Introduction to Networked, Mobile and Wireless Systems

COS 518: *Advanced Computer Systems*
Lecture 5

Kyle Jamieson

[Credits: Selected content adapted from M. Freedman, I. Ganichev, B. Karp]

Today

1. **The UNIX Socket API**

2. Remote Procedure Call

3. Introduction to Wireless Communications

UNIX Socket API

- Socket interface
  - **Network programming interface for applications**
  - Originally provided in Berkeley UNIX
    - Later adopted by all popular operating systems
  - Simplifies porting applications to different OSes

- In UNIX, (almost) everything is like a file
  - All input is like reading a file, output like writing
  - File is represented by an integer file descriptor

- API implemented as system calls to the kernel
  - *e.g.*, connect, read, write, close, …

Sockets

- Endpoint of a connection
  - Identified by IP address and Port number

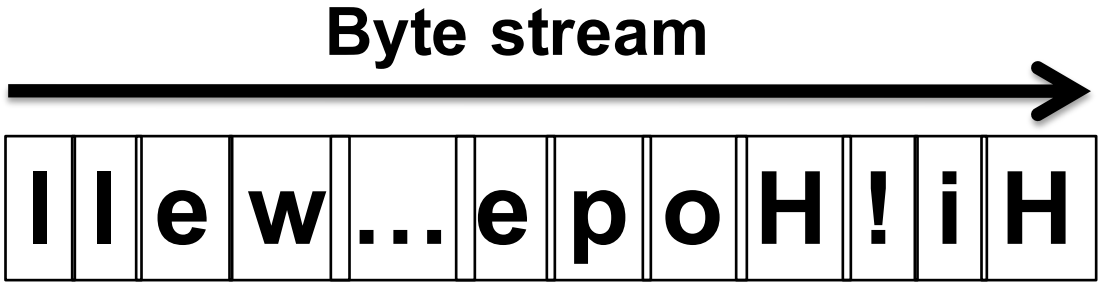
- Primitive to implement high-level networking interfaces
  - e.g., Remote procedure call (RPC)

1. ***Stream sockets*** (TCP sockets): HTTP, SSH, FTP, *e.g.*
- Connection-oriented (require establishment, teardown)
- Reliable, in-order, exactly-once delivery of **data stream**

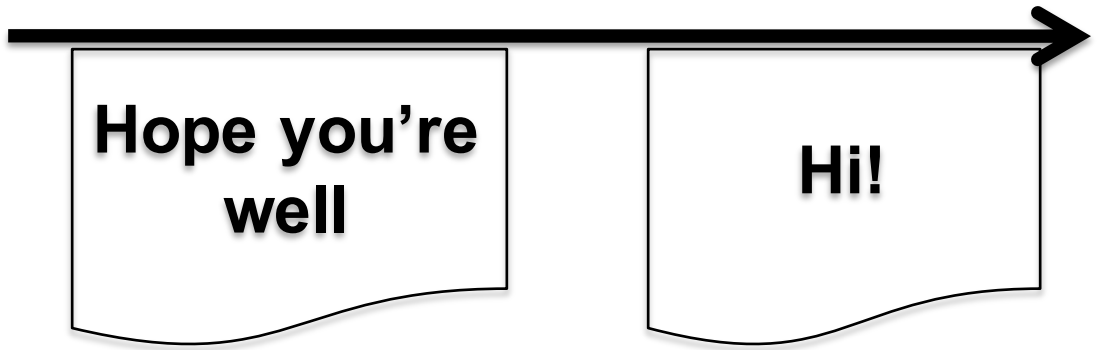
2. ***Datagram sockets*** (UCP sockets): DNS, VoIP, *e.g.*
- Best-effort, connection-less delivery of **datagrams**

Streams versus datagrams (1/2)

- When sending `"Hi!"` and `"Hope you're well"`
- TCP treats them as a single byte stream:



- UDP treats them as separate messages:

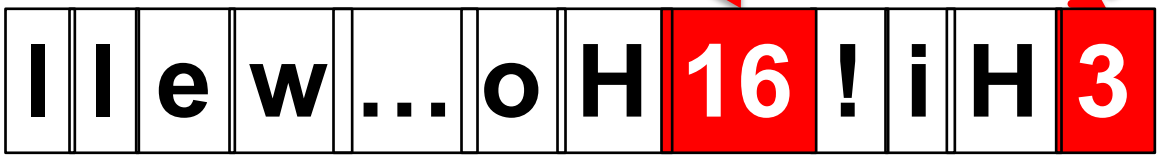


Streams versus datagrams (2/2)

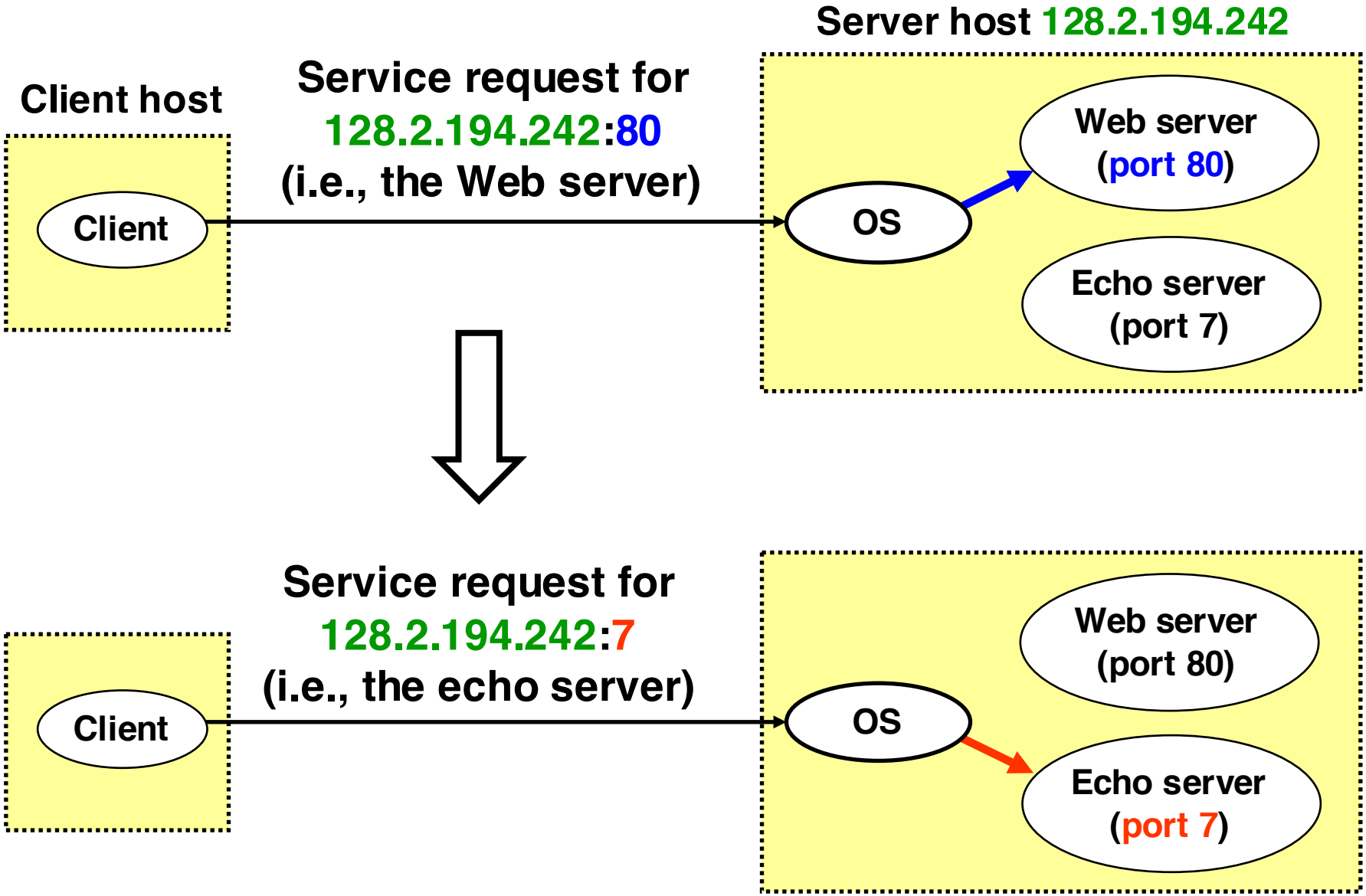
- Thus, TCP needs application-level message boundary.
  - By carrying **length** in application-level header, *e.g.*

```
struct my_app_hdr {
        int length
}
```

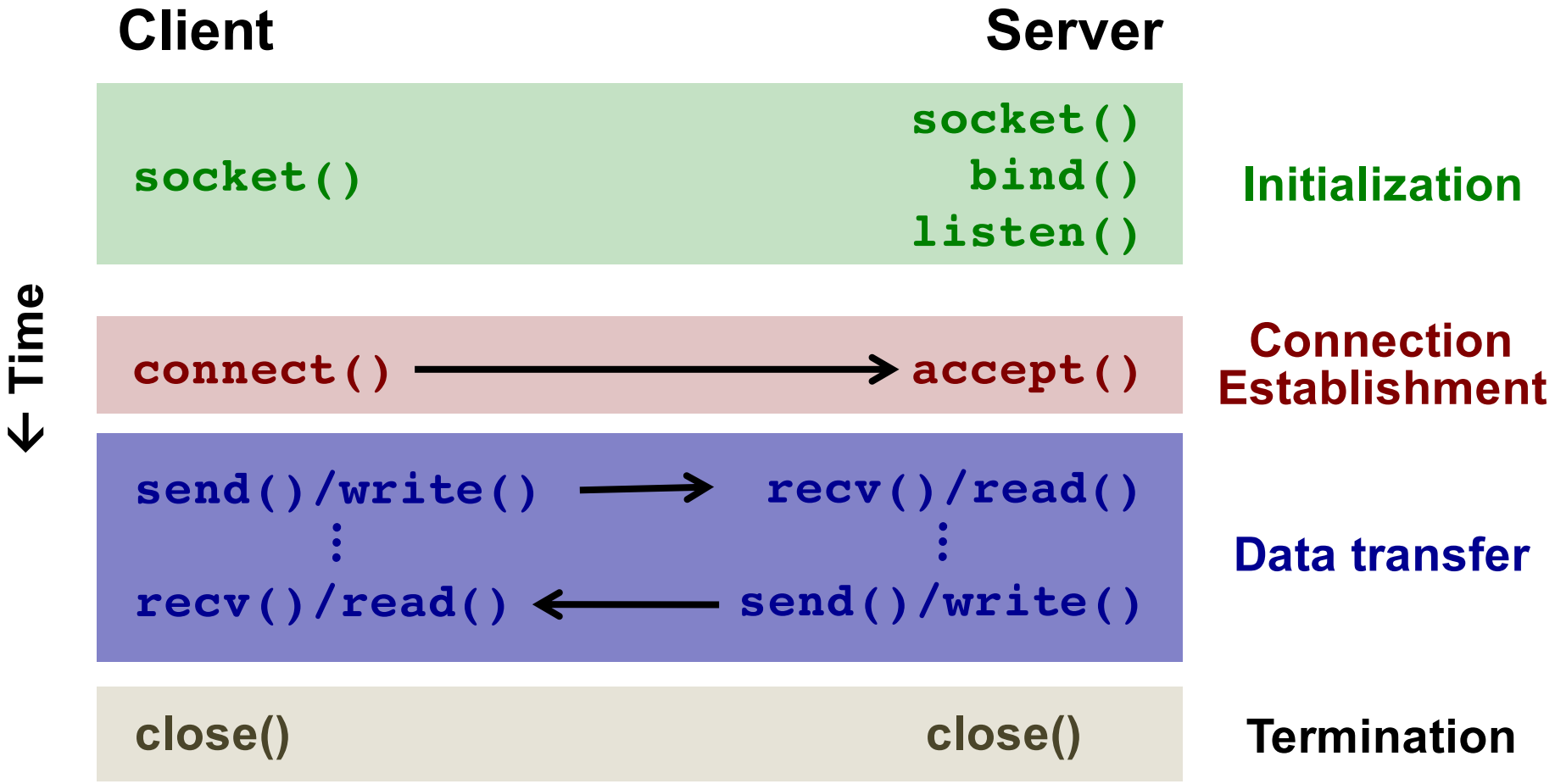
Bytes stream



IP address and port number identify endpoints



Scenario #1: Client-server reliable stream



Initialization at server and client: socket()

