THE UNIVERSITY OF CHICAGO

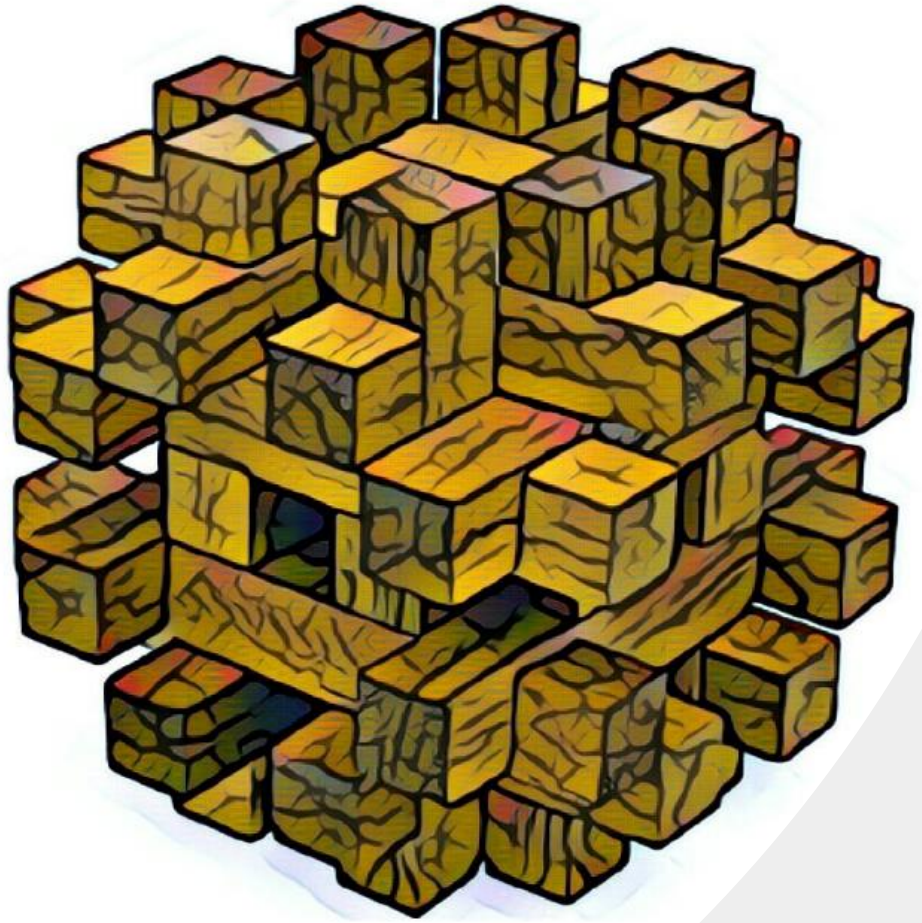
Class 3: Communication
MPCS: 52040
Distributed Systems

Kyle Chard

chard@uchicago.edu

Course schedule (* things might change a little)

Date	Lecture discussion topic	Class	Assessment
January 10	1. Introduction to Distributed Systems	Intro/Logistics	
January 17	2 Distributed architectures 3 Processes and virtualization	1 Docker	Homework 1 due
January 24	4 Networks and Communication	2 RPC/ZMQ/MPI	Homework 2 due
January 31	5 Naming	3 DNS/LDAP	Homework 3 due
February 7	6 Coordination and Synchronization		**Mid term exam**
February 14	7 Fault tolerance and consensus	Raft Project description	Homework 4 due (Project released)
February 21	8 Consistency and replication	4 FaaS	
February 28	9 Distributed data	5 Distributed data	
March 7	10 Data-intensive computing	6 Streaming	**Project due (March 9)** **Final exam (March 14)**



Textbook Chapter 4

Agenda

- Part 1: Shallow dive into networking
- Part 2: Communication techniques: RPC
- Part 3: Communication techniques: Message passing
- Part 4: Communication techniques: Multicast

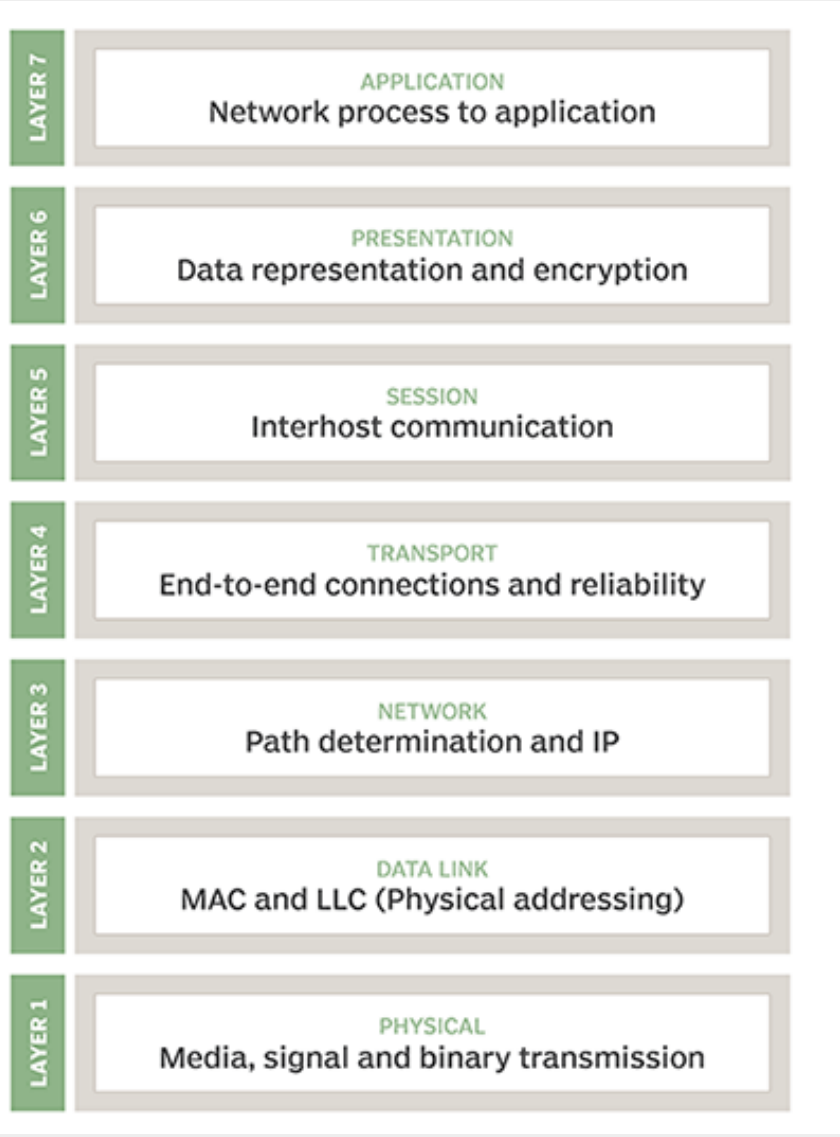
What is a network?

- A computer network links nodes and enables electronic communication between nodes
- Network may be built on physical or wireless links
  - E.g., twisted pair (e.g., Cat6), fiber-optic, WiFi, cellular,..
- A collection of protocols make it possible for nodes to send messages to one another
  - E.g., routing, reliability, security, performance

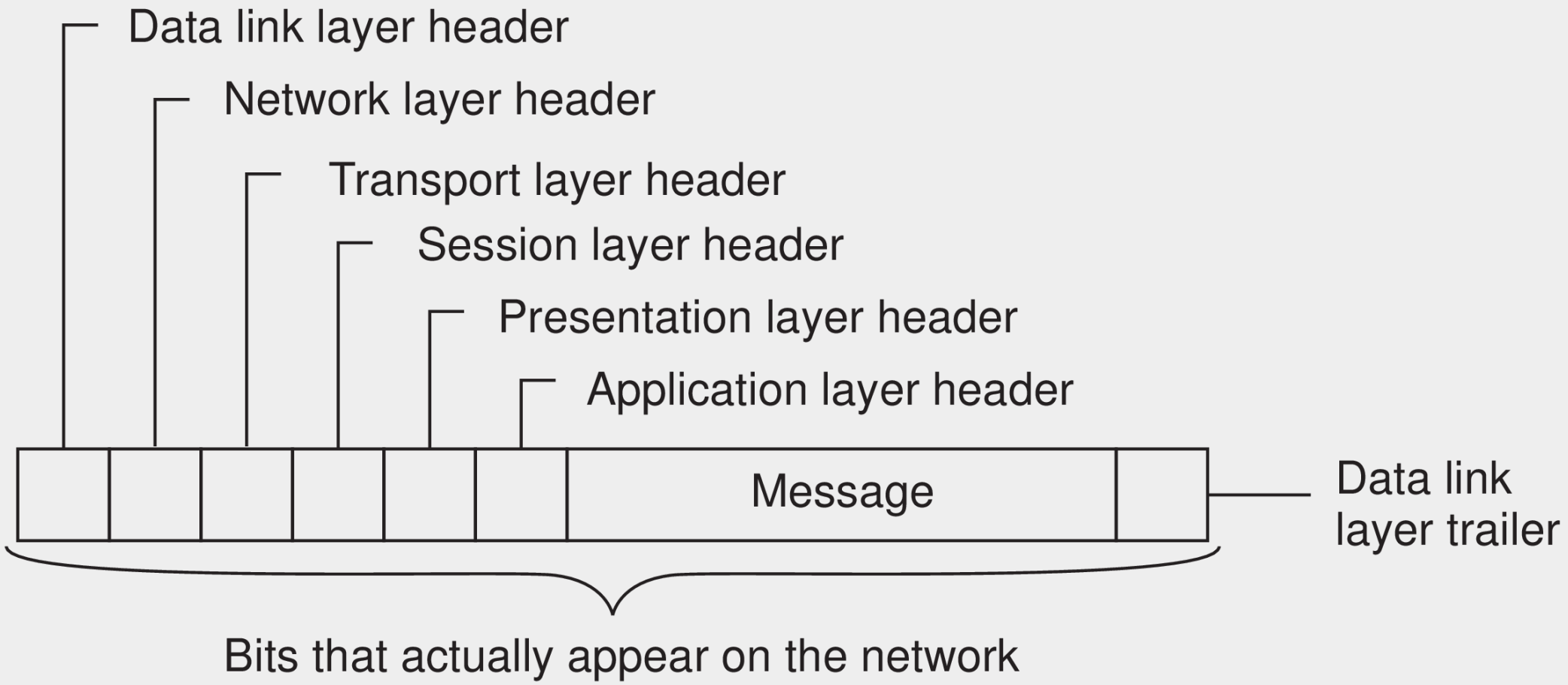
Part 1: Networking

- Network protocols are developed in layers, each packet goes inside the lower layer
- OSI (Open Systems Interconnection) model is a conceptual model with 7 layers

