2009


Duane Stanley Bolick Jr.
Wright State University

Follow this and additional works at: http://corescholar.libraries.wright.edu/etd_all
Part of the Computer Sciences Commons

Repository Citation

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu.
A SOFTWARE FRAMEWORK FOR THE DESIGN, TESTING
AND DEPLOYMENT OF CONTROL SYSTEMS FOR
AUTONOMOUS ROBOTICS

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

DUANE STANLEY BOLICK, JR.
B.A., Case Western Reserve University, 1999

2009
Wright State University

___________________________
John C. Gallagher, Ph.D.
Thesis Director

___________________________
Thomas A. Sudkamp, Ph.D.

Committee on
Final Examination

___________________________
John C. Gallagher, Ph.D.

___________________________
Krishnaprasad Thirunarayan, Ph.D.

___________________________
Mateen M. Rizki, PhD.

___________________________
Joseph F. Thomas, Jr., Ph.D.
Dean, School of Graduate Studies
ABSTRACT

Bolick, Duane Stanley, Jr. M.S., Department of Computer Science and Engineering, Wright State University, 2009.

Simulation and control of robotic agents are common enough tasks among computer science and engineering researchers that there exists a large variety of software applications, toolkits, and programming frameworks designed to facilitate such research. The intent of this project is to provide an autonomous robotics simulation and control framework with sufficiently useful tools and reference implementations so as to be immediately useful to basic users, as well as easily understood and extended by software developers and robotics researchers.

As such, this document describes the implementation of a general networking toolkit, an autonomous robotics extension framework, and an autonomous robotics simulator and control application that can be used in an online collaborative manner, and that is accessible to users with a wide range of technical skills and experience.
TABLE OF CONTENTS

1. INTRODUCTION ...............................................................................................................1
   1.1 Project Overview ...........................................................................................................1
   1.2 Terminology ..................................................................................................................2

2. REQUIREMENTS .............................................................................................................6

3. PREVIOUS WORK ..........................................................................................................9
   3.1 Introduction ...................................................................................................................9
   3.2 Simulation Software .....................................................................................................9
   3.3 Robot Interface Software ..........................................................................................12
   3.4 Communication and Remote Viewing Software .......................................................13
   3.5 Summary ......................................................................................................................14

4. ANALYSIS AND DESIGN ..............................................................................................15
   4.1 Analysis Summary .......................................................................................................15
      4.1.1 General Application Specifications .................................................................15
      4.1.2 Agent Simulation and Interface Specifications .................................................16
      4.1.3 Collaborative Feature Specifications ...............................................................17
      4.1.4 Security Feature Specifications .........................................................................17
   4.2 Design Overview .........................................................................................................18

5. THE CARL NETWORKING LIBRARY ............................................................................21
   5.1 Introduction ..................................................................................................................21
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Client-Server-Context Architecture</td>
<td>22</td>
</tr>
<tr>
<td>5.3</td>
<td>Connection Model</td>
<td>26</td>
</tr>
<tr>
<td>5.4</td>
<td>Inter-Component Communications</td>
<td>29</td>
</tr>
<tr>
<td>5.5</td>
<td>User Authentication and Access Control</td>
<td>32</td>
</tr>
<tr>
<td>5.6</td>
<td>Context Visualization</td>
<td>33</td>
</tr>
<tr>
<td>5.7</td>
<td>Local Communications</td>
<td>34</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>36</td>
</tr>
<tr>
<td>6.2</td>
<td>Agents</td>
<td>38</td>
</tr>
<tr>
<td>6.3</td>
<td>Agent Controllers</td>
<td>40</td>
</tr>
<tr>
<td>6.4</td>
<td>Agent World Framework</td>
<td>41</td>
</tr>
<tr>
<td>6.5</td>
<td>Agent Modules</td>
<td>43</td>
</tr>
<tr>
<td>6.6</td>
<td>Simulation Modules</td>
<td>45</td>
</tr>
<tr>
<td>6.7</td>
<td>Implementation of a 2D Simulated Environment</td>
<td>46</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Overview</td>
<td>46</td>
</tr>
<tr>
<td>6.7.2</td>
<td>The World Model - AgentWorldSim2D</td>
<td>46</td>
</tr>
<tr>
<td>6.7.3</td>
<td>The Visualization - ViewPanelSim2D</td>
<td>49</td>
</tr>
<tr>
<td>6.8</td>
<td>Implementation of the Khepera Agent Module</td>
<td>51</td>
</tr>
<tr>
<td>6.8.1</td>
<td>Control Interfaces and Controllers</td>
<td>51</td>
</tr>
<tr>
<td>6.8.2</td>
<td>Real-World Implementation</td>
<td>51</td>
</tr>
<tr>
<td>6.8.3</td>
<td>Simulated World Implementation</td>
<td>54</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (CONTINUED)

7. THE WART APPLICATION ........................................................................................................58

7.1 Introduction ..........................................................................................................................58

7.2 Application Design and Lifecycle .....................................................................................59

7.3 User Interface Design .........................................................................................................59

7.3.1 Address Bar ..................................................................................................................60

7.3.2 Tabbed Browsing Pane ...............................................................................................61

7.3.3 Additional Interface Features ......................................................................................64

7.4 Contexts ...............................................................................................................................64

7.4.1 Chat Room Context .....................................................................................................65

7.4.2 Simulated Agent Context ............................................................................................67

7.4.3 Real-World Agent Context ..........................................................................................70

8. TESTING AND VALIDATION ..............................................................................................72

8.1 Introduction ..........................................................................................................................72

8.2 Testing ................................................................................................................................72

8.2.1 Unit Testing ...................................................................................................................73

8.2.2 Feature Testing .............................................................................................................75

8.2.3 Stress and Performance Testing ..................................................................................77

8.3 Validation .............................................................................................................................82

8.3.1 General Application Specifications .............................................................................82

8.3.2 Agent Simulation and Interface Specifications .........................................................83

8.3.3 Collaborative Feature Specifications .........................................................................83
8.3.4 Security Feature Specifications ..............................................................84

9. CONCLUSIONS ..........................................................................................86

9.1 Summary ....................................................................................................86

9.2 Future Work ...............................................................................................87

9.2.1 Remote Server Administration .........................................................87

9.2.2 Address Bar Improvements ...............................................................88

9.2.3 Modular Agent Builder .........................................................................89

9.2.4 Runtime Module Loading ......................................................................92

REFERENCES .................................................................................................93
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Example of UML diagram component</td>
<td>4</td>
</tr>
<tr>
<td>2. Example of UML diagram associations</td>
<td>5</td>
</tr>
<tr>
<td>3. The KSim Interface</td>
<td>10</td>
</tr>
<tr>
<td>4. The robot arena</td>
<td>12</td>
</tr>
<tr>
<td>5. Server class diagram</td>
<td>23</td>
</tr>
<tr>
<td>6. Client class diagram</td>
<td>24</td>
</tr>
<tr>
<td>7. Context class diagram</td>
<td>25</td>
</tr>
<tr>
<td>8. The connection process</td>
<td>29</td>
</tr>
<tr>
<td>9. Message and MessageRecipient implementation details</td>
<td>31</td>
</tr>
<tr>
<td>10. View class diagram</td>
<td>33</td>
</tr>
<tr>
<td>11. The View creation process</td>
<td>34</td>
</tr>
<tr>
<td>12. WART library namespaces</td>
<td>37</td>
</tr>
<tr>
<td>13. Fictional Agent FooBot class hierarchy</td>
<td>38</td>
</tr>
<tr>
<td>14. Providing multiple control interfaces</td>
<td>39</td>
</tr>
<tr>
<td>15. AbstractAgentController's run implementation</td>
<td>41</td>
</tr>
<tr>
<td>16. The Context view update process</td>
<td>43</td>
</tr>
<tr>
<td>17. Agent module contents</td>
<td>44</td>
</tr>
<tr>
<td>18. Collision resolution rules</td>
<td>48</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES (CONTINUED)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19. Relationship between solid object model and graphical object model</td>
<td>50</td>
</tr>
<tr>
<td>20. Khepera Serial Protocol response processing</td>
<td>53</td>
</tr>
<tr>
<td>21. Top down view of Khepera robot with gripper turret</td>
<td>55</td>
</tr>
<tr>
<td>22. Gripper collision exception area</td>
<td>55</td>
</tr>
<tr>
<td>23. Address bar with expanded dropdown box</td>
<td>60</td>
</tr>
<tr>
<td>24. Nesting of tabbed pane components</td>
<td>62</td>
</tr>
<tr>
<td>25. Chat room user interface</td>
<td>67</td>
</tr>
<tr>
<td>26. Simulated agent context user interface</td>
<td>69</td>
</tr>
<tr>
<td>27. Simulated world editor</td>
<td>70</td>
</tr>
<tr>
<td>28. Reference-type input parameter partitions and behavior</td>
<td>74</td>
</tr>
<tr>
<td>29. Feature testing use cases</td>
<td>76</td>
</tr>
<tr>
<td>30. Heap memory usage</td>
<td>79</td>
</tr>
<tr>
<td>31. Memory profiler view</td>
<td>80</td>
</tr>
<tr>
<td>32. Frames per second per number of Khepera</td>
<td>81</td>
</tr>
<tr>
<td>33. URL parsing example</td>
<td>89</td>
</tr>
<tr>
<td>34. Modular Khepera and its components</td>
<td>90</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to thank my mother, without whom none of this would have been possible. Also, I would like to thank my research advisor, Dr. John Gallagher, whose guidance and input was crucial to the success of this project. Finally, I would like to acknowledge my fellow lab members and colleagues, Sanjay Boddhu and Dr. Greg Kramer for their encouragement and support.

This work was supported by the National Science Foundation under grants DUE 0341150 and DUE 0341263. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.
1. INTRODUCTION

The purpose of this document is to describe the design and implementation of a framework for the simulation and control of autonomous robotic agents. This project was motivated by the software requirements for an experimental course in teaching the design and development of control systems for mobile autonomous robots.

1.1 Project Overview

The mobile autonomous robotics course arose from a joint research project between the computer science departments of Case Western Reserve University (CWRU) in Cleveland, OH and Wright State University (WSU). This project began in June 2000 and studied the feasibility and efficacy of teaching laboratory-based classes in a distance learning format [1]. The experimental portion of this study involved creating an online course whose goal was to introduce students to design and implementation of control systems for mobile autonomous robots. This course was taught simultaneously at WSU and CWRU, with the course designation of CS 499: WWW Mobile Autonomous Robotics [2] at WSU, and EECS 375: Autonomous Robotics [3] at CWRU.

The selection and development of the necessary software tools for the course is described in the thesis document by Steven J. Perretta entitled Java Tools for the Development of Autonomous Robot Controllers [4]. In summary, several existing robot interface and simulation tools were evaluated with regards to their suitability for use in
this course, but they were determined to be either too specifically and rigidly designed to effectively modify for the purposes of the course, or else they were robotic simulation frameworks designed to be extensible and general, which were overly-broad in their scope and required an unnecessary amount of programming overhead to implement an acceptable solution. As a result, the course utilized a number of separate technologies to provide the necessary tools to instructors and students, including various third-party tools including chat servers, web-cam servers, and remote connection and file transfer applications.

