Stealing Cycles One Object at a Time

Dan-Adrian German
Department of Computer Science, Indiana University Bloomington
150 S. Woodlawn Ave., Bloomington, IN 47405–7104
Email: dgerman@indiana.edu

***Abstract*— We illustrate a simple but powerful program transformation whereby given an infinite number of processors (taken as individual hosts on the network) and virtually any program written in Java, every object in the resulting program runs on its own host. Thus a program can distribute its objects on the network wherever cycles are available. Lack of reliability in the network and relationship with web services, grid computing and XML-based network protocols are briefly discussed at the end.**

I. A Brief Historical Perspective

Object-oriented programming is really about programming over a distributed platform. A paragraph from [1] will help us capture this primordial aspect: "In 1961 [Alan] Kay worked on the problem of transporting data files and procedures from one Air Force air training installation to another and discovered that some unknown programmer had figured out a clever method of doing the job. The idea was to send the data bundled along with its procedures, so that the program at the new installation could use the procedures directly, even without knowing the format of the data files. The procedures themselves could find the information they needed from the data files. The idea that a program could use procedures without knowing how the data was represented struck Kay as a good one. It formed the basis for his later ideas about objects." By contrast, none of the modern benefits of object-oriented programming (e.g., inheritance, polymorphism, etc.) were able to acquire an identity of their own that early.

II. A Note on Distributed Computing

More than 30 years later a very popular document [5] brings forth a strong argument, "that objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space. These differences are required because distributed systems require that the programmer be aware of latency, have a different model of memory access, and take into account issues of concurrency and partial failure." The paper, that precedes the introduction of Java RMI by about 5 years, concludes with a set of guidelines on what is required of both systems-level and application-level programmers and designers if one is to take distribution seriously. Thus we have to accept that local and distributed computing are intrinsically different; however, differences are identified and discussed clearly, and their nature is now completely understood.

III. Java RMI vs. XML-based protocols

The appearance of Java RMI preceded XML-RPC and SOAP by a couple of years. Although it has spawned technologies (see, for example, [6], [7] and [8]) RMI has not really captured the imagination of developers. By contrast, the promise of connecting a large number of otherwise incompatible platforms, seamlessly, into one large hybrid network has proved to be a much more attractive proposition. And while the promise of XML-based network protocols is not to be argued, we want to focus on the advantages that a language-centric approach, such as Java RMI, offers.

In [3] we have demonstrated that while local and distributed computing are essentially distinct concepts, their differences can be safely isolated and encapsulated into a design paradigm that allows a complete separation between design and development (on one hand) and deployment (on the other). Examples presented there described the genesis of the pattern, with examples taken from the multiplayer game design domain. In [4] the pattern was simplified. A class was presented (`NetworkPeer`) isolating the basic functionality of a *free agent*. Using this class we demonstrate a program transformation whereby virtually any program written in Java can have a new (possibly dedicated) network host allocated automatically, at run-time, one for each of the objects it creates. Of course, it is of greater interest when the created objects are threads. In our example they are just normal objects, for simplicity.

IV. Issues of Concurrency

Of the four major differences between local and distributed computing, three are almost straightforward: latency, memory access and partial failure. The fourth one is a somewhat subtler difference: synchronization is not the same in a distributed and a non-distributed system. If we use threads to model a distributed system, the local model must be realistic. It must reflect the truly asynchronous operation in a distributed environment. Thus the only synchronization mechanisms that we encourage are local, at the level of each thread object: objects are the only critical regions. A non-distributed system (even multi-threaded) is layered on top of a single operating system which can be used to determine and aid in synchronization and in the recovery of failure. A distributed system, on the other hand, has no single point of resource allocation, or failure recovery, and thus is conceptually very different.

