RoboCup Rescue Simulator Tutorial

Annibal B. M. da Silva, Luis G. Nardin, Jaime S. Sichman

Laboratório de Técnicas Inteligentes – Escola Politécnica da USP
Av. Prof. Luciano Gualberto, travessa 3, nº 158
05508-970 – São Paulo – SP – Brazil

{annibal.silva, luis.nardin}@usp.br, jaime.sichman@poli.usp.br

Contents

1 Introduction 3

2 Installing and running the simulator 3
2.1 Software needed . . . 4
2.2 Installing the simulator . . . 4
2.3 Running the simulator . . . 5

3 The simulator architecture 6

4 Entities of the simulator 8
4.1 Blockades . . . 8
4.2 Area . . . 9
4.2.1 Buildings . . . 9
4.2.2 Roads . . . 11
4.3 Humans . . . 11
4.3.1 Civilians . . . 12

5 Agents of the simulator 12
5.1 Platoon agents . . . 12
5.1.1 Ambulance Team . . . 12
5.1.2 Fire Brigade . . . 12
5.1.3 Police Force . . . 12
5.2 Central agents . . . 12

6 Agent perception and commands 13

7 Communication **13**

8 Tips for programming the rescue team **14**
8.1 Starting a project 14
8.2 Basic contents 15
8.2.1 AbstractSampleAgent 15
8.2.2 Agent classes 16
8.3 Tinkering the rescue team 16

Appendices **18**

A RoboCup Rescue Simulator Communication Library (RCRSCS) manual **18**
A.1 Introduction 18
A.2 Concept 19
A.3 Installation 20
A.4 Usage 20
A.4.1 Internal Notes 21
A.5 Messages 21
A.5.1 Information Message 21
A.5.2 Task Message 22
A.5.3 Report Message 23
A.6 References 23

1 Introduction

A *disaster* may be defined as a crisis situation that causes serious damage of great extensions of human, material, economical and environmental losses to a social system, with these impacts surpassing the system's capacity to recover. There are two main ways of treating a disaster:

1. Preventing them from happening: this can be done through governmental and communitary organization, identifying causes for certain events and preventing them from taking place.

2. Managing their impacts once they happen, through the application of previously established policies and strategies.

The former option is usually the desirable one: analysing the causes of an event to preventive policies that are incorporated in the society's normal operation can be very effective and very little disrupting. This is not, however, always possible to be done, since some events are too intense to be avoided even if forecasted with great antecedence. The latter option is therefore necessary for the mitigation of the disaster's impacts.

After the events of the 1995 Kobe earthquake, Japanese researchers came up with the idea of developing a simulator of the conditions of a urban post-earthquake scenario, so that strategies to counter the consequences of these disasters (by fighting fires, rescueing victims stuck under construction debris and cleaning obstructed roads) could be tested at will. They also organized a competition, the RoboCup Rescue Agent Simulation League, where people from all over the world would develop teams of agents to run in this simulator.

The objective of this document is to facilitate the first contact with the RoboCup Rescue Agent simulator, since official documentation is scarce and this can be a real buzzkiller for beginners; it was done with the 2011 version in mind (called *rescue-1.0a-2011*). It is aimed at people interested in participating in competitions, but hopefully it will also help people who want to modify the plataform and use it to research purposes.

2 Installing and running the simulator

This tutorial will assume you are running in a Linux machine; using the simulator in a Windows OS is possible, though, but you will not have any problems to install and use a Linux distribution, since they are all free and most of them have a good community of users who sane each others' doubts. If you have never used Linux before, we recommend starting with a user-friendly distribution, such as Ubuntu[1] or Fedora[2].

[1]`http://www.ubuntu.com/`
[2]`http://fedoraproject.org/`

2.1 Software needed

To run the simulator you will need the following software installed:

- Java JDK 1.6 or higher (either the official one, provided by Oracle or a free one, like OpenJDK);
- ant 1.8 or higher;
- rng-tools (not strictly necessary, but strongly recommended as it will allow a much faster connection of the agents with the simulator);

If you are using Ubuntu, all of them are present in the default software repositories, except for the Oracle JDK (as of Ubuntu 12.04), which makes them easy to be downloaded and installed; just open a terminal window, type

```
sudo apt-get install openjdk-7-jdk ant rng-tools
```

and enter your user password when asked to. It is recomendable, however, to use the Oracle version the JDK; a guide for an easy install can be found here[3].

2.2 Installing the simulator

1. Download the simulator at the RoboCup Rescue sourceforge project page[4].
2. Extract the content of the file you have just downloaded into a directory, like your Home folder (the one that has the same name as your username), for instance.
3. Open a terminal window, move to the folder created by the simulator (this can be done by typing `cd rescue-1.0a-2011`), and enter the comand

   ```
   ant
   ```

   This will compile the simulator.
4. In your terminal, enter the command

   ```
   sudo gedit /etc/defaults/rng-tools
   ```

   Enter your user password and a text file will be opened. Type the line

   ```
   HRNGDEVICE=/dev/urandom
   ```

   anywhere in the file and close it, saving the changes made. Then enter the command

[3] `http://www.webupd8.org/2012/01/install-oracle-java-jdk-7-in-ubuntu-via.html`
[4] `http://sourceforge.net/projects/roborescue/files/`

```
/etc/init.d/rng-tools start
```

in your terminal to start the rng daemon.

5. Open the simulator's and access the `boot` folder; edit the `start.sh` file and replace the lines

   ```
   startKernel
   startSims
   ```

   by the lines

   ```
   startKernel --nomenu
   startSims --nogui
   ```

The simulator is now installed and good to go!

