Universidad Politécnica de Madrid

European Master in Software Engineering

RoboCup Rescue Simulator

Quick User Guide

Author:
Gorka Álvarez Marlasca

Madrid, Monday 22nd May, 2023

Table of Contents

Source Code List

List of Figures

1 Document Scope

The purpose of this document is to complement the documentation provided by Robocop for its Rescue Simulation League [1]. Since the documentation provided only gives a very high level view of how the tool works, this document is intended to give a more specific view that will serve as a basis for future developments and implementations related to the server.

It should be noted that this paper is also part of the TFM (Constraint-Based Problem Middleware for Simulation of Intelligent Agent Teams in Urban Rescue) prepared for the Universidad Politecnica de Madrid.

Throughout the document the main modules that compose the server will be shown as well as the way to modify and update each one of them. To emphasize that the present document only shows simple implementations, since its main objective is the understanding of the characteristics of the system.

Throughout the guide two main systems will be referenced, whose base implementations are hosted in the following GitHub repositories:

- ***RCRS Server***: `https://github.com/roborescue/rcrs-server`, from now on *The Server.*
- ***ADF Sample Agent***: `https://github.com/roborescue/adf-sample-agent-java`, from now on *The agent Controller.*

All implementations and descriptions described in the document are based on the implementations published by Robocup as of the date of writing of this document (Monday 22nd May, 2023). Also, a sample implementation of the contents of this guide can be found in the following GitHub repository `https://github.com/gorokotkd/robocup`.

All the commands used through this guide, and other helpful ones, can be found at Appendix A, Useful Commands.

2 System Requirements

According to the documentation provided by Robocup, the necessary requirements to be able to launch the different processes of the system are the following:

- Have a version of Git installed. (Version used in this document 2.36.1)
- Have a version of Gradle installed. (Version used in this document 7.4.2)
- Having installed Java version 17 or OpenJDK 17.

3 Map Customization

In the following section it is specified how to modify the basic settings of a map, these are the location of the elements on the map. Likewise, it is specified how to activate other types of simulators that compose the system.

3.1 Add new Agents and Elements

Agents are added through the map configuration file. To add or remove new agents to the map the following path must be accessed:

```
{basPath}/rcrs-server/maps/test/map/scenario.xml
```

[1] `https://rescuesim.robocup.org/`

Code 1: Map Path

At this point, an example file config file content can be the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scenario:scenario xmlns:scenario="urn:roborescue:map:scenario">
    <scenario:refuge scenario:location="298" scenario:bedCapacity="2" />
    <scenario:refuge scenario:location="255" scenario:bedCapacity="3" />
    <scenario:firestation scenario:location="248"/>
    <scenario:civilian scenario:location="254"/>
    <scenario:civilian scenario:location="905"/>
    <scenario:policeforce scenario:location="934" />
    <scenario:policeforce scenario:location="975" />
    <scenario:ambulanceteam scenario:location="902" />
    <scenario:firebrigade scenario:location="902" />
</scenario:scenario>
```

Code 2: Sceneraio file config

The possible agent types that can be added to this sceneario are the following:

- *Civilian.*
- *Police Force.*
- *Fire Brigade.*
- *Ambulance Team.*

The location of the agents is simply the identifier of the box within the map, the available boxes, as well as the map itself, are found in the *map.gml* file, located in the same folder as the previous file.

In addition to agents, through the file shown in Sceneraio file config you can also add the buildings associated to the agents. Specifically, the buildings that can be added are the following:

- *Refuge.*
- *Police Office.*
- *Fire Station.*
- *Ambulance Centre.*

With this in mind, a possible configuration file with these buildings would be as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scenario:scenario xmlns:scenario="urn:roborescue:map:scenario">
    <scenario:civilian scenario:location="254"/>
    <scenario:policeforce scenario:location="975" />
    <scenario:ambulanceteam scenario:location="902" />
    <scenario:firebrigade scenario:location="902" />
    <scenario:refuge scenario:location="298" scenario:bedCapacity="2" />
    <scenario:ambulancecentre scenario:location="58404" />
    <scenario:firestation scenario:location="248"/>
    <scenario:policeoffice scenario:location="50754" />
</scenario:scenario>
```

Code 3: Sceneraio file config with buildings

3.2 Activate Fire Damage to Agents

By default some damage types are not activated. If fire damage wants to be activated in order to test the correct performance of the firefighters the following must be done.

