1. Introduction and Motivations

A major concern for mobile agents is access to non-mobile resources. When an agent moves from one host to another, resources such as sockets and files that the agent may have been using on the source host are not available to the agent on the destination host. This complicates the task of the agent designer since the designer has to make sure that access to all non-mobile resources are terminated prior to the agent migrating. Furthermore, access to the same resources may have to be reestablished when the agent reaches the destination host.

This problem becomes even more difficult when the mobility of the agent is not requested synchronously by the agent, but asynchronously by the system. An example is the notion of forced mobility in the NOMADS mobile agent system [4], where the system may move agents between hosts for purposes of load balancing, hosts shutting down, etc. In such cases, it is extremely difficult and tedious to design the agent to terminate and reestablish access to non-mobile resources.

This paper describes a mechanism to transparently redirect network communication in the event of agent migration. The system’s goal is to allow agents to open network socket connections to other systems and then to be able to move between hosts without having to interrupt, disconnect, and reconnect any connections. This mechanism greatly simplifies the task of a mobile agent designer. In addition, when agents need to be forcefully moved (as in the NOMADS system), the system may do so assuming that the agent’s network connections will continue uninterrupted.

2. Mocket Overview

The Mocket API consists of 100% pure Java code. Because the implementation uses no native code, it is a cross platform API and can be used by any Java 1.2 compliant VM. Any agent that wishes to use a Mocket does so in a manner similar to using the Java sockets API. The agent must first instantiate a Mocket by providing the remote address and port number to the constructor. Once connected, the agent must use the input and output streams provided by the Mocket to send and receive data. At any point the Mocket may be suspended by the agent system so that the agent can migrate. After the migration has completed the agent system will cause each Mocket being used by the agent to resume its normal operation. The suspend and resume operations are transparent to agents reading from and writing to Mockets.

The following NOMADS Agent, in Figure 1, is an example of how to use the Mocket API. In this example a Mocket is created and used to read text data from a server that is also using Mockets. The agent enters an infinite loop reading one line of data from the Mocket and then migrating to a new host. Figure 2 illustrates the example.

---

1 This research is supported in part by DARPA’s Control of Agent-Based Systems (CoABS) program and NTA/NIMA.
3. Mockets Implementation

Suspending and resuming a Mocket requires that additional control information be sent over the network beyond what is required for normal network communications. The information is necessary so that both ends of a Mocket connection can coordinate the suspend and resume process so that no data is lost. There are many potential ways to exchange additional control information between Mockets. However, as mentioned above, the current Mocket implementation is in Java, which gives the programmer no access to the underlying transport layer. With this restriction there are two obvious solutions: segment the data so that control information can be sent over the same socket as the user data, or use a separate control socket for each Mocket connection (similar to the ftp protocol). If the data is segmented then much of the benefit of using TCP sockets is wasted since TCP also segments the data. If a separate control socket is used, then for each Mocket there are two underlying Java sockets.

The method of implementation chosen is a variation of the second solution mentioned above. In the current implementation, all Mockets running in a given VM share one control socket. Control

```java
public class ReceiverAgent extends Agent {
    public static void main (String[] args) {
        try {
            Mocket m = new Mocket("orion.coginst.uwf.edu", 8765);
            BufferedReader br = new BufferedReader (new InputStreamReader (m.getInputStream()));
            System.out.println ("Receiving numbers...");
            while (true) {
                System.out.println (br.readLine());
                go ("vega.coginst.uwf.edu", "3280", "anonymous", "guest");
                System.out.println (br.readLine());
                go ("andromida.coginst.uwf.edu", "3280", "anonymous", "guest");
            }
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
}
```

*Figure 1: Code example using Mockets*

![Figure 1: Code example using Mockets](image)

![Figure 2: Example of Agent Mobility with Mockets](image)
information is sent from a Mocket on one host to a central message processing point on the remote host. That information is then relayed to the appropriate VM and finally delivered to the appropriate Mocket. This method alleviates the need to segment the user data, while only incurring a one-socket penalty per VM and one extra socket for the host’s central message processing point.

The Mockets implementation consists of 3 major components:

?? **NodeController**: A separate program running on each host using Mockets. It acts as a single entry point for Mocket control messages per host

?? **VMController**: A separate thread running in each VM using Mockets. It acts as a single entry point for Mocket control messages per VM.

?? **Mockets API**: The set of classes available to user code for mobile sockets

The remainder of this paper will discuss the implementation of the components shown in Figure 3.

### 3.1 Node Controller

The Node Controller is a standalone program that runs on every host using Mockets. It acts as a single point of entry for all control and query messages pertaining to Mockets being used by agents running on that host. When messages arrive the Node Controller will determine which VM the message should be sent to and then passes the message along to the VM Controller to be delivered to the destination Mocket. Any replies are sent back to the Node Controller, which are then passed back to the original sender.

