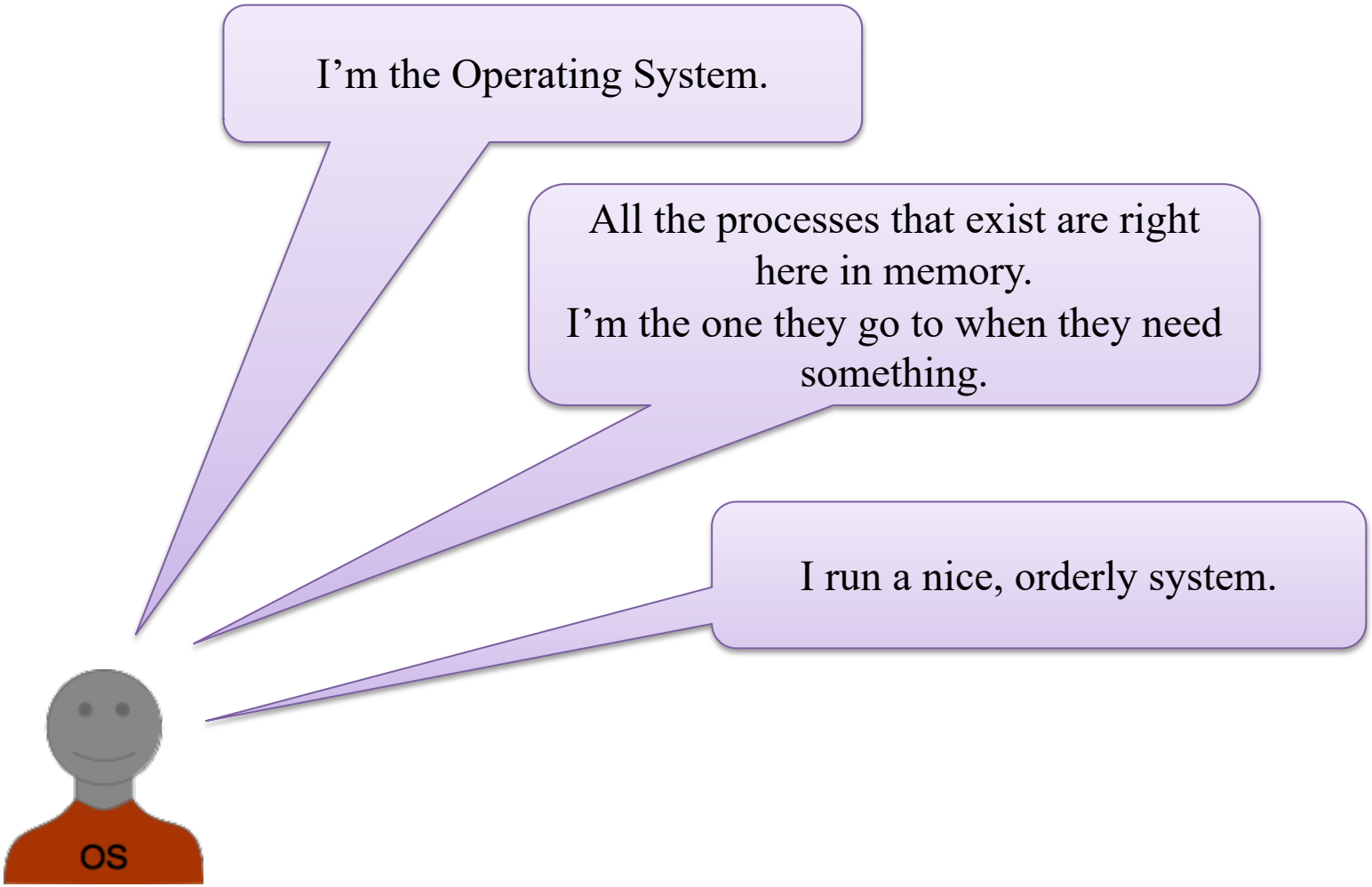
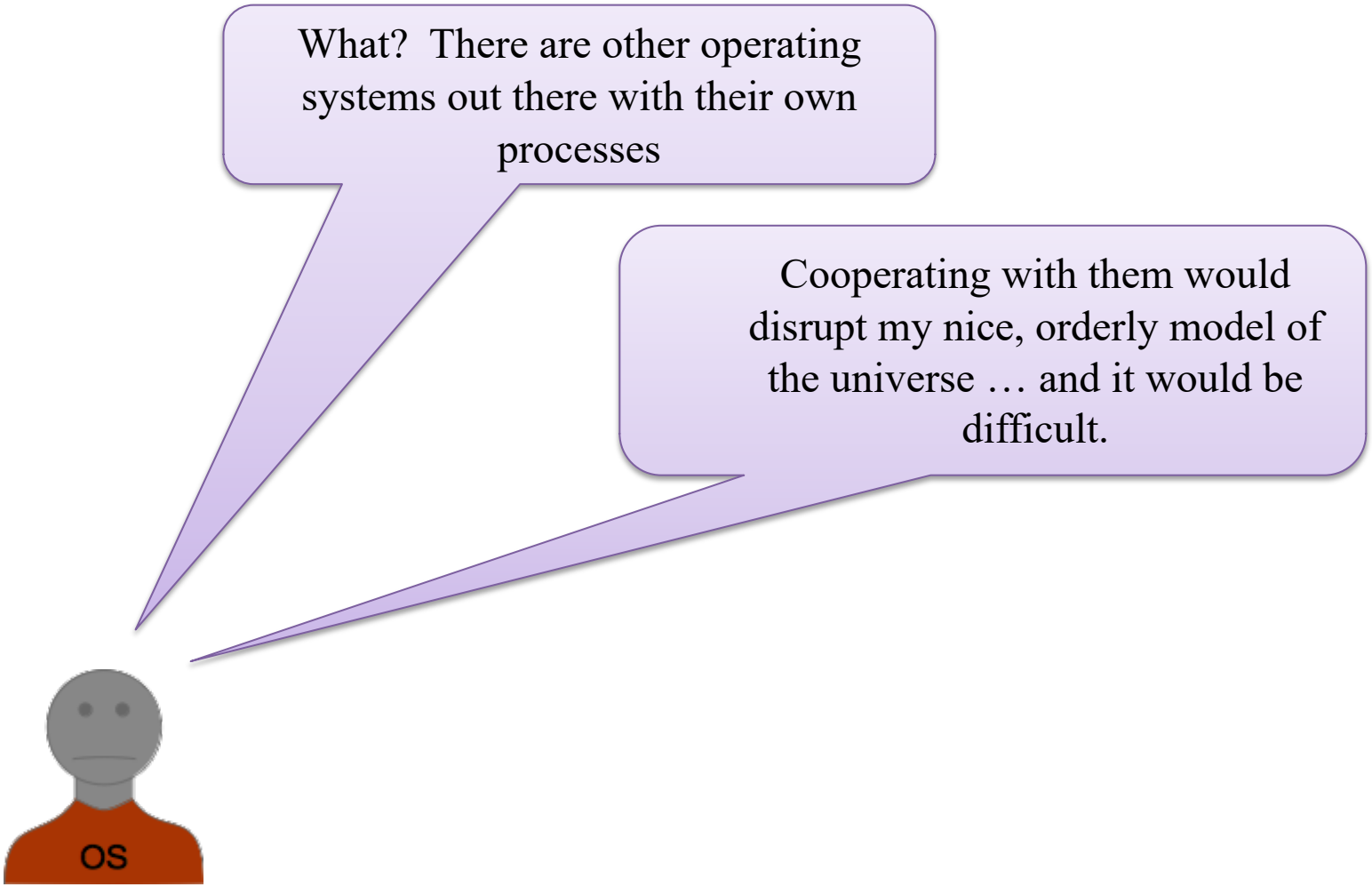
Distributed Systems

Chapter 19

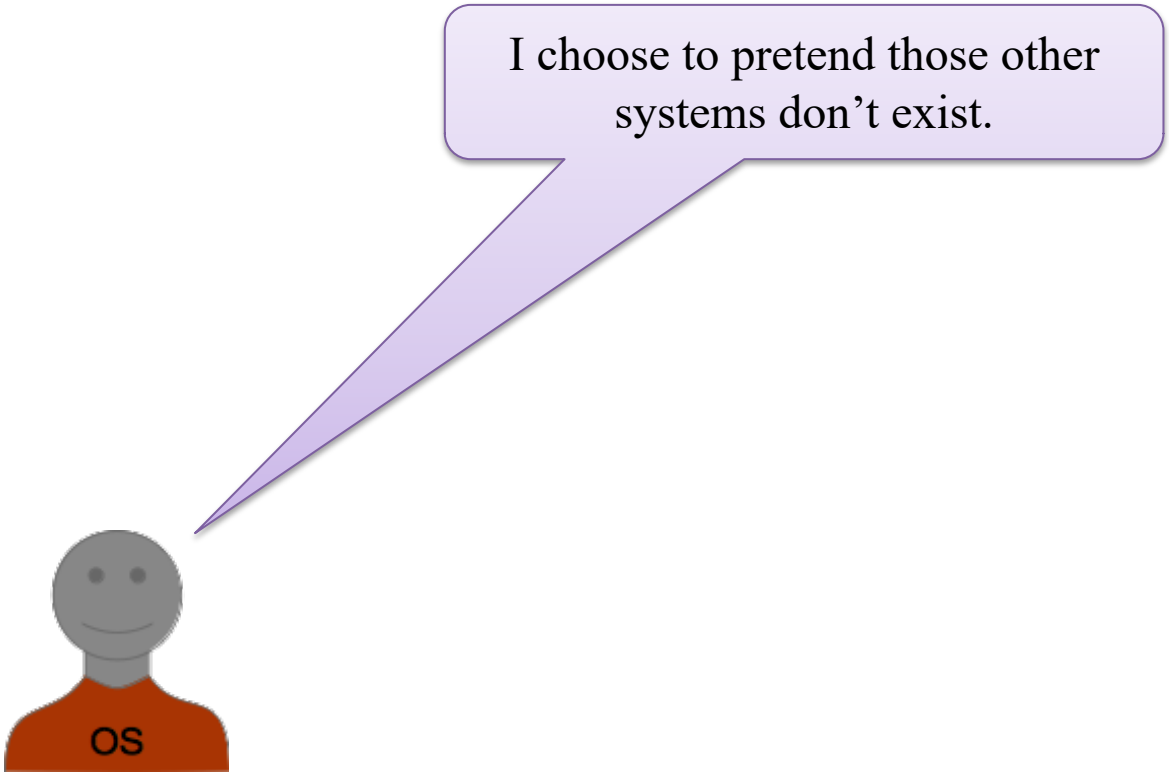
Centralized Computing



Centralized Computing



Centralized Computing



Distributed Computing

- But, there are some advantages to building computer systems so they can work together
  - *Resource sharing* : access remote servers, special-purpose devices, etc.
  - *Computational speedup* : sharing computational load among multiple hosts
  - *Reliability* : detect and recover from a failed host, other hosts to pick up the slack
  - *Communication* : between users on different systems
  - *Cost effectiveness* : aggregation of inexpensive devices, gradually add systems as demand grows

Distributed Operating Systems

- Ideally, the programmer wouldn't need to constantly think about host boundaries
- Remote and local resources could look and behave the same
- These are the goals of a *distributed operating system*.
- It would provide automatic mechanisms for
  - *Data migration* : move the data (e.g., files) to host that's using them.
  - *Computation migration* : execute computation steps (subroutines) on the host with needed resources.
  - *Process migration* : move a whole program after it has started running (for load balancing, data access, etc)

Distributed System Design Goals

- *Scalability* : can we aggregate resources and get more power gradually, as we add more resources
- *Robustness* : can we tolerate failure of individual components (hosts, software, network)
- *Availability* : are data services always there for clients
- *Heterogeneity* different types of hosts and devices can cooperate, including differences in byte order, instruction set, computational speed

Part-Way There

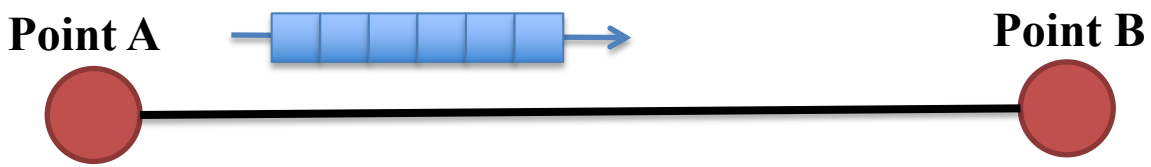
- It's hard to satisfy all these goals for all operating systems services, so …
  - We develop special-purpose solutions for particular types of services
    - e.g., a distributed file system
    - e.g., map-reduce for distributed high-performance computing
  - Elsewhere, we put the responsibility on the application developer
- Generally, we have *network operating systems*
  - We can use the network, like any other device
  - But, accessing remote resources often requires special effort

Why Networking?

- We have a whole class on that …
  - … but it's not a required class
- Everyone should write some code that communicates over a network
  - So, in this class, that's what you get to do.

