MULTI-AGENT SUPERVISION OF GENERIC ROBOTS

by

David Rosas

Submitted in partial fulfillment of the requirements for the

Degree of Master of Computer Science

Advisors:
Dr. Vincenzo Liberatore
Dr. Wyatt Newman

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May, 2002

SIGNATURE SHEET

Table of Contents

List of Illustrations and Figures

Multi-Agent Supervision of Generic Robots

Abstract by

David Rosas

This paper proposes a new architecture for remote robotic control that utilizes sophisticated programs acting as intermediaries between the robot and the supervisor. These processes are mobile agents and are capable of moving from one computer to another autonomously as resource needs change. Through a predefined set of interfaces and a simple language, the agents may communicate with each other as well as the robot and supervisor. This abstraction between the supervisor and the robot allows the virtual robot to move to different computers with different resources depending on the current task the robot is executing. Additionally, it permits a programmer to add new functionality to the virtual robot that wraps up existent robotic functionality. This allows a supervisor to create and inject new methods that will control the robot without actually reprogramming the robot itself. We believe that this framework provides alternative solutions to many remote robotic control problems and allows for many exciting supervisory control possibilities.

1. Introduction

This thesis describes the theory behind and implementation of robotic supervisory control using peer-to-peer processes. The Internet has become a solid infrastructure upon which distributed applications of all sizes and purposes have been constructed. Remote robotic control is becoming increasingly valuable as tasks are discovered that could ideally be performed by robots without requiring human presence[1] [2]. However, there are two main characteristics that separate this project from past remote robotic control undertakings. First, most remote control architectures are unacceptable for real world applications because of the time delays involved. If a robot was given commands to move to specific locations in n-dimensional space, for instance, and some unexpected occurs, the error must be reported back to the supervisor. Upon receiving notification of the robot's error and current state, the supervisor must first decide how to correct the robot. Often the supervisor will want to make some small change, such as causing the robot to move to some modified set of coordinates before continuing along with its job. This feedback must be assembled into a form that the robot can understand and then shipped back across the Internet to the robot. This much delay is often unacceptable, especially when the robot and supervisor have a low-bandwidth or long-distance link. Our project allows "virtual robot" processes to take jobs from the supervisor and command the robot. More importantly, these virtual robots may exist anywhere, such as on a computer that is at a near-distance or high-bandwidth link to the robot. The flexibility that this entails allows us to perform remote, supervisory control on a new class of Internet-capable robots. This class

[1] Bill Adams. [2]
[2] Khurshid Alam, Sudipto Mukherjee. [3]

includes all robots that require quick response times or error correction for their tasks, as long as the errors are somewhat expected and thus may be represented in a fashion understandable to the virtual supervisor process. While the existence and use of these peer-to-peer processes implies many things and grants us many more advantages than just success over some time-delay problems, the second main advantage that this project can tote over traditional attempts is the goal of gaining this control in a generic fashion. By this we mean the following: it is our goal that any robot, as long as that robot falls within a range of our classification, may announce its existence and various details about its abilities to a single, previously written supervisor program which will then be able to control that robot. The details that the supervisor must acquire include tasks that the robot can perform and things it can monitor, as well as basic information about where and how it exists on the Internet. We accomplished this in a way that will require a bare minimum amount of programming on the robot and thus allow a relatively easy way to make any robot that fits our classification compliant with our remote supervisor program. To be able to control any robot regardless of its actual physical shape or its functionality without needing to rewrite the program that does the supervision would be a step forward in remote robotic control. After all, it would provide a standard control mechanism to all robots within our specified class, allowing a familiar set of controls to supervisors and, furthermore, would eliminate the need to write a new supervisor program each time a robot is to be controlled remotely.

Something that must be made clear at this point is that the supervisor program must somehow adapt itself into a form that allows this control to take place in a

manner appropriate for the target robot. It is of little use to have a program that can relay commands to any robot if it is not flexible enough to provide a meaningful interface to the supervisor based on what robot is being controlled. However, the supervisor program itself must not require any additional code or recompilation to achieve this control of any generic robot. If the supervisor is only made to handle robots with a gripper and is unable to change the layout of its GUI, then it does no good to be able to control a mobile pathfinder-like robot, since the supervisor will still only have access to that gripper control representation on the GUI and nothing more.

The roadmap of this thesis is as follows. First, background information will be given and a short description of the foundation upon which this research lies. This includes topics like remote control, supervisory control, and Natural Admittance Control as well as a short discussion of current trends and practices. After this background has been described, all of the third chapter will be devoted to describing several goals for an ideal system for remote, supervisory control of robots. The rest of the thesis paper is devoted to attempting to achieve those goals. As each goal is discussed and a solution proposed, the overall control architecture is updated to reflect our attempt to satisfy this goal. This control architecture starts off assuming a strict master/slave model, but eventually we are forced to change it in order to achieve our objectives. This need is discussed in chapter five, and leads to our final control architecture, which we then illustrate as achieving the remaining goals. This architecture is the primary result of our research and is the crux of the thesis.

2. Background And Basis For Work

In this chapter, the foundation of our work is discussed. It begins with descriptions of a few basic concepts and terms and ends with a description of traditional robotic control and the biggest problems it faces. It is hoped that this chapter will describe the current state of remote robotic control, along with its strengths and weaknesses, and set the stage for outlining our objectives.

2.1 Remote Control

The goal of refining remote robotic supervision is nothing new, with attempts increasingly focused on Internet control. Internet control has proven itself to be an excellent medium for remote control because there exist standard Internet protocols which have been utilized by many robots and robotic controllers, both at the hardware and software levels, and also because it provides us with much of the lower level implementation that sending messages in a time-efficient manner require[3]. Previous related projects at Case Western Reserve University include some of the earliest network-based teleoperation with reflexive collision avoidance[4], a prototype robot that sorts laundry under the supervision of a remote homeowner, an industrial robotic arm that sorts items with the assistance of a remote engineer, and the support for the reprogramming of advanced production line robots from a remote laptop or PDA.

[3] Soraya Ghiasi [12]
[4] Ittichote Chuckpaiwong []

2.2 Supervisory Control

Aside from reusing concepts surrounding remote control, we have also geared our project towards what has become known as robotic "supervision". A complete definition and explanation of this term will be provided within the main section of the thesis. In short, robotic supervision implies that the robot is somewhat intelligent and performs and completes most tasks with a large degree of autonomy, often requiring input from the supervisor only to receive high-level objectives or to escape from an undesirable state in which it finds itself. We have based our decision to use supervisory control on the benefits it affords us relative to the weaknesses of IP communication. For example, at CWRU we have demonstrated an exploratory case of remote supervision using a robot that sorts laundry into whites and colors[5]. The robot (Rhino) is composed of one arm with a gripper (to pick up and put down the clothing) and two cameras (to observe the clothing). The robot uses pictures from the cameras to classify a grasped clothing item as white or color and then drops it in the appropriate basket. The robot usually proceeds autonomously, but there are cases in which the robot is unable to make a clear distinction about the color (or lack thereof) of a piece of clothing. When this case occurs, the robot contacts a remote human, who can assess the state of the robot's job through a Web browser that displays pictures of the environment. The human supervisor is then able to instruct the robot to perform diagnostic or functional actions, which allow the robot to return to productive work. This philosophy of supervisory control applies to many remote control robotics situations and we have found it to be an effective excellent pattern to build into our control architecture.

[5] Rhino

2.3 Natural Admittance Control and Virtual Attractors

Natural admittance control (NAC) is control based upon a set of parameters that prescribe desirable admittance dynamics that the robot should emulate[6][7]. Robot control is achieved by setting these parameters to application-dependent values, which are then used by the NAC controller local to the robot. For example, washing a window, the robot would be instructed to behave relatively stiffly in directions tangent to the glass and compliant normal to the glass. To open a door, the robot should behave relatively stiffly in the expected direction of motion, but should comply gently in directions corresponding with hinge constraints. The NAC controller technology is sufficiently developed that it can be used as a foundation for remote controls.

Within NAC, a "virtual attractor" is a point in n-dimensional space to which the robot end-effector is virtually connected by a means of a set of fictitious springs and dampers of specified stiffnesses and damping. Through this use of virtual springs and dampers, the robot is given goal points upon which it is encouraged to converge. If there is an environmental force in opposition to this convergence (perhaps from contacting kinematic constraint surfaces), there will be position and/or orientation errors between the attractor and the robot, resulting in stretch of the virtual springs. As a result, virtual forces are produced. These virtual forces are part of the model reference dynamics, and thus the robot will equilibrate in contact with the environment with interaction forces equal and opposite to the virtual forces. By this means, one can both visualize and produce desirable interaction dynamics.

---

[6] **Brian Mathewson**
[7] **Nirut Naksuk**

recent experiments at CWRU, the value of virtual dynamics to implement strategic behaviors has been demonstrated with respect to assembly of automotive transmission components. It is proposed here that capability is also highly valuable in accomplishing Internet based collaboration between humans and robots.

2.4 Traditional Robotic Control and Existent Problems

Traditional robotic control involves a myriad of varying components. Some industrial robots are programmed directly while some are controlled through a separate computer to which they must always remain attached. The operating system that controls the robot or its computer controller differs from one manufacturer to the next. Sometimes they are standard and often they are completely proprietary. The engineers who wish to program these robots must learn not only the system calls that correspond to the operating system at hand but also work within the confines of languages that may be compiled or interpreted for that platform. The languages in which robotic controller code is developed also vary greatly, typically being proprietary to the robot manufacturer. With all of these differing factors, it quickly becomes quite difficult to program or control multiple robots from differing manufacturers without a matching set of trained engineers and programs written to be compliant to each system. This is especially so when multiple robots must be coordinated to cooperate in achieving a single objective[8].

Traditional remote robotic control only adds to these problems of complexity. Each robot typically has a static, centralized controller that exposes its own interface with which interaction may take place. Using those interfaces, the remote controller

[8]Terrence Fong, Charles Thorpe, Charles Baur. [10]

may send messages in the language defined by the robot. Various attempts have been made to make a standard language that all robots could recognize and thus simplify the communication process[9] [10] [11]. Even with such attempts, a fully accepted universal language for use in passing messages to robots far from a reality. Part of this problem is the wide range of activities that robots can perform makes the creation of a language that can adequately describe all of them difficult. Trying to remotely control two different robots, then, nearly always implies the need to work with vastly different interfaces that accept messages formed in vastly different languages.

For all of these reasons, traditional remote robotic control solutions have struggled to expand their domain to problems requiring multiple robot coordination. Additionally, it is difficult to augment the existing functionality of the robot, particularly when the owner of the robot who desires these changes is not also the manufacturer of the robot. First, an engineer must be found who understands both the operating system and the language with which the robot was programmed. Depending on how flexible the remote interfaces were programmed and how robust the language that defines the robotic commands is, new interfaces must be added to allow the utilization of any new functionality. Moreover, the robot must have some way of informing any potential controller of its new functionality so that the person controlling the robot will know that this functionality is now available, or else the controller program must be updated and recompiled for usage as well.

---

[9] Michelle Munson, Todd Hodes, Thomas Fischer, Keung Hae Lee, Tobin Lehman, Ben Zhao. [16]
[10] J. Bates, J. Bacon, K. Moody, and M. Spiteri [6]
[11] D. Gelernter. [11]

3. Overview And Rationale of Goals

Now that the foundation for our work has been explained, this chapter lays out the objectives for ideal supervisory robotic control. The goals are meant to describe an architecture that makes the most of the strengths of current systems and perhaps help to supplement some of the weaknesses. Thus, each goal is an attempt either to provide a better solution to one of the currently solved challenges of robotic control or to propose a new feature altogether. Once the goals have been defined, their proposed solutions and implementations may then be discussed in the next chapter.

3.1 Enabling Remote, Generic Supervisory Control

One of our primary goals for an ideal control system is to enable remote, generic supervisory robotic control of a class of robots. By generic control, we mean that control should be possible regardless of the computer, operating system, and language associated with the robot. Clearly, the class of robots with which we wish to enable remote, generic supervisory control must be a class of robots that lends itself to this control. Therefore, there are some basic requirements for robots before they are classified as a match for our control architecture. We first require that the robot be able to be controlled remotely to a satisfactory degree. Most often, this goal will be achieved by using functions that the robot currently implements when the supervisor wishes to execute common, well known tasks, using a virtual attractor and impedance model to drive the robot through unexpected, supervisor defined patterns, or a combination of the two. In addition, there will be some basic operational requirements of the robots. There must be a lowest common denominator for communication that the remote control program may use and with which, therefore, the robots must comply. It is important to note that these operational requirements should not make it difficult for the average robot to be controlled—that is, the bar for the lowest common denominator must be set low enough that it does not eliminate the bulk of robots from compatibility with our architecture. Ideally, these operational requirements should be nearly universally available across many various robotic platforms and programming languages.

The final characteristic that a robot must have in order to be classified as a good match for our system of remote supervisory control is that it lends itself to

supervisory control in the first place. Supervisory control means, in essence, that the robot is given high-level, coarse goals with a high autonomy of control. In supervisory control, the human robot controller acts exactly as such—a supervisor. The supervisor sees the results of the robot's efforts, gives overall objectives and desires for future output, and is able to obtain precise control over the robot when the robot is confused or in an undesirable state. The supervisor is not required to step the robot through each micro operation, nor must he or she be present to watch the robot perform its work. Often this level of job abstraction is achieved through pre-programmed functionality that is present on the robot and which the supervisor simply invokes in an ordered sequence. This results in the robot continuously performing well-known tasks without requiring systematic instructions. This is perfect for remote control scenarios, where often the time delay prohibits any efficient fine-tuned control[12]. Even though supervisory control requires the ability for the robot to complete objectives with only coarse input, it does not eliminate the possibility of low-level control. Occasionally, the supervisor may wish to use low-level, or precision, control in order to correct a supervised robot from an error state or to perform some operation that the robot has not preprogrammed to do. When given complete knowledge about the robot being controlled, the task of giving that robot meaningful, low-level commands in order to manipulate the robot's state is trivial. However, our work has been based around the concept of generic control: Instead of complete information, virtually no knowledge is available to the virtual supervisor. Accordingly, it is our intent to provide this same fine-grain control through a virtual attractor and impedance model, in which the supervisor has a much greater range of

[12] Jeffrey B. Ellis [8]

control over the robot with no pre-programmed knowledge of the overall goal the supervisor has in mind.

It is important to note at this time that while the robot will certainly be programmed with functionality that it may expose to the supervisor, the remote control application that the supervisor uses is meant to be generic. It should be able to apply meaningfully to any robot without any pre-existent knowledge about how the robot works and what it does. This is a noteworthy endeavor—we must create a program that will allow a human supervisor to invoke any range of functionality on a foreign robot that the program previously knew nothing about.

3.2 Generic Robotic Control Application

It is rudimentary that in order to control a robot remotely, there must exist some program that the supervisor will use from a different, remote location, to send commands to the robot. This program will be called the *Virtual Supervisor*, since it is exactly that—a computer program that accepts the supervisor's wishes and relays them to the rest of the system. To the robot, it speaks with the voice and authority of the supervisor himself. Currently, remote robotic control architectures are nearly completely composed of master/slave models. The supervisor uses a program, the virtual supervisor, as the master process to send commands that control one or more robots, which play the roles of slaves. We will be assuming this architecture for the remainder of the analysis and requirements gathering portion of this thesis. It is the nearly universal architecture used for current day remote robotic control applications,

and as such, it is an excellent foundation to stand upon and assess weaknesses and possible modifications.

This virtual supervisor should be, like the robots it hopes to control, generic across operating systems. Regardless of what computer the supervisor himself wishes to use, it would be highly desirable if the virtual supervisor could run on any computer the supervisor desired, regardless of location or operating system. This would give the supervisor a great deal of flexibility in attaining control of a remotely located robot.

3.3 Efficient and High-Level Remote Communication

Our next goal is to utilize as efficient and high-level a model for remote communication as possible. While we recognize the need to pander to the lowest common denominator of robotic functionality, we wish to use protocols that facilitate communication with the robot and the rest of our system in an efficient a fashion as possible. With so many powerful ways to package data, such as Java's RMI (Remote Method Invocation) protocol and the SOAP (Simple Object Access Protocol) standard currently used by many corporations, it is unnecessarily inflexible to continue to program sockets to send text strings from one process to another. Modern communication protocols give programmers a high-level communication standard that allows for easy transmission of a wide range of data types between computers. Additionally, they package the data in a far more efficient manner than sending ASCII text strings across sockets.

3.4 Dynamic GUI Utilizing Intuitive, Modern Day Controls

Just as we are attempting to accomplish communication in a high-level and elegant fashion, the virtual supervisor should be required to adhere to these same standards. The virtual supervisor should be a GUI with the powerful control mechanisms that are associated with modern-day GUI design. The advancement of visual controls has reached a point in software development where the visual representations of commonly requested inputs can be presented intuitively to the average computer user.

This virtual supervisor must be bidirectional. The virtual supervisor must be able to call a function on the robot and then accept a return value from the robot after it has processed the command. Additionally, the robot will have many different state variables with values that it may wish to relay to the supervisor. These state variables are essentially properties of the robot, such as its location and velocity, and the robot's environment, such as external temperature and pictures of its surroundings. Ideally, the supervisor could receive updates of these properties in different fashions, such as polling or upon a change of their value, or upon a change of their value to some definable degree of significance. Regardless, two-way communication is necessary for useful robotic control problems. There are also times when an emergency may occur and the robot may wish to ensure that the human supervisor is contacted in ways that exceed the normal abilities of the virtual supervisor. In these emergency cases, it is desirable for the virtual supervisor to be able to give additional graphic warnings to the supervisor, possibly out of the scope of the virtual supervisor program itself, or even perhaps out of the scope of the computer. For instance, if an

assembly line robot were in a state of severe error and could not continue to complete its work, it could send a message up to the virtual supervisor. The virtual supervisor would immediately request supervisor input, and, should the supervisor not be at the computer to give it, the virtual supervisor might use the Internet to call the supervisor's beeper number and send a message reporting the problem.

One of the biggest issues with making a highly robot-specific GUI within our architecture is that, as was previously stated, the virtual supervisor is to be generic to the point that it has no preprogrammed knowledge about any robot it must control. Thus, instead of making a virtual supervisor that can interact with a single robot that it has been programmed to know about, the virtual supervisor should be a visually intuitive and high-level GUI for generic robots of which it has no foreknowledge. It is a worthwhile effort simply to expose functionality on any robot that the virtual supervisor wishes to control while specifying nothing about how fluid and intuitive the virtual supervisor's control mechanisms must be. However, it is even more challenging and beneficial to produce a GUI that uses modern day graphical controls and provides a user with an intuitive way to control a robot that was foreign to the virtual supervisor at the time of its programming. We must define a way that any robot may inform the virtual supervisor about not only its raw functionality, but also about what visual representation a human being would see as a close parallel to that functionality.

3.5 Single Supervisor, Multiple Robot Control

The architecture of our system, then, must facilitate routing commands from the supervisor to multiple robots, thus allowing for single-supervisor control of those robots. Accordingly, it must be able to route return values and property updates from multiple robots to a single virtual supervisor in a meaningful fashion. There are many cases in which controlling multiple robots from a single interface is beneficial. Many of them are based around scenarios in which the robot participants would have previously required different GUIs in order to permit remote control. In such a scenario, allowing the control of two robots from one GUI is a step forward. More complicated, yet equally desired, scenarios involve coordinating the inputs and outputs of multiple robots not only with one virtual supervisor, but also with each other, thus allowing a network of robots to work collectively even though they are being controlled only in a supervisory fashion.

3.6 Adding Dynamic Functionality to the Robot

Perhaps the most frustrating ramification of the greatly varied operating systems and programming languages that most current robots are built upon is that in order to fix a bug or to add functionality to such a robot, an engineer who is familiar with the robot's platform and code base must be found. Many times this is frustrating to industries that buy a robot and then find out later that it requires some maintenance or an upgrade in a language or on an operating system with which none of their employees are familiar. Since the interfaces for the current functionality must be discovered and stored by the virtual supervisor during the time of its control, it seems

feasible that the virtual supervisor could somehow be able to expose new functionality to the supervisor based off what it knows currently exists on the robot. In other words, it would be a desirable feature if the virtual supervisor could offer the supervisor the ability to perform actions on the robot that the robot was not originally programmed to perform. This translates into the ability to dynamically add functionality to robots by using our knowledge of the robot's preprogrammed abilities.