Two key applications were developed and maintained at WSU specifically for use in this class by Steven J. Perretta and this author. The first was a Java-based graphical simulator allowing students to test control algorithms for a Khepera robot platform, and the second was an application that applied control algorithms to an actual Khepera robot. Details of the development and use of both of these applications can be found in the thesis document mentioned above. While these applications fulfilled the minimum requirements for the needs of the course, they were deficient in several key areas of the long-term goals for the software.

This document recapitulates the original technology requirements for the course, reviews the previously implemented solutions, and then describes in detail the development of a software framework and application that more completely fulfills the requirements, and provides significant extensibility and flexibility.

1.2 Terminology

This document outlines a medium-scale software development project developed in
Java, and therefore utilizes terminology used in the object-oriented programming (OOP) paradigm. With regards to common OOP concepts, the terms: class, interface, implements, and extends may be assumed to have the standard meaning. Of note, the term implements/implementation as in, "Class Foo implements/is an implementation of interface Bar," should be taken to mean that a particular class fulfills the interface contract at some point in its inheritance hierarchy, and not that the concrete implementation of the interface necessarily occurs in the named class.

In addition, many of the figures in this document utilize Unified Modeling Language (UML) diagram components. Class or interface components are represented as rectangular shapes divided internally into two compartments, with the class name, or the class and instance variable name, in the upper compartment. The lower compartment may be empty, or it may contain descriptive text regarding that class or interface. Interfaces are identified by the word, "Interface", contained in double angle brackets above the class component. A solid line with an arrowhead connecting two classes or interfaces implies the relationship of "extends," while a dotted line with an arrowhead connecting a class and an interface implies the relationship of "implements."
Two class or interface components joined by a solid line implies an unspecified association. The cardinality of the association is indicated by numbers, implying a specific numerical relationship such as one to one, or the letter ‘n,’ implying a many to one (or many to many) relationship.
Figure 2: Example of UML diagram associations.

<table>
<thead>
<tr>
<th>Client</th>
<th>Server Handle Manager</th>
<th>Map&lt;String, Callback&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>views</td>
<td></td>
<td>clientCallbacks</td>
</tr>
<tr>
<td>Map&lt;String, View&gt;</td>
<td>ServerHandleManager</td>
<td></td>
</tr>
<tr>
<td>views</td>
<td>Map&lt;String, Callback&gt;</td>
<td></td>
</tr>
<tr>
<td>Maintains a collection of Context Views.</td>
<td>Maintains a collection of RemoteHandles, each representing a remote server.</td>
<td>Maintains a collection of Callback objects, one for each ClientHandle that exists on a remote server.</td>
</tr>
<tr>
<td>Provides ability to add and delete Views.</td>
<td>Provides ability to add or remove servers.</td>
<td></td>
</tr>
<tr>
<td>Provides access to individual Views by Context ID.</td>
<td>Provides access to individual servers by ID.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>View</th>
<th>ServerHandle</th>
<th>Callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td></td>
<td>A Callback object exists to handle method invocations on a RemoteHandle.</td>
</tr>
<tr>
<td>1:1 mapping of a View to a Context.</td>
<td>Defines accessor and mutator methods common to all Server handles.</td>
<td>Note that in the case of RMI, the RemoteHandle and the Callback are the same object.</td>
</tr>
<tr>
<td>View implementations are responsible for providing a GUI interface (ViewPanel) for a given Context implementation.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. REQUIREMENTS

This problem consists of two closely-related domains: First, the specific requirements for the WWW Mobile Autonomous Robotics course must be fulfilled. The course is intended to provide an introduction to the development and implementation of control algorithms for the Khepera mobile robotics platform. Throughout the course, students are first introduced to the Khepera robotics platform, and instructed on the fundamentals of writing control algorithms. They are subsequently presented with a series of tasks for the robotic agent to accomplish, of increasing complexity throughout the remainder of the course period. Student and instructors interact online, and should be able to demonstrate controller design and troubleshoot students’ algorithms. Finally, students must be able to deploy their control algorithms to a real-world robotic agent, and view the results of that algorithm running on that agent.

Java Tools for the Development of Autonomous Robotic Controllers provides a list of technologies that the course requires. Specific hardware requirements have been omitted from this list:

1) Software that enables students to run control algorithms on a Khepera mobile robot platform remotely.

2) Simulation software that enables students to develop and test Khepera control
algorithms in a simulated environment.

3) Communication software for student/instructor interaction.

4) Software providing the capability to remotely view the Khepera.

The second domain consists of the goals of the overarching research project, which are to study and ultimately facilitate the execution of distance-based education involving autonomous robotics. The goals of the research project are broader in scope, and as a result there are a number of additional requirements that this domain defines in addition to those defined above:

1) Extensible software that enables development of simulation of multiple types of simulated robotic agents and potentially other devices.

2) Extensible software that enables development of interactivity with multiple types of real-world robotic agents/devices.

3) Ability to modify user interface to address accessibility concerns.

4) Minimize barrier to entry in terms of ease of use, to enable deployment to a target audience with a widely varied range of technical expertise. This requirement pertains to the potential use of this software in secondary education settings.
5) Provide facilities for future extension and improvement.

6) Maximize portability to different operating systems.

In the next chapter we discuss the previously developed software solutions, and identify their shortcomings with respect to the requirements shown here.
3. PREVIOUS WORK

3.1 Introduction

The four major technological requirements of the WWW Mobile Autonomous Robotics course are robotic agent simulation of a Khepera platform, interface with a real-world Khepera, online communications, and remote viewing of the operation of a real-world robotic agent. The software developed by Steven Perretta in conjunction with selected third-party applications successfully fulfills these requirements, and provides educators with the necessary tools to execute the course [5]. However, due to its necessarily broader scope, several requirements of the research project still remain to be fulfilled [6].

This chapter examines each of the four major course requirements listed above, describes the previously implemented solution to each, and then summarizes the shortcomings of the existing software with respect to the overarching research project requirements. For details of the features and implementation of the software described below, please refer to the document Java Tools for the Development of Autonomous Robot Controllers by Steven J. Perretta.

3.2 Simulation Software

The WSU Khepera Simulator (KSim) was developed as a graphical simulator of a single Khepera robot, operating in a fixed size, flat environment. It allows the placement
of several types of objects, with which the Khepera can interact, and also obstacles, which impede the Khepera's movement. The simulated Khepera presents a Java-based programming interface that is identical to that of the Khepera Serial Interface Program, so that users can control a simulated or real Khepera with the same compiled class file.

Figure 3: The KSim Interface

While it fulfills the requirements for executing the lab course, the simulation engine used in KSim has several significant shortcomings that prevent it from fulfilling the project requirements. These shortcomings were a direct result of the requirement for backwards compatibility of control algorithms, which required that KSim maintain several internal data structures and other aspects of the design of the previous versions. These older version design aspects were the cause of the aforementioned shortcomings,
and are discussed in this section.

First, the simulation engine does not use a geometrically-based model for collisions, sensing and other interactions between objects in the simulated environment. Instead, it defines the environment as a fixed-size area in pixels and uses a two-dimensional array where each cell represents a pixel in the simulated environment to perform collision detection, and distance calculation for sensing purposes. This design decision ties the simulation engine to a fixed size and scale, and also disassociates the simulated world from real-world distance measurements. Because of the fixed size, shape, and scale of the simulated environment, KSim lacks the ability to simulate diverse and varied environments, which reduces its generality as a robotics simulation tool.

The simulation also uses an update timing model that is not directly associated with the behavior of the Khepera over actual time. The engine thread updates the position of the objects in the world, calculates the Khepera's sensor returns based on the new positions, and then updates the graphical view. These updates are based on distances in pixels not tied to an actual time scale. The simulator relies on a set of constant factors built in to its calculations, which were originally determined empirically from repeated observation of the real-world Khepera to reliably reproduce behavior consistent with the actual Khepera. As a result, the timing of updates of the simulated environment is inextricably bound to the operation of a Khepera-type robotic agent, which precludes its use with other robotic agent types.

Finally, KSim does not permit use of multiple agents, the inclusion of additional object types other than the limited set that are included with the simulator, or the sharing of a simulated environment among multiple remote users.
3.3 Robot Interface Software

The WSU Khepera Serial Interface Program (KSIP) allows users to control the actual Khepera using control algorithms developed in the simulator, and is implemented as a Linux-only, single-user, console-based application. KSIP communicates with the Khepera via a serial port using a third-party implementation of the javax.comm library, RXTX [7]. The usage of KSIP is as follows: Users transfer their Khepera control algorithm Java class files via a third-party file transfer application or FTP utility, and then connect to the host remotely using a remote connection utility such as a Secure Shell or Telnet client. Once connected, the user executes the KSIP application to run their control algorithm on the Khepera.

*Figure 4: The robot arena.*
While KSIP, in conjunction with other third-party applications and utilities fulfills the requirement of remote access to a Khepera robotics platform, it lacks several key qualities that prevent it from fulfilling the intent of the project requirements. Its usage, as described above, is not amenable to an online multi-user collaborative environment. Additionally, users are required to be proficient in the use of several other technologies such as FTP clients and remote connection utilities, as well as the use of a Linux-based shell command-line interface. While not an insurmountable obstacle for an audience of computer science undergraduate students, it presents a serious impediment for its use in secondary education settings where both instructors and students alike might not possess the technical skills and experience to use KSIP.

Finally, KSIP does not possess the capabilities to communicate with multiple agents, it uses the Khepera Serial Protocol which restricts its ability to communicate with multiple robotic agent types, and its console-based user interface is not easily translated into accessible content.

3.4 Communication and Remote Viewing Software

Communication between students and instructors is implemented using third-party chat applications. This is a suitable solution for providing online communication facilities, especially since there are a wide range of freely-available, mature chat applications. However, these chat applications are not integrated with any of the previously described software, which diminishes their utility as a collaborative teaching tool.
Remote viewing of the Khepera robot is implemented using a third-party streaming video server and client applications, which allows students to observe the Khepera in operation over a web cam. As with the use of third-party software for communication, this solution fulfills the requirements for use in the course, but lacks somewhat in providing an integrated, collaborative teaching environment.

Additionally, both of these solutions present additional barriers to entry for instructors lacking the technical skills necessary to administer chat and web cam servers.

3.4 Summary

The previously implemented solution to the requirements of the WWW Mobile Autonomous Robotics course was a significant first step towards providing a set of tools for educators teaching an online collaborative laboratory-based course, and it provided sufficient functionality to successfully teach the course for a number of iterations. However, there remain several unfulfilled requirements of the overarching research project that require additional development.

The following chapter presents an analysis of all requirements, distillation of the key specifications, and an overview of the design of a software product that will fulfill those specifications.
4. ANALYSIS AND DESIGN

4.1 Analysis Summary

This section provides specific design specifications generated from analysis of the requirements of the research project. The items in this section are necessarily more specific and contain greater amounts of detail than the requirements, because their intent is to provide a significant degree of guidance in the design of the software. Below are specifications separated into four sections: General application specifications, simulation and real-world agent interface specifications, collaborative feature specifications, and security specifications.