2.3 Running the simulator

To run the simulator, you must open the terminal, move to the `boot` folder inside the simulator's folder and type the following command:

```
./start.sh [OPTIONS]
```

Where [OPTIONS] are optional parameters that can be one or all of the following:

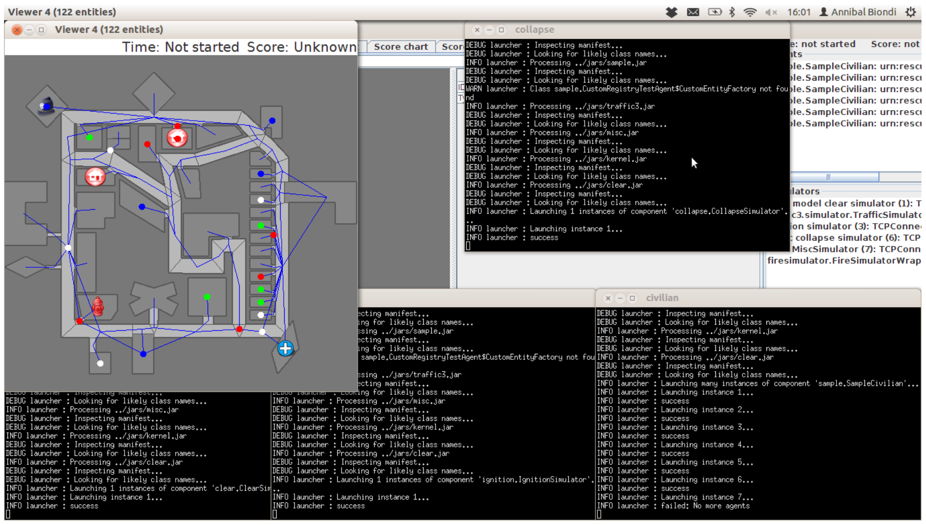
- `-m MAPDIR`, where `MAPDIR` is the path to the folder containing the map you want to run (default is `../maps/gml/test`).
- `-l LOGDIR [LABEL_OPTIONS]`, where `LOGDIR` is the folder where the log files will be stored (default is `/logs`). `LABEL_OPTIONS` are the following:
  - `-s`, to store the log archives in a folder named with the time and date of the simulation inside `LOGDIR`.
  - `-t TEAMNAME`, where `TEAMNAME` is the name of the team to appear in the name of the folder created by `-s` (default is none).

The simulator, however, does not work if `-l` and its options are used, so you should not use them. An example of a run command would be

```
./start.sh -m ../maps/gml/berlin
```

You will get something like figure 1.

We must now launch the agents that will run in the simulator; let us use the ones provided by the simulator itself. Open a new terminal window, move to



```
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/sample.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
WARN launcher : Class sample.CustomRegistryTestAgent$CustomEntityFactory not fou
nd
INFO launcher : Processing ../jars/traffic3.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/misc.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/kernel.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/clear.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Launching 1 instances of component 'collapse.CollapseSimulator'.
..
INFO launcher : Launching instance 1...
INFO launcher : success
```

```
e: not started     Score: not
nts
ple.SampleCivilian: urn:resc
ple.SampleCivilian: urn:resc
ple.SampleCivilian: urn:resc
ple.SampleCivilian: urn:resc
ple.SampleCivilian: urn:resc
ple.SampleCivilian: urn:resc
ulators
 model clear simulator (1): T
c3.simulator.TrafficSimulat
ion simulator (3): TCPConne
 collapse simulator (6): TCP
 MiscSimulator (7): TCPConn
firesimulator.FireSimulatorWrap
```

```
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/misc.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/kernel.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/clear.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Launching 1 instances of component 'clear.ClearSim
INFO launcher : Launching instance 1...
INFO launcher : success
```

```
ecting manifest...
ing for likely class names...
ssing ../jars/sample.jar
ecting manifest...
ing for likely class names...
 sample.CustomRegistryTestAgent$CustomEntityFactory not fou

ssing ../jars/traffic3.jar
ecting manifest...
ing for likely class names...
INFO launcher : Processing ../jars/misc.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/kernel.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/clear.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Launching 1 instances of component 'ignition.IgnitionSimulator'.
..
INFO launcher : Launching instance 1...
INFO launcher : success
```

```
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/kernel.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Processing ../jars/clear.jar
DEBUG launcher : Inspecting manifest...
DEBUG launcher : Looking for likely class names...
INFO launcher : Launching many instances of component 'sample.SampleCivilian'...
INFO launcher : Launching instance 1...
INFO launcher : success
INFO launcher : Launching instance 2...
INFO launcher : success
INFO launcher : Launching instance 3...
INFO launcher : success
INFO launcher : Launching instance 4...
INFO launcher : success
INFO launcher : Launching instance 5...
INFO launcher : success
INFO launcher : Launching instance 6...
INFO launcher : success
INFO launcher : Launching instance 7...
INFO launcher : failed: No more agents
```

Figure 1: Example of the simulator running

the `boot` folder inside the simulator and type the command

```
./sampleagents.sh
```

Now just hit the *Run* button in the Kernel GUI and the simulator will start. To terminate the execution of the simulator or the agents' script, select the terminal window and press `Ctrl-C`.

3 The simulator architecture

The simulator is in fact divided into many other simulators; these are:

- the *clear* simulator, responsible for blockade removal;
- the *collapse* simulator, responsible for managing buildings' structural damage and blockade creation;
- the *ignition* simulator, responsible for firing up random buildings during the simulation;
- the *fire* simulator, responsible for the fire spread between buildings;
- the *traffic* simulator, responsible for humans' movement;
- the *misc* simulator, responsible for human damage and buriedness.