First of all, the following file must be accessed:

```
{basPath}/rcrs-server/modules/misc/src/misc/MiscSimulator.java
```

Code 4: Misc Simulator Path

After accesing the file, we must uncomment line 33 from method *processCommands*

```
 1 @Override
 2 protected void processCommands(KSCommands c, ChangeSet changes) {
 3     long start = System.currentTimeMillis();
 4     int time = c.getTime();
 5     Logger.info("Timestep " + time);
 6 
 7     for (Command com : c.getCommands()) {
 8         if (checkValidity(com)) {
 9             if (com instanceof AKRescue) {
10             Human human = (Human) (model.getEntity(((AKRescue) com).getTarget()));
11             handleRescue(human, changes);
12         }
13         /*
14          * For the implementation of Refuge Bed Capacity
15          **/
16         if (com instanceof AKUnload) {
17             handleUnload(com, changes);
18         }
19 
20         if (com instanceof AKLoad) {
21             handleLoad((AKLoad) com, changes);
22         }
23 
24       } else {
25             Logger.debug("Ignoring " + com);
26       }
27     }
28 
29     updateRefuges();
30 
31     processBrokenBuildings(changes);
32     processBurningBuildings( changes );
33     // processExplodedGasStations( changes );
34     updateDamage(changes);
35 
36     updateChangeSet(changes);
37 
38     // Clean up
39     newlyBrokenBuildings.clear();
40     writeDebugOutput(c.getTime());
41     if (gui != null) {
42         gui.refresh(humans.values());
43     }
```

```
    long end = System.currentTimeMillis();
    Logger.info("Timestep " + time + " took " + (end - start) + " ms");
}
```

After making any change in the source code of the server, we must re-build the project using the command:

```
./gradlew completeBuild
```

Code 5: Misc Simulator Path

The fire simulator must also be activated on the server. To do so, make sure that the following lines are uncommented in the *startSims* function of the *functions.sh* file (This file is located inside the *scripts/* folder):

```
execute fire "java -Xmx1024m -cp
↪  $CP:$BASEDIR/jars/rescuecore2.jar:$BASEDIR/jars/standard.jar:
↪  $BASEDIR/jars/resq-fire.jar -Dlog4j.log.dir=$LOGDIR rescuecore2.LaunchComponents
↪  firesimulator.FireSimulatorWrapper -c $CONFIGDIR/resq-fire.cfg $GUI_OPTION $*"
echo "waiting for fire to connect..."
waitFor $LOGDIR/fire-out.log "success"
```

4 Creation of new Agent Controllers

The agent controller allows to configure the behavior of the different agents that compose the system. All the basic and standard operation of the agents are implemented in the adf-core.[2]

An example of an agent configured using the Agent Development Framework mentioned above is available on the official Roborescue GitHub. View https://github.com/roborescue/adf-sample-agent-java. To use it, it is simply needed to clone the repository.

4.1 Modifying Agent Behaviour

The Agent Development Framework provides all the necessary tools to modify the behavior of each and every agent in the system. Among them, it offers the possibility of modifying the behavior of the agents, being able to fully configure the way in which the agents decide which tasks to perform or the path to follow to achieve their objective.

The following section will show the changes to be made to modify the movement of the agents, i.e., the algorithm they follow to determine the best path to reach their objective.

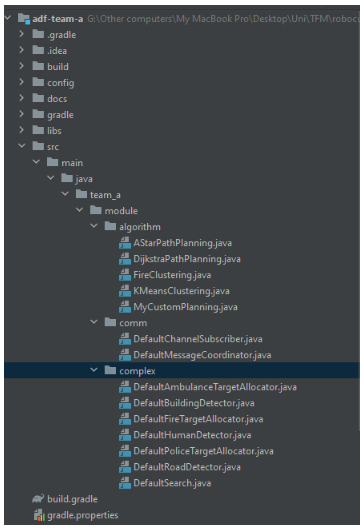
4.1.1 Modification of Agent Path Planning

Through the property *PathPlanning* an agent is able to decide the path he wants to use in order to achieve a given target or group of targets. By default, agents use Dijkstra's algorithm, an algorithm for determining the shortest path. The objective of this section is to show the changes to be made to the controller so that the agents can use different types of path calculation algorithms.

