University of Glasgow

TCP

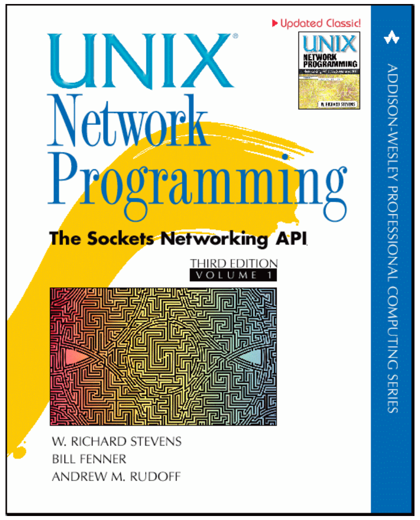
Networked Systems 3
Lecture 13

Lecture Outline

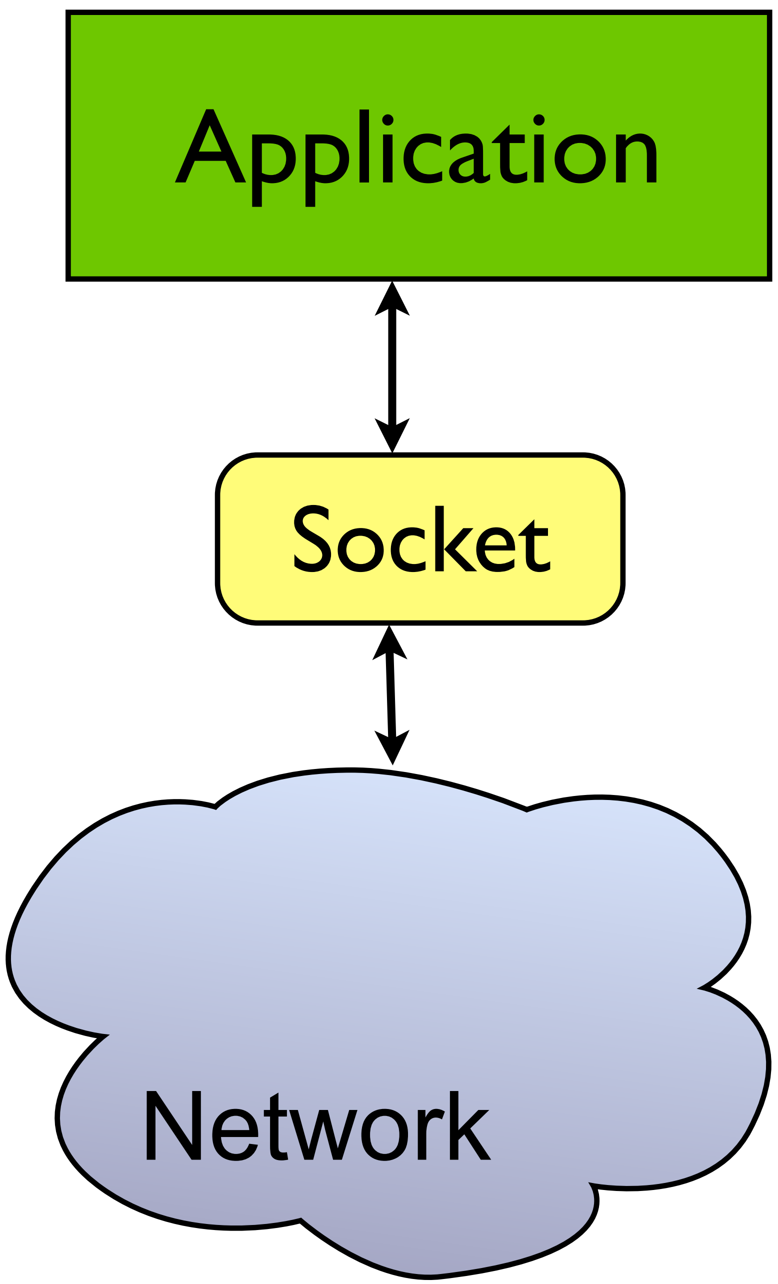
- The Berkeley Sockets API
- The TCP protocol and stream sockets

The Berkeley Sockets API

- Widely used low-level C networking API
- First introduced in 4.BSD Unix
  - Now available on most platforms: Linux, MacOS X, Windows, FreeBSD, Solaris, etc.
  - Largely compatible cross-platform
- Recommended reading:
  - Stevens, Fenner, and Rudoff, “Unix Network Programming volume 1: The Sockets Networking API”, 3rd Edition, Addison-Wesley, 2003.



