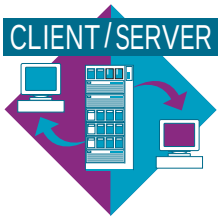
Multithreaded Programming with Java RMI



By Chary Tamirisa

This article presents an overview of the Java Remote Method Invocation (RMI) architecture, a description of its execution environment, the differences between synchronous and asynchronous method invocations, and two design choices for obtaining asynchronous behavior within the RMI framework using threads. Also included are synchronization issues specific to RMI along with a few tips. A detailed example illustrates the concepts discussed in this article. This article is based on Java RMI support in the JDK 1.1.4 running on AIX 4.3.

The Java™ Remote Method Invocation (RMI) provides a way to create distributed applications based on the client/server paradigm. RMI offers synchronous remote method invocation wherein the caller (client) waits until the remote method call returns from the server. RMI architecture is described on the JavaSoft™ Web page,[1] which provides a brief discussion of how remote method calls are mapped to the RMI threads. This article examines RMI's thread model in detail and outlines how to write asynchronous applications based on RMI.

Readers should be familiar with Java's thread support and the RMI model for distributed objects.

RMI Model

A *remote object* is one whose methods can be invoked from another Java Virtual Machine (JVM) running on the same machine or a remote machine. Typically, a server implements a remote object and a client invokes the methods in the remote object through a proxy. A proxy creates an environment in the client that allows the client to use the remote object as if it were local. Therefore, the client can invoke the methods transparently in the remote object.

The RMI system has three layers: the Stub/Skeleton, Remote Reference, and Transport. Together these three layers form the RMI runtime, which allows a client to invoke remote method calls as if they were local method calls. If a remote object becomes unreferenced, it becomes a candidate for the garbage collection.

Figure 1 shows the RMI architecture.

The Stub and Skeleton are Java classes automatically generated using the `rmic` command on the remote object implementation, or the server class. The Stub code runs in the client and the Skeleton code runs on the server. A key function of the Stub is to send the client's arguments to the server and obtain any return values. RMI uses the Java serialization of objects as the mechanism to send data from client to server and vice versa. Non-remote objects are passed by value whereas remote objects are passed by reference.

Chary Tamirisa

[1] RMI Architecture Specification. *http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.htm*

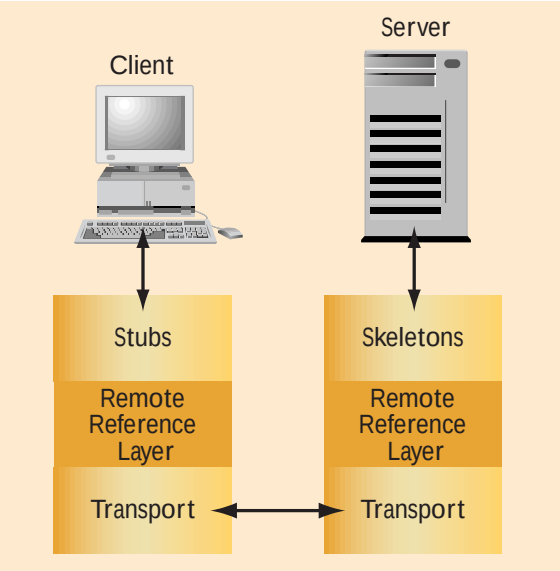


Figure 1. The architecture for the remote method invocation

The process of converting the arguments from their original form into the corresponding serialized form is called *marshalling*. The receiver converts (or deserializes) the serialized objects into the original form through a process called *unmarshalling*. Both the Stub and the Skeleton do marshalling and unmarshalling.

When a client makes a remote method call, the Stub initiates the call to the server. On the server side, the Skeleton makes the upcall to the actual implementation of the method and to marshal either the return value of the call or an exception. Every class that implements the `java.rmi.Remote` interface must have Stubs and Skeletons.

The Remote Reference layer enables the RMI architecture to be extensible in order to provide a variety of server and reference types. For example, RMI currently provides point-to-point (Unicast) server objects. However, in the future, it will allow multicast (one-to-many) reference types, which will allow replicated service.

The Transport layer listens to incoming requests, sets up connections for an incoming call, monitors the liveness of connections, and passes the connection to a dispatcher for the target remote call. Currently, the Transport layer implements the TCP protocol, although several other protocols can coexist with TCP.

RMI Execution Environment

To develop multithreaded applications based on RMI, it is important to understand the RMI execution environment. Figure 2 provides a snapshot of the threads in a Java RMI server application. Figure 3 provides a snapshot of the threads in a Java RMI client application. The snapshots are based on the Java Development Kit (JDK)1.1.4 running on AIX® 4.3.

Note: The actual implementation details may vary from platform to platform. Therefore, these threads cannot be used explicitly in your programs. Treat them as background threads created to facilitate RMI.

Server

The server side of RMI has three threadgroups: the system, the main, and the RMI runtime. A *threadgroup* defines a group of threads allowing operations such as `stop()` `suspend()`, `resume()` on all the threads in the group. Figure 2 shows the threadgroup, the thread name, its priority, and a notation of whether the thread is a daemon. A daemon thread is a background thread not expected to exit. When `main()` of the server application is invoked, two threadgroups already exist: system and main. The system threadgroup has one thread (Finalizer) and the main threadgroup consists of one thread (main). The Finalizer is a daemon thread but the main thread is not. The server application's `main()` is invoked in the main thread.

