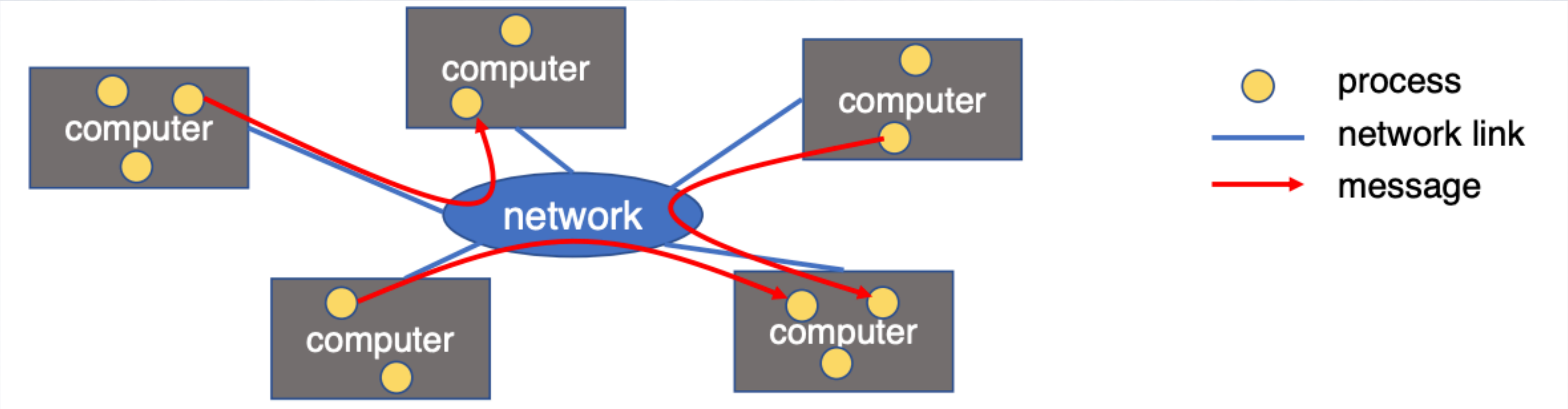
Distributed Systems

Thierry Sans

What is a distributed system?

➡ Cooperating processes in a computer network



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

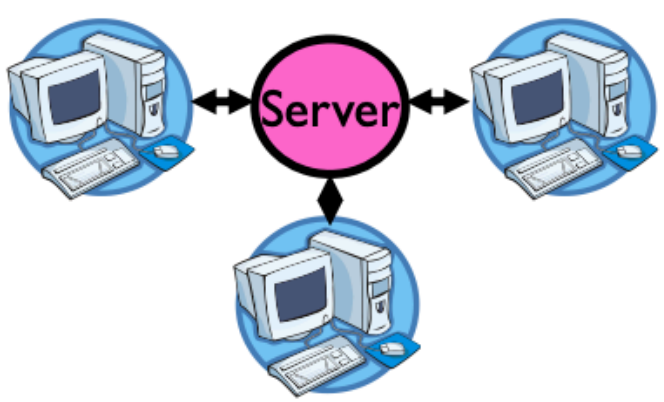
Popular distributed systems today:
Google file systems, BigTable, MapReduce, Hadoop, ZooKeeper, etc.

Forms & models of distributed systems?

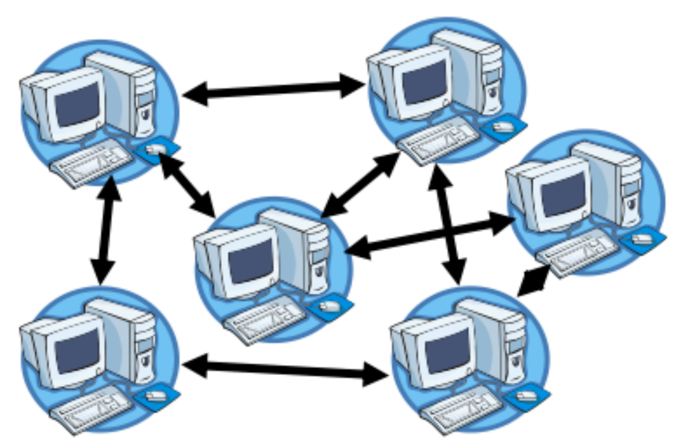
Degree of integration

- **Loosely-coupled**
  internet applications (e.g email, web, FTP, SSH)
- **Mediumly-coupled**
  remote execution (e.g RPC), remote file system (e.g NFS)
- **Tightly-coupled**
  distributed file systems (e.g. AFS)

