CC755: Distributed and Parallel Systems

Dr. Manal Helal, Spring 2016

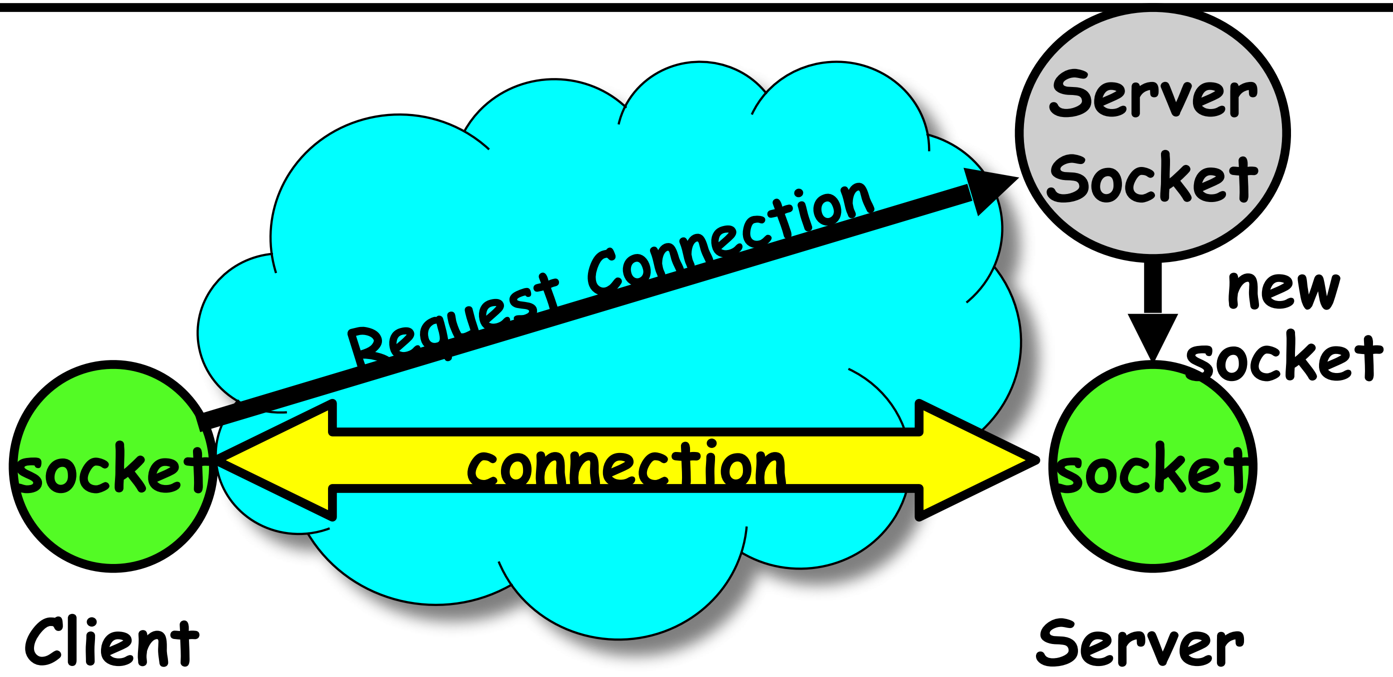
moodle.manalhelal.com

Lecture 6: Socket Programming
Remote Procedure Calls (RPC)

Use of TCP: Sockets

- **Socket:** an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation at time
    - » Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
  - On server: set up "server-socket"
    - » Create socket, Bind to protocol (TCP), local address, port
    - » Call listen(): tells server socket to accept incoming requests
    - » Perform multiple accept() calls on socket to accept incoming connection request
    - » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    - » Create socket, Bind to protocol (TCP), remote address, port
    - » Perform connect() on socket to make connection
    - » If connect() successful, have socket connected to server

Socket Setup (Con't)



- Things to remember:
  - Connection requires 5 values:
    [Src Addr, Src Port, Dst Addr, Dst Port, Protocol]
  - Often, Src Port "randomly" assigned
    - » Done by OS during client socket setup
  - Dst Port often "well known"
    - » 80 (web), 443 (secure web), 25 (sendmail), etc
    - » Well-known ports from 0—1023

C Example:

```
/* A simple server in the internet domain using
TCP The port number is passed as an argument */

int main(int argc, char *argv[]) {
int sockfd, newsockfd, portno, n;
socklen_t clilen;char buffer[256];
struct sockaddr_in serv_addr, cli_addr;
if (argc < 2) {
fprintf(stderr,"ERROR, no port provided\n");
exit(1); }
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)error("ERROR opening socket");
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = atoi(argv[1]);
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
if (bind(sockfd, (struct sockaddr *)
    &serv_addr, sizeof(serv_addr)) < 0)
    error("ERROR on binding");
listen(sockfd,5);
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd,
  (struct sockaddr *) &cli_addr, &clilen);
if (newsockfd < 0) error("ERROR on accept");
bzero(buffer,256);
n = read(newsockfd,buffer,255);
if (n < 0) error("ERROR reading from socket");
printf("Here is the message: %s\n",buffer);
n = write(newsockfd,"I got your message",18);
if (n < 0) error("ERROR writing to socket");
close(newsockfd);   close(sockfd);
return 0;
}
```

```
/* Corresponding client code*/

int main(int argc, char *argv[]) {
int sockfd, portno, n;
struct sockaddr_in serv_addr;
struct hostent *server;
char buffer[256];
if (argc < 3) {
  fprintf(stderr,"usage %s hostname port\n",
     argv[0]);
  exit(0);}
portno = atoi(argv[2]);
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0) error("ERROR opening socket");
server = gethostbyname(argv[1]);
if (server == NULL) {
   fprintf(stderr,"ERROR, no such host\n");
   exit(0);}
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
(char *)&serv_addr.sin_addr.s_addr,
server->h_length);
serv_addr.sin_port = htons(portno);
if (connect(sockfd,(struct sockaddr *)
   &serv_addr,sizeof(serv_addr)) < 0)
   error("ERROR connecting");
printf("Please enter the message: ");
bzero(buffer,256); fgets(buffer,255,stdin);
n = write(sockfd,buffer,strlen(buffer));
if (n < 0)  error("ERROR writing to socket");
bzero(buffer,256);
n = read(sockfd,buffer,255);
if (n < 0) error("ERROR reading from socket");
printf("%s\n",buffer);close(sockfd);return 0;
}
```

Java Socket Programming

Y Daniel Liang, Introduction to JAVA Programming, 10th Edition, Prentice Hall, 2013. (http://www.cs.armstrong.edu/liang/intro10e/)

- Chapter 31 in the 10th Edition

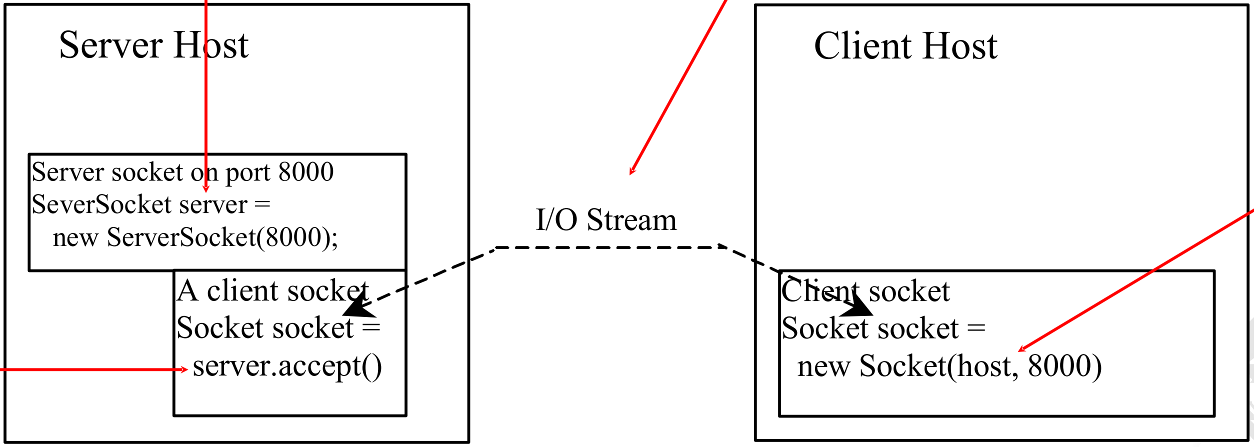
Objectives

- To explain terms: TCP, IP, domain name, domain name server, stream-based communications, and packet-based communications (§31.2).
- To create servers using server sockets (§31.2.1) and clients using client sockets (§31.2.2).
- To implement Java networking programs using stream sockets (§31.2.3).
- To develop an example of a client/server application (§31.2.4).
- To obtain Internet addresses using the **InetAddress** class (§31.3).
- To develop servers for multiple clients (§31.4).
- To send and receive objects on a network (§31.5).
- To develop an interactive tic-tac-toe game played on the Internet (§31.6).

Client/Server Communications

The server must be running when a client starts. The server waits for a connection request from a client. To establish a server, you need to create a server socket and attach it to a port, which is where the server listens for connections.

