A Multi-Agent Distributed Scripting System for COTS-Based Distributed Software Integration

Jim-Min Lin[1]*, Hongji Yang[2], Guo-Ming Fang[1], Che-Tai Lee[1] and Wei-Tsong Lee[3]

[1]*Department of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan 407, R.O.C.*
[2]*Software Technology Research Laboratory, De Montfort University, Leicester, England*
[3]*Department of Electrical Engineering, Tamkang University, Tamsui, Taiwan 251, R.O.C.*

Abstract

This paper presents an approach to distributed commercial off-the-shelf (COTS) based software integration by using the concepts of a multi-agent system and the distributed scripting mechanism. COTS software products are increasingly used to be software components in large-scale systems. Most organisations try to gain the promises of rapidly development and lower cost from reusing COTS components. Nevertheless, COTS-based software system development needs an efficient and useful integration approach. We developed a multi-agent system as an execution platform for distributed COTS software products. Instead of an RPC-like invocation approach, we adopt mobile agents to interoperate COTS software. We also developed a scripting mechanism for helping the software integrator to write a gluer. By using our scripting language constructs and the associated rules, a software integrator can easily write various scripts in various styles. To illustrate this multi-agent system, a distributed CPU-utilisation data collection system, is experimented in our study. Finally, we also successfully developed a graphical user interface tool that would be beneficial and useful for software integrator to edit, debug and display script programs and results.

Key Words: Software Reuse, Distributed Software Integration, Multi-Agent Distributed Scripting System (MADSS), Scripting Language, COTS

1. Introduction

The use of COTS software products in developing software systems is on increase. We have noticed many studies being carried out in COTS-based systems and initial conclusions show that these systems do make the cost of initial development lower and the promise of time to market shorter. The most practical issue on COTS-based software systems is integration of COTS components [1–4,23,24]. Though COTS software components can be sufficient building blocks, an effective mechanism and useful technique for component integration are still needed. Recently, it is realised that *Scripting* is a useful tool for gluing existing software [5,6]. A scripting language is different from programming languages like *C*. A scripting language is typeless and provides a simpler grammar, which enables an easy and quick integration code to assemble diverse software components. Thus a useful scripting mechanism can be a means to meet the requirements of efficient COTS software integration.

A scripting mechanism often consists of a scripting language, an executing platform and a specification of object model. For example, UNIX shell [5] is a popular scripting language for a UNIX expert to write scripts to glue pipes, which are the common components in UNIX.

*Corresponding author. E-mail: jimmy@fcu.edu.tw

Tcl/Tk [7] is another scripting example under UNIX to develop X Windows applications. Similarly, Visual Basic scripting was proposed by Microsoft and is to serve as a programming tool allowing a programmer put visual components into a container and then transform it into a Windows application. However the above instances are based on a single centralised operation. Because software development has moved towards a distributed component environment, to facilitate a high-level task execution through scripting, it will be beneficial to extend a scripting mechanism into a distributed and heterogeneous system, particularly the Internet environment.

There existed basic scripting mechanisms to develop distributed software systems, especially for Internet applications. Microsoft ASP [5] selects VBScript or JavaScript as scripting languages to glue distributed ActiveX, which is an object model commonly applied on Windows applications. Nevertheless this technique is proposed for Microsoft products and is not suitable for COTS software integration. On the contrary, the existing popular object frameworks, such as CORBA [8,25] and Java RMI [9], support many general platforms but no scripting language is available for this. GScript [10], an academic research, is a scripting language that extended from Microsoft Visual Basic, which tries to provide convenience and efficiency for Basic experts. Furthermore, GScript can inherit lexical and grammar rules from Basic. The purpose of GScript is to integrate Microsoft COM and CORBA objects into application logic. By combining the properties of CORBA and COM, the GScript research team has been trying to obtain two benefits, cross-platform and event service support.

In this paper, a multi-agent based distributed scripting system (MADSS) is proposed to achieve distributed software integration based on agent cooperation. In addition, MADSS provides a scripting language to software engineers so that they can perform an application project rapidly through the typeless and command-level scripting attributes [5,6]. In order to achieve the goal of distributed scripting, we adopt an agent-oriented scheme for diverse software integration. An agent in our system is a special software application, which has mental state and flexible interactive abilities for other entities such as human and other agents. Hence, an agent-oriented scripting system will bring at lease the following features:

- More intelligent abilities to deal with tasks – An agent is an intelligent computational entity and has deterministic autonomy. It can decide whether to execute or to deny incoming requests according to the conditions specified by its designers.
- Diverse software tool integration together with designated high-level communications – An agent-oriented system supports high-level communications as the gluers between diverse software since an agent interacts with underlying environment by using semantic messages. However, it is still necessary to consider the lower level access to software. Fortunately, there exist several well-designed products, such as CORBA and Java Remote Method Invocation (RMI), to support this. These products are based on RPC-like communication to build a client-server scheme.
- Load balancing – This is an inherent benefit from agent properties. Since a multi-agent system emphasises cooperative computation, the working loads can be easily distributed to various agents on the network and load balance can thus be achieved. For example, a server can deny an incoming request and suggest other candidates for this request if this server is aware of its overloading. Usually, the exception handling and fault tolerance processing of agents are the responsibilities of clients.

