CS310 Operating Systems

Lecture 44: Distributed System

Ravi Mittal

IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, University of California, Berkeley
  - Book: Computer Networking: A top-down Approach, by Kurose, and Ross
  - 15-440, Distributed Systems, Stanford University, Lecture 6, Class Notes
  - Book: Modern Operating System by Tanenbaum and Bos

Previous Classes

Today we will study

- Local Storage → Cloud Storage
- Socket Overview
- RPC Overview
- Distributed File System - Concepts

Local Storage → Cloud Storage

Access to Storage (Files) Today

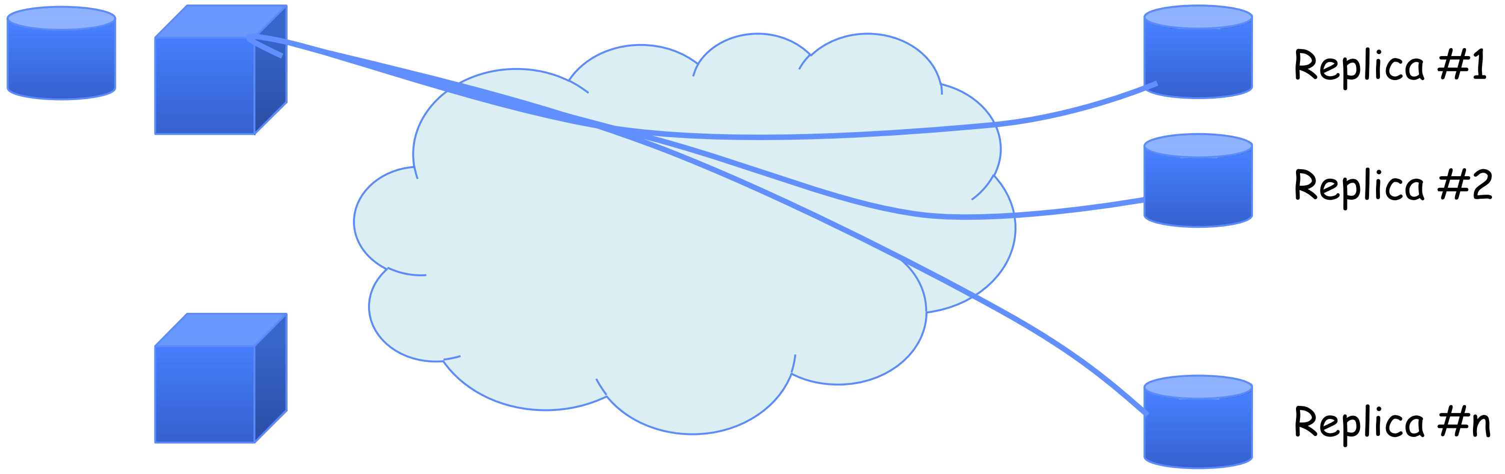
- File system over SSD or HDD on your local machine
- File Server in your organization
  - Remote login (ssh), file transfer (scp) or mount
- Cloud storage
  - Accessed through web or app (drive, box, …)
  - Mounted on your local machine
  - Replicated and/or Distributed

Cloud Storage Options

- Storage Account / Share is like disk “partition”
  - Holds file system: directory, index, free map, data blocks
- Access methods: mount, REST, file transfer, synch
- Security: credentials, encryption
- Performance: HDDs, SSDs
- Redundancy
  - Local RAID
  - Storage cluster in a Data Center
  - Zone redundant (across data centers)
  - Geographic regions

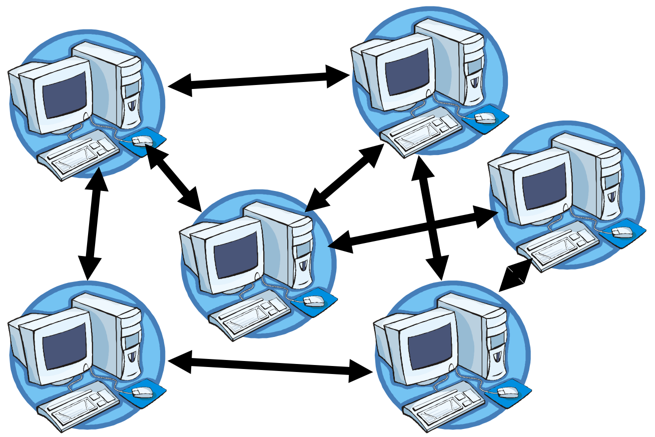
Geographic Replication: Cluster, Zone, Geo

