RMI & RPC

CS 475, Spring 2018
Concurrent & Distributed Systems

Review: Promises

- What if we want to run some task, and do stuff while we are waiting for it to be done?
- You COULD do it with a complicated combination of **synchronized**, **wait**, and **notify**
- You can use the **Promise** abstraction instead
  - Called a **CompletableFuture** in Java 8

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Result of the asynchronous computation";
});
// Block and get the result of the Future
String result = future.get();
System.out.println(result);
```

Review: Networks as Abstractions

- A network consists of communication links
- Networks have several “interesting” properties we will look at
  - Latency
  - Failure modes
- What is the abstraction?

Review: 3 Layer Abstraction

- The typical network abstraction model has 7 layers
  - Take CS 455 if you want to know more about these
- We'll think about 3 abstraction layers, and really focus on the top one

End-to-End layer	**Handles packet loss, etc. Translates from application-data to packets, implements a protocol**
Network layer	**Figures out where to send packets**
Link layer	**Physical links: care about how to deliver packets**

Review: Transport Protocols

- Anything in the end-to-end layer is likely built on top of some lower level protocol (**more** abstractions)
- TCP, or UDP
- Data integrity (checksumming) **UDP ✅ TCP ✅**
- Ordering control **UDP ❎ TCP ✅**
- Flow control **UDP ❎ TCP ✅**
  (not worrying about congestion)

Announcements

- Reminder: HW2 is out
  - http://www.jonbell.net/gmu-cs-475-spring-2018/homework-2/
- Today: RMI
- Reading: Tannenbaum 4.2
- Weds: Back to promises, with REST/Web Services

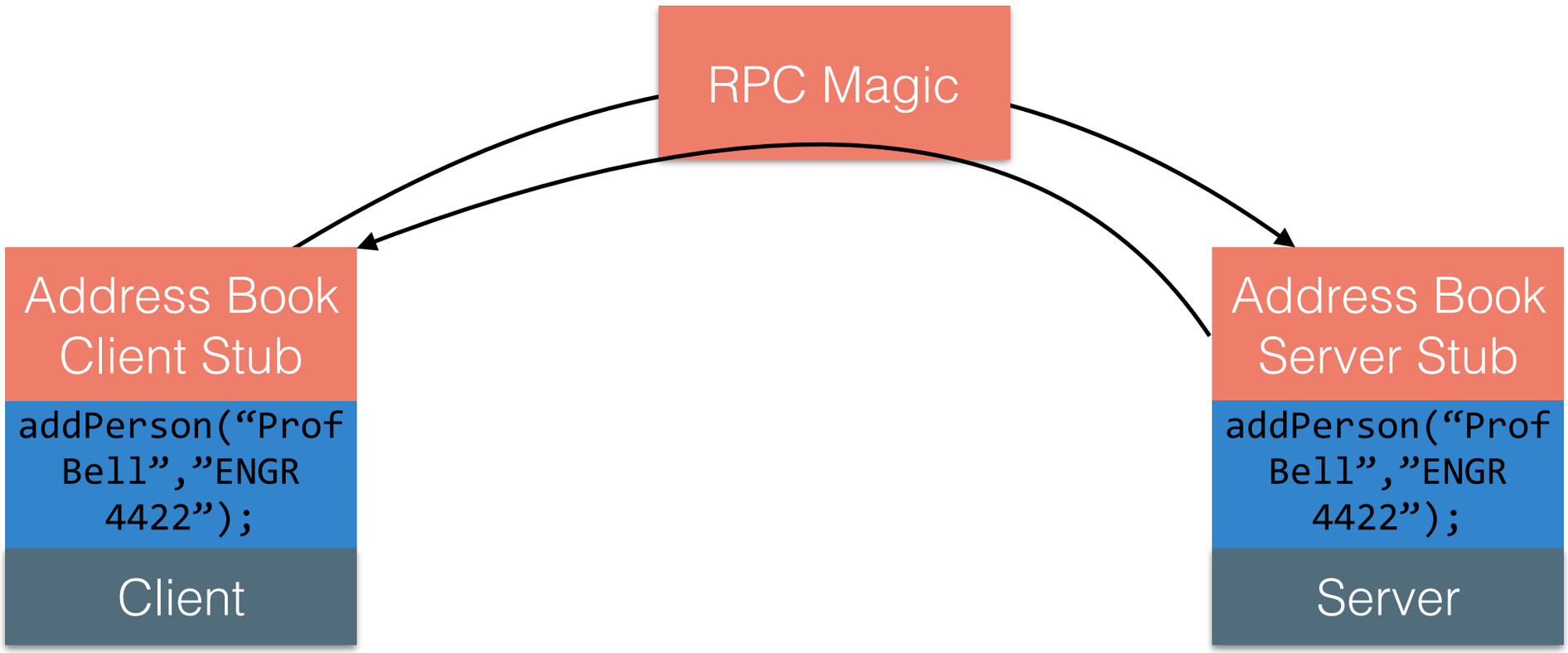
Abstractions

- Using sockets directly is annoying - very low level, likely someone else already figured out a way to solve your problem and their fix is bug free
- High level protocols:
  - FTP, SMTP
  - HTTP (REST, SOAP and other web service stuff)
- For general purpose, lower level:
  - RPC - Remote Procedure Call, and for Java: RMI - Remote Method Invocation
  - Language agnostic: Google ProtocolBuffers

Remote Procedure Calls

- Example: Address book
- We'll store our address book on a server
- But to simplify writing the code, can we pretend that the address book is stored on the client?
