CS 318 Principles of Operating Systems

Fall 2017

Lecture 20: Distributed Systems

Ryan Huang

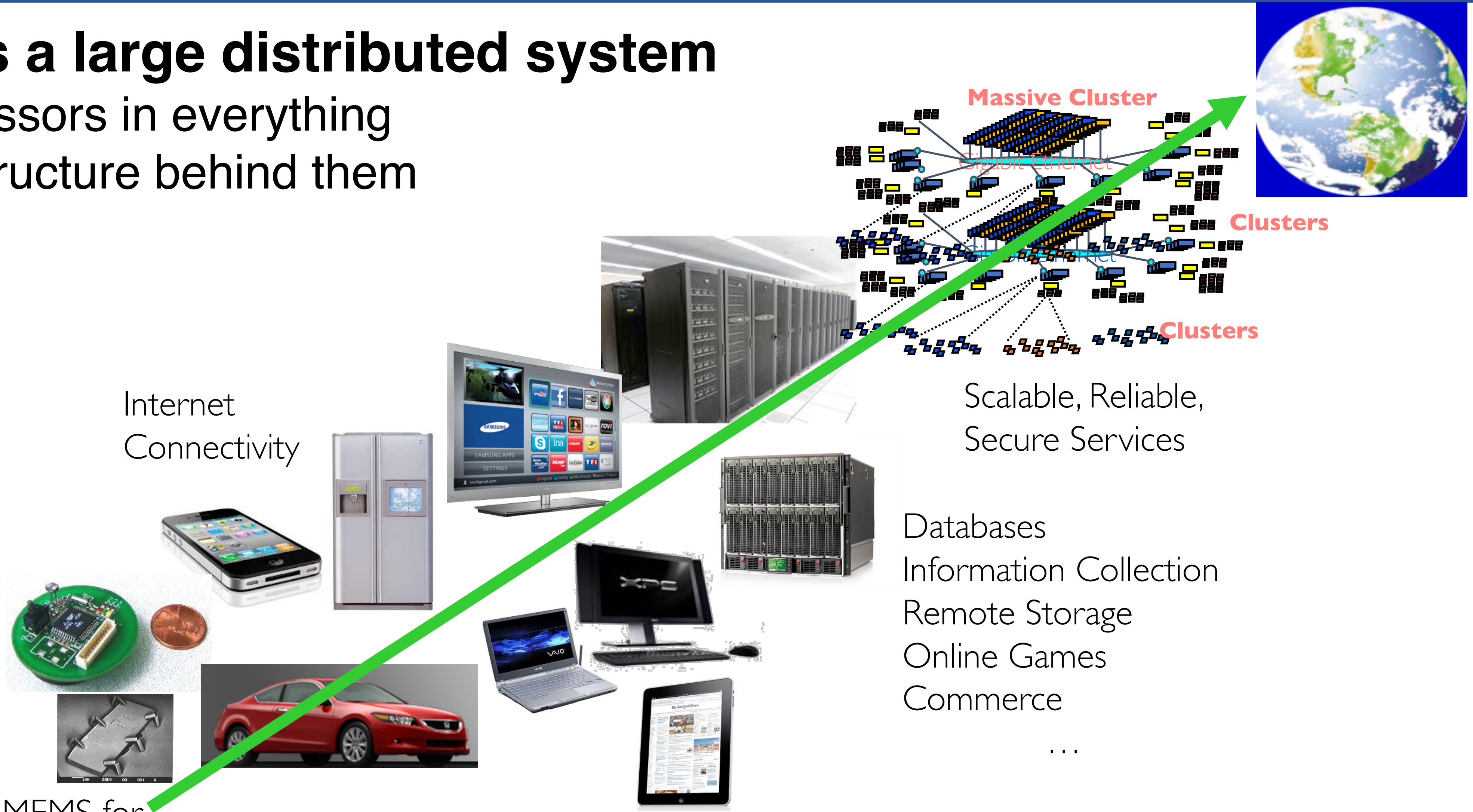
Johns Hopkins
Whiting School of Engineering

Preview

- **Next three lectures are advanced topics on systems in general**
  - Each topic has enough depth to be covered in an entire course by itself
  - We will only cover the high-level basics
  - Focus on abstractions and generic systems techniques
- **Today: distributed systems**
  - What is a distributed system?
  - What are the basic concepts essential to build a distributed system?
  - Examine an important abstraction: Remote Procedure Call (RPC)

