CS 318 Principles of Operating Systems

Fall 2020

Lecture 19: Mobile & Distributed Systems

Prof. Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Administrivia

- **Next week Thanksgiving break**
  - No class
  - Assignments
    - food, lots of it
    - sleep, lots of it
    - warm clothes, winter is coming
    - **Stay safe**
    - Pintos (?😅)

Preview

- **Next two lectures again preview advanced systems topics**
  - Each topic has enough depth to be covered in an entire course by itself
  - We will only cover basic concepts
- **Today: mobile & distributed systems**
  - History of mobile device and OS
  - Mobile OS vs. traditional OS
  - How does Android OS work?
  - What is a distributed system?
  - What are the basic concepts essential to build a distributed system?

Mobile Devices Become Ubiquitous

Google Nexus 6P

NETWORK	Technology	GSM / CDMA / HSPA / LTE
DISPLAY	Type	AMOLED capacitive touchscreen, 16M colors
	Size	5.7 inches (~71.4% screen-to-body ratio)
	Resolution	1440 x 2560 pixels (~518 ppi pixel density)
	Multitouch	Yes
	Protection	Corning Gorilla Glass 4, oleophobic coating
PLATFORM	OS	Android OS, v6.0 (Marshmallow)
	Chipset	Qualcomm MSM8994 Snapdragon 810
	CPU	Quad-core 1.55 GHz Cortex-A53 & Quad-core 2.0 GHz Cortex-A57
	GPU	Adreno 430
MEMORY	Card slot	No
	Internal	32/64/128 GB, 3 GB RAM
CAMERA	Primary	12.3 MP, f/2.0, laser autofocus, dual-LED (dual tone) flash, check quality
	Features	1/2.3' sensor size, 1.55µm pixel size, geo-tagging, touch focus, face detection, HDR, panorama
	Video	2160p@30fps, 720p@240fps, check quality
	Secondary	8 MP, f/2.4, 1080p@30fps
SOUND	Alert types	Vibration; MP3, WAV ringtones
	Loudspeaker	Yes, with front stereo speakers
	3.5mm jack	Yes
COMMS	WLAN	Wi-Fi 802.11 a/b/g/n/ac, dual-band, Wi-Fi Direct, DLNA, hotspot
	Bluetooth	v4.2, A2DP, LE
	GPS	Yes, with A-GPS, GLONASS
	NFC	Yes
	Radio	No
	USB	v2.0, Type-C 1.0 reversible connector
FEATURES	Sensors	Fingerprint, accelerometer, gyro, proximity, compass, barometer
	Messaging	SMS(threaded view), MMS, Email, Push Mail, IM
	Browser	HTML5
	Java	No
		- Fast charging - Active noise cancellation with dedicated mics - MP4/H.264 player - MP3/WAV/eAAC+ player - Photo/video editor - Document editor
BATTERY		Non-removable Li-Po 3450 mAh battery

History of Mobile OS (1)

- **Early "smart" devices are PDAs (touchscreen, Internet)**
- **Symbian, first modern mobile OS**
  - released in 2000
  - run in Ericsson R380, the first 'smartphone' (mobile phone + PDA)
  - only support proprietary programs

History of Mobile OS (2)

- **Many smartphone and mobile OSes followed up**
  - Kyocera 6035 running Palm OS (2001)
  - Windows CE (2002)
  - Blackberry (2002)
    - was a prominent vendor
    - known for secure communications
  - Moto Q (2005)
  - Nokia N70 (2005)
    - 2-megapixel camera, bluetooth
    - 32 MB memory
    - Symbian OS
    - Java games

One More Thing…

- **Introduction of iPhone (2007)**
  - revolutionize the smartphone industry
  - 4GB flash memory, 128 MB DRAM, multi-touch interface
  - runs iOS, initially only proprietary apps
  - App Store opened in 2008, allow third party apps

Android – An Unexpected Rival of iPhone

- **Android Inc. founded by Andy Rubin et al. in 2003**
  - original goal is to develop an OS for digital camera
  - shift focus on Android as a mobile OS
- **The startup had a rough time [story]**
  - run out of cash, landlord threatens to kick them out
  - later bought by Google
  - no carrier wants to support it except for T-Mobile
  - while preparing public launch of Android, iPhone was released
- **Android 1.0 released in 2008 (HTC G1)**

- **Today: ~88% of mobile OS market**
  - iOS ~11%

Android Releases

Why Are Mobile OSes Interesting?

- **They are running in every mobile device as an essential part of people's daily life, even for non-technical users**
  - In many developing countries, the only computing device one has is a phone
- **Mobile OSes and traditional OSes share the same core abstractions but also have many unique designs**
  - Comparing and contrasting helps you understand the whole OS design space
- **It will make you a more efficient mobile user and developer**

Design Considerations for Mobile OS

- **Resources are very constrained**
  - Limited memory
  - Limited storage
  - Limited battery life
  - Limited processing power
  - Limited network bandwidth
  - Limited size
- **User perception are important**
  - Latency ≫ throughput
    - Users will be frustrated if an app takes several seconds to launch
- **Environment are frequently changing**
  - The whole point about being mobile
  - Cellular signals from strong to weak and then back to strong

Process Management in Mobile OS (1)

- **In desktop/server: an application = a process**
- **Not true in mobile OSes**
  - When you see an app present to you, doesn't mean an actual process is running
  - Multiple apps might share processes
  - An app might make use of multiple processes
  - When you "close" an app, the process might be still running
    - **Why?**
    - "*all applications are running all of the time*"
- **Different user-application interaction patterns**
  - Check Facebook for 1 min, switch to Reminder for 10s, Check Facebook again
  - Server: launch a job, waits for result