- RPC
  - Client program will think it's directly talking to the server
  - Server program will think it's directly talking to the client
  - In reality, there's a ton of glue in between them

Remote Procedure Calls



Remote Procedure Calls

- Magic is (surprise?) really complicated
- Challenges in building RPC will re-occur in almost any distributed system
- What is the alternative?

```
void sendRequest(String name, String office)
{
    Socket clientSocket = new Socket(ADDRESSBOOK_SERVER, ADDRESSBOOK_PORT);
    DataOutputStream outToServer = new
DataOutputStream(clientSocket.getOutputStream());
    BufferedReader inFromServer = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
    outToServer.writeBytes("addPerson" + '\n');
    outToServer.writeBytes(name + '\n');
    outToServer.writeBytes(office + '\n');
    clientSocket.close();
}
```

Why use an abstraction

- Ugly boilerplate
- Might have bugs? (Almost definitely)
- Not portable (string encoding)
- Hard to understand and maintain
- Vulnerability-prone

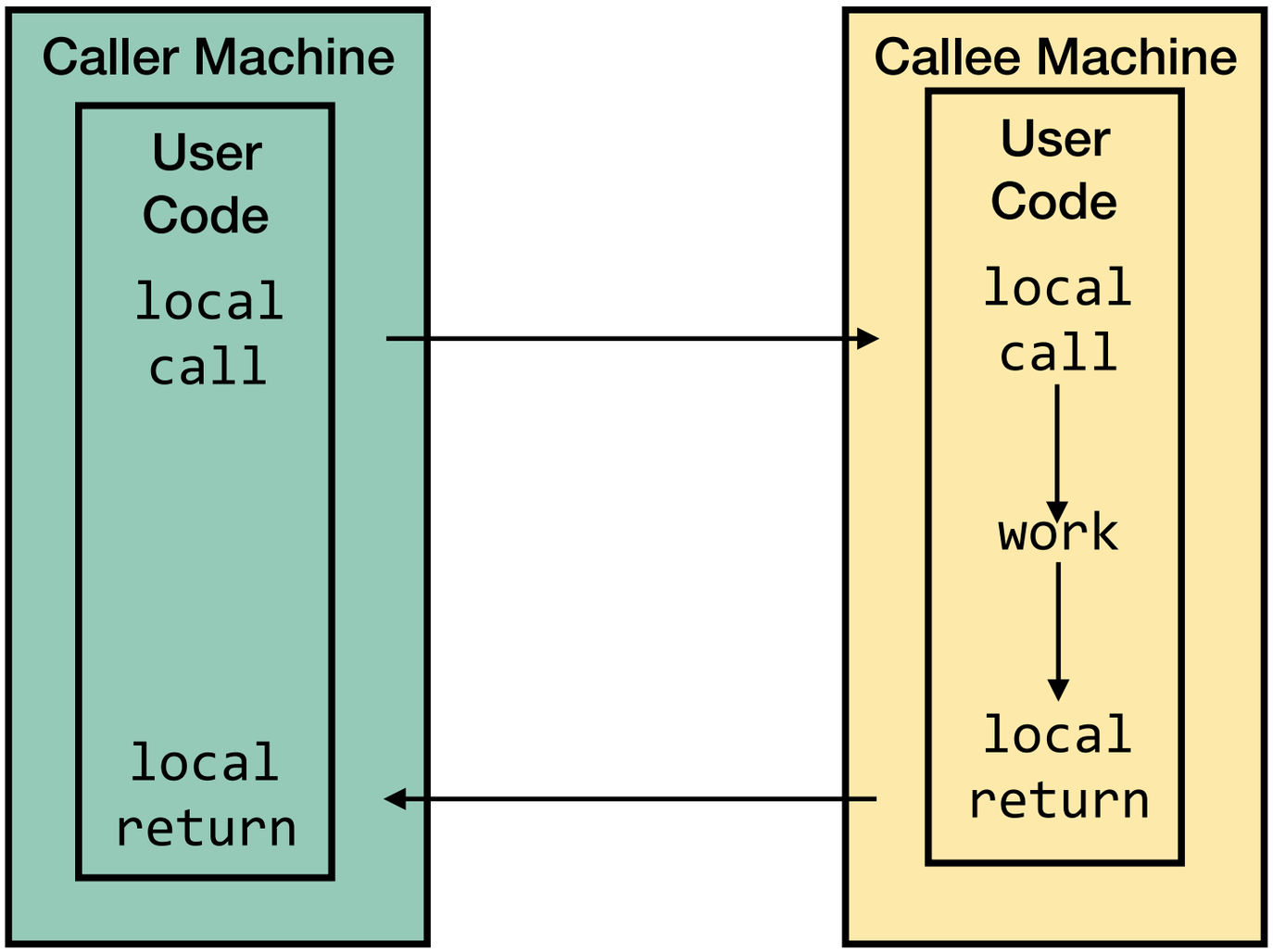
What abstraction to use

- RPC -> use an abstraction all programmers already use
  - E.g. Local Procedure Call (LPC)
    - E.g. just calling functions
- RPC makes transparent whether server is local or remote
- RPC allows applications to become distributed transparently
- RPC makes architecture of remote machine transparent

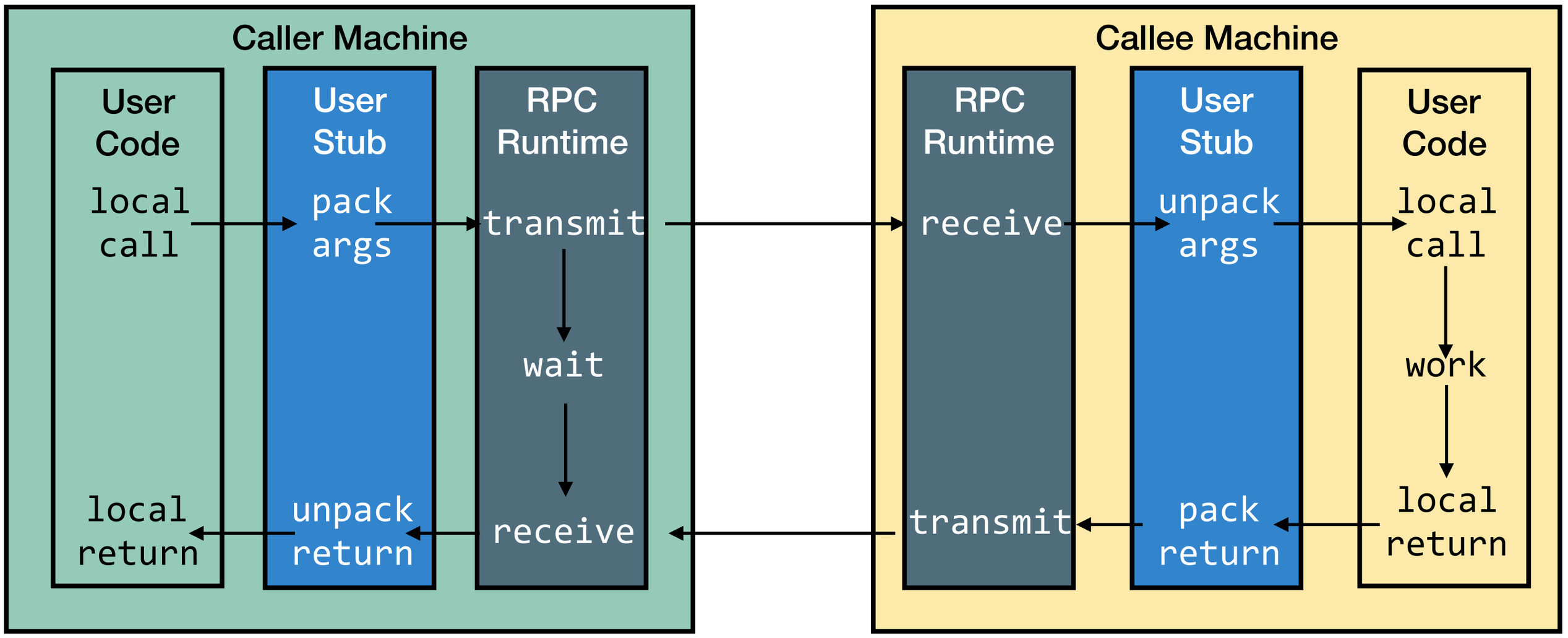
RPC Challenges

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

RPC: High Level Approach



RPC: High Level Approach



RPC Stubs

- Compiler automatically generates these
- 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

Marshalling and Unmarshalling

- How did we decide to send the data?
- How did we decide to encode the data?
- This is marshalling (done poorly)

```
void sendRequest(String name, String office)
{
    Socket clientSocket = new Socket(ADDRESSBOOK_SERVER, ADDRESSBOOK_PORT);
    DataOutputStream outToServer = new