4.1.1 General Application Specifications

This section contains a list of general specifications that do not necessarily apply to a particular feature of the application, or that apply to multiple features. The list is as follows:

1) Provide a graphical user interface that exposes the major functionality of the application in the top-level interface to facilitate ease of use for novice users.

2) Ensure the user interface can be modified or replaced to address accessibility concerns.
3) For network-based features, provide the ability to support multiple network communication protocols, and also provide the ability to add additional protocols in a modular fashion.

4) For features that involve communication between a host computer and an agent or other device, provide the ability to support multiple local communication protocols, and the ability to add additional protocols in a modular fashion.

4.1.2 Agent Simulation and Interface Specifications

This section contains specifications related to the simulation of robotic agents, and interaction with their real-world counterparts.

1) Provide a graphical simulation feature, where users can simulate the deployment of their control algorithms on agents, and observe the results.

2) Provide a real-world agent interface feature, where users can deploy control algorithms on real-world agents.

3) Ensure that both the simulator and interface features support the simultaneous employment of multiple agents, and that they also support the ability to add new agent types.
4) Enforce the practice that agent types provide a common programming interface for both the simulated and real-world components such that a control algorithm created for an agent type can be used both in simulation, and to control the real-world agent.

4.1.3 Collaborative Feature Specifications

This section contains specifications related to features of the application that facilitate online collaboration, and increase the utility of it as an online teaching tool.

1) Provide a chat room feature, where multiple users can communicate via text-based messages.

2) With respect to the agent simulator and interface features, provide the ability for a remote user to deploy a control algorithm in either context.

3) Provide the ability for any user to host multiple instances of the chat room environment, simulation, and real-world interface environment.

4) Provide the ability to add newly-implemented collaboration features in a modular fashion.

4.1.4 Security Specifications

This section outlines specifications intended to provide basic security features to the application, protect host computer resources, and robotic agent hardware.
1) Provide and enforce remote user authentication, with the ability to extend the application to support multiple types of user credential types.

2) Provide the ability to apply function-based access control to features of the application.

3) Provide the ability of a user hosting other remote users to remove a user from a particular feature, or from their hosted application altogether. Additionally, ensure that a user hosting real-world robotic agents can restrict remote control of those agents from the top-level interface.

4.2 Design Overview

This section provides an overview of the design of this software project based on the above specifications. A more detailed description of the design and implementation will be discussed in the following chapters.

At a high level, this application utilizes the Model-View-Controller (MVC) design pattern [8]. MVC is an object-oriented architectural design pattern that is intended to encapsulate the core business logic of an application (the Model), and ensure its separation from the user interface (the View) through use of an intermediate object (the Controller) which acts as a "translator" between user interface and data model. In this case, the Model includes basic client-server networking functions, the agent simulation engine, real-world agent control, and the common agent control system interface. The
application View and Controller utilize the Java Swing Application Framework [9],
which provides much of the basic functionality of a graphical user interface. The intent
behind the use of this design pattern as an architectural guideline is to fulfill the
specification of easy modification or replacement of the user interface to address
accessibility concerns. Additionally, this increases the ability of future development and
extension so that the application can be modified from a standard desktop application to
an entirely different interface type, such as a web-based interface or even to a mobile
computing device.

The design of this application consists of three major components. The first two are
Java class libraries; one implements general networking functionality and the second
implements an autonomous robotics extension framework. The third component is the
application itself. Below is a brief description of each.

The first component is the CARL Networking Library, a general networking library
that provides a basic client-server framework upon which applications can be built. This
library provides a generalized client-server architecture, user authentication functionality,
an access-control list implementation, an OSI Application-Layer connection model, a
generalized message passing architecture, and an interface/abstract class hierarchy for
constructing visualizations of applications developed using this library. This library
provides the functionality that fulfills the networking and security specifications, as well
as the class and interface hierarchy that supports the specification requiring the ability to
add newly implemented collaboration features.

The second component is the WSU Autonomous Robotics Toolkit (WART) library.
This library provides a generalized extensible framework for autonomous robotics
simulation, and interaction with real-world autonomous agents using a common interface. This common interface facilitates development and validation of control systems in simulation, with the ability to seamlessly deploy those control systems on actual robotic agents. This library is focused mainly on implementing the robotic agent related specifications.

The third component is the application itself, WARTApp. This application utilizes the features of the two previously described class libraries to implement a desktop application with a graphical user interface. Within the application package are implementations of the core features listed in the specifications: agent simulator, agent interface, and chat room.

The following three chapters describe in detail the design of these three components.
5. THE CARL NETWORKING LIBRARY

5.1 Introduction

The CARL networking library is intended to provide a standalone class library which the application uses to implement functionality to fulfill the networking, local communications, and security specifications of this project. Developing these functions as a separate class library enforces separation of application business logic and interface, as well as provides a reusable, extensible set of classes. This chapter outlines the major features of the library, provides some details of the implementation in the form of class and sequence diagrams, and highlights the key abstractions that provide the extensibility of this library.

Of note, the class diagrams are intended to provide both a high-level understanding of certain key classes in this library, as well as a reference of their important member classes. However, rather than reproduce a given class diagram several times throughout the text as its various member classes are introduced, it was decided to include a particular class diagram once, at the point that the class is discussed. As a result of this decision, a figure may contain classes and concepts that may not have been introduced by the point in the text at which the figure is located.

Finally, the intent of this chapter is not to enumerate every class and method in the CARL networking library; rather, the intent is to provide a general description of the major features and functionality this library provides, and to provide a conceptual
understanding of the library, and how it is intended to be used. As such, this chapter is organized into subsections, each based on a core design concept of the library.

5.2 Client-Server-Context Architecture

The core purpose of the CARL Networking Library is to provide a generalized Client-Server framework on which applications can be built. The first and most significant abstraction designed into the CARL networking library is separation of the basic minimal client-server functions (creating and maintaining connections, and user authentication) from application-specific functionality (e.g., chat room, shared white board, robotic agent simulation).

To provide this encapsulation the CARL networking library introduces a third core component, the Context, in addition to the standard Client and Server components of a typical client-server architecture.

As a result of this design, the Server class provides basic user authentication and connection functionality, but with respect to actual feature implementation, contains a collection of Contexts, and provides access to them through its interface.
Figure 5: Server class diagram.
Likewise, the Client class does not include any application-specific functionality. It simply maintains connections, and provides communication between itself and the Server.

![Client class diagram](image)

**Figure 6: Client class diagram.**

Conceptually, a Context represents a specific function, made available in a remote fashion via the client-server architecture. Examples of such a function that are included in
this particular application are chat rooms, shared robotic agent simulations, and the ability to control a real robotic agent over a network. In this model, the Client and Server components provide a minimal set of functionality: Connection establishment and management, user authentication, and access to Contexts. It is the Contexts themselves that provide the actual functionality of the application.

In addition, a Context implementation also defines feature-specific behavior.

Figure 7: Context class diagram.
The Context abstraction allows for the addition or removal of various Contexts during runtime; in effect, a Server can make functional units available (or unavailable) while running, as the server administrator chooses. In addition, entirely new functionality may be created (for example, a shared white board Context, or a file transfer Context) with no modification to the existing CARL networking library.

5.3 Connection Model

Rather than constrain client-server connection to a specific implementation of a particular communication protocol (i.e., TCP, RMI), the CARL networking library uses a connection model that is implemented in what can conceptually be considered the Application Layer in the OSI Reference Model. In order to provide this abstraction, all communication between key components is based on the concept of handles and callbacks. At a high level, a handle represents a one-way communication facility between two core components. For example, a Client can communicate to a Server through use of a Server handle and that Server communicates with a Client through use of a Client handle. A callback provides the facilities for handling incoming communications from a component's handle. For example, when a Context communicates with a Client using that Client's handle the underlying protocol transmits the communication to the Client's corresponding callback, which processes and acts on the communication. There is a one to one correspondence between handle and callback.

In the CARL networking library, handles and callbacks are specifically represented by the RemoteHandle and Callback interfaces. The RemoteHandle interface defines the minimal interface for any implementation of a handle, while its descendant interfaces,
ClientHandle, ContextHandle, and ServerHandle, declare additional behaviors as required by each specific core component. The actual lower-level communication protocol is specified in the classes that implement these handle interfaces. For example, there may be multiple implementations of the ClientHandle interface using different protocols such as an RMIClientHandle, a TCPClientHandle, and an RS232ClientHandle. The use of the handle interface type in the core components, rather than the concrete class types allow the commingling of handle implementations that use different protocols.

It should be noted that all communications are handle based, including those of the local user. These local "connections" are implemented as LocalClientHandle, LocalServerHandle, and LocalContextHandle. While the local user could interface directly with the component (i.e., Client, Server, and Context) objects directly, doing so would violate encapsulation of those objects. Additionally, the ability to remotely administer Server and Context functions remotely has been identified as a future development. As such, all interactions, even those conducted locally, are handle-based.

Each concrete handle implementation must implement a corresponding callback, specific for the protocol used for the handle. Of note, the Callback interface is an empty, “tagging” interface that declares no methods. The reason for this is because the implementation specifics of any given Callback are entirely dependent on the protocol. For example when using TCP, a corresponding Callback object must be able to receive and process incoming TCP segments, and determine the appropriate effect on the handle’s component. When using RMI, however, the Callback object is the handle itself, since an RMI Handle is implemented as a Java RMI remote object. For that reason, it is not possible to define any commonalities between different Callback implementations.
Now, given that different protocols define the concept of “connection” differently (or not at all, in the case of UDP), we abstract the underlying protocol-specific connection schemes by defining a generalized ConnectionListener interface. As with the various handle implementations, a concrete ConnectionListener class specifies the actual transport layer protocol. A particular accepting component may, therefore, have multiple ConnectionListener implementations active and “listening” for incoming connections at the same time.

At the handle level, a connection is defined as the state where each component has a reference to a handle corresponding to the other component, and each component has constructed their corresponding callback. A handle-based “connection,” will necessarily consist of the reciprocal handles using the same underlying protocol. The general model for connection establishment between two core components is as follows:

1) Component A, the initiating component, provides Component B, the accepting component, with all necessary information so that Component B can determine if the connection should be established, such as login credentials or permissions. Component A also provides B with the information necessary to create a handle to Component A, such as a port number and network address, or a Java RMI remote object. In addition, the initiating component constructs its callback at this stage. This, in total can be thought of as a connection request.

2) If Component B accepts the connection request, it creates a handle to Component A using the provided information. The connection is now half completed.
3) Component B then transmits the information necessary for construction of a handle to itself, to Component A, and creates the corresponding callback for that handle.

4) Upon receipt, Component A constructs a handle to Component B. The connection is now established.

Figure 8: The connection process.

If any step of the handle-based connection process fails due to an error or failure in the underlying protocol layer, the connection attempt is aborted, and failure is reported to both the initiating and accepting component.