Concepts



- Sockets provide a standard interface between network and application
- Two types of socket:
  - Stream – provides a virtual circuit service
  - Datagram – delivers individual packets
- Independent of network type:
  - Commonly used with TCP/IP and UDP/IP, but not specific to the Internet protocols
  - Discuss TCP/IP sockets today; UDP next lecture

Creating a socket

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int fd;
...
fd = socket(family, type, protocol);
if (fd == -1) {
    // Error: unable to create socket
    ...
}
...
```

family: `AF_INET` for IPv4, `AF_INET6` for IPv6

type: `SOCK_STREAM` for TCP, `SOCK_DGRAM` for UDP

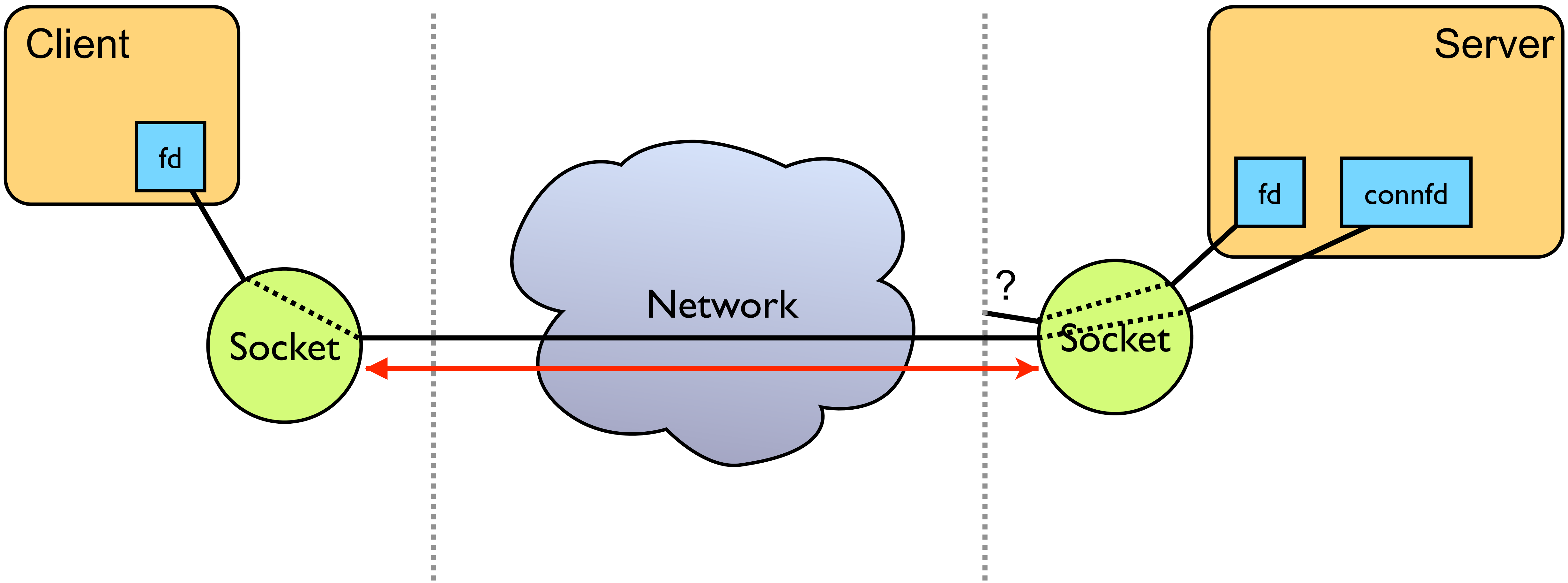
protocol: 0 (not used for Internet sockets)

Create an unbound socket, not connected to network; can be used as either a client or a server

What is a TCP/IP Connection?

- A reliable byte-stream connection between two computers
  - Most commonly used in a client-server fashion:
    - The server listens on a well-known *port*
      - The *port* is a 16-bit number used to distinguish servers
      - E.g. web server listens on port 80, email server on port 25
    - The client connects to that port
  - Once connection is established, either side can write data into the connection, where it becomes available for the other side to read
- The Sockets API represents the connection using a *file descriptor*

Using TCP Connections




```
int fd = socket(...)

