CSEP552

Distributed Systems

Dan Ports

Agenda

- Course intro & administrivia
- Introduction to Distributed Systems
- (break)
- RPC
- MapReduce & Lab 1

Distributed Systems are Exciting!

- Some of the most powerful things we can build in CS
  - systems that span the world,
    serve millions of users,
    and are always up
- …but also some of the hardest material in CS
- Incredibly relevant today:
  everything is a distributed system!

This course

- Introduction to the major challenges in building distributed systems
- Key ideas, abstractions, and techniques for addressing these challenges

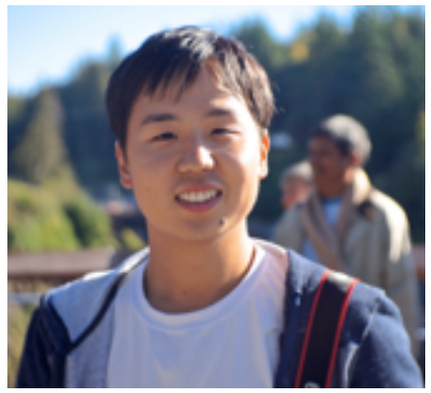
- Prereq: undergrad OS or networking course, or equivalent — talk to me if you're not sure

This course

- Readings and discussions of research papers
  - no textbook — good ones don't exist!
  - online discussion board posts
- A major programming project
  - building a scalable, consistent key-value store

Course staff

Instructor: **Dan Ports**
drkp@cs.washington.edu
office hours: Monday 5-6pm
or by appointment (just email!)



TA: **Haichen Shen**
haichen@cs.washington.edu

TA: **Adriana Szekeres**
aaasz@cs.washington.edu

Introduction to Distributed Systems

What is a distributed system?

- multiple interconnected computers that cooperate to provide some service

- examples?

Our model of computing used to be a single machine

Our model of computing today should be…

Our model of computing today should be…

Why should we build distributed systems?

- Higher capacity and performance
  - today's workloads don't fit on one machine
  - aggregate CPU cycles, memory, disks, network bandwidth
- Connect geographically separate systems
- Build reliable, always-on systems
  - even though the individual components are unreliable

What are the challenges in distributed system design?

What are the challenges in distributed system design?

- System design:
  - what goes in the client, server? what protocols?

- Reasoning about state in a distributed environment
  - locating data: what's stored where?
  - keeping multiple copies consistent
  - concurrent accesses to the same data

- Failure
  - *partial* failures: some nodes are faulty
  - network failure
  - don't always know what failures are happening

