On the Symbiotic Interplay between Good Object-Oriented Design and Distributed Programming

Adrian German, Indiana University School of Informatics, Computer Science Department

Abstract:

Network programming, traditionally considered an advanced topic is usually approached with trepidation and fear by both instructors and students. But languages and tools developed in the last decade have completely taken the gnarl out of a topic that (as we show) can now be easily taught early in CS2 or even late in CS1. We describe an exercise that can be used to discuss good object oriented design but also to show that if simple such design principles are followed one can distribute the objects in the program at will, with virtually no effort, obtaining a networked version of the program almost as an afterthought. The collection of examples mentioned here can successfully be used to offer a brief but fairly complete introduction to Java, with an emphasis on networked programming using RMI.

Introduction

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 (inheritance, polymorphism, etc.) were able to acquire an identity of their own that early.

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.

Java RMI vs. the 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) that isolates the basic functionality of a free agent.

Using this class in [9] we demonstrated 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 were just normal objects (POJOs), for simplicity.

Issues of Concurrency

Of the four major differences identified 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.

We now present the example.

Interfaces

Interfaces are elements of pure design.

Let's consider the following example for a start:

```
public interface Client {
  public void update(Update event);
}
```

It seems to indicate that a Client is supposed to be something that can be sent an update.

Here's what a Client might actually look like:

```
public class ClientImplementation extends Thread implements Client {
  String name;
  int id;
  Server server;
  public ClientImplementation(String name) {
    this.name = name;
  }
  public void update(Update event) {
    System.out.println(this.name + " receives: ***(" + event + ")*** ")
  }
  public void run() {
    while (true) {
      try {
        sleep((int)(Math.random() * 6000 + 10000));
        server.broadcast(new Update(this.name + " says: Howdy!"));
      } catch (Exception e) { }
    }
  }
  public void startAsClientOf(Server server) {
    this.id = server.register(this);
    this.server = server;
    this.start();
  }
}
```

The reader is invited to focus on the definition of the `update(…)` function only, which is the only item of real interest here, for now. The focus at this time is on the concept of interface.

An interface is just a uniform.

A uniform is just an advertisment for a set of conventions.

Here's a picture of Frank Sinatra and Trevor Howard impersonating two Nazi solders in the 1965 movie *von Ryan's Express*. They both *looked* like Nazi soldiers but none of them spoke any German…

Every uniform comes with a set of expectations. An umpire is expected to officiate, a mailman to deliver mail,a graduating senior to graduate, a policeman to direct traffic, etc. (see below).

The use of some uniforms is restricted to certain groups of qualified people:

> BARCELONA, Spain (AFP) - Police in north-eastern Spain have arrested a man who dressed up as a policeman and then directed traffic for about an hour, a Spanish newspaper reported Thursday.
>
> The 29-year-old, who has a history of mental problems, entered a police station in the town of Lleida on Wednesday, where he forced open a locker and stole a uniform.
>
> Greeting other officers as he left the station, he then went on "patrol," directing traffic in the town centre, the Catalan daily Segre reported.
>
> He caught the eye of police because of his dishevelled appearance, his "bizarre gestures" and the fact that he was not carrying a weapon or wearing a belt.
>
> The man was charged with impersonating a policeman.

In our code example `Client`s need to provide an implementation for the advertised `update(…)` function, otherwise objects of this kind cannot (and will not) be recognized as `Client`s.

Here's another interface to consider as an example:

```
public interface Server {
  public int register(Client client);
  public void broadcast(Update event);
}
```

Here's a possible implementation of this interface:

```
public class ServerImplementation implements Server {
  Client[] clients = new Client[100];
  int index = -1;
  synchronized public int register(Client client) {
    clients[++index] = client;
    return index;
  }
  synchronized public void broadcast(Update event) {
    for (int i = 0; i <= index; i++)
      clients[i].update(event);
  }
  String name;
  public ServerImplementation(String name) {
    this.name = name;
    System.out.println("Server being initialized ... ");
  }
  public void startAsLocalServer() {

  }
}
```

The following definition is standalone:

```
public class Update {
  Update(String message) {
    this.message = message;
  }
  String message;
  public String toString() {
    return message;
  }
}
```

We can put all this machinery in action with:

```
public class LocalSetup {
  public static void main(String[] args) {

    ServerImplementation server = new ServerImplementation("dave");
    server.startAsLocalServer();

    (new ClientImplementation("larry")).startAsClientOf(server);

    (new ClientImplementation("michael")).startAsClientOf(server);

    (new ClientImplementation("toni")).startAsClientOf(server);

  }
}
```

Here's how it goes:

```
-bash-3.2$ java LocalSetup
Server being initialized ...
larry receives: ***(toni says: Howdy!)***
michael receives: ***(toni says: Howdy!)***
toni receives: ***(toni says: Howdy!)***
larry receives: ***(michael says: Howdy!)***
michael receives: ***(michael says: Howdy!)***
toni receives: ***(michael says: Howdy!)***
larry receives: ***(larry says: Howdy!)***
michael receives: ***(larry says: Howdy!)***
toni receives: ***(larry says: Howdy!)***
larry receives: ***(toni says: Howdy!)***
michael receives: ***(toni says: Howdy!)***
toni receives: ***(toni says: Howdy!)***
-bash-3.2$
```

