Computer Science Engineer

Mid-term report

Immersive Virtual Reality System

SIGVerse interaction trough Robot Operating System

Intern :
Geneviève Cirera (SI5 - Web)

Tutor :
Frédéric Precioso

Supervisor :
Inamura Tetsunari

Enterprise

National Institute of Informatics
(Tokyo - Japan)

2015 March, 15 - 2015 August, 30

Contents

1 Introduction

1.1 Context

1.1.1 National Institute of Informatics

The NII[1] of Tokyo is an inter university research institute which aim to develop the research in multiple domains. The Institute is focused on informations-related fields including networking software and content. The NII is composed by several laboratories which work on different international research project. One of them is directed by the associate professor Inamura Tetsunari where I am doing my internship.
The Inamura laboratory works on several project, which one is SIGVerse, a simulator.

1.1.2 SIGVerse

SIGVerse is a virtual world which can modelise objects and agents.
This simulator has been created to give a tool for studying interactions between agents but also comprehension and knowledge in many fields.
Understanding the mecanism of intelligence of human being is the key to develop intelligent robot system. That's why this simulator is very useful.
The movement of the user can be reproduced in the virtual world thanks to the kinect and the representation of the world can be projected on the oculus. So, the user can be "inside" the simulator and interact more easily with its.
We can see in figure 1.1, the representation of an agent. This agent could be a robot too, and it is possible to program its movements or send to it a message to execute an action. That's why this simulator is well adapted to host the training and virtual competition of Robocup.
However, the use of the simulator is exclusively for SIGVerse users which limits its growth.

[1]National Institute of Informatics

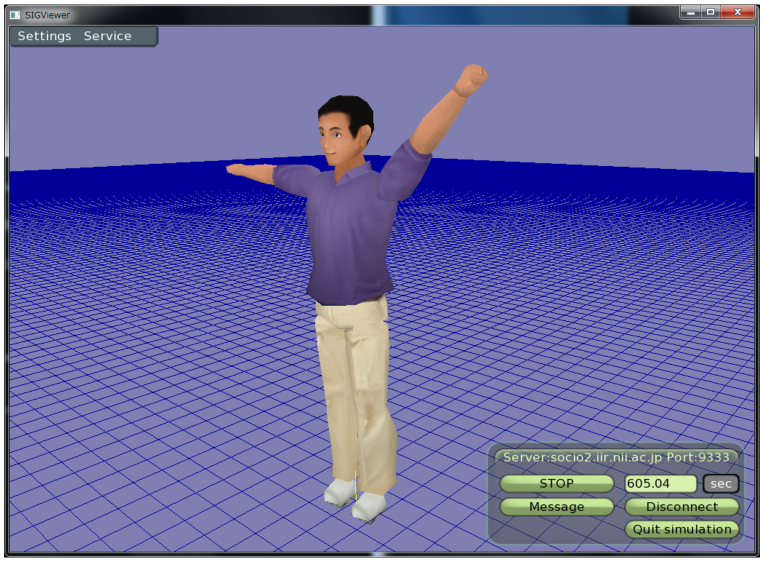


Figure 1.1: Simple agent in the virtual world

1.2 Topic

SIGVerse is limited to SIGVerse users. This topic has the goal to growth the SIGVerse community by joining the ROS[2] community.
This gathering has been chosen because of several reasons.
First of all ROS is open source like SIGVerse and has a big community who works on robots. Then ROS can provide a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.
ROS make possible to manage a robot as SIGVerse do but there is advantages to using ROS. Indeed, ROS is designed to be as distributed and modular as possible so, it encourages the collaboration to develop robot software whereas SIGVerse is not.

In this report, I am going to detail the suggested work, general idea and how ROS works. After that, I am going to detail what I started doing, architecture, topics, services, troubles...

[2]Robot Operating System

2 Suggested work

2.1 Robot Operating System

The ROS[1] is an open source flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.
At the lowest level, ROS is commonly referred to as a middleware. It provides some facilities like publishing/subscribing anonymous message passing and request/response remote procedure calls. We can see in the figure 2.1 the base concept of ROS. Each entity is a node who can publish or subscribe to a topic. If a node publish to a topic, every node who subscribed to it will receive the message. Many nodes can publish in the same topic and many nodes can subscribe to the same topic. Every node can publish and/or subscribe to several nodes.
We can also see a service between two nodes, the first node can send a request with zero or more parameters to the other node who will act and respond with parameters.

