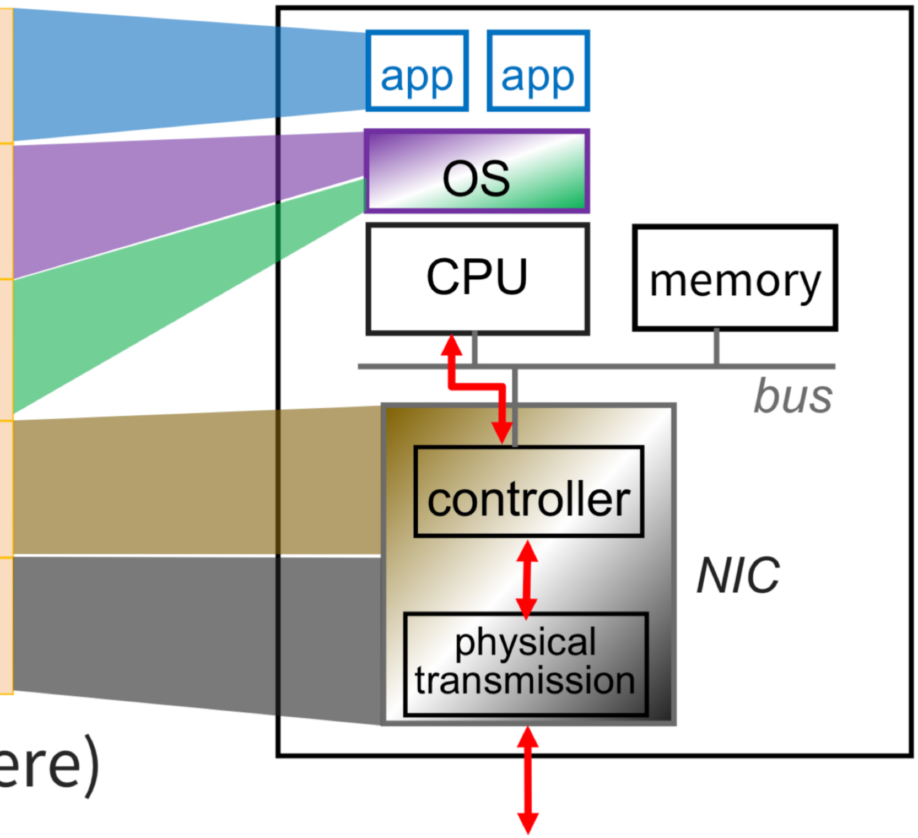
Lecture 19: TCP

CS 105

November 14, 2019

OSI Network Model

Application	HTTP, FTP, DNS (*these*^ are usually in libraries)
Transport	TCP, UDP
Network	IP, ICMP (ping)
Link	Ethernet, WiFi
Physical	wires, signal encoding



(Hard to draw firm lines here)

Transport Layer Protocols

User Datagram Protocol (UDP)

- **unreliable, unordered delivery**
- connectionless
- best-effort, segments might be lost, delivered out-of-order, duplicated
- reliability (if required) is the responsibility of the app

Transmission Control Protocol (TCP)

- **reliable, inorder delivery**
- connection setup
- flow control
- congestion control

UDP: tradeoffs

- fast:
  - no connection setup
  - no rate-limiting
- simple:
  - no connection state
  - small header

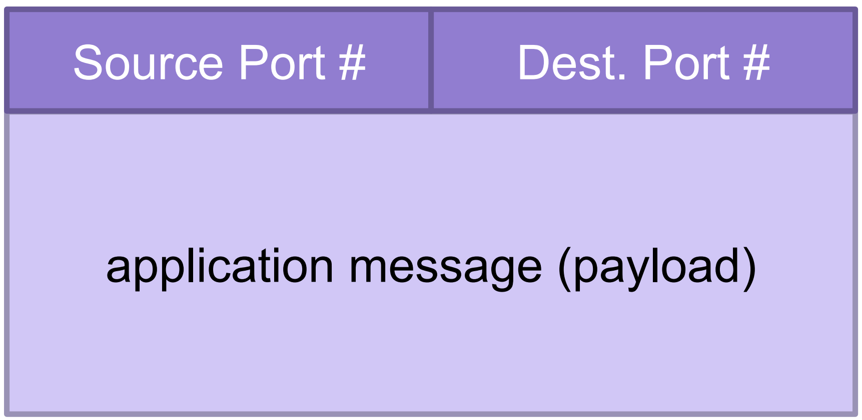
- (possibly) extra work for applications
  - reordering
  - duplicate suppression
  - handle missing packets

Transport Protocols by Application

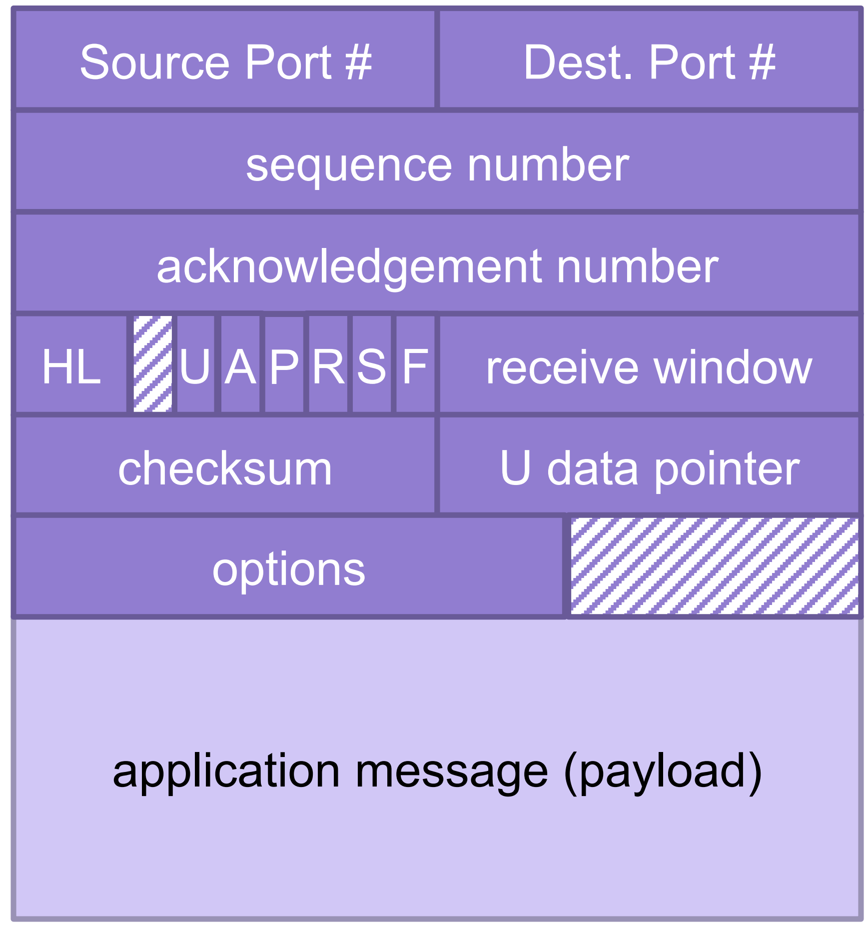
Application	Application-Level Protocol	Transport Protocol
Name Translation	DNS	Typically UDP
Routing Protocol	RIP	Typically UDP
Network Management	SNMP	Typically UDP
Remote File Server	NFS	Typically UDP
Streaming multimedia	(proprietary)	UDP or TCP
Internet telephony	(proprietary)	UDP or TCP
Remote terminal access	Telnet	TCP
File Transfer	(S)FTP	TCP
Email	SMTP	TCP
Web	HTTP(S)	TCP