To build such an MADSS, we adopt Java as the agent programming language. Additionally, we use Knowledge Query and Manipulation Language (KQML) as the agent communication language. One important feature of MADSS is to incorporate mobile agent technology. A mobile agent is a special software agent, which can migrate in a distributed system. This added property makes distributed scripting system higher flexibility, because a program can be migrated to a remote host and execute locally [11–13,26,27]. Two advantages are obtained by using mobile agents on software integration: (a) a mobile agent can be platform-independent and thus is suitable for integrating software on heterogeneous systems and (b) the network communication traffic can be lowered if the size of messages between agents is greater than a mobile agent and we can move this agent to a remote site for execution.

The rest of this paper are organised as follows. Sec-

tion 2 describes the design of MADSS, which will be based on design considerations from viewpoints of system level and user level. Section 3 will give a case study as our demonstration and introduce a graphical tool which supports software engineers to easily write scripts and manipulate MADSS. Finally, we will conclude this research and describe our future research in Section 4.

2. Design of MADSS

In this section, we describe the design of MADSS. The purpose of MADSS is to demonstrate the feasibility of integrating existing software tools on heterogeneous platforms (Operating Systems (OSs)) through mobile agents' cooperation. The experimental OSs can be Microsoft Windows and UNIX typed systems such as Linux and FreeBSD because there exist many useful and popular software tools on these platforms. Two main works should be completed in an MADSS: a multi-agent based distributed software integration system and a scripting language for a software engineer to write integration script code. Hence in this section, we will firstly give an overview of MADSS and then discuss the design considerations from both the system point of view and the user point of view. Based on the discussions, we will describe how to integrate heterogeneous distributed software using an MADSS scripting language.

A client agent on the front-end platform is responsible for interacting with user and accepts user's scripts. Then, the client agent can generate one or more slave mobile agents and delegate these slave agents the designated tasks. The execution scenario is based on the control-flow specified in a script and is performed through the cooperation of multiple agents.

To implement MADSS, several design issues should be considered from the viewpoints of system designers and software integrators respectively. From the system designer's view, this should include two fundamental tasks:

- **Constructing a heterogeneous software integration system as the MADSS base**. The software integration support is the basic requirement in designing MADSS. In MADSS, a wrapper technique is used to bridge various existing software components to a common system integration platform. The use of wrapper makes it easy to invoke/trigger an integrated tool by using these tools' command streams. By making the use of wrappers, an off-the-shelf software product can thus be transformed into software components and be easily accessed by MADSS clients. Consequently, diverse software components can be integrated into an application.
- **Having a multi-agent system as the MADSS operational environment.** A multi-agent system is the core of an MADSS. This multi-agent system is in charge of providing an environment for the interactions among software tools. Typically, we use RPC-like communications between two software entities. In MADSS, however, the integration among software entities is through the software agents' interactions in a speech-act manner [14,15].

On the other hand, from a software integrator's view, MADSS should provide a scripting language to a software integrator such that he can easily write integration code. The software integrator would like to find an efficient way to glue these provided services together rapidly when he uses MADSS. Thus MADSS scripting language supports the constructs to allow integrator to write an execution scenario to manipulate several software agents. These scenarios are to be application logic of the integration code and each scenario specifies how to manipulate COTS software products. Consequently, A script can then be transformed into agent communication language representations, i.e., KQML messages. Furthermore, this distributed scripting language should also provide statements for users to arrange their agents' itinerary planning and data transfer scheme for performing the delegated tasks.

2.1 Integration of Commercial Off-the-Shelf Software Products

This subsection describes our solution to an essential COTS-based software integration problem. We will describe a wrapper technique and then its applications to integrating software on various platforms.

2.1.1 The State of the Art

The key solution to dealing with the COTS-based software integration is to eliminate interface incompatibility. Two types of commonly used solutions are *adap-*

ter [16] and *wrapper* [16,28]. Adapter is a source-porting technology, where e.g. a software component's interface can be converted into another interface expected to a specific system framework. This solution is commonly employed in middleware frameworks, such as CORBA, DCOM and Java connector architectures. In this kind of solutions, we assume that the COTS vendors might open the APIs of adapter and a software integrator referring to these APIs for writing the glue. In Java connector architecture [17], this kind of adapter is called the *resource adapter* and a popular example is JDBC [18]. JDBC makes many database systems easily be integrated into a Java application server. Therefore, a database system could be regarded as a traditional COTS software product and could be integrated into JDBC architecture. Nevertheless, the limitation of requiring an open API, like JDBC, shows that adapter may be not suited to COTS-based software integration. Unfortunately, the source code and APIs of almost COTS software products and legacy software components are unavailable. If we have to integrate a blackbox-like software building block, wrapper might be a suitable solution.