5.4 Inter-Component Communications

Communication between components via handles is limited to the use of the interfaces presented by each component's respective RemoteHandle interface. The handle interfaces are intended to provide the ability to connect, disconnect, and to perform basic
queries, but nothing else. More complicated interactions are performed through the use of
the messaging system, which involves the MessageRecipient and Message interfaces.

The three main message implementations are ClientMessage, ServerMessage and
ContextMessage. Each of these implementations contains data members that allow a
recipient of such a message to determine its source. The specific function of a Message
implementation is contained in its payload, which is a data member of type Object. Use
of type Object for a Message payload is intended to provide maximal flexibility and
future extensibility.

The MessageRecipient interface declares one method, handleIncomingMessage,
which receives a single parameter of Message type, and returns void. It throws several
exceptions, one of which is the MessageTypeException. This exception type is thrown
when the MessageRecipient implementer receives a Message implementation, or a
payload type, that it is not intended to handle. A typical implementation of
handleIncomingMessage will first determine the top-level Message type, e.g.
ContextMessage versus ServerMessage. Once the Message type has been determined, the
lowest-level implementing class of the Message payload is determined, and the method
executes the appropriate code path.
The main intent of this design is to relegate the specifics of a Context's functionality to the Message passing system and, in effect, keep all the core communication “business logic” of a particular Context implementation in one location (the `handleIncomingMessage` method).
5.5 User Authentication and Access Control

The CARL networking library also provides user authentication and access control functionality through implementations of core Java library security interfaces. The primary user authentication interface, java.security.Principal, which represents an individual user, is extended in this library by the UserCredentials interface. A UserCredentials implementation represents a single user, who is identified uniquely by username. In order to provide support for multiple types of authentication credentials, a user is authenticated based on the criteria defined in concrete implementations of UserCredentials. This library provides a default implementation, DefaultUserCredentials, which utilizes an MD5 hashed password for authentication. In addition to allowing varied types of authentication, UserCredentials implementations can include more advanced authentication such as the use of one-time passwords, enforce minimum password length and complexity, and require periodic password changes.

The CARL networking library also provides default implementations for fundamental Java access control interfaces, which include java.security.acl.Permission, java.security.acl.Acl, java.security.acl.AclEntry, java.security.Group, and java.security.Owner. The default implementations in this library are conformant to the Java API with no additional functionality, so detailed and complete information about their usage can be found in the Java API [10]. Support for permission validation is built in to the default Server implementation, and also the abstract Context super-class, AbstractContext through the inclusion of an instance of a java.security.acl.Acl implementation.
5.6 Context Visualization

While the Client and Server components are written without any consideration to user interface, there are several classes and interfaces in the CARL networking library that exist to facilitate encapsulation of graphical interface components used to visualize Contexts. The top-level interface that provides this is the View interface. It should be noted that the class members pictured in Figure 10 are not, obviously, defined in the View interface. Their presence is implied through the definition of accessor methods that obtain references to objects of those types.

![View class diagram](image)

*Figure 10: View class diagram.*

As shown above Client-Server and Client-Context connections are represented by pairs of handles and callbacks. However, to provide support for visualization of Contexts, the Client side ContextHandle is contained within a View object.
This composition of handle and graphic interface component is provided to streamline the process of ensuring all Clients that have joined a particular remote Context have an updated graphical representation of that Context, and is discussed in detail in Chapter 7: The Wart Application.

5.7 Local Communications

Finally, the CARL networking library provides tools for communicating to a device that is connected to a host machine. Various communication protocols such as serial port or USB can be encapsulated as a Communicator type. The Communicator interface defines an event-driven I/O model, which requires that classes using a Communicator implement the CommunicatorEventListener interface, and register with the Communicator as a listener. Writing is performed by invoking the interface's write method, and any incoming communications from the device are sent to registered listeners. All data sent to and from a Communicator are in the form of Java strings, and
specific implementations are expected to handle conversion between that format and whatever native data format (i.e., byte array, ASCII, etc.) the device requires.
6. THE WART LIBRARY

6.1 Introduction

The WSU Autonomous Robotics Toolkit (WART) library is a standalone class library that provides both basic tools for robotic agent simulation and interface, as well as a modular extension framework for development of new robotic agent and simulation types. This library is slightly different than the CARL networking library described above because, in addition to implementing a set of software tools, it also establishes a namespace and interface hierarchy that are intended to be used as an extension framework. As a result, its namespace is organized with the intent of future modular addition of new agent types and new simulation engines, as shown in Figure 10.
Note that in Figure 12 the .mindstorm and .sim3D packages have not been implemented, and are only included to illustrate multiple modules in each module namespace.

The remainder of this chapter outlines the design, implementation, and also usage of the WART library. First, the major interface hierarchies, Agent, AgentController, and AgentWorld, which establish the basis of the framework, are discussed. Then, the creation of new Agent and simulation modules are discussed in general, followed by a description of a concrete implementation of each, the Khepera Agent module, and the Sim2D simulator module.
6.2 Agents

The top level interface that represents a robotic agent is the Agent interface, which is intended to be extended and ultimately implemented by all robotic agent types. This interface is used by the utility classes in this library for such things as maintaining collections of Agent implementations, and deployment of control algorithms. The top level control interfaces for a given Agent, against which control algorithms are written, extend the Agent interface. Then, the concrete Agent implementations implement that control interface.

![Fictional Agent FooBot class hierarchy.](image)

*Figure 13: Fictional Agent FooBot class hierarchy.*
Figure 13 illustrates this concept for a fictional Agent type named FooBot. In the figure, note that there are two concrete implementations of FooBot - one that communicates with the actual agent, FooBotReal, and another that communicates with a simulated agent model, FooBotSim2D.

It is also possible to define multiple control interfaces for a particular Agent type. For example, a host may want to provide two control interfaces to a FooBot - a simplified control interface that hides certain functionality, and presents only basic control commands for students in, say, a autonomous robotics course, and an advanced control interface that exposes all the bare functionality of the FooBot for use by researchers.

![Diagram of FooBot interfaces](image)

*Figure 14: Providing multiple control interfaces.*
Note in Figure 14, that the concrete class, FooBotReal, implements two interfaces. This allows both students and researchers to write control algorithms, using their respective interfaces, for the same agent.

Concrete implementation of an Agent type is discussed further, in section 6.5, Agent Modules.

6.3 Agent Controllers

Control algorithms for robotic agents are implemented as individual concrete classes, and fall under the AgentController interface hierarchy. AgentController is the top-level interface in the hierarchy, and declares methods that enable interaction with several utility classes in this library. These utility classes provide functionality such as maintaining a queue of control algorithms for a particular Agent, and executing them in a first-in-first-out order. This library also defines an abstract class, AbstractAgentController, which is the ultimate super-class for all concrete agent control algorithm implementations.

The reason for this abstract super-class stems from the fact that AgentController extends the java.langRunnable interface, which enables all control algorithms to be run in a separate thread of execution. AbstractAgentController implements the Runnable interface's run method, and handles the details of thread timing, starting and stopping, while declaring an abstract method called doWork. The doWork method must be implemented by all concrete agent control algorithms, and is the place where the actual control logic is located.
6.4 Agent World Framework

The AgentWorld interface is the top-level interface in the hierarchy that contains all simulation implementations, as well as an environment that contains real-world robotic agents. Conceptually, an AgentWorld implementation encapsulates the state and all inner workings of a particular world, and provides the information necessary to construct a visualization of that state.

For a simulated world, this encapsulation is necessarily complex. It must provide the so-called simulation "engine," where the agents and other objects that exist in the simulated world interact. It must also define some method of viewing the world, such as a top-down, 2D graphical representation, or an isometric 3D view. For a representation of a real-world agent environment, the AgentWorld only needs to contain some method of

---

In abstract class

**AbstractAgentController**

```java
public void run()
{
    initialize(); // Performs control thread init

    while(stopRequested == false) {
        doWork();
        controllerSleep(); // Thread.sleep for a set interval
    }

    cleanup(); // Performs control thread cleanup
}
```

*Figure 15: AbstractAgentController's run implementation.*
viewing the arena in which the Agents interact, such as a web cam.

The AgentWorld is the robotic toolkit-specific counterpart to the View interface described in Chapter 5, which exists to facilitate the ability of a Context to provide a visualization of itself to connected Clients. Where there is a one-to-one correspondence between a Context and a View (i.e., ChatroomContext and ChatroomView), there is also a one-to-one correspondence between an AgentWorld and a ViewPanel (i.e., AgentWorldSim2D and AgentWorldViewPanelSim2D).

The specifics of this interaction between AgentWorld and ViewPanel are as follows: An AgentWorld exists within an agent simulation or interface-type Context. The AgentWorld interface declares a method that returns a ContextViewUpdate type, which is called by the Context to obtain the current update, and then sent to all Clients that have joined the Context. On the Client side of this interaction, the View interface declares a method that receives a parameter of type ContextViewUpdate, which is called to update the View's contained ViewPanel using the information contained in that update. As described in Chapter 5, all of this communication utilizes the inter-component messaging system, where the ContextViewUpdate object is sent as a ContextMessage payload object.
Figure 16: The Context view update process.

The AgentWorld/ViewPanel abstraction encapsulates the specifics of a particular simulation engine, or real-world visualization system, and allows for easier modification of an existing implementation, or modular addition of a new implementation.

6.5 Agent Modules

As discussed above, the WART library serves not only as a collection of classes, but also as an extensible framework for creating Agent types. To this point, we have discussed the interface hierarchies that establish the skeleton of the framework, and their usage at a high-level. This section discusses the extension framework, as it pertains to Agent types. As mentioned in Chapter 1, it should once again be noted that since package structure defines namespace in Java, the two terms may be used interchangeably.
Figure 12 shows that an Agent type is provided its own namespace under the edu.wright.cs.carl.wart.agent.mod node in the package hierarchy, and depicts two such nodes: khepera, and mindstorm. Each of these represents a package containing all necessary components for all aspects of the use of that Agent in both real-world environments, and simulation. While the contents of a particular Agent module (i.e., package) may vary, the overall structure of an Agent package contains several invariant elements.

The top-level package of an Agent module may contain any common utility classes, or static constants. The standard sub-packages include .interfaces and .controllers, which contain the various control interfaces and implemented control
algorithms, respectively. In addition, an Agent module must include a sub-package for each type of simulated AgentWorld in which the Agent is intended to be used. These world-specific packages contain sub-packages and classes required for interaction with that particular AgentWorld implementation. The exact disposition of these packages is left up to the simulated AgentWorld.

Finally, in addition to the simulated AgentWorld-specific packages, an Agent must include a real-world implementing class, within the .real sub-package in order to enable control of the actual Agent. In general, this class is expected to translate Java method calls into the specific Agent protocol, transmit those messages using a Communicator instance (see Chapter 5 for a description of the Communicator interface), and handle receipt of any incoming messages from the real-world Agent.