- **Network layer** protocols for routing a message through a computer network, as well as protocols for handling congestion (e.g. IP)
- **Transport layer** protocols for directly supporting applications, such as those that establish reliable communication, or support real-time streaming of data (e.g.,TCP)



Assembling messages in the OSI model

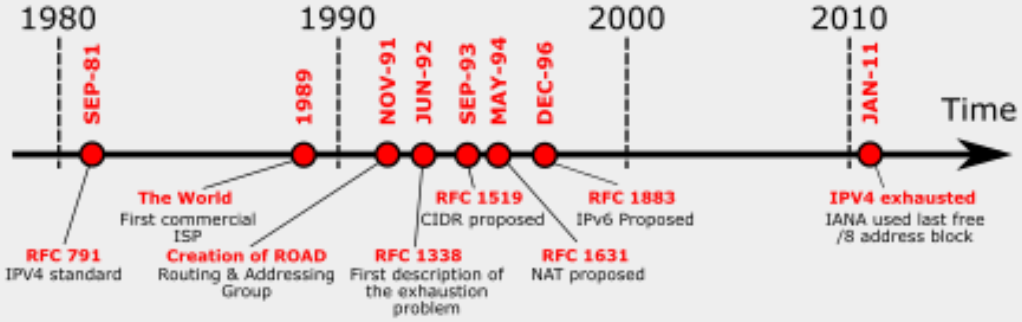


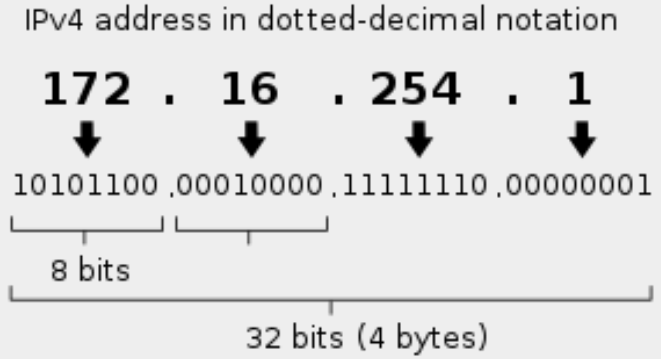
Network layer: Internet Protocol

- Aim to deliver (route) packets from source to destination based on the IP address in the header
- Encapsulates data into *datagrams* (supports fragmentation and reassembly)
- Responsible for routing of packets through a network
- Best-effort delivery (i.e., unreliable)
  - Connectionless protocol (doesn't establish a connection between source/destination)
  - Data corruption, packet loss, duplication possible
  - Routing is dynamic, each packet may take a different path and may arrive out of order
  - Checksum to indicate that packet header is not corrupted

IPv4 and IPv6

- IPv4 uses 32-bit addresses == 4B addresses
  - 192.0.2.235
    - Four 8-bit octets
  - Last top-level address allocated in 2011
    - A records have 16M addresses
  - 18 million addresses reserved for private networks
    - 192.168… 172.. 10..

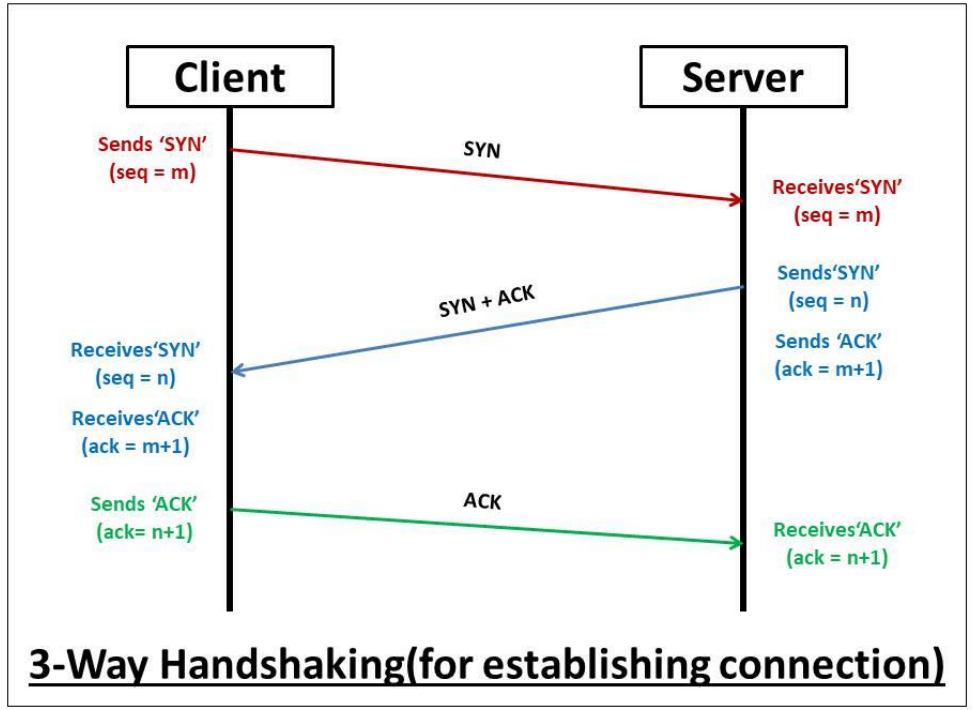




- IPv6 uses 128-bit addresses
  - Not interoperable with v4 (but both coexist together)
  - Addresses are represented as eight groups of four hex digits (128 bits)
    - 2001:0db8:0000:0042:0000:8a2e:0370:7334
  - New packet format (to minimize header processing by router)
  - Many other advantages: IPSec, multicasting, stateless address autoconfig

Transport layer: Transmission Control Protocol (TCP)

- TCP is a reliable stream delivery service which guarantees that all bytes received will be identical and in the same order as those sent
  - Reliable, ordered, and error-checked
- TCP is a connection-oriented protocol
- TCP accepts data from a data stream, divides it into chunks, and adds a TCP header creating a TCP segment
- Advanced features like flow control, congestion control, timeout-based retransmission
- TCP segment placed in IP datagram



Transport layer: User Datagram Protocol (UDP)

- Connectionless communication model with minimal protocol mechanisms
- Provides integrity verification (checksums) of header and payload
- Provides no other guarantees to upper layer protocol
- Features:
  - Transaction-oriented: suitable for simple query-response protocols (e.g., DNS)
  - Provides datagrams: suitable for modeling other protocols (e.g., IP tunneling, RPC)
  - Simple and easy to build upon without full protocol stack (e.g., DHCP)
  - Stateless: suitable for very large numbers of clients (e.g., streaming)
  - The lack of retransmission delays makes it suitable for real-time (e.g., VOIP)
  - Multicast makes it useful for broadcasting information (e.g., service discovery)

Fastest way to move data?