These simulators establish connections to *kernel*, responsible for coordinating the simulators' processes and centralizing the data they generate.

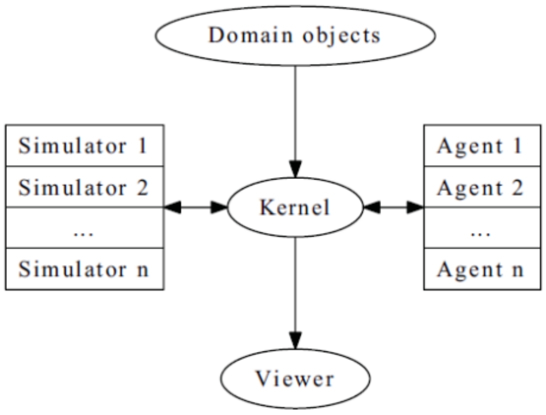


Figure 2: RoboCup Rescue Agent Simulation platform architecture[1]

The simulator was designed to create a *partially observable*, *discrete-time*, *dynamic*, *stochastic*, *multiagent* environment. In other words, in this environment:

- its current state cannot be completely known through a single agent's perception (even if the agent had an infinite range of sight, it still would not be able to see through a building's wall);
- time is divided in intervals, as opposed to continuous time;
- changes may occur while the agent takes its time to make decisions;
- there are random elements that affect its state transition;
- there is more than one agent present, and one's actions may interfere with the others' performance.

Time is divided in *timesteps*; during each timestep, the agent perceives the environment and reasons about what action it will perform. In each timestep, the following happens:

1. The kernel updates all agents' perception (visual and communication) and waits for the agents' commands.
2. The agents updates their world model and make their decisions, sending their commands to the kernel.
3. The kernel sends the agents' commands to the simulators.
4. The simulators process the agents' commands and send the changes suffered by the environment.

5. The kernel broadcasts the current world model, resultant of the information from all the simulators to all of them.
6. The kernel sends the agents' commands the environment changes to the viewers.

Important directories of the simulator are:

- `/boot`: scripts of execution of the simulator
- `/boot/config`: configuration files of the simulator
- `/boot/logs`: log files
- `/build`: the simulator's Java classes
- `/docs`: *javadoc* documentation of the simulator
- `/jars`: the simulator's JAR files
- `/lib`: libraries used by the simulator
- `/maps`: maps that can be ran in the simulator
- `/modules`: the simulator's source code
- `/oldsims`: source code of some of the simulator's older versions

4 Entities of the simulator

Several objects are represented in the simulator: it is important to know what their characteristics are.

4.1 Blockades

Blockades obstruct the path of agents and civilians; they are represented as black polygons in roads. Blockades appear in the beginning of the simulation and are not produced after this. They must be removed by Police Forces.

Properties:

- position: ID of the road to which the blockade belongs
- repair cost: cost to completely remove the blockade from the road
- shape: a rectangle which surrounds the whole blockade
- X & Y: coordinates of the blockade's centroid
- apexes: vector containing the apexes of the blockade

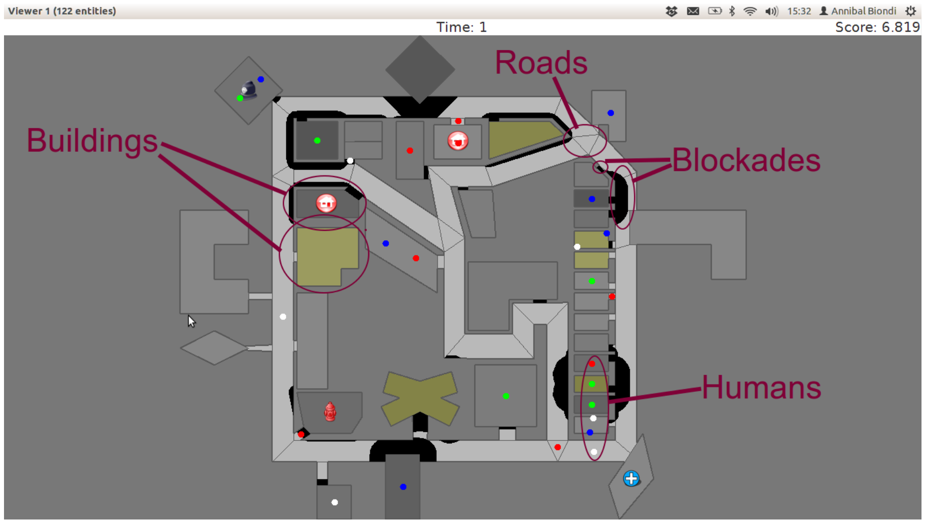


Figure 3: Entities of the simulator

4.2 Area

Area entities represent buildings and roads.

Properties:

- blockades: a list with the blockades in that area
- edges: a list with the edges that limit the area
- neighbours: a list of the areas that can be accessed from this area
- X & Y: coordinates representing the area in the map

While both buidings and roads have the *blockades* attribute, blockades appear only in roads.

4.2.1 Buildings

Buildings group all kinds of buildings in the simulator: besides the regular ones, described below, there are special kinds of buildings (*refuges*, *ambulance centres*, *fire stations* and *police offices*, shown in figure 4) which cannot catch on fire. They will be described in later sections of this document.