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
  perror("Couldn't create socket");
  abort();
}
```

- **socket(…): returns a socket descriptor**
- **AF_INET: IPv4 address family.** (also OK with PF_INET)
  - Compare IPv6 → `AF_INET6`
- **SOCK_STREAM: streaming socket type**
  - Compare datagram socket → `SOCK_DGRAM`
- **perror()**: prints out an error message

Error codes in Unix programming

- `extern int errno;  // by #include <errno.h>`
- Many Unix system calls and library functions set errno on errors
- Macros for error codes ('E' + error name)
  - EINTR, EWOULDBLOCK, EINVAL, …
  - `% man [function name]` shows possible error codes for the function name
- Functions to convert error code into human readable messages
  - `void perror(const char *my_str)`
    - Always looks for `errno`
    - prints out "my str: error code string"
  - `const char *strerror(int err_code)`
    - You must provide an error code
    - returns a string for the `err_code`

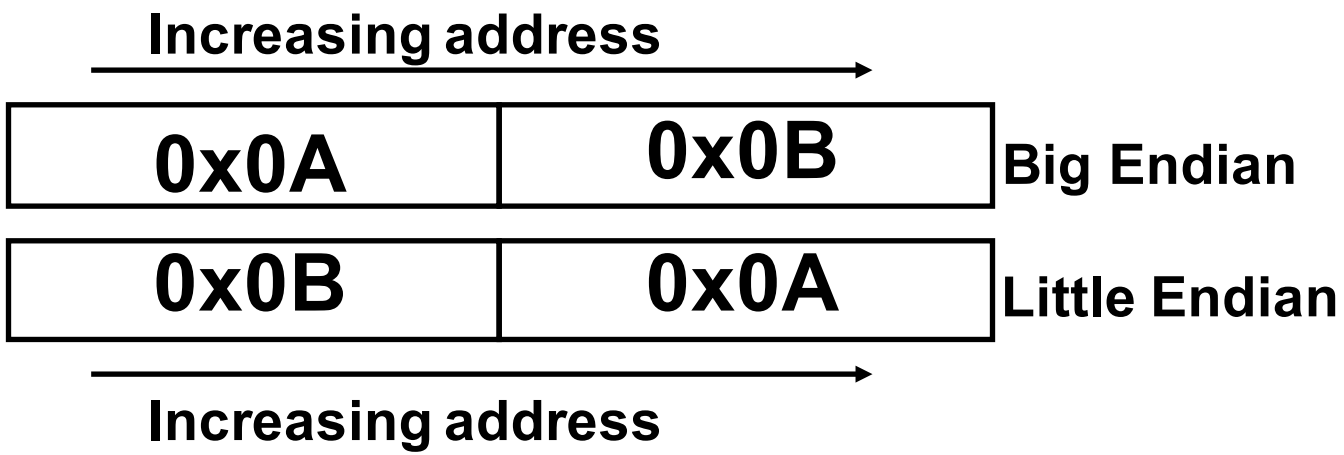
Initialization at the server: `bind()`

```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(server_port);

if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
   perror("bind failed");
   abort();
}
```

- **`bind`**: binds a socket with a particular port number.
  - Kernel remembers which process has bound which port(s)
  - Only one process can bind a particular port number at a time
- **`struct sockaddr_in`**: IPv4 socket address structure
- **`INADDR_ANY:`** If server has multiple IP addresses, binds any address
- **`htons(…)`**: converts host byte order into network byte order

Endianness

- *You have a 16-bit number: 0x0A0B. How is it stored in memory?*



- ***Host endianness (byte order)*** is not uniform
  - Some machines are big endian, others are little endian

- Communicating between machines with different host byte orders is problematic
  - Transferred 256 (0x0100), but received 1 (0x0001)

Endian-ness (cont'd)

- **Convention:** The network byte order is **big endian**
  - To avoid the endian problem, we must use network byte order when sending 16-, 32-, 64-bit numbers

- Utility functions for easy conversion:

```
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
```

- Hint: **h**, **n**, **s**, and **l** stand for host byte order, network byte order, short(16 bit), and long(32 bit), respectively

Reusing the same port

- After TCP connection closes, waits for twice maximum segment lifetime (from 1 to 4 mins, implementation dependent). Why?
  - By default, port number cannot be reused before two × MSL
  - But server port numbers are fixed, so must be reused
    - Solution: Server executes this code **before** `bind(…)`:

```
int optval = 1;
if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
  &optval, sizeof(optval)) < 0) {
  perror("reuse failed");
  abort();
}
```

- **setsockopt(…):** changes socket, protocol options
  - *e.g.*, buffer size, timeout value, …

Initialization at the server: listen()

- Specify a willingness to accept incoming connections and a queue limit for incoming connections

```
if (listen(sock, back_log) < 0) {
  perror("listen failed");
  abort();
}
```

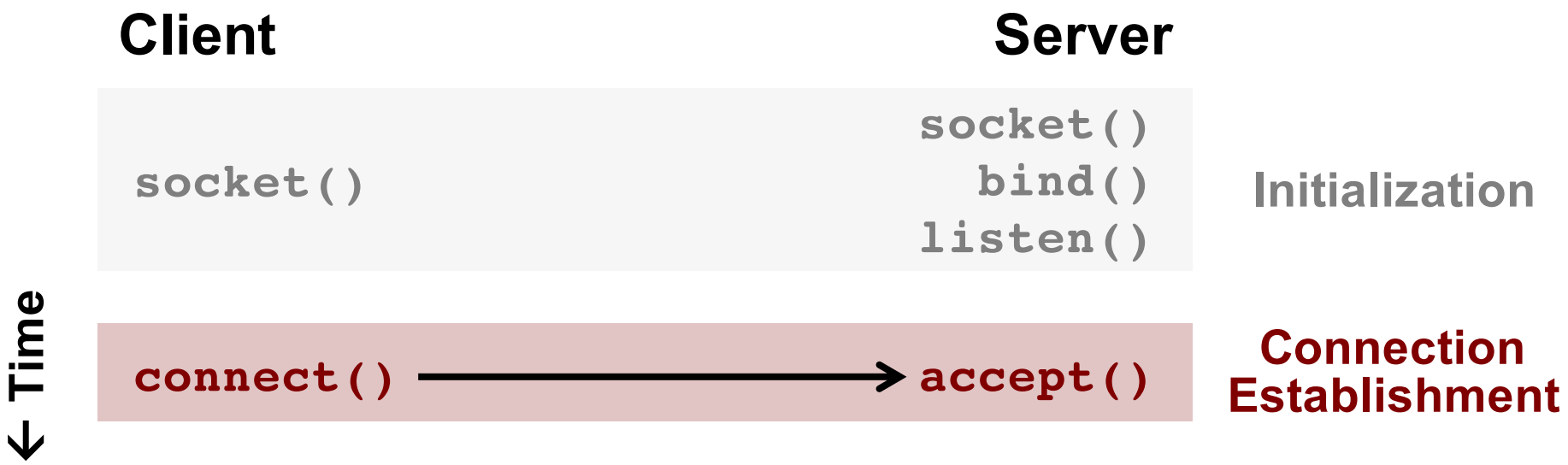
- **`listen(…):`** converts an active socket to passive

- **back_log**: connection-waiting queue size. (*e.g.*, 32)
  - Busy server may need a large value (*e.g.*, 1024, …)

Initialization: Summary

- Client
  - **`socket()`**
- Server
  - **`socket(…)`**
  - **`setsockopt(sock, SOL_SOCKET, SO_REUSEADDR)`**
  - **`bind(…)`**
  - **`listen(…)`**
- Pitfalls:
  - The order of the functions matter
  - Don't forget to use **`htons(…)`** to handle port number

Scenario #1: Client-server reliable stream



Connection establishment at the client

```
struct sockaddr_in sin;
memset(&sin, 0 ,sizeof(sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = inet_addr("128.32.132.214");
sin.sin_port = htons(80);

if (connect(sock, (struct sockaddr *) &sin,
sizeof(sin)) < 0) {
   perror("connection failed");
   abort();
}
```

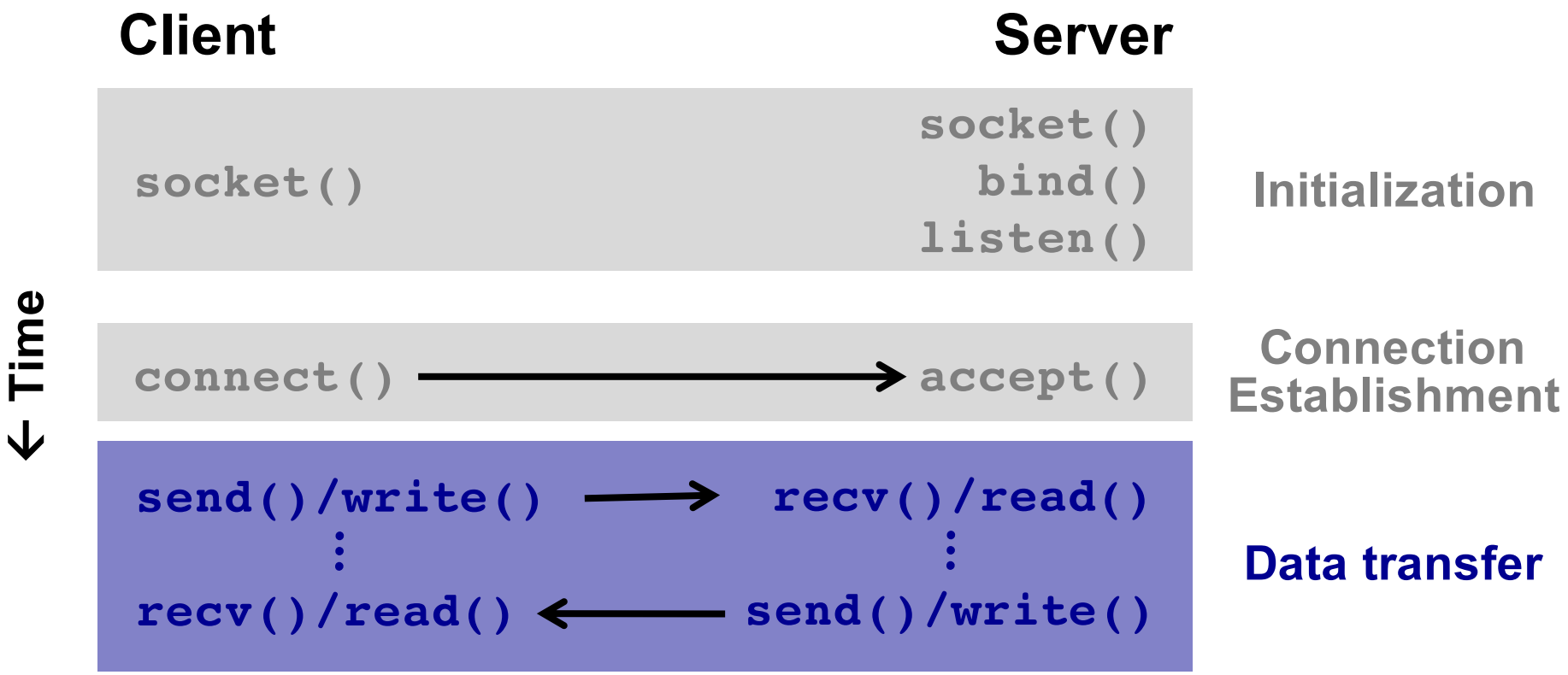
- **connect(…)**: waits until connection establishes/fails
- **inet_addr()**: converts an IP address string into a 32-bit address number (network byte order).

Connection establishment at the server

```
struct sockaddr_in client_sin;
int addr_len = sizeof(client_sin);
int client_sock = accept(listening_sock,
                         (struct sockaddr *) &client_sin,
                         &addr_len);
if (client_sock < 0) {
   perror("accept failed");
   abort();
}
```

- **`accept()` blocks** until a connection is present
- **`accept()`** returns a new socket descriptor for a client connection in the connection-waiting queue
  - This socket descriptor is to communicate with the client
    - `listening_sock` is not to communicate with a client
- **`client_sin`**: contains client IP address and port number
  - Q: Are they big- or little-endian?

Scenario #1: Client-server reliable stream



Sending data (server or client)