ESGF data replicatior

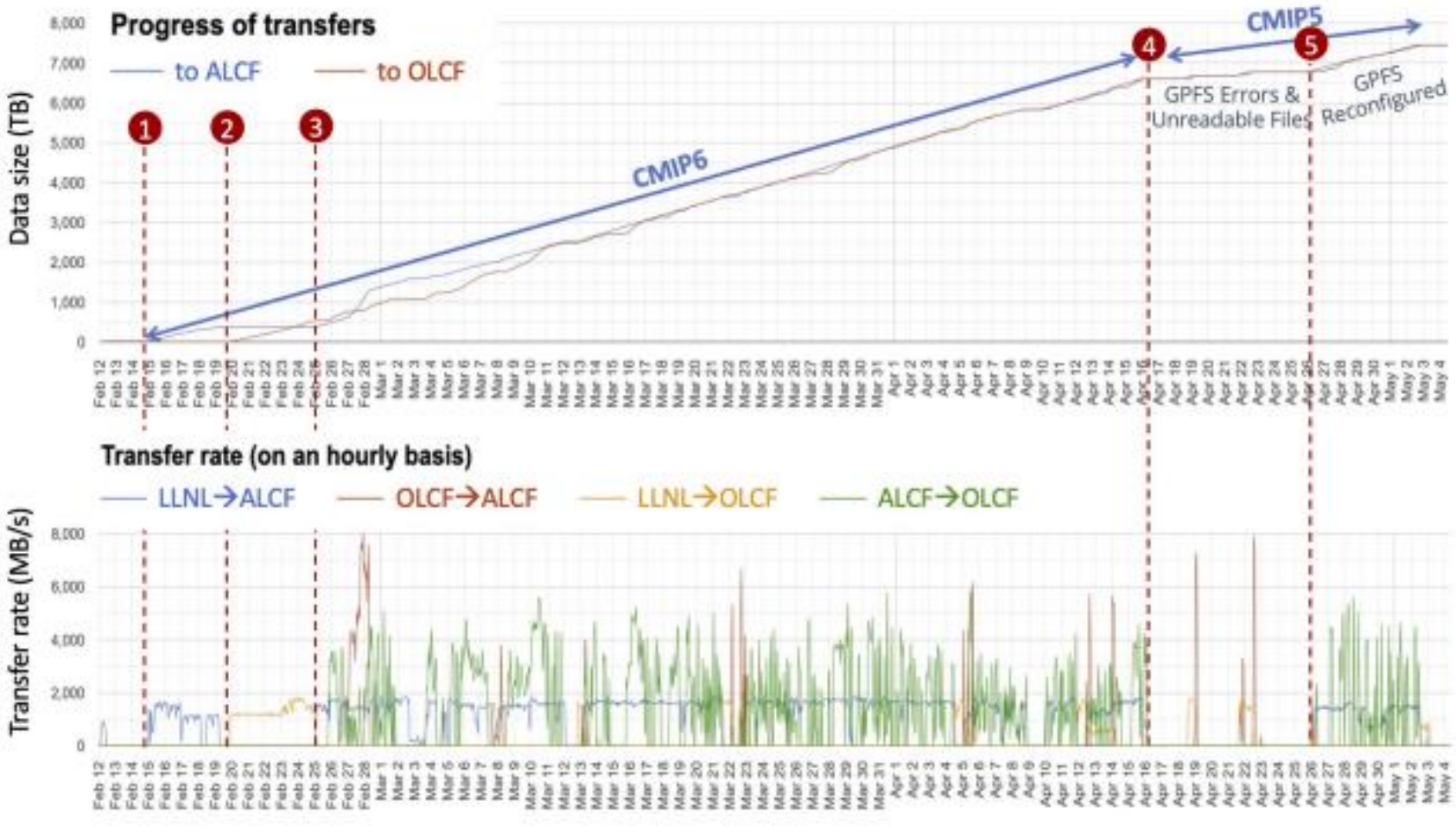
7PB of data fr
LLNL to ANL/(

Lawren
Livermo
Nationa
Laborat

Data transferred to ALCF

Data transferred to ORNL

4-6 GB/s



3	/css03_data/CMIP6/CMIP/MIROC/MIROC-ES2L/esm-hist	ALCF	2022-03-10 14:51:04	2022-03-10 15:13:24	SUCCEEDED	3706	39663	2973432261868	0	2.22 GB/s
4	/css03_data/CMIP6/CMIP/MIROC/MIROC-ES2L/amip	ALCF	2022-03-10 14:47:03	2022-03-10 14:50:22	SUCCEEDED	3126	12284	446324011629	0	2.25 GB/s

Rate
2 MB/s
9 MB/s
0 MB/s
4 MB/s

Rate
93 GB/s
08 GB/s

globus

https://dashboard.globus.org/esgf

Lacinski et al. Automated, Reliable, and Efficient Continental-Scale Replication of 7.3 Petabytes of Climate Simulation Data: A Case Study. https://arxiv.org/abs/2404.19717

March 10, 2022

Fastest way to move data?

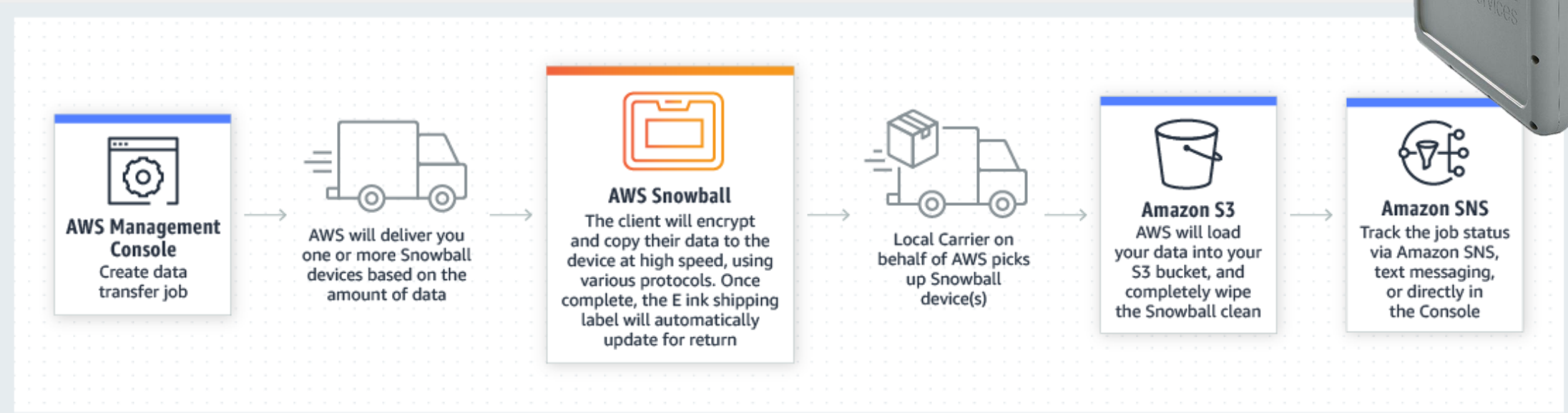
Sneakernet

- Transfer of data by physically moving media (e.g., disks, USB drives, tapes)

"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway."

- Andrew Tanenbaum. Computer Networks. 1989.

Amazon snowball



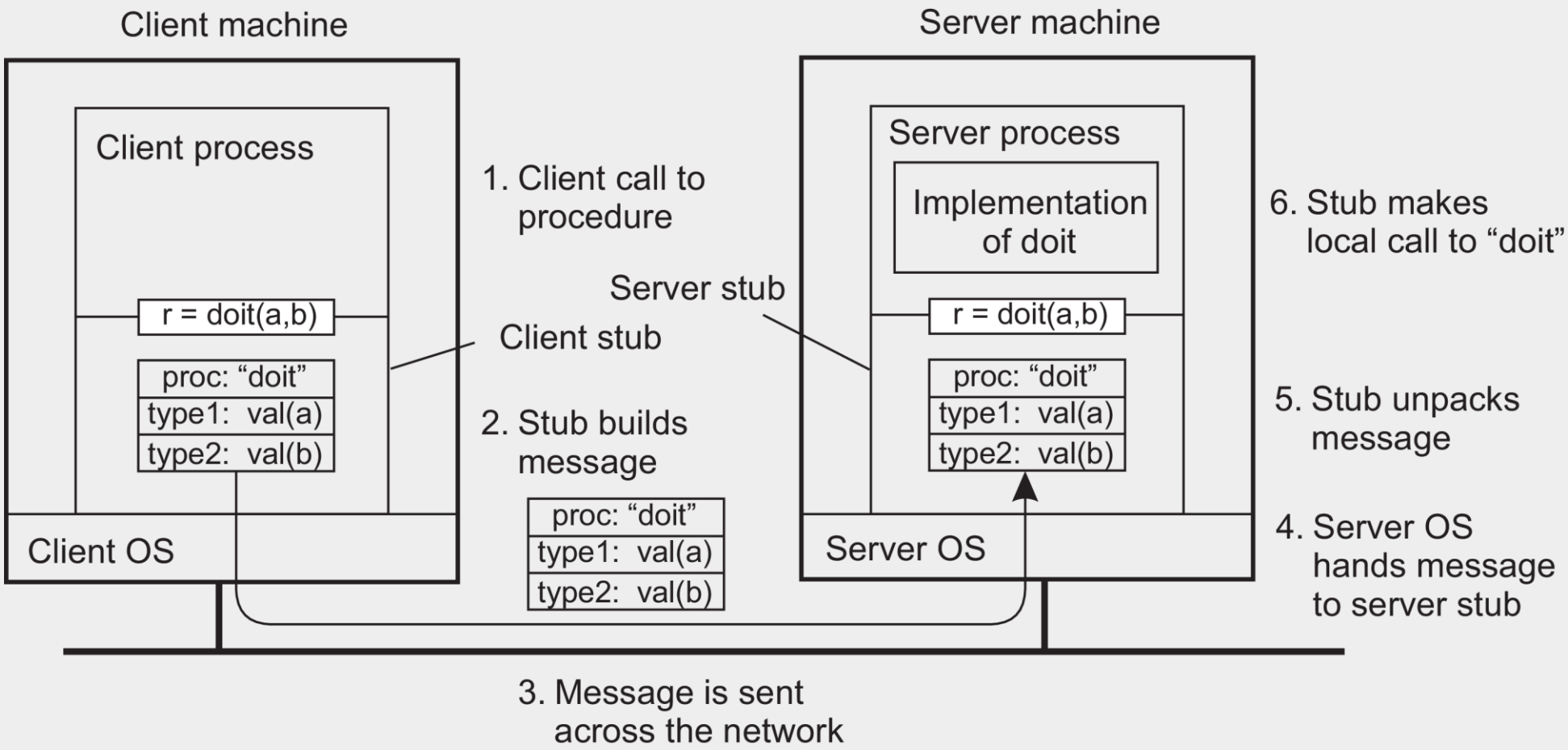
Snowball is a petabyte-scale data transport solution that uses devices designed to be secure to transfer large amounts of data into and out of the AWS Cloud. Using Snowball addresses common challenges with large-scale data transfers including high network costs, long transfer times, and security concerns.