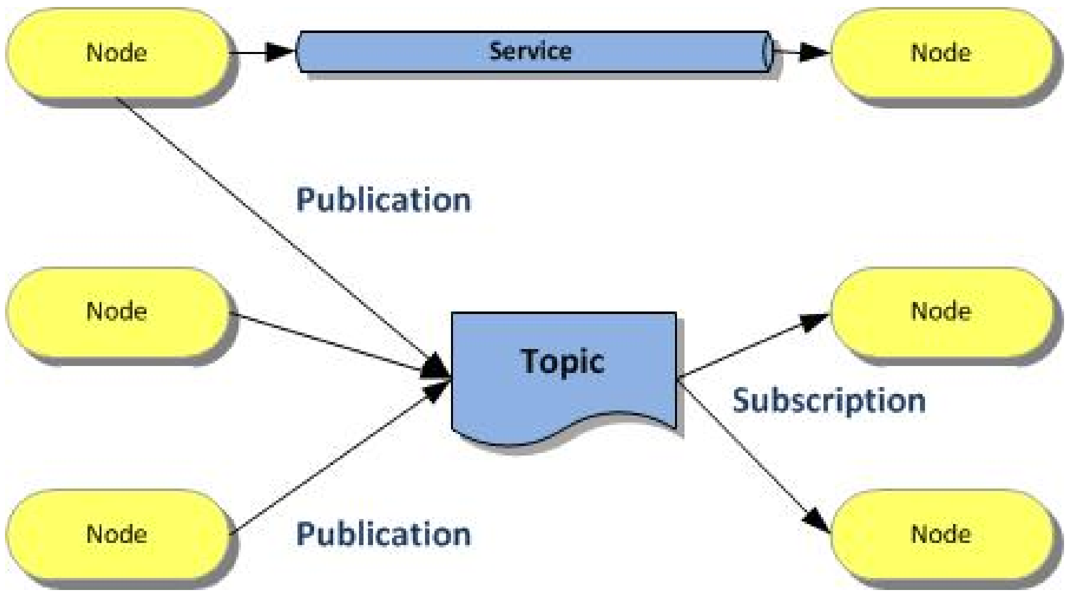


Figure 2.1: Publish/Subscribe system

[1]Robot Operating System

2.2 SIGVerse architecture

Currently, the architecture of SIGVerse is as shown figure 2.2.
On the Linux part, the server is running with the xml file where the agent is defined and the Controller.cpp where the initialisation and actions are defined.
On the Windows part, two things, SIGViewer which is a GUI and aim to show to the user the agent on the simulator. The services are all devices which can provide data to the server. That means the kinect, the oculus,...

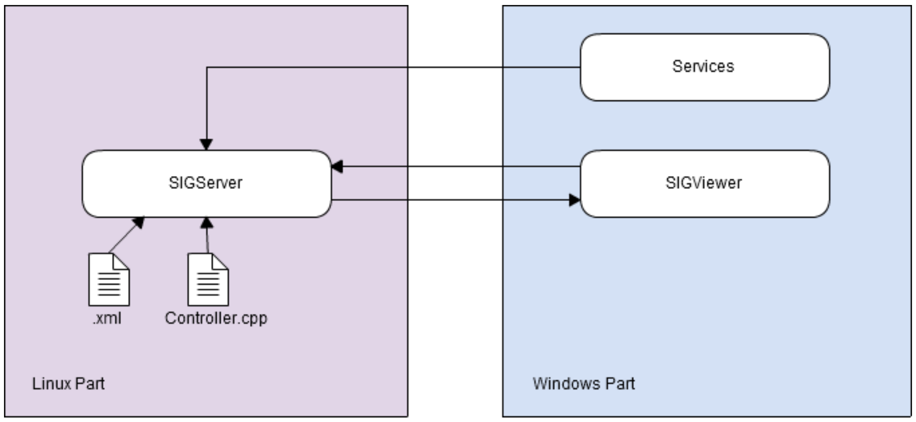


Figure 2.2: SIGVerse

2.3 Objective

As seen section 2.1, using ROS to create and manipulate agents on SIGVerse will be very useful. So, I have to design and develop an interface between SIGVerse and ROS.
This interface has to make available the all fonctionnalities of SIGVerse from ROS, that means initializing one or more agents, make them act and make the services working though ROS.
The agents can be : robot, human or object.
The expected architecture is shown figure 2.3. We can see the SIGServer and the “Controller” inside the ROS interface, that means the user will not need to interact with SIGVerse, he only defined the agent in the xml file and then, interacting with ROS, send messages to the server though the interface. The user will not need to write the “Controller”, it will be automatically generated.
This is an example with one xml file and one controller, but it will be necessary to generalise it to more than one.

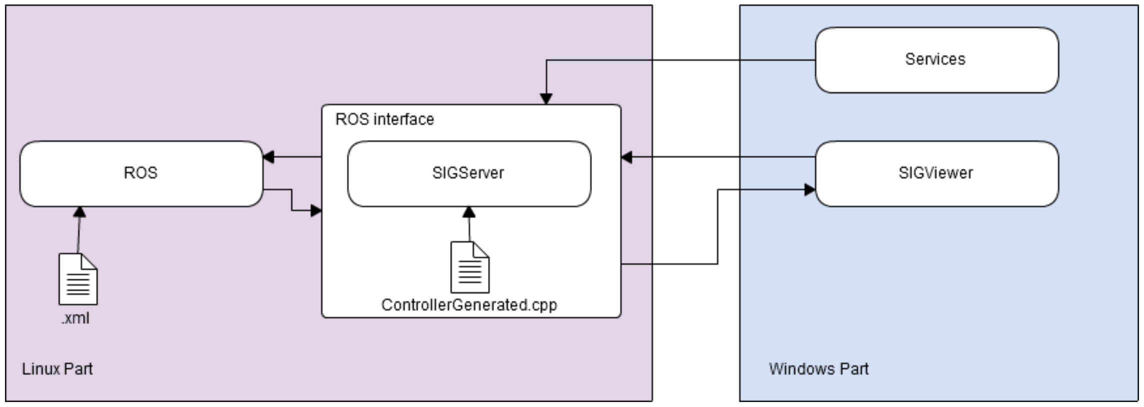


Figure 2.3: SIGVerse with ROS interface

2.4 Fonctionalities requiered

2.4.1 General requierement

Currently, SIGVerse has two parts: SIGServer and SIGViewer and they communicate directly together as seen figure 2.2.
The aim is to find a way for the ROS users for using SIGVerse. That means the ROS user will only write ROS code and this will be enough for manipulate the simulator.
Right now, if the ROS user wants to do it, he has to make the interface himself, mapping the SIGVerse function which is needed. That is why, a common interface with the main functions are useful. After that, the user will have the possibility to enhance it.