```
char *data_addr = "hello, world";
int len = 12;

int sent_bytes = send(sock, data_addr, len, 0);
if (sent_bytes < 0) {
   perror("send failed");
}
```

- **`send()`**: sends data, returns the number of sent bytes
  - Also OK with **`write()`**, **`writev()`**
- **`data_addr`**: address of data to send
- **`len`**: size of the data

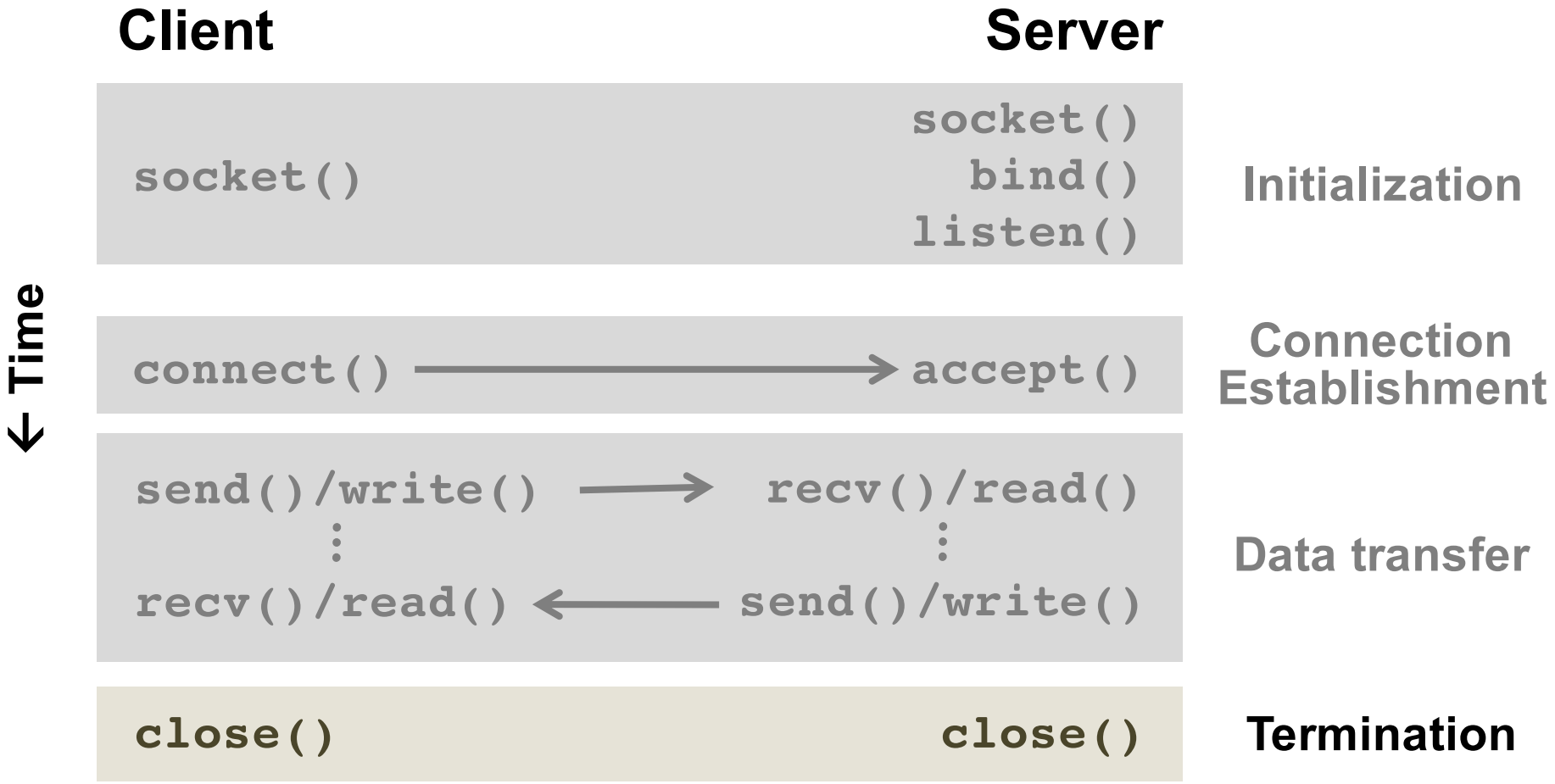
- With ***blocking*** sockets (default), send() blocks until sending data
- With ***non-blocking*** sockets, sent_bytes may not equal `len`
  - If kernel lacks space, accepts only partial data
  - **You** must retry for the unsent data

Receiving data (server or client)

```
char buf[4096];
int max_len = sizeof(buf);

int read_len = recv(sock, buf, max_len, 0);
if (read_len == 0) {  // connection is closed
} else if (read_len < 0) { // error
   perror("recv failed");
} else {  // OK, but no guarantee read_len == max_len
}
```

- **`recv()`**: Reads bytes from **`sock`**, returns count of bytes read
  - read_bytes may not equal to expected_data_len
    - If no data is available, `recv` blocks
    - If only partial data is available, `read_len < max_len`
  - If you get only partial data, you should retry for the remainder

Scenario #1: Client-server reliable stream

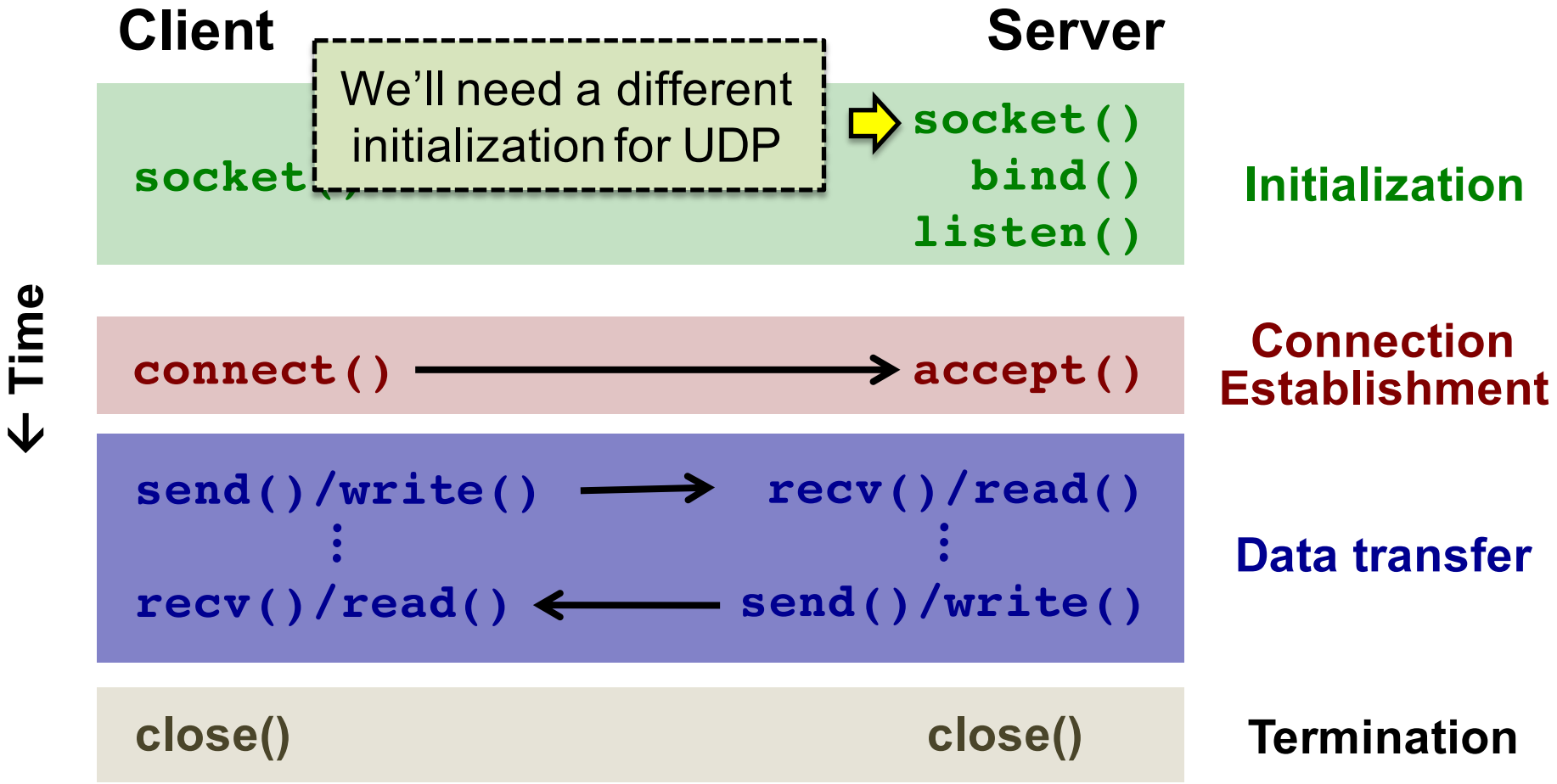


Termination at the server and client

- `close()`: closes the socket descriptor

- We cannot open files/sockets more than 1024*
  - We must release the resource after use
  - Super user can overcome this constraint, but regular user cannot

Scenario #2: Client-server datagrams

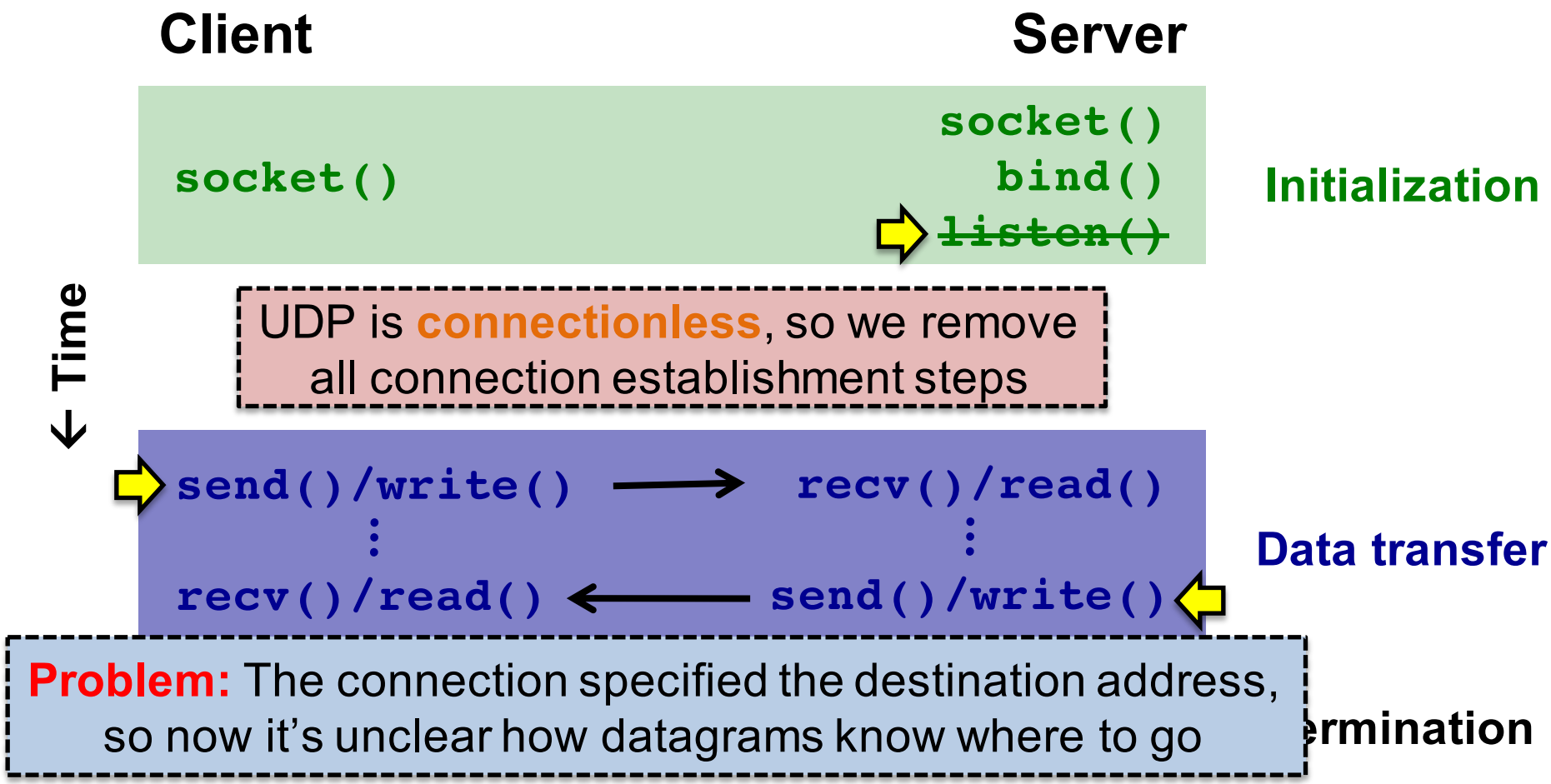


Initialization: UDP

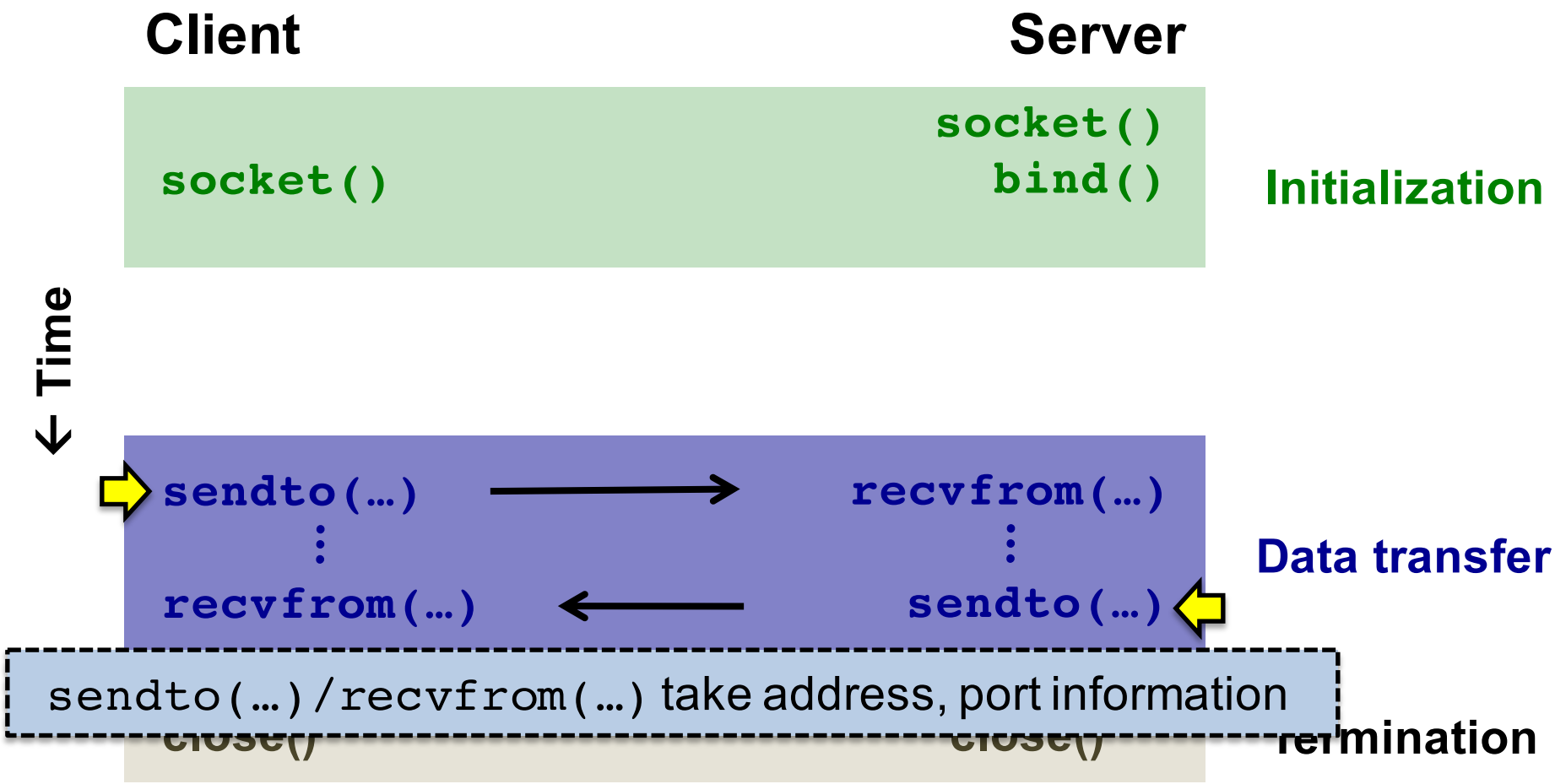
```
int sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock < 0) {
      perror("socket failed");
      abort();
}
```

- UDP uses **SOCK_DGRAM** instead of **SOCK_STREAM**

Scenario #2: Client-server datagrams



Scenario #2: Client-server datagrams



UDP send: `sendto(…)`