Transport-Layer Segment Formats

UDP

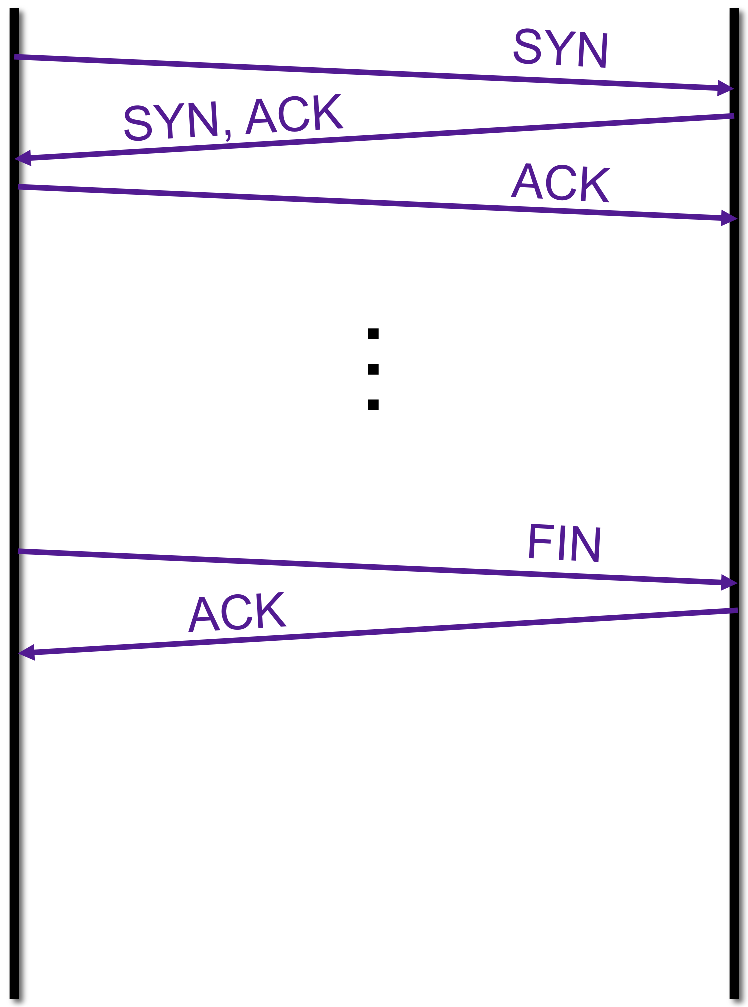


TCP



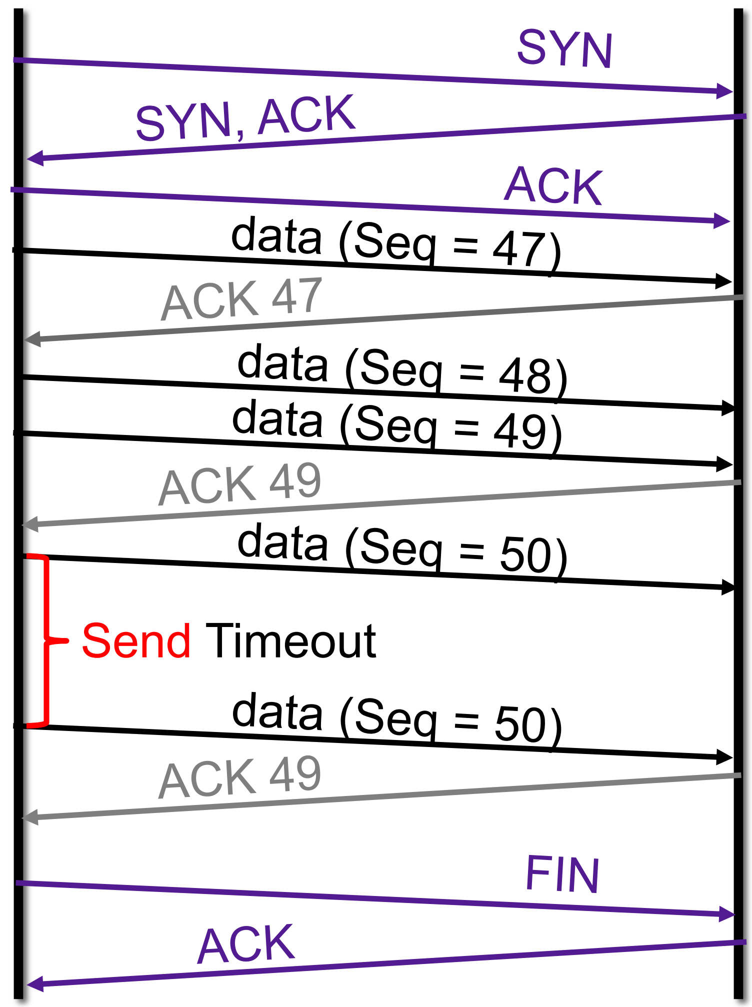
TCP Connections

- TCP is connection-oriented
- A connection is initiated with a three-way handshake
  - Recall: server will typically create a new socket to handle the new connection
- FIN works (mostly) like SYN but to teardown a connection



Reliable Transport

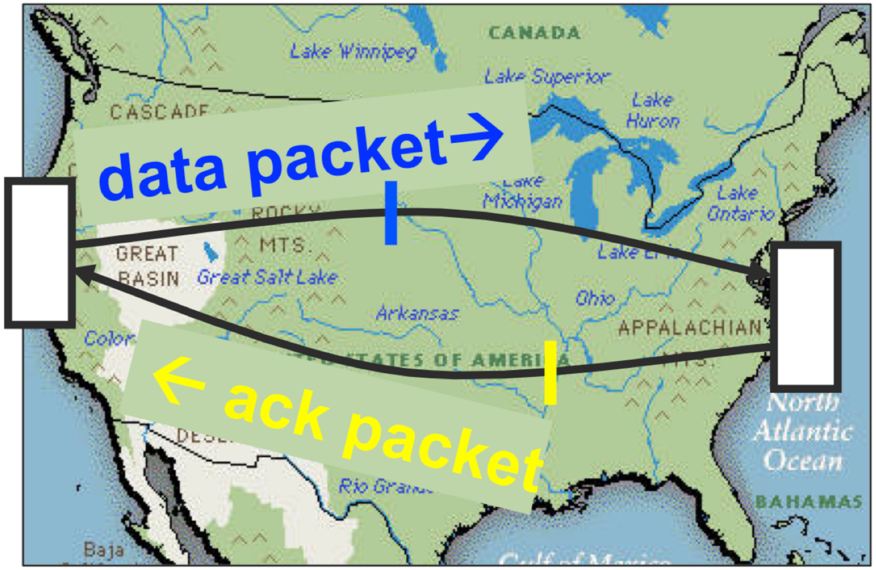
- Each SYN segment will include a randomly chosen sequence number
- Sequence number of each segment is incremented by data length
- Receiver sends ACK segments acknowledging latest sequence number received
- Sender maintains copy of all sent but unacknowledged segments; resends if ACK does not arrive within timeout
- Timeout is dynamically adjusted to account for round-trip delay

