UNIVERSITY OF OREGON

CIS 415: Operating Systems

IPC and RPC

Prof. Kevin Butler
Spring 2012

Today's Lecture

UNIVERSITY OF OREGON

- Inter-process communication
- Remote procedure calls

- Reminders
  - Assignment 1 due April 22
  - Project 1 due April 24

Process Communication

UNIVERSITY OF OREGON

- Processes need to share information
- Process model is a useful way to isolate running programs (separate resources, state, etc)
  - Can simplify programs (no need to worry about other processes running)
  - But processes don't always work in isolation
- Discuss a variety of ways
  - Doesn't include regular files and signals

Process communication

- When is communication necessary?
- Lots of examples in operating systems
  - threads with access to same data structures
  - kernel/OS access to user process data
  - processes sharing data via shared memory
  - processes sharing data via system calls
  - processes sharing data via file system
- And in general computer science
  - DB transactions, P/L parallelism issues

IPC Mechanisms

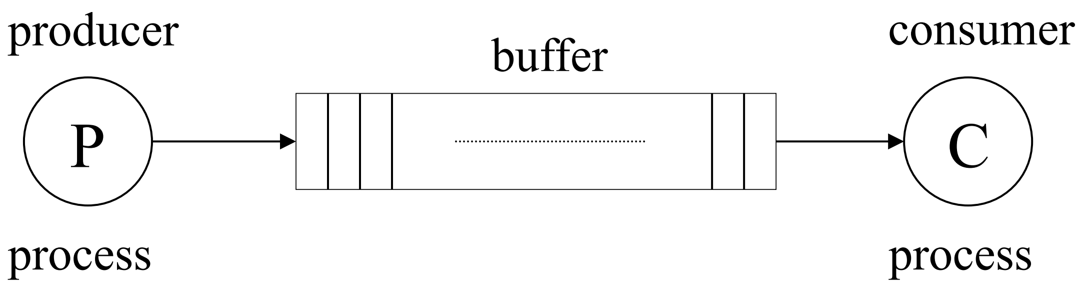
UNIVERSITY OF OREGON

- Two fundamental methods
- Shared memory
  - Pipes, shared buffer
- Message Passing
  - Mailboxes, Sockets
- Which one would you use and why?

Shared Memory

UNIVERSITY OF OREGON

- Two processes share a memory region
  - One writes: Producer
  - One reads: Consumer
- Producer action
  - While buffer not full
  - Add stuff to buffer
- Consumer actions
  - When stuff in buffer
  - Read it
- Must manage where new stuff is in the buffer…



Shared Memory -- Producer

UNIVERSITY OF OREGON

```
item nextProduced;

while (1) {
  while (((in + 1) % BUFFER_SIZE) == out)
                                ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Shared Memory -- Consumer

```
item nextConsumed;

