Facilitating the Development of Software for Multiple Devices

Andrew Correa
acorrea@mit.edu

February 3, 2008

Abstract

As computers are available at cheaper costs, software systems that span multiple devices are becoming more desirable and frequent. As a result, software development tools which facilitate the creation of such multi-device systems will find greater demand. Unfortunately, as far as the authors are aware, there are no such software tools available that are readily available and easy to use. To remedy this, we have started development on a plug-in to Eclipse, the leading Java application integrated development environment, that abstracts the task of negotiating the network transfer of source code and executables from software developers wishing to create a multi-device system. It is our hope that this abstraction will free the developer to focus on the task of creating multi-device software systems, thereby potentially increasing the number of future multi-device software systems. The final version of this plug-in will provide an easy-to-use single-machine interface to interact with multiple machines. We describe the Eclipse plug-in in its current state of development, our current experiences using this plug-in, and the final status and features we plan to incorporate into future versions of the plug-in.

1 Introduction

Several software tools have been built to aid computer users with tasks, ranging from compilers such as GCC to office tools such as Microsoft Word. Computer users do not need to worry about the details of how these tools work or interface with the hardware they run on, and instead can focus on achieve whichever task they have before them. As a result there is a plethora of software and digital media available ranging from games to design documents. Currently there are a wide variety of software tools to aide computer programmers in a wide variety of software-related tasks. The binary code of computer hardware has been abstracted farther and farther away with the development of newer and higher-level programming languages. Repetitive tasks are often far too tedious for a human to do but are simple enough for a computer to perform, so a software program is written which will perform the repetitive task more quickly than a human could, and most often more accurately as well. Short of solving natural language, software tools that are meant for development on one computer are in rich supply in both proprietary and open-sourced versions. We believe that the next great area of software development currently in need of sophisticated software tools is the realm of multi-device software engineering.

As computers become cheaper, more computing devices can be found in the same physical space (such as an office) than before. For example, given that many people have cell phones, laptops, desktop computers, personal organizers, iPods, and other small devices, the amount of co-located devices, or devices that are in the same physical place, can be in the 10's or 20's with relatively few people in one room. With the higher frequency and lower cost of computers comes the opportunity for software systems that span multiple platforms. Such systems could facilitate the interaction between already existing devices such as being able to transfer mp3 music files wirelessly from your desktop to your iPod or allow for more complex interactions between the real and the virtual world such as the novel interactions with museum exhibits demonstrated by the Virtual Raft [6] and EcoRaft [5] projects. Unfortunately, creating such systems involves more expertise

than creating a single software system because of the need to negotiate a network and its protocols, a process analogous to the previously-mentioned beginnings of computing, when it was still required that programmers know how to program in 1's and 0's due to its difficulty. The authors are aware of no software tools which aide the programmer in making multi-device systems by abstracting the details of the network away from the source code creation and distribution process which are, easy-to-use, and effective in their task. To fill this void, we have developed a plug-in for Eclipse, the leading IDE for Java development, which allows a programmer to program on two separate machines connected via a local area network by using only one of them. Future versions will allow arbitrarily many machines to be developed for. Only preliminary in-house tests have been performed on the plug-in, but they give very promising results. The plug-in works successfully (though slowly) on a sub-section of a multi-device system composed of only two machines and nearly 100,000 lines of code. We expect that future iterations will be able to greatly aide the development of future multi-device systems.

2 Multi-Device Systems and Multi-Device Software Engineering

Multi-device software engineering (MDSE) is a relatively new field that involves developing software systems that run on multiple machines concurrently. In this paper MDSE is primarily meant to refer to the development of systems built upon multiple co-located devices (devices that are used in the same physical space). It can also be used to refer to large networked games such as World of Warcraft or other web-based remote software deployment applications. While these other fields are also interesting and important, development for co-located devices is much less developed a field and offers users the option to interact with software systems in novel ways. For this reason we chose to focus on collocated devices.