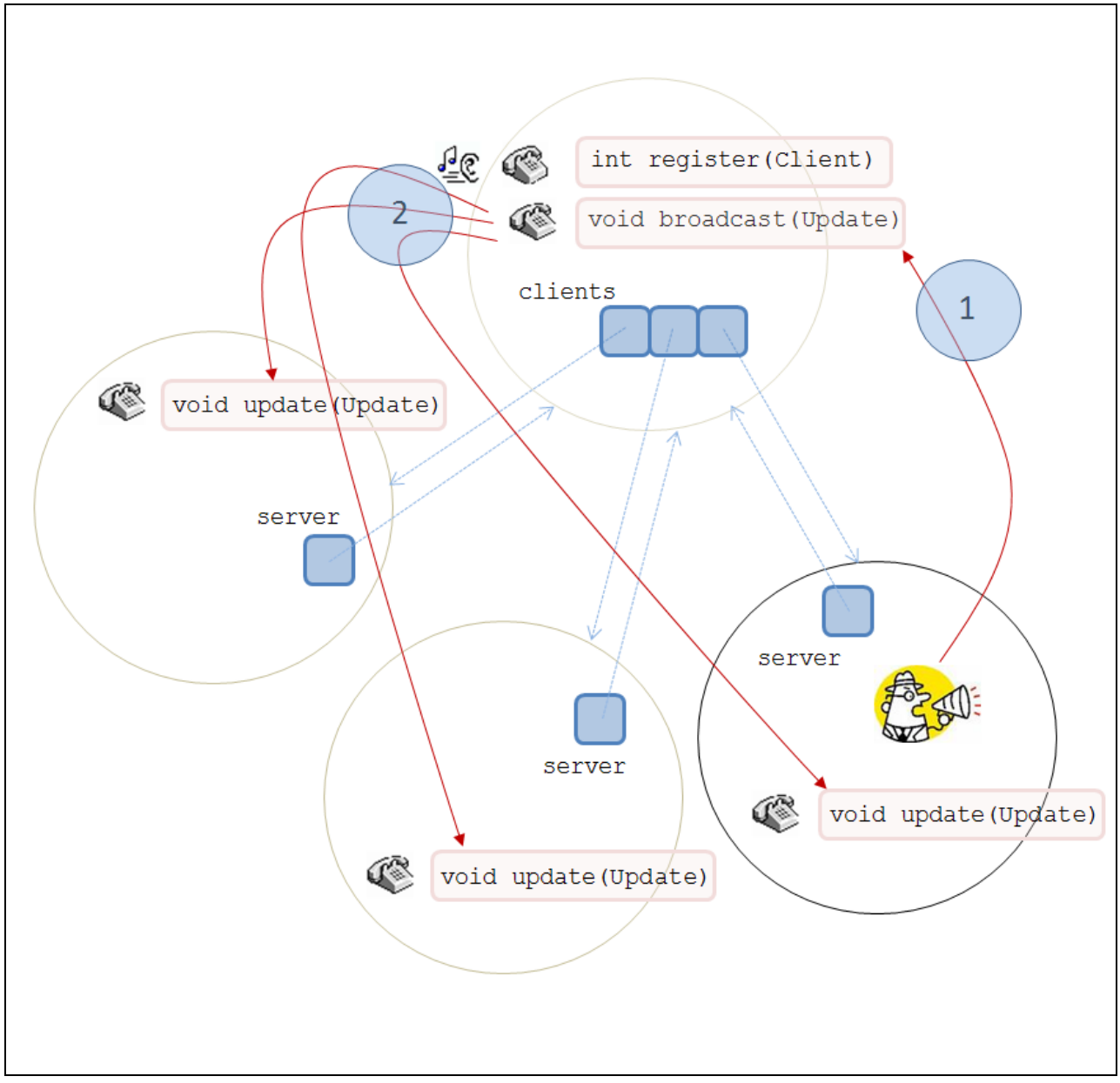
So this is a (realistic) mock-up of a real-time text messaging (IM, chat) system.

It's a scaled-down, simulated, real-time chat system.

To understand how this works we probably need to start with the client and server methods invoked in the local setup's main method, as well as the constructors of the two types of objects (if any).

Diagrams

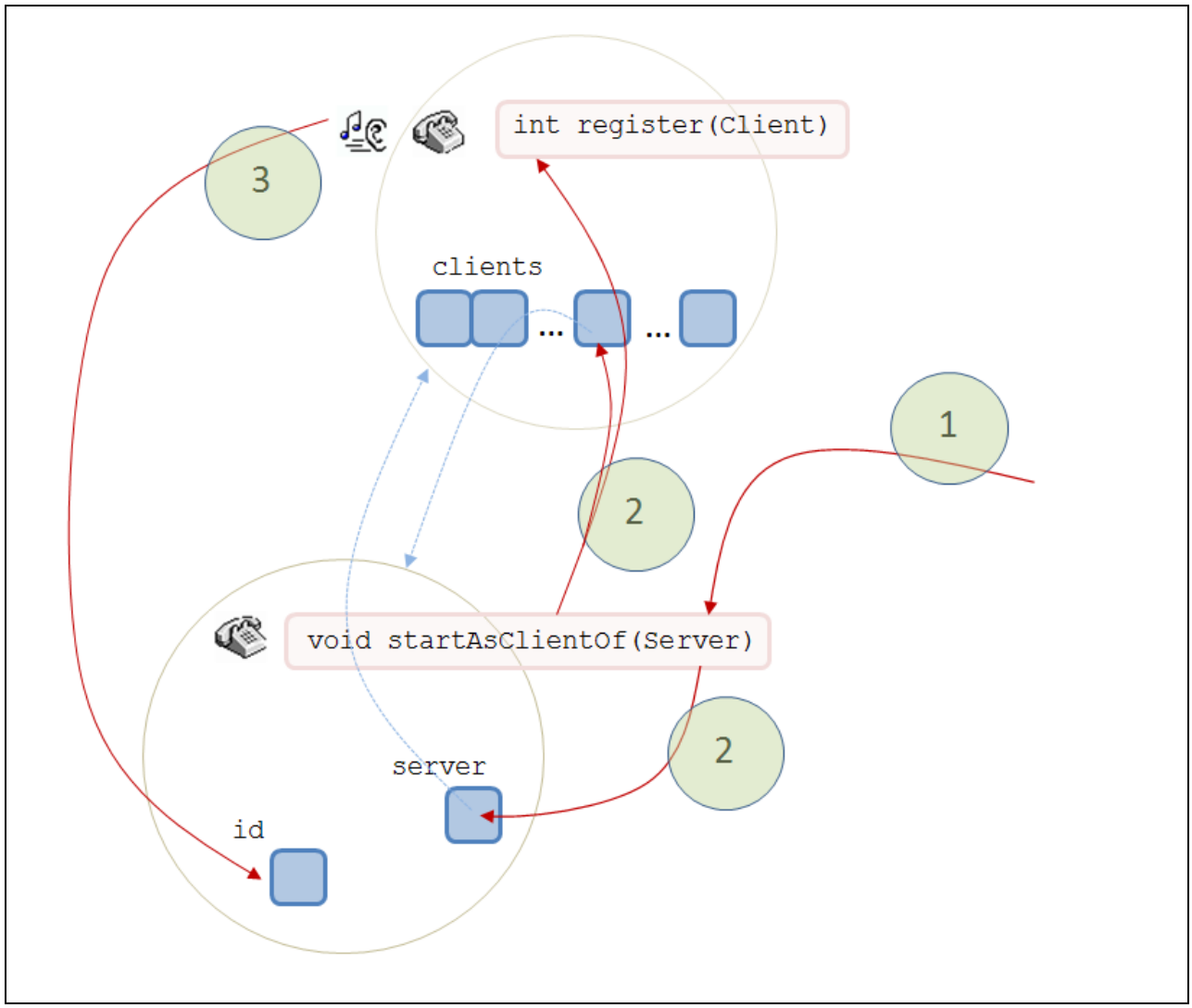
It will help to draw a schematic diagram of how the program works:



The diagram mostly shows that happens when a client has something to say: the client calls the server's broadcast method and passes the information to it. The server's braodcast method in turn calls each client's update method, thus making sure that all the clients (including the originator) will receive the information. For this to work the client needs to have the server's address and the server needs to know where to find the clients. This is taken care of during the registration phase when everything's starts up.

Here's what happens at startup:

1. The server's address is public and known so somebody starts the client with this information.
2. A reference to the server is passed to the client's method startAsClientOf which results in the client storing the reference for later use and the server's register method to be called through this reference. The client passes a references to itself during this call by way of introductions.
3. The client receives an id, which is the return value of the register call. That id is also an index in an array of client references (the server's address book).



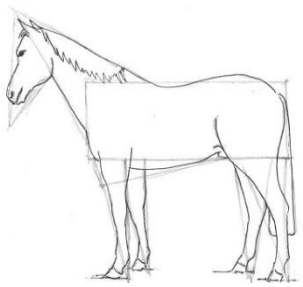
Then what we described earlier can start.

At this point we have a few objects that simulate (as part of the same program, address space) a chat system. The question is: what can or should we do if we want the clients and the servers to be not necessarily on different machines but definitely in different address spaces (created in separate main methods, classes, by different programs)?

Inheritance

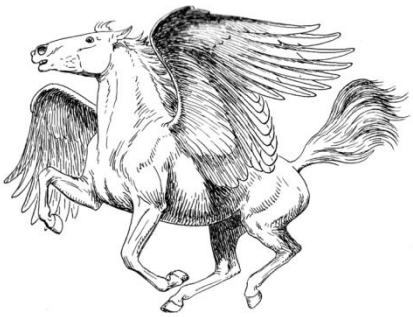
Inheritance is what happens when we use the class extension mechanism: via this mechanism we are allowed to describe any entity in stages. For example a unicorn is a horse with a horn. Defining a class Unicorn that extends Horse and inherits all the attributes (variables and methods, data and behavior) of the class Horse to which we add the distinguishing feature of the new concept (the horn for the Unicorn) is what the class extension mechanism allows and it all amounts to a simple operation of set union of features of the two descriptions. Dynamic method lookup is what happens when the added features have the same name with the inherited features, a phenomenon that's called overriding. Overriding is a very useful feature, and the special rule used in dynamic method lookup (allow the invocation if the type of the reference is compatible, use the type of the actual object to determine the actual method) is entirely justified because the set of features is in this case a multiset. So the ambiguity must be resolved.

So a Horse needs to be defined first:



Then a Unicorn can be defined by extension as a Horse with a horn

Pegasus can be likewise defined as a Horse with wings:



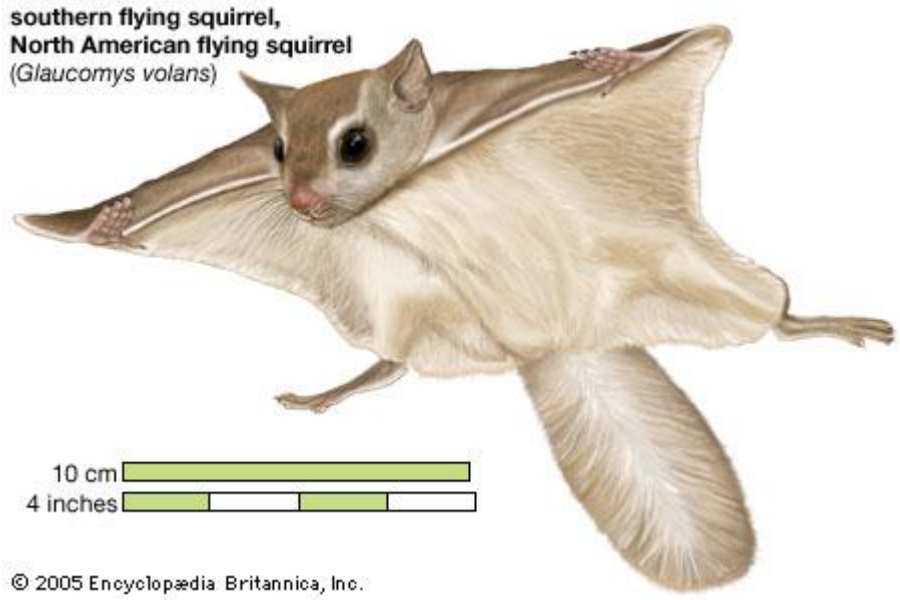
But these are imaginary creatures. Can we find a real example in nature?

The answer is yes: the flying squirrel is a squirrel with wings.

From Wikipedia, the free encyclopedia:

The flying squirrels, scientifically known as Pteromyini or Petauristini, are a tribe of squirrel (family Sciuridae). There are 43 species in this tribe, the largest of which is the woolly flying squirrel (Eupetaurus cinereus). The 2 species of the genus Glaucomys (Glaucomys sabrinus and Glaucomys volans) are native to North America, and the Siberian flying squirrel is native to parts of northern Europe (Pteromys volans).