After the server accepts the connection, communication between server and client is conducted the same as for I/O streams.



```
SeverSocket server =
  new ServerSocket(8000);
```

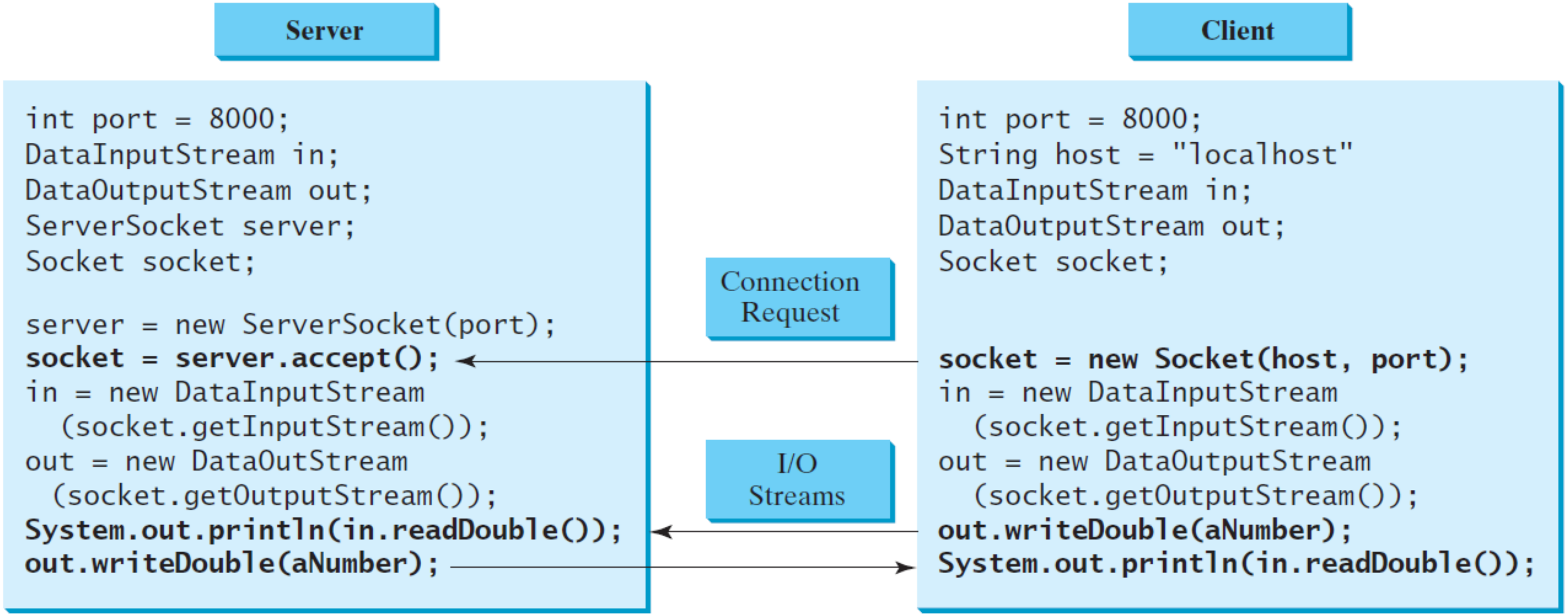
```
Socket socket =
  server.accept()
```

```
Socket socket =
  new Socket(host, 8000)
```

After a server socket is created, the server can use this statement to listen for connections.

The client issues this statement to request a connection to a server.

Data Transmission through Sockets



Server

```
int port = 8000;
DataInputStream in;
DataOutputStream out;
ServerSocket server;
Socket socket;

server = new ServerSocket(port);
socket = server.accept();
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutStream
  (socket.getOutputStream());
System.out.println(in.readDouble());
out.writeDouble(aNumber);
```

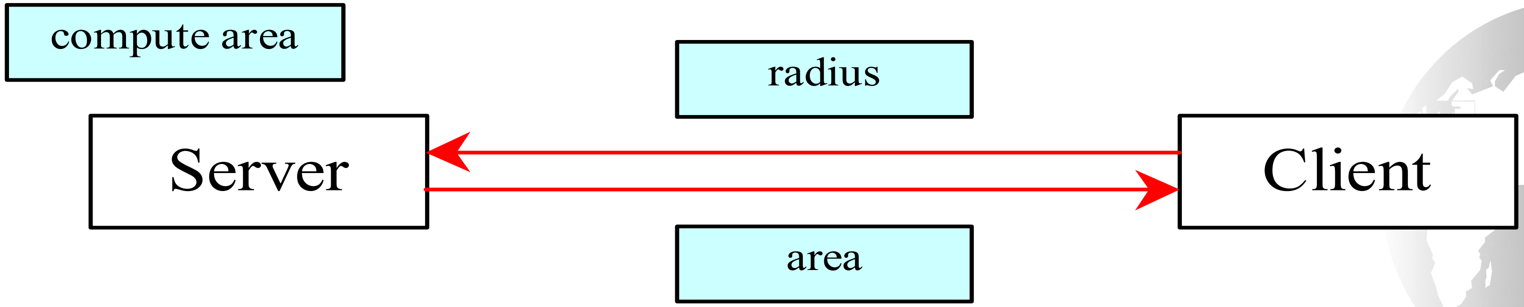
Client

```
int port = 8000;
String host = "localhost"
DataInputStream in;
DataOutputStream out;
Socket socket;


socket = new Socket(host, port);
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutputStream
  (socket.getOutputStream());
out.writeDouble(aNumber);
System.out.println(in.readDouble());
```

```
InputStream input = socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

A Client/Server Example

- Problem: Write a client to send data to a server. The server receives the data, uses it to produce a result, and then sends the result back to the client. The client displays the result on the console. In this example, the data sent from the client is the radius of a circle, and the result produced by the server is the area of the circle.


compute area

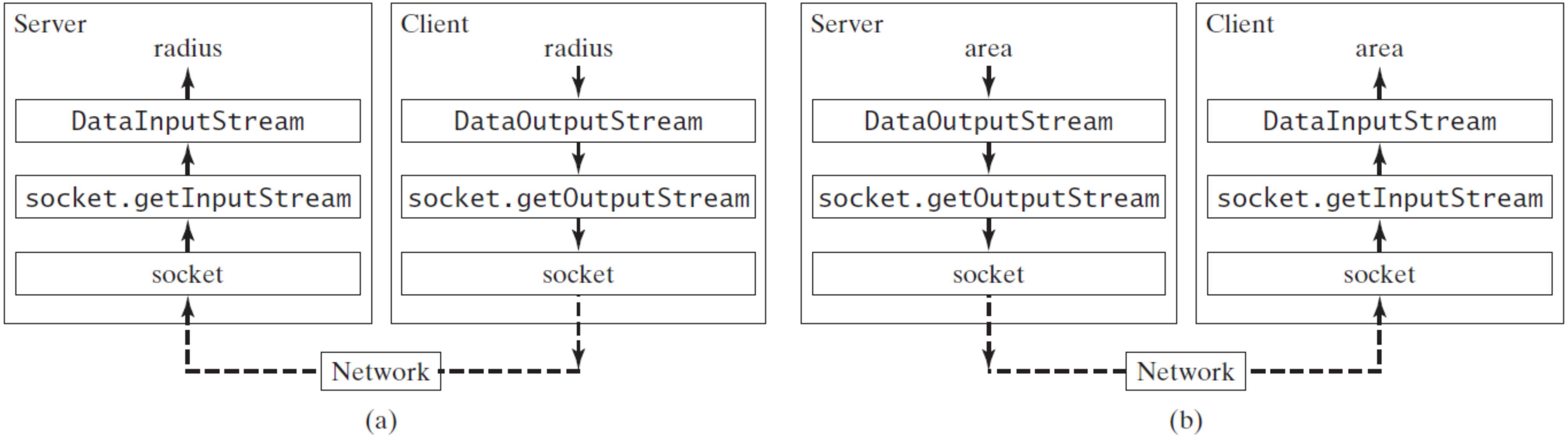
Server

radius

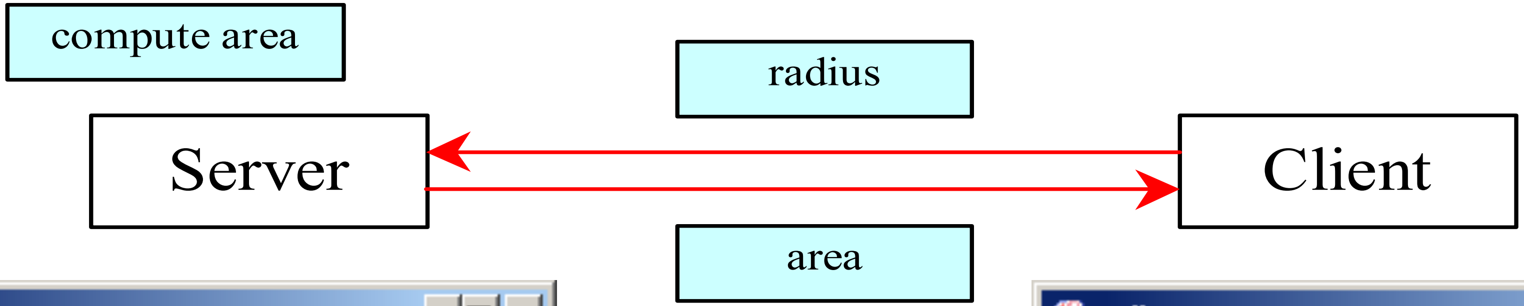
area

Client

A Client/Server Example, cont.



A Client/Server Example, cont.



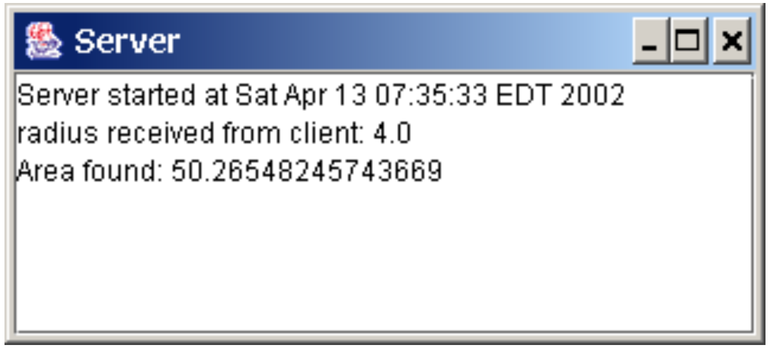
compute area

radius

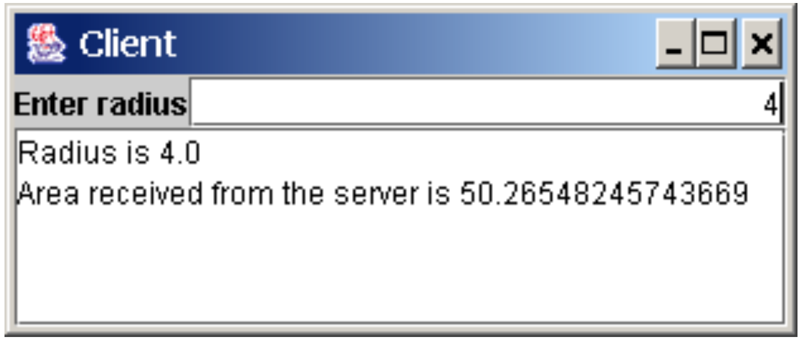
Server

Client

area