DataOutputStream(clientSocket.getOutputStream());
    BufferedReader inFromServer = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
    outToServer.writeBytes("addPerson" + '\n');
    outToServer.writeBytes(name + '\n');
    outToServer.writeBytes(office + '\n');
    clientSocket.close();
}
```

RPC Stubs

- How does the compiler know what to generate?
- Usually there is a formal spec of the calling interface
  - "Interface Definition Language"
- Java: Just any old Interface
  - Other RPC systems: specified in a small language similarly (including XML)

RPC: The Good Stuff

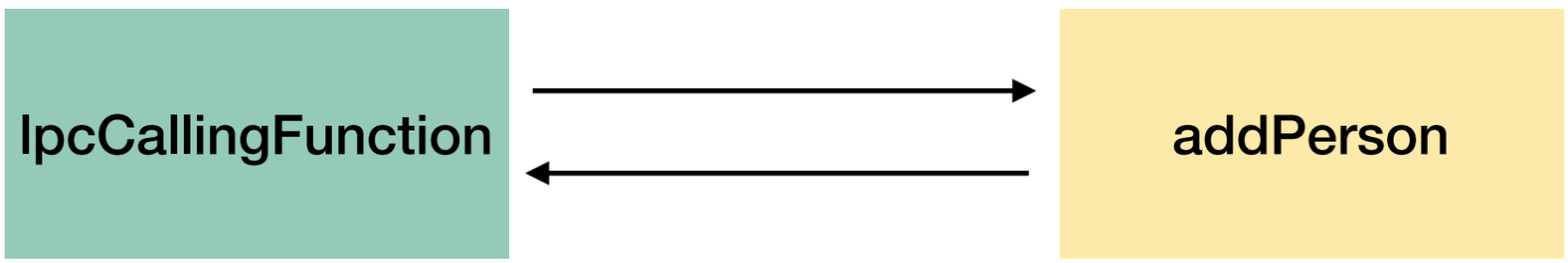
- Hides networking details
- Hides marshaling details
- Lets you evolve the protocol separately from the communication path
- Easy to add extra shared features
  - Authentication
  - Naming/location

RPC: The Bad Stuff

- Latency
- Pointer transfers
- Failures

Local Procedure Call - Failures

- Local code calls `addPerson`
- What can happen?
