The **optional quiz**
is administered in
the labs in the week of Feb. 10th
(cf. web site for instructions)

Network Programming with Sockets

http://ece297.msrg.utoronto.ca

I added some additional slides that we wont cover in the lectures; they are for your information.

Stats :

ῆ 350 submissions (should have been ~380)

ῆ 9 with wrong directory structure (did not
follow submission instructions)

Quiz: Week of Feb. 10th: Dedicated lab: TBD

- Optional & pass/fail
- Entice you to use svn & testsubmit
- Consistent use of svn over a number of days (start now!)
- See instructions online
- One student on behalf of the whole team

Midterm: Get an early start!

- Read the online instructions now !
- Full lecture dedicated to midterm: Feb 12th
- Evaluates Milestones 1 & 2
  - Short team presentation
  - Question-based demo (by individual !)
  - General questions (see online samples)

Important points for Midterm

ñ Each student must run demo from his or her own account; running the demo from someone elseẠs account is not allowed

ñ It is your responsibility that your account is properly working, i.e., is not revoked, sufficient disk space is available, etc.

ñ Should there be a problem with the computer you are setting up on, the supervising TA will allow you to try one other computer

ñ You will have some setup time to verify that all is working, but you will be asked to start your demos from scratch (note the data loading requirement)!

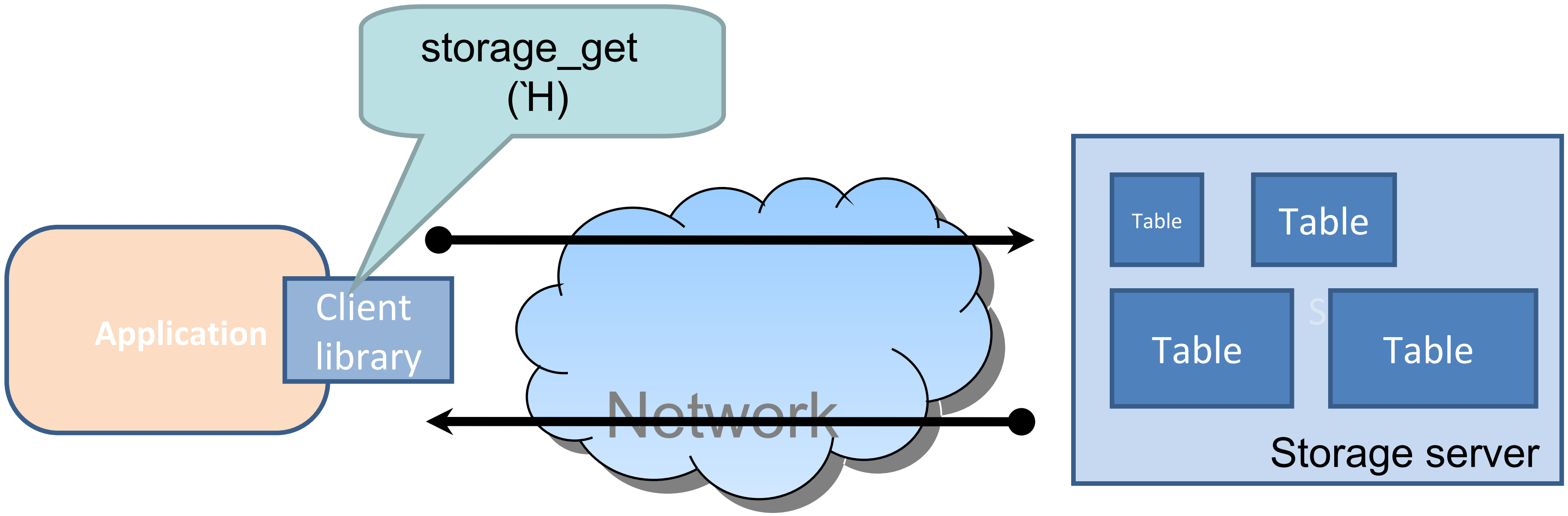
ñ Your shells should provide meaningful output for erroneous cases, e.g., retrieving tables that do not exist, keys that do not exist, etc.; a segmentation fault is not a meaningful output

Agenda

ῃ Network programming
  ᾶ Sockets *et al.*
  ᾶ Protocol design
  ᾶ Socket API

Network programming

How does storage_get(Ẑ) propagate over the network?



On the client side

Why not use a debugger to see what's going on?

gdb client

See client.c

```
status = storage_get (TABLE, KEY, &r, conn);
```

See storage.c: in storage_get

```
... sendall (sock, buf, strlen(buf)) `H && `H recvline(sock, buf,
    sizeof buf) ...
```