while (1) {
  while (in == out)
      ; /* do nothing */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
}
```

Shared Memory

UNIVERSITY OF OREGON

- Communicate by reading/writing from a specific memory location
  - Setup a shared memory region in your process
  - Permit others to attach to the shared memory region
- `shmget` -- create shared memory segment
  - Permissions (read and write)
  - Size
  - Returns an identifier for segment
- `shmat` -- attach to existing shared memory segment
  - Specify identifier
  - Location in local address space
  - Permissions (read and write)
- Also, operations for detach and control

Pipes

- Producer-Consumer mechanism
  - `prog1 | prog2`
  - The output of `prog1` becomes the input to `prog2`
  - More precisely,
    - The standard output of `prog1` is connected to the standard input of `prog2`
- OS sets up a fixed-size buffer
  - System calls: `pipe, dup, popen`
- Producer
  - Write to buffer, if space available
- Consumer
  - Read from buffer if data available

Pipes

- Buffer management
  - A finite region of memory (array or linked-list)
  - Wait to produce if no room
  - Wait to consume if empty
  - Produce and consume complete items
- Access to buffer
  - Write adds to buffer (updates end of buffer)
  - Reader removes stuff from buffer (updates start of buffer)
  - Both are updating buffer state
- Issues
  - What happens when end is reached (e.g., in finite array)?
  - What happens if reading and writing are concurrent?

Shared Memory Machines

- SGI UV 1000 (Pitt SC)
  - 256 blades, each with 2 8-core Xeon processors
  - Each core has 8 GB RAM = 128 GB per blade
- Coherent shared-memory machine = all memory accessible to the machine
  - 32 TB of RAM
- Why? Certain problems hard to chunk up (eg graphs)

IPC -- Message Passing

UNIVERSITY OF OREGON

- Establish communication link
  - Producer sends on link
  - Consumer receives on link
- IPC Operations
  - Y: Send(X, message)
  - X: Receive(Y, message)
- Issues
  - What if X wants to receive from anyone?
  - What if X and Y aren't ready at same time?
  - What size message can X receive?
  - Can other processes receive the same message from Y?

IPC -- Synchronous Messaging

- Direct communication from one process to another
- Synchronous send
  - Send(X, message)
  - Producer must wait for the consumer to be ready to receive the message
- Synchronous receive
  - Receive(id, message)
  - Id could be X or anyone
  - Wait for someone to deliver a message
  - Allocate enough space to receive message
