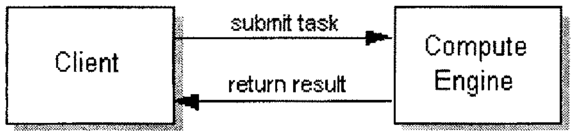
12 Appendix

12.1 Program Listing

At the heart of the compute engine is a protocol that allows jobs to be submitted to the compute engine, the compute engine to run those jobs, and the results of the job to be returned to the client. This protocol is expressed in interfaces supported by the compute engine and by the objects that are submitted to the compute engine, as shown in the following figure.



Here is the remote interface with its single method:

```
import java.rmi.*;
public interface AppServerIntf extends Remote
{
        double executeTask(double d1, double d2) throws RemoteException;
}
```

By extending the interface java.rmi.Remote, this interface marks itself as one whose methods can be called from any virtual machine. Any object that implements this interface becomes a remote object.

As a member of a remote interface, the executeTask method is a remote method. Therefore the method must be defined as being capable of throwing a java.rmi.RemoteException. This exception is thrown by the RMI system during a remote method call to indicate that either a communication failure or a protocol error has occurred. A RemoteException is a checked exception, so any code making a call to a remote method needs to handle this exception by either catching it or declaring it in its throws clause.

A Compute object can run different kinds of tasks as long as they are implementations of the Task type. The classes that implement this interface can contain any data needed for the computation of the task and any other methods needed for the computation.
Here is how RMI makes this simple compute engine possible. Since RMI can assume that the Task objects are written in the Java programming language, implementations of the Task object that were previously unknown to the compute engine are downloaded by RMI into the compute engine's virtual machine as needed. This allows clients of the compute engine to define new kinds of tasks to be run on the server machine without needing the code to be explicitly installed on that machine. In addition, because the executeTask method returns a java.lang.Object, any type of object can be passed as a return value in the remote call.
The compute engine, implemented by the AppServerImpl class, implements the AppServerIntf interface, allowing different tasks to be submitted to it by calls to its executeTask method. These tasks are run using the task's implementation of the execute method. The compute engine reports results to the caller through its return value.
Implementing a Remote Interface
Let's turn now to the task of implementing a class for the compute engine. In general the implementation class of a remote interface should at least

- Declare the remote interfaces being implemented
- Define the constructor for the remote object
- Provide an implementation for each remote method in the remote interfaces

The server needs to create and to install the remote objects. This setup procedure can be encapsulated in a main method in the remote object implementation class itself, or it can be included in another class entirely. The setup procedure should

- Create and install a security manager
- Create one or more instances of a remote object

Register at least one of the remote objects with the RMI remote object registry (or another naming service such as one that uses JNDI), for bootstrapping purposes
The complete implementation of the compute engine follows. The class implements the remote interface Compute and also includes the main method for setting up the compute engine.

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class AppServerImpl extends UnicastRemoteObject implements AppServerIntf
{
        private ArrayList emailObjArray;
        public AppServerImpl() throws RemoteException
        {
                emailObjArray = new ArrayList();
        }
        public double executeTask(double d1, double d2) throws RemoteException
        {
                new editorialBasedEmailNotification();
                return d1+d2;
        }
}
```

Now let's take a closer look at each of the components of the compute engine implementation.

Declare the Remote Interfaces Being Implemented The implementation class for the compute engine is declared as public class AppServerImpl extends UnicastRemoteObject implements AppServerIntf. This declaration states that the class implements the Compute remote interface (and therefore defines a remote object) and extends the class java.rmi.server.UnicastRemoteObject.

UnicastRemoteObject is a convenience class, defined in the RMI public API, which can be used as a superclass for remote object implementations. The superclass UnicastRemoteObject supplies implementations for a number of java.lang.Object methods (equals, hashCode, toString) so that they are defined appropriately for remote objects. UnicastRemoteObjectalso includes constructors and static methods used to *export* a remote object, that is, make the remote object available to receive incoming calls from clients.

A remote object implementation does not have to extend UnicastRemoteObject, but any implementation that does not must supply appropriate implementations of the java.lang.Object methods. Furthermore, a remote object implementation must make an explicit call to one of UnicastRemoteObject's exportObject methods to make the RMI runtime aware of the remote object so that the object can accept incoming calls. By extending UnicastRemoteObject, the ComputeEngine class can be used to create a simple remote object that supports unicast (point-to-point) remote communication and that uses RMI's default sockets-based transport for communication.

If you choose to extend a remote object from any class other than Unicast-RemoteObject or, alternatively, extend from the new JDK 1.2 class java.rmi.activation.Activatable (used to construct remote objects that can execute on demand), you need to export the remote object by calling either the UnicastRemoteObject.exportObject or Activatable.exportObject method explicitly from your class's constructor (or another initialization method, as appropriate).

The AppServer engine example defines a remote object class that implements only a single remote interface and no other interfaces. The AppServerImpl class also contains some methods that can be called only locally. The first of these is a constructor for AppServerImpl objects; the second is a main method that is used to create a ComputeEngine and make it available to clients.

Define the Constructor

The AppServerImpl class has a single constructor that takes no arguments. The code for the constructor is