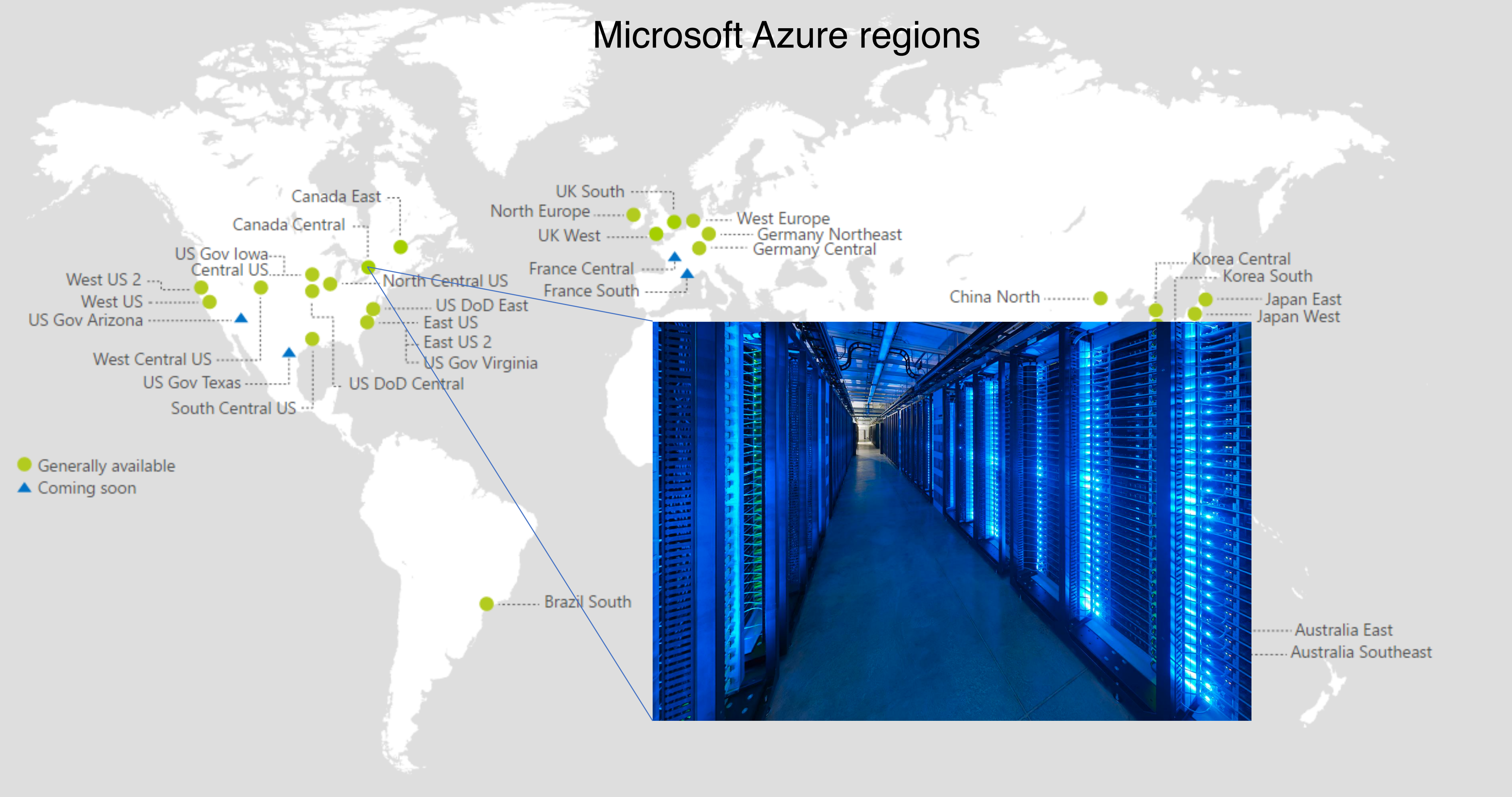
Societal Scale Information Systems

- **The world is a large distributed system**
  - Microprocessors in everything
  - Vast infrastructure behind them




DELL

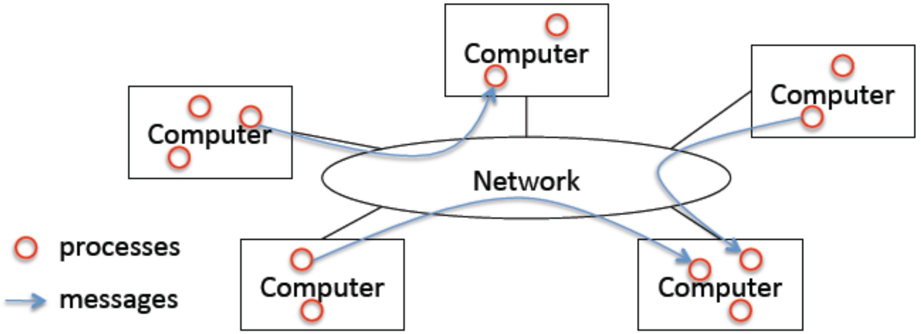
Today



Microsoft Azure regions
Canada East
Canada Central
US Gov Iowa
Central US
West US 2
West US
US Gov Arizona
North Central US
US DoD East
East US
East US 2
US Gov Virginia
West Central US
US Gov Texas
South Central US
US DoD Central
UK South
North Europe
UK West
West Europe
Germany Northeast
Germany Central
France Central
France South
China North
Korea Central
Korea South
Japan East
Japan West
Brazil South
Australia East
Australia Southeast
Generally available
Coming soon

What is a Distributed System?