All RMI threads belong to the RMI runtime threadgroup. The RMI runtime starts when the server creates the remote object (the object whose methods can be invoked from another JVM). The RMI runtime threadgroup consists of the Pinger, Reaper, KeepAlive, LeaseRenewer, LeaseChecker, Cleaner, and a set of TCP Acceptor threads. From a programmer's viewpoint, the important set of threads are the TCP Acceptor threads. In the context of these threads, the remote methods are invoked when the Skeleton makes an upcall from the RMI's runtime to the actual method.

Client

The client-side RMI runtime does not contain all the threads that are found on

ThreadGroup	Thread Name	Priority	Is Daemon?
RMI Runtime	10 (MAX)		No
	Reaper	5	Yes
	KeepAlive	5	No
	Pinger	5	Yes
	LeaseRenewer	5	Yes
	LeaseChecker	5	Yes
	Cleaner	5	Yes
	TCP Accept-1	5	Yes
	TCP Accept-2	5	Yes
	TCP Accept-3	5	Yes
	TCP Accept-4	5	Yes
	TCP Accept-5	5	Yes
	TCP Accept-6	5	Yes
	TCP Accept-7	5	Yes
Main10		(max)	No
	Main	5	No
System		10 (max)	No
	Finalizer	1	Yes

Figure 2. Server-side RMI runtime threads

ThreadGroup	Thread Name	Priority	Is Daemon?
RMI Runtime		10 (max)	No
	Reaper	5	Yes
	Cleaner	5	Yes
	LeaseRenewer	5	Yes
Main		10 (max)	No
	Main	5	No
System		10 (max)	No
	Finalizer	1	Yes

Figure 3. RMI runtime threads in a simple client

the server side. A client that invokes just remote methods in a server has three threadgroups as shown in Figure 3. It is noteworthy that the client has no TCP Acceptor threads, because all remote methods invoked by the client are executed in the context of the current client thread. Since all remote method calls are synchronous, the current client thread blocks until the remote method call returns.

Mapping Method Calls to Threads

To use threads effectively, RMI programmers need to answer the following questions:

- Since a server consists of multiple RMI threads, how does RMI map incoming method calls to these threads?
- If the same client makes multiple calls to a server concurrently, will the server execute them concurrently or serially?
- What happens if the calls come from different clients (from different JVMs)?

RMI guarantees that calls originating from different client virtual machines will execute in different threads, which allows concurrency. However, calls originating from the same JVM are not guaranteed to

run concurrently; these calls may be invoked serially in some unspecified order.

Synchronous vs. Asynchronous Remote Methods

By definition, remote method invocations are synchronous: the client invokes a remote method and waits until the method is completed. Specifically, the calling thread waits until the remote method is completed and the results, if any, are passed back to the client.

The advantages of the synchronous call are that the client application does not have to do additional work to obtain the results of the method, and the remote call looks like a local call. If concurrent calls originating from the same JVM are mapped to different threads in the server's RMI runtime, the result is concurrency. However, RMI does not guarantee concurrency for multiple concurrent calls originating from the same JVM. If concurrent calls originating from the same JVM are mapped to the same RMI thread, the RMI remote method calls are executed serially in the server. But there is a way to handle this limitation.

An asynchronous remote method submits a request to a server and returns without waiting for the request to complete; the client can continue its execution. Java RMI does not support asynchronous semantics in its remote method calls. To get asynchronous behavior, it is necessary to create a multithreaded execution environment and invoke remote method implementations in the context of this multithreaded environment. That is, it is necessary to create a threaded execution environment and layer it over the RMI's runtime.

Asynchronous Application Design

In asynchronous application design, clients submit requests to a server in batch mode, but typically they do not wait for the remote methods to complete. The remote method invocations are asynchronous method calls. The clients may query the server to find status, or even cancel jobs as needed. The server typically notifies the appropriate client of the results after a job is completed.

The following scenario describes an asynchronous application. The client submits a remote request by invoking `submit()`. Once a job is submitted, the client can query the status of the job. When the job is completed, the server notifies the client.

Another scenario examines how a client can submit a request and cancel it later. The client creates a thread to invoke the remote `submit()` method and later creates another thread to invoke the remote `cancel()` method in the server. Figure 4 depicts this scenario.

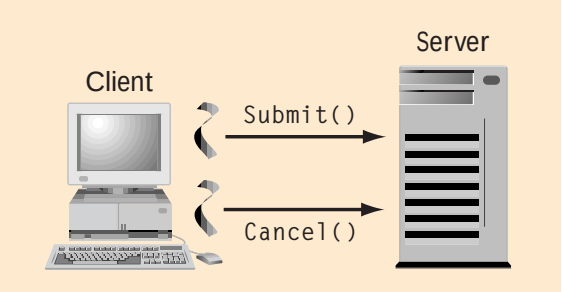


Figure 4. Two remote methods invoked by multithreaded client

On the server, let's assume that `submit()` method is running on thread TCP Accept-1 (RMI runtime thread). On which thread will `cancel()` run? If `cancel()` runs on another thread TCP Accept-2, it can interrupt thread TCP Accept-1 as shown in Figure 5. A thread can be interrupted or stopped by invoking the `interrupt()` or `stop()` method in the `java.lang.Thread` class.