6.6 Simulation Modules

The WART library also supports the modular addition of new simulation engines within the edu.wright.cs.carl.wart.sim modules namespace, although there are fewer specific requirements for the addition of a simulation engine type than for the addition of a new Agent type. The minimal requirements for creating a simulation engine are to provide concrete implementations for the core components depicted above in Figure 17. A simulation engine must provide a concrete implementation of AgentWorld, it must provide a GUI component that extends from the ViewPanel abstract class, and it must establish its own format for transmitting updates from the AgentWorld to the ViewPanel, through implementation of one or more classes that represent those updates. By implementing those components minimally, a simulation engine can be used by any
In the following section, we will examine the reference implementation of a two-dimensional simulation engine used by the application in this project.

6.7 Implementation of the 2D Simulated Environment

This section examines in detail the design and implementation of the two-dimensional simulation engine used by this project, the Sim2D engine. The intent of this section is to provide a concrete example of how a simulation engine can be created for use with this application.

6.7.1 Overview

Two major components comprise this simulation module: The first is the AgentWorld implementation, AgentWorldSim2D. The second is the corresponding ViewPanel implementation, ViewPanelSim2D. For the sake of clarity and brevity in the remainder of this section, AgentWorldSim2D will be referred to as "the world," or "the world model," and ViewPanelSim2D as "the view," or "the visualization."

6.7.2 The World Model - AgentWorldSim2D

The world model can be thought of as a collection of solid two-dimensional objects that exist in a bounded x/y coordinate plane, where each unit is equal to one millimeter in real-world distance. Every object is defined by the coordinates of its center point, its rotation, and the shape of its bounding box, which is implemented using a Java 2D geometric shape from the java.awt.geom library. These characteristics are defined by the
ObjectSim2D interface, from which all objects in the world model descend. Additionally, these objects are partitioned into two categories: Active, or inert. Inert objects generally represent things that do not change their state (including location, or any other internal state variables) unless acted upon by another object. Active objects represent things that have the capability of changing location and state independently, and additionally implement the ActiveSim2D interface, which defines a single method, \textit{update}, which causes the object to update its state based on its own internal logic. Active objects include, but are not limited to, simulated robotic agents - For example, another type of potentially active object might be a light that turned itself on and off at a regular interval.

The world model runs in a separate thread of execution from the rest of the application, and is repeatedly updated at a regular interval. While running, the world's update thread iterates over all of the active objects that it contains, calls each active object's update method, and then resolves any collisions between solid objects that may have resulted from the movement of an active object. In order to tie the simulation engine to a real-world measure of time, the world model defines a specific number of updates per second, from which the duration in milliseconds of each update cycle is calculated. This value is used in two ways: First, to ensure that the update thread performs its updates at this rate, the thread sleeps every update cycle for this duration, less the amount of time required to execute the update itself. Second, every active object is passed this duration value when its update method is called, and it uses the duration in its update logic - for example, a mobile robotic agent, moving at a given rate of speed, uses the update duration to calculate the correct movement distance.

Collision detection between two objects is performed using the \textit{intersectsObject}
method declared in ObjectSim2D. This method, defined by each solid object implementation, receives a single ObjectSim2D parameter, representing another solid object, and returns true if the object supplied in the parameter intersects with the called object. Most solid objects implement this method by simply checking to see if their bounding box intersects with the supplied object's bounding box, using the intersection-checking methods in the Java geometry library, but implementations may choose to utilize more complex shapes or areas when determining collision.

Collision resolution in the Sim2D engine uses a "relaxation" technique, rather than implement a true physics model. This technique is implemented, first, by partitioning all solid objects into three categories: The first category contains immovable obstacles, the second contains only the robotic agents, and the third contains all other movable solid objects. Based on these categories, collision resolution rules are defined as shown in Figure 18.

<table>
<thead>
<tr>
<th>ObstacleSim2D</th>
<th>AgentSim2D</th>
<th>ObjectSim2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>May intersect other obstacles.</td>
<td>May not intersect any other solid objects.</td>
<td>May not intersect any other solid objects.</td>
</tr>
<tr>
<td>Immovable.</td>
<td>When intersecting an obstacle, this agent moves.</td>
<td>When intersecting an obstacle, this object moves.</td>
</tr>
<tr>
<td></td>
<td>When intersecting another agent, both move equally.</td>
<td>When intersecting an agent, this object moves.</td>
</tr>
</tbody>
</table>

*Figure 18: Collision resolution rules.*
In the collision detection and resolution phase of the update cycle, the three sets of solid objects are tested for collision against one another, and the sets of Agents and Objects are tested against themselves. This batch of comparisons occurs only once each update cycle, even though the movement of a solid object to resolve a collision could result in another, different collision. Performing this "relaxation" step a single time per update cycle is an acceptable decision, for three main reasons. First, given enough update cycles, all collisions will resolve. Second, a collision "loop" might occur if the collision resolution phase continued until all collisions were resolved every update cycle - that is, a set of collision resolution movements may repeat infinitely. Third, as the number of relaxation steps is increased, the more objects would seem to "jump" from one location to another, as collisions were resolved.

6.7.3 The Visualization - ViewPanelSim2D

The visualization element of the simulation engine is a great deal more simple than the engine itself. In the same way that the world model can be thought of as a collection of solid objects with position, rotation, and a bounding shape, the visualization may be thought of as a two-dimensional bounded plane represented by a java.awt.geom.Graphics2D drawing pane, where each pixel equals a specified number of millimeters. This visualization "world" contains a collection of graphical objects, each of which is defined by its center point and rotation. Each of these objects, given a reference to the Graphics2D pane, can render itself correctly on that pane. A ViewPanelSim2D obtains instances of that collection of graphical objects from the AgentWorldSim2D in
the form of an initialization view update. This update is transmitted from the Context to
the Client as the payload of a message, using the standard message-passing facilities.

The characteristics and behavior of a graphical object are defined by the
DrawableSim2D interface, from which all visualization object implementations
ultimately descend. There is a one-to-one correspondence between an ObjectSim2D
instance in the world model, and a DrawableSim2D instance in the visualization.

<table>
<thead>
<tr>
<th>BallSim2D</th>
<th>DrawableBallSim2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server-side object.</td>
<td>Client-side object.</td>
</tr>
<tr>
<td>Exists within the AgentWorldSim2D</td>
<td>Exists within the ViewPanelSim2D</td>
</tr>
<tr>
<td>object.</td>
<td>object.</td>
</tr>
<tr>
<td>Has a position and rotation in the</td>
<td>Given a position and rotation in the</td>
</tr>
<tr>
<td>world model coordinate plane.</td>
<td>GraphicsSim2D drawing pane and a</td>
</tr>
<tr>
<td>Has a bounding box.</td>
<td>reference to the pane, can render a</td>
</tr>
<tr>
<td></td>
<td>image of itself.</td>
</tr>
</tbody>
</table>

Figure 19: Relationship between solid object model and graphical object model.

This relationship results in a simple format for the view updates generated by
AgentWorldSim2D, since they need only to contain the position and rotation of each
solid object for the ViewPanelSim2D to be able to correctly render an image of the
simulated world. This update, like the initialization update mentioned above, is
transmitted via the message-passing system.
6.8 Implementation of the Khepera Agent Module

This section describes the implementation of the Khepera agent module used by this project. The intent of this section is to provide a concrete example of how an Agent Type may be added, as well as to demonstrate how to create the classes necessary for a newly-created Agent to interact with a simulation engine.

The Khepera agent module is implemented in the edu.wright.cs.carl.wart.agent.mod.khepera namespace, and contains the sub-packages described above. A detailed discussion of the contents of those sub-packages follows.

6.8.1 Control Interfaces and Controllers

The Khepera agent module defines a single interface in the .interfaces sub-package: KheperaStudent. This interface provides a simplified façade to the actual low-level Khepera commands, and is identical to the control interface defined in the previous iteration of the Khepera software. This was done to allow control algorithms created in the old software to be used here. Khepera controller class files are contained in the .controllers sub-package.

6.8.2 Real-World Implementation

The real-world Khepera implementation is fairly straightforward. There is a single class named KheperaReal, which implements the Agent interface for use with the WART library tools, the KheperaStudent interface for use with AgentController implementations, and the CommunicatorEventListener interface, for use with the local Communicator. This implementation maintains a set of state variables, representing the current state or value
of the Khepera's sensors and effectors.

The KheperaStudent interface methods that set an effector (i.e., change the wheel speeds, or raise the gripper arm) are simply mapped to the appropriate commands in the Khepera Serial Protocol, which are sent via Communicator to the Khepera. Interface methods that query a sensor for its current values, however, work in a slightly different manner: When a sensor query method is called by a Khepera AgentController, the appropriate sensor query command is sent to the Khepera, and then the values currently stored in the local state variables are returned. It is important to note that the returned sensor values are actually the values returned from the Khepera from the previous query, not the one that was just sent. After these "stale" sensor values are returned, the Khepera's response with "fresh" sensor values will be received and processed, and the state variables updated. The assumption is that, since a control algorithm runs in a tight loop, any sensor query that is made once will be made repeatedly, and so returned values will only lag in time by the duration of a single update cycle, which occurs at a tens-of-milliseconds interval.

While writing messages to the Communicator is straightforward, receiving them is not, due to the fact that the Khepera may send a response message in several pieces, depending on its length. As a result, the following scheme must be utilized for reading responses from the Khepera: The real-world implementation, having registered itself with the Communicator, receives responses from the actual Khepera, reassembles them into complete Khepera Serial Protocol responses, and then adds them to a processing queue as the end of each response is received.
The KheperaReal implementation contains a message processing thread, which periodically processes messages in its queue, translating them into the appropriate state changes.

Figure 20: Khepera Serial Protocol response processing.
6.8.3 Simulated World Implementation

The KheperaSim2D class implements the AgentSim2D interface, which is a subtype of the ObjectSim2D and ActiveSim2D interfaces. The AgentSim2D interface represents a single solid object composed of multiple, discrete components. The components, in this case, represent the Khepera base module, and its add-on expansion turrets. An AgentSim2D implementation behaves as if it were a single active ObjectSim2D in the following three ways: 1) When the agent moves, or is moved, all the attached components are moved, maintaining their relative positions to the center of the agent. 2) When the agent's update method is invoked by the AgentWorld, it iterates over each attached component, invoking their update methods in turn. 3) For the purposes of collision detection, the agent provides a bounding box that is the tightest bounding box that contains all of its attached components.

This componentization of the Khepera does not provide any particular benefits over a single, monolithic implementation for this project. However, a feature that has been identified for future development is to allow users to create new Agent types in simulation by combining a collection of separately implemented robotic components. The componentized AgentSim2D was implemented in anticipation of the future addition feature.

The Khepera's passive collision detection behavior is the same as any other solid object, using bounding box intersection to determine if it collides with another object. However, there is one notable exception to this behavior due to a particular Khepera component - a gripper arm extension turret, which has the capability to grab objects.
When the gripper arm is in the down position, the Khepera's bounding box is the smallest rectangle that contains the bounding boxes of the gripper arm, the base unit, and any other components attached to the Khepera. However, in order to effectively grip an object, the Khepera must be able to allow objects within its bounding box between its gripper tongs, when the grip is open. This is accomplished by defining two rays, from the center point of the gripping area, shown in the figure below.
The Khepera's collision detection method then tests to see if an object's center point falls between those two rays. If so, the object may intersect the Khepera's bounding box, but it will not "collide." Note that an object may be within the gripping area, but its center point may fall outside the acceptable area if it gets too close to the Khepera. This behavior results in an object that appears to be "pushed" by the gripper arm.