connect(fd, ..., ...)

write(fd, data, datalen)

read(fd, buffer, buflen)

close(fd)
```

```
int fd = socket(...)
bind(fd, ..., ...)
listen(fd, ...)
connfd = accept(fd, ...)

read(connfd, buffer, buflen)

write(connfd, data, datalen)

close(connfd)
```

Implementing a Server: Bind and Listen

- A new socket can become either client or server
- To implement a server:
  - Bind to a *port* on a network interface
    - Specify a well-known port for the service, and INADDR_ANY to indicate any available network interface
  - Listen for new connections on that port
    - The *backlog* is the maximum number of connections the socket will queue up, each waiting to be `accept()`'ed

```
#include <sys/types.h>
#include <sys/socket.h>
...
if (bind(fd, addr, addrlen) == -1) {
    // Error: unable to bind
    ...
}
...
if (listen(fd, backlog) == -1) {
    // Error
    ...
}
...
```

Implementing a Server: Accept

- Once the server socket is listening for connections, call `accept()` in a loop to accept new connections in turn:

```
int                 connfd;
struct sockaddr_in cliaddr;
socklen_t           cliaddrlen = sizeof(cliaddr);
...
connfd = accept(fd, (struct sockaddr *) &cliaddr, &cliaddrlen);
if (connfd == -1) {
    // Error
    ...
}
...
```

The `connfd` is a new file descriptor for this connection
The original `fd` remains open, and can be used to accept another connection

Implementing a Client

- A client doesn't need to `bind()` or `listen()`, and simply connects to the server
  - The *addr* parameter includes the IP address and port on which the server is listening

```
#include <sys/types.h>
#include <sys/socket.h>
...
if (connect(fd, addr, addrlen) == -1) {
    // Error: unable to open connection
    ...
}
...
```

Specifying IP Addresses

- Specify an address and port in `bind()` and `connect()`
  - The address can be either IPv4 or IPv6
  - Addresses for `bind()` and `connect()` specified via `struct sockaddr`
  - Could be modelled in C as a union, but the designers of the sockets API chose to use a number of structs, and abuse casting instead
  - The `sa_data` field is big enough to hold the largest address of any family; `sa_len` and `sa_family` specify the length and type of the address
  - Treats address as opaque binary string

```
struct sockaddr {
    uint8_t         sa_len;
    sa_family_t     sa_family;
    char            sa_data[22];
};
```

Specifying IP Addresses: IPv4

- Two variations exist for IPv4 and IPv6 addresses
  - Use `struct sockaddr_in` to hold an IPv4 address
  - Has the same size and memory layout as `struct sockaddr`, but interprets the bits differently to give structure to the address

```
struct in_addr {
    in_addr_t       s_addr;
};

struct sockaddr_in {
    uint8_t         sin_len;
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    char            sin_pad[16];
};
```

Specifying IP Addresses: IPv6

- Two variations exist for IPv4 and IPv6 addresses
  - Use `struct sockaddr_in6` to hold an IPv6 address
  - Has the same size and memory layout as `struct sockaddr`, but interprets the bits differently to give structure to the address

```
struct in6_addr {
    uint8_t          s6_addr[16];
};

struct sockaddr_in6 {
    uint8_t          sin6_len;
    sa_family_t      sin6_family;
    in_port_t        sin6_port;
    uint32_t         sin6_flowinfo;
    struct in6_addr  sin6_addr;
};
```

Working with IP Addresses

- Work with either `struct sockaddr_in` or `struct sockaddr_in6`
- Cast it to a `struct sockaddr` before calling the socket routines

```
struct sockaddr_in  addr;
...
// Fill in addr here
...
if (bind(fd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
    ...
```

Creating an Address: INADDR_ANY

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

- Servers often just want to listen on the default address – do this using INADDR_ANY for the address passed to `bind()`
- Convert port number using `htons(…)`

```
struct sockaddr_in  addr;
...
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_family      = AF_INET;
addr.sin_port        = htons(80);

if (bind(fd, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
    ...
```

Creating an Address: Manually

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

- Clients want to connect to a specific IP address – can use `inet_pton()` to create the address, if you know the numeric IP address
- Convert port number using `htons(…)`

