ROBOSIM-
ANALYZE AND RESTRUCTURE ENVIRONMENT SIMULATOR

by

THOMAS KAVUKAT

B.Tech , Kerala University, India, 1999

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
2005

Approved by:

Major Professor
Scott Deloach, Ph.D.

ABSTRACT

The Cooperative Robotic Simulator or RoboSim simulates robots, communication between robots, and the interaction between robots and the environment in which it is present. The Robot Application, Hardware Simulator, Environment Simulator, Geometry Package, and Viewer are the five different modules in the Cooperative Robotic Simulator.

RoboSim is a time step based system. At each time step, the system performs following actions. Hardware simulator sends request messages to the Environment module. Environment performs the actions based on requests and sends reply messages to Hardware simulator. It also updates the Viewer with latest locations of robots in the Environment. Each module therefore, synchronizes with Environment at each time step. Thus, performance of Environment Simulator is very important for the performance of the entire system.

This report primarily covers, analysis done on the performance of Environment simulator, various design changes applied to the simulator, and the implementation of a multithreaded architecture on the existing Environment Simulator. It also looks into the integration of Geometry package to Environment module.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I sincerely thank my Major Professor, Dr. Scott A Deloach, for taking me as a part of this project and encouraging me to work on it. I also would like to thank my other committee members, Dr. David A Gustafson and Dr.William J. Hankley for their time and effort as well as for giving me valuable insight.

All my friends were helpful and were ready to hear and give suggestions on various problems I faced along the way. My sincere thanks to you people. Iam specially thankful to my teammates, Balakumar Krishnamurthi, Vikram Raman and Ryan Shelton for helping me at different levels of this project.

Above all, without the grace of God and prayers of many, I could not have attained anything in my life.

Chapter 1 Introduction

1.1 Simulation

Computer simulation of complex systems or processes is often helpful in better understanding of the system. Sometimes it becomes an essential part of the actual implementation because the effort, time, and cost needed to perform complicated hardware prototyping and testing are tremendous.

The primary use of RoboSim is to test various robot applications on a group of virtual heterogeneous robots, where robots communicate and cooperate among themselves, before having to implement the applications on a set of real robots. RoboSim is used to analyze how a group of heterogeneous robots works in various environmental situations.

1.2 Overview of RoboSim

There are seven major modules in RoboSim. Each of these modules communicates with any of the other modules through sockets so they could be executed on different machines. Below is a brief description of the entire system.

Figure 1 gives an overall system design of RoboSim and Figure 2 shows the major components of the RoboSim. Robot Control code issues commands to the robots by calling different APIs[1] exposed by the Hardware Simulator Interface. Remote Control module connects to RobotControlCode. RobotControl code can issue commands based

[1] These API declarations are same for both the actual and simulated robots so that control code, after testing in RoboSim, could be directly executed on a real robot.

on the inputs from Remote Control module. Remote Control is a user interface that accepts inputs from external user through an I/O device. The Hardware Simulator Interface API on invoking generates new requests, at the Hardware Simulator level, which is then sent to Environment Simulator.

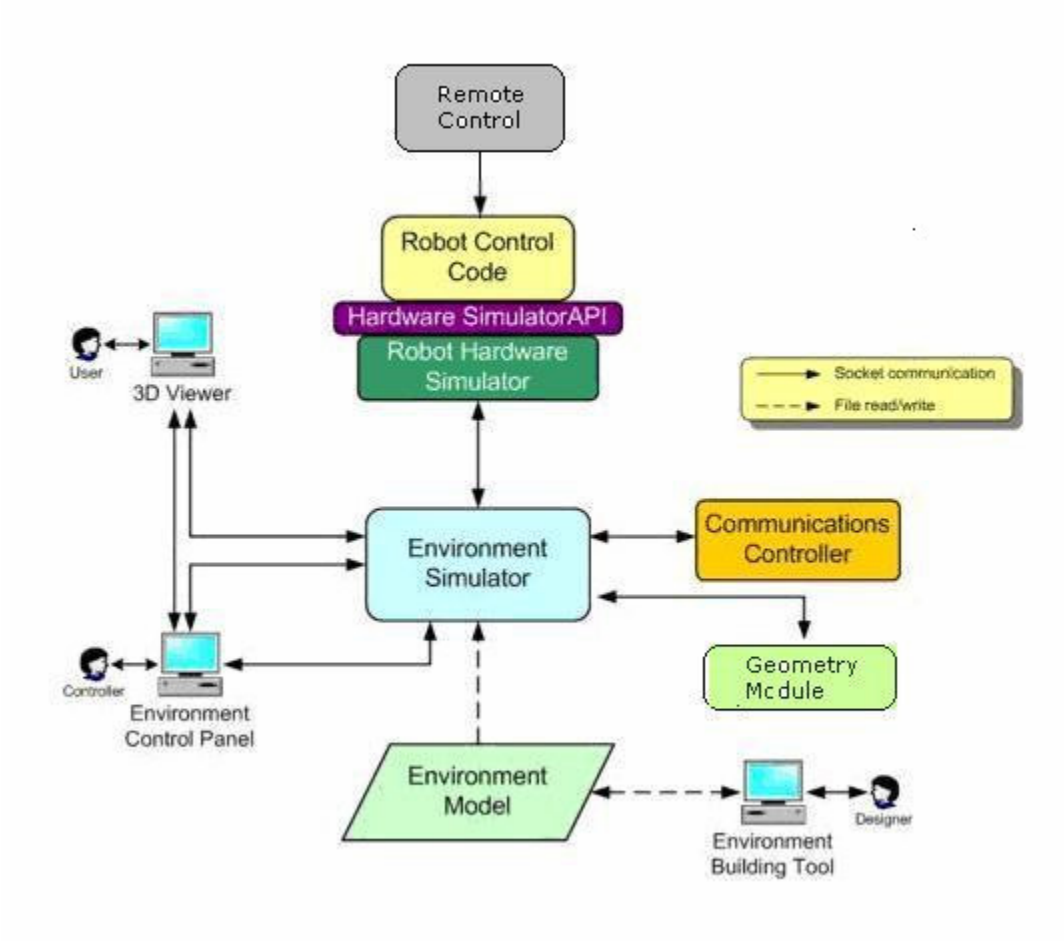


Figure 1: System Overview

The Environment module simulates the environment in which robots are simulated. Details of environment are loaded from an XML file when the module starts up. The number of robots present and the properties of each robot are described in the file. The Hardware Simulator simulates the hardware of the robot by translating the

robotic operations as requests to the Environment. Hardware Simulator and Environment module interacts on a time step basis. The Environment Simulator, on starting, sets up the time step size and sends this information to the Hardware Simulator. Later, when the RobotControl Code issues commands, the Hardware Simulator breaks the operations issued by the robot code into the time step size. For example, let the time

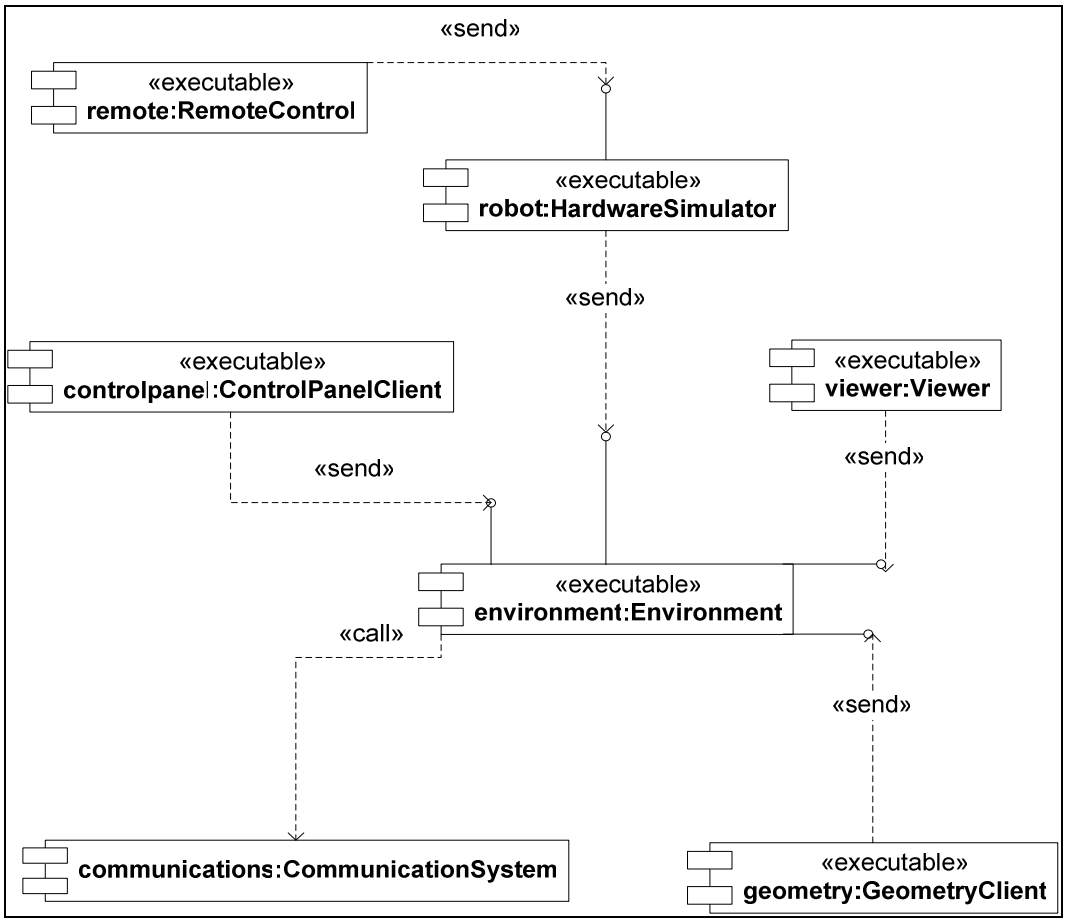


Figure 2: RoboSim Components

step size set by the Environment is 500ms. Now, if velocity of robot set by robot control code is 500 m/s then hardware simulator knows that the robot can move a maximum of 250 mm every time step. Therefore, when a move command with 750mm is issued by the

control code the hardware simulator will split the move command into three separate move commands of 250mm each for three consecutive time step.