```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = inet_addr("128.32.132.214");
sin.sin_port = htons(1234);

sent_bytes = sendto(sock, data, data_len, 0,
                    (struct sockaddr *) &sin, sizeof(sin));
if (sent_bytes < 0) {
    perror("sendto failed");
    abort();
}
```

- **`sendto(…)`**: sends datagram to destination **address**, **port**
  - *cf.*, in TCP, we set destination when calling **`connect()`**
- As opposed to TCP, UDP packetizes data
  - So, `sendto(…)` either sends all the data or none

UDP receive: `recvfrom(…)`

```
struct sockaddr_in sin;
int sin_len;
char buffer[4096];

int read_bytes = recvfrom(sock, buffer, sizeof(buffer), 0,
                          (struct sockaddr *)&sin,
                          &sin_len);
if (read_bytes < 0) {
   perror("recvfrom failed");
   abort();
}
```

- **`recvfrom()`**: reads datagram, sets source information
- Reading 0 bytes does not mean "connection closed" unlike TCP.
  - Recall UDP does not have a notion of "connection".

API functions summary

TCP

- Initialization
  - socket(AF_INET, SOCK_STREAM, 0)
  - setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, …)
  - bind()
  - listen()
- Connection
  - connect()
  - accept()
- Data transfer
  - send()
  - recv()
- Termination
  - close()

UDP

- Initialization
  - socket(AF_INET, SOCK_DGRAM, 0)
  - setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, …)
  - bind()
- No connection
- Data transfer
  - sendto()
  - recvfrom()
- Termination
  - close()

How to handle multiple inputs?

- Many data sources, many sockets
  - *e.g.*, busy web server, busy ssh server

- **Problem: asynchronous data arrival**
  - Program does not know when data will arrive on any socket!
  - If no data available, `recv(…)` blocks.
    - If blocked on one source, cannot handle other sources
    - *e.g.* a web server could not handle multiple connections

- Solutions
  - Polling using non-blocking socket → **Inefficient**
  - I/O multiplexing using select() → **Simple, performance(?)**
  - Multithreading → more complex, not covered today

Polling using non-blocking socket

- This approach **wastes CPU cycles**

```
// Fetch existing socket options
int opt = fcntl(sock, F_GETFL);
if (opt < 0) {
        perror("fcntl failed");
        abort();
}

// Update socket option with non-blocking option
if (fcntl(sock, F_SETFL, opt | O_NONBLOCK) < 0) {
        perror("fcntl failed");
        abort();
}

while (1) {
        int read_bytes = recv(sock, buffer, sizeof(buffer), 0);
        if (read_bytes < 0) {
                if (errno == EWOULDBLOCK) {
                        // OK: Simply no data available now
                } else {
                        perror("recv failed");
                        abort();
                }
        } else { // read data here
        }
}
```

I/O multiplexing using `select(...)`

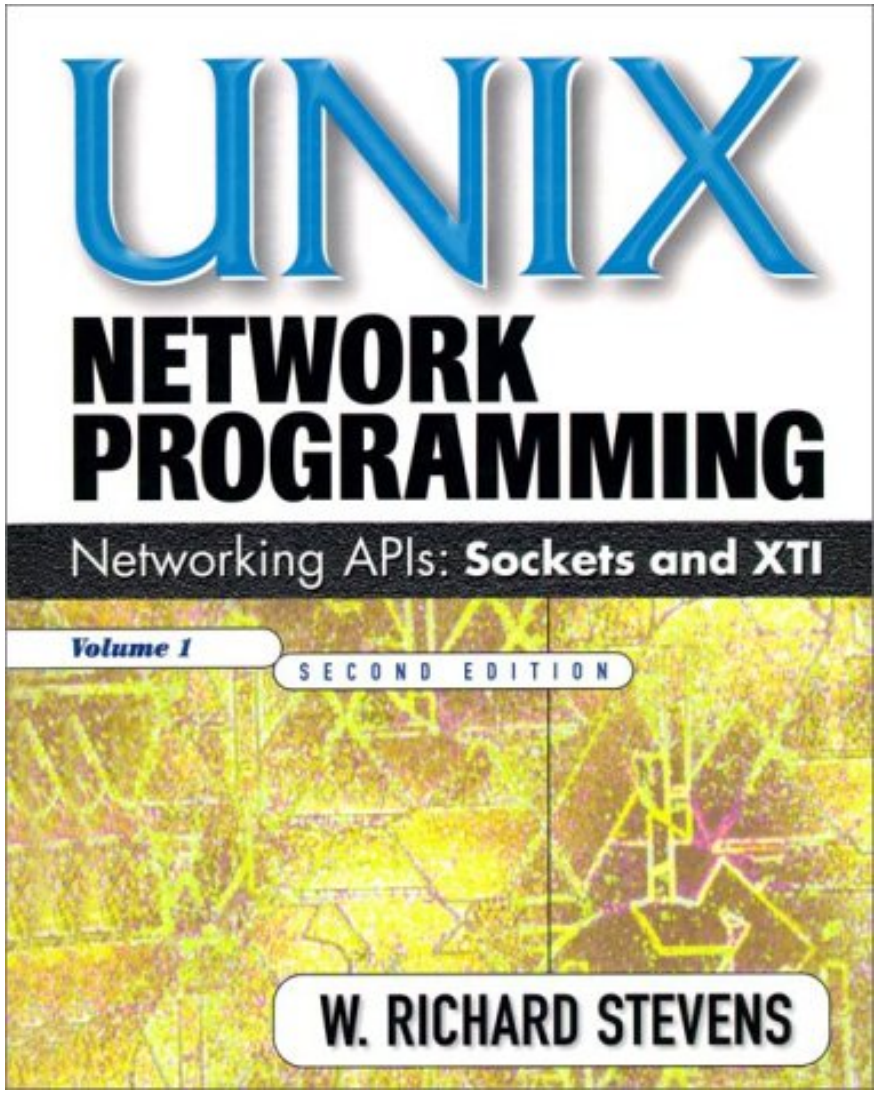
```
fd_set  read_set;
struct  timeval  timeout

FD_ZERO(&read_set);
FD_SET(sock1,  &read_set);
FD_SET(sock2,  &read_set);
timeout.tv_sec  = 0;
timeout.tv_usec  = 5000;

while  (1)  {
   // the following blocks until >= 1 fd is ready for reading
   if  (select(MAX(sock1,  sock2)  + 1,  &read_set,  NULL,
                NULL,  &time_out)  < 0)  {
    perror("select  failed");
    abort();
   }

   if  (FD_ISSET(sock1,  &read_set))  {
    // sock1  has  data
   }
   if  (FD_ISSET(sock2,  &read_set))  {
    // sock2  has  data
   }
}
```

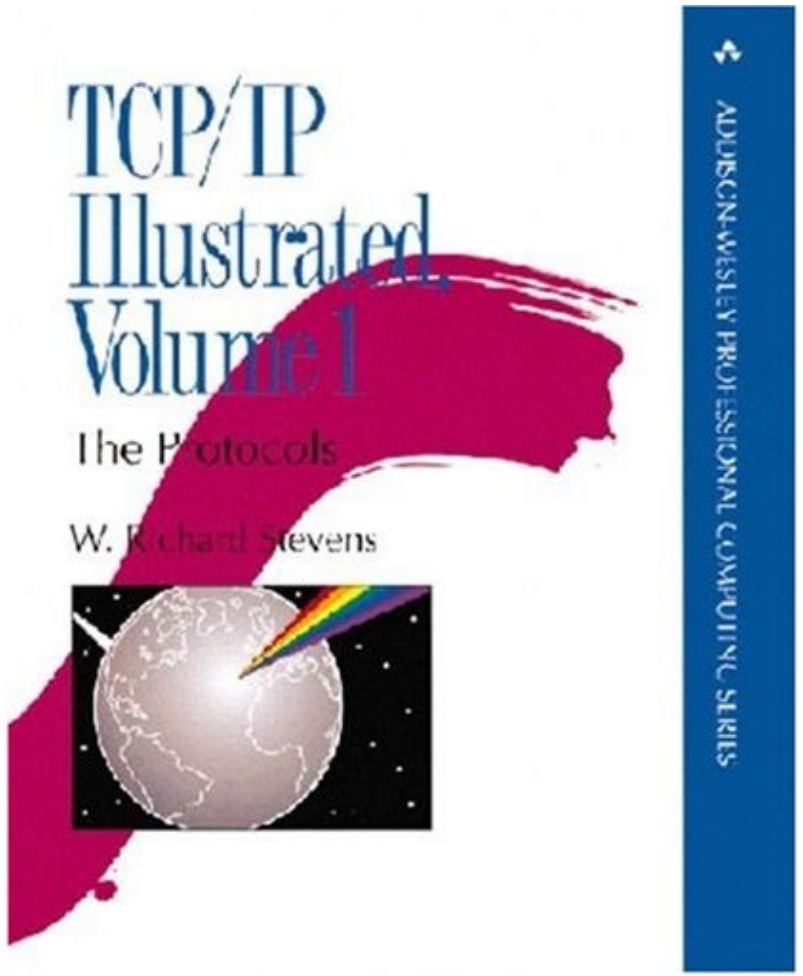
Initializes arguments for select()

Pass `NULL` instead of `&timeout` if you want to wait indefinitely

Check for I/O events and act accordingly

For further information





Today

1. The UNIX Socket API

2. **Remote Procedure Call**

3. Introduction to Wireless Communications

Straw man: Writing a distributed application with sockets

Client