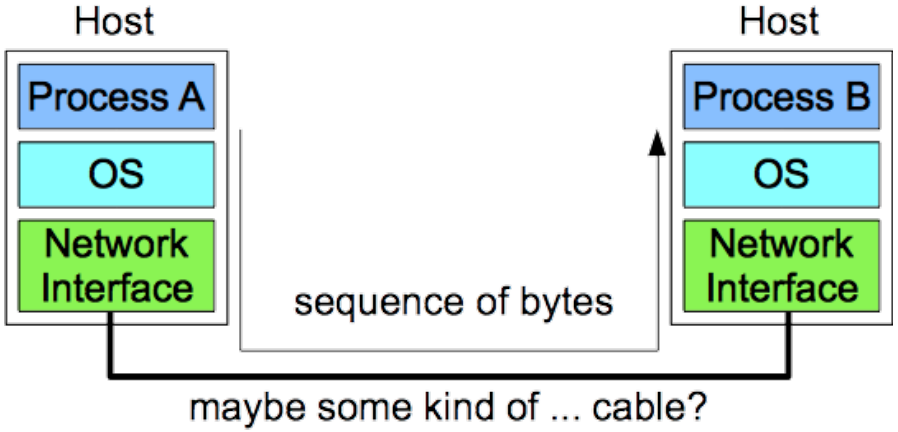
Getting from Point A to Point B

- We want to be able to send messages between processes on different hosts
  - Ideally, we just encode a message as an array of bytes
  - Send the array to the receiver
  - Then, make sense of the message on the receiver



Simple, Right?

- The network is just a new type of device.
- Two processes could:
  - Request it
  - Use it to send / receive messages
  - Release it when they're done



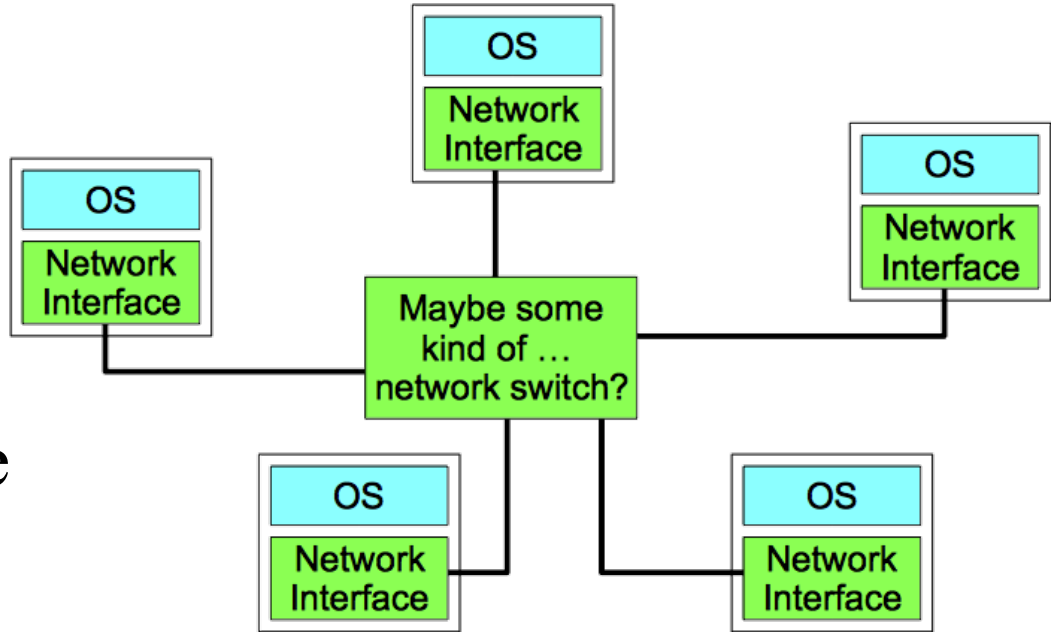
- There are two big problems with this.

Not So Simple

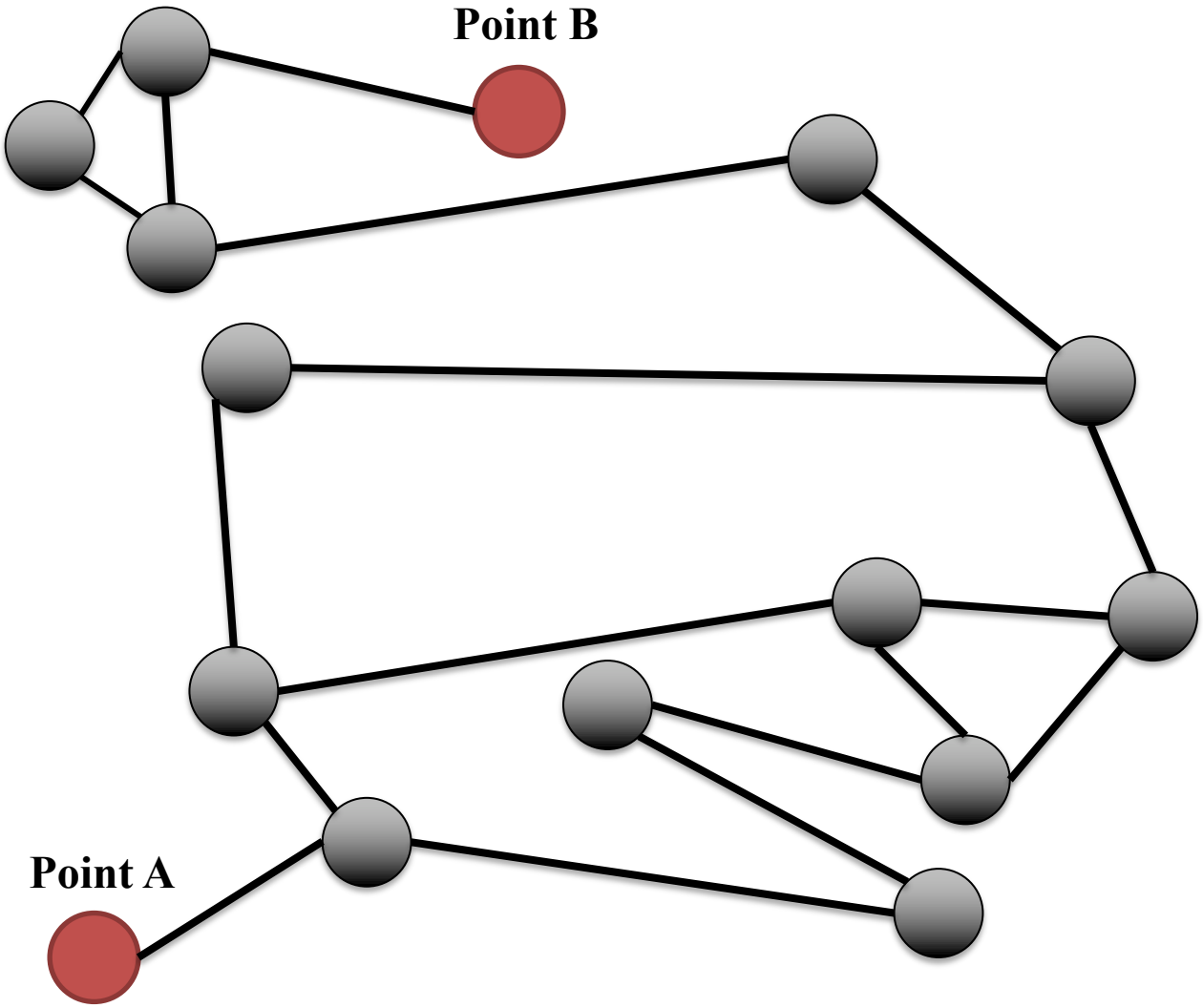
- Many processes may want to use the network
  - We need to make it shareable,
  - Like we did with memory, disk, CPU, etc.
- Hosts may not be directly connected.

Network Topology

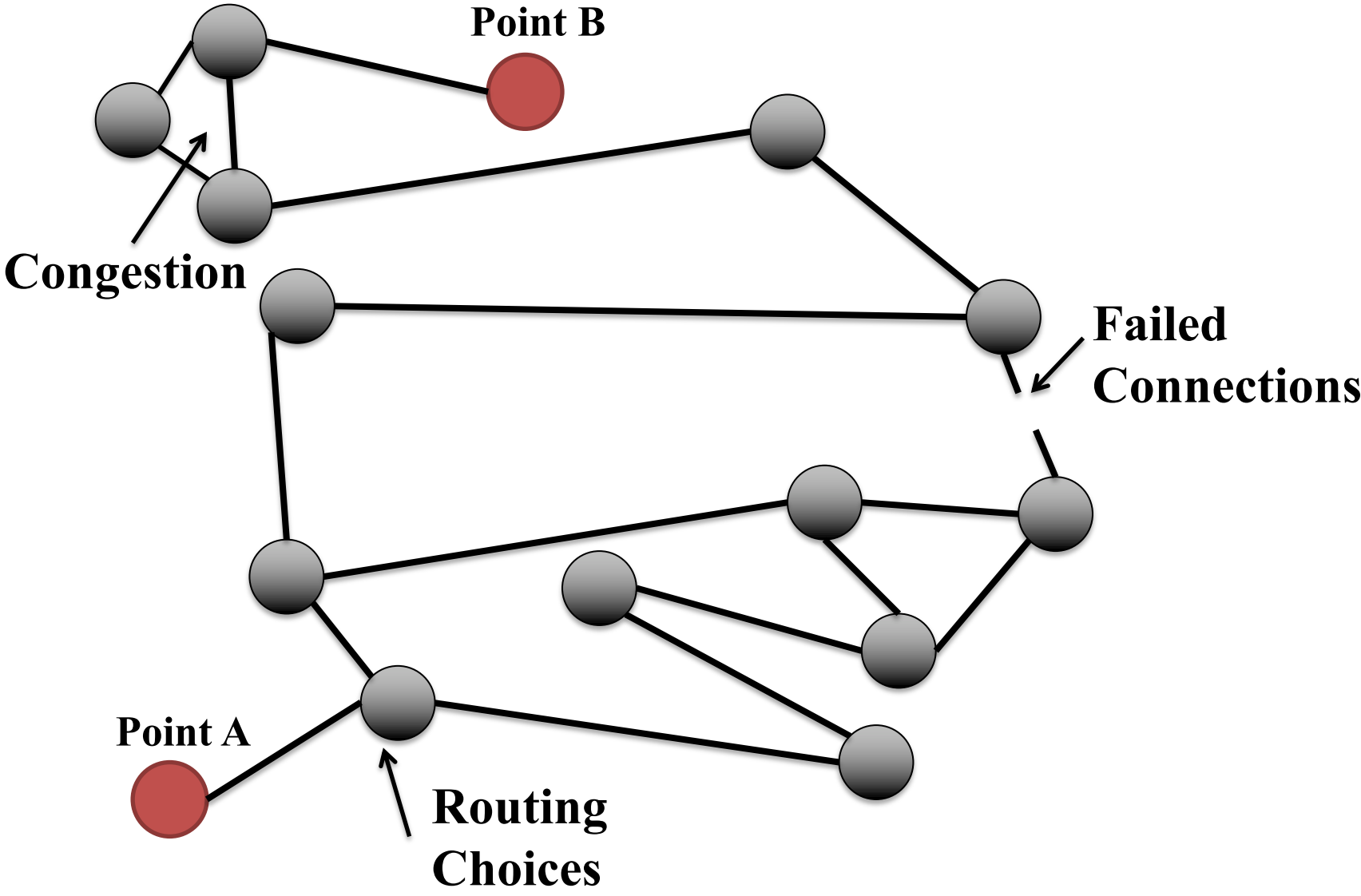
- Focusing on a *Local Area Network* can simplify some issues.
  - These networks have a simple, predictable structure.
  - making *routing* decisions straightforward.
  - and, limiting the scope of possible failures.



A Wide-Area Network may look more like this.