```
struct sockaddr_in  addr;
...
inet_pton(AF_INET, “130.209.240.1”, &addr.sin_addr);
addr.sin_family = AF_INET;
addr.sin_port   = htons(80);

if (connect(fd, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
    ...
```

DON’T DO THIS – USE THE DNS INSTEAD

→ Lecture 16

Role of the TCP Port Number

Port Range		Name	Intended use
0	1023	Well-known (system) ports	Trusted operating system services
1024	49151	Registered (user) ports	User applications and services
49152	65535	Dynamic (ephemeral) ports	Private use, peer-to-peer applications, source ports for TCP client connections

RFC 6335

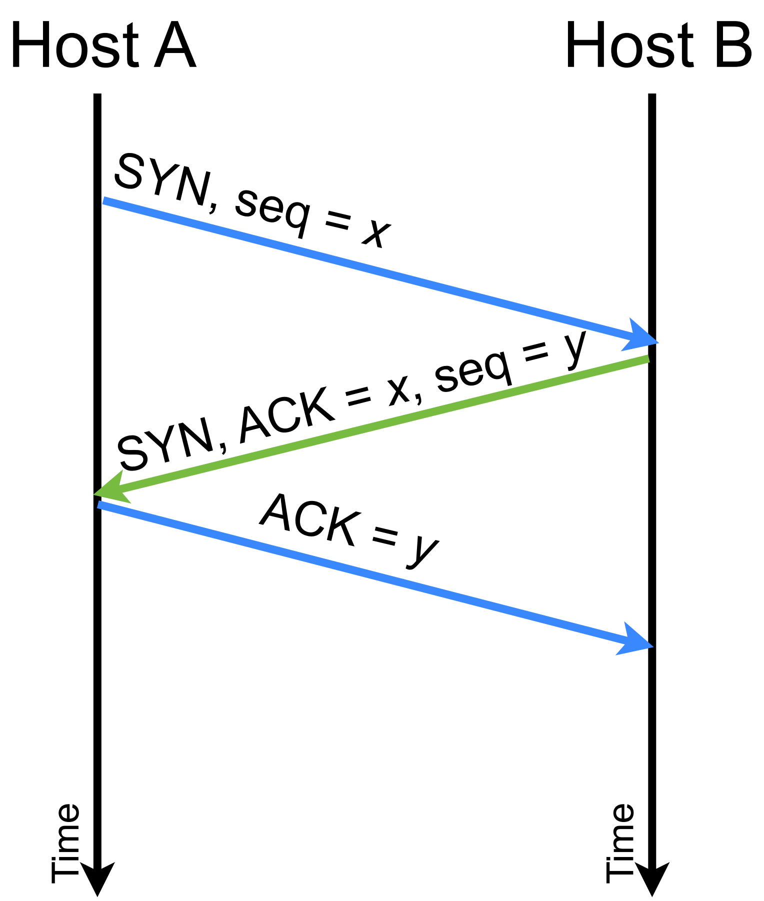
- Servers must listen on a known port; IANA maintains a registry
- Distinction between system and user ports ill-advised – security problems resulted
- Insufficient port space available (>75% of ports are registered)

- TCP clients traditionally connect from a randomly chosen port in the ephemeral range
  - The port must be chosen randomly, to prevent spoofing attacks
  - Many systems use the entire port range for source ports, to increase the amount of randomness available

http://www.iana.org/assignments/port-numbers

TCP Connection Setup

- Connections use 3-way handshake
  - The SYN and ACK flags in the TCP header signal connection progress
  - Initial packet has SYN bit set, includes randomly chosen initial sequence number
  - Reply also has SYN bit set and randomly chosen sequence number, acknowledges initial packet
  - Handshake completed by acknowledgement of second packet
  - Happens during the `connect()`/`accept()` calls
- Combination ensures robustness
  - Randomly chosen initial sequence numbers give robustness to delayed packets or restarted hosts
  - Acknowledgements ensure reliability



Similar handshake ends connection, with FIN bits signalling the teardown

Reading and Writing Data

```
#define BUFLEN 1500
...
ssize_t i;
ssize_t rcount;
char    buf[BUFLEN];
...
rcount = read(fd, buf, BUFLEN);
if (rcount == -1) {
    // Error has occurred
    ...
}
...
for (i = 0; i < rcount; i++) {
    printf("%c", buf[i]);
}
```

- The `read()` call reads *up to* BUFLEN bytes of data from connection – blocks until data available
- Returns actual number of bytes read, or `-1` on error
- Data is *not* null terminated