```
Server started at Sat Apr 13 07:35:33 EDT 2002
radius received from client: 4.0
Area found: 50.26548245743669
```



Enter radius: 4

```
Radius is 4.0
Area received from the server is 50.26548245743669
```

Server Code

Client Code

Start Server

Start Client

Note: Start the server, then the client.

The InetAddress Class

Occasionally, you would like to know who is connecting to the server. You can use the InetAddress class to find the client's host name and IP address. The InetAddress class models an IP address. You can use the statement shown below to create an instance of InetAddress for the client on a socket.

```
InetAddress inetAddress = socket.getInetAddress();
```

Next, you can display the client's host name and IP address, as follows:

```
System.out.println("Client's host name is " +
  inetAddress.getHostName());
System.out.println("Client's IP Address is " +
  inetAddress.getHostAddress());
```

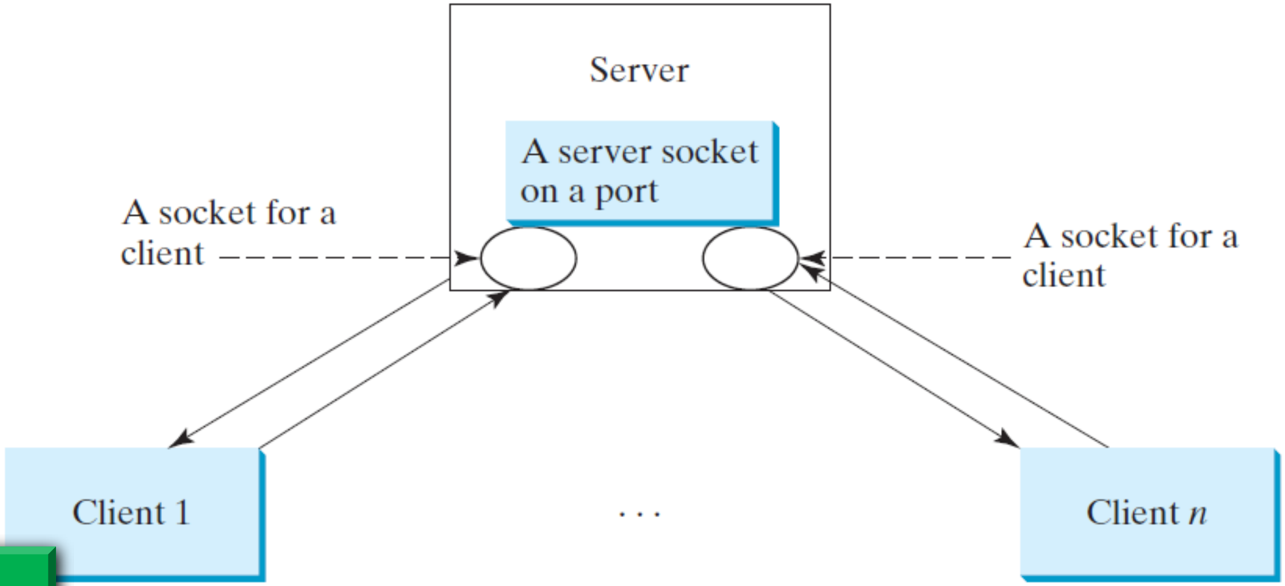
Serving Multiple Clients

Multiple clients are quite often connected to a single server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet may want to connect to it. You can use threads to handle the server's multiple clients simultaneously. Simply create a thread for each connection. Here is how the server handles the establishment of a connection:

```
while (true) {
  Socket socket = serverSocket.accept();
  Thread thread = new ThreadClass(socket);
  thread.start();
}
```

The server socket can have many connections. Each iteration of the while loop creates a new connection. Whenever a connection is established, a new thread is created to handle communication between the server and the new client; and this allows multiple connections to run at the same time.

Example: Serving Multiple Clients



Server for Multiple Clients

Start Server

Start Client

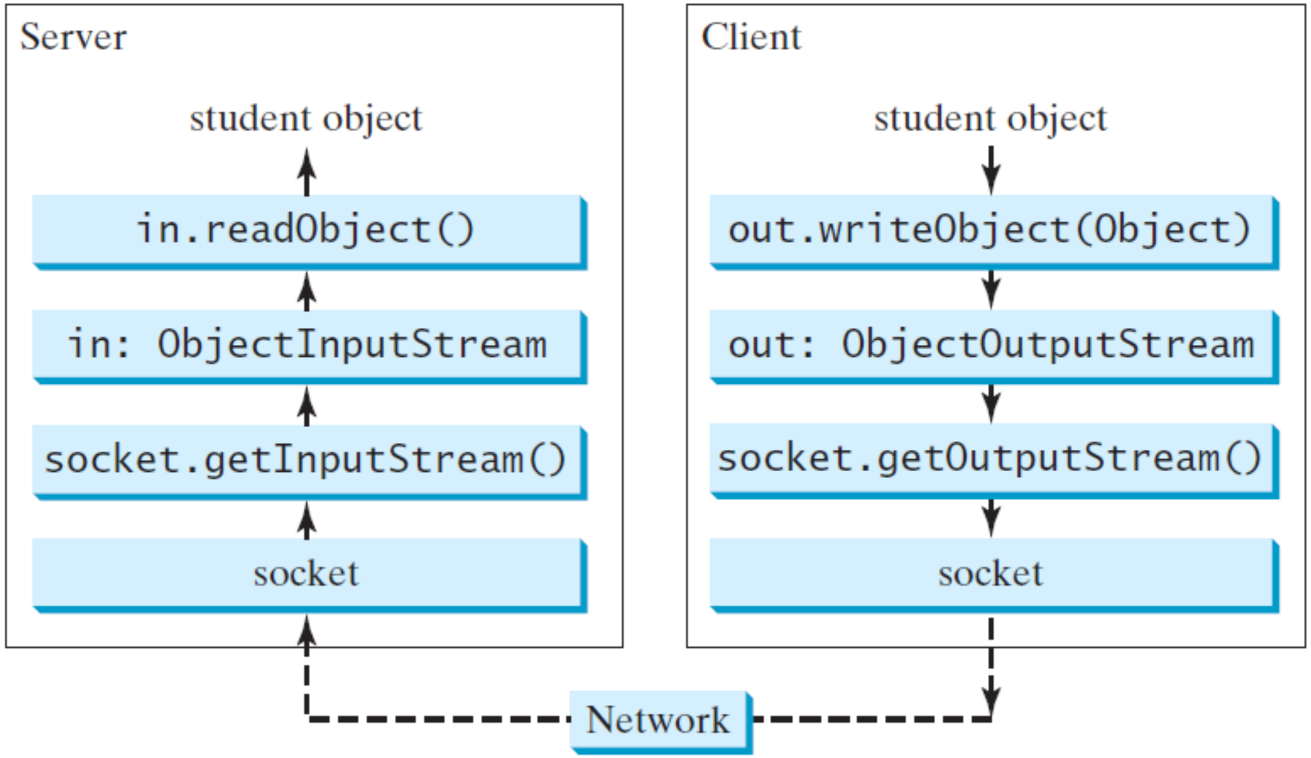
Note: Start the server first, then start multiple clients.

