Mobile Software Testing over the Internet

Huey-Der Chu  
Department of Computing Science  
University of Newcastle upon Tyne  
2nd Floor, Bedson Building  
Newcastle upon Tyne, NE1 7RU, UK  
+44 191 222 5153  
Huey-Der.Chu@newcastle.ac.uk

Jianrong Chen  
School of Computing and Mathematical Science  
Oxford Brookes University  
Gipsy Lane Campus  
Headington, Oxford OX3 0BP, UK  
+44 1865 483708  
97102337@brookes.ac.uk

ABSTRACT

This paper presents a novel paradigm for software testing, mobile software testing, which applies the mobile agents to software testing and can test applications on remote sites. Testing tools with information are registered with the tools broker on the testing server side. Testers on the testing client side only need to launch a mobile testing agent to remote testing servers. The mobile testing agent will request the tool broker for testing services on the users behalf. The paradigm proposed shows a way to address software testing problems in a networked environment that fits more naturally with the real world.

Keywords: Mobile agents, computer aided software testing, software test environment

1 INTRODUCTION

Due to the rapid development of computer networking, the World Wide Web (WWW) has rapidly become the dominant platform for network computing, both Internet and Intranet. It has exponentially increased the possibilities for distributed computing, data distribution and collaboration. With the addition of Java, the WWW has embraced distributed objects and is evolving into a full application development and delivery platform. Therefore, researchers and developers have been constantly looking for ways to program the Internet in novel and efficient ways.

Mobile agents [2,3,4] provide a new alternative paradigm for distributed object computing on the WWW. A mobile agent is a computer object that can move through a computer network under its own control, migrating from host to host and interacting with other agents and resources in order to satisfy requests made by its clients. They may move around on behalf of their users seeking out, filtering and forwarding
information or even doing business in their own name. Therefore, mobile agents show a way to think about solving software problems in a networked environment that fits more naturally with the real world.

Software testing is characterized by the existence of many methods, techniques and tools, that must fit the test situation, including technical properties, goals and restrictions [9]. In practice, there is no single ideal software testing technique or testing tool alone for assessing software quality. Therefore, testers must ensure that the testing strategy is chosen by the combination of testing techniques and tools at the right time on the right work products.

In the past few years, the potential benefits of automated software testing has been more widely recognized therefore the demand for tools and Software Test Environments (STEs) to automate testing activities has grown rapidly. As a result, there are now many commercial or prototypal Computer Aided Software Testing (CAST) tools in existence [11]. However, the knowledge about the suitability of test methods, techniques and about the appropriateness of test metrics and test tools is limited. Therefore, many testers and users fail to recognize that selecting and implementing a tool involves much more than a quick shopping trip to the nearest CAST tool vendor. At the same time, this expensive tool could be redundant after just one project and become “Shelfware” which accrues unnoticed against the budget.

To address these problems, a novel approach, MOBILE Software Testing (MOST) through a Tools Broker (TB), is proposed. Based on the concept of mobile agents, it provides a mechanism, the TB, to enable the Tool Vendors (TVs) to register their CAST tools or STEs with the TB on remote sites. The testers or users only need to launch a Mobile Testing Agent (MTA) to ask for services from the TB on these remote sites. After the TB has selected the suitable tool vendor, the communication between users and TVs is through the MTA. Moreover, the Application Under Test (AUT) can be run on remote sites by the Network Class Server and send the testing report back to the testers through the MTA.

In Section 2 of this paper, the concept of mobile agents is introduced. The architecture of MOST is proposed in the Section 3. The consideration to implement MOST is described in the Section 4. Section 5 shows how to implement the MTA in distributed computing environment. Section 6 summarizes our research and offers suggestions for further study.
2 MOBILE AGENTS

An agent is an object that is autonomous enough to act independently even when the user or application that created it is not available to provide guidance and handle error [10]. Agents can receive requests from external sources, such as other agents, but each individual agent decides whether or not to comply with external requests. In the computer world, an agent is a computer program whose purpose is to help a user perform some tasks (or set of tasks) [10]. To achieve this aim, it maintains a persistent state and can communicate with its owner, other agents and the environment in general. Agents can do routine work for users or assist them with complicated tasks; they can also mediate between incompatible programs and thus generate new, modular and problem-oriented solutions saving work.