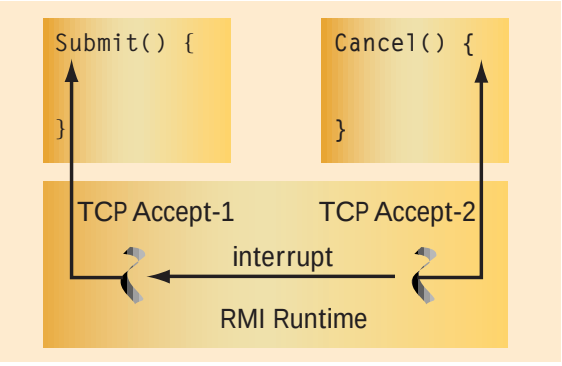


Figure 5. Server side-mapping each remote call from the same JVM to a different thread

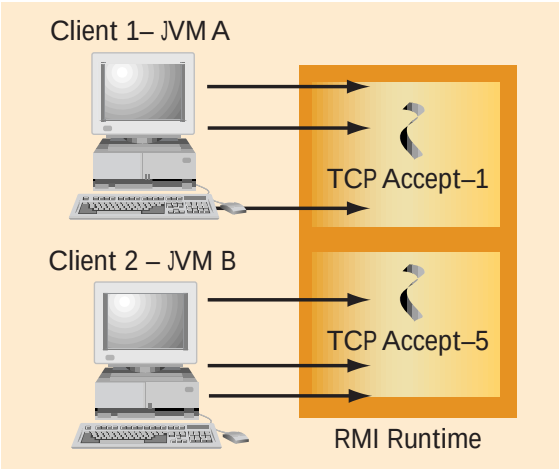


Figure 6. Server side—mapping all remote calls from the same JVM to the same TCP Accept thread

However, RMI does not guarantee mapping `cancel()` method (originating from the same client JVM) to another thread TCP Accept-2. It may map the `cancel()` request to thread TCP Accept-1, depicted in Figure 6.

It is clear that an infrastructure is needed on top of the RMI runtime to obtain asynchronous semantics. We need a new thread so that `submit()` runs on this thread rather than in the RMI runtime. Let's call this thread TS1, as shown in Figure 7. When `submit()` method is invoked on the server, the RMI runtime invokes `submit()` in one of the TCP Acceptor thread's context. From this thread, it is possible to create another thread (TS1) to distribute the actual work of the submit method `submitImpl()`. After starting this new thread, the RMI thread returns from the `submit()` method.

When `cancel()` is invoked on the server, the same or another RMI thread may invoke it. It does not matter which one. The `cancel()` method can determine which thread to cancel, cancel the thread, then return. Also, since `submit()` returns without waiting for completion, the client does not need a separate thread for `submit()` and `cancel()`. This enables asynchronous behavior from RMI.

This design is independent of the RMI implementation. A critical implementation requirement is the need for a unique identifier for each job (job ID) and a shared table (hashtable, for example) that uses this unique key to store all pertinent information associated with a job, including the thread ID. Since this table is shared across multiple clients, access to the table must be synchronized.

Callback Mechanism

The client has two choices in querying the status of the job it submitted. It can poll the server by periodically invoking the remote `status()` method, or alternately, the client can let the server invoke a client's method. The latter is known as a *callback*. To implement the callback mechanism, the client has two choices.

- Peer-to-Peer. In this model, the client and server are peers. The client also is a server and the server also is a client. Just like the server, the client exports its callback interface with a port number and hostname, then has the server look it up through the RMI Naming service. The client also extends `UnicastRemoteObject`. It is necessary to create the Stubs and Skeletons for the client side using `rmic` command.

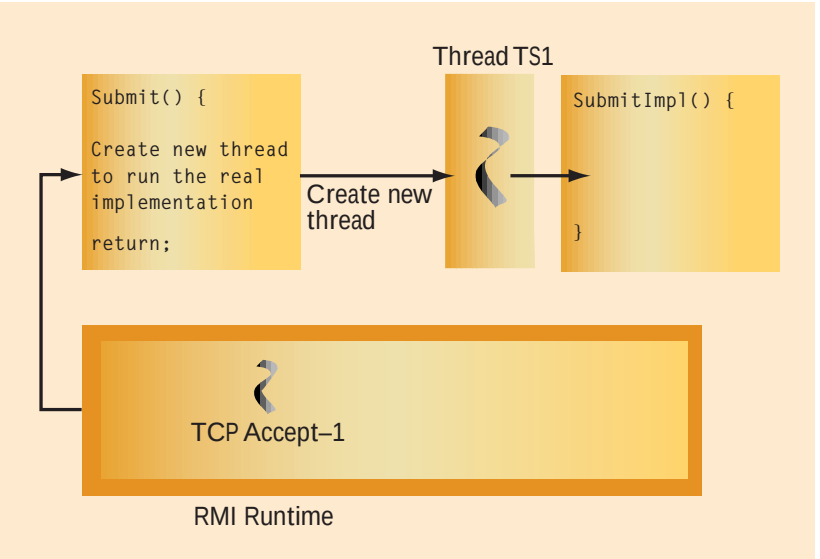


Figure 7. Application creating threads to obtain asynchronous behavior

- Callback Method. In this model, the client is not a peer because it does not export its interface to the RMI Naming and it does not extend `UnicastRemoteObject` (and hence, other clients cannot connect to it). The client implements the `java.rmi.Remote` interface and invokes `UnicastRemoteObject.exportObject()` to export the client object; it is not necessary to use the naming service. The client, of course, must implement the callback interface. The `rmic` command can generate the client-side Skeletons and Stubs in addition to the server-side counterparts. The client then can pass to the server a reference to itself, and the server can invoke the (callback) methods on the client.