Part 2: Communication Techniques: RPC

Part 2: RPC

- Messaging doesn’t provide a great deal of transparency to most programs
- RPC aims to provide a familiar procedure call model by making calls look like they’re happening locally
- One of the biggest challenges is marshaling parameters such that the two sides can understand the contents of the procedure call

Basic RPC model



Parameter passing

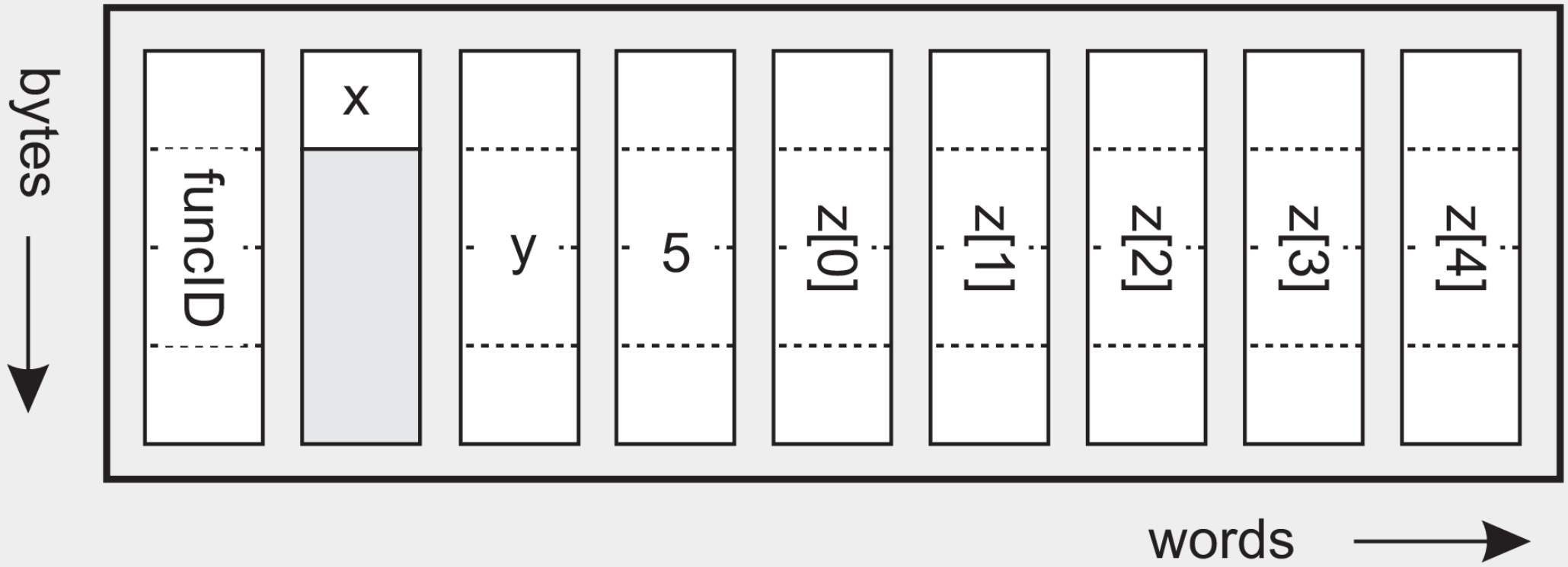
- Client stub takes parameters, packs into a message (*marshal*), and sends to server stub
- Challenges
  - Server receives a stream of bytes from the client (no information about meaning)
  - Client/server might have different data representations (e.g., byte order)
  - Client and server must agree on how they *encode* data:
    - How are basic data values represented (integers, floats, characters)?
    - How are complex data values represented (arrays, objects)?
- Conclusion
  - Client/server need to properly interpret messages, transforming them into machine-and network-independent representations

How do we deal with pointers?

- Forbid them ☺
  - Often not really feasible
- Often references to fixed size data types (e.g., static arrays) or dynamic data types where we can compute size at runtime (e.g., strings)
  - In these cases we can copy the referenced data structure to a “flat” param
- More complicated types (e.g., user-defined classes) are more difficult
  - Ideally allow the language system (e.g., Python) to automate marshaling/unmarshaling of those data
  - Complex, big, nested objects might make this impossible or impractical

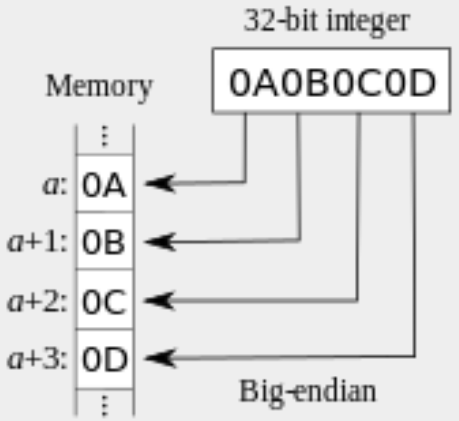
RPC Process: 1) agree on message format

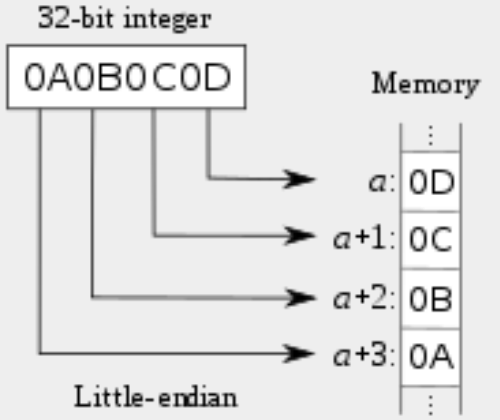
```
void someFunction(char x; float y; int z[5])
```



RPC Process: 2) agree on representation

- Machines have different representations:
  - Integers are represented in two's complement
  - Characters in 16-bit Unicode
  - Floats in IEEE 754
  - With everything stored in little endian





RPC Process: 3) agree on communication

- Finally, we need to agree on the underlying protocols to be used
  - TCP/IP
  - UDP + application specific reliability model

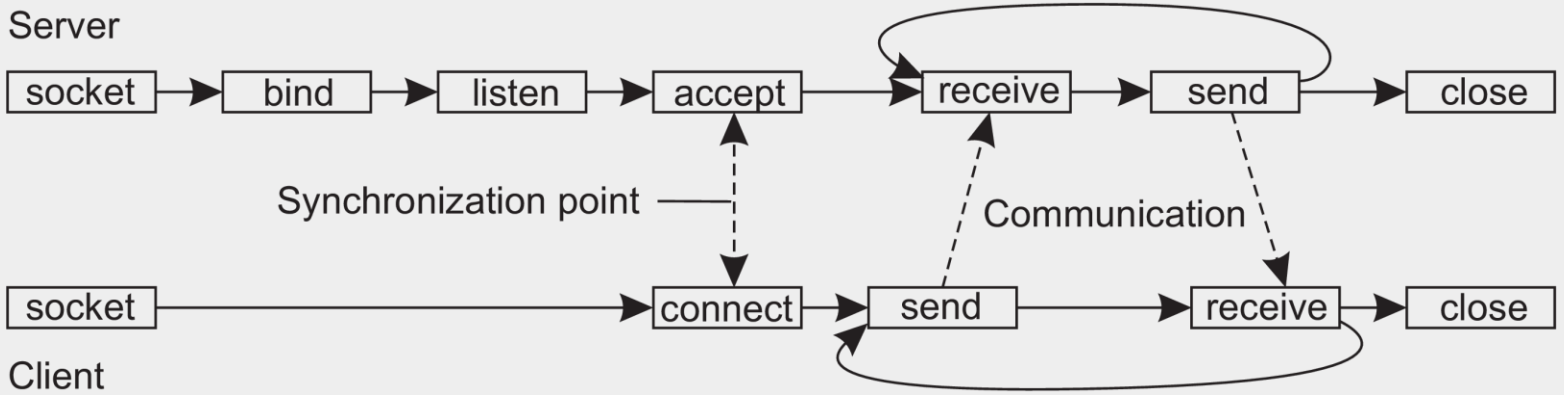
Part 3: Communication Techniques: Message Passing

Agenda

- Part 1: Shallow dive into networking
- Part 2: Communication techniques: RPC
- Part 3: Communication techniques: Message passing
  - Sockets
  - ZeroMQ
  - MPI
  - Message queues
- Part 4: Communication techniques: Multicast

Sockets

- Transport layer provides a good base for messaging
- Socket
  - Communication endpoint that an application can write/read data that will be sent/received from the network
  - Simple abstraction over a network port for a specific transport protocol
  - TCP or UDP



ZeroMQ

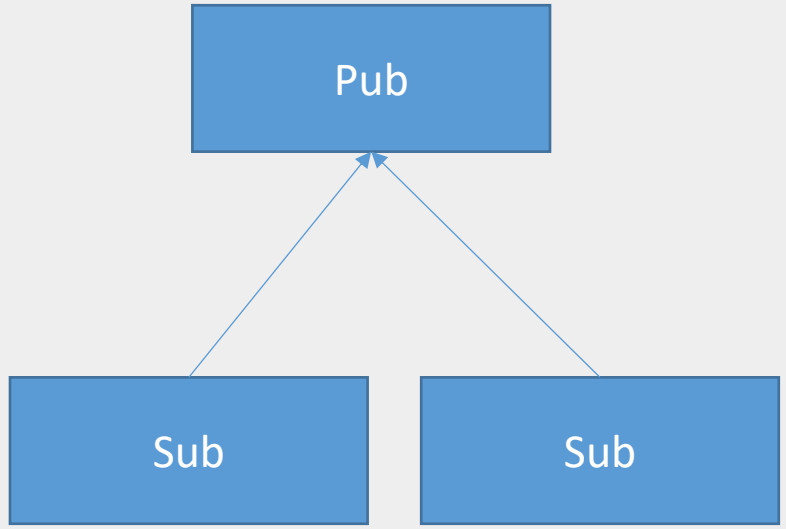
- Abstracts low level socket implementations
  - The same interface irrespective of the underlying OS
- Provides a higher level model by pairing sockets:
  - one for sending messages at process P
  - a corresponding one at process Q for receiving messages
- TCP-based, all communication is connection-oriented
- Allows many-one communication with sockets (server can listen to multiple ports)
- All communication is asynchronous