MDSE is similar to a telnetting program, a program that acts as a workstation running on a remote machine, however MDSE also assumes that both computers engaged in the connection are self-sufficient computers that will be acting on their own. The local machine is treated as though it were the only computer running a program and so is the remote machine, however the remote machine accepts input from and gives input to a network connection instead of a user. Further, telnetting usually takes place when someone wants to access a remote, remotely-located machine from elsewhere in the world via a separate, local machine. The local machine then takes on the facade of being the remote machine, granting users access to all of the remote machine's files and permissions as though they were interacting with it directly. MDSE, however, implements only that part of telnetting which allows the local machine to take on the facade of the remote machine, and only for the purposes of running the program that is under development. It is therefore less broad and less powerful than telnetting. Also, it is intended that software developers have access to both machines so as fully to enable program execution and debugging. Similar to telnetting, Virtual Network Computing (VNC) grants remote users the privileges as abilities of local users. However the same things that were said of telnetting can be said of VNC. MDSE is a much more focused application domain that will not allow the flexibility of these other techniques and assumes immediate access to both local and remote machines.

Multi-device systems built on collocated devices are very powerful. The ability of separate desktop PCs to take on the role of islands and the ability of separate tablet PCs to take on the role of rafts between those islands has been proven to be quite effective to teaching children the nuances of restoration ecology [5] as well as being an overall interesting interaction paradigm [6] that warrants the development of future systems and the exploration of other novel interaction paradigms that can be created with multi-device systems.

3 Related Work

There are two major areas that demonstrate need for technology that facilitates the development of collocated multi-device systems. They are the area of multi-device systems itself that is, we feel MDSE interesting enough and full of enough potential to warrant its own development - and cross-platform systems - those that are written on one machine (usually so the programmer can enjoy the increased performance of a faster

machine's compiler) but meant to be run on a separate machine. Examples of the latter would be business software that needs to run on several operating systems (such as Windows XP, Mac OS X, and Linux) or something like cellular phone applications that could benefit from a more powerful computer to compile code.

The Ecoraft Project [5] and its predecessor The Virtual Raft Project [6] are multi-device systems that run on three tablet PCs and three desktop PCs. The tablets create the illusion of being rafts between island, represented by desktop PC's. These six separate computers work together by each running separate, interacting software programs. This single cohesive system of co-located devices provides a rich learning experience for children to learn the complexities of the ecological and restoration sciences. This richness could not have been achieved by interacting with a simple, single-machine system. Unfortunately, the complexity and tedium of creating a multi-device system such as The Ecoraft Project provides a great disincentive to create future multi-device systems.

Han et al. (2005) describe the methods and theory behind the creation of their new state-of-the-art sensor node operating system. This operating system was meant to run on small sensor nodes that communicate with one another, allowing data to be collected from large areas of land. The paper goes on to say that a few bugs still exist in the system. These bugs could be found more quickly if the developers of the system had a better tool, such as the Eclipse plug-in described here, that supported debugging programs. The work of Han et al. demonstrates another multi-device system that could benefit from multi-device development tools such as the one presented here.

Ringel et al. (2005) describe the difficulty they experienced protecting the campus network of CSU Chico from future Blaster worm like attacks. They claimed that getting the network to act as one centralized system was a difficult challenge due to the remote deployment of the new system and the remote deployment of subsequent updates and commands that are issued to the member-machines of the campus system. This paper demonstrates that even as recently as 2005, creating a cohesive network of multiple machines to act as one system is an arduous and painstaking process.

4 Implementation

We are currently developing a plug-in for Eclipse that adds functionality to the IDE that facilitates the development of software systems for multiple machines. We chose to create an Eclipse plug-in because Eclipse is the most widely-used development tool for Java applications.The members of our development team have written several single-device projects on their own and a few multi-device projects using Eclipse. Resultantly, we have become aware of much of Eclipse's functionality. Further, all of the Java-based code that our group creates is written in Eclipse. The Virtual Raft Project and The EcoRaft Project are two excellent examples of multi-device systems that were written in Java with Eclipse. Production of this plug-in had not been scheduled to start when our group first began the implementation of these massive multi-device projects, so all of the network managing was done by hand by each of the programmers. This was the source of much frustration and wasted time and effort. In order to test integral parts of our system, our software developers had to transfer executable class files between machines. This is much more tedious process than the simple clicking of a button or typing make into a console prompt to deploy a locally running application that many coders have become accustomed to. As a result, mention of the possibility of creating a tool, like a plug-in, to aide the process was mentioned during the development of The Ecoraft Project, but not seriously considered until much later.