Process Management in Mobile OS (2)

- **Multitasking is a luxury in mobile OS**
  - Early versions of iOS don't allow multi-tasking
    - Not because the CPU doesn't support it, but because of battery life and limited memory
  - Only one app runs in the foreground, all other user apps are suspended
  - OS's tasks are multi-tasked because they are assumed to be well-behaving
  - Starting with iOS 4, the OS APIs allow multi-tasking in apps
    - But only available for a limited number of app types
- **Different philosophies among mobile OSes**
  - Android more liberal: apps are allowed to run in background
    - Define Service class, e.g., to periodically fetch tweets
    - When system runs low in memory, kill an app

Memory Management in Mobile OS

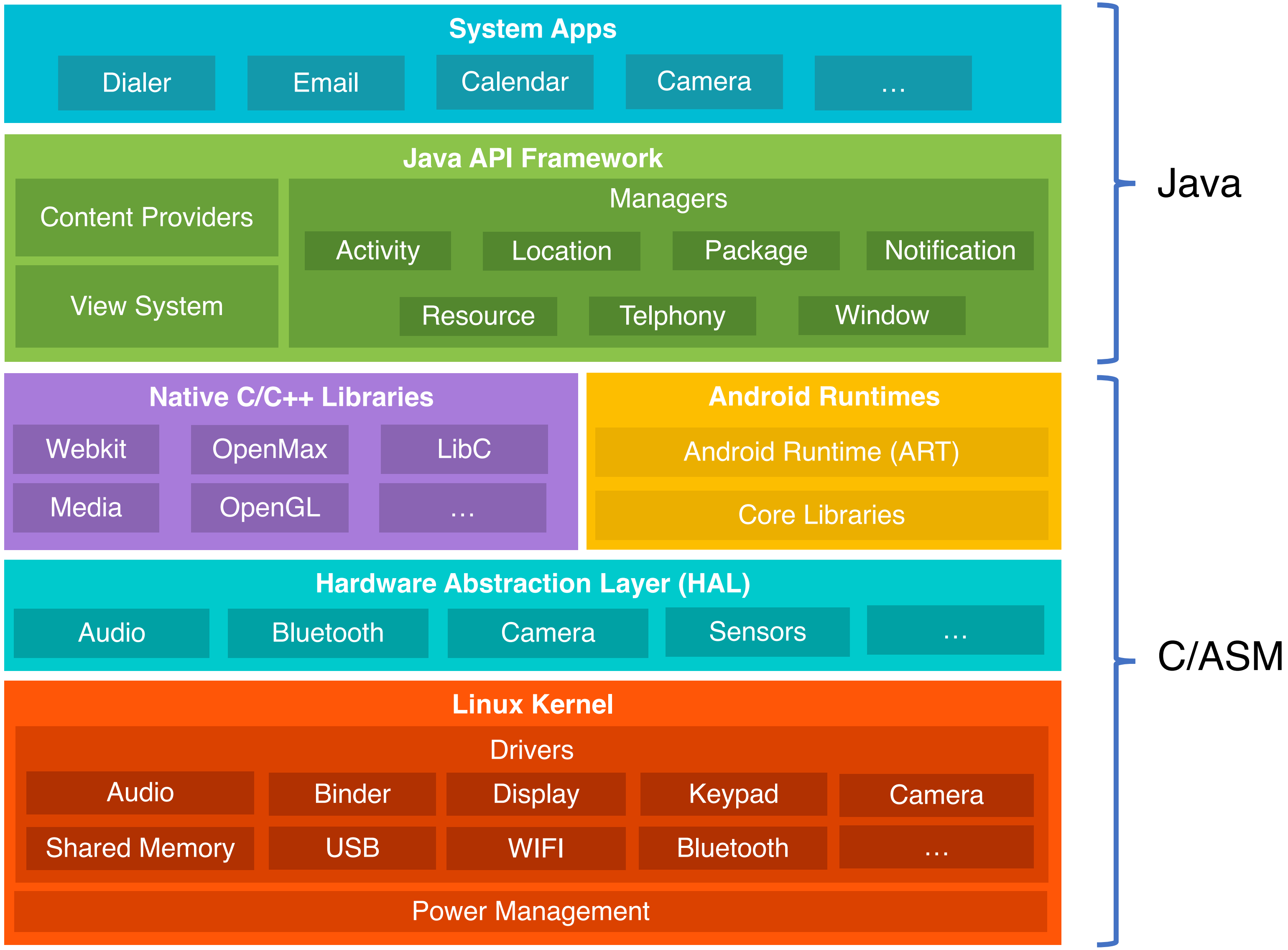
- **Most desktop and server OSes today support swap space**
  - Allows virtual memory to grow beyond physical memory size
  - When physical memory is full utilized, evict some pages to disk
- **Smartphones use flash memory rather than hard disk**
  - Capacity is very constrained: 16 GB vs. 512 GB
  - Limited number of writes in its lifetime
  - Poor throughput between main memory and flash memory
- **Mobile OSes typically don't support swapping!**
  - iOS ***asks*** applications to voluntarily relinquish allocated memory
  - Android will terminate an app when free memory is running low
- **App developers must be very careful about memory usage**