The term "flying" is somewhat misleading, since flying squirrels are actually gliders incapable of true flight. Steering is accomplished by adjusting tautness of the patagium, largely controlled by a small cartilaginous wrist bone. The tail acts as a stabilizer in flight, much like the tail of a kite, and as an adjunct airfoil when "braking" prior to landing on a tree trunk.

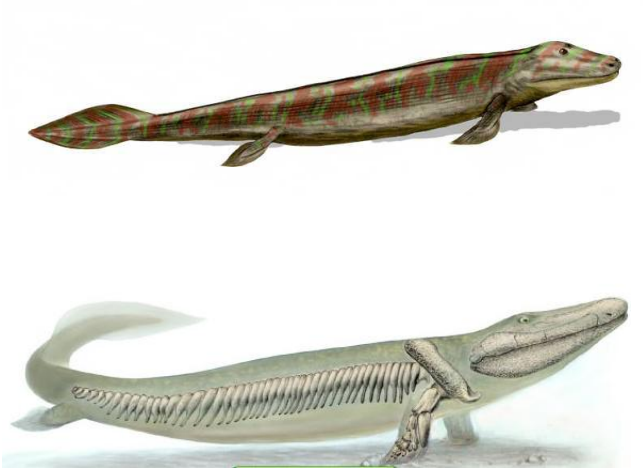


Though their life expectancy is only about six years in the wild, flying squirrels often live between 10 and 15 years in captivity. This difference is due to these creatures being important prey animals. Predation mortality rates in sub-adults are very high. Predators include arboreal snakes, raccoons, nocturnal owls, martens, fishers, coyotes, and the domestic house cat. In the Pacific Northwest of North America, the Northern Spotted Owl (*Strix occidentalis*) is a well-known predator. They are also nocturnal. They have been kept as pets since the US colonial era.

It appears that these animals have developed the extra feature during time, possibly through selection, adaptation and so on. The situation is probably similar to the one described below:

On 6 April 2006 the journal Nature featured on its cover the banner headline "When Fins became Limbs."

Inside were two articles variously authored by Neil Shubin of the University of Chicago, Ted Daeschler of the Academy of Natural Sciences in Philadelphia and Farish A. Jenkins Jr. of Harvard. These articles gave details of an exciting new fossil find that seem to offer to fill a missing link in the understanding of evolution with respect to how fish evolved towards becoming land animals



These extremely well-preserved fossil remains were found on Ellesmere Island, a Canadian island lying within the Arctic Circle, and date from some 383 million years ago during the Devonian Era of Geological Time. They are of a predator that had sharp teeth, a crocodile like head and could grow to a body length of some 2.75 metres (9ft).

The fossil finds - billed in some newspapers as "one of the most important fossil finds in history," have caused excitement because Tiktaalik roseae had a skull, neck and ribs similar to early limbed land animals, (known as tetrapods), but also featured several defining characteristics of fish such as fins, scales and a relatively primitive jaw.

Scientists have been on the look out for the fossil remains of animals that could provide a practical example of the previously recognised missing link in the understanding of evolution concerning the transformation of the original aquatic life forms into life forms that lived on the land. […]
Shubin also said that "here is a creature that has a fin that can do push-ups," and that "this is clearly an animal that is able to support itself on the ground." Thus it is reasonable to suggest that these fossil remains may well give evolutionary biologists a new understanding of how fins turned into limbs.
The fossil remains so far discovered include one nearly complete front half of a fossilised skeleton but do not include reliable details of the hind fins and tail of the creatures body. Given this fact the scientists suggest that Tikkalik would have lived mainly in water but could have moved like a seal on land.

Tikktaalik was a meat eater so the forces driving the species onto land can be speculated upon as to whether they did so to scavenge, to hunt, to breed or to escape their own predators.

Let’s see if we can do that in our program as well.

Growing Wings

Growing wings implies that we are making structural changes to ourselves.

We are the code.

So we have the following changes to the code:

```
-bash-3.2$ cat Client.java
import java.rmi.*;

public interface Client extends Remote {
  public void update(Update event) throws RemoteException;
}

-bash-3.2$ cat ClientImplementation.java
import java.rmi.*;

public class ClientImplementation extends Thread implements Client {
  String name;
  int id;
  Server server;
  public ClientImplementation(String name) {
    this.name = name;
  }
  public void update(Update event) throws RemoteException {
    System.out.println(this.name + " receives: ***(" + event + ")*** ");
  }
  public void run() {
    while (true) {
      // System.out.println("Hello, this is: " + this.name);
      try {
        sleep((int)(Math.random() * 6000 + 10000));
        server.broadcast(new Update(this.name + " says: Howdy!"));
      } catch (Exception e) {
        // System.out.println("Ouch: " + e);
      }
    }
  }
  public void startAsClientOf(Server server) throws RemoteException {
    this.id = server.register(this);
    this.server = server;
    this.start();
    System.out.println("I have started the client " + this.name);
  }
}
-bash-3.2$
```

Changes like these are not difficult to make but they permeate the entire code:

```
-bash-3.2$ cat Server.java
import java.rmi.*;

public interface Server extends Remote {
  public int register(Client client) throws RemoteException;
  public void broadcast(Update event) throws RemoteException;
}

-bash-3.2$ cat ServerImplementation.java
import java.rmi.*;

public class ServerImplementation implements Server {
  Client[] clients = new Client[100];
  int index = -1;
  synchronized public int register(Client client) throws RemoteException {
    clients[++index] = client;
    return index;
  }
  synchronized public void broadcast(Update event) throws RemoteException {
    for (int i = 0; i <= index; i++)
      clients[i].update(event);
  }
  String name;
  public ServerImplementation(String name) {
    this.name = name;
    System.out.println("Server being initialized ... ");
  }
  public void startAsLocalServer() {

  }
}
-bash-3.2$
```

One can't say that these changes are of substance.

- They are mostly decorative (although important).
- They don't interfere in any way with the actual code that we wrote.