In each time step, the Environment Simulator receives three major types of requests from the Hardware simulator, ACTION which includes either move, turn or tag requests, SENSOR which includes either sonar, laser or heat requests and COMMEVENT for sending messages to another robot's hardware simulator.

On receiving the requests, the Environment bundles all the ACTION and sonar SENSOR requests and sends it to geometry module. The geometry module is responsible for the geometric representation of the environmental objects, collision detection, and distance finding. By finding intersection between objects, it can prevent objects from overlapping in the environment and simulate sonar and laser range finding. The Environment Map module of the Environment Simulator processes SENSOR requests other than sonar types. Environment receives the responses for each of the requests send to the Geometry module. Responses for SENSOR requests are then send back to corresponding robot's Hardware Simulator.

The COMMEVENT request from each robot contains the destination robot to where it wants to send the messages. Message Module of Environment does this processing and sends the messages to appropriate Robot's Hardware Simulator.

The Environment calculates the new positions of the Robots at the end of each time step based on the response from Geometry module. These new positions are sent to the Viewer module to be displayed to the end user. One coordinate unit in viewer is equivalent to 1 meter in real world. One or more viewers can be connected to the Environment.

The Control Panel module is a user interface to control the RoboSim system like, to load a new environment file, to start, or to stop the simulation. It communicates with the Environment Module.

1.3 Literature Reviews

There have been many development works in Robotic Simulation. Below I discuss about two papers that describes about an environment for robotic simulation and the previous work done on RoboSim.

[1] discuss about a development environment that helps in software development for mobile robotics. The primary module in this system is a Robot Daemon that acts as a gateway to either physical robot or a simulated one. A command line interface allows the user to interact with the robot using a combination of semantic primitives. This allows some simple functions to be implemented interactively. This development environment allows control code written for a robot to be tested on a simulated robot before running on an actual code. The user can switch between real robot and the simulated one.

[2] discuss about a Mobile Robot Simulation, called Webots[2] that helps in developing a robot controller program in either C, C++ or Java. The controller programs could be tested on the simulated robots and then transferred to real mobile robots using Webots. The Webots includes complete library of sensors and actuators, which enables it to simulate a real robot. You can create complex environments for your mobile robot simulations using advanced hardware accelerated OpenGL technologies. The simulation system used in Webots uses virtual time similar to the RoboSim.

[2] It is a commercially available tool developed by Cyberbotics Ltd.

[6] details the initial implementation of the Environment Simulator in RoboSim. The `Environment` class is the main class in this implementation. The Environment Simulator drives the simulation. It has two phases: initial and execution. In the initial phase, it reads from an XML file and waits for all the robot clients specified in the XML file to connect to the Environment. In the execution phase it does the following infinitely, first, time steps are sent to the robot clients, robots on receiving the time step sends the requests for the current time step, these requests are processed by the Environment and responses are sent back to the appropriate robot clients.

1.4 Objectives

RoboSim simulates a group of robots cooperating to achieve a goal. However, the performance of RoboSim decreases dramatically as the number of robots in the system increases. Another significant concern is the stability and scalability of the system. In this report, we focus on analyzing and restructuring the existing Environment Simulator to improve its performance and stabilize it. In addition, a new module, the Geometry module, is to be added to the RoboSim system.

The report is structured as follows. In the first part, the existing architecture of Environment module and the analysis performed on it using a profiler is discussed. The second part discusses various designs that were considered for the Environment and the final design that was chosen. At the end, the final implementation is discussed with geometry package integrated into the system.

The final chapter summarizes the work done and provides some suggestions for future. Appendix contains an elaborate guide on how to start the simulator. It gives a good idea about adding new functionalities to the existing Environment module.

Chapter 2 Environment Simulator – Analysis

RoboSim is a time step based simulation, i.e. each robot advances to the next time step if all others have finished computations of current time step. Since all robot operations are simulated in the Environment Module, robots have to send their requests for current time step to the Environment Module for a proper simulation. All the modules proceed to the next time step only when the environment has finished computations of the current time step. Overall performance of the RoboSim system therefore is depended on the performance of the Environment Module. Again, Environment Module starts computation for the next time step only after all the robot clients have sent their requests. This makes it very important that robot clients too handle the responses for the current time step from Environment fast and send the requests for the next time step.

Currently, the RoboSim slows down when the number of robots connected to Environment Simulator increases. Therefore, it is important to analyze and make sure that Environment Module has no processing or memory hotspots[3]. There are several methods of finding the processing and memory hotspots in an application. A very basic method is to calculate the time taken by a set of executing statements using print statements before and after the statements. For this method, the programmer should analyze every part of the code and do extensive testing. Another method is finding the probable sets of statements that can cause memory/processing hotspots and then do an extensive analysis on it. This way you can narrow down the search on the entire source code. Profiling tools are helpful in finding the probable memory/processing hotspots. Many profiling tools are

[3] Hotspots are the places in the code that affects the overall performance of the profiled application.

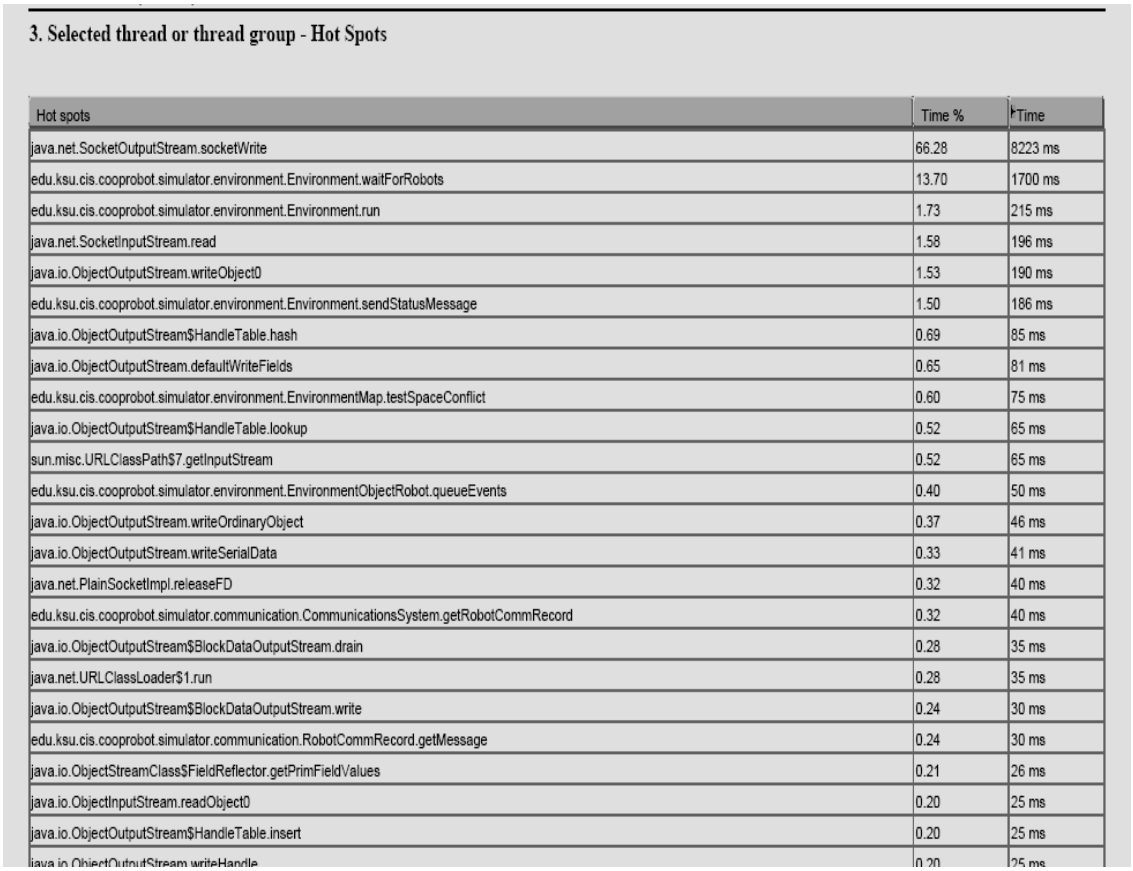
available that can find out memory/processing hot spots for you. In the next section, I discuss the profiling done on the Environment Simulator to find these hotspots and analysis done on these hot spots.

2.1 Profiling

To analyze the performance of any application profiling tools are very useful. It helps us in identifying the bottlenecks of an application which otherwise is too difficult to find. We used Borland Profiler to find the CPU usage and memory hotspots in the Environment and the Hardware Simulator (robot client)[4].

[4] In this report, I use the terms Hardware Simulator and robot client interchangeably; Hardware Simulator is the thread that communicates with Environment.