Storage in Mobile OS

- **App privacy and security is hugely important in mobile device**
  - Each app has its own private directory that other app can't access
  - Only shared storage is external storage
    - /sdcard/
- **High-level abstractions**
  - Files
  - Database (SQLite)
  - Preferences (key-value pairs)

```
ryan@orderlab:~$adb shell
shell@shamu:/ $ cd /data/app
shell@shamu:/data/app $ ls
opendir failed, Permission denied
255|shell@shamu:/data/app $ su
root@shamu:/data/app # ls
com.android.chrome-2
com.android.vending-2
com.facebook.katana-1
com.google.android.apps.docs.editors.docs-1
com.google.android.apps.maps-1
com.google.android.apps.messaging-1
com.google.android.gms-2
com.google.android.googlequicksearchbox-1
com.google.android.instantapps.supervisor-2
com.google.android.play.games-1
com.google.android.youtube-1
com.jumobile.manager.systemapp-1
com.ketchapp.stack-1
com.progames01.tanks.playtank-1
com.rovio.angrybirds-1
com.snapchat.android-1
edu.jhu.order.appstatstracker-1
```

Android OS Stack

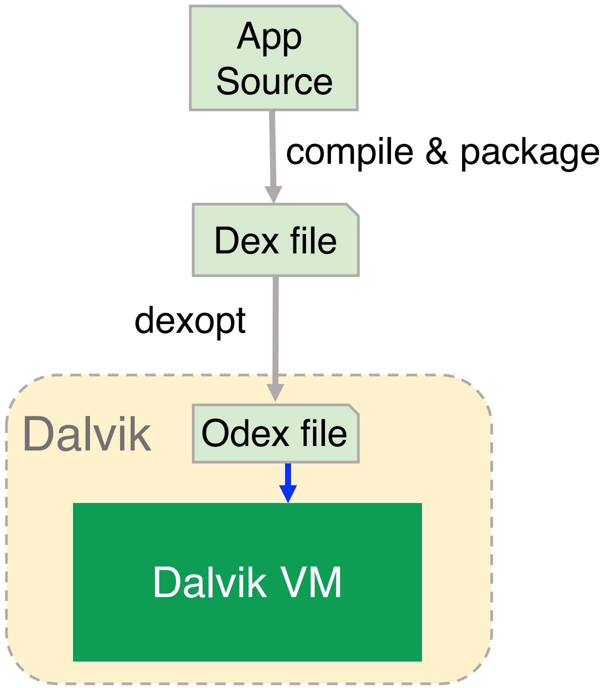


Linux Kernel vs. Android Kernel

- **Linux kernel is the foundation of Android platform**
- **New core code**
  - binder - interprocess communication mechanism
  - ashmem - shared memory mechanism
  - logger
- **Performance/power**
  - wakelock
  - low-memory killer
  - CPU frequency governor
- **and much more . . . [361 Android patches for the kernel]**

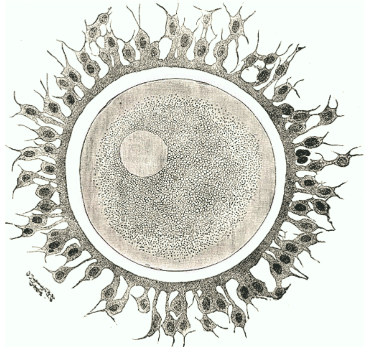
Android Runtime

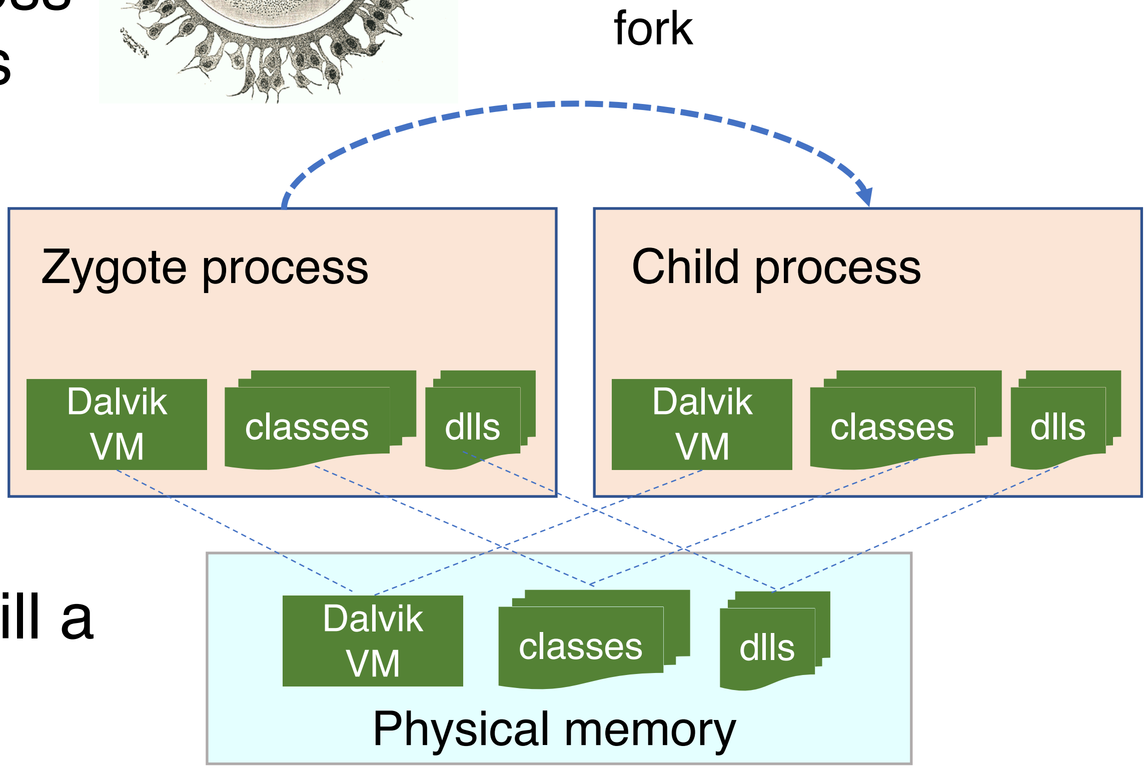
- **What is a runtime?**
  - A component provides functionality necessary for the execution of a program
    - E.g., scheduling, resource management, stack behavior
- **Prior to Android 5.0, Dalvik is the runtime**
  - Each Android app has its own process, runs its own instance of the Dalvik virtual machine (*process virtual machine*)
  - The VM executes the Dalvik executable (.dex) format
  - Register-based compared to stack-based of JVM
- **ART introduced in Android 5.0**
  - Backward compatible for running Dex bytecode
  - New feature: Ahead-of-time (AOT) compilation
  - Improved garbage collection



