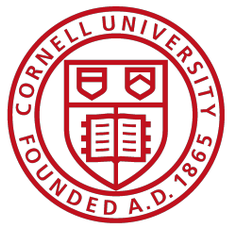
CS4450

Computer Networks: Architecture and Protocols

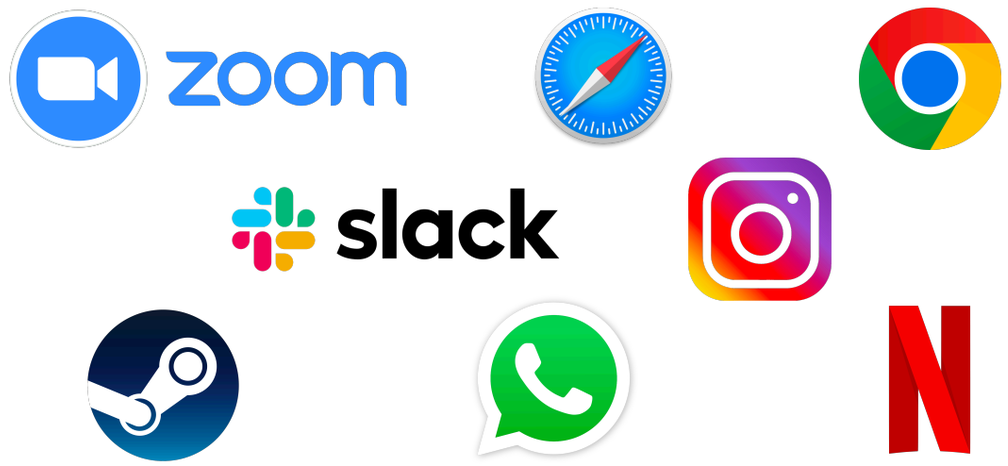
Lecture 16
Socket Programming

Midhul Vuppalapati



Goals for Today's Lecture

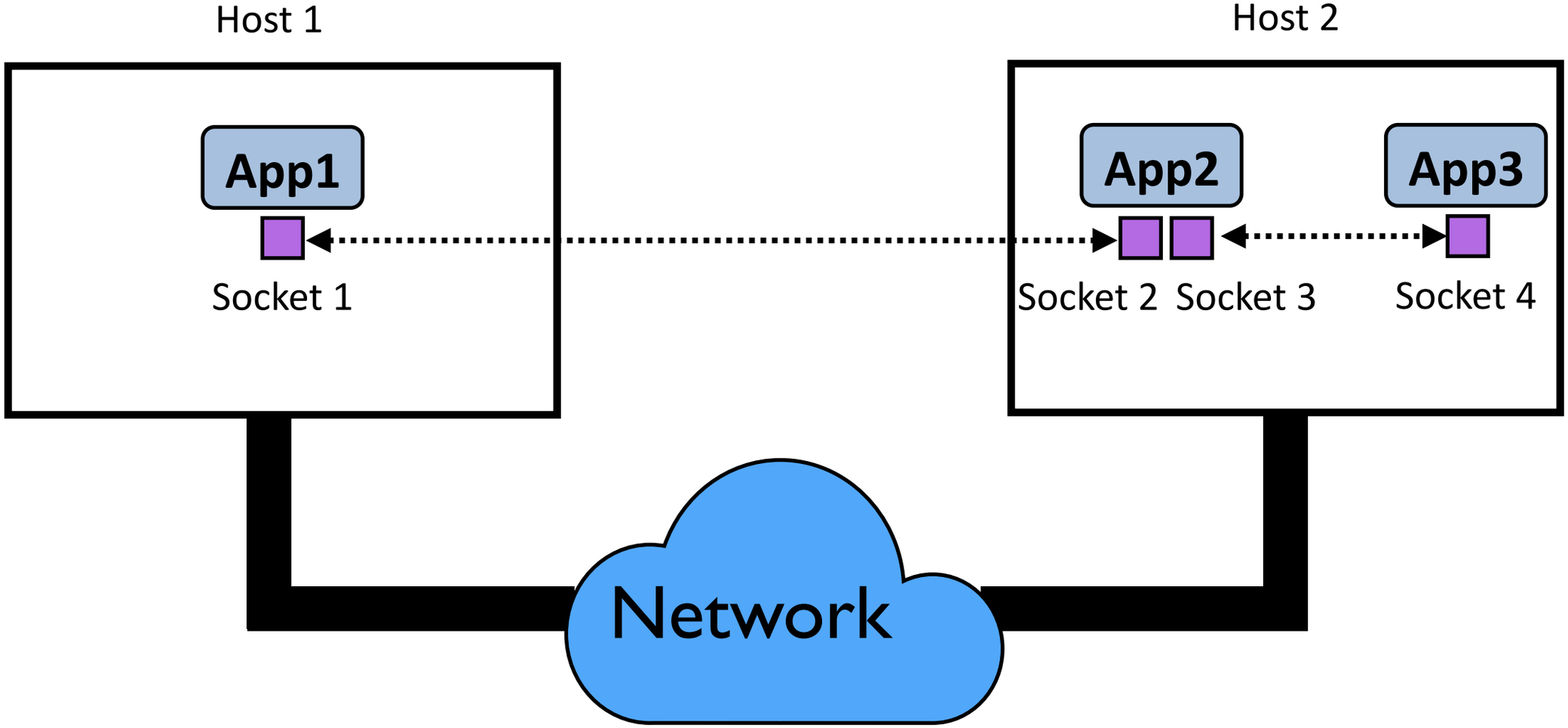
- Understanding the Socket API
- How does the Socket API work under the hood?
- Programming with the Socket API
  - Live demo