In literature, wrappers are software modules that are responsible for bridging individual information sources modules to external users. Many wrappers are used to convert data and queries from one model to another in the database community. Recently, wrapper is regarded as a module that can connect two software components. In another words, we employ a wrapper to enclose COTS software product and add an isolated and portable interface. This technology makes COTS software product portable to lots of platforms and could also be regarded as a software component. In MADSS, we employ wrapper technology as the essence of COTS-based integration.

Another essential technique on component-based software integration is the scripting language. As a software integrator writes a gluer, he may like to use a scripting language. ASP and JSP are two well-known examples in developing Internet applications. The results of software industries show that ASP and JSP really have increased Internet application productivity.

2.1.2 MADSS Wrapper

Wrapping is a common technique used in the research of component-ware. The essence of the technique is to use a "wrapper" program to cover around a component. Then the primary control path between the wrapped component and its "user" is wrapper. By using this technology, we can wrap programs or software packages with a standard interface. Then the wrapped software package forms an object and can be executed under a specific platform such as CORBA. Figure 2 depicts the notion of MADSS wrapper.

In Figure 1, the wrapped component is a COTS-based component, of which its computation unit is a COTS product. A wrapper is an artificial module, consisting of three sub-units – *Middleware connector*, *Synchronisation* and *Adapter*. Adapter is responsible for adapting the input/output interfaces of a component. Adapter accepts the input command strings received using a COTS-based software component and then invokes corresponding operations in the wrapped COTS products. Upon completion of the invoked operations, Adapter will receive the results and reassemble them into the forms appearing on the COTS-based software component interface.

Many COTS software products were originally designed for supporting a single-user operating mode. However, to allow a COTS-based software component to provide services for users on the Internet, a COTS-based software component should support a multi-user operation environment. To deal with this problem, Synchronisation (Figure 1) would be required to maintain data consistency, fault tolerance and security checks for the COTS software, so that many users can correctly operate on an originally single-user application. Middleware

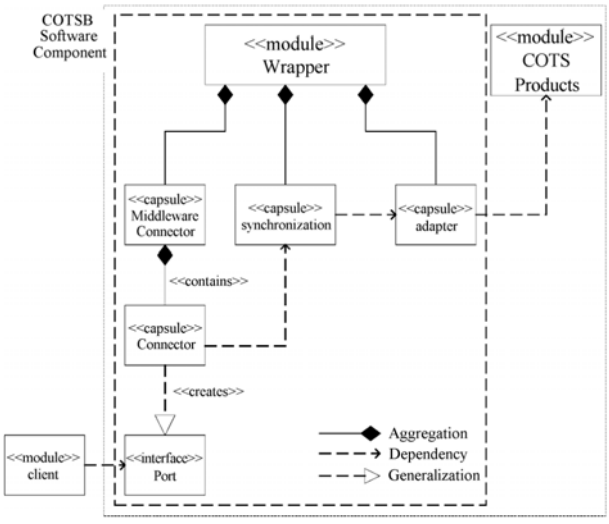

Figure 1. Wrapping.

Connector might contain several component interface modules, i.e., the connectors and invoke a suitable connector depending on what middleware is installed. A connector plays the role of a communication port to the client sites.

2.1.3 Wrapping of Microsoft Windows Application

In the MADSS wrapper, implementing adapter unit (as shown in Figure 1) is a complicated job. For different operating systems, the adapters differ greatly. In this subsection, we are going to describe the Windows adapter.

An adapter accepts user requests and transforms them into corresponding operations of the specific software. Microsoft Windows products have similar input/output operations. We adopt the *clipboard* provided by Microsoft to intercept the external calls and direct them into software products. The clipboard can be regarded as a channel to let a program to access Microsoft Windows applications. However, making black-box software accessible is the first step. The second step is to expose the public functions on an added interface.

JNI technique, an interface written in Java, is used to cope with the methods written in other programming language like C++. Writing an RMI interface is not difficult, but it is complicated to write an adapter. Fortunately, the modules of MADSS adapter are reusable. Thus the complexity of writing an adapter would be acceptable. We have successfully reengineered MS-Windows software packages into the COTS-based components executing under various middleware [19–21].

2.1.4 Wrapping of UNIX Applications

The design of an adapter for a UNIX wrapper differs from that of Windows wrappers. In this scheme, we adopt the concept of a "sandwich adapter". Figure 4 depicts the structure of such a sandwich adapter.

The term, "sandwich", means that there are three parts in the structure, *Front Cover*, a *Back Cover* and a *UNIX Application*. A Front Cover program and a Back Cover program cooperatively wrap an existing UNIX application. To achieve this, a UNIX pipe is employed to send data/instruction streams that are needed to trigger from Front Cover to a UNIX application.

The interaction among these units involves the following steps: the Adapter first accepts input messages from client and then redirects them to Front Cover process through a UNIX message queue. Front Cover process will then send them to a UNIX application through a pipe. After the UNIX application completes the requests and outputs results, the results will be redirected to Back Cover process through a pipe again. Finally the Adapter gets results from Back Cover through the message queue.