Example: ZeroMQ Request-Reply

- Request-reply pattern
  - Traditional client-server communication
  - A client application uses a **request socket** (of type REQ) to send a request message to a server and a response
  - The server uses a **reply socket** (of type REP)
- Advantages
  - Simplifies development by avoiding need to call listen (or accept)
  - When a server receives a message, a subsequent call to send is automatically targeted toward the original sender.
  - When a client calls the recv operation (for receiving a message) after having sent a message, ZeroMQ assumes the client is waiting for a response from the original recipient

Example: ZeroMQ Pub-Sub

- Publish-subscribe
  - Client subscribes to messages that are published by server
    - Server defines topics that client can filter on
  - Supports multicast from server to many clients
  - Server uses PUB socket, client uses SUB socket

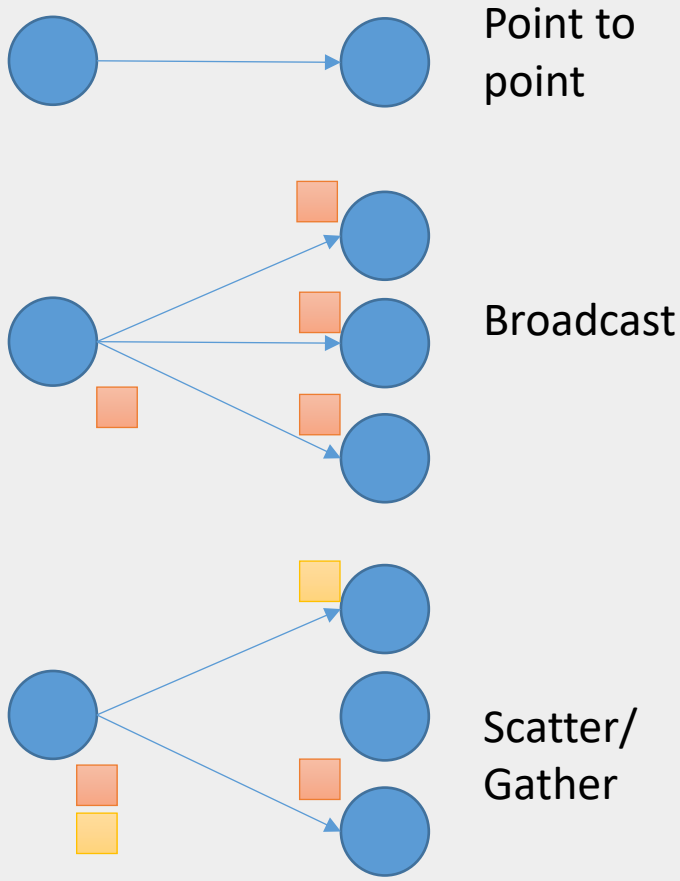


Message Passing Interface (MPI)

- Supercomputers comprised of thousands of nodes with high speed interconnects
  - Each node is a separate machine (but low latency to neighbors)
  - How can we effectively communicate amongst these machines
- Sockets?
  - Level of abstraction is low (read/write), ideally higher level primitives needed for applications
  - Not optimized for high speed networks
  - Opening sockets for thousands of nodes is often not allowed
  - Sharing sockets is complicated
- HPC machines shipped with their own communication libraries
  - Need a standard way to communicate == MPI (but really need a way to build tightly coupled applications)

Message Passing Interface

- MPI
  - Message passing on parallel architectures
  - High scalable for building large-scale distributed applications
  - Optimized for cluster interconnects
  - Little fault tolerance
  - Requires specialized MPI executors



Message-Oriented Middleware (MOM)

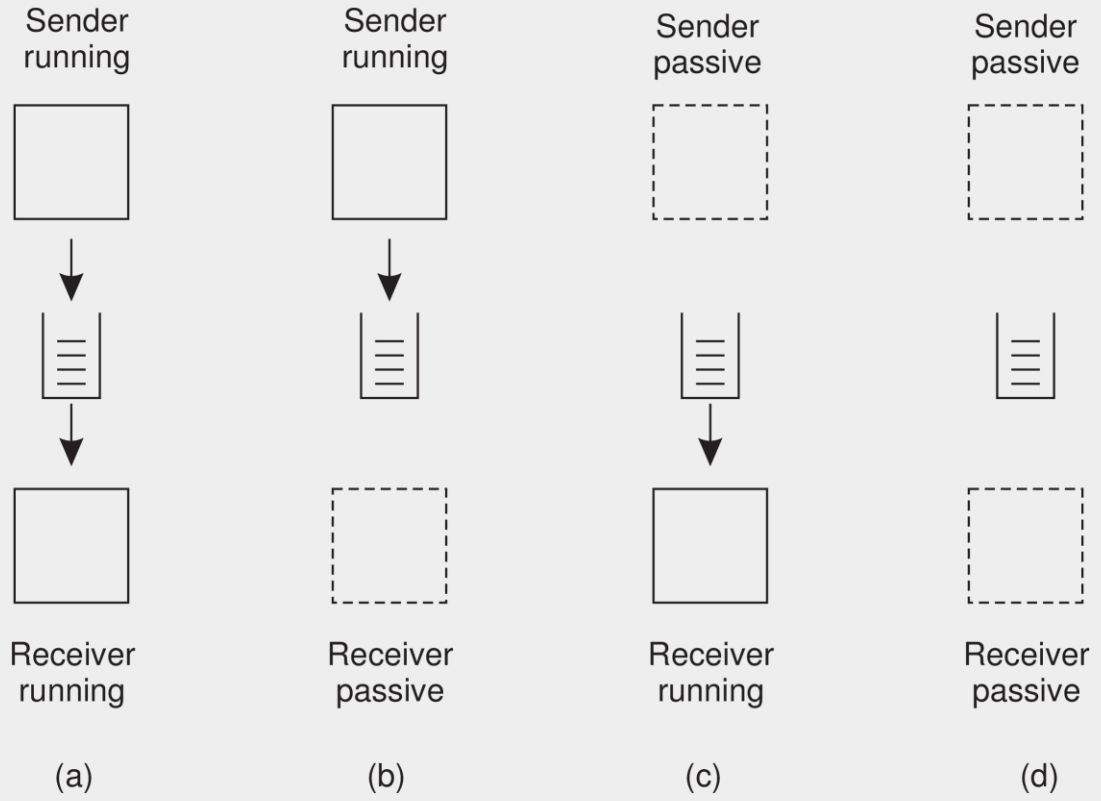
Unlike previous point-to-point communication, message-oriented middleware systems support asynchronous communication using intermediate storage

- Sender/receiver need not be online at the same time
- Message delivery is therefore targeted at larger latency (minutes/hours rather than milliseconds)

- Asynchronous persistent communication is provided through middleware-level queues
  - Queues correspond to buffers at communication servers

General model

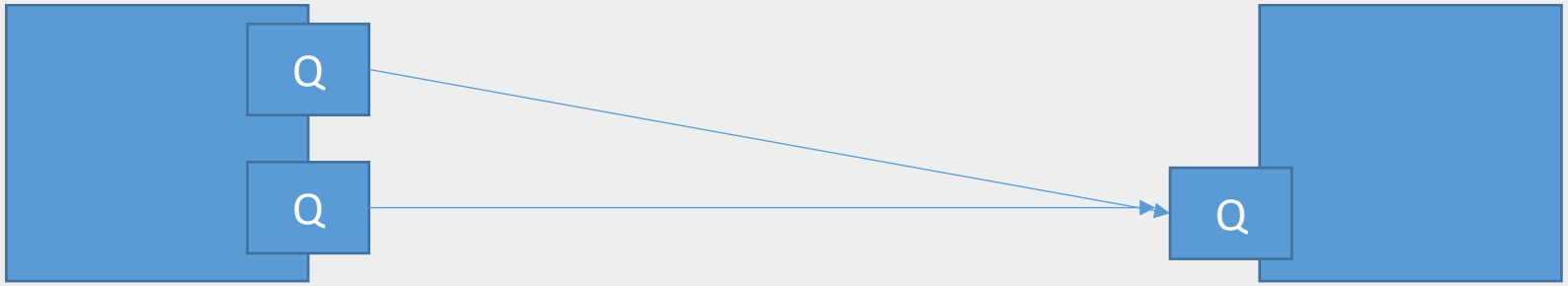
- Applications communicate by inserting messages in specific queues
- Messages forwarded over a series of connected servers
  - In practice, src-dst are normally connected and the queue is on the dst
- Generally, sender guaranteed that the message will eventually be inserted in destination's queue
  - Nothing about when/if it will be read
- Message stays in queue until removed



General architecture

- Queues are managed by **queue managers**
  - Separate process or implemented as a library linked with the application

- Application can only put in local queue and get from a local queue
  - => Messages must contain destination address
  - => Queue managers are responsible for routing messages