```
public AppServerImpl() throws RemoteException
{
        emailObjArray = new ArrayList();
}
```

This constructor simply initializes an ArrayList. The superclass constructor gets called even if omitted from the AppServerImpl constructor, we include it for clarity.

During construction, a UnicastRemoteObject is ***exported***, meaning that it is available to accept incoming requests by listening for incoming calls from clients on an anonymous port.

Note: *In JDK 1.2 you may indicate the specific port that a remote object uses to accept requests.*

The no-argument constructor for the superclass, UnicastRemoteObject, declares the exception RemoteException in its throws clause, so the Compute-Engine constructor must also declare that it can throw RemoteException. A RemoteException can occur during construction if the attempt to export the object fails--due to, for example, communication resources being unavailable or the appropriate stub class not being found.
Provide Implementations for Each Remote Method
The class for a remote object provides implementations for each of the remote methods specified in the remote interfaces. The Compute interface contains a single remote method, executeTask, which is implemented as follows:

```
public double executeTask(double d1, double d2) throws RemoteException
{
        new editorialBasedEmailNotification();
        return d1+d2;
}
```

This method implements the protocol between the AppServer and its clients. Clients provide the executeTask with some values, which has an implementation of the task's execute method. The AppServerImpl executes the Task and returns the result of the task's execute method directly to the caller.
The executeTask method does not need to know anything more about the result of the execute method than that it is at least an Object. The caller presumably knows more about the precise type of the Object returned and can cast the result to the appropriate type.

12.1.1 Passing Objects in RMI

Arguments to or return values from remote methods can be of almost any type, including local objects, remote objects, and primitive types. More precisely, any entity of any type can be passed to or from a remote method as long as the entity is an instance of a type that is a primitive data type, a remote object, or a ***serializable*** object, which means that it implements the interface java.io.Serializable.

A few object types do not meet any of these criteria and thus cannot be passed to or returned from a remote method. Most of these objects, such as a file descriptor, encapsulate information that makes sense only within a single address space. Many of the core classes, including those in the packages java.lang and java.util, implement the

12.1.2 Serializable interface

The rules governing how arguments and return values are passed are as follows.

Remote objects are essentially passed by reference. A ***remote object reference*** is a stub, which is a client-side proxy that implements the complete set of remote interfaces that the remote object implements.

Local objects are passed by copy, using object serialization. By default all fields are copied, except those that are marked static or transient. Default serialization behavior can be overridden on a class-by-class basis.

Passing an object by reference (as is done with remote objects) means that any changes made to the state of the object by remote method calls are reflected in the original remote object. When passing a remote object, only those interfaces that are remote interfaces are available to the receiver; any methods defined in the implementation class or defined in nonremote interfaces implemented by the class are not available to that receiver.

For example, if you were to pass a reference to an instance of the AppServerImpl class, the receiver would have access only to the AppServerImpl engine's executeTask method. That receiver would not see either the AppServerImpl constructor or its main method or any of the methods in java.lang.Object.

In remote method calls objects--parameters, return values, and exceptions--that are not remote objects are passed by value. This means that a copy of the object is created in the receiving virtual machine. Any changes to this object's state at the receiver are reflected only in the receiver's copy, not in the original instance.

Implement the Server's main Method

The most involved method of the ComputeEngine implementation is the main method. The main method is used to start the ComputeEngine and therefore needs to do the necessary initialization and housekeeping to prepare the server for accepting calls from clients. This method is not a remote method, which means that it cannot be called from a different virtual machine. Since the main method is declared static, the method is not associated with an object at all but rather with the class ComputeEngine.

Create and Install a Security Manager

The first thing that the main method does is to create and to install a security manager, which protects access to system resources from untrusted downloaded code running within the virtual machine. The security manager determines whether downloaded code has access to the local file system or can perform any other privileged operations.

All programs using RMI must install a security manager, or RMI will not download classes (other than from the local class path) for objects received as parameters, return values, or exceptions in remote method calls. This restriction ensures that the operations performed by downloaded code go through a set of security checks.

The ComputeEngine uses a security manager supplied as part of the RMI system, the RMISecurityManager. This security manager enforces a similar security policy as the typical security manager for applets; that is to say, it is very conservative as to what access it allows. An RMI application could define and use another SecurityManager class that gave more liberal access to system resources or, in JDK 1.2, use a policy file that grants more permissions.

Here's the code that creates and installs the security manager:

```
if (System.getSecurityManager() == null) {
   System.setSecurityManager(new RMISecurityManager());
}
```

Make the Remote Object Available to Clients

Next, the main method creates an instance of the ComputeEngine. This is done with the statement

Compute engine = new ComputeEngine();

As mentioned, this constructor calls the UnicastRemoteObject superclass constructor, which in turn exports the newly created object to the RMI runtime. Once the export step is complete, the ComputeEngine remote object is ready to accept incoming calls from clients on an anonymous port, one chosen by RMI or the underlying operating system. Note that the type of the variable engine is Compute, not ComputeEngine. This declaration emphasizes that the interface available to clients is the Compute interface and its methods, not the Compute-Engine class and its methods.

Before a caller can invoke a method on a remote object, that caller must first obtain a reference to the remote object. This can be done in the same way that any other object reference is obtained in a program, such as getting it as part of the return value of a method or as part of a data structure that contains such a reference.

The system provides a particular remote object, the RMI registry, for finding references to remote objects. The RMI registry is a simple remote object name service that allows remote clients to get a reference to a remote object by name. The registry is typically used only to locate the first remote object an RMI client needs to use. That first remote object then provides support for finding other objects.

The java.rmi.Naming interface is used as a front-end API for binding, or registering, and looking up remote objects in the registry. Once a remote object is registered with the RMI registry on the local host, callers on any host can look up the remote object by name, obtain its reference, and then invoke remote methods on the object. All servers running on a host may share the registry, or an individual server process may create and use its own registry, if desired.

The AppServer Engine class creates a name for the object. This name includes the host name, host, on which the registry (and remote object) is being run and a name, Compute that identifies the remote object in the registry. The code then needs to add the name to the RMI registry running on the server. This is done later (within the try block) with the statement

Naming.rebind("AddServer", addServerImpl);

Calling the rebind method makes a remote call to the RMI registry on the local host. This call can result in a RemoteException being generated, so the exception needs to be handled. The AppServer class handles the exception within the try/catch block. If the exception is not handled in this way, RemoteException would have to be added to the throws clause (currently nonexistent) of the main method.

Note the following about the arguments to the call to Naming.rebind.

The first parameter is a URL-formatted java.lang.String representing the location and the name of the remote object. You will need to change the value of host to be the name, or IP address, of your server machine. If the host is omitted from the URL, the host defaults to the local host. Also, you don't need to specify a protocol in the URL. For example, supplying Compute as the name in the Naming.rebind call is allowed. Optionally a port number may be supplied in the URL; for example, the name //host:1234/objectname is legal. If the port is omitted, it defaults to 1099. You must specify the port number only if a server creates a registry on a port other than the default 1099. The default port is useful in that it provides a well-known place to look for the remote objects that offer services on a particular host.

The RMI runtime substitutes a reference to the stub for the remote object reference specified by the argument. Remote implementation objects, such as instances of ComputeEngine, never leave the VM where they are created, so when a client performs a lookup in a server's remote object registry, a reference to the stub is returned. As discussed earlier, remote objects in such cases are passed by reference rather than by value.

Note that for security reasons, an application can bind, unbind, or rebind remote object references only with a registry running on the same host. This restriction prevents a remote client from removing or overwriting any of the entries in a server's registry. A lookup, however, can be requested from any host, local or remote.

Once the server has registered with the local RMI registry, it prints out a message indicating that it's ready to start handling calls and then the main method exits. It is not necessary to have a thread wait to keep the server alive. As long as there is a reference to the ComputeEngine object in another virtual machine, local or remote, the AppServerImpl object will not be shut down, or garbage collected. Because the program binds a reference to the AppServerImpl in the registry, it is reachable from a remote client, the registry itself! The RMI system takes care of keeping the Engine's process up.

The Engine is available to accept calls and won't be reclaimed until its binding is removed from the registry, ***and*** no remote clients hold a remote reference to the ComputeEngine object.

The only exception that could be thrown in the code is a RemoteException, thrown either by the constructor of the class or by the call to the RMI registry to bind the object to the name Compute. In either case the program can't do much more than exit after printing an error message. In some distributed applications it is possible to recover from the failure to make a remote call. For example, the application could choose another server and continue operation.

Creating a Client Program

The appserver engine is a pretty simple program: it runs tasks that are handed to it. A client needs to call the methods provided by the interface on the server, but it also has to define the task to be performed by the server. The client is a simple GUI application that invokes methods on the remote objects registered with the application server.

```
/*
*A simple graphical user interface that connects to the Appserver and invokes methods
*defined in the remote interface.
*/
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;

public class AppFrame extends Frame {
TextField a, b ,result;
Button btn;
protected static boolean DEBUG = true;
public AppFrame() {
super("AppClient");
setSize(400, 200);
setLayout(new FlowLayout());
add(new Label("Send Request to Server..."));
```

```
a = new TextField("0", 4);
b = new TextField("0", 4);
result = new TextField("0", 4);
btn = new Button("SEND");
add(a); add(b);add(result); add(btn);
addMouseListener(new MouseAdapter() {
public void mousePressed(MouseEvent e) {
a.setText(String.valueOf(e.getX()));
b.setText(String.valueOf(e.getY()));
}
});
addWindowListener(new WindowAdapter() {
public void windowClosing(WindowEvent e) {
setVisible(false);
dispose();
System.exit(0);
}
});
btn.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e)
{
try
{
String appServerURL="rmi://localhost/AddServer";
                    AppServerIntf
                    appServerIntf=(AppServerIntf)Naming.lookup(appServerURL);
result.setText(""+appServerIntf.executeTask(10, 11));

