
#feedback
int32[] sequence
```

Il permet de définir les noms et types du 'goal' (l'ordre 'n' à calculer) , du 'feedback' (la séquence intermédiaire d'entiers), du résultat final (la séquence finale d'entiers).

3- Modifier le fichier 'CMakeLists.txt' de votre paquet afin de prendre en compte le fichier 'action' :

```
cmake_minimum_required(VERSION 2.8.3)
project(my_first_action)

find_package(catkin REQUIRED COMPONENTS
  actionlib
  actionlib_msgs
  roscpp
)

add_action_files(
  DIRECTORY action
  FILES Fibonacci.action
)

generate_messages(
  DEPENDENCIES actionlib_msgs std_msgs
)


catkin_package(
  CATKIN_DEPENDS actionlib actionlib_msgs roscpp
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

Compiler le paquet.

Il y a maintenant, dans le dossier '**devel/share/my_first_action/msg**', les fichiers 'message' nécessaires pour la communication entre le 'Client' et le 'Serveur' :
- FibonacciActionFeedback.msg
- FibonacciActionResult.msg
- FibonacciResult.msg // Result definition msg
- FibonacciActionGoal.msg
- FibonacciFeedback.msg // Feedback definition msg
- FibonacciAction.msg // Message contenant la définition générale de l'action
- FibonacciGoal.msg // Goal definition msg

Il y a aussi, dans le dossier '**devel/include/my_first_action**' les fichiers 'headers' correspondant aux 'messages' :
- FibonacciActionFeedback.h
- FibonacciActionResult.h
- FibonacciResult.h
- FibonacciActionGoal.h
- FibonacciFeedback.h
- FibonacciAction.h
- FibonacciGoal.h

4- Ajouter un fichier source nommé  'fibonacci_client.cpp' dans votre paquet.
Ce fichier correspondra au 'Client'.

Il contiendra ceci :

```
#include <ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include <my_first_action/FibonacciAction.h>
#include <boost/thread.hpp>

// Called once when the goal completes
void doneCb(const actionlib::SimpleClientGoalState& state,
          const my_first_action::FibonacciResultConstPtr& result)
{
  ROS_INFO("Finished in state [%s]", state.toString().c_str());
  ROS_INFO("Answer: %i", result->sequence.back());
  ros::shutdown();
}

// Called once when the goal becomes active
void activeCb()
{
  ROS_INFO("Goal just went active");
}

// Called every time feedback is received for the goal
void feedbackCb(const my_first_action::FibonacciFeedbackConstPtr& feedback)
{
  ROS_INFO("Got Feedback of length %lu", feedback->sequence.size());
}

void workerFunc()
{
              ros::Duration d = ros::Duration(1, 0); // 1.0 second

              for (int i=0; i<1000; i++)
              {
                            ROS_INFO("--> Client WorkerFunc with i=%i", i);
                            d.sleep();
              }
}

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

  // create the action client
  // true causes the client to spin its own thread
  actionlib::SimpleActionClient<my_first_action::FibonacciAction> ac("fibonacci", true);

  ROS_INFO("Waiting for action server to start.");
  // wait for the action server to start
  ac.waitForServer(); //will wait for infinite time

  ROS_INFO("Action server started, sending goal.");
  // send a goal to the action
  my_first_action::FibonacciGoal goal;
  goal.order = 20;
  ac.sendGoal(goal, &doneCb, &activeCb, &feedbackCb);

  boost::thread workerThread(&workerFunc);

  ros::spin();

  return 0;
}
```

Principalement dans ce fichier on :

- crée un 'Client' paramétré avec l'action 'my_first_action::FibonacciAction' ('fibonacci' est le nom du 'Serveur')

```
actionlib::SimpleActionClient<my_first_action::FibonacciAction> ac("fibonacci", true);
```

- attend que le 'Serveur' soit prêt

```
 ac.waitForServer(); //will wait for infinite time
```

- définit l'ordre (sous forme de 'goal') de la suite et on l'envoie au 'Serveur'

```
my_first_action::FibonacciGoal goal;
goal.order = 20;
ac.sendGoal(goal, &doneCb, &activeCb, &feedbackCb);
```

On associe plusieurs fonctions 'callback'.

```
doneCb → une fois le résultat final envoyé par le 'Serveur'
activeCb → une fois le 'goal' devenu actif
feedbackCb → à chaque progression du calcul
```

- crée un 'thread boost' associé à une fonction 'workerFunc', montrant que le 'Client' n'est pas bloqué durant le calcul de la suite

```
boost::thread workerThread(&workerFunc);
```

5- Ajouter un fichier source nommé  'fibonacci_server.cpp' dans votre paquet.
Ce fichier correspondra au 'Server'.