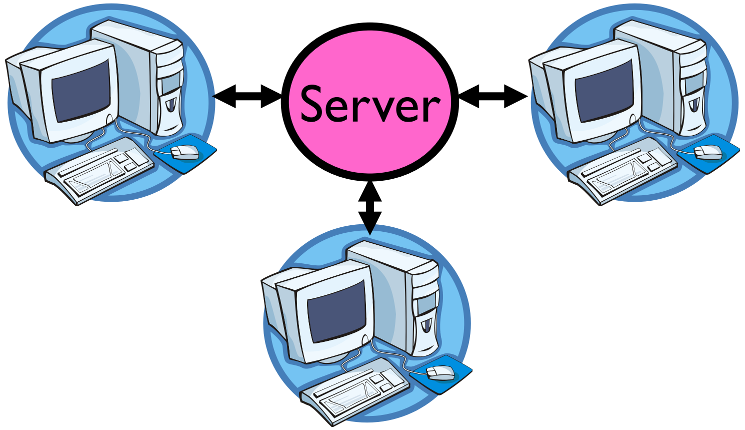
- **Cooperating processes in a computer network**



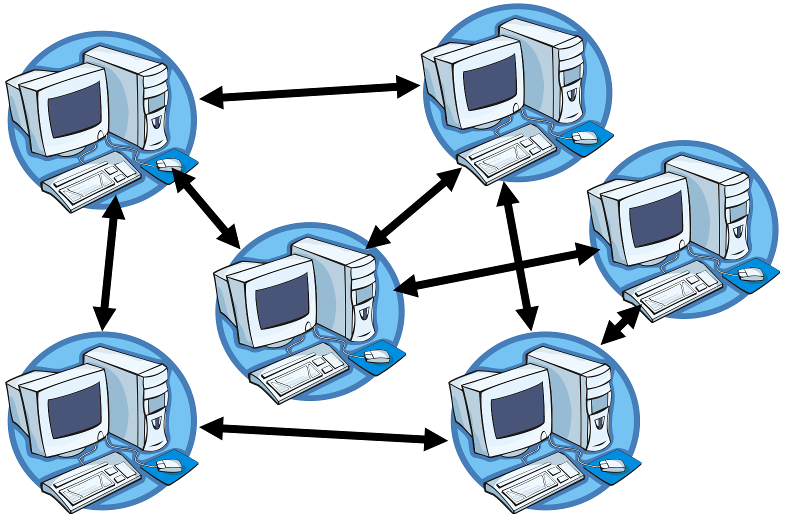
- **Degree of integration**
  - Loose: Internet applications, email, web browsing
  - Medium: remote execution, remote file systems
  - Tight: distributed file systems

- **Popular distributed systems today**
  - Google file systems, BigTable, MapReduce, Hadoop, ZooKeeper, etc.

Centralized vs Distributed Systems



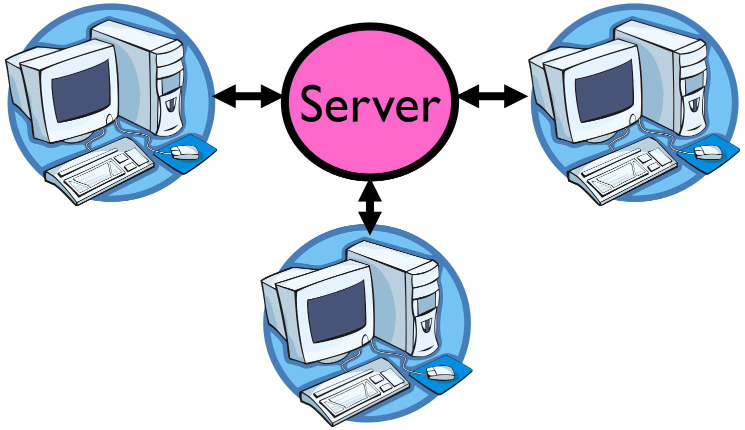
Client/Server Model



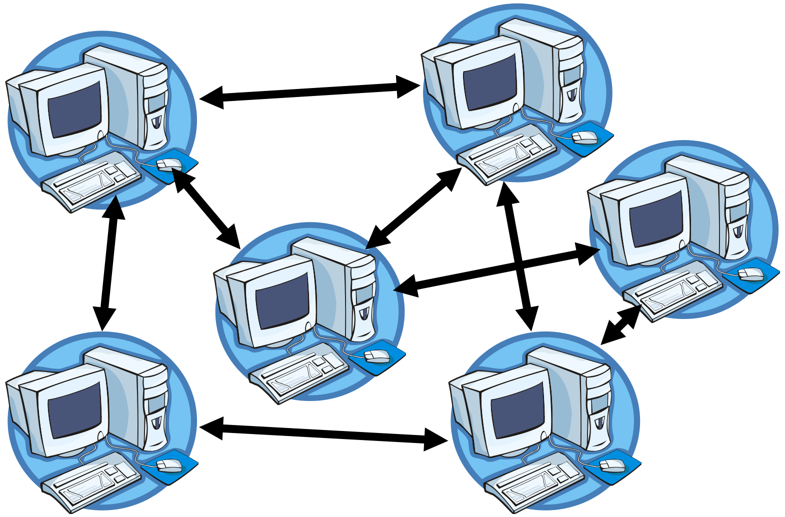
Peer-to-Peer Model

- **Centralized System: System in which major functions are performed by a single physical computer**
  - Originally, everything on single computer
  - Later: client/server model

