A Web-Based Distributed Programming Environment*

Kiyoko F. Aoki and D. T. Lee

Institute of Information Science, Academia Sinica,
Nankang, Taipei, Taiwan
{kiyoko, dtlee}@iis.sinica.edu.tw
http://iis.sinica.edu.tw/

Abstract. A Java-based system called the *GeoJAVA System* was introduced in [1]. This system allows a user to remotely compile his/her own C/C++ programs and execute them for visualization among a group of remote users. *DISPE*, which stands for DIStributed Programming Environment, expands on the *GeoJAVA System* by allowing the resulting executables to be run on systems other than the host on which they were compiled, thus making the system more versatile. *DISPE* uses Common Object Request Broker (CORBA) services to enable executables compiled on this system to invoke methods in libraries on remote sites in an architecturally heterogeneous environment. Not only does this allow users to compile and execute their programs remotely, but the maintenance and duplication of libraries is lowered since agents are used to search for symbols in libraries located remotely and to compile them with the user's source code. As long as there is an Internet connection between the hosts on which these libraries reside, the agents can search and compile with these libraries.

1 Introduction

The *GeoJAVA System* introduced in [1] is a system developed for researchers of computational geometry to enable them to develop geometric algorithms without the hassle of the administrative aspects of programming, such as downloading, setting up and maintaining libraries and compiling programs. Briefly, this system consists of web-based interfaces where users upload their C/C++ programs that visualize geometric algorithms to the web server, compile them remotely, and execute their program, thus broadcasting the results to remote users via the provided visualization tool. The programs only need to provide minimal code for the actual visualization, and the management of remote users is provided by the system.

DISPE expands on this system to provide a more generalized programming environment that any researcher can use. Whereas the *GeoJAVA System* required

* This work supported in part by the National Science Foundation under the Grant CCR-9731638, and by the National Science Council under the Grant NSC-89-2213-E-001-012.

that the user upload their files to a remote host and compile with the libraries there, *DISPE* allows the user to take advantage of mobile agents to compile the code with distributed remote libraries. So these agents can be dispatched from a local computer to compile the local source code with remote libraries. Work related to *DISPE* will be introduced next, followed by a brief description of the system's design. The conclusion and future work are given in the last section.

2 Related Work

DISPE is a system that encompasses several areas. It is an extension of the *GeoJAVA System*, which incorporates a visualization tool with a collaboratory, allowing remote users to interact and solve geometric problems. It is also a compilation system for distributed libraries, different from existing compilers for high performance computing or parallel computers. It takes advantage of mobile agents to search for libraries during compilation, and it uses CORBA for dynamic, distributed execution. In this section, we discuss some work related to the main components of *DISPE*, namely agents and CORBA. To our knowledge, no known system provides all of the functionality that *DISPE* does.

2.1 Agents

The terminology of agents should be discerned between agents of artificial intelligence (AI), which are more like robots, and agents as described later in this article. Projects whose foci are more in the former area include work done by the Software Agents group at MIT Media Lab[22] and Softbots at the University of Washington[2].

Quite a few agent products are being developed, as is evident on the Agent Society home page[11]. Many of these are agent systems that provide a framework for agent projects. We list a few major ones.

Aglets [3] are the agents provided by IBM's Aglets Software Development Kit (ASDK), which is an environment for programming mobile Internet agents in Java. Aglets are Java objects that can move from one host on the Internet to another. When an aglet moves, it takes along its program code as well as its data. *DISPE* uses Aglets in its implementation.

Voyager [24] is a 100% Java agent-enhanced Object Request Broker (ORB) that combines mobile autonomous agents and remote method invocation with complete CORBA support and comes complete with distributed services such as directory, persistence, and publish-subscribe.

MOA [5] was designed to support migration, communication and control of agents. It was implemented on top of the Java Virtual Machine, and compliant with the Java Beans component model, which provides for additional configurability and customization of agent system and agent applications, as well as interoperability which allows cooperation with other agent systems.

2.2 CORBA

The Object Management Group (OMG) developed the CORBA standard in response to the need for interoperability among the rapidly proliferating number of hardware and software products available. CORBA allows applications to communicate with one another no matter where they are located or who has designed them.

Since the inception of CORBA, a number of different vendors have implemented their own versions of CORBA, each a little different from the other, especially where the CORBA specification was not very detailed. In implementing *DISPE*, requirements in deciding upon a CORBA implementation included adherence to version 2.2 of the CORBA specification, ease of use of the API, and availability. Based on these requirements, TAO and MICO seemed the most applicable.