The Khepera's active behavior consists of three main activities: Movement is based on the motor speeds at the time of the update cycle, and is calculated based on the differential wheel speeds, axle length, and duration of movement. Sensor values are calculated by determining the distance to the nearest object within a sensor's arc, and then applying that distance to a table of sensor return values obtained from the Khepera user manual. Finally, gripper turret state is updated, which can result in an object being gripped, or dropped, depending on the current state of the gripper.

The DrawableKheperaSim2D class provides the graphical representation of the Khepera in the ViewPanelSim2D. This class contains a number of static images, each corresponding to a particular Khepera gripper turret state. There is a corresponding image for each combination of the arm states (being raised or lowered), and the grip states (being open, closed, or holding an object), resulting in a total of six images. The view update for the Khepera includes the standard position and rotation data, but also includes the gripper and arm states so that the correct image reflecting the simulated Khepera's state is drawn.

In addition, the KheperaSim2D's view update contains other data, such as current sensor readings, which can be used to render other visual components in the ViewPanel,
such as a sensor value panel.
7. THE WART APPLICATION

7.1 Introduction

The WSU Autonomous Robotics Toolkit Application (WARTApp) is designed to provide a single application that addresses all of the needs of a group wishing to work collaboratively and simultaneously on problems related to autonomous control of robotic agents, in a distance education based format. It utilizes the class libraries and extension framework described in the preceding two chapters to implement its features, which include online text-based chat, robotic agent simulation, and control of real-world robotic agents.

The application provides these features in a format that is accessible to users with widely ranging technical skills and backgrounds so that it may be used in a number of different settings, including for autonomous robotics research, a graduate or undergraduate class-setting, or even in secondary education. This application is designed and implemented with the intent of fulfilling the application feature specifications listed in Chapter 2 of this document.

This chapter describes the design and implementation of the WARTApp, starting with the use of the Model-View-Controller architectural pattern to properly segregate business logic from user interface, followed by a discussion of the user interface design, concluding with a detailed description of the implementation of the Context-based functional units that provide the required features of this application.
7.2 Application Design

In the MVC architectural design pattern, the Model component of this application is implemented through the use of the class libraries described in the preceding chapters, as well as the Context implementations contained within the application namespace itself. The two core classes that define the "model" are the Client and Server classes (implemented by DefaultClient and DefaultServer). An instance each of these two classes contains the business logic of, and provides the interface for the core functionality of the application.

The View component is implemented by the WartappView class, which contains all of the top level user interface components. This class is implemented as a component of the Swing Application Framework.

The Controller component is implemented as the Wartapp class. This class acts as an interpreter between user interface (WartappView) and data model (Client and Server instances). As such, this is the class that instantiates and maintains reference to the Model's class instances. This class is implemented as part of the Java Swing Application Framework, as well.

7.3 User Interface Design

The main guiding principle behind the design of the user interface is that of ease of use for non-technical users. The application user interface was therefore modeled after a commonly-used application type, a tabbed web browser. By modeling the user interface in such a way, the user will be able to intuit the functions and behavioral semantics of the
various parts of the interface, which are common to both this application, and that of the browser-style applications after which this one is modeled. The remainder of this section highlights several key features of the user interface.

7.3.1 Address Bar

The top-most control is the address bar. The intent of this component was to allow users to connect to a remote host running this application by simply entering the host's network address or name. However, a web browser is able to make several assumptions that do not apply to this application, such as the port number (80), and the protocol type. As a result, the behavior of this application's address bar differs slightly. When a user enters a new host address, they are prompted by a dialog to enter the port number, protocol type, and login information. This connection is then saved, and the connection attempt is made.

Also, much like a web browser, this input device combines a text input field with a drop-down combo box. This is intended to replicate the behavior of many browsers that store frequently visited host names and network addresses in the address bar control.

![Image of Address Bar with Expanded Dropdown Box]

*Figure 23: Address bar with expanded dropdown box.*
The dropdown behavior of this component was designed around the concept of a saved connection. A saved connection is a persistent entity that contains the host name, port, and connection type of a remote server as well as the user credentials for login. The dropdown box contains saved connections, which the user can use by selecting and clicking the Connect button. Users may also manage their collection of saved connections through the menu bar by selecting the Manage Saved Connections under the Client menu item.

At the time of this writing, the address bar does not have the parsing capabilities of a browser, such as the ability to specify port number, protocol, or login credentials on the address bar, though this feature has been identified for future development.

7.3.2 Tabbed Browsing Pane

In addition to the address bar, another major user interface element was modeled after a tabbed browser application, the main tabbed browsing pane. This interface element contains a number of tabs, each containing a scrolling pane-enclosed interface component. This nesting is required to ensure that each tab exhibits correct scrolling behavior typical of browser applications.
There are three main tab types. First, every application has a persistent home tab. This tab contains controls and status displays for the user's own local server. This pane allows the user to start, stop and rename their server instance and manage their hosted Context-based functional units and connected users. The second tab type is the remote server tab. This tab is created when a user connects to a remote server, and it acts as the "lobby" of that server. The user can view connected users, and hosted Contexts. This tab provides the entry point to the use of a remote server's Contexts. The third tab type is a Context visualization tab. This tab type is created whenever a user joins a Context, whether it is hosted locally, or on a remote server. This tab contains as its top level
interface panel, a ViewPanel implementation that provides the visualization of that
Context. Two additional tab types have been identified for future development - a remote
server administrative tab, and a remote Context administrative tab. These tabs are
proposed to provide remote users administrative controls (i.e., kicking users, adding new
Contexts) if they possess sufficient permissions to do so remotely.

Tabs are automatically created when connections are established - for example, on
collection to a remote server, a remote server tab is created and added to the tabbed
pane. When joining a Context, a Context tab is added. However, the behavior exhibited
on closing a tab in the pane is slightly different depending on the nature of the tab.

Closing the home tab simply removes the visual element from the tabbed pane, but
does not actually dispose of it. When a user wishes to reopen their home tab, they can
click the Home button located near the address bar, much like the home button of
browser applications. This action returns the home tab to the first tab position. Closing a
Context tab causes the user to leave (i.e., disconnect from) a Context. This behavior is
common, regardless if the Context being left is hosted locally or on a remote server. Of
note, closing the tab of a locally-hosted Context does not deactivate or remove that
Context, and other remote users may continue to utilize it. In order to perform those
actions, the user must use the controls on their home tab. Closing a remote server tab
causes the user to disconnect from that server, which includes disconnecting from all
Contexts on that server. This ensures that all inactive tabs will be closed when leaving a
server.

In addition to voluntary connection and disconnection, users may be forcibly
removed from both Contexts and a remote server, by the hosting user. This action causes
tabs to be closed as appropriate to reflect the lost connection or connections.

7.3.3 Additional Interface Features

Immediately under the address bar, all functional units (i.e. Context types) are available to the user as graphical button controls, and are created, joined, and immediately in use following a single click. This user interface component is in accordance with the design specification that requires exposure of core functionality in the top level of the interface.

Also, the home tab and remote server tabs provide a private messaging facility. Users can send private text-based messages to other users using the Message button located in both home and remote server tabs. Private messages appear as message dialog boxes on the recipient’s machine.

7.4 Contexts

WARTApp provides three major functional units: A chat room in which multiple users can communicate via simple text messaging, a simulated agent arena, in which users can create a simulated environment with robotic agents, deploy control algorithms to those agents, and observe the results of the execution, and a real-world agent interface environment, where users can deploy control algorithms to actual agents connected to a remote host to validate the behavior of their controllers. Each of these functional units is represented by a Context type. A user may host one or more of each Context type locally, and allow multiple users to connect, and utilize those Contexts.

In terms of the implementation, it should be noted that all of these Context types
descend from an abstract super-class, AbstractContext. AbstractContext implements all of
the general basic functionality of a Context, requiring its inheritors to implement only the
behaviors that are unique to that particular Context type. Additionally, each functional
unit represented by a Context type must also have corresponding View and ViewPanel
implementations, representing its user interface (which, as described above is displayed
in a tab pane), and so the following sections also include a description of their View and
ViewPanel-based user interfaces.

7.4.1 Chat Room Context

The chat room Context is a relatively simple functional unit, and is implemented
by the ChatContext class. Its primary function is to receive text-based chat messages sent
from connected users, and then broadcast those messages to all users. This is
accomplished by using the standard message-passing system in the following manner: A
Client sends a ClientMessage containing the text payload via the message-passing system
to the ChatContext. The ChatContext's message handling routine determines the sender's
username from the Message wrapper, and then obtains the chat text message from the
message's payload. It then creates a ContextMessage, containing a ChatWindowUpdate
payload, and broadcasts this message to all connected Clients. Each Client receives the
message, and routes it to the correct ChatContextView, where the text message is
displayed by the ChatContextViewPanel, on the remote Clients' application interfaces.

There are a number of other message payload types that are passed between the
ChatContext and its connected Clients. When a user joins or leaves, the
UserStatusChange payload is employed to communicate this, and it is used to both
announce the joining or leaving of a user to all other users, and to update the Clients' display list of connected users. Additionally, the host of the ChatContext can forcibly remove other users from the ChatContext. This is transmitted via a UserRemovalRequest message payload from Client to Context, which checks to see if the user requesting removal of another user has sufficient permissions to do so. If so, the ChatContext removes the user, and transmits a UserStatusChange payload to all remaining users, informing them of this change.

Given that, at this time, only the local hosting user can request removal of another user, it may seem odd that user removal requires the use of the messaging system. However, as mentioned above, a future development will be to allow remote administration of servers and Contexts. Therefore, in anticipation of this feature all communications from a local user are routed through the message-passing system.

The user interface defined by the ChatContextViewPanel is relatively sparse, and contains only a few necessary elements.
Figure 25: Chat room user interface.

The main chat window displays chat messages from other users and messages regarding the status of other users, such as when they join, leave, or are forcibly removed. Chat messages are sent to the Context using the text input field below the main chat window, and a list of users is displayed to the right of both.

7.4.2 Simulated Agent Context

The primary function of the simulated agent context, which is implemented by the
AgentContext class, is to provide access to robotic agents in a simulated environment so that users can test developed control algorithms. The context also provides chat room capabilities, as well, to facilitate more seamless online collaboration among remote users. There are several features in the AgentContext that exist in the ChatContext. Specifically, the chat display window, chat text entry field, and user list are also present in the AgentContext. These features are implemented in a similar fashion, so the description of the details of their implementation in the preceding section applies to these features here, as well.

The unique feature that the AgentContext provides is the ability to control robotic agents. Users are provided a list of agents by name, and they can request to control one or more of them. This request is communicated through the standard messaging-passing system, using a payload type of AgentControlRequest. This payload contains an instance of a control algorithm and the name of the selected agent, which is transmitted to the AgentContext, and added to the selected agent's control queue, if the requesting user possesses sufficient permissions. If the request is successful, the user's controller is placed into a first-in-first-out queue of controllers, it is run for a specified amount of time once all preceding controllers have run, and is then stopped and removed from the queue.