By using a sandwich adapter, an off-the-shelf UNIX component can be wrapped as an accessible software component. A wrapped component can also be added a designated object interface and be executed under a specific middleware such as CORBA, Java RMI and so on.

To form a "sandwich" structure, Back Cover should firstly fork a child process and then the Back Cover connects this newly created process's file '1' (i.e., *stdout*) to the Back Cover process's file '0' (i.e., *stdin*). In the same way, the child process forks a grandchild process and then connects the grandchild's *stdout* to its own *stdin*. Finally, the grandchild executes as Front Cover process and child process uses *exec()* to execute a UNIX application.

To summarise our wrapping technique: firstly, instead of putting the focus on the issues of selecting and identifying a suitable COTS component, we focus on the software integration; secondly, we have shown the design of wrappers for two popular operating systems (similarly, the concept of wrapper can be applied to other platforms); and finally, by providing different programmatic interfaces, the wrapped applications can be executed for clients under various middlewares.

2.2 Multi-Agent Support

The core of the proposed MADSS is a multi-agent system. This multi-agent system is responsible for executing scripts and providing an environment for the interactions among diverse software tools in a network. It consists of four kinds of software agents (see Figure 2). There are three reasons for the proposal of combining the concepts of an agent system with software integration. Firstly, an agent's mobility facilitates the integration of distributed tools, because a mobile agent glues a software tool with a remote one by bringing requests/results around distributed tools. Secondly, multiple mobile agents act like multithreads executing with diverse scenario styles in a distributed system. This means MADSS can have more kinds of behaviours than traditional distributed scripting mechanisms in which only few ones are supported. Thirdly, MADSS possess features of higher reli-

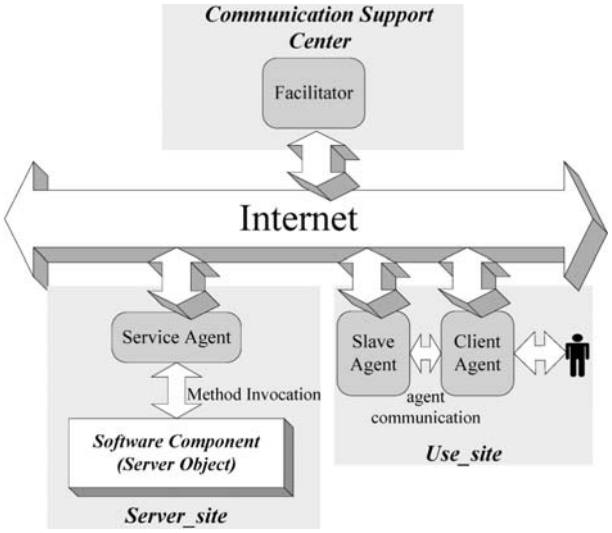

Figure 2. Structure of MADSS.

ability and load balancing. Since an agent can have the ability of knowing environment situation, it can go to another site with the same service as it encounters troubles in site failure or site/service busy.

Next we focus on how to construct a multi-agent system and how to execute scripts through the cooperation of software agents.

Four kinds of software agents in MADSS are *Service agent*, *Client Agent, Slave Agent* and *Facilitator*. The services wrapped by off-the-shelf software tools were maintained by a service agent. The execution scenario written in MADSS scripting language is executed by the Client Agent. Client Agent gets tasks from user script, translating a script to the corresponding KQML messages and then delegates the tasks to the mobile Slave Agent. Then Slave Agent can move to a remote site and request service agent the desired services. In our design, agents' communications are based on the KQML protocol. An agent can talk to other agents to query the desired knowledge including the public services wrapped from existing software.

To implement a multi-agent platform, after examining several multi-agent products like IBM Aglet, Jackal, etc., we adopted IBM's Aglet as the target multi-agent platform. We will introduce the agents and address how to let Slave Agent access remote software.

(1) Service Agent

Service Agent is responsible for maintaining the interfaces of wrapped software applications on a machine. Whenever a newly wrapped software application is adapted to the host's service, Service Agent will advertise it on the Facilitator Agent. The details of the facilitator will be introduced later. During the execution phase, a mobile slave agent will migrate to the Service Agent's site and communicate with Service Agent using KQML messages. A KQML message contains Salve Agent's parameters and translates the parameters into corresponding data type of the Service Agent. Service Agent has to extract the parameters/input streams from KQML message. This is a document-based message passing process. When the input streams were extracted, the Service Agent uses typical method invocation technique to redirect these streams and triggers the wrapped software application.

(2) Client Agent and Slave Agent

For an agent system design, Lange invented a useful scheme to decompose a big task into several subtasks performed by software agents. This scheme is a master-slave pattern [22]. In this pattern, a stationary agent, *master*, can delegate tasks to several mobile slave agents in parallel. System performance will be the main advantage as we apply this pattern. While slave agents migrate to remote sites, the master agent can continue its operation.