- Security

- Performance
  - latency of coordination
  - bandwidth as a scarce resource

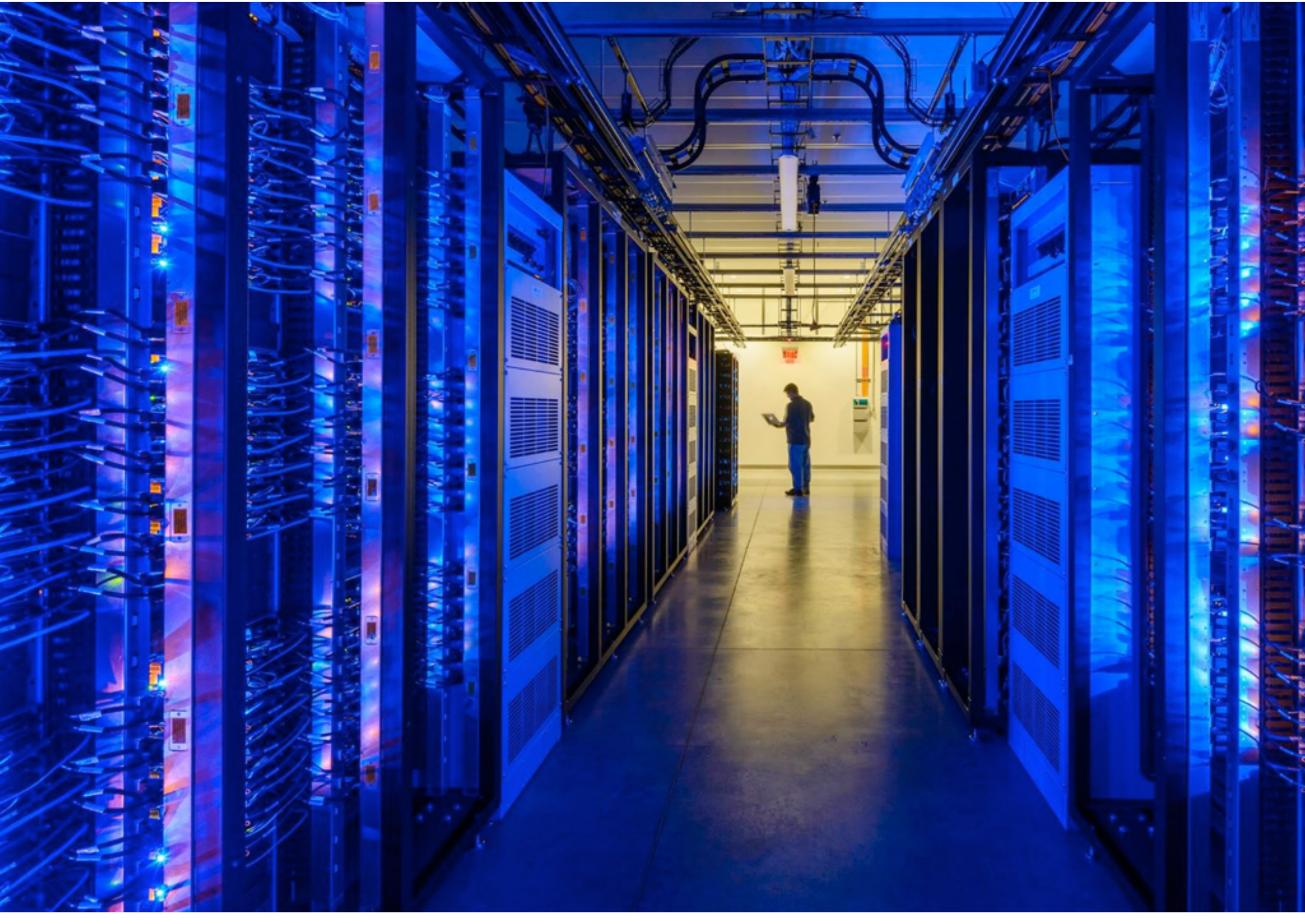
- Testing

We want to build distributed systems to be more scalable, and more reliable.

But it's easy to make a distributed system that's *less scalable and less reliable* than a centralized one!

Major challenge: failure

- Want to **keep the system doing useful work** in the presence of **partial failures**



A data center

- e.g., Facebook, Prineville OR
- 10x size of this building, $1B cost, 30 MW power
  - 200K+ servers
  - 500K+ disks
  - 10K network switches
  - 300K+ network cables
- What is the likelihood that all of them are functioning correctly at any given moment?

Typical first year for a cluster

[Jeff Dean, Google, 2008]

- ~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
- ~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
- ~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
- ~1 network rewiring (rolling ~5% of machines down over 2-day span)
- ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
- ~5 racks go wonky (40-80 machines see 50% packetloss)
- ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
- ~12 router reloads (takes out DNS and external network for a couple minutes)
- ~3 router failures (have to immediately pull traffic for an hour)
- ~dozens of minor 30-second blips for dns
- ~1000 individual machine failures
- ~10000 hard drive failures

Part of the system is *always* failed!

“A distributed system is one where the failure of a computer you didn’t know existed renders your own computer useless”

—Leslie Lamport, c. 1990

And yet…

- Distributed systems today still work most of the time
  - wherever you are
  - whenever you want
  - even though parts of the system have failed
  - even though thousands or millions of other people are using the system too

Another challenge: managing distributed state

- Keep data available despite failures:
  make multiple copies in different places

- Make popular data fast for everyone:
  make multiple copies in different places

- Store a huge amount of data:
  split it into multiple partitions on different machines

- How do we make sure that all these copies of data are consistent with each other?