```
def main:
    balance = acctBalance(id)
    print balance

def acctBalanceStub:
    msg ← id
    send request
    (wait for reply)
    balance ← reply
    return balance
```

Server

```
def acctBalance(id)
    …
    return balance

def acctBalanceStub
    wait for request
    id ← request
    balance = acctBalance(id)
    reply ← balance
    (send reply)
```

Remote Procedure Call (RPC)

- Within a single program, running in a single process, there is a well-known notion of **function call**:
  - ***Caller*** pushes arguments onto stack,
  - jumps to address of ***callee*** function
  - **Callee** reads arguments from stack,
  - executes, puts return value in register,
  - returns to next instruction in caller

- RPC aims to make communication in a **distributed** system appear just like **a local procedure call**

The RPC abstraction

- Library makes an API available to locally running applications
- Let servers **export** their local APIs to be accessible over the network, as well
- On the **client**, procedure call generates request over network to server
- On the **server**, the called procedure executes, result returned in response to client

Implementing RPC

- Data **types** may be **different sizes** on different machines (*e.g.*, 32-bit vs. 64-bit integers)
- Data may have **different endian-ness** on different machines
- Need mechanism to pass procedure parameters and return values in **machine-independent fashion**
- Solution: **Interface Description Language (IDL)**

Interface Description Languages

- Compile interface description, produces:
  - Types in native language (*e.g.*, Java, C, C++)
  - Code to marshal native data types into machine-neutral byte streams for network (and vice-versa)
  - Stub routines on client to forward local procedure calls as requests to server
- For Sun RPC, IDL is ***XDR*** (eXternal Data Representation)

Defining a wire format with Sun XDR

proto.x:

```
acctBalanceFormat {

   request {
      u_int64_t id;
   };

   response {
      u_int32_t balance;
   };

} = 837501;
```

SUN XDR example

- Define API for procedure calls between client and server in XDR file, *e.g.*, **`proto.x`**

- Compile: **`rpcgen proto.x`**, producing:
  - **`proto.h`**: RPC procedure prototypes, argument and return value data structure definitions
  - **`proto_clnt.c`**: client stub code to send RPC requests to remote server
  - **proto_svc.c**: server stub code to receive and dispatch RPC request to specified procedure
  - **proto_xdr.c**: argument and result marshaling/ unmarshaling, host ↔ network byte order conversion

Modern RPC: Google protobufs

- Goals (per Jeff Dean):
  - Self-describing, multiple language support
  - Efficient to encode/decode, compact serialized form