To perform this modification, the best option is to use the Dijkstra algorithm already created as a basis. To do this, simply copy the file located in `adf/impl/module/algorithm`[3] towards the custom agent implementation in the controller. The following image shows an example directory tree of how the classes can be saved:

[2]Agent Development Framework

[3]This path is located inside the core of the agent development framework, view https://github.com/roborescue/adf-core-java



```
adf-team-a G:\Other computers\My MacBook Pro\Desktop\Uni\TFM\roboc
  .gradle
  .idea
  build
  config
  docs
  gradle
  libs
  src
    main
      java
        team_a
          module
            algorithm
              AStarPathPlanning.java
              DijkstraPathPlanning.java
              FireClustering.java
              KMeansClustering.java
              MyCustomPlanning.java
            comm
              DefaultChannelSubscriber.java
              DefaultMessageCoordinator.java
            complex
              DefaultAmbulanceTargetAllocator.java
              DefaultBuildingDetector.java
              DefaultFireTargetAllocator.java
              DefaultHumanDetector.java
              DefaultPoliceTargetAllocator.java
              DefaultRoadDetector.java
              DefaultSearch.java
  build.gradle
  gradle.properties
```

Figure 1: ADF Modules Directories

Based on the previous file, you simply need to create a new path calculation file, for example *MyCustomPlanning.java*. Once the file has been created, before starting to modify it, the configuration (rootDir)config/module.cfg) must be updated to assign the new calculation configuration to the different elements that require it. The following is an example of a modification of the search for the different types of platoons. If you want to modify only the movement characteristic, you must modify the *DefaultExtActionMove.PathPlanning* entry.

```
## SampleSearch
SampleSearch.PathPlanning.Ambulance : team_a.module.algorithm.MyCustomPlanning
SampleSearch.Clustering.Ambulance : team_a.module.algorithm.KMeansClustering
SampleSearch.PathPlanning.Fire : team_a.module.algorithm.MyCustomPlanning
SampleSearch.Clustering.Fire : team_a.module.algorithm.KMeansClustering
SampleSearch.PathPlanning.Police : team_a.module.algorithm.MyCustomPlanning
SampleSearch.Clustering.Police : team_a.module.algorithm.KMeansClustering
```

Code 6: Configuration file for custom path planning

Once the configuration file has been updated, the new path calculation class will be modified. To do so, the attributes and methods to be taken into account are the following:

- **Class Attributes**. In addition to the attributes of the class itself, the attributes inherited from the abstract class *PathPlanning* must also be taken into account.
  - ***From***. Contains the id of the box in which the agent is located.
  - ***Targets***. Contains the list of target box identifiers. By default, this list consists of the buildings visible and not visited by the agent.
  - ***Result***. Contains the list of result boxes, i.e. the path chosen by the agent.
  - ***Graph***. Contains the relationship between the *Cell - Neighbors* for each of the cells on the map.
  - ***AgentInfo***. Contains the agent controller basic information.
  - ***WorldInfo***. It contains the information of the world visible to the agent. This is, through this attribute the agent knows the identifiers of all the elements of the world, although it does not know all the specific information of each one of them.
  - ***ScenarioInfo***. It contains the static information of the scenario, it contains the parameter information such as movement time, waiting time, etc.

- **Class Methods**. The methods are inherited and overraided from the PathPlanning class, and new auxiliary methods can also be generated to perform the calculations.
  - `void init()`. Initializes the attribute *graph*, based on the attribute *worldInfo* constructs the lisat of relationship Cell - Neighbors.
  - `PathPlanning calc()`. It contains all the logic related to the calculation of the path to reach the *targets*, by default, you will find the implementation of Dijkstra's algorithm.

4.2 Example of Path Planning

Once the configuration of the previous section is done, the following is the implementation of the calc() method to make the agents choose a random route to their target.

```java
private PathPlanning calc(){
    List<EntityID> path = new ArrayList<>();
    path.add(from);
    while (!isGoal(path.get(path.size() - 1), targets)){
        Collection<EntityID> neighbours = graph.get(path.get(path.size() - 1));
        EntityID next = getRandom(neighbours);
        path.add(next);
    }
    this.result = path;
    return this;
}

public static <T> T getRandom(Collection<T> coll) {
    int num = (int) (Math.random() * coll.size());
    for(T t: coll) if (--num < 0) return t;
    throw new AssertionError();
}
```

Code 7: Random PathPlanning Implementation

The implementation consists of two methods. The first, the `T getRandom(Collection<T> coll)` method, obtains a random element from a given collection as a parameter. The second, the `PathPlanning calc()` method, adds random cells to the result path. These cells are chained together starting from the cell where the agent is located.