The wings per se are these two classes which are new:

```
-bash-3.2$ cat NetworkClient.java
import java.rmi.*;
import java.rmi.server.*;

public class NetworkClient extends ClientImplementation
                           implements Client {
```

```
  public NetworkClient(String name) throws RemoteException {
    super(name);
    UnicastRemoteObject.exportObject(this);
  }

  public static void main(String[] args) {
    try {
      Server far =
        (Server)Naming.lookup(
          "rmi://" + args[0] + ":" + args[1] + "/Dirac");

      NetworkClient here = new NetworkClient(args[2]);

      here.startAsClientOf(far);

    } catch (Exception e) {
      System.out.println("Error in client... " + e);
      e.printStackTrace();
    }

  }

}
-bash-3.2$
```

The other class provides the wings for the server objects:

```
-bash-3.2$ cat NetworkServer.java
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class NetworkServer extends ServerImplementation
                           implements Server {

  public NetworkServer(String name) throws RemoteException {
    super(name);
    UnicastRemoteObject.exportObject(this);
    System.out.println("Server being initialized... ");
  }

  public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());
    try {
      NetworkServer pam = new NetworkServer("dave");
      Registry cat =
        LocateRegistry.createRegistry(Integer.parseInt(args[0]));
```

```
        cat.bind("Dirac", pam);

         pam.startAsLocalServer();

        System.out.println("Server is ready... ");
      } catch (Exception e) {
        System.out.println("Server error: " + e + "... ");
      }

    }

}
-bash-3.2$
```

The set up changes a little, as each object is launched separately, in a different main method:

```
-bash-3.2$ javac *.java
-bash-3.2$ rmic NetworkClient NetworkServer
-bash-3.2$ cat java.policy
grant {
  permission java.net.SocketPermission "*", "connect,accept";
};
-bash-3.2$ java -Djava.security.policy=java.policy NetworkServer 12398
Server being initialized ...
Server being initialized...
Server is ready...
[...]
-bash-3.2$
```

Clients are also to be started separately. Here's how we start one of them:

```
-bash-3.2$ java NetworkClient 127.0.0.1 12398 larry
I have started the client larry
[...]
```

Any other client is started in the same way:

```
-bash-3.2$ java NetworkClient 127.0.0.1 12398 michael
I have started the client michael
[...]
```

Of Mice and Men[1]

The title only wants to indicate we plan to make a comparison. Squirrel are rodents.

We started with normal squirrels and we helped them grow wings.

This is what we have now:

These mutated, flying objects can still be run in the same address space.

They still act as squirrels when they don't need to fly.

[1]Of Rodents and Humans?

What we're trying to say is that the original local setup program works the same, with minimal updates:

```
-bash-3.2$ cat LocalSetup.java
import java.rmi.*;

public class LocalSetup {
  public static void main(String[] args) throws RemoteException {

    ServerImplementation server = new ServerImplementation("dave");
    server.startAsLocalServer();

    (new ClientImplementation("larry")).startAsClientOf(server);

    (new ClientImplementation("michael")).startAsClientOf(server);

    (new ClientImplementation("toni")).startAsClientOf(server);

  }
}
-bash-3.2$
```

What do we, people, do in similar circumstances?

We can't grow wings. (I mean this is not Star Trek!)

So we approach the problem in reverse:

- instead of starting with a normal person and adding a new feature: wings
- we start with a normal set of wings and let anybody add a new feature: a person

So we have this:

In other words instead of creating NetworkClient and NetworkServer that extend the already existing classes that implement the clients and the server we define a more general class (say, NetworkPeer) and we make any implementation extend them. People remain unchanged. (Squirrels did not.)

```
-bash-3.2$ cat NetworkPeer.java
public abstract class NetworkPeer implements java.rmi.Remote {
    public void exportMethods() throws java.rmi.RemoteException {
        java.rmi.server.UnicastRemoteObject.exportObject(this);
    }
    public java.rmi.Remote locatePeer(String peerHost,
                                      int peerPort,
                                      String peerName) throws Exception {
        return java.rmi.Naming.lookup("rmi://" + peerHost + ":" + peerPort + "/" + peerName);
    }
    public void startAsNetworkClientOf(String peerHost,
                                       int peerPort,
                                       String peerName) throws Exception {
        this.exportMethods();
        java.rmi.Remote peer = this.locatePeer(peerHost, peerPort, peerName);
        this.startAsClientOf(peer); // see below ...
    }
    public abstract void startAsClientOf(java.rmi.Remote peer) throws java.rmi.RemoteException;
    public void startAsNetworkServer(String name, int port) {
        System.setSecurityManager(new java.rmi.RMISecurityManager());
        try {
            this.exportMethods();
            java.rmi.registry.Registry registry = java.rmi.registry.LocateRegistry.createRegistry(port);
            registry.bind(name, this);
            this.startAsLocalServer(); // see below ...
            System.out.println("Server is ready ... ");
        } catch (Exception e) {
            System.out.println("Server error: " + e + " ... ");
        }
    }
    public abstract void startAsLocalServer(); // perhaps a better name would be startAsServer...
}
-bash-3.2$
```

The client code becomes:

```
-bash-3.2$ cat Client.java
public interface Client extends java.rmi.Remote {
  public void update(Update event) throws java.rmi.RemoteException;
}

-bash-3.2$ cat ClientImplementation.java
public class ClientImplementation extends NetworkPeer implements Runnable, Client {
  String name;
  int id;
  Server server;
  public ClientImplementation(String name) {
    this.name = name;
  }
  public void update(Update event) throws java.rmi.RemoteException  {
    System.out.println(this.name + " receives: ***(" + event + ")*** ");
  }
  public void run() {
    while (true) {
      try {
        Thread.sleep((int)(Math.random() * 6000 + 10000));
        server.broadcast(new Update(this.name + " says: Howdy!"));
      } catch (Exception e) { }
```

```
    }
  }
  public void startAsClientOf(java.rmi.Remote server) throws java.rmi.RemoteException {
    this.id = ((Server)server).register(this);
    this.server = (Server)server;
    (new Thread(this)).start();
  }
  public static void main(String[] args) throws Exception {
     String
       ownName = args[0],
       serverHostName = args[1],
       serverPortNumber = args[2],
       serverName = args[3];

     ClientImplementation client = new ClientImplementation(ownName);
     client.startAsNetworkClientOf(serverHostName,
                                   Integer.parseInt(serverPortNumber),
                                   serverName); // calls startAsClientOf
  }
  public void startAsLocalServer() { }
}
-bash-3.2$
```