```
char data[] = "Hello, world!";
int  datalen = strlen(data);
...
if (write(fd, data, datalen) == -1) {
    // Error has occurred
    ...
}
...
```

- The `write()` call sends data over a socket; blocks until all data can be written
- Returns actual number of bytes written, or `-1` on error

Record Boundaries in TCP Connections

- If the data in a `write()` is bigger than the data link layer MTU, TCP will send the data as fragments
- Similarly, multiple small `write()` requests may be aggregated into a single TCP packet
- Implication: the data returned by a `read()` doesn't necessarily match that sent in a single `write()`
    - There often appears to be a correspondence, but this *is not* guaranteed (it may work in the lab, but not when you use it over a different link)

Application Level Framing

Data may arrive in arbitrary sized chunks; must parse and understand the data, no matter where it is split by the network – it's a byte stream (colours indicate one possible split of the data into chunks)

```
HTTP/1.1 200 OK
Date: Mon, 19 Jan 2009 22:25:40 GMT
Server: Apache/2.0.46 (Scientific Linux)
Last-Modified: Mon, 17 Nov 2003 08:06:50 GMT
ETag: "57c0cd-e3e-17901a80"
Accept-Ranges: bytes
Content-Length: 3646
Connection: close
Content-Type: text/html; charset=UTF-8