Currently, our goals permit the expected case: If the robot has any functionality it wishes to expose at all, the virtual supervisor currently will learn of this functionality and expose its own controls to invoke the robot's functionality. Should this new goal be achieved, then the following scenario could play out: The supervisor, using the virtual supervisor program, connects to some robot. Suppose this robot exposes two functions, DrawLine() and Rotate(), which draw a line on a piece of paper and rotate the robotic hand appropriately. The robot informs the virtual supervisor of its functionality, functions DrawLine() and Rotate(). The virtual supervisor then these functions along with a way for the supervisor to access them, passing in whatever parameters are necessary (using some graphical representation per the previous requirement of a high-level GUI). Now, the supervisor wishes to add some further functionality to the robot based off the robot's current abilities. In the simplest case this would involve perhaps making some function DrawSquare (), which called DrawLine() and Rotate() four times. We now propose that instead of reprogramming the robot using whatever robot-specific language and OS calls necessary, the supervisor instead reprogram the virtual supervisor to expose a

function DrawLine() that calls the robotic functions in the proper order. The virtual supervisor, after all, has complete access to the robot and knows about both DrawLine() and Rotate(). It is trivial for the virtual supervisor to make calls to these robotic functions. After the virtual supervisor has been programmed to make those calls and expose it as a DrawSquare() command, this new command is exposed to the supervisor as if it existed on the robot. In fact, since a supervisor who is not familiar with the robot will gain all of his or her information about the robot from the virtual supervisor, it would be impossible to tell what functions existed on the robot and which ones were programmed into the virtual supervisor based on previously existent functionality. After this method creation had taken place, the supervisor could invoke his newly created DrawSquare() function on the virtual supervisor, which would, in turn, invoke the DrawLine() and Rotate() functionality on the existent robot.

Clearly, this goal is easily accomplished should the virtual supervisor program's source code be updated and then recompiled by the supervisor wishing to add features to the robot, but this approach has three drawbacks. First, should a functionality tweak be necessary while the robot is operating, the virtual supervisor (and thus, presumably the robots) would need to come to a safe state for shutdown and then proceed to shutdown before the virtual supervisor program could be closed. After closure, a newly tweaked version that had been compiled could be run and connect to the robots to begin controlling them anew. In short, the changes could not be applied while the system was running. Secondly, it breaks the idea of a generic virtual supervisor that may control any robot. Granted, the challenge of loading up functionality dynamically is a separate and arguably more ambitious goal than the

previous requirements, but to require a virtual supervisor that has a permanently different binary from other virtual supervisors that are in use is extremely undesirable. The aim of this project is to make one virtual supervisor program that can be used for any robot. To require a recompilation into a different virtual supervisor program leads to different versions of the virtual supervisor that are compatible with different robots. This is not acceptable. Customization clearly needs to take place on a per-robot level, but statically modifying the current generic product is not an attractive solution—adding onto it with a separate component would be much preferred.

The third and final drawback of recompiling the virtual supervisor each time one wishes to add robotic functionality is the need for supervisor knowledge of the virtual supervisor code. Recompilation would require the supervisor to not only understand how the robot they wish to reprogram functions and what interfaces it currently provides, but would demand moderate to high knowledge of the inner workings of the virtual supervisor application as well. Such knowledge should not be required of the supervisor, who merely wishes to deal with the robot and have as little contact with the inner workings of the virtual supervisor as possible. It is arguable that requiring an engineer to learn the non-changing architecture and interfaces of the virtual supervisor application would still be significantly easier than learning new languages and OS calls each time a different robot was in need of additional programming, but ideally, neither should be necessary.

We believe that this feature should be especially attractive to supervisors in the industry who would much prefer simply to upload new code to transform their

robot version 1.0 to robot version 2.0 without shutting the robot down or installing new hardware. This kind of patching has certainly become the standard in the software world, where the shipped product is constantly being updated, patched, and exposing new functionality that was simply not quite ready when the time came to ship. Dynamically loading code to fulfill maintenance or update demands on remote robotic controller applications is a fair application of this model.

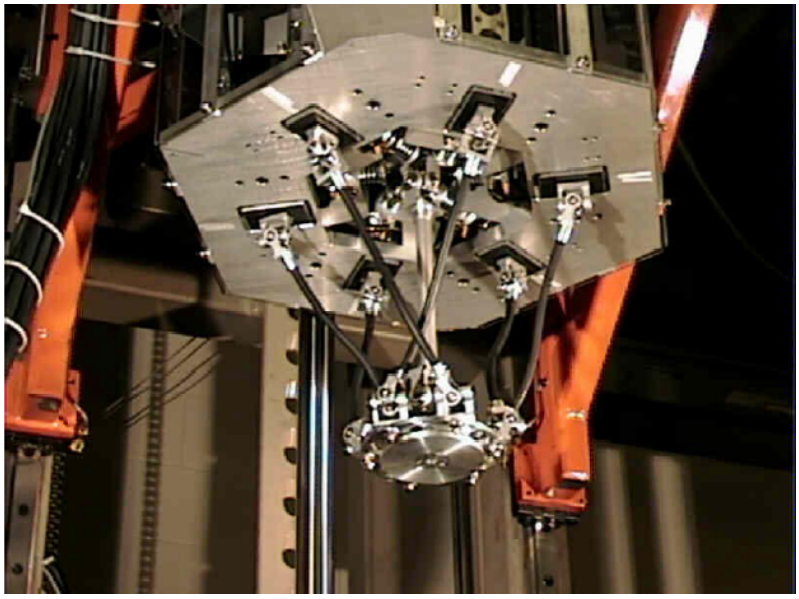
As a final point on this requirement, it is important to note that when the virtual supervisor may be required to control multiple robots it will contain functionality from more than one robot. Should the supervisor wish to create a single new method that wraps up functionality on multiple robots, this should be entirely feasible. By achieving this, the supervisor would achieve something that would have been significantly more difficult if he or she was programming at the robot level instead of the virtual supervisor level—multiple robot coordination. At the robot level, this involves setting up new communication protocols that do translation between the command languages of the two robots and then having a new communication link (robot to robot) in existence during execution. By programming at the virtual supervisor level, the robots simply use the pre-existent communication links and languages to communicate with a coordinator—the virtual supervisor process itself. To the supervisor, this coordination still appears to be wrapped up into one function call. Thus, he or she may execute a function call to achieve some objective and without being aware (or needing to be aware), multiple robots may all play a part in achieving the requested goal.

3.7 Enabling Fine-Grain control

The final requirement that we wish to achieve is to allow for differing levels of robotic control. Depending on what task is requested of some robot, the kind of resources required by the robot to complete their objective changes drastically. Network distance as well as computation cycles required are two excellent examples of metrics whose importance changes drastically depending on the type of job requested of the robot. Thus far, the requirements have been assuming a rigid master/slave model, wherein the virtual supervisor talks directly to the robot responsible for carrying out his or her commands. Assuming this model, the control flow is static: regardless of what job is being requested of the robot, the supervisor begins by manipulating the GUI on the computer that is running the virtual supervisor. Those commands are then sent to the robot, which executes them and sends feedback to the virtual supervisor. Often the response times are unacceptable[13]. The desire to take advantage of the specific resource requirements of a remotely controlled task is widely held[14].

At this time, it is helpful to introduce a practical example. Therefore, we will introduce a robot that will serve as such. It is the Paradex robot, and may be seen below, in figure 3.1.

[13] Ken Taylor, Barney Dalton, Australian National University. [21]
[14] Rajrup Banerjee, Amitabha Mukerjee. [5]



3.1 The Paradex robot

The Paradex is one of the robots at Case Western Reserve University and has several arms that reach down to a central tool disc. There is nothing about the Paradex robot that makes it especially well or poorly suited for our architecture. It does have the advantage of having force sensors located inside each of the six links from its base to its tool disc, and with that an enhanced ability to sense and "feel" its way around the environment. This makes it an excellent candidate for NAC and thus supervisory control, as was discussed in the introduction. However, there is nothing about the Paradex, either physically or programmatically, that makes it special case for control within our architecture. Consider the following case. The Paradex is set up in an environment in which it may move its tool disc about fairly freely. The Paradex exposes only a single interface that allows the virtual supervisor to send it Cartesian coordinates, to which the Paradex will move whatever tool is attached to its tool disc. The virtual supervisor contacts the robot and immediately gives the supervisor the option of using the robot's interface and sending a destination point to the robot.

Additionally, as per the dynamic loading of functionality requirement above, a method that allows the supervisor the option to specify a complex path for the Paradex's tool to follow has been added to the virtual supervisor. How the supervisor defines the path is irrelevant, it is only necessary to stipulate that the supervisor has the ability to meaningfully describe to the virtual supervisor the exact path it wishes the Paradex's tool to follow, perhaps through a sketch pad control. Assume that the virtual supervisor's newly added operation may be broken down into two stages. In the first, the virtual supervisor, with the image of the path the supervisor specified in memory, calculates the actual coordinates that the Paradex must follow. In the second stage, the virtual supervisor actually commands the Paradex to move its tool to each point and thus travel the path that the supervisor specified.

Assuming that the input the virtual supervisor receives from the user is in any significant abstraction layer above actual coordinates (a fair assumption), the first part of this process is very computationally intensive in comparison with an idle state or with the second stage. The virtual supervisor must deduce a series of actual coordinates from the relatively complex representation of the supervisor's desired path. Since this stage does require a great deal of calculations, the processing power of the computer on which the virtual supervisor exists suddenly becomes important. Conversely, the second stage requires relatively little CPU cycles and a great deal of data transmission speed. When the robot has reached a target point, the next point should come quickly, taking into consideration any error in convergence upon the previous point. Should a serious error occur, such as the Paradex being bumped or some problem occurring in the environment space, the robot needs to receive

commands to resolve the problem as quickly as possible. If the supervisor is in another geographical location and separated by a large network distance, then many seconds may pass while the state goes from the robot to the supervisor, a response intended to correct the current state is given, and that response travels back to the robot. Alternatively, in the normal case when the Paradex has converged upon a target point, it will wait, while the signal is sent to the virtual supervisor and next coordinate is sent back, calibrated to fix any error in the convergence to the first point. Computational ability is now a far less important metric than a short round trip time between the Paradex and the supervisor.

Suppose, then, that there were two computers from which the virtual supervisor could be executed. One of them is on the same local area network (LAN)[15] as the Paradex and as such has a very good network connection to the robot. However, it is an outdated machine and not capable of fast performance. The other computer is located at the supervisor's home in a different geographical location from the Paradex and is on a modem connection to the Internet. While the round trip time between that machine and the robot is large, the computer is very powerful and capable of performing massive amounts of calculations in a relatively small amount of time. If the virtual supervisor is launched from the computer on the robot's LAN, it will perform well in the second stage of operation, but not the first. The opposite is true should the virtual supervisor be launched from the computer at the supervisor's home.

[15] A Local Area Network is a network that connects computers that are close to each other, usually in the same building, linked by a cable.

What we would like is for the remote control architecture to contain some mechanism that will allow not only engaging in this high-level supervisory control from remote locations, but will also enable us to gain fine-grain control over the controlled robot when the situation demands it. In this case, it seems almost as if a dual virtual supervisor solution would be called for—one virtual supervisor that resides on the high performance machine and does the calculations required in stage one and then a second virtual supervisor that takes the results of those calculations and moves the Paradex's tool during stage two. Note that one highly undesirable consequence of the need for running the virtual supervisor on different machines during the course of one supervisory action is that the actual person who is the supervisor must travel to those machines to control the robot and receive feedback. In other words, during a phase that required emphasis on CPU, the supervisor would go to a performance-oriented machine and launch the virtual supervisor. When controlling the robot in a situation where fast feedback was a priority, a different computer would be desirable from which to host the remote control. This is highly undesirable.

4. Goal Achievement and Corresponding Design Architecture

Now that the full requirements for remote robotic control have been specified, we may propose an architecture that we believe will allow us to achieve these goals. Once that architecture has been put into place, some implementation details must be listed and explained to prove that it is indeed a solution to the requirements. For each requirement listed above, the high-level design as well as experimental results and control scenarios will be discussed. Accordingly, the resulting robotic control architecture will be revealed and discussed incrementally, so that each goal solution yields a modified architecture that accommodates that solution.

4.1 Enabling Remote, Generic Supervisory Control

As discussed in the requirements section, enabling remote, generic, supervisory control of robotics requires the ability to interact meaningfully with an unspecified robot regardless of operating system and programming language. This requires a specific communication protocol and language that describes the robotic commands to be put into practice that allows information communication between our generic virtual supervisor and this robot, whose attributes and environment are completely unspecified.

There is no one protocol on any level or in any category that all operating systems utilize or with which all programming languages are compliant. However, a method of communication that will act as the best possible lowest common denominator must be selected. It should be basic enough that the vast majority of systems and configurations that we encounter will be able to support it with minimal difficulty while powerful enough to handle basic Internet communication concerns. TCP/IP[16] sockets are a universal standard for communicating information and are supported under nearly all operating systems, whether Unix or Windows based, hard, soft, or not real time and are an attractive solution for the communications aspect of this requirement. They will allow the programmer a standard communications protocol with which he or she will almost certainly be familiar and which takes care of much of the work required in transmitting data to a foreign host. Sockets are the highest level of communications protocol that we can use to still be compliant with

[16] TCP/IP is Transmission Control Protocol over Internet Protocol. It is the most common Internet transport layer protocol, defined in STD 7, RFC 793.This communications is based on the Internet Protocol as its underlying protocol. TCP is connection-oriented and stream-oriented, and provides for reliable communication

most robots. Something more powerful, such as Java's RMI or even Sun's RPC would not allow compatibility with many robots and, since the protocol must serve as the lowest common denominator for any generic robot, are unacceptable.

Using sockets achieves the generic communication aspect of this requirement. However, to interact with a foreign robot, the virtual supervisor must do more than be able to send data to the robot, it must have a predetermined communication scheme so that the two parties may interpret each other. The attributes of the robot (that is, the properties it contains and the methods it exposes) must be discovered and exposed in such a way that the virtual supervisor may interact with them. This will require a defined language for expressing robotic commands. As was stated in the problem definition, this concept is nothing new to remote robotic control. This language must be simple enough to be utilized by any robot but at the same time complex enough to adequately describe that robot. The simplest language would contain absolutely no words or phrases, and would certainly be usable by all robots while not able to describe any. A very complicated language would allow absolute control down to the rate at which the robot reads signals from the hardware, but would hardly be applicable to the average robot that could be controlled. A middle ground must be found.

The language we chose is text-based, which is again in accordance with the standard communication protocols used in modern remote robotic control. In this language, commands sent to the robot may be one of three things: property requests, method executions, or control changes. In the first two cases, the name of the requesting supervisor is sent to the robot along with the name of the property/function

to execute, and in the case of a function execution, any parameters necessary. The final case involves control changes such as logging onto or off of a robot, or indicating the desire to begin supervisory control. Logging onto a robot is a security measure taken to ensure that persons who are not authorized to control a robot are not able to do so. After logging in and being authorized by the robot, a supervisor will then be able to execute methods and receive properties. Properties may be reported through polling, upon their change, or upon a change of some significance (as specified by the supervisor). Nearly all the outputs and inputs of robots may be classified by methods and properties. Strictly speaking, nearly all software in the modern world is built upon the concept of a class, which contains either member functions or member variables, relating to methods and properties respectively. If all modern day software can be described through these two categories, robotics should be definable through the same methods.

When the robot wishes to report to the supervisor, it can send a property update, a method return value, or a control communication. As in the case of supervisor to robot communication, the first two cases are straightforward. The robot sends all required information to the supervisor to inform him of the method completion or of the state change. In a control communication, the robot transmits messages that confirm or deny a supervisor's rights to control the robot. The robot also uses control communications to define all of its attributes. After a supervisor first establishes the want and the authorization to control the robot they have connected to, the robot then sends the supervisor all the information that defines it.

This information takes two forms, as is obvious by now. They are either a property definition or a method definition.

In the case of a property, the robot is responsible for sending enough details about the property to define it to the virtual supervisor[17]. One of these details that define a property is a notification level that indicates how important it is to notify the supervisor about changes to the property. The notification level will eventually determine how the virtual supervisor will respond to the property notification, ranging from simply displaying a message on the screen to attempting to contact the supervisor's beeper. For instance, when there is a change in ambient temperature, only a passive notification is needed. However, if a piece of hardware malfunctioned that was endangering the robot and its surroundings, much stronger attempts to reach the actual supervisor would be merited.

Method definitions are mostly similar[18] in that the robot must transmit enough information to the virtual supervisor to fully define its methods. Upon receiving this information, it is the virtual supervisor's requirement to display the attributes of the robot in a meaningful fashion to the supervisor and also to map the supervisor's requests into strings compliant with this language that may then be sent to the robot for processing. Using this language, then, any robot should be able to define itself to the supervisor in a way that permits satisfactory control.

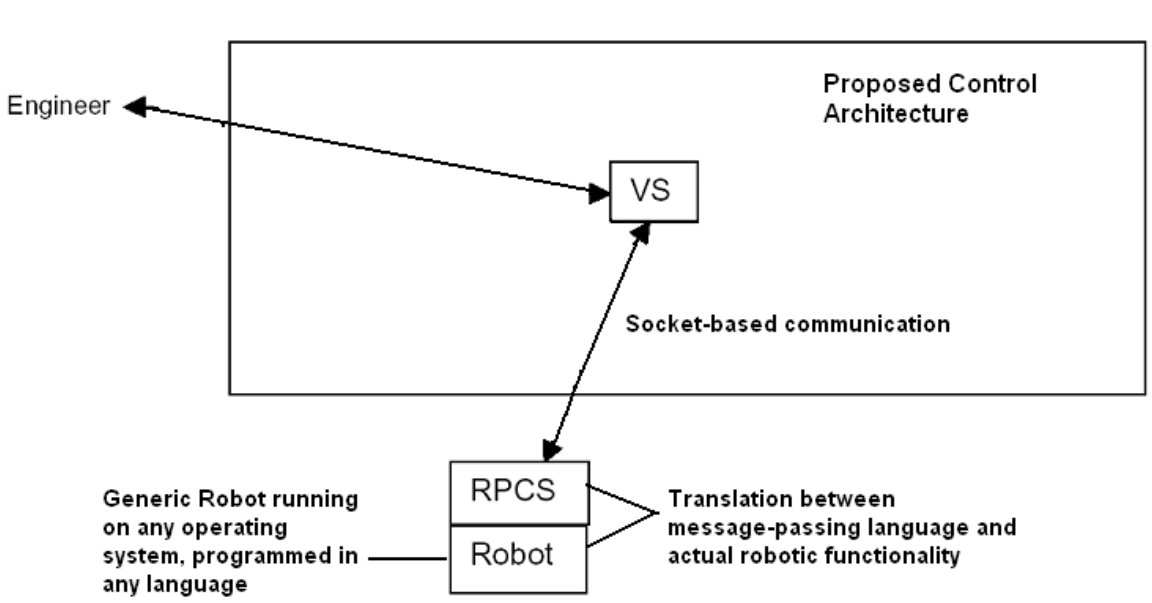
---

[17] Specifically, a property definition contains a dispatch name by which it is to be called, a friendly name that the supervisor will see, a description, also for the supervisor's benefit, the type of data the property represents, whether or not to show the property to the supervisor by default, the initial value of the property, and finally a representation of the notification level that is by default associated with this property.

[18] Specifically, the robot is required to inform the supervisor the dispatch and friendly name of the method, all of the parameters required to execute the method, whether or not to show this method by default, and the type of the return value. Each parameter description must indicate a name, description, type, and default value.

It is important at this point to note the obvious: additional programming is required on the robot before it will be compatible with the architecture. This was expected, however. It is obvious that some code must be written on a per-robot basis; there is no other way each robot can conform to the communication standard being put forth and expose their functionality. The goal is to make this code as simple as possible and to allow each robot to communicate with the virtual supervisor using a language that is powerful enough to fully define the robot. In practice, the owner of the robot who wishes to plug it into this generic robotic control framework must write the necessary code on the robotic controller, sometimes referred to as the *robotic proxy control stub*, or RPCS. This code should be significantly easier to construct than implementing a remote control architecture from scratch. The RPCS resides wherever the bulk of the robotic control code exists; either on robot itself or on the machine attached to the robot that directly controls it. The RPCS is the component of the architecture that is responsible for translating between the methods and properties of the robot itself and the language required by the rest of the control architecture.