TAO[23], is a real-time ORB end system designed to meet end-to-end application quality of service (QoS) requirements by vertically integrating CORBA middleware with operating system I/O subsystems, communication protocols, and network interfaces. **MICO** [20] has a clean API that supports the CORBA 2.2 specification, and it is freely available.

There are several other popular vendors who provide CORBA implementations. A full list of CORBA implementations can be found at [21]. There are several projects underway using CORBA, as can be seen in [14]. The ones that seem to be the most closely related to *DISPE* are the DOMIS Project at MITRE[15] and GOODE at the University of Lille[16], the latter of which has developed CorbaWeb[13] and CorbaScript[12].

3 Design and Implementation

The *DISPE* system is composed of the original *GeoJAVA System* plus the Compilation Agents and the *CORBAizer*. The *CORBAizer* is an application that can be used by any user, and the Compilation Agents consist of the Java agents which reside in Agent Contexts on each remote host. These Agent Contexts provide a layer of security between the agents and the host so that (1) agents do not gain unlimited access to the host's resources, and (2) the host cannot directly manipulate the agents and its data. Figure 1 illustrates the general architecture for the system. The work involved in implementing these components of *DISPE* is given next, followed by a brief example explaining the general flow of the compilation process using agents.

3.1 The Agents

The Compilation Agents are Java objects that use native compilers. The main Compiler agent communicates with several types of agents in order to obtain information and data from remote sites and to complete the compilation process. These other agents are the Include agent, the Client Agent and the Library Proxy

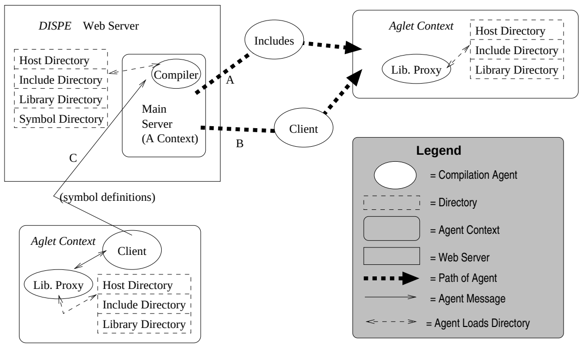

Fig. 1. Agents Architecture

Agent. While the main Compiler agent is stationary and remains at the original site at which the compilation begins, the Include Agent searches for missing include header files, the Client agent travels and searches for missing symbols during the linking phase, and the Library Proxy Agent serves as a stationary agent at each server host that responds to Include Agent requests for header file locations and to Client agent requests for library locations. Library Proxy Agents are created and disposed of by the agents who need them.

There is also a Vulture object for the Compiler agent which performs some checks for fault tolerance. For example, if an agent somehow "dies" unexpectedly, the Compiler agent would not know of it, so the Vulture object is used as a separate thread which occasionally sends a "ping" message to the dispatched agents, and if it does not receive a reply within a certain time frame, it sends a replacement agent to the same site, or, if the site has gone down, to any existing backup site.

In order to search for hosts, include header files, libraries and symbols, the Compiler agent maintains a hash table of this information which is read from disk upon startup and written to disk before being disposed of. This hash table serves as the Directory. The advantage of this approach is that the web server's agent host system keeps a central database file of this information in a simple hash table, and no new protocols or additional management software need to be introduced into the system. Java proved to be very convenient in this respect because of its serialization capabilities. So when the Compiler agent is first started, it reads four hash tables representing the Host, Include, Library and Symbol Directories containing host, header file, library, and symbol information, respectively, collected from previous compilations.

3.2 Performing the compilation

The compilation process performed by the agents are discussed next in a step-by-step manner.

Invoking the native compiler The first step is to invoke the compiler. The original gcc compiler goes through preprocessing, compilation, assembly, and linking[9], each invoked by separate executables. The linking phase was modified for the agents to use. In a normal compilation, the linker is automatically called by the compiler with various flags and options, so in order to keep consistent with the flags used during a "normal" compilation, we first determine the flags needed to be passed to the linker by running a sample compilation with the verbose (-v) flag. The same flags were then used when the agent called the linker.