2.1.1 CPU Profiling



3. Selected thread or thread group - Hot Spots

Hot spots	Time %	Time
java.net.SocketOutputStream.socketWrite	66.28	8223 ms
edu.ksu.cis.cooprobot.simulator.environment.Environment.waitForRobots	13.70	1700 ms
edu.ksu.cis.cooprobot.simulator.environment.Environment.run	1.73	215 ms
java.net.SocketInputStream.read	1.58	196 ms
java.io.ObjectOutputStream.writeObject0	1.53	190 ms
edu.ksu.cis.cooprobot.simulator.environment.Environment.sendStatusMessage	1.50	186 ms
java.io.ObjectOutputStream$HandleTable.hash	0.69	85 ms
java.io.ObjectOutputStream.defaultWriteFields	0.65	81 ms
edu.ksu.cis.cooprobot.simulator.environment.EnvironmentMap.testSpaceConflict	0.60	75 ms
java.io.ObjectOutputStream$HandleTable.lookup	0.52	65 ms
sun.misc.URLClassPath$7.getInputStream	0.52	65 ms
edu.ksu.cis.cooprobot.simulator.environment.EnvironmentObjectRobot.queueEvents	0.40	50 ms
java.io.ObjectOutputStream.writeOrdinaryObject	0.37	46 ms
java.io.ObjectOutputStream.writeSerialData	0.33	41 ms
java.net.PlainSocketImpl.releaseFD	0.32	40 ms
edu.ksu.cis.cooprobot.simulator.communication.CommunicationsSystem.getRobotCommRecord	0.32	40 ms
java.io.ObjectOutputStream$BlockDataOutputStream.drain	0.28	35 ms
java.net.URLClassLoader$1.run	0.28	35 ms
java.io.ObjectOutputStream$BlockDataOutputStream.write	0.24	30 ms
edu.ksu.cis.cooprobot.simulator.communication.RobotCommRecord.getMessage	0.24	30 ms
java.io.ObjectStreamClass$FieldReflector.getPrimFieldValues	0.21	26 ms
java.io.ObjectInputStream.readObject0	0.20	25 ms
java.io.ObjectOutputStream$HandleTable.insert	0.20	25 ms
java.io.ObjectOutputStream.writeHandle	0.20	25 ms

Figure 3: Environment Simulator CPU usage.

Figure 3 is the output of profiler depicting the CPU usage hotspots of Environment Simulator. A close look of Figure 3 shows that almost 67% of CPU usage of EM is in writing to the `ObjectOutputStream`. Figure 4 is the profiler output for the Hardware Simulator. Similar to the Environment Simulator analyzing the profiler output of Hardware Simulator shows that the bottleneck in here too is writing into the socket. Around 95% of the execution time of Hardware Simulator is spent on writing to the `ObjectOutputStream`.

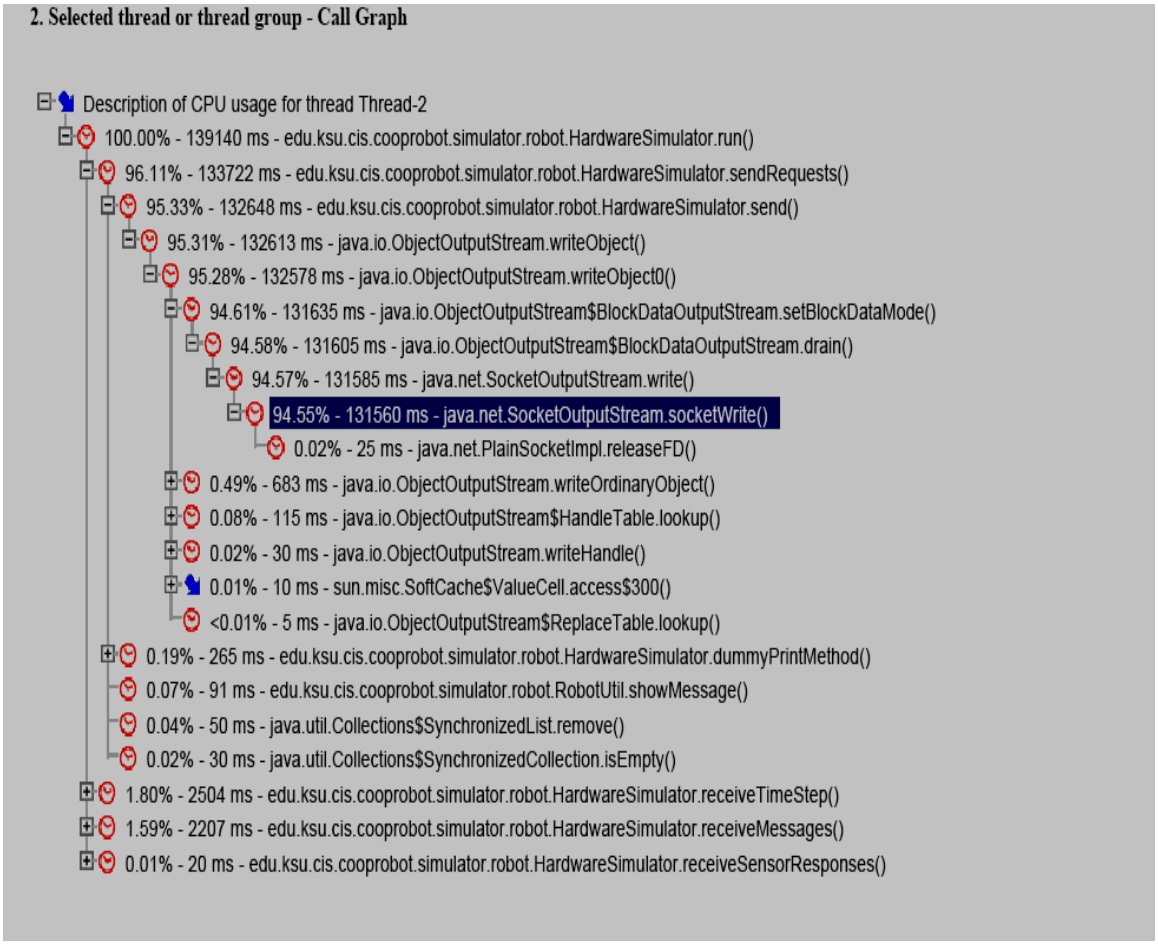


Figure 4: Hardware Simulator CPU usage.

One of the possible solutions to improve the performance of Environment is to assume that writing to `Java` output stream takes more time and so do them parallel in different threads. The existing Environment was a sequential application; it receives requests from each client, one after the other. This sequential execution might be slowing down the Environment. The other solution is to look at the performance issues of using `Java ObjectOutputStream` and fine-tune the same. In java, Object serialization is the mechanism that allows you to read/write full-blown objects to byte streams. Implementing the user objects as java `Serializable` interface allows you to serialize

the objects by passing them to the `writeObject()` method of `ObjectOutputStream`. `ObjectOutputStream` automates the process of writing the class metadata and instance fields to the stream. In other words, it does all the serialization work for you [2]. This causes the `writeObject()` function to be slow in streaming the data across the network. Instead of using `Serializable` objects, we could use `Externalizable` objects to be sent across the network. When you declare that, an object is `Externalizable` you assume full responsibility for writing the object's state to the stream. `ObjectOutputStream` no longer automates the process of writing your class's metadata and instance fields to the stream.

Another important observation can be made from the protocol between Environment and Robot shown in Figure 5. The Environment sends all the sensor responses to robots at the end of each time step. When the Environment is ready to start next time step, the robots will be processing these responses causing a delay in sending requests for the next time step. This delay could be avoided by creating a new thread in hardware simulator so that a different thread handles receiving responses.

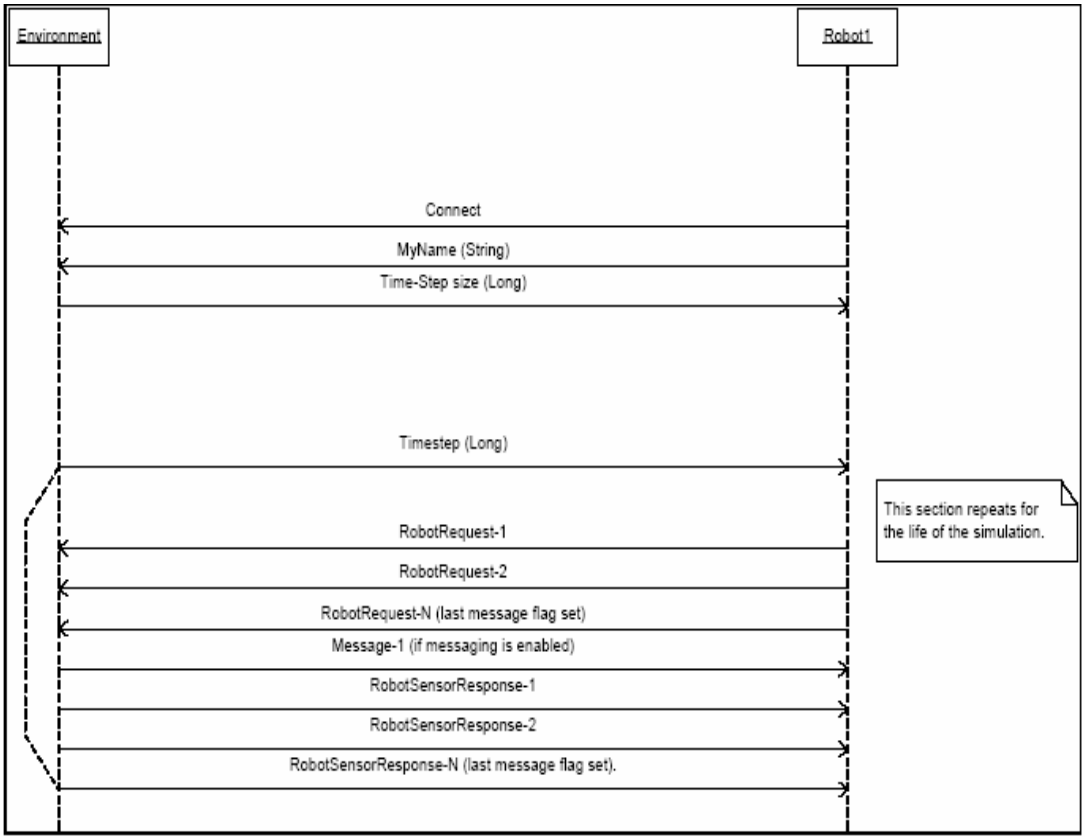


Figure 5: Robot/Environment Protocol.