Three agents are available on SIGVerse: the robot, the human and the object and different actions are available for each one. But some same actions are available for many of them. Indead the robot agent inherit from the object agent and the human can be a special robot.

2.4.2 Use case

The main use case is for the Robocup competition. Indeed, with this interface, ROS users could participate to the competition.
RoboCup is an annual international robotics competition founded in 1997.
There are many stages of competition, RoboCupRescue, RoboCup@Home, RoboCupJunior,...
The best known is the football competition where two team of robots are playing football, but the Inamura lab works on RoboCup@Home using SIGVerse. Three kinds of task are competing, the clean up task, the follow me and the EGPSR task.

In a simple clean up task, the robot detects the trash, go to take it and puts it in the trashbox detected. Points are given for every good things done like “take the trash” and “put it in the trashbox”, but points are removed if a collision occurs.
The aim of the follow me task is to follow someone without collision, don’t lose him and knowing in which direction after entering the elevator. Indeed, if the man entry the elevator, the robot will enter too but the man will not be able to go out before the robot do it.
The EGPSR task is the interaction between the human and the robot. The man asks the robot for an object in a room and the robot has to go to the right room and get it back.
First of all, the clean up task has to work with ROS, that means the methods used with the robot and some methods of objects.
After that, there is a third agent, the human, so the follow up task will be a pertinent example.

2.5 ROS and SIGVerse

On SIGVerse an agent is defined by an XML file for his representation and a “controller” for his dynamic. The dynamic allows an agent to act, receive message or send message.
Currently, the user has to transform the “Controller” into a ROS node himself, defining the interface for each method needed.
We can see in figure 2.4 an example. Indeed, we can see the “Controller” inside the server who is also a ROS node. That is why it can publish a message to a topic and subscribe to “Velocity Topic”. After that, any ROS node can be created and publish or subscribe to topics, the SIGVerse agent can receive instructions from a topic or a service.

In this example figure 2.4 given on the SIGVerse wiki page[3], the “Controller” launch the ROS node when the simulation starts and the topics are created. However, we want the user to write only ROS code and do not bother himself with the “Controller” and SIGVerse functioning.

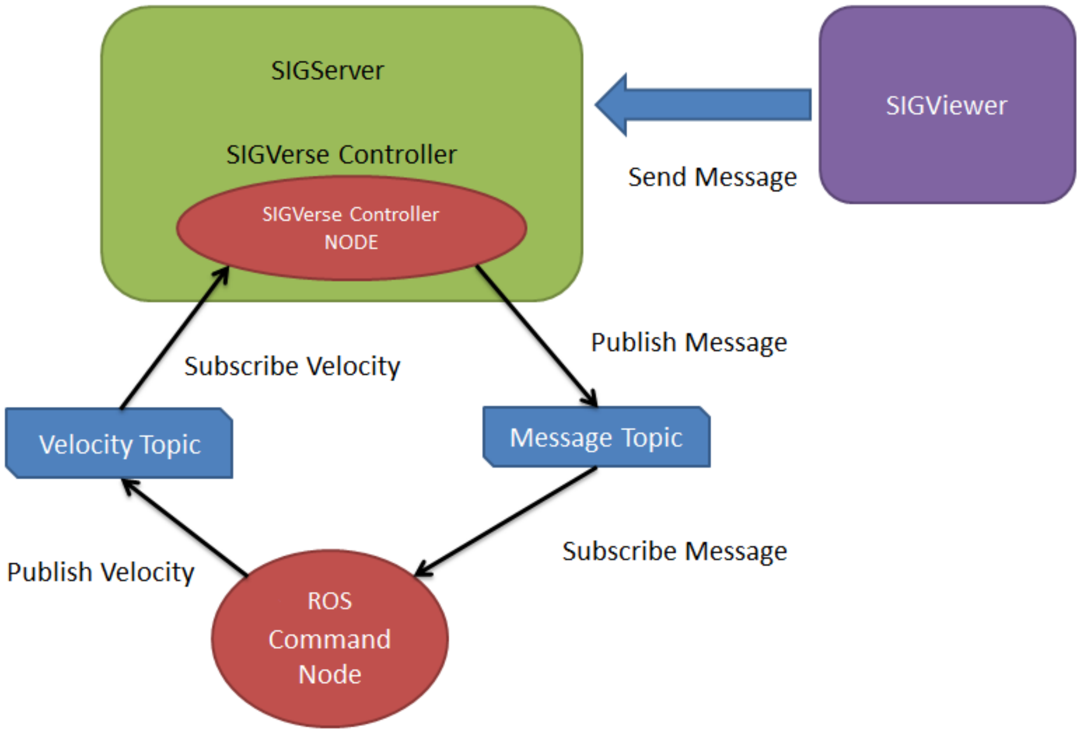


Figure 2.4: SIGVerse controller sending and receiving message

3 Work performed

3.1 Controller

Each "Controller" associated to an agent on the world inherit from the main class "Controller" of SIGVerse. That is why each of them has the same dynamic shown figure 3.1.

When the simulation starts on the SIGViewer, each "Controller" is initialized running "onInit" method. After that the "onAction" method is running regularly, it can be every 1 seconds like 0.1, 0.5,... It is defined by the return value of "onAction" method.
If a collision occurs between the agent and something else, "onCollision" is executed.
If the agent receives a message, "onRecvMsg" is executed.