```
 try {
// Create a server socket
ServerSocket serverSocket = new ServerSocket(8000);
ta.appendText("MultiThreadServer started at "
  + new Date() + '\n');
while (true) {
  // Listen for a new connection request
  Socket socket = serverSocket.accept();

  // Increment clientNo
  clientNo++;

  Platform.runLater( () -> {
// Display the client number
ta.appendText("Starting thread for client " + clientNo +
  " at " + new Date() + '\n');

// Find the client's host name, and IP address
InetAddress inetAddress = socket.getInetAddress();
ta.appendText("Client " + clientNo + "'s host name is "
  + inetAddress.getHostName() + "\n");
ta.appendText("Client " + clientNo + "'s IP Address is "
  + inetAddress.getHostAddress() + "\n");
  });

  // Create and start a new thread for the connection
  new Thread(new HandleAClient(socket)).start();
}
  }
  catch(IOException ex) {
System.err.println(ex);
  }
```

```
  try {

  // Create a socket to connect to the server
  Socket socket = new Socket("localhost", 8000);
  // Socket socket = new Socket("130.254.204.36", 8000);
  // Socket socket = new Socket("drake.Armstrong.edu", 8000);

  // Create an input stream to receive data from the server
DataInputStream fromServer = new DataInputStream
                  (socket.getInputStream());

  // Create an output stream to send data to the server
  DataOutputStream toServer = new DataOutputStream
                  (socket.getOutputStream());

        double radius =  in.nextDouble()
// Send the radius to the server
toServer.writeDouble(radius);
toServer.flush();

// Get area from the server
double area = fromServer.readDouble();

// Display to the text area
ta.appendText("Radius is " + radius + "\n");
ta.appendText("Area received from the server is "
  + area + '\n');
}
  catch (IOException ex) {
  ta.appendText(ex.toString() + '\n');
}
```

Example: Passing Objects in Network Programs

Write a program that collects student information from a client and send them to a server. Passing student information in an object.



Server

student object

in.readObject()

in: ObjectInputStream

socket.getInputStream()

socket

Client

student object

out.writeObject(Object)

out: ObjectOutputStream

socket.getOutputStream()

socket

Network

Student Class

Student Sever

Student Client

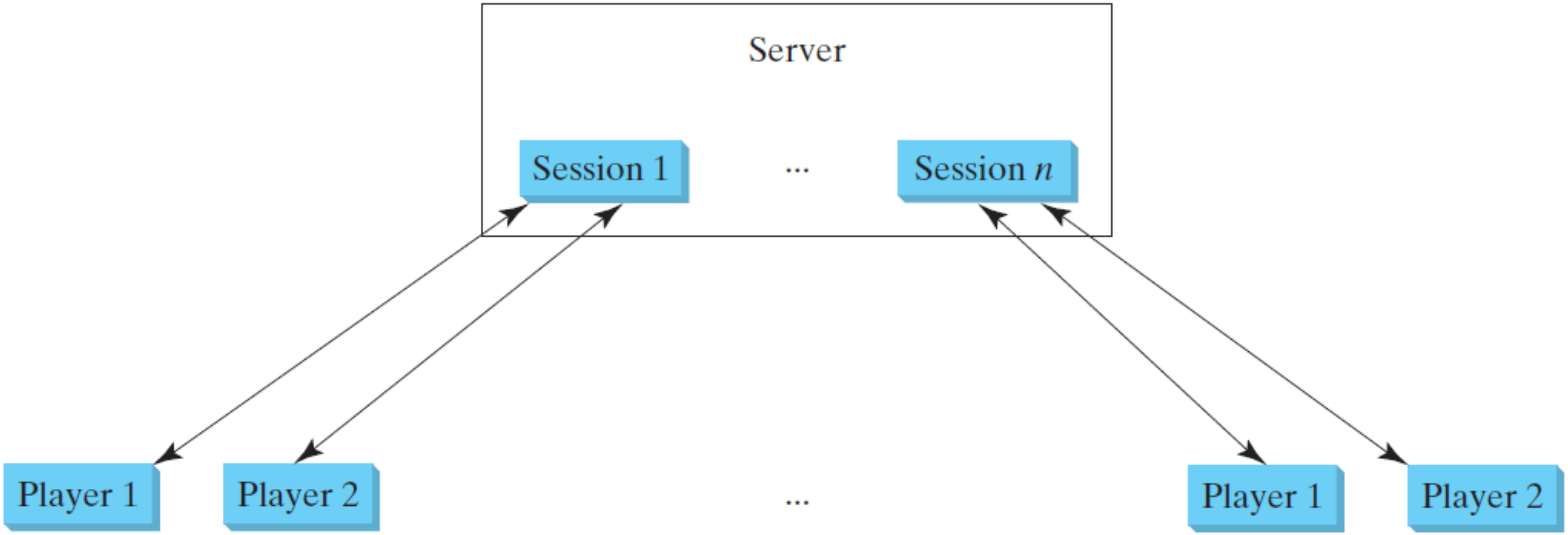
Start Server

Start Client

Note: Start the server first, then the client.

Optional

Case Studies: Distributed TicTacToe Games



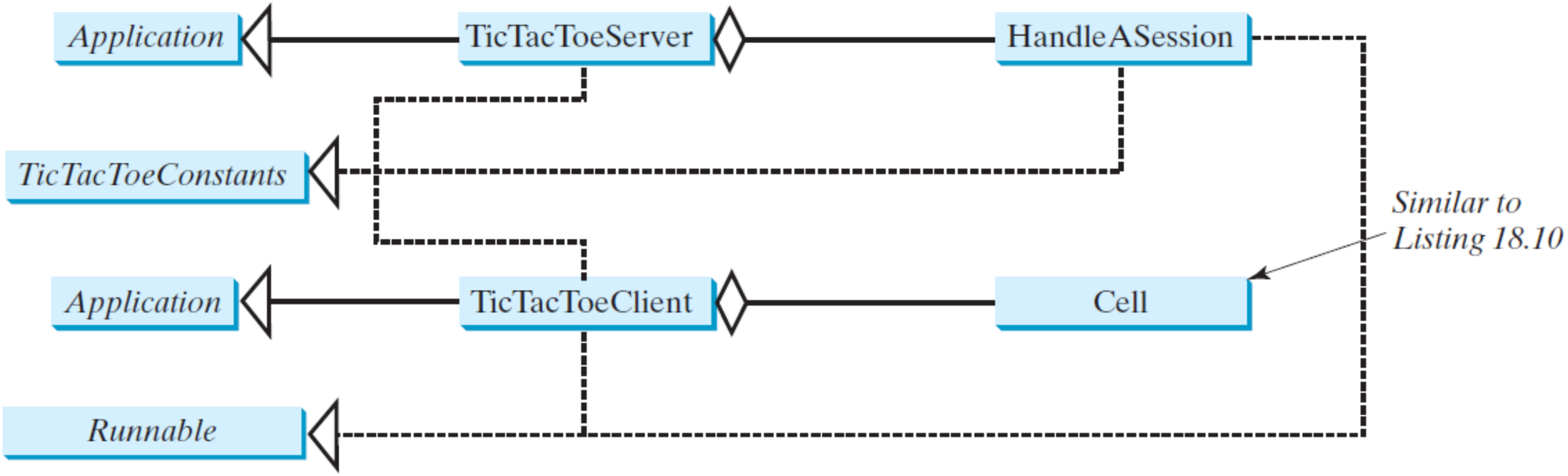
TicTacToeServer

Run Server

TicTacToeClient

Run Client

Distributed TicTacToe, cont.



TicTacToeServer

```
start(primaryStage: Stage):
  void
```

«interface»
TicTacToeConstants

```
+PLAYER1 = 1: int
+PLAYER2 = 2: int
+PLAYER1_WON = 1: int
+PLAYER2_WON = 2: int
+DRAW = 3: int
+CONTINUE = 4: int
```

HandleASession

```
-player1: Socket
-player2: Socket
-cell: char[][]
-continueToPlay: boolean
```

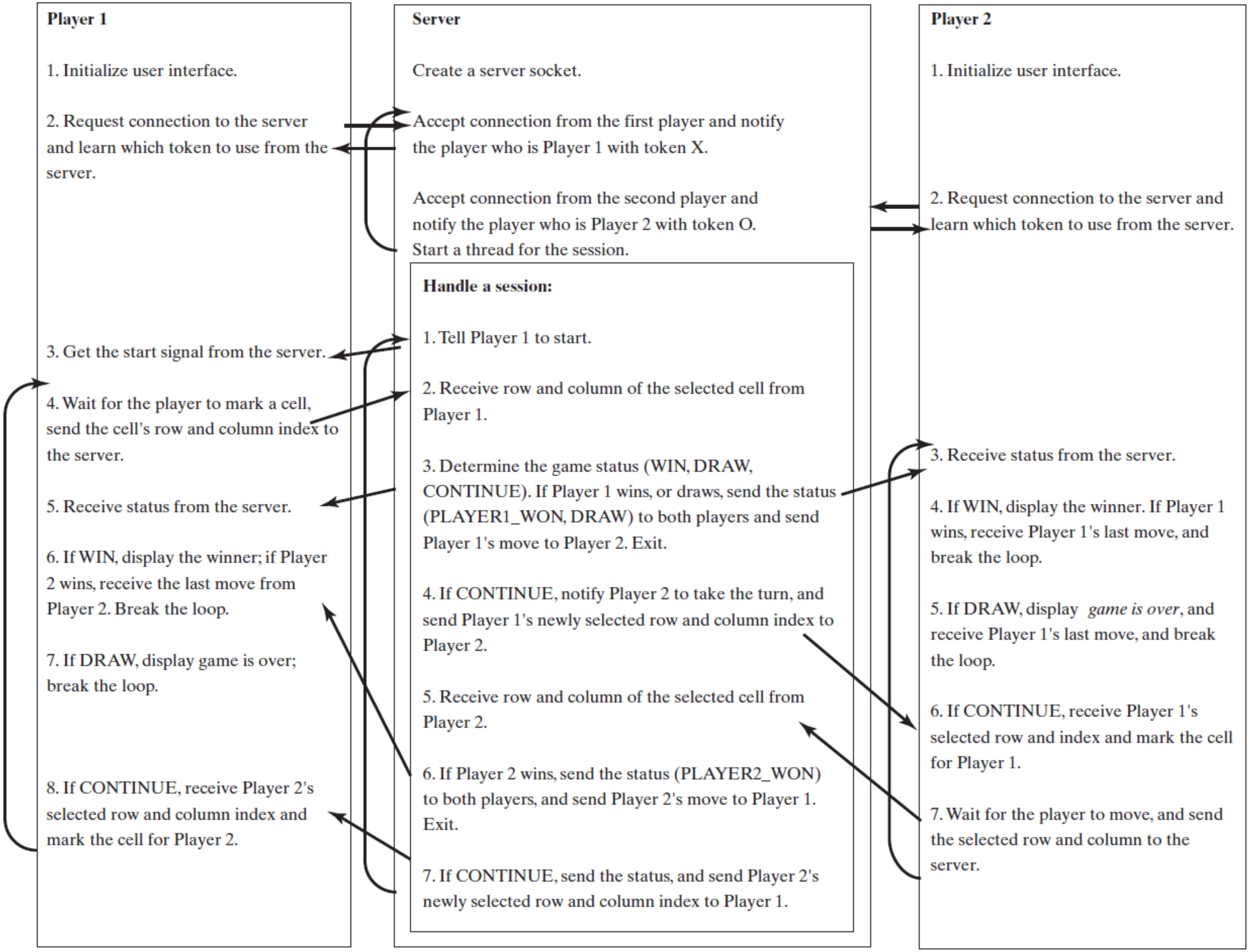
```
+run(): void
-isWon(): boolean
-isFull(): boolean
-sendMove(out:
  DataOutputStream, row: int,
  column: int): void
```

TicTacToeClient