When several VMs on a given host need to use Mockets, it is necessary to execute the Node Controller as a separate program since it acts as the single control point for all Mockets being used on a host. If the Node Controller was run in a VM using Mockets and that VM quit normally then all other VMs using Mockets would no longer be able to receive control messages. If only one VM will be using Mockets on a given host, the Node Controller can be run as a separate thread inside that VM.

### 3.2 VM Controller

The VM Controller serves two purposes: it receives all incoming control messages and delivers them to the appropriate Mocket, and it suspends and resumes all Mockets when an agent wishes to migrate. A VM Controller runs on any VM using Mockets. It runs as a separate thread that gets started when the

---

*Figure 3: Relationship between the three major components*  
*Figure 4: Flow of control messages and user data*
first Mocket or ServerMocket is created. The suspend and resume operations may be invoked by any piece of code that has been granted the appropriate permission. Usually this would be the agent system, but any piece of code can be authorized to suspend and resume Mockets by setting the Java permissions in a Java security policy file.

3.3 Mockets
A Mocket encapsulates a normal TCP socket. All access to the underlying socket is done via wrapper methods in the Mocket class. The input and output stream of the TCP socket are also encapsulated by MocketInputStream and MocketOutputStream respectively. By using wrapper classes, the underlying socket and its streams can be replaced transparently to the user when an agent migrates. Figure 5 shows the structure of the Mocket class.

As mentioned above, all control information is delivered to a Mocket via a separate network connection. Because only user data is sent over the Mocket connection, a Mocket can communicate with normal TCP sockets. However, a Mocket that is connected to a normal TCP socket cannot be suspended. That Mocket must be closed prior to migration. Only connections between two Mockets can be suspended and resumed.

When a Mocket connects to another end-point it must determine whether or not that end-point is also a Mocket or if it is a normal TCP socket. Since a Mocket cannot be suspended if it is connected to a normal TCP socket, the remote end-point’s type must be determined first before the agent has a chance to migrate. Figure 6 outlines the Mocket initialization process.

3.3.1 Suspending a Mocket
A Mocket can be suspended in one of two ways: it can be suspended because the agent using the Mocket wishes to migrate (suspension was triggered locally) or it can be suspended because the other side of the connection needs to migrate (suspension was triggered remotely). When a connection is suspended, one end is always local and the other end is always remote. The procedure for suspending a Mocket differs depending on how it was triggered.

If the Mocket was suspended locally, it will first contact the remote Node Controller and tell it to suspend the other end of the connection. Next the Mocket will suspend the input stream. Suspend will

![Figure 5: Mocket structure](image)
not return until all data has been read from the underlying stream and all threads reading from the stream have been put to sleep. After the input stream has been suspended the Mocket will suspend the output stream. Like MocketInputStream.suspend(), MocketOutputStream.suspend() will not return until all threads writing to the stream have been put to sleep. Finally, before closing the underlying socket, the Mocket will record the remote address to which it was connected. Lastly the underlying socket connection is closed. If the Mocket was suspended remotely, it will first suspend the output stream. Next it will suspend the input stream. Finally it will close the underlying socket.

When a connection is suspended it is important that the streams in one Mocket be closed opposite the order in which they are closed in the other Mocket. If the streams were suspended in the same order on both sides a deadlock will occur. By suspending the streams in the opposite order one input stream is always trying to finish reading while the corresponding output stream is trying to finish writing. Figure 7 illustrates the process of suspending a Mocket before an agent migrates.

![Figure 6: Mocket initialization process](image)

![Figure 7: Suspend process](image)
3.3.2 Resuming a Mocket

A Mocket is resumed in one of two ways depending on how it was suspended. If the Mocket was suspended locally (i.e. it was suspended because the agent using the Mocket migrated) then it is responsible for initiating the resume. If the Mocket was suspended remotely (i.e. it was suspended because the remote agent migrated) then it must wait for the other side to initiate the resume process.

If the Mocket being resumed was suspended locally, it will first contact the remote Node Controller to find out what port it should use to resume the Mocket connection. The port used is the same port that the Node Controller uses to deliver messages to the VM Controller. By using this port, the VM Controller can receive the resume message and hand off the connection to the Mocket waiting to be resumed. Once the Mocket knows which port to use, it will contact the VM Controller and send it a resume message. The VM Controller will then hand off the connection to the appropriate Mocket on the remote side. Finally the Mocket will resume only its input stream. At this point the Mockets are “resumed”.