The user interface of the simulated AgentContext is implemented by the SimAgentContextView and SimAgentContextViewPanel classes. It contains primarily a graphics panel component, on which the simulation visualization is drawn.
Figure 26: Simulated agent context user interface.

An important component of the simulation is the world editor. This feature allows a user to place, delete, move, and rotate items in a simulated world, using a graphical editor interface.
Maps are saved and loaded by respectively serializing and deserializing instances of AgentWorldSim2D.

7.4.3 Real-World Agent Context

The real-world agent control interface is identical to the simulation, and it uses the same Context implementation, AgentContext. Since AgentContext specifies a member instance of type AgentWorld, which both AgentWorldSim2D and AgentWorldReal implement. In addition, AgentContext's facilities for storing references to robotic agents
use the interface type, Agent, for its collection. This allows both real-world, and simulated agents to be stored and utilized by AgentContext.

There are two major differences between the real-world Context and the simulated agent Context. The first difference is in creation. When the simulated AgentContext is created, a simulated world map is either loaded or created, and the AgentWorldSim2D is created using that. In contrast, real-world agent control Context creation provides the user the opportunity to both auto detect any connected Agents, and manually add connected Agents, so that they will be made available in the Context for deployment of a control algorithm.

The second major difference is in the ViewPanel implementation, RealAgentContextViewPanel. As shown above, the simulated AgentContext's view is top-down of a two-dimensional simulation. As of this writing, the real-world ViewPanel implementation does not provide any visualization of the robotic agents. However, this feature has been identified for future development.

The layout and functionality of the user interface for the real-world agent context is the same as that for the simulation, excluding the main graphics pane.
8. TESTING AND VALIDATION

8.1 Introduction

This chapter focuses on description and discussion of the measures taken to ensure that the application functions properly, through testing, and fulfills its specifications, through validation. The section of this chapter on testing discusses testing methodologies in general and also specific examples and results, where applicable. The validation section contains a general discussion of the features of the application and class libraries that fulfill the required specifications.

8.2 Testing

Testing was employed throughout the implementation of this project, from initial design and implementation, through final validation. Classes were tested as they were implemented to ensure, at least, basic features functioned properly. Smaller assemblies of classes that composed a subsystem of the application were likewise tested in a similar manner. The application, as a whole was tested by core feature, and finally the application's resource usage and limitations were tested, both to obtain performance and resource utilization benchmarks, as well as to identify any potential resource-releasing issues.

This section outlines these testing procedures in general, and also describes how they were applied to the class libraries and application.
8.2.1 Unit Testing

The most commonly used testing methodology in this project was unit testing. These tests were usually conducted as an individual class, or small subsystem (generally < 5-10 concrete classes) was completed. Unit testing was conducted on two tiers: Superficially, all classes were tested for correctness of their basic functionality, using the code paths that defined normal operation. But depending on the availability of resources, and the importance of a particular class, more in-depth unit testing was performed.

Unit testing of this more in-depth variety typically focuses on two aspects of the source code being tested: Boundary-value analysis (BVA), and code coverage. BVA is conducted by identifying the input partitions for a method, defining the boundary values of those partitions, and then testing methods with inputs from all identified partitions and their boundaries. However, the bulk of the code in this project utilizes input parameters that are Java reference types, which simplified the task of identifying input partitions for each parameter. For the purposes of boundary values analysis-based unit testing, any reference type can be considered to have three input partitions: a valid reference to a valid object, or a valid reference to an object that is itself invalid, or an invalid, null reference. The concept of validity for an object here is taken to mean that it does not violate any of its class invariants.
Given that objects in these libraries and application are of widely-varying, usually significant complexity, the prospect of testing the middle case (valid reference, invalid object instance) would have required testing harnesses to obtain instances of a particular class in all possible valid and invalid states, which for n member classes, each with m states would result in $n^m$ test cases, per input parameter, per method. Due to resource constraints, it was not feasible to specifically and exhaustively test all of these cases, and errors of this type result in easily-traceable and reparable runtime exceptions and errors. As a result, unit testing was focused primarily on code-coverage testing, which inherently

---

*Figure 28: Reference-type input parameter partitions and behavior.*

```java
public void addAgent(Agent);  

Requires a non-null, Agent type input parameter, and also required that the Agent type have a non-null, non-empty String name.
```

```
<<Implements Agent>>  
FooBot  
String name = "Larry";
Valid reference, valid object.
```

```
<<Implements Agent>>  
FooBot  
String name = null;
Valid reference, invalid object.
```

```
<<Implements Agent>>  

null reference.
```

```

public void addAgent(Agent);  

Requires a non-null, Agent type input parameter, and also required that the Agent type have a non-null, non-empty String name.
```

```
Correct behavior occurs.
```

```
Specifically, undefined. Typically, this results in an eventual runtime error or exception, such as a NullPointerException.
```

```
NullPointerException, the method fails fast.
```
includes the remaining two input partitions.

In code coverage-based unit testing, the focus is on ensuring that every statement in a particular class’s source code is executed during testing, and that it functions properly. In order to accomplish this, the test harnesses provide a set of test cases that exercise every statement in a particular class's source code. In the case of standard syntax control and branching statements, it is generally a straightforward task to cause a method to branch in a particular way based on either the method parameters, or by simulating a response from a member component of the class being tested. This technique was used to provide code coverage

Of note, however, this analysis provides code coverage of normal execution paths, and does not include error paths such as exception handling code.

8.2.2 Feature Testing

WARTApp is a multi-user, multi-threaded application, and regardless of how thoroughly classes are unit tested, there remain potential software faults in error-handling code, and concurrency issues that will generally not be exposed until the application is tested as a whole. With this in mind, feature testing was conducted on the application. This involved generating a set of use case scenarios, executing those scenarios using the application, and observing any resulting errors.

A use case scenario is a plain text description of an end user activity, or series of activities that represent a typical utilization of the application. The intent of use-case-scenario-based feature testing is to exercise the most typically-used code paths, and to ensure that major features are operational. Feature testing using this method exposes
errors and elicits incorrect behaviors that would not typically be discovered in unit testing. Where the focus of unit testing is explicit code coverage, and is considered to be "clear box" testing such that the tester is aware of the actual source code and uses it to guide his or her testing efforts, the focus of feature testing is closer to the concept of "black box," or "opaque box" testing, where the tester's efforts are not driven by source code. Rather, the tester's focus in feature testing is to verify the proper, error-free functioning of application features.

Client/Server Use Cases
Login Locally, Connect to Remote Server, Leave Remote Server, Kick User, Send Private Message, Change Server Name, Change Server Port

Connection Manager Use Cases
Add Saved Connection, Modify Saved Connection, Delete Saved Connection

Common Context Use Cases
Add Context, Remove Context, Join Context, Leave Context, Send Chat Message, Kick User

Agent Context Use Cases
Control Agent, Adjust Zoom (Simulated Context only)

Sim2D Editor Use Cases
Open Map Editor, Save Map, Load Map, Add Object, Add Obstacle, Add Agent, Remove Object/Obstacle/Agent, Move Object/Obstacle/Agent, Rotate Object/Agent, Adjust Map Size

Figure 29: Feature testing use cases.

In order to test the actual feature itself for validation purposes, each use case was
executed with the minimal activity necessary to execute the feature. In order to test for potential concurrency faults, the features that involve multiple users or a remote connection were tested multiple times, with a varied sequence of events. For example, the use case of "Control Agent," in the Agent Context Use Cases category was tested with a single user locally, with one local user and one remote user, with multiple concurrent control requests, etc.

Admittedly, this sort of testing intersects significantly with end-to-end "fuzz" testing, where random (or unexpected) inputs from the user interface are applied to the application in an attempt to elicit software errors. Additionally, there are a vast number of conditions and orders in which events can be performed, for which it was not possible with the resources available to exhaustively test, or even to enumerate. However, this type of testing, in the case of this project, acted as a replacement for "alpha" and "beta" testing, where errors are discovered through repeated and varied use by a large number of testers.

8.2.3 Stress and Performance Testing

Stress and performance testing are two related, but slightly different types of test. While both tests are measured using a specific set of quantifiable performance benchmarks, such as resource usage, animation frame rate, graphical component rendering time, etc, they differ in their overall goals. The goal of performance testing is to determine the values for the selected performance benchmarks during normal, average-case use. The concept of "normal," or "average-case," use is measured in terms of some quantifiable load factor, which for this application could be, for example, the number of
remote connected users, or the number of simulation Contexts currently running. Stress testing is intended to determine the maximum values for those selected load factors where the application’s performance falls outside of some specific range as measured by the selected benchmarks.

The most resource-intensive aspect of this application is the simulated agent context, due to its need to render a graphical representation of the simulated agent world in addition to performing all of the same functions that the other Context types do as well. Therefore, stress and performance testing were conducted on the simulated agent Context only.

Two scenarios were selected for stress and performance testing. In each scenario a single load factor and a single performance benchmark were selected. The specific performance benchmark in each case was selected as the benchmark that was the most sensitive and specific to changes in the load factor – that is, the selected benchmark represented the first point of failure when the particular load factor increased to a point outside of normal operating parameters.

The first scenario involves a number of active simulated agent Contexts, each with one Agent. The load factor in this scenario was chosen as the number of Contexts, and the performance benchmark was heap memory usage. The second scenario involves a single simulated Agent Context, with some number of active Agents, each running a control algorithm. The load factor in this case was selected to be the number of active Agents, and the performance benchmark was simulation graphical panel frame rate. In both scenarios, acceptable bounds for the benchmark were chosen, and then that benchmark was measured over increasing load factor. Finally, for each scenario, an "average-case"
The load factor was determined.

The hardware used for stress and performance testing was a PC with an Intel Core 2 Duo dual core processor, running at 2.66 GHz with 2 GB of RAM and an NVidia GeForce 8600 GT video card with 256 MB of GDDR3 RAM running Windows Vista 32-bit edition.

In the first scenario, the Netbeans 6.5 IDE profiler plug-in was used to measure memory use, as the number of agent simulation Contexts was increased. The upper bound of the acceptable range of the performance benchmark was chosen to be the default maximum Java Virtual Machine (JVM) heap size for a 32 bit JVM, which is 64 MB.

![Figure 30: Heap memory usage.](image-url)
As demonstrated in Figures 30 and 31, heap memory use increased in a nearly monotonic fashion as simulation Contexts were created, starting from a baseline of 14.7 MB with no Contexts, increasing roughly 7.6 MB per Context addition. When attempting to add the seventh simulation Context, an OutOfMemoryError was thrown, and resulted in a failure to create the Context. Of note, all other aspects of performance (i.e., frame rate, updates per second, etc.) were well within the bounds of normal, acceptable performance throughout this particular stress test.

The major implication of this test is that the default maximum heap size for the JVM may be insufficient for users running a server that hosts a large number of simulation instances. Also, development of a more memory-intensive type of simulation engine may require tuning of the JVM maximum heap size.