Properties:

- brokeness: how structurally damaged the building is; does not change during the simulation
- fieryness: the intensity of the fire and fire-related damage in the building
    - UNBURNT (not burnt at all)

(a) Refuge (b) Ambulance Central
(c) Fire Station (d) Police Office

Figure 4: Special buildings

  - WATER_DAMAGE (not burnt at all, but has water damage)
  - HEATING (on fire a bit)
  - BURNING (on fire a bit more)
  - INFERNO (on fire a lot)
  - MINOR_DAMAGE (extinguished but minor damage)
  - MODERATE_DAMAGE (extinguished but moderate damage)
  - SEVERE_DAMAGE (extinguished but major damage)
  - BURNT_OUT (completely burnt out)

- floors: the number of floors the building has; all the ones observed have only one
- ground area: the area of each floor
- ignition: indicates if the simulator has lit this building on fire[5]
- importance (unknown function; has equal values to all buildings)
- temperature: temperature of the building; if it crosses a certain treshold, the building catches on fire
- total area: the total area of the building ($floors \times ground\ area$)

Regular buildings are represented as polygons of various colors, depending of their status, as shown in figure 5; the darker the color, the greater the structural fire or water damage.

In the beginning of the simulation, broken buildings trap humans inside it under debris; these debris must be removed by ambulance forces, who must then proceed to rescue the human.

A *refuge* is a special kind of building: it represents a place destined to support the rescue activity, providing medical care for the wounded and water to the fire brigades. In the simulator, humans inside a refuge have their damage zeroed, which means they do not lose health while they stay there; damage will, however, resume when the human entity leaves the refuge.

Also, fire brigades have their water supply replenished by a certain amount during each cycle while they are inside the refuge.

[5] A building can catch on fire by being ignited by the simulator or by being close to a burning building; ignition will be set to "1" if the building was, at some pointi of the simulation, ignited by the simulator

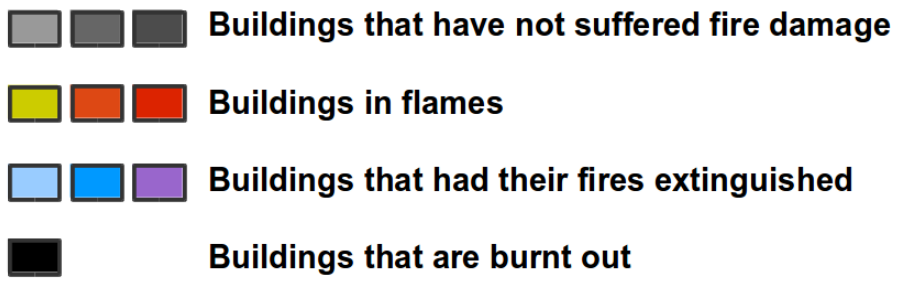

Buildings that have not suffered fire damage
Buildings in flames
Buildings that had their fires extinguished
Buildings that are burnt out

Figure 5: Possible status of regular buildings

4.2.2 Roads

Area entities representing roads; no new attributes besides those of *area* entities.

4.3 Humans

These are the entities representing humans; in the simulator, they can be *civilians*, *ambulance teams*, *fire brigades* or *police forces*. They are all represented by circles of different colors, and cannot move if they are dead or buried.

- buriedness: how deep the individual is buried
- damage: how much HP the individual loses per second; zeroes when a refuge is reached
- direction: direction to where the agent is moving (infered); the Y-axis positive half is zero, and the value increases until 129599 (360*60*60 - 1) seconds anti-clockwise
- HP: health points of the individual; if it reaches 0, he dies
- position: ID of the entity where the individual is; may be an area entity or a human entity (if it is inside an ambulance)
- position history: a list of the entities the individual has passed during the last cycle, in chronological order
- stamina: not implemented; would decrease each time the agent took an action and would be partially replenished at the beginning of each cycle
- travel distance (unknown)
- X & Y: coordinates representing the individual in the map

The color of each human in the simulator is defined by its type and its health: the lower it is, the darker they get. Dead humans are black.

4.3.1 Civilians

Civilians are human entities that are not part of a rescue team; they are represented by the color green. Their standard behaviour is to walk to the closest refuge on their on if they are not wounded or buried; otherwise, they will have to be transported by an ambulance.

5 Agents of the simulator

These are the entities that will compose your rescue team; in other words, this is what you will program. Agents are divided in two types: *platoon agents* and *central agents*.

5.1 Platoon agents

Platoon agents are able to interact with the simulated environment through perception and executing actions on it. They can also exchange messages with other agents by vocal or radio communication. They are comprised of three different categories: the *ambulance teams*, *fire birgades* and *police forces*.

5.1.1 Ambulance Team

Ambulance Teams are responsible for rescueing humans (agents and civilians) and take them to a refuge. They are able to unbury victims and carry one person.

5.1.2 Fire Brigade

Fire brigades are responsible for extinguish fires on buildings. They can carry a certain amount of water and must replenish them in a refuge.

5.1.3 Police Force

Police foreces are responsible for removing blockades from the roads. When ordered to do so, they will remove a certain amount of repair cost from the target blockade at each cycle. Differently from ambulance teams and fire brigades, having two police forces acting on the same blockade brings no advantage to the process: it will be as though there were only one police force acting on it.

5.2 Central agents

Central agents are a type of agents whose only interaction with the world is through radio communication. There are three types of central agents: *ambulance centers*, *fire stations* and *police offices*, and they are represented as buildings.

6 Agent perception and commands

The simulator has two perception modes: *standard* and *line of sight*. We will discuss just the latter, since it is the one used in competitions.

Line of sight perception simulates visual percetion of the agent: a vision range and a number of rays are defined and the agent percepts anything that is reached by these rays.

