CO-OPERATIVE ROBOTICS SIMULATOR - ENVIRONMENT SIMULATOR

by

SCOTT JOSEPH HARMON

B. S., Kansas State University, 2002

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2004

Approved by:

Major Professor
Scott A. DeLoach, Ph.D.

ABSTRACT

In this project, we built a simulator that models the interactions between robots and an environment, and between robots and other robots. Every robot has two major components: effectors and sensors. Effectors try to make modifications to the environment, while sensors try to perceive different aspects of the environment. The goal of this simulator was to simulate robotic interactions in a scalable and accurate fashion.

Our simulator consists of five separate modules: the Environment module, the Robot Hardware module, the Viewer module, the Control Panel module, and the Messaging module. This report covers one specific module, the Environment module. The Environment module is responsible for modeling the simulated world. It also drives the simulation. The Environment module manages all the interactions between robots as well as the interactions between the robots and the simulated world.

KEYWORDS: Cooperative Robotic Simulator, Environment, Distributed Simulations

TABLE OF CONTENTS

LIST OF FIGURES..III

ACKNOWLEDGEMENTS...IV

CHAPTER 1 INTRODUCTION...1

1.1 BACKGROUND... 1

1.2 SYSTEM DESIGN OVERVIEW 2

CHAPTER 2 ENVIRONMENT MODULE...4

2.1 ENVIRONMENT OVERVIEW 4

2.2 ENVIRONMENT STATES AND MAIN LOOP... 4

2.2.1 State Diagram 4

2.2.2 Main Loop 5

2.3 COMPONENTS 7

2.3.1 System Component Diagram...7

2.3.2 Environment Components...8

2.3.3 EnvironmentMap Component.. 10

2.3.4 Robot and Sensor Components... 12

2.4 CURRENTLY IMPLEMENTED EFFECTORS AND SENSORS..................................... 13

2.4.1 Effectors.. 13

2.4.2 Sensors.. 14

2.4.3 Integrating New Sensors.. 15

2.5 ENVIRONMENT PROTOCOLS AND INTERFACES 15

2.5.1 Robot Hardware Simulator and Environment Protocol................................. 16

2.5.2 Viewer and Environment Protocol 19

2.5.3 Control Panel and Environment Protocol 21

2.5.4 Environment Model File Format 22

CHAPTER 3 CONCLUSIONS 23

CHAPTER 4 FUTURE WORK 24

4.1 NETWORK IMPROVEMENTS 24

4.2 ENVIRONMENTMAP EXTENSIONS AND COLLISION DETECTION 26

4.3 FURTHER ABSTRACTION AND USE OF REFLECTION 27

APPENDIX A: RUNNING THE ENVIRONMENT 28

REFERENCES 29

LIST OF FIGURES

FIGURE 1: SIMULATOR DEPLOYMENT OVERVIEW. 2

FIGURE 2: HIGH LEVEL SYSTEM DESIGN. [1] 3

FIGURE 3: ENVIRONMENT STATE DIAGRAM 5

FIGURE 4: MAIN SIMULATION LOOP. 6

FIGURE 5: SYSTEM COMPONENT DIAGRAM. 8

FIGURE 6: ENVIRONMENT COMPONENT DIAGRAM. 9

FIGURE 7: ENVIRONMENTMAP COMPONENT CLASSES. 10

FIGURE 8: ROBOT AND ROBOT SENSOR CLASS DIAGRAM 12

FIGURE 9: THE ROBOT/ENVIRONMENT PROTOCOL. 16

FIGURE 10: ROBOTREQUEST CLASS 17

FIGURE 11: ROBOTSENSORRESPONSE CLASS. 18

FIGURE 12: VIEWER/ENVIRONMENT PROTOCOL 19

FIGURE 13: VIEWEROBJECT CLASS DIAGRAM 20

FIGURE 14: VIEWERUPDATELOCATION CLASS. 21

FIGURE 15: CONTROL PANEL/ENVIRONMENT PROTOCOL. 22

FIGURE 17: EXAMPLE OF A PROTOCOL TO IMPROVE NETWORK LATENCY. 25

FIGURE 18: RUNNING THE ENVIRONMENT 28

ACKNOWLEDGEMENTS

This project has been a very good learning experience for me. I would like to mention some of the people that have helped me along the way.

First, I would like to thank my wonderful wife, who has been patient and understanding while I have been working on this project. Without her support, this project would not have been the success that it is.

Second, I would like to thank my advisor, Dr. DeLoach, for allowing me to work on this project and for guiding it. He was always reasonable and understanding in any discussions that came up during our project meetings.

I would also like to thank Dr. David Gustafson and Dr. William Hsu for serving in my committee and for their aid in guiding the development of this project.

I am grateful to my teammates, Venkata Prashant Rapaka, Arun Prakash Ganesan, Esteban Guillen, and Aaron Chavez, for putting up with me and picking up the slack for me when my time started to run short.

I also want to thank my friends, who helped me when I needed fresh ideas or when I simply needed a break.

Finally, I would like to thank all of my instructors. Every insight I have had into aspects of this project have been because of your teaching.

Chapter 1 Introduction

1.1 Background

Simulators are useful tools for compressing time, testing without physical repercussions, and modeling things not yet found in the real world. They are used in a wide range of applications, anywhere from modeling explosions of atomic bombs, to modeling weather patterns.

For our simulator, we wanted a system that was scalable and accurate. The primary use of this system is to test various cooperative robotic programs that could also run on physical robots. We needed to be able to test scenarios where there are many robots and scenarios with various environments.

We divided our simulator into five separate modules: the Environment module, the Robot Hardware module, the Viewer module, the Control Panel module, and the Messaging module. All of the modules interact through the Environment module. Each module is designed to be able to run on a separate computer and thus, they all communicate through a network layer.[1] This enables our simulator system to be distributed across computers, and as such, harness the power of multiple computers. Figure 1, shows the high-level interaction

[1] The current implementation of the Communications module does not support separation of the Communications module from the Environment module.

between each module.

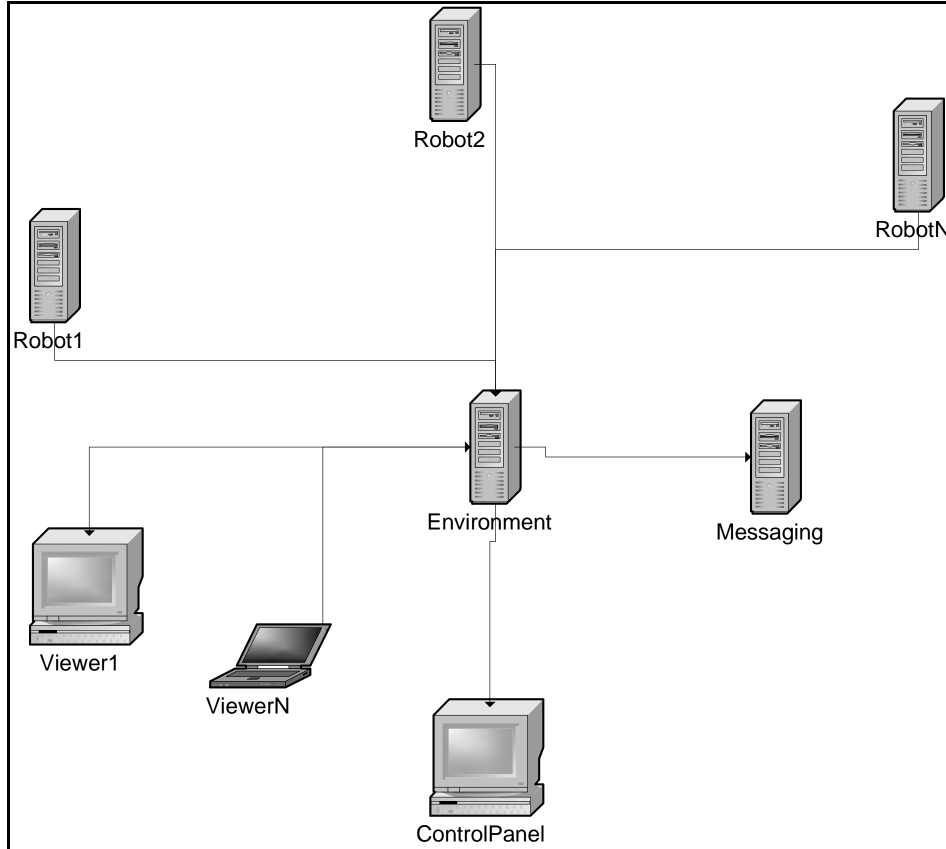

Figure 1: Simulator deployment overview.

