International Journal of Information and Electronics Engineering, Vol. 1, No. 3, November 2011

Mobile Objects: Objects That Can Migrate to Other Hosts

Tak Sing Li and Sze Mui Wong

***Abstract*—A new Java tool called *MobileObject* for use in distributed computing is to be presented in this paper. This tool is more user friendly and powerful than other similar tools like *RMI* and mobile agents. It uses Java dynamic proxy to forward method calls to remote objects. The tool allows the user to deploy a distributed computing program from a single computer. The code will migrate to other computers accordingly and therefore the deployment of a distributed system can be done from one single computer.**

***Index Terms*—Distributed computing, *RMI*, mobile agents .**

I. Introduction

RMI [1],[2] and mobile agents [3],[4] have been used for many years. *RMI* allows the creation of a remote object so that its methods can be invoked remotely. When a remote object is sent to another computer, only a proxy is sent. When a remote computer invokes a method via the proxy, this request is forwarded to the computer that stores the remote object. After the invocation of the method, the result is returned back to the caller of the method. *RMI* is suitable to be used in distributing computing because communication of two processes in different computers is simplied as method invocation.

The drawbacks of using *RMI* are:

- All remote objects have to be subclass of *UnicastRemoteObject*. The programmer is not free to have it derived from other classes. This discourages software reuse through inheritence.
- We cannot synchronize on remote objects. Even we have declared a method of an *RMI* object as synchronized, it does not *work* remotely. There is no guarantee that the same remote thread will be used to invoke the method when we use the same local thread to invoke a remote method for several times.

On the other hand, a mobile agent is a fragment of program code that can move from one computer to another. Therefore, one important use of them is to deploy a distributed computing system. With a mobile agent, we can push some code to a remote computer and has the code executed there. So it has the advantage of easy deployment. However, most mobile agents comunicate using message exchange. So the programmer has to deal with the communication protocol. In addition, synchronization has to be dealt with explicitly.

A project called *JavaParty* [5][6] was developed to solve the above *RMI* problems. It does allow for syncrhonization on remote objects. However, the preprocessor of JavaParty will convert a mobile class to be a subclass of a given class. So it has the same problem of RMI that the programmer is not free to develop a JavaParty class so that it is a subclass of an existing class.

K.M. Chandy, A. Rifkin, P.A.G. Sivilotti, J. Mandelson, M.Richardson, W. Tanaka, L.Weisman introduced another

Manuscript received September 18, 2011; revised October 22, 2011.

The authors are with Open University of Hong Kong, 30 Good Shepherd Street, Hong Kong (email: tsli@ouhk.edu.hk, anwong@ouhk.edu.hk).

II. MobileObject

We have developed a Java tool called MobileObject. This tool has both the advantages of RMI and mobile agents but without the stated problems.

MobileObject is similar to RMI in that proxy is used to represent a remote MobileObject. When a MobileObject or a proxy moves to a remote host, some transformation regarding the MobileObject or proxy will occur.

There are four kinds of transformation for a MobileObject or a proxy. The first kind of transformation is when a MobileObject moves to another host. The actual MobileObject is copied to a remote host. The original host will receive a proxy of that moved MobileObject. Fig. 1 depicts how a MobileObject moves to a remote host. The local host will use the proxy to invoke methods of the remote MobileObject.

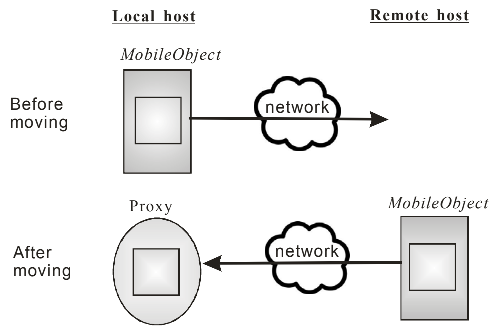

Fig. 1. When a *MobileObject* is moved to another host

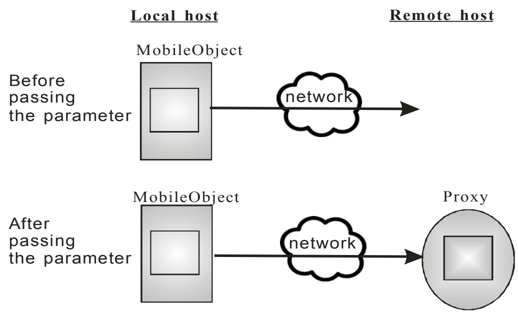

Fig. 2. A *MobileObject* is passed as parameter to a remote host.