In any case, "onAction" is running until the simulation stops. If the simulation restarts, "onInit" is not executed, the simulation continues where it stopped.

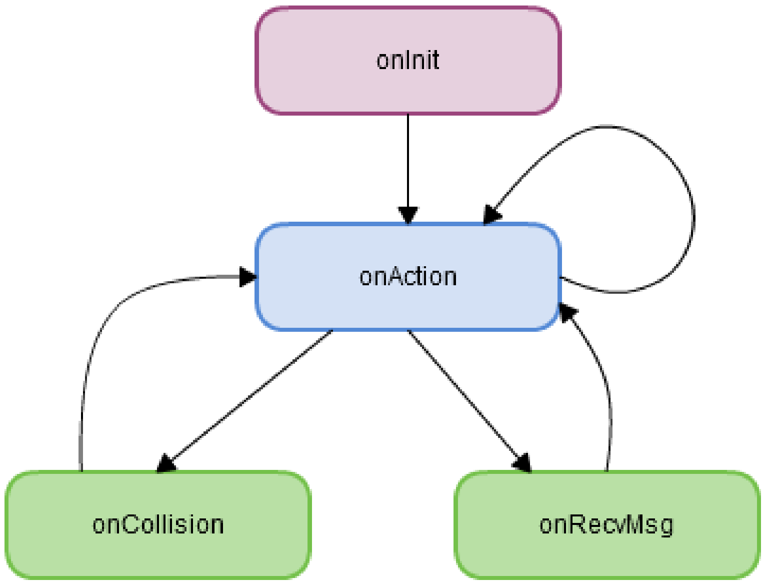


Figure 3.1: Dynamic of the "Controller"

3.2 Architecture

3.2.1 General objective

On the SIGServer, severals “Controller” can run at the same time, because there are one for each agent in the simulator and their types can be different. For exemple, we can have a “Controller” for a Robot and “Controller” for an object. The principal difference between them is that a Robot controller do not apply the same method to the agent than an object, indeed the robot can move, not an object. But they have the same dynamic, section 3.1.
If we have three robots in the simulator, that means three “Controller”.

We can see figure 3.2 how the interface has to work. Several nodes can send information to topics which can be the same or not. This is the ROS part. And these topics are subscribed by the “Controller” of SIGServer.
As we can see, the same node can publish to the same node “Topic 4” or one node can publish to several nodes “Node 1”.
However, different “Controller” cannot publish or subscribe to the same topic. The reason is that the ROS user do not have to write the “Controller” or edit it neither, it will be generated and I can not know if the user want the same behaviour between two or more agents. If the user wants this behaviour, he will have to send the same message to severals topics.

As we can see figure 3.2 a service can be called too. The difference with publishing/subscribing is that a request is sent and the “Controller” will answer with a response to the node who asked “Node 3”.

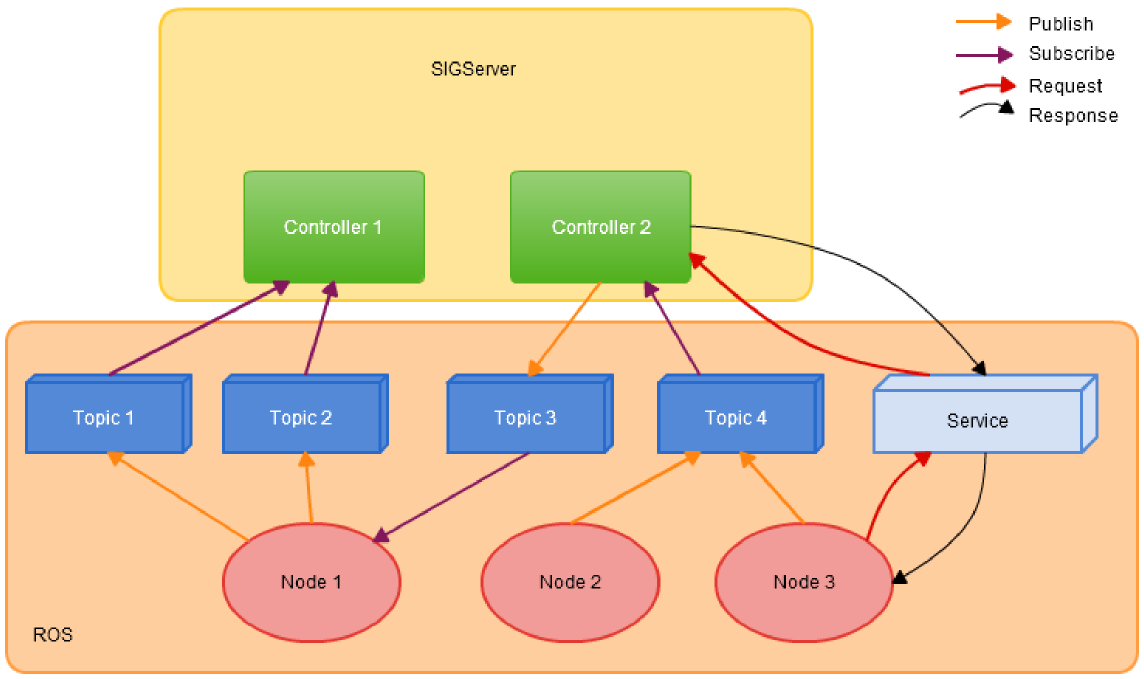


Figure 3.2: Many ROS nodes send and receive information to many controllers

3.2.2 Controllers used

As explained earlier, one “Controller” is associated to one agent for making him act. This association is defined in the xml file which describes the agent.
The same “Controller” can be associated to several agents, that means that all agent associated to this “Controller” will act the same.
On this project, “Controllers” has to be developped for creating topics which sent information and make the agent acting. So, two “Controllers” will be necessary, one for the robot and one for the object. For generals methods like getting the all entities, they can be included on the object controller.
The best would be a general controller for the generals methods to avoid the duplication of topics, but for this mid-term report, only the robot controller and the object controller are implemented with the generals methods inside them.
So, in our case, I have as many topics (or services) for getting entities as number of agent in the simulator.