The Environment module may have an arbitrary number of Robot clients and an arbitrary number of Viewer modules connected to it. Currently, the number of robots in a scenario is specified by the Environment model file that is loaded at run time.

1.2 System Design Overview

Each module is responsible for a different part of the overall simulation. Figure 2 shows the original high-level design of the system.

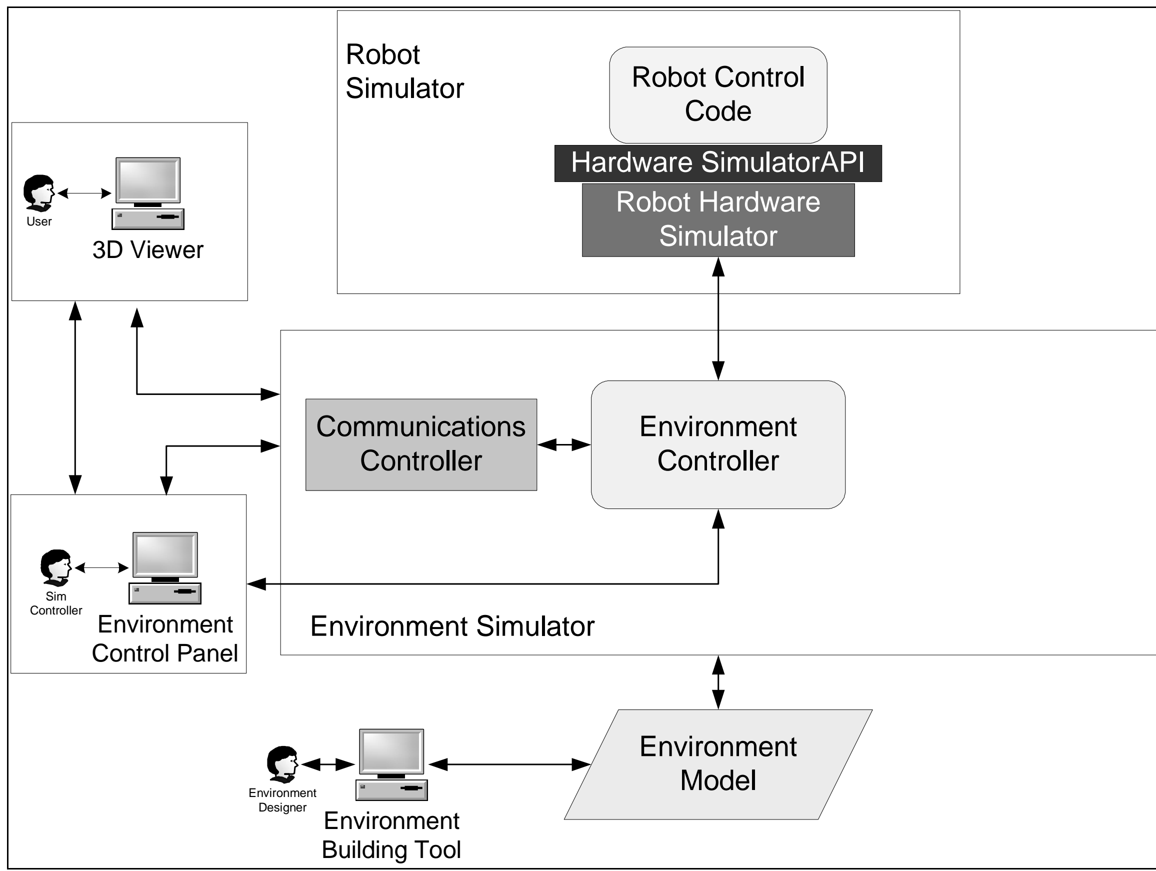

Figure 2: High level system design. [1]

The Environment module is responsible for conducting the simulation and for simulating the virtual environment in which the robots will operate. The Robotic Hardware Simulator is responsible for modeling the hardware that is found on different models of physical robots, to translate robotic operations into operations that the Environment module provides, and to break those operations into the time-step size specified by the Environment module. The Messaging module is a package to allow for simulated communications between robots. The Viewer module allows us to view the simulation in real time, or to play back a saved simulation. The Control Panel module allows properties of the simulation to be set, such as, time-step size, the ports the Environment module will listen on for connections, and what environment model file to use for the simulation.

Chapter 2 Environment Module

2.1 Environment Overview

The Environment module is the center of the system. It must communicate with all of the other components. To accomplish this communication we developed various protocols. These protocols are described in more detail later in the paper. The Environment module must also orchestrate the entire simulation. I took a time-step based approach to accomplish this. Each time-step occurs through one iteration of the Environment module's main loop. This loop is described in more detail in subsequent sections of this paper.

2.2 Environment States and Main Loop

2.2.1 State Diagram

The Environment may be in one of a set of five states. These states allow the Environment to determine whether certain variables may be changed and whether or not it can move on to different stages of the simulation. Figure 3 shows the transitions between states.

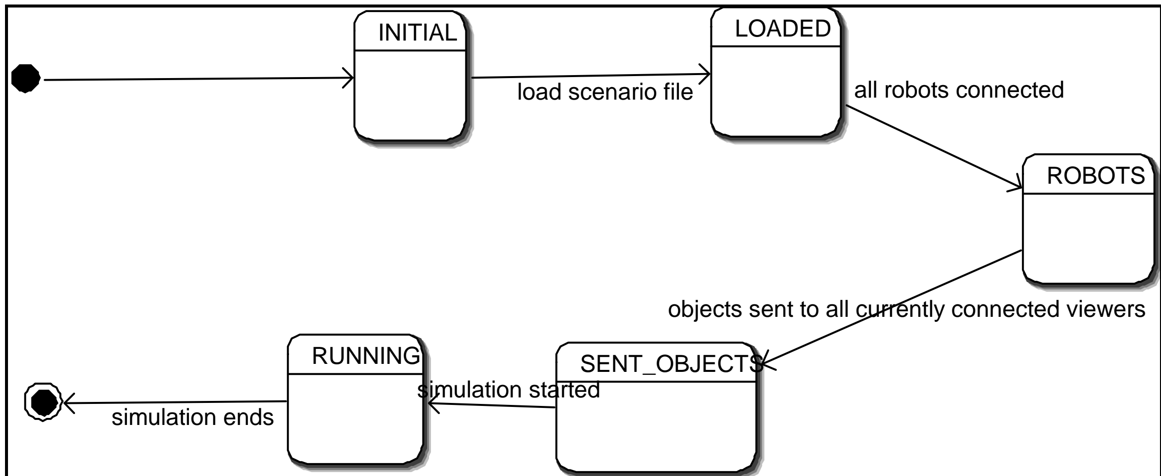

Figure 3: Environment State Diagram

Robots may only connect when the Environment has left the INITIAL state. Parameters, such as time-step size and port numbers may only be set in the LOADED state. If a viewer connects and the Environment has passed the SENT_OBJECTS state (is in the SENT_OBJECTS, or any state after that), the Viewer Objects are sent to the Viewer, otherwise this is postponed until we are ready to make the transition from the ROBOTS state to the SENT_OBJECTS state.

2.2.2 Main Loop

The Environment drives the simulation. It does this through iteration of a main loop. This loop is given in Figure 4.

```
1  waitForRobots();
2  setState(STATE_ROBOTS);
3  sendObjectsToViewerClients();
4  currentTime = 0;
5  running = true;
6  setState(STATE_RUNNING);
7  while(running) {
8     //get the event for this timestep from each Robot
9     //lock the robotqueue
10    synchronized(robots) {
11          //This double-stepped get allows the stuff to travel here
before I block trying to get it.
12          for (int i = 0; i < robots.size(); i++) {
13                EnvironmentObjectRobot robot =
(EnvironmentObjectRobot) robots.get(i);
14                robot.prepGetEvents(currentTime);
15          }
16          for (int i = 0; i < robots.size(); i++) {
17                EnvironmentObjectRobot robot =
(EnvironmentObjectRobot) robots.get(i);
18                robot.queueEvents();
19          }
20    }
21    //process queue from front (top).
22    processActionEventQueue();
23    //send out sensor readings to robots.
24    processSensorEventQueue();
25    //add timestep to ViewerUpdates.
26    for (int i = 0; i < viewerUpdateQueue.size(); i++) {
27          ((ViewerUpdateLocation)viewerUpdateQueue.get(i)).timestep
= currentTime;
28    }
29    //send out ViewerUpdates to the Viewer
30    sendViewerEvents(viewerUpdateQueue);
31    currentTime++;
32    try {
33          Thread.sleep(steppausetime);
34    } catch (InterruptedException e) {}
35 }
```

Figure 4: Main simulation loop.