4.1 Interface

We attempted to make the interface of the plug-in consist of similar-looking objects as the already existing interface of Eclipse. There are three major visual components to it: the property page section, the run configuration tab group, and the command line-based remote server module. These three components are used to see the status of a file or directory, configure a new remote run configuration and launch that

run configuration, and configure and set-up a second machine to be able to receive connections, files, and commands from Eclipse remotely, respectively. They are listed in further detail below.

4.1.1 The "Remote Properties" Page

In Eclipse, right-clicking on a file brings up the context menu for that file. The context menu has an entry for the properties of the currently-selected file. When clicked on, the property pages for that file are displayed. These pages show much of the information about the file that is relevant to the programmer. One such item of information is the location on the disk where the currently-selected file is stored. With this information, a developer can decided to alter that file outside of Eclipse, either by using another text editor that he or she may prefer (such as VIM) or by ecopying the file for back-up purposes. For the same reason of providing the software developer with useful information, we decided to create a remote-specific property page for each file, called the Remote Properties page. By viewing this page, the programmer can determine if the currently-selected file is a remote file (that is, a file which is also transferred to the remote machine and used or executed there) and, if it is remote, where on the remote machine the file can be found (its remote file path). With this information, the user of our plug-in can alter the file on the remote machine with the same freedom as he or she can interact with files on the local machine, provided that the user has access to the remote machine. However, our plug-in does not support remote actions that are outside of the scope of local actions. That is to say, if a certain action is not possible to perform with Eclipse on a locally resident file, then our plug in does not make that action possible on remote files (though just because a certain action is possible on local files, that does not necessarily mean they are also possible on remote files).

4.1.2 The Remote Java Application Run-Configuration Dialog

Before a software system of any sort can be run, the programmer must tell Eclipse how to run that program. For a normal, single-machine application, the information that is required is the name of the project, the name of the class containing the main() method, and any command-line parameters that are to be passed to the Java virtual machine or the program being written. To obtain that information from the user, Eclipse has the run-configuration dialog window. In this window the previous parameters can be specified in an easy-to-use graphical interface to allow Eclipse to run a program. There are many different types of run configurations that Eclipse supports by default, including the Java Application run configuration (for single-system projects) and the Eclipse Application run configuration (for plug-in projects such as the one described here). For our run configuration dialog we tried to emulate the single-machine Java Application run configuration as closely as possible for two reasons. Firstly, since one single Java application is being run on either machine, each must have the same amount of information to run as does one Java Application run configuration. Secondly, changing the look and feel of the interface, however slight the change, could possibly cause many users to become confused as to what the purpose of any given field is. We hope conforming to the style of the already-existing parts of the interface will cause our plug-in to be more intuitive to existing Eclipse users and will therefore be more widely accepted because the similarity to other Eclipse modules will cause the function of our run configuration interface to be self-evident. Unfortunately, though each of these run configurations and the tabs they are composed of have identifying icons next to their labels, neither our run configuration nor any of its tabs have icons. This was an extra step that did not add to the functionality of the plug-in, and was therefore put off in favor of tasks that advanced the overall functionality.

In our Remote Java Application run configuration dialog, we created four tabs, the Main tab, the Arguments tab, and the Remote Settings tab. Although all other run configurations have the generic Common tab and several have the Environment tab, Source tab, Classpath tab, or JRE tab, this first version of our plug-in did not include them as it is just a prototype version and we wanted to produce a testable, running plug-in as soon as possible.

4.1.3 The Main Tab

This tab is a rough emulation of the Main tab of the Java Application run configuration. It consists of text fields to enter the name of the project, the name of the class containing the main() method, and the class containing the main() method of the application to be run on the remote machine. This tab will report with an error message to the user if it detects any of the following things have happened: (1) the project name field is blank, (2) the main class name field is blank, (3) the specified project does not exist, (4) the path specified for the class containing the main() method does not exist, (5) the path specified for the remote class containing the main() method does not exist, (6) the remote file specified as containing the main() method has not been flagged as remote. This tab is different from the Main tab of the Java Application run configuration in several ways. It is slightly less user-friendly because it requires that you enter the project-relative path of the file that contains the class with the main() method instead of asking for the package-representation of the class. It does not offer an auto-complete feature for helping the user to find the particular class or project he or she is looking for. And it does not offer the same flexibility of main() method options that the Java Application run configuration dialog does. These issues are mentioned in the future works section of this paper.