2.1.2 Memory Profiling

Figure 6 is the profiler output that shows the memory hotspots of Hardware Simulator. From Figure 6, it is clear that too many instances of `Long` objects are created in Hardware Simulator. This resulted in the Robot Application crashing with java.lang.OutOfMemoryError. In every time step the Hardware Simulator starts sending requests to the Environment after receiving the time step information from Environment and the Environment creates a new instance of `java.lang.Long` every time step. This causes the `ObjectOutputStream` to cache the data at sending and receiving end of the socket [1]; the garbage collector does not clean up this cache and application runs out

of memory. Instead of sending a `java.lang.Long` object, streaming the time step value directly to the socket using `writeLong`() function of `ObjectOutputStream` is a better method.

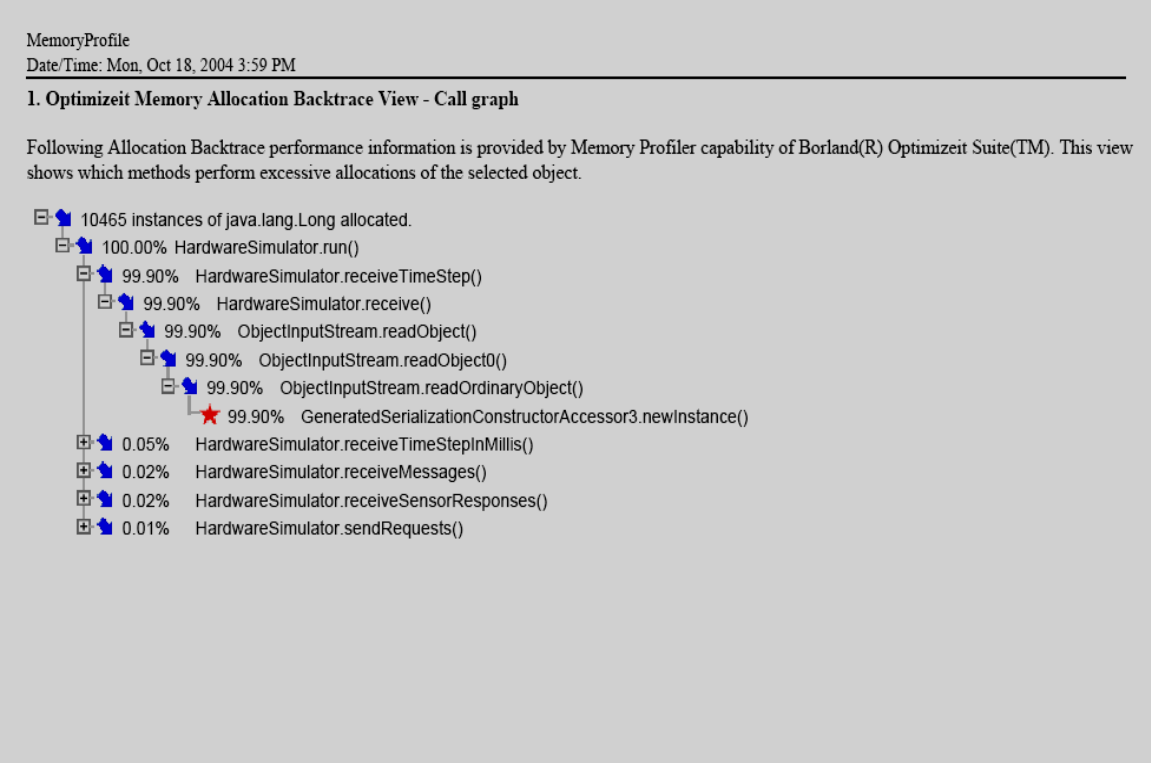


Figure 6: Hardware Simulator – Memory hotspots.

According to [2], it is generally suggested that the `ObjectOutputStream` connection be reset after data is written to it. Doing a `reset`() operation on `ObjectOutputStream` allows the garbage collector to clean up the cached data of the connection. Failure to perform this cleanup can cause the cache to grow and eventually crash the receiving and sending applications. However, a `reset`() operation takes time and can cause system to slow down. Therefore resetting the socket should not be done after every `writeObject`() call, rather it should only be done in proper intervals.

The profiling discussed above enabled me to find the major bottlenecks in the Environment Simulator. The Environment was implemented as a sequential application. It processes all the robot socket connection in a single thread. Since writing to the socket is the performance issue, we could handle this in separate threads to improve the performance. In the next chapter, I discuss two multi-threaded architectures, and an implementation of one of the architectures is discussed in detail.

Chapter 3 Redesigning Environment Simulator

As discussed in Chapter 2, the major bottlenecks of the Environment Simulator are based on writing to the socket connection. This time consuming process, writing to socket, can be done in multiple threads to improve the performance. We consider two designs:

1 Two threads per client – In this design, the Environment starts two threads for each robot client, one thread handles the input connection, and second thread handles the output connection.

2 One thread per client – In this design, the Environment starts a single thread to handle both input and output connections of each robot client.

Below I discuss these two designs in detail.

3.1 Two threads per client

In this design, two threads, one to handle the input socket connection and the other to handle the output connection, are started by the Environment Simulator for each robot client. The overall scenario is depicted in Figure 7.

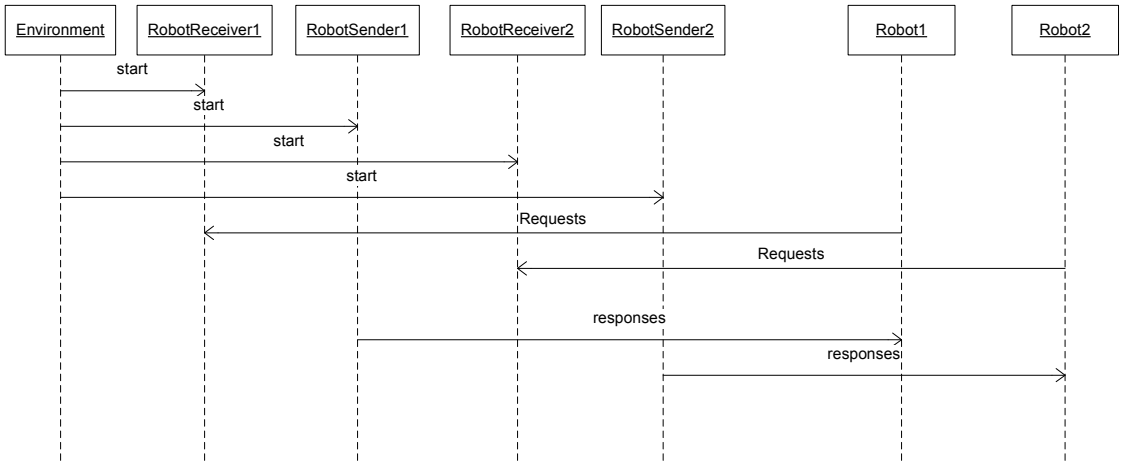


Figure 7: Two Thread design.

In Figure 7 for Robot1 and Robot2, the Environment starts four threads RobotSender1, RobotSender2, RobotReceiver1, and RobotReceiver2. The receiver threads, RobotReceiver1 and RobotReceiver2 read the request from clients Robot1 and Robot2 respectively. The Environment processes these requests and the sender threads, RobotSender1 and RobotSender2, send corresponding responses. This design ensures that responses are streamed as soon as they are generated.

One of the major advantages of this design is achieving asynchrony between the Environment and the robot client. If Robot client has two separate sender and receiver threads instead of current single thread model, the sender thread could send the requests for time steps ahead of the current time step if they do not need to wait for any responses from the Environment in the current time step.

3.2 Single Thread per Client

In this design, a single thread handles both input and output socket connection. This is a much simpler design as the existing Hardware Simulator in the Robot client need not be modified to take full advantage of this design. Moreover maintaining more threads is always more complex as the communication between threads is an overhead in multithreading. In Figure 8, which shows the design overview, two threads, RobotThread1 and RobotThread2 are started for each client, Robot1 and Robot2 respectively. Same threads receive requests from robots and sends responses back to them.

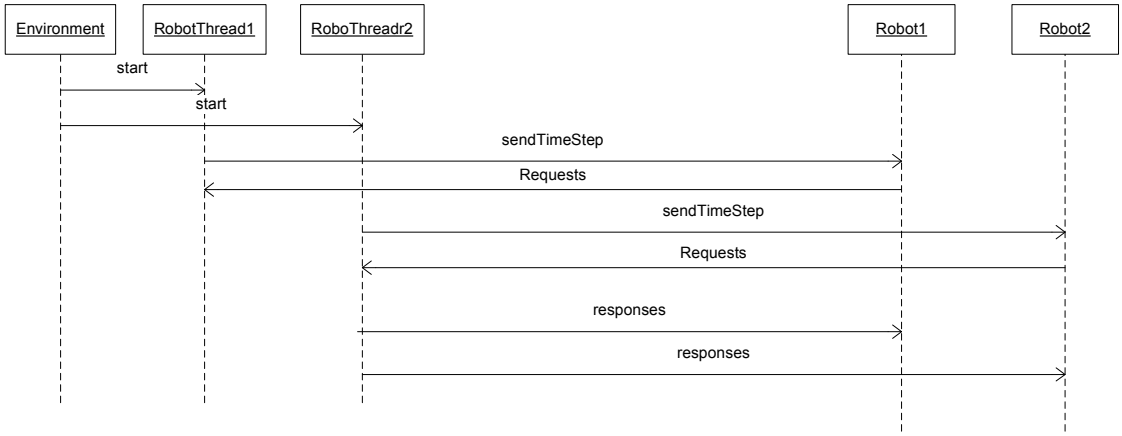


Figure 8: Single Thread Design.

In the next section, I discuss how Environment Simulator was modified to implement the Single Thread per Client design.