  - LPC returns (OK, can see return value/error code)
  - LPC doesn't return (OK, know it's still running)

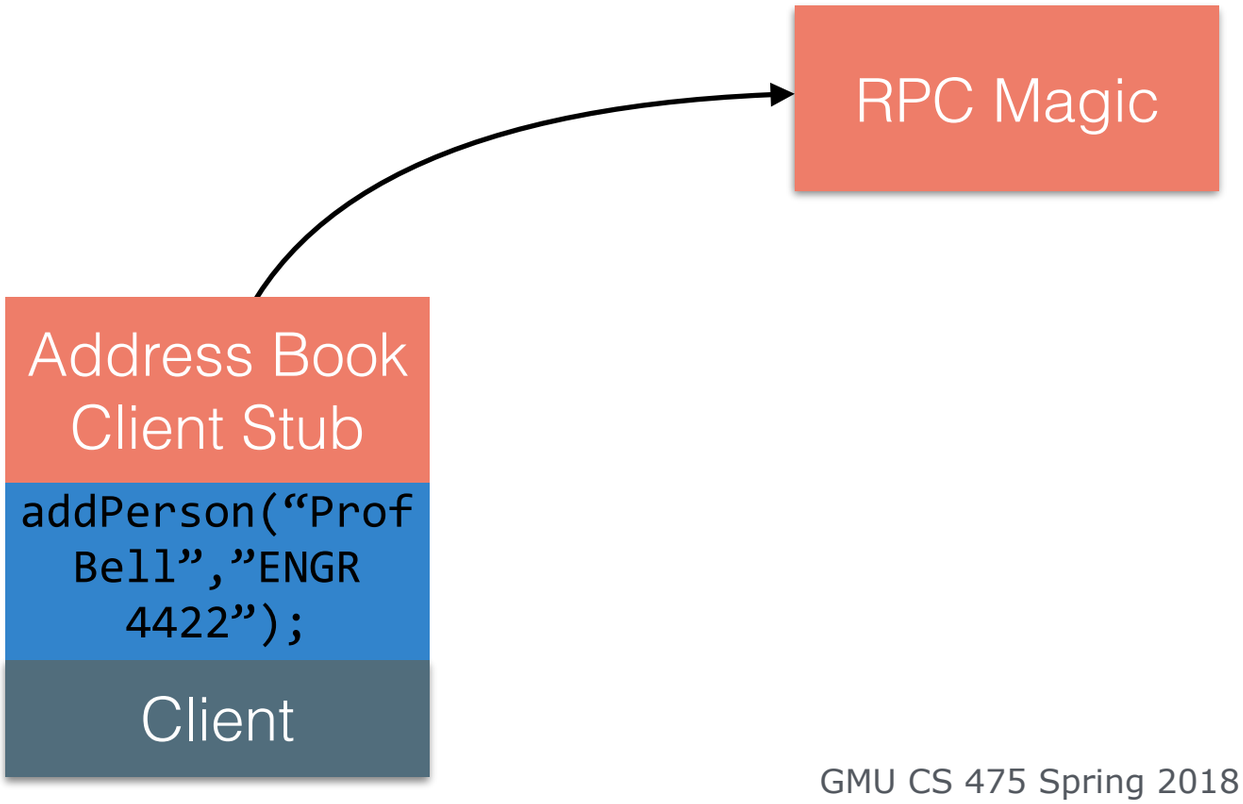


Shared Fate

- Two methods/threads/processes running on the same computer generally have **shared fate**
- They will either both crash, or neither will crash

RPC Failures

- If our client makes an RPC request, but hasn't heard back yet, how do we know what happened?



