Agenda

- Last time (Tues)
  - Interprocess Communication (chpt 4)
  - Sockets demo
- This time and tonight (5-6:15)/tomorrow (3-4:15): chpts 4-5
  - General RPC/RMI example
  - General RPC/RMI terminology
  - Case study: SUNRPC
  - Case study: Java RMI
  - General pros/cons of RPC/RMI
- No class Tues Feb 13 (Marty at conference)
- Note: we are skipping chpt 6
- Note: I’m generally out of contact between 7:30pm – 4am

[3] How are parameters passed?

- External data representation: An agreed standard for the representation of data structures and primitive values.
- Marshalling: The process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
  - Java, XML: “serialization”
- Unmarshalling: The process of disassembling data on arrival to produce an equivalent collection of data items at the destination.
  - Java, XML: “deserialization”

[3] How are parameters passed?

- External data representation and marshalling – approaches
  - SUN RPC’s XDR (RFC 1014)
  - CORBA’s common data representation (CDR).
  - Java’s object serialization.
  - .NET’s object serialization.
- Marshalling and unmarshalling activities intended to be carried out by a middleware layer without any involvement on the part of the application programmer.

[4] Binding

- How do we locate a remote service / object?
- May be done at various times
  - later -> more flexible
  - earlier -> more efficient communication
- Usually done at run-time
  - At initialization - typical rpc, some object systems
  - At invocation - to support mobility
- Both can use the same paradigm
  - locate on first contact and cache
  - relocate if cache fails


- Is server available?
  - Solution - factory
- Are versions consistent?
  - Solution - version number
  - Reject or multiple servers
- Multiple Servers
  - Load balancing
  - First reply

[5] What are the invocation semantics?

- Local invocations are executed exactly once.
- Cannot always be the case for remote method invocations.
- Possible run-time errors
  - Can’t find the server
  - Request to server is lost
  - Reply from server is lost
  - Server crashes
  - Client crashes
What are the invocation semantics?

**RPC/RMI invocation semantics:**
- Maybe.
- At-least once.
- At-most once.

**RPC/RMI invocation semantics depends on use of:**
- Retry request message.
- Duplicate filtering.
- Retransmission of results.

---

<table>
<thead>
<tr>
<th>Retransmit request message</th>
<th>Duplicate filtering</th>
<th>Re-execute procedure or retransmit reply</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Maybe</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Re-execute procedure</td>
<td>At-least-once</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Retransmit reply</td>
<td>At-most-once</td>
</tr>
</tbody>
</table>

---

(potential) transparencies of RPC/RMI

- Location transparency:
  - RMI and RPCs invoked without knowledge of the location of invoked method/procedure
- Transport protocol transparency:
  - e.g., request/reply protocol used to implement RPC can use either transport protocol
- Transparency of computer hardware and operating system:
  - e.g., use of external data representations
- Transparency of programming language used:
  - e.g., by use of programming language independent Interface Definition Languages, such as CORBA IDL

---

**RPC Case Study: SUNRPC**

- At one time, most widely used RPC system, developed for use with NFS
- Built on top of either UDP or TCP
  - TCP: stream is divided into records
  - UDP: total size of input < 8192 bytes
- Single parameter is passed (how can we do multiple arguments?)
- Reliability:
  - UDP: timeout + limited number of retransmissions
  - TCP: error condition returned if connection terminated by server
- Failure semantics:
  - at-least once if reply received
  - at-least zero when no reply received
  - options under UDP to try to enforce at-most-once semantics
- Uses Sun’s eXternal Data Representation (XDR)
  - Big endian order for 32 bit integers
  - Handles arbitrary data structures

---

**Binder – Port Mapper**

- Server: at start-up, server creates a UDP/TCP port (handle)
  - Server stub calls svc_reg with program number and version number as arguments to register server with local port mapper
  - Port mapper stores program number, version number, and port
- Client: at start-up, client calls svc_reg to request server port number
  - Upon return, client can call procedures available at server