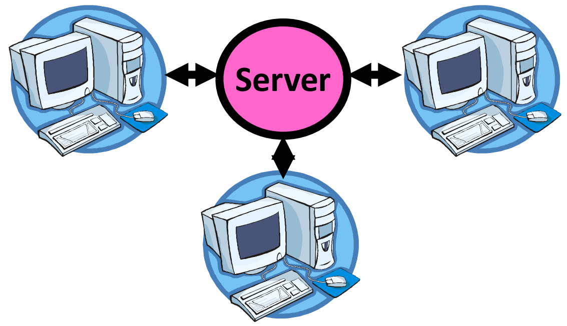
- Highly durable: Hard to destroy all copies
- Highly available for reads: Just talk to any copy
- What about for writes? Need every copy online to update all together?



Centralized vs. Distributed

- Centralized System
  - Major functions performed on one physical computer
- Distributed System
  - Physically separate computers working together to perform a single task


Peer-to-Peer Model



Client/Server Model

Distributed Systems: Motivation

- The *promise* of distributed systems
    - *Higher availability*: one machine goes down, use another
    - *Better durability*: store data in multiple locations
    - *More security*: each piece easier to make secure

- Other advantages
    - Cheaper/easier to build lots of simple computers
    - Allows for adding more resources incrementally
    - Easier for users to collaborate

Distributed Systems: Goals/Requirements

- Transparency: the ability of the system to mask its complexity behind a simple interface

- Possible transparencies:
  - Location: Can’t tell where resources are located
  - Migration: Resources may move without the user knowing
  - Replication: Can’t tell how many copies of resource exist
  - Concurrency: Can’t tell how many users there are
  - Parallelism: System may speed up large jobs by splitting them into smaller pieces
  - Fault Tolerance: System may hide various things that go wrong

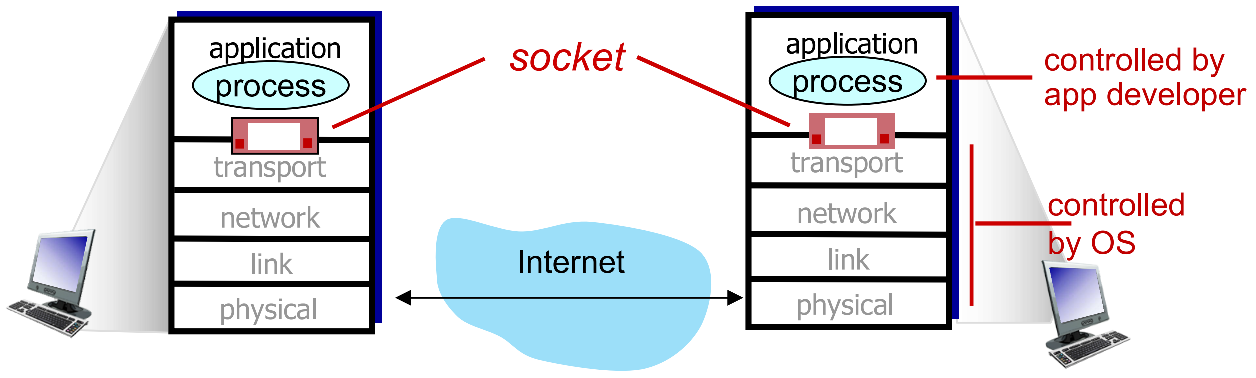
Examples of Transparency

- RPC: Simple function-like interface
  - Masks complexity of marshalling/unmarshalling, sending data, using sockets…
- Sockets: Simple file-like interface
  - Masks complexity of segmentation, retransmissions, windowing, etc.