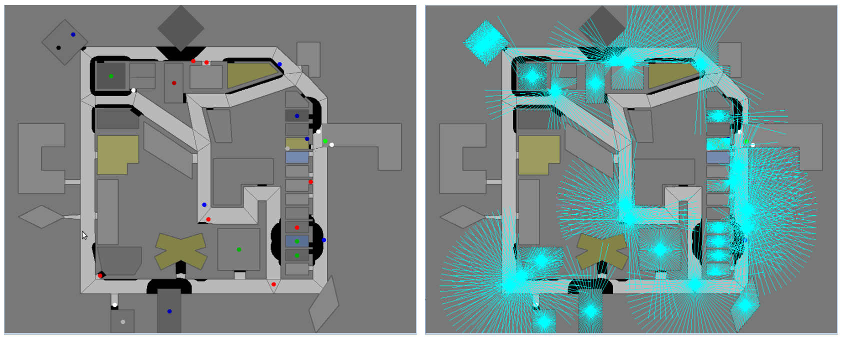

(a) No LoS (b) LoS

Figure 6: Normal viewer and LoS perception

The set of currently visible entities for an agent is stored in a structure named *change set*; entities present in it are automatically updated in its world model; that is, if an agent sees a blockade he did not know that was there before, this blockade is automatically added to its world model. The opposite, though does not happen: if the agent does not see a blockade nothing in its world model changes, even if he knew that there was a blockade there before. In that case, the agent will still think that there is blockade in that road, even though he should be able to see it but does not; it is up to the agent to figure this out and modify its world model accordingly.

7 Communication

There are two forms of communication available in the simulator: *direct communication* and *radio communication*. Direct communication, done with the command *speak*, is communication audible to humans within a radius from the emiter agent, as if the emiter shouted something.

Radio communication is done with the command *tell*, and transmits information to all agents that are signed up to the channel on which it was broadcasted. Radio communication channels are present in limited number, each one with a limited bandwidth.

In both types of communication, the message has to be coded into a string of bytes before being sent; the receptor must decode it once it receives the message. Both types might be succeptible to message *dropout*, where the message is not received by its destinataires; radio communication is also succeptible to message *failure*, where some bytes of information are modified during the transmition proccess.

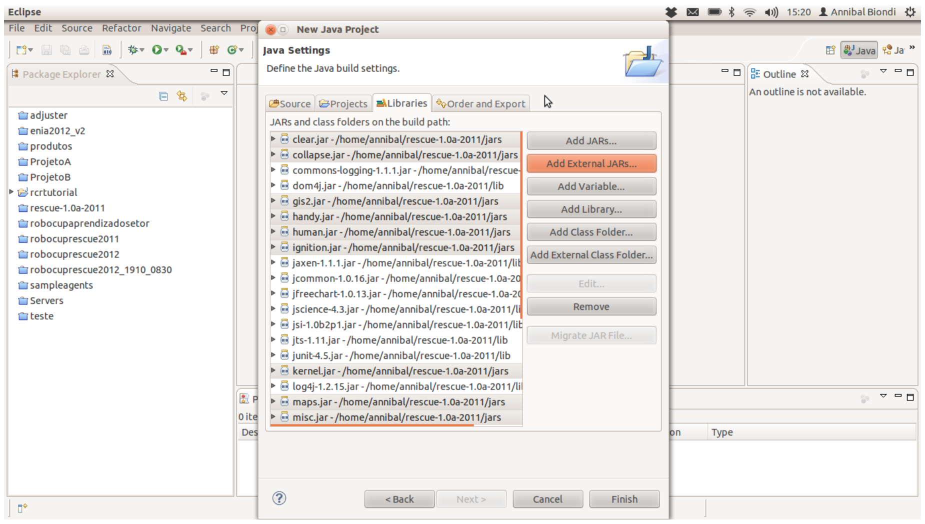
8 Tips for programming the rescue team

When starting to program a rescue team, we advise to start by creating a project and modifying the sample agents provided by the simulator. Programming will be done in Java; there are a lot of IDEs for this language, the most popular being Eclipse[6] and NetBeans[7]. This section will be described using Eclipse.

8.1 Starting a project

After installing Eclipse, you should create a project for developing your rescue team: click on `File > New... > Java Project`; name your project and click on `Next >`.

Select the "Libraries" tab and add all the content from the `jars` and the `lib` folders using the `Add external JARs...` button, as shown in figure 7, and click "Finish".



JARs and class folders on the build path:

```
clear.jar - /home/annibal/rescue-1.0a-2011/jars
collapse.jar - /home/annibal/rescue-1.0a-2011/jars
commons-logging-1.1.1.jar - /home/annibal/rescue-
dom4j.jar - /home/annibal/rescue-1.0a-2011/lib
gis2.jar - /home/annibal/rescue-1.0a-2011/jars
handy.jar - /home/annibal/rescue-1.0a-2011/jars
human.jar - /home/annibal/rescue-1.0a-2011/jars
ignition.jar - /home/annibal/rescue-1.0a-2011/jars
jaxen-1.1.1.jar - /home/annibal/rescue-1.0a-2011/lib
jcommon-1.0.16.jar - /home/annibal/rescue-1.0a-20
jfreechart-1.0.13.jar - /home/annibal/rescue-1.0a-20
jscience-4.3.jar - /home/annibal/rescue-1.0a-2011/li
jsi-1.0b2p1.jar - /home/annibal/rescue-1.0a-2011/lib
jts-1.11.jar - /home/annibal/rescue-1.0a-2011/lib
junit-4.5.jar - /home/annibal/rescue-1.0a-2011/lib
kernel.jar - /home/annibal/rescue-1.0a-2011/jars
log4j-1.2.15.jar - /home/annibal/rescue-1.0a-2011/li
maps.jar - /home/annibal/rescue-1.0a-2011/jars
misc.jar - /home/annibal/rescue-1.0a-2011/jars
```

```
adjuster
enia2012_v2
produtos
ProjetoA
ProjetoB
rcrtutorial
rescue-1.0a-2011
robocupaprendizadosetor
robocuprescue2011
robocuprescue2012
robocuprescue2012_1910_0830
sampleagents
Servers
teste
```