The method "waitForRobots()" simply waits for the expected number of robots to connect as specified in the Environment model file. After the robots are connected, the state of the Environment changes to "STATE_ROBOTS". All the objects in the environment are now sent to any attached Viewers with the "sendObjectToVRMLClients()" command. The time-step is initialized to 0, running is set to true, and the state is changed to "STATE_RUNNING", in lines 4-6.

We now enter the main loop. The first step is to ask all the robots for the commands or requests they wish to perform in the present time-step. The Environment performs this in two stages. The first stage, "prepGetEvents(currentTime)", sends the current time-step to the robot. This prompts the robot to send requests for that time-step to the Environment. The second stage receives those requests and places them into queues, which are stored within each EnvironmentObjectRobot. I chose to do this in two steps to help with network latency, since I make requests to all the robots before blocking waiting to receive the requests. This gives the requests time to propagate over the network.

After all of the events have been received, the Environment processes all the action events in line 22. It then goes on to process the sensor events in line 24. After processing all the events, the Environment must send all the updates to any attached Viewer. It does this in its call to "sendViewerEvents(viewerUpdateQueue)". Finally, the time-step is incremented and the loop begins again.

2.3 Components

2.3.1 System Component Diagram

The Environment can only be used together with the other components. Without the Robot component, nothing can happen. Generally, you will want to set the parameters of the Environment through the Control Panel, and view the simulation through the Viewer. Figure 5 depicts a component diagram of the entire system.

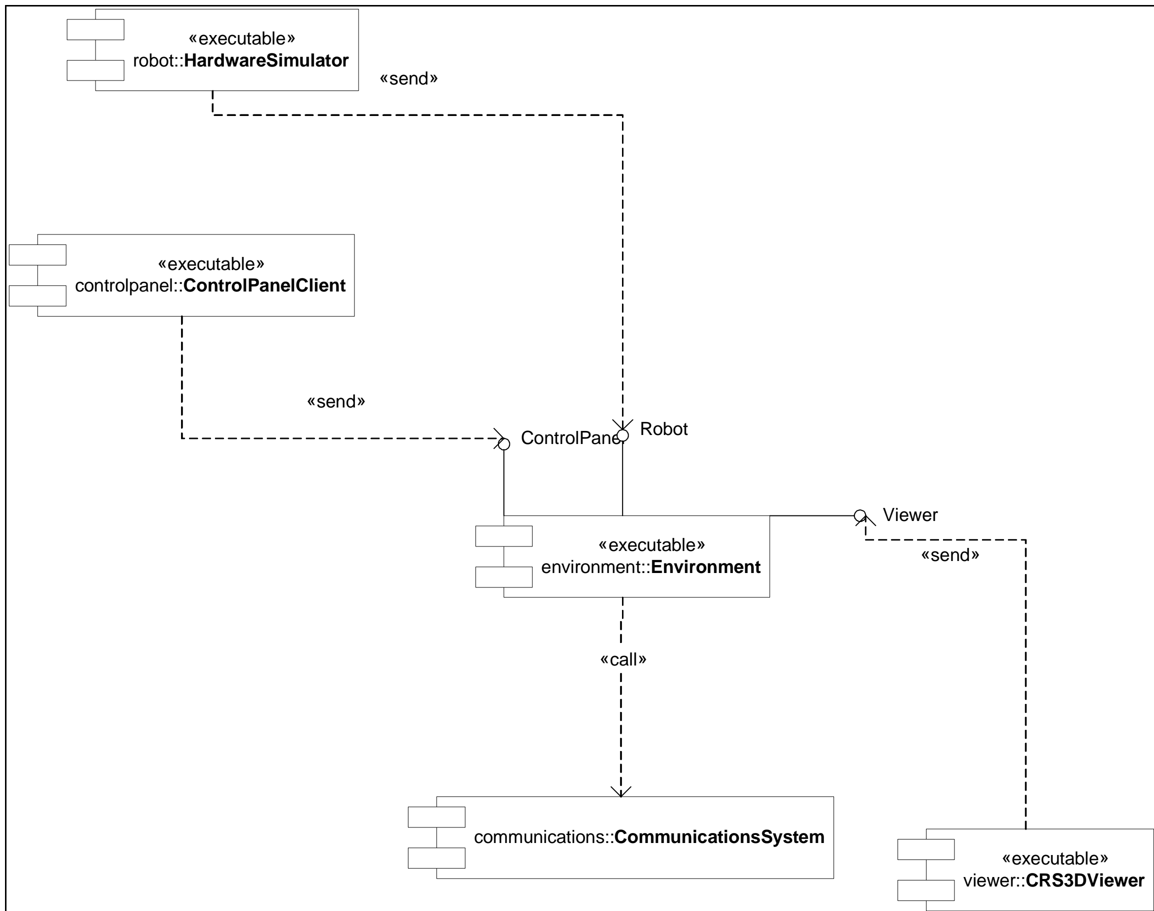

Figure 5: System Component Diagram.

Each component is contained in a package with the names shown above and with the prefix of "edu.ksu.cis.cooprobot.simulator.". Since my part of the simulator was to construct the environment module, I will break this down into its respective modules.

2.3.2 Environment Components

The Environment module has several major components. These components are the EnvironmentMap component, the CollisionDetection component, the EnvironmentObject components, the Robot components, and the Sensor components. All these components work together to produce the simulation of the environment. Figure 6 shows a high-level component diagram of the environment package.

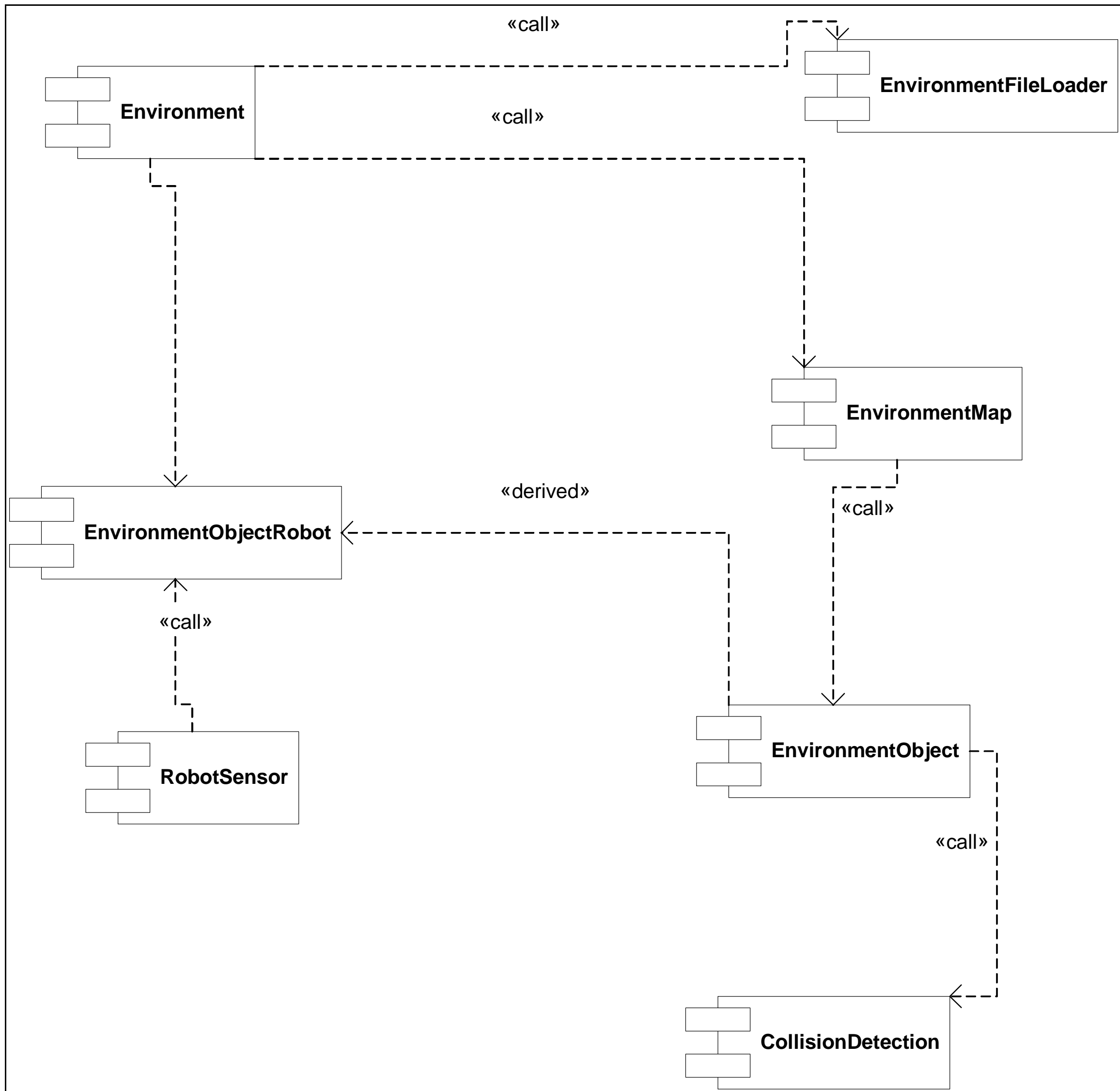

Figure 6: Environment Component Diagram.

