JDOS: A Jini Based Distributed Operating System

M. Saywell and J.S. Reeve

The Department of Electronics and Computer Science
University of Southampton, Southampton SO17 1BJ, UK
email jsr@ecs.soton.ac.uk

Abstract. J-DOS provides and integrated JAVA environment for the execution of a program across an interconnected network of heterogeneous computers. The system provides a file system, shared memory and a distributed execution scheme, all of which is transparent to the user. The framework used to provide these service is sufficiently general as to allow the provision of extra services by the user. We describe the client-server execution, remote execution and the shared file system, paying particular attention to the techniques used to distribute threads over many nodes. Distributed Mandelbrot set generation and rendering is used to benchmark and validate the the remote execution and load balancing aspects of the system.

1 Introduction

The purpose of a computer cluster is to present a number of physically distinct machines as a single networked virtual computer[2]. However such systems are typically only distributed at the processing level, requiring independently configured third party software to provide distributed functionality. For example NFS[6] is commonly used to share file systems and X11[3] provides support for remote GUIs. Our purpose is to provide the essential distributed services of remote processing, shared memory and a common storage medium and offer a framework which is readily extensible so that supplementary services can be added with minimal programming overhead. In contrast to high performance computing environments like Beowulf[5] which usually provide The Message Passing Interface[8], J-DOS provides the ability to adapt and scale in an un-managed way. Additionally the system doesn't attempt to disguise its distributed nature and instead allows the programmer to distribute processes when it is most suitable to do so.
J-DOS provides a dynamic extensible distributed computing environment in 100% pure Java. The system includes a file system, shared memory and a distributed processing environment. Provision is made for extra services to be added by the user without detailed knowledge of the system or the underlying Jini and RMI behaviour. The J-DOS environment is "zero configuration", in the sense that the introduction of a new service is made simply by running it to trigger the automatic registration of the service which then becomes available to

clients. Finally, J-DOS provides classes which automate the thread distribution and collection processes.

2 Other Systems

There are two systems, similar to our own, that aim at providing a distributed Java environment.
Javaparty[4] was developed by the Institute for Program Structures and Data Organisation at the University of Karlsruhe, and aims to provide transparent remote objects in a Java environment. An additional class modifier ,"remote", is introduced into the language which is interpreted by a pre-processor that generates the necessary RMI calls and interfaces and deals with network exception handling. Javaparty distributes instances of any "remote" objects among other machines, thereby spreading the load over all machines in the cluster. "While regular Java classes are limited to one Java virtual machine, remote classes and their instances are visible and accessible anywhere in the distributed JavaParty environment. As far as remote classes are concerned, the JavaParty environment can be viewed as a Java virtual machine that is distributed over several computers"[4].
Cluster Virtual Machine for Java[10, 11], cJVM, was developed at the IBM Haifa Laboratory and "provides a single system image of a traditional JVM while execution on a cluster"[10]. The underlying architecture of cJVM is similar to RMI, so when an object is located on a remote node, the local node communicates with it via a proxy object. From the applications perspective, the object and its proxy are indistinguishable. This allows multi-threaded programs to benefit from the cluster with no change to the code. Although threads are executed on different nodes, they cannot migrate between them. The load sharing function allocates a new thread to the best suited node when the thread was started, consequently load imbalance can arise when over time as the system load changes.

3 Description

In the following we describe the components of JDOS; a communications backend, an infrastructure to allow processes to run on remote hosts and a shared file system. Our system builds on Remote Method Invocation RMI[9], which allows a JVM running on one host to invoke methods on another host and the Jini[7] set of classes, which provide a client server architecture in which the client can locate and utilise services by name reference.