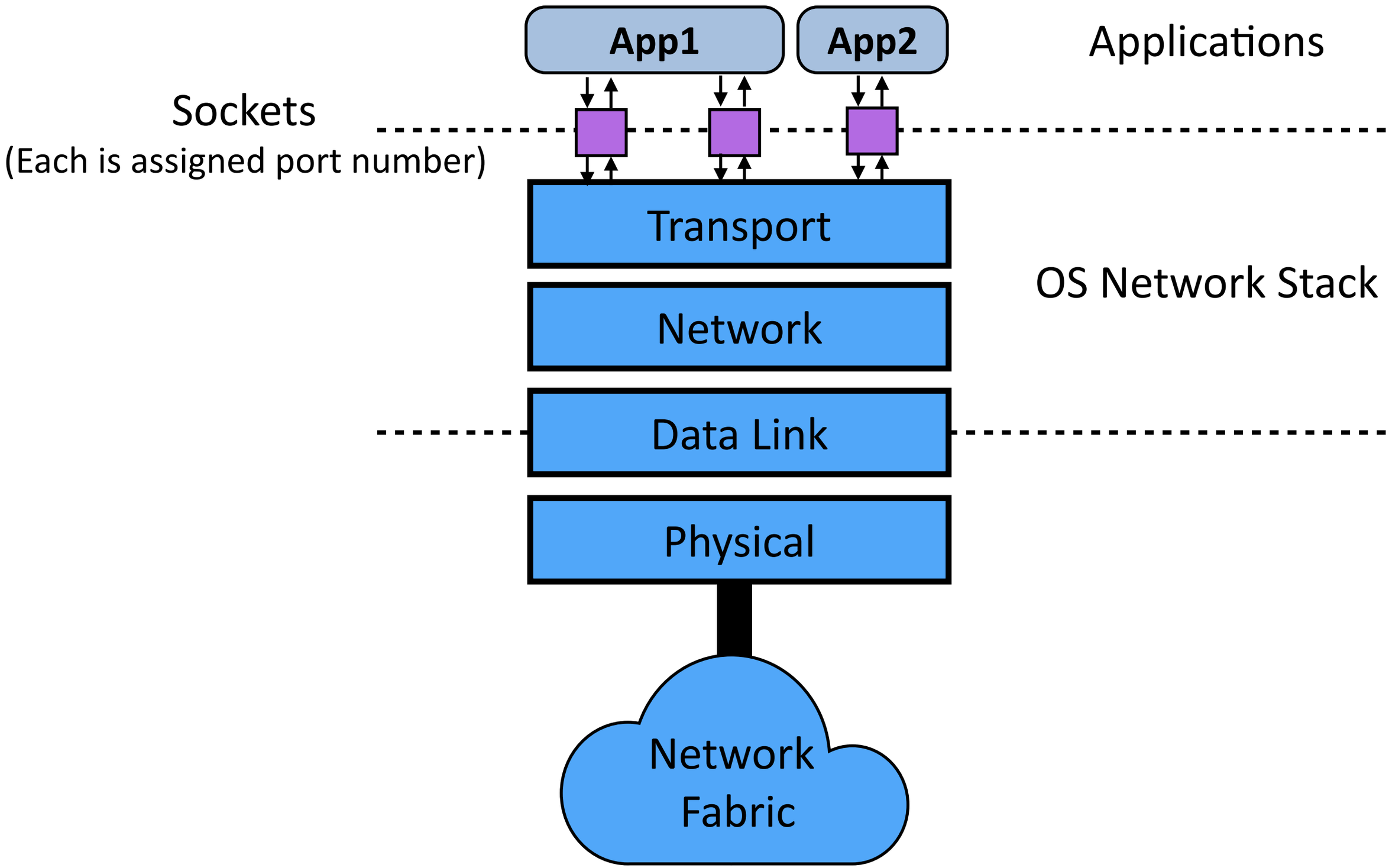
Nearly all applications you use today rely on the Socket API