Next to this implementation, a slightly more complex implementation of the path calculation is shown below. In it, in order to establish a minimum coordination between police officers, the choice of a cell is preferred if there is a police officer on it, so that whenever two police officers are close to each other, they will try to stay close to each other. In case there are no police officers nearby, a cell will be selected at random.

```java
private PathPlanning calc(){
    StandardEntity entity = agentInfo.me();
    PoliceForce policeForce = ((PoliceForce) entity);
    List<EntityID> path = new ArrayList<>();
    List<StandardEntity> policeForces =
    ↪  worldInfo.getEntitiesOfType(StandardEntityURN.POLICE_FORCE).stream()
            .filter(k -> !k.getID().equals(agentInfo.getID()))
            //.filter(k -> ((PoliceForce)k).getTeam().equals(policeForce.isTeamDefined()
            ↪  ? policeForce.getTeam() : ""))
            .collect(Collectors.toList());

    path.add(from);


    while (!isGoal(path.get(path.size() - 1), targets)){
        Set<EntityID> neighbours = graph.get(path.get(path.size() - 1));
        Boolean isPoliceClose = false;
        for(StandardEntity police : policeForces){
            EntityID position = ((PoliceForce)police).getPosition();
            if(neighbours.contains(position) && !path.contains(position)){
                path.add(position);
                isPoliceClose = true;
                break;
            }
        }

      if(!isPoliceClose){
            path.add(getRandom(neighbours));
      }

    }

    this.result = path;
    return this;
}
```

4.3 Modifying Civilians Behaviour

Police officers, medical team and firefighters are not the only ones who can modify their behavior. The system also allows the modification of the behavior of civilians, this is done from the server side, since it is the server that creates the controllers for these entities. The behavior of civilians is very simple "Run to the nearest shelter".

By default, the server has a civilian type called *SampleCivilian*, the configuration of which is found in the package *sample* (`{rootFolder}/modules/sample/src/sample`). To modify the behavior, it is only necessary to modify the `void think()` method of this class.

If new civilian classes have been created, the launcher must be modified to detect these new civilian types. To do this, it will be necessary to navigate to the *scripts* folder and then to the *functions.sh* file. Once in this file, the modification must be made in the *starSims* function, which is in charge of executing all the simulators that compose the server. To add different types of civilians, the following line must be updated:

```
execute civilian "java -Xmx1512m -cp
↪  $CP:$BASEDIR/jars/rescuecore2.jar:$BASEDIR/jars/standard.jar:
↪  $BASEDIR/jars/sample.jar:$BASEDIR/jars/kernel.jar -Dlog4j.log.dir=$LOGDIR
↪  rescuecore2.LaunchComponents sample.SampleCivilian*n -c $CONFIGDIR/civilian.cfg $*"
```

Code 8: Civilian execution original line

The modification must be made in the *LaunchComponents* part, where the type of civilians to be launched and the number of instances of them are indicated. By default the value is *sample.SampleCivilian*n*, which launches n instances of Civilians of type *SampleCivilian*, where n is the number of civilians on the map.

If the new civilians are also in the sample package, changing one class for another is enough, in case they are in different packages, the path of the *sample.jar* package must be changed for the path of the package where the compiled code of the civilian is located.

The value of n can also be modified without any problem, its range varies from 0 to n. In case the launching of these instances is less than the total number of agents on the map, the civilians will have to be selected manually from the user interface.

```
execute civilian "java -Xmx1512m -cp
↪  $CP:$BASEDIR/jars/rescuecore2.jar:$BASEDIR/jars/standard.jar:
↪  $BASEDIR/jars/sample.jar:$BASEDIR/jars/kernel.jar -Dlog4j.log.dir=$LOGDIR
↪  rescuecore2.LaunchComponents sample.SampleCivilian*0 -c $CONFIGDIR/civilian.cfg $*"
execute civilian "java -Xmx1512m -cp
↪  $CP:$BASEDIR/jars/rescuecore2.jar:$BASEDIR/jars/standard.jar:
↪  $BASEDIR/jars/sample.jar:$BASEDIR/jars/kernel.jar -Dlog4j.log.dir=$LOGDIR
↪  rescuecore2.LaunchComponents sample.Team1Civilian*0 -c $CONFIGDIR/civilian.cfg $*"
```

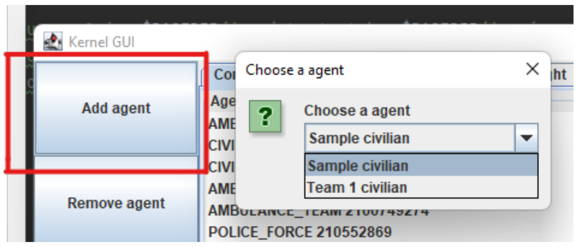


Figure 2: Civilian manual launching

5 Creating Teams

Throughout the following section it will be described the changes made in the server and in the agents in order to be able to perform a team management. This will not only allow to divide the agents in two teams but also to manage the decisions they make depending on the team they belong to.

Note that the steps presented bellow allows the assignment of an agent to a team. The agent itself will know the team he belongs to, but he would not know the team of the other agents in the system.

5.1 Changes in the Server

The base implementation of the Server does not allow the creation of teams in the agents. Adding this possibility requires making a series of changes in the properties of those entities to which it is required to assign teams.