Thinking about distributed state involves a lot of subtleties

Thinking about distributed state involves a lot of subtleties

- Simple idea: make two copies of data so you can tolerate one failure

Thinking about distributed state involves a lot of subtleties

- Simple idea: make two copies of data so you can tolerate one failure

- We will spend a non-trivial amount of time this quarter learning how to do this correctly!

  - What if one replica fails?

  - What if one replica just thinks the other has failed?

  - What if each replica thinks the other has failed?

  - …

A thought experiment

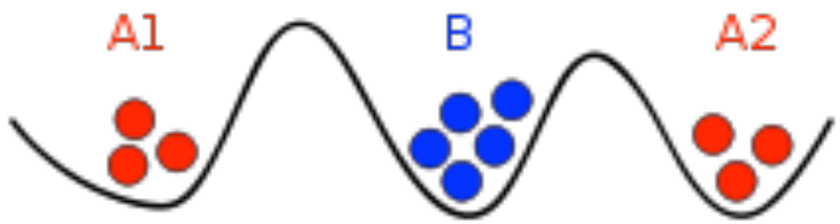
- Suppose there is a group of people, two of whom have green dots on their foreheads
- Without using a mirror or directly asking each other, can anyone tell whether they have a green dot themselves?
- What if I tell everyone: “someone has a green dot”?
  - note that everyone already knew this!

A thought experiment

- Difference between individual knowledge and common knowledge
- Everyone knows that someone has a green dot, but not that *everyone else knows* that someone has a green dot,
  or that everyone else knows that everyone else knows, ad infinitum…

The Two-Generals Problem

- Two armies are encamped on two hills surrounding a city in a valley



- The generals must agree on *the same time* to attack the city.

- Their only way to communicate is by sending a messenger through the valley, but that messenger could be captured (and the message lost)

The Two-Generals Problem

- No solution is possible!
- If a solution were possible:
  - it must have involved sending some messages
    - but the last message could have been lost, so we must not have really needed it
    - so we can remove that message entirely
- We can apply this logic to any protocol,
  and remove *all* the messages — contradiction

What does this have to do with distributed systems?

Distributed Systems are Hard!

- Distributed systems are hard because many things we want to do are *provably impossible*
  - consensus: get a group of nodes to agree on a value (say, which request to execute next)
  - be certain about which machines are alive and which ones are just slow
  - build a storage system that is always consistent and always available (the “CAP theorem”)
  - (we’ll make all of these precise later)

We will manage to do them anyway!

- We will solve these problems in practice by making the right assumptions about the environment
- But many times there aren’t any easy answers
- Often involves tradeoffs => class discussion

Topics we will cover

- Implementing distributed systems: system and protocol design
- Understanding the global state of a distributed system
- Building reliable systems from unreliable components
- Building scalable systems
- Managing concurrent accesses to data with transactions
- Abstractions for big data analytics
- Building secure systems from untrusted components
- Latest research in distributed systems