The second kind of transformation is when a *MobileObject* is passed as a parameter to a remote host. In this case, only the proxy is passed to the remote host. This case is depicted in Fig. 2. Code in the remote host can invoke the methods of the *MobileObject* via the proxy.

The third kind of transformation is when a proxy of a *MobileObject* is passed as a parameter to the host where the *MobileObject* is stored. In this way, the proxy will be changed back to the original *MobileObject* when it arrives at the remote host. This is shown in Fig. 3.

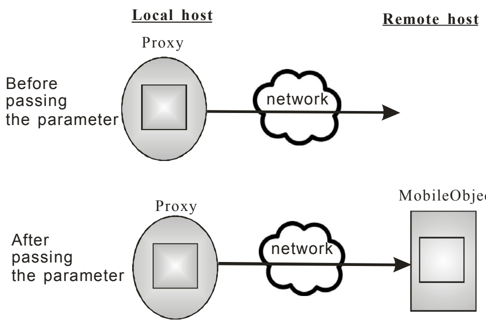

Fig. 3. A proxy of a *MobileObject* is passed as parameter to the host where the *MobileObject* is stored.

The forth kind of transformation is when a proxy of *MobileObject* is passed as parameter to a host other than the one that stores the *MobileObject*. In this case, the proxy will just be copied to the destination. This is depicted in Fig. 4.

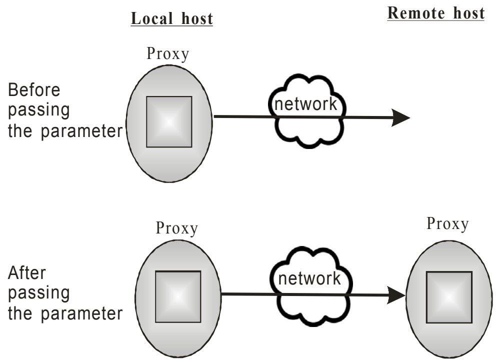

Fig. 4. A proxy of a *MobileObject* is passed as parameter to a host other than the one that stores the *MobileObject*.

III. LOCAL OBJECT IN MOBILE OBJECT

In a *MobileObject*, there is an attribute of type *LocalObject*. The proxy also contains the same *Local Object* in the *MobileObject*. This is shown in Fig. 5. The *Local Object* is used to store the data of the *MobileObject* that cannot change over time. Whenever there is a request by a remote user to such data, there is no need to go back to the remote *MobileObject* to get it. The methods of *MobileObject* are also classified into two groups, one that can be invoked locally and the other remotely. A local method call is directed to the *LocalObject* while a remote method call is directed to the remote *MobileObject*.

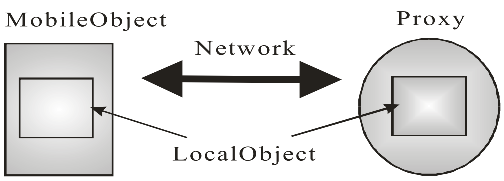

Fig. 5. *Local Object* in *Mobile Object*

In our implementation, *LocalObject* is a class while *MobileObject* is an interface. All the internal operations of *Mobile Object* is implemented in *Local Object*. The programmer just needs to design a *MobileObject* with an *LocalObject* and following a number of rules. We will show an example to see how *java.util.Vector* can be extended to its *MobileObject* version.