Next, the process to follow to add new properties to the *Human* entities of the server, i.e. Civilians, Police Officers, Paramedics Team... is shown. Likewise, all this process can be applied to the different buildings that can be generated in the server, allowing the creation of totally different algorithms.

Likewise, it is necessary to point out that although the process is generalized to humans, certain changes are at the final entity level, so the example that will be shown will only include the police officers.

5.1.1 Creation of the new Property

All the entities that make up the server are made of properties, in the case of humans, they are made up of properties like *hp, stamina, damage...* Since the property that is going to be added is *team* and it is represented by a String, a new *StringPopery* should be generated. All the properties of the server are located in `rcrs-server/modules/rescuecore2/src/rescuecore2/worldmodel/properties`.

Inside that folder the property *StringPorperty* will be created, this property will allow the creation of new properties with a String based content.

```java
public class StringProperty extends AbstractProperty {
    private String value;

    public StringProperty(int urn) {
        super(urn);
    }

    public StringProperty(URN urn) {
        super(urn);
    }

    public StringProperty(int urn, String value) {
        super(urn, true);
        this.value = value;
    }

    public StringProperty(URN urn, String value) {
        super(urn, true);
        this.value = value;
    }

    public StringProperty(StringProperty other) {
        super(other);
        this.value = other.value;
    }

    @Override
    public String getValue() {
        if (!isDefined()) {
            return null;
        }
        return value;
    }

    public void setValue(String value) {
        String old = this.value;
        boolean wasDefined = isDefined();
        this.value = value;
        setDefined();
        if (!wasDefined || old != value) {
            fireChange(old, value);
        }
    }

    @Override
    public void takeValue(Property p) {
        if (p instanceof StringProperty) {
            StringProperty i = (StringProperty) p;
            if (i.isDefined()) {
                setValue(i.getValue());
            } else {
                undefine();
            }
        } else {
            throw new IllegalArgumentException(
                    this + " cannot take value from " + p);
        }
    }

    @Override
    public void write(OutputStream out) throws IOException {
        write(out);
    }

    @Override
    public void read(InputStream in) throws IOException {
        setValue(in.toString());
    }

    @Override
    public StringProperty copy() {
        return new StringProperty(this);
    }

    @Override
    public PropertyProto toPropertyProto() {
        PropertyProto.Builder builder = basePropertyProto();
        if (isDefined()) {
            
            ↪  builder.setByteList(ByteString.copyFrom(value.getBytes(StandardCharsets.UTF_8)));
        }
        return builder.build();
    }
    @Override
    public void fromPropertyProto(PropertyProto proto) {
        if (!proto.getDefined())
            return;
        setValue(proto.getByteList().toString(Charset.defaultCharset()));
    }
}
```

Code 9: StringProperty for team property creation

With this base, the next step is as simple as creating a new attribute inside the Human class (`rcrs-server/modules/standard/src/rescuecore2/standard/entities`) `private StringProperty    team;`

Every property of the server must have it associated URN, the URNs of the properties associated to Humans are located inside a file called *StandardPropertyURN*, which is located inside the same folder as the Human class. This is an *Enum* class, so the only thing to be done is adding a new entry to the enum `TEAM(PROPERTY_URN_PREFIX | 36, PROPERTY_URN_PREFIX_STR + "team");`