The linker's `main()` method was modified into a method that was compiled into a shared library, which the Compiler agent loads when it is instantiated. To invoke the compiler, then, the Compiler agent first makes a system call to invoke the C++ compiler with the -c flag, indicating that it should compile up to the linking stage, but not link. Then the linker is called via the modified method in the shared library.

Finding Header files Before we can reach the linking stage, the compiler should first handle missing header files. A parser is used to parse the messages displayed during this earlier compilation phase for any messages indicating that an include file could not be found.

For missing include header files, the Compiler agent will consult its own copy of the Directory to see if any of the include files have been found before. If so, an Include Agent is dispatched to the host where the include files are located, taking along a list of the missing filenames. If these include files are not located in the Directory, the Include Agent is sent to the Main Server, which is a central host where libraries and other host information is stored. The Include Agent will then begin travelling and searching for the include file names that it has been given.

When an Include Agent arrives at a host, it searches for a Library Proxy Agent, and if it cannot find one, it creates it. The Library Proxy Agent reads the Directory of include files located at the host and sends the Include Agent a formatted list of include files. When the Library Proxy Agent first starts up, it also informs the parent of the Include Agent, whose proxy it receives from the creating agent, of the list of hosts that it "knows" about from its own Directory. The Compiler agent will then add any new hosts from this list. In this way, new host information can be dispersed in a "natural" manner by the agents themselves.

Any include files found are sent back to the Compiler agent and incorporated into the compilation. This agent keeps track of the "repeat count" which indicates how long it should wait before it should determine that the compilation cannot continue.

Linking Once this compilation phase completes successfully, the Compilation agent will begin the linking phase by making a call to the modified gcc linker via JNI. The linker will first attempt to link the source code with whatever libraries are available on the local host.

The linker uses a hash table to store all of the symbols in a compilation, and within the hash table is a linked list of undefined symbols. During the compilation, this linked list is used to merge in data from libraries to “pull in” symbol definitions into the compilation. We took advantage of this linked list to use with the agents.

To illustrate a compilation procedure using Figure 1, first, the Compiler agent is instantiated upon a call for a compilation. Given the source code information, the Compiler agent dispatches an Include agent when it detects any missing header files in the compilation stage (before the linking stage). So the Include agent (A) is sent to a host (based on information in the Directory on the DISPE server), where it finds the Lib Proxy agent. The Lib Proxy agent tells the Include agent which header files are located there based on the Directory information at the host. If there is a match for the file that the Include agent is searching for, the Include agent sends the header file back to the Compiler agent. If there are missing header files remaining, the Include agent will dispatch itself to the next host. The result of the compilation determines whether or not to continue on to the linking stage. The linking stage will begin with a link in order to determine which symbols are missing. If there are symbols missing, then a Client agent (B) is dispatched to the hosts where the header files were found. These Client agents will communicate with the Lib Proxy agents in a manner similar to that of the Include agent. However, once a symbol is found in a remote library, instead of sending each symbol back to the Compiler agent, the Client agent will begin its own “mini-compilation” in order to draw in as many needed symbols as possible. This process will result in an archive of object files containing the needed symbols for the compilation, which is sent back (C) to the Compiler agent. When the necessary symbols are found, the Compiler agent re-compiles the source code with the archive(s) found, and if more missing symbols remain, a message is sent to the Client agent(s) to find the new missing symbols (thus incrementing the “repeat count”). The Client agents are told to dispose of themselves once the compilation completes successfully (i.e., no missing symbols remain), or the “repeat limit” is reached (which can be set according to the user’s wishes).

3.3 The GUI

A web interface used as a GUI to the Compiler agent is described next. We use a daemon which creates a socket and listens for messages to create Compiler agents. An applet connects to the socket and simply displays any messages that it receives while listening on the socket. The daemon takes the applet’s socket ID and passes it to the newly created Compiler aglet. Thus, the Compiler aglet can send messages to display on the applet to indicate the status of the compilation.

Figure 2 is an instance of this web interface. Note, however, that a user can also use these agents from their local machine as long as they have downloaded the necessary components.