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer Model

- **Distributed System: physically separate computers working together on some task**
  - Early model: multiple servers working together
    - Probably in the same room or building
    - Often called a "cluster"
  - Later models: peer-to-peer/wide-spread collaboration

Distributed Systems: Motivation

- **Why do we want distributed systems?**
  - Performance: parallelism across multiple nodes
  - Scalability: by adding more nodes
  - Reliability: leverage redundancy to provide fault tolerance
  - Cost: cheaper and easier to build lots of simple computers
  - Control: users can have complete control over some components
  - Collaboration: much easier for users to collaborate through network resources

- **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

Distributed Systems: Reality

- **Reality has been disappointing**
  - Worse availability: depend on every machine being up
    - Lamport: "a distributed system is one where I can't do work because some machine I've never heard of isn't working!"
  - Worse reliability: can lose data if any machine crashes
  - Worse security: anyone in world can break into system
- **Coordination is more difficult**
  - Must coordinate multiple copies of shared state information (using only a network)
  - What would be easy in a centralized system becomes a lot more difficult

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: May speed up large jobs by splitting them into smaller pieces
  - Fault Tolerance: System may hide various things that go wrong
- **Transparency and collaboration require some way for different processors to communicate with one another**

Clients and Servers

- **The prevalent model for structuring distributed computation is the client/server paradigm**
- **A server is a program (or collection of programs) that provide a service (file server, name service, etc.)**
  - The server may exist on one or more nodes
  - Often the node is called the server, too, which is confusing
- **A client is a program that uses the service**
  - A client first binds to the server (locates it and establishes a connection to it)
  - A client then sends requests, with data, to perform actions, and the servers sends responses, also with data

Naming

- **Name systems in network**
  - often hierarchical name. cs.jhu.edu is *domain*
- **Network Address (Internet IP address)**
  - 192.17.4.131 -- 192.17.4.**
  - 128.174.240.**
- **Physical Network Address**
  - Ethernet address or Token Ring Address
- **Address processes/ports within system (host, id) pair**
- **Domain name service (DNS) specifies naming structure of hosts and provides resolution of names to network address**