4.1.4 The Arguments Tab

In the Java Application run configuration dialog, the Arguments tab specifies which command-line arguments are to be passed to both the locally running program and the Java virtual machine that program is running on. In our version, the Arguments tab takes these exact same parameters for both the locally and the remotely running programs and virtual machines. For simple programs, this tab can be ignored as it only enhances the functionality of the Java virtual machine or the budding system and simple programs will not need any added functionality or special running conditions.

Unfortunately, unlike the Arguments tab in the Java Application run configuration, the Remote Java Application run configuration does not have a list of possible variables that can be passed either to the Java virtual machine or the program under development. Resultantly, the fields in this tab remain unchecked for validity, so an error message will never appear due to a mis-configuration. Also, no choice of working directory is given in this tab. For the remote machine, the working directory is a property of the network configuration, thus it made more sense to insert the working directory choice into the Remote Settings tab. These shortcomings are addressed further in the future work section.

4.1.5 The Remote Settings Tab

This is the only tab that has no equivalent in the Java Application run configuration. It is responsible for storing the IP address of the remote machine and the path on the remote machine that any remote files are to be transferred to. Further, this tab displays a tree structure of all the files and folders in the project. This tree has check boxes next to each of the files and folders that signify whether the file is to be transferred to the remote machine. This is the most efficient way for the user to determine and change which files have been flagged as remote files and which have not.

This tab is still very slow for large projects and can be cumbersome to work with. The automated process of setting boxes to checked or unchecked based on whether they are flagged or not requires an extremely long amount of time when this tab is first opened. This is a top priority fix for the next version of the plug-in and is addressed in the future work section.

4.2 The Remote Command-Line Server

To run applications remotely, it is necessary to have a server on the remote machine that accepts connections and then instructions from Eclipse. This server's purpose is to bypass the permissions security protocol implemented in all modern operating systems, granting another machine (in this case the local machine the one running Eclipse) the ability to run arbitrary processes remotely. For our first version we decided it would be best not to worry about creating a graphical user interface for the remote server, so instead we

created a command-line program that opens a port and awaits a connection. It then acts both as a gateway for file transfers and a handle into remote process creation, blindly passing along any commands it is given to the command line. As a result, this server module creates a large opening in the security of the remote computer. Any (potentially hostile) machine could use knowledge of this machine to perpetrate any number of malicious acts. Therefore, this server (and consequently this plug-in) is not yet ready for widespread use among all programmers.

4.2.1 The Transfer & Remote-Link Mechanism

Four Java classes control the transfers of files and the transferring, interpreting, and execution of commands sent between machines. The four classes are contained in the org.elcipse.mdse.net package and are named Server, Client, RemoteServerMain, and FileCopier. Their names are meant to give some idea of what their purpose is. They are described in more detail below.

4.2.2 The RemoteServerMain class

This class contains the main() method of the remote command-line Java application of the remote computer. It accepts two flags that alter it's performance. The -IP flag is used to denote the IP address of the local machine that is the machine that Eclipse will be running on, and the programmer will be developing his or her code on. This flag is required for the program to run. If this flag is not passed to the Remote-ServerMain instance, the program will complain by informing the user that the flag must not be omitted and exit without opening a Server object. The other flag currently supported is used mainly for debugging purposes. The -v or -verbose option prints out each step that the application is taking in its computation. Examples of output messages the Remote Server will output are: "received message: '#create' ", "creating file: 'C:\users\acorrea\AClass.class", "wrote to file", etc.

4.2.3 The FileCopier class

This class is responsible for managing the details of transferring files over the network from a local directory to a remote directory on another machine. This class requires that a Server and Client object have already been instantiated correctly (see the Server and Client sections for more details), and it requires that it have a handle on the Client object it is to use to communicate with the remote machine.

Currently this is the most inefficient class of the plug-in. Its primary method the method that transfers a file from a given location on the local machine to a given location on the remote machine via the Client object works by transferring string-encoded bits. String-encoded bits are strings which contain only 1's and 0's and represent the bits in memory. This extremely inefficient mode of transportation was selected to carry bits across the network because bit-precision was needed and because using this method was comparatively much quicker to get working than finding a pre-existing library or class which had already solved the problem. The next phase of development is focused on improving this efficiency, which is discussed in more detail in the future work section.