The server code is similar (your only chance is to read this online, and zoom):

```
-bash-3.2$ cat Server.java
public interface Server extends java.rmi.Remote {
  public int register(Client client) throws java.rmi.RemoteException;
  public void broadcast(Update event) throws java.rmi.RemoteException;
}

-bash-3.2$ cat ServerImplementation.java
public class ServerImplementation extends NetworkPeer implements Server, java.rmi.Remote {
  Client[] clients = new Client[100];
  int index = -1;
  synchronized public int register(Client client) throws java.rmi.RemoteException {
    clients[++index] = client;
    return index;
  }
  synchronized public void broadcast(Update event) throws java.rmi.RemoteException {
    for (int i = 0; i <= index; i++)
      clients[i].update(event);
  }
  String name;
  public ServerImplementation(String name) {
    this.name = name;
    System.out.println("Server being initialized ... ");
  }
  public void startAsLocalServer() {

  }
  public static void main(String[] args) {
    String
      portNumber = args[0],
      ownName = args[1];
    ServerImplementation here = new ServerImplementation(ownName);
    here.startAsNetworkServer(ownName,
                              Integer.parseInt(portNumber));
  }
  public void startAsClientOf(java.rmi.Remote peer) {

  }
}
-bash-3.2$
```

What we notice now is that instead of having two types of remote objects (one for the server and one for the clients) we have only one (which we generically call NetworkPeer). All people fly the same kind of device. Not all flying squirrels are the same, however!

There are no other differences, the remaining code is like before:

```
-bash-3.2$ cat Update.java
public class Update implements java.io.Serializable {
  Update(String message) {
    this.message = message;
  }
  String message;
  public String toString() {
    return message;
  }
}
-bash-3.2$ cat LocalSetup.java
public class LocalSetup {
  public static void main(String[] args) throws /*Remote*/Exception {
    ServerImplementation server = new ServerImplementation("dave");
    server.startAsLocalServer();

    (new ClientImplementation("larry")).startAsClientOf(server);

    (new ClientImplementation("michael")).startAsClientOf(server);

    (new ClientImplementation("toni")).startAsClientOf(server);

  }
}
-bash-3.2$
```

Overall the two methods are somewhat different, as explained, yet they're also similar.

A remote object.

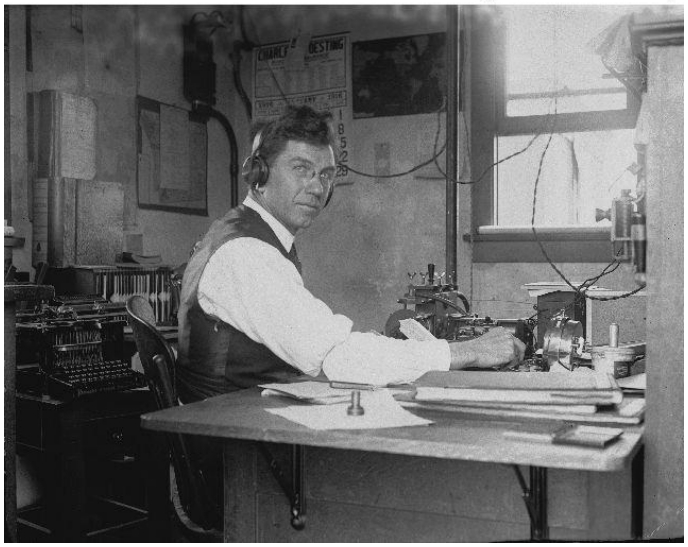
This person extends a pair of headphones instead of extending a glider. The effect is the same.

Here are two remote objects that are arguing in the same address space:

It is important to understand that having the headphones on is virtually transparent to objects, as in the picture above. Good object-oriented design allows us to distribute the objects at will, easily, as needed.

The two remote objects do not need to adjust in any way the protocol that they normally use. This is the great advantage of RMI. When using sockets the network protocol is harder to ignore, communication is not as transparent:

San Diego Historical Society

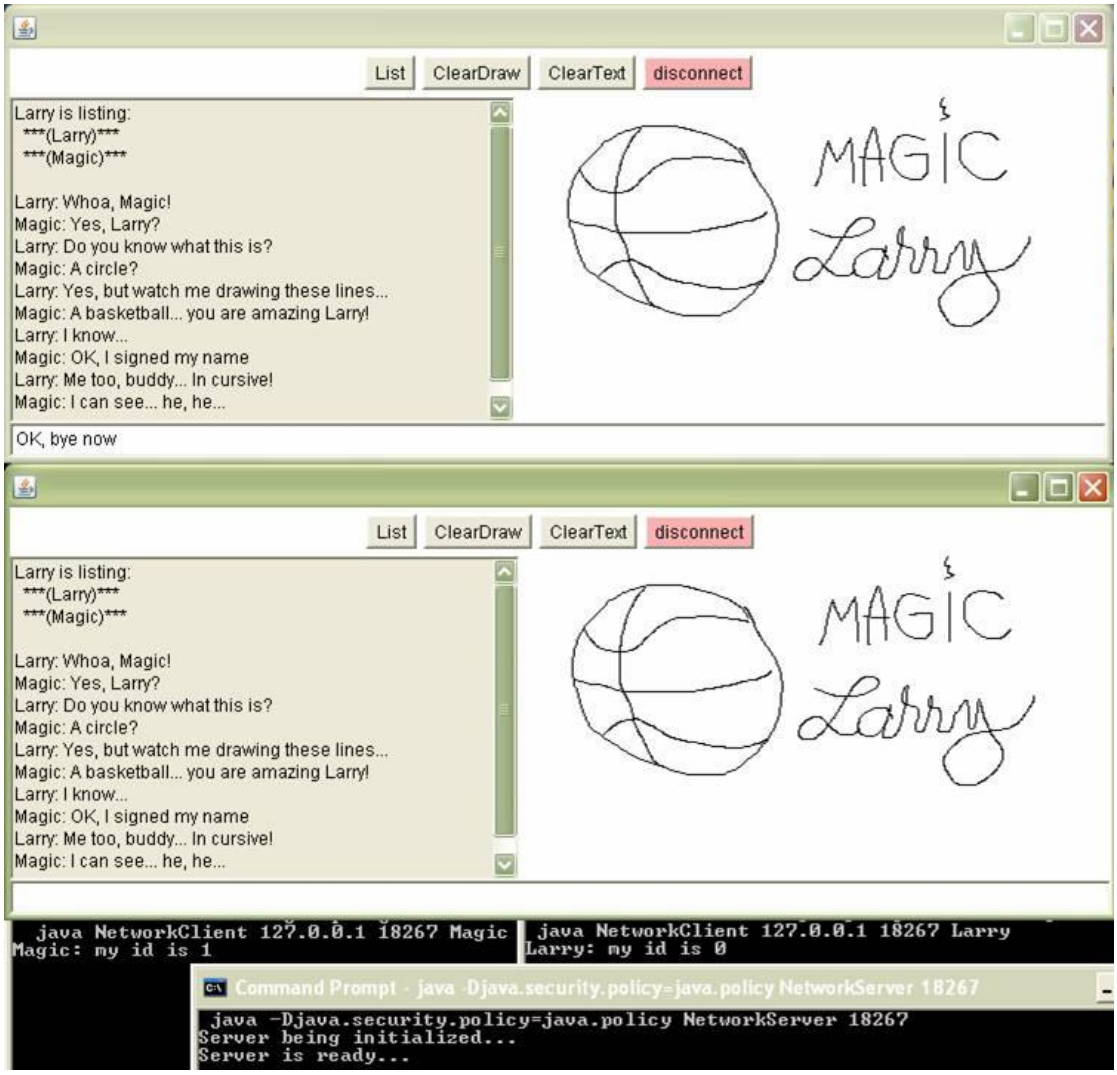


Outdated means of communicating over the network use various inflexible protocols.

There is only one problem with RMI: firewalls. However the cajo package solves this problem easily. The basic structure changes a bit but the transparent aspect of networking is maintained.

Here are other examples used in class.

This was a shared whiteboard with chat implemented locally.

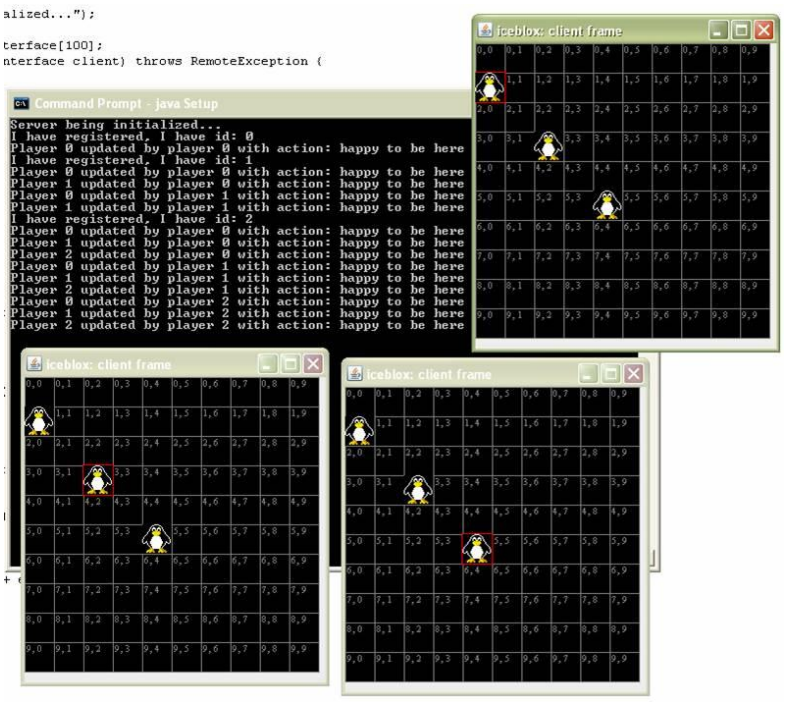


The same implemented and run as a distri buted program.

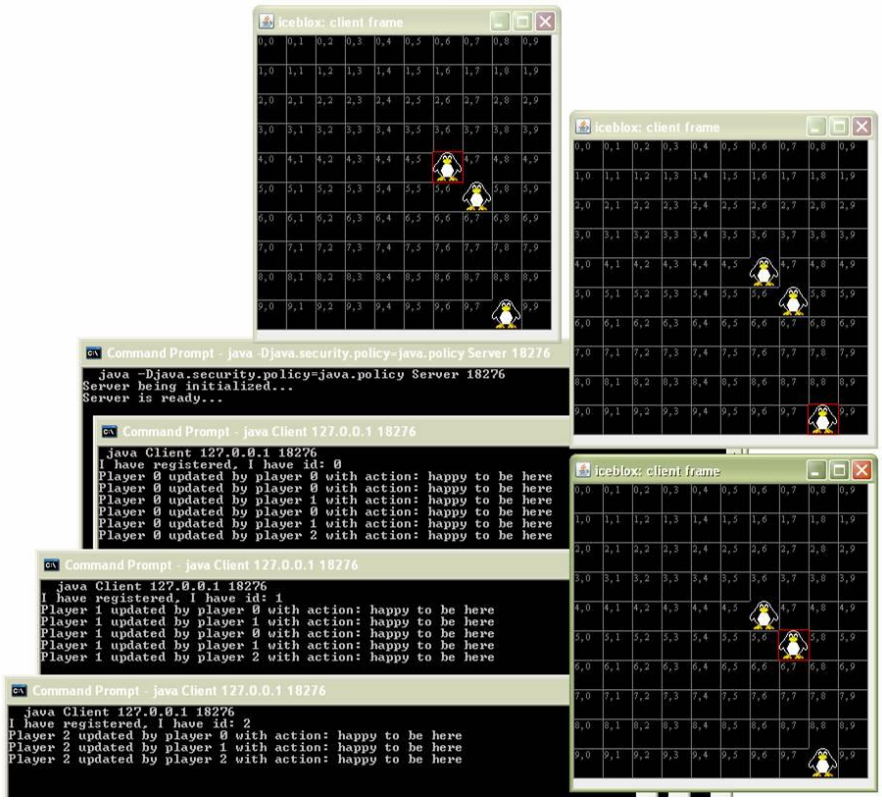
Another example we use is a very simple multiplayer game:

- Every time a player connects (s)he becomes a Penguin and is rendered as such.
- The players then can move their avatars and everything is in real time.
- Own avatar is identified by red square surrounding it.

