03 – Remote invocation

- Request-reply
- RPC
- RMI

- Coulouris 5
- Birrel_Nelson_84.pdf

Remote Procedure Call

- Request Reply protocols
  - Not transparent (w.r.t. distribution)
  - Local invocation ≠ remote invocation
- Extends procedure call abstraction
  - RPCs are called as if in local address space
  - RPC system hides
    - Distribution
    - Encoding/decoding of parameters/reply
    - Passing of messages
    - Semantics of call

Remote Procedure Call

Interfaces

- RPCs should mimic programming language structures
  - Modules/ libraries / classes
  - Function/procedures/methods
  - Use the concept of interface
    - A set of well defined ointeraction points
  - Encapsulate implementation details
- Service interfaces
  - Set of procedures offered by a server
  - Defining parameters and return values (and types)

Remote Procedure Call

Interfaces

- Provide/implement good programming practices
  - Implementation are hidden from the programmer
  - Server programming language /architecture is hidden
  - SW evolution is possible
- Distribution imposes
  - Impossible access to remote variables
  - Call by reference not suitable
    - Input parameters: client → server transfer
    - Output parameters client ← server transfer
  - Addresses in one process are not valid on another
    - Addresses can not be passed

Remote Procedure Call

Interface Definition Languages

- Static languages require the *a priori* declaration of functions and types
- Publication of a service requires its description
- Interface Definition Languages (IDL) allow:
  - Specification of interfaces
  - Communication of interfaces
  - Production of .c/.h files

- Examples
  - SUN XDR,
  - CORBA IDL
  - Webservices WSDL
  - Java interfaces
  -

Remote Procedure Call

RPC call semantics

- DoOperation can be implemented differently
  - Retry request message
    - Are retries made until reply is received?
  - Duplicate filtering
    - What happens when server receives duplicate message
  - Retransmission of results
    - Is history kept to prevent re-execution of operations?

Remote Procedure Call

RPC call semantics

Call semantics	Fault tolerance measures		
	Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply
Maybe	No	Not Applicable	Not Applicable
No fault tolerance measures applied Omission failures (message lost), crash failures (server fails)			
At-least-once	Yes	No	Re-execute procedure
Invoker receives a result (code was executed at least once) or an exception (result was not received) Retransmission of requests Crash failures (server fails), arbitrary failures (requests are retransmitted, procedure may be executed more than once)			
At-most-once	Yes	Yes	Retransmit reply
Invoker receives a result (code was executed once) or an exception (result was not received, but code not executed or executed once)			