3.1 Communications

In order to understand the design decisions made here we must first give a more detailed account of the the communications process used between a typical Jini Service and a client. Initially the only knowledge a client has is that it requires

a service matching an interface which it possesses. This is used to query the lookup server and download a proxy class. The proxy then uses RMI to invoke methods on the service, so a Remote Interface must be declared containing the signature of each method which is to be invoked over the network.
For example:-

```
public interface Protocol extends Remote {
  public String getName() throws RemoteException;
  public void setName(String s) throws RemoteException;
}
```

The proxy class which is downloaded by the client contains a reference to this interface:-

```
public class Proxy
     implements Serializable, PublishedInterface {
  Protocol comms;
  int      age;
  public Proxy(Protocol p) {
    this.comms = p;
    this.age = 8;
  }
  public String getName() {
    return comms.getName();
  }
  public void setDetails(int i, String s) {
    this.age = i;
    comms.setName(s);
  }
}
```

Finally the service will either implement or contain a sub-class which implements this interface:-

```
public class Backend extends UnicastRemoteObject
                 implements Protocol {
  String name = "Ben";
  Backend() throws RemoteException {}
  public String getName() throws RemoteException {
    return name;
  }
  public void setName(String s) throws RemoteException {
    name = s;
  }
}
```

The service is responsible for creating and publishing the proxy object. Notice that the proxy requires an object implementing the Protocol interface in its constructor - this is the Backend object which receives remote invocations and its proxy which is used to communicate with the backend:-

```
Backend backend = new Backend();
```

```
Proxy proxy = new Proxy(backend);
```

Having created the proxy it must be registered and uploaded to at least one lookup server before a client can download and execute it. The standard serialization process recursively converts each field of an object to a byte stream which can then be piped across a network or saved to disk.

The backend object extends UnicastRemoteObject, which implements the methods necessary to customise Serialization, which in short replaces the reference to the backend with a "stub" object. Stubs are complete Java objects, dynamically generated at compile time they implement the same interface as the service (in this example the "Protocol" interface) and relay invocations complete with parameters over the network to the service which created them. Since both stub and service implement the same interface, from the client's perspective there is no difference between the two. The overall resultant behaviour however, is that all invocations made by a client are realised on the same backend service object.

There is one final complication to this procedure which is the means by which the client obtains the class definition of the stub itself. This is required in order for the JVM to be able to instantiate a stub object once it has fully streamed from the server since one of the main design goals of Jini is that a client only needs an interface in order to fully communicate with a service. RMI solves this problem by providing a custom ClassLoader which retrieves class definitions from an HTTP server. This ClassLoader is then used automatically during the de-serialization process of the stub. The address of the server must be provided at run time, typically specified as a command line option to the client JVM.

The resulting framework preserves the strong typing provided by Java while simultaneously blurring the remote nature between client and service, however it does not provide a good base for a distributed system such as J-DOS in which clients are able to invoke any method of an arbitrary objects on a remote node and as such the interface used for communication must be as generic as possible. In fact it is possible to reduce it to a single method, namely:-

```
public Object invoke(String methodName, Object[] args)
                                    throws RemoteException;
```

As with traditional RMI this invokes the method named "invoke" on the Backend, however by using reflection this single method can be used to invoke any other method on the service.

```
public Object invoke(String methodName, Object[] args)
                                   throws RemoteException {
  Class[] cTypes = new Class[args.length];
  for(int i=0; i < args.length; i++) {
    cTypes[i] = args[i].getClass();
  }
  Method m=oService.getClass().getMethod(methodName,cTypes);
  Object foo = m.invoke(oService, args);
  return foo;
}
```

In the above code “oService” is a reference to the service which owns the object and performs no processing itself other than to invoke the specified method on its parent service object.
The following code is an example of how the proxy can utilise this universal communications protocol:

```
public Collection getMatching(Class c)
                                  throws OFSException {
  try {
    Object[] args = {cwd, c};
    Object o = remoteComms.invoke("getMatching", args);
    return (Collection) o;
  }
  catch (RemoteException ex) {
    rethrow(ex);
  }
  return null;
}
```

The previous code was taken from the File System proxy and is used to retrieve all leaves in the current node which are an instance of the specified Class. The arguments passed to the communications backend are “cwd”, the path of the current node, and “c”, the class to match against. his call will invoke the “getMatching” method and the Collection of matching leaves is passed back to the client application.
The advantage of this implementation is that additional methods can be added to the service without re-compiling or even restarting existing clients. If existing method signatures are changed then corresponding changes are required only in the proxy. Additionally, the RemoteCommsBackend class can be re-used for every service. Its constructor takes a single argument which is the Service it belongs to, and therefore the object upon which it will invoke methods.
If an exception should be thrown by code executing on the server it is caught and passed back to the client encapsulated in a RemoteException. It is then re-thrown, hence providing the client with meaningful Exceptions from the server.

