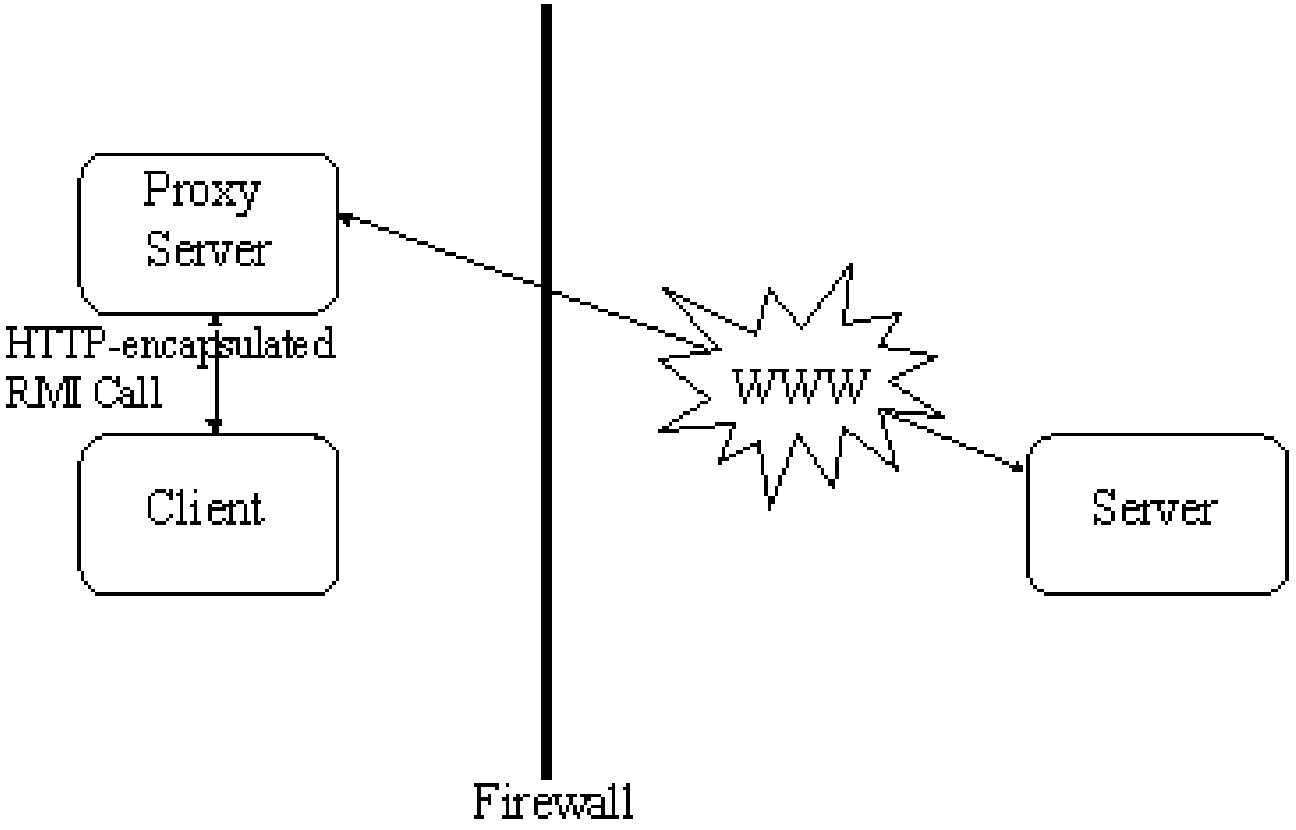
Firewall Issues

- **Firewalls are inevitably encountered by any networked enterprise application that has to operate beyond the sheltering confines of an Intranet**
- **Typically, firewalls block all network traffic, with the exception of those intended for certain "well-known" ports**
- **Since the RMI transport layer opens dynamic socket connections between the client and the server to facilitate communication, the traffic is typically blocked by most firewall implementations**
- **But luckily, the RMI designers had anticipated this problem, and a solution is provided by the RMI transport layer itself**
- **To get across firewalls, RMI makes use of HTTP tunneling by encapsulating the RMI calls within an HTTP POST request**
- **The possible scenarios: the RMI client, the server, or both can be operating from behind a firewall**