All of the components within the Environment module interact using references and method calls. I will explain in more detail, how the major components interact in the following sections.

2.3.3 EnvironmentMap Component

The EnvironmentMap component contains the representation of the environment as well as methods to manipulate and query the state of the environment. Figure 7 shows a close up of the class diagram for the classes related to the EnvironmentMap component.

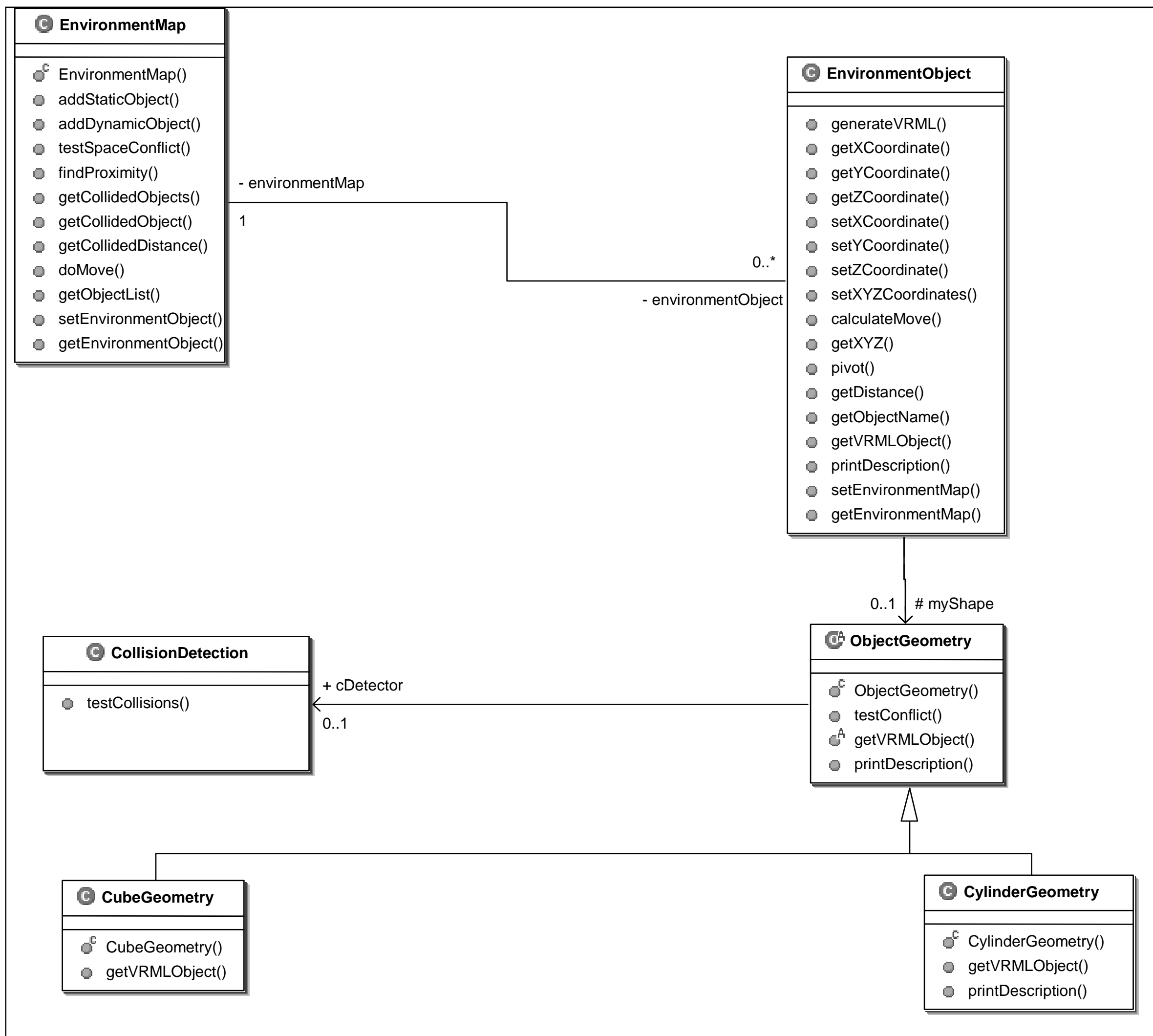

Figure 7: EnvironmentMap component classes.

The EnvironmentMap manages all the EnvironmentObjects. Each EnvironmentObject contains information related to position in three-dimensional space, as well as orientation and geometry. An EnvironmentObject may have only a single ObjectGeometry associated with it. The ObjectGeometry class is an abstract class that is currently implemented by two

classes, CubeGeometry and CylinderGeometry. These classes contain the information needed to represent their respective geometric properties. Each geometry contains a method, 'testConflict'. This method, in the current implementation, refers to a static class, 'CollisionDetectionNaive'. CollisionDetectionNaive implements the CollisionDetection interface shown in the figure above. CollisionDetectionNaive performs simple collision detection between two geometries that have certain positions and orientations in three-dimensional space.

EnvironmentMap contains methods that are used by different sensors. In particular, the "getCollidedObject(s)" methods are used by the sonar and heat sensors in determining their return values. EnvironmentMap also contains some methods used by the Environment class. The "addStaticObject()" and "addDynamicObject()" methods are used to add EnvironmentObjects to the simulation. The "doMove()" method is used by the Environment class, to move an object within the virtual environment. This move call obeys collision rules, and specifies how far the robot wishes to move. This means that the robot may not move as far as it tried, and it is possible that it may not move at all.

The EnvironmentObject class contains methods that act on any EnvironmentObject, whether they are robots or otherwise. It contains methods for setting and getting positional information as well as for current rotation ("pivot()") about the y-axis (the y-axis being perpendicular to the ground).

If you wish to expand the set of geometries that an EvironmentObject may have, you must perform several steps. First you must create a new class that extends ObjectGeometry, you should include in your new class fields that allow you to describe this new shape. Second, you need to add your new shape into the collision detection code. Currently, this resides in CollisionDetectionNaive. Third, you need to add appropriate an Viewer object for your new

geometry (for example ViewerCylinder). Last, you must update the EnvironmentFileLoader to recognize the new geometry and be able to initialize the dimensions of the new geometry.

2.3.4 Robot and Sensor Components

The representation for robots in the environment is an extension of the EnvironmentObject class. A robot may have a number of sensors and effectors, these sensor and effectors may be located on the robot in different places. The information for sensors are stored along with each robot. A class diagram for the sensor and robot structure is shown in Figure 8.