Now, everything is ready to add the *team* property to the Human Entity, with all the above steps done the Human constructors and the method *getProperty()* should be the following:

```
protected Human( EntityID id ) {
    super( id );
    x = new IntProperty( StandardPropertyURN.X );
    y = new IntProperty( StandardPropertyURN.Y );
```

```java
    travelDistance = new IntProperty( StandardPropertyURN.TRAVEL_DISTANCE );
    position = new EntityRefProperty( StandardPropertyURN.POSITION );
    positionHistory = new IntArrayProperty(
        StandardPropertyURN.POSITION_HISTORY );
    direction = new IntProperty( StandardPropertyURN.DIRECTION );
    stamina = new IntProperty( StandardPropertyURN.STAMINA );
    hp = new IntProperty( StandardPropertyURN.HP );
    damage = new IntProperty( StandardPropertyURN.DAMAGE );
    buriedness = new IntProperty( StandardPropertyURN.BURIEDNESS );
    team = new StringProperty( StandardPropertyURN.TEAM );
    registerProperties( x, y, position, positionHistory, travelDistance,
        direction, stamina, hp, damage, buriedness, team );
  }

public Human( Human other ) {
    super( other );
    x = new IntProperty( other.x );
    y = new IntProperty( other.y );
    travelDistance = new IntProperty( other.travelDistance );
    position = new EntityRefProperty( other.position );
    positionHistory = new IntArrayProperty( other.positionHistory );
    direction = new IntProperty( other.direction );
    stamina = new IntProperty( other.stamina );
    hp = new IntProperty( other.hp );
    damage = new IntProperty( other.damage );
    buriedness = new IntProperty( other.buriedness );
    team = new StringProperty( other.team );
    registerProperties( x, y, position, positionHistory, travelDistance,
        direction, stamina, hp, damage, buriedness, team );
  }


@Override
public Property getProperty( int urn ) {
    StandardPropertyURN type;
    try {
      type = StandardPropertyURN.fromInt( urn );
    } catch ( IllegalArgumentException e ) {
      return super.getProperty( urn );
    }
    switch ( type ) {
      case POSITION:
        return position;
      case POSITION_HISTORY:
        return positionHistory;
      case DIRECTION:
        return direction;
      case STAMINA:
        return stamina;
      case HP:
        return hp;
      case X:
        return x;
      case Y:
        return y;
      case DAMAGE:
        return damage;
      case BURIEDNESS:
        return buriedness;
      case TRAVEL_DISTANCE:
        return travelDistance;
      case TEAM:
        return team;
      default:
        return super.getProperty( urn );
    }
}
```

Code 10: Content of Human Class after adding the team property

Ass well as this, the corresponding getters and setter method should be created for this new property:

```
  public String getTeam() {
    return team.getValue();
  }

  public void setTeam( String team ) {
    this.team.setValue( team );
  }

  public boolean isTeamDefined() {
    return team.isDefined();
  }

  public void undefineTeam() {
    team.undefine();
  }
```

Code 11: Getters and Setters for the Team propery

Last but not least, in order to ensure the correct creation of the new property that has been added, is essential to add this new property to the factory: `rcrs-server/modules /standard/src/rescuecore2/standard /entities` inside the *StantardPropertyFactory.java* class. The content of the method *makeProperty()* must be modified, being the result the one listed bellow:

```
  @Override
  public Property makeProperty(StandardPropertyURN urn) {
    switch (urn) {
      case START_TIME:
      case LONGITUDE:
      case LATITUDE:
      case WIND_FORCE:
      case WIND_DIRECTION:
      case X:
      case Y:
      case FLOORS:
      case BUILDING_ATTRIBUTES:
      case FIERYNESS:
      case BROKENNESS:
      case BUILDING_CODE:
```

```
      case BUILDING_AREA_GROUND:
      case BUILDING_AREA_TOTAL:
      case DIRECTION:
      case STAMINA:
      case HP:
      case DAMAGE:
      case BURIEDNESS:
      case WATER_QUANTITY:
      case TEMPERATURE:
      case IMPORTANCE:
      case TRAVEL_DISTANCE:
      case REPAIR_COST:
      case CAPACITY:
      case BED_CAPACITY:
      case REFILL_CAPACITY:
      case OCCUPIED_BEDS:
      case WAITING_LIST_SIZE:
        return new IntProperty(urn);
      case APEXES:
      case POSITION_HISTORY:
        return new IntArrayProperty(urn);
      case IGNITION:
        return new BooleanProperty(urn);
      case POSITION:
        return new EntityRefProperty(urn);
      case BLOCKADES:
        return new EntityRefListProperty(urn);
      case EDGES:
        return new EdgeListProperty(urn);
      case TEAM:
        return new StringProperty(urn);
      default:
        throw new IllegalArgumentException("Unrecognised property urn: " + urn);
    }
  }
```

Code 12: Code modification for StandardPropertyFactory.java

5.1.2 Assigning a new team

In the previous section how to create a new *team* property has been shown, in this section the steps to use the new property will be indicated.

The first point to take into account is that the name of the team will be defined from the agent controller, the way to define the name of the team from the controller is already created by default, so it will not be necessary to make any changes in that place.

However, changes will have to be made when the agent controller connects to the server and to the agent it is going to control. This is embedded in the *ComponentManager.java* class, located in `rcrs-server/modules /kernel/src/kernel`