If the Mocket being resumed was suspended remotely, it will receive a TCP socket connection from the VM Controller. The Mocket will keep this socket as its underlying socket connection and resume only its input stream. At this point the Mocket is “resumed”. The following diagram illustrates the process of resuming a Mocket after an agent has migrated.

3.3.3 Mocket Streams

The two major components of the Mocket API are the Input and Output Streams. They are important because they control how the data is actually sent using a Mocket. This section will describe some of the overall design decisions as well as the implementation of both the Input and Output streams.

Since Mockets wrap around normal TCP Sockets, it is desirable to take advantage of the benefits of TCP. One major benefit of TCP is that it will buffer data until it is read from the socket. Since the underlying socket buffers the data, the Mocket does not need to buffer data until it is suspended. Until the Mocket is suspended, calls to read and write can be mapped directly to the underlying socket. When a Mocket is suspended the output stream will flush all data pending on the TCP buffer. At the same time, the corresponding Input stream will buffer data from the socket until no more data is available.

![Diagram of Resume process](image)
When the Mocket is resumed, the output stream will begin writing normally, and the input stream will map all calls to read to its internal buffer until it is empty. After the buffer is emptied it will map all calls to read directly to the socket again.

Another benefit of TCP is that it will segment data and make sure that if the data arrives at its destination, the data will be delivered correctly and in order. As mentioned earlier, if Mockets choose to send control information on the same socket as the data it would be necessary to segment the data. Not only would Mockets not be able to communicate with normal TCP sockets, they would be unnecessarily re-implementing something that TCP already does. Because TCP guarantees that if the data arrives at its destination, it will be delivered correctly and in order, Mocket streams can blindly read and write to the socket without worrying about segmenting the data to ensure correct, in-order delivery.

The Mocket Stream implementation was simplified by relying on some of the features that TCP provides, but blindly writing and reading to and from the socket causes a problem with mobility. There are two types of read and write methods, one that accepts one byte to be read or written and another that accepts an array of bytes to be read or written. When read or write is called on the underlying stream, that call will block until it has been satisfied. A Mocket cannot be suspended until all calls to the underlying read and write methods complete. If a call to read or write is made with a large array, it could be several minutes before the call is satisfied. It is usually not convenient for an agent to wait for several minutes before it can suspend all of its Mockets. Therefore, the Mocket streams must break large arrays into smaller chunks and write one chunk at a time. After writing one chunk it will check to see if the stream has been suspended before writing the next one.

Checking to see if the stream has been suspended after reading or writing a chunk can be inefficient depending on the chunk size. Currently the chunk size is set to 2048 bytes. That size was chosen because it would allow a Mocket stream to check to see if it is suspended once each second on a 28.8 baud modem connection. If Mockets are being used on a higher bandwidth connection this size should be adjusted to allow more data to be sent each time. The current Mocket implementation does not adjust the chunk size depending on the connection. The best way to optimize this feature would be to monitor how many bytes are being sent and increase/decrease the chunk size accordingly.

4. Related Work

The Mockets API is not the only project that is addressing transparent network redirection. As agent based systems grow in popularity, network transparency is becoming more and more desirable because it helps to greatly simplify solutions. As an example consider CBorg developed at the Vrije University in Brussels.

CBorg is a mobile agent system that supports location independent addressing and message forwarding. CBorg accomplishes this by using a custom built router that doubles as a name server. Instead of sending lookup requests to the name server, entire messages are sent, and the name server will route the messages appropriately.

Another example is FarGo developed by the Israel Institute of Technology. FarGo is a mobile code system that also supports location transparency via updating remote RMI references. The basic building blocks of a FarGo application are called complets. Complets can be interconnected via complet references, which can be local or remote (using RMI). A complet communicates by using these complet references to invoke methods on other complets. Each time a source or destination complet migrates, the FarGo system updates the references of affected complets so that they remain valid.
5. Future Work

The Mockets API is the first step towards a network redirectable socket for use with mobile agents. Some of the future enhancements include:

?? **Embeddable Node Controller:** The Node Controller currently is implemented in Java and runs as a stand-alone application. It is required on any host that is using Mockets. However, the implementation does not have to be in Java. It is possible to write the Node Controller in C or C++ and embed it as part of an agent system such as NOMADS.

?? **Mobility with normal TCP sockets:** Currently Mockets can be migrated when they are communicating with other Mockets. Mockets can also communicate with normal sockets, but when they do they lose the ability to migrate. A Proxy could be used to allow Mockets to connect to normal sockets while still allowing the agent using the Mocket to move. The Proxy could be a standalone program or embedded in another application like the Node Controller. The proxy would use Mockets to communicate with the agent and normal TCP sockets to talk to the server and would act as a middleman for communication.

6. References