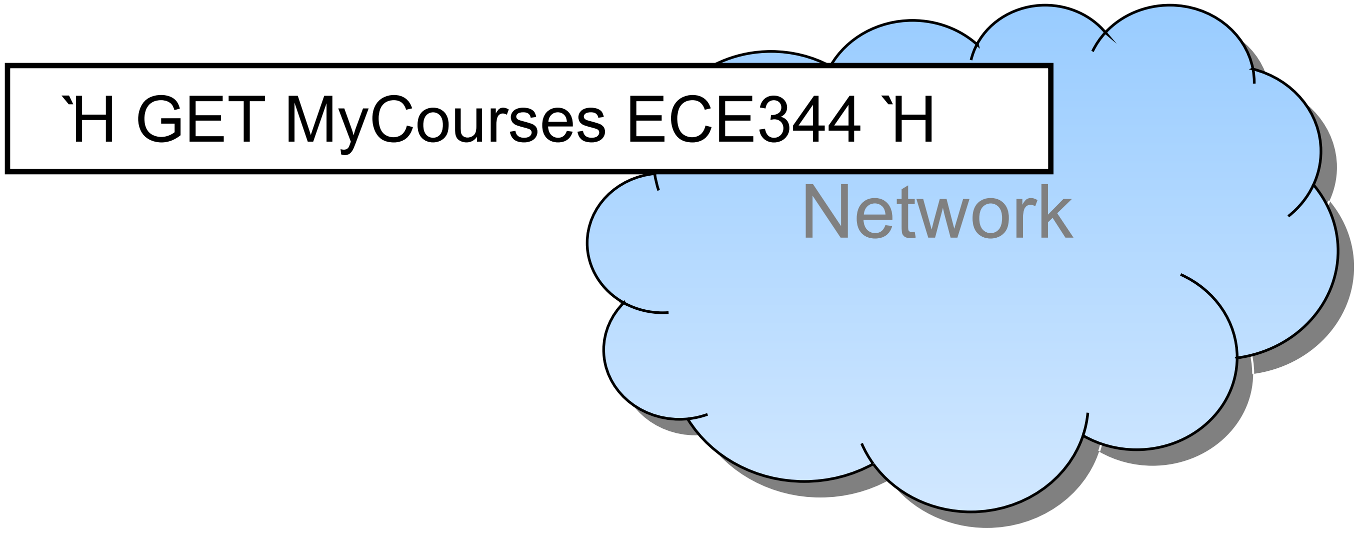
Our string array / buffer

See utils.c: in sendall

```
`H
while (tosend > 0) {
        ssize_t bytes = send (sock, buf, tosend, 0);
        if (bytes <= 0)
                break; // send() was not successful, so stop.
        tosend -= bytes;
        buf += bytes;
};
```

The string buffer

storage_get(’MyCourses˜, ’ECE344˜, &record, conn)



On the server side

See server.c in main

`H

```
while (wait_for_connections) { ...
  // Get commands from client.
  do { `H
    // Read a line from the client.
    int status = recvline (clientsock, cmd,
    MAX_CMD_LEN);
    `H
    // Handle the command from the client.
    int status = handle_command (clientsock, cmd);
```

Our string array / buffer

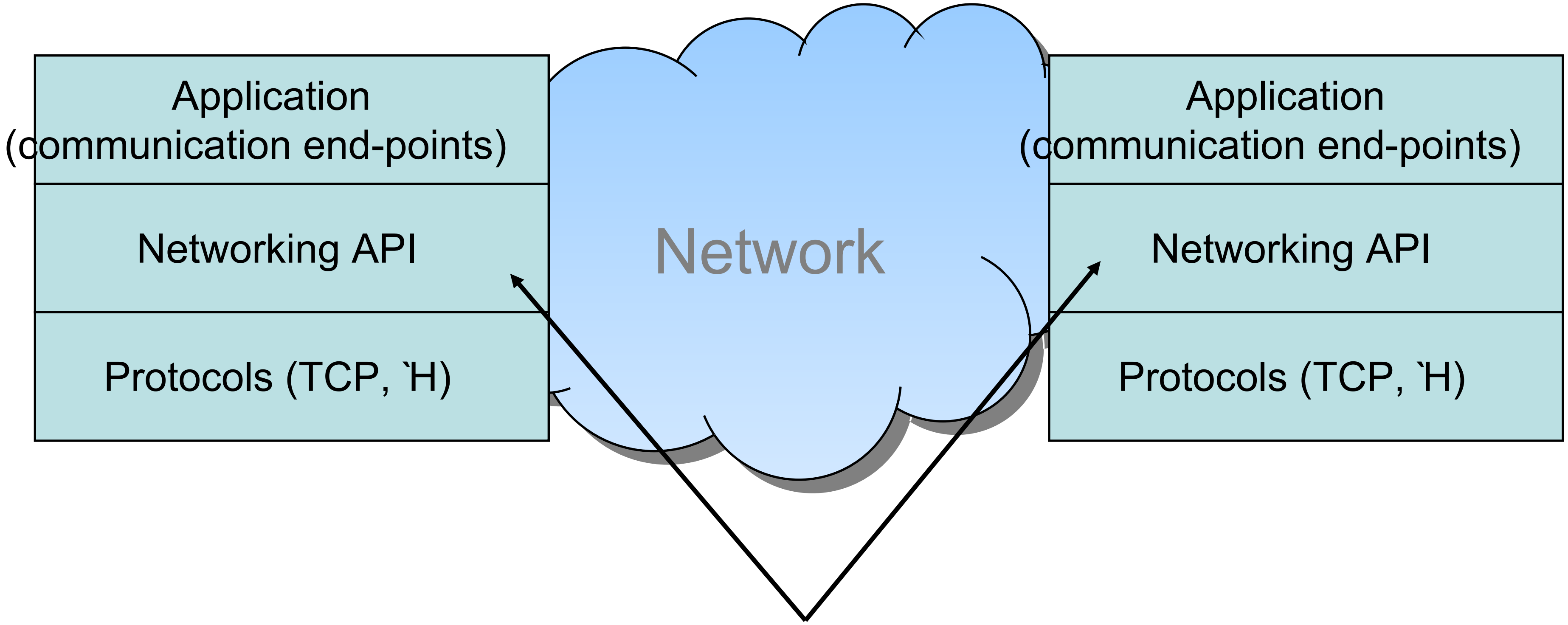
Why not use a debugger to see what's going on?

gdb server

On the server side

```
int  handle_command      (int sock, char *cmd){ `H
   LOG(("Processing command '%s'\n", cmd));

   // For now, just send back the command to the
   client.
   sendall   (sock, cmd, strlen(cmd));
   `H
}
```

Using the ’network˜

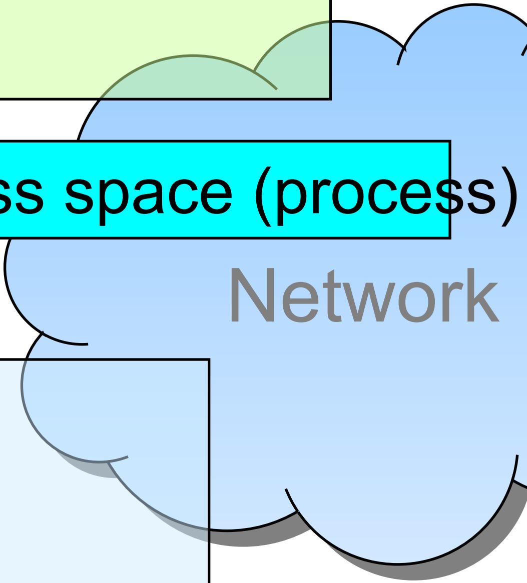


Sockets are network programming abstractions.