In MADSS, Client Agent plays the role of the master agent. It interacts directly with human users. Client agent accepts user script and displays execution results by providing a user interface, for which users can input the scripts to arrange the controls of remote services. Client agent is also designed as an interpreter. The scripts written can be interpreted into a KQML message and the slave agent can carry this message to destination.

Slave Agent is used to request service agent. It interacts with Service Agent by using KQML messages. To migrate to a network environment, Client Agent provides knowledge about itinerary to Slave Agent. Moreover, if Slave Agent has to carry any file, the file's information such as file's name has to be included.

(3) Facilitator

To gain high scalability in a distributed system, it is necessary to reduce the proliferating interconnections among distributed components. To achieve this feature, the agents in a multi-agent system must freely collaborate without the direct knowledge of agent existence. To cope with this design issue, a *Facilitator* program can be applied.

In our work, Facilitator has the following basic functions.

- Preserve the registered agent names.
- Assist an agent in looking for an appropriate helper agent.
- Assist an agent in delivering the message to the correct destination.

In MADSS, a facilitator is assigned jobs to store and maintain the services exposed from service agents. A slave agent asks the facilitator for the public services advertised by a remote host. If this action is successful, a slave agent can obtain necessary information, such as the I/O type and the location of the remote host, for remote execution. A slave agent can then move to the destination and does its jobs.

2.3 MADSS Scripting Language

The proposed MADSS scripting language is designed by referring to KQML. Most of the constructs in MADSS scripting language are used to describe agent's operations. To write an MADSS script, a programmer should give two kinds of contents: the type of agent execution and the details of task delegation. The first part is regarding to how to initiate a software agent and what the agent should perform. Then the construct "*execute*" will initiate an agent's execution. Furthermore, if a job consists of more than one agent and these agents should be executed in parallel, the construct "*pexecute*" can be used. We will introduce aforementioned constructs as follows:

- astart [*agent_name*] {*task1, task2, …*}

"astart" denotes the start of an agent. It specifies which software agent will be initiated and what tasks this agent will perform. "agent_name" indicates the agent's name and *task* indicates a software service that is made by wrapping of-the-shelf software packages. After delegating tasks, a mobile software agent can migrate to a remote host and invoke desired services. A task stays in a remote host and goes to a next one after the completion of the invoked service. "astart" does not actually specify the content of tasks. Another construct "delegate" was given to specify the contents of a task. As the delegated task has completed, "dispose" is given to kill a processing agent. The two constructs will be introduced later.

- execute [*agent_name*] *-parameter*

An agent will start its execution by initiating construct, "execute". Agent-name denotes the executing agent's name.

- pexecute [*agent1_name, agent2_name, …*] *-parameter*

Like "execute", "pexecute" indicates the execution of an agent and it can be used to execute several mobile agents in parallel. A big task can be subdivided into several sub-tasks and executed in parallel. Hence in MADSS, a task can delegate several mobile agents by using "pexecute".

"execute" and "pexecute" have two parameters, "now" and "schedule". Two parameters specify the time to execute.

- *-now*
  Option *now* notifies a mobile agent to start its task immediately.
- *-schedule* [*time*]
  If we hope to schedule the execution of an agent, this option is used to set the delay time. The default unit of the *time* is second.

In addition to above constructs, two constructs "dispose" and "delegate" are adopted to specify the content of a task. "dispose" is used to stop an agent's execution. This construct can be regarded as a specific task an agent has to perform. Another construct, "delegate", is used to define the content of a task. This construct specifies which host an agent should go and what services it should invoke. "Delegate" provides several operational parameters described as follows. The syntax structure of "delegate" is:

- delegate [-[*option*] value] …

the parameters may include:

- *-d* destination
  Option *d* specifies the remote host's address where the delegated mobile agent migrates. "Delegation" can be in form of either an IP address or an Internet domain name.
- *-n* service agent name
  Option *n* specifies the remote host's name. Because of open environment, MADSS adopts name register strategy. Moreover, we use KQML to be ACL, each remote host maintain a service agent. Service agent manages services and waits for mobile agent's requests. Whenever the mobile agent arrives at the remote host, it has to interact with a service agent by using KQML. KQML

encapsulates input data and service agent directs data to invoke services.

- *-s* target services
  Option *s* specifies the content of a task that the mobile agent should perform. The service is provided by a remote host and the integrator does not need to know how this service was created. This service exposed its interface on the multi-agent environment. Thus the mobile agent carries correct input data and interacts with this service. Finally, mobile agent must carry the results back to the home host.
- *-c* message content
  Option *-c* specifies the contents in KQML messages to a remote agent. It is actually the input's content. For example, if we would like to initiate a mathematical function, "-c" might specify a mathematic equation.
- *-f* file name
  Option *f* specifies a file name that will be sent to remote agent. For some occasions, the remote service needs a large amount of data to be processed. A mobile agent can carry a file with these data to the remote host. The file's name will be translated into a parameter that is filled into a field of KQML message.
- *-r* file name
  Option *r* specifies a file name that delivers the processing results to another agent.
- *-t* time to timeout
  Option *t* specifies the time an agent will expire. This option will enforce an agent to dispose if it cannot finish the designated task in expire time.