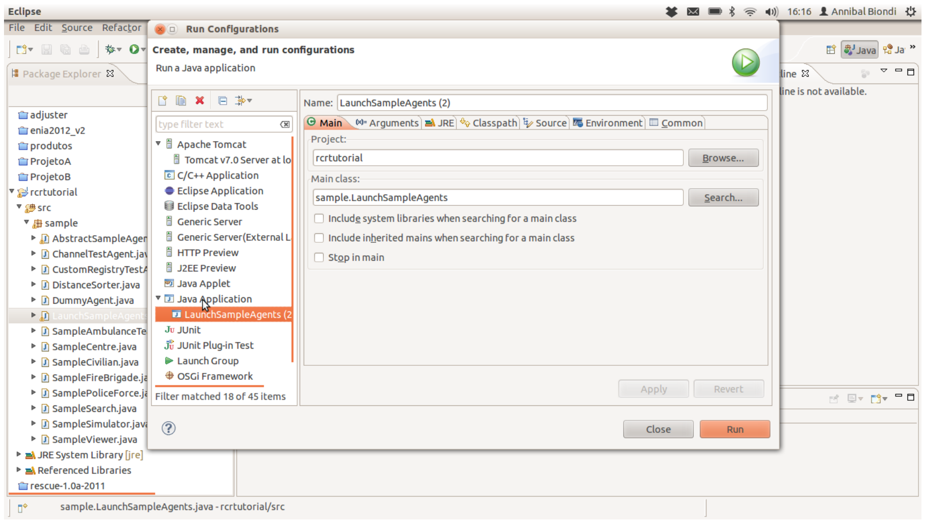
Figure 7: The "Libraries" tab after adding the contents of the `lib` and `jars` folders.

You now have a project, in which you can create and edit Java classes. Let us now copy the source code of the sample agents into it: opening the simulator

[6]`http://www.eclipse.org/`
[7]`http://netbeans.org/`

folder and copy the contents of the `modules/sample/src` folder to the `src` folder of your project. Go back to Eclipse and refresh the project.

Start the simulator, but do not launch any agents; instead, right click on the `LaunchSampleAgents` class inside your Eclipse project and chose the option ”Run As > Run Configurations...”. A new window will be opened; double click the ”Java Application” icon, and press ”Run”. Doing this, you have just launched your project’s agents through Eclipse; once the simulation ends, terminate the application by clicking on the red button on the Console tab.


Run Configurations

Create, manage, and run configurations

Run a Java application

Name: LaunchSampleAgents (2)

Project: rcrtutorial

Main class: sample.LaunchSampleAgents

☐ Include system libraries when searching for a main class

☐ Include inherited mains when searching for a main class

☐ Stop in main

Filter matched 18 of 45 items

sample.LaunchSampleAgents.java - rcrtutorial/src

Figure 8: The ”Run Configurations” window.

With this project, you can look into the source code of the sample agents, make modifications in it and observe the behaviour changes that arise from them.

8.2 Basic contents

Now that the the project is up, let us take a look at some of the contents:

8.2.1 AbstractSampleAgent

There is an abstract class called `AbstractSampleAgent`, which contains methods and attributes that all platoon agents must have: for example, its developers thought it would be good if each agent had lists with all the buildings and roads of the map for quick access instead of retrieving them from the world model everytime they were needed, so they created the `buildingIDs` and `roadIDs` attributes.

The `postConnect` method must prepare the agents for the simulation: the agents are already connected to the simulator and their world model has been created. That means they can get information from it to start their attributes, such as the `buildingIDs` list.

8.2.2 Agent classes

Programming a rescue team revolves around developing the `think` method: this is a method all platoon and central agents must implement, and will determine the reasoning the agent does and the actions it takes during each timestep. Looking into the agent classes (`SampleAmbulanceTeam`, `SampleFireBrigade`, `SamplePoliceForce`), one can see how its behaviour was modeled.

The agents can issue the following commands to the simulator:

- Communication commands:
  - **Say:** the agent issues a message through direct communication;
  - **Tell:** the agent broadcasts a message through radio communication;
- Action commands:
  - **Move:** the agent moves through a determined path;
  - **Rest:** the agents takes no action;
  - **Rescue:** the agent unburies a victim by a certain amount (exclusive to ambulance teams);
  - **Load:** the agent picks up a victim (exclusive to ambulance teams);
  - **Unload:** the agent unloads a victim he is currently carrying (exclusive to ambulance teams);
  - **Extinguish:** the agent directs a certain amount of water to a building (exclusive to fire brigades);
  - **Clear:** the agent removes a certain amount of debris from a blockade (exclusive to police forces).

During a timestep, an agent can issue both communication commands and one action command.

8.3 Tinkering the rescue team

Now that you have a template of how a rescue team is programmed, you can start modifying it to learn about the simulator and create your own team. Think of what could be modified to increase the team's performance, make changes and observe the results.

Also, since communication is not implemented in the sample agents, a good way of observing its functioning is using the *RoboCup Rescue Simulation Communication Library* (RCRSCS), developed by the SUNTORI team. A manual for using this library was added as an appendix at the end of this document.