Local call vs. sending data

```
int handle_command(int s, char *cmd){
        LOG (("Processing command '%s'\n",
cmd));
        sendall   (sock, cmd, strlen(cmd) );
```

Local call: Function call in the same address space (process)



```
int sendall(int s, char *buf, size_t len){
        ssize_t tosend = len;
        while (tosend > 0) {
                size_t bytes =      send  (sock, buf, tosend, 0);
```

Sending data: A system call that initiates transfer of data across the net

Communication end-points & sockets

ñ Identify the *application* that sends and receives data and the *host* the application runs on

 ã Multiple applications running on the same machine

ñ End-points are programmatically represented by sockets (#include <sys/socket.h>)

ñ Create the end-points

ñ Connect the end-points

ñ Listen and accept connections on end-points

ñ Transmit messages via sending and

Sockets are file descriptors

- Socket is a programming abstraction for communicating among processes (applications) based on (Unix) *file descriptors*

- A file descriptor is an integer representing an open file managed by the OS
  - In Unix any I/O is done by reading/writing from/to file descriptors
  - For sockets, also called socket descriptor

Our focus in ECE 297

ñ Our focus is Internet sockets
  ã there are others

ñ Our focus is sockets of type stream, i.e., SOCK_STREAM (i.e., based on TCP)
  ã there are other types

ñ Our focus is IPv4

ñ All this is provided for you in the skeleton code.

SOCK_STREAM Sockets

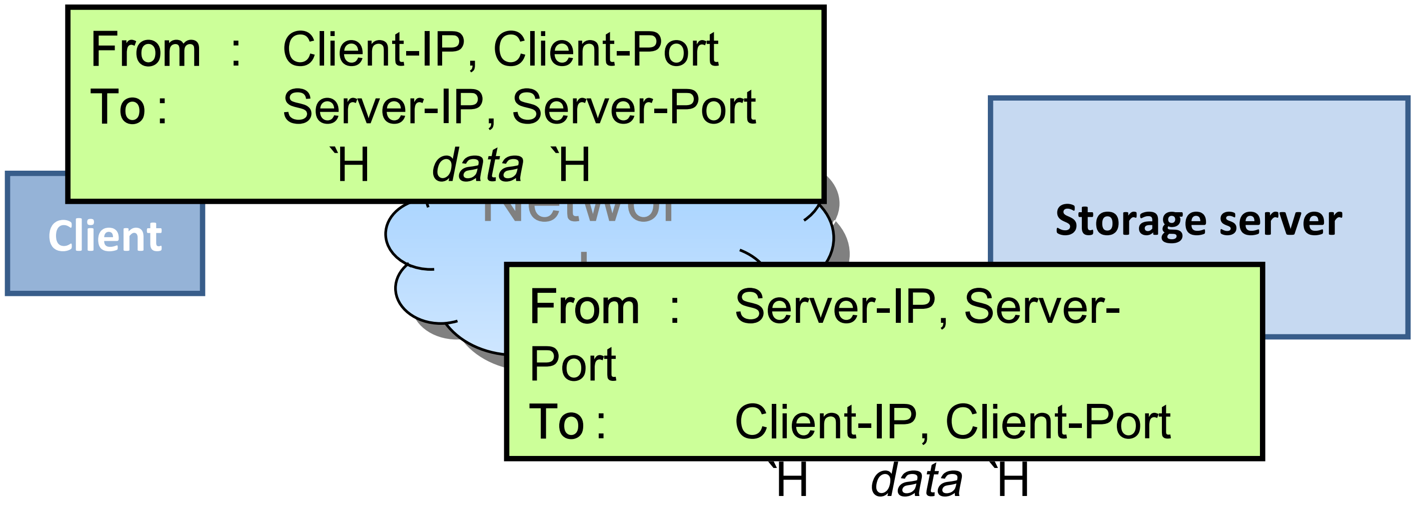
ῆ Are reliable , two-way connected communication streams

  ᾶ Output two items into the socket in the order "1, 2", they arrive in the order "1, 2" at the opposite end (’`H *if they arrive at allẇ* ☹)

  ᾶ Transmission is error-free (i.e., messages are not corrupted)

Two end-points determine a connection

ñ End-point is determined by
  - ã IP address (host address); e.g., 192.168.100.100
  - ã Port number , e.g., 8888



From : Client-IP, Client-Port
To : Server-IP, Server-Port
`H data `H
Client
Networ
k
Storage server
From : Server-IP, Server-Port
To : Client-IP, Client-Port
`H data `H