What is the Socket API?

- **Socket API:** Interface for applications to communicate with each other
  - Provided by the Operating System (OS)
  - **Key abstraction:** Socket (End-point for communication)
  - Applications can be on same or different hosts



Where do Sockets fit in the end-to-end picture?



How do applications interact with sockets?

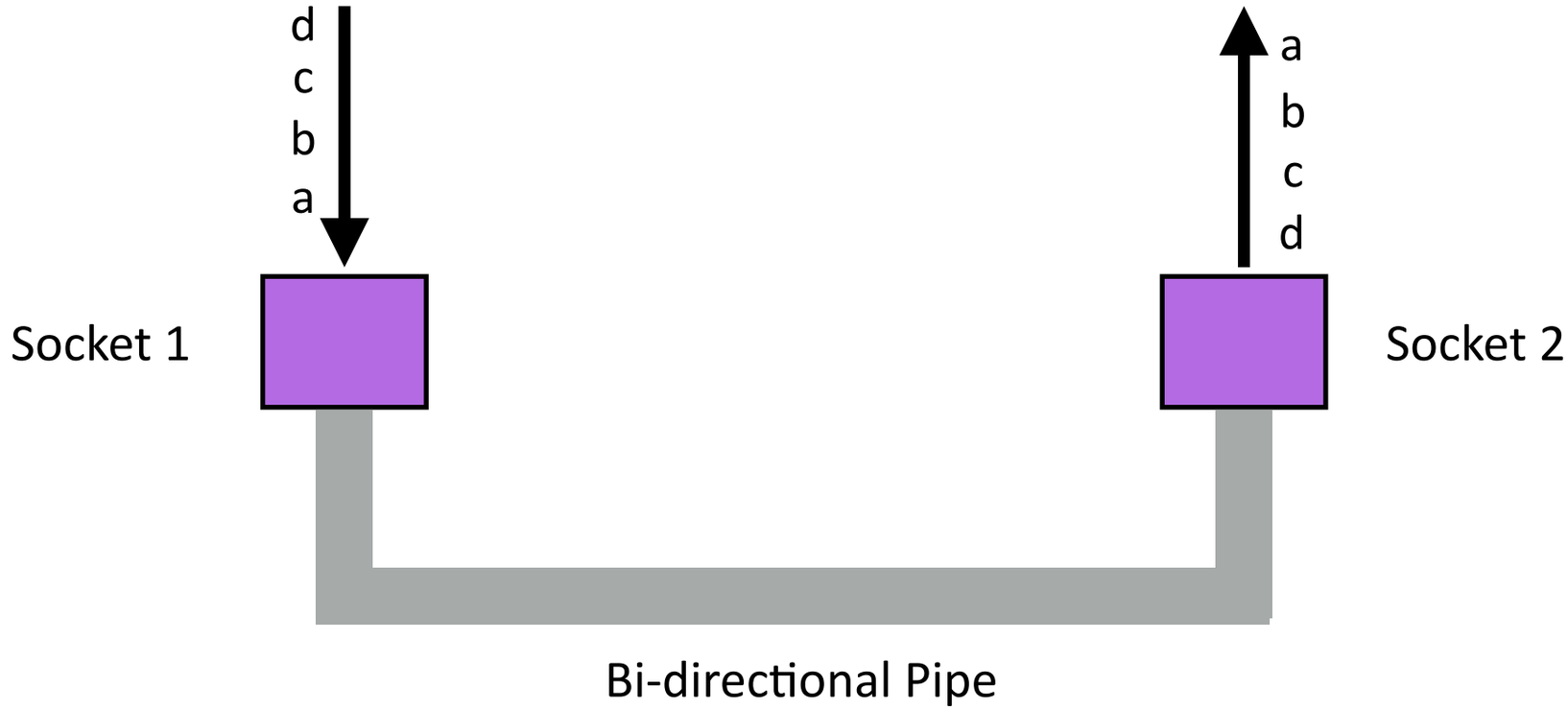
- Socket works very similar to file
  - **~~open()~~ socket()**: Open a socket
  - **read()**: Read data from a socket (i.e. receive data)
  - **write()**: Write data to a socket (i.e. send data)
  - **close()**: close a socket
- Note
  - **recv()** is alternative for **read()**
  - **send()** is alternative for **write()**