In particular, the change must be made once the ACK is received from the agent's controller,exactly, the *handleAKConnect()* method.

```
1  private void handleAKConnect(AKConnect connect, Connection connection) {
2      // Pull out the request ID and requested entity type list
3      int requestID = connect.getRequestID();
4      List<Integer> types = connect.getRequestedEntityTypes();
5      // See if we can find an entity for this agent to control.
6      Message reply = null;
7      Logger.debug("AKConnect received: " + types);
8      synchronized (agentLock) {
9          ControlledEntityInfo result = findEntityToControl(types);
10         if (result == null) {
11             Logger.debug("No suitable entities found");
12             // Send an error
13             reply = new KAConnectError(requestID, "No more agents");
14         } else {
15             Logger.debug("Found entity to control: " + result);
16             Entity entity = result.entity;
17             if(entity instanceof Human){
18                 ((Human)entity).setTeam(connect.getAgentName().split("\\.")[0]);
19             }
20             AgentProxy agent = new AgentProxy(connect.getAgentName(),
21                     entity, connection);
22             agentsToAcknowledge.add(new AgentAck(agent, entity.getID(),
23                     requestID, connection));
24             Logger.info("Agent '" + connect.getAgentName() + "' id "
25                     + entity.getID() + " (" + connection
26                     + " request ID " + requestID + ") connected");
27             // Send an OK
28             reply = new KAConnectOK(requestID, entity.getID(),
29                     result.visibleSet, result.config);
30         }
31     }
32     if (reply != null) {
33         try {
34             connection.sendMessage(reply);
35         } catch (ConnectionException e) {
36             Logger.error("Error sending reply", e);
37         }
38     }
39     updateGUIUncontrolledAgents();
40     updateGUIAgentAck();
41 }
```

Code 13: Modification to handleAKConnect in CommponentManager

The change made has been to add lines 17 to 19, where, after receiving the ACK from the agent controller, it has been searched for an entity to control. In doing so, it is verified that this entity is of type *Human* and the team defined in the controller is added. The reason for using the *split* method is due to the fact that the format in which the name is received is *"teamName.instanceName"*.

The last step is to launch the agents. To do this, from the controller, we must include the *-tn* option which allows to add a team name to the set of agents to be launched. An example of launching the command would be the following:

```
bash launch.sh -at 1 -pf 2 -tn team-a
```

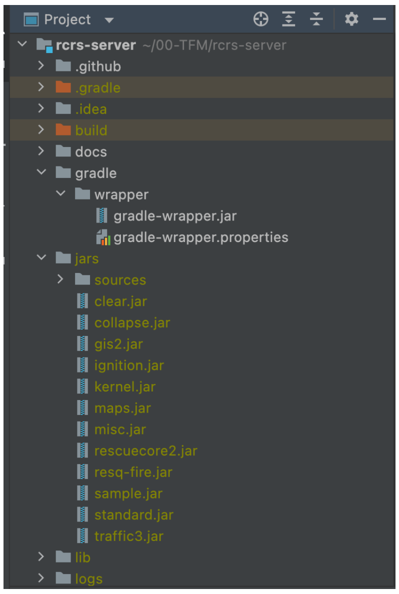
5.2 Changes in Agent Controller

Throughout the previous section, it has been shown the various changes to be made to the server to ensure that it supports all the features of equipment management.

Those modifications must be visible from the agent controller, which, by default, works using the last version deployed on GitHub (view 1. Document Scope).

In order to modify this version the *.jar* files generated by the server must be manually added. For generating them, from the server root folder, the following command must be run mintinlinebashgradle clean and after, `./gradlew completeBuild`.

If after running the second command the *.jar* files have not been generated, the command `gradle jar` should be executed. If the process has been executed correctly, the needed content would have been created in the root directory



```
Project
rcrs-server ~/00-TFM/rcrs-server
  .github
  .gradle
  .idea
  build
  docs
  gradle
    wrapper
      gradle-wrapper.jar
      gradle-wrapper.properties
  jars
    sources
    clear.jar
    collapse.jar
    gis2.jar
    ignition.jar
    kernel.jar
    maps.jar
    misc.jar
    rescuecore2.jar
    resq-fire.jar
    sample.jar
    standard.jar
    traffic3.jar
  lib
  logs
```

Figure 3: Generated Server Jar Files

Next, all the *.jar* files inside the */jars* folder will be selected (the content of the folder */sources* is not needed) and they will be copied to the agent controller[4]. These will be pasted inside a new directory called */libs*.

[4]Only the *.jars* called *rescuecore2.jar* and *standard.jar* are needed, but for avoiding possible dependencies errors it is recommended to paste all the generated jars.