A mobile agent is an agent that can move through a computer network under its own control, migrating from host to host and interacting with other agents and resources in order to satisfy requests made by its clients. They are defined in formal terms by computer scientists as objects that have behavior, state and location. A subset of behaviors of every agent is inherited from the model notably those behaviors that define the means by which agents move from place to place.

The advantages of mobile agents are [2,3,4]: firstly, they offer an effective paradigm for distributed applications, particularly in partially connected computing; secondly, they can provide a pervasive, open, generalized framework for the development and personalization of network services; thirdly, they move the programmer away from the rigid client–server model to the more flexible peer–peer model in which programs communicate as peers and act as either clients or servers depending on their current needs and fourthly, they allow ad–hoc, on–the–fly applications that represent would be unreasonable investment of time if a code had to be installed on each network site rather than dynamically dispatched.

Nowadays, there are already some mobile agent technologies such as Aglet [1] and Java–to–go [8]. There are also many applications for which mobile agents are claimed to be well–suited, including information retrieval, data–mining, network management, electronic commerce, mobile computing, remote control and monitoring. However, there is no application available in software testing, particularly in applying for the CAST tools which are expected from the testers. Therefore, MOST is proposed in the next section.
3 THE ARCHITECTURE OF MOST

The architecture of mobile software testing is shown as in Figure 1, which consists of testing clients and testing servers. Testing clients are testers who want to make use of the testing services that the testing servers provide. The testing servers belong to vendors which provide various software testing services through the Internet.

![Figure 1. The architecture of Mobile Software Testing](image)

3.1 Testing Clients

The testing clients prepare for the AUT, which generally consists of a couple of classes in the Java case. The classes will be loaded dynamically into testing servers during testing by the MTA through the Internet with the help of a network class server.

When a client wants to test his application to begin with he has to register with the testing server. Afterwards, the client will launch a standard customized MTA into the testing server, which is used as a communication pipe between the server and the client.

During testing, the network class server is listening to the Internet, the MTA will ask it to find, load and transfer the classes to the testing server dynamically.
3.2 Testing Servers

There are four components in a testing server: a TB, TVs, a security server and MTAs.

The TB is a broker between the tools in TVs and the MTAs. Tools in TVs were registered with the TB. When a new tool comes out, it will register its information, such as its testing technique, testing strategy and the cost of the tools usage with the broker. The TVs is distributed in the sense that the tools in TVs may distribute over the network. When the MTA arrives at a testing server, it will request a specific tool set, the broker will select the suitable tool set and answer with the addresses of this tool set.

After getting the addresses of the specific tool set, the agent will communicate with the TVs directly. In this case, the MTA is only used as a pipe between the testing server and the testing client. When the agent asks the specific testing tool set for test, the tool set will send back a URL of a tool–specific specification editor to the agent. Users utilize the specific editor to enter their specification of AUT. The specification will save at a particular site which is transparent to users and available to the tool set. According to the specification, the TVs will test the application. During the testing the MTA will load the classes dynamically and pass them to TVs on demand.

The security server is listening to the Internet. When users want to test their software they have to register with the testing server, the security server will give a key to a user at that moment which the MTA will bring with it on its journey. When the agent arrives at the testing server, the server will check the key to see if it is valid. If it is valid, the process will continue, otherwise send back an error message to the user.

4 IMPLEMENTATION CONSIDERATION

In order to implement the architecture proposed above using Java there are four core technologies: the testing server supporting the mobile agent computing, MTA, dynamic class loading and communication between MTA, TVs and TB.

4.1 Testing Servers

The layers of a testing server is shown in Figure 2. When the testing server starts up, it forks a thread to
execute the receptionist, which in turn forks a thread to execute the porter.

![Figure 2. The layers of Testing Server](image)

Based on Java sockets and severSockets, the transmission provides semantics of the agent–packet transmission. The porter is monitoring the network to see if a new packet is coming. If so, the porter makes use of the methods provided by the transmission layer to receive the packet and pass it to the receptionist. The receptionist unpacks the packet and instantiates the coming MTA first, then checks if the key MTA it carries is valid. If not, the receptionist would send back an error message to the testing client. If it is valid, the receptionist would initiate the MTA, create its execution environment and pass it to the testing server. The testing server forks a new thread to execute the MTA.