---

**rpcgen: Generating Stubs**

- Q.x is the RPC specification file
- Q_xdr.c contains the data conversion calls to XDR

---
Let's expose two local functions via RPC:

- the "local time" – think "man 2 time"
- a "time stringifying function" – think "man 3 ctime"

First, define the .x file...

```
program DATE_PROG {
  version DATE_VERS {
    int BIN_DATE(void) = 1; /* procedure number = 1 */
    string STR_DATE(long) = 2; /* procedure number = 2 */
  } = 1; /* version number = 1 */
  program number = 0x31234567; /* program number = 0x31234567 */
}
```

- program number is 32-bit integer:
  - [0x00000000 -- 0x1fffffff] defined by SUN
  - [0x20000000 -- 0x3fffffff] defined by user
  - [0x40000000 -- 0x5fffffff] transient
  - [0x60000000 -- 0x7fffffff] reserved

- procedure numbers begin with 0 (procedure 0 is null procedure generated by `rpcgen`, which allows client to call it to verify existence of program)

Software Architecture of RPC Example

```
server stub
rdate.c
r_date

client stub
date_svc.c
date_clnt.c
date.h
libraries: nsl, "standard"
```

```
Output of rpcgen: date.h
/* Please do not edit this file. */
/* It was generated using rpcgen. */
#include <memory.h> /* for memset */
#include "date.h" /* this file is generated by rpcgen */
/* Default timeout can be changed using clnt_control() */

static struct timeval TIMEOUT = { 25, 0 };

int *
bin_date_1(void *argp, CLIENT *clnt)
{
  static int clnt_res;
  memset((char *)&clnt_res, 0, sizeof(clnt_res));
  if (clnt_call (clnt, BIN_DATE,
                 (xdrproc_t) xdr_void, (caddr_t) argp,
                 (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
                 TIMEOUT) != RPC_SUCCESS) {
    return (NULL);
  }
  return (&clnt_res);
}

char **
str_date_1(long *bintime, struct svc_req *rqstp)
{
  static char     *ptr;           /* must be static */
  char            *ctime();       /* Unix function */
  ptr = ctime(bintime);           /* convert to local time */
  return(&ptr);           /* return the address of pointer */
}
```

```Server side: date_proc.c
/* Please do not edit this file. */
/* It was generated using rpcgen. */
#include <rpc/rpc.h>     /* standard RPC include file */
#include "date.h"        /* this file is generated by rpcgen */

int *
bin_date_1_svc(void *argp, struct svc_req *rqstp)
{
  static int      timeval;        /* must be static */
  timeval = time((long *) 0);
  return(&timeval);
}

char **
str_date_1_svc(long *bintime, struct svc_req *rqstp )
{
  static char     *ptr;           /* must be static */
  char            *ctime();       /* Unix function */
  ptr = ctime(bintime);           /* convert to local time */
  return(&ptr);           /* return the address of pointer */
}
```

```
Output of rpcgen: date_clnt.c
/* Please do not edit this file. */
/* It was generated using rpcgen. */
#include <rpc/rpc.h>     /* standard RPC include file */
#include "date.h"        /* this file is generated by rpcgen */

int *
bin_date_1(void *argp, CLIENT *clnt)
{
  return (NULL);
}
```

```
Output of rpcgen: date_svc.c
```
Client side: rdate.c

```c
#include <stdio.h>
#include <rpc/rpc.h>     /* standard RPC include file */
#include "date.h"        /* this file is generated by rpcgen */
main(argc, argv)
      int argc;
      char *argv[];
{
  CLIENT          *cl;            /* RPC handle */
  char            *server;        /* server name */
  long            *lresult;        /* return value from bin_date_1() */
  char            **sresult;      /* return value from str_date_1() */
  if (argc != 2) {
    fprintf(stderr, "usage: %s hostname
", argv[0]);
    exit(1);
  }
  server = argv[1];
  /* Create the client "handle."   */
  if ( (cl = clnt_create(server, DATE_PROG, DATE_VERS , "udp")) == NULL) {
    /* Couldn't establish connection with server.        */
    clnt_pcreateerror(server);
    exit(2);
  }
  /* First call the remote procedure "bin_date".        */
  if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(3);
  }
  printf("time on host %s = %ld\n", server, *lresult) ;
  /* Now call the remote procedure "str_date". */
  if ( (sresult = str_date_1(lresult, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(4);
  }
  printf("time on host %s = %s", server, *sresult);
  clnt_destroy(cl);               /* done with the handle */
  exit(0);
}
```