```
message SearchResult {
   required int32 estimated_results = 1;
   optional string error_message = 2;
   repeated group Result = 3 {
       required float score = 4;
       required fixed64 docid = 5;
       optional message<WebResultDetails> = 6;
    }};
```

- Unique tags specify fields in message binary format
- Forward/backward compatibility: ignore new fields when parsing

Google Protobufs

- Automatically generated language wrappers
- Graceful client and server upgrades
  - Systems ignore tags they don't understand, but pass the information through (no need to upgrade intermediate servers)
- Serialization/deserialization
  - Lots of performance work here, benefits all users of protocol buffers, e.g., Snappy/Zippy compression
  - Format used to store data persistently (not just for RPCs): No need to convert between formats

Today

1. The UNIX Socket API

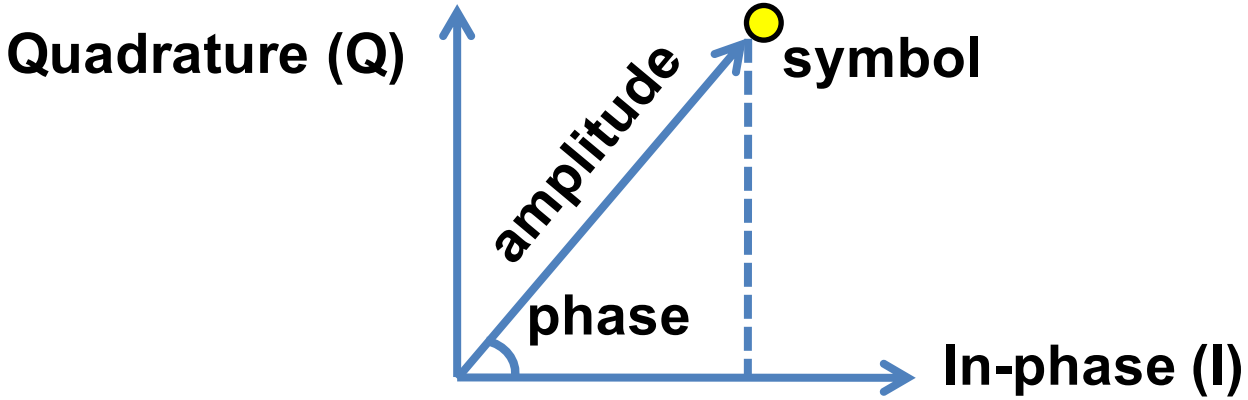
2. Remote Procedure Call

3. **Introduction to Wireless Communications**

   – “Sensor hints” paper

What is modulation?

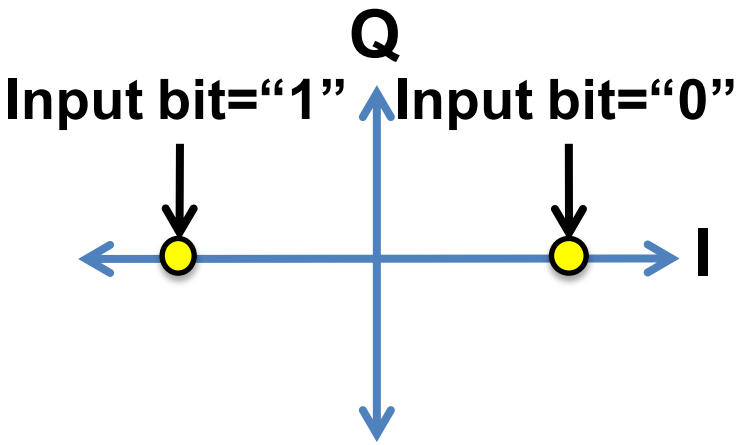
- To ***modulate*** means to **change.** Change what?
  - The **amplitude** and **phase (*i.e.*, angle)** of a ***carrier signal***
  - For 802.11 WiFi local area networks, this carrier signal is usually at 2.4 GHz or 5 GHz



- ***Digital modulation:*** Use only a finite set of choices (*i.e.*, ***symbols***) for how to change the carrier and phase
  - Transmitter and receiver agree upon the symbols beforehand

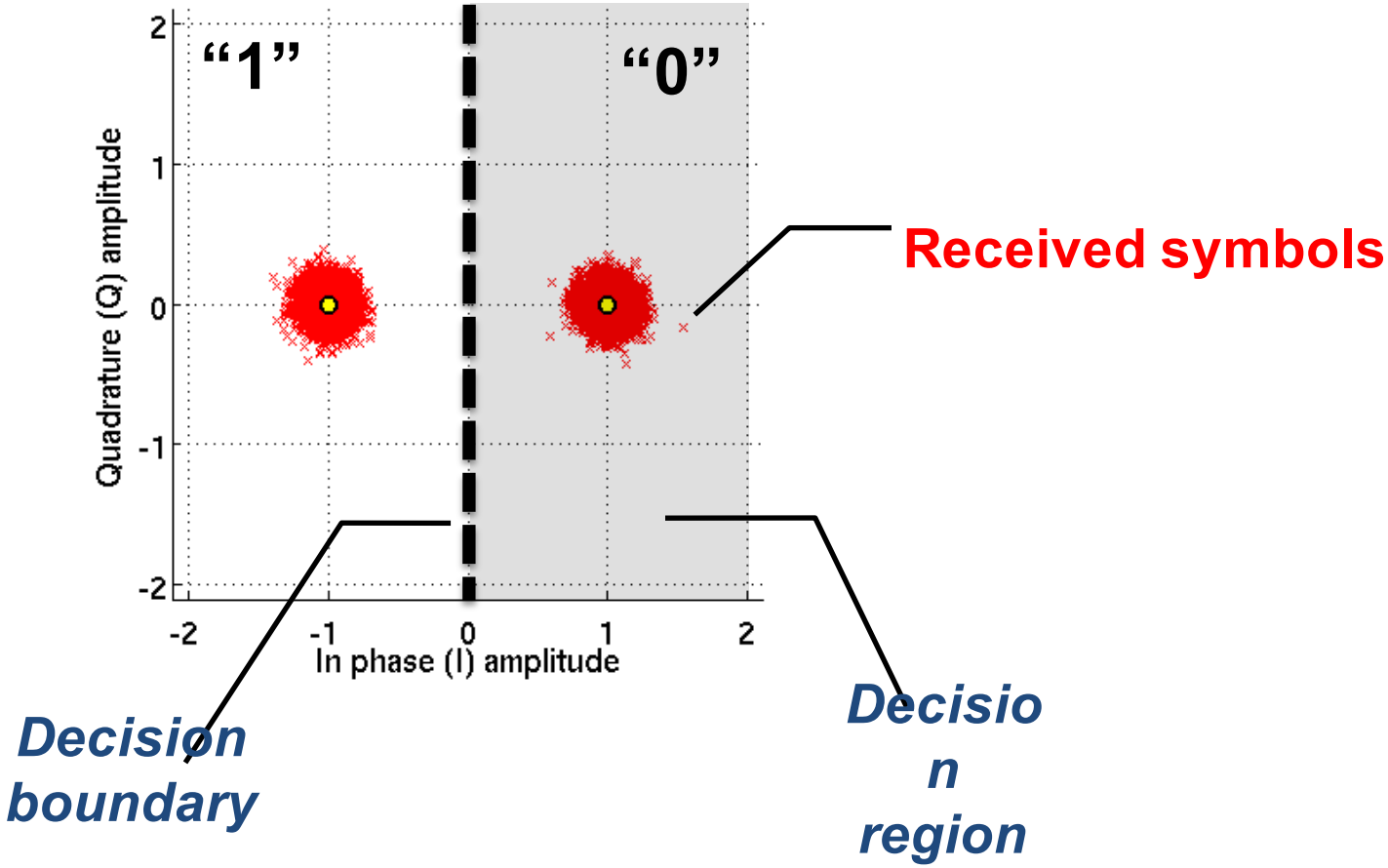
From information bits to symbols…

- Simplest possible scheme
  - Pick two symbols (**binary**)
  - The information bit decides which symbol you transmit
  - **Phase shift** of 180 degrees between the two symbols
  - So, this is called ***binary phase shift keying***