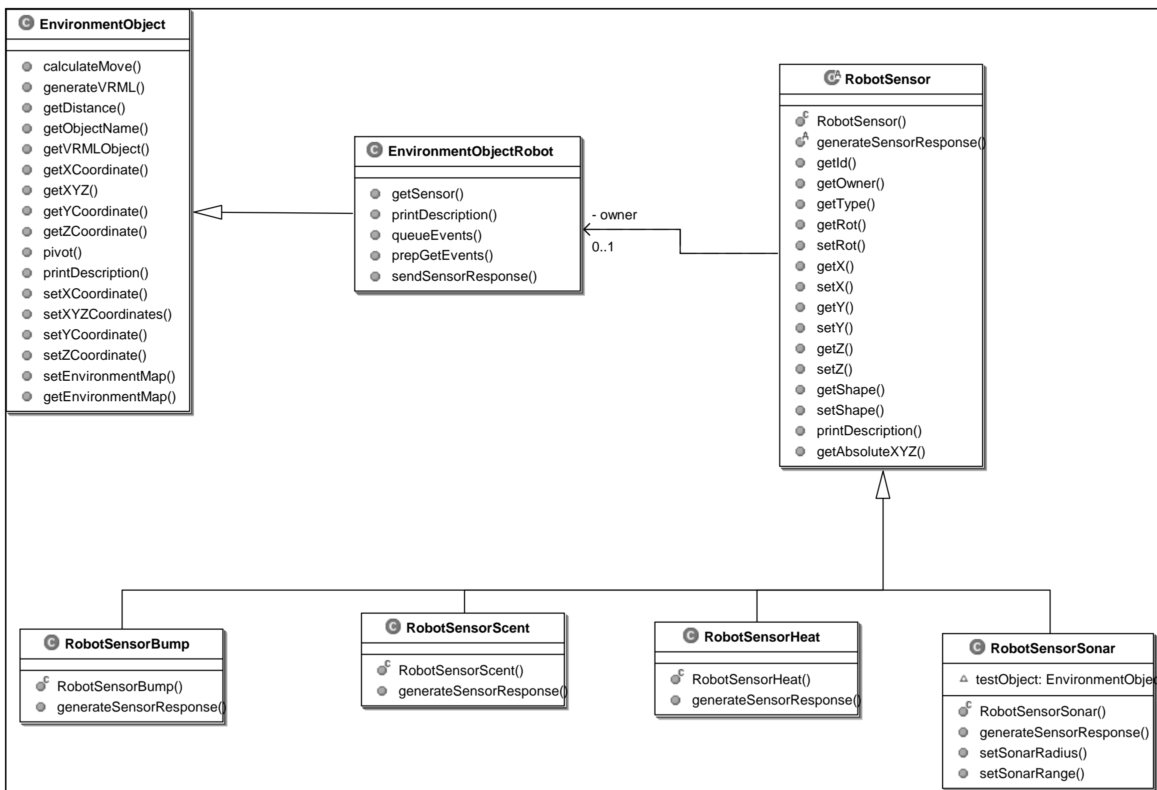

Figure 8: Robot and robot sensor class diagram

Each sensor has some common information such as position and rotation relative to the robot. This information is stored in the RobotSensor class. Each one of the subclasses of RobotSensor (for example, RobotSensorBump) extends RobotSensor to specialize it for the particular sensor function it is supposed to perform. RobotSensor is an abstract class, with one method left incomplete: 'generateSensorResponse()'. This method is called by the

Environment whenever the robot requests a response from this particular sensor. The Environment, therefore, does not need to know the details of how each sensor operates. The Environment just issues the same command ("generateSensorResponse()") for every sensor, which returns a RobotSensorResponse object. This RobotSensorResponse is simply passed on to the robot that made the request, with no further processing by the Environment. In fact, RobotSensorResponse is a class within the robot package.

2.4 Currently Implemented Effectors and Sensors

I have currently implemented three sensors and three effectors. I will outline how each one of these sensor and effectors operate.

2.4.1 Effectors

A robot may send action events to the environment every time-step. These actions are preformed on the virtual environment. When the actions are preformed on the virtual environment, the state of the virtual environment is changed ($S_1 \rightarrow S_2$). The currently implemented commands are move, turn, and tag.

The move command attempts to move the object within the virtual environment. The move command has one parameter, which is the distance to move. Since each object has an orientation (rotation about the y-axis), the move is performed in the direction of the orientation. There are two outcomes of the move operation. If the object is placed at the coordinates determined by the distance and orientation and there is no collision, then the move is allowed and the virtual environment changes from state S_1 to S_2. If the move is not allowed, then a smaller move is tried until either the move can be made or no move is possible. If no move is possible, the environment's state remains the same.

The turn command changes the object's orientation within the virtual environment. It also has one parameter, which is the amount to turn in radians. Currently turns are always allowed and no checking is done in the virtual environment for collisions caused by simply turning.

The tag command currently 'tags' objects in a specified radius from the centroid of the robot object. It tags the object by setting the object's heat component to 0. The tag command has no limitations and nothing is checked in the virtual environment.

2.4.2 Sensors

A sensor is basically an instrument that gives a robot information about the current state of the virtual environment. A robot may request sensor responses each time-step. The sensors that I have currently implemented are bump sensor, sonar, and heat.

The bump sensor is used to detect whether or not one object is against another object. The bump sensor is currently stuck to the front of the robot. It is important to note here that, on the actual Scout robot, there are several bump sensors located around the robot. The response that my bump sensor sends back to the robot is merely a true if the sensor is activated, otherwise it sends back false.

The sonar sensor is a bit more sophisticated than the bump sensor. It allows a robot to determine in what direction and how far away an object is. A robot may have several sonar sensors which are oriented in different directions and are located in different positions on the robot. When a request for a sonar sensor is made, the distance to the nearest object in the direction that the sonar is facing is sent back to the robot. Each sonar has a maximum range, thus, if nothing is found within that maximum range, then the maximum value is sent back to the robot.

The heat sensor is non-directional. It senses the presence of heat within a certain radius. Heat comes from objects in the robot's surrounding and is additive. This means that if there are more than one object in the range of the heat sensor, their values will be added. The heat also decays with distance from the source. Therefore, the farther away a robot is from a heat source, the lower the value sent back to the robot.

2.4.3 Integrating New Sensors

In order to integrate new sensors into the Environment, you must create a new class which represents your new sensor and which extends the RobotSensor class. The main part to fill in in your new sensor class, is the generateSensorResponse method. This method is declared abstract in the RobotSensor class and therefore must be defined in your new subclass. This method is called whenever a robot requests sensor state information for your new sensor. This method has as a input parameter, a reference to the current EnvironmentMap component. This allows you to call methods in EnvironmentMap to obtain information about the current state of the virtual world. After you have implemented your new sensor class, you must add the appropriate code into the EnvironmentFileLoader within the 'setupSensors()' method to be able to instantiate the new sensor class. Of course, you must also alter your robot to be able to handle the new sensor. For more information on the robot's side of sensors, please refer to Venkata Prashant Rapaka's 2004 Masters' Report.

2.5 Environment Protocols and Interfaces

The Environment module must communicate with the Robot Hardware Simulator(s), the Viewer(s), Control Panel(s), and the Messaging module. Each point of communication involves a protocol or interface. The protocol definition allows the Environment to

communicate with these modules efficiently and effectively. All of the protocols have been implemented by making use of Java socket connections.

2.5.1 Robot Hardware Simulator and Environment Protocol

This protocol enables the Robotic Hardware Simulator module to send commands to be performed on the virtual environment provided by the Environment module. The protocol also enables the Robotic Hardware Simulator to receive simulated sensor readings from the virtual environment. In Figure 9, a typical run of the protocol is depicted.

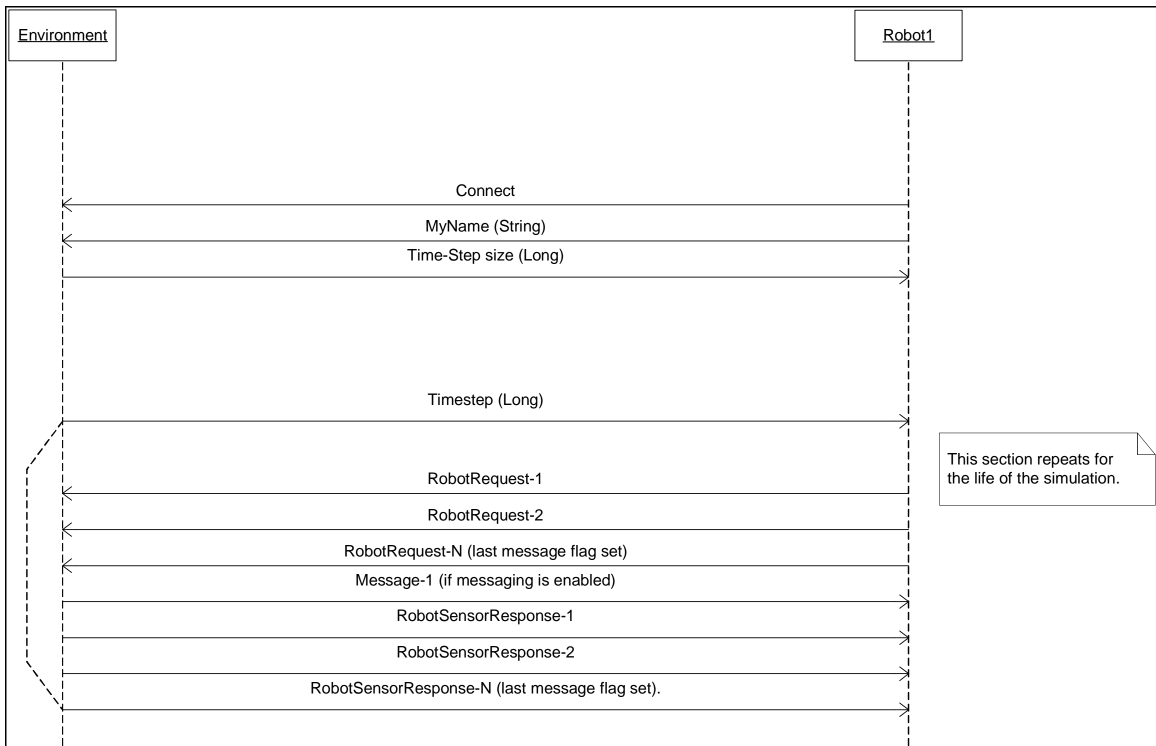

Figure 9: The Robot/Environment Protocol.