Socket Overview

Sockets

- Process sends/receives messages to/from its socket
- A Socket is analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side

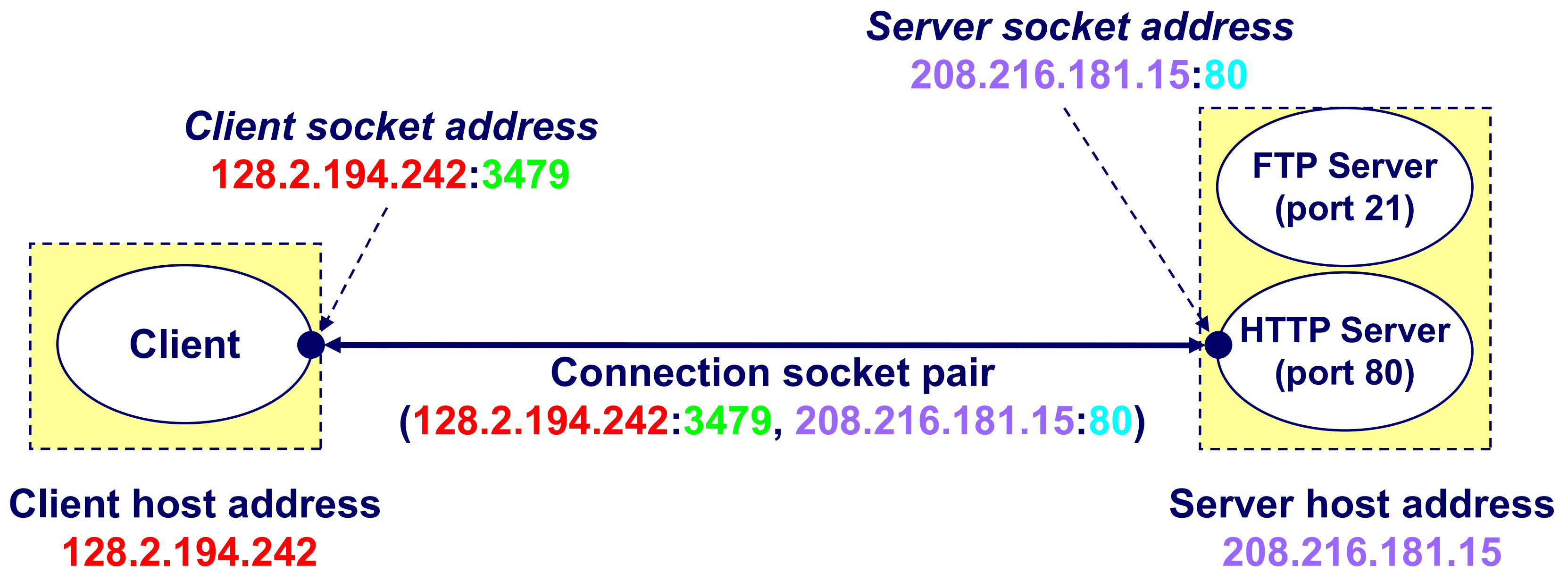


Sockets

- 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

Socket: Identify the Destination

- Addressing
  - IP address
- Multiplexing
  - port

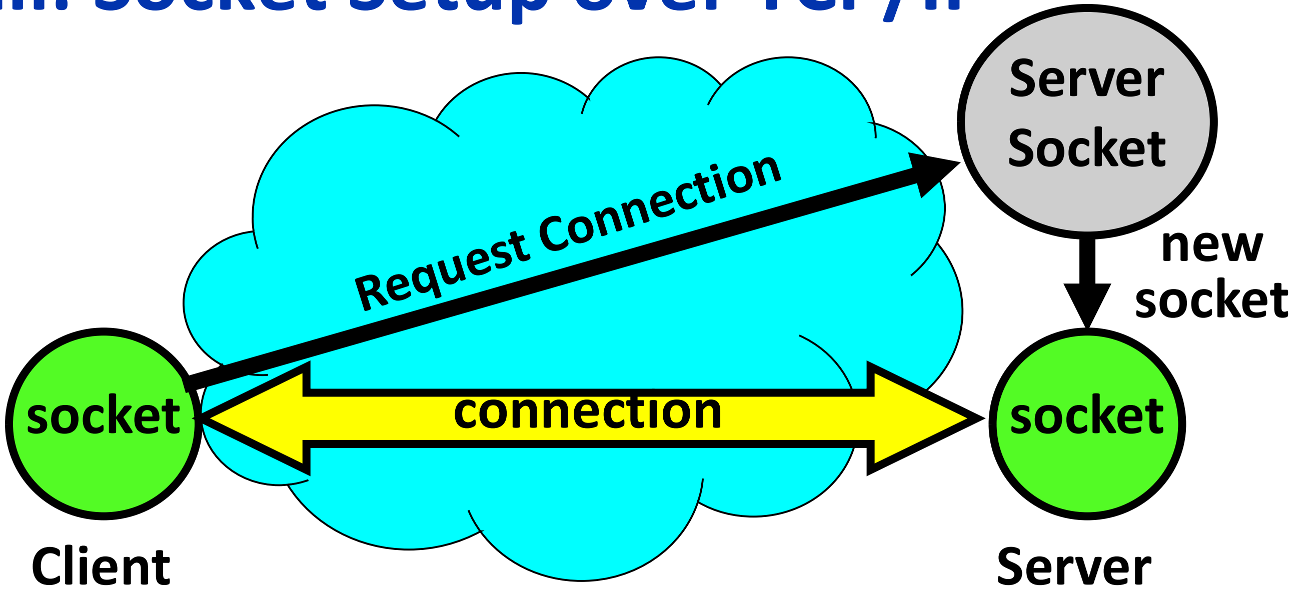