}
catch (java.rmi.NotBoundException exc) {
if (DEBUG) {
```

```
System.err.println("Couldn't find Server running on host ");
System.err.println("RMI seems to be running fine.");
}
} catch (java.rmi.UnmarshalException exc) {
System.err.println("Unmarshal exception on host "
+ ".");
System.err.println("Try recompiling everything and starting over?");
} catch (java.rmi.ConnectException exc) {
if (DEBUG) {
System.err.println("RMI isn't running on "
+ ".");
System.err.println("Or maybe it is, but it "
+ "started after this program.");
}
} catch (java.rmi.UnknownHostException exc) {
System.err.println("Unknown host");
} catch (java.rmi.ConnectIOException exc) {
System.err.println("Couldn't get to host "
+ ".");
} catch (Exception exc) {
System.err.println("Unexpected exception ");
System.err.println("Exception type: "
+ exc.toString());
exc.printStackTrace();
}
}
});
}
public static void main(String[] args) {
AppFrame app = new AppFrame();
app.setVisible(true);
```

```
}
}
```

There is a multithreaded process which creates and few XML files and writes to the file system. This is a fake service provided by this application server for the purpose of demonstration.

```
//This pool is used to initiate worker threads and manage their life cycle.
//Also the pool creates a Update thread that deals with database tractions.
//
//Worker threads are in wait state and are deposited to an ArrayList "threads"
//after they are finished the work. When a new task comes in, the pool pulls
//a worker thread from the end of list to perform the work.The worker thread is
//waken up when being handed over the work.

//The pool monitors one conditions for the "threads" arraylist: No_empty. If
//"threads" list has no thread left and there is a new task coming in, the
//requestor of that task must wait until there is a worker thread retunring.

//Upon notification as terminateNow function being called, the pool notifies
//every worker thread in the list and the returning worker threads to stop.
//The maximum wait time for worker thread to stop is set by the variable
//called maxTimeToWait.

//After all the worker threads are stopped or wait time is out, the pool notifies
// the update thread to stop too.

import java.util.*;
public class WorkerThreadPool
{
  ArrayList threads= new ArrayList();
  DeliveryThread d;
```

```
MailPool mailPool;
int threadCount;

private int successfulHtmlMails=0;
private int successfulTextMails=0;
private int unSuccessfulMails=0;

//wait maximum 100 seconds for all the workerThread to terminate
private static final int maxTimeToWait=100;

//!Warning: Modifications in any of the following sections may result in improper
//execution of the application such as race condition, deadlock, trash
//buildup and/or other runtime unexpected errors. Please use care.

public WorkerThreadPool(MailPool mailPool, DeliveryThread d, int count)
{
  this.mailPool = mailPool;
  this.d=d;
  threadCount=count;
  for (int i=0; i<count; i++)
  {
    new WorkerThread(this, i);
  }
}

public WorkerThreadPool()
{}

public synchronized WorkerThread getWorker()
{
            if(threads.size()>0)
            {
```

```
                        return (WorkerThread)threads.remove(threads.size()-1);
                }
                else
                {
                        try
                        {
                                while(threads.size()==0)
                                        wait();
                        }
                        catch(InterruptedException ie)
                        {
                                ie.printStackTrace();
                        }

                        return (WorkerThread)threads.remove(threads.size()-1);
                }
        }

        public int getThread()
        {
                return threads.size();
            }

            public synchronized void deposit(WorkerThread worker)
        {
          //System.out.println("Depositing workerThread..");
          if(worker.getSubnId()!= -999 && worker.getLastDeliveryStatus())
          {
            if(worker.getMailFormat()==editorialBasedEmailNotification.HTML)
              successfulHtmlMails++;
            else if(worker.getMailFormat()==editorialBasedEmailNotification.TEXT)
              successfulTextMails++;
          }
          else if(worker.getSubnId()!= -999)
          {
                        unSuccessfulMails++;
                }
                if(worker.getContent() != null)
                mailPool.pushRecycle(worker.getContent());
                worker.setContent(null);
                threads.add(worker);
                if(threads.size()==1)
            notify();
         //System.out.println("Waking up deliveryThread..");
        }
```

```
    public void setErrorFlag()
    {
      d.setErrorFlag();
      terminateNow();
    }


    public void terminateNow()
    {
      int threadsTerminated=0;
      long then=System.currentTimeMillis();

      while((System.currentTimeMillis()-then)<maxTimeToWait)
      {
        synchronized(this)
        {
          while(threads.size()>0)
          {
            ((WorkerThread)threads.remove(threads.size()-1)).stop();
            ++threadsTerminated;
          }
        }
        if(threadsTerminated>=threadCount)
          break;
        else
        {
          try
          {
            Thread.sleep(50); //Wait before retrying.
          }
          catch(InterruptedException i)
          {
            i.printStackTrace();
            threads=null;
            break;
          }

        }

      }
              d.updateSuccessfulTextMails(successfulTextMails);
              d.updateSuccessfulHtmlMails(successfulHtmlMails);
              d.updateUnSuccessfulMails(unSuccessfulMails);
    }
  }
```

MailPool.java