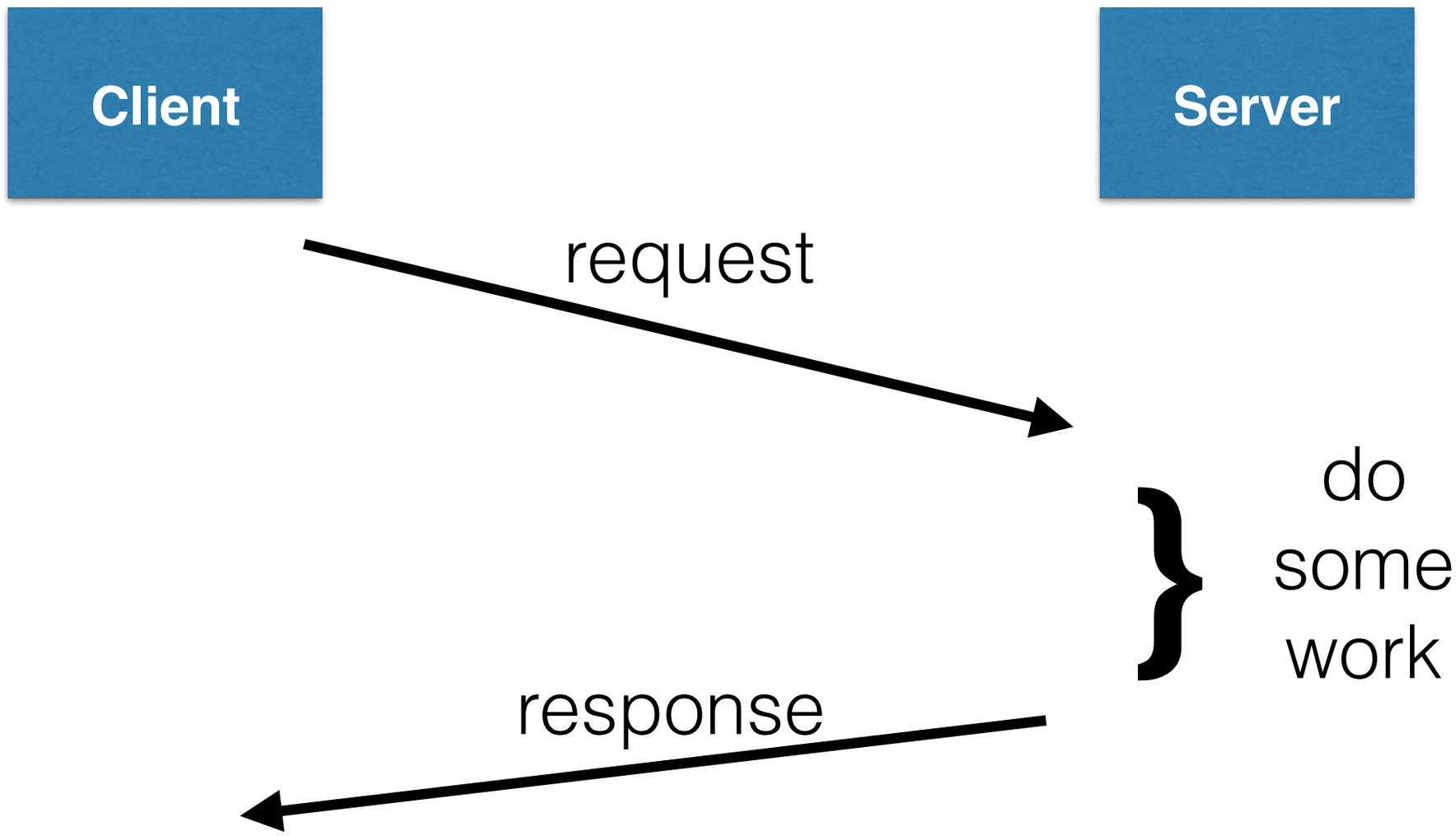
Agenda

- Course intro & administrivia
- Introduction to Distributed Systems
- (break)
- RPC
- MapReduce & Lab 1

RPC

- How should we communicate between nodes in a distributed system?
- Could communicate with explicit message patterns
- CS is about finding abstractions to make programming easier
- Can we find some abstractions for communication?

Common pattern: client/server



Obvious in retrospect

- But this idea has only been around since the 80s

- This paper: Xerox PARC, 1984
  Xerox Dorados, 3 mbit/sec Ethernet prototype

- What did “distributed systems” mean back then?