Il contiendra ceci :

```
#include <ros/ros.h>
#include <actionlib/server/simple_action_server.h>
#include <my_first_action/FibonacciAction.h>

class FibonacciAction
{
protected:

  ros::NodeHandle nh_;
  // NodeHandle instance must be created before this line. Otherwise strange error may occur.
  actionlib::SimpleActionServer<my_first_action::FibonacciAction> as_;
  std::string action_name_;
  // create messages that are used to published feedback/result
  my_first_action::FibonacciFeedback feedback_;
  my_first_action::FibonacciResult result_;

public:

  FibonacciAction(std::string name) :
    as_(nh_, name, boost::bind(&FibonacciAction::executeCB, this, _1), false),
    action_name_(name)
  {
    ROS_INFO("Create FibonacciAction, and start -> actionlib::SimpleActionServer<my_first_action::FibonacciAction> !");
    as_.start();
  }

  ~FibonacciAction(void)
  {
  }

  void executeCB(const my_first_action::FibonacciGoalConstPtr &goal)
  {
    // helper variables
    ros::Rate r(1);
    bool success = true;

    // push_back the seeds for the fibonacci sequence
    feedback_.sequence.clear();
    feedback_.sequence.push_back(0);
    feedback_.sequence.push_back(1);

    // publish info to the console for the user
    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]);

    // start executing the action
    for(int i=1; i<=goal->order; i++)
    {
      // check that preempt has not been requested by the client
      if (as_.isPreemptRequested() || !ros::ok())
      {
        ROS_INFO("%s: Preempted !", action_name_.c_str());

        // set the action state to preempted
        as_.setPreempted();
        success = false;
        break;
      }

      feedback_.sequence.push_back(feedback_.sequence[i] + feedback_.sequence[i-1]);
```

```
    ROS_INFO("%s: Executing, creating feedback %i", action_name_.c_str(), feedback_.sequence[i+1]);

    // publish the feedback
    as_.publishFeedback(feedback_);
    // this sleep is not necessary, the sequence is computed at 1 Hz for demonstration purposes
    r.sleep();
   }

   if(success)
   {
    result_.sequence = feedback_.sequence;
    ROS_INFO("%s: Succeeded", action_name_.c_str());

    // set the action state to succeeded
    as_.setSucceeded(result_);
   }
  }


};


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

  FibonacciAction fibonacci(ros::this_node::getName());
  ros::spin();

  return 0;
}
```

Principalement dans ce fichier on :

- crée une classe 'FibonacciAction ' qui se chargera de démarrer un 'Serveur' dans son constructeur :

```
actionlib::SimpleActionServer<my_first_action::FibonacciAction> as_;
…
FibonacciAction(std::string name) :
    as_(nh_, name, boost::bind(&FibonacciAction::executeCB, this, _1), false),
    action_name_(name)
  {
    ROS_INFO("Create FibonacciAction, and start -> actionlib::SimpleActionServer<my_first_action::FibonacciAction> !");
    as_.start();
  }
```

Ce 'Serveur' sera associé à une fonction 'callback' (ici la méthode 'executeCB' de la classe 'FibonacciAction'). A chaque nouveau 'goal' envoyé au 'Serveur' cela appellera la fonction 'executeCB'.

- publie les calculs intermédiaires dans la fonction 'executeCB' en utilisant le message 'my_first_action::FibonacciFeedback'

```
feedback_.sequence.push_back(feedback_.sequence[i] + feedback_.sequence[i-1]);
ROS_INFO("%s: Executing, creating feedback %i", action_name_.c_str(), feedback_.sequence[i+1]);

// publish the feedback
as_.publishFeedback(feedback_);
```

- publie le résultat final en utilisant le message 'my_first_action::FibonacciResult result_'

```
if(success)
   {
     result_.sequence = feedback_.sequence;
     ROS_INFO("%s: Succeeded", action_name_.c_str());

     // set the action state to succeeded
     as_.setSucceeded(result_);
   }
```