<HTML>
<HEAD>
<TITLE>Computing Science, University of Glasgow </TITLE>
...
</BODY>
</HTML>
```

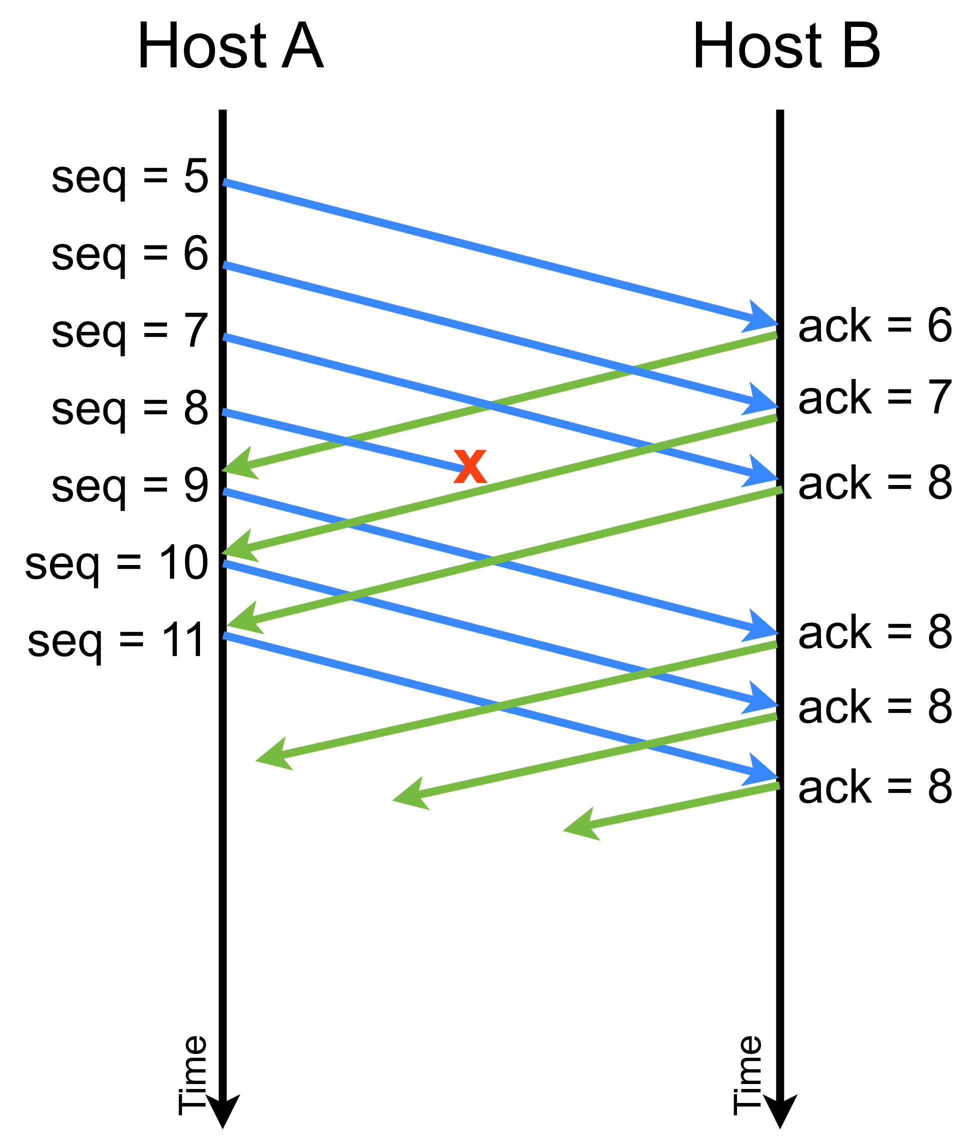
Example: HTTP response

Known marker (blank line) signals end of headers

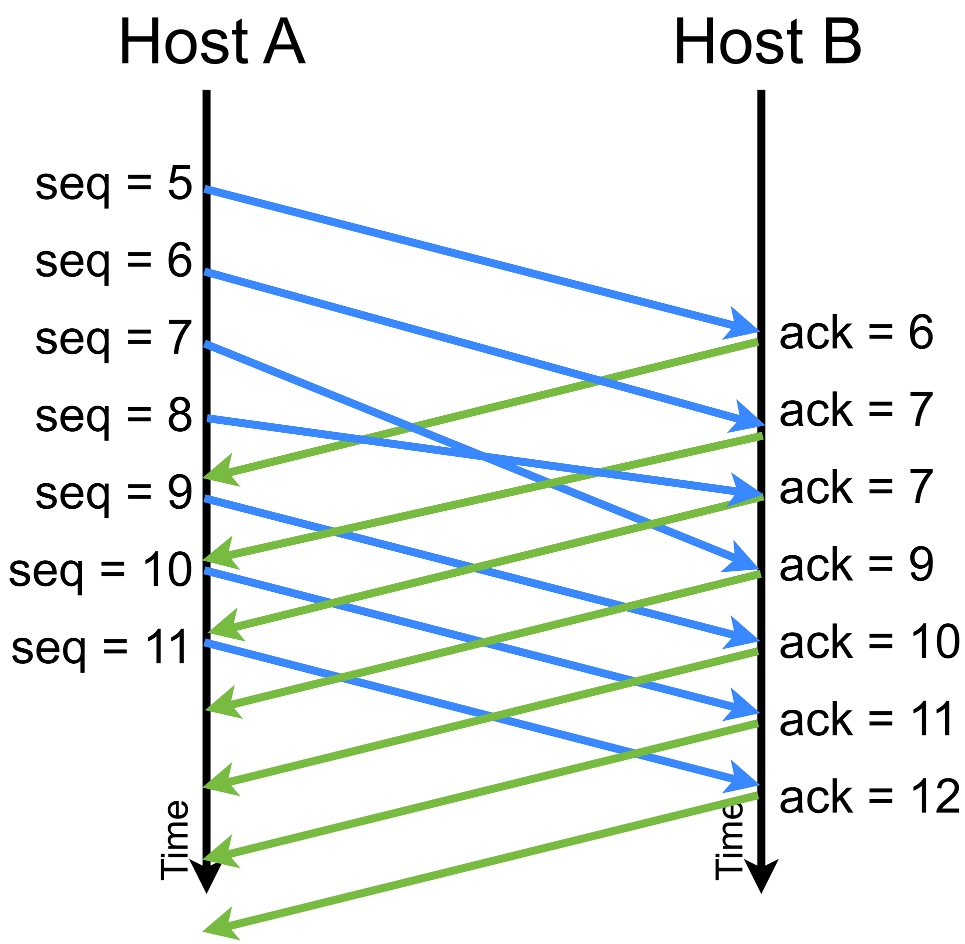
Size of payload indicated in the headers

TCP Reliability

- TCP connections are reliable
  - Application data gathered into packets
  - Each packet has a sequence number and an acknowledgement number
    - Sequence number counts how many bytes are sent (this example is unrealistic, since it shows one byte being sent per packet)
  - Acknowledgement number specifies next byte expected to be received
    - Cumulative positive acknowledgement
    - Only acknowledge contiguous data packets (sliding window protocol, so several data packets in flight)
    - Duplicated acknowledgements imply loss