Communication

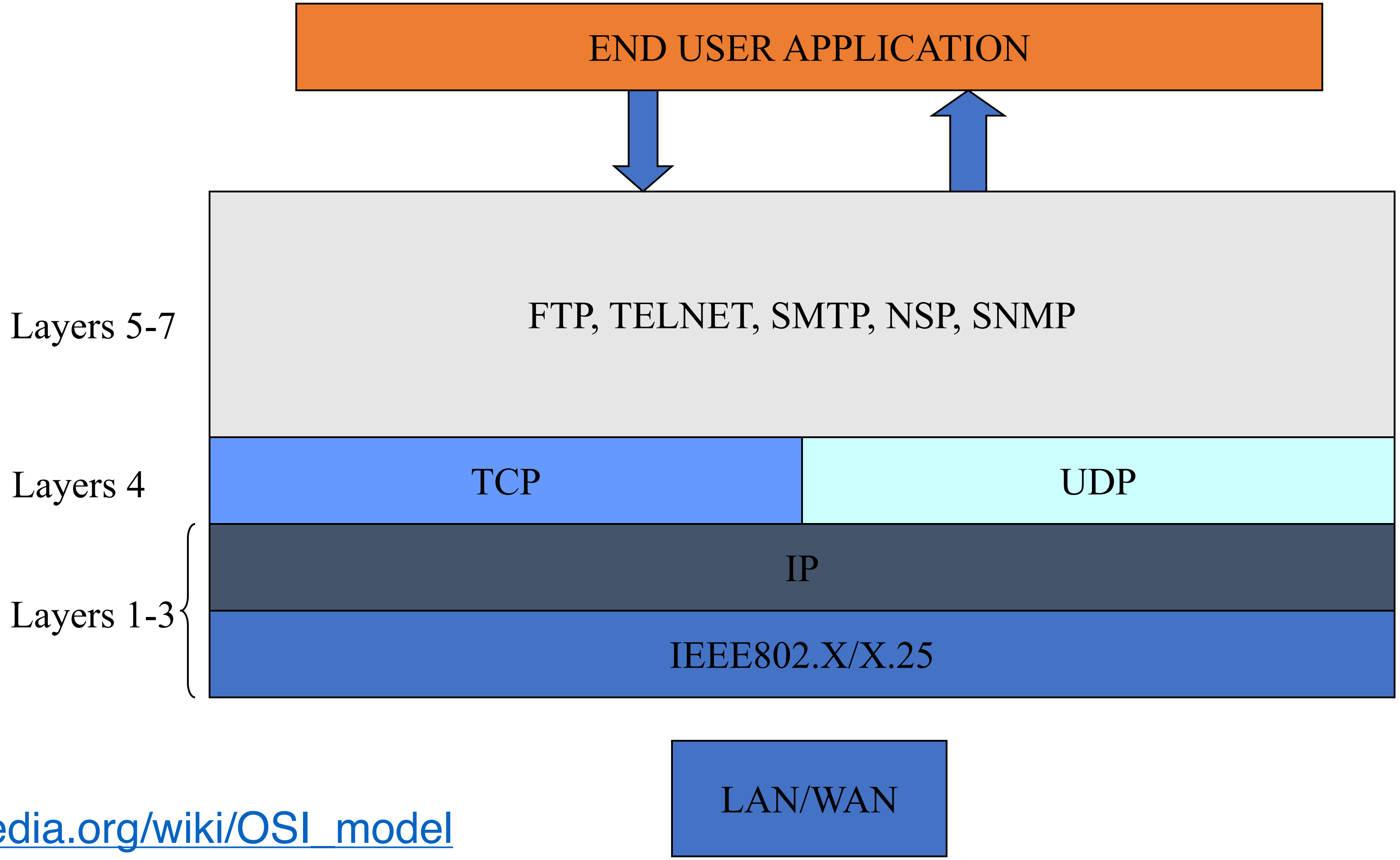
- **Socket (TCP/IP)**

- **Remote Procedure Call (RPC) /Remote Method Invocation(RMI)**

TCP/IP (Socket)

- **Transport Protocols**
  - User Datagram Protocol (UDP)
    - UDP/IP is an unreliable, connectionless transport protocol, which uses IP to transport IP datagrams but adds error correction and a protocol port address to specify the process on the remote system for which the packet is destined.
  - Transmission Control Protocol (TCP)
    - TCP/IP is a reliable stream protocol for communicating information between two processes