Android Runtime - Zygote

- **All Android apps derive from a process called Zygote**
  - Zygote is started as part of the init process
  - Preloads Java classes, resources, starts Dalvik VM
  - Registers a Unix domain socket
  - Waits for commands on the socket
  - Forks off child processes that inherit the initial state of VMs
- **Uses Copy-on-Write**
  - Only when a process writes to a page will a page be allocated





Java API Framework

- **The main Android “OS” from app point of view**
  - Provide high-level services and environment to apps
  - Interact with low-level libraries and Linux kernel
- **Example**
  - Activity Manager
    - Manages the lifecycle of apps
  - Package Manager
    - Keeps track of apps installed
  - Power Manager
    - Wakelock APIs to apps

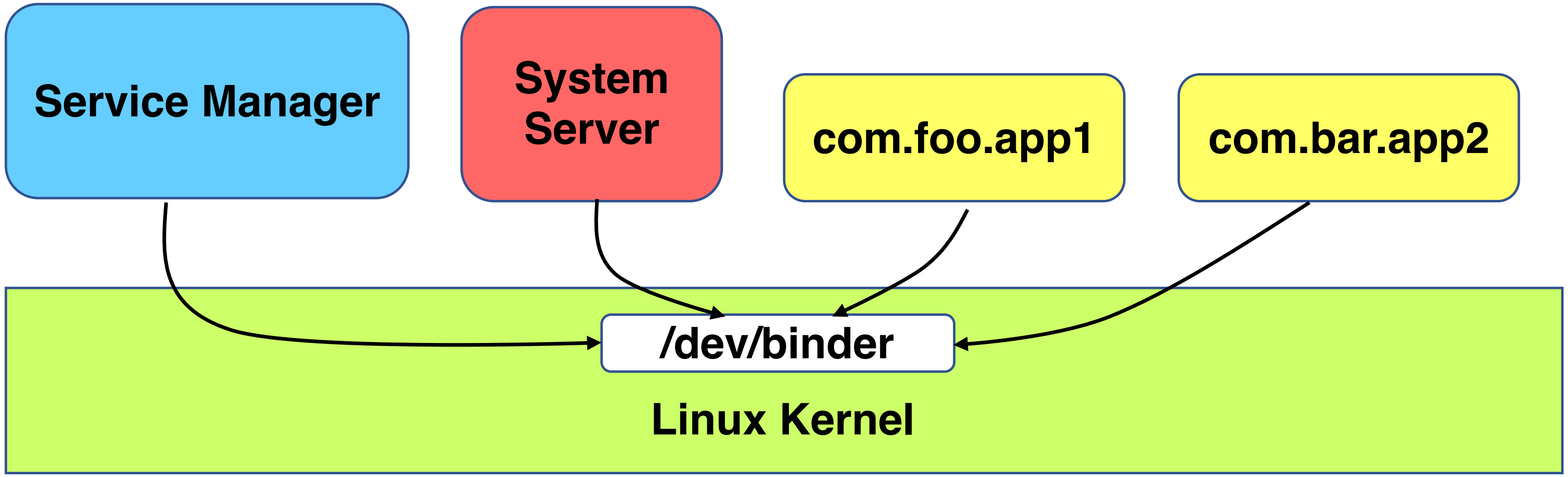
Native C/C++ Libraries

- **Many core Android services are built from native code**
  - Require native libraries written in C/C++
  - Performance benefit
  - Some of them are exposed through the Java API framework as native APIs
    - E.g., Java OpenGL API
- **Technique: JNI – Java Native Interface**
- **App developer can use Android NDK to include C/C++ code**
  - Common in gaming apps