A single-host system

```
float balance(int accountID) {
   return balance[accountID];
}

void deposit(int accountID, float amount) {
   balance[accountID] += amount
   return OK;
}

client() {
   deposit(42, $50.00);
   print balance(42);
}
```

standard function calls

Defining a protocol

```
request "balance" = 1 {
   arguments {
     int accountID (4 bytes)
   }
   response {
     float balance (8 bytes);
   }
}

request "deposit" = 2 {
   arguments {
     int accountID (4 bytes)
     float amount (8 bytes)
   }
   response {
   }
}
```

Hand-coding a client & server

```
client() {
   s = socket(UDP)

   msg = {2, 42, 50.00}            // marshalling
   send(s, server_address, msg)
   response = receive(s)
   check response == "OK"

   msg = {1, 42}
   send(s -> server_address, msg)
   response = receive(s)
   print "balance is" + response
}

server() {
   s = socket(UDP)
   bind s to port 1024
   while (1) {
      msg, client_addr = receive(s)
      type = byte 0 of msg
      if (type == 1) {
          account = bytes 1-4 of msg    // unmarshalling
          result = balance(account)
          send(s -> client_addr, result)
      } else if (type == 2) {
          account = bytes 1-4 of msg
          amount = bytes 5-12 of msg
          deposit(account, amount)
          send(s -> client_addr, "OK")
      }
}
```

Hand-coding a client & server

```
client() {
   s = socket(UDP)

   msg = {2, 42, 50.00}              // marshalling
   send(s, server_address, msg)
   response = receive(s)
   check response == "OK"

   msg = {1, 42}
   send(s -> server_address, msg)
   response = receive(s)
   print "balance is" + response
}

server() {
   s = socket(UDP)
   bind s to port 1024
   while (1) {
      msg, client_addr = receive(s)
      type = byte 0 of msg
      if (type == 1) {
          account = bytes 1-4 of msg     // unmarshalling
          result = balance(account)
          send(s -> client_addr, result)
      } else if (type == 2) {
          account = bytes 1-4 of msg
          amount = bytes 5-12 of msg
          deposit(account, amount)
          send(s -> client_addr, "OK")
      }
}
```

Hard-coded message formats!

Hand-coding a client & server

```
client() {
   s = socket(UDP)

   msg = {2, 42, 50.00}                    // marshalling
   send(s, server_address, msg)
   response = receive(s)
   check response == "OK"

   msg = {1, 42}
   send(s -> server_address, msg)
   response = receive(s)
   print "balance is" + response
}

server() {
   s = socket(UDP)
   bind s to port 1024
   while (1) {
      msg, client_addr = receive(s)
      type = byte 0 of msg
      if (type == 1) {
          account = bytes 1-4 of msg     // unmarshalling
          result = balance(account)
          send(s -> client_addr, result)
      } else if (type == 2) {
          account = bytes 1-4 of msg
          amount = bytes 5-12 of msg
          deposit(account, amount)
          send(s -> client_addr, "OK")
      }
}
```

Hard-coded message formats!