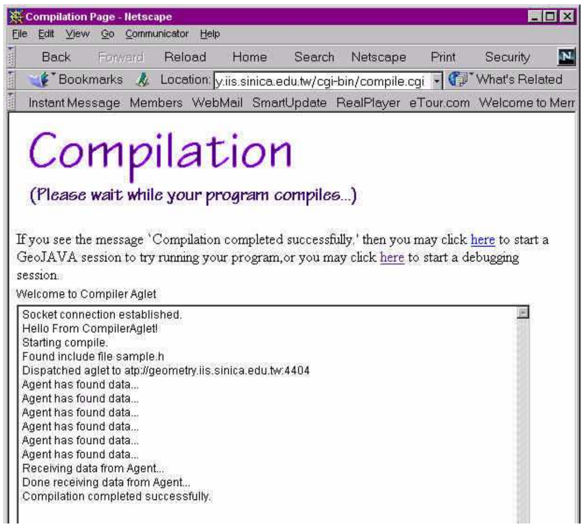

Fig. 2. Agent Compiler GUI

3.4 Registration Tool

The Registration Tool is a Java application that the user runs when he/she wishes to make a library available to the agents. Once a library is registered into the system via this Registration Tool, agents can find the new library when they arrive at the host through the Context providing the registered information. In the case that a new host is being created, the host can be added to the system during a compilation, where, in the beginning, the Compiler Agent will ask if any new hosts are to be added, and the user can input the information at that time.

3.5 The CORBAizer

Although the compilation agents can compile with static libraries, compiling with shared, or dynamic link, libraries is a different story. Because shared libraries are only useful (at the time of this writing) in a system where they are loadable into the address space of the running executable [8], they cannot be used in a distributed system. In addition, we still have architecture issues where libraries on different architectures cannot be compiled together, even if they have symbol definitions to offer. Fortunately, these issues dissolve with the *CORBAizer*.

The *CORBAizer* is based on the Exerciser Generator introduced in the RunClass tool[10]. This tool dynamically instantiates objects through the user's commands via the GUI and allows the methods within these objects to be examined. In order for a library to be inspected, the header files of the library had to be parsed and converted into a format that the system's engine could understand. The CORBAizer is based on this parser that is used in the RunClass tool.

The *CORBAizer* parses the header files of a library and generates two sets of C++ code: server code and client code. These source code files can be compiled separately, even on separate architectures, and used to communicate with each other via CORBA. The CORBA implementation that we used was MICO. Once the CORBAizer generates the server and client source files from the header files, the server code is compiled with the original library corresponding to the input header files, and the client code is used to generate a new library that is registered with *DISPE*, as was described previously. The server code can either be added to the CORBA Implementation Repository, which is a daemon that will automatically call the server when a request for a method in its library is called, or the server can be run manually to wait for requests from clients. Then, once the client library is added to the system, any program using the classes and methods defined in the original, possibly shared, library will be compiled with the client library just generated, and when the program is executed, the user's program will use CORBA to make a connection with the server program which executes the appropriate methods remotely. The user need not be concerned with any details of the CORBA implementation since it is all handled by the client library.

The basic idea behind the *CORBAizer* is that from a library's header files, a "skeleton" for the library's contents can be retrieved, which is used to generate the server and client code. The client is a copy of the skeleton with the implementations of the objects replaced by CORBA calls for searching for and connecting with the server and invoking the "real" methods. The server code takes advantage of CORBA's *tie* feature [6], which allows legacy classes to be wrapped by a CORBA class that takes the legacy class as the object of a template. The legacy class is then called when the CORBA class in the server receives a request for a method invocation.

4 Conclusion and Future Work

DISPE has great potential in paving the way for a new style of programming. Much of the latest and most solid technology in computer programming has been incorporated into the system, such as Java and CORBA. This should prove to be a plus for *DISPE* as it has been built on technology with a solid foundation.

The use of Java-based agents is indeed an innovative concept, and one that can grow and be useful for years to come. There are no signs of the C/C++ language weakening in the future, despite the growing use of Java. Thus, the integration of two of the most popular languages today in a distributed system should be very beneficial for C/C++ programmers. Remote compilation, whether using the "Traditional" or "Agent Compilation" frees users from worrying about the setup of libraries and include files and their directories, allowing them more time to focus on programming. The remote execution is all location transparent, where groups of users at remote sites can easily demonstrate geometric algorithms to one another, and the CORBAizer is especially useful in allowing remote libraries to be accessed during a program's execution. The

CORBAizer should be very practical for users of existing legacy libraries written in C/C++. It is not easy to manually develop CORBA libraries from scratch, let alone to port a legacy library to CORBA. Thus, the CORBAizer will definitely be useful for all programmers in any field where there is a need to integrate legacy systems with CORBA. Finally, remote debugging capabilities, a necessity for most programmers, complete the *DISPE* package. No known system provides such a complete programming framework.