Use of Sockets

- How to use sockets
  - Setup socket
    - Where is the remote machine (IP address, hostname)
    - What service gets the data (port)
  - Send and Receive
    - Designed just like any other I/O in unix
    - send -- write
    - recv -- read
  - Close the socket

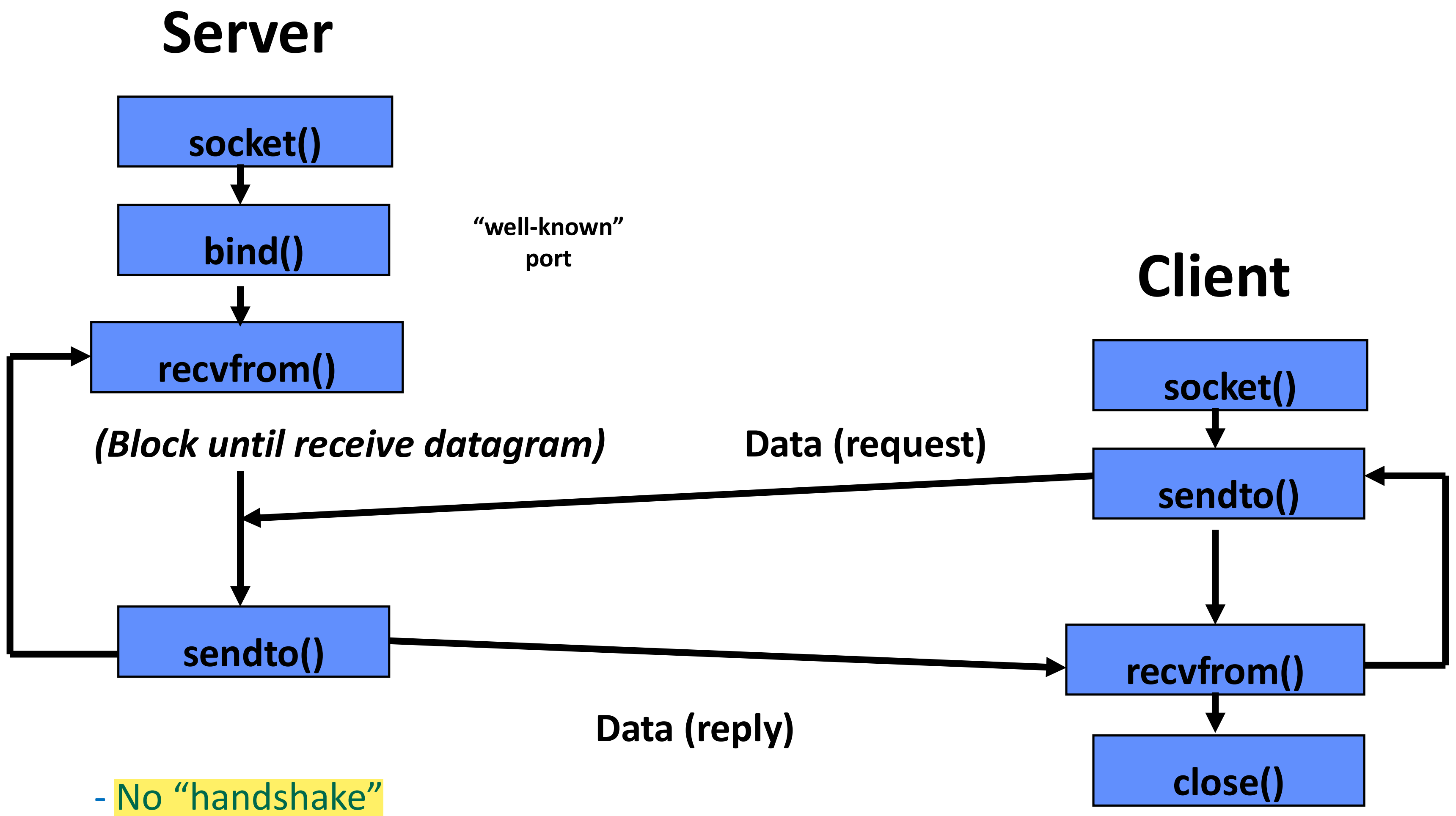
- Sockets
  - UDP Client Server
  - TCP Client Server