```
/**
* This pool hold the SubnContentDetail object.The pool helps to recycle the scd objects.
* There are method like pushRecycle() and getRecycle() that are used to manage this
pool.
* Once a new scd object is created the push() method in this class can be used to register
* that object with this class.
**/


import java.util.*;

public class MailPool
{
  private volatile ArrayList scdArray;
        private volatile ArrayList scdRecycleArray;
  private int poolSize=4;
  private boolean stop=false;

  public MailPool(int poolSize)
  {
    if(poolSize>0 && poolSize<10)
      this.poolSize=poolSize;

    scdArray=new ArrayList(poolSize);
    scdRecycleArray = new ArrayList();
  }

  /*
        *This method Stores an scd object from the pool and removes it from the
pool.This method
        *is synchronized as there could be at any given time multiple number of threads
attempting
        *to modify the pool.
        *@return void
        */
  public synchronized void push(SubnContentDetail scd)
  {
    while(scdArray.size()==poolSize)
    {
      try
      {
        wait();
```

```
         }
         catch(InterruptedException ie)
         {
            ie.printStackTrace();
         }
      }

         scdArray.add(scd);

      if(scdArray.size()==1)
      {
         notify();
      }
   }

         /*
         *Once the worker thread has finished processing the task it returns the scd to the
pool
         *for reuse.
         *@param SubnContentDetail object.
         */

         public void pushRecycle(SubnContentDetail scd)
   {
         scdRecycleArray.add(scd);
   }

  /*
  *To get an scd object from the pool for reuse. Returns null if the pool is empty.
  *@return SubnContentDetail object.
  */
  public SubnContentDetail getRecycle()
         {
                  try
                  {
                           return
(SubnContentDetail)scdRecycleArray.remove(scdRecycleArray.size()-1);
                  }
                  catch(IndexOutOfBoundsException idx)
                  {
                           return null;
                  }
   }

  /*
  *Confirm is the pool has any scd available for processing.
```

```
 *@return boolean.
 */
  public synchronized boolean hasItems()
  {
     return scdArray.size()>0;
  }

        /*
        *Reads the size of the pool.
        *@return pool size.
        */
        public int getScd()
  {
     return scdArray.size();
  }

        /*
        *Reads the size of the recycle pool.
        *@return recycle pool size.
        */
        public int getscdRecycleArray()
        {
                return scdRecycleArray.size();
        }

  /*
        *Returns an scd object from the pool and removes it from the pool.This method is
        *synchronized as there could be at any given time multiple number of threads
attempting
        *to pick up the scd object.
        *@return recycle pool size.
        */
  public synchronized SubnContentDetail get()
  {
     SubnContentDetail scd=null;

     while(scdArray.size()==0)
     {
        if(stop)
           return null;
        try
        {
           wait();
           if(stop)
              return null;
        }
```

```
            catch(InterruptedException ie)
            {
                ie.printStackTrace();
            }
        }

        scd= (SubnContentDetail)scdArray.remove(scdArray.size()-1);

        if(scdArray.size()==poolSize-1)
        {
            notify();
        }

        return scd;
    }


    public synchronized void setStop()
    {
        stop=true;
        notify();
    }
}
```
EditorialBasedEmailNotification.java

```
import java.sql.*;
import java.util.*;
import java.text.SimpleDateFormat;
import javax.naming.*;
import javax.ejb.FinderException;
import java.rmi.*;
import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;
import java.lang.*;

public class editorialBasedEmailNotification
{
    private MailPool mailPool[];
    private static DeliveryThread deliveryThread[];
    public final static int TEXT= 1;
    public final static int HTML= 2;
    public long subnThreadCount=0;
    public int threadCount=0;
    public int mailPoolSize=0;

    private HashMap contentHM = new HashMap();
    private SimpleDateFormat formatter = new SimpleDateFormat ("EEEE, MMM d,
yyyy");
    private SimpleDateFormat SQLDateFormatter = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss");
    private static boolean errorFlag=false; //error from lower levels.
    private static ArrayList subnThreadArray=new ArrayList();

    public editorialBasedEmailNotification()
    {
      subnThreadCount = 2;
                threadCount = 2;
                mailPoolSize = 2;

      long maxSubnId=0;
                mailPool = new MailPool[mailPoolSize];
                deliveryThread = new DeliveryThread[mailPoolSize];
                for(int i=0; i <mailPool.length; i++)
                {
                        mailPool[i] = new MailPool(4);
                        deliveryThread[i]=new
DeliveryThread(this,mailPool[i],threadCount);
                }
      SubscriptionThread subnThread;
      long SubnIdIncrement=maxSubnId/subnThreadCount;
      for (int i=0; i<subnThreadCount-1; ++i)
      {
         subnThread=new
SubscriptionThread(i,i*SubnIdIncrement+1,(i+1)*SubnIdIncrement);
         subnThreadArray.add(subnThread);
         subnThread.start();
      }

      subnThread=new SubscriptionThread((int)subnThreadCount-1,(subnThreadCount-
1)*SubnIdIncrement+1,maxSubnId);
      subnThreadArray.add(subnThread);
      subnThread.start();
   }

   public void setErrorFlag()
   {
      errorFlag=true;
      System.out.println("A non-recoverable error has occured and the application " +
                     "has to stop now. The cause can be some supporting files missing.");
   }