IV. A MOBILE VERSION OF VECTOR

Let's assume that we want the extended *Vector* to have a method which returns the name of the *Vector* and assume that this name will not change. So we can put this name field in the *LocalObject*. In order to do this, we need to create a custom made *LocalObject* with a custom made interface.

```
Public interface VectorLocalInterface {
   Public String name ();
}

Public class VectorLocalObject extends LocalObject,
VectorLocalInterface {
   Private String name;
   public VectorLocalObject (String n, MobileVector vector)
{
      super (vector);
      Name=n;
   }
   public String name () {
      Return name;
   }
}
```

To create *MobileVector*, we need to define an interface for *MobileVector* which will be used as the type of the proxy. Assume that we are only interested in the *size*() method, the *add*() method and the *get*() method of *Vector*.

```
Public interface MobileVectorInterface {
   Public int size ();
   Public void add (Object obj);
   Public Object get (int i);
}
```

Then, we can create the *MobileVector*:

```
Public class MobileVector extends Vector implements
VectorLocalInterface, MobileObject, MobileVectorInterface
{
   Private VectorLocalObject localObject;
   Public MobileVector (String name) {
      LocalObject=new VectorLocalObject (name, this);
   }
   Public LocalObject localObject () {
      Return localObject;
   }
   Public MobileObject move (Host h) {
      Return localObject.move (h);
   }
   Public Object replaces (MyObjectOutputStream output) {
      Return localObject.realObjectReplace (output);
   }
   Public String name () {
      Return localObject.name ();
   }
}
```

In the constructor, we need to create a *VectorLocalObject*. The three methods *localObject* (), *move* () and *replace* () are methods required by the *MobileObject* interface. The *localObject* () method is trivial which returns the *LocalObject* attribute. The *move* () method is used to move the object to the specified host as depicted in Fig. 1. It returns the proxy of the moved object. The *replace*() method is used internally when the *MobileObject* is sent as parameter to a remote method as depicted in Fig. 2, Fig. 3 and Fig. 4. The implementation of these methods involves the invocation of the corresponding methods in *localObject*. The programmer does not need to worry about its implementation details. Lastly, the *name* () method is required by *Vector Local Interface*.

Although *MobileVector* has implemented the *MobileVectorInterface*, there is no need to provide implementation for the methods *size* (), *add* () and *get* (). They are already defined by the super class of *MobileVector*, namely *java.util.Vector*.

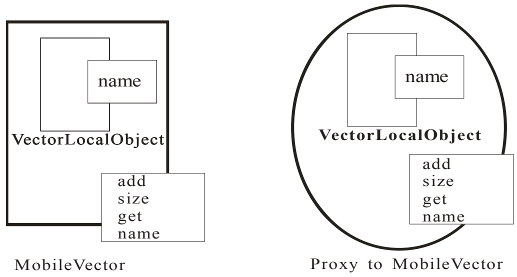

Fig. 6. Graphical representation of *MobileVector* and its proxy.

Fig. 6 shows the graphical representation of a *Mobile Vector* and its proxy. They both have an attribute of type *VectorLocalObject*. *MobileVector* has four methods to be used externally, namely *add* (), *size* (), *get* () and *name* (). The figure does not show the methods that are used internally, like *move* (), *replace* (), *local Object* (), Note that *name* () is also defined in *VectorLocalObject*. The implementation of *name* () in *MobileVector* is simply an invocation of *name* () in *VectorLocalObject*. When a *MobileVector* is sent to a remote host, it is replaced by a proxy to *MobileVector*. However, this proxy also contains an exact copy of the *VectorLocalObject* in the *MobileVector*. The proxy also has the methods of *MobileVector* as shown. When the *name* () method of the proxy is invoked, it will forward the call to the *VectorLocalObject*. When other methods are called, it will forward the call to the remote *MobileVector*.

Consider the following fragment of code that uses *MobileVector*.

```
....
MobileVector vector=new MobileVector ("my vector");
MobileVectorInterface vectInt=vector.move (h);
System.out.println (h.name ());
vectInt.add (new Integer (3));
System.out.println (vectInt.size ());
....
```

The first line creates a *MobileVector*. Then it is moved to a remote host denoted by *h*. *vectInt* is the proxy that points to that moved object. In the third line, the *name*() method is called. Since this method is defined in *VectorLocalInterface*, it will be forwarded to the local object in the proxy. In the fourth line, the number 3 is added to the remote *Vector*. Finally, the remote method *size*() is called. Both the *add*() method and the *size*() method here are remote method calls.