Recall: Socket Setup over TCP/IP



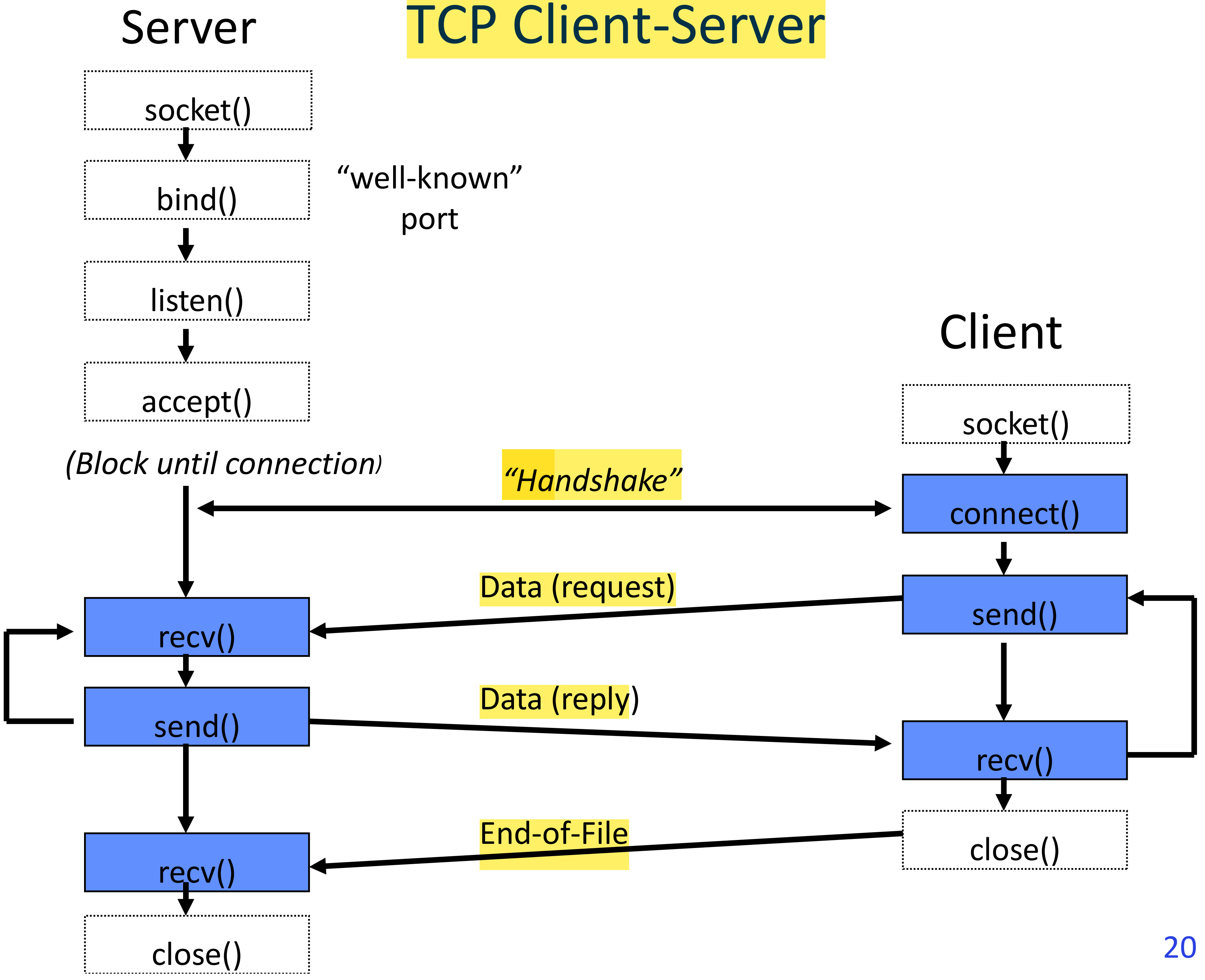
- Things to remember:
  - Connection involves 5 values:
    [Client Addr, Client Port, Server Addr, Server Port, Protocol]
  - Often, Client Port "randomly" assigned
  - Server Port often "well known"
    - 80 (web), 443 (secure web), 25 (sendmail), etc
    - Well-known ports from 0—1023