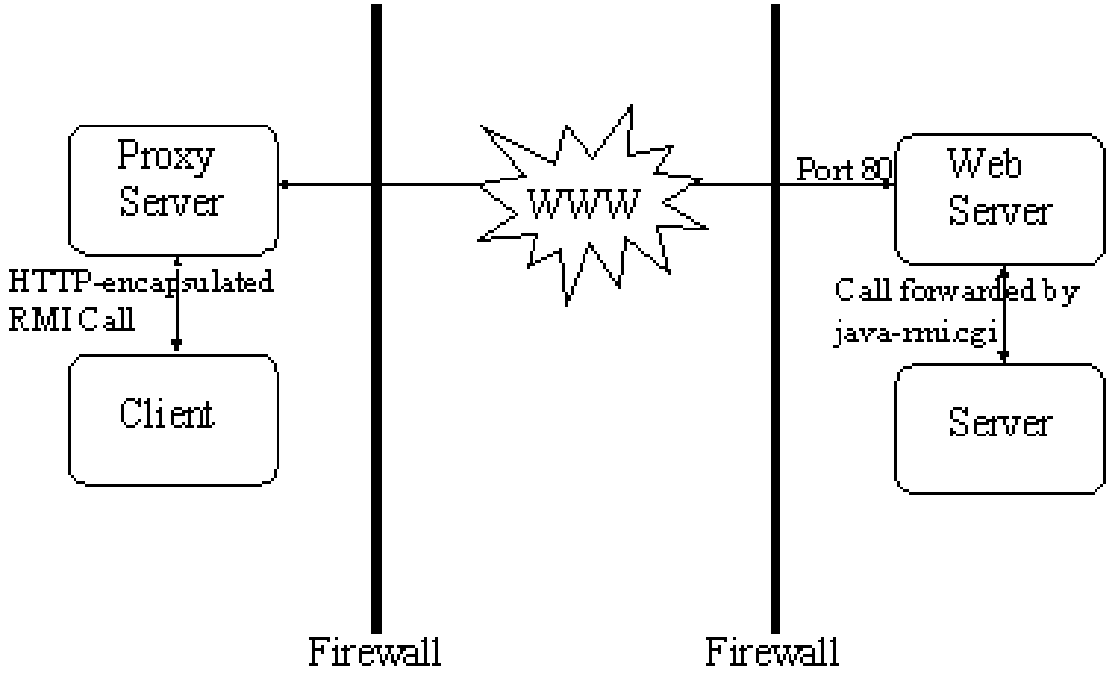
- **When the transport layer tries to establish a connection with the server, it is blocked by the firewall. When this happens, the RMI transport layer automatically retries by encapsulating the call data within an HTTP POST request**

- **The HTTP POST header for the call is in the form**

  `http://hostname:port`

- **If a client is behind a firewall, it is important that you also set the system property `http.proxyHost` appropriately**
- **Since almost all firewalls recognize the HTTP protocol, the specified proxy server should be able to forward the call directly to the port on which the remote server is listening on the outside**
- **Once the HTTP-encapsulated data is received at the server, it is automatically decoded and dispatched by the RMI transport layer**
- **The reply is then sent back to client as HTTP-encapsulated data**

- **The following diagram shows the scenario when both the RMI client and server are behind firewalls, or when the client proxy server can forward data only to the well-known HTTP port 80 at the server**



- **In this case, the RMI transport layer uses one additional level of indirection. This is because the client can no longer send the HTTP-encapsulated JRMP calls to arbitrary ports as the server is also behind a firewall**

- **Instead, the RMI transport layer places the call inside the HTTP packets and sends those packets to port 80 of the server**
- **The HTTP POST header is now in the form**