With Hazards Along the Way

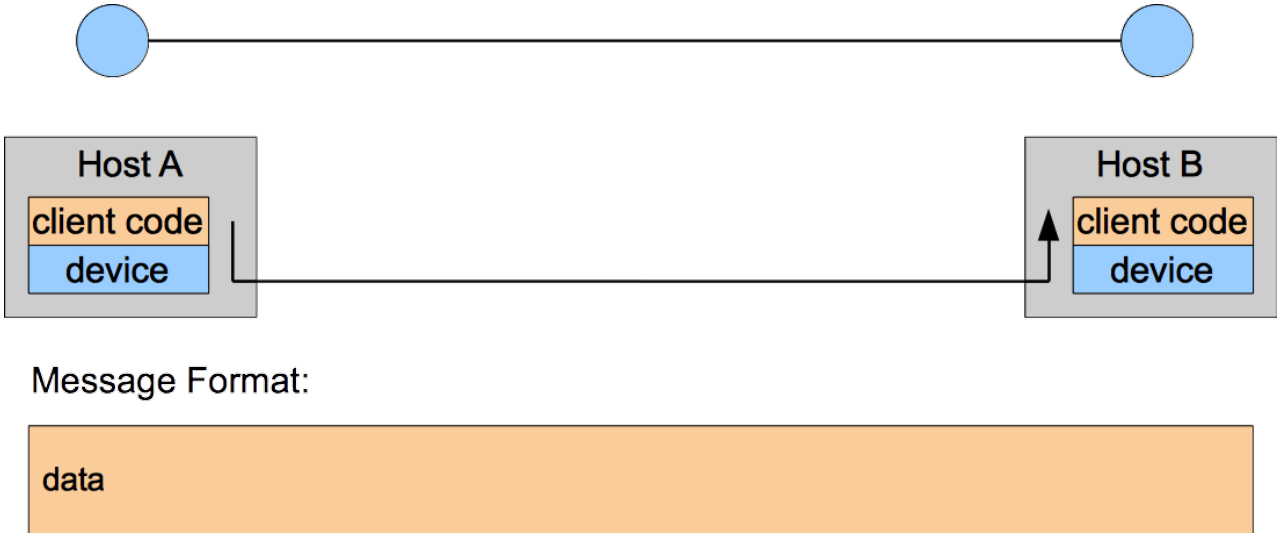


Building Blocks

- Independent hosts
  - Each with its own memory, running its own software
  - Possibly with different architectures and operating systems
  - Multiple applications on each host
  - Not reliable. Host or software components may fail
- Network interconnection
  - Some way to send messages between systems
  - Typically built as an irregular structure of links
  - Routing decisions may need to be made … and they may change
  - Other, competing application can cause congestion
  - Not reliable. Messages may be lost or delayed, links may fail

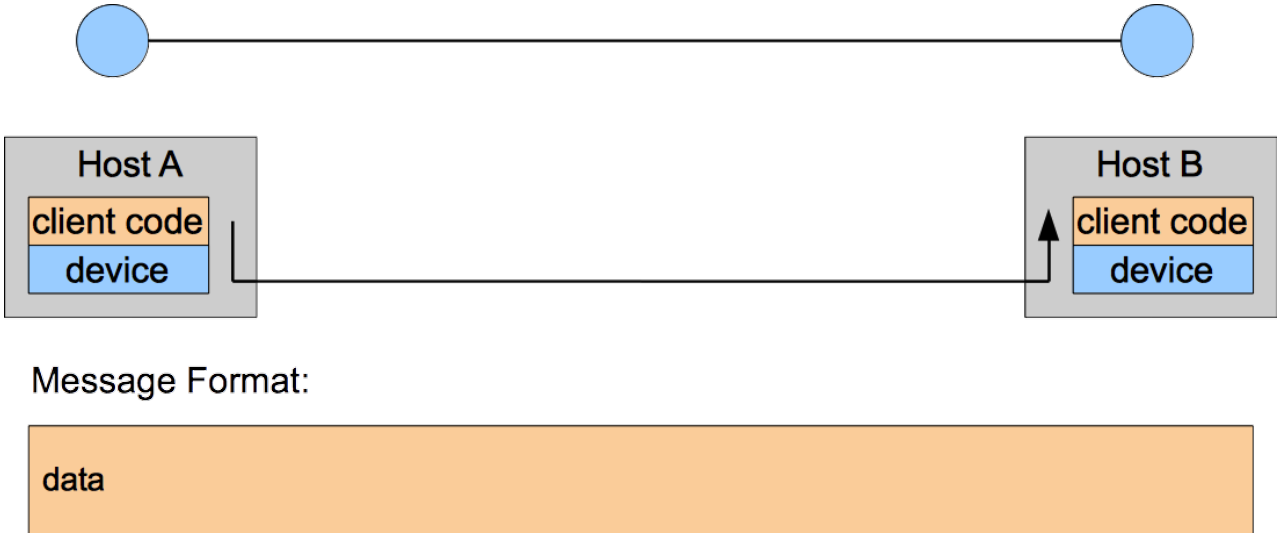
Getting from Point A to Point B

- How about if we agree on:
  - How to build hardware communication hardware, device controllers (interface controller) and links
  - How to encode/decode a bit stream from the medium
  - Transmissions speeds, etc.



Getting from Point A to Point B

- Is this enough?
  - Well, no. ☹
  - What if there are errors during transmission?
  - What if hosts A and B aren't directly connected?
  - How do multiple applications share this connection?

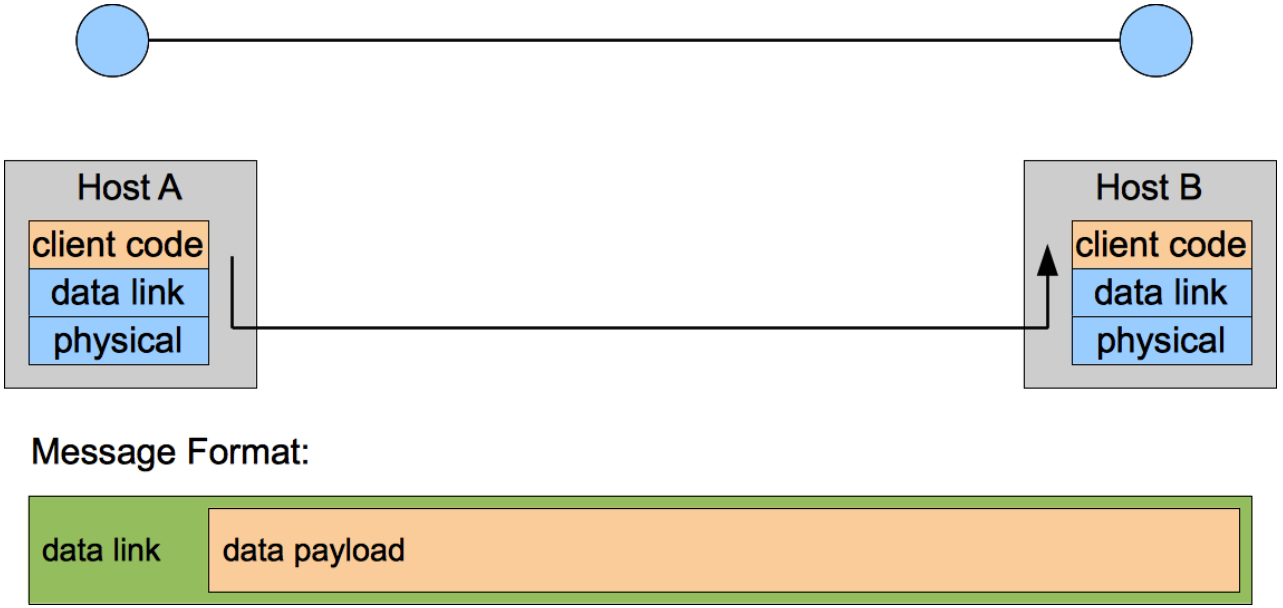


Getting from Point A to Point B

- OK, we're just getting started
  - This is what our network will look like, physically …
  - but it's not what it will look like to applications
- It's just the *physical layer*
  - Examples: (part of) Ethernet, (part of) 802.11 (wireless)
- The OS will include more software to dress up this interface:
  - Make it shared
  - Make it more reliable
  - Offer different semantics to applications
- But, here's something new
  - Different operating systems will need to agree about what this software does, and how.

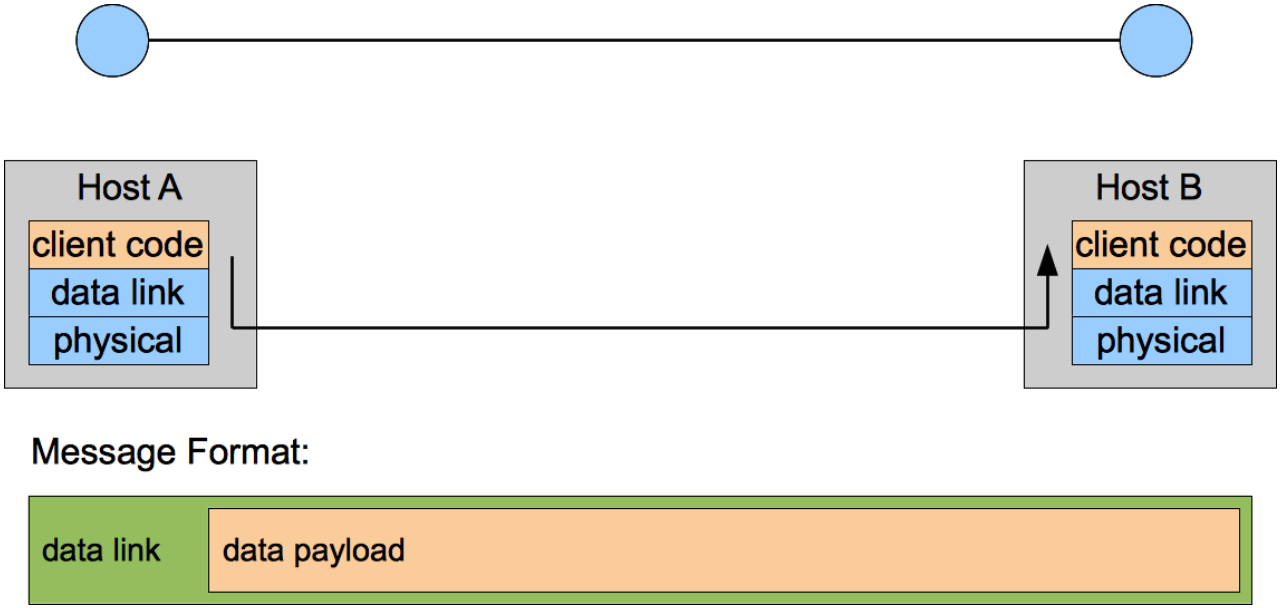
Getting from Point A to Point B

- The *data link layer*
  - Organize data into fixed- or variable-length *frames*, to let systems share a connection.
  - Support local routing decisions
  - Correct errors introduced in the physical layer
  - A header to help do all of this.



Getting from Point A to Point B

- The *data link layer*
  - Delivery of frames within a local-area network (with only basic routing decisions)
  - Example: (the rest of) Ethernet or 802.11

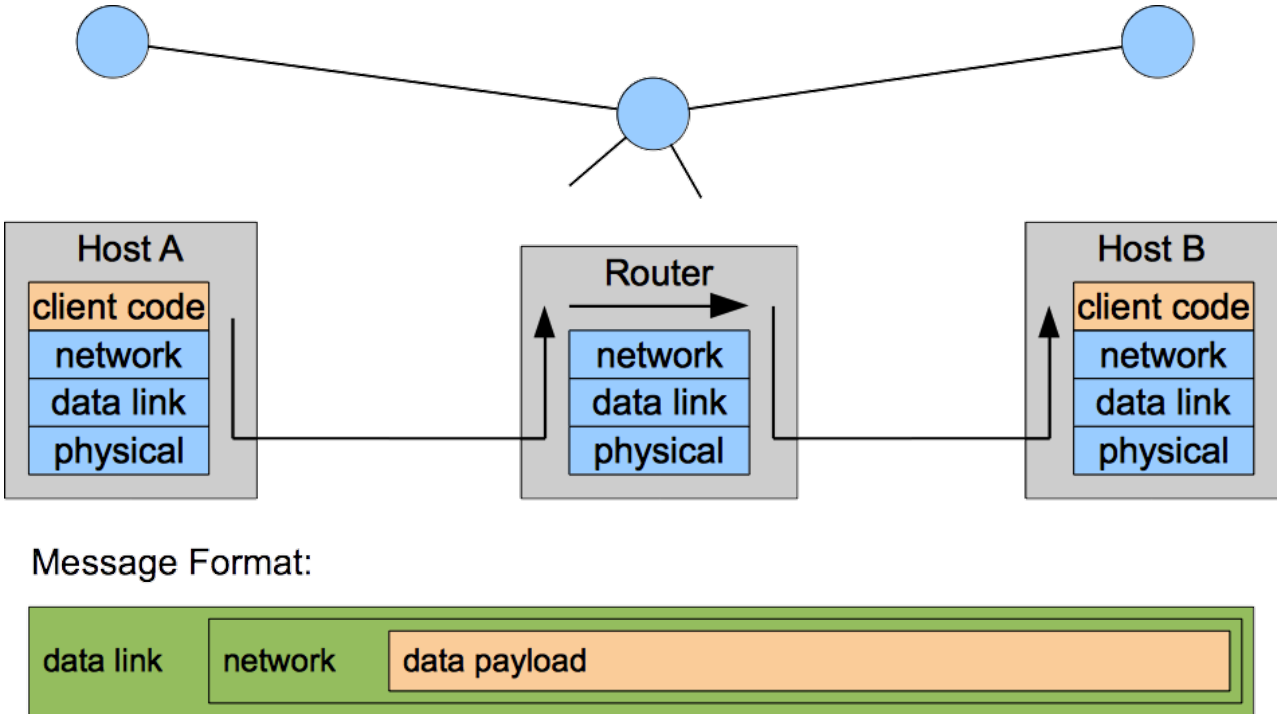


Getting from Point A to Point B

- Happy Now?
  - Well, not as happy is I could be. 😐
  - What if hosts A and B aren't part of the same local network?
  - How do multiple applications share this connection?
- OK. We're just getting warmed up.
  - We just need more layers!