In the second scenario, a frames-per-second monitor was added to the Context visualization class’s rendering thread. The lower bound of the performance benchmark was selected to be 20 frames per second, at which point pauses between renderings become visibly noticeable. The upper range for this benchmark is set by the minimum
rendering thread sleep interval, which is 20 ms per iteration, yielding a maximum refresh rate of 50 frames per second. The load factor was the number of simulated Khepera in the simulation each running a simple obstacle avoidance control algorithm, which were increased until the frame rate decreased below the lower bound of the performance measure.

![Rendering Rate](image)

**Figure 32: Frames per second per number of Khepera.**

It was observed that the frames per second rate decreased as the number of simulated Khepera increased. A frame rate at or below the lower bound for the performance benchmark of 20 frames per second was consistently observed between 35 and 40 simulated Khepera in a single simulation instance.

As demonstrated in this section, the application performs sufficiently well, within a reasonably wide range of load factors, for the most processor-intensive feature of the application. There were no observed memory or other resource leaks, and the application
performed consistently over a reasonable period of continuous usage.

8.3 Validation

The final step in this project is to perform validation of the application and class libraries to determine if they meet the design specifications, and provide the required features. Validation is also known as acceptance testing, and is, in general, performed by the end user of a particular software product. In this case, validation will be performed by examining the design specifications, and then identifying the feature or features that fulfill a particular specification item.

The initial specification list was organized into four categories - general application specifications, which encompass specifications that apply to the user experience and general functionality including networking features and characteristics, agent-specific (i.e., simulation and controller deployment related) specifications which include features directly related to the simulation and control of robotic agents, collaboration feature specifications which include features that affect the ability of users to communicate and perform work in groups, and security specifications, which include features that provide users the ability to control access to their hosted Contexts and prevent misuse. This section contains this analysis on a category-by-category basis.

8.3.1 General Application Specifications

The features in this category included general application features and characteristics. Through its use of the Swing Application Framework, which provides a Model-View-Controller pattern-type framework, the user interface is kept separate from
the model, so that new user interfaces may be developed without modifying the underlying code. The interface itself includes controls to allow users to immediately begin simulations or control attached agents, at the top level of the GUI. These aspects of the application fulfill user interface-related specifications.

The CARL networking library provides encapsulation of both network, and local (i.e., host to attached device) communications protocols, which fulfills the general networking and communications protocols extensibility specification.

8.3.2 Agent Simulation and Interface Specifications

This category contained specifications related to the simulation of robotic agents, and interaction with their real-world counterparts. The WART robotics extension framework library provides most of the functionality that fulfills the specifications in this category.

There is a reference implementation of a two-dimensional simulation engine included in the WART library, and also an implementation of a control system for real-world robotic agents. Both of these support the use of multiple agent instances, and multiple agent types. The implemented Khepera agent module provide a reference implementation of a common control interface between real-world and simulated agent, which, if used as a model for future agent type modules, enforces a common interface.

8.3.3 Collaborative Feature Specifications

This section contained specifications related to features of the application that facilitate online collaboration, and increase the utility of it as an online teaching tool.
These specifications are fulfilled mostly by the Context implementations included in the application package. The implemented Context types include a chat room, a robotic agent simulation and real-world interface Context types. These implementations fulfill all of the collaboration-related specifications, excepting one.

The last collaboration specification requires that the application support the addition of newly-implemented collaboration features in a modular fashion. It should be noted that the design of the CARL networking library, especially the Client-Server-Context architecture, fulfills this specification. However, there is currently no facility for utilizing newly-added Context types at runtime, without modification of the user interface of the application. While this is not part of the specification, it should be noted that this feature would greatly expand the utility and usability of the application due to the fact that developers could implement new Context types, and deploy them to users without also rewriting the application user interface. Additionally, users and developers could exchange and share Context types. This feature has been identified as a potential future development, and is discussed in Chapter 9.

8.3.4 Security Specifications

This section outlined specifications intended to provide basic security features to the application, protect host computer resources, and robotic agent hardware. The CARL networking library provides the user authentication and access control features that fulfill the related specifications. The application, through its implemented Context types, provides the remote user removal feature.

However, there are two aspects of the security-related specifications that require
additional discussion here. First, the function-based access control specification is fulfilled by the inclusion of access control list and permission verification for certain operations and the application maintains a persistent access control list. However, the actual ability to modify permissions has not been implemented at the time of this writing. The "ability," requirement of the specification, has been fulfilled in the respect that a developer can implement access control list modification, but not in the respect that an end user can do so through the user interface. This feature has been identified as a candidate for future development.

Second, a point in one security specification states that a hosting user has the ability to restrict remote control of his or her hosted agents. This feature is currently in place, but not in a finely-grained implementation. That is, a hosting user may restrict remote control of agents in a real-world control interface Context by simply removing or inactivating the Context. However, there are no facilities to restrict access on an Agent by Agent basis.
9. CONCLUSIONS

9.1 Summary

The purpose of this document was to describe the design and implementation of a framework for the simulation and control of autonomous robotic agents. This project was motivated by the software requirements for an experimental course in teaching the design and development of control systems for mobile autonomous robots.

The first chapters (1-4) outlined the motivation for this project, and provided an outline of the requirements taken from the technological needs of the course, as well as the larger-scale requirements obtained from the encompassing research project. Those general requirements were analyzed to identify design specifications for the software. From those specifications, the overall design strategy was decided, which involved creating a general networking library and a mobile autonomous robotics extension framework, upon which applications could be built.

Following the collection of initial requirements and analysis that yielded the design specifications, the next several chapters (5-7) described the specific implementation details of each component - the CARL networking library, the WART autonomous robotics extension framework, and the WARTApp application built using those two libraries.

Finally, testing of the resulting application was discussed, and validation against the design specifications was performed, which demonstrated that the application and
libraries fulfilled the initial requirements, and the specifications.

The remainder of this chapter discusses potential future developments using this software application, and the underlying class libraries.

9.2 Future Work

The extensible nature of the CARL library and the WART extension framework lend themselves to easy, modular extension. This section contains a survey of potential future developments. First, items that can be regarded as improvements and modifications to the existing application itself, which can be implemented within the current application's user interface and data model, are listed and discussed. The final item, Runtime Module Loading, falls outside the scope of the existing application, and would require a new application and user interface (or at least significant modification to the existing one).

9.2.1 Remote Server Administration

At this point, the application only supports local control of a server. That is, only the local user has the ability to manage hosted Contexts, the server account list, and control access of remote users. The access control list model, which is not currently in use in the application, could be used to allow different remote users to have different sets of permissions, depending on their user credentials. This would permit the exposure of an administrative-type ServerHandle, where additional server functionality could be provided to a remote user possessing sufficient permissions.

This would require the implementation of an access control list modification
interface, the implementation of a new type of ServerHandle providing additional functionality, and also a new user interface tab, corresponding to the administrative ServerHandle type.

It should be noted that while the access control list model is not currently used to provide differential permissions, all the necessary code is in place in the application, and permissions are actually checked as users attempt to perform various actions. Additionally, servers store a persistent copy of their master access control list, which contains valid entries for each user account. However, at this point all user accounts are given full permissions when they are created, there is no user interface component that allows access control list modification, and remote users are simply not provided the interface to perform administrative functions by the Server and Context remote handles. The implication of this is that, in order to securely implement the remote server administration feature, access control list modification must first be added.

9.2.2 Address Bar Improvements

As mentioned above in Chapter 7, the user interface of the application is designed to mimic that of a web browser application. However, it currently lacks several key features that web browser address bars implements. First, it does not have an auto-complete feature. Most modern web browsers provide auto-completion and suggestion via the drop down combo box as the user types a web address. This would be a useful feature that would improve the usability and lower the technical barrier to entry for end users.

Additionally, web browser address bars have the ability to parse more complex
input than just simple host names of IP addresses. A user can specify a domain name and port number, various protocol types, and even provide a username and password (however ill-advised that practice might be) on the address bar.

![Example: rmi://dbolick:password@roboserver.cs.wright.edu:1099](Figure 33: URL parsing example)

Additional parameters could be used to specify a particular Context by name, or to specify other preferences such as the use of a particular simulation engine

As noted in the description of the address bar above, the process of connecting to a new server involves typing in the address bar, and then completing the remainder of the connection information in a dialog box. It would, again, improve usability to allow users to short-cut that process by entering more information on the address bar. Additionally, the address bar could be used to specify any number of additional parameters. For example, if a user wanted to connect only to a specific Context on a particular server, they could specify that in the address bar. Or, if a particular server provided multiple simulation engine types, this could be used to specify the engine visualization that a remote user wanted to use.

9.2.3 Modular Agent Builder

The concept of Agent types composed of separate, discrete components was briefly
discussed in the implementation of the simulated Khepera in Chapter 6. To summarize, the KheperaSim2D (the Sim2D engine implementation of a Khepera robotic agent) extends the AgentSim2D interface. This interface declares methods that are necessary to compose an agent type from separate components, each of which must implement the ComponentSim2D interface.

Specifically, the KheperaSim2D's components represent the physically indivisible, modular components that make up the real-world Khepera robotics platform.

![Figure 34: Modular Khepera and its components.](image-url)
Each component performs whatever functions in the simulated environment that it performs in the real-world agent itself. For example, the KheperaBodySim2D, which is the component that represents the base Khepera unit, is used in the Sim2D engine to calculate the movement and rotation of the Khepera, based on its set wheel speeds. The KheperaSensorSim2D calculates its estimated distance and ambient light sensor values based on the relative location of objects and lights in the simulation. The KheperaGripperSim2D senses object presence between its grippers, and also actively moves any gripped object as the gripper component itself moves in the Sim2D world model.

The componentization of a particular Agent type provides several key benefits. First, it encapsulates the functions of a single component, so that it could be modified, or even replaced by another comparable component, easily. Additionally, componentization allows the easy addition or removal of components from an Agent type. However, this composition only occurs in the source code of the KheperaSim2D implementation.

An identified future development is to provide a programmatic way of composing Agents at runtime, and providing a user interface to do so. This feature would provide users with a modular agent builder feature, and would allow them to create new agents, with functioning sensors and effectors, without the need to actually write and compile new classes. The underlying interface and class hierarchy already exists for composition of Agent types, but a new concept for control interfaces and agent controllers will need to be developed to accommodate the fact that componentized, compositionally-created Agent types will have varied control interfaces depending on the components they
possess. Additionally, a new concrete class will need to be created to allow Agent creation by runtime composition of components, and a new implementation for the DrawableSim2D object providing the graphical representation of the componentized agent will need to be developed.

9.2.4 Runtime Module Loading

The final topic for future work is of a larger scope than the preceding features, which are intended for implementation in the existing application. As mentioned in Chapter 6, the WART library's namespace is organized so that new agent types and simulation engines may be added to the library easily, and without conflict. This package/namespace design was intended to be used for future development of a runtime class loading system, which would allow developers to add new agent and simulation types without having to modify the library, or any application that uses the library.

Along the same lines, a potential future development is to create an application that provides the same namespace organization for Context types, so that developers can add new Context types without modifying the application, and those Context types can be loaded at runtime.
REFERENCES