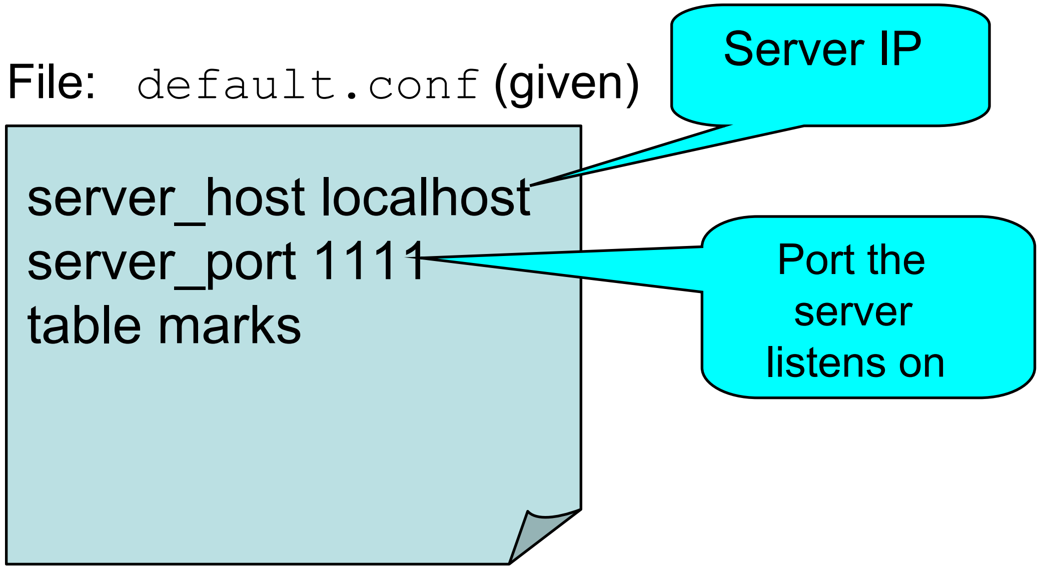
Try **`netstat`** to see open sockets on your machine

man netstat for help

The configuration file & server IP, port

- The configuration file sets *properties* for the storage serv
- It is read at startup to initialize the storage server.
- It is not available to client (i.e., it resides across network

File: `default.conf` (given)



```
server_host localhost
server_port 1111
table marks
```

Most software has configuration files and ways to overwrite their settings.

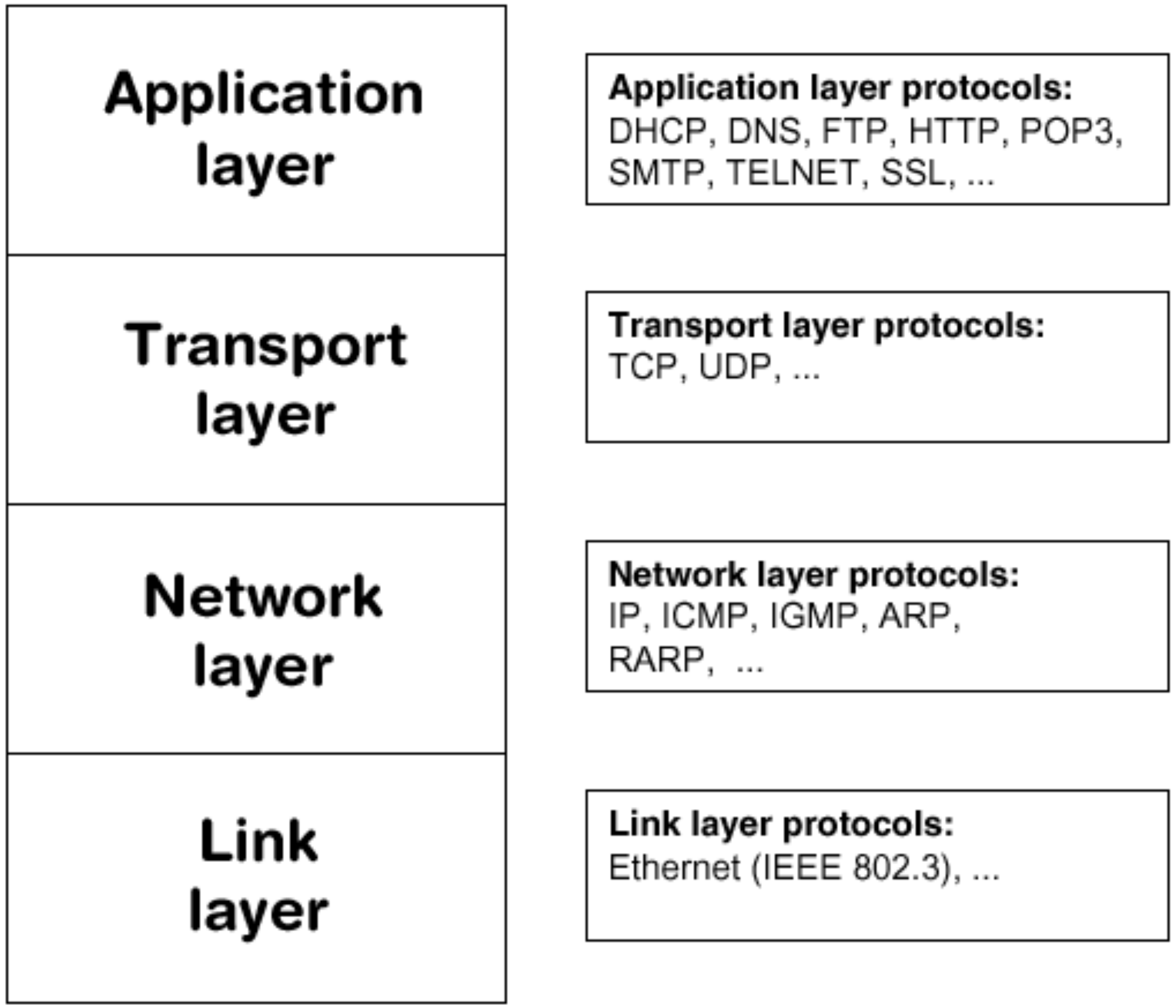
For a further example see onetable.conf in the skeleton code distribution

IP address

(Only what we need to know at this point)

- Internet Protocol (IP) address of a host (32bit number)
- Split up into four eight-bit numbers
- At home you may see 192.168.0.0 `H
  - Reserved block of IP address for private networks
  - Your modem / router is assigned a public IP by Bell/Rogers
- For development we can use localhost
  - Meaning this computer / host
  - Translated by OS to loopback IP address 127.0.0.1
- Domain Name System (DNS) maps names to IP addresses
  - It is easier to remember www.example.com than 192.168.212.4

The network stack



Application layer protocols:
DHCP, DNS, FTP, HTTP, POP3, SMTP, TELNET, SSL, ...