4.2.4 The Client and Server Classes

In order for a connection to be made between two separate machines connected by some network, one must act as a server and await connection requests, while the other must act as a client and send connection requests to the server. The entity that creates the opportunity for a connection is the Server class. The server class wraps the complexities of creating a hub to which other machines can connect from. It is run on the remote machine so that local machine can connect to it at the programmer's request. The Java class that is responsible for making connections to the remote machine is the Client class. In the current version of the plug-in there is one instance of the Client class connected to one remote machine. Commands and files are sent through this object when Eclipse needs to communicate with the remote machine and when the remote machine needs to report back to Eclipse. Future versions will have multiple instances of this class, allowing for there to be one connection with each of multiple machines.

5 Process Summary

Our experiences creating this plug-in can be divided into three main categories: Programming, Searching, and Collaborating. Ideally, since this plug-in (like any plug-in) is at its heart a software system, most of our time would have been spent writing functional, error-free code. However, since Eclipse is such a massive body of code with much functionality that must be searched through, to find that piece of functionality that is desired takes a long time. Therefore, most of our time was spent searching online forums, Eclipse help documentation, and even already-existing Eclipse source code to find pieces of functionality that would be useful to my plug-in. Those processes and features are described in further detail below.

5.1 The Process of Programming

The programming process was arguably the most important process in the plug-in development cycle. Programming consists of writing the code that is run when the plug-in is loaded and the user is using it. As with many programming tasks, however, more time is spent at the computer deciding what code to write than is spent actually writing it. Further, once the code is written, it is almost always strewn with errors that must be fleshed out. This increased the amount of time spent in the programming part of plug-in development, however due to the large amount of experience we had with Java and the relatively little amount of experience we have creating Eclipse plug-ins, it still did not bring the hours spent on programming anywhere near the hours spent on searching.

5.2 The Process of Searching

It is acknowledged that Eclipse is a rich environment with many useful features and modules that allow for numerous pieces of functionality. Often, when we desired some piece of functionality, especially pieces that would help our plug-in look more contiguous with the already existing interface of Eclipse, we found that the functionality in question would most likely already be implemented somewhere within Eclipse's core. Resultantly, the most time spent on the development of this plug-in was taken up by the task of finding already-existing code, classes, and modules built into Eclipse that perform some function or task that could be useful to the plug-in. Unfortunately, there are still no good ways to document large bodies of code in a way that is easily interpretable by a human user. When trying to find a piece of functionality, we would first try to use Google's search engine to find an online blog post or forum discussing the answer. Most often, we had to search through the Eclipse code by hand to find which modules and classes we could use. This started off by determining which plug-in or over-arching module used that functionality. This was the most useful clue since Eclipse is open source and anyone can review the code of nearly every plug-in like one would review a piece of example code online. Once we found the code and the example, the online Eclipse 3.2 API reference would aide us in the rest of our search to determine how, exactly, to implement the feature we wanted.

This brings up a new problem that was encountered at various stages in the plug-in development process; the Eclipse API page. Online and in the help documentation of Eclipse there is the Application Program Interface specification for Eclipse 3.2. This interface, while extremely useful and lucid to the experienced Eclipse plug-in programmer (as any javadoc style document is), is cryptic to anyone not familiar with common Eclipse lingo. As a result of Eclipse being an IDE richly endowed with functionality, the vocabulary too of Eclipse if very rich and wide. Once we had found a solution to a problem or the correct class that would give us the functionality we desired, we often times did not know it, because the vocabulary used to describe said functionality was cryptic and unknown to us. Much of the battle of the process of searching for functionality was learning the vocabulary of Eclipse well enough to interpret our findings.

5.3 The Process of Collaborating

The process of collaboration is vital in any large software undertaking. We discussed this plug-in with everyone in our research group on multiple occasions and outside coders that we knew from classes or other

places. Collaborations usually took the form of 10 to 40 minute talks about how the plug-in should evolve, what was working, what was not, and which features are desirable and which need improving. Without the guidance of our friends and advisors and the suggestions of our peers, this plug-in would not have become as good as it has.