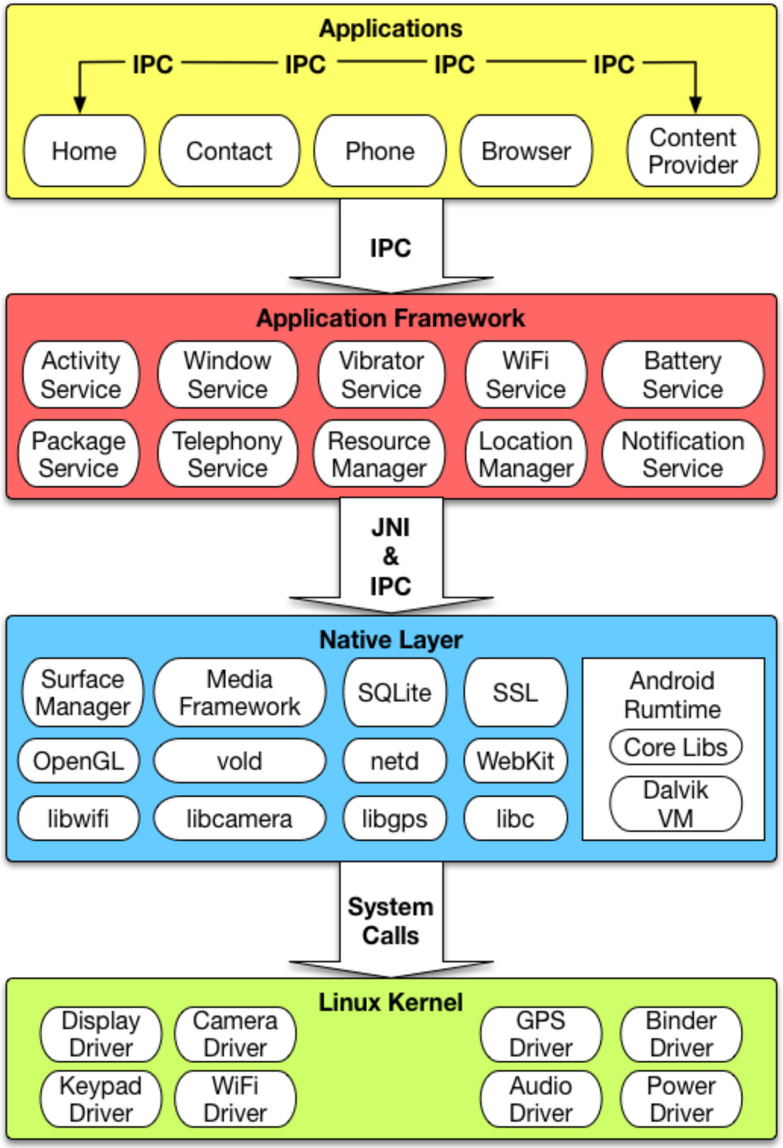
Android Binder IPC

- **An essential component in Android for Inter-Process Communication (IPC)**
  - Allows communication among apps, between system services, and between app and system service

- **Data sent through "parcels" in "transactions"**



IPC Is Pervasive in Android



How Is Binder Implemented: As RPC!

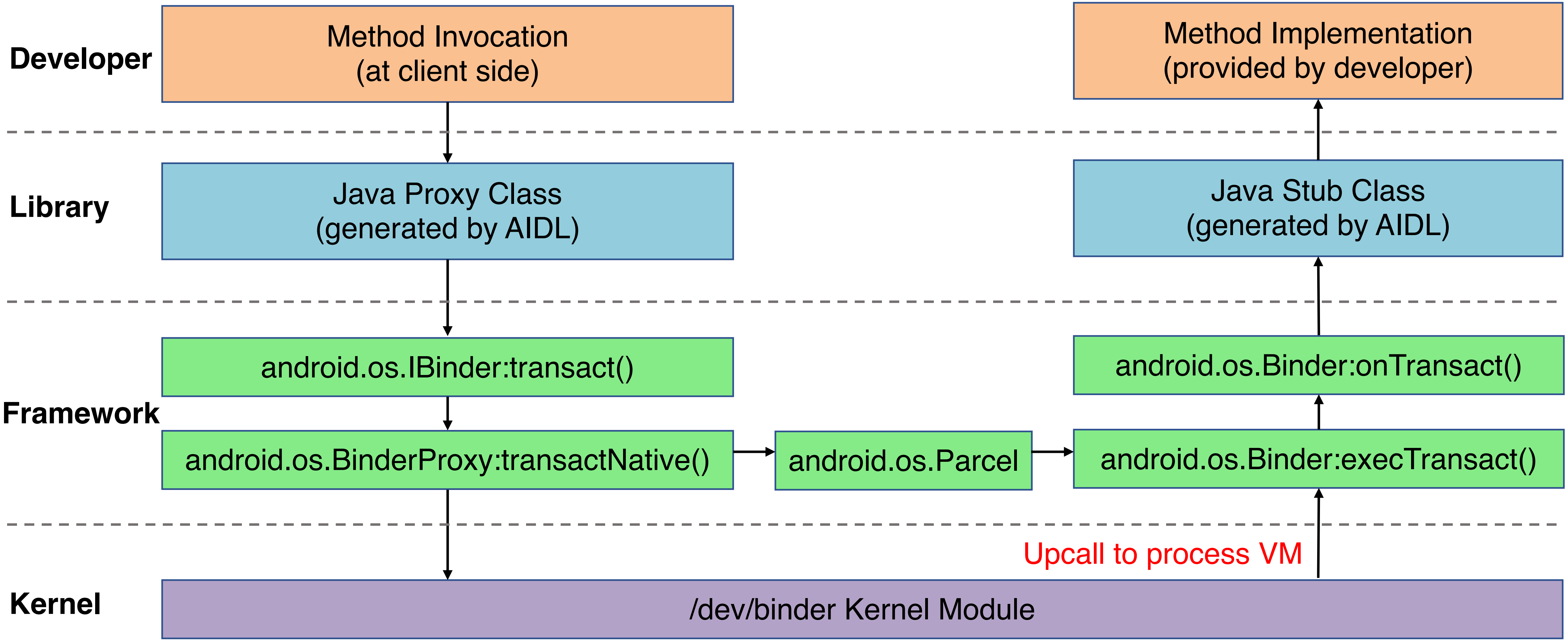
- **Developer defines methods and object interface in an .aidl file**

```
package com.example.android; // IRemoteService.aidl

/** Example service interface */
interface IRemoteService {
    /** Request the process ID of this service, to do evil things with it. */
    int getPid();
    /** Pause the service for a while */
    void pause(long time);
}
```