Getting from Point A to Point B

- The *network layer* handles non-trivial routing.
  - Organize data into variable-length *packets*
  - Additional header information to make routing data for sending data across multiple connections
  - Example: Internet Protocol (IP)

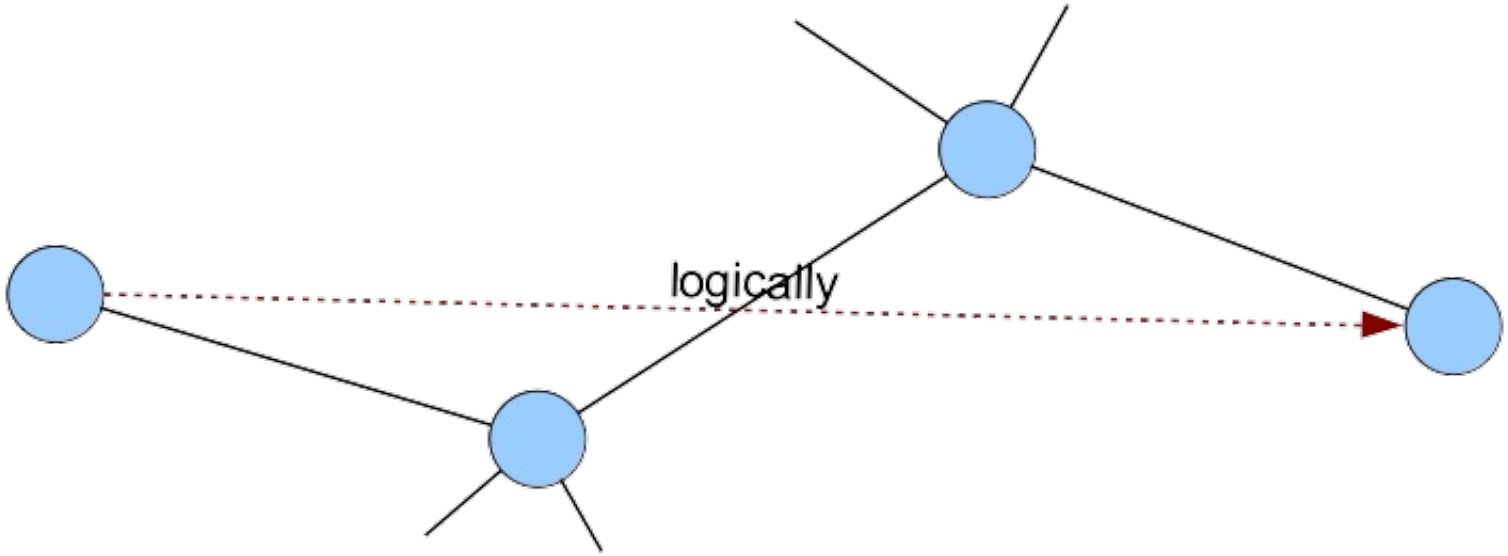


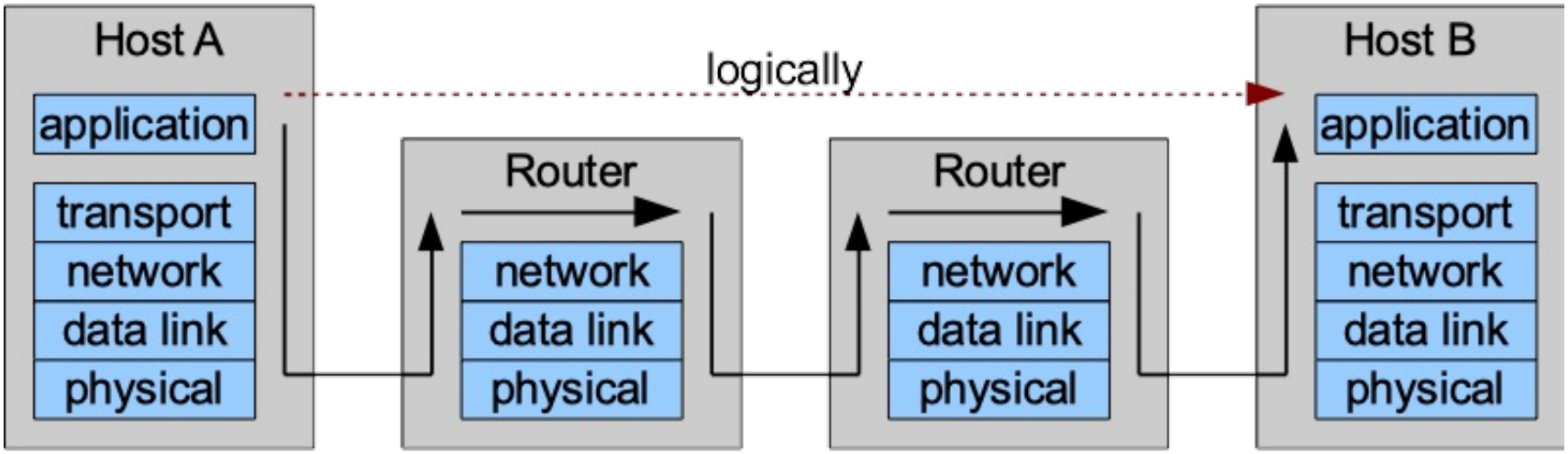
About IP

- Internet Protocol Provides
  - A best-effort packet delivery service between hosts
  - Each host represented by a 32-bit IP address (128-bit for IPv6)
  - Each packet stamped with its source and destination address
  - Congestion may cause packets to be dropped
    - Or errors, but congestion is the typical reason for packet loss
  - Routing changes may reorder packets during transmission
    - But, retransmission to recover from lost packets is a more typical reason for reordering.
  - No error detection/correction for payload
  - No congestion control

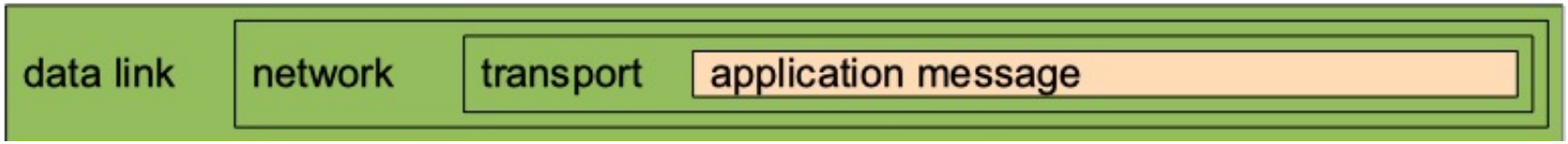
Getting from Point A to Point B

- The *transport layer* makes this useful to individual applications.
  - Dispatch packet data to individual processes
  - Provide useful application semantics on top of the network layer
  - Create the illusion of messages or connections for clients
  - Break messages into packets and re-assemble for on arrival
  - Examples:
    - Transmission Control Protocol (TCP)
    - User Datagram Protocol (UDP)

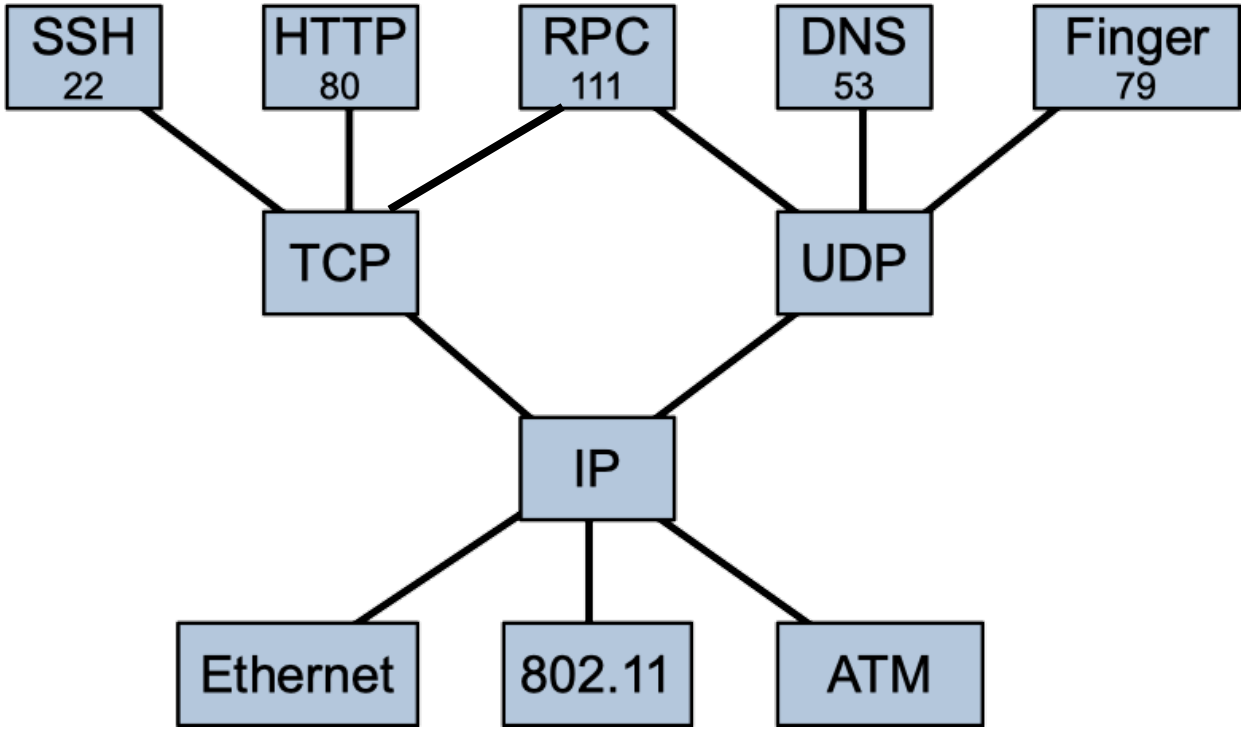




Message Format:



Internet Hierarchy



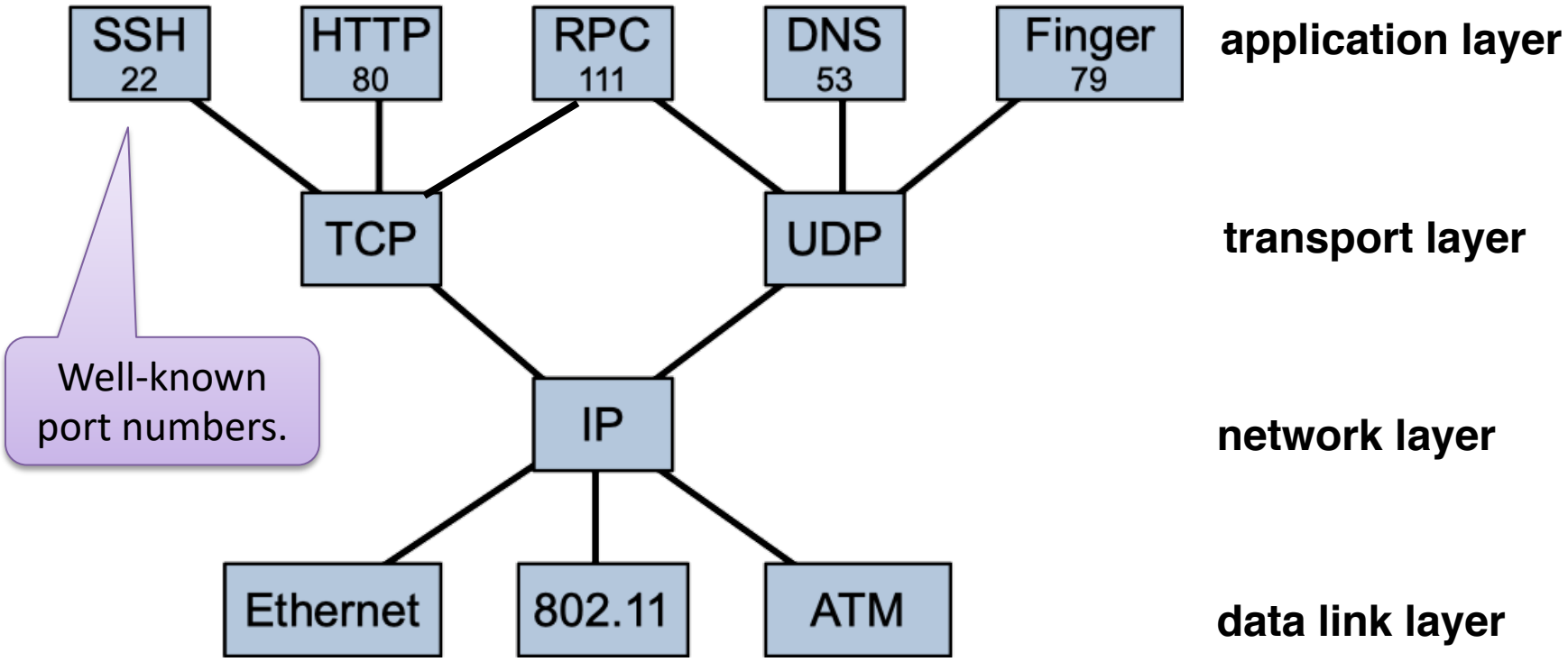
application layer

transport layer

network layer

data link layer

Internet Hierarchy

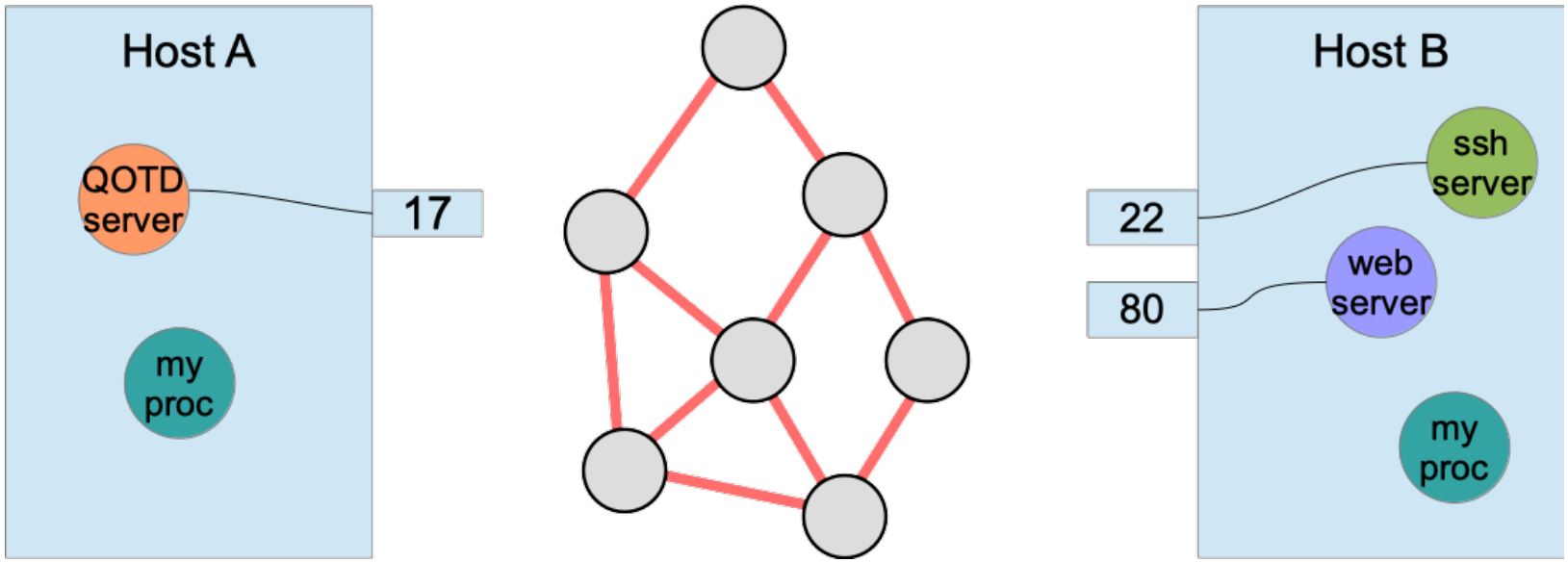


About UDP

- *User Datagram Protocol* Provides
  - A best-effort message delivery service between applications
  - No need to establish a connection first, but messages may be lost
  - Isn't this the same as IP?
  - Well, UDP gives us a little bit more
    - Error detection within the message
    - 16-bit *port numbers* to dispatch messages to the right process on the host