With both the VS and the RPCS defined, the initial state of our proposed architecture may now be established. In this architecture, a Virtual Supervisor is used to communicate with a robot through an RPCS using sockets. The supervisor will supervise the robot by interacting with the VS. The robot represents any generic robot, regardless of programming language or operating system, with the RPCS providing the interface to the rest of the architecture. In diagram 4.1, shown below, the current architecture design is displayed.



4.1 Initial State of Proposed Architecture

4.2 Achieving a Generic Remote Control Application

The virtual supervisor itself should be generic across all operating systems and computers. Additionally, nearly all descriptions of the robot's attributes include a value of a specified type. Methods involve parameters of various types as well as return values of a specific type. Property updates have types and values. The ability to interpret the value depends primarily on the ability to recognize the type of that value. A block of information that is calling itself a number should be read much differently than the same block of information calling itself a string. As such, the ability to use a language that contains variant or generic types that may be used to refer to all types within that language would certainly be an advantage. Additionally, a language that emphasizes strong type safety and reflection would be a huge aid.

Type safety of course refers to dealing with variables of different types in a strict fashion, such that when performing operations on multiple different types the programmer must explicitly state any situation in which he or she wishes to convert data types. Additionally, type safe languages generally have built in optimizations when comparing types. Reflection is the ability of a programming language to investigate the type of a variable at run-time and allow the programmer access to that information. Thus, the programmer may deal with different types of data in different ways, even if these different data types were obtained through a pointer to a base class that they share. If a language that met these requirements were found, then we would not have to store the type information of each value as it was specified by the robot. We could instead use generics to represent all values and then use reflection to determine what the value is and to display it appropriately. The two mainstream languages at this time that best fit this profile are Java and C# (pronounced C-Sharp). Both are based completely on an object model such that objects may be used to represent all data and then reflection used to determine what data is being stored[19]. We decided to use C# to create the virtual supervisor. First, we used C# because, being a newer language, C# has had the opportunity to learn from Java's mistakes and clearly assess its advantages during usage in the programming world over many years. A second reason for using C# was the universally accepted SOAP protocol for communication, unlike Java's proprietary RMI. The biggest weakness to a C#

---

[19] At this point, a debate is usually raised about Java's adherence to this strict object-oriented behavior. The fact that there exist within Java native types, such as the standard int, that are not represented by objects and are grossly different from the int class found within Java's libraries is troublesome even to the most dedicated Java zealot. Still, Java has been found able to meet strongly object oriented requirements again and again in the past, despite its compromises on various points, and as such is considered a viable lingual option.

implementation is that at this time, a CLR[20] exists only on Windows-based systems, and thus it is at this time not as generic as we hope it shall become as C# ages and expands in popularity.

4.3 Efficient and High-Level Remote Communication

Now that we have defined the existence of a virtual supervisor that is capable of high level remote communication, this objective is more feasible than ever. However, we have already defined the communication protocol utilized by the robots to be text strings sent across normal socket connections. Therefore, in a master/slave model such is currently being assumed, this high-level communication requirement currently has no place. We shall pass over this requirement and return to it at a later point.

4.4 Achieving a Dynamic GUI Utilizing Intuitive, Modern Day Controls

Summarizing this objective, the virtual supervisor GUI should be able to display the attributes of the robot(s) to which it is connected in an intuitive way utilizing the standard graphical controls that are so common in nearly every modern day piece of software. The above architectural statements have revealed C# as the language of implementation for the virtual supervisor. With C# comes a very rich library of graphical controls. Specifically, therefore, the half of this requirement that demands using intuitive, modern day controls equates to nothing more than utilizing the graphical controls found within the domain of C#. The other half of this

---

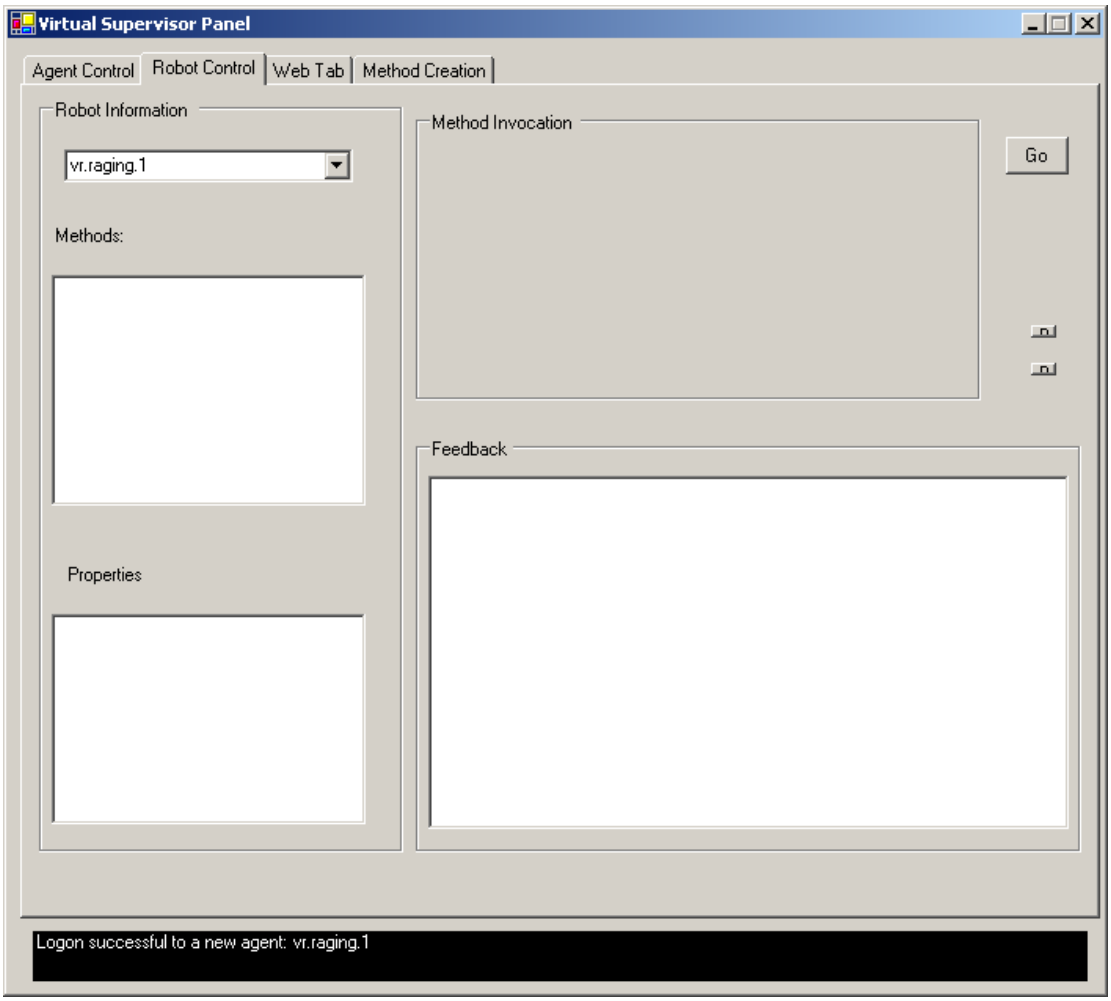
[20] CLR – Common Language Runtime. A (very) coarse equivalent to a Virtual Machine for Java

requirement is dynamically changing the look and feel of the virtual supervisor according to the attributes of the robot to which it connects. This half now equates to creating these C# controls and placing them on the GUI after the virtual supervisor has been exposed to the robot and has had the robots attributes (functions and properties) defined. Once placed on the GUI, the supervisor may then interact with these dynamically loaded controls and thereby control the robot that defined the underlying functionality.

Using C#'s built-in graphical interface library, these objectives are very achievable. When a robot reports a method with some number of parameters and the supervisor indicates to the virtual supervisor that he or she has an interest in executing that method, the virtual supervisor must simply dynamically create controls that are good representations of each parameter's type. The supervisor can then manipulate these newly created controls to reflect the parameter values he wishes to send to the method. However, the area of difficulty has now become clear: how does the virtual supervisor actually map a data type to a control that provides the best representation for that type? When the robot requests input of some specified type, what control should the virtual supervisor create to receive input from the supervisor? One simple response to this question (indeed, the solution sometimes taken by robotic control applications) is to simply accept all input in text. Textboxes are well recognized as established methods of gaining input from a user on Web Pages and computer applications alike. All parameter values must eventually become text strings anyway, when they are to be passed to the robot. While this is a workable solution, it does an

unsatisfactory job of achieving our true objective: to provide intuitive control of the robot using a control that best represents the type of each required value.

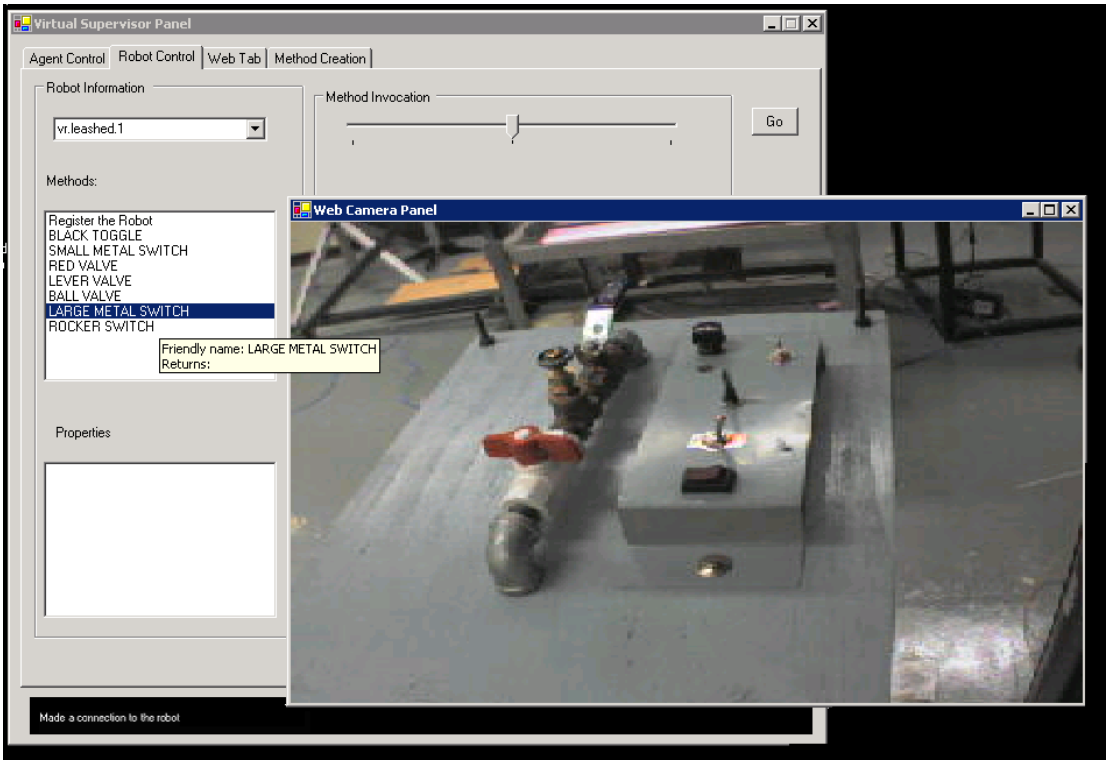
In order to use different controls for different data types, then, we need an effective way to map data types to controls. We wish to be more flexible than assigning one control to one data type, and instead offer the user various controls that map well to the data type he must provide. For instance, if the data type is an integer, then horizontal and vertical sliders may both be options, as would various other controls, and the virtual supervisor would by default display one of the controls in that group. In other words, we need to classify which groups of controls are best able to represent each data type. At present, all basic data types are associated with a class of controls that may be displayed on the virtual supervisor GUI. However, this classification work is ongoing work. Controls and classifications alike are being added to the architecture as users of the virtual supervisor request more controls are used to represent various data types and also as robot methods require more and more complex types as input. Below, in figure 4.3, is the GUI before a robotic connection is made. Note that there are no methods listed, nor are there any parameters displayed on the form. The GUI is a template—it is initially blank and useless and molds itself to fit the profile of whatever robot(s) it is instructed to control.



4.2 The Virtual Supervisor GUI in its blank state; no connections to robots have been made.

The Paradex robot was introduced earlier in this chapter and, since it has an RPCS, is able to receive controlling connections from this architecture. Accordingly, the Virtual Supervisor may be used to make a connection to the Paradex, which will result in the GUI automatically updating to reflect the Paradex's exposed methods. In figure 4.2 below, we see the updated form. Several new methods have appeared, each of which was exposed by the Paradex robot through the RPCS. Most of the methods are related to manipulating the workspace of the Paradex robot, a board with several

controls attached to it. In this figure, a window is shown overtop of the GUI that is a webcam image of the Paradex's workspace. The Parradex robot itself is hanging in the air just above the board. After the supervisor indicates an interest in one of the reorted methods, "Large Metal Switch", the virtual supervisor program updates its GUI. This is the state that Figure 4.4 captures.

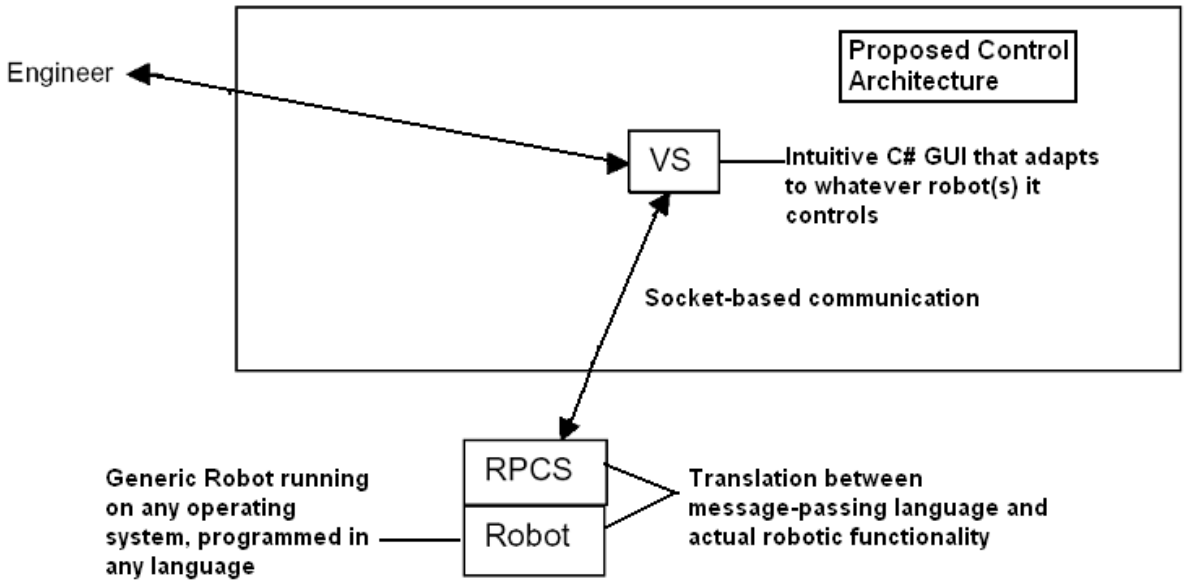


4.3 The Virtual Supervisor after a connection has been made to the Paradex robot

The parameter required to execute the 'Large Metal Switch' method has been graphically displayed in the upper right. It is an input of type 'range'. Range types describe data that must fall within a certain range of values, in this case one of three values (corresponding to three states of the large metal switch, either neutral or switched to the left or right). The virtual supervisor selected its default control to

represent a range input, a horizontal slider. Each of the three stops along the slider indicates a possible input. Upon holding the mouse cursor over the method name, a tool tip has appeared, informing the supervisor of the name and return value of the method (in this case there is no return value). The virtual supervisor will render controls to represent many different types of data and thus mold itself into the form of whatever robot to which it connects.

It may also be noted that there is a box for robotic properties to be selected and monitored, as well as for the feedback from method calls to be displayed. As mentioned above, much of the virtual supervisor is crude and still under development. There is currently a project at CWRU underway to continue to develop this concept of dynamic controls based on type classification. The virtual supervisor is currently the least developed component of our control architecture. Below, figure 4.5 displays the updated diagram of our proposed control architecture. The Virtual Supervisor is now defined to be a visual C# program that adapts to whatever robot(s) it controls, thus fulfilling our requirement for a dynamic GUI.



4.4 The Control Architecture after definition of the VS

5.Achieving Remaining Goals: A Need for Architecture Modification

Many of the goals thus far have been achieved and the control architecture defined and updated per the solutions. However, there is a significant challenge associated with achieving the remainder of the goals from within the standard master/slave architectural model. The desire to add dynamic functionality to the robot through reprogramming the Virtual Supervisor has been discussed in detail, but very little has been mentioned about exactly how to achieve this objective. In short, the goal of allowing supervisors to add functionality to robots dynamically is as follows: Upon evaluating the current functionality of the robot(s) under their control, supervisors may decide that there exists a need for the robot to have additional functionality based on its current functionality. In that case, the supervisor should be able to program a new method, based on existing methods and properties, that may be loaded into the robot for execution. This new method would wrap up the currently existent methods and properties in a supervisor-defined fashion. Since it certainly may not be assumed that the robot itself may have the ability to dynamically load code and expose this ability remotely, this dynamic code loading was to occur inside the virtual supervisor. Since the VS already is aware of all of the robot's functionality and is written in C#, dynamically loading a function based on existent functions is feasible. If the supervisor could continually create new functions that wrapped up existent functionality, the result is very comparable to an object oriented programming model. At each level, functionality is wrapped up by member functions that make use of the functionality from the level below. To illustrate, we return to our DrawSquare() example proposed in the goals chapter. Suppose that a robot initially

had functions DrawLine() and Rotate(). A supervisor who knew nothing about the implementation details of either of these functions could write a new function: DrawSquare(). The DrawSquare() method could simply call the DrawLine() and Rotate() functions in whatever order and frequency and with whatever parameters are required to actually draw a square. Then, the Supervisor could write a DrawDiamond() function that wrapped up Rotate() and DrawSquare(). Furthermore, he could write a function DrawBaseballField() that used DrawSquare() and DrawDiamond() to draw the basic infield diamond and squares for the bases. At each level, the commands are layered into higher and higher levels of abstractions just as the classes in modern day applications repeatedly abstract away lower level functionality.

When applied to the Paradex robot, the supervisor could have the opportunity to write functions like EverythingOn() that moved all switches and buttons to the 'on' position, or ToggleAll(), that viewed the properties of the robot (the current positions of the switches) and then toggled the state of each switch. Using just this one high-level command certainly saves the supervisor the trouble of having to execute each switch's controlling method one by one whenever he or she wishes to effect global change.

Note that at each level, we have a layer that sees only the level below it. Moreover, the commands at the layer below are what are considered the 'robot' functionality, even though the pre-existent robot functionality may have already been abstracted by one or more layers before the current layer is reached. Note that when the layer in question receives commands to execute functionality from the layer

above, it appears as if those commands have come directly from the supervisor. In summary, each level acts as a supervisor to the level below it and a robot to the level above it. Each of these layers has its own drastically different set of functionality and, associated with that, processor and networking requirements associated with it. Each layer quickly takes on both the resource requirements and attributes that are normally associated with an actual robot.

Taking a step away from this requirement for a second, we pause to look at our intention to enable fine-grain control of the robot. Recapping the requirement, we wish to provide a mode in which the process supervising the robot may achieve tight control over the robot when needed or computational power may be provided to that controlling robot when that characteristic is deemed important. Previously, this requirement was determined equal to allowing the process that is actually responsible for controlling the robot to be able to exist at different machines depending on its job or stage of such. Although it is desirable for the process to exist at different machines, it is highly undesirable to require the supervisor to have to physically move from machine to machine as the process did. Ideally, fulfilling this requirement would involve having a process that exists independently of the virtual supervisor that is responsible for carrying out commands for the virtual supervisor but may move to whatever machine best fits its needs. This process could abstract away the control of the virtual supervisor, and report to the robot as a supervisor while reporting to the supervisor as a robot. With full freedom for movement given to this process, the control architecture would be flexible enough to allow for fine-grain control when merited while still allowing for standard supervisory control from a remote location.

Summarizing these two sets of requirements, the following is true: In order to achieve dynamic code loading, the supervisor must be able to write object oriented code fragments which act as robots to both the supervisor and to the code fragments above them and act as supervisors to the robot and code fragments below them. Each of these code fragments has its own task and CPU/latency requirements as well as robot-like functionality and properties. To best fulfill the flexibility required allowing for fine-grain control inside a primarily supervisory control architecture, a different process needs to abstract away control from the Virtual Supervisor. This process, or processes, would act as supervisors to the robot and as a robot to the supervisor, and would be able to move from machine to machine as their requirements change along with the task they are currently executing.