   public static void main(String[] arg)
   {
     long then=System.currentTimeMillis();
     try
     {
        for (int i=0; i<subnThreadArray.size(); i++)
        {
           ((SubscriptionThread)subnThreadArray.get(i)).join();
        }

        if(!errorFlag)
        {
         for (int i=0; i<deliveryThread.length; i++)
         {
            deliveryThread[i].terminate();
         }
      }

      for (int i=0; i<deliveryThread.length; i++)
      {
               deliveryThread[i].join();
               }
   }
   catch(InterruptedException ie)
   {
      ie.printStackTrace();
   }
            System.out.println("\n\nTotal time spent was :" +
      (System.currentTimeMillis()-then)/(1000.0*60) + " minutes.");
}


class SubscriptionThread extends Thread

{

   long startIndex;

   long endIndex;

   int id;

   private SubnContentDetail scd;


   public SubscriptionThread(int id, long from, long to)

   {

      startIndex=from;

      endIndex=to;

      this.id=id;
               System.out.println("SubscriptionThread " + id + " was created. (from: "+
      from + " to: " +to+")");
   }

               public void run()
   {
               int count = 0;
      try
```

```
        {
          for (int i=0; i <= 100; i++)
          {
            if(errorFlag)
            break;
            scd = mailPool[++count%mailPool.length].getRecycle();
            if (scd == null)
                                        scd = new SubnContentDetail();
            scd.setSubnId(i);
            scd.setEmail("test@hotmail.com");
            int format_id=1;
            if (format_id == 1)
            {
              scd.setMailFormat(TEXT);
            }
            else if (format_id == 2)
            {
              scd.setMailFormat(HTML);
            }
            mailPool[count%mailPool.length].push(scd);
          }
                        System.out.println("SubscriptionThread done
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");
        }
        catch (Exception ex)
        {
         ex.printStackTrace();
        }
        }
        }
}
DeliveryThread.java
import java.util.*;

public class DeliveryThread implements Runnable
{
        private static int successfulHtmlMails=0;
        private static int successfulTextMails=0;
        private static int unSuccessfulMails=0;
   boolean beTerminated=false;
   WorkerThreadPool workerThreadPool;
   WorkerThread worker;
   SubnContentDetail scd;
   Thread t;
   MailPool mailPool;
   private boolean errorFlag=false;
```

```
  editorialBasedEmailNotification driver;
  private static int counterForEmail = 0;
  private static Object counterForNumEmailSentLock = new Object();

  public DeliveryThread(editorialBasedEmailNotification driver, MailPool mailPool, int
threadCount)
  {
    this.mailPool=mailPool;
    this.driver=driver;
            synchronized(counterForNumEmailSentLock)
            {
                  counterForEmail++;
            }
    if(threadCount<0)
      threadCount=2;
    else if(threadCount>=10)
      threadCount=5; //max 5 worker threads.

    workerThreadPool=new WorkerThreadPool(mailPool,this, threadCount);

    t=new Thread(this);
    t.start();
  }

  public void setErrorFlag()
  {
    driver.setErrorFlag();
    errorFlag=true;
    terminate();
  }

      public synchronized void updateSuccessfulTextMails(int successfulTextMails)
      {
            this.successfulTextMails += successfulTextMails;
      }

      public synchronized void updateSuccessfulHtmlMails(int successfulHtmlMails)
      {
            this.successfulHtmlMails += successfulHtmlMails;
      }

      public synchronized void updateUnSuccessfulMails(int unSuccessfulMails)
      {
            this.unSuccessfulMails += unSuccessfulMails;
      }
    public synchronized void terminate()
    {
      mailPool.setStop();
      long then=System.currentTimeMillis();
      while(mailPool.hasItems() && (System.currentTimeMillis()-then)<300)
      {
        try
        {
          Thread.sleep(1000);
        }
        catch(InterruptedException ie)
        {
          ie.printStackTrace();
        }
      }
      beTerminated=true;
      notify();
    }

    public void join() throws InterruptedException
    {
      t.join();
    }

    public void run()
    {
                    int count = 0;
      while(true)
      {
        if(beTerminated)
          break;
        scd=mailPool.get();
        if(scd!=null)
        {
          worker=workerThreadPool.getWorker();

          worker.setContent(scd);
                        //System.out.println("Waking up worker.");
          worker.resume();
                                  count++;
                                  if(count %20 == 0)
                                  {
                                          System.out.print("MailPool Recycle
:"+mailPool.getscdRecycleArray()+"Waiting MailPool :"+mailPool.getScd());
                                          System.out.print("ThreadPool
:"+workerThreadPool.getThread());
```