6 Experience Report

This section presents examples of my own experiences using the plug-in.The plug-in has not yet been exported as a plug-in, a simple procedure entailing creating a zipped Java Archive and inserting it into the plug-in directory of the Eclipse installation, because it is still under development. Resultantly, it is run via its source in Eclipse. As such two instances of Eclipse must be running when testing out the plug-in one instance is responsible for creating the plug-in and running the second instance which runs normally as though it were the only instance, but has the plug-in running. To begin we started by creating a new Java project and name it "Test." It was decided not to create a Remote Java project option because it would only add more complexity to the plug-in. To start we wanted to test the simple transfer mechanism of the plug-in to ensure that it transferred and ran files correctly. We wrote two separate classes that did not interact, one called Here and one called There each with its own main() method. Here is meant to run locally and only print out the string "here" when run while There is meant to run remotely and print out the string "there" when run. To test that these programs worked correctly on the local machine, we made two Java Application run configurations, one for each of the two classes, and ran them separately. Using this pre-built functionality, we verified the correct usage of the test classes. The next step was to attempt to run them concurrently on two separate machines. To do this, we opened the run configuration dialog and double-clicked the Remote Java Application item to create a new Remote Java Application. Then we enter "Test" for the name of the project, "Here.java" for the name of the main class, and "There.java" for the name of the remote main class in the Main tab that was presented. Next we clicked on the Remote Settings tab and entered the IP address of the remote machine. We used a small tablet PC connected to the same local area network that sat next to the desktop PC that Eclipse ran on. Using the ipconfig tool we determined the tablet PC's IP address to be 192.168.1.11 and entered it in the IP address field. Since this was the first test, we put "C:\temp" as the remote location for all files to be transferred after we have made sure that no conflicting files were on that path on the tablet PC. Then in the file system viewer, we checked the box next to the "There.java" file, marking it as a remote resource. We then saved the run configuration and exited the run configuration dialog. We now needed to set up the tablet PC to accept connections from Eclipse.

To initialize the server program on the tablet PC, we needed to start the program with the -A flag and the IP address of the local machine. Upon doing so the server started and awaited a connection request with a flashing carat on a new line of the command prompt.

We switched back to the local machine running Eclipse, opened up the run configuration that was made earlier, and hit the run button. The dialog disappeared and the console view came to the front with the word here in its body. When we checked the remote machine, we saw that there was the word there written in the console, just as though the There class had been run locally on the tablet PC. To make sure that the system was dynamic and that the There class was actually responsible for printing out the "there" string, we changed the string from "there" to "not here" in Eclipse and ran the program again. This time "not here" appeared on the screen instead of "there".

The next test of the plug-in is still coming to fruition. As mentioned previously and in the future works section, the file transfer mechanism is highly inefficient and requires further development. This inefficiency does not hinder the plug-in from performing quite effectively on very small-scale systems of less than 20 files. We realized that creating this tool was rather tedious as it required us to negotiate a network, and we realized that we already had a working plug-in to Eclipse that abstracts away the tedium of the network. The next test of our plug-in became to better its networking capabilities by developing the network section of the plug-in with itself. Testing is still underway, but the work that has been done so far has allowed us to better the plug-in, find and correct many bugs and programmer errors, and become more familiar with which aspects of functionality are difficult to work with and how we can improve those aspects.

7 Future Work

This plug-in is still very much under production. As such, most of the functionality that was planned for the final version has not yet been implemented. This section describes the final version of the plug-in and how that version differs from the current version.

7.1 Graphical Contiguity

Several features that are found in the Java Application run configuration have not been implemented in the same manner. This is bad both because it could be confusing to veteran Eclipse users and because the current Java Application interface is much more user-friendly the the one we have implemented. These faults have been mentioned earlier and are as follows: classes are specified by their file's workspace-relative location instead of their package and class identifiers, there is no auto-complete for any of the features located in the Main tab or Arguments tab, the same amount of flexibility to run a Java Application is not found in the Remote Java Application run configuration, and the entire Arguments tab does not offer suggestions for the arguments, but (as a result) it does not check for the validity of argument that are passed. Future versions of the plug-in will link to already-existing Eclipse functionality and use it to create a much more normalized user interface.