We can also use anonymous class with *MobileObject*. For example:

```
...
 Final Mobile Interface vector1=new Mobile Vector("my vector1");
MobileVector vector2=new MobileVector ("my vector2") {
   Public int size () {
      Return vector1.size ();
         }
   };
MobileVectorInterface vectInt2=vector2.move (h);
System.out.println (vectInt2.size ());
...
```

In the above fragment of code, when *vector2* is created, its *size* () method is overridden so that it returns the size of *vector1* instead. Then, *vector2* is moved to host *h* and the proxy for *vector2* is *vectInt2*. When *vectInt2.size* () is invoked the call will first forward to the remote object. Then in the remote object, a remote call is forwarded back to the local *vector1* object. Eventually, the size 0 is printed.

When using anonymous class with *MobileObject*, we must ensure that the enclosing class has implemented the *Serializable* interface. This is because every object of an anonymous class is associated with an object of the enclosing class. If the enclosing class is not a *Serializable*, then the object cannot be sent over network to a remote host.

V. Identification of Mobile Objects and Their Proxy

Each host has a table containing references to all of its *MobileObjects*. All *MobileObjects* and their proxy are identified by a unique hash code. A *MobileObject* and its proxy will have the same hash code. When a proxy arrives from a remote host, the system will first check whether the proxy refers to a local *MobileObject*. This is done by checking the hash code of the proxy against the hash code of all local *MobileObjects*. If one is found, then the proxy is replaced by the actual *Mobile Object*.

VI. Garbage Collection of Mobile Objects

Since each host has a table that contains all *MobileObjects* stored in the host, this mean that the *MobileObjects* will never be garbage collected because they are always referenced by the table. In order to allow unused *MobileObjects* to be garbage collected, we need to remove a *MobileObject* from the table if it is not remotely referenced. We therefore need to record each external reference of a *MobileObject*. Whenever a *MobileObject's* proxy is sent to a remote host, this remote reference is recorded in *MobileObject*. When that proxy in the remote host is garbage collected, it will send back a message to the *MobileObject* about this. Then the record about this remote reference will be removed from the *MobileObject*. When there is no remote reference left for the *MobileObject*, it is removed from the table. This allows the *Mobile Object* to be garbage collected later.

VII. Synchronization of Mobile Objects

We have already mentioned that *RMI* does not allow synchronization on remote objects. [9],[10]

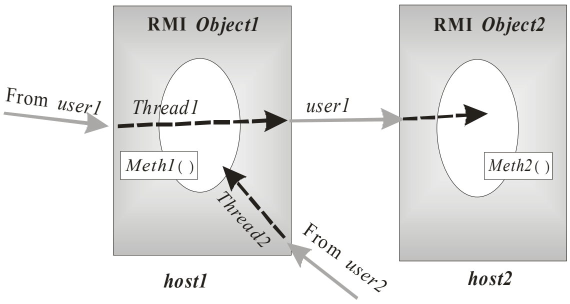

Fig. 7. Synchronization on *RMI* objects

In Fig. 7, there are two hosts and each has an *RMI* object. Two users have invoked the synchronized method *Meth1* of *Object1*. *Thread1* is used to serve *user1* in *host1*. This thread has successively accquired the monitor of *Object1* and invoked the method. In this method, it invokes *Meth2* of *Object2* in *host2*. *Thread1* will go to the waiting state to wait for the reply from *Object2*. Note that at this point *Thread1* did not release the monitor of *Object1*. At this time, a request to invoke *Meth1* of *Object1* comes from *user2*. *Thread2* is used to serve the request. Would *Thread2* be able to acquire the monitor of *Object1*? The answer is not certain. It is because there is no guarantee that *Thread2* is not *Thread1* in *RMI*. Note that *Thread1* is in the waiting state, it is therefore possible that *Thread1* is used to serve the requst from *user2*. If this is the case, *user2* will also be able to invoke *Meth1* because *Thread1* has already held the moitor of *Object1*.

Another problem is shown in Fig. 8.