```
                                }
        }
      }
      if(!errorFlag)
        workerThreadPool.terminateNow();
      try
      {
        Thread.sleep(500);
                }
                catch(InterruptedException ex)
                {}
                synchronized(counterForNumEmailSentLock)
                {
                        counterForEmail--;
                }
      System.out.println("DeliveryThread is stopping..");
   }
}
```

Running the client and server

Start the rmiregistry

To start the registry, Windows users should do the following (assuming that your java\bin directory is in the current path):-

start rmiregistry

To start the registry, Unix users should do the following:-

rmiregistry &

Compile the server

Compile the server, and use the rmic tool to create stub files.

Start the server

From the directory in which the classes are located, type the command to run the server

Start the client

You can run the client locally, or from a different machine. In either case, you'll need to specify the hostname of the machine where you are running the server. If you're running it locally, use localhost as the hostname.

12.2 Typical Performance Characteristic

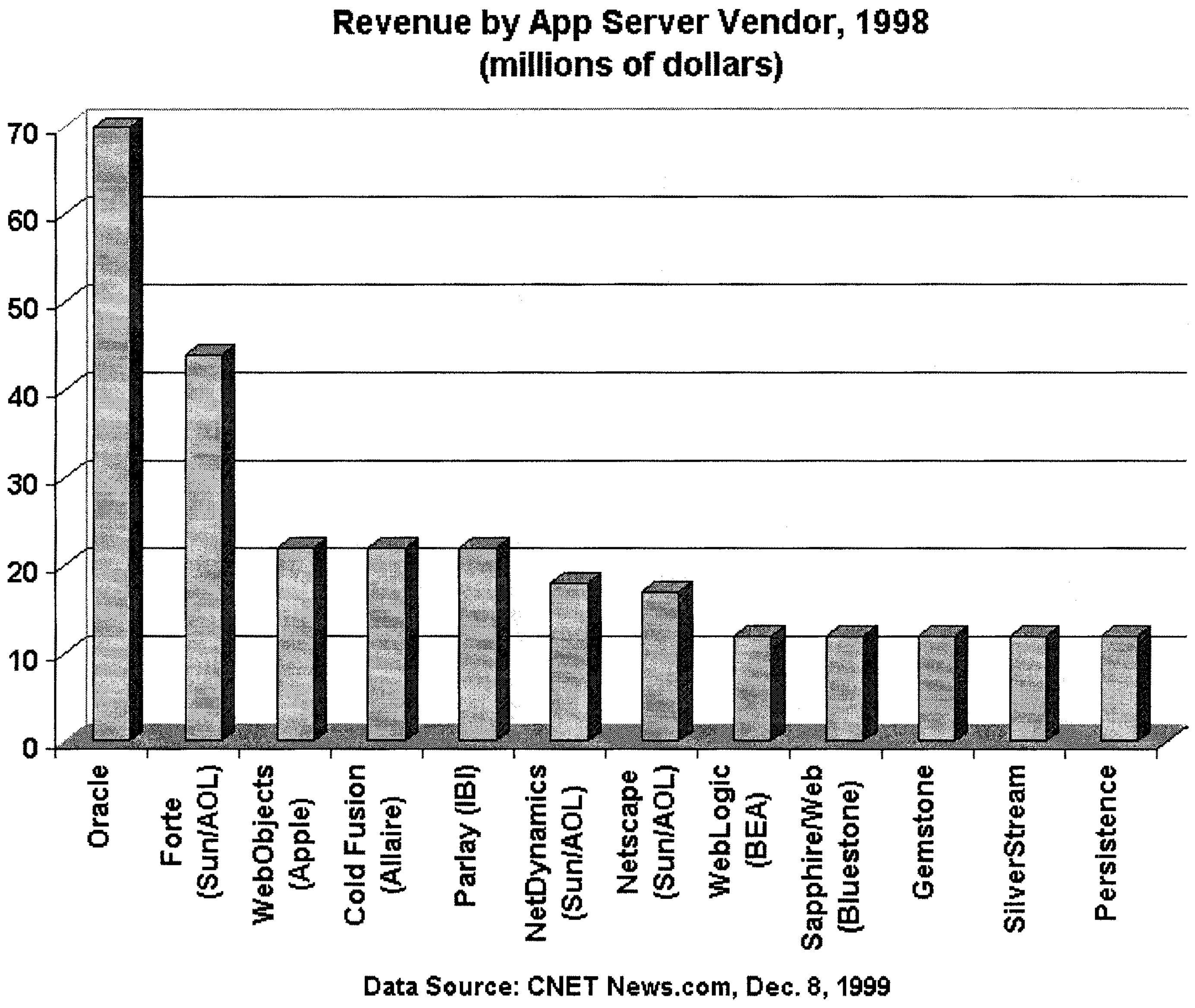


Figure 4 : Performance Characteristic

12.2.1 Emerging Market

The term "application server" has become one of the hottest buzzwords of the decade. It is being used to describe a wide variety of products that offer a broad spectrum of features and capabilities. Application servers have grown out of a number of different product groups, including Web servers, database servers, TP monitors, and CORBA runtime systems. The one feature that these products have in common is that they provide an optimized execution environment for server-side application components.

12.2.2 Product Trade-Offs

Each application server offers a different set of features and capabilities. No single application server is the right choice for all circumstances. The right choice is determined by the specific requirements of a particular installation, but some reasonable guidelines can help narrow the field. Certain application servers offer better scalability, but often with a comparable increase in complexity and license fees. Other applications servers offer simplicity and ease-of-use, but they may not support extended services, such as heterogeneous transactions or fault tolerance.

Company Name	**Product Name**	**Web Site**	**Interface Method**
Allaire	ColdFusion	http://www.allaire.com/products/coldfusion/40/	ActiveX, C++, Java
Apple	Web Objects	http://www.apple.com/webobjects/	ANSI C, C++, Java
Art Technology Group	Dynamo Personalization software	http://www.atg.com/products/highlights/highlights_main.html	Java
BEA Web Login	Tengah	http://www.weblogic.com/products/tengah/tengahabout.html	Java
Blue Stone	Sapphire	http://www.bluestone.com/products/sapphire/	C, C++, Java
Bullet Proof Corporation	JDesigner Pro	http://www.bulletproof.com/	Java

Elemental SoftWare	Drumbeat 2000	http://www.drumbeat.com/	ActiveX
GemStone Software	GemStone/J	http://www.gemstone.com/products/j/main.html	Java
HAHT	HAHT Site Application Server	http://www.haht.com/Go.html?Page=HS_Pr_HSOverview	ANSI C, C++, ActiveX
Halcyon Software	I-ASP	http://www.halcyonsoft.com/asp/whitepaper.html	Java
IBM	WEB SPHERE	http://www.software.ibm.com/webservers/appserv/	Java
Inprise	Inprise Application Server	http://www.inprise.com/appserver/	Java