7.2 Security

Currently the network protocols this plug-in uses are based solely on the passing of Java String objects between the local machine running Eclipse and the remote machine running a server program. The remote machines server program executes any Java Strings it receives as though they were entered into a command line prompt on the remote machine. Not only is this an extreme security risk, but it is also quite inefficient when transferring binary files. Any malicious programmer could write a simple Java application that connects to the remote servers port number and gains access to all the read, write and execute rights of the user that ran the server program. Future versions of the plug-in that are meant for widespread use will implement a more efficient and secure bit-based (as opposed to string-based) transfer protocol for executing remote programs. This protocol is still under development. Preliminary ideas for more secure protocols would be passing only bit sequences, allowing only Java applications to be run, or encrypting the transmission with a secure MD5 checksum.

7.3 Multiple Machine Use

The ability to create two-device systems is not enough. Even a conceptually simple multi-device system such as the Ecoraft Project ran on six machines. This plug-in was meant to meet such demands. Therefore, a future version of this plug-in will allow the user to remotely connect to and develop for multiple machines. The biggest question faced is how to design the user interface to use this functionality. One idea that has been suggested is to use the tabbed set-up of run configuration dialog and create one tab for each new machine that requires the configuration mentioned in the implementation section. This poses the problem of figuring out how to alter the number of tabs the run configuration dialog has. Also, it may be confusing to the programmer to see the number of tabs fluctuating. This behavior may make it appear as though the user interface of Eclipse is acting erratically or incorrectly. It is therefore doubtful that this particular interface will be implemented.

A more promising idea is to present the Remote Java Application run configuration dialog in an almost identical way to how a conventional Java Application run configuration dialog looks. The only change will be that select tabs will have an extra text box displaying the variable number of total machines (including the local one) and a combo-box of all the machines (including the local one) that allows the programmer to switch between which machines properties are being altered. Each tab would then toggle between the various machines based upon what the combo-box read. To disambiguate which machine corresponds to which entry in the combo-box, entries would be of the form [main-class]<[IP-address]> for remote machines

and [main-class]<local> for the local machine. So for the example given in the first half of the experience report section, the text specifying how many machines there are would read "2" and the combo-box would have two entries: one reading "Here<local>" and the other reading "There<192.168.1.11>".

7.4 Remote Project

Currently this plug-in only contributes to the run configuration dialog, the properties page, and the Java process launcher. The idea of creating a contribution to the project types has been discussed within our group. It may be useful in future versions of this plug-in to give the user the option of creating a Remote Java Project. This project would come with its own Remote Java Perspective which sets up views conducive to multi-device software projects. However, no final decision has been made as to whether or not to implement a Java Project.

7.5 Difficulty of Distinguishing Machines

A vital piece of information when creating a multi-device system is the question of which artifacts are being transferred to which machines. There are currently two ways to gather this information in the plug-in: by (1) checking the Remote Settings property page, or by (2) looking at the Remote Java Application run configuration that has been set up for the project in question. Checking the property page takes time for one file. It requires stopping what you are doing, right-clicking and finding the property page. This is disruptive and becomes even worse when you want to check the status of multiple files at once. It requires you to check each one individually, narrowing your field of view, in a sense, and making it impossible to view the status of multiple files or folders at once. An improvement to this is using the run configuration dialog. A check is next to each file name that is a remote asset making it much easier to determine at a glance which assets are remote and which are not. This second method, while better than the first, still suffers from three flaws. The first and most noticeable flaw is the great length of time that it takes to correctly populate the tree with checks. In large systems such as the Ecoraft project, this process can take up to ten minutes. Future versions will fix this either by improving the code, or by removing the check-box tree viewer entirely and using some other graphical element for its purpose. The second flaw is the fact that this is the run configuration dialog. It is meant to be used to create run configurations and run applications using those configurations, not to be the only way to view the status of files. It is true that the remote property of files is part of the way the application is run and hence should be in the Remote Java Application dialog, however the fact remains that it should not be the only way to view the remote status of multiple files. The third flaw of using the run dialog as a status viewer is that this method requires the user to open up a separate dialog to view the status of his or her files. This is not necessary, as other Eclipse plug-ins that use multiple file properties to operate, such as the CVS and SVN plug-ins, do not require the user to open a separate dialog to determine which files have the property and which do not. Instead, the files are shown in the Package Explorer view with small sub-icons displayed on top of and in the corner of their file icon. This lets the user know at a glance whether a file is under revision control in the case of SVN and whether it is up-to-date or whether it has undergone changes. We will use this idea to display which files are marked as remote and which are not. When the plug-in supports multiple machines it will show a different icon for each machine. This may become confusing or over-complex with too many machines, so there will be a limit put on the number of machines that are used or the mini-icon used will simply be a unique number identifier of the machine to which it belongs. Also, some files may be used by more than one machine, so displaying a single icon may not be enough. Perhaps multiple icons will be used. This may become cluttered or visually confusing, however, so this too is not an ideal solution.