Types of communication

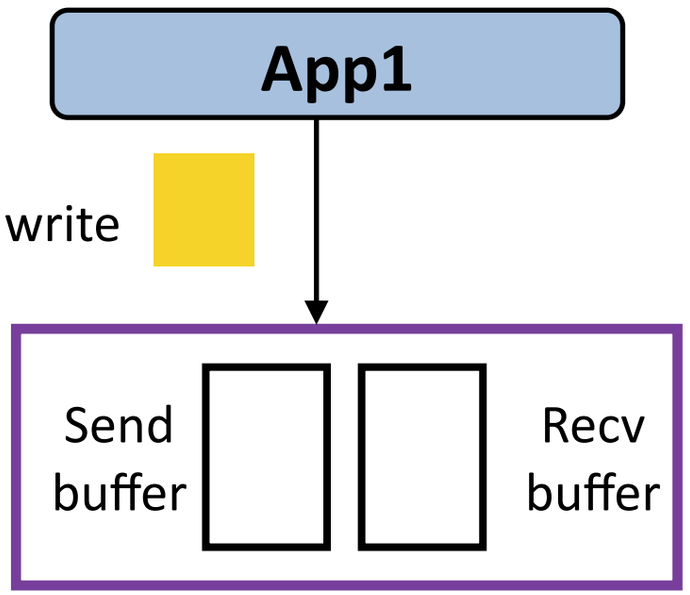
- Two main modes of communication
  - **Connection-oriented (Stream sockets)**
    - First, connection is established between a pair of sockets
    - Then, data is exchanged between them
    - “Pipe” abstraction (reliable & in-order delivery of data)
    - Implemented on top of TCP Transport
  - **Connection-less (Datagram sockets)**
    - No connection establishment
    - Data can be directly exchanged between sockets
    - No guarantee of reliable or in-order delivery
    - Implemented on top of UDP transport

Stream sockets: Pipe Abstraction

- Bi-directional “pipe” between a pair of sockets
  - Sequence of bytes sent on one end will be received on other end
    - Reliable delivery
    - In-order delivery
    - No duplication