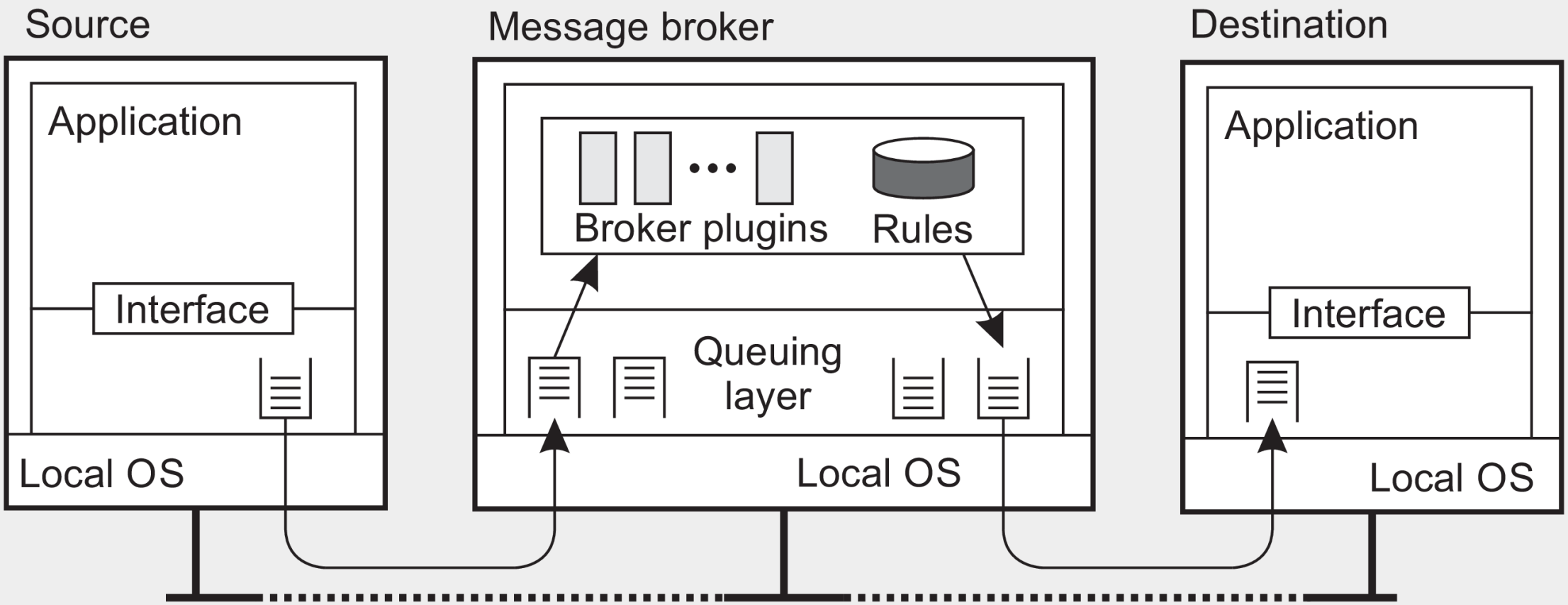
Challenges

- How do we address a queue?
  - Ideally use logical, location-independent names
  - Need a mapping from logical name to physical address (host/port)
- How do we relay the name/address mapping to the queue manager?
  - Copy the mapping table to all queue managers?
- Does every queue manager need to know all the address of all other queue managers?
  - Scalability and synchronization issues as size increases,
  - Routing models allow queue managers to know about their neighbor and they can forward

Challenges with MOM

- Message queues are often used in situations where the applications are different and thus it serves as a common layer for interoperability
- Requirement of messaging is that all participants speak the same language (i.e., protocol and message format)
  - How do we integrate new and old applications?
- Why not just update each time we add a new application?
  - All other applications need to be updated to understand that message
  - With *N* applications we have *N* x *N* message protocol converters
- Why not just agree on a common message?
  - Only makes sense if the collection of processes that use the protocol are very similar
- Why not use a common representation of messages?
  - This is a good basis, but doesn't solve all problems above (e.g., vocabulary)
  - E.g., messages in self describing XML or JSON formats

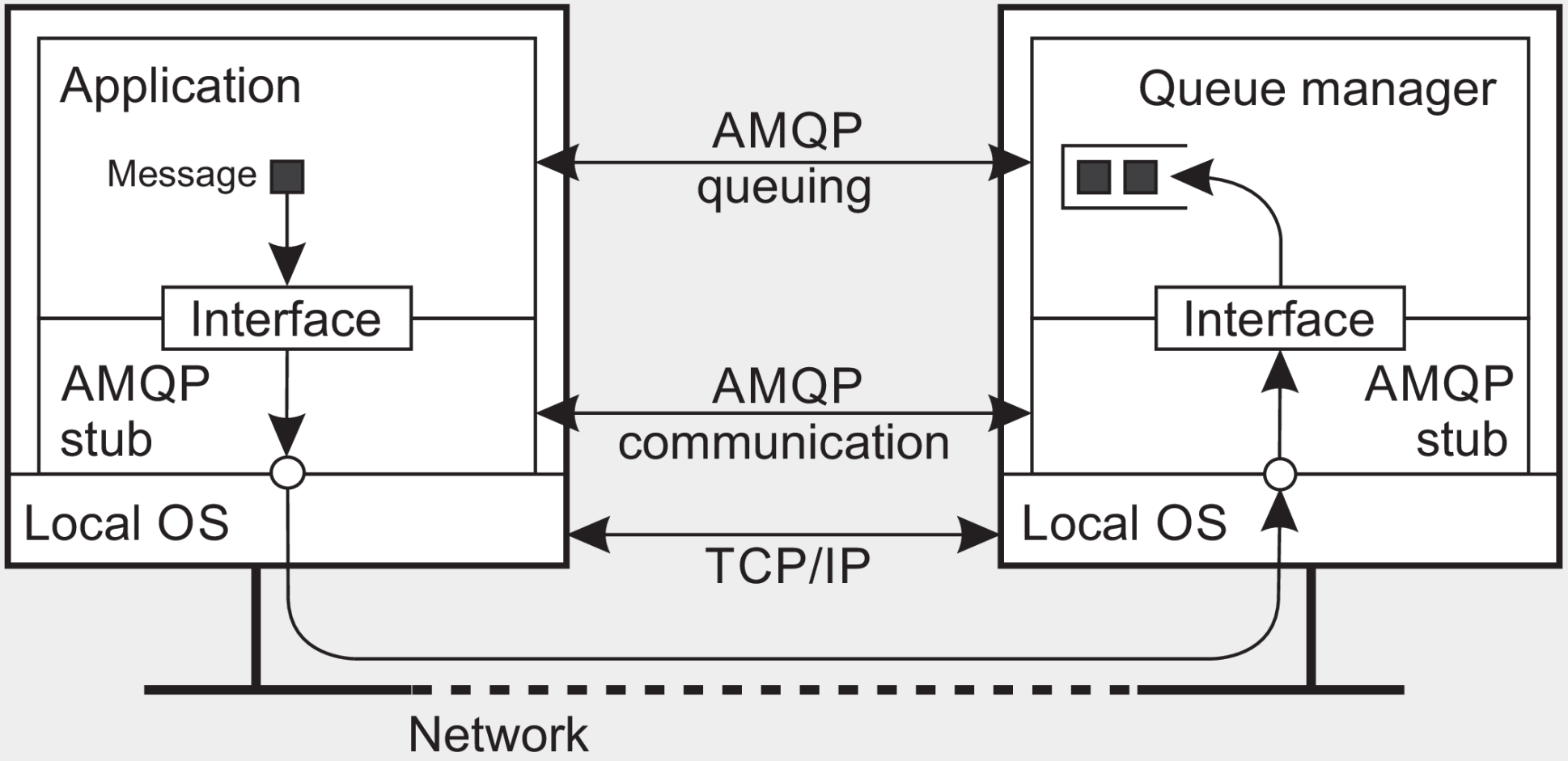
Solution: message broker



Example: Advanced Message Queuing Protocol (AMQP)

- Many different message queuing systems:
  - IBM MQ, Rabbit MQ, Active MQ, Qpid
- Given message queues are used for interoperation it is sub-optimal if you're locked into a message queue system
- AMQP (2006) is an effort to create a standard protocol

AMQP Model


Application
Message
Interface
AMQP
stub
Local OS
AMQP
queuing
AMQP
communication
TCP/IP
Queue manager
Interface
AMQP
stub
Local OS
Network

AMQP Communication model: Transferring a message

1. Sender:
    1. Message assigned unique ID and recorded locally in unsettled state
    2. Stub transfers the message to the server (receiver)
    3. AMQP stub also records it as being in an unsettled state
    4. Receiver server-side stub passes it to the queue manager
2. Receiver (queue manager):
    1. Handles the message and normally reports back to its stub that it is finished (it has actioned the message)
    2. The stub passes this information to the original sender
    3. Sender: message enters a settled state.
3. AMQP stub of the original sender now tells the sub of the original receiver that message transfer has been settled
    1. (meaning that the original sender will forget about the message as the receiver has processed)
    2. Receiver's stub can now also discard anything about the message, formally recording it as being settled as well

AMQP Messaging

- Layered on top of underlying communication protocols (e.g., TCP/IP)
- Messaging takes place between two nodes
  - Producer, consumer, or a queue (store/forward messages)
- Receiver can indicate to the sender whether its message was accepted or rejected (i.e., notification sent to the original sender)
- Persistence
  - Mark a message as durable: indicate that intermediate nodes (e.g., queue) can recover in case of failure (nodes that can't will reject the message)

Part 4: Communication techniques: Multicast

Agenda

- Part 1: Shallow dive into networking
- Part 2: Communication techniques: RPC
- Part 3: Communication techniques: Message passing
- Part 4: Communication techniques: Multicast

Multicast communication

- Many applications require sending data to multiple receivers
  - E.g., content distribution, P2P, location services, distributed analyses