The protocol is initiated by a Robot connecting to a pre-specified port to the Environment. Once a connection is established, the Robot informs the Environment of its name (this name must be unique). After receiving the name, the Environment informs the Robot of the time-step size. This ends the initial phase of the protocol.

In the second phase, the Environment sends the current time-step to the Robot as a Long object. The robot then sends a series of commands to the Environment. These commands may consist of actions or requests. An action may be to move forward one centimeter, while a request may be a request for the current status of the robot's bump sensor. The list of commands ends with the last message flag being set. Figure 10 shows the RobotRequest class.

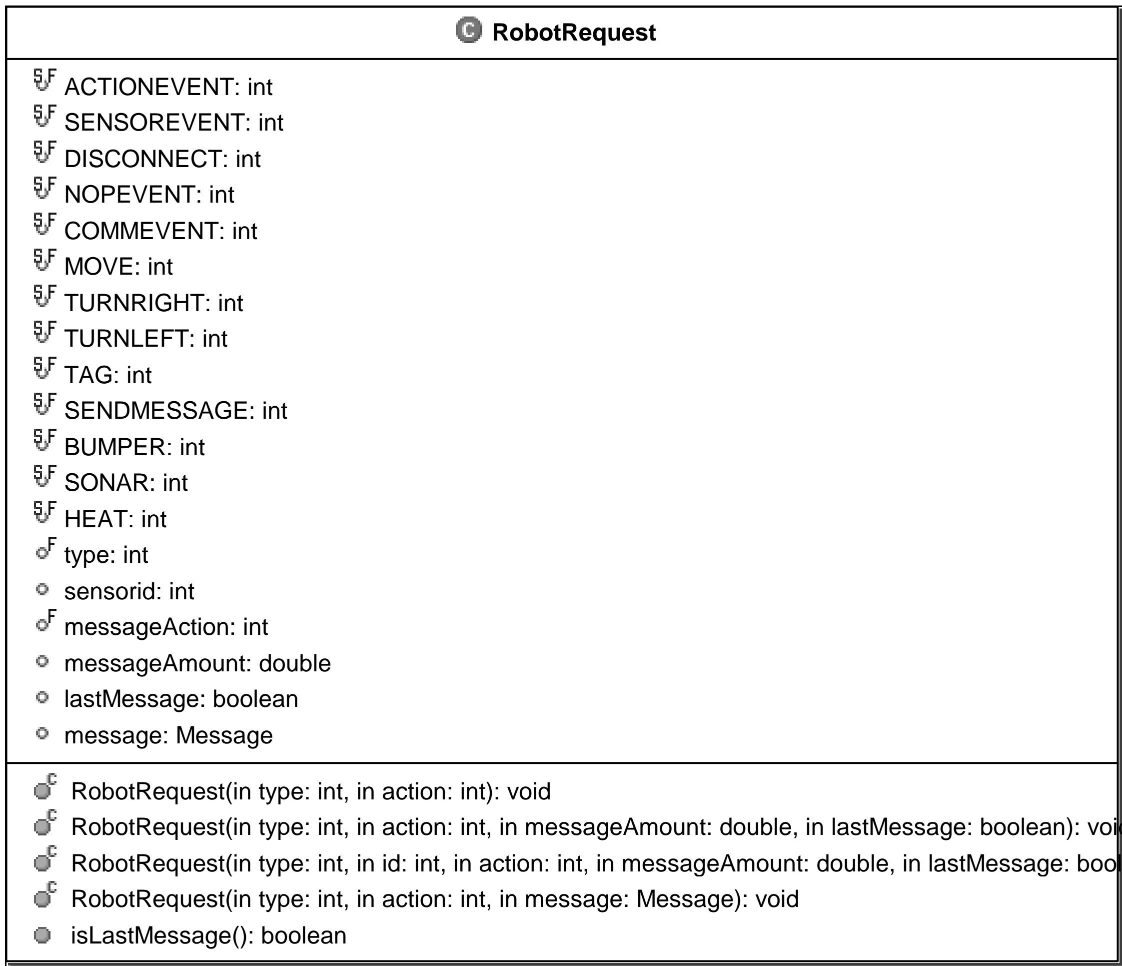

Figure 10: RobotRequest class.

A RobotRequest currently may have four types: ACTIONEVENT, SENSOREVENT, NOPEVENT, or COMMEVENT. These types are defined as static final integers. Each message also has a specific messageAction: MOVE, TURNRIGHT, TURNLEFT, and TAG

for the ACTIONEVENT type; BUMPER, SONAR, and HEAT for the SENSOREVENT type; and SENDMESSAGE for the COMMEVENT type. The sensorid field is currently only used if the type is set to SENSOREVENT. The messageAmount field is currently only used when the type is set to ACTIONEVENT. The message field is only used when the type is set to COMMEVENT.

After receiving the RobotRequests, the Environment processes the actions and requests and then sends back a series of responses, again being terminated by the last message flag being set. If robot communications is enabled, any messages for the robot are sent before any RobotSensorResponses are sent. The second phase is repeated for the life of the simulation. Figure 11 shows the RobotSensorResponse class.

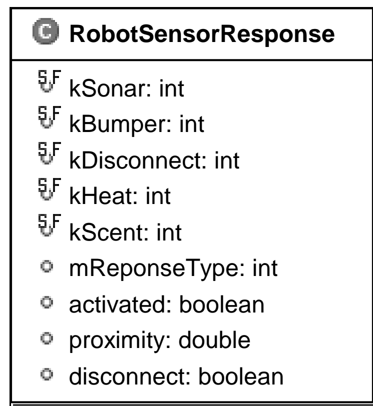

Figure 11: RobotSensorResponse class.

The RobotSensorResponse may be used for different types of sensors, mResponseType is set to the corresponding sensor type which generated the response (kSonar, kBumper, kHeat, or kScent). The activated field is used for boolean valued sensors, such as the bump sensor, and proximity is used for real valued sensor readings, such as sonar.

2.5.2 Viewer and Environment Protocol

This protocol allows viewers to connect to the Environment module, allowing them to observe the simulation. The protocol is simple and consists of two phases as shown in Figure 12.

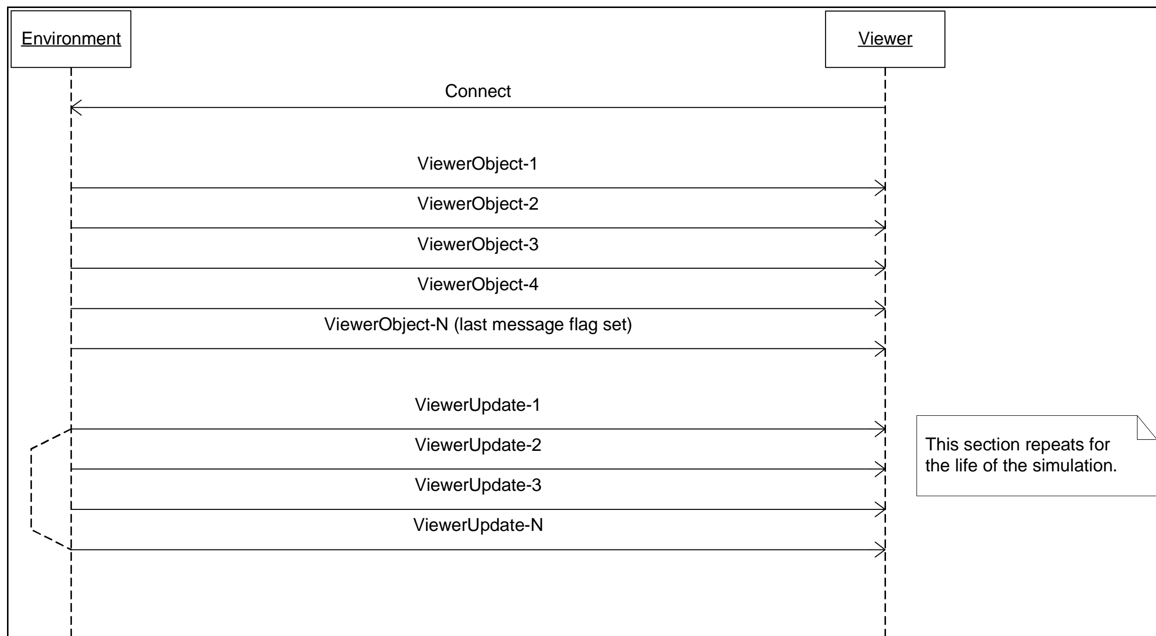

Figure 12: Viewer/Environment Protocol.