- Synchronous means that both have to be ready!

IPC -- Asynchronous Messaging

- Indirect communication from one process to another
- Asynchronous send
  - Send(M, message)
  - Producer sends message to a buffer M (like a mailbox)
  - No waiting (modulo busy mailbox)
- Asynchronous receive
  - Receive(M, message)
  - Receive a message from a specific buffer (get your mail)
  - No waiting (modulo busy mailbox)
  - Allocate enough space to receive message
- Asynchronous means that you can send/receive when you're ready
  - What are some issues with the buffer?

IPC -- Sockets

- Communcation end point
  - Connect one socket to another (TCP/IP)
  - Send/receive message to/from another socket (UDP/IP)
- Sockets are named by
  - IP address (roughly, machine)
  - Port number (service: ssh, http, etc.)
- Semantics
  - Bidirectional link between a pair of sockets
  - Messages: unstructured stream of bytes
- Connection between
  - Processes on same machine (UNIX domain sockets)
  - Processes on different machines (TCP or UDP sockets)
  - User process and kernel (netlink sockets)

Files and file descriptors

- Remember open, read, write, and close?
  - POSIX system calls for interacting with files
  - open( ) returns a *file descriptor*
    - an integer that represents an open file
    - inside the OS, it's an index into a table that keeps track of any state associated with your interactions, such as the file position
    - you pass the file descriptor into read, write, and close

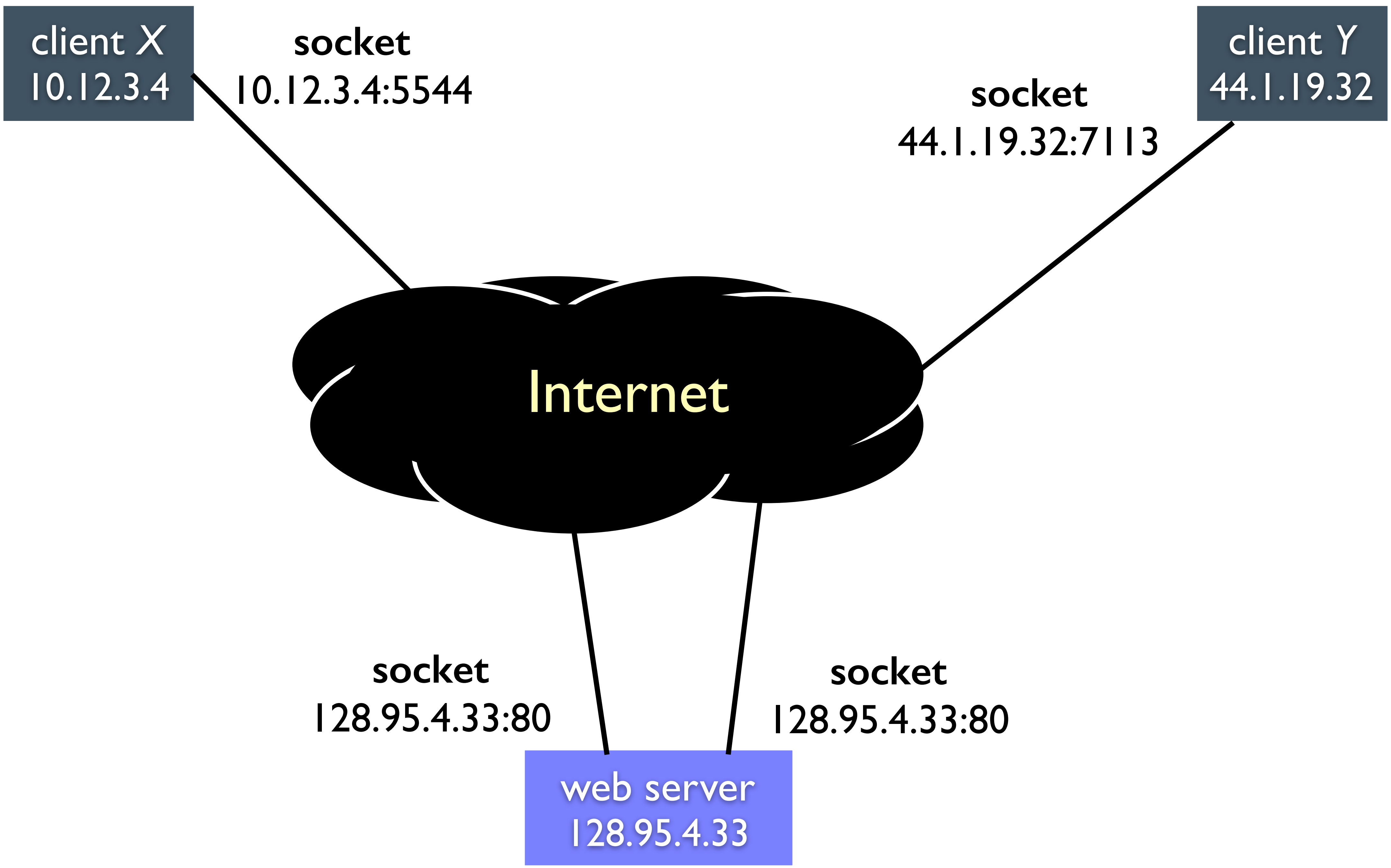
Networks and sockets

- UNIX likes to make all I/O look like file I/O
  - the good news is that you can use read( ) and write( ) to interact with remote computers over a network!
  - just like with files....
    - your program can have multiple network channels open at once
    - you need to pass read( ) and write( ) a ***file descriptor*** to let the OS know which network channel you want to write to or read from
  - a file descriptor used for network communications is a **socket**

Examples of sockets

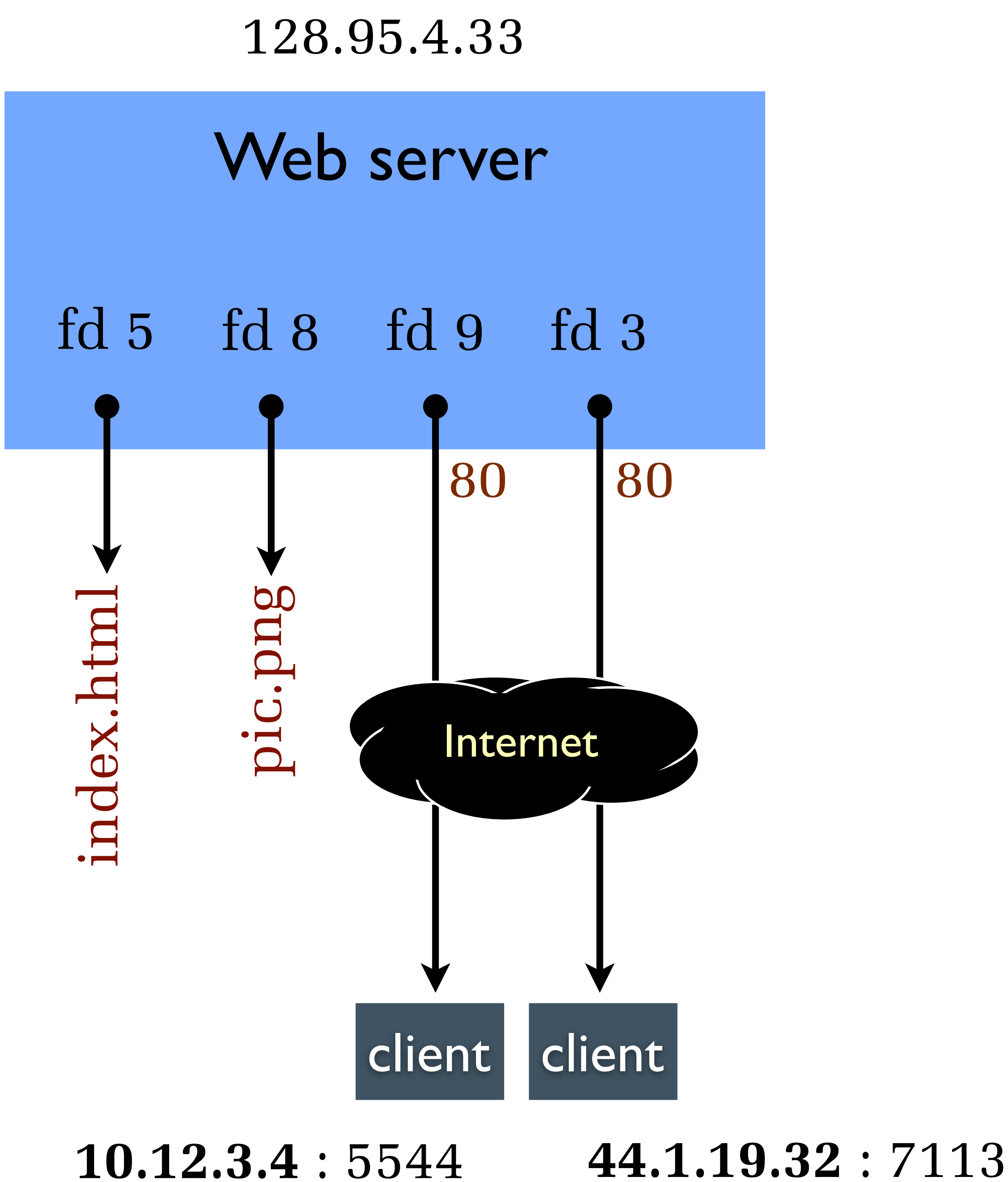
- HTTP / SSL
- email (POP/IMAP)
- ssh
- telnet

IPC: Sockets


client X
10.12.3.4
socket
10.12.3.4:5544
client Y
44.1.19.32
socket
44.1.19.32:7113
Internet
socket
128.95.4.33:80
socket
128.95.4.33:80
web server
128.95.4.33

Pictorially

UNIVERSITY OF OREGON



file descriptor	type	connected to?
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

OS's descriptor table

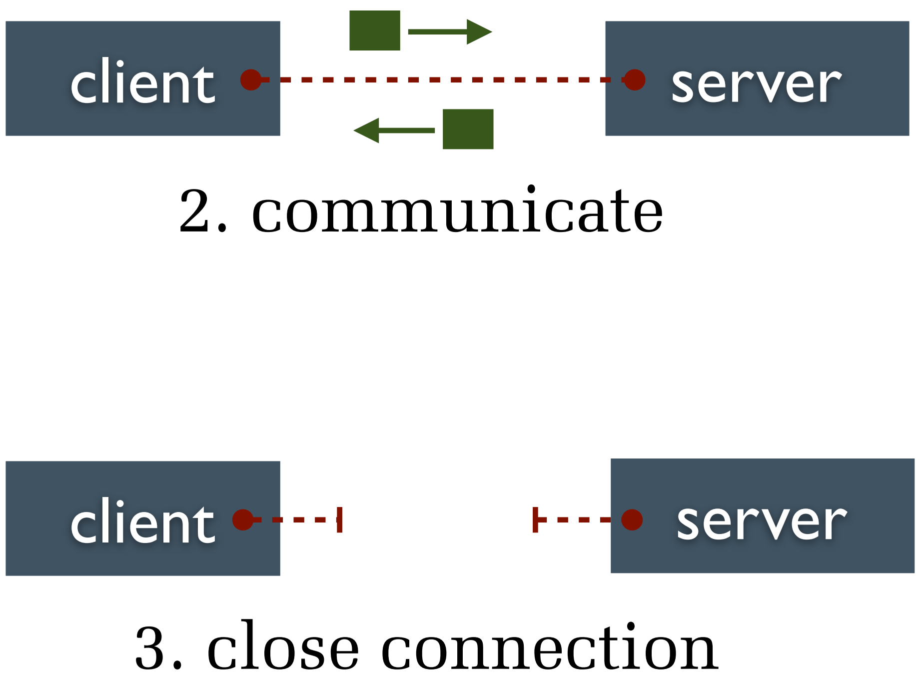
Types of sockets

UNIVERSITY OF OREGON