3.2 Remote Execution

Our design of a remote execution system has three essential considerations:-

1. Remote methods executing of service nodes must not block. This is desirable from a processing perspective, to extract maximal gain in performance.
2. Methods often have side-effects, that is they may alter the internal state of the object in some way e.g. by setting an instance variable, producing output to the screen etc. Our design provides means for the client to retrieve the object after the method has been invoked and consequently these side effects are preserved.
3. The client must be able to upload the class definitions to the server so they can be accessed locally by the JVM.

In certain situations it may be advantageous to have side effects occurring on a remote machine, particularly with regards to resources available on that machine. For example, a method may read in or save to a file which is on the remote machine.
The implementation of our design is thread based, extending the concept of Runnable objects:-

```
public void remoteRun(RemotelyRunnable rThread)
```

RemotelyRunnable is an interface which extends both Serializeable and Runnable. When the client invokes this method the rThread object is serialized and sent to the service as with a remoteInvocation, however the method returns void. This is because the remote Service spawns a new thread which in turn executes the run() method of the RemotelyRunnable rThread object. Therefore the method will return before the rThread has completed execution, thus allowing the client to farm out many requests in a truly distributed fashion.
In a direct contrast to the remoteInvocation method, all results which are to be passed back to the client by some means must now be stored within the the rThread object itself. Once the run() method has completed the object must then be made available to the client by some means.
To do this we provide an RMI callback method within the client which is called by the service whenever a thread completes, the method itself could be integrated into the RemotelyRunnable class allowing the client to specify the action to be performed.
This approach is both the fast and efficient since the service sends the result directly to the client as soon as the thread had completed. The additional complexity required in the client is the result of it becoming an RMI server object, although once again much of this complexity will be common across all clients and so consequently can be encompassed into helper classes.
For a distributed system to be of practical use it should be possible for a user to run programs which they have compiled or otherwise obtained themselves, without providing each node in the system which is to run the program with the necessary class definitions. RMI addresses this issue by providing a mechanism to download these class definitions from an HTTP server, however this requires the downloading party to know the address of the HTTP server in advance. If the class definition is being held by the client (of which there may be many, all with different class definitions) then the server has no way of knowing where to look. Ideally the service should download the class definition from the client, which we provide by manually streaming the object which the client wishes to be run and instantiating it using a special ClassLoader which can load definitions from the client. In order to do this it is obvious that a form of 2 way communication is required, consequently the client must become an RMI server object.
With our approach the client sends the server an RMI stub, instead of the object to be run. The stub has several methods which the server must call in order to set up a connection over the network, through which the RemotelyRunnable object can be streamed. the RMI interface is as follows:

```
public interface ClientComms extends Remote {
```

```
  public InetAddress getAddress(Integer pID)
                                  throws RemoteException;
  public int getPort(Integer pID) throws RemoteException;
  public void receiveResult(RemotelyRunnable result)
                                  throws RemoteException;
  public ClassFile getClassDefn(String s)
                                  throws RemoteException;
}
```

On receiving the stub, the service immediately makes a network connection to the host and port returned by the first 2 methods, of the above interface. The client is waiting for a connection on this address.
Next the service instantiates a new instance of RemoteClassLoader. This is a customised ClassLoader which uses the getClassDefn method in the client stub to resolve class definitions. The client is only queried if the definition can not be found locally, hence it is not possible for the client to override local definitions and performance is much greater than if every definition was taken from the client. The method returns a ClassFile object, a simple wrapper for an array of bytes which constitute the class definition itself.
Following this a custom ObjectInputStream is created. This is a very simple extension of the ObjectInputStream provided by the Java API:-

```
class JDOSObjectInputStream extends ObjectInputStream {
  private ClassLoader loader;

  public JDOSObjectInputStream(InputStream in,ClassLoader loader)
                        throws IOException {
    super(in);
    this.loader = loader;
  }

  protected Class resolveClass(ObjectStreamClass v)
                        throws ClassNotFoundException {
    return Class.forName(v.getName(),true,this.loader);
  }
}
```