The result of combining these two requirements is clear: there exists a need for a new type of process, a process that exists between the virtual supervisor and the robot. These processes would take on the characteristics of the robot or other processes below them and expose them to the supervisor or other processes above. They must be capable of moving from one computer to another as their resource requirements change. Furthermore, they would provide a very natural host for the code fragments the supervisor would write to add functionality dynamically. Since this functionality is at base a way to abstract the robot functionality from one layer to another, this functionality could be added to the functionality of these processes, since they are already existent in layers between the virtual supervisor and the robot. It follows, then, that each process would encapsulate the functionality of the process below it. This leads to a chain of processes between the supervisor to the robot, each

reporting to the process above and the process below in the chain of command as if they were the supervisor or the robot. These processes will be referred to as v*irtual robots*, or VRs, since that is what they appear to be from the perspective of the supervisor. It is certainly plausible that more than one virtual robot could exist between a virtual supervisor and the robot under its control. These virtual robots are peer-to-peer processes that will compose the very core of our architecture and will allow us to circumvent many of the problems in remote robotic control surrounding master/slave models[21].

5.1 The Case for Mobile Agents

The idea of putting a level of abstraction between two processes in order to achieve a greater degree of flexibility is not novel. In fact, there exists an architectural control model that meets the architectural needs. This model is called the agent model, and it is based around the concept of processes called agents that act in proxy for the user. The term agent is a confusing one, with many different definitions. Griss and Pour while working for HP Labs provide us with one of the earliest definitions for an agent: "A proactive software component that interacts with its environment and other agents as a surrogate for its user". But this definition is very broad. Again, there is a lack of universal agreement on exactly what is required to deem some process as an agent, but generally a process is considered an agent if it displays one or more of the following characteristics:

- Autonomous: acts on user's behalf independently

[21] Giri Nipi, Amitabha Ghosh, K.Sriram. [17]

- Adaptable: may be customized or changed run-time
- Mobile: agent can move around to different machines at different locations
- Collaborative: agents can work together
- Persistent: ability to retain state over time
- Knowledgeable: can reason about its goals and users

Examples of simplistic agents include[22]:

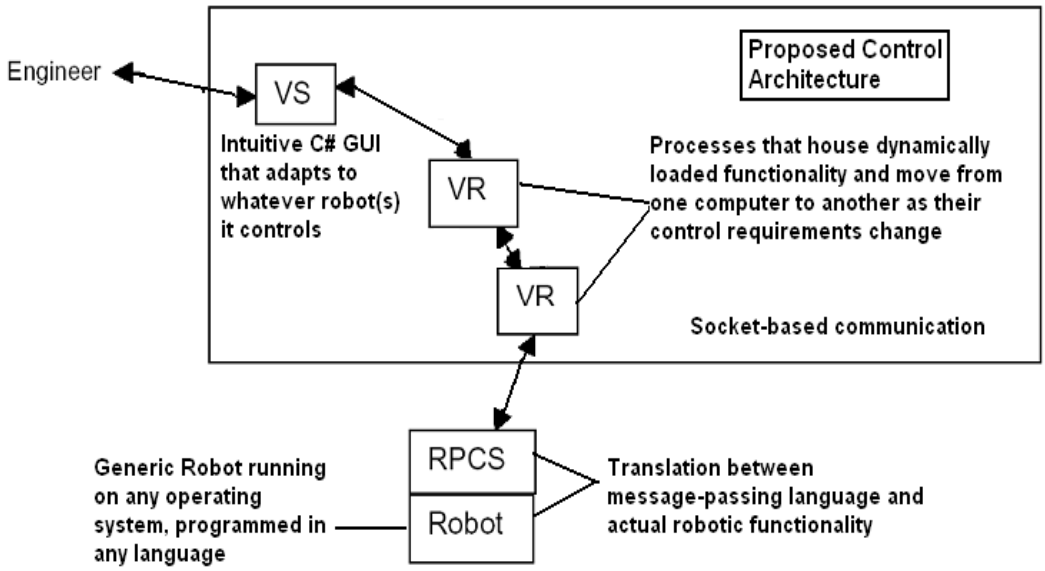
- Shopbots and pricebots, which monitor product availability and price, then negotiate and complete sales of goods and stocks to optimize business-to-business and business-to-consumer interactions.
- Personal agents which interact directly with a user, presenting some personality or character, monitoring and adapting to the users' activities (eg Microsoft Office Assistant)
- Internet spiders that autonomously move from computer to computer, gathering information about the web sites they find there and reporting them to a central data warehouse for access.

The role that the virtual robots fulfill within the current control architecture meet nearly all of the requirements associated with mobile agents, and thus they may be classified as sophisticated mobile agents. The virtual robots are autonomous: they act independently from the supervisor and decide without input when it is beneficial to move from one computer to another. The fact that the VRs will allow the loading of

[22] Wayne Pease. [18]

code at run-time and then take on the functionality exposed by this code makes them highly adaptable. They are mobile by definition; they will move from one computer to another as the task they are charged with executing changes and gives different priorities to various resources. Since the decision about whether or not the VR should move itself to another computer is based upon the resources available at its current location and others, it is knowledgeable about itself and surroundings. VRs are collaborative, since they must transmit information between themselves and also the supervisor and the robots. Currently they are not persistent (once they are destroyed, they will start over in a blank state), but this is the only attribute that agents may have that the VRs do not.

Including these virtual robots as mobile agents in our architecture is the foundation of our proposed control architecture for supervisory control of generic robots. A supervisor will use a virtual supervisor to control a robot. However, the virtual supervisor does not deal directly with the robot. Instead, it communicates its desires to a virtual robot. That fix virtual robot has the ability to move to a different computer when it concludes that the task it has been assigned could be more optimally achieved elsewhere. That virtual robot will express its commands to what looks to be the robot below him. This 'robot' may be the actual robot, or could be another virtual robot posing as one. Should it be a virtual robot, then that process will be independently determining its own resource demands and where it would best be located. If it is the robot itself, then the command from the supervisor will have traversed the full length of the VR chain. Figure 5.1 below shows the control architecture after its biggest change yet—the addition of a layer of virtual robots.



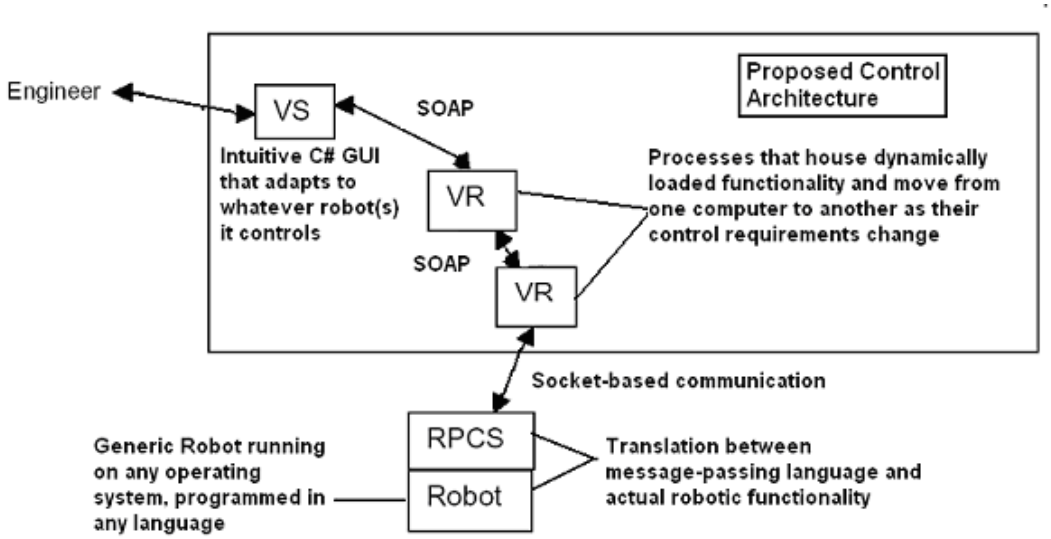
5.1 Control Architecture after introduction of Virtual Robots (VRs)

5.2 Revisiting Efficient and High-Level Remote Communication

The remaining requirements can now be integrated into this revised control architecture. Since we will have virtual robots that will exist between the supervisor and the robot, there is the obvious need for them to communicate. These agents have been written in C# for the same reasons that C# was chosen to implement the virtual supervisor. Since the C# virtual robots must communicate, the previous requirements of high-level, efficient communication whenever possible is immediately applicable. Since the virtual robots are both C# applications, they may make use of the remote communication standard for that language. This communication protocol is, in fact, not something C# or CLR specific, but is a protocol known as SOAP, or Simple Object Access Protocol. SOAP is a high-level communication protocol that is

analogous to Sun's RPC or Java's RMI. SOAP is a universally accepted protocol that came out of the efforts of several major corporations in the software industry. Besides working over IP, SOAP allows any data within the program to be transported to remote instances of classes through a binary or XML-based payload, which produces an efficient packaging of data.

Since we have this high-level communication available to us, it is utilized for communication by the virtual supervisor and virtual robots. However, the robot itself cannot communicate using SOAP; it is required only to be compliant with the lowest common denominator of network communication found on typical robots. Accordingly, the architecture utilizes a communications hybrid model that uses SOAP based communication throughout with the exception of the link between the robot and the virtual robot that communicates with the robot directly. This bottom-most virtual robot is referred to as the 'Base Virtual Robot' or 'Base VR' and must communicate with the robot through sockets. Figure 5.2 below shows the architecture with the communication protocols defined between each process.



5.2 The Control Architecture with the VR and VS links communicating via SOAP

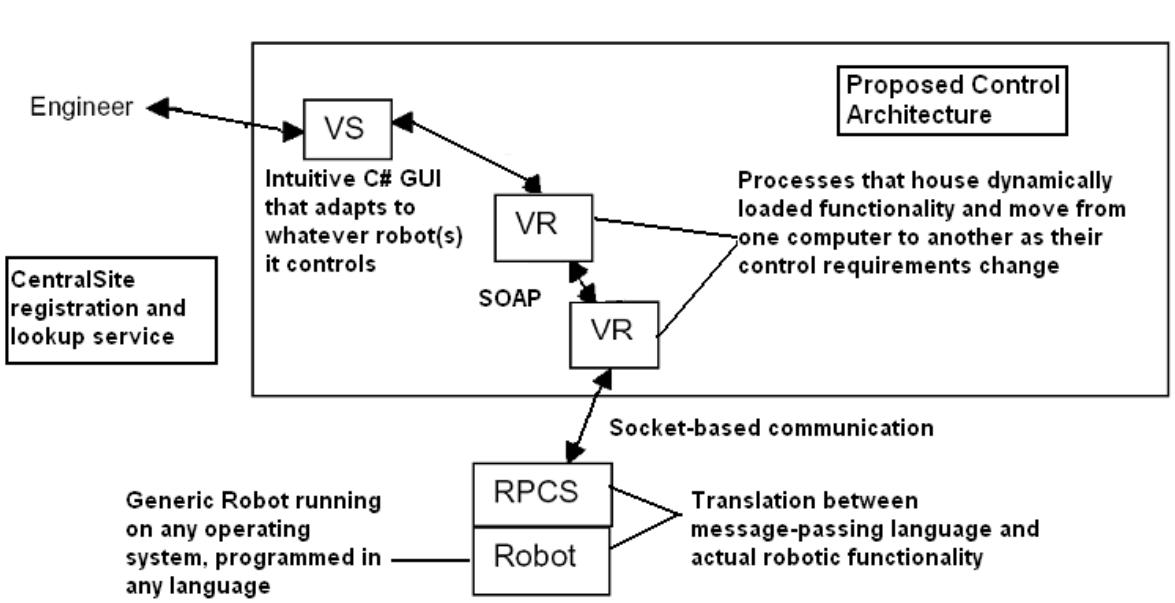
From a technical standpoint, all communication within the architecture is interface driven. Each VR, therefore, has no idea if it is reporting up to another VR or to the VS itself; only that it is an object that inherits from an interface that accepts upstream functionality. The same is true downstream, the virtual supervisor knows nothing about the process to which it passes method execution commands; only that it inherits from an interface that allows for downstream communication. This complete abstraction from the type of process in the chain above or below a process in question (the process' *neighbors*) allows for virtual robots to be added or removed from the chain without disturbance to the rest of the chain. Commands continue to be passed down the chain until a robot receives them, and return values travel up the chain. When a virtual robot becomes mobile and changes its machine of residence, it merely alerts its neighbors of its new IP, they re-establish communication, and the chain

remains intact. If a new virtual robot is spawned and placed just downstream of the virtual supervisor, it notifies its neighbors, and the virtual robot that previously reported to the virtual supervisor now reports to the new virtual robot using the same interface.

5.3 Setting Up the Control Architecture

Since it has been assumed that setup of these processes had already occurred throughout this section, a short time will be spent discussing how that setup occurs. Using the terminology associated with mobile agents, agents may only be constructed on *agencies*, which are willing hosts ready to spawn new agents or accept existent agents that are looking for a new home. The existence of hosts is what excludes these mobile agents from the possibility of being categorized as a virus—the host computer must be willing to accept their presence. Since these agencies are not implicitly known to the supervisor, nor to the virtual supervisor, a registration server must be employed that allows registration and lookup of agencies and currently existent agents by any interested party. This server is called the CentralSite server. Currently, there has been no effort made toward networking the registration servers with each other. In the future, we hope to network each server together so that each CentralSite server will have information about agencies that have been registered with any server. The name CentralSite server was coined to refer to the hope that they would become the center of a site for registration that may talk to other centers of registration sites and freely share registration information. Upon startup, the CentralSite server simply waits for registration or lookups to occur. Upon startup, the Agency process requests

to know the address of a CentralSite server where it may register itself and where it may go to look up other agencies when it is considering a move. It is not necessary to register an agency with a server, but then only supervisors who know about the agency may spawn agents upon it (since they may not look it up), and the agents will not have a server upon which to look up possible move locations. The addition of this lookup server to our architecture is required, and thus figure 5.3 below shows its inclusion.



5.3 The Control Architecture including the CentralSite registration and lookup process.

Below is the agency process, upon start:

```
D:\code\my code\thesis\vr\agency\bin\Debug>agency.exe
Please enter the hostname or IP of the CentralSite machine...
(enter nothing to not use a central site):
raging.cwru.edu
```

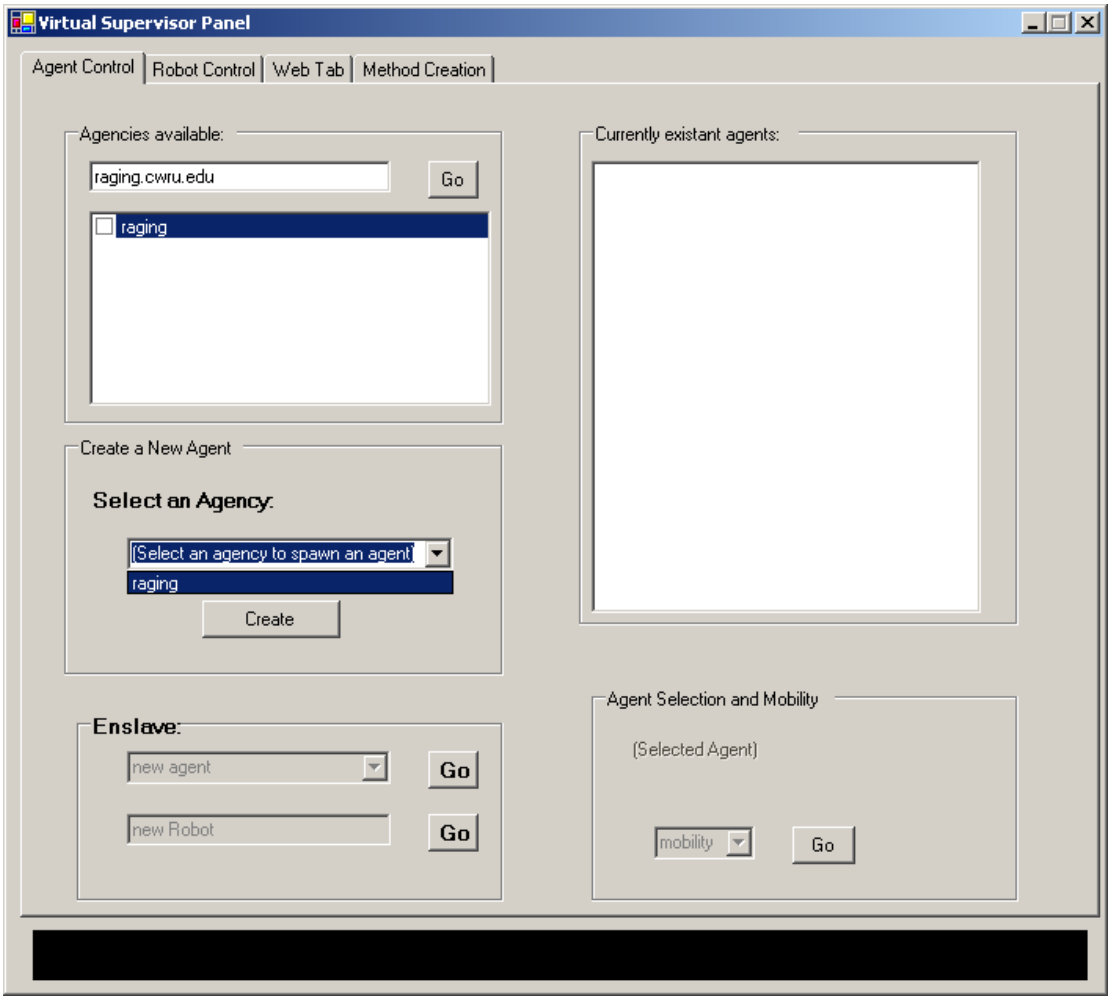
```
Successfully connected to centralsite server and registered this
agency.
Agency activated.  Press enter to quit (and deactivate).
```

And the following is the CentralSite server process, upon start:

```
D:\code\my code\thesis\vr\CentralSite\bin\Debug>CentralSite.exe
Central Site server activated. Press enter to quit (and deactivate).
User: ******* & Pass: ******* added.
1 user added to permissions.
<2:21.14>(an agency @ 'raging'  was added to the lookup service)
```

The registration server requires a user name and password before it allows the agency to register, thus providing a security mechanism against hostile agencies from registering to host agents.