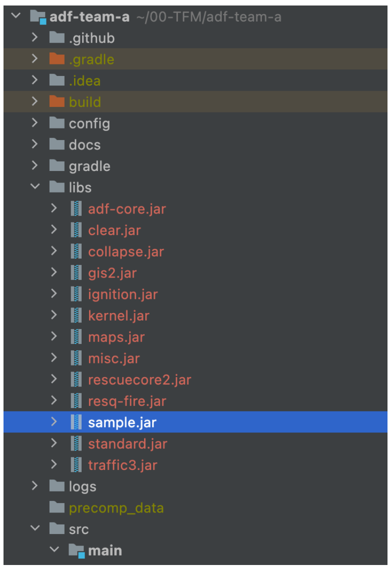


Figure 4: Jar Files Added to the Agent Controller

This change requires modifying the file *build.gradle* where all the project dependencies are listed. If the *.jar* files have been added inside a folder called */libs* the dependencies task of the file should be the following:

```
dependencies {
  implementation fileTree(dir: 'libs', include: '*.jar')
  implementation files('libs/rescuecore2.jar')
  implementation files('libs/standard.jar')
  implementation files('libs/clear.jar')
  implementation files('libs/collapse.jar')
  implementation files('libs/gis2.jar')
  implementation files('libs/ignition.jar')
  implementation files('libs/kernel.jar')
  implementation files('libs/maps.jar')
  implementation files('libs/misc.jar')
  implementation files('libs/resq-fire.jar')
  implementation files('libs/sample.jar')
  implementation files('libs/traffic3.jar')
  implementation files('libs/adf-core.jar')
  implementation 'org.uncommons.maths:uncommons-maths:1.2.2a'
  implementation 'com.fasterxml.jackson.core:jackson-annotations:2.13.0'
  implementation 'com.fasterxml.jackson.core:jackson-core:2.13.0'
  implementation 'com.fasterxml.jackson.core:jackson-databind:2.13.0'
  implementation 'com.google.code.findbugs:jsr305:3.0.2'
  implementation 'com.google.common:google-collect:0.5'
  implementation 'javax.activation:activation:1.1.1'
  implementation 'javax.xml.bind:jaxb-api:2.4.0-b180830.0359'
  implementation 'log4j:log4j:1.2.17'
  implementation 'org.msgpack:jackson-dataformat-msgpack:0.9.0'
  implementation 'log4j:log4j:1.2.17'
  implementation 'com.google.code.gson:gson:2.8.9'
  implementation 'com.google.guava:guava:31.0.1-jre'
  implementation 'com.google.protobuf:protobuf-java:3.19.1'
  implementation 'com.google.protobuf:protobuf-java-util:3.19.1'
  implementation 'com.vividsolutions:jts:1.13'
  implementation 'jaxen:jaxen:1.2.0'
```

```
    implementation 'jfree:jfreechart:1.0.13'
    implementation 'log4j:log4j:1.2.17'
    implementation 'net.sf.trove4j:trove4j:2.1.0'
    implementation 'net.sourceforge.jsi:jsi:1.0.0'
    implementation 'org.dom4j:dom4j:2.1.3'
    implementation 'org.json:json:20210307'
    implementation 'org.jfree:jcommon:1.0.24'
    implementation 'org.jscience:jscience:4.3.1'
    implementation 'org.slf4j:slf4j-log4j12:1.7.32'
    implementation 'org.uncommons.maths:uncommons-maths:1.2.2'
    implementation 'org.tukaani:xz:1.9'
    implementation 'org.projectlombok:lombok:latest.integration'
    implementation 'org.apache.commons:commons-compress:1.21'
    annotationProcessor 'org.projectlombok:lombok:latest.integration'

    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'
  }
```

Code 14: Dependencies task of build.gradle file

It should be noted that in addition to the path of the *.jar* files, all the dependencies that are in the server have been added manually, this is because the project is not able to detect the dependencies, so the best way to adopt is the one shown above.

Once the dependencies are added, simply execute the command `gradle clean build` from the root directory of the agent controller to load the dependencies. With this, you will be able to interact with the agents' computers from the controller, allowing you to greatly expand the agents' configurations.

A Useful Commands

A.1 Commands for the Agent Controller

- `gradle clean build`. Rebuilds the controller code in after making any changes in the code.
- `bash launch.sh`. Runs the agent controller with the default options.
- `bash launch.sh -pf -1` Runs the agent controller with the PF option as -1, which tells the server that this controller will control every instance of the Police Force.
- `bash launch.sh -tn example-name` Runs the agent controller with the TN option, which allows the user to assign the agents to a team called “example-name”.
- `bash launch.sh -all` Runs the agent controller with all options. This command is very usefull when having a great number of agents to control.

A.2 Commands for the RoboCup Rescue Server

- `./gradlew completeBuild`. Custom script to build completely the server after making any change in the source code. When modifying config files it is not mandatory to run this command.
- `bash start.sh -m "../maps/istanbul/map"`. Run from the *scripts* folder, it runs the server using the Istanbul map, by default, the used map is the test one.
- `bash start.sh` Run from the *scripts* folder, it runs the server with the default configuration.