Transport layer protocols:
TCP, UDP, ...

Network layer protocols:
IP, ICMP, IGMP, ARP, RARP, ...

Link layer protocols:
Ethernet (IEEE 802.3), ...

Ports

- Example : Computer with a given IP address handles Mail, IM, Torent, and Web browsing
  - *How should the OS differentiate among these networked applications, if packets arrive?*
- A port is a 16 bit number to locally (per host) identify the connection
- Numbers (vary by OS)
  - 0-1023 ’reserved˜, must be *root* to use
  - 1024 – 65535 are available to regular user
- Well-known, reserved services
  - http 80/tcp
  - ftp 21/tcp
  - ssh 22/tcp
  - Others: telnet 23/tcp; finger 79/tcp; snmp 161/udp
- See `/etc/services` in Unix for all available services

’Error binding socket"

ῆ Port is used by another process

–`ps aux | grep server`

ῆ Kill it, if it is your own

ῆ Use a different port number, if it is someone elseẠs who might be logged in to the machine via ssh

Byte order

Good interview question

- The order bytes are represented by a computer architecture in memory, on the wire, `H
- E.g., 0x *A3 F1* as two bytes // 0x HEX number
  - Stored as A3 F1 // Big-Endian (' *Big end first* ˜)
    - Motorola 6800, 68k, PowerPC (Intel), `H
    - Network byte order
  - Stored as F1 A3 // Little-Endian (' *Little end first* ˜)
    - Intel or Intel-compatible processors
    - x86, 6502, Z80, VAX, PDP-11
- Host byte order : the order used by the host vs. the network byte order , the order used by the network
- Important to honor network byte order when building messages and filling out structures!

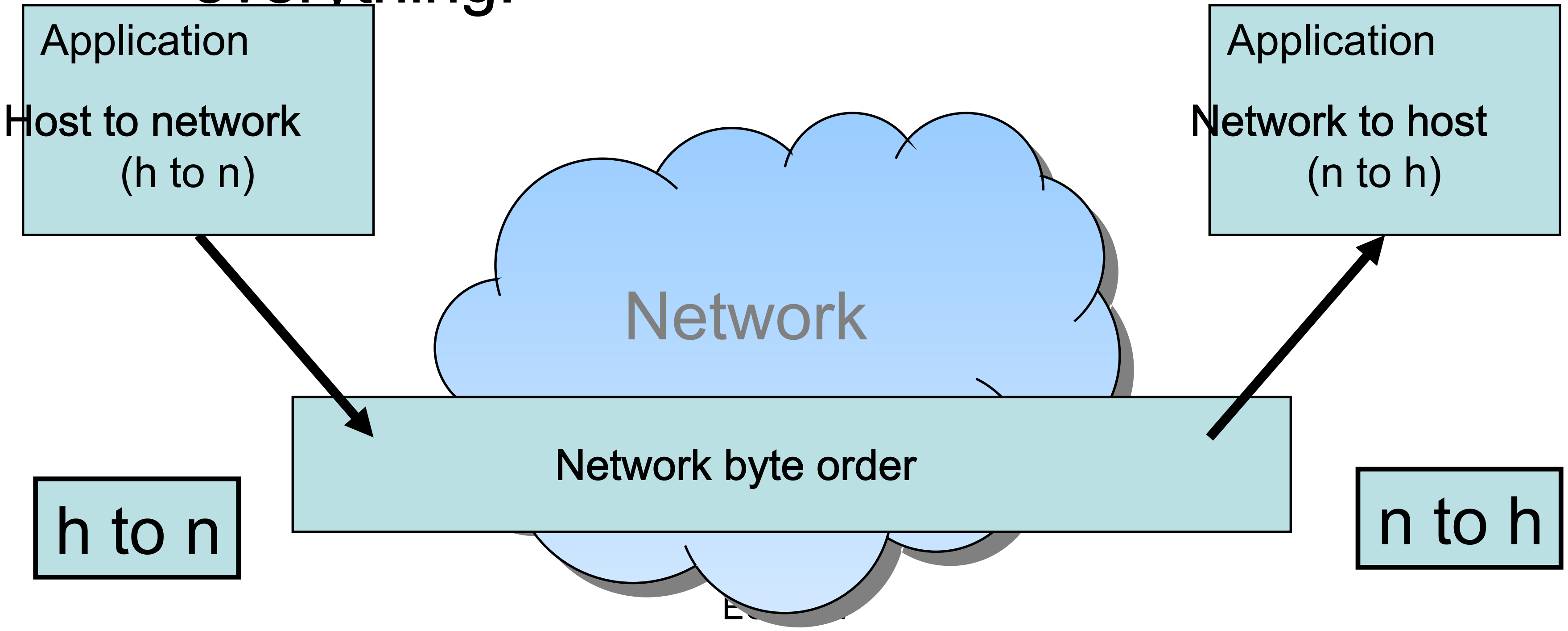
How do we know the host byte order?

Example (wrong): (Motorola) 0xA3F1 `H (Network) 0xA3F1 `H (Intel) 0xF1A3

41969 61859

1010 0011 1111 0001 1111 0001 1010 0011

But, donĄt worry; just transform everything!



h-to-n and n-to-h

htons(`H)	host to network short
htonl(`H)	host to network long
ntohs(`H)	network to host short
ntohl(`H)	network to host long

Great, but how do we handle floats?