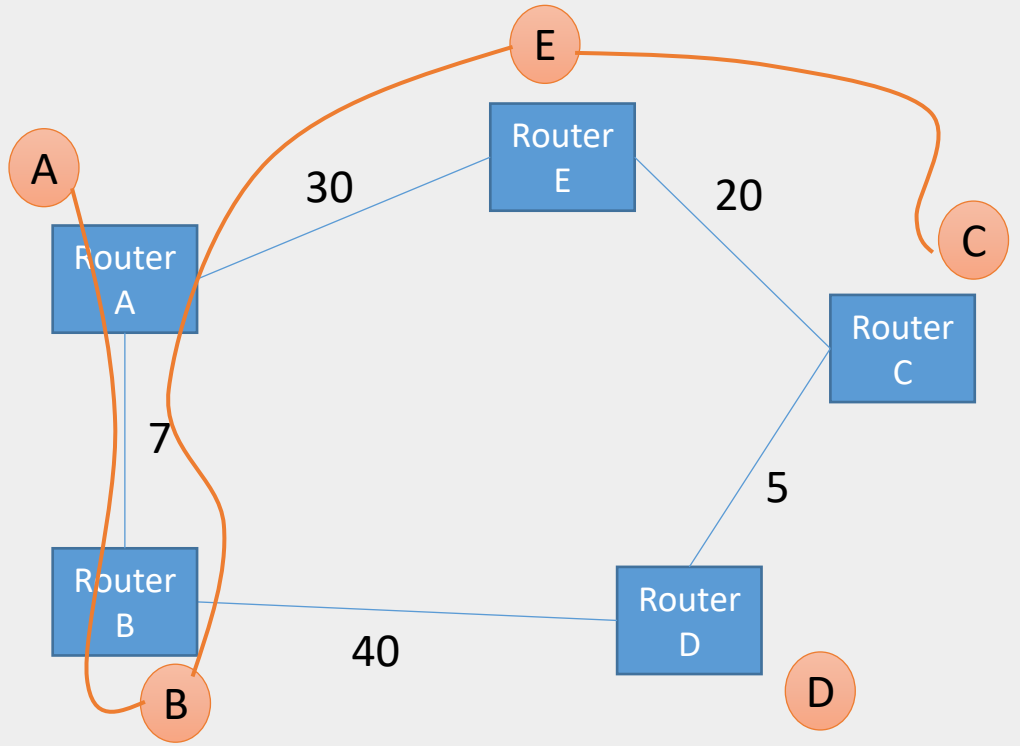
- Unicast:
  - Direct data transfer from **point to point**
- Broadcast:
  - Data transferred from point to **all** other points
- Multicast
  - Data sent from point to a **set** of other points

Application level multicast

- Basic idea is to organize nodes of a distributed system into an overlay network and use that network to disseminate data:
  - A tree, leading to unique paths between every pair of nodes
  - A mesh network, in which every node has multiple neighbors and there exists multiple paths between every pair of nodes
    - Provides better robustness but requires a form of routing

Performance issues with overlay networks

- Building an overlay network is not complicated…
  - building an efficient overlay network is a little harder
- E.g., overlay on the right with 5 nodes connected to various routers
- Multicast overlay might duplicate path comm. (e.g., Ra-Rb) while sending between nodes in the tree (e.g., we should not have an overlay link between B and E)



Metrics to compare performance

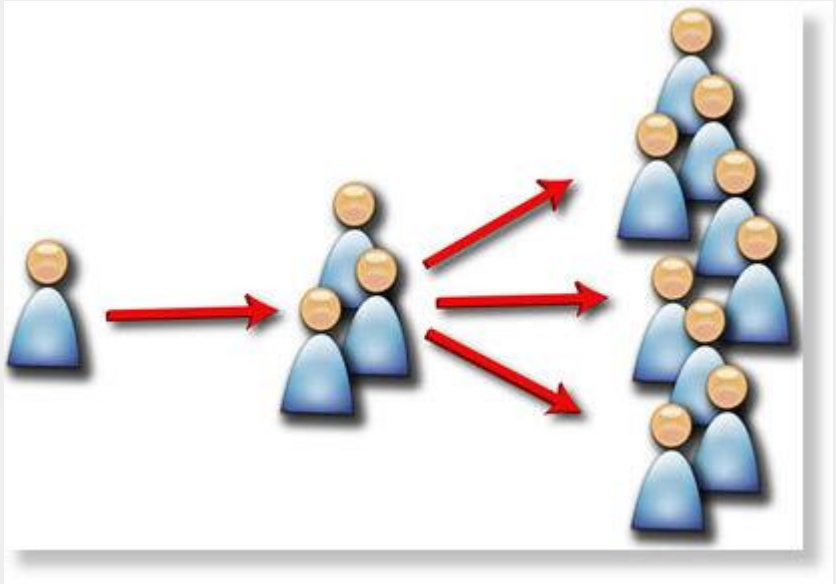
- Link stress
  - Counts how often a packet crosses the same link
- Relative Delay Penalty
  - Ratio in delay between two nodes in the overlay to the delay in the underlying network
  - Captures the inefficiency of the overlay network
- Tree cost
  - Global metric that aims to find a minimal spanning tree in which total time for dissemination to all nodes is minimal

Setting up overlays

- If we are really trying to multicast to a subset of nodes, then tree overlays will likely result in some nodes storing/forwarding messages they don't have any interest in
- One simple solution: construct an overlay for each multicast group and broadcast to that group
  - Nodes belonging in several groups would need to maintain separate lists for each overlay
- How do we now send messages? Naïve approach is flooding:
  - Simply forward a message to each of your neighbors (except the one from which it is received) and ignore duplicates
  - Large number of messages will be sent (most of which are not useful)

Gossip-based (or epidemic) dissemination

- Aim to spread information
  - Node is infected if it contains the data to be spread
  - Node that has not yet seen the data is susceptible
- General model:
  - Update operations are performed at a single server
  - A replica passes updated state to only a few neighbors
  - Update propagation is lazy (i.e., not immediate)
  - Eventually, each update should reach every replica



Exercise 2: Messaging

Completing this exercise

- Run locally on your computer or on a remote computer, or..
- Docker
  - Dockerfile

    FROM python:latest

    RUN apt-get update -y

    RUN pip install rpyc pyzmq
  - Build
    - docker build -t communication-class .
  - Run first terminal
    - docker run -it -v <local>:<container> communication-class bash
      - docker run -it -v "$(pwd)"/code:/code communication-class bash
  - Connect to container and run second terminal
    - docker exec -it <CONTAINER ID> bash

Exercise 2a: Python RPC

- Python RPC module: rpyc
  - https://rpyc.readthedocs.io/en/latest/docs/howto.html

- If you aren't using docker..
  - pip install rpyc

- We will build a simple server (wrapping a list) and a client to add to the list and print the contents of the list

Make a class to represent an object

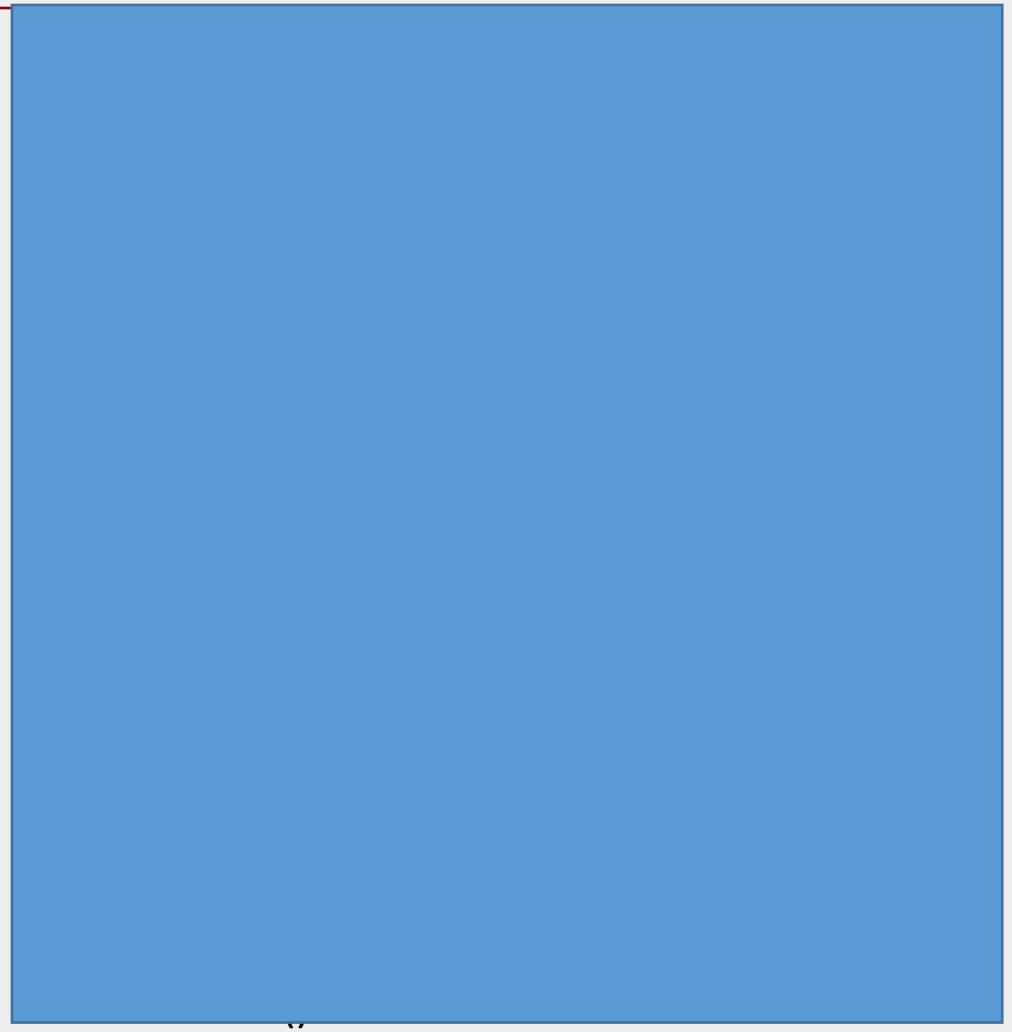
```
class RPC_List():
    values = []

    def append(self, data):
        self.values.append(data)
        return self.values

    def values(self):
        return self.values
```