  `http://hostname:80/cgi-bin/java-rmi?forward=<port>`

- **This causes the execution of the CGI script, `java-rmi.cgi`, which in turn invokes a local JVM, unbundles the HTTP packet, and forwards the call to the server process on the designated port**
- **Replies from the server are sent back as HTTP REPLY packets to the originating client port where RMI again unbundles the information and sends it to the appropriate RMI stub**
- **Of course, for this to work, the `java-rmi.cgi` script, which is included within the standard Java 2 platform distribution, must be preconfigured with the path of the Java interpreter and located within the web server's cgi-bin directory**

- **It is also equally important for the RMI server to specify the host's fully-qualified domain name via a system property upon startup to avoid any DNS resolution problems**

  ```
  java.rmi.server.hostname=host.domain.com
  ```

- **Rather than making use of CGI script for the call forwarding, it is more efficient to use a servlet implementation of the same**
- **RMI suffers a significant performance degradation imposed by HTTP tunnelling**
- **The RMI application will no longer be able to multiplex calls on a single connection, since it would now follow a discrete request/response protocol**
- **Additionally, using the cgi script exposes a fairly large security loophole on your server machine, as now, the script can redirect any incoming request to any port, completely bypassing your firewalling mechanism**
- **Using HTTP tunneling precludes RMI applications from using callbacks, which in itself could be a major design constraint**

  ```
  java.rmi.server.disableHttp=true
  ```

Distributed Garbage Collection

- One of the joys of programming for the Java platform is not worrying about memory allocation
- The JVM has an automatic garbage collector that will reclaim the memory from any object that has been discarded by the running program
- One of the design objectives for RMI was seamless integration into the Java programming language, which includes garbage collection
- Designing an efficient single-machine garbage collector is hard; designing a distributed garbage collector is very hard
- The RMI system provides a reference counting distributed garbage collection algorithm based on Modula-3's Network Objects
- This system works by having the server keep track of which clients have requested access to remote objects running on the server
- When a reference is made, the server marks the object as "dirty" and when a client drops the reference, it is marked as being "clean"

- **The interface to the DGC (distributed garbage collector) is hidden in the stubs and skeletons layer**
- **However, a remote object can implement the `java.rmi.server.Unreferenced` interface and get a notification via the `unreferenced` method when there are no longer any clients holding a live reference**
- **In addition to the reference counting mechanism, a live client reference has a lease with a specified time**
- **If a client does not refresh the connection to the remote object before the lease term expires, the reference is considered to be dead and the remote object may be garbage collected**
- **The lease time is controlled by the system property `java.rmi.dgc.leaseValue`**
- **The value is in milliseconds and defaults to 10 minutes.**
- **Because of these garbage collection semantics, a client must be prepared to deal with remote objects that have "disappeared"**

DGC Example

- There are two remote objects, `Hello` and `MessageObject`
- Their implementations are designed to print out information when they are created, unreferenced, finalized and then deleted
- A remote object can implement the `Unreferenced` interface and its one method, `unreferenced`
- This method is called by the DGC when it removes the last remote reference to the object
- `MessageObjectImpl` and `HelloImpl` are designed to print a message when this happens
- `MessageObjectImpl` and `HelloImpl` also implement the `finalize` method. This is called when the local garbage collector is about to destroy an object and reclaim its memory space
- In this implementation, `MessageObjectImpl` and `HelloImpl` print a message to the console

- **We run RMIServer and two copies of RMIClient**
- **We will experiment with the setting of the Java heap size (use the -mx command line argument) and with explicitly setting the DGC remote reference leaseValue. To change this, use the following command line:**

```
java -Djava.rmi.dgc.leaseValue=10000 RMIServer
```

where the unit of time for leaseValue is in milliseconds

- **The constant `HOST_NAME` must match the computer name**

```
import java.rmi.*;

public interface Hello extends java.rmi.Remote
{   String sayHello() throws RemoteException;
    MessageObject getMessageObject() throws RemoteException;
}
```

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

public class HelloImpl extends UnicastRemoteObject
    implements Hello, Unreferenced
{   public HelloImpl() throws RemoteException
    {   super();
    }

    public String sayHello() throws RemoteException
    {   return "Hello!";
    }

    public MessageObject getMessageObject()
        throws RemoteException
    {   return new MessageObjectImpl();
    }
    [to be continued…]
```