Below we have the program run on the same machine in the same address space:



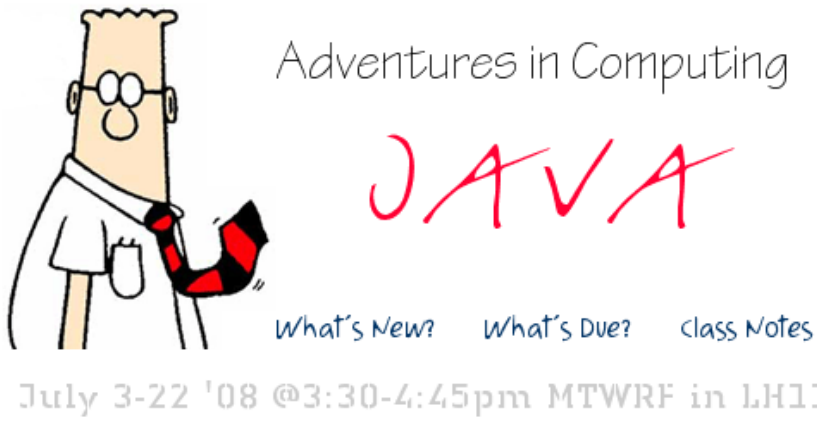
Here's how the exact same program is started on the samemachine but with each client in a different address space, as a separate program:



ou know how this works by now.

Conclusions and Future Work

Integrate with `cajo` in such a way that we can still stay at this introductory level.



Assume you have the package.

The following pair of programs is a minimal chat system implemented in cajo:

```
// javac -classpath .;cajo.jar Server.java
// java -classpath .;cajo.jar Server localhost 18239
// java -classpath .;cajo.jar Server silo.cs.indiana.edu 1198

import gnu.cajo.invoke.Remote;
import gnu.cajo.utils.ItemServer;
import gnu.cajo.invoke.RemoteInvoke;
import gnu.cajo.utils.extra.ClientProxy;

import java.util.*;

public class Server {
```

```
  Server(String host, int port) throws Exception {
    this();
    Remote.config(host, port, host, port);
    ItemServer.bind(this, "Melville");
  }

  Server () {
    System.out.println("Initializing server...");
  }

  Vector chatters = new Vector();

  public static void main(String[] args) throws Exception {
    new Server(args[0], Integer.parseInt(args[1]));
  }

  public RemoteInvoke register(String message) throws Exception {
    System.out.println(message);
    ClientProxy cp = new ClientProxy();
    chatters.add(cp);
    return cp.remoteThis;
  }

  public void broadcast(String message) {
    for (int i = 0; i < chatters.size(); i++) {
      ClientProxy cp = (ClientProxy) (chatters.elementAt(i));
      try {
        Remote.invoke(cp, "update", message);
      } catch (Exception e) {
        System.out.println("Client unavailable...");
        System.out.println(cp);
        System.out.println(e);

      }
    }
  }

}
```

That was the Server, here now is the Client:

```
// javac -classpath cajo.jar;. Client.java
// java -classpath cajo.jar;. Client silo.cs.indiana.edu 1198 Larry
// java -classpath cajo.jar;. Client localhost 19823 Michael

import gnu.cajo.invoke.Remote;
import gnu.cajo.invoke.RemoteInvoke;
import gnu.cajo.utils.extra.ItemProxy;

import java.util.Scanner;

public class Client {

  String name;
```

```
  public Client(String name) {
    System.out.println("Starting up client...");
    this.name = name;
  }

  Object server;

  ItemProxy ip;

  public Client (String host, int port, String name) throws Exception {
    this(name);
    server = Remote.getItem("//" + host + ":" + port + "/Melville");
    RemoteInvoke cp =
      (RemoteInvoke) Remote.invoke( server,
                                    "register",
                                    this.name + " is registering now.");
    ip = new ItemProxy(cp, this);
    this.talk();
  }

  public void update(String message) {
    System.out.print(message);
  }

  public static void main(String[] args) throws Exception {
    new Client(args[0], Integer.parseInt(args[1]), args[2]);
  }

  public void talk() {
    try {
      Scanner scanner = new Scanner(System.in);
      String line = scanner.nextLine();
      while (! line.equals("quit")) {
        System.out.println("You typed: " + line);
        // server.broadcast(name + "> " + line + "\n");
        Remote.invoke(server, "broadcast", name + "> " + line + "\n");
        line = scanner.nextLine();
      }
    } catch (Exception e) {
      System.out.println("Client: error in talk..." + e);
    }
  }

}
```

This works fine *through any firewall*. But as it is can't be run both ways (same address space, different address space) with no changes. But it can easily be made to do that (which is what we will do next). Additionally, it has all the other advantages it had plus a few more. Same drop in architecture, can it be simplified any further? And, isn't it important to explain even **cajo**? So these are the goals for our next steps:

- simplify the use of cajo to the extreme and propose a general pattern
- explain the inner workings of **cajo**

References

There will be a list of 12-14 references here. The links below take us to the code presented in the paper and to the overview of the cajo page:

http://www.cs.indiana.edu/~dgerman/tutorials/001/m01.html for source code.

https://cajo.dev.java.net/overview.html for the cajo project. The code becomes a bit more complex but we can tunnel through firewalls with absolutely no problem (as seen in the crude example above, that we need to try simplify further -- reflection at run time).