We can easily show an instance to describe how to use these constructs to form a script:

```
astart agentA {
    delegate -d atp://stewart.iecs.fcu.edu.tw -n
    Unix_agent -s UnixCommand -c who > result
    -r $result1 -f null -t 240
    delegate -d atp://home.iecs.fcu.edu.tw -n
    clientAgent -s home
   dispose
            }
execute agentA -schedule 120
```

In this example, a mobile agent named *agentA* is initiated. The purpose of agentA is to gather the data of logon-users in a remote UNIX machine by initiating a service "UnixCommand" in a service agent "Unix_ agent". To achieve this goal, a UNIX command, "who > result", is used. This command redirects the logon-user list into a file "result". After the completion of the designated task, agentA returns to its original site and invokes "home" function to save result file in the client site.

This example shows an agent is delegated a simple task with a linear and straightforward scenario. To complete complicated tasks with agents, diverse execution scenarios for mobile agents are desired.

Parallel execution is a basic and important execution style in a distributed system. In this case, an MADSS construct "pexecute" asks more than one mobile agents to execute the corresponding tasks simultaneously on different sites. As mobile agents complete their execution, they can return to home site and merge their execution results.

To write this type of script, the software engineer has to delegate two or more slave agents. Each agent will get its own job and move to its own destination. Thus the structure of this type is similar to the following form:

```
astart SlaveAgentA {
    delegate -d Target_Remote_SiteA -n Service-
    AgentA -s ServiceA -c InputContent -f Input-
    File -r ResultA
    dispose
        }
astart SlaveAgentB {
    delegate -d Target_Remote_SiteB -n Service-
    AgentB -s ServiceB -c InputContent -f Input-
    File -r ResultB
   dispose
        }
                ... /* other slave agents' delegation */
pexecute SlaveAgentA, SlaveAgentB
```

3. Implementation

In this section, we are going to give an experimental MADSS system, a distributed CPU-utilisation data collection system, to demonstrate the feasibility of the proposed concept. The example system is to gather CPU-utilisation charts of several remote hosts by delegating a mobile agent to remote sites and integrating a Microsoft

Windows COTS software application, Hardware Manager, on each host.

3.1 Experiment Environment

The experimental system is configured with 3 personal computers (equipped with Pentium 4 2.8G, 768MB RAM and MS-Windows 2000 Professional Edition operating system) connected with an 100Mbps LAN.

3.2 An Example

In this experiment, a facilitator and a client agent resided in the same machine and two service agents installed on the other two machines. Each service agent maintained its own service that can capture the CPU-utilisation data. In addition, one of the service agents maintains image processing service. In order to display the final result, home agent (i.e., client agent) is responsible for supporting displaying service.

At first, a script in the proposed script language constructs is developed as follows:

```
astart AgentA {
    delegate -d atp://stewart.fcu.edu.tw -n Server-
    Agent 1 -s CPU_Utility -c CPU_Utility1.jpg -t
    120
    delegate -d atp://galex.fcu.edu.tw -n Server-
    Agent2 -s CPU_Utility -c CPU_Utility2.jpg -r
    CPU_Utility2.jpg
    delegate -d atp://stewart.fcu.edu.tw -n Server-
    Agent1 -s PictureMerge -c CPU_Utility1,
    CPU_Utility2,Result -r Result.jpg
    delegate -d atp://home.fcu.edu.tw -n Client-
    Agent -s ShowPic -c Result.jpg
    dispose
        }
execute AgentA -now
```

In this example, a slave agent firstly moves to a remote site and inquires the information of CPU utility rate. The results will be saved as *"CPU_Utility1.jpg"*. Due to this service needs more time to execute, slave agent keeps moving to a next site to request service. After the execution of the second site completes and generates a file named *"CPU_Utility2.jpg"*, slave agent carries this file back to the first site and requests service *"PictureMerge"*. This service is to merge two charts into a single image. Finally, slave agent carries this image file back to the home site and displays it for users.

By executing this script, a client agent will firstly translate a script into the corresponding task list, including *dispatch* and *KQML message.* 'Dispatch' task type includes information about itinerary and destination. 'KQML message's task type is carried by a slave agent and is used to interact with service agents. A service agent will invoke the corresponding COTS-based component according to parameters.

In MADSS, we also implemented a graphical tool using Java. This tool is invoked by a client agent. There are two purposes for this tool: for providing a user graphical interface for MADSS users and for debugging a script.

The result of our experimental script shows a picture for CPU-utilisation of two remote sites (Figure 3).