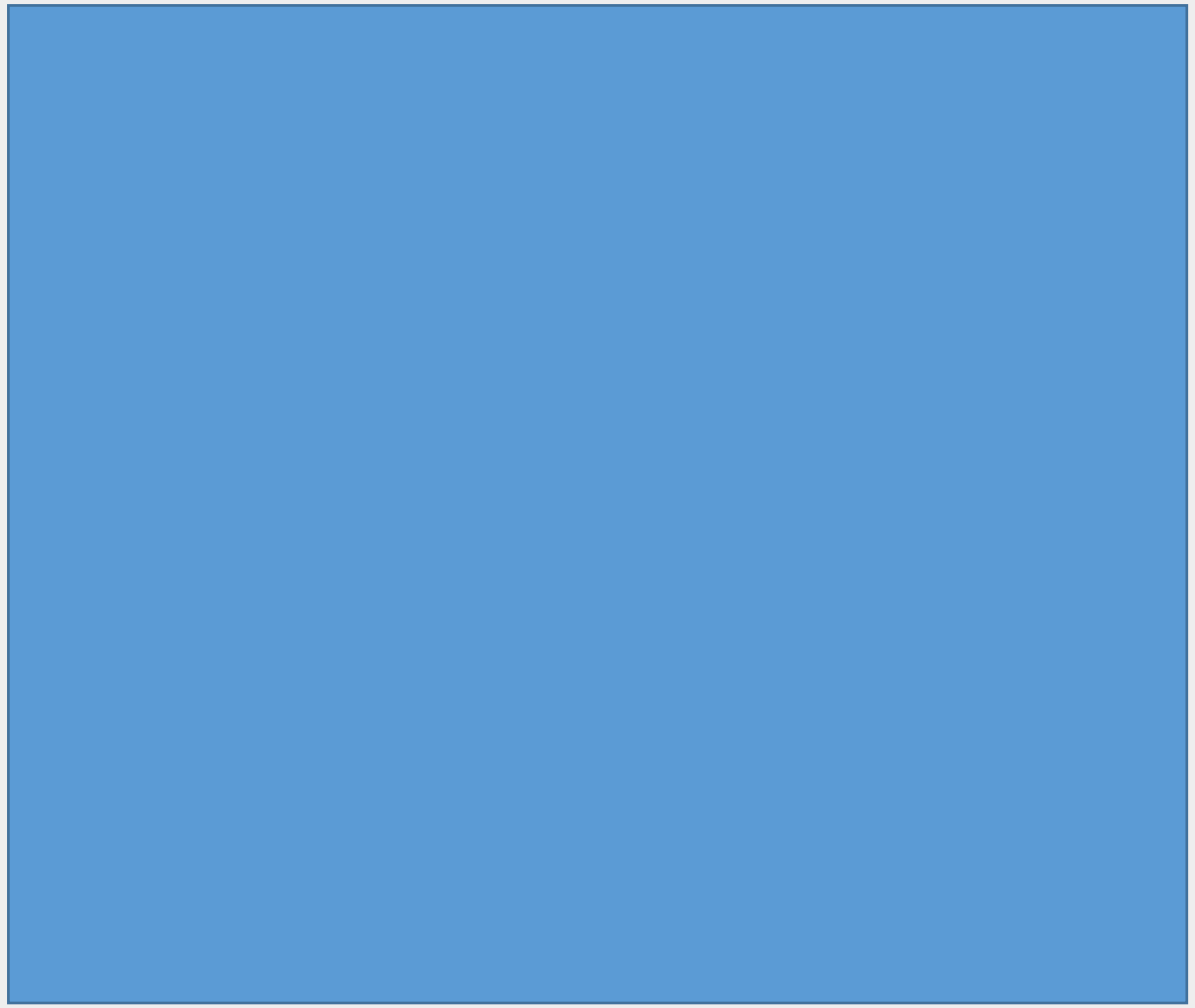
Add RPC

- Import RPC
  - import rpyc
  - from rpyc.utils.server import ThreadedServer
- Inherit from rpyc.Service class
- Add “exposed_” to methods
- Start a server
  - server = ThreadedServer(RPC_List, port = 12345)
  - server.start()



Create the Client

- Simple:
  - Create a class “Client”
  - Create connection
    - c= **rpyc.connect('localhost', 12345)**
  - Call remote functions over that connection
    - c.root.exposed_<name>()



Exercise 2b: ZMQ

- Abstracts low level socket implementations
  - The same interface irrespective of the underlying OS
- Provides a higher level model by pairing sockets:
  - one for sending messages at process P
  - a corresponding one at process Q for receiving messages
- TCP-based, all communication is connection-oriented
- Allows many-one communication with sockets (server can listen to multiple ports)
- All communication is asynchronous

Example: ZeroMQ Request-Reply

- Request-reply pattern
  - Traditional client-server communication
  - A client application uses a **request socket** (of type REQ) to send a request message to a server and a response
  - The server uses a **reply socket** (of type REP)
- Advantages
  - Simplifies development by avoiding need to call listen (or accept)
  - When a server receives a message, a subsequent call to send is automatically targeted toward the original sender.
  - When a client calls the recv operation (for receiving a message) after having sent a message, ZeroMQ assumes the client is waiting for a response from the original recipient

ZMQ setup

- Pip install pyzmq

Request-reply Server

- Create ZMQ context
  - Keeps the list of open sockets and manages async I/O thread that handles messages on sockets
  - **context = zmq.Context()**
- Create response socket
  - **s = context.socket(zmq.REP)**
- Bind to address/port
  - **s.bind('tcp://127.0.0.1:7777')**
- While loop to **recv** messages
  - **while True:**

    message = s.recv_string()

    s.send_string(‘pong’)

Request-reply client

- Setup context
  - **context = zmq.Context()**
- Create request socket
  - **s = context.socket(zmq.REQ)**
- Connect to socket
  - **s.connect('tcp://127.0.0.1:7777')**
- Send message
  - **s.send_string("Ping")**
- Receive message
  - **s.recv_string()**

Config for optional exercises

- Docker
  - Dockerfile
    - FROM python:latest
    - RUN apt-get update -y
    - RUN DEBIAN_FRONTEND="noninteractive" apt-get -y install tzdata
    - RUN apt-get install openmpi-bin libopenmpi-dev rabbitmq-server -y
    - RUN pip install mpi4py rpyc pyzmq pika
  - Build
    - docker build -t communication-class .
  - Run first terminal
    - docker run -it -v <local>:<container> communication-class bash
      - docker run -it -v "$(pwd)"/code:/code communication-class bash
  - Connect to container and run second terminal
    - docker exec -it <CONTAINER ID> bash

Example – MPI4Py

- First you need an MPI runtime (e.g., MPICH, OpenMPI)
  - E.g., on Ubuntu openmpi-bin and libopenmpi-dev
- Second you will need to install mpi4py
  - Pip install mpi4py
- Third when running you need to use an mpi executor
  - mpiexec -n 4 python p2p.py

Example 1: Point-to-point communication

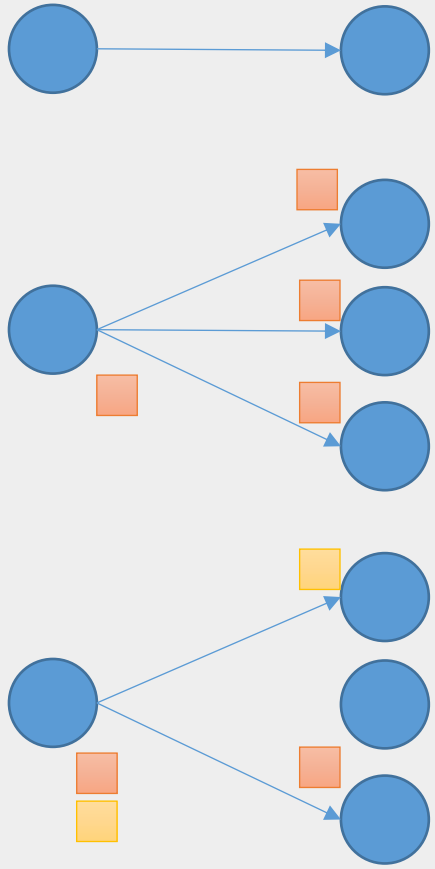
```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    print("Rank 0: sending")
    data = {'a': 7, 'b': ‘cat’}
    comm.send(data, dest=1)
elif rank == 1:
    data = comm.recv(source=0)
    print("Rank %s recieved %s" % (rank, data))
```

Comparing MPI Communication Models

- Point to point
  - Send message from to a specific rank
- Broadcast
  - Send messages to all other ranks
- Scatter/Gather
  - Send messages randomly to individual nodes



Broadcast

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [1,2,3],
            'key2' : ('type', 'cat')}
else:
    data = None

data = comm.bcast(data, root=0)
print("Rank %s recieved %s" % (rank, data))
```

Scatter

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
   data = [(x+1)**x for x in range(size)]
   print ('Scattering:',data)
else:
   data = None

data = comm.scatter(data, root=0)
print ('rank %s got data %s' % (rank, data))
```

Example: Rabbit MQ

- Implements AMQP
- One of the most well-known MQ systems

- Setup
  - Install rabbitmq_server
  - Install Python client (Pika)

- Start MQ server
  - rabbitmq-server - start

Receiving

- Setup a connection and channel
- Declare a queue
- Define a callback method
- Associate callback with the queue

Receiving

```
import pika
connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.queue_declare(queue='hello')

def callback(ch, method, properties, body):
    print("Received %s" % body)

channel.basic_consume(
    queue='hello', on_message_callback=callback, auto_ack=True)

print('Waiting for message ...')
channel.start_consuming()
```

Sending

- Setup connection and channel
- Declare a queue
- Send message to the queue

Sending

```
import pika
connection =
pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

channel.basic_publish(exchange='', routing_key='hello', body='Hello World!')

print("Message sent.")
connection.close()
```

Exercise 2

- Please submit RPC and ZMQ examples:
  - https://classroom.github.com/a/dXa9WhWB

Homework 3

Use ZMQ to create a chat server

1. Create client/server for posting messages (like homework 1 but with zmq)
2. Create a pub/sub model for distributing messages
3. Combine 1 and 2 with some form of concurrency
4. Add individual channels