- Stream sockets
  - for connection-oriented, point-to-point, reliable bytestreams
    - uses TCP, SCTP, or other stream transports
- Datagram sockets
  - for connection-less, one-to-many, unreliable packets
    - uses UDP or other packet transports
- Raw sockets
  - for layer-3 communication (raw IP packet manipulation)

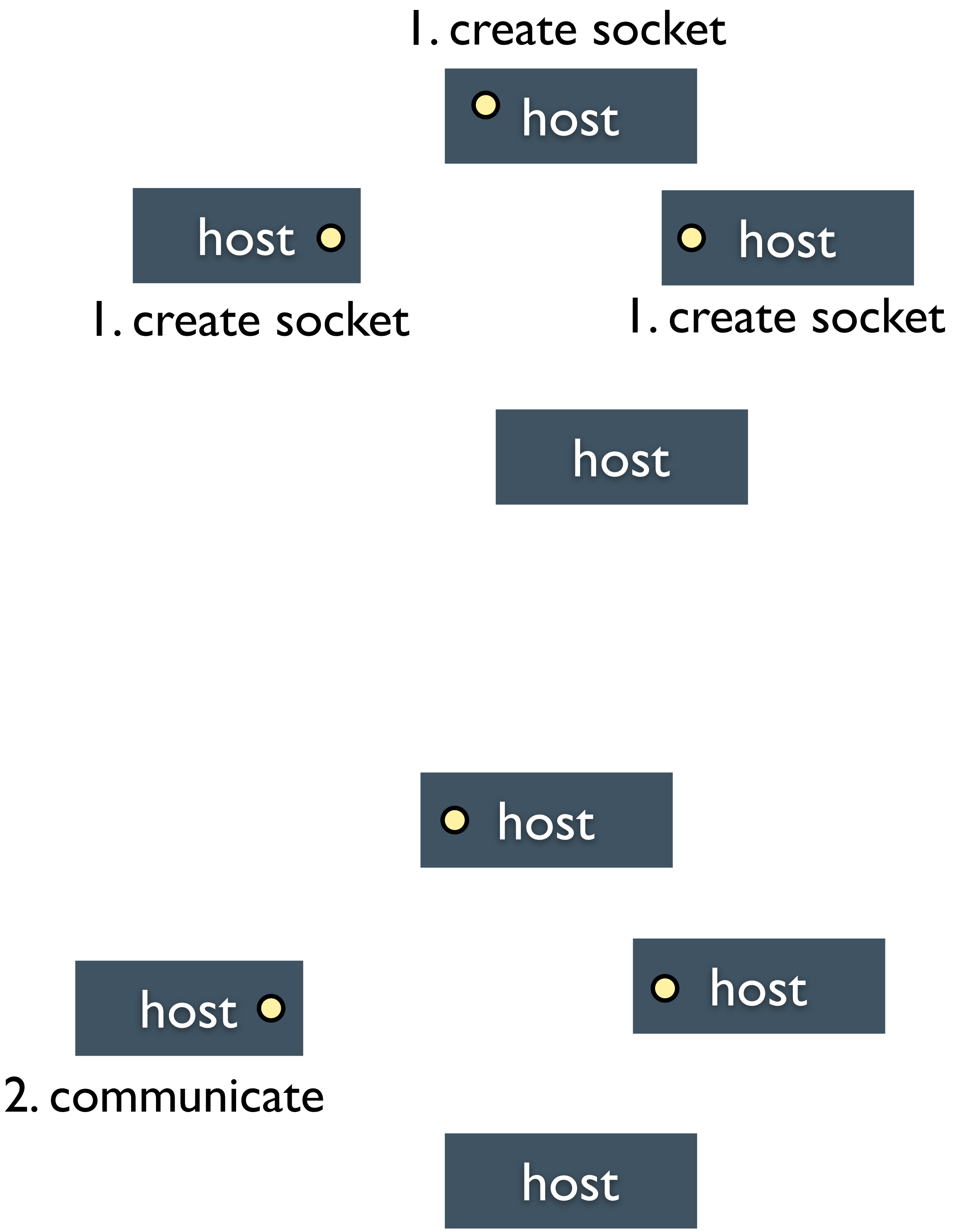
Stream sockets

- Typically used for client / server communications
  - but also for other architectures, like peer-to-peer
- Client
  - an application that establishes a connection to a server
- Server
  - an application that receives connections from clients



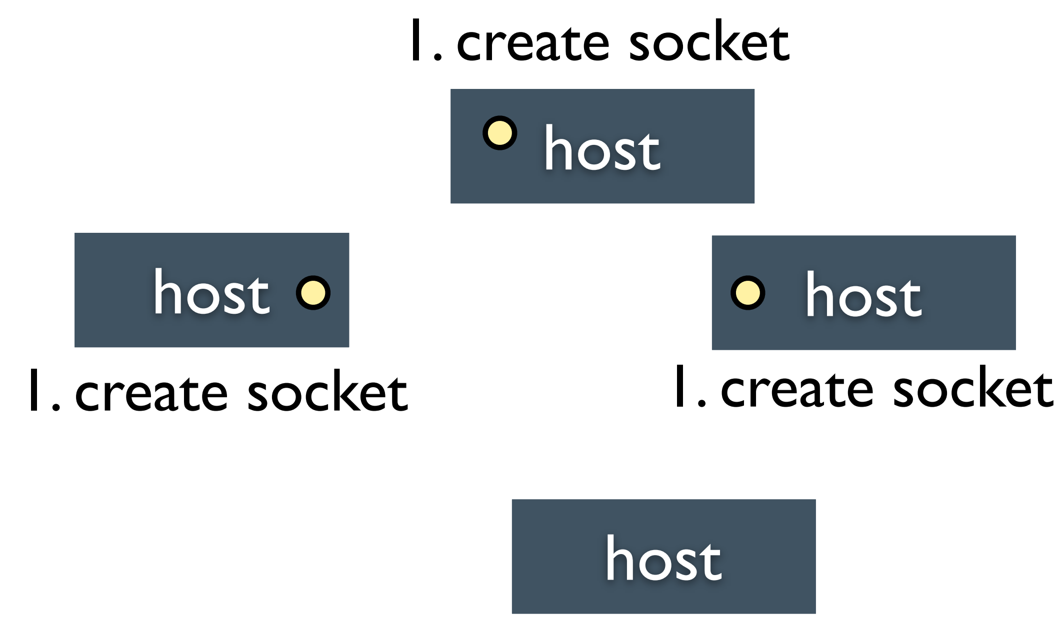
Datagram sockets

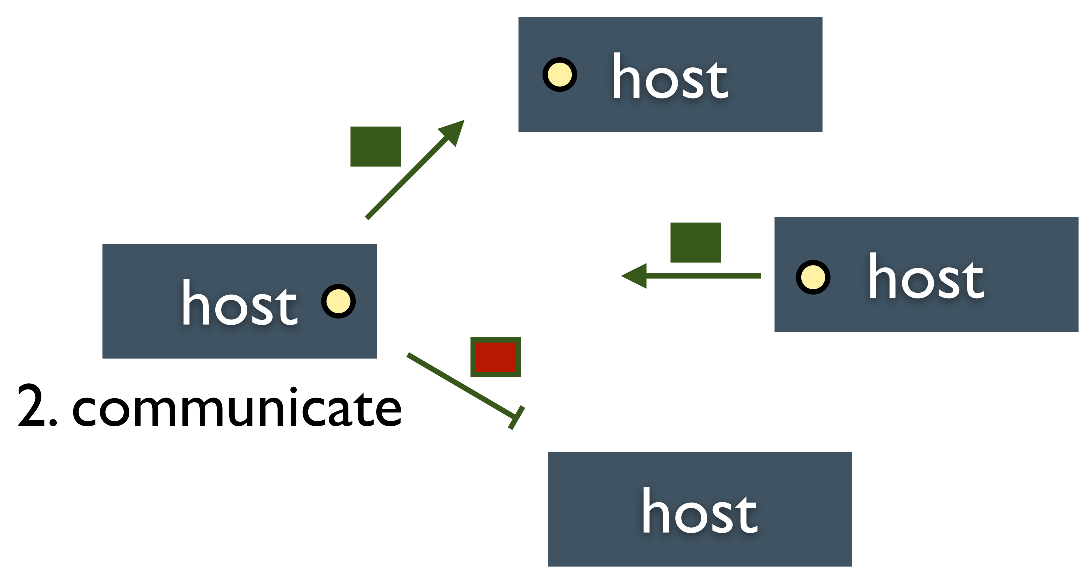
- Used less frequently than stream sockets
  - they provide no flow control, ordering, or reliability
- Often used as a building block
  - streaming media applications
  - sometimes, DNS lookups



Datagram sockets

- Used less frequently than stream sockets
  - they provide no flow control, ordering, or reliability
- Often used as a building block
  - streaming media applications
  - sometimes, DNS lookups





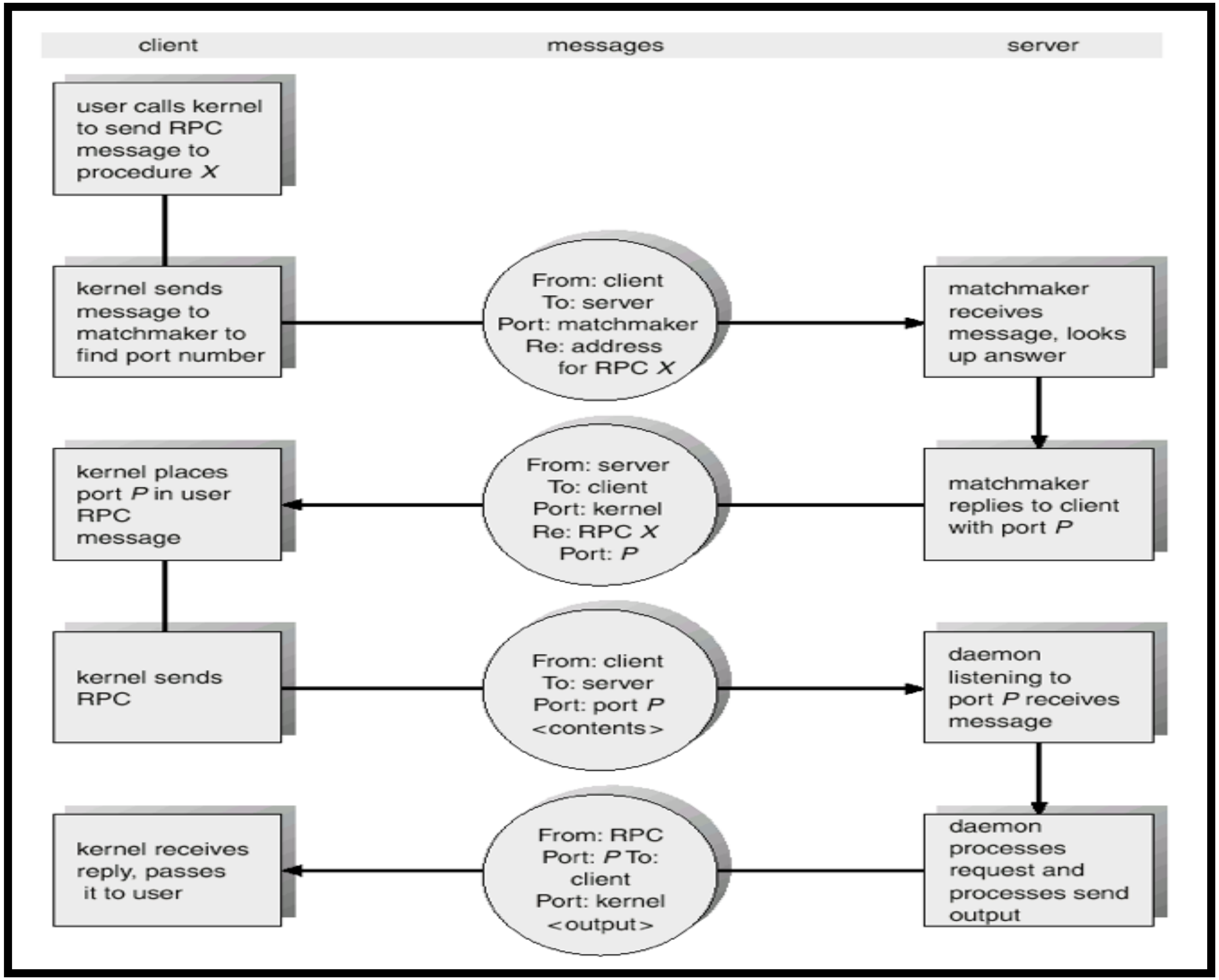
IPC -- Sockets

- Issues
- Communication semantics
  - Reliable or not
- Naming