### 4.2 MTA

The MTA template in the Java programming language is structured as follows.

```java
public interface AgentInterface extends Runnable {
    public void initiate(byte[] kbase);
    public void cleanup();
}

public class MTA implements AgentInterface{
    String name; // the name of the agent
    String key;
    protected ContextInterface context;
    File kbase_dir;
    ......
    public Agent(){
        ......
    }

    public void initiate(byte[] kbase){
        // the code to be executed in remote testing server can be override
        ......}
    //other methods......
```
It consists of a name of the MTA, a key which MTA carries, the MTA execution environment context and a knowledge base kbase_dir.

The MTA execute within a AgentContext which is the execution environment of the MTA. The MTA could make use of services in the testing server by the AgentContext.

The knowledge base includes initial information of the MTA and the information about the test requirement. Users can customize the MTA by changing the information in the knowledge base. When the MTA moves from the testing client to the testing server, the information in the knowledge base will move along.

When an MTA arrives at the testing server, the latter will initiate the former with the knowledge base. When an MTA leaves the testing server, it will clean up the environment. The initiation and cleanup methods are provided by the AgentInterface.

Dynamically, an MTA is a thread. When it moves to the testing server, a new thread is created on which the MTA will run.

### 4.3 Dynamic Class Loading

The principle of Dynamic class loading is shown as in Figure 3.

![Dynamic Class Loading](image)

Figure 3. Dynamic Class Loading

The network class server is listening on the Internet, when a network class loader asks for classes it will find, load and transport the classes. The network class server not only can load classes from the local class library
but can also load classes from a remote class library.

The layer structure of a network class server is shown as in Figure 4. Like the testing server, it is also based on Java Sockets and serverSockets.

<table>
<thead>
<tr>
<th>Network Class Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listener</td>
</tr>
<tr>
<td>Transmission</td>
</tr>
<tr>
<td>Socket and Server Socket</td>
</tr>
</tbody>
</table>

Figure 4. The layers of Network Class Server

In the context of the mobile software testing, when the receptionist in the testing server instantiates the coming MTA, it will load the classes relevant to the MTA dynamically. During the testing, the MTA will load the class from the testing client and then pass them to the TVs.

4.4 Communication between TB, MTA and TVs

Communication between the TB, the MTA and the TVs is based on Knowledge Query Manipulation Language (KQML) [5], which is a high–level language intended for the run–time exchange of knowledge between intelligent systems. The KQML was developed as a part of the Knowledge Sharing Effort.

Logically the KQML message consists of three layers: the content layer, the message layer, and the communication layer. The content layer includes the actual content of the message in the programs’ own knowledge representation of the message. KQML can carry expressions encoded in any representation language such as KIF, KQML or even ASCII strings.

The communication layer encodes a set of message features which describe the lower level communication parameters, such as the identity of the sender and recipient, and a unique identity associated with the communication.

It is the message layer that is used to encode a message that one application would like to transmit to another. The message layer forms the core of the KQML and determines the kinds of interaction one can have with a KQML–speaking agent. A primary function of the message is to identify the protocol to deliver the message and to supply a speech act or performative which the sender attaches to the content (such as an assertion, a
query, a command or any of a set of known *performatives*). In addition, since the content may be opaque to the KQML–peaking agent, this layer also includes optical features which describe the content language, the ontology it assumes and some type of description of the content such as a descriptor naming a topic with the ontology.

Syntactically, a KQML message is a ASCII string called a *performative*, which consists of a *performative’s* name and a list of its parameters. A parameter is represented as keyword/value pair. The keyword, that is the parameter name must begin with a colon and must precede the corresponding parameter value.

Here is an example of KQML messages, which is used as an initial message in our architecture:

```
```

In this message, the KQML *performative is evaluate*, the content is (tell–resource :type address :name TB :value glororan.ncl.ac.uk:5001), another KQML message which tells the MTA that the TB’s address is glororan.ncl.ac.uk:5001, the ontology assumed is agent, the receiver and sender of the message are MTA and init–file respectively, and the content is written in the language KQML.

The value of the content keyword is content level, the values of :sender and :receiver belong to communication level, and the *performative’s* name (evaluate) with :language and :ontology form message layer.

When an MTA moves to the *testing server*, it would transmit a message like the following to TB for telling the resource:
Suppose that there was already a TV1 which sent the following message to TB when it started up.