The robot “Controller” inherits from the object “Controller” called “SimObjController”. The reason is that in SIGVerse, a robot is an object and has only two methods more than on the object “Controller”.

3.2.3 Package

ROS is an open source framework who works with packages, every extension is a package. So, the ROS users just need to download the package wanted.
That is why I choose to develop a package, and only running the node called "ros_controller" will be necessary to launch the SIGServer, create the "Controllers", the topics and services.
We can see figure 3.3 the composition of the package.

src : The generic "Controller" for each kind of agent.

srv : The definition of each services needed.

msg : The definition of each messages needed.

tests : Tests files to check the validity of the implementation.

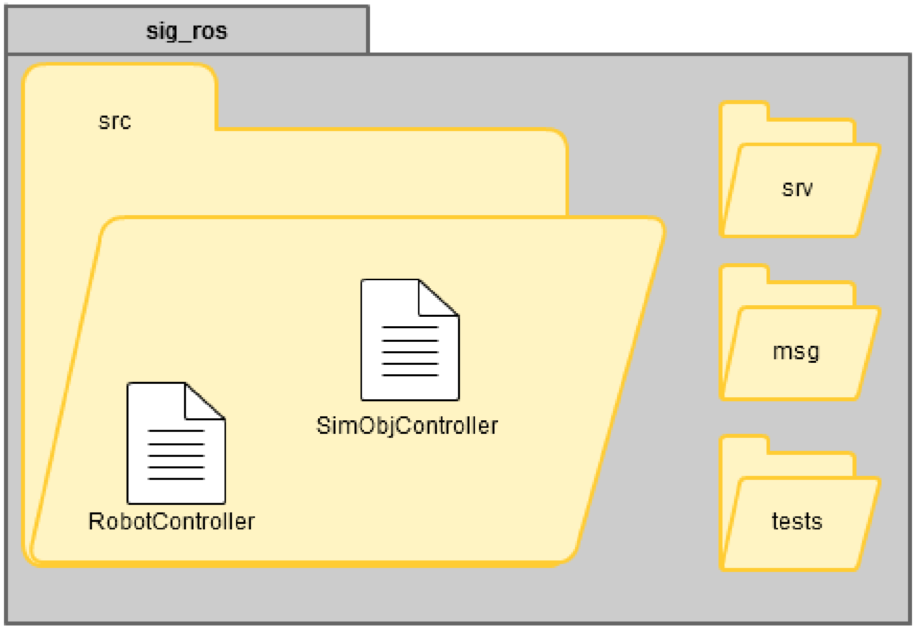


Figure 3.3: sig_ros package

3.3 Usage

From the user point of view, three steps are important, see figure 3.4.
First, the user launches the sig_ros package. So, the SIGServer is automatically launched and the SIGViewer can be connected.
Second, the user starts the simulation from the SIGViewer. So, all topics and services are created and linked to the "Controller" thanks to the package sig_ros.

Finally, the user can create all the ROS node he wants and publish and subscribe to the topics and call services created by the step 2.