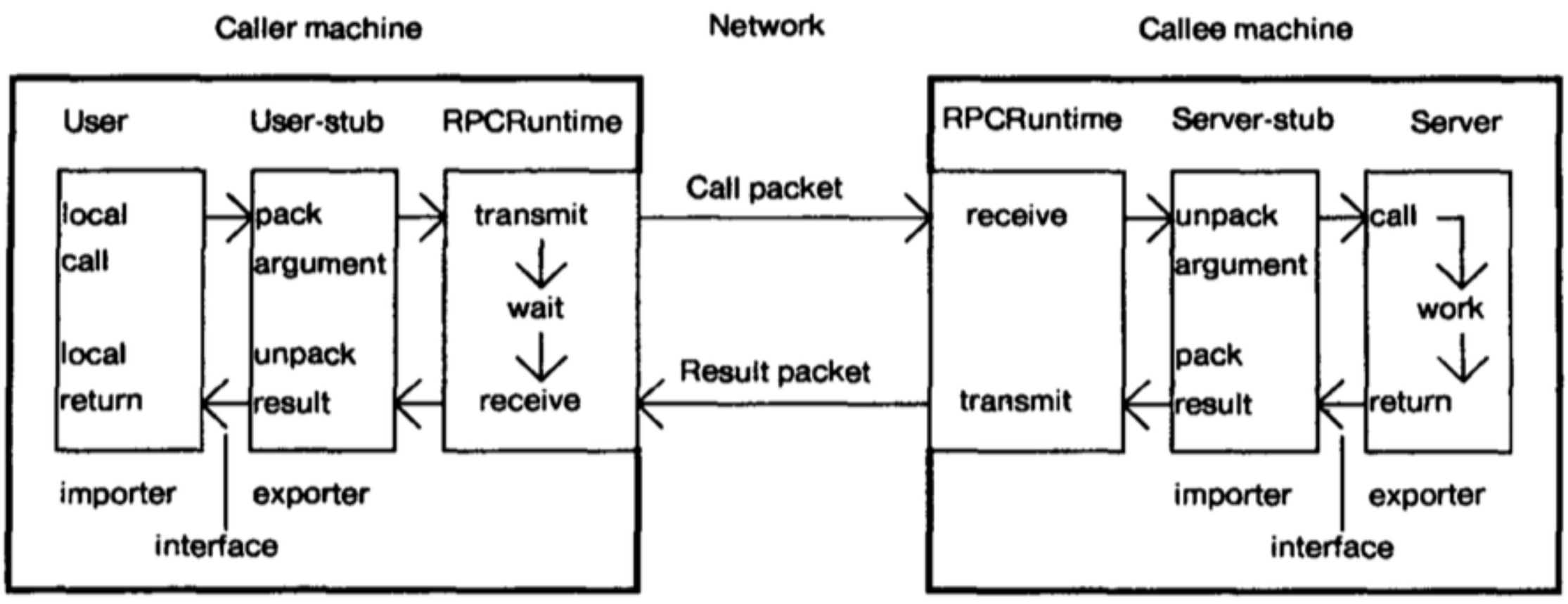
Lots of boilerplate code!

RPC Approach

- Compile protocol into stubs
  that do marshalling/unmarshalling

- Make a remote call look like a local function call

RPC Approach



Client & Server Stubs

Client stub:

```
deposit_stub(int account, float amount) {
    // marshall request type + arguments into buffer
    // send request to client
    // wait for reply
    // decode response
    // return result
}
```

To the client, looks like calling the deposit function we started with!

Server stub:

```
loop:
  wait for command
  decode and unpack request parameters
  call procedure
  build reply message containing results
  send reply
```

pretty much exactly the code we just wrote for the server!

Hides complexity of remote messaging

```
float balance(int accountID) {
   return balance[accountID];
}

void deposit(int accountID, float amount) {
   balance[accountID] += amount
   return OK;
}

client() {
   RPC_deposit(server, 42, $50.00);
   print RPC_balance(server, 42);
}
```

standard function calls

Is all the complexity really gone?

Is all the complexity really gone?

- Remote calls can fail!
- Performance: maybe much slower
- Resources might not be shared
  (file system, disk)

Dealing with failure

- Communication failures
- Host failures
- Can’t tell if failure happened before or after
  - was it the request message or the reply message that was lost?
  - did the server crash before processing the request or after?

At-least-once RPC

- Have client retry request until it gets a response
- What does this mean for client and server programmers?

At-least-once RPC

- Have client retry request until it gets a response
- What does this mean for client and server programmers?
    - Requests might be executed twice

At-least-once RPC

- Have client retry request until it gets a response
- What does this mean for client and server programmers?
  - Requests might be executed twice
  - OK if they’re idempotent

Alternative: at-most-once

- Include a unique ID in every request
  (how to generate this?)

- Server keeps a history of requests it's already answered, their ID, and the result

- If duplicate, server resends result

At-least-one vs at-most-once

- Discussion: which is most useful? When?

Can we combine them?