```
    public void unreferenced()
    {    System.out.println( "HelloImpl: Unreferenced" );
    }

    public void finalize() throws Throwable
    {    super.finalize();
         System.out.println( "HelloImpl: Finalize called" );
    }
} // class HelloImpl
```

```
import java.io.Serializable;
import java.rmi.server.*;

public interface MessageObject extends java.rmi.Remote
{   int getNumberFromObject() throws java.rmi.RemoteException;
    int getNumberFromClass() throws java.rmi.RemoteException;
}
```

```
public class MessageObjectImpl extends UnicastRemoteObject
    implements MessageObject, Serializable, Unreferenced
{   static  int number = 0;
    private int objNumber;

    public MessageObjectImpl() throws RemoteException
    {   objNumber = number;
        System.out.println( "MessageObject:
            Class Number is #" + number + " Object Number
            is #" +  objNumber );
        number++;
    }

    public int getNumberFromObject()
    {   return objNumber;
    }

    public int getNumberFromClass()
    {   return number;
    }
    [to be continued…]
```

```
    public void finalize() throws Throwable
    {     super.finalize();
          System.out.println( "MessageObject: Finalize for
              object #: " + objNumber );
    }

    public void unreferenced()
    {     System.out.println( "MessageObject: Unreferenced for
              object #: " + objNumber );
    }
} // class MessageObjectImpl
```

```
import java.net.*;
import java.io.*;

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.LocateRegistry;

public class RMIServer
{   private static final int    PORT        = 10007;
    private static final String HOST_NAME   = "name";

    private static RMIServer  rmi;

    public static void main ( String[] args )
    {   System.setSecurityManager( new RMISecurityManager() );
        try
        {   rmi = new RMIServer();
        }
        [to be continued…]
```

```
    catch ( java.rmi.UnknownHostException uhe )
    {   System.out.println( "The host computer name you
            have specified, " + HOST_NAME + " does not
            match your real computer name." );
    }
    catch ( RemoteException re )
    {   System.out.println( "Error starting service" );
        System.out.println( "" + re );
    }
    catch ( MalformedURLException mURLe )
    {   System.out.println( "Internal error" + mURLe );
    }
    catch ( NotBoundException nbe )
    {   System.out.println( "Not Bound" );
        System.out.println( "" + nbe );
    }
}// main

public RMIServer() throws RemoteException,
    MalformedURLException, NotBoundException
{   LocateRegistry.createRegistry( PORT );
    [to be continued…]
```

```
            System.out.println( "Registry created on host computer
                 " + HOST_NAME + " on port " + Integer.toString(
                 PORT) );
            Hello hello = new HelloImpl();
            System.out.println( "Remote HelloService
                implementation object created" );
            String urlString = "//" + HOST_NAME + ":" +
                Integer.toString( PORT ) + "/" + "Hello";
            Naming.rebind( urlString, hello );
            System.out.println( "Bindings Finished, waiting for
                client requests." );
        }
}  // class RMIServer
```

```
import java.util.Date;
import java.net.MalformedURLException;

import java.rmi.*;

public class RMIClient
{   private static final int    PORT        = 10007;
    private static final String HOST_NAME   = "name";

    private static RMIClient  rmi;

    public static void main ( String[] args )
    {   rmi = new RMIClient();
    }

    public RMIClient()
    {   try
        {   Hello hello = (Hello)Naming.lookup( "//" +
            HOST_NAME + ":" + Integer.toString( PORT ) +
            "/" + "Hello" );
            [to be continued…]
```

```
            System.out.println( "HelloService lookup
                successful" );
            System.out.println( "Message from Server: " +
                hello.sayHello() );