The following describes how a client can export itself:

Callback Interface. The `callback` interface has one method `(jobDone())` which is invoked by the server when the job is completed. Note that the callback interface extends `java.rmi.Remote`, shown in Figure 8.

Client Class. The `PrintClient` class uses the RMI Naming service to look up the server. It then exports itself using the `exportObject()` method of `UnicastRemoteObject`, as shown in Figure 9.

Design Patterns

We have outlined two methods for obtaining asynchronous behavior. Let us apply these techniques to real applications. We will focus on the `callback` method of the client/server model, not the peer-to-peer model, because the Callback mechanism offers simplicity and retains the simpler client/server model. However, the following patterns can be used in the peer-to-peer model.

Two design patterns emerged from our experience. In the first pattern, remote methods took a long time for spawn threads to execute the actual call. In the second pattern, the remote method on the server placed the job in a queue, then woke up an executor thread to execute the job.

Spawn Threads from Remote Methods

This model assumes that the client needs asynchronous remote methods. Callbacks pass the results back to the client. Figure 10 captures this pattern.

To implement this pattern, follow these steps on the server:

- Identify the methods that need to run asynchronously.
- Create a separate thread (worker thread) each time these methods are invoked.
- Use `java.rmi.UID` class to create a unique identifier for the job.
- Record sufficient information (job ID) so the server can locate the client that originated the request. This information includes the client reference and the worker thread ID. The UID can be the key for a hashtable that stores this job ID.
- Once the job is completed, use the `callback` method of the client to send the results back to the client.

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

public interface Callback extends Remote {
            public void jobDone(UID id) throws RemoteException;
 }
```

Figure 8. Callback interface

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


public class PrintClient implements java.io.Serializable, Callback {

      PrintClient() {
         try {
             String url = "rmi://:2005/PRINTERS";
             Services server = (Services)Naming.lookup(url);
             UnicastRemoteObject.exportObject(this);
              ..
          }
          catch(Exception e) {}

          public void jobDone(UID id) {

      }
       /* Callback interface method */
      public void jobDone()throws RemoteException {

      }

}
```

Figure 9. Client class

The following applies to the client:

- Client implements a callback interface, such as `java.rmi.Remote interface`.
- Client exports itself using `java.rmi.server.UnicastObject.exportObject()`.
- Client submits a request to the server and records the UID returned by the server.
- Client can use the job UID to later query or cancel the job.

The advantage of this pattern is its simplicity; however, it also has a disadvantage. If the number of worker threads is directly proportional to the number of clients submitting jobs, then the number of threads created in a process places an upper limit on how many jobs can be supported simultaneously.

Queue Requests

Although this pattern is similar to the one above, it uses a queue instead of creating threads for each of the asynchronous methods, as shown in Figure 11. The remote method implementation (on the server) puts the job in a queue and wakes up a worker thread. As in the previous pattern, the remote method returns to the client with a unique identifier. When a client cancels a remote job using the UID, the server first tries to locate the job in the queue. If the server finds the job, it removes the job from the queue. If the job is not in the queue, the server then tries to see if the job is currently being executed by a worker thread, and if so, interrupts the thread,thereby, canceling the job.

If the server cannot find the job, it returns an appropriate status to the client. Similar to the previous pattern, the server uses a `callback` method to return results to the client.

We have outlined a single queue and one worker thread associated with it.

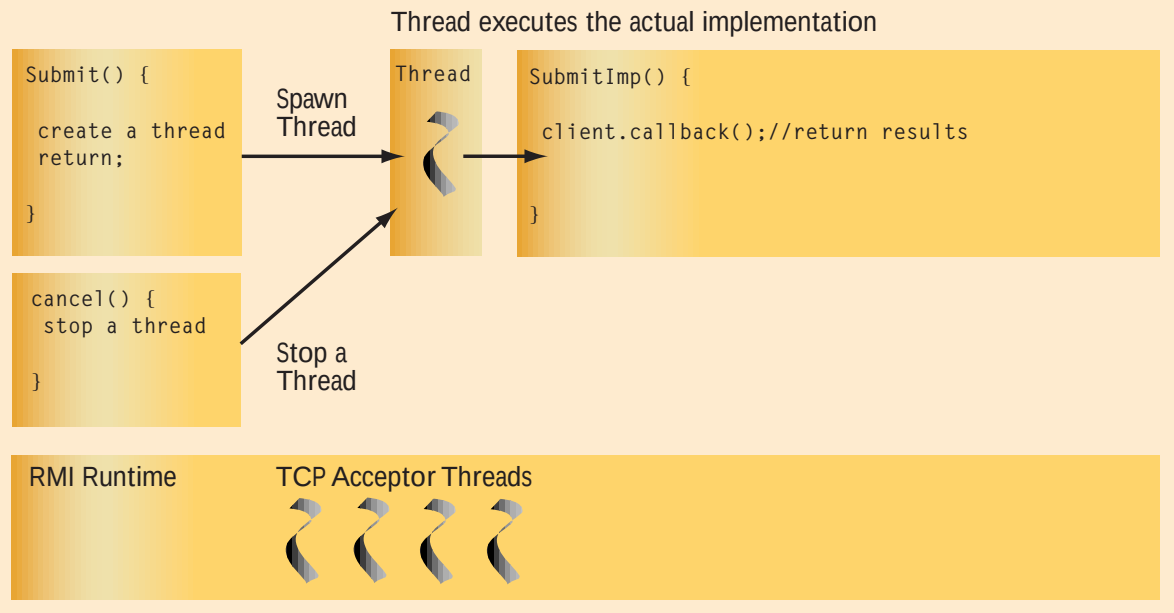


Figure 10. Design pattern: spawning a thread to obtain asynchronous behavior

Depending on the server needs, it is possible to have several queues. The worker threads can be dedicated to service a given queue, or shared by all queues from a general pool.