RPC Failures

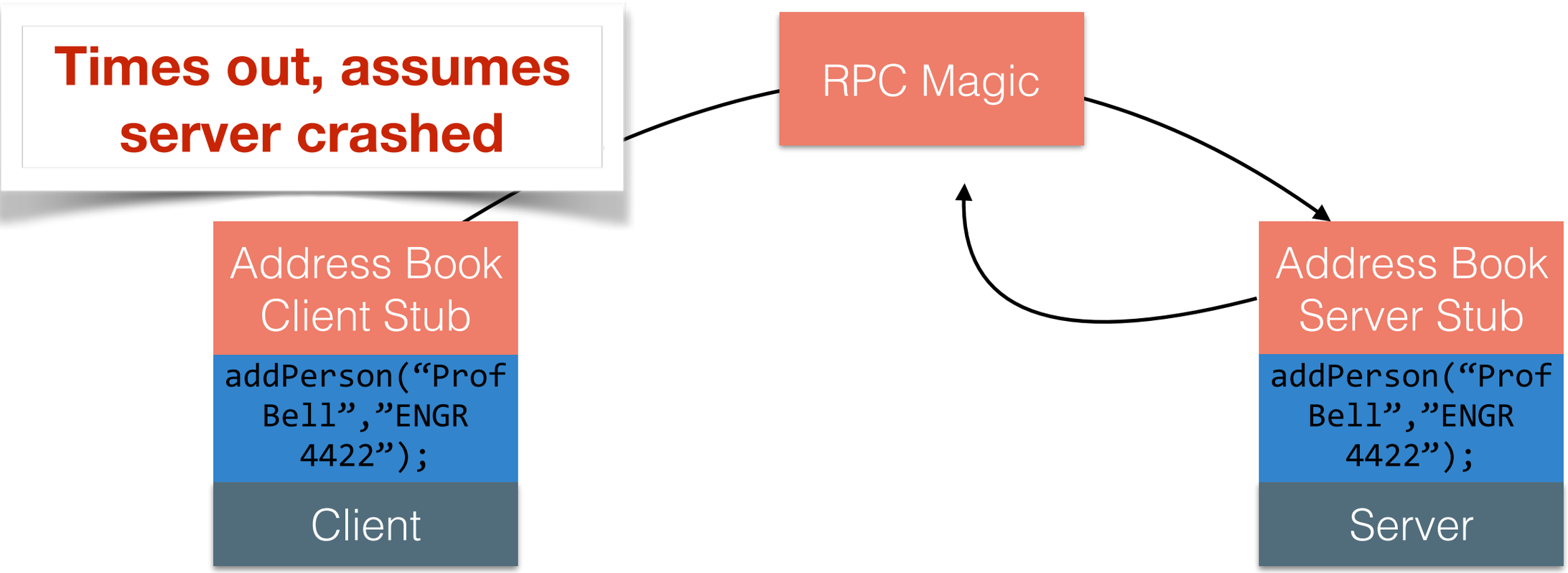
- If we haven't heard back from the server yet, possibilities are:
  - Server never received request
  - Server received request and crashed
  - Server received request, processed it, crashed
  - Server received requested, processed it, sent response but never received it

Split Brain

- When two machines in a distributed system can't talk to each other, they might start believing different things
- Two sides can not reconcile view of world because they can't talk to each other
- We call this a **split brain** problem

Split Brain in RPC

Times out, assumes server crashed



Split brain: Client thinks addPerson didn't succeed, server did complete it though!

RPC Semantics

- No matter what we do, if we want RPC, we have networks, networks might have timeouts/failures
- How do we handle the potential for split brain?
  - If we don't hear a response, just freeze?
- What can the abstraction guarantee?
  - Leak some of this complexity through

RPC Semantics: Exactly once delivery

- Can our RPC abstraction guarantee call is sent and processed exactly once?
- In general, not possible:
  - Server can crash at **any** point
  - Never will know exactly what happened

RPC Semantics: At least once

- RPC system might guarantee **at least once** delivery
- Client library keeps track of unconfirmed messages
- If message is not confirmed, keep re-sending
- Works fine for **idempotent** requests (requests that can be repeated with no side effect)

RPC Semantics: At most once

- RPC system might guarantee at most once delivery
- Client library re-sends if it doesn't hear an ACK
- Client library adds message IDs
- Server library keeps track of received message IDs
- Problems?
  - Server needs to perpetually track received IDs

RPC - Implementation Issues

- Performance is the usual issue:
  - When do you generate the stubs?
  - How do you patch in the stubs?
  - What data is copied from client to server?
- Environmental concerns:
  - What if client/server are very different (OS/ language)?

Java RMI

- Synchronous (client method doesn't return until server completes)
- At most once delivery
- Hence, in the event of a communication failure, an exception is thrown on your client
- Implications:
  - Client code needs to be aware that failures might happen (and exception might be thrown)
  - Client code needs to have some plan to handle when a message fails to get through (application specific)

Java RMI

- Threading model:
  - What happens when there are multiple simultaneous RMI requests to the same server?