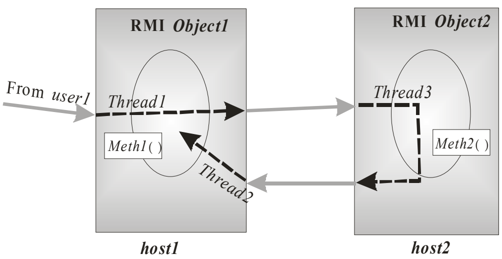

Fig. 8. Synchronization of *RMI* objects

In this case, *user1* has successfully invoked the synchronized method *Meth1* of *Object1*. Then in *Meth1*, it invokes a call to *Meth2* of *Object2* in *Host2* which then invokes a call back to *Meth1* of *Object1*. *Thread1* and *Thread2* are used respectively for the first and second invocation of *Meth1*. Will the second attempt to invoke *Meth1* be successful? Again, the answer is not sure because there is no guarantee that *Thread2* and *Thread1* are the same. If *Thread2* is not *Thread1*, then the call will have to wait for *Thread1* to release the monitor. We can see that this is a deadlock because *Thread1* is waiting for *Thread2* to return the result and *Thread2* is waiting for *Thread1* to release the monitor.

VIII. MobileThread

In order to tackle the above problem, MobileObject guarantees that when a thread Thread1 in Host1 invokes a method of another MobileObject in Host2, it will always use the same thread to serve the requests from Thread1. This is done by using Mobile Thread, a subclass of Thread. Mobile Thread has two subclasses, namely Master Thread and Slave Thread. The thread created by the programmer is a MasterThread. The thread created by the system to serve a remote request is a SlaveThread.

There is always a one-to-one relation between a *MasterThread* and its *SlaveThread* in another host. A *Master Thread* has a corresponding *Slave Thread* in every remote host to which it has invoked a remote method. So if a *MasterThread* in host 1 invokes multiple methods in host 2, these requests will always be served by one *SlaveThread* in host 2. This *SlaveThread* will never serve requests from other *MasterThread*. The *MasterThread* and its *SlaveThreads* in other hosts are identified by the *hash code* of the *MasterThread*. So when a *MasterThread* or a *SlaveThread* invokes a remote method, the *hash code* of the thread is also sent to the remote host. When the remote host receives this request, it will check whether there is a thread with this *hash code*. If there is one, then, this thread will be used to serve this request. If there is no such thread, a *SlaveThread* will be created with the *hash code* and is then used to serve the request.

In the example illustrated in Fig. 8, both *Thread1* and *Thread3* are guaranteed to be the same *SlaveThread*.

IX. Life Cycle of Mobile Thread

The life cycles of a Master Thread and a Slave Thread are shown in

Fig. 9 and

Fig. 10 respectively.

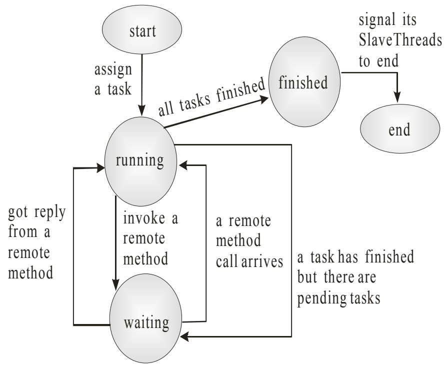

Fig. 9. Life cycle of a MasterThread

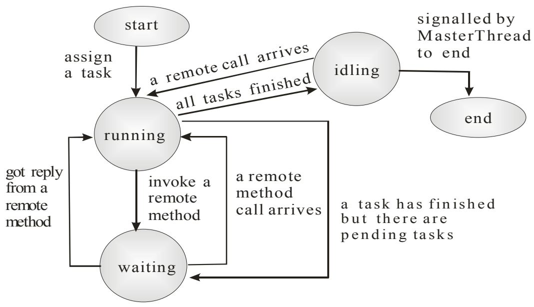

Fig. 10. Life cycle of a SlaveThread

The life cycles of *MasterThread* and *SlaveThread* are mostly the same. They only differ in when the thread should end. A *MasterThread* ends when all of its tasks have finished. On the other hand, a *SlaveThread* will not end even when all its tasks have been completed. It should only end when its corresponding *MasterThread* has ended. So before a

MasterThread ends, it should signal all of its *SlaveThread* to end.