```
-myTurn: boolean
-myToken: char
-otherToken: char
-cell: Cell[][]
-continueToPlay: boolean
-rowSelected: int
-columnSelected: int
-fromServer: DataInputStream
-toServer: DataOutputStream
-waiting: boolean
```

```
+run(): void
-connectToServer(): void
-receiveMove(): void
-sendMove(): void
-receiveInfoFromServer(): void
-waitForPlayerAction(): void
```

Distributed TicTacToe Game



Player 1

1. Initialize user interface.

2. Request connection to the server and learn which token to use from the server.

3. Get the start signal from the server.

4. Wait for the player to mark a cell, send the cell's row and column index to the server.

5. Receive status from the server.

6. If WIN, display the winner; if Player 2 wins, receive the last move from Player 2. Break the loop.

7. If DRAW, display game is over; break the loop.

8. If CONTINUE, receive Player 2's selected row and column index and mark the cell for Player 2.

Server

Create a server socket.

Accept connection from the first player and notify the player who is Player 1 with token X.

Accept connection from the second player and notify the player who is Player 2 with token O. Start a thread for the session.

Handle a session:

1. Tell Player 1 to start.

2. Receive row and column of the selected cell from Player 1.

3. Determine the game status (WIN, DRAW, CONTINUE). If Player 1 wins, or draws, send the status (PLAYER1_WON, DRAW) to both players and send Player 1's move to Player 2. Exit.

4. If CONTINUE, notify Player 2 to take the turn, and send Player 1's newly selected row and column index to Player 2.

5. Receive row and column of the selected cell from Player 2.

6. If Player 2 wins, send the status (PLAYER2_WON) to both players, and send Player 2's move to Player 1. Exit.

7. If CONTINUE, send the status, and send Player 2's newly selected row and column index to Player 1.

Player 2

1. Initialize user interface.

2. Request connection to the server and learn which token to use from the server.

3. Receive status from the server.

4. If WIN, display the winner. If Player 1 wins, receive Player 1's last move, and break the loop.

5. If DRAW, display *game is over*, and receive Player 1's last move, and break the loop.

6. If CONTINUE, receive Player 1's selected row and index and mark the cell for Player 1.

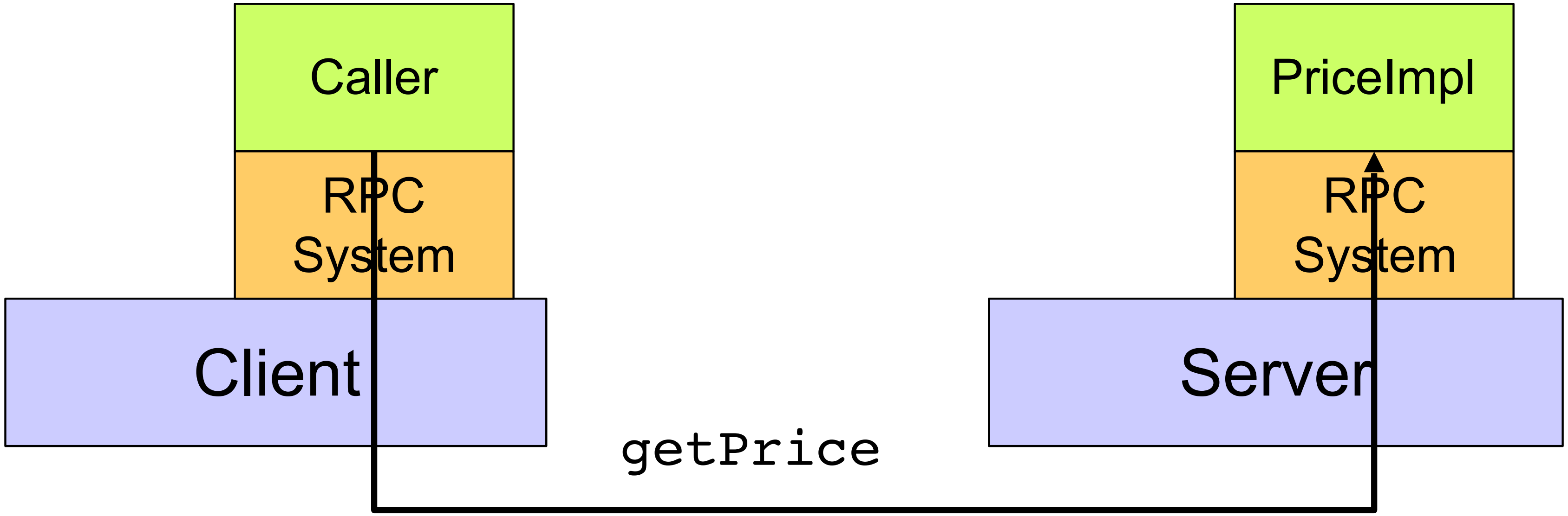
7. Wait for the player to move, and send the selected row and column to the server.

RPC

Remote Procedure Call

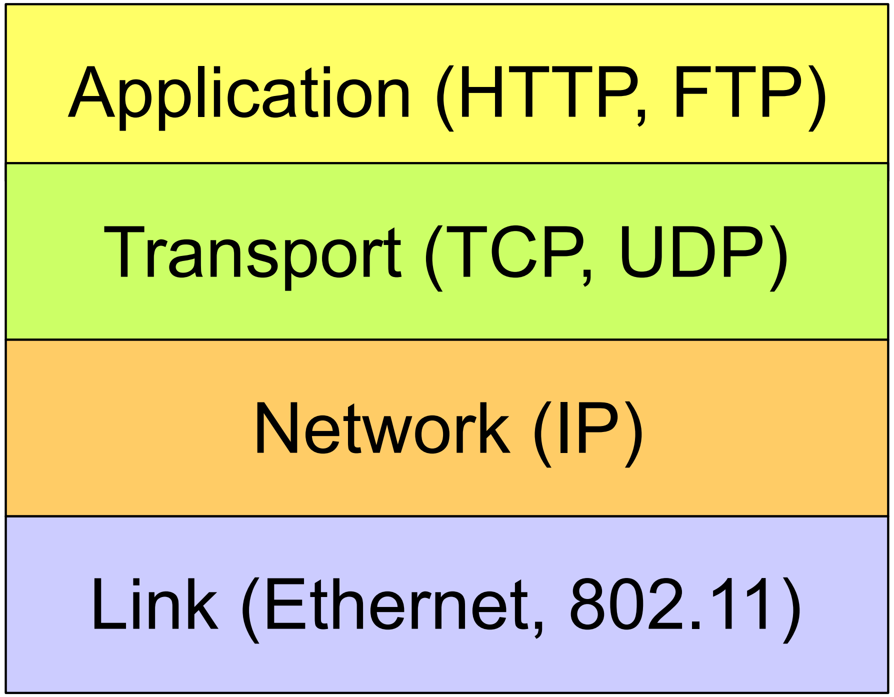
- RPC exposes a programming interface *across* machines:

```
interface PriceService {
   float getPrice(ASIN uniqueID);
}
```



Networking in two slides

- Network software is arranged in layers
- Higher layers offer more convenient programming abstractions
  - TCP provides in-order, reliable delivery of a byte stream



Application (HTTP, FTP)

Transport (TCP, UDP)

Network (IP)

Link (Ethernet, 802.11)

What TCP Does (Not) Provide

- TCP allows one machine to send a reliable byte stream to another machine:
  - Socket.send(byte[] byteBuffer);
- TCP does not provide:
  - Mapping to/from programming language types
    - Called “marshalling”
  - Thread management
  - Intelligent failure semantics
  - Discovery
- RCP packages build on TCP (or sometimes UDP) to provide these services

Remote Procedure Call (RPC)

- *The* most common framework for newer protocols and for middleware
- Used both by operating systems and by applications
  - NFS is implemented as a set of RPCs
  - DCOM, CORBA, Java RMI, etc., are just RPC systems

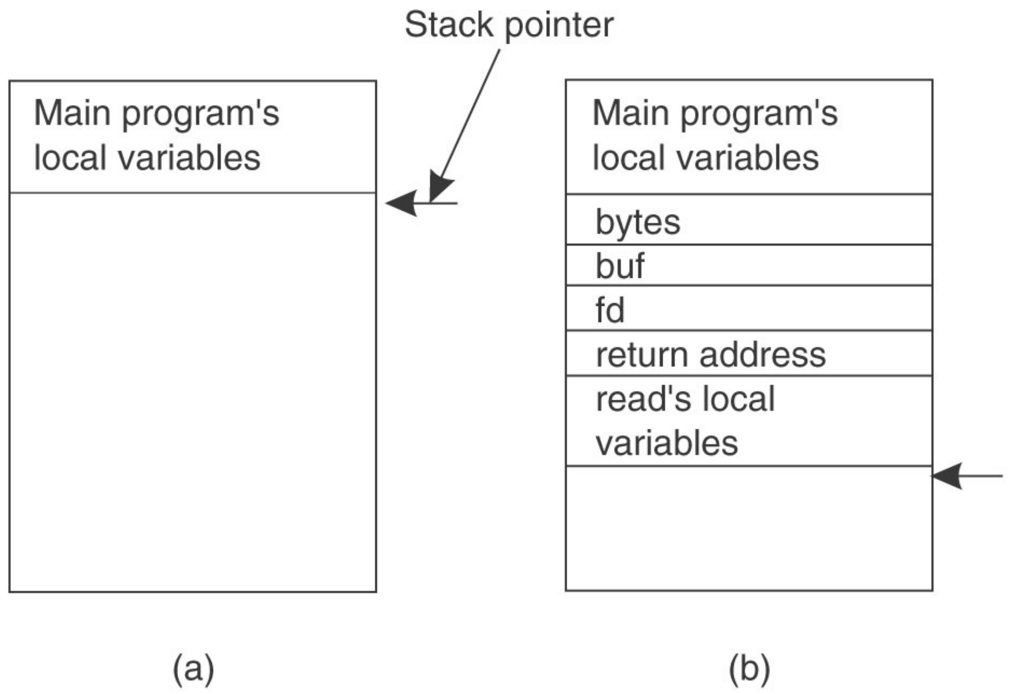
- Reference
  - Birrell, Andrew D., and Nelson, Bruce, “Implementing Remote Procedure Calls,” *ACM Transactions on Computer Systems,* vol. 2, #1, February 1984, pp 39-59. (.pdf)

Remote Procedure Call (RPC)

- Fundamental idea: –
  - Server process exports an *interface* of procedures or functions that can be called by client programs
    - similar to library API, class definitions, etc.
- Clients make local procedure/function calls
  - *As if* directly linked with the server process
  - Under the covers, procedure/function call is converted into a message exchange with remote server process

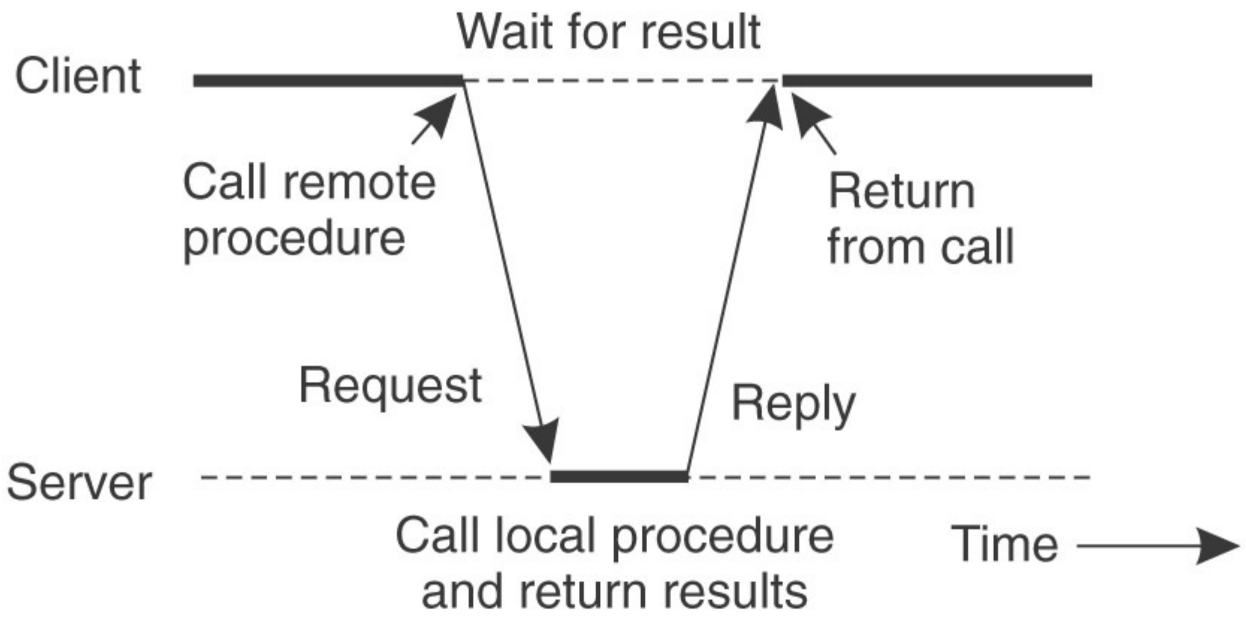
Ordinary procedure/function call

```
count = read(fd, buf, bytes)
```



Remote Procedure Call

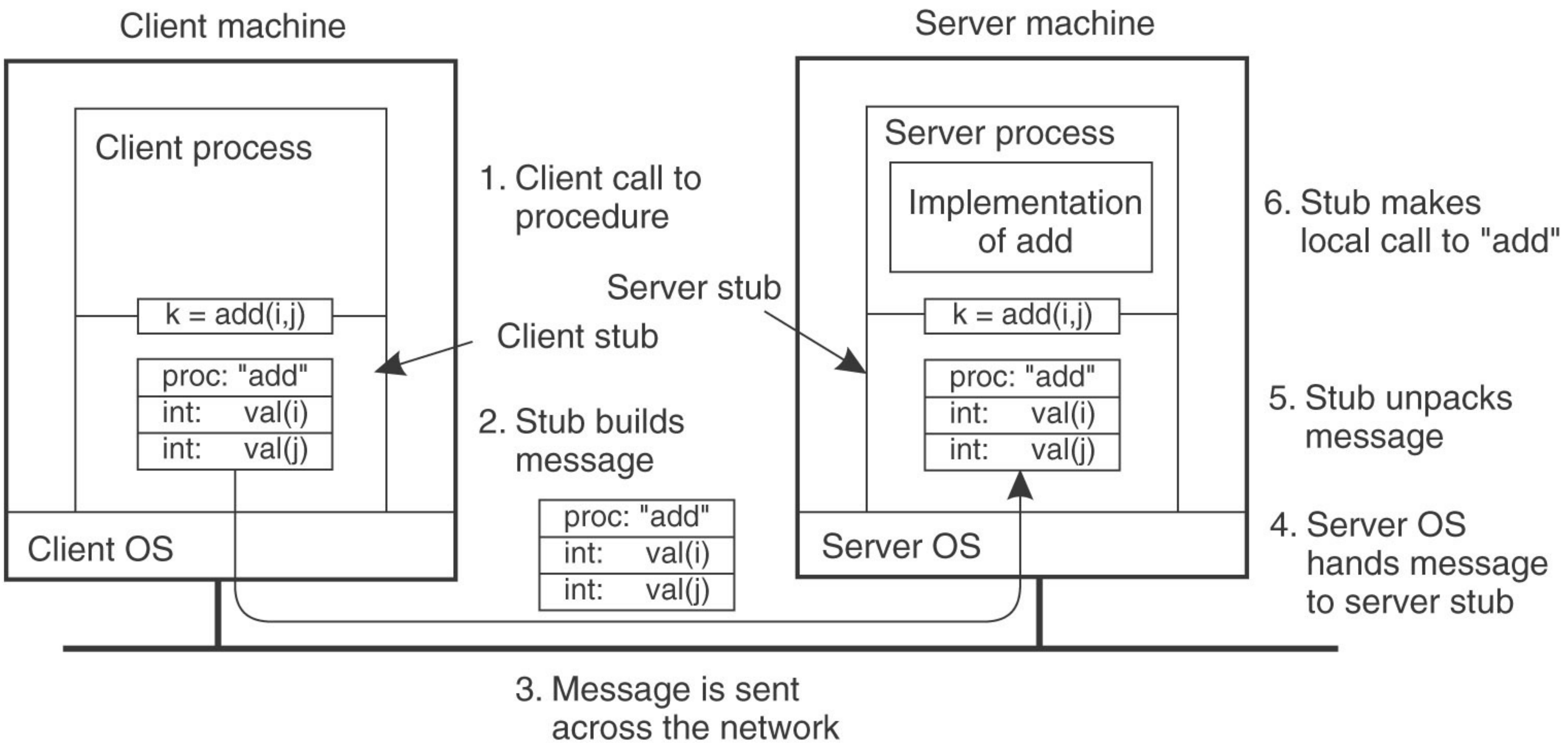
- Would like to do the same if called procedure or function is on a remote server



Solution — a pair of *Stubs*

- A *client-side stub* is a function that looks to the client as if it were a callable servicefunction
  - I.e., same API as the service's implementation of the function
- A *service-side stub* looks like a caller to the service
  - I.e., like a hunk of code invoking the service function
- The client program thinks it's invoking the service
  - but it's calling into the client-side stub
- The service program thinks it's called by the client
  - but it's really called by the service-side stub
- The stubs send messages to each other to make the RPC happen transparently (almost!)

RPC Stubs



Tanenbaum & Van Steen, Fig 4-7

RPC *Stubs* – Summary

- *Client-side stub*
  - Looks like local server function
  - Same interface as local function
  - Bundles arguments into a message, sends to server-side stub
  - Waits for reply, un-bundles results
  - returns

- *Server-side stub*
  - Looks like local client function to server
  - Listens on a socket for message from client stub
  - Un-bundles arguments to local variables
  - Makes a local function call to server
  - Bundles result into reply message to client stub

Result – a very useful Abstraction

- The hard work of building messages, formatting, uniform representation, etc., is buried in the stubs
  - Where it can be automated!
- Designers of client and server can concentrate on *semantics* of application
- Programs behave in familiar way

RPC – Issues

- Transparency: to be or not to be?
- How to make the “remote” part of RPC invisible to the programmer?
- What are semantics of parameter passing?
  - E.g., pass by reference?
- How to bind (locate & connect) to servers?
- How to handle heterogeneity?
  - OS, language, architecture, …
- How to make it go fast?

RPC Model

- A server defines the service interface using an *interface definition language* (IDL)
  - the IDL specifies the names, parameters, and types for all client-callable server procedures
- A *stub compiler* reads the IDL declarations and produces two *stub functions* for each server function
  - *Server-side* and *client-side*

RPC Model (continued)

- Linking:–
  - Server programmer implements the service's functions and links with the *server-side* stubs
  - Client programmer implements the client program and links it with *client-side* stubs
- Operation:–
  - Stubs manage all of the details of remote communication between client and server

Transparency

- General distributed systems issue: does a remote service look *identical* to a local service
- Transparency allows programmers to ignore the network
- But, transparency can impose poor performance and complexity
- In practice
    - File systems try for transparency
    - RPC systems do not

Marshalling Arguments

- *Marshalling* is the packing of function parameters into a message packet: the task of converting programming language types into a byte stream
  - ○ How many bits are in an integer?
  - ○ How are floating point numbers represented?
  - ○ Is the architecture big-endian or little-endian?
  - – the RPC stubs call type-specific functions to marshal or unmarshal the parameters of an RPC
    - Client stub marshals the arguments into a message
    - Server stub unmarshals the arguments and uses them to invoke the service function
  - – on return:
    - the server stub marshals return values
    - the client stub unmarshals return values, and returns to the client program

Complex Types

- Object-oriented languages allow programmer-defined types
- Two basic strategies:
  - Push the type definition into the IDL (CORBA)
  - Add implicit support to the language
    - Java Serialization

Java Serialization

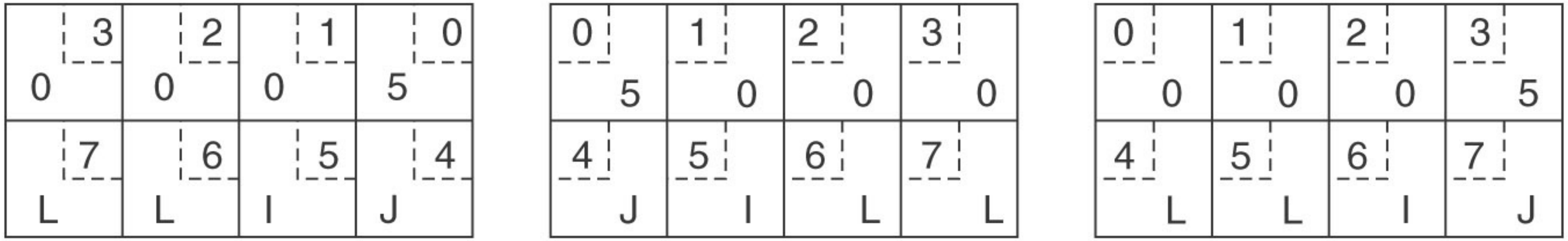
- Instances of `Serializable` can automatically converted into a byte stream
  - Thus, RMI allows serializable arguments