Example Usage / Output

1. Boot the server on grad01
   grad01; date_svc &
   • creates a socket, binds a local port to the socket
   • then calls function in the RPC library to register number and version
   • port mapper keeps track of program number, version number, and port number
2. Invoke the client on viper
   • viper ; ./r_date grad01
   time on host grad01 = 1031527961
   time on host grad01 = Sun Sep  8 19:32:41 2002
   • first calls cint, create, which specifies name of remote system (grad01), program number, version number, and protocol
   • function contacts the port mapper on grad01 to determine port for server
   • client then calls bin_date (handled by client stub, which determines procedure being called and calls appropriate function on remote machine)

More on SUN RPC

• Recap of RPC history
  • Birell and Nelson: “Implementing remote procedure calls”, 1983
  • SUN RPC originated thereafter
  • SUN RPC has since become Open Network Computing (ONC) RPC
  • SUN delegated control of ONC RPC to IETF in 1995
• SUN RPC is nice, but is it transparent? What are some ways in which it is not transparent?

Chapter 5: RMI (Remote Method Invocation)

• Distributed object model: allow objects at different processes to communicate with each other using calls to remote methods
• Object: state, methods, interface
  • A must have a reference to object E so that it can invoke one of its methods
  • A and E: distant objects

Chapter 5: RMI (Remote Method Invocation - basics)

• Remote method invocation: Method invocation between objects in different processes (whether in the same computer or not).
• Local method invocation: Method invocations between objects in the same process.
• Remote objects: Objects that can receive remote invocations.
• Remote object reference: Unique identification of a remote object, needed by other objects wanting to invoke the methods of the remote object.
  • Very interesting: Naming issues (for example: )
  ![Diagram](image)
• Remote interface: Every remote object has a remote interface that specifies which of its methods can be invoked remotely.

![Diagram](image)
Remote method invocation - overview

Remote method invocation – basic modules

- Communication modules:
  - Carry out the request-reply protocol.
  - Provide specific invocation semantics.
- Remote reference module:
  - Translates between local and remote object references.
  - Has a remote object table with entries for all remote objects held by the process, and a record for each local proxy.
  - Creates remote object references.

Remote method invocation – RMI software

- A layer of software between the application-level objects and the communication and remote reference modules.
- Consists of:
  - Proxy.
  - Dispatcher.
  - Skeleton.
- The classes for the RMI software can be generated automatically by an interface compiler.

Remote method invocation – RMI software

RMI – dispatcher and skeleton

- Dispatcher:
  - Receives request message from communication module.
  - Uses methodId to select the appropriate method in the skeleton.
  - Passes on the request.
- Skeleton:
  - Unmarshals the arguments in the request message.
  - Invokes the corresponding method in the remote object.
  - Waits for result.
  - Marshals result.
  - Sends reply to sender’s proxy.

Remote method invocation – proxy

- Makes remote method invocation transparent to clients.
- Behaves like a local object to the invoker.
- Implements the methods in the remote interface of the remote object it represents.
- Does not execute invocation, but forwards request to remote object.
- Marshals a reference to the target object, its own methodId and its arguments into a request message, sends it to the target, waits the reply message, unmarshals it, and returns the result to the invoker.
Events and Notifications

- Heterogeneous: Components in a distributed system that were not designed to interoperate can be made to work together.
- Asynchronous: Publishers and subscribers are decoupled.