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

The promise of distributed systems

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

The reality of distributed systems

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)

Requirements

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

Possible transparencies

- Location - cannot tell where resources are located
- Migration - resources may move without the user knowing
- Replication - cannot tell how many copies of resource exist
- Concurrency - cannot 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

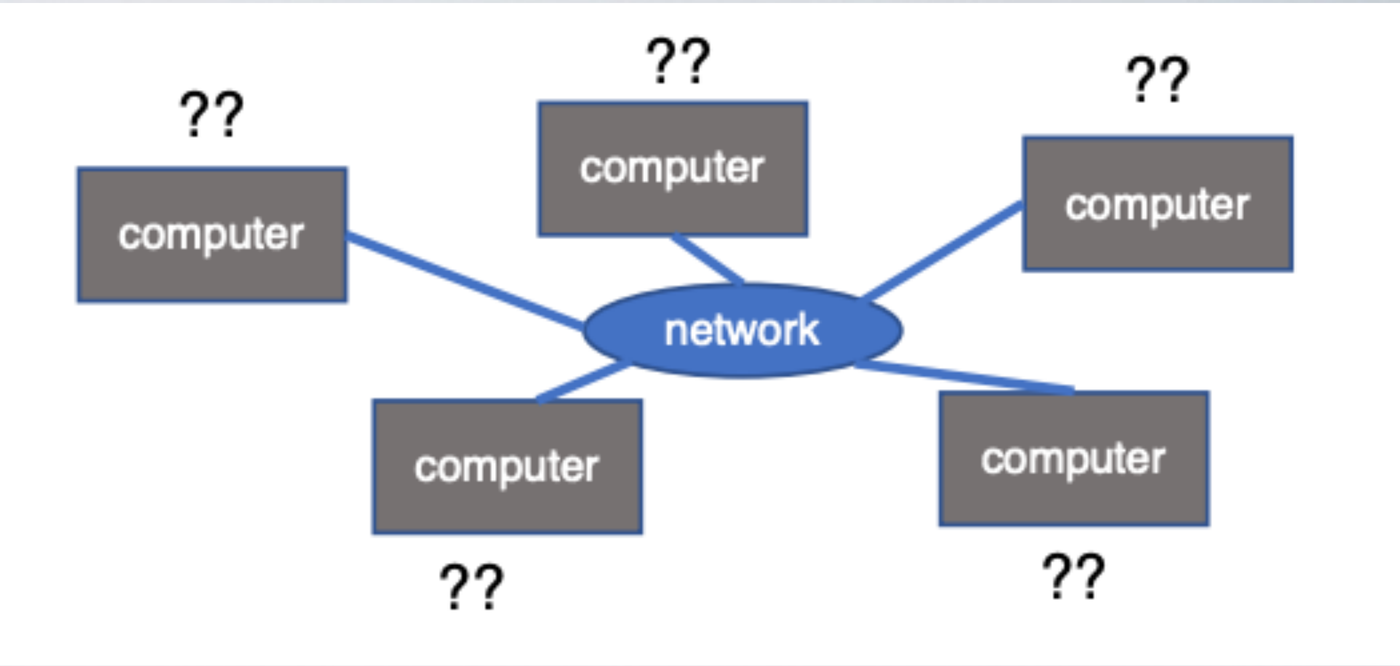
➡ A **server** is a program (or collection of programs) that provide a service (file server, name service, etc.)