When a request for invoking a method of an *MobileObject* arrives, the system will check the hash code of the thread which initiates this request. It will then check if this hash code corresponds to a local thread, which can be a *MasterThread* or a *SlaveThread*. If such a thread exits, it should be either in the waiting state or the idling state. Then this thread is used to serve the request. If such a thread does not exist, a new *SlaveThread* with the hash code will be created. Then, the thread will enter the running state to process the request.

Whenever a thread in the running state needs to invoke a remote method, it will send the request to the remote host and then enter the waiting state. If a thread is in the waiting state waiting for the reply of a remote method invocation, then when the reply arrives, the thread will go back to the running state to process the remaining task.

X. Synchronization using Locks

Locks provide more advanced features in synchronization than those provided by synchronized methods. Naturally, locks provided by JAVA APIs can only work locally. However, it is easy to create a *MobileObject* version of *Lock*.

We have described how to create a *MobileObject* version of a normal Java class. Before we can create *MobileLock*, we need to create *MobileCondition* first.

```
Public class Mobile Condition implements Mobile Object,
Condition {
  Public MobileCondition (Condition c) {
    Condition=c;
    LocalObject=new LocalObject (this);
  }
  Private Condition condition;
  Private LocalObject localObject;

  Public MobileObject move (Host h) throws Exception {
    Throw new Exception ("Condition cannot move");
  }
  Public LocalObject localObject () {
    Return localObject;
  }
  Public Object replaces (MyObjectOutputStream output) {
    Return localObject.realObjectReplace (output);
  }
  Public void await () throws InterruptedException {
    condition.await ();
  }
  ....// other methods required by Condition
  Public void signalAll () {
    condition.signalAll ();
  }
}
```

The main ingredient is the *localObject* attribute which contains all the implementation of *MobileObject*. The methods *move*(), *localObject*(), and *replace*() are the methods required by the interface *MobileObject*. We can simply direct the calls to the corresponding methods in *LocalObject*. The constructor accepts an existing *Condition* as parameter. This is the local *Condition*. Remote calls to this *Condition* will be directed to the remote *MobileLock* through the proxy.

Methods *wait* () to *signalAll* () are all the methods required by the interface *Condition*. These calls are all forwarded to the *condition* attribute. Then we can create *MobileLock*. The code is shown below.

```
Public class MobileLock extends ReentrantLock implements
Lock, MobileObject {
  Private LocalObject localObject;
  Public MobileLock () {
    LocalObject=new LocalObject (this);
  }
  Public MobileObject move (Host h) throws Exception {
    Throw new Exception ("Lock cannot move");
  }
  Public LocalObject localObject () {
    Return localObject;
  }
  Public Object replaces (MyObjectOutputStream output) {
    Return localObject.realObjectReplace (output);
  }
  Public Condition newCondition () {
    Condition c=super.newCondition ();
    MobileCondition mc=new MobileCondition(c);
    Return mc;
  }
}
```

The LocalObject attribute and methods move(), localObject() and replace() are treated similarly as those in MobileCondition. The method newCondition () is required to create a MobileCondition instead of the Java default Condition. All other methods of Lock are just inherited from ReentrantLock and therefore there is no need to define them here. We can then use MobileLock and MobileCondition to do synchronization remotely.

XI. An Example Using Mobile Lock

The following example illustrates how *MobileLock* is used.

```
Final Lock lock1=new MobileLock ();
Final Condition cond1=lock.newCondition ();
MasterThread thread1=new MasterThread () {
  Void run () {
    Try {
      lock1.lock (); //this is a local call
      Thread.sleep (20000);
      cond1.await (); //this is a local call
      lock1.unlock (); //this is a local call
    }
    Catch (Exception e) {
    }
  }
};
thread1.start ();

MobileObject obj=new MyMobileObject () {
  Public void meth () {
    Try {
      lock1.lock (); //this is a remote call.
      cond1.signal (); //this is a remote call
```

```
        lock1.unlock (); //this is a remote call
      } catch (Exception e) {
      }
   }
};

MobileObject objproxy=obj.move (host1); //obj is moved to host1
objproxy.meth (); //invoke meth of the obj in host1. This is a remote call
```

Assume the above code is executed in host *host0*. There are two threads in *host0*. The first one is the one that executes the code. The second one is the *MasterThread* created in the code. We denote the former thread *T1* and the latter thread *T2*. In *T2*, the object *obj* is moved to host *host1*; it is then referenced locally with *objproxy*. Then, the *meth* () method is invoked. This is a remote method call. A *SlaveThread T3* will then be created in *host1* to serve this request. So *meth* () of *obj* is executed by *T3* in *host1*.