This class overrides the resolveClass method (used to load class definitions) forcing the use of a custom ClassLoader, in this case the RemoteClassLoader. The method is called when a complete object has been received from the client and is about to be instantiated, the RemoteClassLoader returns the Class necessary for the received object to be instantiated, having uploaded the definition from the client if necessary. This allows the remote host to run any program that the client can provide class definitions for, without restarting the service or manually uploading files to another host, hence greatly increasing the transparency of the system.
The Jini classes, contained in JavaSpaces, are used as a means of distributing the stubs: the client writes an object containing the stub and a unique process

ID to a JavaSpace which then notifies all interested parties (in this case the execution services) that a new object has been written. Each service can then choose whether or not to take the object. This process is done on a first-come first-served basis, for example if 5 objects are written to the JavaSpace and there are 10 executing services then only the first 5 services will successfully retrieve a matching object from the JavaSpace. The process ID contained in this object is used when communicating with the client over RMI (see the RMI protocol interface above), as all the services communicate with the same RMI backend on the client. This approach could be seen as limiting the scalability of the system, however the RMI stubs are very small so large amounts of processing would be needed before a bottleneck was realised. If this scenario were to arise it could be easily dealt with by having multiple JavaSpaces, services would receive notifications from all of them whereas clients would upload stubs to a single JavaSpace selected at random.
This approach makes the client's role in distributed processing much simpler. It need only to write the stubs to a JavaSpace and then wait for the results to be returned by means of another RMI call to the client. This call is relayed back to the client application, which should implement the RRCallback interface, this specifies a single method:-

```
public void receiveResult(RemotelyRunnable rr)
```

All the behaviour described above has been incorporated into a series of "helper" classes, the most important of which is RemoteThreadRunner, as it automates the entire process I have described. The client need only to instantiate this class and provide the above callBack method to be able to farm out and receive threads. New RemotelyRunnable objects can be distributed by simply invoking the remoteRun method of the RemoteThreadRunner object.

3.3 Remote File System

Another requirement of a distributed environment is some form of shared file system. While JavaSpaces partially meets this need, it does not provide any kind of structure and introduces unnecessary complications such as requiring objects be renewed with leases. As such, it was decided that an additional service be added to the system providing an interface which mimics that found in traditional file systems.
The suggested structure is a tree comprised of nodes which contains leaves and other nodes, with a single root node, named "/" by default. Objects are encapsulated into leaves which then may be placed into any node. We have made extensive use of Interfaces and abstract classes to allow different underlying storage mechanisms, with a RAM based backend for testing but potential for a persistent implementation.
At the lowest level of the file system is a series of interfaces which define the basic properties of leaves and nodes. Note that at this level serialization is not enforced, hence it is possible to create branches containing more fluid entries similar to /dev or /proc in Linux. For example a node could contain a list of currently

executing threads, or a list of currently known processing nodes. Building on top of this layer is the series of RemoteFileSystem interfaces, these are identical except that Serialization is enforced.

As the file system is entirely Java-based it seemed logical to support the reading and writing of objects directly, note however that if the object is written to a remote node it must still be Serializeable, so this does not offer any particular advantage except for convenience.

Like any other Jini service the Remote File System may be accessed by multiple clients at any one time. Care must taken concerning the serialization of objects. If a client is passed the root node, the entire tree will be serialized including all sub-nodes and leaves, and any changes the client makes to the tree will not be reflected at the server as the client now has a local copy.

A second issue is that each client must be responsible for remembering its current location in the file system, if this is not so when one client changes node this will be reflected in all other clients. This is clearly an undesirable behaviour.

In order to address these issues the concept of a "window" is used, this is an object used by the client which holds state (current directory) and is used to communicate with the server. Many of the methods, particularly those associated with navigation, take and return String objects, thus avoiding the serialization problem. Whenever the window communicates with the server it also passes a String denoting the directory in which the action should be performed, this addresses the state issue while maintaining a clean interface to the client.

```
public abstract class ObjectFSWindow extends ObjectFSNode
                                    implements Serializable {
  abstract public void moveTo(String s) throws OFSException;
  abstract public void openNode(String s) throws OFSException;
  abstract public void openParent() throws OFSException;
  ....
}

public abstract class ObjectFSNode
                                    implements ObjectFSMember {
  abstract public String[] peekNodes() throws OFSException;
  abstract public String[] peekLeaves() throws OFSException;
  abstract public void removeLeaf(String s) throws OFSException;
  abstract public void removeNode(String s) throws OFSException;
  abstract public String getName() throws OFSException;
  abstract public void setName(String s) throws OFSException;
  ....
}
```

In addition any implementation of this service must take care that operations are thread safe. In particular, care must be taken to handle the scenario which arises when when one client performs an action which affects a node currently in use by another client.