  - TCP layer retransmits lost packets – this is invisible to the application



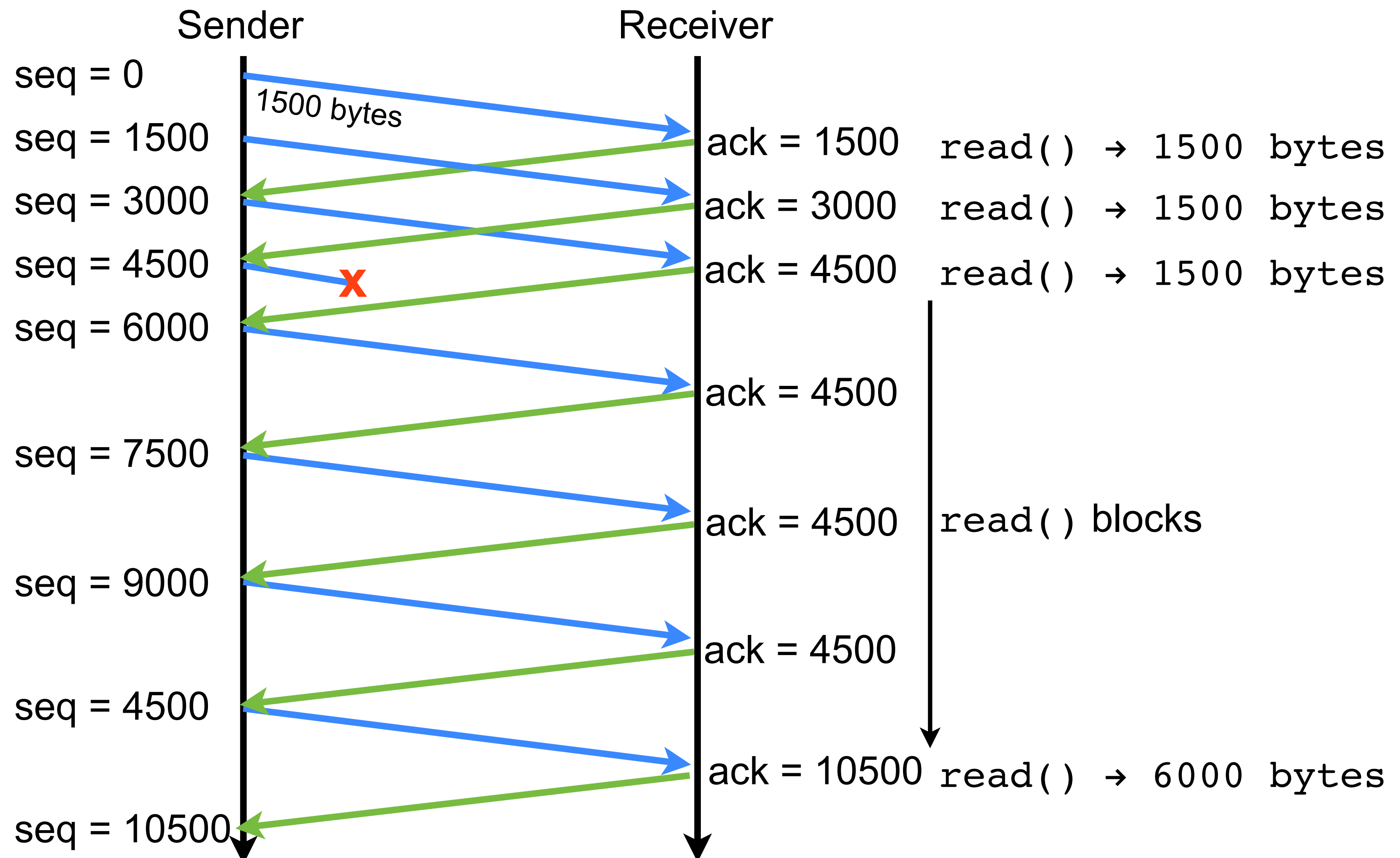
TCP Reliability: How is Loss Detected



- Packet reordering also causes duplicate ACKs
  - Gives appearance of loss, when the data was merely delayed
- TCP uses *triple duplicate ACK* to indicate loss
  - Four identical ACKs in a row
  - Slightly delays response to loss, but makes TCP more robust to reordering

Head of Line Blocking in TCP

- Data delivered in order, even after loss occurs
  - TCP will retransmit the missing data, transparently to the application
  - A `read()` for the missing data will block until it arrives; TCP delivers all data in-order



Summary

- The Berkeley Sockets API
- Implementing TCP client and server sockets
- The TCP API:
  - Reliability
  - Unframed byte stream
  - Head of line blocking