3.3 Implementation of Single Thread Design

This implementation is the Single Thread per Client design we discussed in the previous section. This will help in making the Environment multi threaded and is much simpler than the Two Threads per Client design. Maintenance of threads is an overhead in multi threading; therefore, reducing the number of threads makes it easier to port a sequential application to a multi threading application.

3.3.1 Requirements

There are two main requirements for this implementation. First, to convert the current sequential implementation of Environment to a multi threaded implementation. Second, to add Geometry module to RoboSim and to establish a communication protocol between the Environment and Geometry modules.

3.3.1 Overview

The Environment Simulator was modified to include a new package called EnvironmentClientInterface. The main function of the classes in this package is to handle the communication with other clients as separate threads.

Figure 9 shows the component diagram of this new Environment. There are many modules in the current Environment module which work together to simulate the Environment. In this report, we discuss in detail only the EnvironmentClientInterface module. This new module makes use of the other modules to produce a proper Environment Simulation. The other major modules of the Environment Simulator are the Environment Map component, the Collision Detection component, the Environment Object component, the Robot components and the Sensor Components, which remains the same as discussed in [6].

The `EnvironmentMap` component represents the environment. It implements methods to manipulate and query the state of the Environment. `EnvironmentMap` contains dynamic or static objects implemented by Environment Object module. Each `EnvironmentObject` class contains information about its position in environment, its orientation, and a geometry describing its shape. The `CollisionDetection` component performs collision detection on geometry structure of the `EnvironmentObject`. A special type of `EnvironmentObject` is an `EnvironmentObjectRobot`. An `EnvironmentObjectRobot` may have one or more sensors located at different places of the robot. These sensors are implemented in `RobotSensor` component. For a detailed discussion of these modules, please refer to [6].

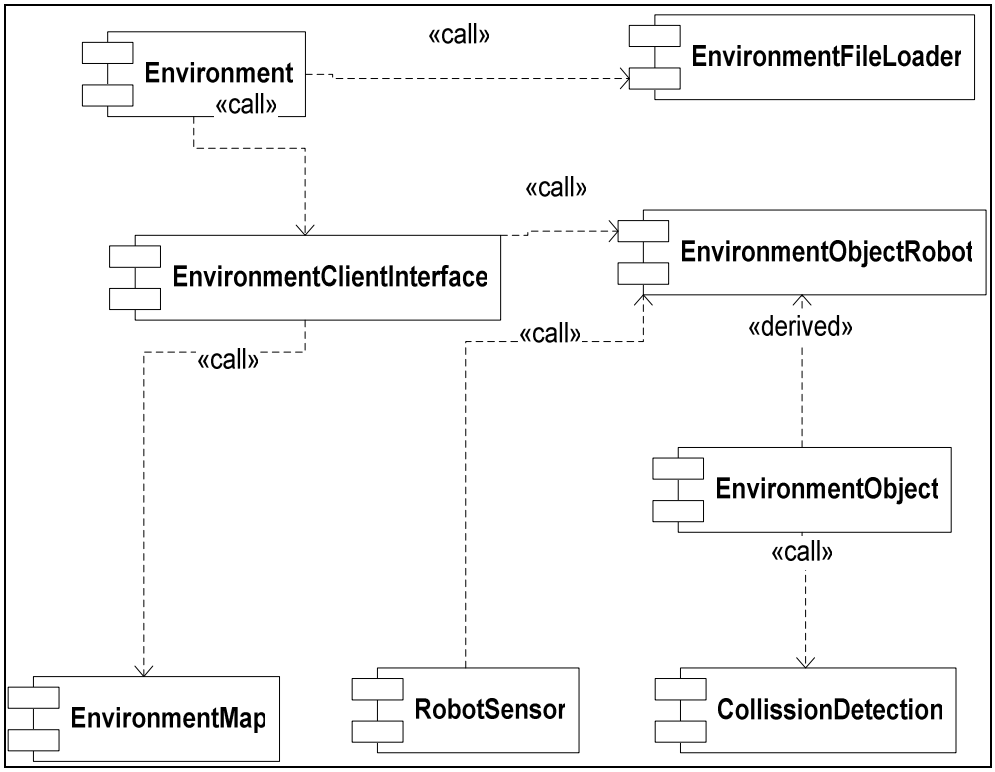


Figure 9: Environment modules

A new component called `Geometry` was added to the existing RoboSim system. Since Environment has to communicate with this new module, the Environment module had to be modified accordingly. The geometry module is responsible for the geometric representation of the environmental objects, collision detection, and distance finding. By finding intersection between objects, it can prevent objects from overlapping in the environment and simulate sonar and laser range finding. In the previous version of RoboSim, this module was part of the Environment. In the new system, `Geometry` runs as a separate process.

Below I describe in detail the implementation of `EnvironmentClientInterface` module. A detailed class diagram depicting all the

classes of `EnvironmentClientInterface` module is given in Figure 10. There are two important classes in this module. One is the `ClientWorkerCoordinator` and the other is the `ClientWorker` class. Based on the different types of clients connecting to the Environment, we added new specializations of the ClientWorker class to this module. Currently the system has three different kinds of clients: Robots[5], Viewers[6], and Geometry Client. Therefore, the `EnvironmentClientInterface` has three different classes handling requests of these clients.

[5] Robots classes are inherited from ArRobot or Scout class.

[6] Viewers can be of type Viewer2D class or CRS3DViewer class.

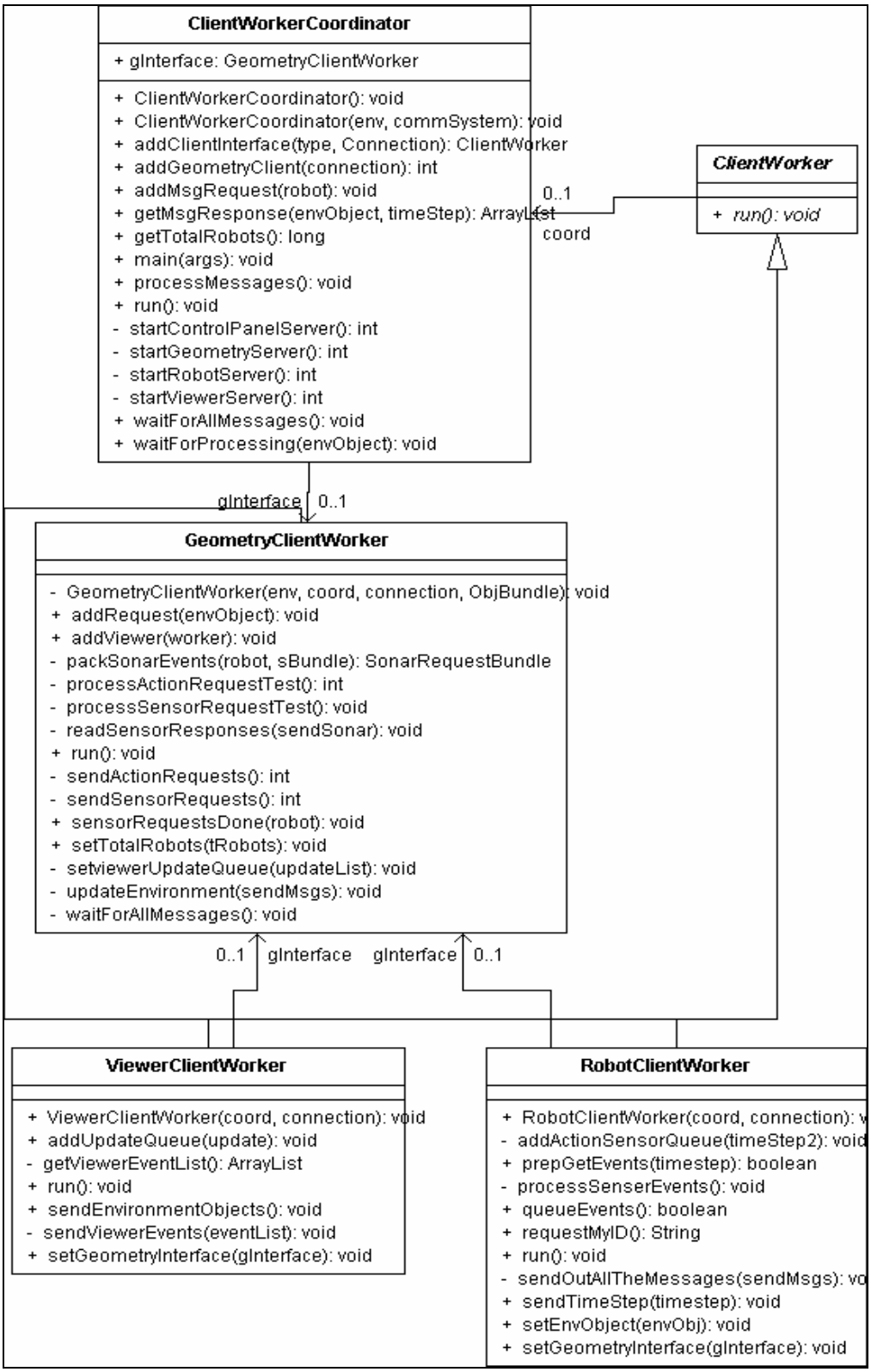


Figure 10: Class Diagram.

Below these classes are briefly described with sequence diagrams depicting their functionality.

1) ClientWorkerCoordinator

The `ClientWorkerCoordinator` manages all the client worker threads. Three different types of clients connect to the environment robots, viewers, and geometry modules. It starts three servers that listen for these clients to connect to the environment at predefined ports, 8000 for the robots, 3000 for the viewers and 10000 for the geometry modules. For each client that connects to the environment, the `ClientWorkerCoordinator` starts a new thread that will receive and sends the messages to client. The threads will start execution only after all the robot clients have connected to the environment. In the current implementation, robots cannot dynamically be added or removed from the environment. There are three kinds of robot requests that environment handles, COMMEVENT, ACTION and SENSOR. COMMEVENT requests contain messages one robot wants to send to another robot. Coordinator processes COMMEVENT requests of robots.