- **Android SDK generate a stub Java file for the .aidl file**
  - Developer implements the stub methods
  - Expose the stub in a Service
- **Client copies the .aidl file to its source, Android SDK generates a stub (a.k.a proxy) for it as well**
  - Client invoke the RPC through the stub

Binder Information Flow



Some Other Interesting Topics in Mobile OS

- **Energy management**
  - ECOSystem: Managing Energy as a First Class Operating System Resource
  - Drowsy Power Management
  - A Case for Lease-Based, Utilitarian Resource Management on Mobile Devices
- **Dealing with misbehaving apps**
  - DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior
  - eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones
- **Security**
  - CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management

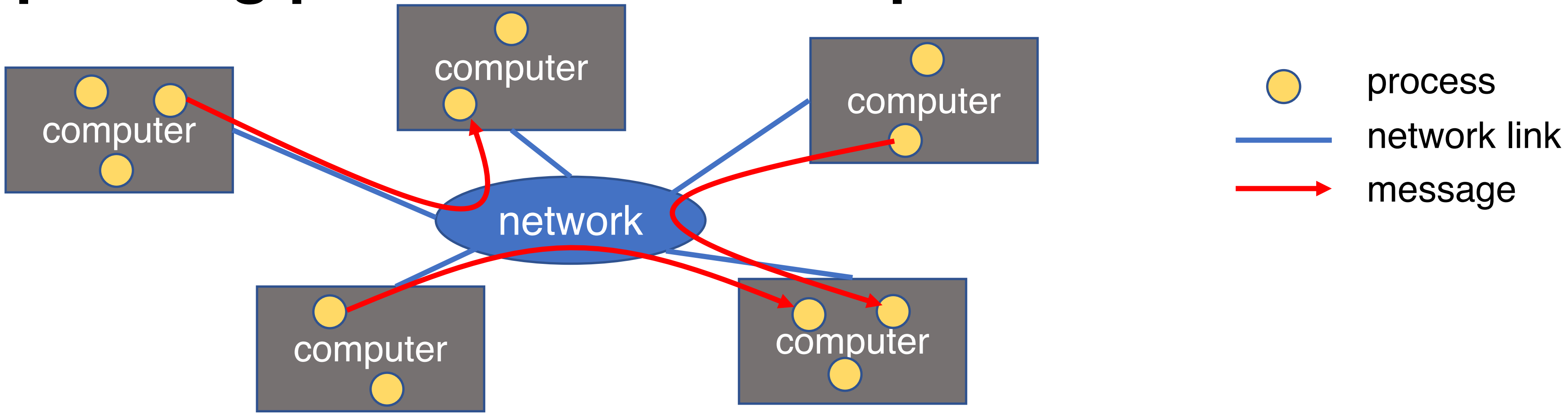
Summary

- **Smartphone has become an ubiquitous computing device**
  - Long history but past decade is disruptive
- **Mobile OS is an interesting and challenging subject**
  - Constrained resources
  - Different user interaction patterns
  - Frequently changing environment
  - Untrusted, immature third-party apps
- **Some unique design choices**
  - Application ≠ process
  - Multitasking
  - No swap space
  - Private storage

Distributed Systems

What is a Distributed System?

- **Cooperating processes in a computer network**



- **Leslie Lamport: "a distributed system is one where I can't do work because some machine I've never heard of isn't working!"**

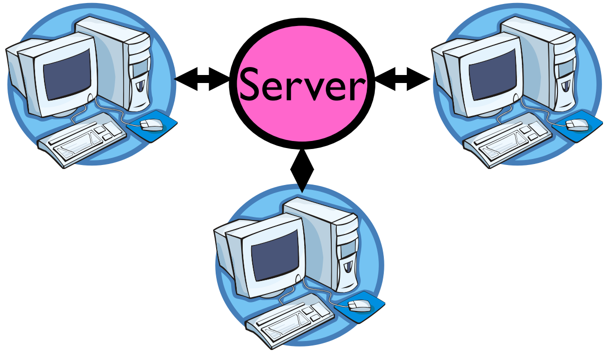
- **Popular distributed systems today**
  - Google file systems, BigTable, MapReduce, Hadoop, ZooKeeper, etc.

Forms & Models of Distributed Systems?

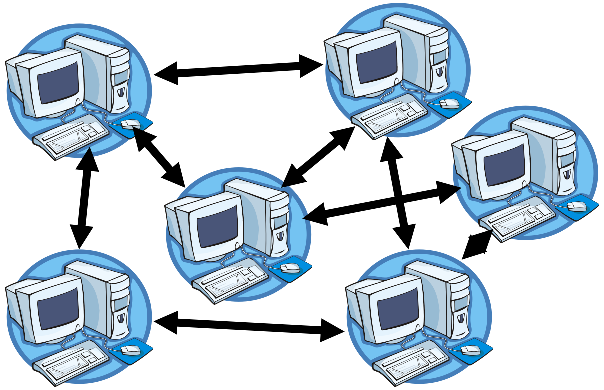
- **Degree of integration**
  - Loosely-coupled: Internet applications, email, web browsing
  - Mediumly-coupled: remote execution, remote file systems
  - Tightly-coupled: distributed file systems

- **Client/Server model vs. Cluster/Peer-to-Peer model**

major functions performed by a single physical computer



Client/Server Model



Cluster/Peer-to-Peer Model

physically separate computers working together on some task

Why Distributed Systems?