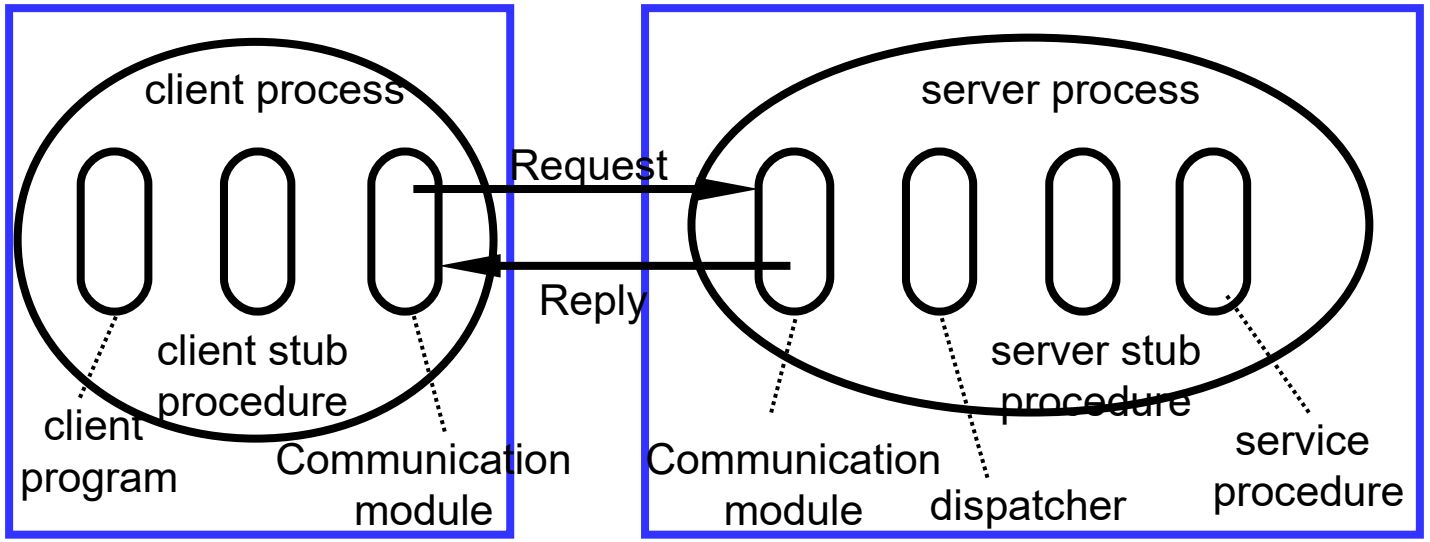
Remote Procedure Call

Transparency

Minimum requirement

- RPC tries to hide differences from local/remote procedure call
  - Syntax is similar
  - Data marshaling and retransmission is hidden
- However RPC are more vulnerable to failure
  - Involve network, other computer, other process
  - Code should be able to recover
- RPC latency is higher
- Parameter passing is different

Remote Procedure Call
Implementation



- Generated automatically
  - Stubs
    - Marshal arguments /return values
  - Dispatcher
    - Redirects call
- General
  - Communication module
- Hand coded
  - Interface definition
  - Client program
  - Service procedure

Centralized application

```
#include <stdio.h>
int add(int a, int b) {
   return a+b;
}
int main(int argc, char* argv[]) {
int i1 = 1;
int i2 = 2;
printf("\nThe sum of %i and %i is "%i.\n",
   i1, i2, add(i1, i2));
}
```

RPC

- Interfaces defined in Remote Procedure Call Language
  - Definition of types
  - Definitions of calls
  - Definition of versions
  - C like

Remote call interface (add_demo.x)

```
program ADDDEMO {
    version ADD {
        int ADD(int, int) = 1;
    } = 1;
} = 0x20004028;
```

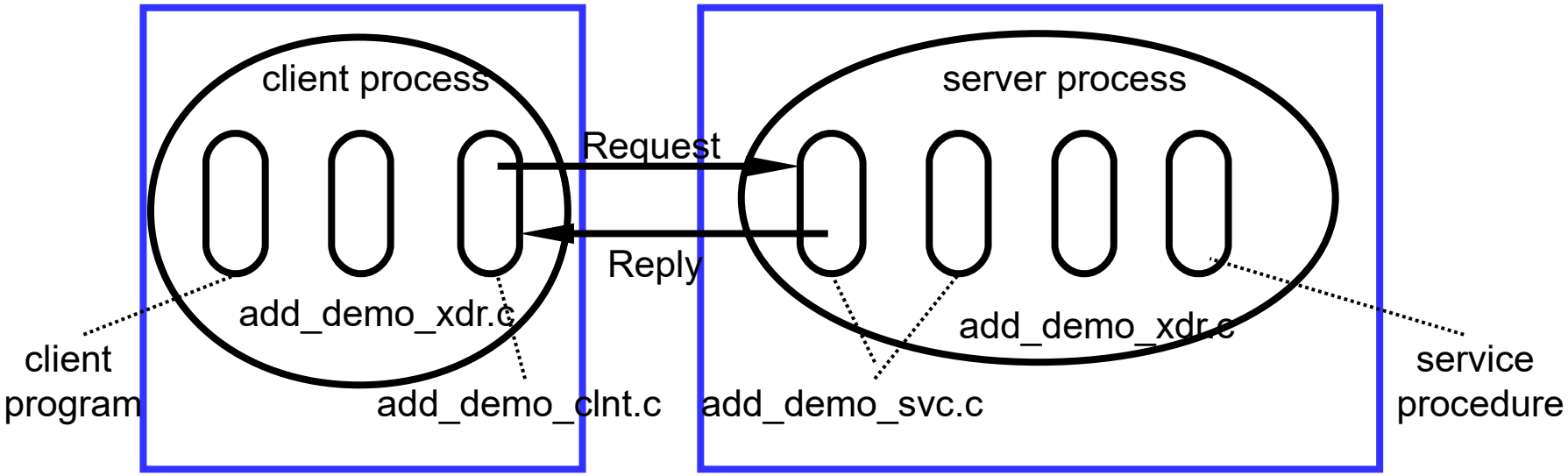
Unique on the each version
Procedure identifier identifier