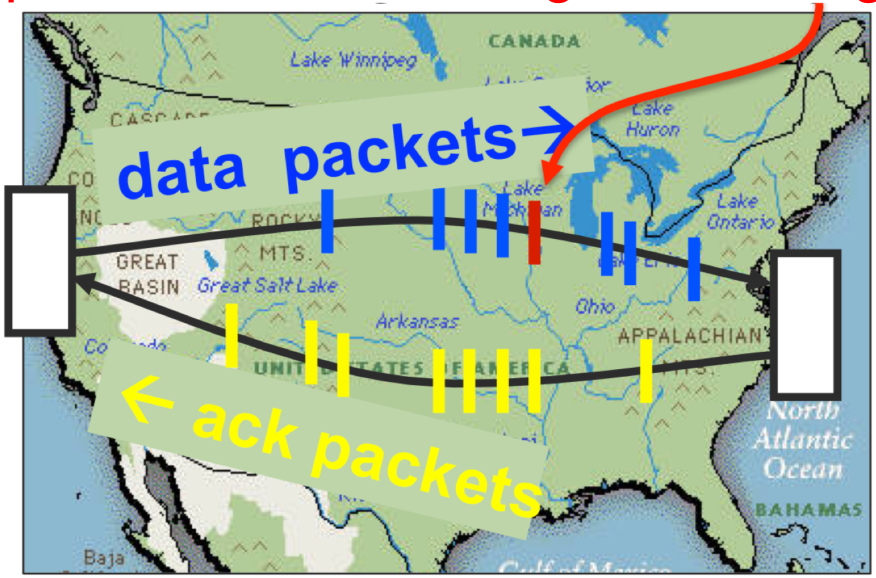


Pipelined Protocols

- Pipelining allows sender to send multiple "in-flight", yet-to-be-acknowledged packets
  - increases throughput
  - needs buffering at sender and receiver
- how big should the window be?

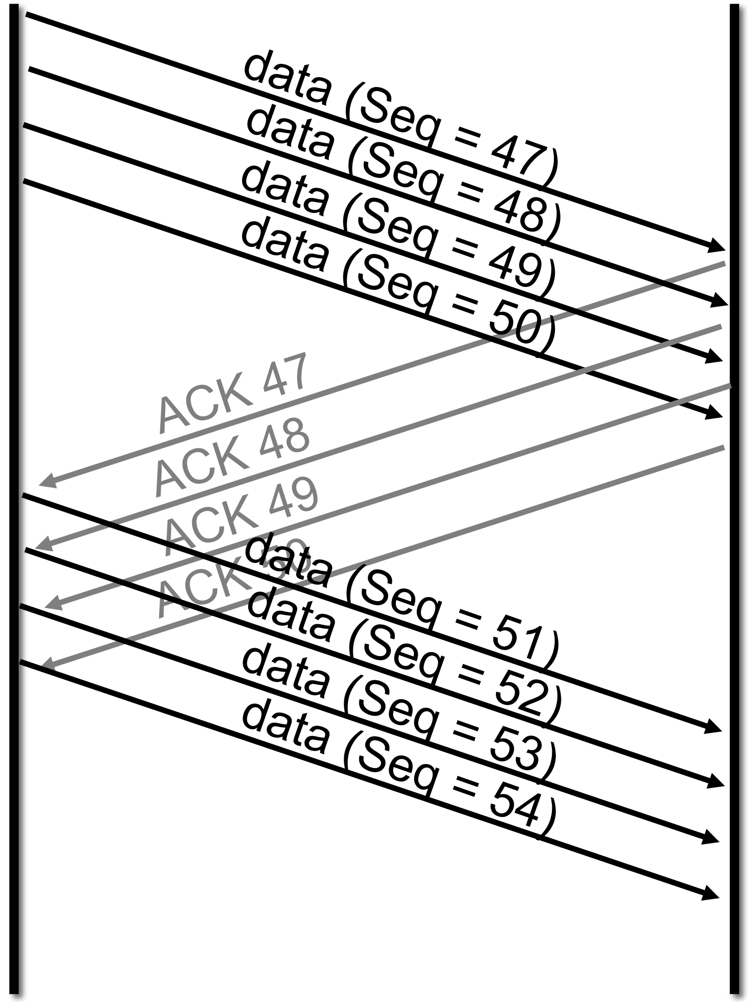
what if a packet in the middle goes missing?





Example: Window Size = 4

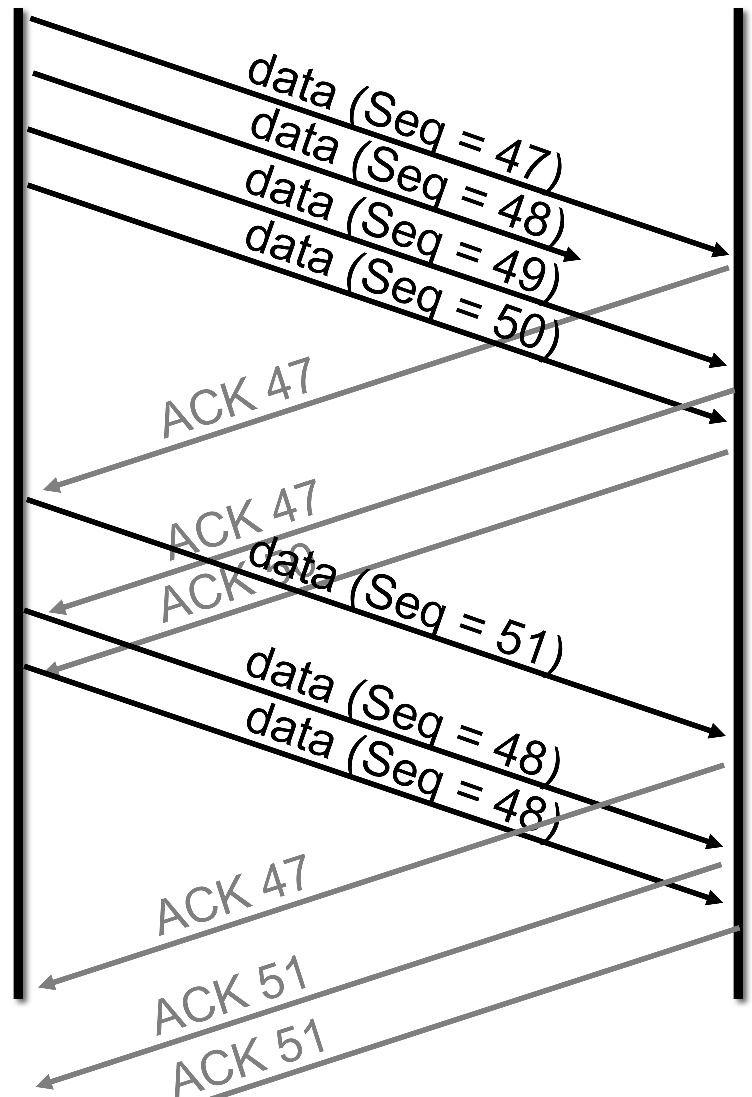
- sender can have up to 4 unacknowledged messages
- when ACK for first message is received, it can send another message