- 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

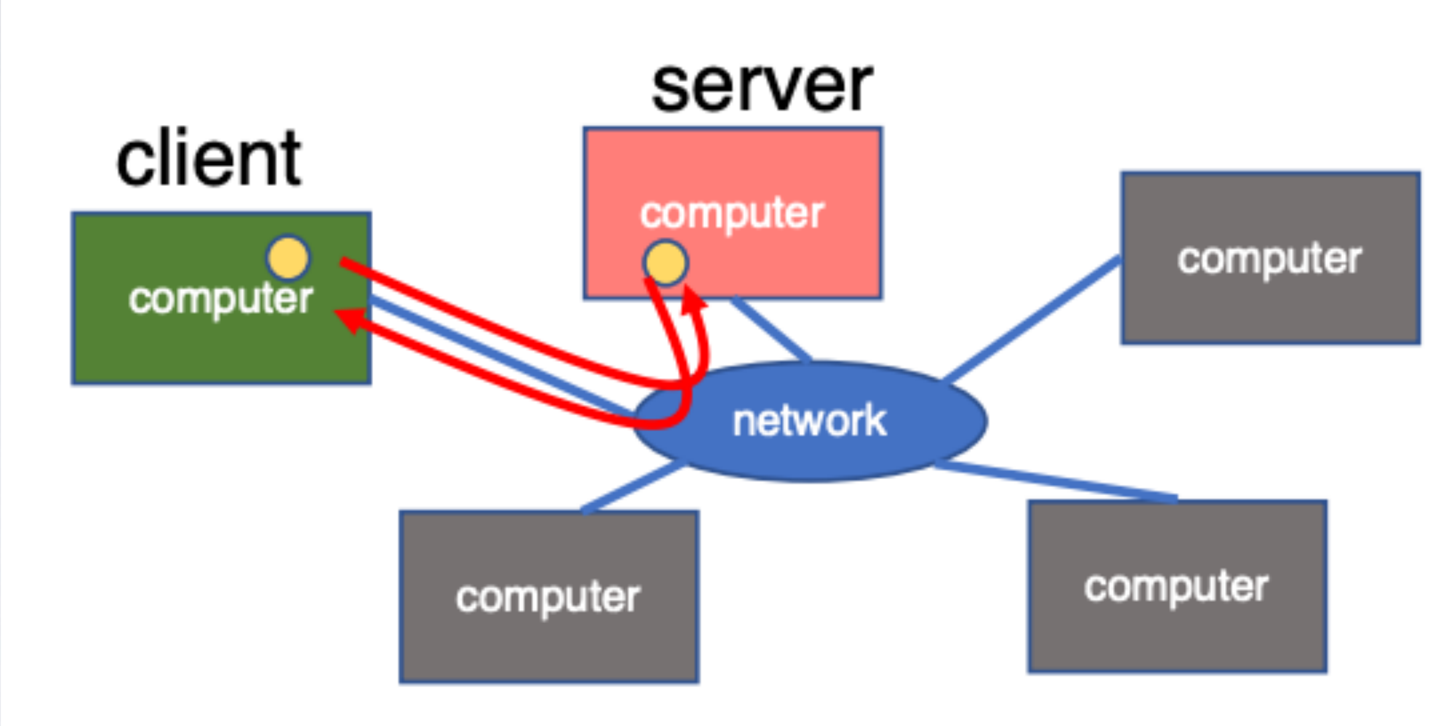
Naming



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

- Address processes/ports within system (host, id) pair
- Physical network address (Ethernet address)
- Network address (Internet IP address)
- Domain Name Service (DNS) provides resolution of canonical names to network address

Communication



How can one computer communicate with another?

- Raw Message - UDP
- Reliable Message - TCP
- Remote Procedure Call (RPC)
  and Remote Method Invocation(RMI)

Raw messaging

➡ Network programming = raw messaging (socket I/O)
programmers hand-coded messages to send requests and responses

- 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 (RPC)

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

- DCOM, CORBA, Java RMI, etc., are all basically just RPC
- NFS is implemented as a set of RPCs

➡ Someday you will most likely have to write an application that uses some form of RPC for remote communication (or you already have)

RPC example

Server Interface:

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

Client Program:

```
…
sum = server->Add(3,4);
…
```

Server Program:

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

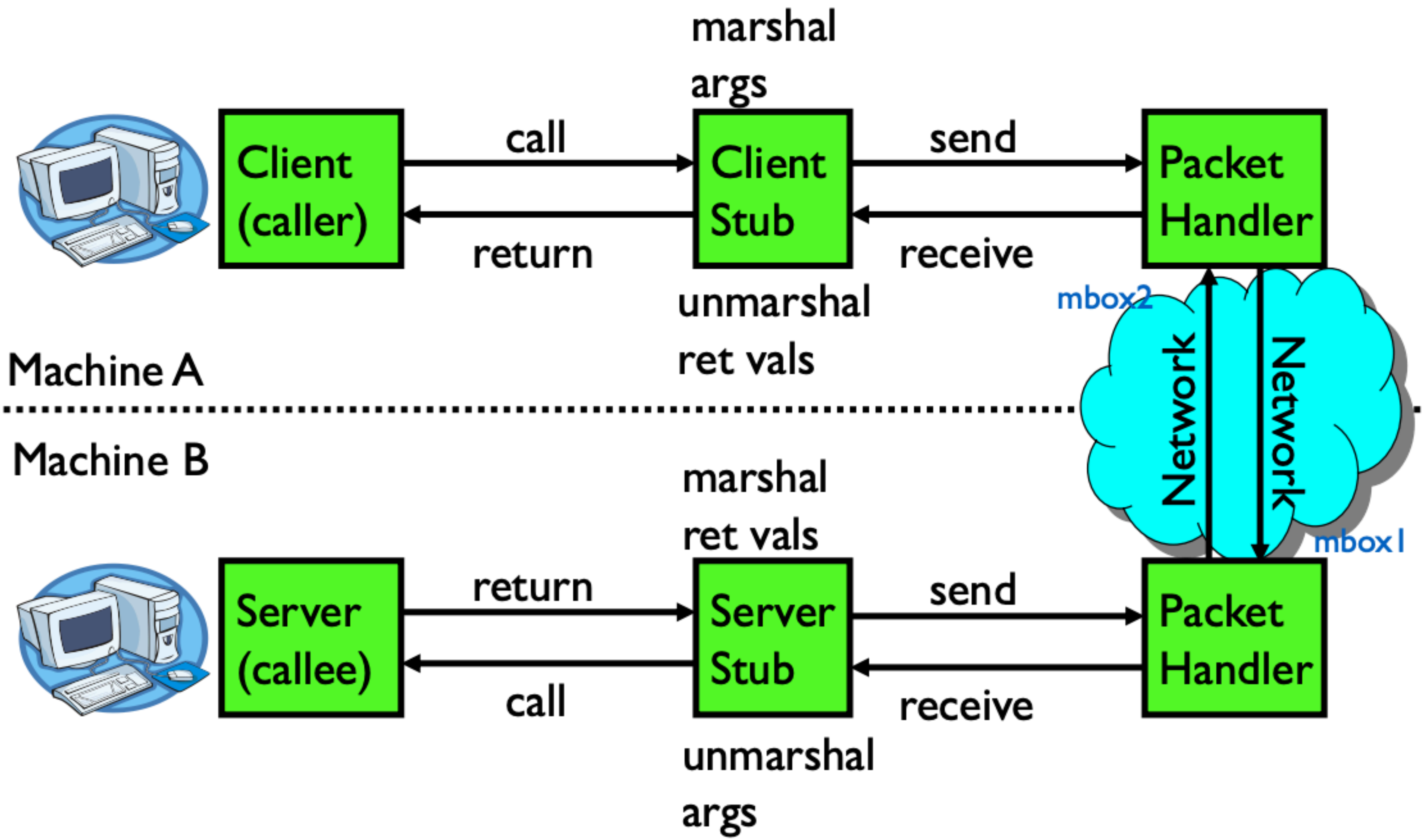
RPC model

- ➡ A server defines the server's interface using an Interface Definition Language (IDL) that 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 information flow



RPC stubs

➡ The stubs send messages to each other to make RPC happen transparently

- A client-side stub packs message, send it off, wait for result, unpack result and return to caller
- A server-side stub unpack message, call procedure, pack results, send them off