Unique on the each program
Version identifier

Unique on the system
Program identifier

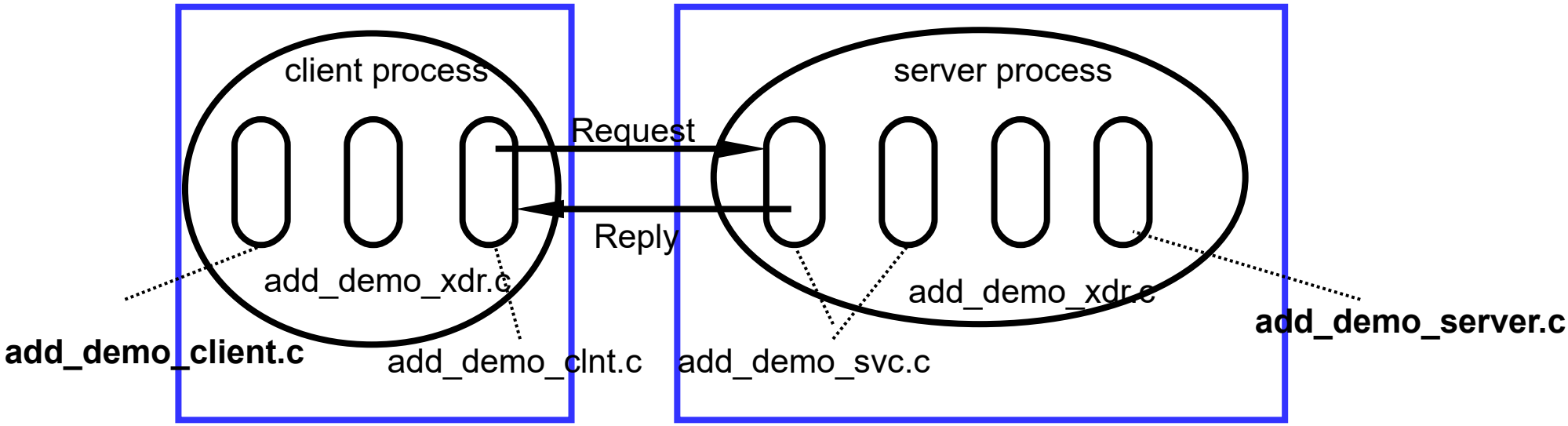
Generation of code

- rpcgen add_demo.x
  - add_demo.h
  - add_demo_clnt.c
  - add_demo_svc.c
  - add_demo_xdr.c



Generation of code

- rpcgen -a add_demo.x
  - add_demo.h
  - add_demo_clnt.c
  - add_demo_svc.c
  - add_demo_xdr.c
  - add_demo_client.c
  - add_demo_server.c



add_demo_client.c

- main

```
…
args x;
x.a = 1;
x.b = 2;
int res;
…
host = argv[1];
Res = adddemo_1(x,host);
…
```

- adddemo_1(args add_1_arg, char *host)

```
CLIENT *clnt;
int  *result_1;

clnt = clnt_create (host, ADDDEMO,
DEMO_1, "udp");
if (clnt == NULL) {
   clnt_pcreateerror (host);
   exit (1);
}
result_1 = add_1(&add_1_arg, clnt);
if (result_1 == (int *) NULL) {
   clnt_perror (clnt, "call failed");
}
clnt_destroy (clnt);
return * result_1;
```

add_demo_server.c

```
/*
 * This is sample code generated by rpcgen. These are only templates and you can use them
 * as a guideline for developing your own functions.
 */
#include "add_demo.h"

Int * add_1_svc(intpair *argp, struct svc_req *rqstp) {
    static int  result;
    /*
     * insert server code here
     */
    result = 12;
    return &result;
}
```

add_demo_server.c

- main

```
…
args x;
x.a = 1;
x.b = 2;
int res;
…
host = argv[1];
Res = adddemo_1(x,host);
…
```

- adddemo_1(args add_1_arg, char *host)

```
CLIENT *clnt;
int  *result_1;

clnt = clnt_create (host, ADDDEMO,
DEMO_1, "udp");
if (clnt == NULL) {
   clnt_pcreateerror (host);
   exit (1);
}
result_1 = add_1(&add_1_arg, clnt);
if (result_1 == (int *) NULL) {
   clnt_perror (clnt, "call failed");
}
clnt_destroy (clnt);
return * result_1;
```