```
public class Person implements Serializable {
   private int age;
   private String name;
   private float salary;
   private transient boolean gender;
}
```

transient turns off serialization

Issue #1 — representation of data

- Big endian *vs.* little endian



(a) (b) (c)

Sent by Pentium Rec’d by SPARC After inversion

Representation of Data (continued)

- IDL must also define representation of data on network
  - Multi-byte integers
  - Strings, character codes
  - Floating point, complex, …
  - …
    - example: Sun’s XDR (external data representation)
- Each stub converts machine representation to/from network representation
- Clients and servers must *not* try to cast data!

Issue #2 — Pointers and References

`read(int fd, char* buf, int nbytes)`

- Pointers are only valid within one address space
- Cannot be interpreted by another process
    - Even on same machine!

- Pointers and references are ubiquitous in C, C++
    - Even in Java implementations!

Pointers and References — Restricted Semantics

- Option: *call by value*
  - Sending stub dereferences pointer, copies result to message
  - Receiving stub conjures up a new pointer
- Option: *call by result*
  - Sending stub provides buffer, called function puts data into it
  - Receiving stub copies data to caller's buffer as specified by pointer

Pointers and References — Restricted Semantics (continued)

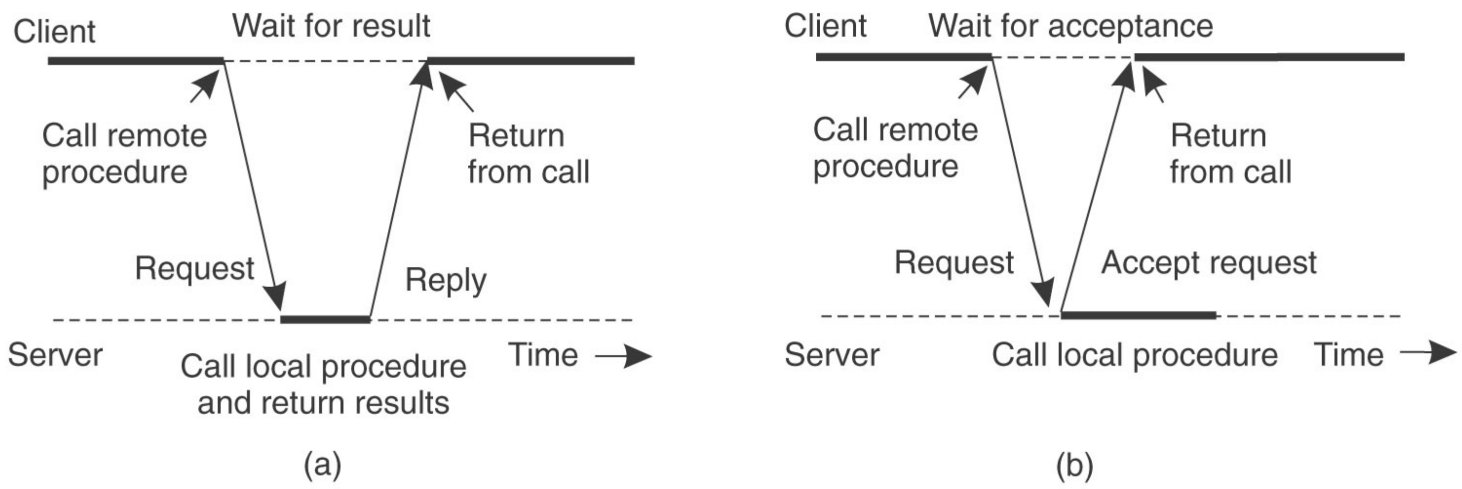
- Option: *call by value-result*
  - Caller's stub copies data to message, then copies result back to client buffer
  - Server stub keeps data in own buffer, server updates it; server sends data back in reply

- Not allowed:–
  - *Call by reference*
  - *Aliased arguments*

Transport of Remote Procedure Call

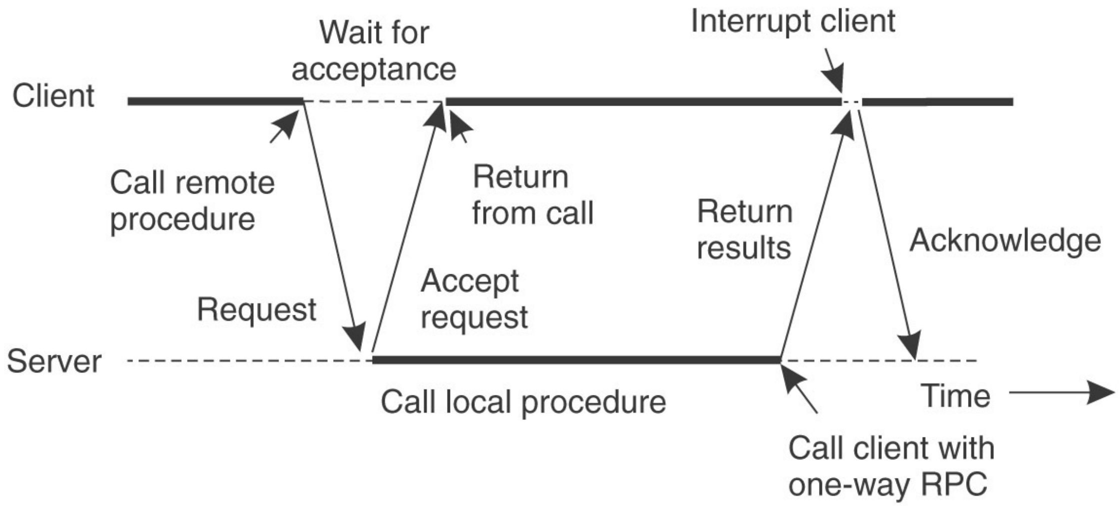
- Option — TCP
    - Connection-based, reliable transmission
    - Useful but heavyweight, less efficient
    - Necessary if repeating a call produces different result
- Alternative — UDP
    - If message fails to arrive within a reasonable time, caller's stub simply sends it again
    - Okay if repeating a call produces same result

Asynchronous RPC



- Analogous to spawning a thread
- Caller must eventually *wait* for result
  - Analogous to *join*

Asynchronous RPC (continued)



- Analogous to spawning a thread
- Caller must eventually *wait* for result
  - Analogous to *join*
  - Or be interrupted (software interrupt)

RPC *Binding*

- Binding is the process of connecting the client to the server
  - the server, when it starts up, exports its interface
    - identifies itself to a *network name server*
    - tells *RPC runtime* that it is alive and ready to accept calls
  - the client, before issuing any calls, imports the server
    - RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs

Remote Procedure Call is used …

- Between processes on different machines
  - E.g., client-server model
- Between processes on the same machine
  - More structured than simple message passing
- Between subsystems of an operating system
  - Windows XP (called *Local Procedure Call*)

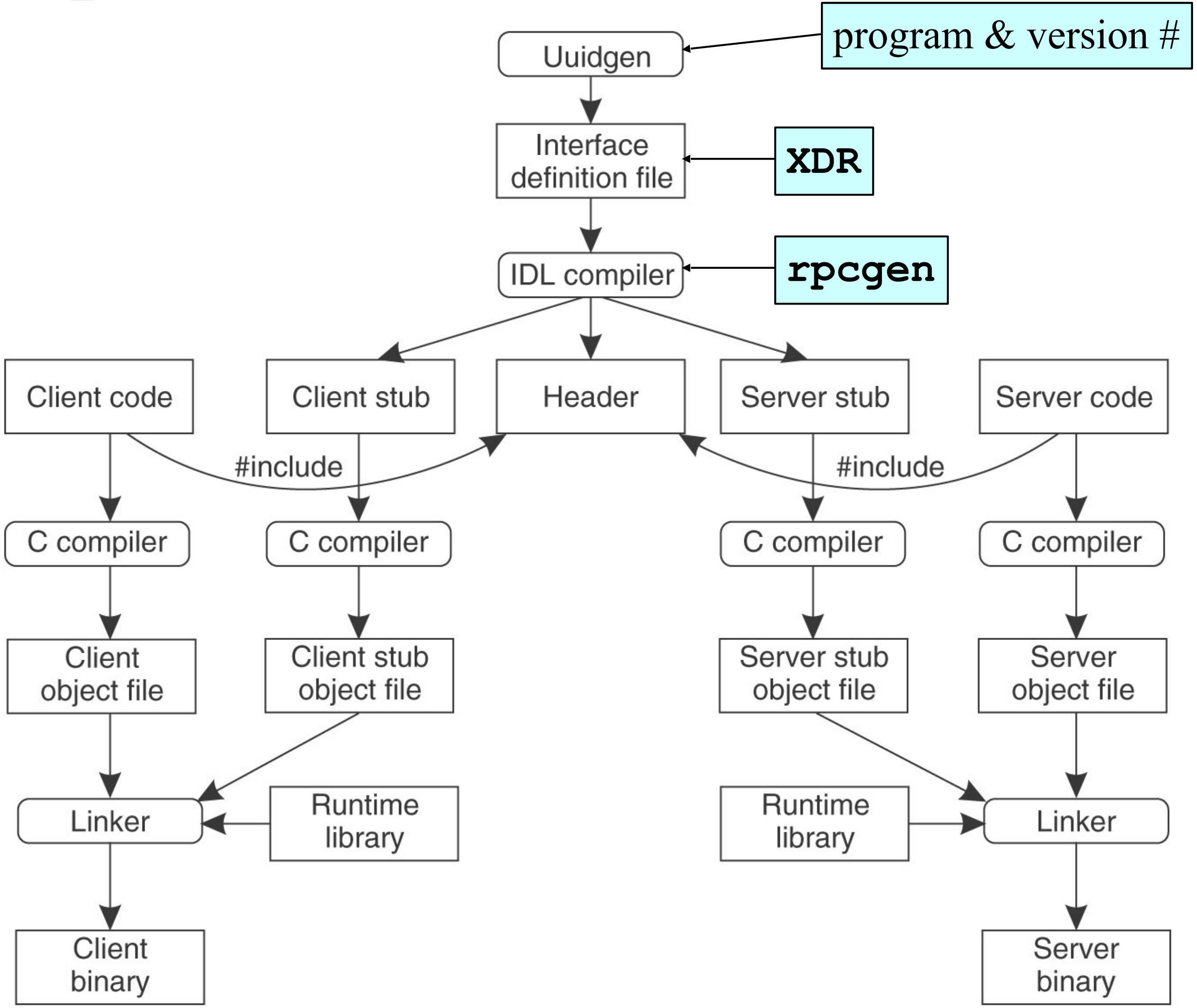
Practical RPC Systems (continued)

- Java RMI (Remote Method Invocation)
  - `java.rmi` standard package
  - Java-oriented approach — objects and methods
- CORBA (Common Object Request Broker Architecture)
  - Standard, multi-language, multi-platform middleware
  - Object-oriented
  - Heavyweight
- Web services
  - Allow arbitrary clients and servers to communicate using XML-based exchange formats
- Distributed file systems
  - e.g., NFS (network file system)
- Multiplayer network games
- Many other distributed systems

Practical RPC Systems

- DCE (Distributed Computing Environment)
    - Open Software Foundation
    - Basis for Microsoft DCOM
    - Tanenbaum & Van Steen, §4.2.4
- Sun’s ONC (Open Network Computing)
    - Very similar to DCE
    - Widely used
    - **`rpcgen`**
    - http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/HTML/AA-Q0R5B-TET1_html/TITLE.html

Implementation Model for ONC



Validating a Remote Service

- Purpose
  - Avoid binding to wrong *service* or wrong *version*
- DCE
  - Globally unique ID
    - Generated in template of IDL file
- Sun ONC
  - Program numbers registered with Sun
  - Version # and procedure # administered locally

RPC Binding — Sun ONC

- Service registers with `portmapper` service on server OS
  - Program # and version #
  - Optional static `port` #
- Client
  - Must know host name or IP address
  - `clnt_create(host, prog, vers, proto)`
    - I.e., RPC to `portmapper` of `host` requesting to bind to `prog, vers` using protocol `proto` (`tcp` or `udp`)
  - (Additional functions for authentication, etc.)
  - Invokes remote functions by name

Sun ONC (continued)

- **`#include <rpc/rpc.h>`**
    - Header file for client and server
- **`rpcgen`**
    - The stub compiler
        - Compiles **`interface.x`**
        - Produces **`.h`** files for client and service; also stubs
- See also
    - **`rpcinfo`**
    - *RPC Programming Guide*

Note on XDR
(the Interface Definition Language for ONC)

- Much like C header file
- Exceptions
  - **`string`** type – maps to **`char *`**
  - **`bool`** type – maps to **`bool_t`**

Sun ONC

- Online tutorial
  - http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/HTML/AA-Q0R5B-TET1_html/TITLE.html
- Code samples
  - http://web.cs.wpi.edu/~rek/DCS/D04/SunRPC.html
  - http://web.cs.wpi.edu/~goos/Teach/cs4513-d05/
  - http://web.cs.wpi.edu/~cs4513/b05/week4-sunrpc.pdf
- ONC RPCGEN :
  - https://www.cs.cf.ac.uk/Dave/C/node33.html