  - How do we know a machine's IP address? DNS
  - How do we know a service's port number?
- Protection
  - Which ports can a process use?
  - Who should you receive a message from?
    - Services are often open -- listen for any connection
- Performance
  - How many copies are necessary?
  - Data must be converted between various data types

Remote Procedure Calls

- IPC via a procedure call
  - Looks like a "normal" procedure call
  - However, the called procedure is run by another process
    - Maybe even on another machine
- RPC mechanism
  - Client stub
  - "Marshall" arguments
  - Find destination for RPC
  - Send call and marshalled arguments to destination (e.g., via socket)
  - Server stub
  - Unmarshalls arguments
  - Calls actual procedure on server side
  - Return results (marshall for return)

Remote Procedure Calls



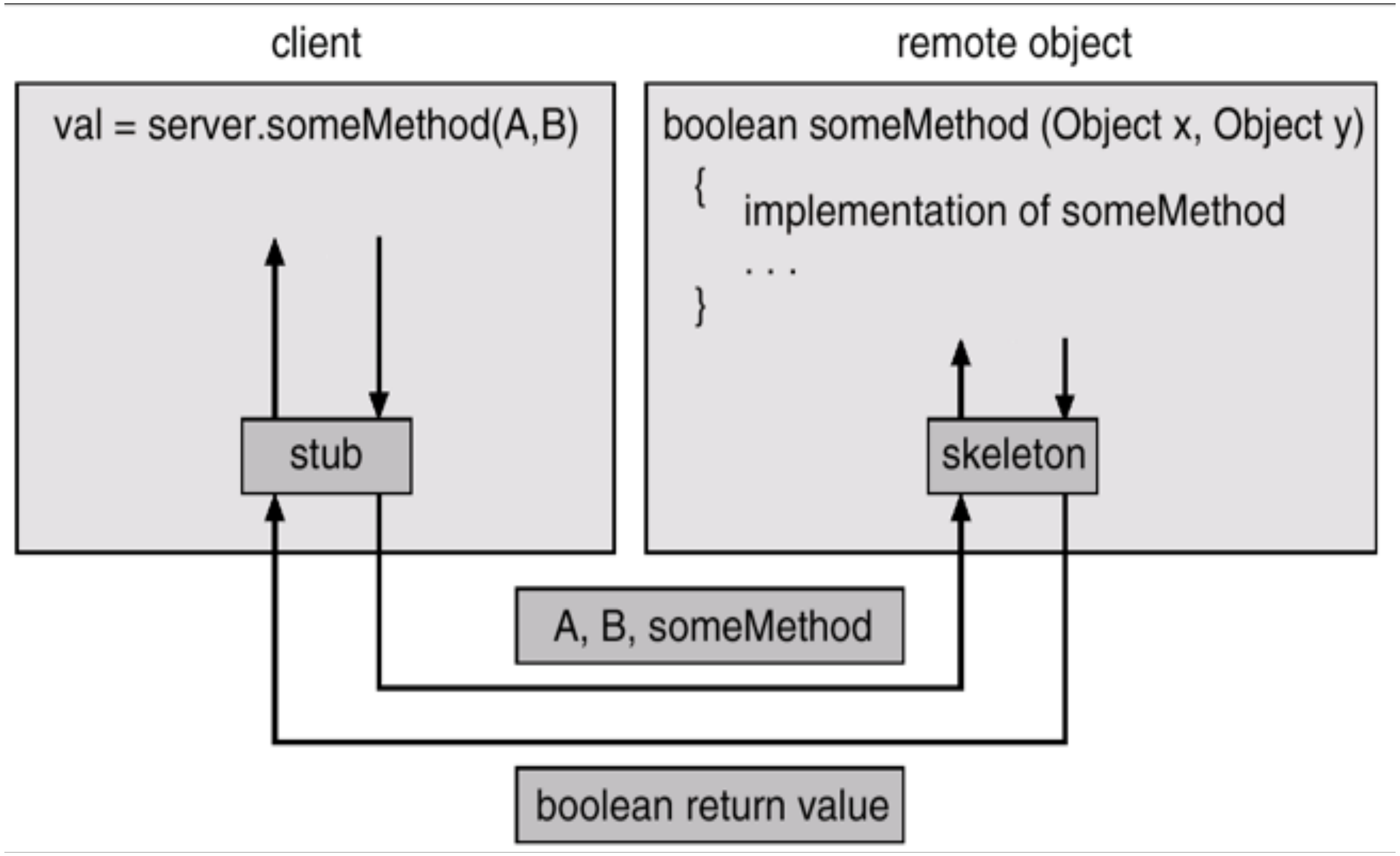
Remote Procedure Calls

UNIVERSITY OF OREGON

- Supported by systems
  - Java RMI
  - CORBA
- Issues
  - Support to build client/server stubs and marshalling code
  - Layer on existing mechanism (e.g., sockets)
  - Remote party crashes… then what?
- Performance versus abstractions
  - What if the two processes are on the same machine?

Remote Procedure Calls

- Marshalling



Example (RMI Server)