Left as an exercise for the reader.

Protocol design

Whatṽs relevant for us for the assignments?

- Simplicity
  - Simple encoding of *client requests*
  - Simple encoding of *server response*
  - Need to worry about *error conditions*
  - DonẠt worry about saving a bit of bandwidth or computation
  - Consider debugging your communication
- Interoperability
  - We can not assume client and server run on the same hardware / OS
- Portability
  - We restrict ourselves to IPv4

The Protocol

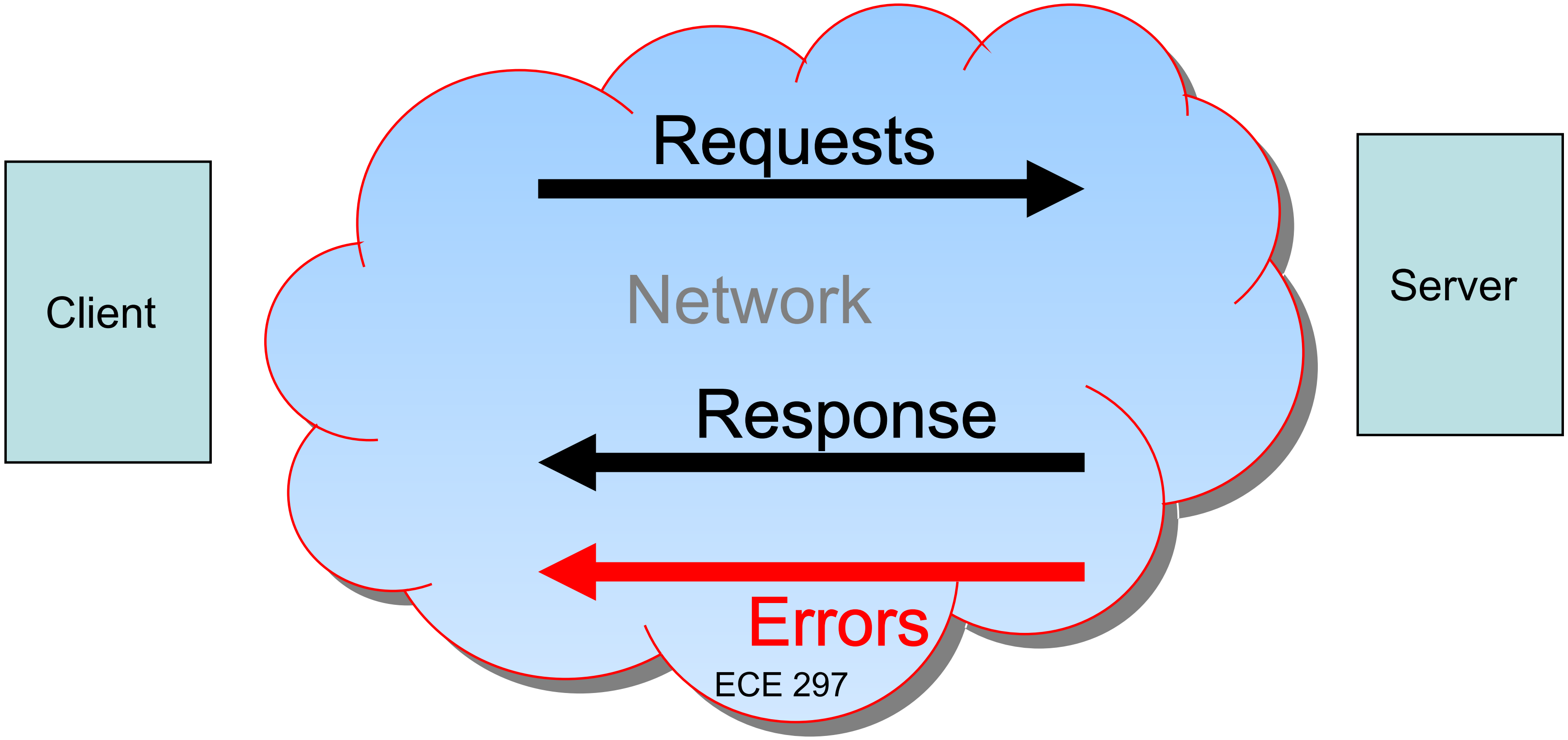
Example

Request: GET <path>/index.html HTTP/1.0

Response: HTTP/1.0 200 OK

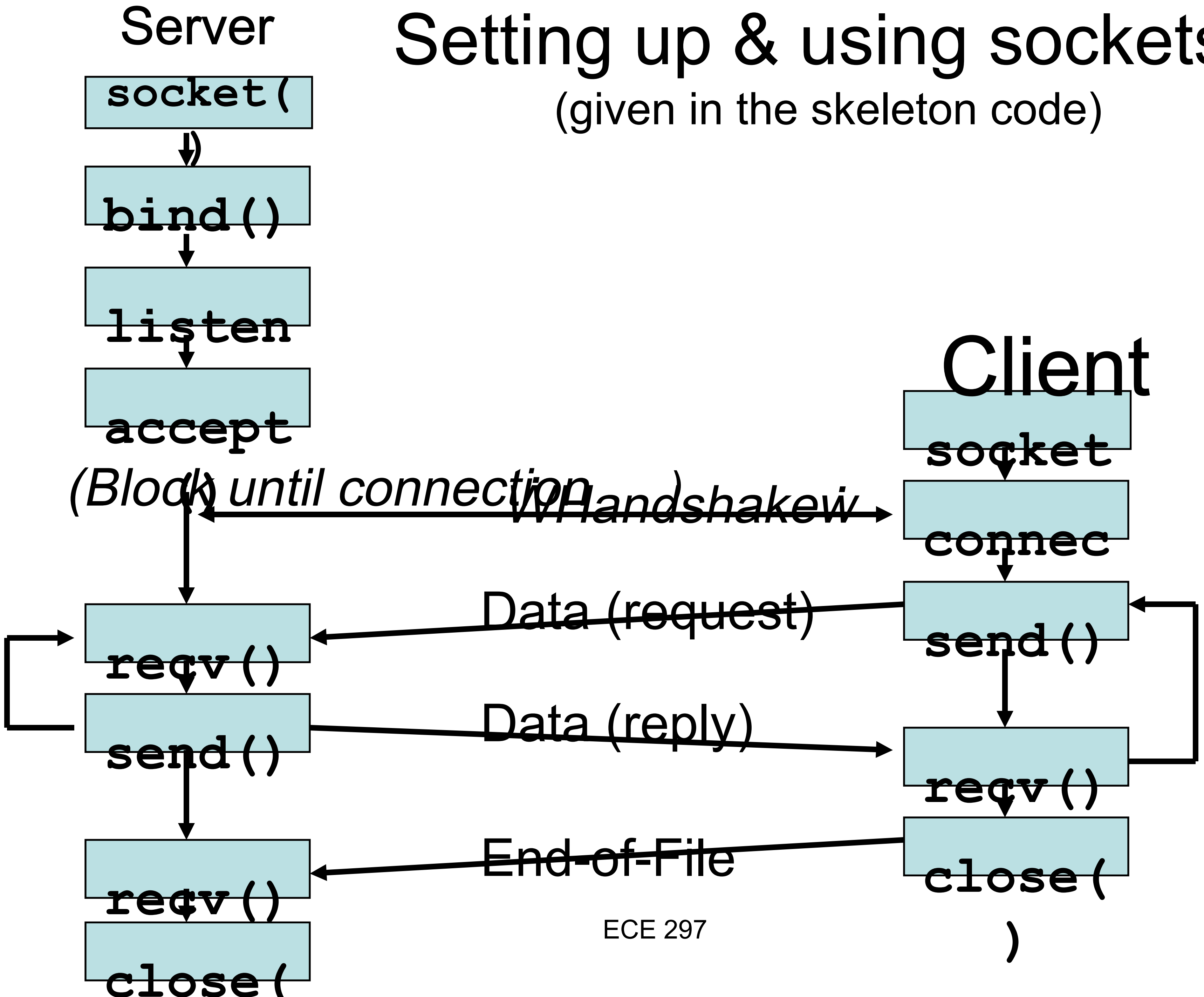
Messages

Error response: HTTP/1.0 404 Not Found



Setting up & using sockets

(given in the skeleton code)



Sending data

client & server

```
int send( int sockfd,
          const void *msg,
          int len, int flags);
```

ῆ msg is a pointer to the data to send

ῆ len is the length of the data to send

ῆ flag controls specifics, set to 0 for us

ῆ send(`H) returns the number of bytes sent, which might be less than what was requested

-1 is returned on error, and *errno* is set to the error number.

```
char *msg = ’    ECE297 is great fun, we learn useful stuff
int len, bytes_sent;
. . .
len = strlen(msg);
bytes_sent     = send(sockfd, msg, len, 0);
```

serẠs responsibility to send the rest

!

On the client side

See utils.c: in **sendall**

```
`H
while (tosend > 0) {
        ssize_t bytes =    send  (sock, buf, tosend, 0);
        if (bytes <= 0)
                break; // send() was not successful, so stop.
        tosend -= bytes;
        buf += bytes;
};
```

ADDITIONAL MATERIAL

Protocol design considerations

ñ How are API calls (e.g., get / set) sent from the client to the server?
  - ã *May they have given us any hints in the code?*

ñ What parts of a call need to be conveyed to the server?
  - ã E.g., int storage_get(*table, *key, *record, *conn);

ñ WhatĄs the simplest possible way of sending this information?