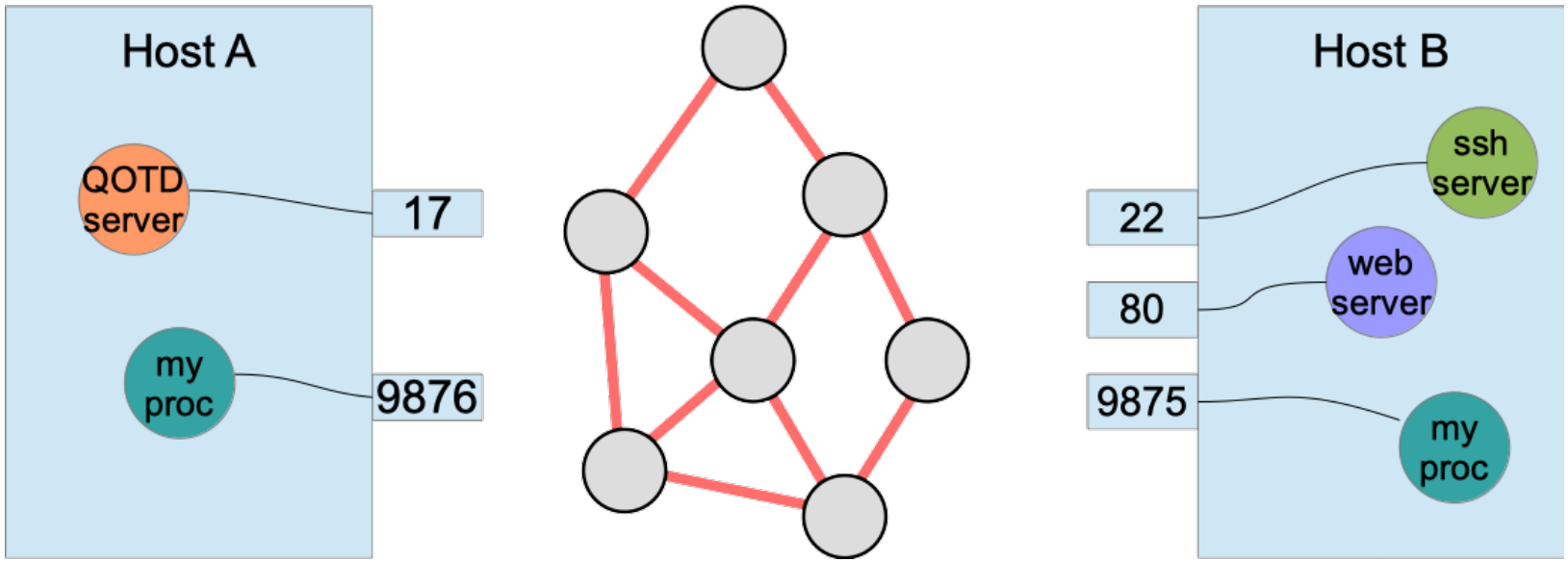
Using UDP

- You have a pair of processes that want to communicate



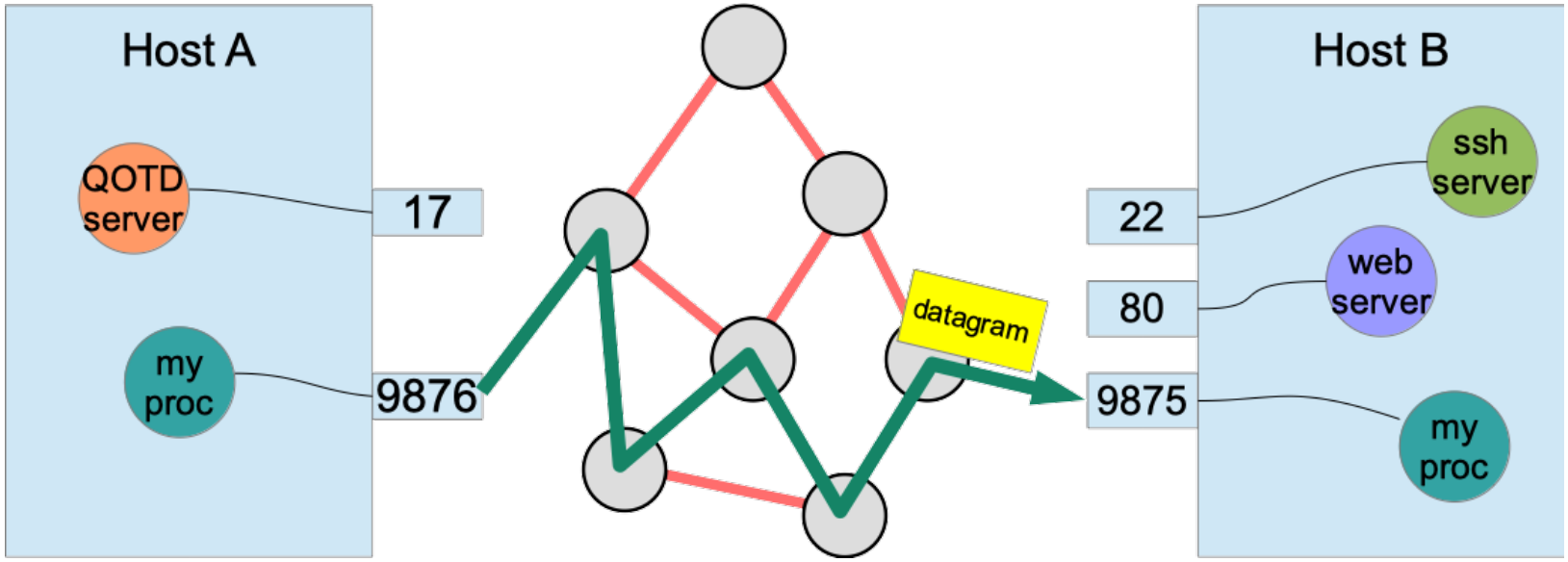
Using UDP

- They will both create a datagram socket



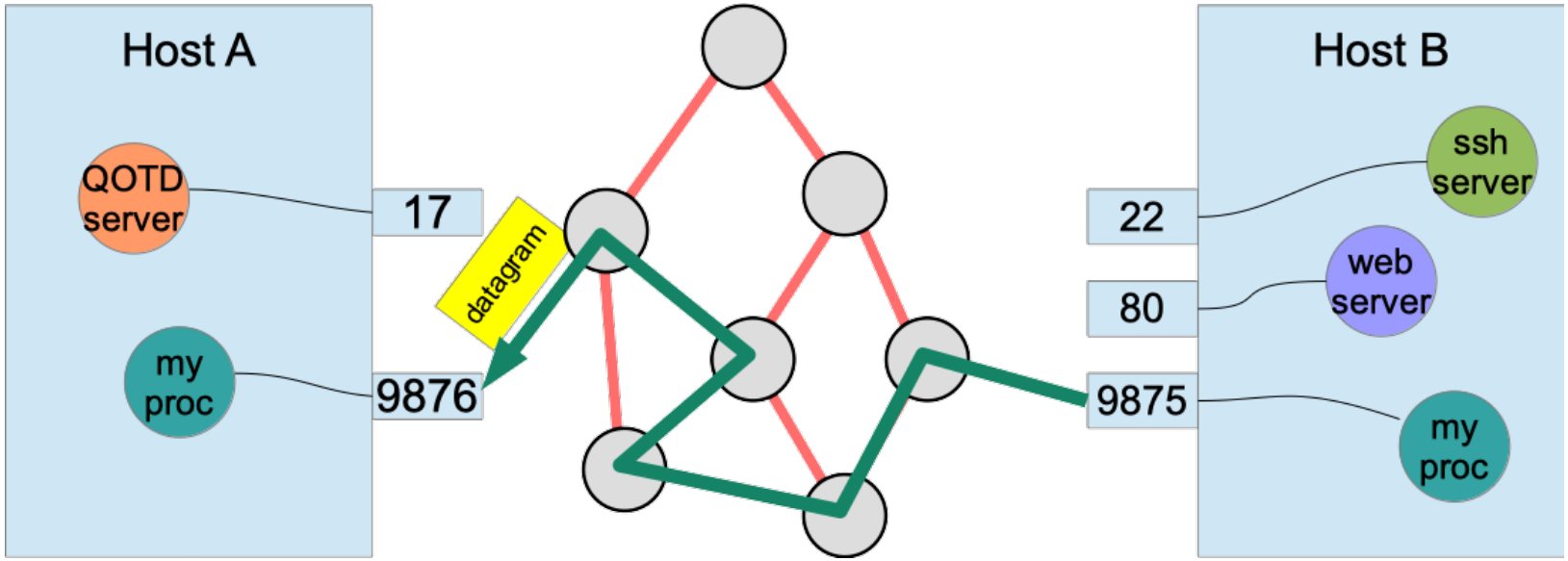
Using UDP

- No need to establish a connection
- They can just send and receive datagrams



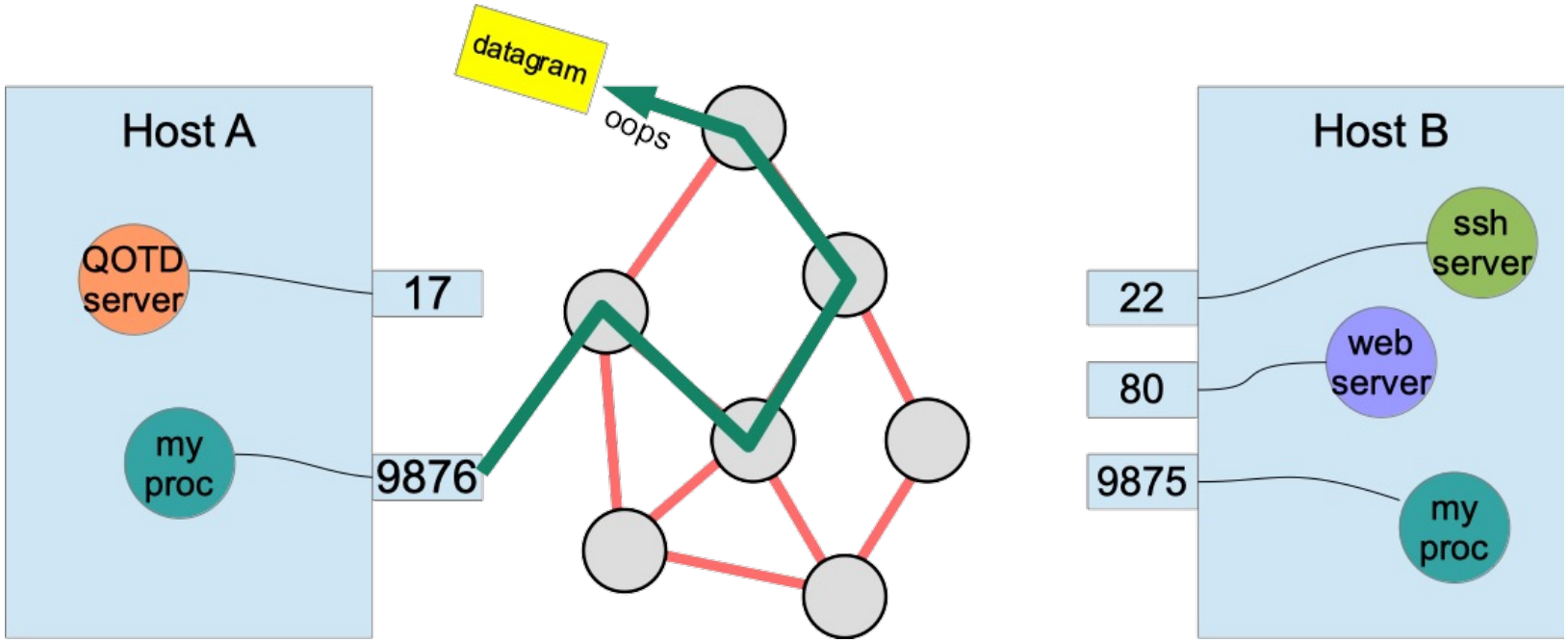
Using UDP

- Back and forth, all day long



Using UDP

- Of course, we're using a best-effort packet delivery service

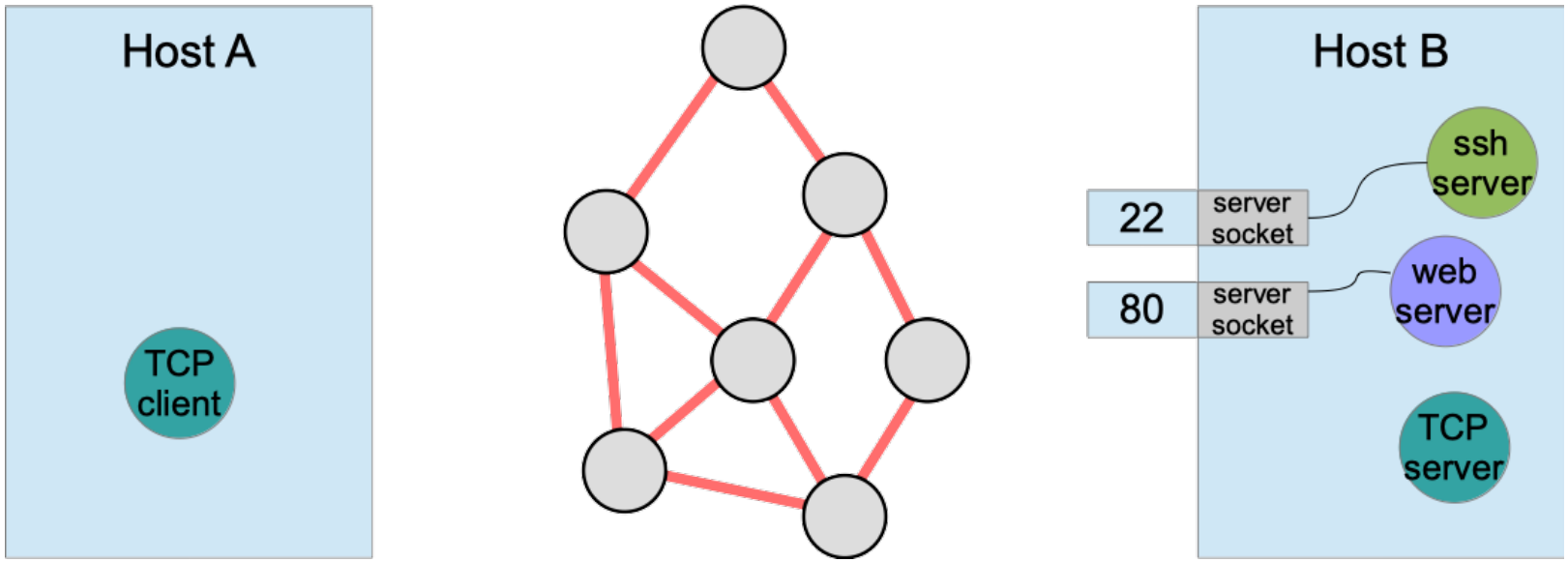


About TCP

- Transmission Control Protocol Provides a different model
  - Connection must be established first
  - In-order byte stream delivery
    - Packet boundaries are hidden
    - Message boundaries are not respected
  - Reliability via acknowledgement and timeout → retransmit
  - Congestion control, slow down the sender if the network is overwhelmed
    - How? Periodically block the sender just like bounded buffer

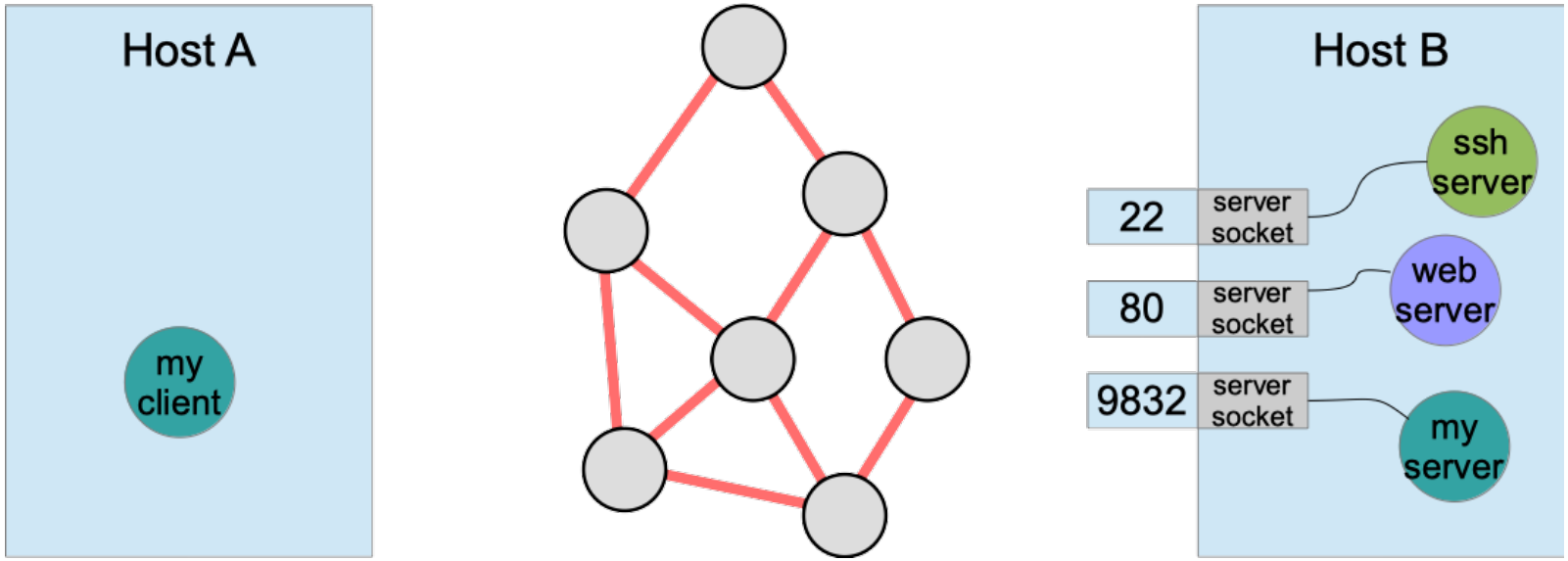
Using TCP

- You have a pair of processes, one of them will play the role of the server



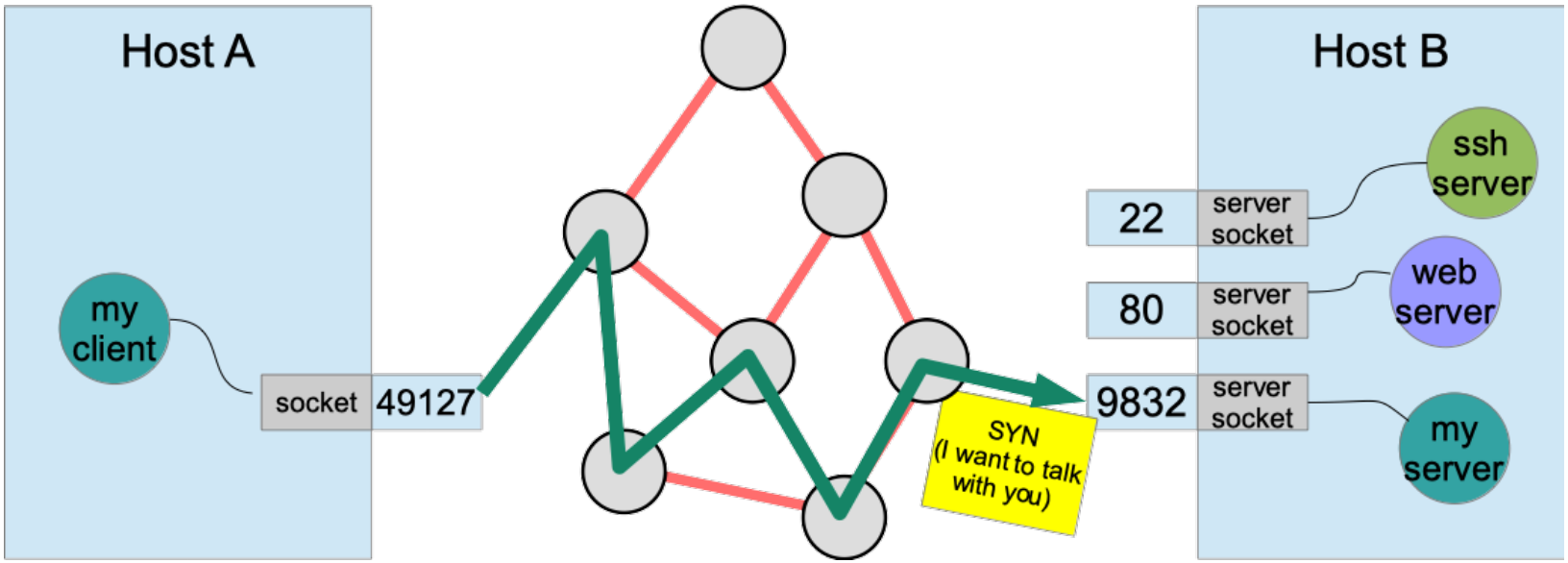
Using TCP

- The server will *listen* for connections at a particular port



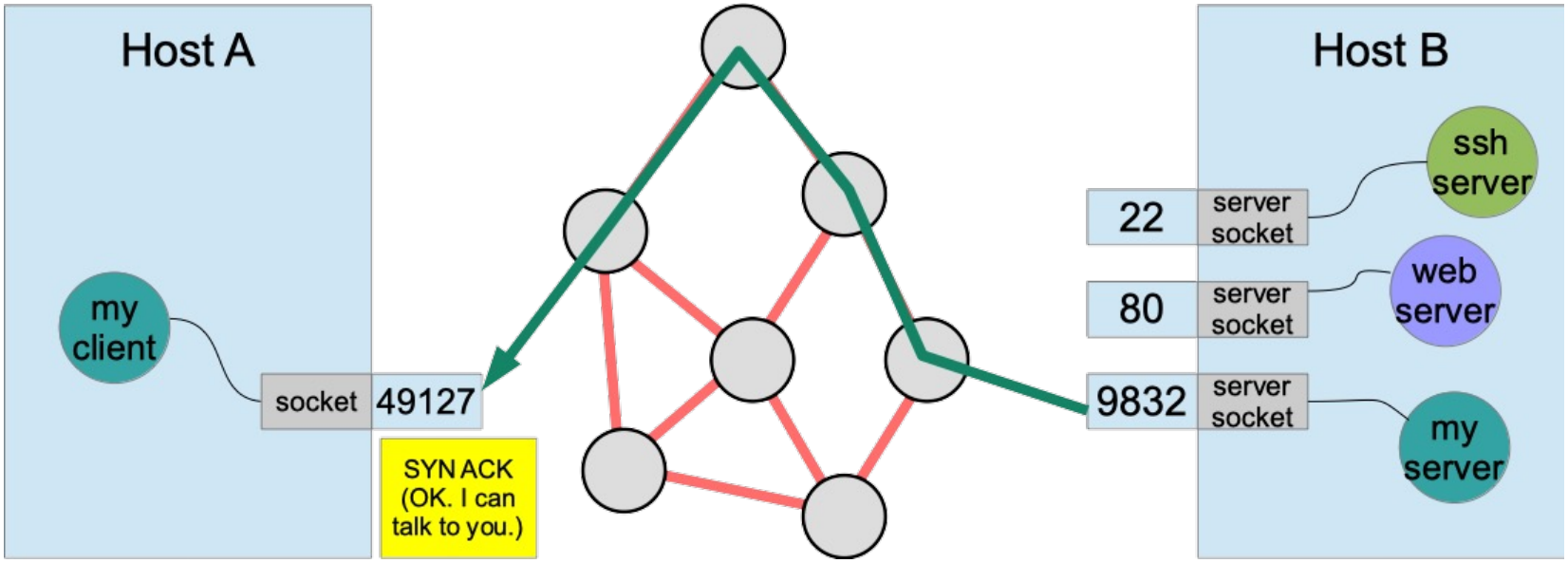
Using TCP

- The client will make a *socket* and try to *connect* with the server (by sending it a message at that port)