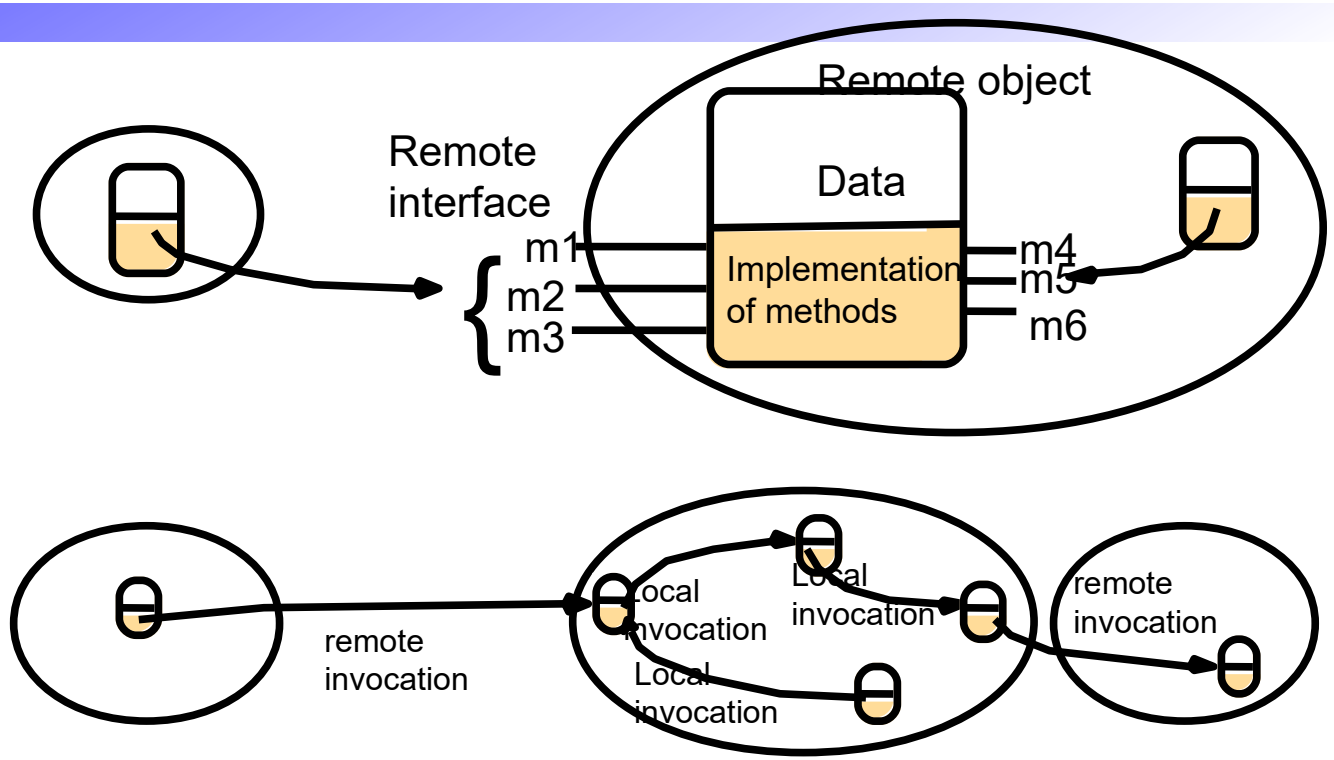
Remote Method Invocation

- Extends RPC to the OO world
- RPC and RMI
  - Support programming with interfaces
  - Built on top of Request-reply protocols
  - Offer similar semantics
  - Offer similar level of transparency
- RMI
  - Use OO mechanisms
    - Object, classes inheritance, OO development methods/tools
  - Provide concept of object identity
    - All objects have references (local or remote)
    - That can be globally accessed