TCP Fast Retransmit

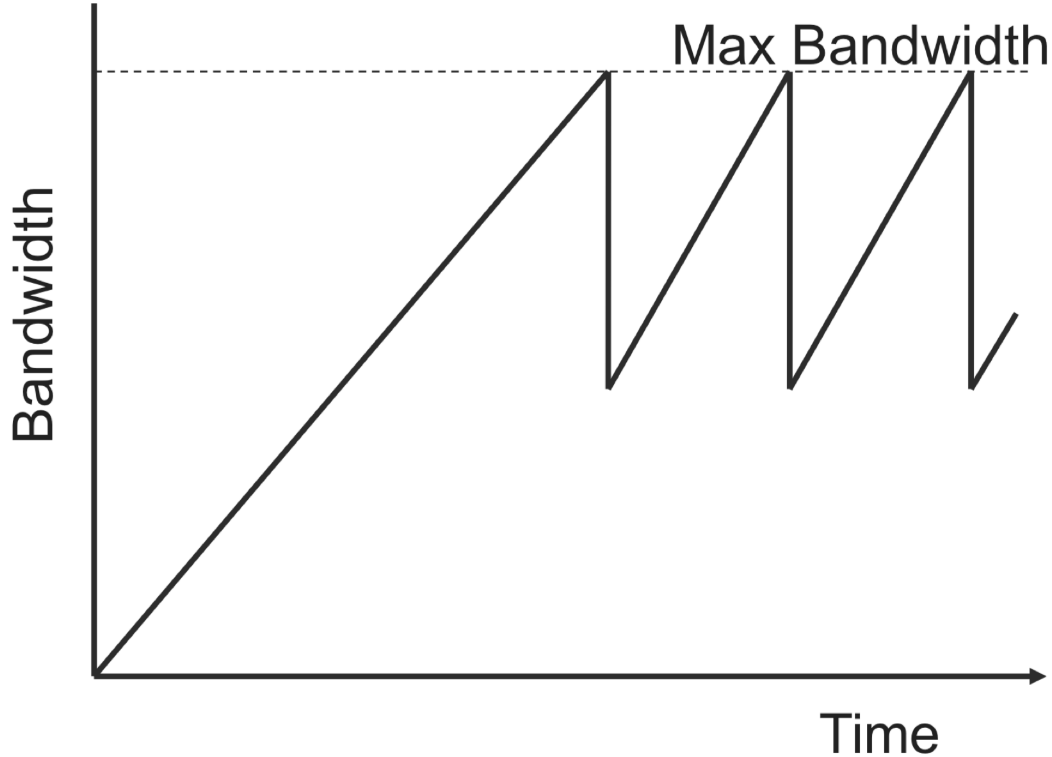
- Receiver always acks the last id it successfully received

- Sender detects loss without waiting for timeout, resends missing packet



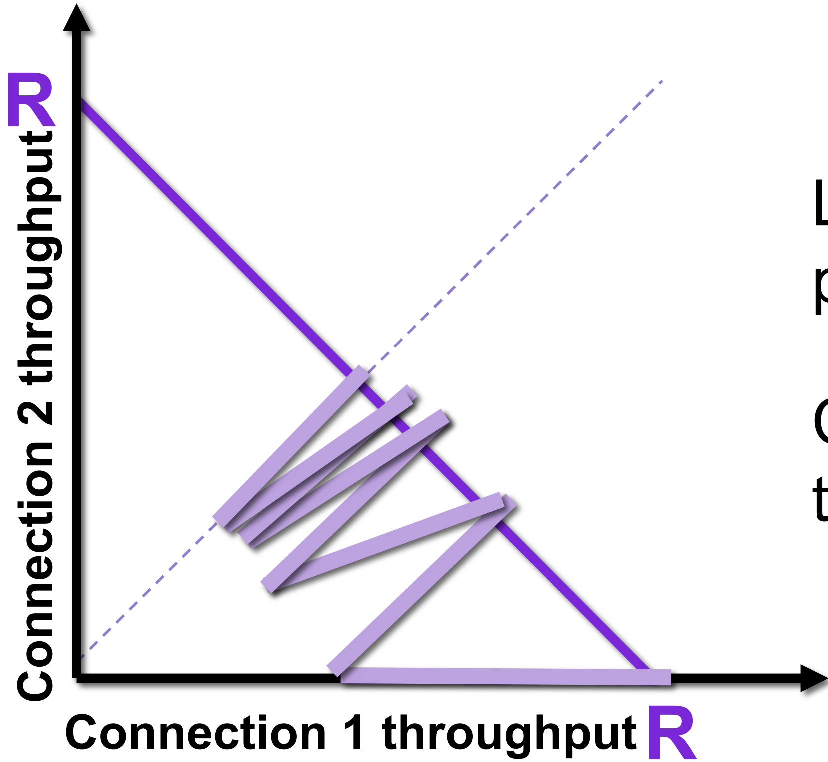
TCP Congestion Control

- TCP operates under a principle of additive increase-multiplicative decrease
  - window size++ every RTT if no packets lost
  - window size/2 if a packet is dropped



TCP Fairness

- Goal: if k TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/k

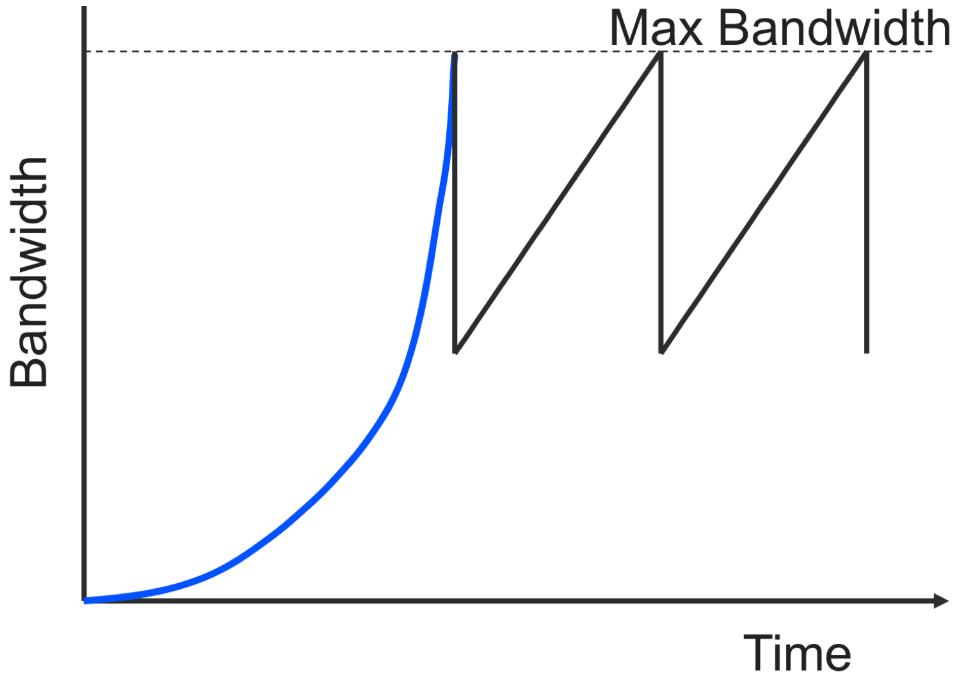


Loss: decreases throughput proportional to current bandwidth

Congestion avoidance: increases throughput linearly (evenly)