Some final tips worth mentioning are:

- An agent's world model will be automatically modified if it *percepts* (i.e. sees) something new; it will not change to account that something has gone missing.
- You can obtain a list of all the entitites an agent is currently percepting with the `ChangeSet` class.
- Any command issued by an agent, granted it has acceptable arguments, will be executed by the simulator even if it has no effect on the world; the `extinguish` command, for instance, has no effect if fire brigade that issues it does not carry a large enough quantity of water.

Good luck!

References

[1] Cameron Skinner and Sarvapali Ramchurn. The robocup rescue simulation platform. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, pages 1647–1648, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

Appendices

A RoboCup Rescue Simulator Communication Library (RCRSCS) manual

A.1 Introduction

The RoboCup Rescue Agent Simulator is a discrete-time distributed simulation system and it aims to provide a simulation to reproduce conditions that arise after the occurrence of an earthquake in an urban area. In order to make the simulation more realistic, some issues are considered, such as heterogeneity (different types of rescue agents), dynamic environment (fires spread; injured victims), limited information (agents can only see a short distance), uncertain information (agents do not see the true state of the world), limited communication (messages can be dropped or have noise) and limited processing time (agents have a limited time to issue commands). It is thus characterized as a complex multiagent domain [Kitano et al., 1999].

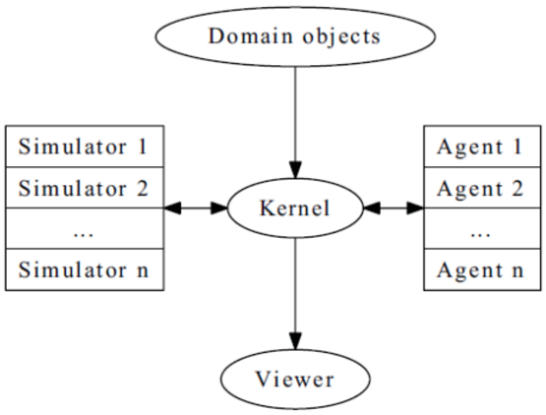


Figure 9: RoboCup Rescue Agent Simulation platform architecture [Skinner and Ramchurn, 2010]

The architecture of the RoboCup Rescue Agent simulator (Fig. 9) is comprised of one kernel, one viewer, domain objects represented in the city map, some sub-simulators, each assigned to a specific task within the simulation, and some agents responsible to carry out tasks on the environment in order to mitigate the disaster impact [Skinner and Ramchurn, 2010]. There are sub-simulators responsible for fire simulation, traffic simulation, collapse simulation, blockage simulation and one responsible for miscellaneous simulations, such as humans' damage and buriedness and road clearing times. Each sub-simulator runs as an independent process, being the simulator kernel responsible for managing the interaction among them through the network. This setting allows for the computational load to be distributed among several computers, which helps the simulation workload distribution.

In order to perform their tasks, the agents are allowed to exchange messages among themselves through communication channels. The simulator allows the agents to use two types of messages: *tell* and *say*. In order to use the tell message type, the agent must be subscribed to a channel and when it sends a *tell* message through this channel, the content is broadcasted via radio to all the platoon agents subscribed to it. On the other hand, the *say* message type limits its receivers to all the agents located within a predefined radius from the sender.

Despite being able to communicate, the simulation platform does not define a protocol to be used for communication, letting the teams free to develop their own communication protocol in order to fulfill their strategic needs. However, such definition and development is time consuming, discouraging many teams to continue to develop their agent teams in order to participate in competitions. Furthermore, the communication protocol is not the teams' goal when they first become interested by such simulation platform.

Therefore, the *RoboCup Rescue Simulation Communication Library (RCRSCS)* was developed by the SUNTORI team in order to minimize this problem allowing the teams to focus their efforts of the strategic part of their teams.

A.2 Concept

In disaster environments, there are different possible ways to control the individuals responsible to perform the actual tasks on the environment. One way of performing such coordination is by delegating it to a central coordinator who is responsible to make decisions about the tasks each individual should take. This coordination structure has several advantages and disadvantages. Among the advantages is the possibility for the central coordinator to select the best task available to each individual since it knows their positions and all the tasks available in the environment; however, this coordination strategy depends heavily on communication and create a single point-of-failure in the system.

Despite the drawbacks listed, it seems to be the most intuitive and simple form of coordination available. Therefore, the RCRSCS was implemented based on such coordination strategy in which the central location is intended to receive all the available information the individuals know from the environment and decide the task each individual should perform at each moment. Besides, the main aim of the library is to relief the teams from developing their own communication protocol before starting develop their team strategy.

Thus, technically speaking, *platoon agents* have a duty to perform actions based on the "Task Message" received from center agents. In addition, platoon agents send the obtained surrounding information (e.g. building, blockade, civilian, and so on) to other agents (platoon agents and center agents). The *center agents* have a duty to analyze the current situation from received information, and send "Task Message" to platoon agents as a result of their analysis. Moreover, the center agents consolidate the information in order to be able to make the bast possible decision about the task a platoon agent should perform [Suntori, 2012].

A.3 Installation

In order to install the RCRSCS library, it is necessary to download it from the website `http://en.sourceforge.jp/projects/rcrscs/`. Once downloaded, you can extract it in any directory and add the `rcrscs.jar` extracted file into your build path in order to be able to use the RCRSCS communication library.