TCP/IP Protocol Layers



https://en.wikipedia.org/wiki/OSI_model

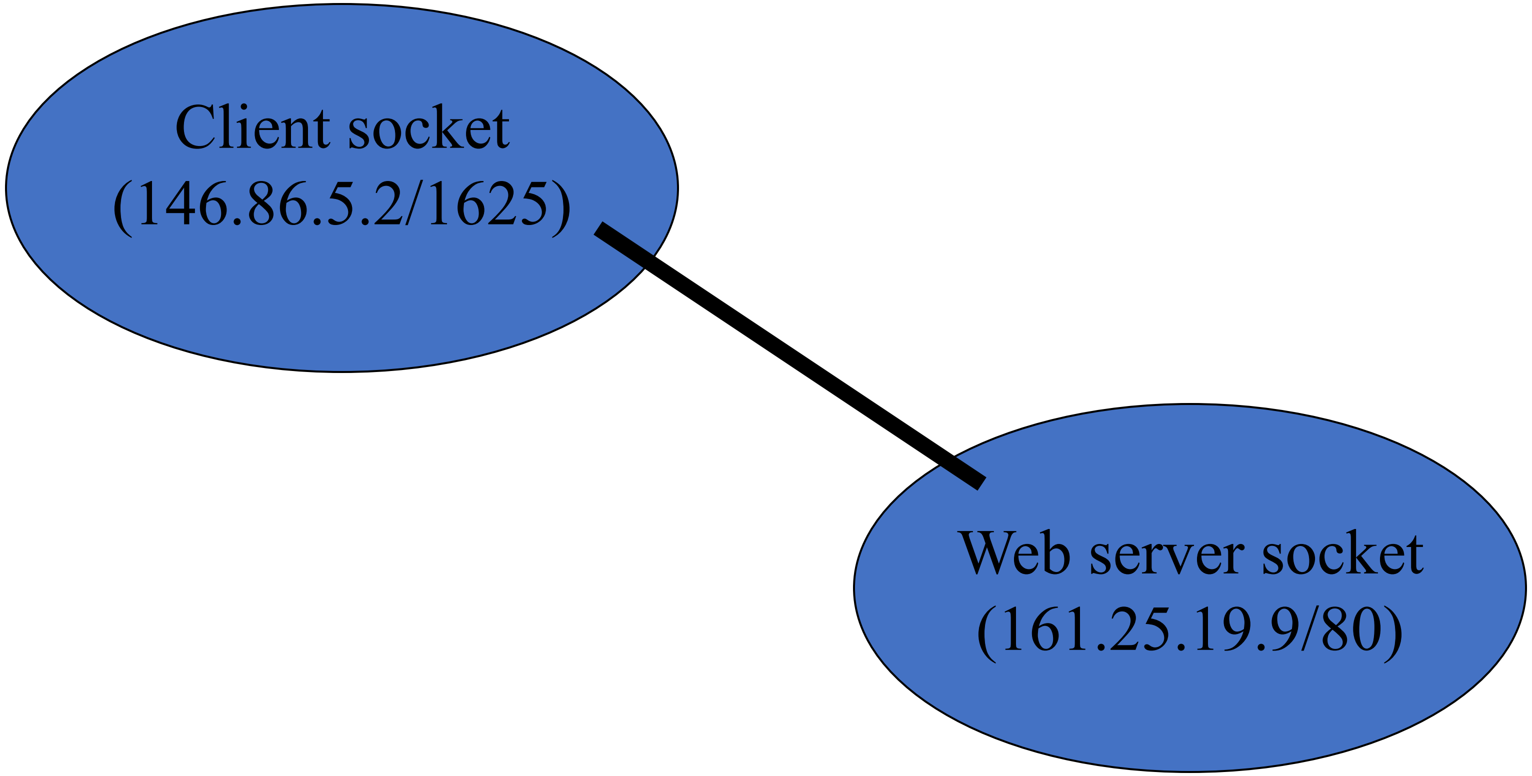
TCP Sockets

- **Communication endpoint**
  - (IP address, Port number)
- **Client-server**
  - server listens to a port
- **Telnet port 23, ftp port 21, web server port 80**

TCP/IP Ports

- **Ports < 1024, standard**
- **Ports > 1024, user created**
- **All connections unique**
  - 161.25.19.8:20
  - IP Address: 161.25.19.8
  - TCP/IP Port: 20 (ftpdata)
  - http://www.iana.org/assignments/port-numbers

TCP Socket Communication



Raw Messaging

- **Initially network programming = raw messaging**
  - Programmers hand-coded messages to send requests and responses
- **Problem: too low-level and tiresome**
  - Need to worry about message formats
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - Have to pack and unpack data from messages
  - May need to sit and wait for multiple messages to arrive
- **Messages are not a very natural programming model**
  - Could encapsulate messaging into a library
  - Just invoke library routines to send a message
  - Which leads us to RPC…

Procedure Calls

- **Procedure calls are a more natural way to communicate**
  - Every language supports them
  - Semantics are well-defined and understood
  - Natural for programmers to use
- **Idea: let servers export procedures that can be called by client programs**
  - Similar to module interfaces, class definitions, etc.
  - Clients just do a procedure call as it they were directly linked with the server
  - Under the covers, the procedure call is converted into a message exchange with the server

Remote Procedure Calls

- **So, we would like to use procedure call as a model for distributed (remote) communication**
- **Lots of issues**
  - How do we make this invisible to the programmer?
  - What are the semantics of parameter passing?
  - How do we bind (locate, connect to) servers?
  - How do we support heterogeneity (OS, arch, language)?
  - How do we make it perform well?

Why is RPC Interesting?

- **Remote Procedure Call (RPC) is the most common means for remote communication**
- **It is used both by operating systems and applications**
  - NFS is implemented as a set of RPCs
  - DCOM, CORBA, Java RMI, etc., are all basically just RPC
- **Someday (soon?) you will most likely have to write an application that uses remote communication (or you already have)**
  - You will most likely use some form of RPC for that remote communication
  - So it's good to know how all this RPC stuff works
    - More "debunking the magic"