UDP Client-Server (self reading)



- No "handshake"
- No simultaneous close
- No fork()/spawn() for concurrent servers!

TCP Client-Server



RPC Overview

RPC: Concept

- A type of client/server communication
- Attempts to make remote procedure calls look like local ones

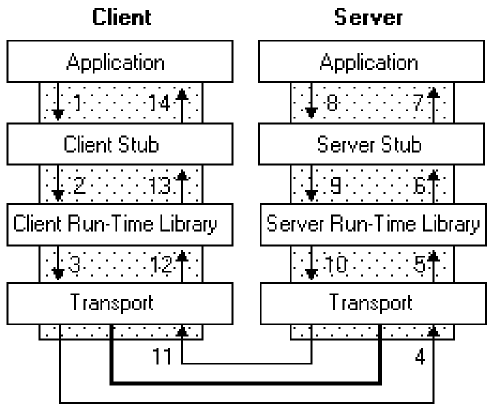


figure from Microsoft MSDN

```
{ ...
   foo()
}
void foo() {
  invoke_remote_foo()
}
```

Remote procedure call

- A remote procedure call makes a call to a remote service look like a local call
  - RPC makes transparent whether server is local or remote
  - RPC allows applications to become distributed transparently
  - RPC makes architecture of remote machine transparent