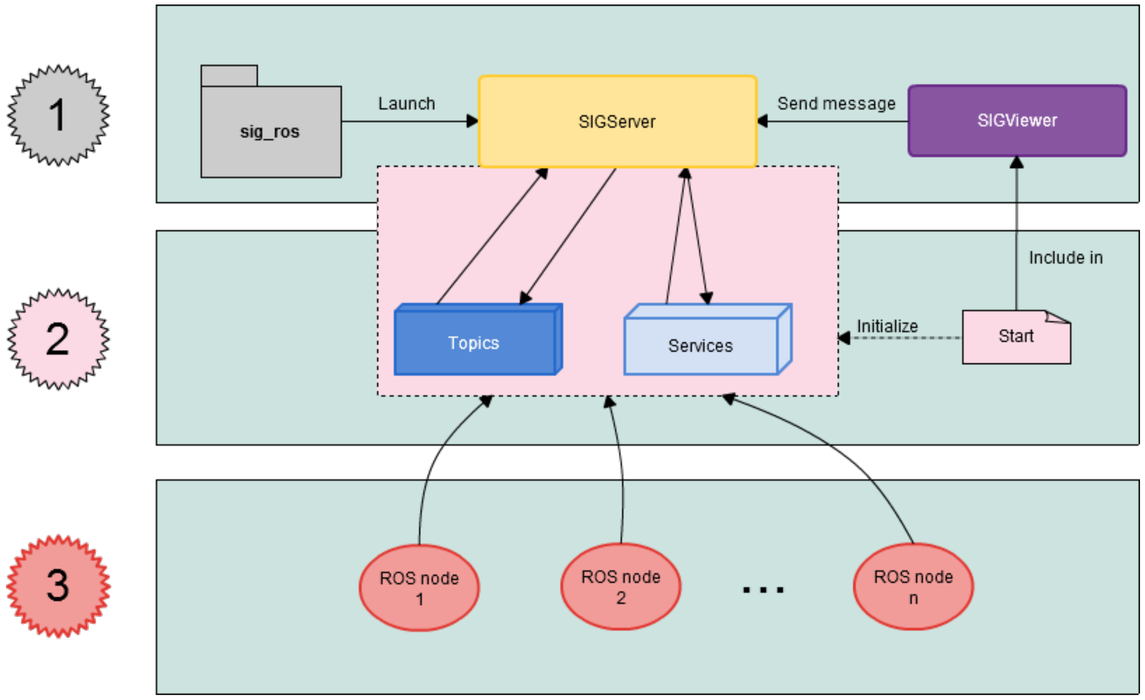


Figure 3.4: Steps to follow to start a simulation with ROS

3.4 Topics & Services

3.4.1 Generalities

Topics and services have to be defined. A node which subscribe to a topic receive the message as soon as it is published to the topic but no answer is given.
On the contrary, the service do the same as the topic bu anwser to the node who called the service.
The types of messages or service request can be of several simple type (double, string, ...) or a combination of these type or message created by these types.
It is possible to create new types, so the tranfert of SIGVerse object could be possible, but the use of the package has to be as easy as possible. So, the methods needed by the clean up task which return bad type for a message are replaced by a topic or service which execute a group of action, for exemple getPart and getPosition applied to the part are mixed in getPartPosition and the user only has to ask for a part and the position is returned with simple type “double”.

The first step is to make the example of the clean up task working, so many topics and services have been created. Each of them starts by the name of the agent and follow by the name of the topic/service.

Now, we are going to see the topics and services I implemented for the robot agent, but a more exhaustive list with more details is given in annex A.3 and annex A.4. In those annexes, there are the topics for the robot agent but also for the object agent of the clean up task which I have started implementing.

3.4.2 Topics

The list of the topics needed for the robot agent of the clean up task is:

_onRecvMsg : The "Controller" send the message received by the SIGViewer.

_onCollisionMsg : The name of the agent which one is in collision with are sent to this topic. If there is severals collision at the same time, severals messages are sent.

_setWheel : Publish the radius and the distance in a message and they will be applied to the robot.

_setWheelVelocity : Publish the velocity for the left and the right wheel and it will be applied.

_setJointVelocity : set the velocity "angular velocity" to the join called "jointName".

_releaseObj : Publish the part which you want to release an object and it will be done.

The two first topics was obvious, they are the unique topics where the "Controller" publish. Indeed, the dynamic of the "Controller" run two methods when particular events occurs, see section 3.1. That is why, each time it occurs, messages are sent to this topics.

3.4.3 Services

The list of the services needed for the robot agent of the clean up task is:

_get_time : Get the simulation time.

_get_obj_position : Get the position of the object named name, if name is empty, return the position of the agent which the service's name start with.

_get_parts_position : Get the position of the part in parameter.

_get_rotation : Get the quaternion of the agent's rotation.

_get_angle_rotation : Get the angle of the rotation of the agent.

_get_joint_angle : Get the angle between the joint.

_grasp_obj : Grasp the object “obj” with the part “part”.

3.5 Internship progress

3.5.1 Discovery

The first two weeks were dedicated to the installation of the environment: Operating System, Virtual Machine, SIGServer, SIGViewer. After that, I could see how SIGVerse works.

Ones the environment installed, I was able to run examples of the wiki page [1]. With this examples, I could see the function of the “Controller” and then understand how an agent can act, changing places, saying “Hello”,... But also, how an agent communicates in both ways, sending and receiving messages.

After installing the environment and learning how make agents and make them move, I had to know what was ROS and how it is working. So, I installed its and I did the all beginners tutorials of the wiki page [2].
After that, I could run an example of SIGVerse running with ROS and then beginning to investigate how to design an interface between SIGVerse and ROS.

3.5.2 Tools & organisation

I'm developping on a virtual machine Ubuntu 12.04 because of the more stable version where SIGVerse works well. No IDE[1] is used, only a text editor gedit and a terminal for compilation. ROS and SIGVerse are open source so, I decided to use GitHub.
I'm using Windows 8.1 for the compatibility with the kinect which I will need by the end of the project.

The project was very free, my supervisor showed me the simulator SIGVerse and the first step after the installation was finding a subject. Because of the short time due to the DoW[2] deadline, I chose to work on this project.

[1]Integrated Development Environment
[2]Description of Work

Once a week, the all laboratory attends a meeting where each of members expose what he planed to do last week, what he actually did and what he will do the next week. I make my own objectives every week for achieving the main goal.
My supervisor answers to my questions and gives me indications during the meeting.

I make unit tests with ROS to be sure of the good functioning.

3.5.3 Schedule

What I planned

In the DoW report, I planned to achieve the job L2 before the mid-term report deadline. That means make SIGVerse works with ROS by a simple way like one "Controller" with one node and generate a "Controller" for each agent on the simulator.

For the next months, I planned to design and develop the topics and services that are necessary to control the agents. I also planned to make an interface between the real human and the simulator. For example, making work the kinect. We can see figure A.2 the schedule of the DoW.

What I did

Finally, this part was easier than I though and I could start the next job. So, I started implementing some topics and services for the clean up task.
Only the methods for the clean up task have been done and some others. You can see at `https://www.youtube.com/watch?v=PL4MCjire2M&feature=youtu.be` a demonstration of the clean up task. The all actions of the robot are sent by ROS.

For the next months, I will follow the DoW schedule, starting with finishing the clean up task example. The time saved by the first two month will be usefull for making more tests.