T1	T2	T3
start	start	
	obj.move()	
lock1.lock()	objproxy.meth() //remote call	start
sleep		obj.meth() //serve T2
		lock1.lock() //remote call
	lock1.lock()//serve T3	
wake up		
cond1.await()	lock1 locked	
		cond1.signal() //remote call
T1 is waken up	cond1.signal() //serve T3	
		lock1.unlock() //remote call
T1 get lock1 again	lock1.unlock() //serve T3	
T1.unlock()		end
end	end	

Fig. 11. Events on different threads

Note that *T1*, *T2*, *lock1* and *cond1* are all in *host0*. *T3* and *obj* are in *host1*. So you can see that all method calls to *lock1*, *cond1* in *T3* are remote calls which will be forwarded to *T2*. Fig. 11 shows the sequences of evens in different threads. You can also see how *T2* changes between the waiting state and running state when invoking a remote method and serving a remote method.

XII. Conclusion

We have developed a useful tool for distributed computing. With this tool, a programmer can write a distributed computing program as if it is to be executed on a single machine. *MobileObjects* can then be moved to other computers and to have code executed there. Compared with *JavaParty* and *RMI*, *MobileObject* has the following advantages:

- All existing non-final Java classes can be extended to the MobileObject version very easily.
- Synchronization of threads in different hosts can be done via synchronized methods or mobile version of Lock.

References

[1] T. Downing, *Java RMI: Remote method invocation*, 1998, IDG Books Worldwide (Foster City, CA).

[2] Sun Microsystems, *Remote Method Invocation Specification*, http://java.sun.com/products/jdk/rmi/, 1997.

[3] J. Kiniry,D. Zimmerman, *A hands-on look at Java mobile agents*, *Internet Computing*, IEEE, Jul/Aug, pp. 21-30, 1997

[4] D.B. Lange, M. Oshima, *Mobile agents with Java: The Aglet API,* World Wide Web, vol 1, no 3, pp. 111-121.

[5] M. Zenger, M. Philippsen, *JavaParty – Transparent Remote Objects in Java*, ACM 1997 Workshop on Java for Science and Engineering Computation, June 1997.

[6] M. Philippsen, B Haumacher, *Locality optimization in JavaParty by means of static type analysis,*Concurrency: Practice & Experience, val 11, 1999, pp1213-1224.

[7] K.M. Chandy, A. Rifkin, P.A.G. Sivilotti, J. Mandelson, M.Richardson, W. Tanaka, L. Weisman, *A word-wide distributed system using Java and the Internet,* 5th IEEE international Symposium on High Performance Distributed Computing, 1996, pp 11.

[8] J. Blosser, *Exploring the Dynamic Proxy API,* http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-proxy.html, Nov 2000.

[9] A. Tanenbaum, M.V. Steen. *Distributed Systems: Principles and Paradigms* (2nd Edition), Prentice Hall, 2006.

[10] B Haumacher, T Moschny, J Reuter, W.F. Tichy, *Transparent distributed threads for Java*, Parallel and Distributed Processing Symposium, 2003.

Tak Sing Li received a BSc(Hons) degree in mechanical engineering from the University of Hong Kong. He then received an MSc degree in computer science in 1991 and a PhD in computer science in 1994. Both degrees were from the Queen's University of Belfast.
He joined the Open University of Hong Kong in 1995 as a lecturer in the computing team. He is now an associate professor in the same university.

Sze Mui Wong received a BSc (Hons) degree in Mathematical Studies in 1988, and received an MSc degree in Operational Research in 1989 at the University of London. She completed the PhD in Computational Modeling in 2001 in the City University of Hong Kong. She joined the Open University of Hong Kong in 1993 as a lecturer. She is now an associate professor in the same university.