RPC marshalling

Marshalling is the packing of procedure parameters into a message packet

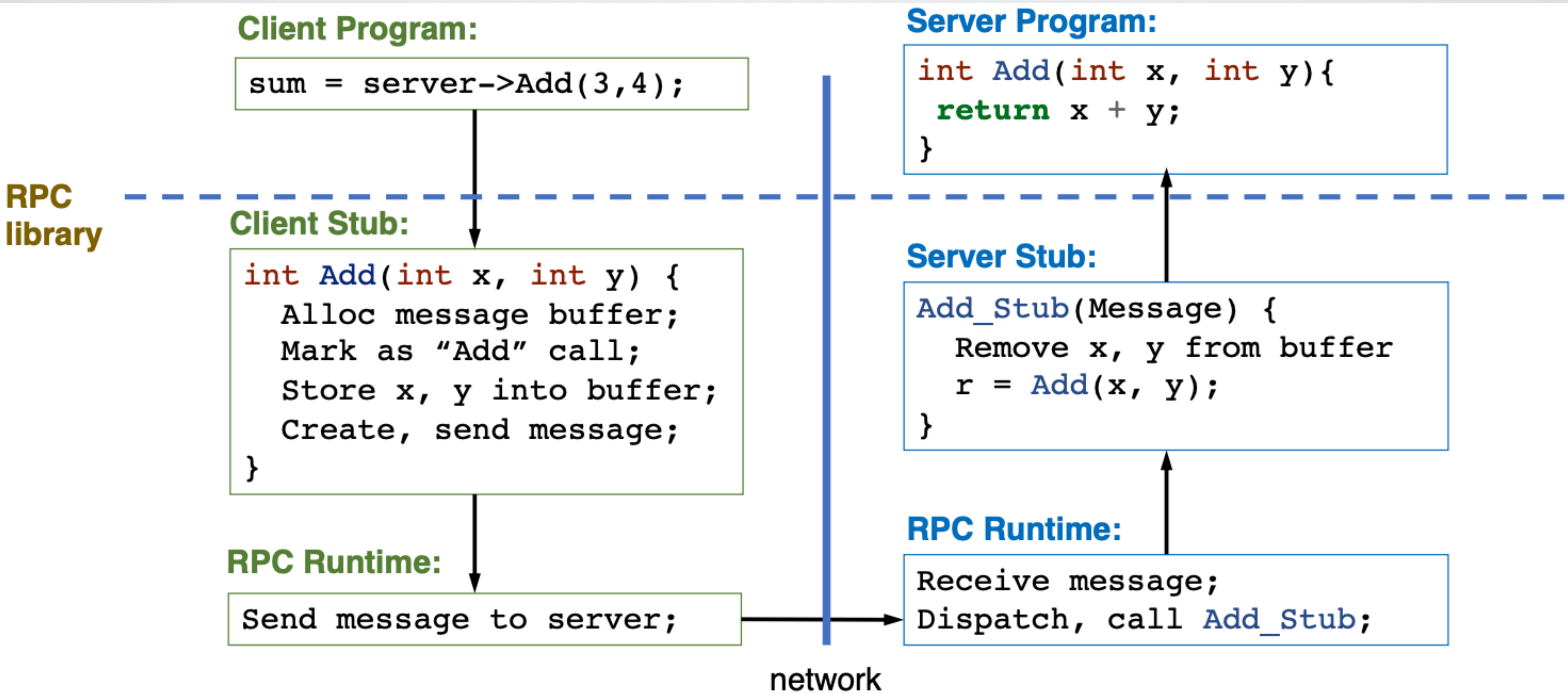
The RPC stubs call type-specific procedures to marshal (or unmarshal) the parameters to a call