- “Exactly-once RPC”
- Use retries and keep a history on the server
- Not possible in general:
  what if the server crashes?
  how do we know whether it crashed right
  before executing or right after?
- Can make this work in most cases with a fault-tolerant server (Lab 3)

RPCs in Lab 1

- Our labs use Go's RPC library to communicate
- Go provides at-most-once semantics
  - sends requests over TCP; will fail if TCP connection lost
- Requests are executed in separate threads ("goroutines")
- Need to use synchronization mechanism (e.g, channels & sync.Mutex) to synchronize accesses between threads

RPCs Summary

- Common pattern for client-server interactions (but not all distributed systems work this way)

- RPC is used everywhere
  - automatic marshalling is really useful; lots of libraries
  - client stubs and transparency are useful, but transparency only goes so far

- Dealing with failures is still hard and requires application involvement

Agenda

- Course intro & administrivia
- Introduction to Distributed Systems
- (break)
- RPC
- MapReduce & Lab 1

MapReduce

- One of the first “big data” processing systems
- Hugely influential
  used widely at Google
  Apache Hadoop
  lots of intellectual children

Motivation

- Huge data sets from web crawls, server logs, user databases, web site contents, etc

- Need a distributed system to handle these (petabyte-scale!)

- Challenges
  - what nodes should be involved?
  - what nodes process what data?
  - what if a node fails?
  - …

MapReduce Model

- input is stored as a set of key-value pairs (k,v)

- programmer writes map function
  map(k,v) -> list of (k2, v2) pairs
  gets run on every input element

- hidden shuffle phase:
  group all (k2, v2) pairs with the same key

- programmer writes reduce function
  reduce(k2, set of values) -> output pairs (k3,v3)

Discussion

- Is this a useful model?
- What can we express in it?
- What can we *not* express in it?

Counting words

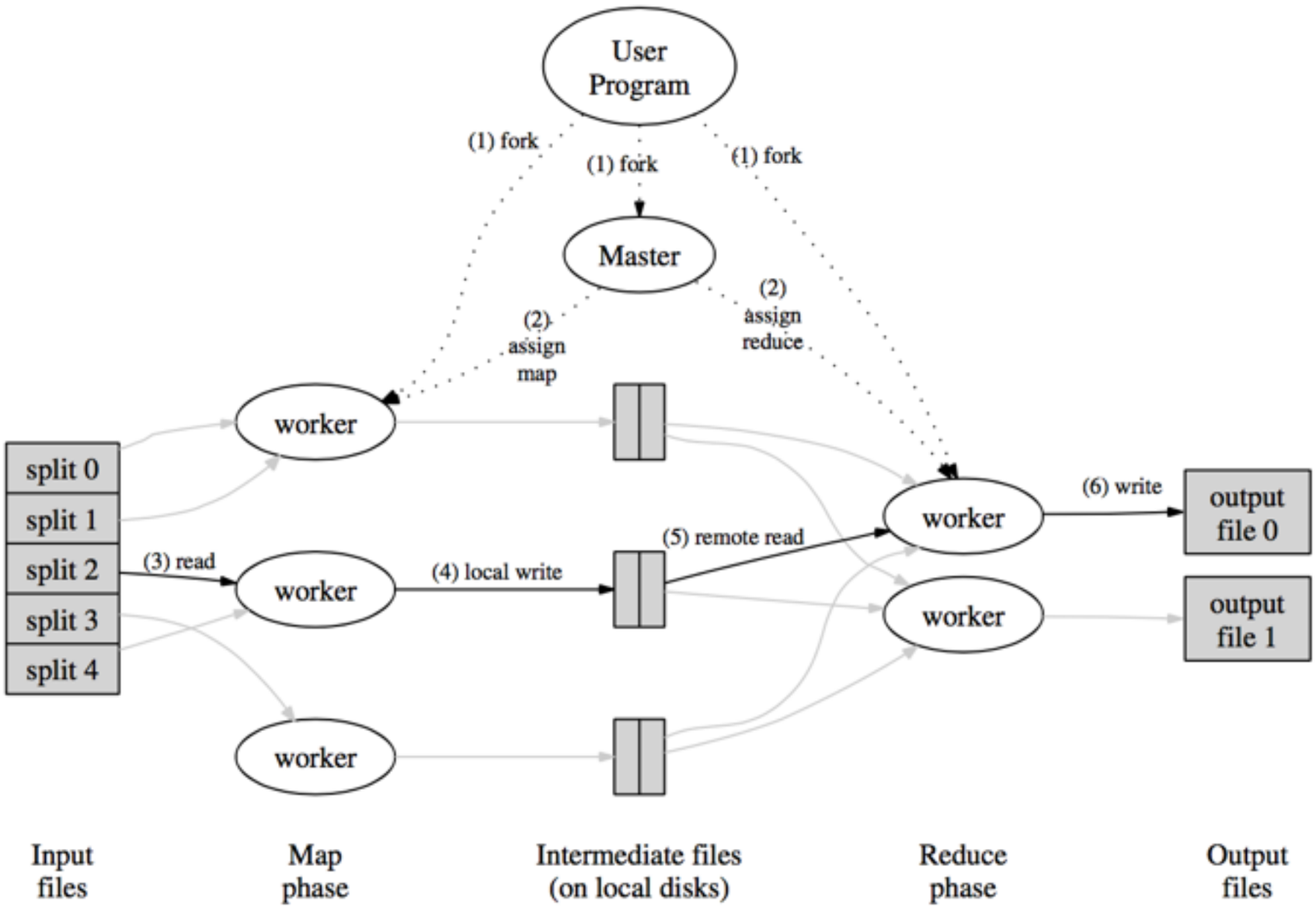
- Input: (text, _)
- Desired output: (word, # of occurrences)
- Map function:
  split text into words
  emit(word, 1)
- Reduce function:
  receives 1 element per word
  emit(word, number of elements received)