This pattern allows fewer worker threads to execute jobs coming from several clients. The server creates worker threads as needed. This pattern is advantageous when the worker thread has to invoke native methods, or use code that cannot be invoked from multiple threads concurrently. A single queue with a single worker thread allows serialized incoming requests.

Canceling Jobs

To cancel a job in process, the executing thread must be told to stop. Java provides two ways to do this: invoke the `interupt()` or `stop()` methods of the `java.lang.Thread` class.

Invoking `stop()` on a thread essentially terminates the thread. Therefore, within the queue model, a new thread must be started for other jobs in that particular queue to be executed.

Invoking `interrupt()` on a thread interrupts the thread if it is in a wait or sleep state and after the current job is deleted; the same thread can be used to execute the next job. However, if the thread is in a compute bound loop or blocked in an I/O, it may not be interrupted immediately. Hence, `interrupt()` may not be appropriate in some cases.

The Example section shows a sample application based on the queue design pattern. It illustrates the use of the `callback` method. In addition, it shows a simple way to persist (save) the client's job information so that a client can submit jobs to a server, exit the process, and later find the status of the jobs using the persisted file.

Synchronization Tips

Even if a server does not create any threads explicitly, the server runs in a multithreaded RMI environment. Typically, remote method invocations from different JVMs run concurrently on different threads in the server. When using synchronization in the server, be careful not to create deadlocks. For example, a deadlock arises when a thread T1 holds lock L1 and tries to acquire lock L2, while concurrently, thread T2 holding lock L2 tries to acquire lock L1. Both threads will become blocked forever because each thread needs a lock (resource) held by the other.

The deadlock situation especially can arise when using `callback` methods. Suppose a thread in the server holds a lock,

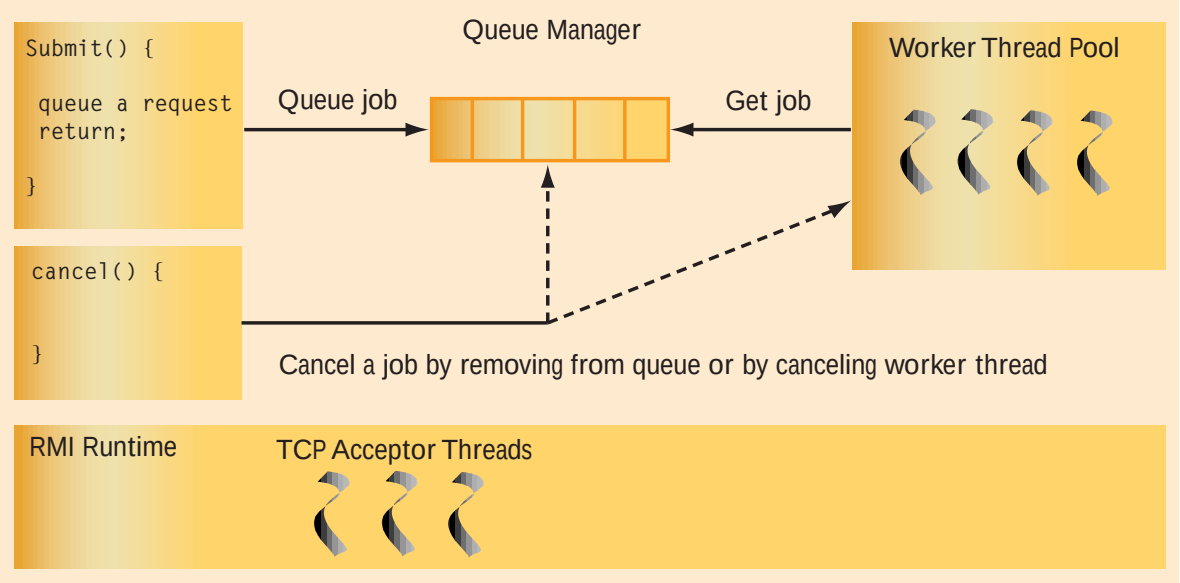


Figure 11. Design pattern: queue requests to worker threads

then invokes a client's `callback` method. If the `callback` method invokes a remote method on the server, the server can deadlock if contention exists for locks held by the calls. It is best not to hold locks while invoking `callback` methods.

Example

The following example was developed on AIX 4.3 using Java Development Kit 1.1.4 build a114-19980407.

The example illustrates the queue pattern to design an application. A `PrintServer` services requests for print jobs from clients. The `PrintClient` submits jobs to the server. The client does not wait for the job to complete, but persists its state (job information) and exits. At a later time, the client may cancel the jobs using the persisted information.

This example uses two interfaces: the Services and the `Callback` interfaces. The server implements the Services interface, which consists of three methods: `submit()`, `query()`, and `cancel()`. The `submit()` method executes the job asynchronously. It places a job request in a queue and returns to the client. `Query()` returns the current status of the job, and `cancel()` cancels the job either by removing it from the queue or by stopping the thread from executing the job.