            MessageObject mo;
            for ( int i = 0; i< 1000; i++ )
            {   mo = hello.getMessageObject();
                System.out.println( "MessageObject: Class
                Number is #" + mo.getNumberFromClass() +
                " Object Number is #" +
                mo.getNumberFromObject() );
                mo = null;
                Thread.sleep(500);
            }
        }
        catch ( Exception e)
        {       System.out.println(e);
        }
    }

}  // class RMIClient
```

The `RMIClientSocketFactory` interface

- An `RMIClientSocketFactory` instance is used by the RMI runtime in order to obtain client sockets for RMI calls
- A remote object can be associated with an `RMIClientSocketFactory` when it is created/exported via the constructors or `exportObject` methods of `java.rmi.server.UnicastRemoteObject` and `java.rmi.activation.Activatable`
- An `RMIClientSocketFactory` instance associated with a remote object will be downloaded to clients when the remote object's reference is transmitted in an RMI call
- This `RMIClientSocketFactory` will be used to create connections to the remote object for remote method calls

- An `RMIClientSocketFactory` instance can also be associated with a remote object registry so that clients can use custom socket communication with a remote object registry
- An implementation of this interface should be serializable and should implement `Object.equals(java.lang.Object)` to return true when passed an instance that represents the same (functionally equivalent) client socket factory, and false otherwise
- It should also implement `Object.hashCode()` consistently with its `Object.equals` implementation

Method Summary	
Socket	createSocket(String host, int port) Create a client socket connected to the specified host and port.

The `RMIClientSocketFactory` interface

- An `RMIServerSocketFactory` instance associated with a remote object is used to obtain the `ServerSocket` used to accept incoming calls from clients
- An `RMIServerSocketFactory` instance can also be associated with a remote object registry so that clients can use custom socket communication with a remote object registry

Method Summary	
ServerSocket	createServerSocket(int port) Create a server socket on the specified port (port 0 indicates an anonymous port).

Using a Custom RMI Socket Factory

- **Implementation and use of a custom RMI socket factory e.g. when**
  - **RMI clients and servers need to use sockets that encrypt or compress data**
  - **the application requires different socket types for different remote objects**
- **Prior to the Java™ 2 SDK, v1.2 release, it was possible to create and install a custom `java.rmi.server.RMISocketFactory` subclass used globally for all connections created by the RMI transport**
- **It was not possible, however, to associate a different RMI socket factory on a per-object basis**
- **For example in JDK™ v1.1.x, an RMI socket factory could not produce SSL sockets for one object and use the *Java Remote Method Protocol* (JRMP) directly over TCP for a different object in the same virtual machine**
- **As of the Java 2 SDK, v1.2 release, an RMI application can use a custom RMI socket factory on a per-object basis, download a client-side socket factory, and continue to use the default rmiregistry**

- **The type of socket to use is an application-specific decision**
- **For instance, if your server sends or receives sensitive data, you might want a socket that encrypts the data**
- **For this example, the custom RMI socket factory will create sockets that perform simple XOR encryption**
- **This type of encryption will protect data from a casual snooper sniffing packets on the wire, but is easily decoded by a knowledgeable cryptanalyst**
- **XOR sockets use special input and output stream implementations to handle xor-ing the data written to or read from the socket**

```
import java.io.*;

public class XorInputStream extends FilterInputStream
{   private final byte pattern;

    public XorInputStream(InputStream in, byte pattern)
    {   super(in);
        this.pattern = pattern;
    }

    public int read() throws IOException
    {   int b = in.read();
        if (b != -1)
            b = (b ^ pattern) & 0xFF;
        return b;
    }

    [to be continued…]
```

```
    public int read(byte b[], int off, int len)
        throws IOException
    {   int numBytes = in.read(b, off, len);
        if (numBytes <= 0)
            return numBytes;
        for(int i = 0; i < numBytes; i++)
            b[off + i] = (byte)((b[off + i] ^ pattern)
                & 0xFF);
        return numBytes;
    }
} // class XorInputStream
```

```
import java.io.*;

public class XorOutputStream extends FilterOutputStream
{   private final byte pattern;

    public XorOutputStream(OutputStream out, byte pattern)
    {   super(out);
        this.pattern = pattern;
    }

    public void write(int b) throws IOException
    {   out.write((b ^ pattern) & 0xFF);
    }
}
```