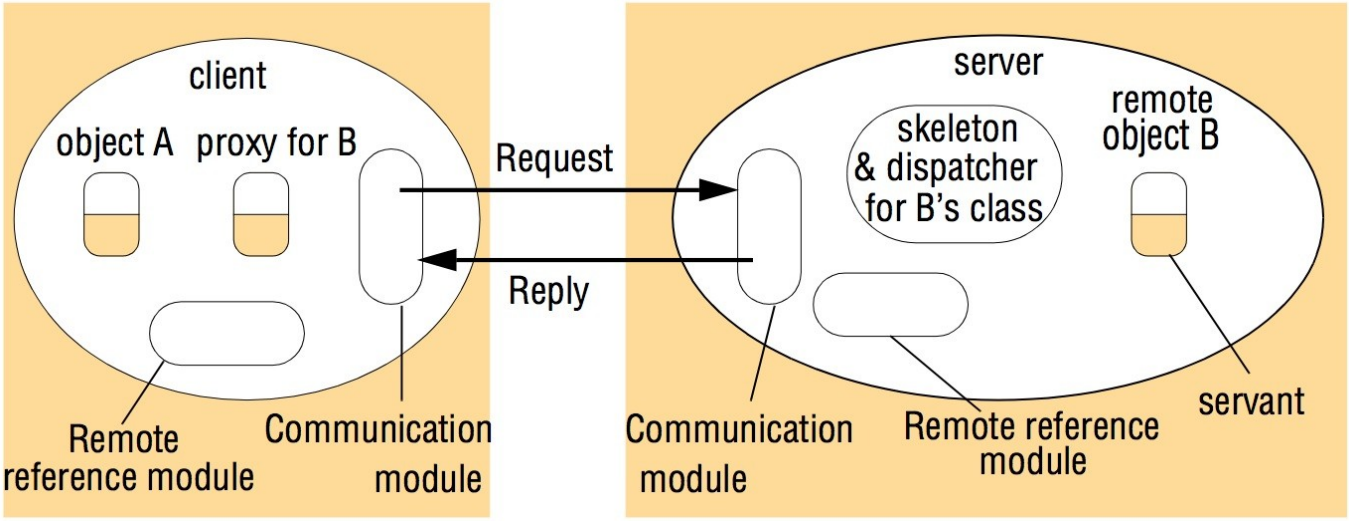
Remote Method Invocation

Object model

- Object references
  - remote
- Interfaces
  - remote
- Actions
  - Object creation
  - State change
  - Further invocations
- Exception
  - Can be generated and “returned”
- Garbage collection



Remote Method Invocation
Implementation



- Proxy
- Dispatcher
- Skeleton
- Static Generation
  - IDL+compiler (Similar to SUN RPC)
- Dynamic invocation
  - Language support for reflection
    - method interception (client)
    - Dynamic code loading

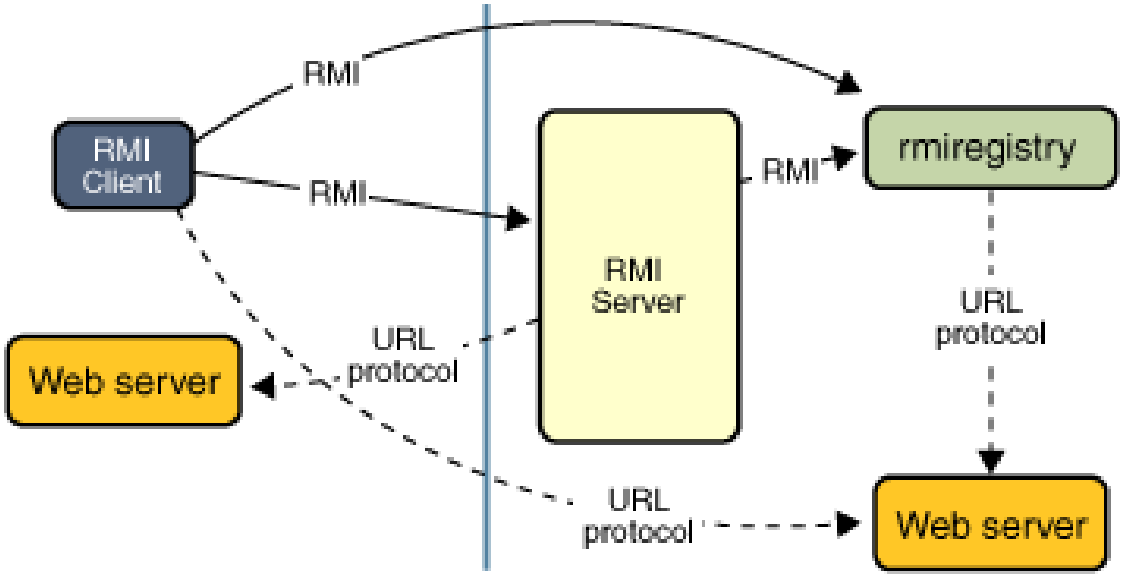
Remote Method Invocation

Execution

- https://www.linuxjournal.com/article/2204
- Server/client programs
- Binder/ns
  - Process that provides object reference from textual name
- Server threads
  - In order to allow concurrent method execution
  - Requires synchronization
- Activation
  - When are objects killed?