Troubles

During this two first month, I had to solve some troubles.
First of all, due to my lack of knowledge about ROS, it was very difficult to gain a full view of the project at the beginning. That is why, I had to spend some days to learn from the ROS tutorials and to practice with this framework.

When I started implementing the “Controller”, I had a problem of inheritance. Indeed, I had to inherit the robot “Controller” from the object “Controller” and keep this two “Controller” usable for an agent. This inheritance was not easy because of ROS.
I found a SIGVerse bug using a function “getSimulationTime()”. I spent a lot of time diagnosing this bug because I though it came from my code but finally it was a SIGVerse error. So, I made a bug report.
The documentation of SIGVerse is in japanese, so it is quiet difficult to understand what every method does, so I did a lot of tests to figure it out.

4 Conclusion

These two first month of internship make me see a research environment: discovering a new project, finding a subject and designing a solution by myself.
I learnt a lot with the autonomy I had: designing, implementing, testing and finding a solution to the bugs who occured, but also technical skills like C++.

The interface I began to develop has to be functional by the end of the internship. Indeed, ROS users will use my package or may be SIGVerse users who thinks using ROS is an easier way to use SIGVerse. This objective of achievement is very motivated.

For the rest of the internship, many things have to be done, finishing the clean up task, testing, designing and developping the interaction human-SIGVerse.

Bibliography

[1] SIGVerse wiki page : `http://www.sigverse.org/wiki/en/index.php?Tutorial`.

[2] ROS wiki page : `http://wiki.ros.org/fr/ROS/Tutorials`.

[3] SIGVerse wiki page ROS integration tutorial : `http://www.sigverse.org/wiki/en/index.php?ROS%20integration`.

Abstract

I did this internship in the National Institute of Informatics, an inter university research institute which aim to develop the research in multiple domains. But more specifically, in the Inamura lab.
My subject was making an interface between ROS and SIGVerse for growing the SIGVerse community. ROS is a framework for writing robot software and SIGVerse a simulator.
The interface is made in C++ and I tried to make it very easy for the ROS users. Their knowledge about SIGVerse can be very limited or totally inexistant. The ROS users will be able to use SIGVerse as easy as possible. They only have to publish or subscribe topics or to call services that I defined like any ROS package.
This project make me very responsible and independant in the area of sofware design, need analysis...

J'ai réalisé ce stage dans le laboratoire d'Inamura, au National Institute of Informatics, un institut de recherche dans divers domaines.
Le but de ce projet est de créer une interface entre ROS et SIGVerse afin d'agrandir la communauté de SIGVerse.
ROS est un framework pour le développement de logiciel pour les robots et SIGVerse un simulateur.
L'interface est réalisée en C++ et son utilisation doit être la plus simple possible pour un utilisateur ROS, c'est-à-dire sans ou peu de connaissances de SIGVerse prérequises. Les utilisateurs ROS doivent être capable de communiquer avec SIGVerse sans difficultés, juste en publiant et/ou souscrivant à des "topics" ou appelant des services.
Ce projet m'a permis d'appliquer mes connaissances de manière responsable et autonome dans l'analyse des besoins, conception...