```
public class RmiServer extends UnicastRemoteObject
    implements RmiServerIntf {
    public static final String MESSAGE = "Hello world";

    public RmiServer() throws RemoteException {
    }
    public String getMessage() {
        return MESSAGE;
    }
    public static void main(String args[]) {
        System.out.println("RMI server started");

        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
            System.out.println("Security manager installed.");
        } else {
            System.out.println("Security manager already exists.");
        }

...
        try {
            //Instantiate RmiServer
            RmiServer obj = new RmiServer();

            // Bind this object instance to the name "RmiServer"
            Naming.rebind("//localhost/RmiServer", obj);

            System.out.println("PeerServer bound in registry");
        } catch (Exception e) {
            System.err.println("RMI server exception:" + e);
            e.printStackTrace();
        }
    }
}
```

Example (RMI Server)

UNIVERSITY OF OREGON

```
public class RmiServer extends UnicastRemoteObject
    implements RmiServerIntf {
    public static final String MESSAGE = "Hello world";

    public RmiServer() throws RemoteException {
    }
    public String getMessage() {
        return MESSAGE;
    }
    public static void main(String args[]) {
        System.out.println("RMI server started");

        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
            System.out.println("Security manager installed.");
        } else {
            System.out.println("Security manager already exists.");
        }

...
        try {
            //Instantiate RmiServer
            RmiServer obj = new RmiServer();

            // Bind this object instance to the name "RmiServer"
            Naming.rebind("//localhost/RmiServer", obj);

            System.out.println("PeerServer bound in registry");
        } catch (Exception e) {
            System.err.println("RMI server exception:" + e);
            e.printStackTrace();
        }
    }
}
```

Binding to registry

Example (RMI Interface)

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

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

Example (RMI Client)

UNIVERSITY OF OREGON

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

public class RmiClient {
    // "obj" is the reference of the remote object
    RmiServerIntf obj = null;

    public String getMessage() {
        try {
            obj = (RmiServerIntf)Naming.lookup("//localhost/RmiServer");
            return obj.getMessage();
        } catch (Exception e) {
            System.err.println("RmiClient exception: " + e);
            e.printStackTrace();

            return e.getMessage();
        }
    }

    public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        RmiClient cli = new RmiClient();

        System.out.println(cli.getMessage());
    }
}
```

IPC Summary

UNIVERSITY OF OREGON

- Lots of mechanisms
  - Pipes
  - Shared memory
  - Sockets
  - RPC
- Trade-offs
  - Ease of use, functionality, flexibility, performance
- Implementation must maximize these
  - Minimize copies (performance)
  - Synchronous vs Asynchronous (ease of use, flexibility)
  - Local vs Remote (functionality)

Summary

- Process
  - Execution state of a program
- Process Creation
  - fork and exec
  - From binary representation
- Process Description
  - Necessary to manage resources and context switch
- Process Scheduling
  - Process states and transitions among them
- Interprocess Communication
  - Ways for processes to interact (other than normal files)

- Next time: Threads