Internova	Colibri Engine	http://www.internova.com/colibri/main.asp	Java, ActiveX
Intersolv	NetExpress	http://www.microfocus.com/products/enterapp.htm	C++
Lona Technologie s	Orbix OTM	http:// www.iona.com/products/transactions/orbixotm/index.html	Java
Lotus	Domino	http://www.lotus.com/home.nsf/tabs/domino	ActiveX
Micorsoft	MTS/IIS	http://www.microsoft.com	ActiveX
Netscape	Application Server	http://www.netscape.com/appserver/v2.1/index.html	Java
New Atlanta	Servlet Exec 2.0	http://www.newatlanta.com/products.html	Java
Novera	J Business	http://www.novera.com/jbusiness.html	Java
Open Connect System	WebConnect	http://www.openconnect.com/pressrel/120898.html	Java
Oracle	Oracle WAS	http://www.oracle.com/products/asd/oas/oas.html	Java

Pervasive Software	Tango	http://tango.pervasive.com/products/tango/webjump/	Java, ActiveX
Pramati technologies	Proton	http://www.pramati.com/products.htm	Java
Progress Softwares	Aptivity	http://www.progress.com/java/apptivity/apptivity.htm	Java
Prosyst	Enterprise Beans Server	http://www.prosyst.com/prosyst/champion.htm	Java
Seagate Software	Seagate Info APS	http://www.seagatesoftware.com/crystalinfo/	ANSI C, C++
Secant Technologies	Secant Extreme Server	http:// www.secant.com/secant/extreme_enterprise_server_ejb.htm	Java
SilverStream	Silver Stream	http://www.silverstream.com/information/press/v2press_f.htm	Java
Sun	NetDynamics	http://www.netdynamics.com/	Java
Sybase	Enterprise Application Server	http://www.sybase.com/products/application_servers/	ActiveX, ANSI C, C++, Java
Tempest	Tempest Messanger System	http://www.tempest.com/products.html	ANSI C, C++, Java
Unify	Vision App Server	http://www.unify.com/Products/vision.htm	ANSI C, ActiveX
Unify	Ewave Engine	http://www.unify.com/Products/ewave/index.htm	Java
Visient	Arabica EJB Server	http://www.visient.com/Arabica_server_main.htm	Java
Vision	Jade	http://www.vision-soft.com/products/products.htm	Java
Visisoft Inc.	Com Studio	http://www.visisoft.com/cando.htm	C++

Table 1 : List of Application Servers

12.3 Specification Summary

Load Balancing: - Ability to send the request to the to different servers depending upon the load and availability of the server.
Fault Tolerence Ability of the Application server with no single point of failure, defining policies for recovery and fail-over recovery in case of failure of one object or group of objects.
Transaction Management
Mulithreaded Architecture
Managability
Security like support for SSL, Firewall X.509 certificates,Access Control Lists (ACL)
Security Level i.e. ServerLevel, Service Level , Directory Level,Object Level etc
Development and Support Tools i.e. Any development Environment.
CORBA Support
Application Portablility for e.g. Application developed in one application server environment can be easily be ported on to other application server environment is possible or not.
EJB Support
External Data Integeration Support like support to Integrate with legacy systems via: CICS,IMS,Tuxedo, MQ Series
Distributed Standards Protocols Supported
Platforms Supported like support for Windows , Solaris etc
Protocol Support like CORBA,IIOP,LDAP,JNDI,RMI,HTTP,SMTP,SNMP,NSAPI and ISAPI etc
Database Supported like support for ODBC, JDBC, Oracle, Sybase, MS-Access etc.
Modeling Tool Supported like support for Rational Rose etc
Component Management : - Provides the manager for handling all the components and run time services like session management. synchronous/asynchronous client