Under the hood



Src IP: 10.0.0.1

Src port: 5001

Dst IP: **11.0.0.1**

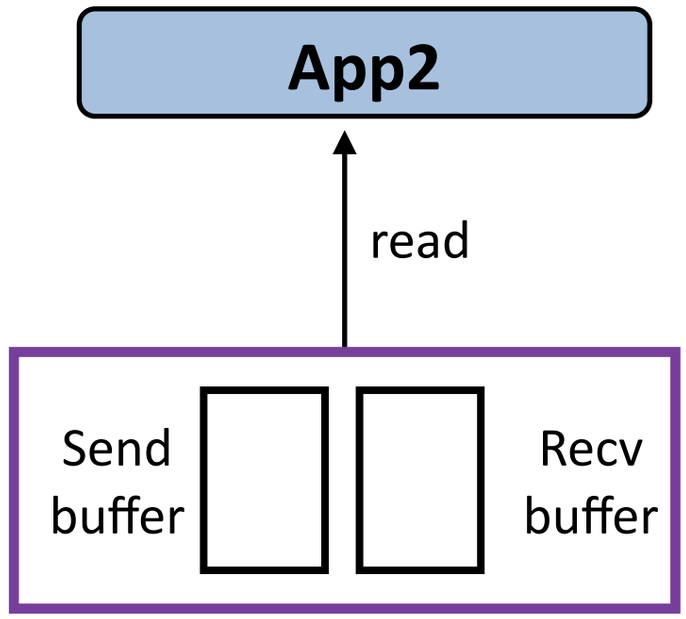
Dst port: **6001**

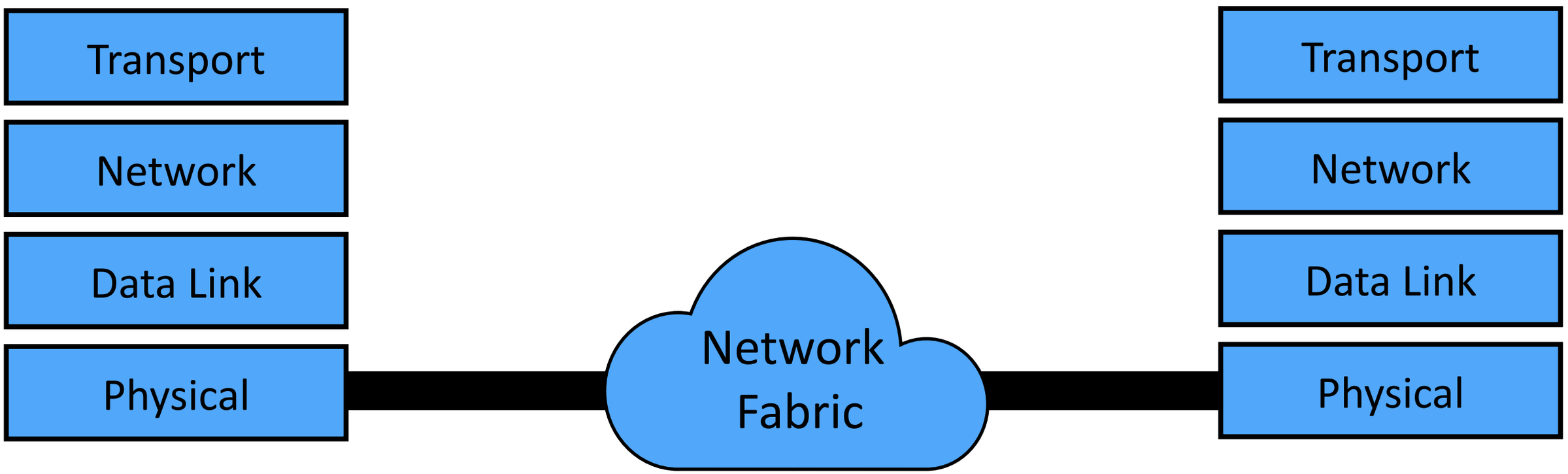
Src IP: 11.0.0.1

Src port: 6001

Dst IP: **10.0.0.1**

Dst port: **5001**





Details of reading / writing to / from sockets

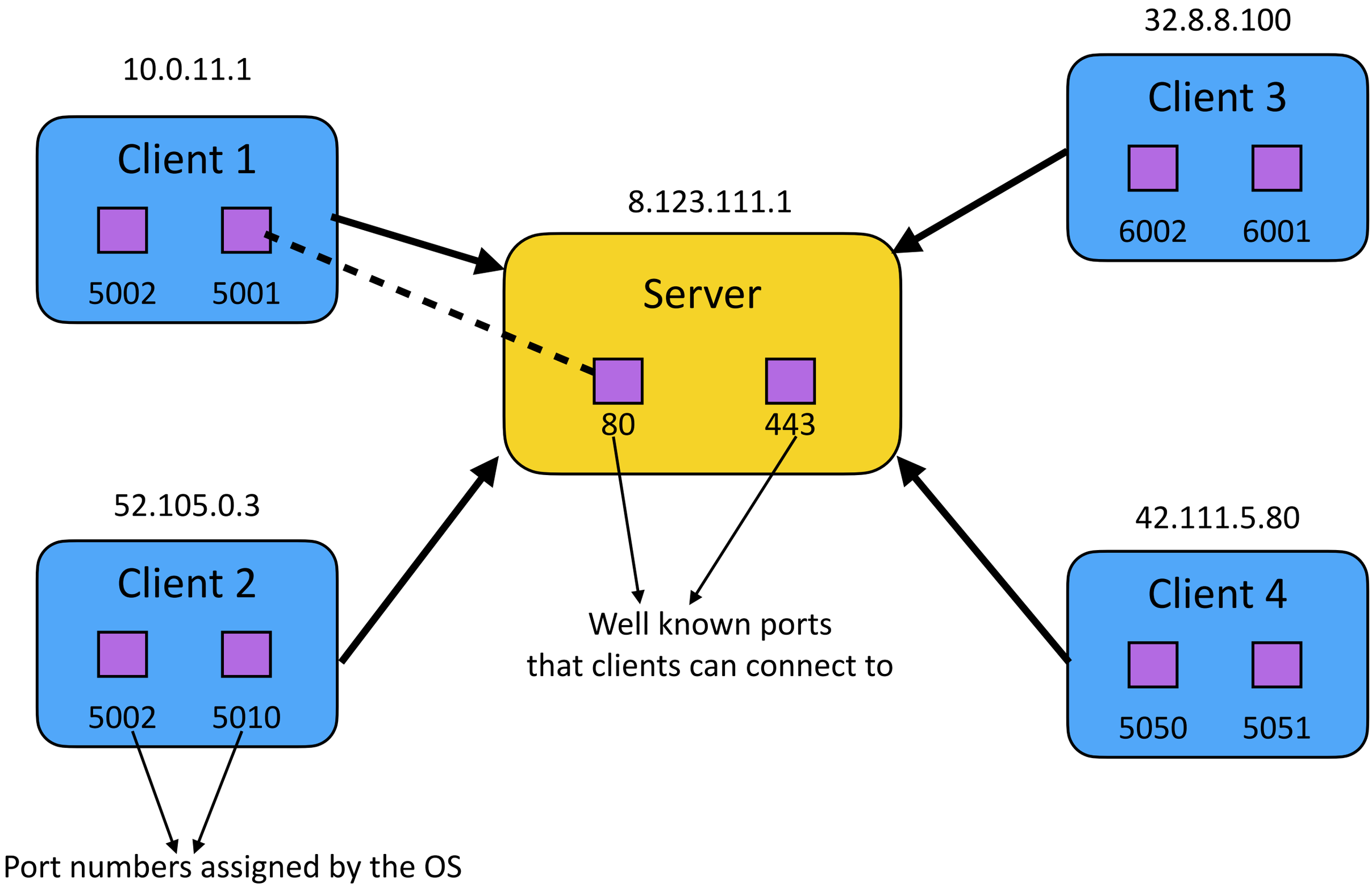
- **read(X bytes)**
  - Reads **up to** X bytes from the socket receive buffer
  - Returns the number of bytes actually read upon completion
  - Upon successful completion: can return **any non-zero value <= X**
  - E.g: If you read(100 bytes) and socket receive buffer has only 50 bytes, then it will read 50 bytes and return
- **write(X bytes)**
  - Writes **up to** X bytes to the socket send buffer
  - Returns the number of bytes actually written upon completion
  - Upon successful completion: can return **any non-zero value <= X**
  - E.g: If you write(100 bytes) and socket send buffer has only 50 bytes of space left, then it will write 50 bytes and return

What if socket receive buffer is empty or send buffer is full?