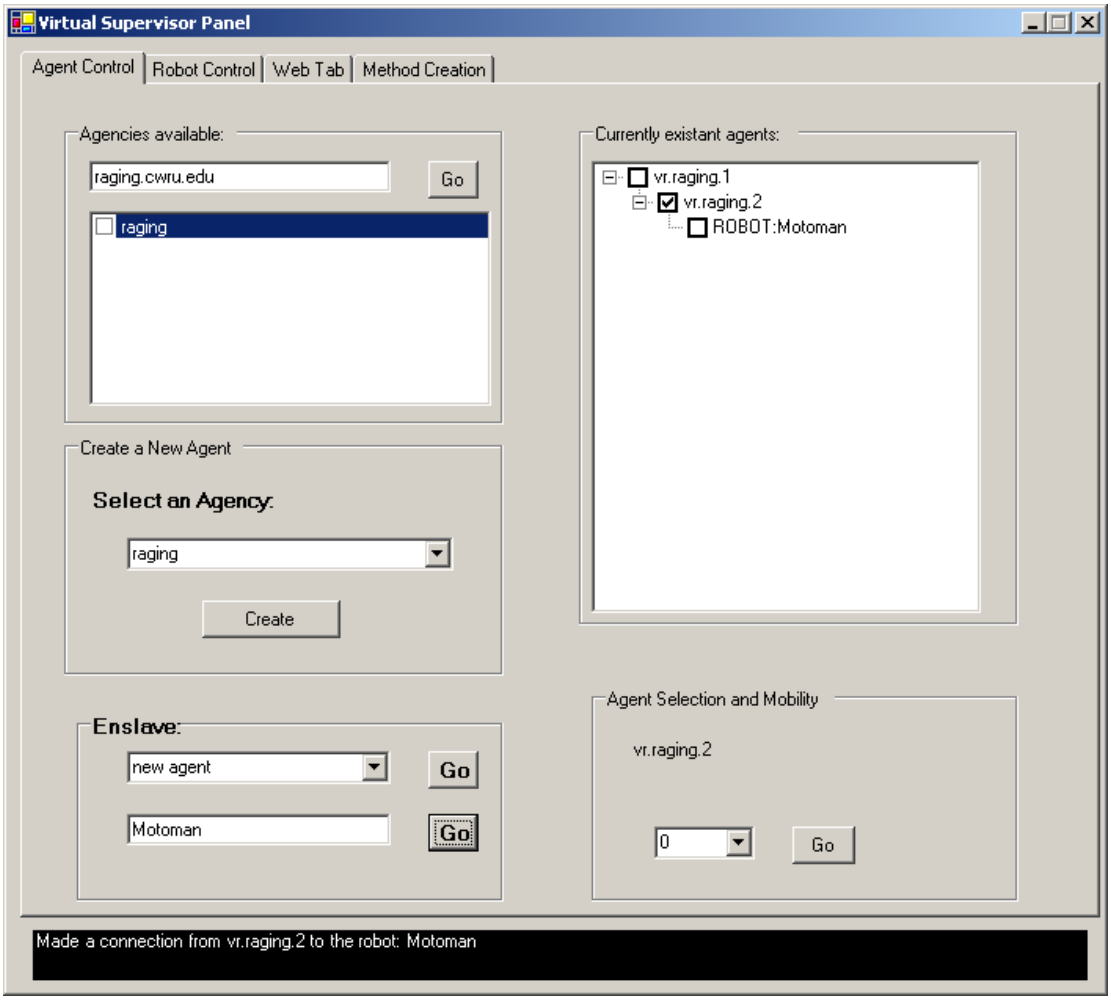
When the Virtual Supervisor program is started, the user is asked to enter the address of a central site server. Upon doing so, a listbox with all registered agencies is populated. In this case, the CentralSite server and the Agency are both running on the same computer, raging.cwru.edu. All of this is shown in figure 5.4 below.



5.4 The CentralSite server shows one agency registered; an agency located at the computer 'raging'.

Once an agency has been located, the supervisor may create an agent upon it. When that agent is created, it may enslave another existent agent, a new agent it can create, or an actual robot. Once successfully enslaved, the robot (virtual or not) is subject to the control of the virtual supervisor. Of course, to enslave another process, the supervisor must first log on, and as was described earlier, the process of logging on involves the transmission and approval of credentials. This means that it is not

possible for any agency or robot to be enslaved against its will. In the example of figure 5.5 below, two VRs have been created on the same computer and then base VR enslaves a robot called ‘MotoMan’:

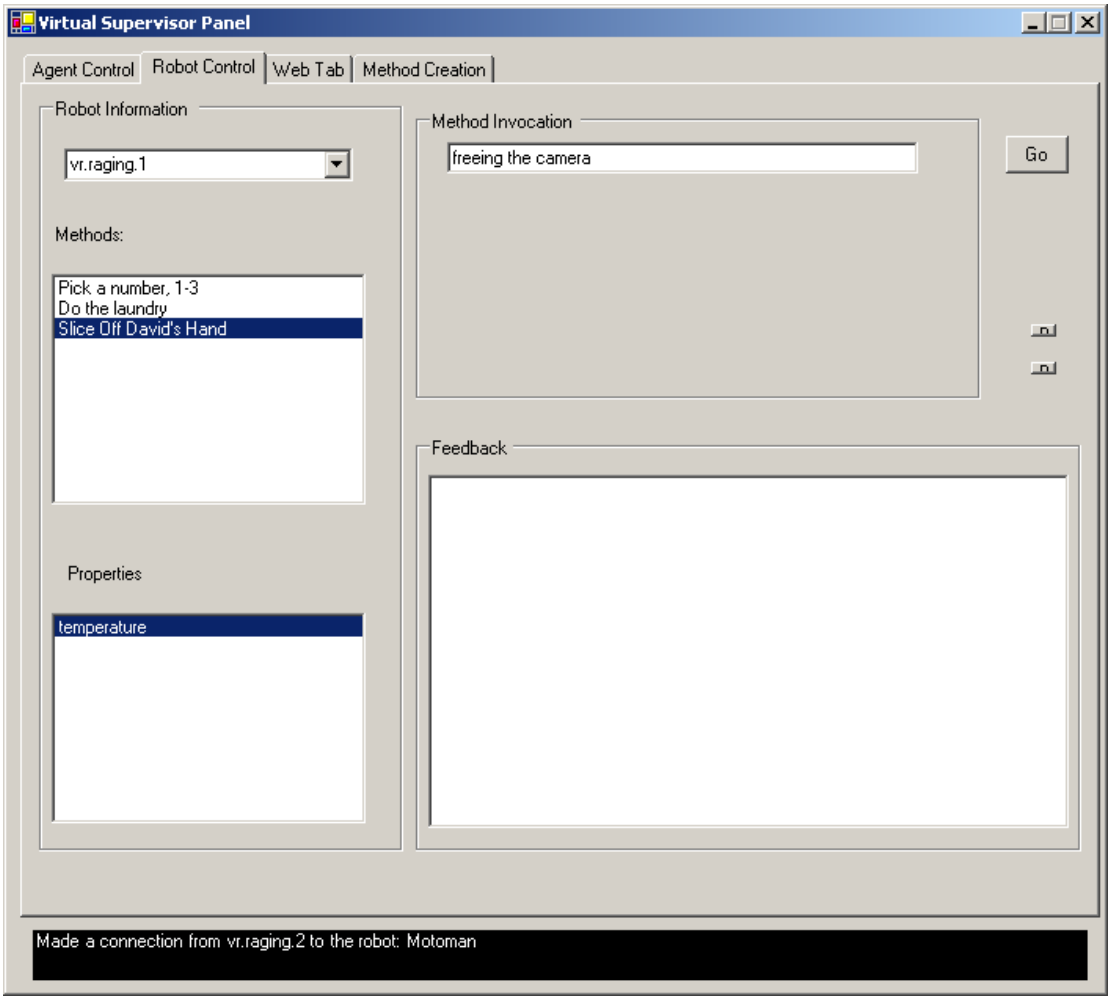


5.5 The Virtual Supervisor showing a chain of two VRs and an actual robot under its control.

The agency's output below shows that the supervisor (whose name is abstracted by the word 'User') created and logged onto the first VR, vr.raging.1. In this experiment, he then asked vr.raging.1 to create a new virtual robot using his credentials, and enslave it. Thus, we see that vr.raging.2 is created by vr.raging.1. Then the supervisor instructs the base VR, vr.raging.2, to connect to a robot, Motoman. That VR reports a connection with the robot, and then the VR upstream from it, vr.raging.1, reports a connection as well (since connections are passed upstream, signifying that if one VR is a master to a robot, then all VRs who are a master to the Base VR are also masters to the robot):

```
(enter nothing to not use a central site):
raging.cwru.edu
Successfully connected to centralsite server and registered this
agency.
Agency activated.  Press enter to quit (and deactivate).
<2:26.7>{vr.raging.1}User has logged onto this VR.
<2:26.11>{vr.raging.2}User has logged onto this VR.
<2:26.11>{vr.raging.1}Successfully created and logged on to a new VR
who is now one of my slaves, named: vr.raging.2
Going to attempt to listen.
<2:26.15>{vr.raging.2}vr.raging.2: reports connection with Motoman
formal name
<2:26.15>{vr.raging.1}vr.raging.1: reports connection with Motoman
formal name
```

This notification and control continues to be passed up to the supervisor himself who now contains the definitions of the attributes for the robot. These attributes include three methods as well as one property, which have all appeared on the Robot Control tab. The control for the parameter to the selected function have been dynamically created, as was discussed during the intuitive GUI portion of the thesis. In figure 5.6 below, this state may be seen from the perspective of the Virtual Supervisor.



5.6 The virtual supervisor after contact has been established with a robot.

At this point, setup has been completed and the supervisor can now command the virtual supervisor to execute functionality on the robot. The virtual supervisor will pass those commands down through each VR until the command propagates to the robot itself. The robot will then execute the command, send the return value (if any) to the base VR, and the value will travel back upstream from there until it reaches the supervisor.

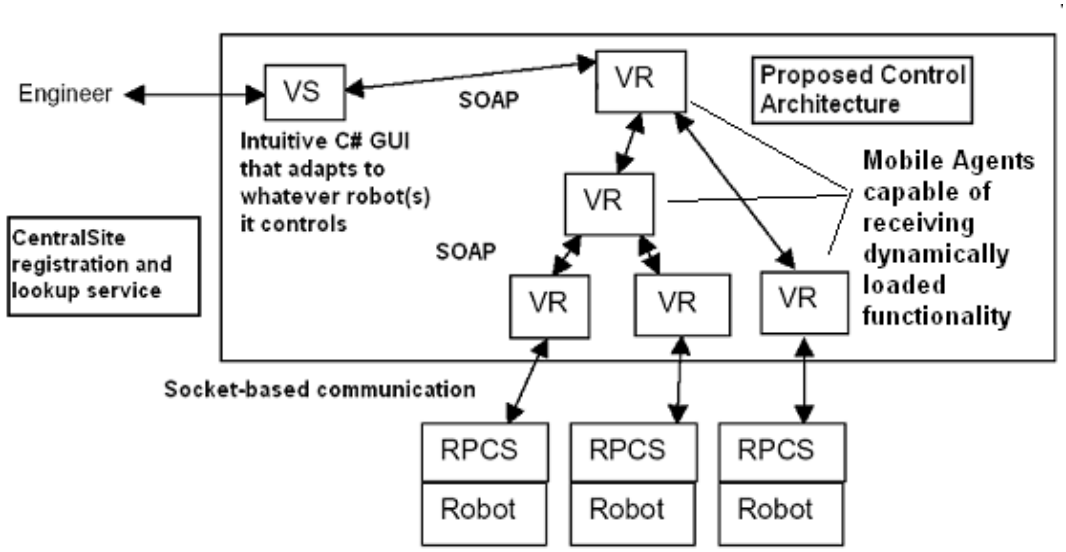
What follows is an example of such. The method was exposed on the robot in reference to near lab mishap famous to some of the team members working on this project. In this mishap, this author's hand was nearly injured while attempting to free a camera from a supposedly unpowered robotic arm that came to life at a dangerous time. The function itself simply returns the text string it was passed. The parameter name requests a location and its value has been entered above as 'freeing the camera'.

```
<2:34.8>{vr.raging.1}Relaying a call to Motoman formal name.Slice
Off David's Hand
<2:34.8>{vr.raging.2}Relaying a call to Motoman formal name.Slice
Off David's Hand
<2:34.8>{vr.raging.2}Got a return value from Motoman formal name for
official_sliceDavidsHand(freeing the camera)
<2:34.8>{vr.raging.1}Got a return value from Motoman formal name for
official_sliceDavidsHand(freeing the camera)
```

5.4 Achieving Single Supervisor, Multiple Robot Control

The goal of achieving single supervisor, multiple robot control has now become fairly easy. Instead of requiring a one-to-one master-to-slave ratio, we allow a master to enslave multiple processes. Thus, the master takes on the functionality of each of its slaves and more than one slave will pass messages upstream through the same VR. We do not allow, however, a slave to have more than one master. It is unnecessary, for there are no reasonable use scenarios in which it is advantageous to allow a virtual robot to report to more than one master. Robots may only be

controlled, after all, by a single supervisor. Elimination of this possibility thus excludes many frustrating problems encountered in peer-to-peer process algorithms, such as cycles. When a single VR is a master to more than one robot (either directly or through other VRs who are enslaved), the functionality from all its slaves is accumulated and exposed to upper VRs and the Supervisor. Thus, our final architectural control diagram has been constructed. It is displayed, in finished form, by figure 5.7 below.



5.7) The final control architecture, complete with Virtual Robots in a 1:Many relationship

5.5 Achieving Fine-Grain control

With the control framework finalized, the remaining requirements may be addressed without much effort. Since the virtual robots are mobile, they can move from one computer to another when their duties change. Now, consider the previous

fine-grain control example in which the Paradex was being commanded to move its tool in a complex pattern. The virtual supervisor accepted a path from the supervisor, and the processes that load functionality dynamically, must first translate the image into a point-by-point path for the robotic arm to travel. While before it was unknown what processes would load this functionality and thus it was assigned to the Virtual Supervisor, we know now this to be the role of the virtual robots. In the second stage, then, a virtual robot must send the target points to the Paradex and monitor closely for problems while accounting for errors in convergence. In one example, the virtual supervisor creates the bare minimum – one virtual robot to accept commands from the virtual supervisor and control the robot. This virtual robot has been loaded with the functionality to calculate the coordinates that the Paradex's tool is to follow (stage one), and knows how to send coordinates to the Paradex and monitor its progress (stage two). Stage one requires a powerful processor, and during stage two, minimizing round trip time between the VR and the robot is of top priority. The actual loading of this functionality into the virtual robots will be discussed in the next section of this thesis. It is only important at this point to note that the VR itself is executing these commands, not sending a request to calculate the coordinates for the path to the Paradex and then waiting for the robot to return the calculations. Were that the case, then the location of the VR would be irrelevant—no matter where it was located, the robot would still be doing all the work.

To start the scenario, the virtual supervisor executes the dynamically loaded function on the VR, passing it some representation of the path for the Paradex's tool to follow. When the VR breaks down this function call into the first stage,

calculating the coordinates, it is aware of its task and notices very quickly that it is a CPU intensive task. The VR does a lookup on the registration server and finds that there are other agencies registered. The VR proceeds to collect information about round trip times from each agency to its slave, the Paradex, as well as the performance capabilities of each agency. Should this agent deduce that it is in its best interests to move to a foreign agency, it will do so. In the first stage, the virtual robot will move to the powerful computer at the supervisor's home and calculate the desired coordinates. During the second stage, the virtual robot will move to the computer on the Paradex's LAN and engage in fine-grain control of the robot, passing it coordinates and monitoring its progress. Thus, without the Virtual Supervisor (and thus the supervisor) moving to any other computers, the control of the Paradex, housed by the virtual robot, moves freely throughout the agencies and is optimized for the task it is engaged in. This kind of mobility has been implemented and tested – successfully— as resource needs change a virtual robot will indeed survey its surroundings and move to the computer best suited for its task.

The most complicated portion of this process is the logic pertaining to the VRs decision to move. In the current implementation, if the agent finds that the available CPU and network location of a foreign agency's computer are better suited for its current task than its current computer, it decides that a move would be advantageous and proceeds. In other words, a greedy algorithm bent on local optimization is utilized in which the VR optimizes its own state. There are several outstanding issues with this method that will be discussed in the Future Work section at the end of the thesis.

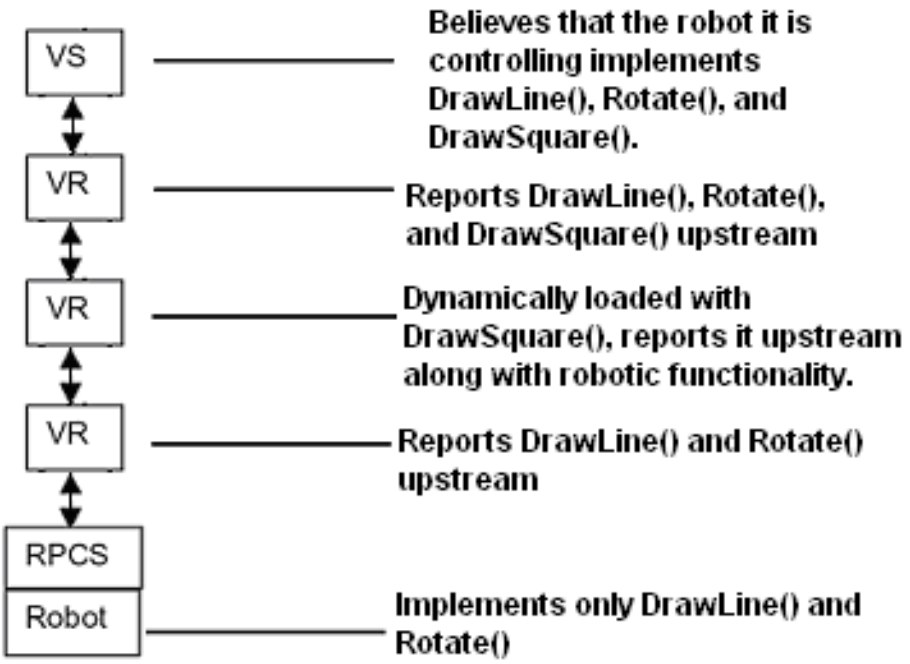
One technical concern about agent mobility is simply coordinating their moves. Although there are many well known distributed algorithms, applying distributed programming techniques to a series of processes that decide autonomously when to move and must be responsible for passing messages both up and down the chain is challenging. Virtual robots need to coordinate their move such that two neighbors may not move at once, or else they would be unable to find each other to report their new location. VRs also may not move when they are currently executing any functionality. If a message is attempting to pass through a VR that is currently moving, the VR must store the message and deliver it to the master or slave process upon completion of its move. The classic distributed algorithm that solves this coordination problem involves setting up a coordinator process that grants or denies the processes the right to move. However, there is the danger of a very slow coordination effort. When the task being executed has a low response time priority, the virtual robots may have moved themselves to powerful computers at large network distances from each other. Should network response time be a priority, it is probable that one virtual robot will move in close proximity to the robot to achieve this fine-grain control while the others remain far apart. Therefore, because our virtual robots are mobile and flexible, there is never a guarantee of a centrally located coordinator process. In fact, the processes will most likely be at a close proximity to their neighbors, but the virtual robots at opposite ends of the control chain may be at large network distances from one another. Therefore, any algorithm that relies too heavily on the virtual supervisor as a coordinator is potentially spending a long time sending packets back and forth and wasting more time than might be gained by the

benefits of a virtual robot's requested move. An algorithm that utilizes minimal supervisor moderation and heavy coordination with neighboring processes was utilized. This is merely a minor technical detail, but noteworthy because of the unusual perspective required when approaching such a classic distributed application coordination problem.

5.6 Achieving the Addition of Dynamic Functionality to the Robot

The only requirement that is not currently in the control model is allowing the supervisor to dynamically load functionality that may apply to the robot. Originally, in a master/slave architecture, the virtual supervisor was required to dynamically load functionality in order to achieve this requirement. However, in our current architecture, the virtual robots are currently abstracting away the control of the robot in layers, acting as intermediaries between the virtual supervisor and the robot. They are also capable of moving from one computer to another as their job requirement changes, and thus efficiently execute code themselves instead of merely passing on instructions to the robot. Because of these two facts, adding dynamic functionality to the virtual robots is the logical progression of their role within the architecture. Thus, it is not the robot that is to be modified, nor the virtual supervisor, but rather the virtual robots that exist between the two. Because the virtual robots do exist as supervisors to robots and virtual robots below them and as robots to any virtual robots or the virtual supervisor above, they provide us a natural medium in which to partition up added functionality that will wrap up existent robot commands and abstract them to a supervisor.

If a robot exposes functions DrawLine() and Rotate() through the architecture's remote control interface, then a virtual robot could be programmed with a code fragment that calls DrawLine() and Rotate() four times and encapsulates it in a function it exposes as DrawSquare(). This VR, then, which previously was responsible only for transmitting the abilities of the robot may now report functionality of its own along with that of the robot. The VR exposes the sum of these functionalities to the process upstream, which eventually reaches the supervisor. The supervisor may now execute the DrawLine(), Rotate(), or DrawSquare() methods, according to what the VR reported. This is illustrated by figure 5.8 below, which lists each process involved in the architecture along with the commands they are aware of. When read top to bottom, it shows how the methods the supervisor is capable of executing may be decomposed into function calls that the robot itself understands.