V. The Starting Point of the Example

To illustrate the program transformation we choose a simple, and often-used example. Two specialized objects `Odd` and `Even` are being defined, with the ability of knowing about each other. Each specialized type of object is working on the task of

finding out the parity of the argument in the following mutual recursive way (in which we assume $n \geq 0$)

$$\text{even}(n) = \begin{cases} \texttt{true} & \text{if } n = 0 \\ \text{odd}(n-1) & \text{otherwise} \end{cases}$$

$$\text{odd}(n) = \begin{cases} \texttt{false} & \text{if } n = 0 \\ \text{even}(n-1) & \text{otherwise} \end{cases}$$

These definitions immediately translate into Java:

```
class Odd {
  Even even;
  void setEven(Even even) {
    this.even = even;
  }
  boolean isOdd(int n) {
    if (n == 0) {
      return false;
    } else {
      return even.isEven(n-1);
    }
  }
}

class Even {
  Odd odd;
  void setOdd(Odd odd) {
    this.odd = odd;
  }
  boolean isEven(int n) {
    if (n == 0) {
      return true;
    } else {
      return odd.isOdd(n-1);
    }
  }
}
```

With the definitions above the test program becomes:

```
class One {
  public static void main(String[] args) {
    Odd odd;
    Even even;
    odd = new Odd();
    even = new Even();
    odd.setEven(even);
    even.setOdd(odd);
    System.out.println(odd.isOdd(5));
  }
}
```

Our goal is to illustrate a simple but powerful transformation using the example code provided above.

VI. Using Activators as Visitors

We start by defining an `Activator` type of object that can be used as active postage when sent from one host to the other.

```
class Activator implements java.io.Serializable {
  void activate() { }
}

class Server {
  void receive(Activator activator) {
    activator.activate();
  }
}
```

As we shall see, the major strength of this approach is that the meaning of the `activate()` method can be determined later. The host is indicated by a variable of type `Server` whose only method, `receive()` is responsible for running the activator thunk in a dynamic context of the new host.

Thus the basic scheme involves augmenting the `new` operator, by creating a thunk visitor object that will carry a delayed, frozen invocation of `new` to the host of choice.

Here's the resulting code:

```
class One {
  public static void main(String[] args) {
    Server s = new Server();
    s.receive(new Activator() {
                void activate() {
                  System.out.println("Here we go.");
                }
              }
             );
  }
}
```

VII. The Transformed Example

Following our previous approaches (see [4], [3]) we first create a local version of the transformed program:

```
class One {
  public static void main(String[] args) {
    Processor processor = new Processor();
    Odd odd;
    Even even;
    // odd = new Odd();
    odd = (Odd) processor.receive(
                              new Activator() {
                                Object activate() {
                                  return new Odd();
                                }
                              }
                             );
    // even = new Even();
    even = (Even) processor.receive(
                                new Activator() {
                                  Object activate() {
                                    return new Even();
                                  }
                                }
                               );
    odd.setEven(even);
    even.setOdd(odd);
    System.out.println(odd.isOdd(5));
  }
}
```

Instead of creating the objects we ask one of the available processors to receive an *activator* containing a frozen invocation of the statement we want executed on the new host.

Small changes are to be noticed in the `Activator`:

```
class Activator implements java.io.Serializable {
  Object activate() {
    return null;
  }
}
```

And correspondingly in the definition of a `Processor`:

```
class Processor {
  Object receive(Activator activator) {
    return activator.activate();
  }
}
```

The classes `Odd` and `Even` remain unchanged.

No attention is given in this paper to determining and allocating an eligible processor. In the interest of clarity we will be working with exactly two processors, one for each object being created.

VIII. The Distributed Outcome

Following [3] and [4] we now prepare interfaces for our free agents. Interfaces are elements of pure design and their use as agents' glorified business cards is encouraged here:

```
interface EvenInterface extends java.rmi.Remote
{
  boolean isEven(int n) throws
                        java.rmi.RemoteException;
  void setOdd(OddInterface peer) throws
                        java.rmi.RemoteException;
}

interface OddInterface extends java.rmi.Remote
{
  boolean isOdd(int n) throws
                        java.rmi.RemoteException;
  void setEven(EvenInterface peer) throws
                        java.rmi.RemoteException;
}
```

Their definitions take into account the new types and acknowledge the possibility of exceptional situations in the distributed version of our program.

```
public class Even extends NetworkPeer
                  implements EvenInterface {

  OddInterface odd;

  public void setOdd(OddInterface odd) throws
                        java.rmi.RemoteException {
    this.odd = odd;
  }

  public boolean isEven(int n) throws
                        java.rmi.RemoteException {
    try {
      System.out.print(
        java.net.InetAddress.getLocalHost() + ": ");
    } catch (Exception e) { }
    if (n == 0) {
      System.out.println("0 is even.");
      return true;
    } else {
      System.out.println("Is " + (n-1) + " odd?");
      return odd.isOdd(n-1);
    }
  }
  public void startAsLocalServer() { }
  public void startAsClientOf(java.rmi.Remote peer) {

  }

  Even() throws java.rmi.RemoteException {
    this.exportMethods();
  }
}
```