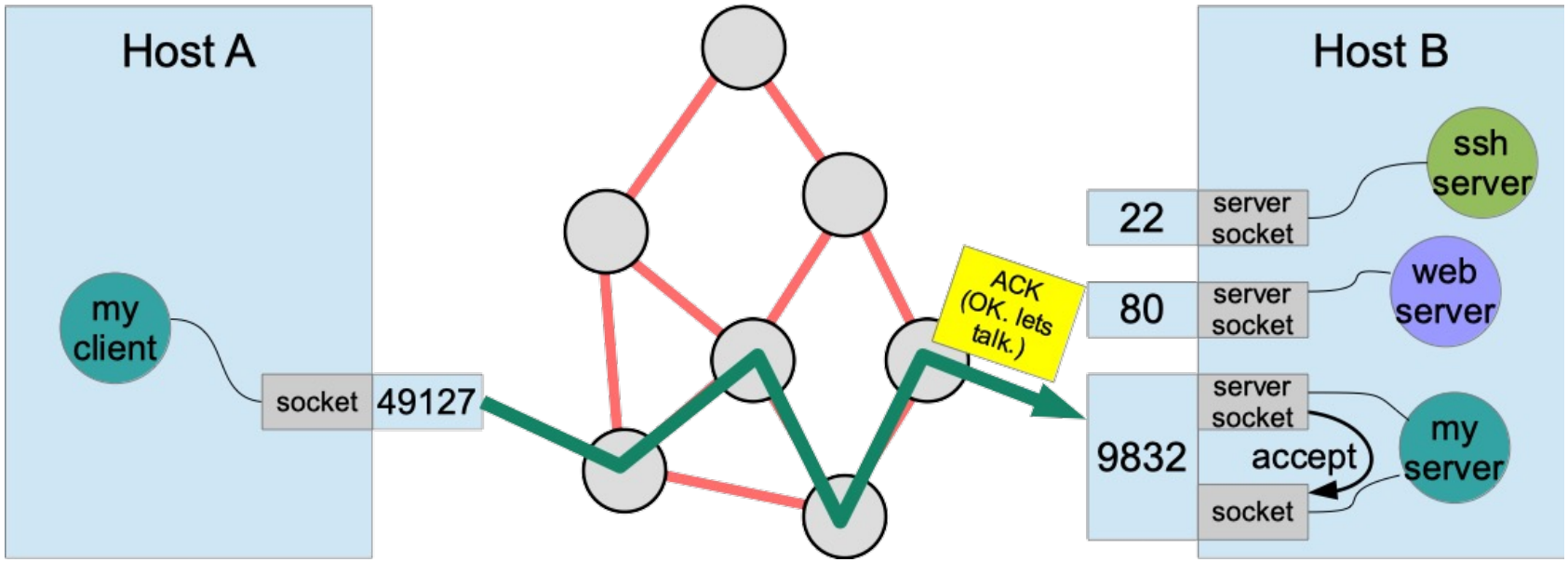
Using TCP

- The server will send a response (acknowledging the connection request and sending back some information)



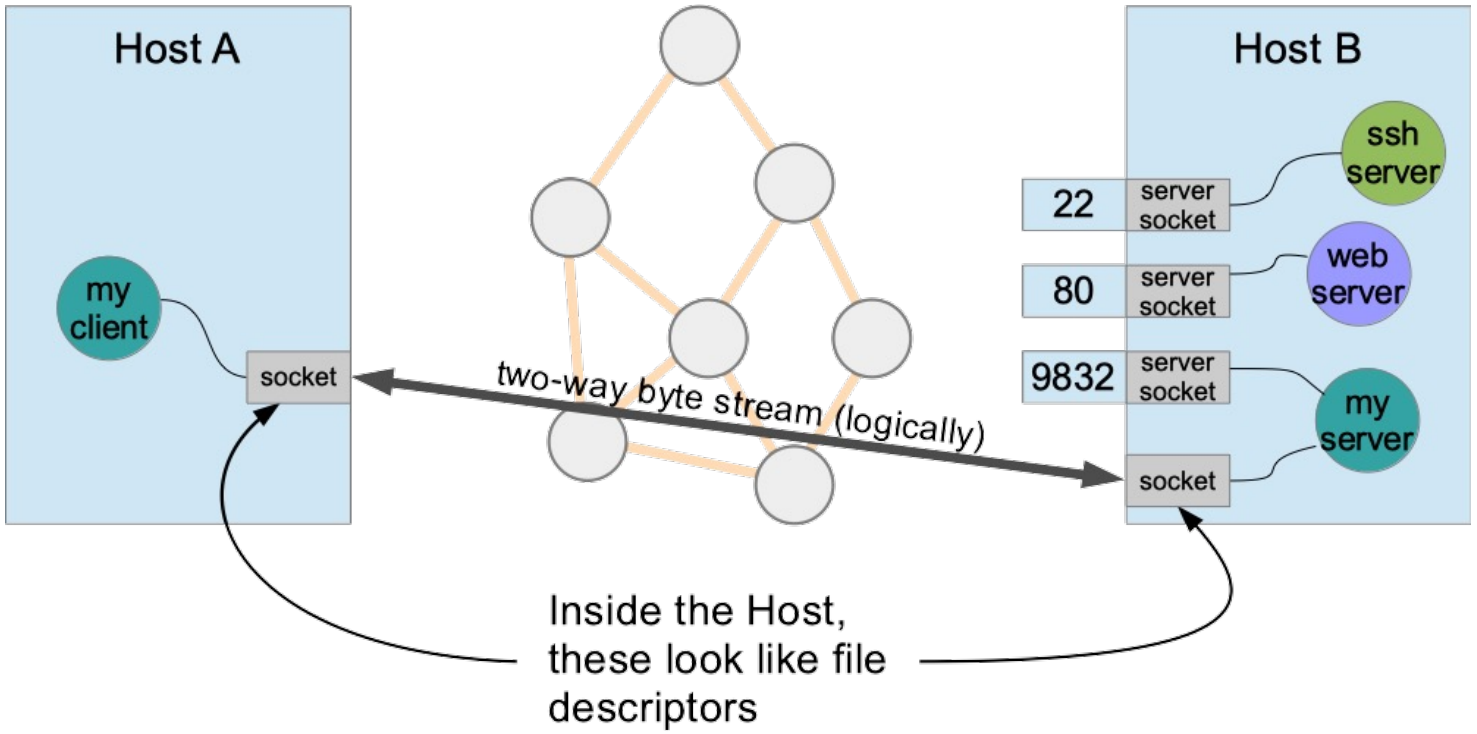
Using TCP

- After another response from the client, the server can *accept* the connection.



Using TCP

- Now, the client and server each have sockets
  (their endpoint in a two-way communication channel)



The Sockets API

- A collection of system calls to create and use TCP connections (and other protocols).
  - socket()
    - Make a new socket and return it as a file descriptor
    - used to make the server-side socket for accepting new connections
    - and to make an endpoint in a communication channel
    - Lots of possible parameter options, to tell it exactly what to make

The Sockets API

- More sockets calls
  - bind()
    - On the server, associate this socket with an address
  - listen()
    - On the sever, tell the socket to start listening for incoming connections
  - accept()
    - On the server, wait for a new connection and return a new socket for it
  - connect()
    - On the client, after making a socket, connect it to a socket on a listening server

The Sockets API

- More sockets calls
  - send()/recv() or write()/read() or sendto()/recvfrom()
    - Send and receive data over the socket
  - close()
    - Free the socket when you're done
  - gethostbyname()
    - An older interface to map host name to IP addresses
  - getaddrinfo()
    - A newer technique for the same thing, returns a list of addresses

C Socket Example

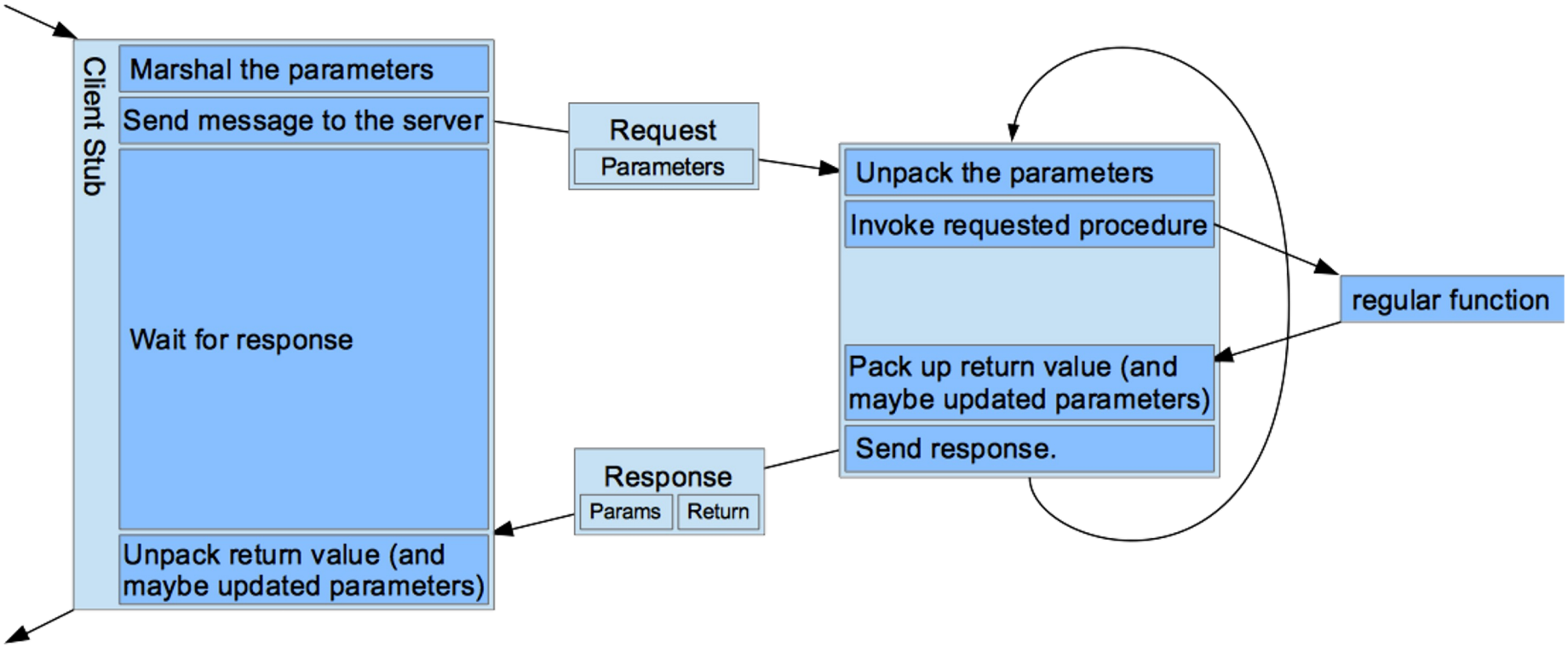
- A server, listens for one connection
- A client, connects to the server
- Then, push 1 MB from server to client
  - 64 KB at a time

Remote Procedure Call (RPC)