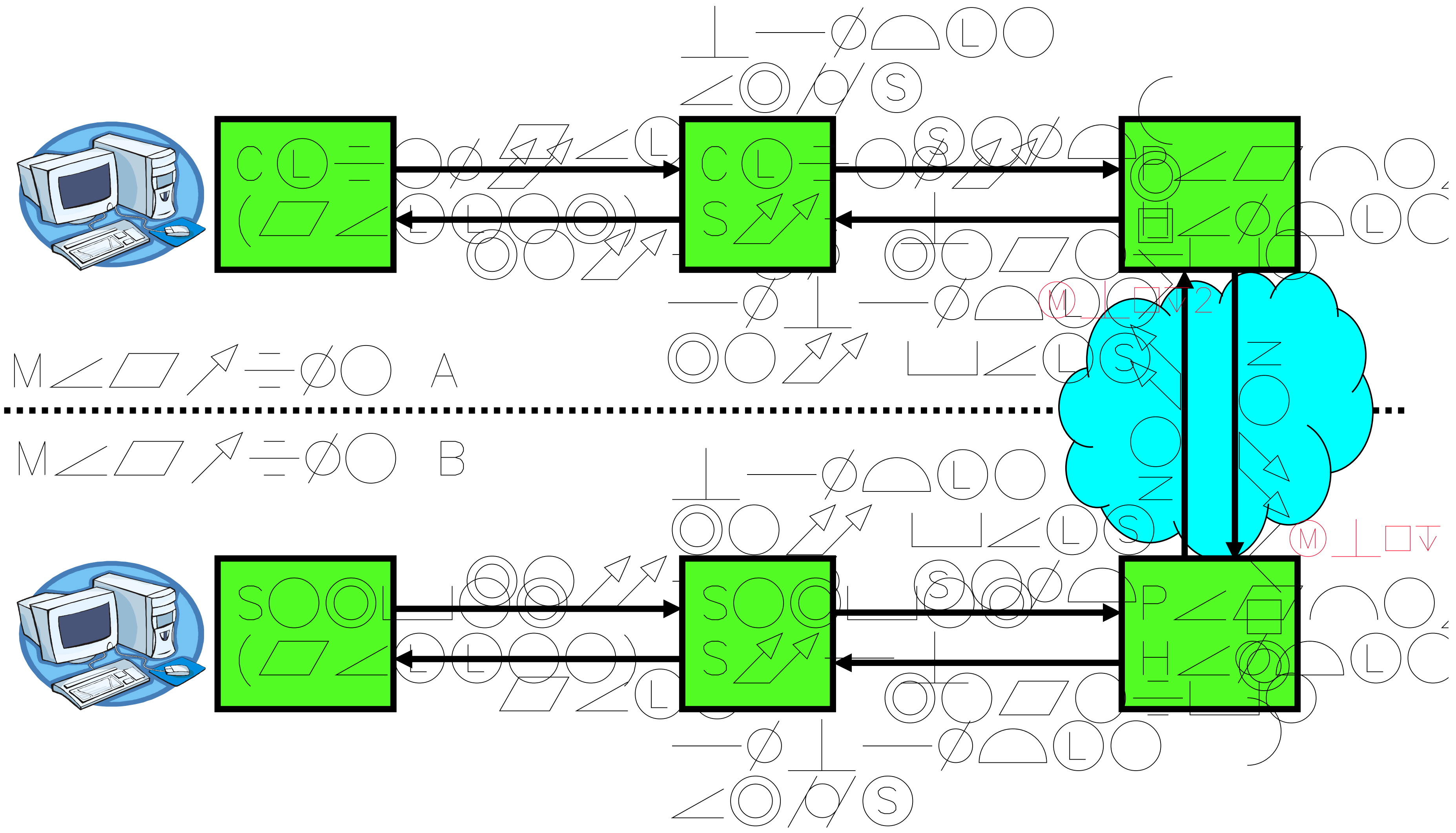
But it’s not always simple

- Calling and called procedures run on different machines, with different address spaces
  - And perhaps different environments .. or operating systems ..
- Must convert to local representation of data
- Machines and network can fail

Stubs: obtaining transparency

- Compiler generates from API stubs for a procedure on the client and server
- Client stub
  - **Marshals** arguments into machine-independent format
  - Sends request to server
  - Waits for response
  - **Unmarshals** result and returns to caller
- Server stub
  - **Unmarshals** arguments and builds stack frame
  - Calls procedure
  - Server stub **marshals** results and sends reply

RPC Information Flow



RPC steps

- Step 1: Client calling client stub
  - Normal procedure call with parameters
- Step 2: the client stub packing the parameters in the message and making a system call to send the message
  - Packing parameters is called marshalling
- Step 3: Kernel sending the message from the client machine to the server machine
- Step 4: the Kernel passing the incoming packet to Server stub
- Step 5: the Server stub calling the server procedure
- Reply traces the same path

A few points !

- Client stub represents server procedure in client address space
- Client procedure is a normal procedure which calls client stub
  - Client stub has the same name as the server procedure
- Is Address Space an issue ?
  - Client procedure and client stub are in the same address space – parameters are passed in the usual way
  - The server procedure and and server stub are in the same address space

- How to pass a pointer (as a parameter) to a remote procedure ?
- How to handle Global variables?

RPC Details (self study) – self study

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)

- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an “interface definition language (IDL)”
    - Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - Code for server to unpack message, call procedure, pack results, send them off

RPC Details (2/3) – self study

- Cross-platform issues:
  - What if client/server machines are different architectures/ languages?
    - Convert everything to/from some canonical form
    - Tag every item with an indication of how it is encoded (avoids unnecessary conversions)

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    - This is another word for “naming” at network level
    - Static: fixed at compile time
    - Dynamic: performed at runtime

RPC Details (3/3) – self study

- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - Name service provides dynamic translation of service $\rightarrow$ mbox
  - Why dynamic binding?
    - Access control: check who is permitted to access service
    - Fail-over: If server fails, use a different one
- What if there are multiple servers?
  - Could give flexibility at binding time
    - Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - Choose unloaded server for each new request
    - Only works if no state carried from one call to next
- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Distributed File System

Distributed System Protocols are Built by Message Passing

- Sending/receiving messages is atomic
  - Each message is either fully received exactly once…
  - or not received at all (!)