A.4 Usage

In order to use the RCRSCS communication library, you must:

1. Extend the *AbstractCSAgent* class instead of the *StandardAgent* when developing a platoon or center agent;
2. Override the following two methods:
   (a) `protected EnumSet<StandardEntityURN> getRequestedEntityURNEnum()`
   (b) `protected void thinking(int time, ChangeSet change, Collection<Command> heard)`
3. Optionally, override the following method:
   (a) `void postConnect()`

For example,

```
public class simpleAmbulanceTeamAgent extends AbstractCSAgent<
    AmbulanceTeam>{

    @Override
    protected EnumSet<StandardEntityURN> getRequestedEntityURNsEnum(){
        return EnumSet.of(StandardEntityURN.AMBULANCE_TEAM);
    }

    /**
     * Implements the agent specific post connection code
     */
    @Override
    protected void postConnect(){
        ...
    }

    /**
     * Implements the agent specific processing
     */
    @Override
    protected void thinking(int time, ChangeSet change, Collection<
        Command> heard){
        ...
    }
}
```

There are two methods that should be used in order to enable the RCRSCS library functionalities. The methods are:

1. `setMessageChannel(int channel)`
   This method indicates to the RCRSCS library which channel the agent is going to use in order to send and receive messages. It should be used only once, since all the message sends and receives are going to use this channel as a reference.

2. `addMessage(RCRSCSMessage msg)`
   This method adds a message into a queue. At the end of the `thinking` method processing, the RCRSCS library automatically sends all the messages in this queue through the channel set using the `setMessageChannel` method. There are several types of `RCRSCSMessage` available, which is described in section Message.

The RCRSCS communication library makes available one attribute that allows the access to the heard messages from the channel set in the `setMessageChannel`. The attribute name is `receiveMessageList` and it is a list of `RCRSCSMessage` and it should be accessed directly in the agent code.

A.4.1 Internal Notes

The `AbstractCSAgent` extends the `StandardAgent` and implements the `think` method as required by the latter class.

```
@Override
/**
 * Represent each step thinking.
 * (1.Receive message, 2.think, 3.send new messages)
 */
protected final void think(int time, ChangeSet changed, Collection<
    Command> heard) {
    super.receiveMessage(heard);
    this.thinking(time, changed, heard);
    super.sendMessage(time);
}
```

Basically, what this code does is to filter all the messages in the `heard` collection and add to the `receiveMessageList` attribute the messages received from the channel set using the `setMessageChannel` method. Next, it executes the implemented `thinking` method from the agent. Finally, it sends all the messages added to the queue during the thinking processing to the channel set using the `setMessageChannel`.

A.5 Messages

The RCRSCS communication library predefines three types of messages that can be sent: Information Messages, Task Messages and Report Messages.

A.5.1 Information Message

The Information Message represents information from the disaster space. It does not include static information from those objects in order to avoid the transmission of unnecessary data, thus reducing the size of message sent. They have specific meaning in the context of the RCRSCS communication library and parameters as described below.

- **AmbulanceTeamInformation(int time, EntityID atID, int hp, int damage, int buriedness, EntityID areaID)**

  Ambulance Team information

- **BlockadeInformation(int time, EntityID blockadeID, EntityID roadID, int repairCost)**

  Blockade information

- **BuildingInformation(int time, EntityID buildingId, int fieryness, int brokenness)**

  Building information

- **FireBrigadeInformation(int time, EntityID fbID, int hp, int damage, int buriedness, int water, EntityID areaID**

  Fire Brigade information

- **PoliceForceInformation(int time, EntityID pfID, int hp, int damage, int buriedness, EntityID areaID)**

  Police Force information

- **PositionInformation(int time, EntityID platoonID, Pair<Integer, Integer> cor)**

  Rescue agent location information

- **TransferInformation(int time, EntityID platoonID, EntityID... areas)**

  Pathway information

- **VictimInformation(int time, EntityID vicID, EntityID area, int hp, int buriedness, int damage)**

  Victims information

A.5.2 Task Message

The Task Message represents orders the platoon agents should perform. This message does not directly change the agent behavior, but it provides the direction the agents should follow to cooperate among the other agents. They have specific meaning in the context of the RCRSCS communication library and parameters as described below.

- **ClearRouteTaskMessage(int time, EntityID ownerID, EntityID pfID, EntityID departure, EntityID destination)**

  Command a Police Force to clear roads from an area A to an area B

- **RescueAreaTaskMessage(int time, EntityID ownerID, EntityID atID, EntityID... areas)**

  Command a Ambulance Team to rescue a civilian

- **ExtinguishAreaTaskMessage(int time, EntityID ownerID, EntityID fbID, EntityID... areas)**

  Command a Fire Brigade to extinguish fire from a specific building

A.5.3 Report Message

The Report Message reports the order results sent to the agents using a Task Message. The types of Report Message, their parameters and their meanings are described below.

- **DoneReportMessage(int time, EntityID platoonID)**

  This message indicates the agent has accomplished the task assigned to it

- **ExceptionReportMessage(int time, EntityID platoonID)**

  This message indicates the agent was unable to complete the task assigned to it

IMPORTANT NOTE: Despite the Report Messages being defined, they do not actually work in the RCRSCS v1.0. The use of `DoneReportMessage` or `ExceptionReportMessage` message generates an exception error in the `RCRSCSMessageConverter` class, which does not seem to be prepared to handle those two types of messages.

A.6 References

H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, volume 6, pages 739–743, Tokyo, Japan, October 1999. IEEE.

C. Skinner and S. Ramchurn. The robocup rescue simulation platform. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1*, AAMAS '10, pages 1647–1648, Richland, SC,

2010. International Foundation for Autonomous Agents and Multiagent Systems.

Suntori Team. *Library for Communication – ReadMe*. 2012.