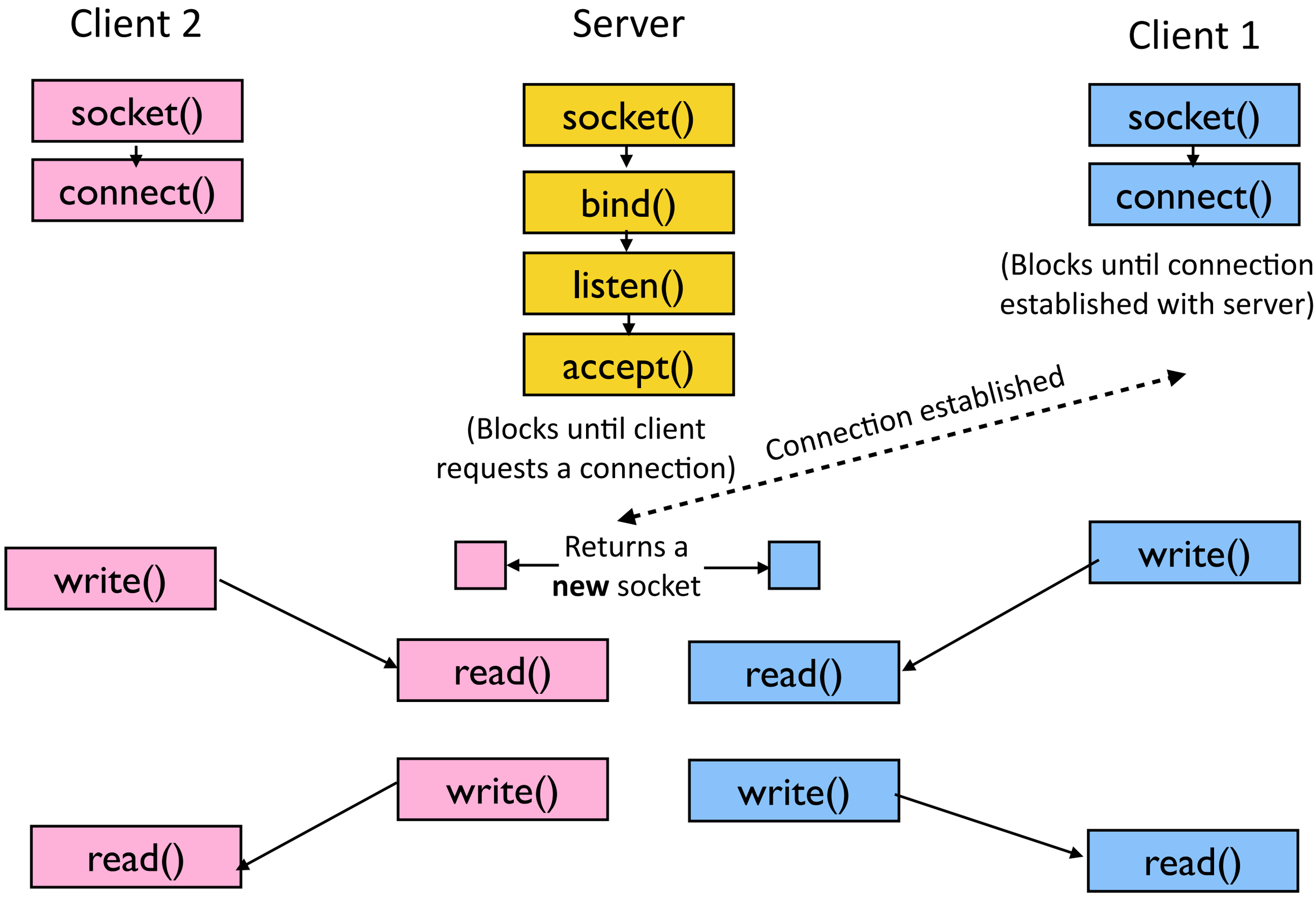
Blocking vs Non-blocking sockets

- What if socket receive buffer is empty / send buffer is full?
  - read / write call cannot complete successfully
  - Two modes:
    - (Default) **Blocking mode**
      - Application is blocked until read / write call can complete successfully
    - **Non-blocking mode**
      - read / write call returns immediately with ERROR

Client-Server Model



Client-Server Model - APIs



Demo

Designing servers for massive scale

- **Scalability of the server that we just wrote (in demo)**
  - Works fine for 100s-1000s of connections
  - **What if we want to handle 100,000+ connections?**
    - **Problem:** 1 thread per-connection
    - 100,000s of connections => 100,000s of threads => **Inefficient**
- **Can single thread handle multiple connections?**
  - Yes. How: Use non-blocking sockets
  - **Challenge:** How to monitor multiple sockets?
    - **Naive approach:** Thread constantly iterates over all sockets and check if any of them have data available (Not efficient)
    - **Solution:** OS provides APIs to monitor sockets (*select*/*poll*/*epoll*)
  - **Con:** Difficult to program with

Questions?