- RMI creates a *thread pool*, a set of threads ready to handle each request
  - Subsequent calls from the same client might or might not use the same thread
  - Subsequent calls from other clients might use the same thread as others
- Implications:
  - Can process multiple requests simultaneously
  - Need to be cognizant of thread safety

Java RMI

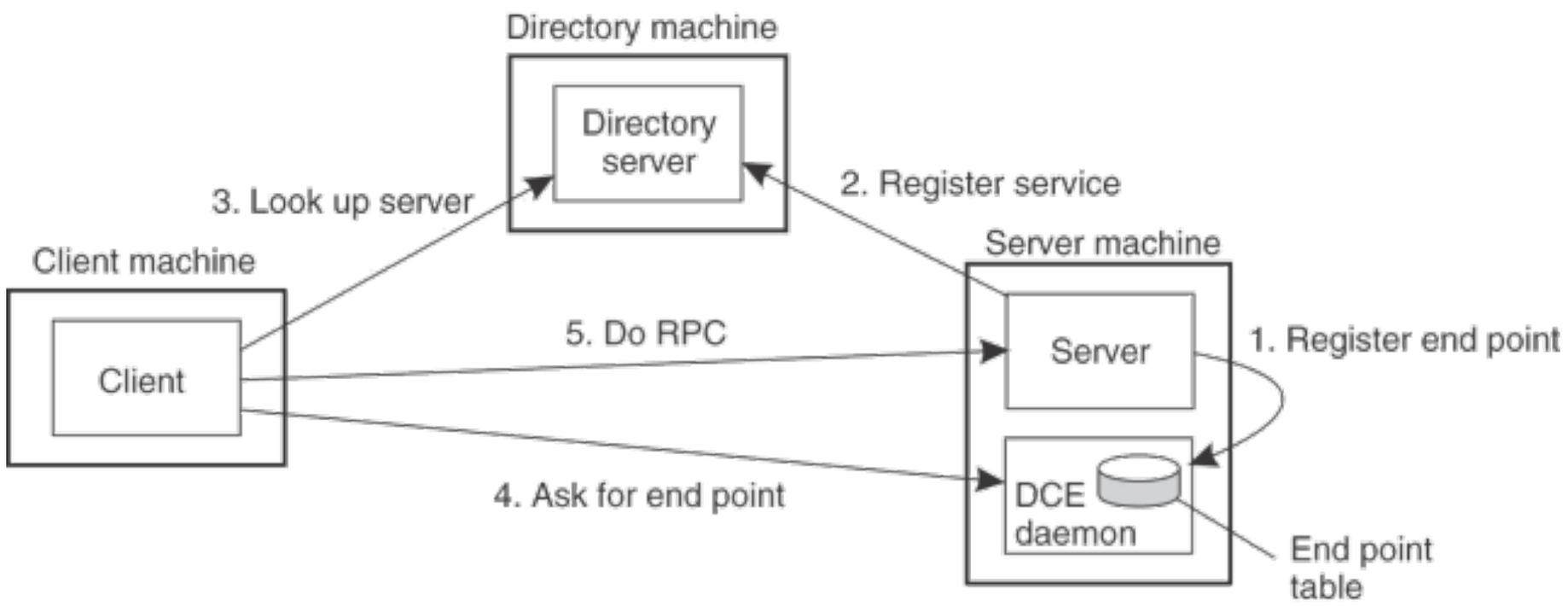
```
public interface AddressBook extends Remote {
   public LinkedList<Person> getAddressBook() throws RemoteException;

   public void addPerson(Person p) throws RemoteException;
}
```

```
AddressBook book = new AddressBookServer();
AddressBook stub = (AddressBook) UnicastRemoteObject.exportObject(book, 0);
Registry registry = LocateRegistry.createRegistry(port);
registry.rebind("AddressBook", stub);
```

```
Registry registry = LocateRegistry.getRegistry("localhost", 9000);
AddressBook addressBook = (AddressBook) registry.lookup("AddressBook");
```

Java RMI

- Registration of a server makes it possible for a client to locate the server and bind to it
- Server location is done in two steps:
  - Locate the server's machine.
  - Locate the server on that machine.



RPC Summary

- Expose RPC properties to client, since you cannot hide them
  - Application writers have to decide how to deal with partial failures
  - Consider: E-commerce application vs. game

Summary

- Procedure calls
  - Simple way to pass control and data
  - Elegant transparent way to distribute application
  - Not only way…
- Hard to provide true transparency
  - Failures
  - Performance
  - Memory access
  - Etc.
- How to deal with hard problem: give up and let programmer deal with it

Socrative

- Reminder - class name is CS475