TCP Slow Start

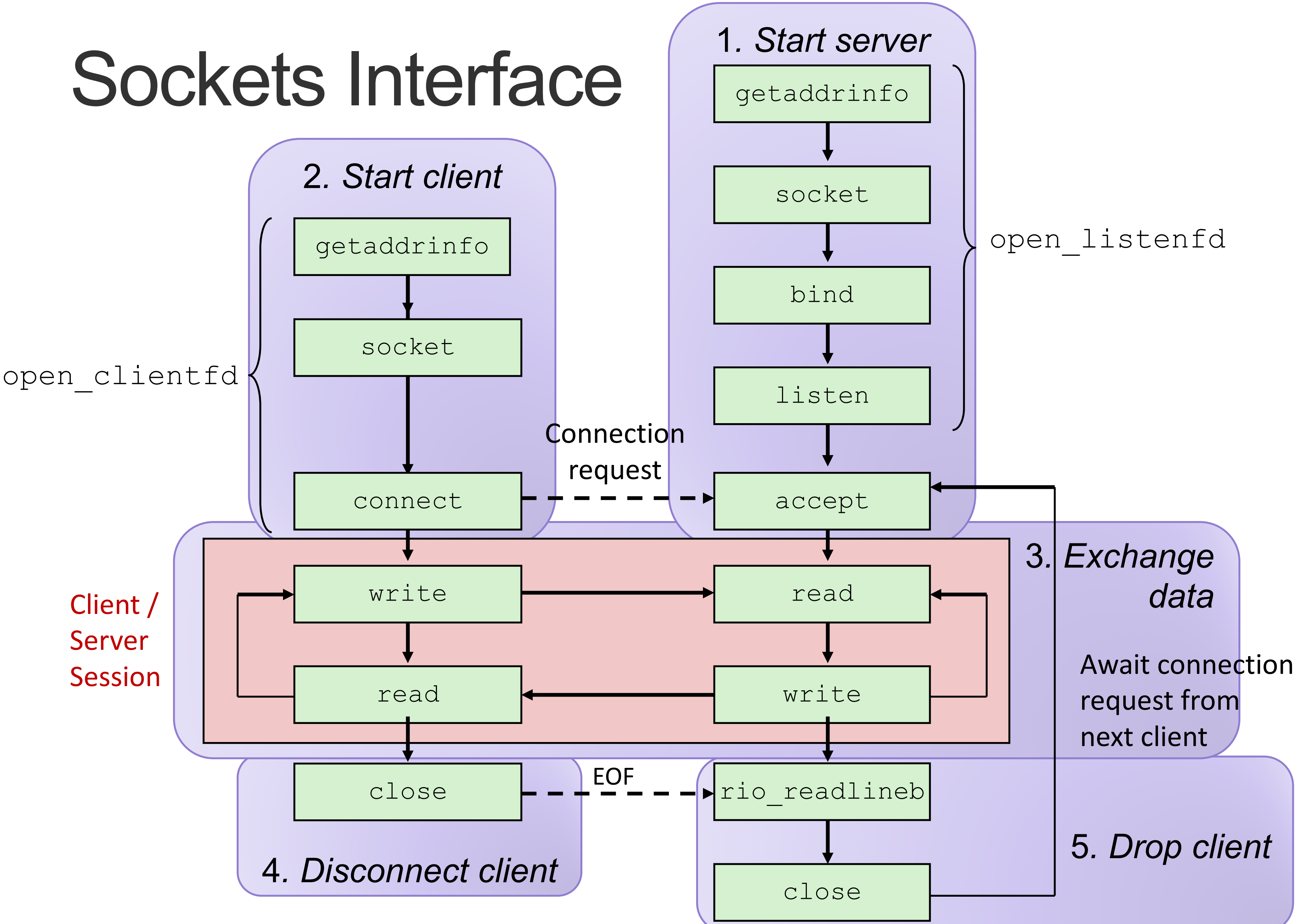
- Problem: linear increase takes a long time to build up a decent window size, and most transactions are small
- Solution: allow window size to increase exponentially until first loss



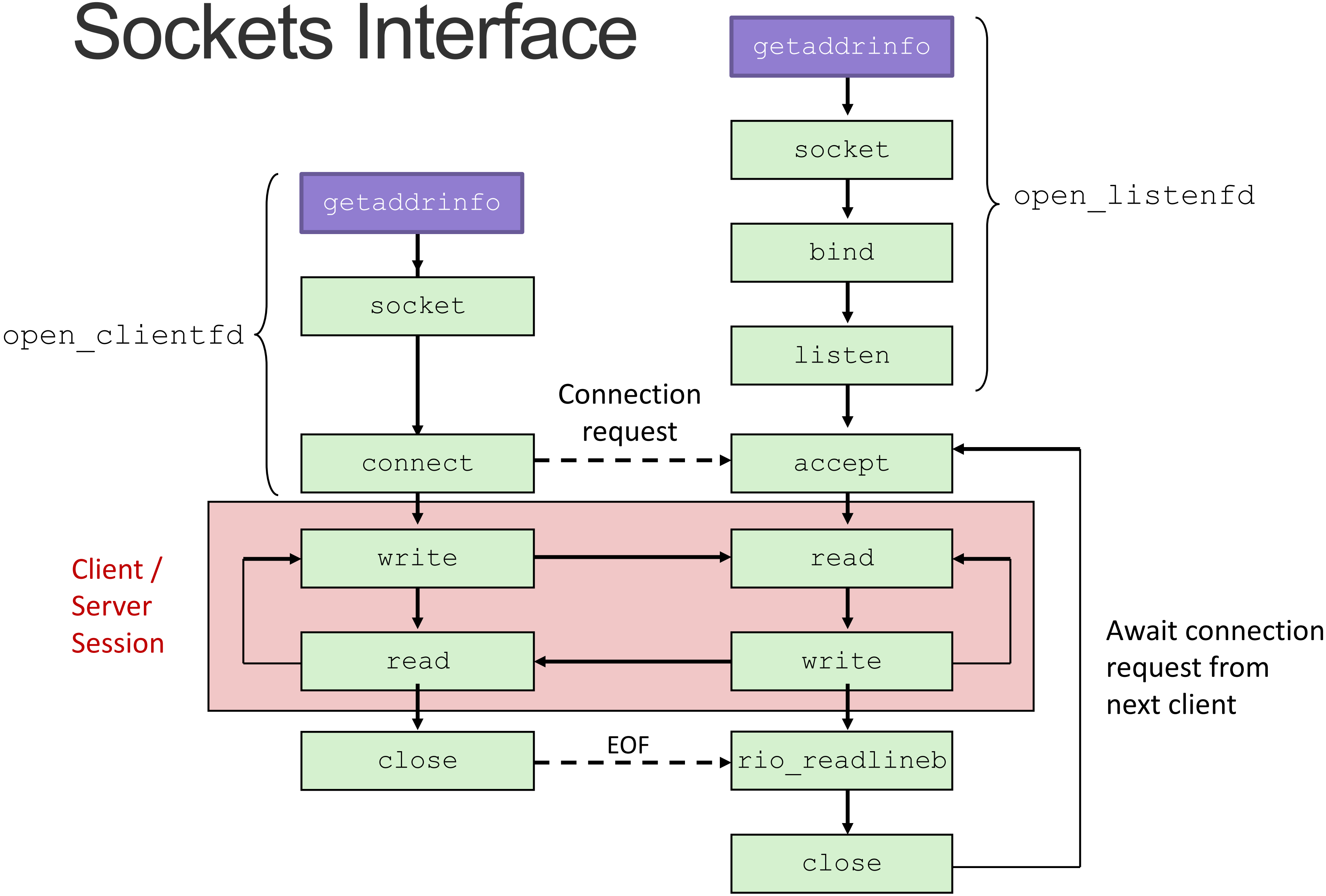
TCP Summary

- Reliable, in-order message delivery
- Connection-oriented, three-way handshake
- Transmission window for better throughput
  - timeouts based on link parameters (e.g., RTT, variance)
- Congestion control
  - Linear increase, exponential backoff
- Fast adaptation
  - Exponential increase in the initial phase