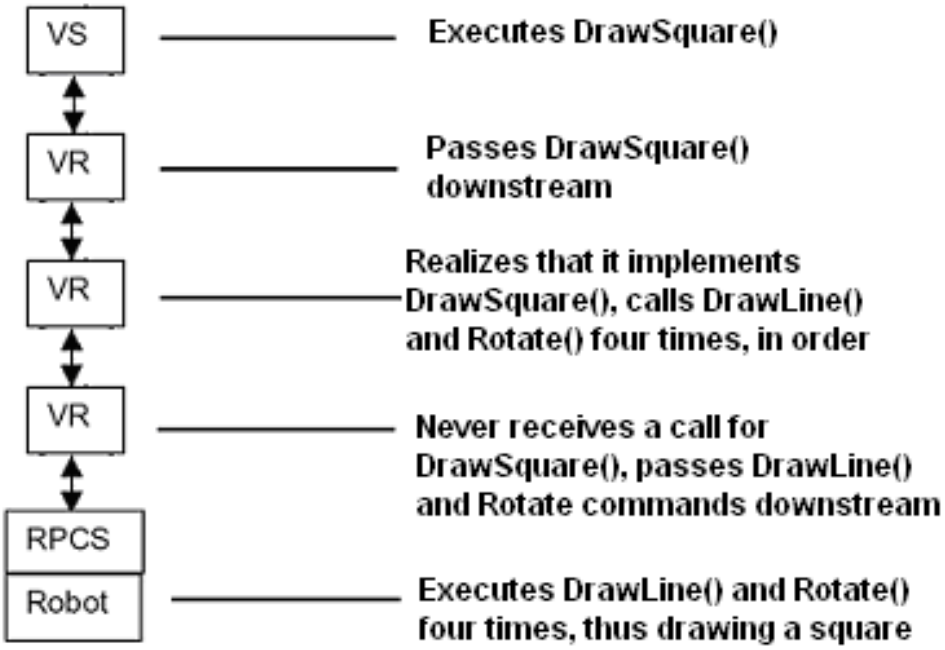


5.8 The methods reported when a VR has dynamically loaded a function. Note that each process believes that the process below it is the actual robot and that the process above is the actual virtual supervisor, thus the DrawLine(), Rotate(), and DrawSquare() commands all appear as actual robotic commands to the VS.

Should the supervisor choose to execute the DrawLine() or Rotate() commands, the normal case commences: the command is issued down the chain until it reaches the robot, who will execute the requested function and then pass the return value (if any) back up the chain to the virtual supervisor. Only the robot itself executes any meaningful code, the virtual robots act only as messengers, passing the execution command down and the return value up. However, should the supervisor choose to execute DrawSquare(), the command that exists on the virtual robot, a slightly different case will commence. The command execution will travel downstream from VR to VR. Each VR between the VR that actually implements

DrawSquare() and the VS is under the impression that the robot actually exposes DrawSquare(). This is natural, since each one of those VRs will believe that the process just downstream is the actual robot, and that 'robot' was the process that exposed and reported the DrawSquare() command. When the DrawSquare() execution command reaches the VR that houses its functionality, that VR calls into its functionality instead of calling the next VR (who, of course, would not know what DrawSquare() was). That functionality, in this example, will call DrawLine() and Rotate() four times on the appropriate slave VR. At this point, then, the VR, controlled by the code it has loaded, will pass these calls on to the process downstream until they reach the robot. Thus, this VR now has a more significant role than merely passing messages to the robots; it is executing meaningful code that is giving its own commands to the robot.

This second, more complicated scenario is illustrated in figure 5.9 below. This figure shows the propagation and decomposition of a Supervisor's command for the robot to DrawSquare().



5.9 The execution of a dynamically loaded function. Note that only the top two VRs are aware of the function DrawSquare (), which was loaded into the middle VR, and thus when the middle VR decodes DrawSquare() into DrawLine() and Rotate() components as the supervisor programmed, the lower VRs and the robot itself will be able to recognize and execute the commands.

As was indicated previously, dividing new functionality among the virtual robots results in different virtual robots each implementing different tasks with different resource requirements. The implications of such are significant. Revisiting the example of cutting shapes out of sheet metal illustrates this point. Previously, a supervisor could dynamically load both stage1 (CPU intensive) and stage2 (priority on network response time) functionality into the same VR. That VR, then, would move to a different machine as it changed stages. Alternatively, the supervisor could

spawn two different VRs between the VS and the Robot. He or she could then load the stage1 functionality into the VR closest to the supervisor and the stage2 functionality into the Base VR. Each VR would detect the resource needs of its functionality and move to the most suitable computer, which would be different for each of the VRs. Even though both VRs are controlling the same robot with only a single command actually exposed (simply move arm), the VRs themselves differ in the requirements of their wrapper functionality and thus optimize their location based on their part of the overall task. This partial specialization of tasks is clearly more efficient than a non-mobile agent that must execute the functionality on a computer that will be well suited for one stage and not the other. It is also more efficient than a mobile agent, which must incur the time overhead of testing out other agencies and coordinating a move to a different computer when its task and thus resource requirements change.

When the virtual robots load further functionality from the supervisor, their responsibilities change from merely command passing to actually executing code that is relevant to the control of one or more robots. Yet because currently the functionality of the robot is exposed and defined through standard interfaces that each virtual robot uses, no change in the virtual agent's interfaces is required after it loads robot-specific functionality under the order of the supervisor. The new methods or properties are simply reported and defined up the stream as if they were coming from the robot itself. Each virtual robot, then, contains knowledge of the actual robotic functionality as well as any dynamic functionality that they have loaded and any functionality loaded into other virtual robots further down the chain.

It is interesting that as functionality is wrapped up along the chain of VRs, the functionality that each VR contains represents that of class inheritance in any modern object oriented language. Expanding the DrawSquare() example even further, suppose the supervisor spawns three VRs named VRa, VRb, and VRc. The supervisor then lines them up in a chain so that the virtual supervisor communicates directly with VRa and VRc is the Base VR. First, the supervisor loads into VRc the functionality for DrawSquare(), which is implemented as calls to DrawLine() and Rotate(). Then the supervisor wishes VRb to implement the DrawDiamond() function, which uses DrawSquare() and Rotate(), both of which are exposed to it by its "robot", VRc. The supervisor may then add a new function, DrawBaseballField () to VRa that calls upon DrawSquare() and DrawDiamond(). VRa still exposes DrawLine(), Rotate(), DrawSquare() and DrawDiamond() to the supervisor, as well as this new function. Note that at each level, the VR implements all of the functionality of the VR downstream as well as any functionality that has been dynamically loaded into itself. This closely mimics the concept of inheritance. The robot acts as a base class, providing a standard set of functionality. Each VR built on top of the robot contains all of the functionality of the downstream process and possibly additional functionality, in parallel to a derived class. Additionally, when the supervisor adds commands to a VR, he or she has the option of hiding previous methods from upstream VRs. So when the supervisor loads the DrawDiamond () function into VRb that wraps up Rotate() and DrawSquare(), he or she has the option to hide DrawSquare() from upstream VRs (and therefore the supervisor as well). In that case, the example would fail because VRa, which is upstream of VRb, could not

call DrawSquare() and thus could not wrap that up into DrawBaseballField(). This mimics private inheritance, where the members of the base class are taken as private members of the derived class and may not be inherited by further derivations.

Also, note that this dynamic method loading takes place at a VR level, and it has been previously established that a VR may control multiple slaves that may control, directly or indirectly, a robot each. This means that when we add new functionality that wraps up existent functionality, we have the ability to wrap up functionality from multiple robots. This parallels multiple inheritance, where the functionality from more than one base class is added to a single derived class. Even more interesting is that some of the typical multiple inheritance problems that object oriented languages contain, such as duplicate method names also appear within our control architecture.

Although the technical implementations of theory are not always required nor desirable, the concept of requiring robotic supervisors to write code that will be added dynamically to an existent process has been treated with some skepticism by some in industry. As a reaction to the hesitation to believe that it is feasible, with little effort, to write code that can interact meaningfully with an existent virtual robot process, some technical detail is merited, and follows. From a technical standpoint, the ability to load code into the VR is quite feasible in a modern, reflexive language like C#. The language supports the ability to inject compiled assemblies into the executing process at any time. In this context, an assembly refers to a compiled collection of classes that form an exe or dll. The classes and member functions within those classes are then accessible to the currently executing program immediately after the

assembly is loaded. There are two main challenges associated with dynamically loading functionality within the context of the current architecture. First, there is the fundamental problem of how this code is to be written and, once written and compiled, how it can be loaded into the virtual robot from a remote point without requiring a shutdown of the robot or a recompilation of the host VR. Secondly, the communication between the loaded assembly and the virtual robot that loads that assembly must be coordinated. The loaded assembly must inform the VR that there is new functionality to expose, how that functionality should be reported, and how the VR can invoke that functionality when a call comes to it from the virtual supervisor.

Since the virtual robots are written in C#, any language written on top of the CLR may be compiled into an assembly and loaded by the process. The supervisor, therefore, must write code in one of those languages, such as C#. While this sounds undesirable, the alternative to reprogramming the virtual robot is reprogramming the robot itself. To add functionality to a virtual robot, the supervisor must write code in any of the dozens of languages on top of the CLR that will interact with the code resident in a VR… code that will remain constant and familiar even though the VR may be controlling vastly different robots. Should the supervisor be required to reprogram the robots when an upgrade is necessary, then supervisor would be forced to learn each robot's multiple interfaces and multiple languages that make calls to whatever proprietary operating system the robot or robotic controller was running. Furthermore, the hardship involved in adding code that will interact with a virtual robot's functionality can be made significantly easier by careful planning by the programmers writing the VR. These simplifications will be shown after the concern

involving VR communication with the loaded assembly is addressed.

The standard approach to interacting with dynamically loaded code is to require the unknown code to implement well-known interfaces that will serve as a contract for communication. This case is no different. When a supervisor wishes to write code to be loaded into an agent as additional functionality, then, he must simply write a class that derives from a known interface. In this case, there is one key function, CreateMethods() that the interface requires of any newly loading classes. The CreateMethods() function returns objects that represent the methods supported by the dll. These objects are similar to the objects that the VR uses internally to represent robot functionality, with one exception. The object contains one member that is not found within a normal method description: a delegate that points to a method contained within the loaded class. Crudely, a delegate is a C# wrapper around a standard function pointer.

When a virtual robot loads up an assembly, it immediately executes the assembly's NewMethods method (which is guaranteed to exist since the loaded assembly must derive from an interface that requires that method). The objects returned, that describe the newly added methods, are stored by the VR. The VR then reports this functionality to the process upstream, either another VR or the VS. This is the standard protocol for exposing static robotic functionality. That is how the virtual supervisor is made aware of the dynamically loaded functionality. When the supervisor chooses to execute the new functionality through the VS, the call travels downstream until it reaches the virtual robot to which the supervisor sent the compiled assembly. This is the same virtual robot that stores the object describing

this functionality. The virtual robot recognizes the command, finds the object describing the command, and routes control to the function that the object specifies with the parameters that the VS sent. This will in turn execute the code inside the assembly, passing it the parameters that have been passed down the stream from the VS.

Below is the code for an assembly eligible to be dynamically loaded. In its CreateMethods function, it creates one new method with a friendly name of “Laundry if David’s close”. It accepts one parameter, of TypeText, with a name “Location” and a description of David’s location. It has no default value. This method is added to an arraylist along with a delegate to a method called DualMethod which is also existent in the loaded class (and will be shown shortly). These methods are returned and every VR above whatever VR loaded this assembly, as well as the VS, will reflect a new method called “Laundry if David’s Close”. Note that the objects that contain all the information necessary to describe this new method are of type DynamicBotMethod, which is what the host VR will store and use to recognize when commands being executed are intended for dynamically loaded code instead of its downstream process.

```
public override ArrayList CreateMethods()
{
   ArrayList newMethods = new ArrayList();
   ArrayList parms = new ArrayList();
   parms.Add(new BotParam(new TypeText(), "Location", "David's
      location.", null));
   DynamicBotMethod dbm = new DynamicBotMethod("dispatch_newDual",
      "Laundry if David's close", parms, true, new TypeBoolean(),
   MyName, new DynamicMethodDelegate(DualMethod));
   newMethods.Add(dbm);
   return newMethods;
}
```

While this analysis reveals the communication feasibility from the virtual robot to the dynamically loaded functionality, the reverse communication is much more challenging. The difficulty arises when the assembly, which was written without any of the virtual robot code to compile against, must be able to call methods that are exposed to the virtual robot. To accomplish this, the writers of the assembly must be provided *something* to compile against that will expose methods allowing module writers to call functions that have been exposed to the current VR. These 'helper functions' will give module writers a way to ensure that they have correctly formatted their calls. Additionally, the helper functions will take away a significant amount of the complexity involved when executing a function. Although in-depth discussion of all the helper functionality existent is beyond the purpose of this thesis, a technical explanation of one should help ease concerns about the feasibility of interfacing with the virtual robots. The main helper function within the base class is RunMethod, which requires a method name to run, a list of parameters to pass that method, and a boolean value indicating whether to continue exposing the functions that have been exposed to the VR from downstream VRs (the equivalent of a switch between public and private inheritance of methods).

```
public object RunMethod(string methodName, ArrayList parameters,
bool keepshowing)
```

The RunMethod function is an excellent example of how a base class helper function can drastically simplify the work that module writers must undergo in order to make their code compatible with the virtual robot architecture. First, RunMethod

(at runtime) retrieves a listing of all functions exposed by the VR that loaded the assembly and ensures that the methodName passed to it by the derived assembly matches to one of the exposed function. If not, then the dynamically loaded assembly was attempting to call functionality that the virtual robot does not know. Secondly, it packages up the information into the exact object that VR the needs to understand in order to execute the method, so the module author does not need to concern himself or herself with it. RunMethod then spins off a new process to handle the command execution. When the command is executed on the robot and the return value is passed back through the VRs, it eventually is passed to the VR that contains the loaded assembly that initiated the command. There is one additional problem: the architecture is set up so that multiple commands may be sent to the robot before any return values are received. That is, the virtual robot does not have to wait for a return value from one function before it can send the next command (which may be an abort). However, the author of the module wishes to have one command executed and its return value stored before the next line is processed. For example, "int a = base.RunMethod(foo,…)" is intended to have foo execute (via sockets) and return value be stored in "a" before the next command is read. In order to make this non-blocking process blocking, RunMethod uses a multithreading algorithm to block until the return value is passed back to the base class. After receiving a return value for foo, the base class allows the thread that called into RunMethod(foo,…) to continue, and passes the thread the return value. This creates the illusion that the execution took place within one thread and allowing a return value to come back to the caller of the function. Without the base class undergoing this functionality, there is no way

that the writer of the module could wait to receive return values from function calls before executing the next line of code.

Using these helper functions, the module author may thus have an easier time writing code to be injected into the VR chain. The helper functions and base classes, in fact, are the means of simplification that erases the difficulty involved for supervisors wishing to write modules to be injected into a VR. The base class does nearly all of the work in integrating the assembly with the system and handling the communication while leaving the supervisor to worry only about writing code related to the desired robotic functionality.

The DualMethod method that corresponds to the example “Laundry if David’s close” example above then, is:

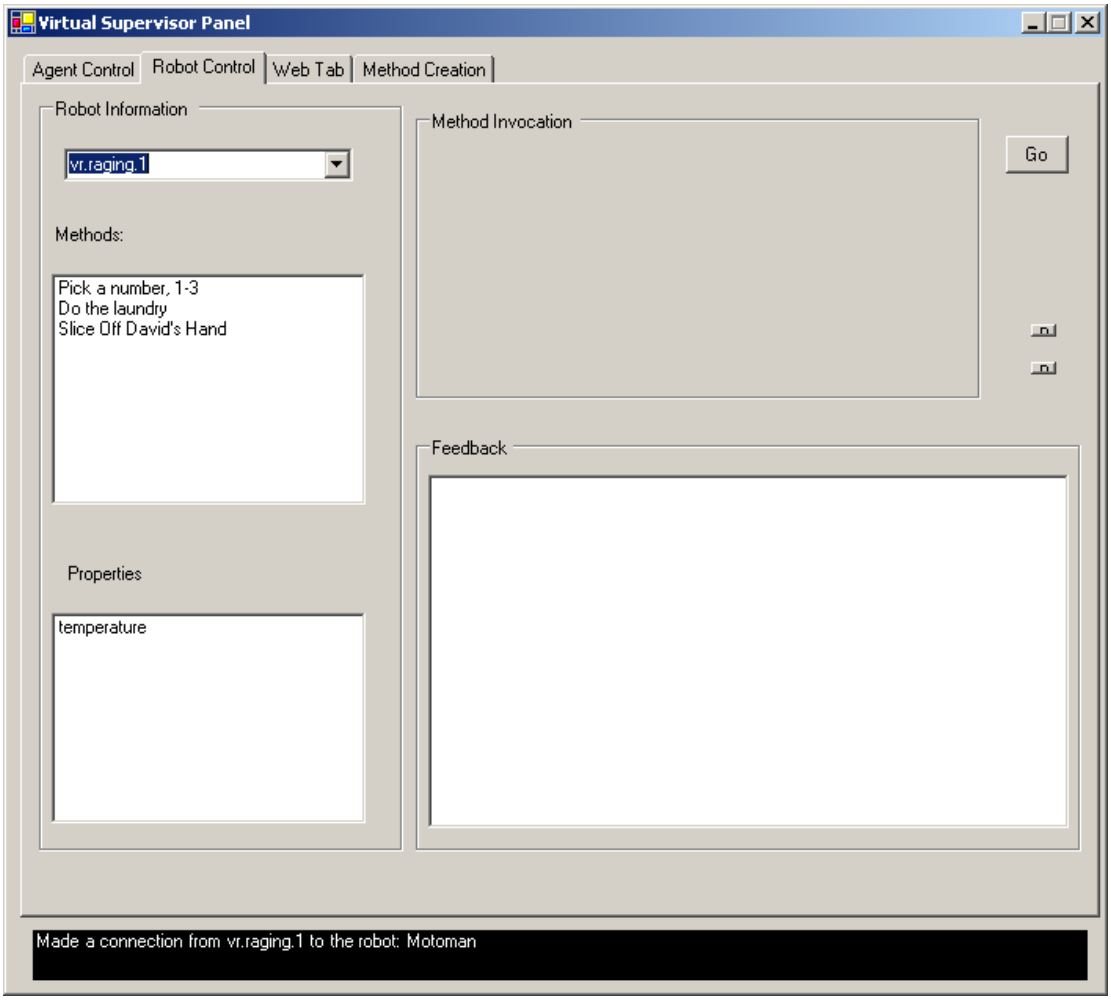
```
private object DualMethod(BotMethod bm)
{
   ArrayList p = new ArrayList();
   object ret = base.RunMethod("Slice Off David's Hand",
bm.parameters, true);
   string sret = ret.ToString();
   if (sret == "laundry machine")
   {
      p.Clear();
      ret = base.RunMethod("Do the laundry", p, true);
      return true;
   }
   return false;
}
```

This example shows that when DualMethod is executed (in response from the supervisor executing the “Laundry if David’s Close” method exposed within CreateMethods), it will pass in the parameters (which were specified in create

methods to be a single text parameter) to an existent function. If the return value from that function is “laundry machine”, then it will run an additional function and return true. If not, the function will return false. The function that it calls happens to return whatever its parameter was passed in to be. So, if “Laundry if David’s Close” is given a parameter of “Laundry Machine”, it should execute two of the robot’s functions, the second one being a “Do Laundry Machine”. If the parameter is not “Laundry Machine”, it will execute only the first method, “Slice off David’s Hand” (as mentioned earlier, a reference to a tense night in the lab).

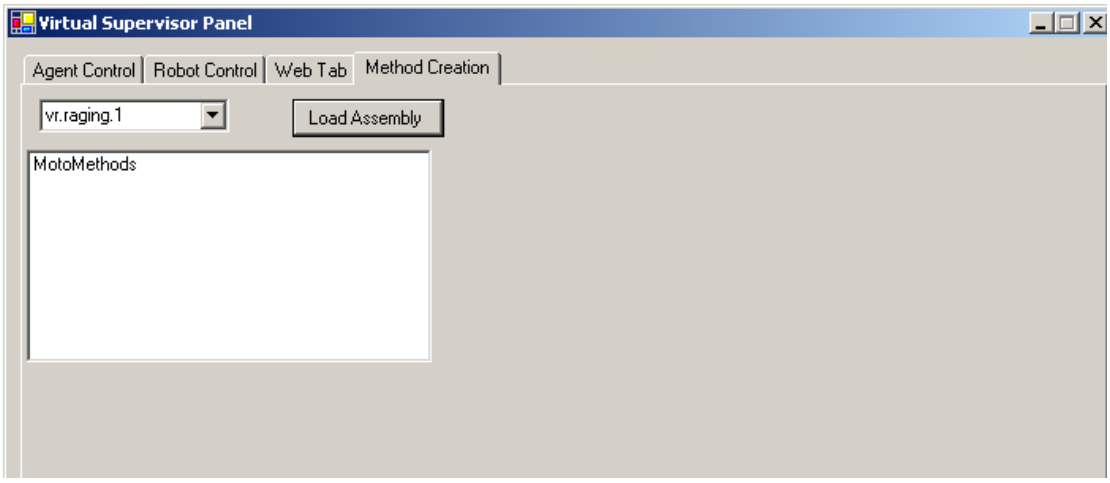
The amazing thing about this example is that the totality of the code for the loaded assembly has now been displayed—just those two functions. CreateMethods announced the existence of a new method, called “Laundry if David’s Close” and mapped it to the second and final method of the class, DualMethod(). All the code necessary to add functionality to a VR has been written in a few dozen, simplistic lines.