MADSS might have more useful applications for system managers. For example, a system manager might do routine system administration works, like system

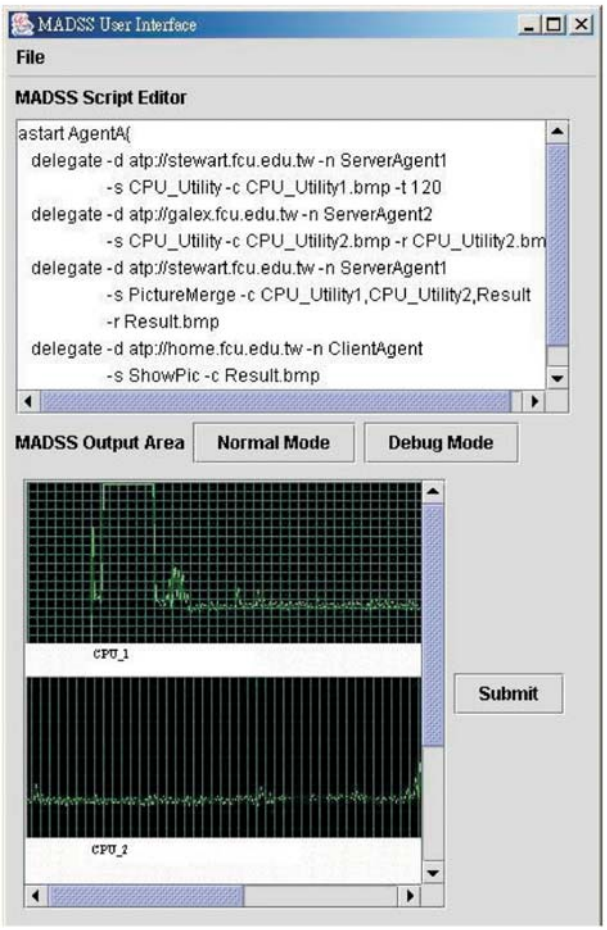

Figure 3. MADSS result display – normal mode.

shutdown procedure, disk data backing up, user account management and so on, for a cluster computer on a single site through writing an MADSS script.

In the example, the execution of the multi-agent scripting system involves the delivery of the slave agent, the communication among agents, and the operations of several COTS software. The overall execution time in the test environment was about 77.062 seconds. Most of the time is taken to deliver the slave agent and operate COTS software. The slave agent migrates among several computers. During the execution, the slave agent bringing the codes and pictures are delivered among the Aglet mobile platforms through network. It took about 0.984 seconds. The *CPU_Utility* and *PictureMerge* services involve the operations of the *Task Manager* and *MS Paint* software. A service may consist of several operations. Some extra time delays may also be required to support the synchronization and scheduling between the operations. Therefore, the operations of COTS software take larger proportion delay in the overall execution time. It took about 76 seconds. The remaining time is taken on the communication between the slave and server agents.

4. Conclusion

In this paper, we proposed a new approach to designing a COTS-based software system by using a multi-agent based distributed scripting mechanism. The multi-agent system is used as an operational platform for incorporating distributed COTS software components within a software system. We proposed an MADSS script language for software designers to guide mobile agents' operations and workflow. The results of an experimental system show that the proposed idea would be useful when a user wants to quickly build up a software system by reusing COTS software tool. From our experience in the given experimental system, a well-trained programmer could rapidly complete a simple CPU-Utilisation Gathering System in just three or less weeks.

In summary, four academic merits of this research are concluded as follows:

- We firstly propose the idea to achieve the goal of COTS software integration through the combination of a mobile agent system and a scripting mechanism.
- In MADSS, the integration of COTS software products is based on the relation among COTS software function services. Such a high-level integration mechanism would be easier to use for users.
- The software components in MADSS are loosely coupled and highly reusable because MADSS supports software reuse for heterogeneous systems.
- The behaviours of agents are controlled by a scripting language.

In the next stage of perfecting MADSS, it is planned to enhance a mechanism for discovering the correct services advertised on the network, to handle the problem of service's ontology, to present the multi-agent architecture through a formal design methodology, such as UML and to enhance the proposed MADSS script language with more diverse and useful control flow styles.

Acknowledgement

This research was supported by National Science Council under Grant NSC91-2213-E-035-024 and NSC 95-2221-E-035-076.

References

[1] Baker, Thomas G., "Lessons Learned Integrating COTS into Systems," *Proceedings of 1st International Conference on COTS-Based Software System (ICCBSS 2002),* Orlando, FL, USA, pp. 21–30 (2002).

[2] Davis, L. and Rose Gamble, "Identifying Evolvability for Integration," *Proceedings of 1st International Conference on COTS-Based Software System (ICCBSS 2002),* Orlando, FL, USA, pp. 65–75 (2002).

[3] Thomas Pfarr and Reis, James E., "The Integration of COTS/GOTS within NASA's HST Command and Control System," *Proceedings of 1st International Conference on COTS-Based Software System (ICCBSS 2002),* Orlando, FL, USA, pp. 209–221 (2002).

[4] Egyed, A. and Balzer, R., "Unfriendly COTS integration – instrumentation and interfaces for improved plugability," *Proceedings of 16th International Conference on Automated Software Engineering (ASE 2001),*

Nov., pp. 223–231 (2001).