Computing an inverted index

- Input: (document name, list of words)
- Desired output: (word, list of documents)

- Map: split document into words, for each word: emit(word, docname)
- Reduce: input is (word, {list of docnames}) identity function; that's the output we wanted!

How does this get implemented?



Word count example

Word count example

- Input files: f1 = “a b”, f2 = “b c”

Word count example

- Input files: f1 = “a b”, f2 = “b c”

- master sends f1 to map worker 1 & f2 to worker 2
  map(f1) -> (a,1) (b,1)
  map(f2) -> (b,1) (c,1)

Word count example

- Input files: f1 = “a b”, f2 = “b c”

- master sends f1 to map worker 1 & f2 to worker 2
  map(f1) -> (a,1) (b,1)
  map(f2) -> (b,1) (c,1)

- once map workers finish, master tells each reduce worker what keys they are responsible for

Word count example

- Input files: f1 = “a b”, f2 = “b c”

- master sends f1 to map worker 1 & f2 to worker 2
  map(f1) -> (a,1) (b,1)
  map(f2) -> (b,1) (c,1)

- once map workers finish, master tells each reduce worker what keys they are responsible for

- each reduce worker accesses appropriate map output from each map worker (the shared filesystem is useful here!)

Word count example

- Input files: f1 = “a b”, f2 = “b c”

- master sends f1 to map worker 1 & f2 to worker 2
  map(f1) -> (a,1) (b,1)
  map(f2) -> (b,1) (c,1)

- once map workers finish, master tells each reduce worker what keys they are responsible for

- each reduce worker accesses appropriate map output from each map worker (the shared filesystem is useful here!)

- each worker calls reduce once per key, e.g.
  reduce(b, {1, 1}) -> 2

MapReduce Performance

- Why should performance be good?
  - Parallelism: many separate map and reduce workers n workers => 1/n * runtime?
- What are the performance limitations?
  - moving map output to reduce workers is expensive
  - stragglers: can’t finish a phase until the last one finishes

Failures

- What happens if a worker fails?
- What happens if a worker is just slow?
- What happens if the master fails?
- What happens if an input causes workers to crash?

Lab 1

- Intro to Go programming and a bit of fault tolerance problems in distributed systems
- Part A: just Map() and Reduce() for word count
- Part B: write the master code to hand out map and reduce jobs to worker threads via RPC
- Part C: deal with worker failures (failed RPCs)

Discussion

- What simplifying assumptions does MapReduce make and how do they help?
- Would a more general model be useful, and how?
- We'll revisit MapReduce later in the quarter and compare it to other data analytic systems