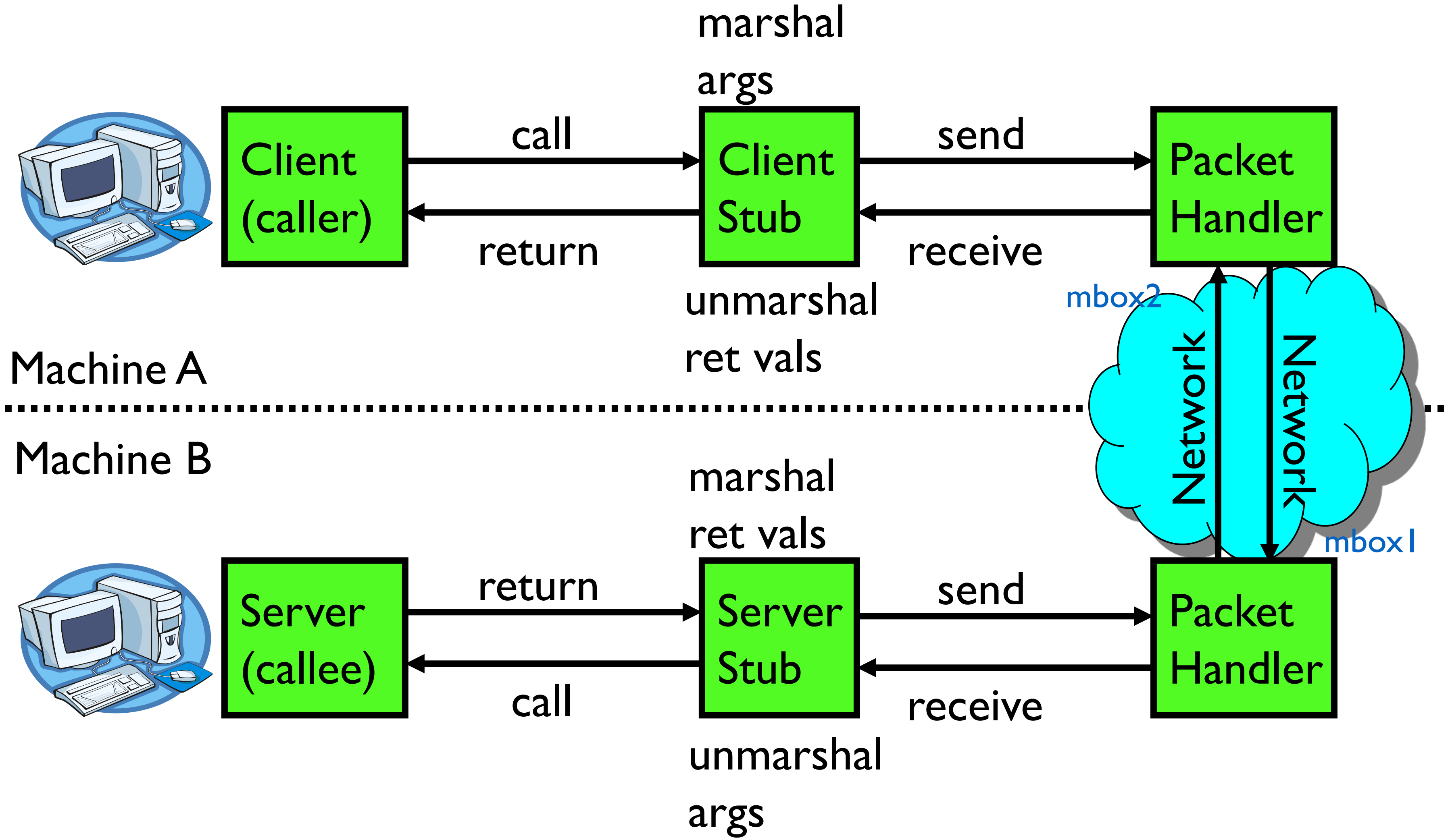
RPC Model

- **A server defines the server's 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 and produces two stub procedures for each server procedure (client and server)**
  - Server programmer implements the server procedures and links them with server-side stubs
  - Client programmer implements the client program and links it with client-side stubs
  - The stubs are the *"glues"* responsible for managing all details of the remote communication between client and server

RPC Stubs

- **A client-side stub is a procedure that looks to the client as if it were a callable server procedure**
  - Task: pack message, send it off, wait for result, unpack result and return to caller
- **A server-side stub looks to the server as if a client called it**
  - Task: unpack message, call procedure, pack results, send them off
- **The client program thinks it is calling the server**
  - In fact, it's calling the client stub
- **The server program thinks it is called by the client**
  - In fact, it's called by the server stub
- **The stubs send messages to each other to make RPC happen transparently**

RPC Information Flow



RPC Example

Client Program:

```
…
sum = server->Add(3,4);
…
```

Server Interface:

```
int Add(int x, int y);
```

Server Program:

```
int Add(int x, int y) {
  return x + y;
}
```

- **If the server were just a library, then `Add` would just be a procedure call**

RPC Example: Call

Client Program:

```
sum = server->Add(3,4);
```

Client Stub:

```
int Add(int x, int y) {
  Alloc message buffer;
  Mark as "Add" call;
  Store x, y into buffer;
  Create, send message;
}
```

RPC Runtime:

```
Send message to server;
```

RPC Runtime:

```
Receive message;
Dispatch, call Add_Stub;
```

Server Stub:

```
Add_Stub(Message) {
  Remove x, y from buffer
  r = Add(x, y);
}
```

Server Program:

```
int Add(int x, int y){
 return x + y;
}
```

RPC Example: Return

Client Program:

```
sum = server->Add(3,4);
```

Client Stub:

```
int Add(int x, int y) {
  Alloc message buffer;
  Mark as "Add" call;
  Store x, y into buffer;
  Create, send message;
  Remove r from reply;
  return r;
}
```

RPC Runtime:

```
Return reply to stub;
```

Server Program:

```
int Add(int x, int y){
 return x + y;
}
```

Server Stub:

```
Add_Stub(Message) {
  Remove x, y from buffer
  r = Add(x, y);
  Store r in buffer;
}
```

RPC Runtime:

```
Send reply to client;
```

RPC Marshalling

- **Marshalling is the packing of procedure parameters into a message packet**
- **The RPC stubs call type-specific procedures to marshal (or unmarshal) the parameters to a call**
  - The client stub marshals the parameters into a message
  - The server stub unmarshals parameters from the message and uses them to call the server procedure
- **On return**
  - The server stub marshals the return parameters
  - The client stub unmarshals return parameters and returns them to the client program

RPC Implementation Details

- **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 server 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 Binding (1)

- **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 it's 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 a server and establish a connection

- **The import and export operations are explicit in the server and client programs**
  - Breakdown of transparency

RPC Example in Go Including Binding

```
type Args struct {
      A, B int
}
type Arith int
```

Client Program:

```
client, err := rpc.DialHTTP("tcp",
        serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
    log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)
```