In determining how we were to implement the search for hosts and libraries ("searching" for symbols is performed once an agent reaches a remote host and access a library), at first, the lightweight directory access protocol (LDAP)[19, 18] seemed very attractive. The attractiveness was compounded by the fact that Sun Microsystems also announced the availability of the Java Naming and Directory Interface (JNDI)[17] which supports LDAP. Therefore, as more hosts and libraries are added to the system, the use of LDAP may become useful in implementing the Directories used in the system. This would address versioning and consistency issues related to the header files and libraries introduced into the system.

As in any distributed system, the issue of security needs to be addressed, especially in the compilation and execution of programs that use foreign libraries. Both the code being compiled and the libraries registered in the system need to be checked for any potentially dangerous code such as system calls. In approaching this issue, agents may use a little more intelligence in compiling the user's code, perhaps by detecting any potentially dangerous methods and checking with the user if the detected code should be compiled into the program. A similar procedure may be followed during the registration and/or CORBAization of libraries.

The CORBAizer that has been implemented generates server and client code that is compatible with MICO's CORBA implementation. So future work can be put into generating code for other CORBA implementations as well.

The Registration Tool currently reads three sets of files to determine the libraries and include files available to the compilation agents. However, in the case when large lets of libraries and include files are residing on a system, it may be more efficient to use a database. Therefore, the Registration Tool may be replaced by a database in the future.

References

1. K.F. Aoki, D. T. Lee, *Towards Web-Based Computing*, accepted to Int'l Journal of Computational Geometry and Applications, Special Edition, 1999.
2. O. Etzioni, D. Weld, *A Softbot-Based Interface to the Internet*, Communications of the ACM, July, 1994.
3. D.B. Lange, M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison Wesley Longman, Inc., 1998.
4. D.T. Lee, C.F. Shen, S.M. Sheu, "GeoSheet: A Distributed Visualization Tool for Geometric Algorithms", *Int'l J. Computational Geometry & Applications*, **8,2**, April 1998, pp. 119-155.

5. D. Milojicic, W. LaForge, D. Chauhan, *Mobile Objects and Agents (MOA), Design Implementation and Lessons Learned*, The 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe, New Mexico, April, 1998.
6. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Revision 2.2, OMG Technical Document 98-07-01.
7. Object Management Group. 1998. *Mobile Agent System Interoperability Facility - OMG Revision Task Force Document*. Framingham, MA: Object Management Group.
   ftp://ftp.omg.org/pub/docs/orbos/98-03-09.pdf.
8. *Solaris Linker and Libraries Guide*, Sun Microsystems, Inc., 1997. http://docs.sun.com:80/ab2/coll.45.4/LLM/@Ab2TocView?
9. R.M. Stallman, *Using and Porting GNU CC, Version 2.8.1*. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA, 02111-1307, 1998.
10. T.R. Chuang, Y.S. Kuo, C.M. Wang, *Non-Intrusive Object Introspection in C++ – Architecture and Application*, Proceedings of the 20th International Conference on Software Engineering, pp. 312-321, Kyoto, Japan, April 1998.
11. The Agent Society. http://www.agent.org/.
12. CorbaScript. http://corbaweb.lifl.fr/CorbaScript/index.html.
13. CorbaWeb. http://corbaweb.lifl.fr/index.html.
14. Cetus Links - CORBA.
    http://www.cetus-links.org/oo_corba.html#oo_corba_projects.
15. DOMIS. http://www.mitre.org/research/domis/index.html.
16. GOODE. http://corbaweb.lifl.fr/GOODE/index.html.
17. Java Naming and Directory Interface (JNDI).
    http://java.sun.com/products/jndi/.
18. An LDAP Roadmap & FAQ.
    http://www.kingsmountain.com/ldapRoadmap.shtml.
19. Lightweight Directory Access Protocol (LDAP) FAQ.
    http://www.critical-angle.com/ldapworld/ldapfaq.html.
20. MICO. http://www.mico.org/.
21. Cetus Links - CORBA - ORBs.
    http://www.cetus-links.org/oo_object_request_brokers.html.
22. Software Agents Group. http://agents.www.media.mit.edu/groups/agents/.
23. TAO. http://www.cs.wustl.edu/~schmidt/TAO.html.
24. Voyager. http://www.objectspace.com/voyager.