```
#include <rpc/rpc.h>
#include "rls.h"

main()
{
   extern bool_t xdr_dir();
   extern char * read_dir();

   registerrpc(DIRPROG, DIRVERS, READDIR,
            read_dir, xdr_dir, xdr_dir);

  svc_run();
}
```

```
/*
 * rls.c: remote directory listing client
 */
#include <stdio.h>
#include <strings.h>
#include <rpc/rpc.h>
#include "rls.h"

main (int argc, char *argv[] {
        char    dir[DIR_SIZE];
        /* call the remote procedure if
           registered */
        strcpy(dir, argv[2]);
        read_dir(argv[1], dir);

        /* spew-out the results and bail out
           of here! */
        printf("%s\n", dir);
        exit(0);
}

read_dir(char   *dir, *host) {
        extern bool_t xdr_dir();
        enum clnt_stat clnt_stat;

        clnt_stat = callrpc ( host, DIRPROG,
                  DIRVERS, READDIR,
                  xdr_dir, dir, xdr_dir, dir);
        if (clnt_stat != 0) clnt_perrno
           (clnt_stat);
}
```

The easiest way to define and generate the protocol is to use a protocol complier such as `rpcgen`

For the protocol you must identify the name of the service procedures, and data types of parameters and return arguments.
The protocol compiler reads a definitio and automatically generates client and server stubs.
`rpcgen` uses its own language (RPC language or RPCL) which looks very similar to preprocessor directives.
`rpcgen` exists as a standalone executable compiler that reads special files denoted by a `.x` prefix.
So to compile a RPCL file you simply do

```
rpcgen rpcprog.x
```

This will generate possibly four files:

- `rpcprog_clnt.c` -- the client stub
- `rpcprog_svc.c` -- the server stub
- `rpcprog_xdr.c` -- If necessary XDR (external data representation) filters
- `rpcprog.h` -- the header file needed for any XDR filters.

The external data representation (XDR) is an data abstraction needed for machine independent communication. The client and server need not be machines of the same type.