Java RMI

- Define interfaces
- Implement interface
- Register server
- Develop client
- Connect to server



http://docs.oracle.com/javase/tutorial/rmi/overview.html

Design Remote interface

- Define interface
  - Import java.rmi.Remote
  - Extend Remote

```
import java.rmi.*;

public interface AddInterface extends Remote {

  public int add(int a, int b) throws
RemoteException;

}
```

Implement Remote Interface

- Implement pre-defined interface (AddInterface)
  - Implement constructor
  - Implement interface methods

```
import java.rmi.*;
import java.rmi.server.*;
public class Adder extends UnicastRemoteObject implements AddInterface {
  private int counter ;
  public Adder () throws RemoteException {
    counter = 0;
  }
  public int add(int a, int b) throws RemoteException {
    System.out.println ("Called: " + counter++);
    return a+b;
  }
}
```

Compilation

- Compile Interface implementation
  - javac Adder
    - AddInterface.class
    - Adder.class
- Create Stubs
  - rmic Adder
    - Adder_Stub.class
  -

Old Java

Programming the server

- Creates a server object (Adder)
- Binds it to a well know name

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

public class AddServer{
  public static void main (String[] argv) {
    try {
      Naming.rebind ("Adder", new Adder ());
      System.out.println ("Add Server is ready.");
    } catch (Exception e) {
      System.out.println ("Add Server failed: " + e);
    }
  }
}
```

Running the server

- Create the rmiregistry
  - Rmiregistry &
- Execute the server code
  - java AddServer
  - Accesses
    - AddInterface.class
    - AddServer.class
    - Adder.class
    - Adder_Stub.class
  - Registers the server object on the registry
  - Waits for accesses

Programming the client

- Declare a reference to the server interface
  - AddInterface a
- Create a proxy for the remote object
  - naming.lookup ("//localhost/Adder");
- Call the remote method

```
import java.rmi.*;
public class AddClient{
  public static void main (String[] argv) {
    try {
      AddInterface a = (AddInterface) Naming.lookup ("//localhost/Adder");
      System.out.println (a.add(1,56));
    } catch (Exception e) {
      System.out.println ("AddClient exception: " + e);
    }
  }
}
```

Running the client

- Access the registry
  - Receives informatrion about server
- Server proxy creation
    - That implements AddInterface
- Methods call
    - Handled by the proxy
    - Redirected to the remote object
- Accesses
  - AddClient.class
  - AddInterface.class
  - Adder_Stub.class

What is going on under the hood?

- Special compiler will generate, based on code from previous slide, new stub code for handling communication
- Application programmers do not have to look at stub code
- Stub code provides "glue" that:
- packages information about function and parameters
- sends to server
- handles message on the server side, calling appropriate function
- inverse process for the reply

Marshaling

- When sending a message, it is necessary to marshal the arguments, and then unmarshal upon receiving it
- Valid for both request and reply messages
- Marshaling consists of:
  - Serializing objects
  - Converting to common format

RPC: Challenges

- Ideally, RPC calls should be identical to a local procedure call (i.e., transparent)
- In many ways, RPCs are inherently different
- In particular, RPCs must handle:
  - Failures of client or server
  - Concurrency
  - Lack of a common address space
  - Noticeable performance difference

Handling failures

- In a local procedure call, both caller and callee have "fate sharing"
- Either both work or both fail simultaneously
- In a remote procedure call there are several things that can go wrong:
  - client fails
  - server fails
  - network is unreliable
  - retrying raises questions on what effects are

Handling concurrency

- Clients are inherently concurrent
- Server code may also handle functions concurrently
- Must synchronize access to state

Lack of a common address space

- How to implement passing by reference?
- There is no common address space between client and server
- However, marshaling can simulate an approximation