[5] Ousterhout, John K., "Scripting: Higher Level Programming for the 21st Century," *IEEE Computer,* Vol. 31, pp. 23–30 (1998).

[6] Ousterhout, J., *Additional Information for Scripting White Paper,* Sun Micro Systems, http://www.sunlabs.com/people/john.ousterhout/scriptextra.html.

[7] Brent Welch, Ken Jones and Jeffery Hobbs, *Practical Programming in Tcl and Tk 4th Edition,* Prentice Hall PTR, ISBN 0-130-38560-3 (2003).

[8] CORBA Home Page, http://www.corba.org/.

[9] Java Home Page, http://java.sun.com/products/jdk/rmi/.

[10] Fu Yan, "GSCRIPT: A Script Language that Supports both COM and CORBA," *Proceedings of the 4th International Conference on High Performance Computing in the Asia-Pacific Region,* Vol. 1, pp. 558–562 (2000).

[11] Stavros Papastavrou, George Samaras and Evaggelia Pitoura, "Mobile Agents for World Wide Web Distributed Database Access," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 12, pp. 802–820 (2000).

[12] Paolo Bellavista, Antonio Corradi and Cesare Stefanelli, "Mobile Agent Middleware for Mobile Computing," *Computer,* Vol. 34, pp. 73–81 (2001).

[13] Sabrina De Capitani di Vimercati, Alessandro Ferrero and Massimo Lazzaroni, "Mobile Agents Technology for Remote Measurements," *IEEE Transactions on Instrumentation and Measurement,* Vol. 55, pp. 1159–1565 (2006).

[14] Jennings, Nicholas R. and Michael Wooldridge, "Agent-Oriented Software Engineering," *Proceedings of the 9th European Workshop on MAAMAW* (2000).

[15] Lin, J.-M., Hong, Z.-W. and Fang, G.-M., "MADSS: A Multi-Agent Based Distributed Scripting System," *Proceedings of the 26th International Conference on Computer Software and Applications (COMPSAC 2002),* pp. 578–583 (2002).

[16] Klaus Bergner Andreas Rausch, Mare Sihling and Alexander Vilbig, "Adaptation Strategies in Componentware," *Proceedings of Software Engineering Conference,* Australian, pp. 87–95 (2000).

[17] Java Home Page, http://java.sun.com/j2ee/connector/.

[18] Java Home Page, http://java.sun.com/products/jdbc/.

[19] Lin, J.-M., Hong, Z.-W., Fang, G.-M., Jiau, Christine H.-J. and Chu, Wiliam C., "Reengineering Windows Software into Reusable CORBA Objects," *Journal of Information and Software Technology,* Vol. 46, pp. 403–413 (2004).

[20] Hong, Z.-W., Lin, J.-M., Jiau, Hewijin C., Fang, G.-M. and Chiou, C. W., "Reengineering Windows-Based Software Application into Reusable Components Using Pattern Language," *Journal of Information and Software Technology,* Vol. 48, pp. 619–629 (2006)

[21] Lin, J.-M., Hong, Z.-W., Fang, G.-M. and Lee, C.-T., "A Style for Integrating MS-Windows Software Application to Client-Server Systems Using Java Technology," *Software Practice and Experience,* Vol. 37 (2007) (To appear).

[22] Aridor, Y. and Lange, D., "Agent Design Patterns: Elements of Agent Application Design," *Proceedings of 2nd International Conference on Autonomous Agents (Agent'98),* ACM press, pp. 110–115 (1998).

[23] Jin, Dean and Cordy, James R., "A Service-Sharing Methodology for Integrating COTS-Based Software Systems," *Fifth International Conference on COTS-Based Software Systems (ICCBSS),* (2006).

[24] Sihem Ben Sassi, Lamia Labed Jilani and Henda Hajjami Ben Ghezala, "Towards a COTS-Based Development Environment," *Fifth International Conference on COTS-Based Software Systems (ICCBSS),* (2006).

[25] Dionissis Vassilopoulos, Thomi Pilioura and Aphrodite Tsalgatidou, "Distributed Technologies CORBA, Enterprise JavaBeans, Web Services: A Comparative Presentation," *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06),* (2006).

[26] Wang, Z., Chen, Q., Gao, Chuanshan, "Implementing Grid Computing Over Mobile Ad-Hoc Networks Based on Mobile Agent," *Fifth International Conference on Grid and Cooperative Computing Workshops (GCCW'06),* pp. 321–326 (2006).

[27] Ibrahim, Mohammed A. M., "Distributed Network Management with Secured Mobile Agent Support,"

International Conference on Hybrid Information Technology (ICHIT'06), Vol. 1, pp. 244–251 (2006).

[28] Shin, Michael E. and Fernando Paniagua, "Self-Management of COTS Component-Based Systems Using Wrappers," *30th Annual International Computer Software and Applications Conference (COMPSAC'06),* Vol. 2, pp. 33–36 (2006).

Manuscript Received: Nov. 29, 2006
Accepted: Feb. 5, 2007