Sockets Interface



Sockets Interface



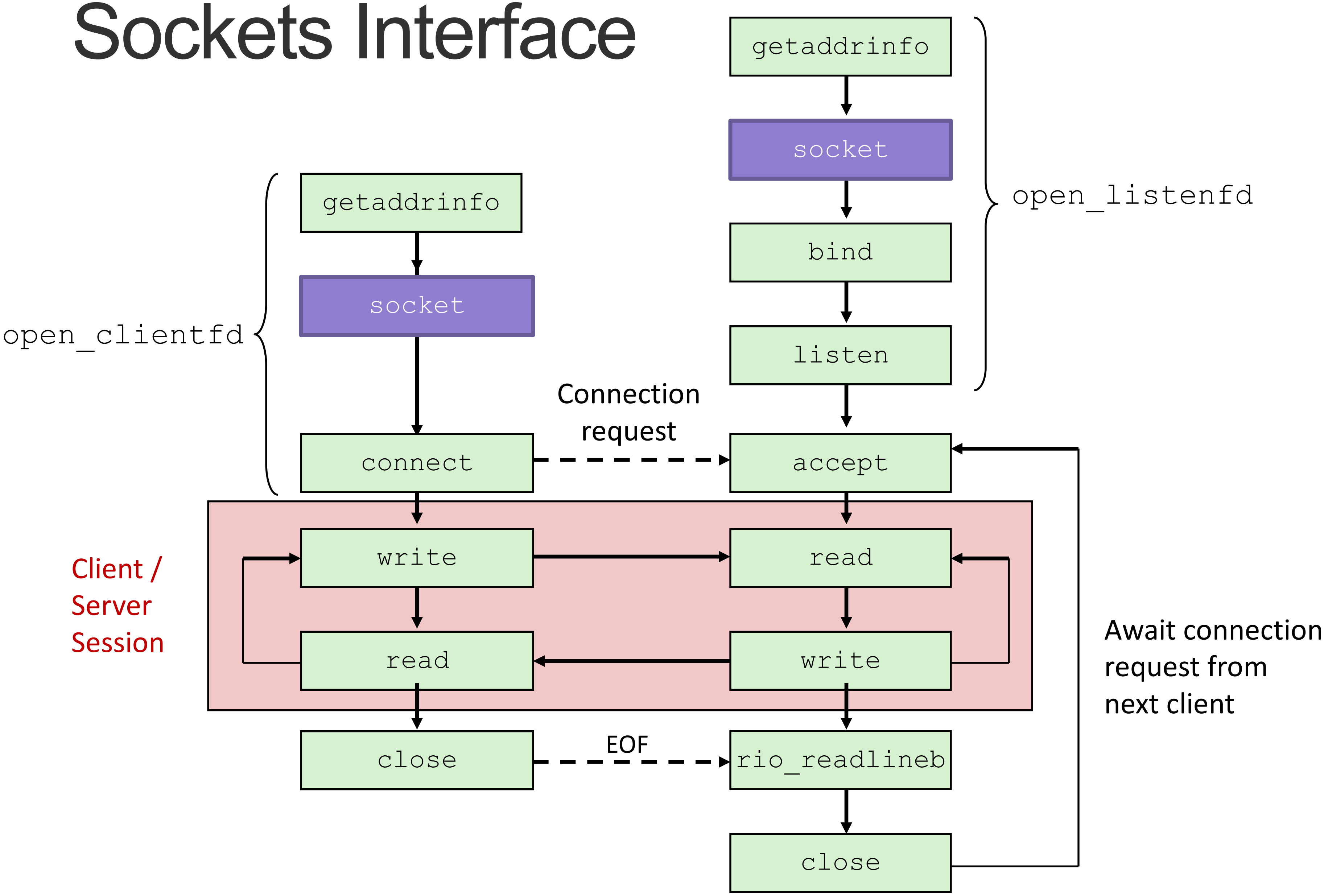
Host and Service Conversion: `getaddrinfo`

- `getaddrinfo` is the modern way to convert string representations of hostnames, host addresses, ports, and service names to socket address structures.
  - Replaces obsolete `gethostbyname` and `getservbyname` funcs.

- Advantages:
  - Reentrant (can be safely used by threaded programs).
  - Allows us to write portable protocol-independent code
    - Works with both IPv4 and IPv6

- Disadvantages
  - Somewhat complex
  - Fortunately, a small number of usage patterns suffice in most cases.

Sockets Interface



Sockets Interface: `socket`

- Clients and servers use the `socket` function to create a *socket descriptor*:

```
int socket(int domain, int type, int protocol)
```

- Example:

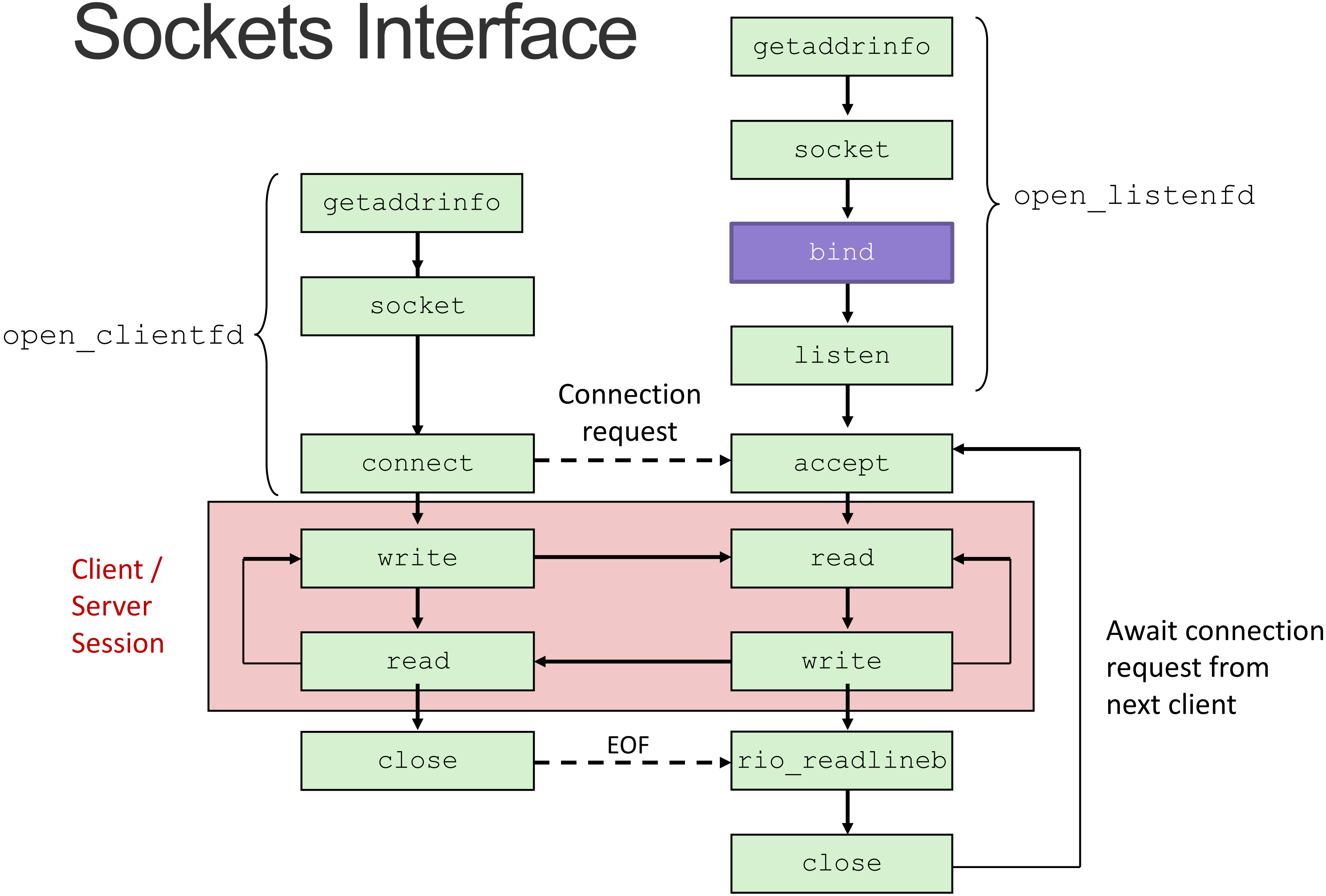
```
int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Indicates that we are using 32-bit IPV4 addresses (AF_INET)

Indicates that the socket will be the end point of a connection (SOCK_STREAM)

Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.

Sockets Interface



Sockets Interface: `bind`

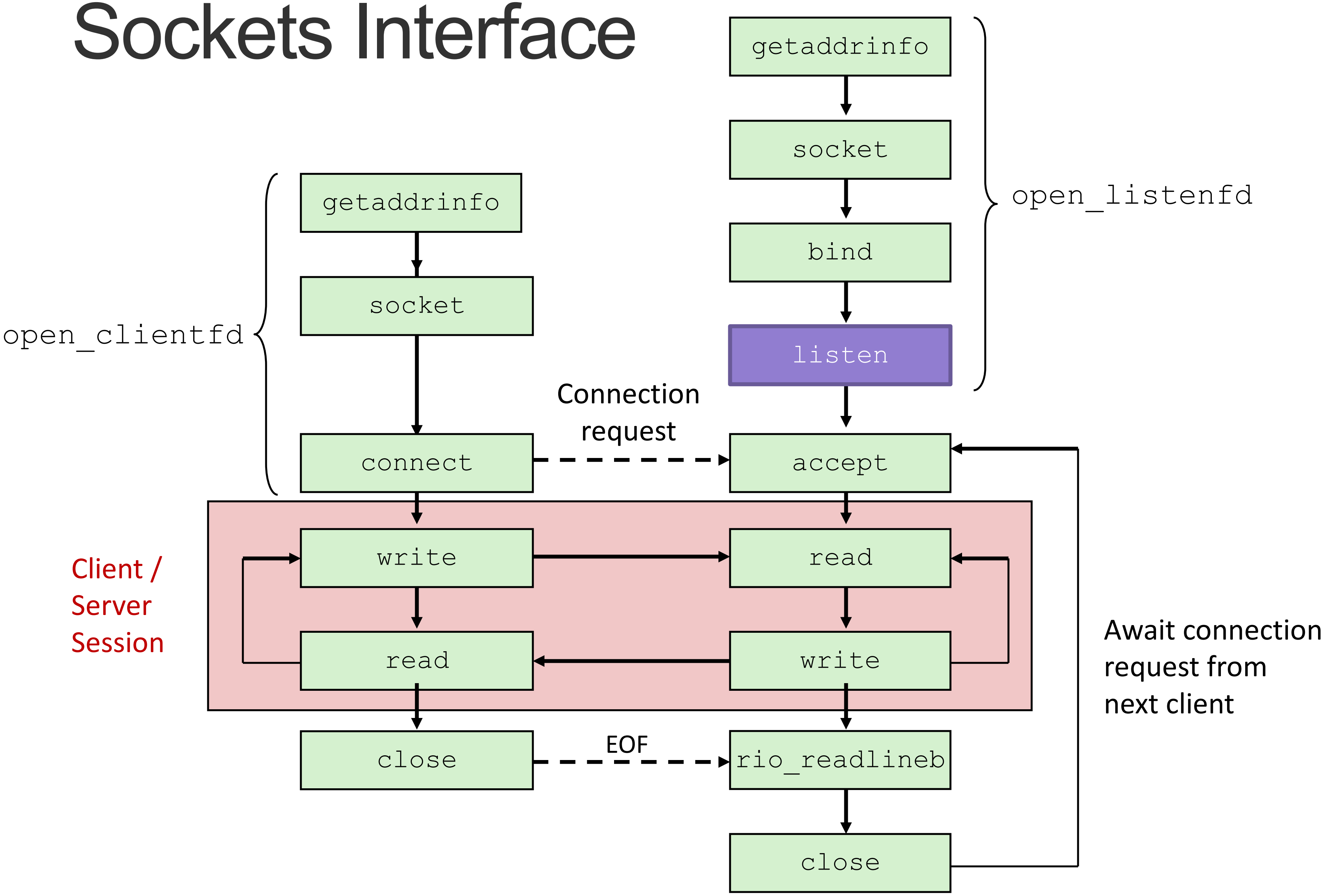
- A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

- The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.
- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

Sockets Interface



Sockets Interface: `listen`

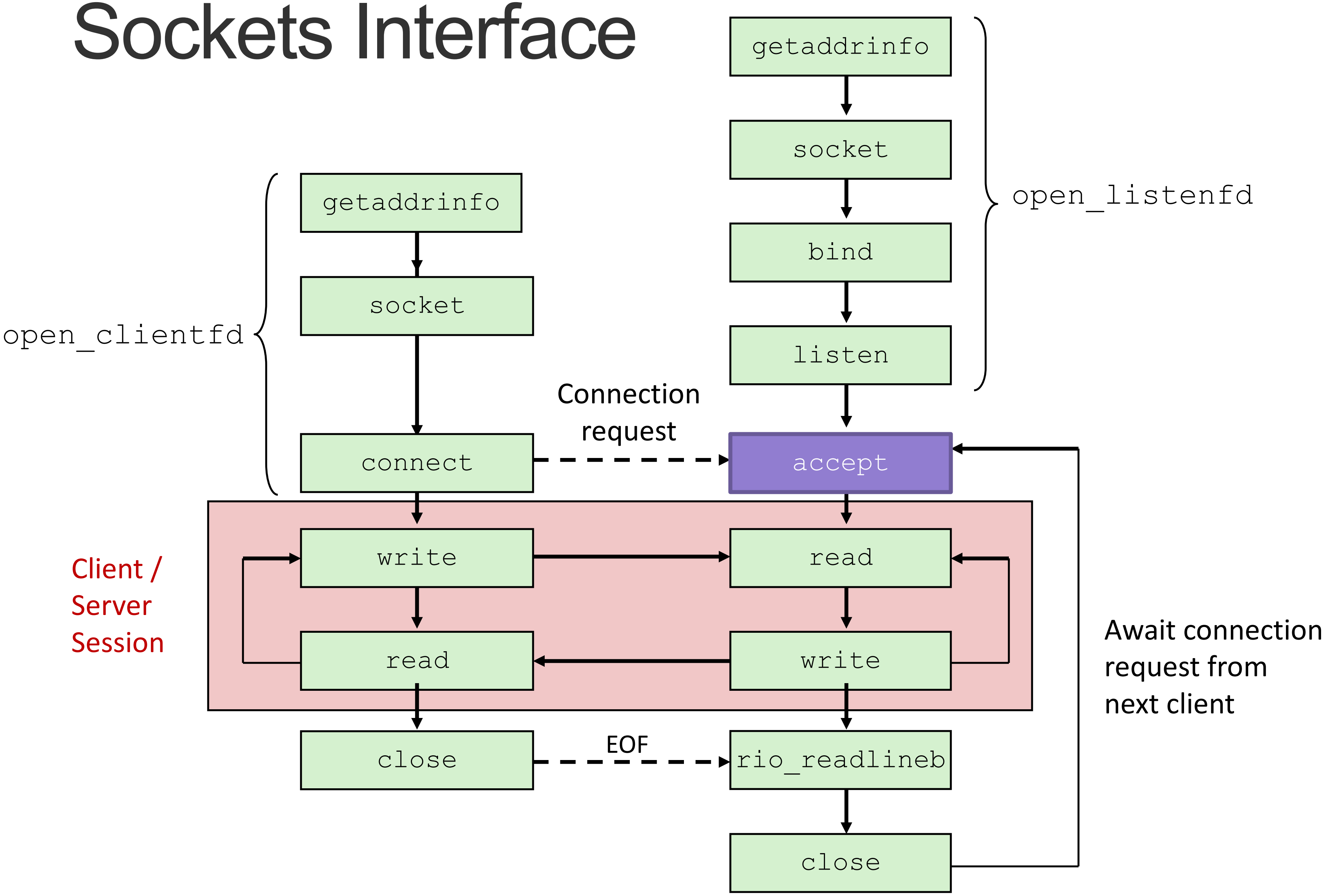
- By default, kernel assumes that descriptor from socket function is an *active socket* that will be on the client end of a connection.
- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