The `Odd` class is transformed in a similar way. The reader should note that all changes are purely cosmetic: decorations to ensure that the types and restrictions imposed by Java RMI are met. And since the framework offered by Java RMI is precisely the one announced in [5] the implication should be that we take them to be adequate and reasonable. Here's what the peer class `Odd` becomes:

```
public class Odd extends NetworkPeer
                 implements OddInterface {

  EvenInterface even;

  public void setEven(EvenInterface even) throws
                        java.rmi.RemoteException {
    this.even = even;
  }

  public boolean isOdd(int n) throws
                        java.rmi.RemoteException {
    try {
      System.out.print(
        java.net.InetAddress.getLocalHost() + ": ");
    } catch (Exception e) { }
    if (n == 0) {
      System.out.println("0 is not odd.");
      return false;
    } else {
      System.out.println("Is " + (n - 1) + " even?");
      return even.isEven(n-1);
    }
  }
  public void startAsLocalServer() { }
  public void startAsClientOf(java.rmi.Remote peer) {

  }

  Odd() throws java.rmi.RemoteException {
    this.exportMethods();
  }
}
```

Additional changes to the code enable the objects to report the hosts they're running on (simply for illustration purposes). Remote processors are the foundation of this transformation, so the `Processor` class also needs an interface:

```
  interface ProcessorInterface extends
                                java.rmi.Remote
  { Object receive(Activator activator)
           throws java.rmi.RemoteException;
  }
```

Processors themselves need to extend the network peer class and provide some of the decorations required:

```
class Processor extends NetworkPeer
                implements ProcessorInterface
{ public void startAsLocalServer() { }
  public void startAsClientOf(java.rmi.Remote peer) {

  }
  public Object receive(Activator activator)
                throws java.rmi.RemoteException {
    return activator.activate();
  }
  public static void main(String[] args) {
    String portNumber = args[0], ownName = args[1];
    (new Processor()).startAsNetworkServer(
                        ownName,
                        Integer.parseInt(portNumber)
                      );
  }
}
```

A `main` method is responsible for starting a processor on a

specific port, with a given name.

The activator class also needs an update, to acknowledge exceptions that could be thrown during activation,

```
class Activator implements java.io.Serializable {
  Object activate()
         throws java.rmi.RemoteException
  {
    return null;
  }
}
```

and we can now run the program:

```
class Program {
  public static void main(String[] args)
                     throws Exception
  {
    ProcessorInterface processorOne =
      (ProcessorInterface) NetworkPeer.locatePeer
         ("blesmol.cs.indiana.edu", 10776, "Dave");

    ProcessorInterface processorTwo =
      (ProcessorInterface) NetworkPeer.locatePeer
         ("molerat.cs.indiana.edu", 10776, "Dave");

    OddInterface   odd;
    EvenInterface even;

    odd  = (OddInterface)
             processorOne.receive(
               new Activator() {
                 Object activate()
                        throws
                        java.rmi.RemoteException
                 {
                   return new Odd();
                 }
               }
             );

    even = (EvenInterface)
             processorTwo.receive(
               new Activator() {
                 Object activate()
                        throws
                        java.rmi.RemoteException
                 {
                   return new Even();
                 }
               }
             );

    odd.setEven(even);
    even.setOdd(odd);

    System.out.println(odd.isOdd(5));

  }
}
```

We see that processors (like HTTP web servers) could get a default port number—and name. Preparing the test for execution involves the following two step process:

```
frilled.cs.indiana.edu%javac *.java
frilled.cs.indiana.edu%rmic Odd Even Processor
frilled.cs.indiana.edu%
```

Our installation involves two hosts `blesmol` and `molerat` on which the processors will be started. They share the code developed and compiled on `frilled` via NFS. Once we start the two processors separately on the two hosts the `Program` is run from `frilled`. Its output on `frilled` is not impressive at all. It contains the answer to the question `odd(5)`:

```
frilled.cs.indiana.edu%java Program
true
frilled.cs.indiana.edu%
```