The initial phase involves the Viewer establishing a connection with the Environment on a pre-determined port. Once the connection is established (and after all the robots are connected), the Environment sends all the objects for display to the Viewer. If a Viewer connects after the simulation is started, it simply moves through both phases of the protocol right away. If it connects before all the robots are connected, then the first phase is delayed until all the robots have connected. The objects sent to the Viewer include specifications on shape, size, position, and color and are shown in more detail in Figure 13.

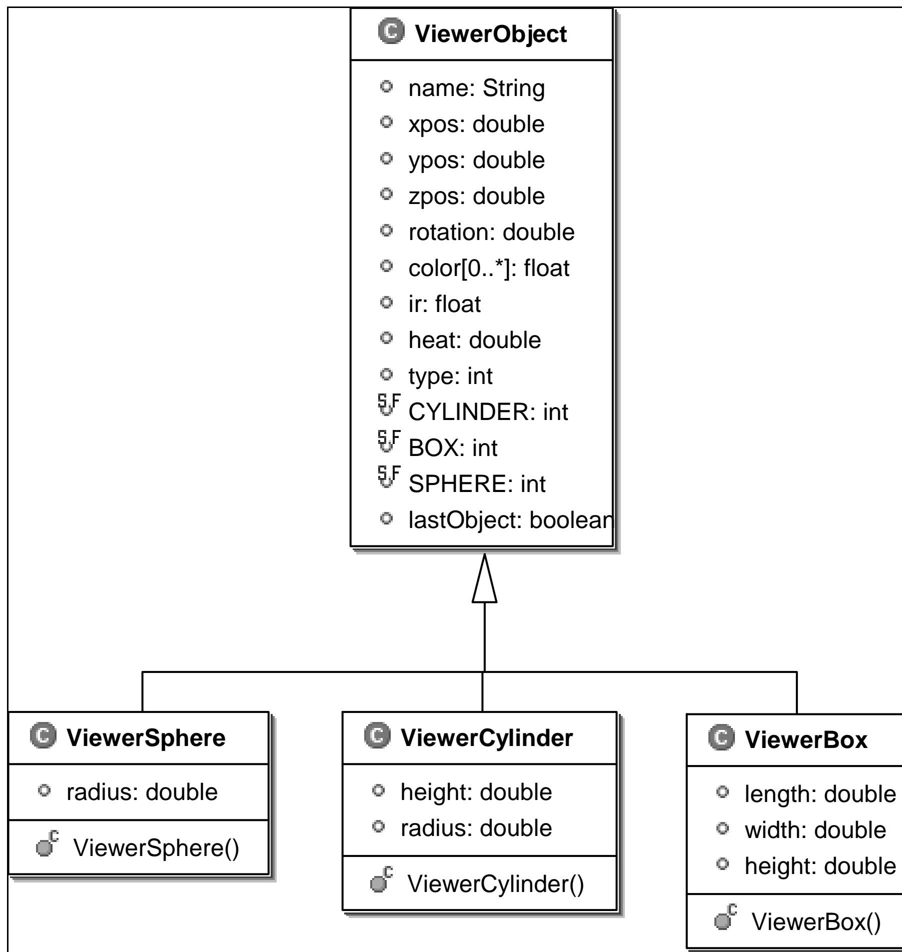

Figure 13: ViewerObject class diagram.

The name field of the ViewerObject is a unique identifier, which allows the Environment to specify updates to that ViewerObject by using its name. The xpos, ypos, and zpos, are the current coordinates of the object in three-dimensional space, with respect to the centroid of the object. The color field specifies the color of the object in an array of three floats. The ir and heat fields indicate the infrared reflectance and infrared output respectively. Finally the type field is set to one of the final static integers: CYLINDER, BOX, or SPHERE. Each of the subclasses adds information about their respective geometries. All the units of these measurements depend on what units the Environment Model File and the Robot module have agreed on.

Once all the objects are sent, the protocol enters the second phase. In the second phase, any positional updates that happen during a time-step are sent to the Viewer. This allows the viewer to display the object at its new coordinates. This information may also be stored for future playback. Figure 14 shows the ViewerUpdateLocation class.

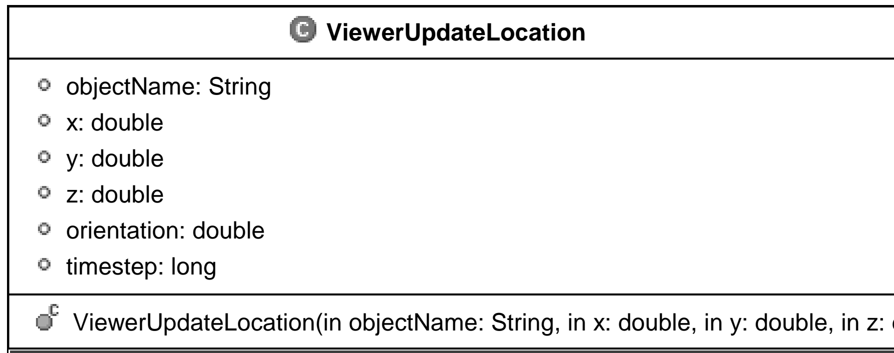

Figure 14: ViewerUpdateLocation class.

The "objectName" field is used by the Viewer to determine which object has moved. The "timestep" field is used for playback in order to determine which events happened together. The x, y, and z fields give the new position in three dimensional space, while the orientation field gives the new rotation about the y-axis. These use the same units as the ViewerObject class.

2.5.3 Control Panel and Environment Protocol

The Control Panel now communicates with the Environment through socket connections. The protocol is a simple request/response protocol as shown in Figure 15.

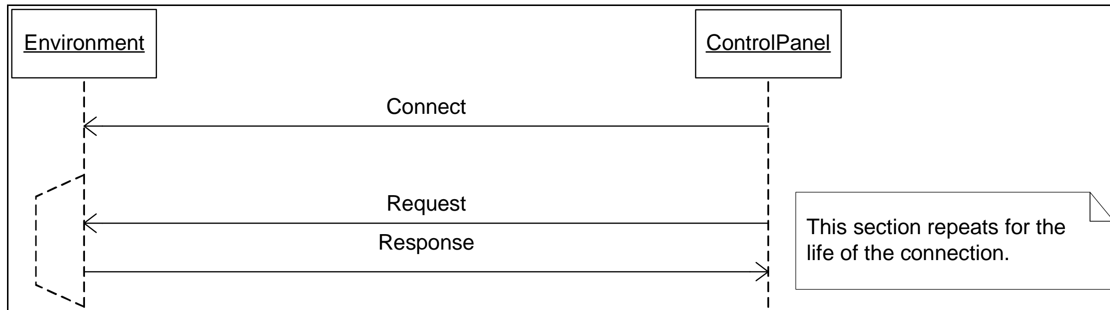

Figure 15: Control Panel/Environment Protocol.

This allows the Control Panel to run on a machine other than what the Environment module is running on. For more information about the Control Panel/Environment protocol, please refer to Aaron Chavez's 2004 Honors Report.

2.5.4 Environment Model File Format

The environment model file contains all the information required to represent an environment. In the present implementation, it also contains definitions of the robots that will be in the environment. The Document Type Definition (DTD) may be found in Esteban Guillen's 2004 MSE Report.