- **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

Distributed Systems: Promise

- **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
  - 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

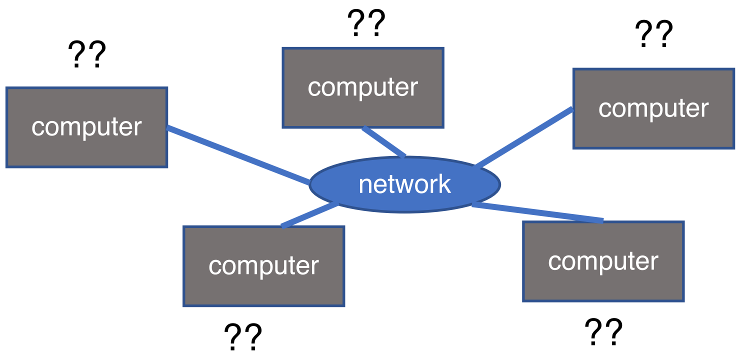
- **Transparency:**
  - the ability of the system to mask its complexity behind a simple interface
- **Possible transparencies:**
  - Location: Can’t tell where resources are located
  - Migration: Resources may move without the user knowing
  - Replication: Can’t tell how many copies of resource exist
  - 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

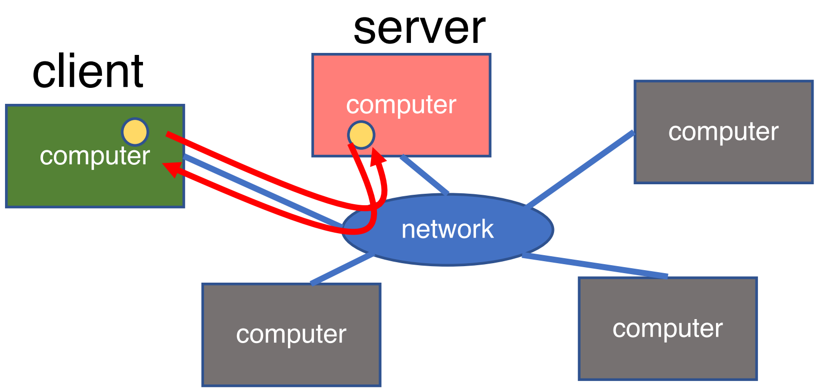
- **How to refer to a node in a distributed system?**
  - Essentially naming systems in network



- **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

- **How can one computer communicate with another?**
- **Raw Message: UDP**
- **Reliable Message: TCP**
  - Covered in networking class
- **Remote Procedure Call (RPC) /Remote Method Invocation(RMI)**



Raw Messaging

- **Initially network programming = raw messaging (socket I/O)**
  - 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"

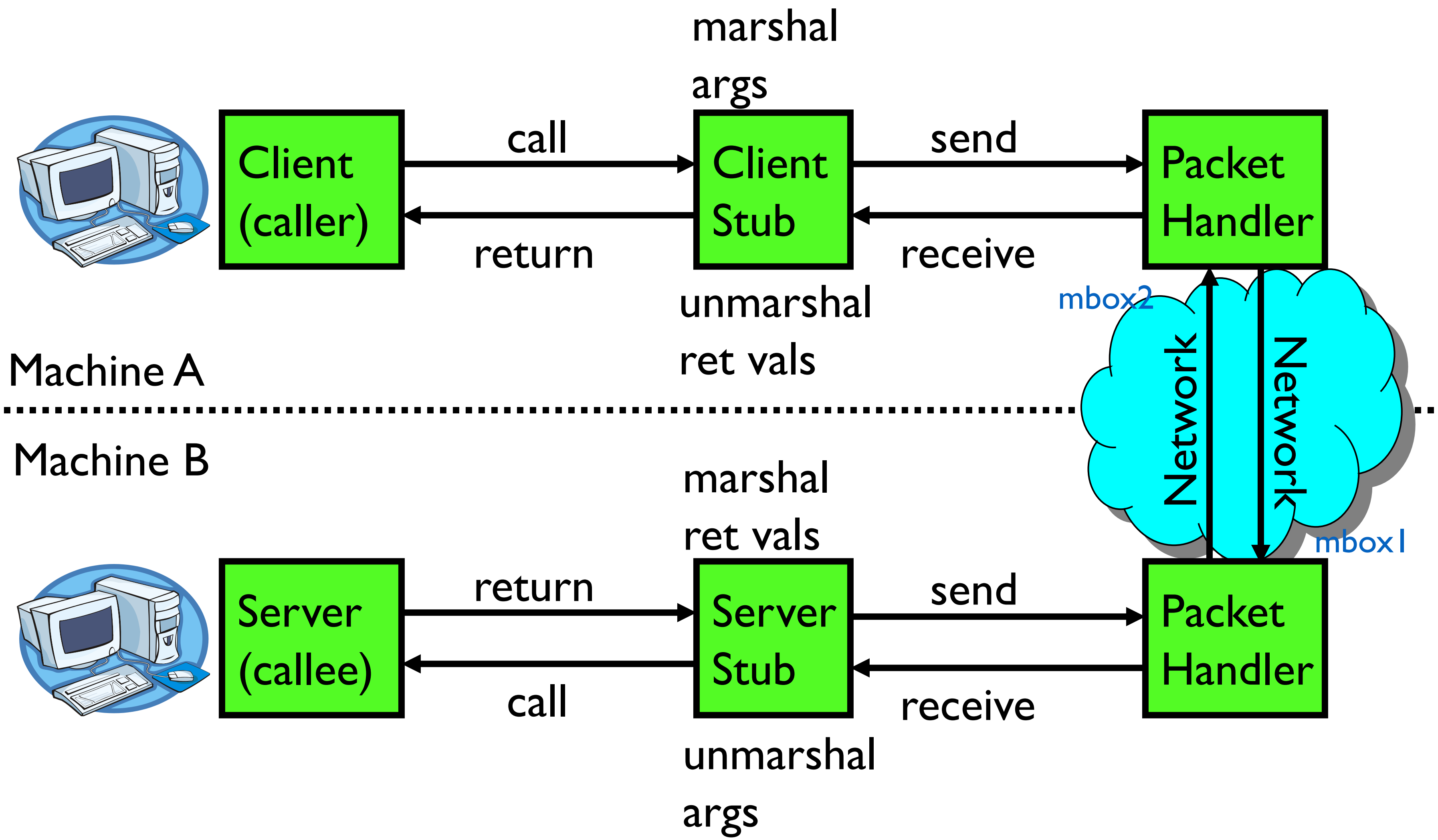
RPC Model

- **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

RPC library

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;
```

network

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);
}
```

