

The Readers/Writers Problem

- The producer/consumer problem is one classic distributed computing problem.
- Another is the readers/writers problem.
- The problem is to allow concurrent access to some data.
- You want to allow multiple readers, but you do not want a reader and a writer to overlap. This could result in the reader reading inconsistent data.

Readers/Writers Equal Priority

```
public class ReadWriteNoPref
{
    public synchronized void startRead()
    {
        while(writers > 0)
            wait();
        readers++;
    }
    public synchronized void endRead()
    {
        readers--;
        notify();
    }
}
//continued
```

```
public synchronized void startWrite()
{
    while(readers > 0 || writers > 0)
        wait();
    writers++;
}
public synchronized void endWrite()
{
    writers--;
    notify();
}
private int readers, writers, writersWaiting;
}
```

notify() vs notifyAll()

- notify() awakens at most one thread.
- notifyAll() awakens all threads blocked on the lock for the specified object.
- Use notifyAll() when
 - there are many threads to wake up simultaneously, or
 - only one thread should continue but the determination of which one is up to the threads themselves.

Give Writers Priority

```
public class ReadWritePriority
{
    public synchronized void startRead()
    {
        while(writers > 0 || writersWaiting > 0)
            wait();
        readers++;
    }
    public synchronized void endRead()
    {
        readers--;
        notifyAll();
    }
}
// continued
```

```
public synchronized void startWrite()
{
    writersWaiting++;
    while(readers > 0 || writers > 0)
        wait();
    writersWaiting--;
    writers++;
}
public synchronized void endWrite()
{
    writers--;
    notifyAll();
}
private int readers, writers, writersWaiting;
}
```

Protecting read/write

```
public class ReadWritePriority2
{
    public synchronized void startRead()
    {
        while(writers > 0 || writersWaiting > 0)
            wait();
        readers++;
        readList.add(Thread.currentThread());
    }
    public synchronized void endRead()
    {
        readers--;
        readList.remove(Thread.currentThread());
        notifyAll();
    }
}
```

```
public synchronized void startWrite()
{
    writersWaiting++;
    while(readers > 0 || writers > 0)
        wait();
    writersWaiting--;
    writers++;
    theWriter = Thread.currentThread();
}
public synchronized void endWrite()
{
    writers--;
    theWriter = null;
    notifyAll();
}
```

```
public Object read(ObjectInput in)
{
    if(!readList.contains(Thread.currentThread()))
        throw new IllegalReadException();
    return in.readObject();
}
public void write(ObjectOutput out)
{
    if(theWriter != Thread.currentThread())
        throw new IllegalWriteException();
    out.writeObject();
}
private int readers, writers, writersWaiting;
private Vector readList = new Vector();
private Thread theWriter;
}
```

More about java.lang.Thread

- yield()
- sleep(milliseconds)
- join(), join(milliseconds)
- suspend(), resume()
- setPriority()
- Thread.currentThread()

Remote Method Invocation

- RMI allows a program running on one computer, to contain a reference to an object on another computer.
- After the initial setup, this makes communicating with another program, as easy as calling a method.

A Brief Look at RMI

- First let's assume that someone else has created a remote object and registered it with some lookup service.
- In addition, we will assume that the client will not be passing objects to the remote objects methods, that are instances of classes the server doesn't know about.

Get the remote interface

```
import java.rmi.*;

public interface MessageServer extends Remote
{
    public String getMessage() throws RemoteException;
}
```

Write the Client

```
import java.rmi.*;

public class Client
{
    public static void main(String[] args)
        throws java.rmi.RemoteException
    {
        MessageServer server = null;

        // Using rmi requires a security manager.
        if(System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());
    }
}
```

```
// Lookup the remote object.
try {
    String name = "/" + args[0] + "/MessageServer";
    server = (MessageServer) Naming.lookup(name);
}
catch (Exception e) {
    System.err.println("client failed " + e);
    e.printStackTrace();
}

// We can now use the object referenced by server
// just like any other object. It may throw a RemoteException
// hence the throws clause above.
System.out.println("Message Received: " + server.getMessage());
System.out.println("Message Received: " + server.getMessage());
}
```

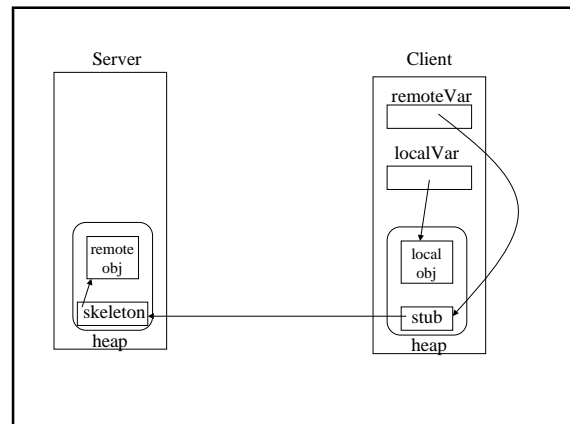
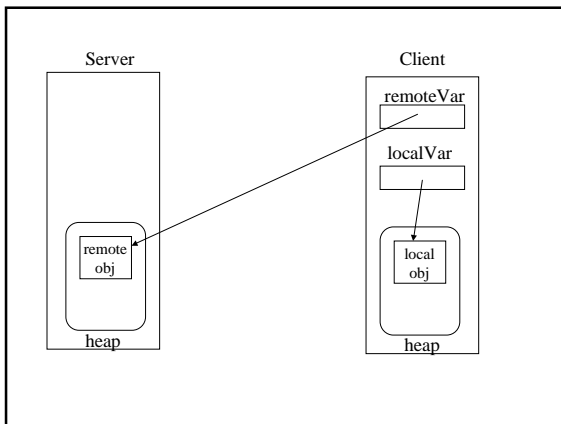
Run the Client

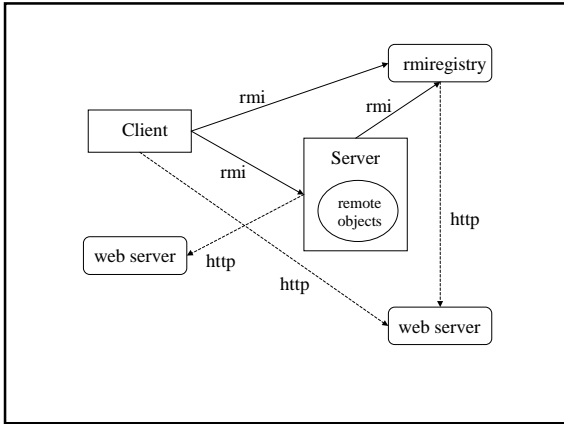
You must provide a security policy file. The one shown is sufficient for opening sockets between the client/server and web servers, serving up the required classes.

```
java -Djava.security.policy=java.policy Client sundance
```

Where java.policy contains:

```
grant {
    permission java.net.SocketPermission "*:1024-65535",
        "connect,accept";
    permission java.net.SocketPermission "*:80",
        "connect";
};
```





Creating the Remote Object

- 1 Create the interface that will be used by both the client and the server.
- 2 Create the remote class that implements the remote interface from step 1.
- 3 Create stub and skeleton files and put them where the rmiserver can find them via http.
- 4 Create the server that instantiates the remote object and registers it with the lookup service.
- 5 Start the lookup service (rmiregistry).
- 6 Run the server created in step 4.