The `ClientWorkerCoordinator` loops infinitely doing the following. It starts a new `ViewerClientWorker` thread for any new viewer connected to the environment. Then the coordinator waits for all `RobotClientWorker` objects to call the function `processMessage`(). The last thread that calls this function triggers the `ClientWorkerCoordinator` to start processing the COMMEVENT requests of all the robot clients for the current time step. `RobotClientWorker` objects will then call

`getMessage()` method of coordinator to get the messages that has to be sent to the robot clients.

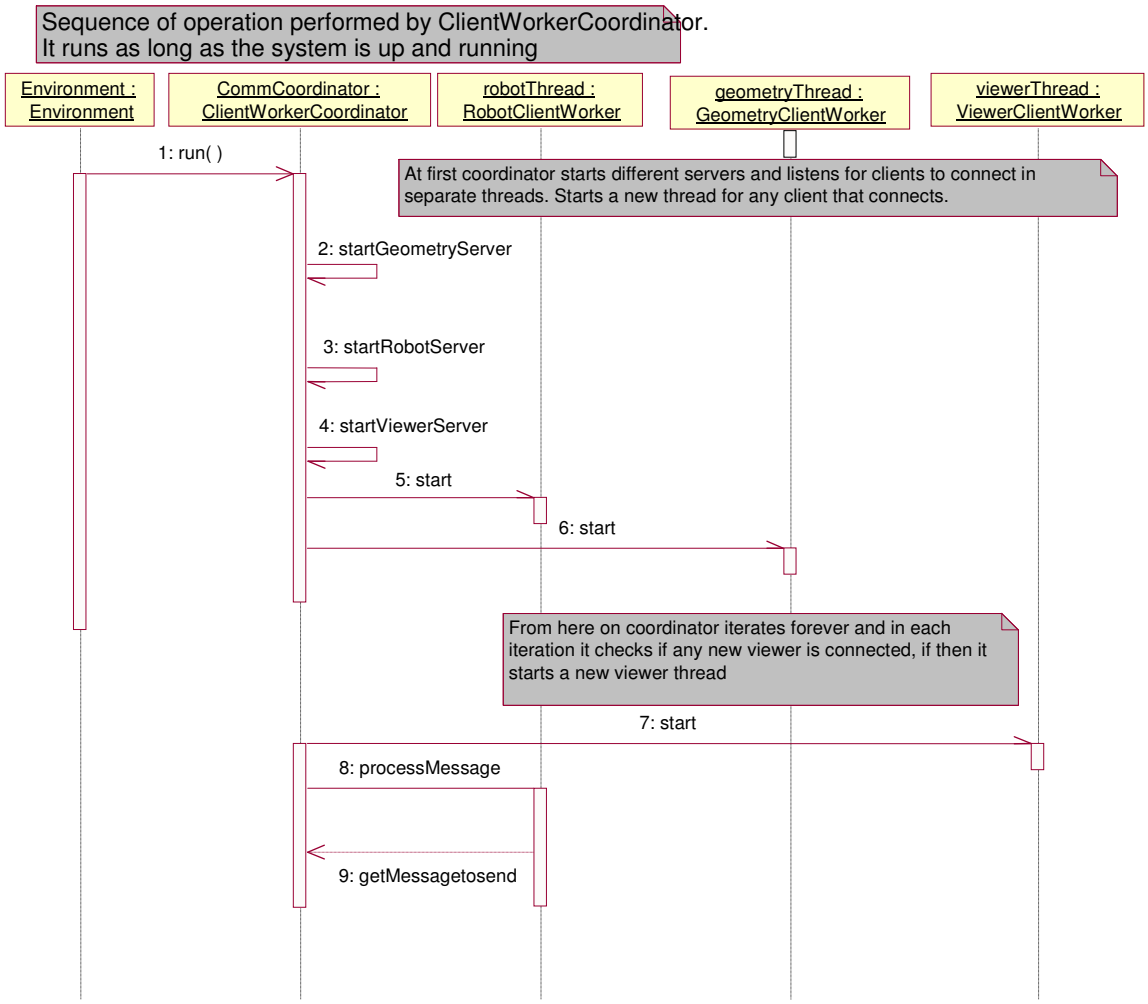


Figure 11: ClientWorkerCoordinator - Sequence Diagram.

2 RobotClientWorker

The `RobotClientWorker` handles the communication and the requests of each client. Each of the `RobotClientWorker` object does the following infinitely. First, it sends the current time step to the robot client, and then waits for client's requests for the current

time step. The next two method calls trigger the `GeometryClientWorker` and `ClientCommCoordinator` threads to resume execution as these will be blocked in the beginning of every iteration by a `wait`() function call waiting for all the `RobotClientWorker` thread to receive requests for the current time step. Then, the `GeometryClientWorker` processes the SENSOR and ACTION requests and the `ClientCommCoordinator` processes the COMMEVENT requests. The `Robotclientworker` then calls the `getMessages`() function of coordinator object to get the messages to be send to the robot client and sends it to robot client. Similarly the responses for SENSOR requests are got by calling `getResponses`() function of GeometryClient. It then goes back to the top of the loop.

The communication between Hardware Simulator and the `RobotClientWorker` thread follows a specific protocol. There are two phases in the communication between the Environment and the Hardware Simulator. In the first phase when a new robot connects, it sends its name to the Environment. The Environment then sends the time step size. The `run()` method of the `ClientWorkerCoordinator` handles this initialization phase. The `RobotClientWorker` thread handles the second phase. In each time step, the `RobotClientWorker` thread sends the time step information to the Hardware Simulator. This causes the HardwareSimulator to send all the requests to Environment for current time step. After processing these requests, the `RobotClientWorker` thread sends the responses back to HardwareSimulator. A detailed description of the protocol is discussed in [6].

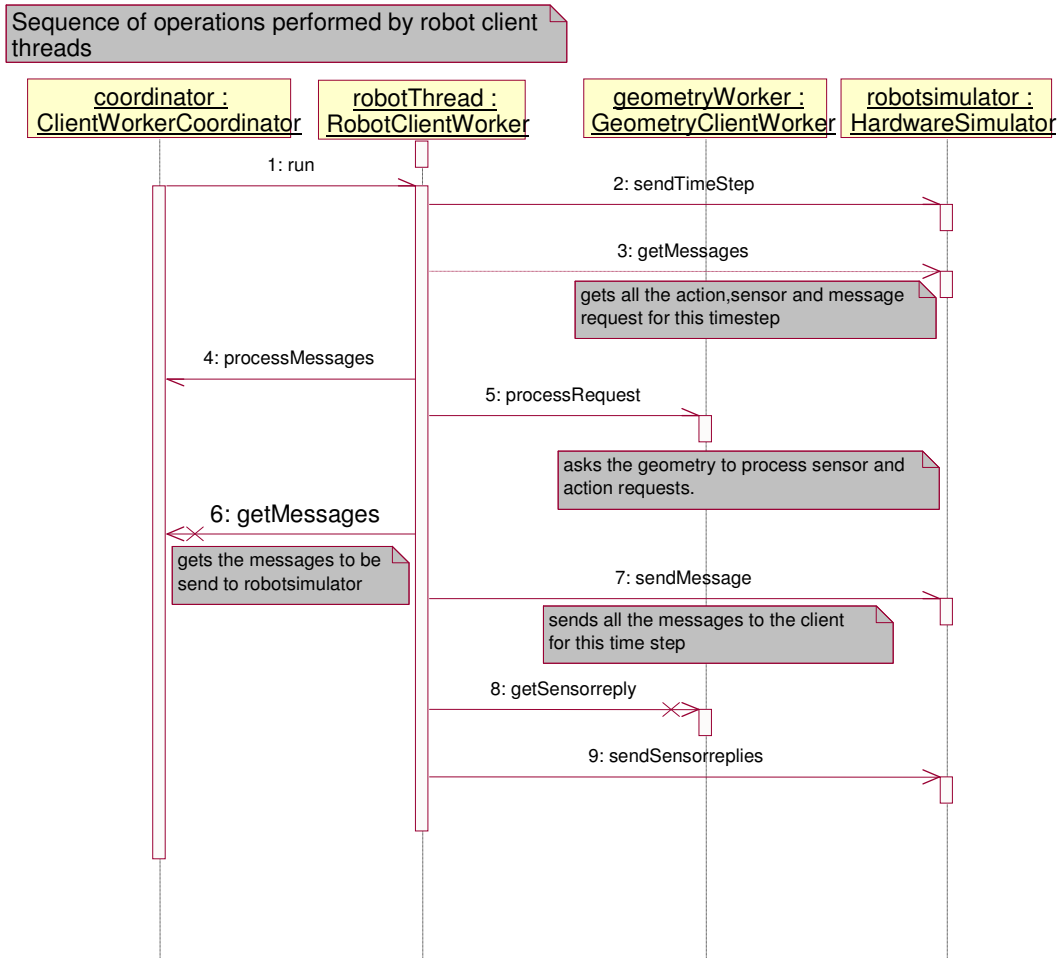


Figure 12: RobotClientWorker – Sequence Diagram.

3 GeometryClientWorker

The `GeometryClientWorker` thread interfaces with the Geometry module, which implements the simulation of ACTION events and SONAR SENSOR events of robots. The communication between `GeometryClientWorker` thread and Geometry Module follows a simple protocol as shown in Figure 13.