Events and notifications - issues

- Delivery semantics.
- Roles for observers:
  - Forwarding.
  - Filtering of notifications.
  - Patterns of events.
  - Notification mailboxes.

Case Study: Java RMI


Network Programming Paradigms

- Sockets programming:
  - Arguably, design a protocol first, then implement clients and servers that support the protocol.
- RMI:
  - Develop an application, then (statically) move some objects to remote machines.
  - Not concerned with the details of the actual communication between processes – everything is just method calls.

RMI Call Semantics

- RMI does a great job of providing natural call semantics for remote objects/methods.
  - Simply a few additional Exceptions that you need to handle.
  - Objects implementing the Remote interface are passed by reference. Non-remote (serializable) objects and primitive types are passed by value.

Finding Remote Objects

- It would be awkward if we needed to include a hostname, port and protocol with every remote method invocation.
- RMI provides a Naming Service through the RMI Registry that simplifies how programs specify the location of remote objects.
  - This naming service is a JDK utility called rmiregistry that runs at a well known address (by default).
Overview of RMI Programming

- Define an interface that declares the methods that will be available remotely.
- The server program must include a class that implements this interface.
- The server program must create a remote object and register it with the naming service.
- The client program creates a remote object by asking the naming service for an object reference.

Java Interfaces

- Similar to Class
- No implementation! All methods are abstract (virtual for C++ people).
- Everything is public.
- No fields defined, just Methods.
- No constructor
- An Interface is an API that can be implemented by a Class.

Interfaces and Inheritance

- In Java a class can only extend a single superclass (single inheritance).
- A class can implement any number of interfaces.
  - End result is very similar to multiple inheritance.

Sample Interface

```java
public interface Shape {
    public double getArea();
    public void draw();
    public void fill(Color c);
}
```

Server Details – extending Remote

- Create an interface that extends the java.rmi.Remote interface.
- This new interface includes all the public methods that will be available as remote methods.

```java
import java.rmi.*;
public interface MyRemote extends Remote {
    public int foo(int x) throws RemoteException;
    public void blah(int y) throws RemoteException;
    ...
}
```

How the interface will be used
• Create a class that implements the interface.
  • The class should extend UnicastRemoteObject (not strictly necessary but the general way)
• This class needs a constructor that throws RemoteException
• This class is now used by rmic to create the stub and skeleton code.

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
  public MyRemoteImpl() throws RemoteException {}
  public int add_one(int x) {
    return(x+1);
  }
  public String stringify_an_int(int y) {
    return("Your number is "+y);
  }
}

Server Details – Implementation Class

DEMO: Remote Object Implementation Class

Client Details

• The client needs to ask the naming service for a reference to a remote object.
• The client needs to know the hostname or IP address of the machine running the server.
• The client needs to know the name of the remote object.
• The naming service uses URIs to identify remote objects.

Server Detail – main()

• The server main() needs to:
  • create a remote object.
  • register the object with the Naming service.

public static void main(String args[]) {
  try {
    MyRemoteImpl r = new MyRemoteImpl();
    Naming.bind("joe",r);
  } catch (RemoteException e) {
    ...
  }
}

Using The Naming service

• Naming.lookup() method takes a string parameter that holds a URI indicating the remote object to lookup.
  rmi://hostname/objectname
• Naming.lookup() returns an Object!
• Naming.lookup() can throw
  • RemoteException
  • MalformedURLException
try {
    Object o = Naming.lookup("rmi://algol.cs.virginia.edu/joe");

    MyRemote r = (MyRemote) o;
    // . . . Use like any other Java object!
} catch (RemoteException re) {
    . . .
} catch (MalformedURLException up) {
    throw up;
}