(evaluate :sender TV1 :content (tell-resource :type address :name MT A :value glororan.ncl.ac.uk:54100) :ontology software test :receiver TB :language KQML)

When the MTA wants to make use of TV1, it would first send the following message to TB:

(evaluate :sender MTA :content (ask-resource :type address :name TV1) :ontology software test :receiver TB :language KQML)

The TB would answer with the message below:

(evaluate :sender TB :content (tell-resource :type address :value glororan.ncl.ac.uk:54103 :name TV1):ontology software test :receiver MTA :language KQML)

After that, the MTA would dialogue with TV1 directly.

5 THE IMPLEMENTATION OF MTA

The MTA has been implemented with the Java Agent Template (JAT) [7] and the Java Remote Method Invocation (RMI). The JAT provides a fully functional template for constructing agents which communicate peer–to–peer with a community of other agents distributed over the Internet. However, JAT
agents are not migratory but rather have a static existence on a single host. As an improvement, we use the Java RMI to let JAT agents could dynamically migrate to foreign hosts in this implementation. Because the Java RMI could not work well on the Netscape Browser currently, the implementation of MTA is not available with Java Applet, but with stand-alone style. It can be downloaded at [6].

5.1 An Operational Environment

To implement MTA, we present an operational environment based on the Java software as shown in Figure 5, allowing MTA to roam cross the disparate platforms.

![Figure 5. An operational environment for implementing MTA](image)

The hardware platform of the testbed at the lowest level in Figure 5, is a network of UNIX machines running the Solaris 2.x operating system which often plays a part in distributed system. The widespread use of Personal Computer (PC) has also prompted an ongoing effort to port the environment to the PC/Windows platform. On the top of the hardware platform is Java Development Kit (JDK) and JAT. JDK consists of the Java programming language core functionality, the Java Application Programming Interface (API) with multiple package sets and the essential tools such as Java Abstract Windowing Toolkit (AWT) and Java RMI. On top of this platform is MOST which consists two testing servers and one testing client. MTA is launched by the user on the Testing Client side, migrates to Testing Servers and sends results back to the user.
5.2 How It Works

Before launching MTA on the Testing Client side, we set up two Tools Brokers on the Test Server side on two different hosts: Tools Broker A on Glororan and Tools Broker B on Walton. The GUI includes a menu bar and a text area for displaying system messages as shown in Figure 7 (a).

![Tools Broker A on Glororan](image)

(a) GUI for Tools Brokers and MTAs

![System Messages](image)

(b) GUI for displaying System Messages

Figure 6. GUI for the implementation of MTA

The system messages can be displayed on another window as shown in Figure 6 (b) after we click on the “System Messages” button. Each message is written to a log file.

When we launch an MTA from the Testing Client on Aidan to the Testing Server A on Glororan, the standard MTA GUI the same with Tools Broker. The MTA gets the message here, sends it back to user on the Testing Client side and migrates to the Testing Servers B.
The scenario for launching MTA on Aidan is shown in Figure 7. It shows MTA can roam through the Internet and send the result back to the user.

![Figure 7. A scenario for launching MTA](image)

6 CONCLUSION

Many testers and users fail to recognize how to select and implement testing tools for their applications due to the lack of the knowledge about the suitability of test methods, techniques and about the appropriateness of test metrics and test tools.

Mobile software testing provides a novel alternative paradigm for remote testing applications through the testing tools brokers on the remote sites. Testers do not need to know what the testing tools use and where the testing tools are. An MTA will do these jobs for them. The major advantages of MOST are: firstly, the testing tools vendors could register their testing tools with information to the tools brokers to look for their clients; secondly, testers on the testing client side only need to launch an MTA migrating amongst testing servers to seek suitable testing tools and thirdly, the tools broker can help testers to select suitable testing tools from different tool vendors on the remote sites to be a logical integrated test environment.
The MTA has been implemented with the JAT and the Java RMI in distributed computing environment. Currently we are integrating MTA with the WWW using the Java Applet. The next important step to be made is the implementation of the Tools Broker which is an intelligent agent to select the suitable testing tools to MTA. The *agent–based blackboards* technology will be used in our future work.

**Acknowledgement**

The work of one author, H.D. Chu, was funded by the National Science Council in Taiwan from whom he received a fellowship to work toward a doctoral degree.

**REFERENCES**

6. MOST: MObile Software Testing, Available at <http://www.ncl.ac.uk/~n4521677/MOTS.tar.gz>
9. Liggesmeyer, P. Selecting test methods, techniques, metrics, and tools using systematic decision support, in Proc. *QW’96* (San Francisco USA, May 1996).