A more promising idea that came up was to determine which machines system was being worked on at the moment based on which file was being edited and gray-out those files in the Package Explorer that are not marked as being on the same machine as the current file. This runs into the problem of what to do when the current file is a part of multiple systems. In that case it may be best to show all of the files on all of the machines the current file is a part of. Using this method and the previously-mentioned mini-icon method would disambiguate the process a bit. Also adding a dialog for the soul purpose of editing which files are

assigned to which machines will be useful. In future work, we will determine the correct way of displaying this information, and implement it.

7.6 File Transfers

A network transferring tool that allows for interchangeable transfers of byte data and character data is needed for the network transferring. This is difficult to make, so to get a working copy up, we just used the already-existing Java 5 standard Socket, InputStream, and OutputStream classes, along with BufferedReader and BufferedWriter classes to transfer only character streams between them, and encode bit data into strings. This means that each bit of data requires 2 bytes of data to transfer correctly. This is highly inefficient. Also, String objects are immutable in Java, that is, once they are created, and allocated memory, that memory is permanently assigned to that unchangeable String object. After several transfers, Eclipse runs out of space to allocate more string data. A future version of the plug-in will implement a more efficient method for transferring data and commands across the network. This will be achieved, either by creating or finding a custom network transferring class or package, or revising the network protocol used so that the network only needs to transfer data bits between Eclipse and the remote machine. At the time of writing, the former seems more promising, because file paths and names are denoted by strings, making string passing valuable, but binary files like class files are made up of byte data, necessitating byte transfers.

8 Conclusion

The plug-in presented here is still in its infancy, however the fact that it already shows great promise for facilitating multi-device system development suggests that future versions have a great potential for being extremely useful to very many multi-device systems' software developers. This plug-in and other tools like it (specifically tools meant for rapid development and deployment of systems spanning multiple collocated devices) have the potential to change the way people use computers. With the availability of such tools comes the possibility that future software developers will take interest in using them and creating new, innovative multi-device systems, that in turn could change the way people become accustomed to interacting with collocated devices. This plug-in is an early step in creating a world in which we are accustomed to multiple collocated devices taking advantage of their relative proximity to one another by interfacing seamlessly and effortlessly.

Acknowledgments

We would like to thank Calit2 for providing us our research space, Professor Bill Tomlinson for his guidance and support, Abe Qavi for his valuable editing, and the school of ICS.

References

[1] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM.

[2] Dong-Hyun Lee, Hwan-Joong Lee, Young-Woo Lee, and Dong-Houn Shin. Wireless broadband services and network management system in kt. *Int. J. Netw. Manag.*, 16(6):429–442, 2006.

[3] Frank Mueller and Antony L. Hosking. Penumbra: an eclipse plugin for introductory programming. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 65–68, New York, NY, USA, 2003. ACM.

[4] Jerry Ringel, Andrea Mox, and Jason Musselman. Desktop management: reeling in the great white whale. In *SIGUCCS '05: Proceedings of the 33rd annual ACM SIGUCCS conference on User services*, pages 326–332, New York, NY, USA, 2005. ACM.

[5] Bill Tomlinson, Man Lok Yau, Eric Baumer, Sara Goetz, Lynn Carpenter, Riley Pratt, Kristin Young, and Calen May-Tobin. The ecoraft project: a multi-device interactive graphical exhibit for learning about restoration ecology. In *CHI '06: CHI '06 extended abstracts on Human factors in computing systems*, pages 1445–1450, New York, NY, USA, 2006. ACM.

[6] Bill Tomlinson, Man Lok Yau, Jesse Gray, Eric Baumer, Jessica O'Connell, Ksatria Williams, So Yamaoka, and Sara Goetz. The virtual raft project: a network of mobile and stationary computer systems inhabited by communities of interactive animated agents. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Emerging technologies*, page 30, New York, NY, USA, 2005. ACM.