    - **Sending rate:** $\log_2 2 = 1$ bit/symbol



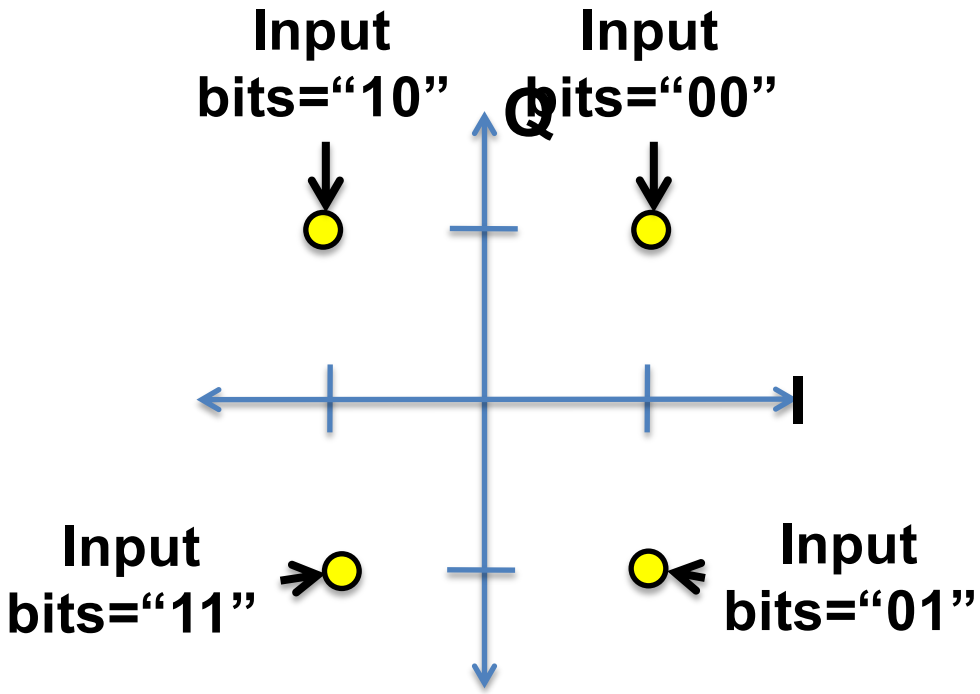
…and back to bits!

Received BPSK constellation



From information bits to symbols, twice as fast

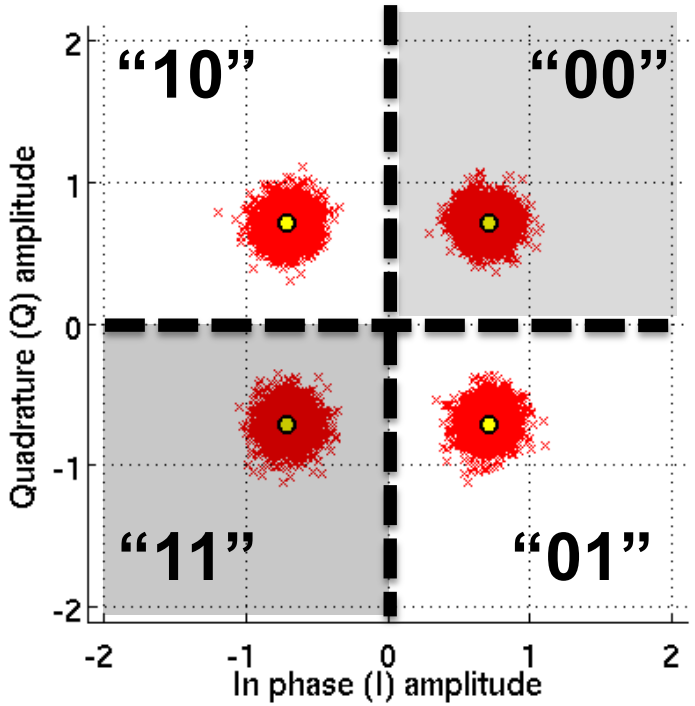
Quadrature phase shift keying (QPSK)



Sending $\log_2 4 = 2$ bits/symbol

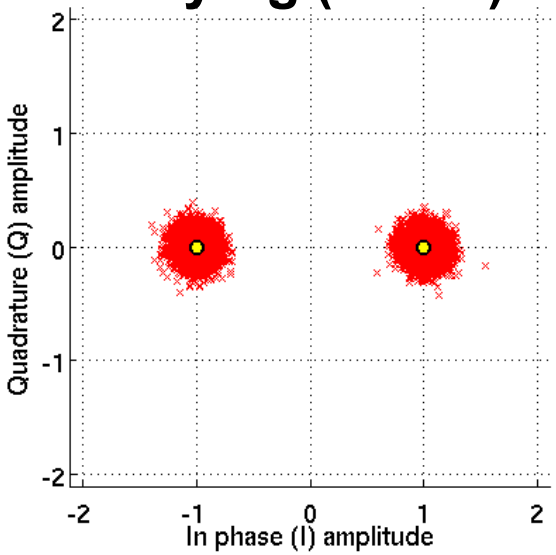
…and back to bits, twice as fast!

Received QPSK constellation

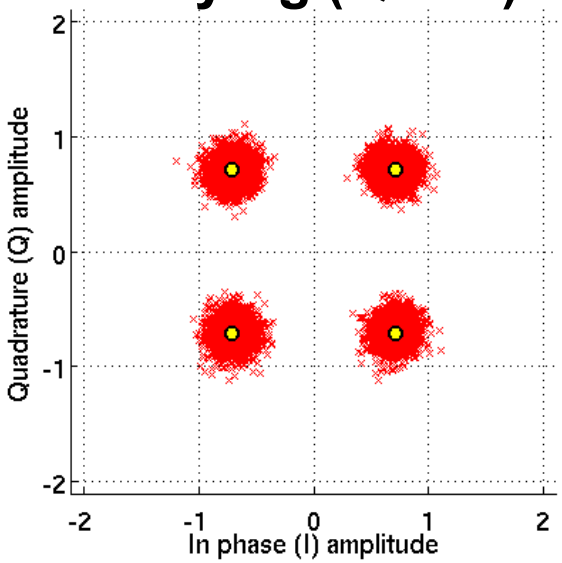


Change modulations, increase bitrate

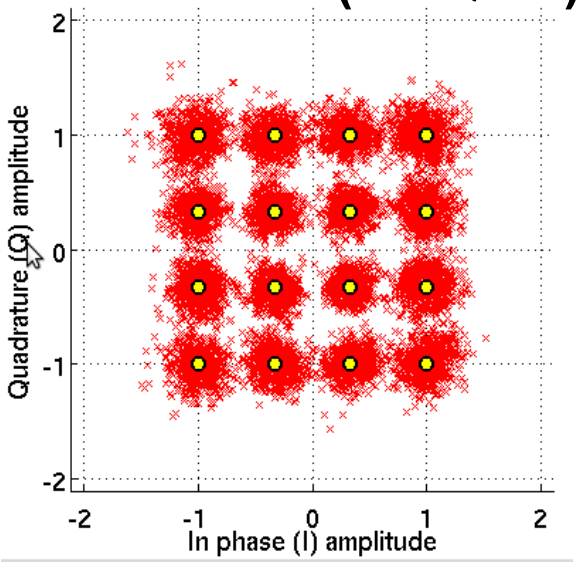
Binary Phase-Shift Keying (BPSK)



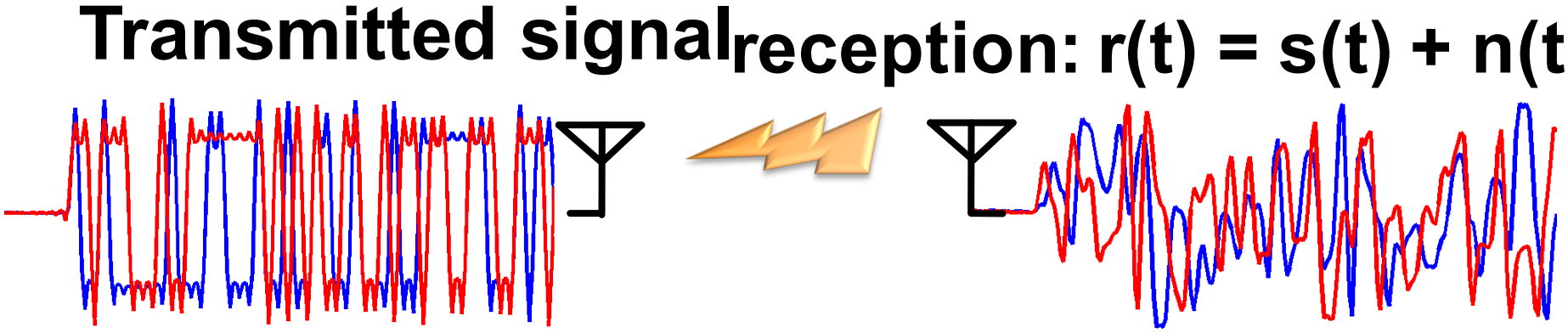
Quadrature Phase-Shift Keying (QPSK)

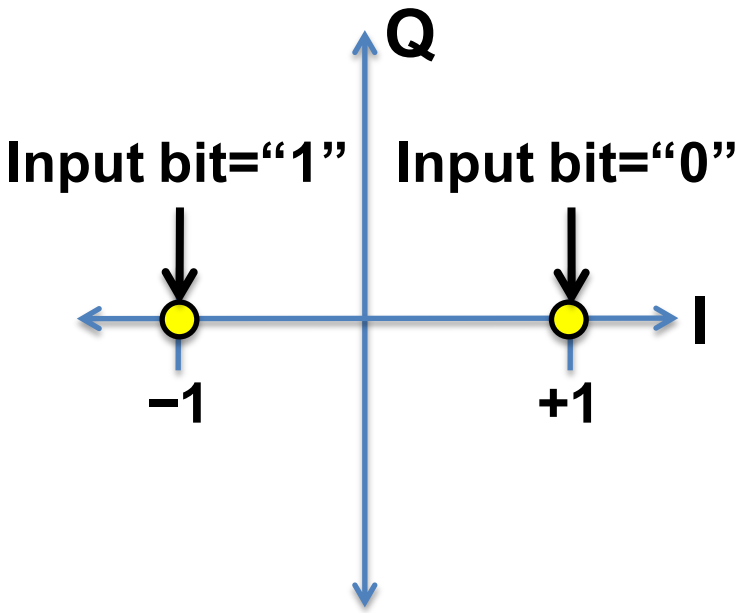


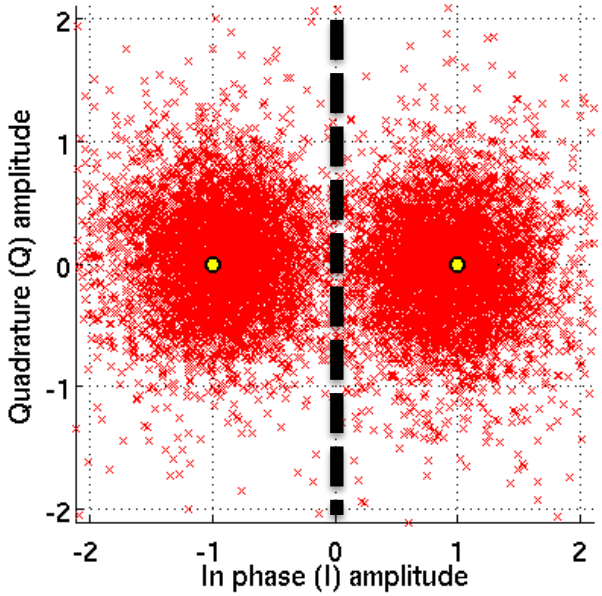
16-Quadrature Amplitude Modulation (16-QAM)



The wireless channel



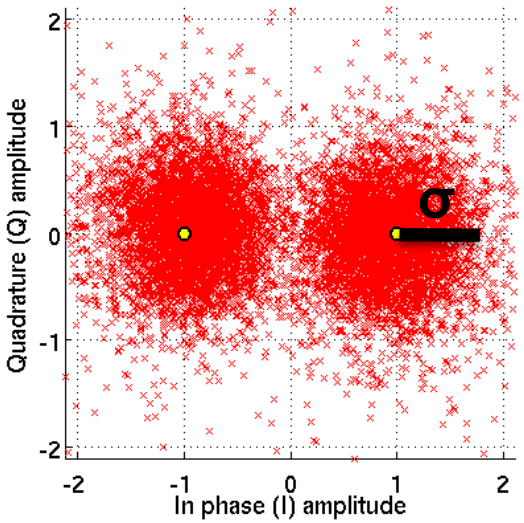




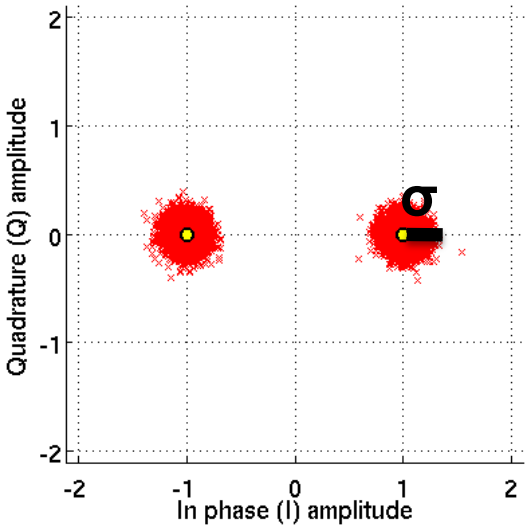
Signal to noise ratio (SNR)

- ***Signal to noise ratio:*** The ratio of the power of the signal to the power of the noise
  - Measured in ***decibels (dB):*** 10 times $\log_{10}$ of a quantity

SNR = 8 dB

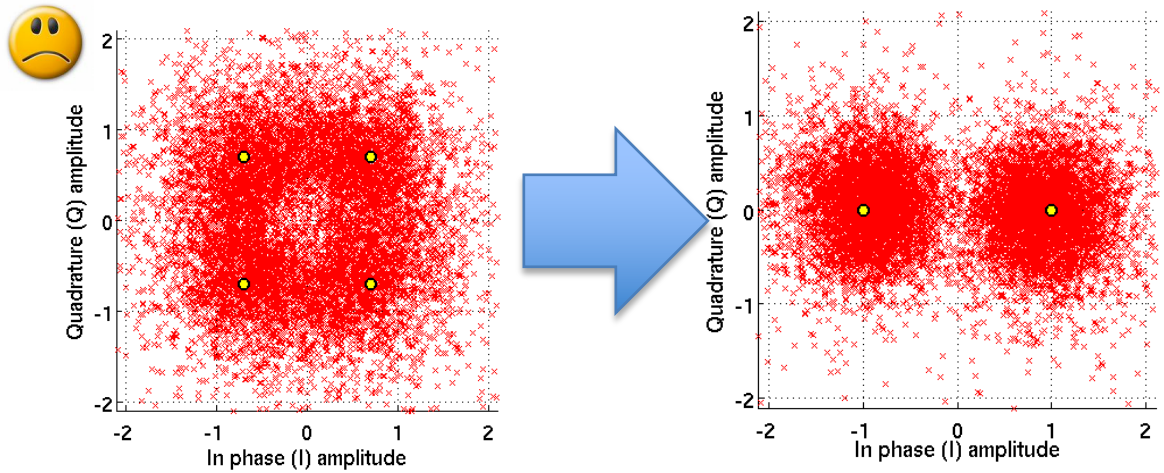


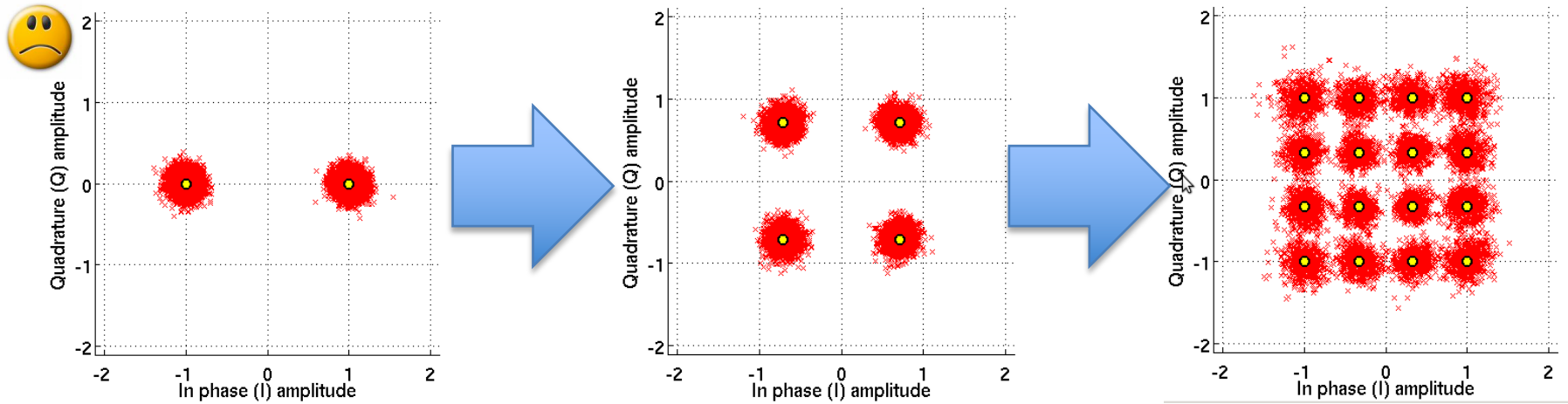
SNR = 20 dB



SNR (dB) = 10 $\log_{10}$ (signal power / noise power)
= 10 $\log_{10}$ (1^2 / σ^2) (assuming Gaussian noise)
= −20 $\log_{10}$ σ

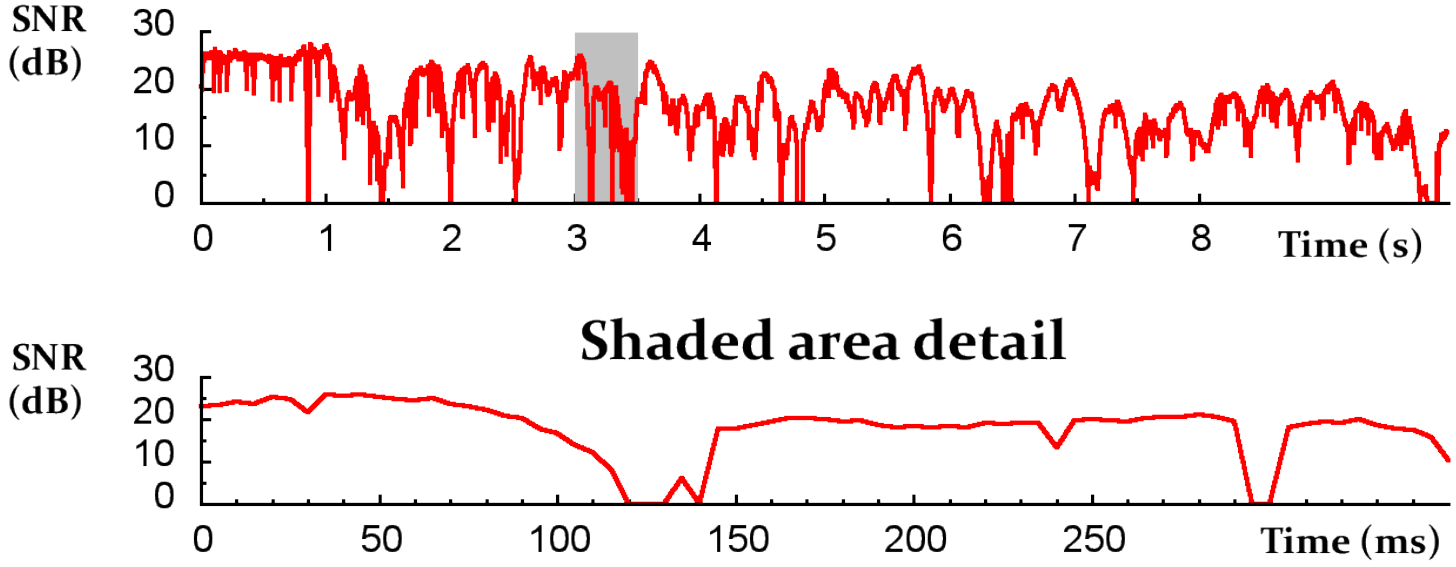
Modulation adaptation





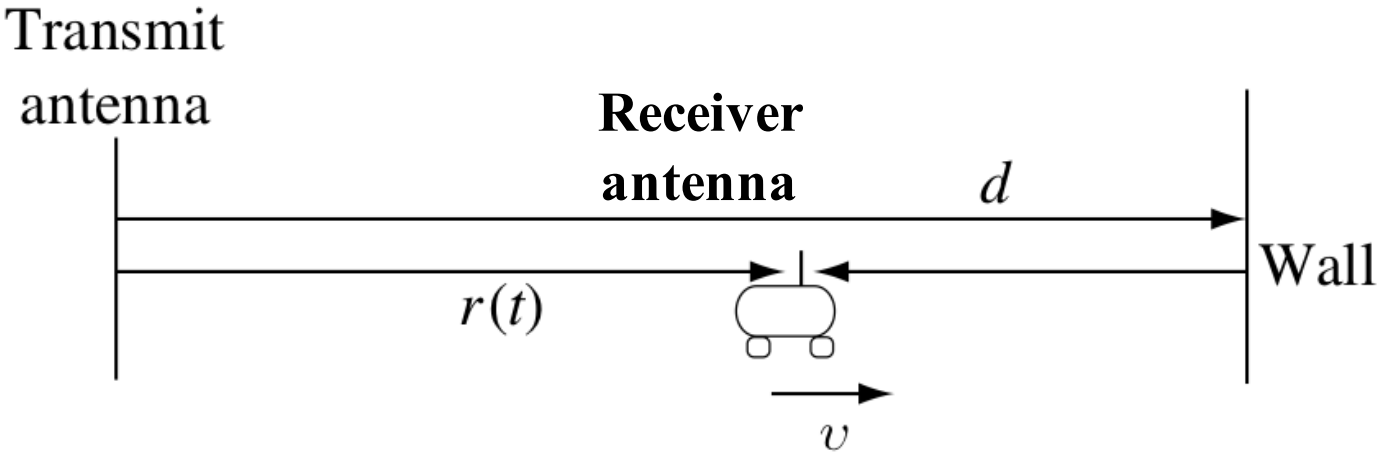
The wireless channel varies quickly and unpredictably

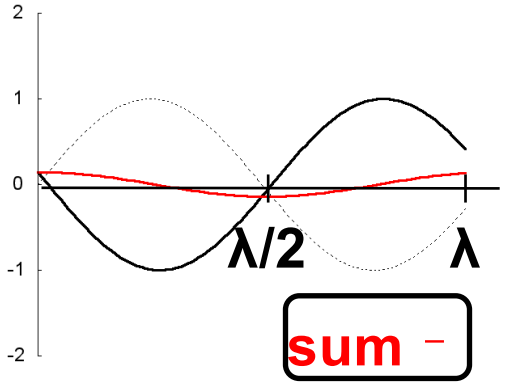
- Walking speed wireless trace measuring SNR



- **Fades last just tens of milliseconds**
- Fades are **large in magnitude** (> 20 dB difference)

Channel *coherence time*

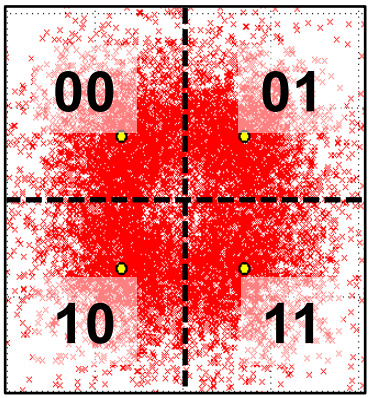


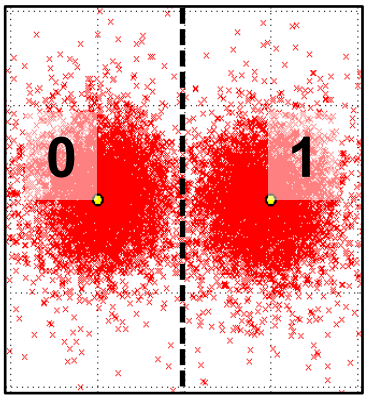