This example uses one worker thread to take jobs off the queue and execute them. A pool of worker threads can be used instead of just one thread. The server creates the thread early in its execution. After a job is completed, the worker thread invokes a `callback` to the client with the execution results.

A `JobID` class saves the thread ID, the client object, and the status of the job. When a client invokes the `submit()` method, the server assigns a unique ID to the job (`java.rmi.server.UID()`). The server maintains a hashtable of these UIDs and the associated `JobIDs`, then places the job requests in a queue using the `put()` method of the `QueueManager` class. The `put()` method notifies the worker thread of work to do. The worker thread uses the `QueueManager's get()` method to get to the next executable job.

The `java.rmi.server.UID()` obtains a unique ID to tag a client's request. This UID is guaranteed to be unique across all clients. It is used as the hash key in a hashtable containing `JobID` objects.

The client (`PrintClient`) implements the `Callback` interface, which consists of the `jobDone()` method. It uses the `UnicastRemoteObject'sexportObject()` method to export itself so that the server can issue `callback` methods on it.

When the client invokes `submit()`, it receives a UID, uses this UID as a key, and saves the jobs it submits in a hashtable. The client invokes the `query()` and `cancel()` methods to get status and to cancel jobs respectively.

Since a user can submit a set of jobs using the client and also terminate the client, we introduce Java's persistence to save the client's hashtable. This allows users to cancel or query the job status at a later time. We use the `SaveJob` class for this purpose.

Note the three important methods in this class:

- `save()`: Serializes the object given to it and saves it in a specified file
- `restore()`: Takes a filename and returns the deserialized object
- `getNextRJEID()`: All file names have two parts: a simple integer value followed by the suffix .RJE (for example, `1.RJE` is a filename). To get the next ID, `getNextRJEID()` gets a list of all file names in the `/tmp/RJE` directory. It steps through all of the prefixes to determine the maximum integer value of the prefixes. It increments this maximum integer value by one to get the prefix for the next filename. It creates files such as `1.RJE`, `2.RJE`, and so on, in the `/tmp/RJE` directory.

After submitting a set of remote jobs, the client process can exit. A user can find status and cancel jobs, as needed, at a later time by invoking the `CancelJobs` class with an argument indicating the persisted filename (such as `1.RJE`).

Files

The files in Figure 11 are used in the example.

Compiling Files Here is how the files are compiled:

```
javac *.java
rmic PrintServer
rmic PrintClient
```

Running the Example Here is how to run the example:

```
rmi registry & (background)
java PrintServer (in a dedicated
  window)
```

You may run the client commands in separate windows: `java PrintClient`

Note the filename to use in canceling jobs: java CancelJobs 1.RJE.

The files that follow illustrate the queue pattern for designing an application.

Filename	Type	Description
`CallBack.java`	Interface	Used by server to call back client
`Job.java`	Class	Used by client to create a job
`JobID.java`	Class	Created by server to store the client object, the thread ID, the job status
`PrintClient.java`	Class	The client application
`PrintServer.java`	Class	The server application
`QueueManager.java`	Class	Manages the queue
`Services.java`	Interface	Server interface
`SaveJob.java`	Class	Used by client to save its job information
`CancelJobs.java`	Class	Client-side application to cancel jobs

Figure 11. Files used in example

Java Code

CallBack.java

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

public interface Callback extends Remote {
      public void jobDone(UID uid, String result) throws RemoteException;
}
```

Job.java

```
import java.io.*;

public class Job implements Serializable {
      int id;
      Job(int i) {
            id = i;
      }
      int getID() {
            return id;
      }
}
```

JobID.java

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

public class JobID implements java.io.Serializable {

      UID uid;
      Job job;
      String status;
      transient Thread threadID;
      Callback client;

      JobID() {
            uid = null;
            job = null;
            threadID = null;
            status = "Not Initialized";
      }

      UID getUID() {
            return this.uid;
      }
      void setUID(UID uid) {
            this.uid = uid;
      }
      synchronized void setJob(Callback client, Job job ) {
```

JobID.java continued on following page

JobID.java

```
            uid = new java.rmi.server.UID();
            this.uid = uid;
            this.job = job;
            this.client = client;
      }

      synchronized void setStatus(String status) {
            this.status = status;
      }
      synchronized String getStatus() {
            return this.status ;
}
      synchronized void setThread( Thread thread) {
            this.threadID = thread;
      }
      synchronized Thread getThread() {
            return this.threadID ;
      }
      void callback() {
            String curStatus;
            synchronized(this) {
                 curStatus = new String(status);
            }

            try {
                  client.jobDone(uid, curStatus);
            }
            catch(RemoteException re) {
                  System.out.println("JobID: callback except=" + re);

            }
      }


}
```

PrintClient.java

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


public class PrintClient implements java.io.Serializable, Callback {


      Hashtable jobTable= new Hashtable();
      String s;
      String filename;

      public static void main(String[] args) {
            new PrintClient();

      }
```

PrintClient.java continued on following page

PrintClient.java

```
      void showStatus(UID uid, String s) {
            Job job;
            job = (Job)jobTable.get(uid);
            System.out.println("UID= " + uid.hashCode() + "    " +
"JobID="+job.getID()+ ": status= " + s);
      }