- 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

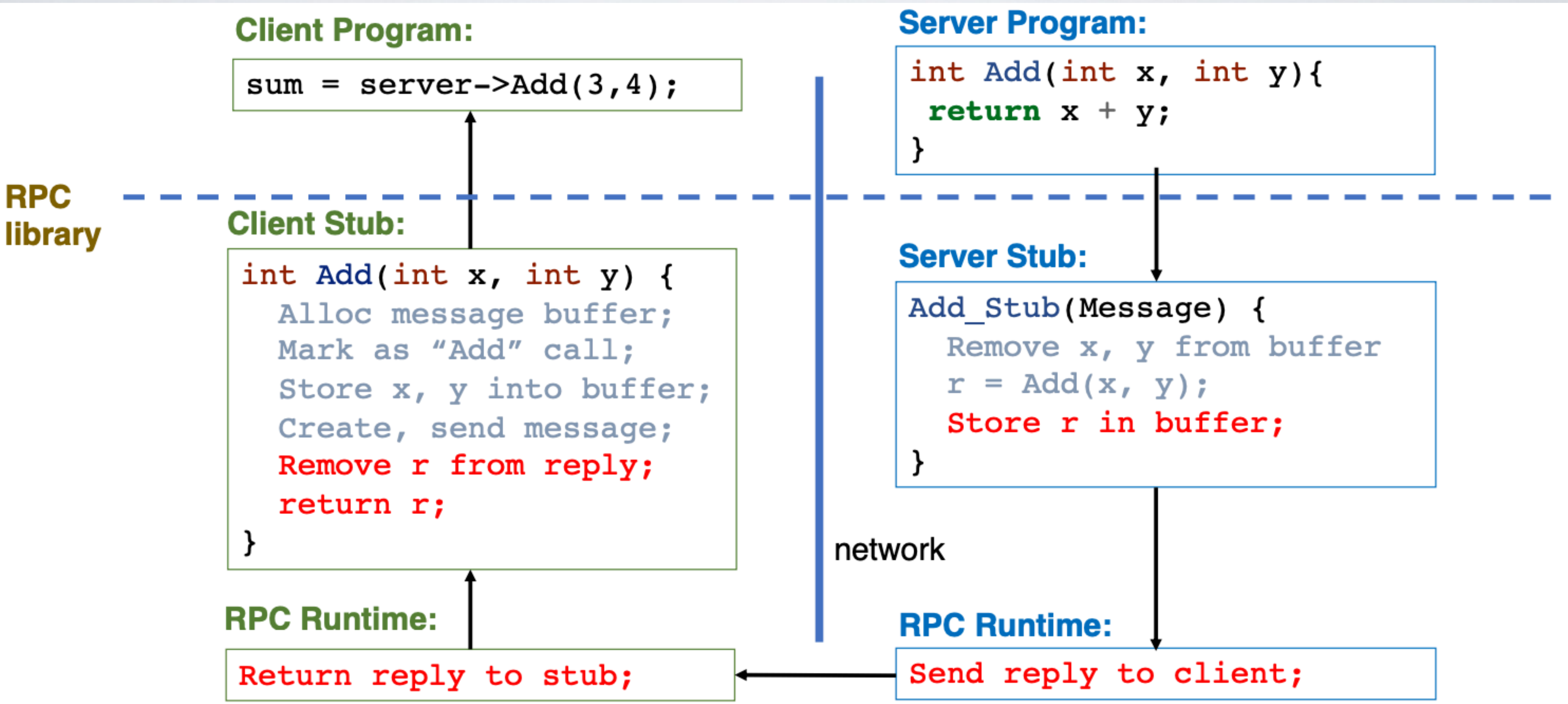
On return

- The server stub marshals the return parameters
- The client stub unmarshals return parameters and returns them to the client progra

RPC example - call



RPC example - return



RPC implementation details

What if client/server machines are different architectures and/or languages?
Need to convert everything to/from some canonical form and tag every item with an indication of how it is encoded (avoids unnecessary conversions)

- ➡ Abstract Syntax Notation One (ASN.1)

How does client know which server to send to?
Need to translate name of remote service into network endpoint (IP, port)

- ➡ Binding - the process of converting a user-visible name into a network endpoint
  - Static - fixed at compile time
  - Dynamic - performed at runtime

RPC transparency

One goal of RPC is to be as transparent as possible

➡ Make remote procedure calls look like local procedure call although binding can break transparency

What else?

- Failures – remote nodes/networks can fail in more ways than with local procedure calls
- Performance – remote communication is inherently slower than local communication

RPC failure semantic - at-least-once

What does a failure look like to the client RPC library?

- Client never sees a response from the server
- Client does not know whether the server processed the request

Simplest scheme - **at-least-once behavior**

- RPC library waits for response for time T, if none arrives, re-send the request
- Possibly repeat this a few times
- If still no response then return an error to the application

RPC failure semantic - at-most-once

- Problem with at-least-once behavior
  What if the request is "deduct $100 from bank account" ?

➡ At-least-once works well with <u>idempotent requests</u>

Another (better) RPC behavior - **at-most-once**

- ➡ Having 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

Problems with RPC - performance

Cost of Procedure Call ≪ same-machine RPC ≪ network RPC

➡ Means programmers must be aware that RPC is not free

RPC summary

RPC is the most common model for communication in distributed applications

- Some popular libraries such as *gRPC*
- "Cloaked" as DCOM, CORBA, Java RMI, etc.

➡ RPC is essentially language support for distributed programming

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*