4 Performance Results

The algorithm chosen to demonstrate the distributed processing ability of the system was the generation of Mandelbrot fractals. The set is generated by the iteration formula:-

$$Z_n = Z_{n-1}^2$$

where $Z_n = X_n + iY_n$ is a complex number, with $X_n$and Y_n real. For each point on the X, Y plane Z is iterated until it converges and that point of the plane is coloured in proportion to the number of iterations that it takes to converge. If Z fails to converge in 256 iterations then the point is black. The region covered was the plane $-1 < X < 1$ and $-1 < Y < 1$. This was partitioned into square sub-regions and the colour map for each sub-region was computed independently on different processors.

The platform used to test J-DOS was a Beowulf[5] cluster running MOSIX[1] consisting of 30 dual processor, 1GHz Pentium3 machines each with 1Gbyte of ram and linked with 100Mbit Ethernet. MOSIX itself is capable of migrating processes so care taken to ensure that it didn't do so during our experiments.

Table 1 shows that the problem does indeed benefit from distribution albeit in a way that isn't in proportion with the number of processors. Part of the problem is the bus architecture of the cluster and part is the overhead in distributing threads over several JVMs. The second contribution is clearly displayed in table 2. Where the problem only involves a single thread and we use more than one processor that single thread is not run on the host machine and the extra second that it takes to run can be attributed to the time it takes to transfer the 15Mbytes of data back to the host node. The benefits of parallelisation show up somewhat better with higher resolution images but the bottleneck associated with collecting data at a common node remains.

Threads	1	4	16	25
Time (secs)	8.4	3.4	3.9	3.9

Table 1. Times to render a 2000×2000 image when the number of processors matches the number of threads

5 Conclusions and Future Developments

The system is currently being developed by adding a remote GUI interface so that graphical applications could be displayed on one node and the J-DOS interface on another. Inter-thread communication is also being developed so that programmers are not restricted to the client server model. This would allow parallel programs with any logical communication patterns to be run under the

	Nodes			
Threads	1	4	16	25
1	6.8	8.3	8.3	8.3
4	3.8	3.3	3.5	3.4
16	4.3	3.9	2.8	2.4
25	4.2	31.	2.6	2.7

Table 2. Time to render a 2000×2000 image when the number of threads doesn't match the number of processors

system. The file system, as so far developed, is not distributed and would benefit from being so by allowing more overall storage capacity and allowing data searches to be distributed over all available nodes. Redundancy could also be built into such a file system making it more robust.
Overall J-DOS provides a dynamic extensible distributed computing environment in Java, with well defined interfaces by which programmers can write and distribute there own programs, without detailed knowledge of the cluster and using either client server or shared memory communication or both.

References

1. A.Barak and O.La'adan. The MOSIX multicomputer operating syatem for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13:361–372, 1998.
2. C.Catlett and L.Smarr. Metacomputing. *Communications of the ACM*, 35:44–52, 1992.
3. D.Young. *X Window system programming and applications with XT.* Prentice Hall, 1990.
4. M.Philippesen and M.Zenger. JavaParty-transparent remote objects in java. *Concurrency:Practice and Experience*, 9:1225–1242, 1997.
5. T.Sterling nad G.Bell and J.Kowalik. *Beowulf Cluster Computing with Linux.* MIT Press, 2001.
6. R.Sandberg. The SUN network file system: Design, implementation and experience. Technical report, SUN Microsystems, Inc., 1985.
7. S.Oaks and H.Wong. *Jini in a nutshell.* O'reilly, 2000.
8. W.Gropp, M.Snir, B.Nitzberg, and E.Lusk. *MPI: The Complete Reference.* MIT Press, 1998.
9. W.Grosso. *Java RMI.* O'reilly, 2002.
10. Y.Aridor, M. Factor, and A.Teperman. cJVM: A single image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing, Aizu-Wakamatsu, Japan*, pages 4–11, 1999.
11. Y.Aridor, M. Factor, and A.Teperman. Implementing java on clusters. *Lecture Notes on Computer Science*, 2150:722–731, 2001.