RPC Runtime:

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

RPC Example: Return

RPC library

network

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 Example in Go Including Binding

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

binding

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)
```

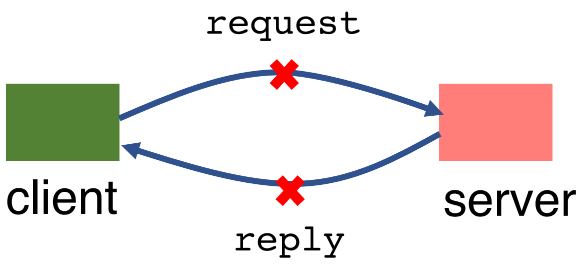
Server Program:

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

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

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

RPC Failure Semantic (2)

- **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 - 10);    put(key, v);`
- **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

RPC Failure Semantic (3)

- **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

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 Summary

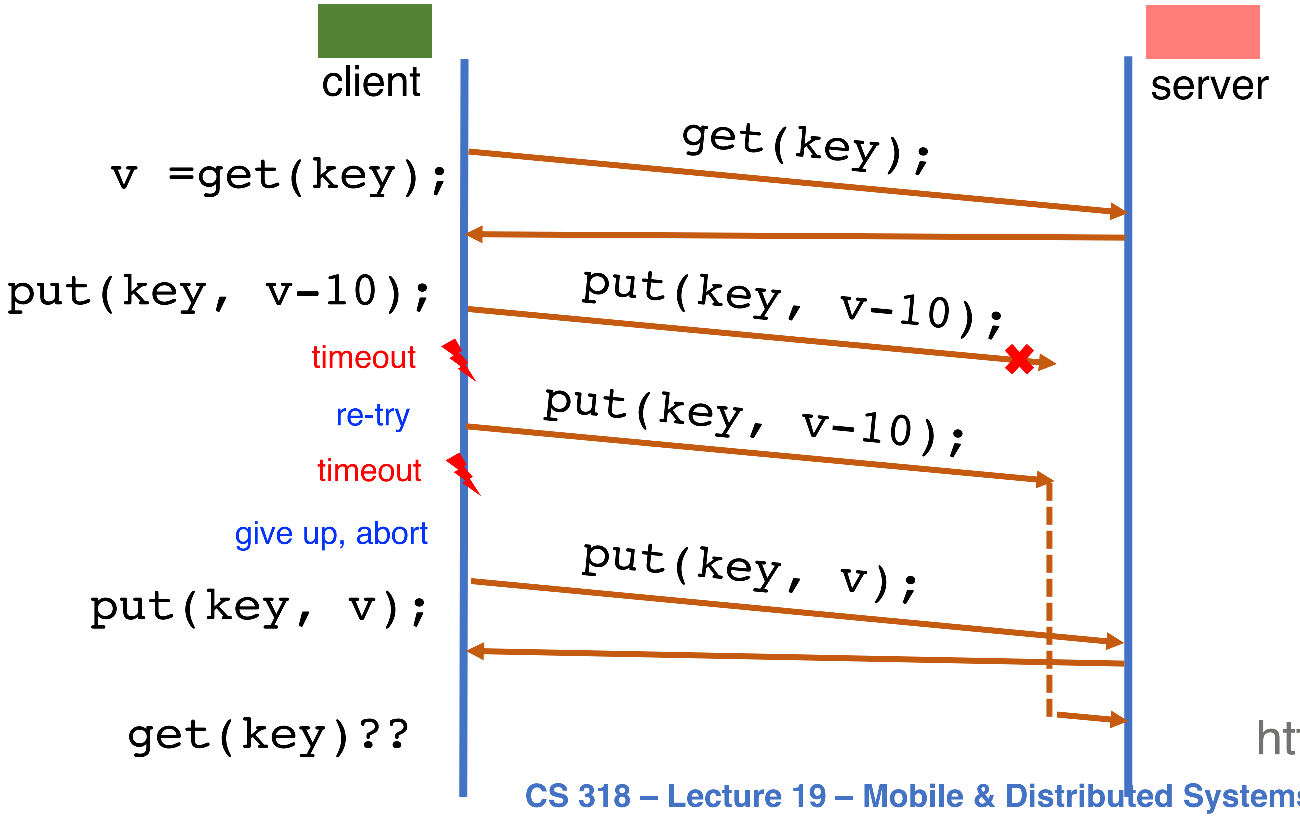
- **RPC is the most common model for communication in distributed applications**
  - "Cloaked" as DCOM, CORBA, Java RMI, etc.
  - Some popular libraries: gRPC, Golang RPC
  - Also used on same node between applications (e.g., gRPC)
- **RPC is essentially language support for distributed programming**
- **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…

- **System Reliability**

RPC Failure Semantic (2)

- **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 - 10);   put(key, v);`



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