```
import java.io.*;
import java.net.*;

public class XorServerSocket extends ServerSocket
{   private final byte pattern;

    public XorServerSocket(int port, byte pattern)
        throws IOException
    {   super(port);
        this.pattern = pattern;
    }

    public Socket accept() throws IOException
    {   Socket s = new XorSocket(pattern);
        implAccept(s);
        return s;
    }
}
```

```
import java.io.*;
import java.net.*;

public class XorSocket extends Socket
{   private final byte pattern;
    private InputStream in = null;
    private OutputStream out = null;

    public XorSocket(byte pattern)
        throws IOException
    {   super();
        this.pattern = pattern;
    }

    public XorSocket(String host, int port, byte pattern)
        throws IOException
    {   super(host, port);
        this.pattern = pattern;
    }
    [to be continued…]
```

```
    public synchronized InputStream getInputStream()
        throws IOException
    {   if (in == null)
            in = new XorInputStream(super.getInputStream(),
                pattern);
        return in;
    }

    public synchronized OutputStream getOutputStream()
        throws IOException
    {   if (out == null)
            out = new XorOutputStream(super.getOutputStream(),
                pattern);
        return out;
    }
} // class XorSocket
```

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class XorClientSocketFactory
    implements RMIClientSocketFactory, Serializable
{   private final byte pattern;

    public XorClientSocketFactory(byte pattern)
    {   this.pattern = pattern;
    }

    public Socket createSocket(String host, int port)
        throws IOException
    {   return new XorSocket(host, port, pattern);
    }
    [to be continued…]
```

```
    public int hashCode()
    {   return (int) pattern;
    }

    public boolean equals(Object obj)
    {   return (getClass() == obj.getClass() &&
            pattern == ((XorClientSocketFactory)
                obj).pattern);
    }
} //class XorClientSocketFactory
```

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class XorServerSocketFactory
    implements RMIServerSocketFactory
{   private byte pattern;

    public XorServerSocketFactory(byte pattern)
    {   this.pattern = pattern;
    }

    public ServerSocket createServerSocket(int port)
        throws IOException
    {   return new XorServerSocket(port, pattern);
    }

    public int hashCode() {[Same as client]}

    public boolean equals(Object obj) {[Same as client]}
}
```

```
public interface Hello extends java.rmi.Remote
{   String sayHello() throws java.rmi.RemoteException;
}
```

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class HelloImpl implements Hello
{   public HelloImpl() {}

    public String sayHello()
    {   return  "Hello World!";
    }

    public static void main(String args[])
    {   System.setSecurityManager(new SecurityManager());
        [to be continued…]
```

```
            byte pattern = (byte) 0xAC;
            try
            {   HelloImpl obj = new HelloImpl();
                RMIClientSocketFactory csf =
                    new XorClientSocketFactory(pattern);
                RMIServerSocketFactory ssf =
                    new XorServerSocketFactory(pattern);
                Hello stub = (Hello) UnicastRemoteObject.
                    exportObject(obj, 0, csf, ssf);
                LocateRegistry.createRegistry(2002);
                Registry registry = LocateRegistry.
                    getRegistry(2002);
                registry.rebind("Hello", stub);
                System.out.println("HelloImpl bound in registry");

            }
            catch (Exception e)
            {   System.out.println("HelloImpl exception: " +
                    e.getMessage());
                e.printStackTrace();
            }
        }
} //class HelloImpl
```

```
import java.rmi.*;
import java.rmi.registry.*;

public class HelloClient
{   public static void main(String args[])
    {   System.setSecurityManager(new SecurityManager());
        try
        {   Registry registry =
                LocateRegistry.getRegistry(2002);
            Hello obj = (Hello) registry.lookup("Hello");
            String message = obj.sayHello();
            System.out.println(message);

        }
        catch (Exception e)
        {   System.out.println("HelloClient exception: " +
                e.getMessage());
            e.printStackTrace();
        }
    }
}
```