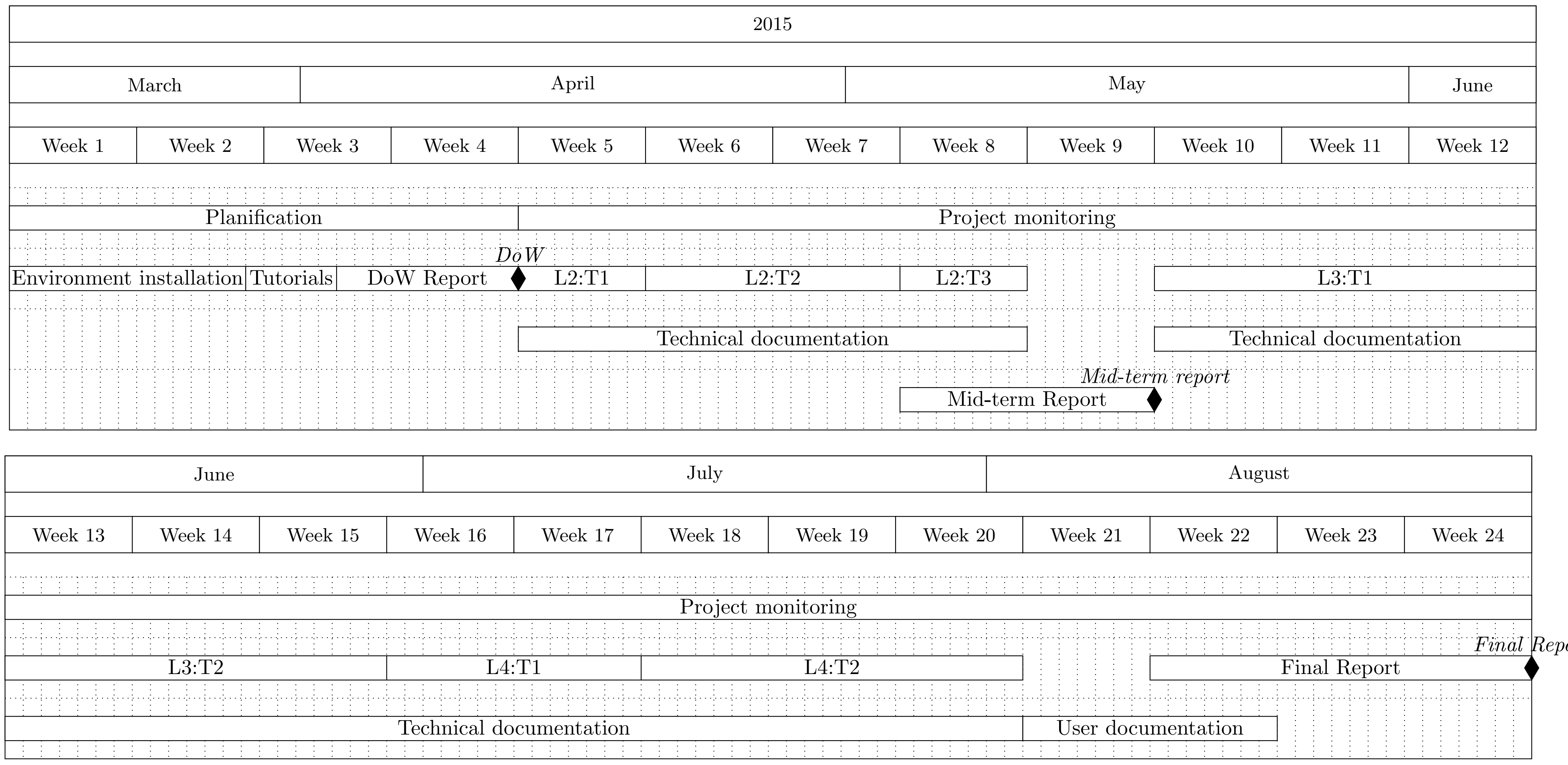
A Annex

A.1 Jobs

N	Title	Start	End
L1	Management		
T1	Planification	S1	S4
T2	Project monitoring	S5	S24
L2	SIGServer and agents integration		
T1	Run SIGServer from ROS	S5	S5
T2	Generation of one controller associated to one agent	S6	S7
T3	Generation of controller for each agents	S8	S8
L3	Make agents act		
T1	Though Publish/Subscribe	S10	S12
T2	Though ROS Services	S13	S15
L4	Services interface		
T1	Design	S16	S17
T2	Kinect	S18	S20
L5	Documentation		
T1	Technical documentation	S5	S18
T2	User documentation	S19	S23
L6	Report		
T1	Mid-term report	S8	S9
T2	Final report	S22	S24

Table A.1: Jobs

A.2 Schedule


2015
March
April
May
June
Week 1
Week 2
Week 3
Week 4
Week 5
Week 6
Week 7
Week 8
Week 9
Week 10
Week 11
Week 12
Planification
Project monitoring
DoW
Environment installation
Tutorials
DoW Report
L2:T1
L2:T2
L2:T3
L3:T1
Technical documentation
Technical documentation
Mid-term report
Mid-term Report
June
July
August
Week 13
Week 14
Week 15
Week 16
Week 17
Week 18
Week 19
Week 20
Week 21
Week 22
Week 23
Week 24
Project monitoring
Final Repo
L3:T2
L4:T1
L4:T2
Final Report
Technical documentation
User documentation

A.3 Topics

Topic name	Message	Description
_onRecvMsg	**sender** : string **content** : string	The "Controller" send the message received by the SIGViewer.
_onCollisionMsg	**name** : string **part** : string	The name of the agent which one is in collision with are sent to this topic. If there is severals collision at the same time, severals messages are sent.
_setWheel	**wheelRadius** : double **wheelDistance** : double	Publish the radius and the distance in a message and they will be applied to the robot.
_setWheelVelocity	**leftWheel** : double **rightWheel** : double	Publish the velocity for the left and the right wheel and it will be applied.
_setJointVelocity	**jointName** : string **angularVelocity** : double **max** : double	jointName, angular velocity, max ???
_releaseObj	**arm** : string	Publish the part which you want to release an object and it will be done.
_setAxisAndAngle	**name** : string **axisX** : double **axisY** : double **axisZ** : double **angle** : double	Set the axis defined by "axisX", "axisY" and "axisZ" and set the angle "angle" to the entity called "name", if no name is provided, the main entity of the topic will be set.
_setPosition	**name** : string **posX** : double **posY** : double **posZ** : double	Set the position "posX", "posY" and "posZ" to the entity called "name", if no name is provided, the main entity of the topic will be set.

A.4 Services

Service name	Request	Response	Description
_get_time		**time** : double	Get the simulation time.
_get_obj_position	**name** : string	**posX** : double **posY** : double **posZ** : double	Get the position of the object named name, if name is empty, return the position of the agent which the service's name start with.

_get_parts_position	**part** : string	**posX** : double **posY** : double **posZ** : double	Get the position of the part in parameter.
_get_rotation	**axis** : string	**qW** : double **qX** : double **qY** : double **qZ** : double	Get the rotation of ...
_get_angle_rotation	**axis** : string **x** : double **y** : double **z** : double	**angle** : double	Get the angle of ...
_get_joint_angle	**nameArm** : string	**angle** : double	Get the angle between the joint.
_grasp_obj	**part** : string **obj** : string	**ok** : bool	Grasp the object "obj" with the part "part"
_get_entities	**axis** : string	**entitiesNames** : string[] **length** : int	Get the names of the entities in the simulator.
_check_service	**serviceName** : string	**connected** : bool	Check if the service "serviceName" is connected.
_connect_to_service	**serviceName** : string	**connected** : bool	Connect the "serviceName", true if it is connected, false otherwise.
_get_collision_state _of_main_part		**collisionState** : bool	Get the collision state of the main part.
_is_grasped	**entityName** : string	**answer** : bool	True if "entityName" is grasped, false otherwise. If no entity name is provided, it will return the answer for the agent which is asked