      PrintClient() {
            String url = "rmi://:8010/PRINTERS";
            Services server ;
            Job job ;
            UID jobID1, jobID2, jobID3;


            try {

            /* Connect to a Print Server */
            server = (Services)Naming.lookup(url);
            /* Export the client's reference */
            UnicastRemoteObject.exportObject(this);

            SaveJob saveJob = new SaveJob("/tmp/RJE");

            /* Create jobs and submit them */
            job = new Job(1);
            jobID1 = server.submit(this, job);
            jobTable.put(jobID1, job);
            filename =  saveJob.getNextRJEID();
            job = new Job(2);
            jobID2 = server.submit(this, job);
            jobTable.put(jobID2, job);
            job = new Job(3);
            jobID3 = server.submit(this, job);
            jobTable.put(jobID3, job);

            s = server.query(jobID1) ;
            showStatus(jobID1, s);
            s = server.query(jobID2) ;
            showStatus(jobID2, s);
            s = server.query(jobID3) ;
            showStatus(jobID3, s);

            saveJob.save(jobTable, filename);
            System.out.println("PrintClient saved file=" + filename);

            }
            catch(Exception  r) {
              System.out.println("PrintClient client got exception = " + r);
            }
      }

      public void jobDone(UID uid, String result){
            showStatus(uid, result);
      }
}
```

PrintServer.java

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


public class PrintServer extends java.rmi.server.UnicastRemoteObject implements
java.io.Serializable, Services {

      public static void main(String[] args) {

            try {
            System.out.println("Welcome to Print Server ");
            PrintServer prs =new PrintServer();

            System.out.println("Locate registry");
            LocateRegistry.createRegistry(8010);
            String url = "rmi://:8010/PRINTERS";
            System.out.println("Bind to 8010");
            Naming.rebind(url, prs);
            }catch(Exception e) {
            System.out.println("exception got..");
            }
            System.out.println("Server in business..");
      }

      QueueManager qmanager;
      JobExecuter jobExecuter;
      Hashtable map = new Hashtable();


      PrintServer() throws RemoteException {
            qmanager = new QueueManager();
            jobExecuter= new JobExecuter (qmanager);
      }

      public UID submit(Callback client, Job job) throws RemoteException {
            JobID jobID = new JobID();
            jobID.setJob(client, job);
            jobID.setStatus("Waiting");
            map.put(jobID.getUID(), jobID);
            qmanager.put(jobID);
            /* Restart the executer thread if not alive*/
            if(!jobExecuter.isAlive())
                  jobExecuter = new JobExecuter(qmanager);

            return jobID.getUID();
      }
      public String query(UID uid) throws RemoteException {
            JobID jobID= (JobID)map.get(uid);
            if(jobID == null)
                  return "Job Not Found";
            else
            return jobID.getStatus();
      }

      public String cancel(UID uid) throws RemoteException {
```

PrintServer.java continued on following page

PrintServer.java

```
            JobID jobID = (JobID)map.get(uid);
            if(qmanager.cancel(jobID)) {
                  System.out.println("cancel waiting job: uid=" + uid);
                  jobID.setStatus("Canceled");
                  map.remove(uid);
                  return jobID.getStatus();
            }else if(jobID.getStatus() == "Executing"){
                  System.out.println("Cancel Executing Job: uid=" + uid);
                  jobID.getThread().stop();
                  if(!jobExecuter.isAlive()) {
                        System.out.println("Restart Executer thread");
                        jobExecuter = new JobExecuter(qmanager);
                  }
                  jobID.setStatus("Canceled");
                  map.remove(uid);
                  return jobID.getStatus();
            }else
                  return "Job Not Found!";
      }


      class JobExecuter extends Thread {
        QueueManager qmanager;
            JobExecuter(QueueManager qmgr) {
                  super();
                  this.qmanager = qmgr;
                  start();
            }

            public void run() {
                  System.out.println("running jobs..");
                  JobID jid = null;


                  for(;;)
                    {
                       synchronized(qmanager) {
                        try {
                          while( (jid = qmanager.get())== null)
                                qmanager.wait();

                        }
                        catch(InterruptedException e) {
                          System.out.println("Job Executer:interrupted=" + e);
                        }

                       }

                        jid.setThread(this);
                        jid.setStatus("Executing");

                       try{
                          // This must be replaced by the actual job */
                          sleep(2*60 * 1000); // Sleep for 2 minutes
                          }
                       catch(InterruptedException e) {
                          System.out.println("Job interrupted=" + e);
```

PrintServer.java continued on following page

PrintServer.java

```
                        jid.setStatus("Interrupted");
                        jid.callback();
                        continue;
                    }

                    // Now the job is done. So set status
                    jid.setStatus("JobDone");
                    // Let the client know
                    jid.callback();
                }
            }
        }

}
```

QueueManager.java

```
import java.util.*;
import java.io.*;

public class QueueManager implements java.io.Serializable {

    Vector jobs = new Vector();


    public synchronized void put(JobID id)  {
        jobs.addElement(id);
        notify();
    }

    public synchronized JobID get() {
        JobID id ;
        if(jobs.isEmpty()) return null;
        id = (JobID)jobs.firstElement();
        jobs.removeElement(id);
        return id;
    }
    private int findJobIDIndex(JobID id) {
        JobID jid;
        for(int i=0;i<jobs.size(); i++) {
            jid = (JobID)jobs.elementAt(i);
            if(jid.uid.equals(id.uid)) {
                return i;
            }
        }
        return -1;
    }
    public synchronized boolean cancel(JobID id) {
        JobID jid;
        int index = findJobIDIndex(id);
        if(index == -1) return false;
        jid = (JobID)jobs.elementAt(index);
        jobs.removeElement(jid);
        return true;

    }

}
```