- permet la préemption du calcul (c'est à dire son éventuel arrêt)

```
// check that preempt has not been requested by the client
      if (as_.isPreemptRequested() || !ros::ok())
      {
        ROS_INFO("%s: Preempted !", action_name_.c_str());

        // set the action state to preempted
        as_.setPreempted();
        success = false;
        break;
      }
```

6- Ajouter dans le fichier 'CMakelists.txt' la prise en compte des fichiers source 'Client' et 'Serveur' comme ceci :

```
add_executable(fibonacci_server src/fibonacci_server.cpp)
add_dependencies(fibonacci_server ${${PROJECT_NAME}_EXPORTED_TARGETS})

add_executable(fibonacci_client src/fibonacci_client.cpp)
add_dependencies(fibonacci_client ${${PROJECT_NAME}_EXPORTED_TARGETS})

target_link_libraries(
  fibonacci_server
  ${catkin_LIBRARIES}
)

target_link_libraries(
  fibonacci_client
  ${catkin_LIBRARIES}
)
```

Le 'add_dependencies' permet d'ajouter à la target cmake 'fibonacci_server' ou 'fibonacci_client' une dépendance sur la variable 'my_first_action_EXPORTED_TARGETS'  (elle contient 'my_first_action_generate_messages_cpp').
Cela veut dire que l'on souhaite générer les messages et leurs 'headers' avant de lancer la compilation du 'Client' ou 'Serveur'.

7- Vérifier que le fichier 'package.xml' contient bien :

```
<?xml version="1.0"?>
<package>
  <name>my_first_action</name>
  <version>0.0.0</version>
  <description>The my_first_action package</description>

  <maintainer email="laurent.lequievre@univ-bpclermont.fr">Laurent LEQUIEVRE</maintainer>
  <license>BSD</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>actionlib_msgs</build_depend>
  <build_depend>roscpp</build_depend>
  <run_depend>actionlib</run_depend>
  <run_depend>actionlib_msgs</run_depend>
  <run_depend>roscpp</run_depend>


  <export>
    <!-- Other tools can request additional information be placed here -->
  </export>

</package>
```

8- Compiler le tout

9- Tester le tout

Lancer dans des fenêtres terminal séparées :

```
roscore
rosrun my_first_action fibonacci_server
rosrun my_first_action fibonacci_client
```

Observer les infos des 'topics' et leurs contenus

```
rostopic info /fibonacci/goal → Type: my_first_action/FibonacciActionGoal
rostopic info /fibonacci/cancel → Type: actionlib_msgs/GoalID
rostopic echo /fibonacci/feedback
```

10- Arrêter le calcul avec l'envoie d'un 'cancel' via un 'topic'

Pour envoyer un 'cancel' , on envoie un message vide :

```
rostopic pub -1 /fibonacci/cancel actionlib_msgs/GoalID -- {}
```

(si on envoie un message vide cela 'cancel' tous les 'goal' !)

ou bien, si on souhaite exprimer les paramètres du message, on peut faire aussi :

```
rostopic pub -1 /fibonacci/cancel actionlib_msgs/GoalID '{ stamp: {}, id : '' }'
```

(ici on envoie un temps vide et un 'id' qui doit être une 'string' vide, cf détail des paramètres du message : http://docs.ros.org/diamondback/api/actionlib_msgs/html/msg/GoalID.html )

Pour envoyer un 'goal' via un 'topic' (sans le 'Client') :

```
rostopic pub -1 /fibonacci/goal my_first_action/FibonacciActionGoal '{goal : { order : 20 } }'
```

(pour les paramètres cf le détail des messages 'FibonacciGoal.msg' et 'FibonacciActionGoal.msg')

Quelques liens détaillant les 'actionlib' :
http://wiki.ros.org/actionlib/DetailedDescription
http://wiki.ros.org/actionlib/Tutorials
http://wiki.ros.org/actionlib
http://www.dis.uniroma1.it/~nardi/Didattica/CAI/matdid/1-ROS-ActionLib.pdf
http://library.isr.ist.utl.pt/docs/roswiki/actionlib.html

11- On souhaite lancer le calcul d'une n ! sans être bloqué. Créer un nouveau paquet 'my_facto' qui contiendra un 'Client' et un 'Serveur'.
Le 'Client' demandera comme 'goal' le n ! à calculer.
Le 'Serveur' retournera les calculs intermédiaires de n ! comme 'feedback'.
exemple : 4 ! → [1,2,6,24]
Le calcul peut-être interrompu à tout moment.