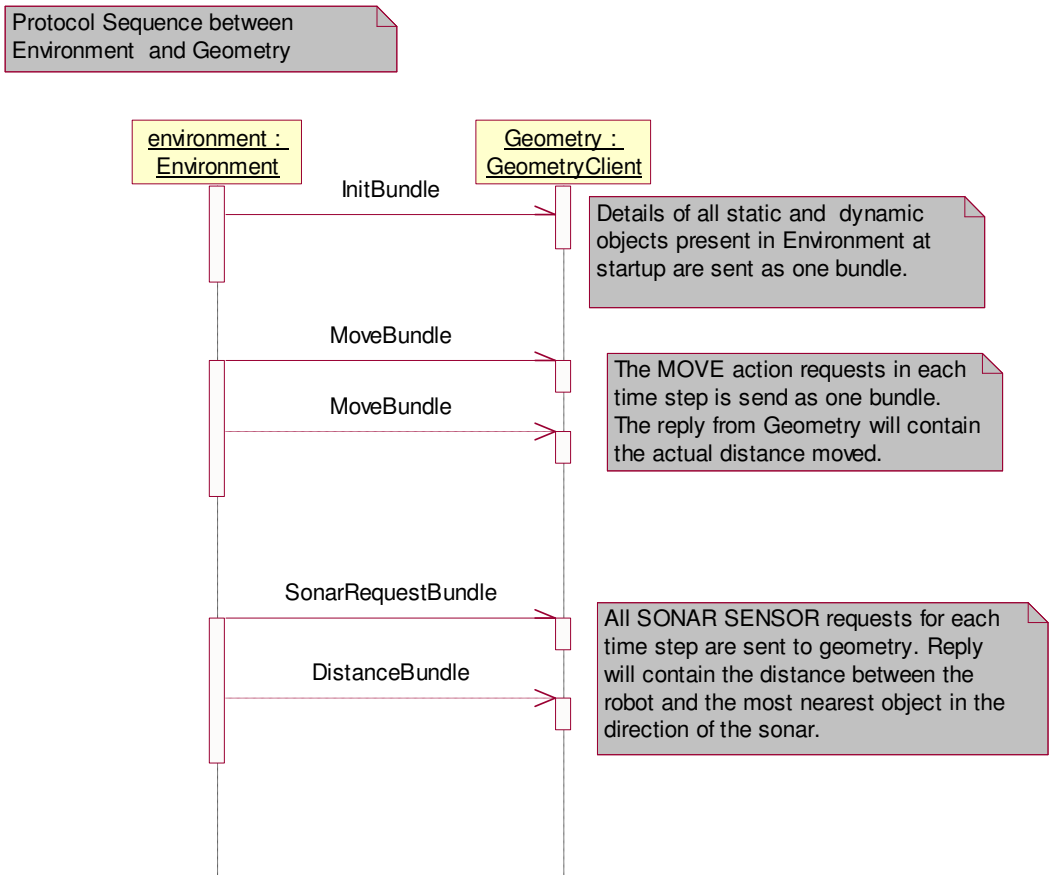


Figure 13: Environment/Geometry Protocol.

The `GeometryClientWorker` thread is initially blocked by `wait()` function so that all the `RobotClientWorker` threads could add their ACTION and SENSOR requests for the current time step. It then sends the ACTION and SONAR requests to the Geometry Module. Other SENSOR requests are simulated as a component within the Environment module. After requests are sent, the thread receives the responses. If the ACTION requests are successful then the Geometry Module sends back the distance

moved by the robots. The positions of the robots in the Environment are then updated by adding the returned value to the current position of the robots.

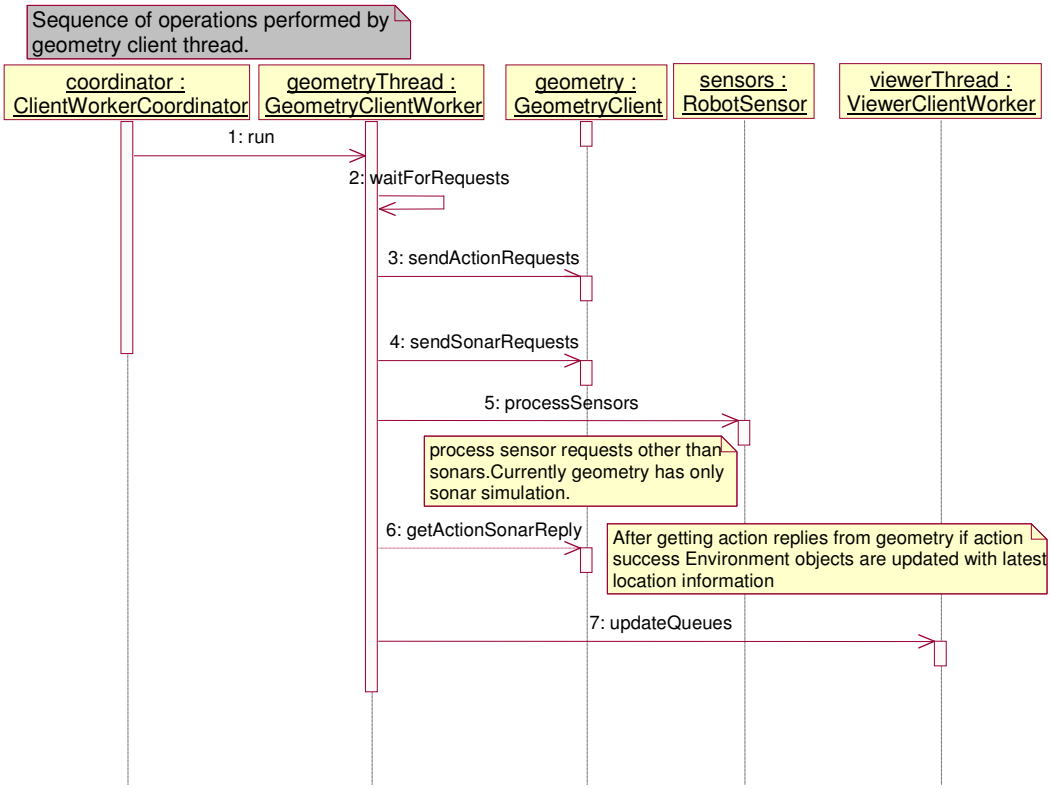


Figure 14: GeometryClientWorker – Sequence Diagram.

These updated locations are added to the queue of the all `ViewerClientWorker` objects. `ViewerClientWorker` thread then sends these updated locations of robots in Environment to their corresponding viewer client.

4 ViewerClientWorker

The `ViewerClientWorker` thread interfaces with different types of viewers; 2DViewer, 3DViewer, and robot specific viewer. The communication between `ViewerClientWorker` thread and all these viewers follows the same protocol.

Similar to the Hardware Simulator/Environment protocol, this protocol also has two phases. In the initial phase, information such as position, shape, and orientation of all objects in Environment is sent to all types of viewer clients that connect to the Environment. After this initialization phase, any updated information of the objects in each time step is sent. The protocol is discussed in detail in [6]. The `ViewerClientWorker` thread waits for updated locations by calling a `wait()` call. When new information is added to the queue of `ViewerClientWorker` thread, it is unblocked.

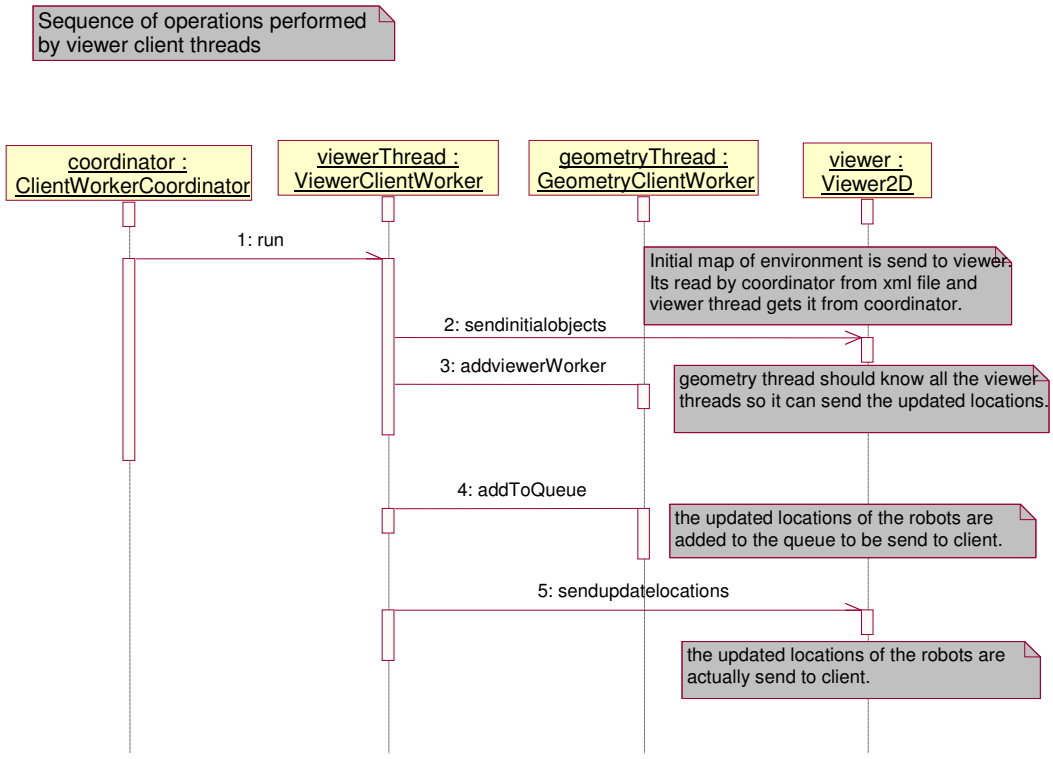


Figure 15: ViewerClientWorker – Sequence Diagram.

3.4 Analysis of new design

The new Environment was integrated with the existing setup of RoboSim and tested for all required functionalities and for any improvement in performance.