Services.java

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

public interface Services extends Remote {
      public UID submit(Callback client, Job job) throws RemoteException;
      public String query(UID jobID) throws RemoteException;
      public String cancel(UID jobID ) throws RemoteException;
}
```

SaveJob.java

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

/**
Purpose: SaveJob allows persisting of jobs
*/

public class SaveJob
{


      public static String rootDir;
      int lastRJEID=0;    // last assigned persistent ID
      FileOutputStream f_out ;
      ObjectOutputStream out ;
      FileInputStream f_in ;
      ObjectInputStream in ;
      public  final String ioExc = " IO Exception:";
        public  final String misMatchExc = " Mismatch Exception:"
;
        public  final String classNotFoundExc = " Class Not Found Exception:";


      public SaveJob(String root)
      {
            this.rootDir = root;

      }
      public  void  save(Object obj, String fileName)
      {
            try
            {
                  f_out = new FileOutputStream(fileName);
            }
            catch(IOException e)
            {
                  System.out.println(ioExc + "FileName=" + fileName );
            }

            try
            {
                  out = new ObjectOutputStream(f_out);
```

Savejob.java continued on following page

SaveJob.java

```
            }
            catch(IOException e)
            {
                  System.out.println(ioExc + "FileName=" + fileName);
            }

            try {
                  out.writeObject(obj);
                  out.flush();
                  out.close();
            }
            catch(IOException e)
            {
                  System.out.println(ioExc + "FileName=" + fileName);
            }


      }

      public  Object restore(String fileName )
      {
Object obj=null;

            try
            {
                     String fullName= rootDir +  File.separatorChar+ fileName;
               f_in = new FileInputStream(fullName);
               in = new ObjectInputStream(f_in);
            }
            catch(IOException e)
            {
               System.out.println(ioExc + "FileName=" + fileName);
               return null;
            }
            try {
                obj = in.readObject();
                in.close();
            }
            catch(ClassNotFoundException el)
            {
               System.out.println(classNotFoundExc + "FileName=" + fileName);
            }

            catch(IOException e)
            {
                  System.out.println(ioExc + "FileName=" + fileName);
            }

            return obj;

      }
      public String getNextRJEID()
      {
                  if ( lastRJEID == 0 ) {
                  File file      = new File( rootDir );
                  String files[] = file.list();
                  if ( files != null ) {
```

Savejob.java continued on following page

SaveJob.java

```
                                for ( int i = 0; i < files.length; i++ ) {
                                String curFile = files[i];
                                int len     = curFile.length();
                                if(((curFile.substring( len - 4, len )).
toUpperCase()). compareTo( ".RJE" ) == 0 )
                                                lastRJEID = Math.max( lastRJEID,
                                                Integer.parseInt(
curFile.substring( 0, len - 4) ) );
                                        }
                        }
                        }
                        return rootDir +  File.separatorChar+(++lastRJEID) + ".RJE";
        }

}
```

CancelJobs.java

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


public class CancelJobs implements java.io.Serializable {


        Hashtable jobTable;
        String s;


        public static void main(String[] args) {
                new CancelJobs(args);

        }

        void showStatus(UID uid, String s) {
                Job job;
                job = (Job)jobTable.get(uid);
                System.out.println("UID= " + uid.hashCode() + "    " +
"JobID="+job.getID()+ ": status= " + s);
        }

        CancelJobs(String[] args) {
                String url = "rmi://:8010/PRINTERS";
                Services server ;
                Job job ;

                try {

                /* Connect to a Print Server */
                server = (Services)Naming.lookup(url);
                /* Export the client's reference */

                SaveJob saveJob = new SaveJob("/tmp/RJE");
```

CancelJobs.java continued on following page

CancelJobs.java

```
            System.out.println("Restore file=" + args[0] );
            jobTable = (Hashtable)saveJob.restore(args[0]);

            int size = jobTable.size();
            System.out.println("size=" + size);

            Enumeration keysEnum = jobTable.keys();
            UID uid;
            do{
                  uid = (UID)keysEnum.nextElement();
                  s = server.query(uid) ;
                  showStatus(uid, s);
                  s = server.cancel(uid);
                  showStatus(uid, s);
                  s = server.query(uid) ;
                  showStatus(uid, s);

            }while(keysEnum.hasMoreElements());


            }
            catch(Exception  r) {
              System.out.println("PrintClient client got exception = " + r);
            }
      }

}
```

Conclusion

The description of the RMI runtime, RMI's execution environment, synchronous and asynchronous method calls, and synchronization tips should provide enough knowledge to develop applications that require asynchronous behavior.

Reference

Downing, Troy Bryan. *Java RMI: Remote Method Invocation*. IDG Books Worldwide Inc. (*http://www.idgbooks.com*).

Chary Tamirisa, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com.

Mr. Tamirisa is currently the manager of the Performance and System Test group in the Global Business Intelligence Solutions (Data Mining) area. He has extensive experience in developing Java applications and the JavaBean component model. Mr. Tamirisa worked on the DCE threads package on AIX and OS/2®. He has also worked in the fields of communication protocols, system software, and National Language Support. Mr. Tamirisa has an MS in Computer Science from McGill University in Montreal, Quebec, Canada, and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.