ñ What else needs to be represented in the protocol between the client library and the storage server?

Associate socket with port on host

```
int bind( int sockfd,
          struct sockaddr *myaddr,
          socklen_t addrlen);
```

Assigns a socket an address.

ñ sockfd is socket descriptor from `socket(…)`

ñ myaddr is a pointer to address struct with:

- ã port number and IP address
- ã if port is 0, then host picks port (>1023)
- ã IP address = INADDR_ANY

Especially when rerunning the server in development.

ñ addrlen is length of structure

ñ returns 0 on success, -1 on failure and sets errno

- ã Common is EADDRINUSE (' *Address already in use* ˜)

Client connecting to server

```
int connect(    int sockfd,
                struct sockaddr *serv_addr,
                int addrlen);
```

Contains destination IP address / port

ῃ Failures indicated by the returning of -1 and the setting of errno

ῃ E.g., bind(`H) was not called!

ῃ Setting local port is not important

ῃ The destination IP/port matters

ῃ OS selects local port & conveys to remote site

connect(`H) is used by the client

Accepting connections

```
int accept(      int sockfd,
                 struct sockaddr cliaddr,
                 socklen_t *addrlen);
```

ñ Return value & arguments are similar to the above (cf. bind(`H))

ñ Returns brand new descriptor created by OS

ñ Former descriptor continues to listen for new connections

ñ New descriptor is used to send and receive data

accept(`H) is used by the server

Sending data

client & server

```
int send( int sockfd,
          const void *msg,
          int len, int flags);
```

-1 is returned on error, and *errno* is set to the error number.

- ñ msg is a pointer to the data to send
- ñ len is the length of the data to send
- ñ flag controls specifics, set to 0 for us
- ñ send(`H) returns the number of bytes sent, which might be less than what was requested

```
char *msg = ' ECE297 is great fun, we learn useful stuff
int len, bytes_sent;
. . .
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
```

serĄs responsibility to send the rest

!