Server Program:

```
func (t *Arith) Multiply(args *Args,
        reply *int) error {
    *reply = args.A * args.B
    return nil
}
func main() {
    arith := new(Arith)
    rpc.RegisterName("Arithmetic", arith)
    rpc.HandleHTTP()
    l, e := net.Listen("tcp", ":1234")
    if e != nil {
        log.Fatal("listen error:", e)
    }
    http.Serve(l, nil)
}
```

RPC Binding (2)

- **Dynamic Binding**
  - Most RPC systems use dynamic binding via name service
    - Name service provides dynamic translation of service → 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

RPC Transparency

- **One goal of RPC is to be as transparent as possible**
  - Make remote procedure calls look like local procedure calls
- **We have seen that binding breaks transparency**
- **What else?**
  - Failures – remote nodes/networks can fail in more ways than with local procedure calls
    - Need extra support to handle failures well
  - Performance – remote communication is inherently slower than local communication
    - If program is performance-sensitive, could be a problem

Problems with RPC: Non-Atomic Failures

- **Different failure modes in dist. system than on a single machine**
- **Consider many different types of failures**
  - User-level bug causes address space to crash
  - Machine failure, kernel bug causes all processes on same machine to fail
  - Some machine is compromised by malicious party
- **Before RPC: whole system would crash/die**
- **After RPC: One machine crashes/compromised while others keep working**
- **Can easily result in inconsistent view of the world**
  - Did my cached data get written back or not?
  - Did server do what I requested or not?
- **Answer? Distributed transactions/Byzantine Commit**

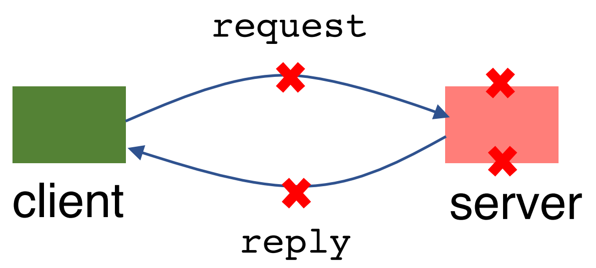
Problems with RPC: Performance

- **Cost of Procedure call ≪ same-machine RPC ≪network RPC**

- **Means programmers must be aware that RPC is not free**
  - Caching can help, but may make failure handling complex

RPC Failure Semantic (1)

- **What does a failure look like to the client RPC library?**
  - Client never sees a response from the server
  - Client does *not* know if the server saw the request
    - Maybe server/net failed just before sending reply



- **Simplest scheme: at-least-once behavior**
  - RPC library waits for response for time `T`, if none arrives, re-send the request
  - Repeat this a few times
  - Still no response → return an error to the application

- **Problem with at-least-once behavior?**
  - E.g., request is "deduct $100 from bank account"
  - What about this sequence?: `v = get(key); put(key, v - 100);` ✖ `put(key, v);`

https://pdos.csail.mit.edu/6.824/notes/l-rpc.txt

RPC Failure Semantic (2)

- **When is at-least-once behavior *OK*?**
  - If it's ok to repeat an operation, e.g., `get(key);`
  - If the application has its own way of dealing with duplicates
- **Another (better) RPC behavior: at most once**
  - **Idea:** server RPC code detects duplicate requests returns previous reply instead of re-running handler
  - How to detect a duplicate request?
    - client includes unique ID (XID) with each request, and uses the same XID for re-send
    - server checks an incoming XID in a table, if an entry is found, directly returns the reply

https://pdos.csail.mit.edu/6.824/notes/l-rpc.txt

RPC Failure Semantic (3)

- **Some complexities about implementing at-most-once**
  - How to ensure XID is unique?
  - Server must eventually discard info about old RPCs, when is it safe to discard?
  - How to handle duplicate request while original is still executing?
- **What if an at-most-once server crashes and re-starts?**
  - If duplicate info is in memory, server will forget and accept duplicate requests after re-start
  - It could write the duplicate info to disk
  - Replica server could also replicate duplicate info
- **What about "exactly once"?**
  - at-most-once plus unbounded retries plus fault-tolerant service
- **RPC semantics beyond two entities**
  - Master sends RPC to a worker, worker doesn't respond, master re-send to another worker
    - original worker may have not failed, and is working on it too

https://pdos.csail.mit.edu/6.824/notes/l-rpc.txt

RPC Summary

- **RPC is the most common model for communication in distributed applications**
  - "Cloaked" as DCOM, CORBA, Java RMI, etc.
  - Some popular libraries: gRPC, Golang RPC
  - Also used on same node between applications (e.g., gRPC)
- **RPC is essentially language support for distributed programming**
- **RPC relies upon a stub compiler to automatically generate client/server stubs from the IDL server descriptions**
  - These stubs do the marshalling/unmarshalling, message sending/receiving/replying
- **At-least-once, at-most-once, exactly-once RPC failure semantic**
- **NFS uses RPC to implement remote file systems**

Next Time…

- **Mobile Systems**