The following figure (5.10) shows the functions available after setting up a VS, one VR, and the same robot to which we have been demonstrating connections throughout. They are the three functions that the robot itself supports.



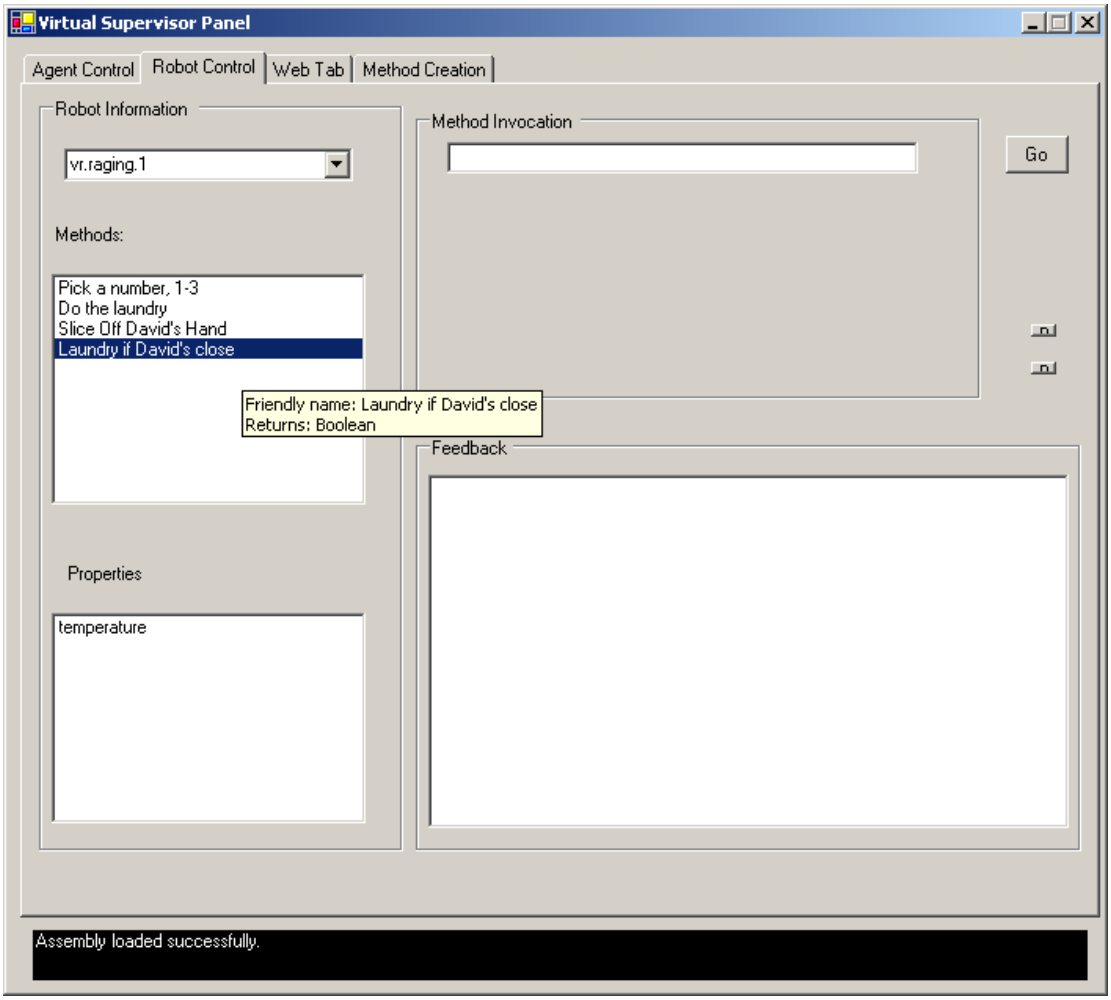
5.10 The static methods available to the VS after connecting to a robot

Note that “Do the laundry” as well as “Slice off David’s Hand” are both exposed to this VR, and are the two functions that the example module may call. The next step is to send over an assembly for the program to load up while it continues to run. This is done from the “Method Creation” tab on the virtual supervisor, which allows the supervisor to browse his or her computer for pre-compiled assemblies. Below, Figure 5.11 below shows the “Method Creation” tab and the result of this action.



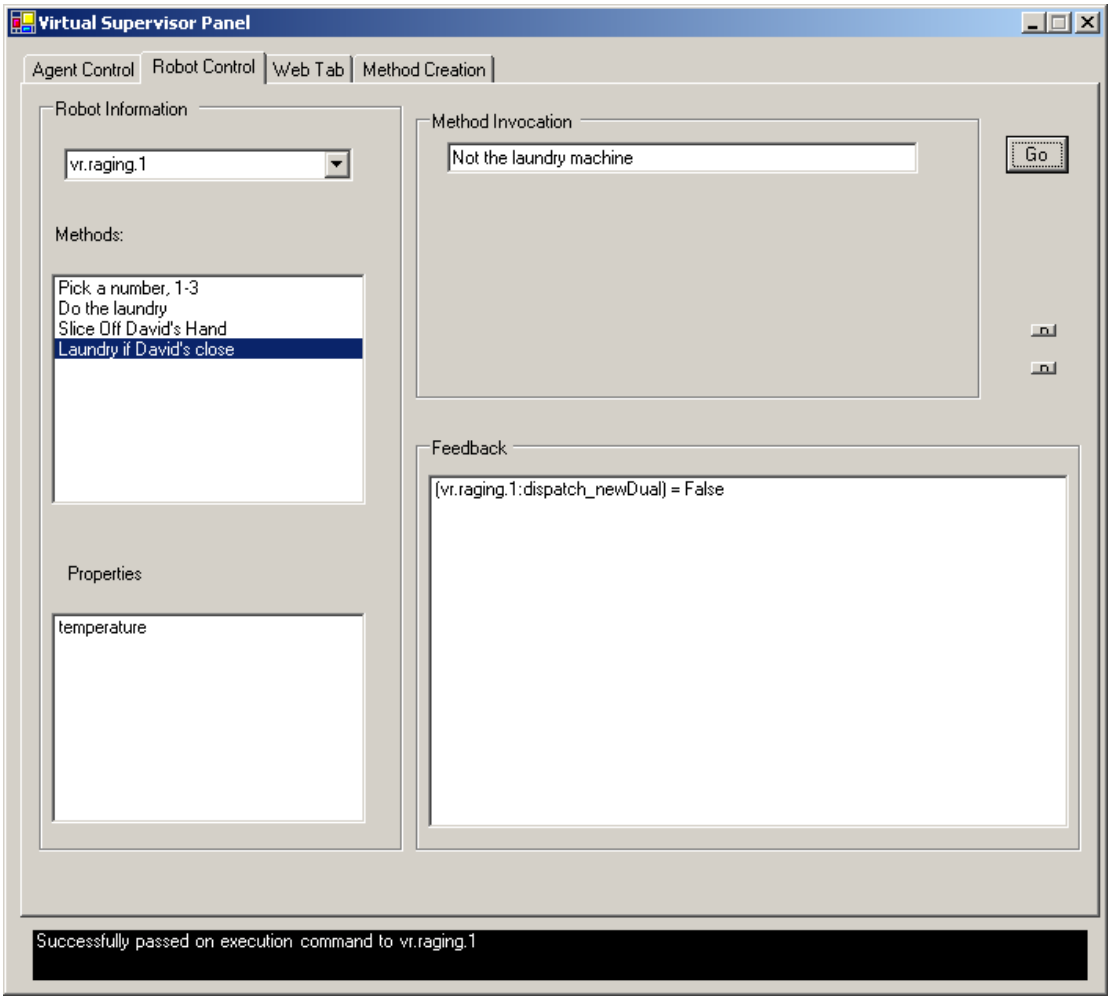
5.11 Sending over a pre-compiled assembly containing new functionality to the VR for loading

Now, the Robot Control tab display should have updated to reflect the dynamically loaded functionality. This is indeed the case, as may be seen in figure 5.12 below, which shows the “Robot Control” tab, this time with the new method.



5.12 The VS after a VR in its chain has loaded up the dynamic functionality. The new function has been reported to the VS in the same fashion as the static robotic functions.

The new method, entitled “Laundry if David’s close” has appeared. It accepts one text parameter, which the virtual supervisor uses a text box to represent, and returns a boolean value. When the supervisor enters some value other than “Laundry Machine” (in this case “Not the laundry machine”) and executes the method the virtual supervisor responds as shown in figure 5.13 below:



5.13 Execution of the dynamically loaded functionality

Note that the return value (displayed in a listbox) came back false. When looking at the output from the agent, it is clear to see what happened:

```
<6:32.48>{vr.raging.1}Relaying a call to vr.raging.1.Laundry if
David's close
<6:32.48>{executing Slice Off David's Hand}
Just got a base call to run a method; checking for validity.
<6:32.48>{executing Slice Off David's Hand}Valid request.  Going to
attempt to execute.
<6:32.48>{executing Slice Off David's Hand}Waiting for a return to
come in.
<6:32.48>{executing official_sliceDavidsHand}Spun off a thread to
handle execution.
<6:32.48>{vr.raging.1}Relaying a call to Motoman formal name.Slice
Off David's Hand
<6:32.48>{vr.raging.1}Realizes that he's returning from a method
that he invoked dynamically.
```

```
<6:32.48>{Return in base}recevied a return for
official_sliceDavidsHand ... going to release waiting procs.
<6:32.48>{executing Slice Off David's Hand}A return came in for
official_sliceDavidsHand
<6:32.48>{executing Slice Off David's Hand}Returning to the derived
class with our return value.
<6:32.48>{vr.raging.1}I actually received a return from the derived
class just now... it was False
<6:32.48>{vr.raging.1}Got a return value from vr.raging.1 for
dispatch_newDual(False)
```

This output is worth a simple analysis. First, a call was made to the vr for "Laundry if David's close", which came from the VS. This call mapped to DualMethod(), which used the base class to attempt to make a function call. The base class recognized it as a valid method name and spun off a thread to handle the execution and locked down the thread that made the call to the base class. This thread then ordered the vr to relay the call to the robot, and then the next thing we see a return value is coming back. The virtual robot realizes that the return value is for a function that was created dynamically, and thus it must handle the return value instead of continuing to pass it upwards. The VR releases the waiting processes and the process that was waiting for the return value realizes that the value has come back and returns back to the derived class with that return value. Since it was not "Laundry Machine", as the module hoped for, the module returned false without going through any additional calls.

As one final demonstration, observe what happens when a second assembly is added to the virtual robot that had previously loaded the first. This time, a method called "Tell Dual David's by Laundry" is created that calls into the created DualMethod() function and passes it a parameter of "Laundry Machine". The new method requires no parameters from the user and returns a boolean value, specifically, the value that DualMethod() returns. The following is the complete code

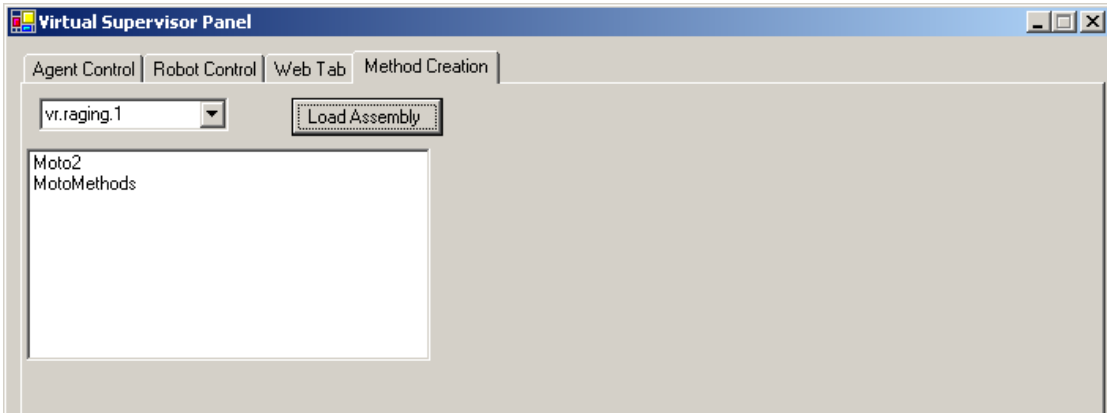
for this new module:

```
public override ArrayList CreateMethods()
{
   ArrayList newMethods = new ArrayList();
   ArrayList parms = new ArrayList();
   DynamicBotMethod dbm = new DynamicBotMethod("Big_dual", "Tell
      Dual David's by Laundry", parms, true, new TypeBoolean(),
       MyName, new DynamicMethodDelegate(BigDual));
   newMethods.Add(dbm);
   return newMethods;
}

private object BigDual(BotMethod bm)
{
   Console.WriteLine("Executing the big dual method.");

    bm.parameters.Add(new BotParam(new TypeText(), null, null,
        "laundry machine"));
    object ret = base.RunMethod("Laundry if David's close",
        bm.parameters, true);
    if (ret is string)
    {
       ret = bool.Parse((string)ret);
    }
     bool bret = (bool)ret;
     return bret;
}
```

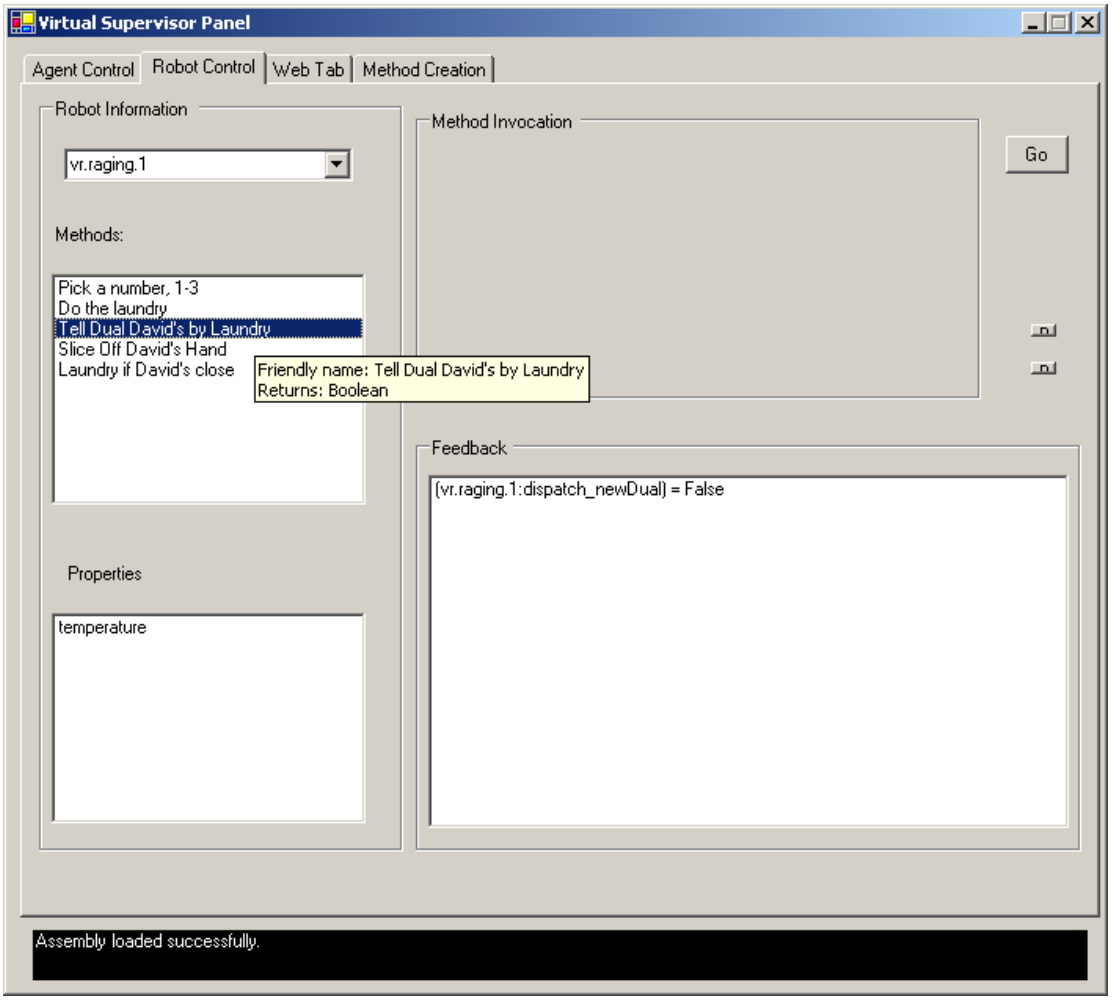
Again, load up the assembly, shown in Figure 5.14 below:



5.14 A second assembly is sent to a VR for loading. Note that the VR tracks the assemblies it has loaded.

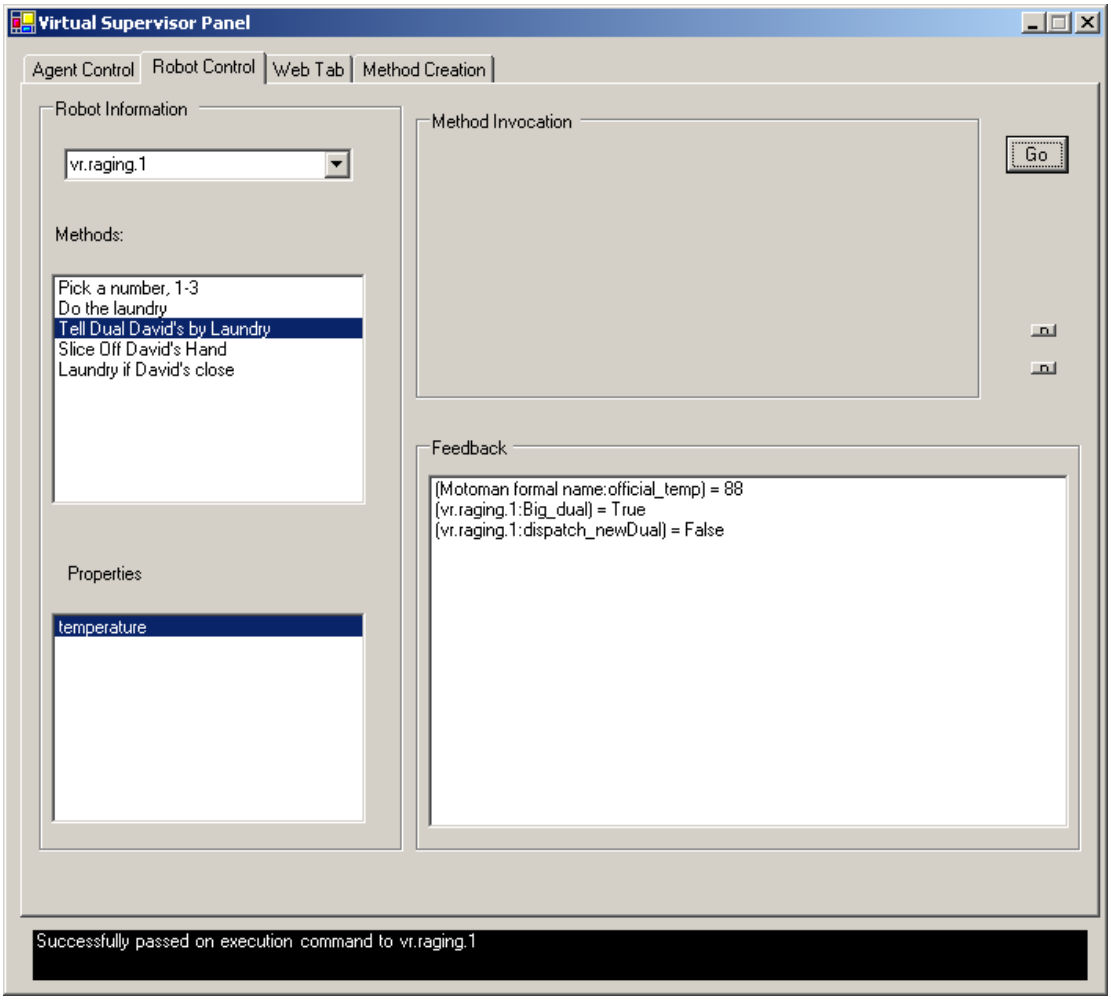
Note the new method, “Tell Dual David’s by Laundry”. Additionally, the temperature property has been set up to report when any changes occur. When the

"Do Laundry" function is invoked, the temperature property is programmed to increase by three degrees. Since the "Do Laundry" function has no return value, a reported change in temperature is the easiest way of telling if this function has executed, as is expected. The "Robot Control" tab, with both of the dynamically loaded functions available, is shown in figure 5.15 below.