- We'd like to be able to distribute our applications across multiple hosts
  - Sockets provide a communication mechanism
  - But, for normal applications, this isn't how different parts of an application would normally communicate
- Alternatives, break application at procedure call boundaries
  - Caller and procedure reside on different hosts
  - Invisible to the programmer (well, that's the idea)
- Remote Procedure Call
  - A framework to simplify this type of application distribution
  - Compile-time tools to create:
    - A client-side stub function to initiate the RPC
    - Message formats for passing parameters & return values

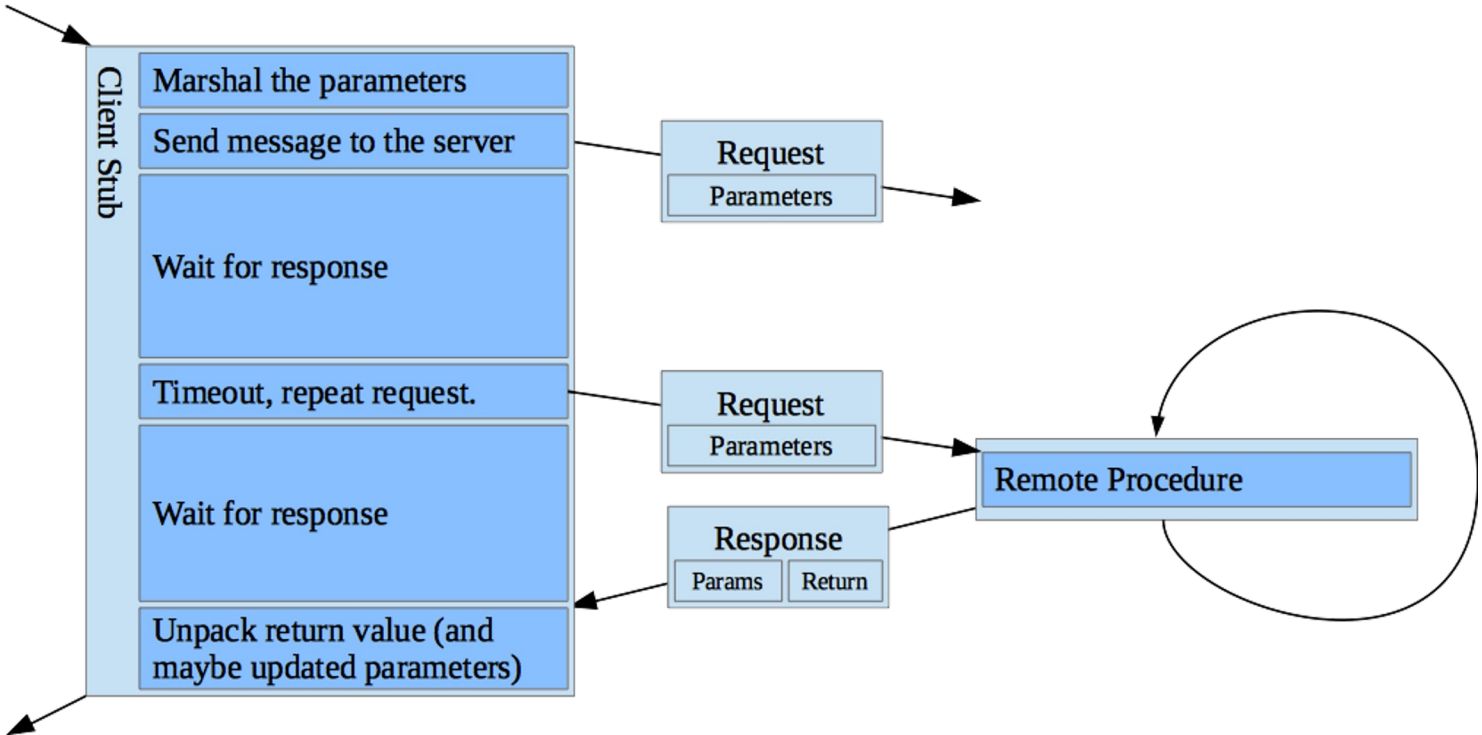
Remote Procedure Call

- Compile time tools to create:



Coping with Loss

- RPC can be implemented over TCP or UDP
  - Less overhead with UDP, but messages can get lost.
  - But, we can retransmit the request, without any additional application complexity.



Lost Messages and Semantics

- Now we have a real problem, we can't tell if the request or the reply got lost
- So, a remote procedure call may have different semantics than a local one.
  - *Exactly once semantics*
  - *At-least-once semantics*
    - We're going to want our procedures to be *idempotent*
  - *At-most-once semantics*

Client/Server Computing

- Client machine
  - Typically a single-user workstation
  - Providing an interface for the end user
- Each server provides
  - Shared, remote services for multiple clients
- Server permits many client to:
  - Share access to the same resource (e.g., database)
  - Use special-purpose computing resources (e.g., high-performance computing)

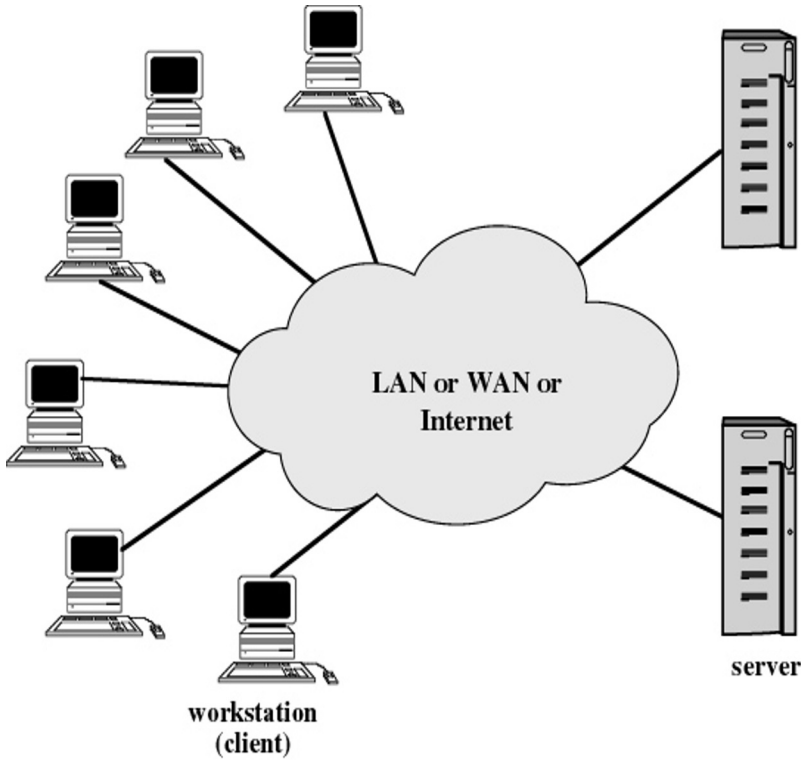
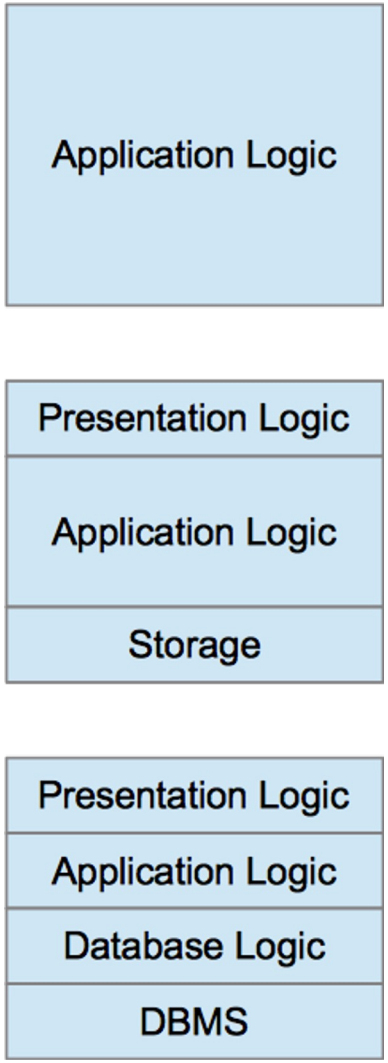


Figure 13.1 Generic Client/Server Environment

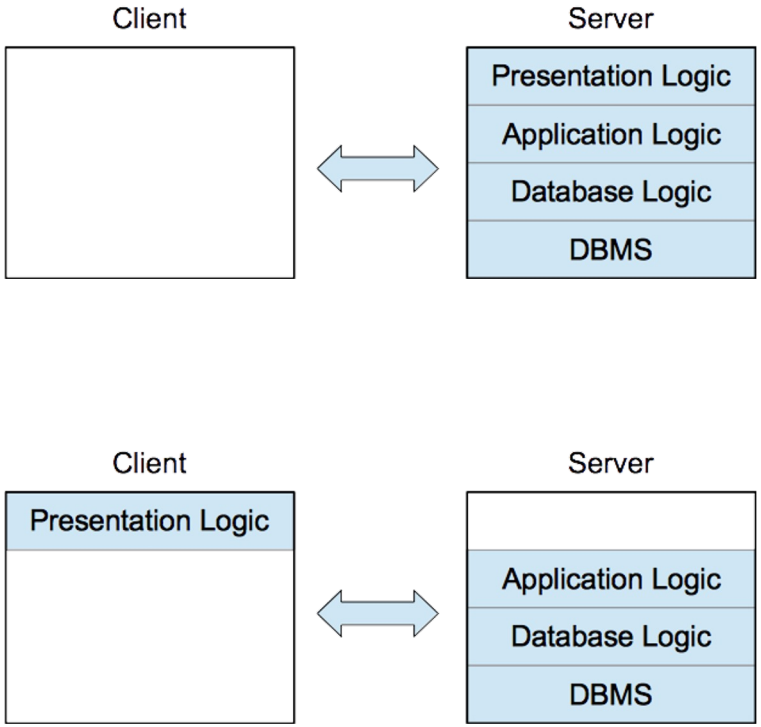
Client/Server Computing

- We're going to divide application responsibilities between two hosts.
- How?
- Well, what does an application look like?
- Normally, it has
  - Application logic
  - Some kind of presentation logic (if it has users)
  - Some kind of storage, commonly provided via a database.



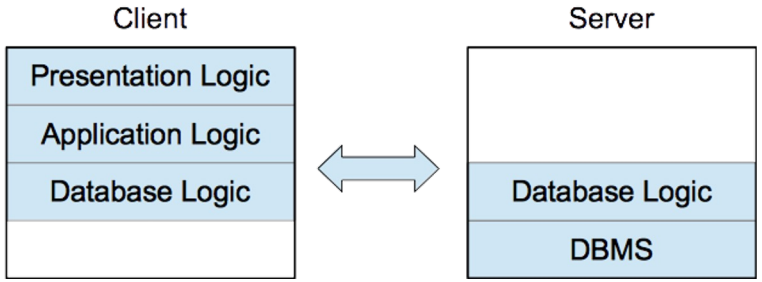
Sharing the Responsibility

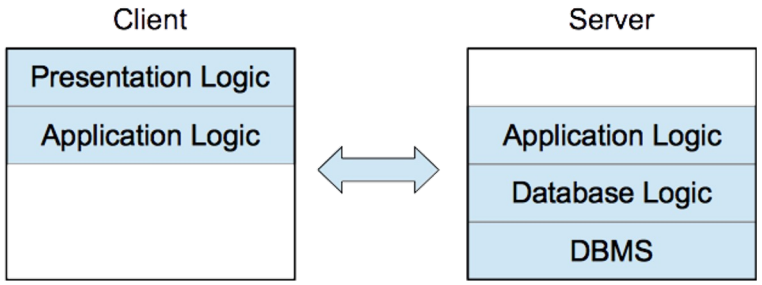
- *Host-based processing*
  - Server does (almost) all the work
  - E.g., a remote login
  - Not really client-server
- *Server-based processing*
  - Server does all application processing
  - Client just does user interface (e.g., a GUI)



Sharing the Responsibility

- *Client-based processing*
  - Client does all application processing
  - Server just does validation and execution of requests
- *Cooperative processing*
  - Split application logic between client and server
  - Shared computational load, potential for reduced communication overhead ☺
  - More complex to design and implement ☹





Distributed Application Models

- So far, these are all models with *Centralized Control*
  - A single application coordinates everything
  - But, this might inhibit scalability, robustness, availability
- We could fix some of this with *Decentralized control*
  - Responsibilities shared among a collection of servers
  - More challenging to design, but there are some common models
- *Peer-to-peer*
  - Clients themselves share in control responsibilities
- *Transactions*
  - Model application logic as a collection of possible transactions
  - Distribute transaction operations across multiple hosts
    Use a standard framework to guarantee execution order correctness

What We've Learned

- OS services in a networked environment
- Advantages of distributed applications
- Building blocks
  - Layered network protocols (e.g., IP, TCP, UDP)
  - TCP example
  - RCP as a simplified model for distributed applications
  - Models of distributed applications (client server, peer-to-peer, distributed transactions)
- Network Operating System vs a Distributed Operating System