But the output of processors running on `blesmol`

```
blesmol.cs.indiana.edu% java Processor 10776 Dave
Server is ready ...
blesmol.cs.indiana.edu/129.79.245.102: Is 4 even?
blesmol.cs.indiana.edu/129.79.245.102: Is 2 even?
blesmol.cs.indiana.edu/129.79.245.102: Is 0 even?
```

and `molerat`, respectively

```
molerat.cs.indiana.edu% java Processor 10776 Dave
Server is ready ...
molerat.cs.indiana.edu/129.79.245.107: Is 3 odd?
molerat.cs.indiana.edu/129.79.245.107: Is 1 odd?
molerat.cs.indiana.edu/129.79.245.107: 0 is even.
```

indicates that the test has been successful.

Here are the relevant parts of the `NetworkPeer` class. The complete code for this example can be found at [9].

```
public abstract class NetworkPeer
                     implements java.rmi.Remote
{
  public void exportMethods()
              throws java.rmi.RemoteException
  {
    java.rmi.server.
      UnicastRemoteObject.exportObject(this);
  }
  public static java.rmi.Remote locatePeer
  (String peerHost, int peerPort, String peerName)
                                 throws Exception {
      return java.rmi.Naming.lookup
        ("rmi://" + peerHost +
              ":" + peerPort + "/" + peerName);
  }
  // ...
}
```

IX. Completing the Transformation

The described transformation is thus straightforward: given a program (code fragment) p involving a set of types (classes) C_i we identify the public methods $m_{i,j}$ of each type and collect them in corresponding interfaces I_i. If we impose that every I_i extends `java.rmi.Remote` then each $m_{i,j}$ needs to throw a `java.rmi.RemoteException`. This, however, has nothing to do with the contents/definition of $m_{i,j}$. That's why, in the end (as explained in both [3] and [4]) p can be run both locally and in distributed fashion without any change.

Part of determining the public methods $m_{i,j}$ involves removing any direct access to instance variables of $o_{i,k}$ objects of type C_i with accessor and mutator methods $a_{i,p}$ which will also have to be included in the interfaces I_i. The code for each class C_i remains unchanged except for the required compliance with I_i (decorations and types of variables that hold the references to the agents—local or remote).

If objects are threads they need to be written in a special way: the only synchronization mechanisms that we encourage

```
interface OddInterface extends java.rmi.Remote
{
    boolean isOdd(int n) throws java.rmi.RemoteException;
    void setEven(EvenInterface peer) throws java.rmi.RemoteException;
}

public class Odd implements OddInterface
{
    EvenInterface even;
    public void setEven(EvenInterface even) throws java.rmi.RemoteException {
        this.even = even;
    }
    public boolean isOdd(int n) throws java.rmi.RemoteException {
        if (n == 0) {
            System.out.println("0 is not odd.");
            return false;
        } else {
            System.out.println("Is " + (n - 1) + " even?");
            return even.isEven(n-1);
        }
    }
}

interface EvenInterface extends java.rmi.Remote
{
    boolean isEven(int n) throws java.rmi.RemoteException;
    void setOdd(OddInterface peer) throws java.rmi.RemoteException;
}

public class Even implements EvenInterface
{
    OddInterface odd;
    public void setOdd(OddInterface odd) throws java.rmi.RemoteException {
        this.odd = odd;
    }
    public boolean isEven(int n) throws java.rmi.RemoteException {
        if (n == 0) {
            System.out.println("0 is even.");
            return true;
        } else {
            System.out.println("Is " + (n-1) + " odd?");
            return odd.isOdd(n-1);
        }
    }
}

class Activator implements java.io.Serializable {
    Object activate() throws java.rmi.RemoteException {
      return null;
    }
}

interface ProcessorInterface extends java.rmi.Remote {
    Object receive(Activator activator) throws java.rmi.RemoteException;
}

class Processor extends NetworkPeer implements ProcessorInterface {

  public void startAsLocalServer() { }
  public void startAsClientOf(java.rmi.Remote peer) { }

  public Object receive(Activator activator) throws java.rmi.RemoteException {
    return activator.activate();
  }

  public static void main(String[] args) {
    String portNumber = args[0], ownName = args[1];
    (new Processor()).startAsNetworkServer(ownName, Integer.parseInt(portNumber));
  }

}
```