```
int listen(int sockfd, int backlog);
```

- Converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients.

- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.

Sockets Interface



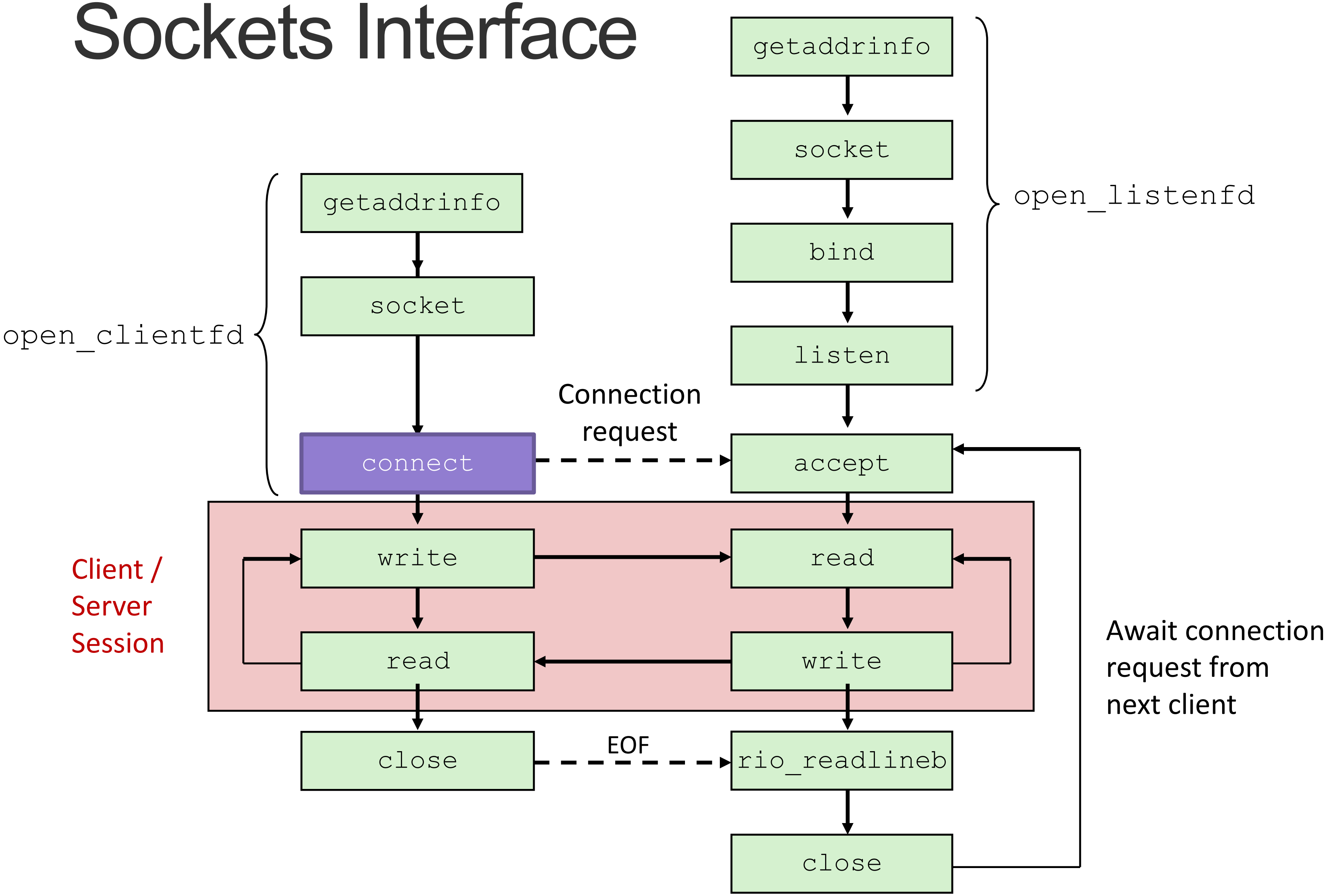
Sockets Interface: `accept`

- Servers wait for connection requests from clients by calling `accept`:

```
int accept(int listenfd, SA *addr, int *addrlen);
```

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`.
- Returns a *connected descriptor* that can be used to communicate with the client via Unix I/O routines.

Sockets Interface



Sockets Interface: `connect`

- A client establishes a connection with a server by calling connect:

```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

- Attempts to establish a connection with server at socket address `addr`
  - If successful, then `clientfd` is now ready for reading and writing.
  - Resulting connection is characterized by socket pair

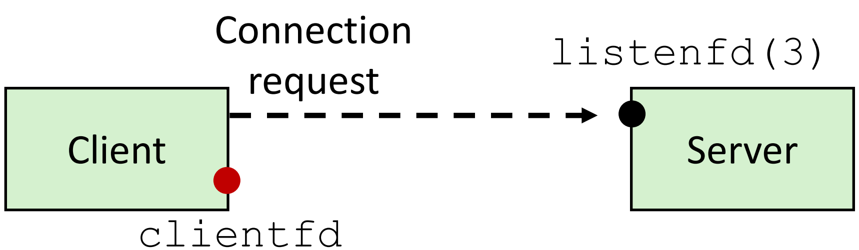
    `(x:y, addr.sin_addr:addr.sin_port)`

    - `x` is client address
    - `y` is ephemeral port that uniquely identifies client process on client host

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

`accept` Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



2. Client makes connection request by calling and blocking in `connect`

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

Connected vs. Listening Descriptors

- Listening descriptor
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- Connected descriptor
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

- Why the distinction?
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request

Exercise: Concurrent Connections

```
int main(int argc, char **argv){
   int listenfd, connfd;
   socklen_t clientlen;
   struct sockaddr_storage clientaddr;
   char client_hostname[MAXLINE], client_port[MAXLINE];

   listenfd = Open_listenfd(argv[1]);
   while (1) {
      clientlen = sizeof(struct sockaddr_storage);
      connfd = Accept(listenfd, clientaddr, &clientlen);
      Getnameinfo(&clientaddr, clientlen, client_hostname, MAXLINE,
                  client_port, MAXLINE, 0);
      printf("Connected to (%s, %s)\n", client_hostname, client_port);

      echo(connfd);
      Close(connfd);
    }
    return 0;
}
```