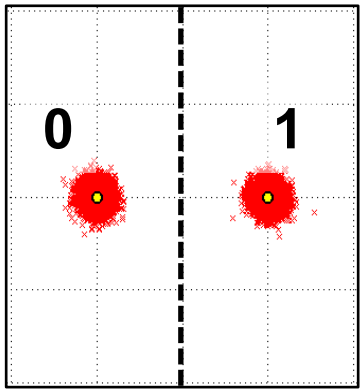
- Sender transmits the wireless signal at carrier frequency ***f = c / λ***
  - Speed of light: *c*; Wavelength of the signal: λ

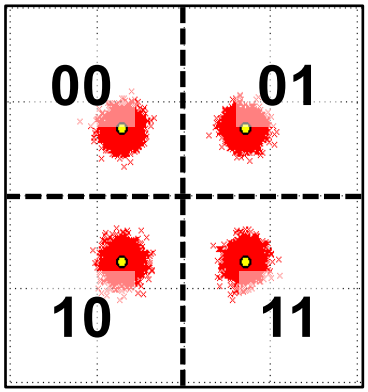
- **Change in path length difference** of λ/2 moves from ***constructive*** to ***destructive*** interference
  - Receiver movement of λ/4: ***coherence distance***
  - Time it takes to move a coherence distance: ***coherence time***
    - Walking speed @ 2.4 GHz: ≈ 15 milliseconds

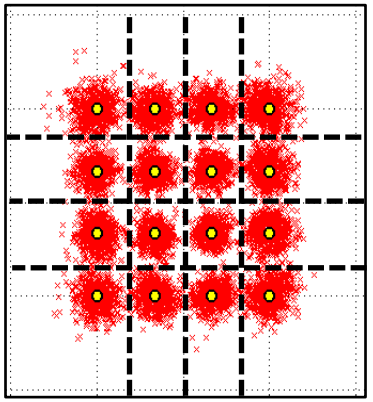
Current approach: Modulation adaptation





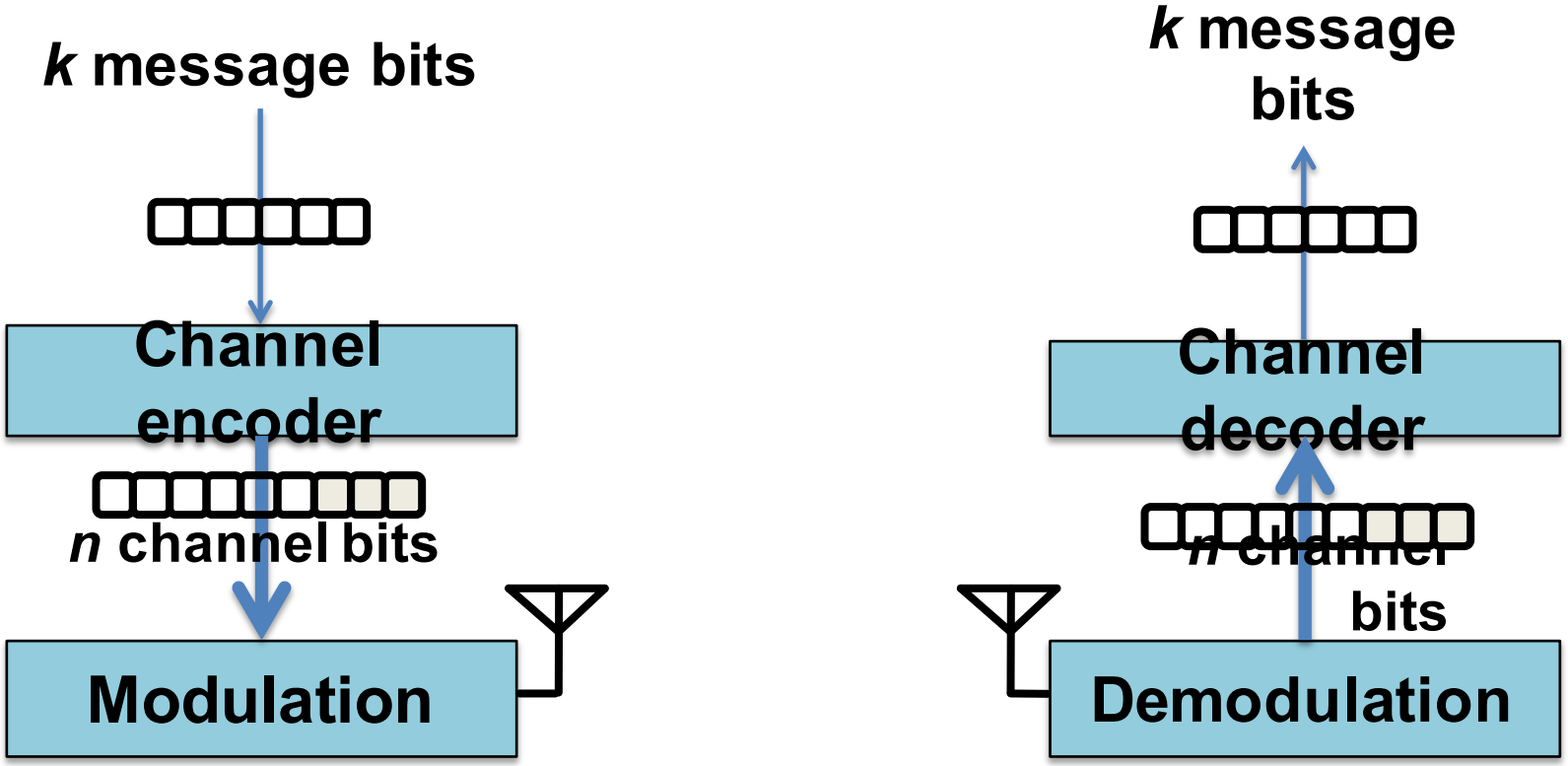






Metric: Signal-to-noise power ratio (SNR)

Current approach: Coding adaptation



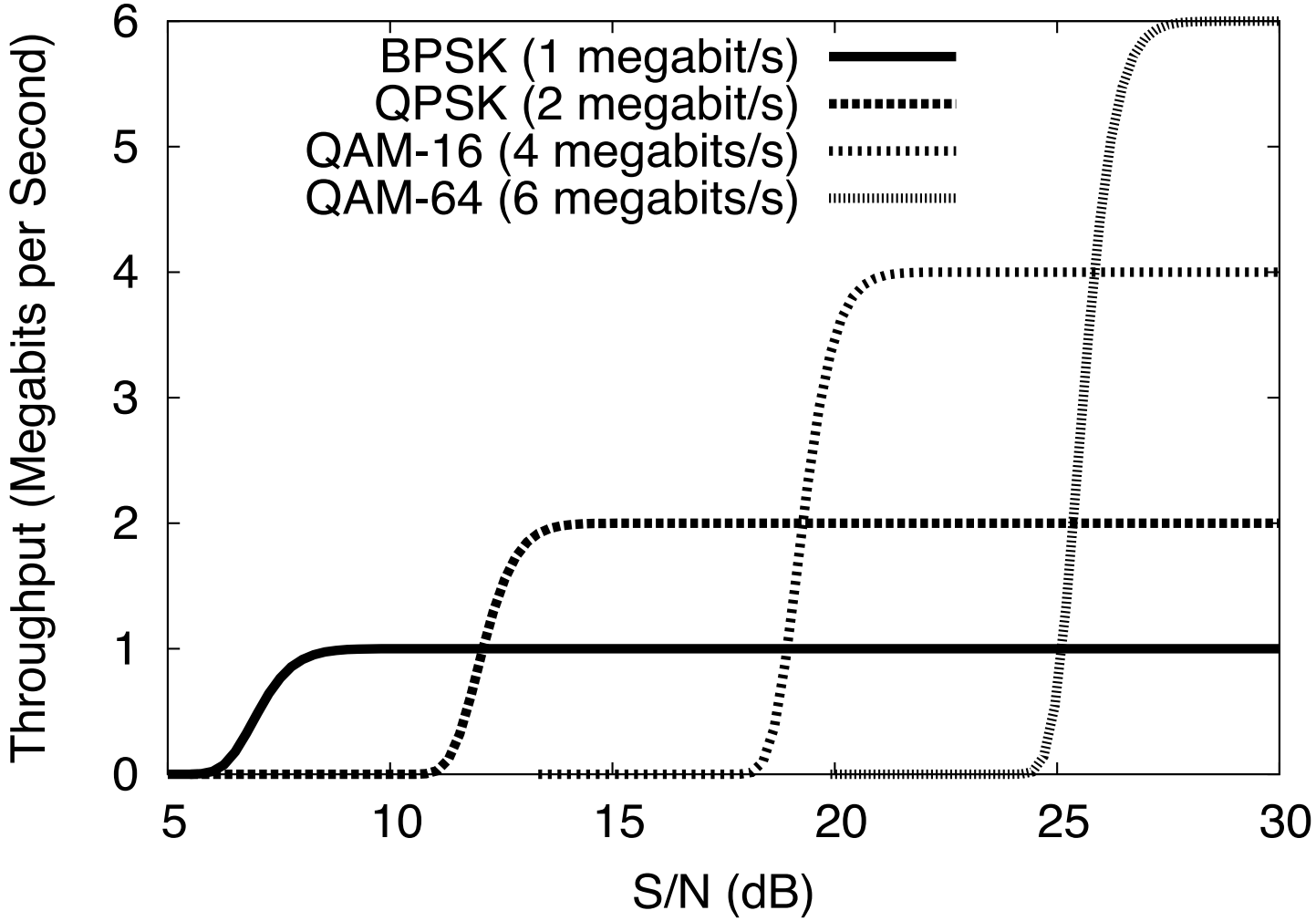
Current approaches adapt a fixed code rate: $R = \frac{k}{n}$

Lower code rate ☞ more parity bits ☞ more redundancy

WiFi bit rates change code rate, modulation

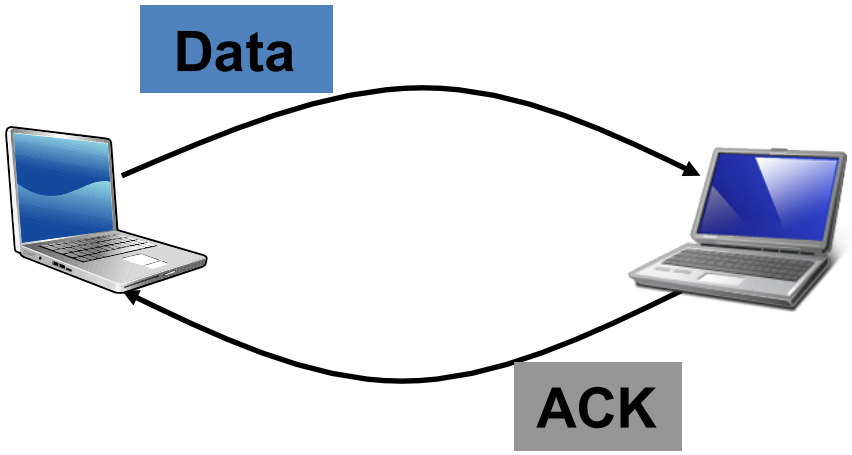
Bit-rate	802.11 Stan-dards	DSSS or OFDM	Modulation	Bits per Symbol	Coding Rate	Mega-Symbols per second
1	b	DSSS	BPSK	1	1/11	11
2	b	DSSS	QPSK	2	1/11	11
5.5	b	DSSS	CCK	1	4/8	11
11	b	DSSS	CCK	2	4/8	11
6	a/g	OFDM	BPSK	1	1/2	12
9	a/g	OFDM	BPSK	1	3/4	12
12	a/g	OFDM	QPSK	2	1/2	12
18	a/g	OFDM	QPSK	2	3/4	12
24	a/g	OFDM	QAM-16	4	1/2	12
36	a/g	OFDM	QAM-16	4	3/4	12
48	a/g	OFDM	QAM-64	6	2/3	12
54	a/g	OFDM	QAM-64	6	3/4	12

Fixed-rate codes *require* channel adaptation



Existing rate adaptation algorithms

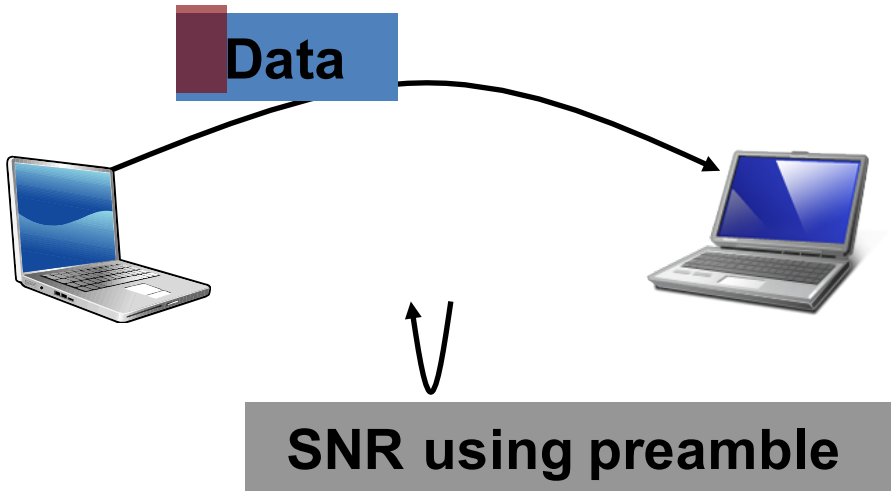
Frame-based



Estimate frame loss rate at each bit rate

- **RRAA**, Wong *et al.*, 2006.
- **SampleRate**, Bicket, 2005.
- **ARF**, **ONOE**

SNR/BER-based



Lookup table: SNR/BER → best rate

- **RBAR**, Holland *et al.*, 2001.
- **CHARM**, Judd *et al.*, 2008.
- **SoftRate**, Vutukuru *et al.*, 2009

SENSOR HINTS DISCUSSION