The Environment Builder tool generates an XML file to the DTD specification, which then can be loaded by the simulator. The simulator uses the EnvironmentFileLoader class to parse this XML file.

Chapter 3 Conclusions

This simulator will be a very useful tool. It will allow multi-agent systems (MAS) to be tested, easily, and without the consequences of testing in the physical world. It may also be used for any testing that one may need to do for any performance comparisons of different robot control code.

Our simulator system is made up of several distributable components, one of which is the Environment module. The Environment module plays a key role in the overall simulator system. It is responsible for coordinating and directing the simulation as well as modeling the real world environment. Within the Environment module are several other modules. These modules help the Environment to model the world and orchestrate the simulation.

Chapter 4 Future Work

Several improvements would make this simulator more useful and efficient. I will try to outline these improvements in detail that would be sufficient for anyone to begin work on them. I have divided the improvements into five separate categories: network improvements, map extensions, collision detection, further abstraction, and use of reflection.

4.1 Network Improvements

This simulator is a distributed system. As such, its modules must communicate with each other. This communication is done by sending messages via a network. In this section, I will outline a method of improving this communication and thus adding a speed boost to the simulator.

As noted above, every time-step the Environment module requests events from the robots and blocks waiting for those events. I will propose here a method of pre-fetching those events in some circumstances.

Many of the time-steps a robot does not make a sensor request. This means that the robot is not waiting for any message from the Environment. Thus, the Environment may make several requests for events from the robot *before* processing previous events. This could be done in a separate thread so that event pre-fetching happens in the background. Most of the time spent in event fetching is waiting for the network, not the CPU. So, I believe that this would save much of the time that is spent waiting for messages to travel over the network. Figure 16 shows an example protocol implementing this idea.

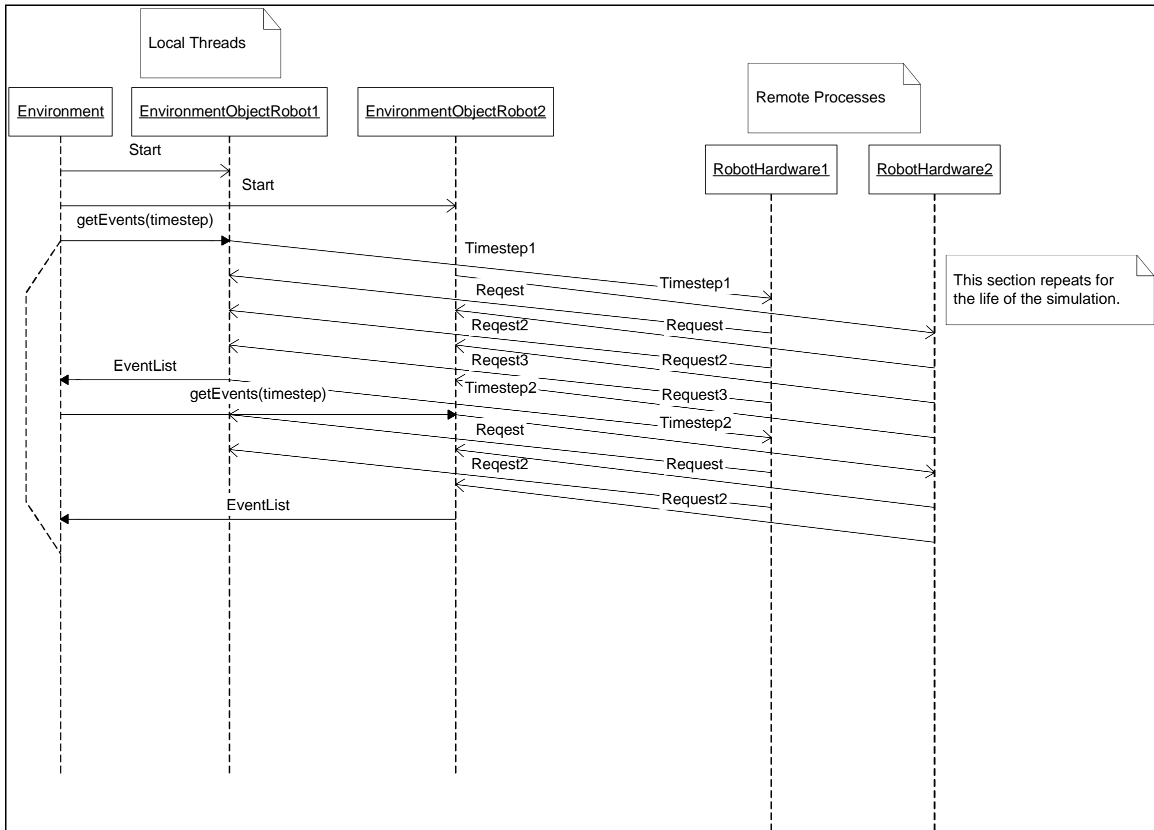

Figure 16: Example of a protocol to improve network latency.

Another idea is to parallelize RobotSensorResponse messages. This, also, could help speed up the Environment module. This could be done by making each EnvironmentObjectRobot a separate Thread. These threads could do the work of sending out the responses.

If many viewers are attached to the Environment, the Environment may be slowed down by sending out so many updates. This could be prevented by setting up a proxy Viewer update module. The Environment would then send any updates to the proxy, who would then be responsible for sending those updates to any Viewer clients.

4.2 EnvironmentMap Extensions and Collision Detection

As detailed previously, the EnvironmentMap component is responsible for keeping track of objects in the environment and for allowing operations on those objects (such as move). Possible future work on this component could include creating domain specific EnvironmentMap components that could be swapped at runtime. The domain specific EnvironmentMap component could potentially be much faster, as it could make domain specific assumptions concerning the Environment.

I will detail one possible extension here. Take, for instance, a grid world. In a grid world, things such as collision detection can be greatly simplified. My idea is to have a grid that can have its granularity set at runtime. The finer grain, the more memory required for the grid, the courser grained, the less accurate. A simulation could be run in a course-grained mode for speed and to observe general characteristics. Each grid would represent some fixed amount of the environment. If we are dealing with an infinite environment, we could also include grids that would represent all points outside our current grid. An object would, at every point of time, be located in one or more grids. If an object wished to move, the grids into which it wished to move would be checked for objects already occupying them. If nothing occupied them, the move would be allowed. If any of the grids were occupied then you could either employ traditional collision detection between the object that wished to move and the objects within the grids, or you could assume that they would collide and disallow the move.

Another method to improve the collision detection could be to use Java3D's built in collision detection mechanism. This mechanism currently implements collision detection for a variety of geometries. It is native code, so it is possibly faster. Also, it may utilize the

graphics processing unit (GPU) of the computer's video card, further offloading computation from the computer running the Environment.

Further work on the EnvironmentMap component could be implementation of more geometries, as well as support for objects that are made up of composed geometries. Support for more physics may also be desired in some simulated domains.

4.3 Further Abstraction and Use of Reflection

There is one component that could be abstracted further, the effectors. Currently all of the effectors are implemented in the Environment class. These should be moved out to a structure similar to the sensors.

Reflection could be used for loading the sensors, effectors, and maps. This would allow extensions (a.k.a. pluggable components) without the need to modify the original code.

APPENDIX A: Running the Environment.

The Environment module may be run by placing all of the simulator packages in your classpath and executing the Environment class. The Environment class takes one argument, which is the environment file to load for the current simulation. After that, additional parameters may be set by using the ControlPanel. By default, the Environment will listen on port 3000 for viewers, port 8000 for robots, and port 9000 for control panels. Once the Environment is successfully started, you should see output similar to that given in Figure 17.

```
C:\eclipse\workspace\RoboSim\bin>java -classpath .
edu.ksu.cis.cooprobot.simulator.environment.Environment ..\TestLoadFiles\
environment\proto-one-robot.xml

0(0 seconds) - Starting Control Panel Server...
Sensor Amount - 3
Sensor Type - bump
Adding Sensor -
ID - 0
Type - 0
Shape:
Type - 3
Sensor Type - heat
Sensor Type - sonar
0(0 seconds) - Environment changed state: 0 --> 1
0(0 seconds) - Starting Robot Server...
0(0 seconds) - Starting VRML Server...
```

Figure 17: Running the Environment.

After the Environment is started, the robots, viewers, and any control panels may be started. The Environment output will indicate whenever a viewer or robot has successfully connected. As soon as all the expected robots have connected, the Environment begins the simulation.

REFERENCES

[1] The Official Project website, *http://www.cis.ksu.edu/~sdeloach/ai/projects/crsim.htm*