5.15 Selecting a 2nd dynamically loaded function that will call the first.

When the supervisor chooses to execute this new command, the Virtual Supervisor will respond according to the loaded code. It is shown in figure 5.16 below, and will be explained immediately afterwards.



5.16 The result of executing the 2nd dynamically loaded function. Note that the temperature has changed, indicating that it successfully called the 1st dynamically loaded function, which evaluated the parameters and called into a static function, "Do Laundry"

We see in figure 5.16 that Big_dual returned true, indicating that its call to DualMethod with "Laundry Machine" returned true, indicating that the laundry should have been done. This has occurred because the temperature property reported an update. Thus, we have successfully illustrated two dynamically loaded functions, the first wrapping up the second, which in turn wraps up actual robotic functionality. Each piece of dynamically loaded code is able to check return values of the methods they call and act accordingly, using any methods or properties that existed before it

was loaded, regardless of whether the attributes were exposed by the actual robot or simply another Virtual Robot.

6. Future Work and Conclusion

6.1 Future Work

There is a significant amount of future work associated with expanding this project into its final form. The work to this point has served the intended purpose: researching an alternative architecture for robotic control and implementing the basic solutions to prove their worth. The next step, however, is to continue to test the solutions proposed in this thesis with differing robotic control scenarios as well as using it as a foundation to be developed into a full-featured, fully tested architecture.

Much of this work will surround the virtual supervisor. There is a need to add additional types into the architecture as robots that desire to transmit data in different formats are discovered, just as the Paradex wished to deal with a RangeType instead of merely an integer when controlling the large switch. These data types must be turned into classifications and have default controls associated with them for display on the GUI. Also, adding non-default controls to existent classifications and providing the supervisor ways of saving his preferred control types for each classification are both needed. Aside from specific data classification, there is also the need for the Virtual Supervisor to handle different priorities of data. When a return value with a high priority is reported to the Virtual Supervisor, it should react accordingly. Currently, these priorities are being reported, but the Virtual Supervisor does not react differently based on a priority. It is hoped that eventually high-priority responses will cause the virtual supervisor to demand acknowledgement from the Supervisor and, should it not be received, attempt to contact him via a cell phone or pager.

Loading code into the VRs is in many ways a much easier process than it could have been. There are well defined interfaces to access robotic functionality and pass values to the Virtual Supervisor the Virtual Robot will recognize and thus the Supervisor may code against. This code may be written in any .NET language, of which there are dozens, since the architecture was written in C#, a .NET language itself. However, to minimize the effort required to author dynamically loaded code, a small visual development environment would be highly desirable. If supervisors could use a visual interface to choose methods to call and parameters to pass in, it would make writing these pieces of code even easier.

Although the architecture was built to accommodate multiple robots, very little has been done to do multiple robot coordination, or to load dynamic functionality that would do anything exciting with the two robots. Currently, the extent of writing functions that wrap up basic methods on multiple robots has been limited to functionality on each robot that is unrelated to the other. Thus, true coordination has not yet been attempted. Moreover, a demonstration of writing dynamically loaded code that reorganizes groups of robots to work in different teams would be impressive, and is well within the scope of the architecture.

One of the problems with allowing such abstraction from the robots' basic commands, is that it becomes possible for the supervisor, who is unaware of the actual implementation of the high level commands he or she is issuing, to request that the robot perform actions that contradict each other. For instance, he could execute a function to load into a VR controlling the Paradex robot that wrapped up calls to turning a switch on and off. Should this function be executed, the Paradex will be

given conflicting commands that may or may not achieve what the supervisor intended. If the virtual robots could track the dependencies of their dynamically loaded functions and report an error when an ambiguous set of commands was wrapped up, it would be a great addition to the architecture.

Finally, there is a great deal of work to surrounding the mobility of the virtual robots. This is by far the area of future work that is the most complicated. Currently, a simple algorithm is in place. Using this algorithm, a virtual robot looks at each agency and records the available CPU as well as the round trip time to each of its neighbors. The virtual robot then moves to the agency that provides it the best resources. First, this algorithm does nothing to counter possible looping that may occur. For example, AgentA is dissatisfied with the CPU performance of its current machine. It looks up another agency on another machine and, upon running tests on that agency, finds that the computer on which the agency runs is somewhat faster than its current computer. Thus, it moves itself from its current computer to that agency. But AgentB, which is located on that computer, now finds that with the drop of CPU performance on its computer, it is better suited for AgentA's old computer. Moving there, AgentB then makes that computer undesirable for a third Agent, AgentC, who had been sharing the computer with AgentA originally. AgentC gets up and moves to computer B. This is the initial state for the same cycle to occur, with processes reversing the direction of their move. Clearly, the local greedy solution is not universally stable.

Additionally, there is the issue of priority. Perhaps AgentB is performing a critical task, such as navigating a robot through a dangerous stretch of road. AgentA,

then, could possibly be performing some CPU consuming task as well, but of much less importance to its supervisor relative to the importance of robotic navigation to AgentB's supervisor. If AgentA simply sees AgentB's powerful computer and moves there blindly, it may starve AgentB from getting its massive requirement of CPU cycles, leading to a failure of an important task in the name of an efficiency increase for an unimportant task. This situation is also undesirable.

Ideally, a stable global optimal solution would be calculated before any movement occurred, in which the importance, or priority, of each agent was taken into consideration as well as their current task and current task requirements. It may not be best for any one process or even for any process at all, but overall would be the best solution available and thus stable, until requirements changed or additional processes were added. This problem is known as the *agent-planning problem*, and is an emerging field of research. It has been compared to many well-known problems, including the problem of distributed data storage and the traveling salesperson[23]. A complete analysis of this difficult, currently unsolved problem is well beyond the scope of this thesis. However, when applied specifically to the proposed virtual robots, the problem reduces fairly well to that of obtaining a globally optimal solution for similar processes that are engaged in performing different tasks that are each requesting space on a finite number of computers with finite resources. This problem has been researched for many years under the heading of web page caching on servers. Users each want their common pages cached on a local server instead of requiring those pages to be reloaded from the page's main server each time. The local servers have a limited number of resources, most notably space, to be spread

[23] Katsuhiro Moizumi. [15]

across all the users that want to cache their web pages there. Therefore, researching the best way to find the globally optimal arrangement of VRs meant relating it to the problem of web caching and applying the research and results of that work to our problem.

Currently, there are a number of different web caching algorithms in usage. These include Greedy Dual Size[24], LRU, LFU, LRU-Threshhold[25], LogLRU[26], HYPER-G[27], Pitkow/Recker[28], Lowest-Latency First[29], and Hybrid[30]. Out of these, the best algorithms are generally agreed to be Greedy Dual Size and the Hybrid when calculation time for the algorithm itself is disregarded[31] [32]. In the Hybrid model, a function is computed for each document in the cache. The function is designed to capture the utility of retaining a given document in the cache, and the document with the smallest function value is evicted. The function for a document depends on the time to connect with the server on which the page is located, the bandwidth of that server, the number of times the page has been requested since it was brought into the cache, and the size of the document. Greedy Dual Size bases its decisions about which page to evict from the cache on an equation as well, but uses some slightly different factors. These factors are the locality of the page, the size of the page, and the latency/cost concerns of caching the page instead of fetching it each time. The two methods are very similar and use similar parameters, differing in ways too minute

[24] comparison of online
[25] LRU THRESH
[26] loglru
[27] hyper-g
[28] pitkow/recker
[29] lowest latency first
[30] hybrid
[31] greedy
[32] compare

to be listed here. Also favored as a web-caching algorithm is the LRU method, which simply removes the least recently used page from the cache before adding the new page. It does not produce as optimal a solution as other algorithms (like Greedy Dual Size), but it achieves its calculations in O(1) vs. O(log n)[33] for the more complicated algorithms where n is the number of pages in the cache. Also, LRU does not require additional space, of which many other algorithms require a significant amount.

The bottom line for web caching is that all of the algorithms look at the currently cached pages and attempt to find the page that has fewest of the characteristics the algorithm holds to be important. Most algorithms are based completely on LRU or incorporate it heavily into their equations along with other factors, such as size and latency. In future research, we hope to apply this principle to the arrangement of our virtual robots. We must analyze the currently existent VRs against a set of characteristics that we hold valuable and give the best spots to the VRs that are rated the highest. Some of the parameters in web caching have no clear counterpart in our situation. For instance, the size of a page is usually considered, but the size of virtual agents will be very consistent. However, other web page factors have clear associations with our agents; locality of the page's server, for instance, parallels well to the distance between an agent and his slaves and master. The agent's message must travel different distances depending on which agency it utilizes, just as the time to reload a page depends on the time to contact the page's server or the time to contact a cache of that page on a local server. There is also a parallel between LFU (Least Frequently Used) and LRU (Least Recently Used) to the frequency of message

---

[33] Some implementations of GDS may obtain a running time of O(1), but the constants in the time complexity analysis are so large that those implementations are rarely seen in practice

passing and command execution by the virtual robots. And there are certainly other factors to take into account with virtual robots that have no clear counterpart in the web-caching world, like job priority. We believe that calculating a globally optimal solution for virtual robot placement is plausible should factors be correctly assessed in an equation that can meaningfully incorporate information about each agent.

6.2 Conclusion

There are many challenges involved in the remote control of robotics. Many of these challenges have been met by current solutions, yet there remain many difficult objectives that have not been satisfactorily achieved. We believe that a revision to the architectural model of robotic control allows the accomplishment of many of these objectives as well as permitting the incorporation of several features to the realm of remote robotic control that had not been previously possible. This revision involves two main components. First, the introduction of peer-to-peer processes that serve as intermediaries between the remote control application (the *virtual supervisor*) and the robot itself. These processes are viewed as the supervisor from the perspective of the robot and as robots to the supervisor, and are therefore called *virtual robots*. Secondly, the architecture is designed to allow control of robots generically. This means that the virtual supervisor, as well as the virtual robots that exist between the virtual supervisor and the robot, are merely templates. They were written with no preexistent knowledge of the robot they are intended to control, and therefore do not have any programming to reflect the methods and properties any specific robot has exposed for remote interaction.

We believe that since our model is focused on having a single application that may control robots generically, it will be appealing to industry users. Currently, to enable remote supervision of a robot, a language to describe the robot's abilities and pass supervisory commands must be established, robotic functionality must be exposed to communicate with a controlling application, and, finally, the controlling application must be programmed. This controlling application must accept user input and then translate it into a message that can be understood by the robot before sending that message over the Internet. In contrast, when using our architecture, the robot needs only to have a small stub written that defines its attributes in a well-defined language. The virtual supervisor will then mold itself to accept user input relevant to the robot in question and send commands to the robotic stub. The virtual supervisor will place visual controls on its GUI that correspond to the robot's methods and are intuitive to users of modern-day software. In addition, the virtual supervisor will communicate with the rest of the remote architecture as efficiently as possible when sending the messages across the Internet. In short, the architecture makes it trivially easy to enable remote control of nearly any robot through a rich, user-intuitive GUI that is completely pre-written.

Because our control model uses peer-to-peer processes, virtual robots, to abstract away the control of the supervisor and the functionality of the robot, we are able to boast several advantages over traditional master/slave remote control. One immediate advantage is that because the virtual supervisor does not interact directly with a robot, it is easy to build a structure of a virtual supervisor directly or indirectly

controlling multiple virtual robots, each of which control a robot. Thus, achieving single supervisor control of multiple robots is effortlessly achieved.

There are also advantages to this structure that rely on the advanced capabilities of these virtual robots. The virtual robot processes are actually *mobile agents* that are capable of evaluating their current task and the resource requirements associated with that task, as well as gauging the resources available on various, registered host computers. Upon finding a more desirable computer, these processes are able to move themselves, along with the task they are executing, to that new host before continuing their execution. This results in an inherent flexibility to the types of resources required by the objective the supervisor assigned. When a supervisor orders a robot to perform a job that requires fine-grain control, for instance, the virtual robot that is carrying out the supervisor's command will recognize the need for a short round trip time to the robot and move itself to a computer that can provide it with such. The same is true when a virtual robot realizes that its supervisor is requesting a computationally intensive task. All of this flexibility and run-time reaction to the supervisor's commands happens in such a fashion that the virtual supervisor program itself, and thus the supervisor, need not themselves move to a different computer or direct the mobility of the virtual robots in any way.

The virtual robots have also been programmed to accept additional functionality from the supervisor. Thus, to add new functionality to a robot based on currently existent abilities, the supervisor may simply write code for the new function, compile it into an assembly, and send it to a virtual robot. The virtual robot is aware of the robot's functions as well as how to invoke them, since it must use this

knowledge whenever a supervisor wishes to know what abilities a robot may have or wishes to invoke one such ability. This virtual robot then reports the ability to execute this new function as a robotic ability to the virtual supervisor along with the functionality that is actually programmed into the robot. When the virtual supervisor executes this new function, the command reaches the virtual robot that loaded the supervisor's assembly, and that VR passes control to the assembly. The assembly executes the supervisor's program, including calls to the robot's current functionality. This is an especially exciting feature, since it allows a supervisor to upgrade the abilities of a robot without directly reprogramming the robot in whatever language and operating system the robot requires. Instead, the supervisor may reprogram the virtual robots and follow the same steps to add functionality to any robot that fits within our architecture. This upgrade will take place dynamically, without a need to recompile the virtual robot program or even halt operation of the robot in question. Additionally, since a virtual robot may control multiple robots, a function may be loaded dynamically that, upon execution by the supervisor, actually executes commands on multiple robots in a coordinated effort.

This ability to load functionality into a virtual robot results in a supervisor's ability to repeatedly wrap up existent functionality in an object-oriented fashion, so that layers of abstraction are established. By layering functionality and thus providing the ability to program the robots against higher and higher level abstractions (that may include one or more robots), the supervisor is able to write powerful command sequences using relatively little effort, much as using classes greatly simplifies modern software development. Furthermore, by enabling the

robots' abilities to be modified while they are running, the control architecture allows the supervisor to react to the conditions his or her robots encounter with a new degree of flexibility. If the environment to which the robot must be deployed is mostly unknown to a supervisor, he or she may program in a significant portion of the robot's actual functionality after the robot has arrived in its target environment and reported its surroundings. In addition, by loading functionality that involves multiple robots, a supervisor could dynamically reorganize teams of robots depending on what unexpected challenges the environment may pose to a team of robots that must accomplish some task.

We feel that because of its fresh approach, our control architecture represents a step forward in remote robotic control. First, it provides an easy way to allow the remote, supervisory control of generic robots. This control takes place from within a rich GUI that molds itself to the target robot. Our architecture also solves many of the current difficulties of robotic control, such as flexible, fine-grain control and facilitating the coordination of multiple robots from within a single application. Finally, it offers the impressive feature of enabling the supervisor to dynamically load functionality into virtual robots. By modifying the abilities of a robot as it executes, the supervisor may wrap up and abstract away lower-level functionality into higher-level methods that allow a more powerful programming model.

5.7 Bibliography

[1] Acharya, Anurag M; Saltz, Ranganathan Joel. "Sumatra: A Language for Resource-aware Mobile Programs" (1997).

[2] Adams, Bill. US Army Maneuver Support Center. "Robotics and Unmanned Vehicles: The Future". (December 13, 2001).

[3] Alam, Khurshid; Mukherjee, Sudipto. "ROMP: Remotely Operated Mobile Platform." (June 1995).

[4] Anderson, R. J.; Spong, M. W.. "Bilateral Control of Teleoperators with Time Delay". Proceedings of the 27th Conference on Decision and Control, pp. 167-170. (Dec 1988).

[5] Banerjee, Rajrup; Mukerjee, Amitabha. "Implementation Of Parallel Watershed Algorithm On a Network Based Environment." (March 2001).

[6] Bates, J.; Bacon, J; Moody, K; Spiteri, M. "Using Events for the Scalable Federation of Heterogeneous Components." (September 1998).

[7] Brooks, Rodney A. Massachusetts Institute of Technology Artificial Intelligence Labs. "A Robust Layered Control System For A Mobile Robot". (1985).

[8] Ellis, Jeffrey Brent. University of Cincinnati. "An Investigation of Predictive and Adaptive Model-Based Methods for Direct Ground-to-Space Teleoperation with Time Delay" (1988).

[9] Ferrell, William R. Massachusetts Institute of Technology "Remote manipulation with transmission delay". (January 1964).

[10] Fong, Terrance; Baur, Charles. "Multi-robot driving with collaborative control." (July 2002).

[11] Gelernter, D. "Generative Communication in Linda". (January 1985).

[12] Ghiasi, Soraya; Zorn, Benjamin. University of Colorado. "A Reusable Framework for Web-based Teleoperation of Robotic Devices" (June 2000).

[13] Lehman, Tobin J; McLaughry, Stephen W.; Wyckoff, Peter. "T Spaces: The Next Wave". (1999).

[14] Meynard, Jean Paul. Linkoping Studies in Science and Technology. "Control of industrial robots through high-level task programming." (May 2000)

[15] Moizumi, Katsuhiro. Dartmouth College. “Mobile Agent Planning Problems”. (1998).

[16] Munson, Michelle; Hodes, Todd; Fischer, Thomas; Lee, Keung Hae; Lehman, Tobin; Zhao, Ben. IBM/Berkley. "Flexible Internetworking of Devices and Controls". (1999).

[17] Nipi, Giri; Ghosh, Amitabha; Sriram, K. “Design and Developement of a master slave teleoperated robot.” (October, 1999).

[18] Pease, Wayne. University of Southern Queensland. “E-Commerce Enabling Technologies” (2001).

[19] Silva, Alberto; Delgado, Jose. INESC & IST Technical University of Lisbon. “The Agent Pattern for Mobile Agent Systems”. (1998).

[20] Stein, Matthew. Grasp Laboratory, University of Pennslyvania. “Behavior-Based Control for Time-Delayed Teleoperation” (1993).

[21] Taylor, Ken; Dalton, Barney. Australian National University. “Issues in Internet Telerobotics.” International Conference on Field and Service. (1997).

[22] Venema, Steven C.; Bejczy, Antal K.. Jet Propulsion Laboratory, California Institute of Technology. “The Phantom Robot: Predictive Displays for Teleoperation with Time Delay”. (1997).

[23] Whittaker, Red; Wettergreen, David; Bares, John. Carnegie Mellon University Robotics Institute. “Configuration of a Walking Robot for Volcano Exploration” (1998).

[] Chuckpaiwong, Ittichote. Case Western Reserve University. “Reflexive Collision Avoidance for a Novel Parallel Manipulator” (2001).

[] Naksuk, Nirut. Case Western Reserve University. “The Implementation of a Natural Admittance Controller for an Industrial Robot” (2000)

[] Mathewson, Brian. Case Western Reserve University. “Integration of Force Strategies and Natural Admittance Control” (1993)

[]Cao, Pei; Irani, Sandy. University of Wisconsin-Madison and University of California-Irving. “Cost-Aware WWW Proxy Caching Algorithms” (1997)

R. Wooster and M. Abrams. Proxy Caching the Estimates Page Load Delays. In the *6th International World Wide Web Conference*, April 7-11, 1997, Santa Clara, CA.

http://www6.nttlabs.com/HyperNews/get/ PAPER250.html. → Hybrid and lowest latency first

M. Abrams, C.R. Standbridge, G.Abdulla, S. Williams and E.A. Fox. Caching Proxies: Limitations and Potentials. WWW-4, Boston Conference, December, 1995->LRU Thresh & LOG size

S. . Williams, M. Abrams, C.R. Standbridge, G.Abdulla and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the ACM Sigcomm96*, August, 1996, Stanford University.->hyper G and Pitkow Recter

P. Lorenzetti, L. Rizzo and L. Vicisano. Replacement Policies for a Proxy Cache. http://www.iet.unipi.it/ luigi/research.html. →comparison 2