- Interface:
  - Mailbox: temporary holding area for messages
    - Includes both destination location and queue
  - `Send(message,mbox)`
    - Send message to remote mailbox identified by `mbox`
  - `Receive(buffer,mbox)`
    - Wait until `mbox` has message, copy into buffer, and return
    - If threads sleeping on this mbox, wake up one of them

But, doesn't TCP give us reliable delivery?

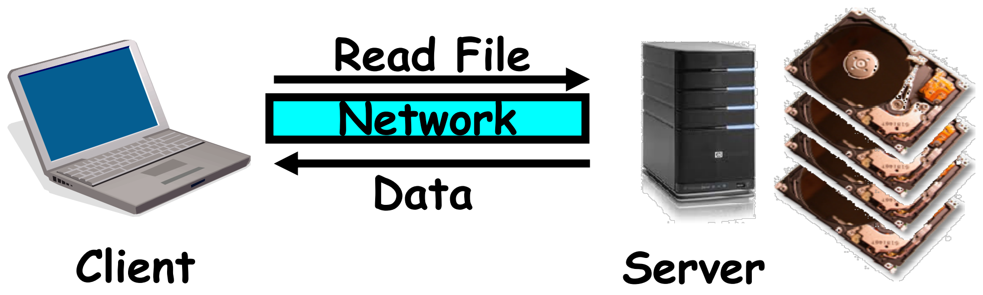
- TCP provides a convenient interface to use an unreliable network…
- … but it does *not* make the network reliable!

- Messages can still be lost if you use TCP
  - After many retransmissions, the OS "gives up" and breaks the connection

- Losing messages is fundamental problem in distributed systems
  - TCP's retransmissions turn packet losses into packet delays (even if it never "gives up")
  - ***And very long delays look just like losses!***
  - TCP makes the network easy to use, and it can help improve performance
  - But TCP doesn't solve this fundamental problem (losing messages)

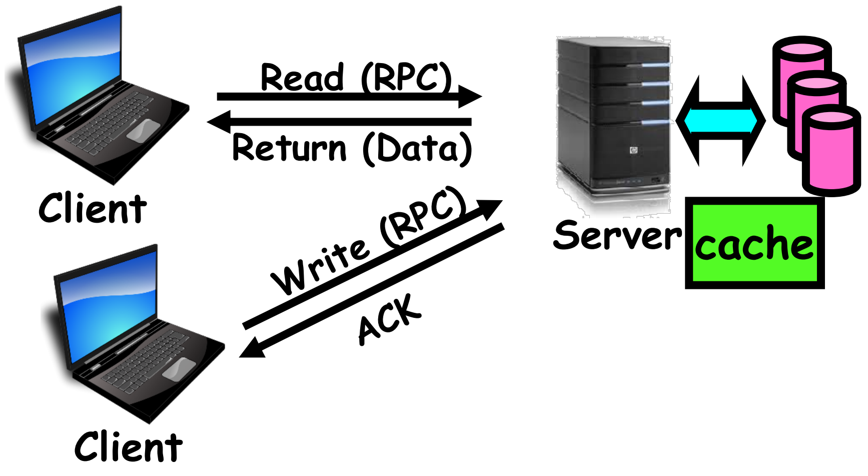
Distributed File Systems

- Transparent access to files stored on a remote disk
- *Mount* remote files into your local file system
  - Directory in local file system refers to remote files
  - e.g., `/home/oksi/162/` on laptop actually refers to `/users/oski` on campus file server



Simple Distributed File System

- Remote Disk: Opens, Reads, Writes, Closes forwarded to server
  - Use Remote Procedure Calls (RPC) to translate file system calls into remote requests
  - Server may cache files in memory to response more quickly
  - Server provides consistent view of file system to multiple clients
- Problem: performance (network slower than memory, server is bottleneck)



Lecture Summary

- Distributed systems are becoming very important in the present interconnected world
- Remote Procedure Call (RPC): Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)
- Socket Programming
  - Connection involves 5 values:
    [Client Addr, Client Port, Server Addr, Server Port, Protocol]