An extensive testing of performance of the new Environment was not performed. A basic testing scenario was conducted by calculating the time taken to complete 5000 time steps for both the old and new system, with ten robots that will do the following infinitely, move forward for 10 units(10 meters in real world), then turn around and again move back to old position. This test showed no significant increase in performance. This could be due to communication between threads. Blocking and unblocking threads are necessary for the Environment to synchronize with all the modules. For example, collision detection can be performed on only one object at a time. This causes other threads to be blocked and then unblocked. Functions like collision detection are called synchronization regions. In the current Environment architecture, the number of synchronization regions are more than the old Environment since there was only one thread in the old Environment implementation. We need to look at how to reduce this. As discussed in Section 3 fine-tuning of socket communication by converting `Serializable` objects to `Externalizable` objects could improve the performance. The new architecture has to be analyzed more thoroughly using profiler and checked to see if any new bottlenecks have been introduced to the changes made.

The new Environment was checked to see for memory leakage, which was significantly reduced. The heap memory of the new Environment was growing in a reduced manner when compared to the old Environment. Resetting socket connection after write operation could completely stop the memory leakage. Resetting of socket was not added to this implementation since it causes the application to slow down. We need to look at the appropriate intervals at which reset operation can be performed.

4 CONCLUSION

The analysis done on the Environment simulator helped in identifying the bottlenecks of the system. The system was not able to run for long due to memory leaks. Profiling helped in identifying the exact cause of the problem. Solving this issue made the Environment more stable.

Some of the clients connected to the Environment can now communicate with the Environment without synchronizing with other modules or processes that communicates with the Environment. For example, in the original configuration, the viewer clients were updated at the end of each time step. This caused robot clients to wait until the Environment updated the viewer clients. Now the viewer clients can be updated in a separate thread that does not block the threads serving robot client.

The Geometry Module was moved out of the environment and run as a separate process. Earlier the geometry module was part of the Environment and hence all collision detection functions were done within the Environment.

4.1 Future Enhancements

The Environment acts as a coordinator with many modules communicating with it. Instead of Environment doing the actual simulation work, it is offloaded to other processes like Geometry Module. Therefore, in the future, communication performance will be the primary concern for the Environment.

4.1 Communication Performance Improvement

Here I discuss three important modifications for the entire system that can help improve the communication performance of the system by decreasing the response time of Environment.

1. Convert all java `Serializable` objects to java `Externalizable` objects. In `Java`, Object serialization is the mechanism that allows you to read/write full-blown objects to byte streams. Implementing the user objects as `Java Serializable` interface allows you to serialize the objects by passing them to the `writeObject()` method of `ObjectOutputStream`. The `ObjectOutputStream` automates the process of writing the class metadata and instance fields to the stream. In other words, it does all the serialization work for you [2]. This causes the `writeObject()` function to be slow in streaming the data across the network. Instead of using `Serializable` objects, we could use `Externalizable` objects to be sent across the network. This could improve the performance of the Environment. A more detail discussion can be found in Section 2 of this document. This change can be done initially in Hardware Simulator and Environment, and then analyzed to see if it actually improves the performance.

2. Modify the Hardware Simulator and the Environment so that time step information is not sent between them in every time step. For this, we need to modify the protocol between Robot and Environment. Since the `receive()` function of the `ObjectInputStream` is a blocking function, we do not need to send time step from Environment to robot to synchronize all robots with Environment. Time step can be incremented as a local variable by these two modules.

3. We could also divide the Hardware Simulator into two separate threads so that receiving and sending is done separately. Whether or not this will actually improve the performance of the system is unknown.

4.2 Protocol Modification

Currently the Environment knows the details about robots by reading them from an XML file. However, this could become complex when there are certain specific properties attached to particular type of robot. For example, the Sonar sensors of Scout are at different locations than Pioneer. Since Environment Simulator has to serve different types of robots, this kind of information is not defined as constants within the Environment. Adding this information to an XML file will make the file more complex and error prone. Therefore, robots' properties could be sent from the Hardware Simulator to the Environment at the startup. To accomplish this, we must modify the protocol between the Hardware Simulator and the Environment.

The protocol between the Hardware Simulator and the Environment has two phases: initial and execution. In the initial phase, the Environment sends time step size to the Hardware Simulator after receiving the name of the robot from the Hardware Simulator. Details of the robot could be sent from the Hardware Simulator to the Environment in this initial phase.

REFERENCES

[1] Dr. Heinz M. Kabutz, http://www.javaspecialists.co.za/archive/Issue088.html

[2] Stuart Halloway, a Java specialist at DevelopMentor, http://java.sun.com/developer/TechTips/2000/tt0425.html

[3] Dr.Scott Deloach, KSU, http://www.cis.ksu.edu/~sdeloach/ai/projects/crsim.html

[4] Dudek, G., and Jenkin, M., A multi-level development environment for mobile robotics, Proc. Int. Conf. on Intelligent Autonomous Systems: IAS-3, 542-550, Pittsburgh, PA, 1993.

[5] Michael.O, Professional Mobile Robot Simulation, pp.39-42, international Journal of Advanced Robotic Systems, 2004

[6] Scott Harmon's MS report, Co-operative Robotics Simulator - Environment Simulator, May 2004, http://robosim.user.cis.ksu.edu/tiki-index.php .

[7] RoboSim Team, KSU, http://robosim.user.cis.ksu.edu/tiki-index.php

APPENDIX

User Manual

The RoboSim system contains one Environment process, one or more viewer processes like Viewer2D, CRS3DViewer, one or more Robot processes, one or more Remote Control processes, and one Geometry process. These processes can run on different machines that are connected by a common network. Java 1.4 or above is required for all the processes. To execute CRS3DViewer Java3D should be installed. RoboSim can be executed on any operating system that supports java.

Below is a step-by-step explanation on how to start the RoboSim system:

1 Download the \bin directory, \scripts and \TestLoadFiles directories from Central Versioning System, fingolfin.user.cis.ksu.edu.

2 Create a new Environment file in \TestLoadFiles\Environment folder (e.g. test.xml). Suppose you want to setup two robots, say robot1 and robot2, in the Environment, then you have to enter the details of these two robots in this file. A sample XML file will be already present in \TestLoadFiles\Environment directory.

All the commands from step 3 to step 7 should be executed from the \bin directory.

3 Start Environment by issuing the following command

\bin> java edu.ksu.cis.cooprobot.simulator.environment.Environment ..\TestLoadFiles\environment\test.xml

From step 4 to step 6, ***hostname*** is the machine in which the Environment is running. For example if the Environment is running in the same machine as the starting application, then ***hostname*** will be **localhost**.

4 Start a 2-D viewer (there is a 2 D viewer and a 3 D Viewer currently in the system).

\bin> java edu.ksu.cis.cooprobot.simulator.viewer.Viewer2D <hostname> 3000

- ***3000*** is the port number at which the environment is going to wait for viewer clients to connect.

5 Start geometry client.

\bin> java edu.ksu.cis.cooprobot.simulator.geometry.GeometryClient <hostname> 10000

- ***10000*** is the port number at which the environment is going to wait for Geometry Client to connect.

6 By now all the modules necessary to support the RoboSim is ready. Robots should only be started after these modules are started. The Environment will be waiting for all the robots mentioned in the XML file to connect, only then will it start processing requests from all the robots. There are two things to keep in mind while doing this. First, make sure that all the robots mentioned in the files are started as separate processes. Second, the names of the robots in the XML file should match to the names of the robots that start. For example, consider test.xml; it contains robot1 and robot2. Now, the script to start robots looks like the following:

\bin> java edu.ksu.cis.cooprobot.simulator.applications.maze.RemoteControlRobot robot1 <hostname> 8000

\bin> java edu.ksu.cis.cooprobot.simulator.applications.maze.RemoteControlRobot robot2 <hostname> 8000

- ***8000*** is the port number at which the environment is going to wait for robot clients to connect.

Here, RemoteControlRobot is the robot control code. This is the controller program we test using the RoboSim.

7 A Remote Control program is available that can be used by external user to control the robots. This remote control can be started as follows:

\bin> java edu.ksu.cis.cooprobot.simulator.applications.maze.RemoteControl ..\TestLoadFiles\environment\RobotInfo.xml

RobotInfo.xml contains the following information:

- Information of all the robots that are running in the RoboSim, i.e. the port number and the hostname of the machine where each robot is running.
- `RemoteControl` initially controls only one robot. If the user wants, he/she can switch to a different robot using the I/O device. These information is to be mentioned in the above XML file.

Programmer Manual

The RoboSim project can be downloaded from the CVS repository at `fingolfin.user.cis.ksu.edu`. You need access to the CVS directory to download it. Eclipse can be used as the development tool and to connect to CVS. There will be different packages present in the project. Current Environment uses two packages, the main class `Environment` is within `edu.ksu.cis.cooprobot.simulator.environment` and the client worker classes are within

edu.ksu.cis.cooprobot.simulator.environclientinterface. Below I briefly describe how to add new classes to the existing Environment.

1 Adding Robots

The EnvironmentObject class contains methods that act on any Environment Object, whether they are robots or otherwise. EnvironmentObjectRobot is an extension of EnvironmentObject class. The information for sensors are stored along with each robot. If a specific type of robots needs to be added to the Environment, create a new class that extends EnvironmentObject class.

2 Adding Sensors

Each sensor has some common information such as position and rotation relative to the robot. This information is stored in the RobotSensor class. RobotSensor class can be extended to handle specialized sensors(e.g. RobotSensorSonar, RobotSensorBump). RobotSensor is an abstract class with an incomplete method, generateSensorResponse(). The Environment calls this method whenever the robot requests a response from this particular sensor. This way the details of sensors are hidden from the Environment.

Therefore, to add a new Sensor you need to create a new class that extends from RobotSensor class and implement the generateSensorResponse() method.

3 Handling new Types of Clients

If new type of client needs to communicate with the Environment, create a new class that extends the ClientWorker class in the environclientinterface package. This class objects can be instantiated and started from ClientCommCoordinator class.

4 To disconnect Geometry Module

There is a variable GEOMETRY_USED in `GeometryClientWorker` class. Setting this variable to zero makes the Environment work without using the Geometry module.