Fig. 1. The transformed objects. The need for `NetworkPeer` is essentially removed (except for the convenience in defining the `Processor` class). The reader should note that `Odd` and `Even` (the types C_i) can easily extend other classes so lack of multiple inheritance in Java is no longer a potential problem.

```
class Program {
    public static void main(String[] args) throws Exception {

        ProcessorInterface processorOne =
          (ProcessorInterface) NetworkPeer.locatePeer("blesmol.cs.indiana.edu", 10776, "Dave");

        ProcessorInterface processorTwo =
          (ProcessorInterface) NetworkPeer.locatePeer("molerat.cs.indiana.edu", 10776, "Dave");

        OddInterface   odd;
        EvenInterface even;

        odd  = (OddInterface)  processorOne.receive(
                                 new Activator() {
                                   Object activate() throws java.rmi.RemoteException {
                                     OddInterface odd = new Odd();
                                     java.rmi.server.UnicastRemoteObject.exportObject(odd);
                                     return odd;
                                   }
                                 }
                               );

        even = (EvenInterface) processorTwo.receive(
                                 new Activator() {
                                   Object activate() throws java.rmi.RemoteException {
                                     EvenInterface even = new Even();
                                     java.rmi.server.UnicastRemoteObject.exportObject(even);
                                     return even;
                                   }
                                 }
                               );
        odd.setEven(even);
        even.setOdd(odd);

        System.out.println(odd.isOdd(5));
    }
}
```

Fig. 2. The transformed program. Note that proxies for the distributed agents are created on the fly (at creation/activation time) on the receiving host.

are local, at the level of each thread object. A non-distributed system (even multi-threaded) is layered on top of a single operating system which can be used to determine and aid in synchronization and in the recovery of failure. A distributed system, on the other hand, has no single point of resource allocation, or failure recovery, and thus is conceptually very different. The transformation described here does not interfere with the communication pattern of existing threads (assumed to comply with the set of requirements mentioned above).

So now the given code p can be transformed as follows: every occurrence of `new` $C_i(\ldots)$ is to be replaced with a request to a processor π's `receive` method. A new activator is sent, its `activate` method is coded to create a new object of type C_i and export its methods, then return a reference to the newly created object. It would be possible for these decisions to be done at run-time in which case a more general, and reflective, distributed framework can be built.

X. Conclusions and Directions for Future Work

We have illustrated a simple but powerful and general transformation by which a program can distribute its objects on the network wherever cycles are available. The exception mechanism provided by Java allows for a complete separation of the program logic (the desired) from the unexpected (run-time errors). The approach presented above can be automated completely and future work will show that using reflection and first-class continuations the same approach can be successfully used even when the available processors don't have any knowledge whatsoever of the compiled code.

The approach described here is somewhat opposed to that taken by web services traditionally. By web services we usually mean a large collection of resources written by the world's programmers, in various languages, and integrated using SOAP, WSDL and UDDI. Our perspective is that of a single developer, whose program only needs cycles (processing power) to distribute its requests over them for resources at run-time, in the most straightforward grid-like sense.

References

[1] D. Sasha, C. Lazere, *Out of Their Minds*, Springer-Verlag: 1995.
[2] M. Felleisen, D. Friedman, *A Little Java…*, MIT Press: 1998.
[3] D. German, *The Net Worth …*, ICPADS 2004, Newport Beach, CA.
[4] D. German, *RMI: Observing the …*, FIE 2004, Savannah, GA.
[5] S. Kendall, J. Waldo, A. Wollrath and G. Wyant, *A Note on Distributed Computing*, Sun Microsystems, SMLI Technical Report TR-94-29.
[6] K. Arnold et al., *The Jini Specification …*, Addison-Wesley, 1999.
[7] http://www.javaworld.com/javaworld/jw-08-2002/jw-0809-trmi.html
[8] http://www.jot.fm/issues/issue_2002_11/article2
[9] http://www.cs.indiana.edu/~dgerman/icpads05.html
[10] P. Perrone et al., *J2EE Developer's Handbook*
[11] Berman, Fox and Hey, *Grid Computing*.