Receiving data & close

client & server

```
int recv( int sockfd,
          void *buf,
          int len, int flags);
```

ñ buf is a buffer to read data into

ñ len is the maximum length of the buffer

ñ flag controls specifics, set to 0 for us

ñ recv(`H) returns the number of bytes read

ñ recv(`H) may return 0; remoteẠs side closed connection on us

```
int close(int sockfd);
```

ñ attempts to send any unsent data

ñ closes socket for sending and receiving

ñ returns -1 if error

-1 is returned on error, and *errno* is set to the error number.

Read the code !

ῆ A lot of what we ask you to do is already in the code

ᾶ or at least hinted in the code and the comments

ῆ Scared of code

ᾶ look at the doxygen output and then read the code

ῆ Read the handout

ᾶ a lot of what we ask you to do is explained in English

ADDITIONAL MATERIAL

Actually `send(…)` is a system call

ñ Program control transitions to the OS

ñ Switch from user space to kernel space

ñ Implementation requires 100s of assembly instructions

ñ Much more expensive than a pure function call, like `sendall(…)`

ñ A function call is a few assembly instructions

ñ Latency cost : Remote call > > Function call

For far more details, see ECE344: Operating Systems

The ẆLatency is Zeroẇ Fallacy

’ Fallacies of Distributed Computing Explained˜

ñ In the context of remote calls a fatal misconception

ñ Latency is the time it takes to move data from client to storage server vs amount of data transferred (bandwidth)

ñ Typical WAN round trip times are 30ms+

ñ *Transfer as much data with as few calls as possible vs. many*

Misconception!

ẆThe Network is Reliableẇ

’ Fallacies of Distributed Computing Explained˜

- ñ Consider: Power supply failures, disk failures, CPU / power fan failures, A/C failures, accidents (cables get disconnected etc.), flash crowds, bugs, `H
- ñ Dependency on third party services (e.g., credit card validation, ad serving, `H)
- ñ Requires hardware and software redundancy (cost-benefit trade-off)
- ñ Loose messages (acknowledgements, time-outs & re-tries), detect duplicates (idempotent operations), re-order messages (donẠt depend on order), `H

Stream sockets in practice

- Telnet uses stream sockets
- Web browsers & HTTP use stream sockets to fetch pages
- Stream sockets are based on the Transmission Control Protocol (TCP)
- TCP ensures that data arrives sequentially and error-free (given no failures)
- *When is this kind of reliability not required?*

/etc/services

```
# This file contains port numbers for well-known services defined by
IANA
# Format:
# <service name>  <port number>/<protocol>  [aliases...]
[#<comment>]
echo            7/tcp
echo            7/udp
discard         9/tcp     sink null
discard         9/udp     sink null
systat          11/tcp    users                   #Active users
systat          11/udp    users                   #Active users
daytime                   13/tcp
daytime                   13/udp
qotd            17/tcp    quote                   #Quote of the day
qotd            17/udp    quote                   #Quote of the day
chargen                   19/tcp    ttytst source             #Character
generator
chargen                   19/udp    ttytst source             #Character
generator
ftp-data        20/tcp                      #FTP, data
ftp             21/tcp                      #FTP. control
ssh             22/tcp                      #SSH Remote Login Protocol
telnet          23/tcp
```

Client/Server

- Client
  - Issues requests to server (e.g., send & receive)
- Server
  - Starts up and listens for connections, requests, and sends/receives
- Client/Server examples
  - telnet/telnetd
  - ftp/ftpd (sftp/sftpd)
  - Firefox/Apache
- Client and server are **roles**, our following discussion also applies in **peer-to-peer** contexts
- Same application may play **both roles**

Client/Server

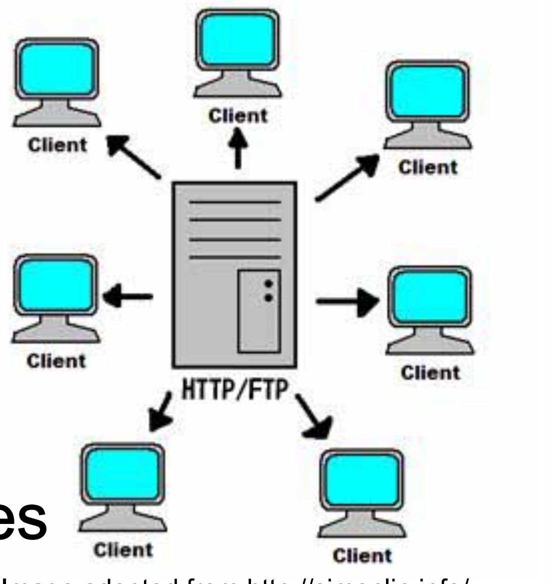


Image adopted from http://simonlin.info/

Creating a socket

```
int socket( int family,
            int type,
            int protocol);
```

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

Creates a socket Descriptor.

ῃ family
  - ᾶ AF_INET (IPv4), AF_INET6 (IPv6), AF_LOCAL (local Unix), AF_ROUTE, AF_KEY

ῃ type
  - ᾶ SOCK_STREAM (TCP) , SOCK_DGRAM (UDP), SOCK_RAW

ῃ protocol set to 0 (retrieve the default protocol for the given family and type); other values see <netinet/in.h>

ῃ Upon success returns socket descriptor, otherwise -1 indicating failure and sets errno

socket(`H) often used as follows

```
int s;
struct addrinfo *    res ;

// do the lookup
getaddrinfo("www.example.com",
            "http",
            &hints,
            &res );

s = socket(res->ai_family,
           res->ai_socktype,
           res->ai_protocol);
```

socket(..) is used by client & by server

Associate socket with port on host

```
int bind( int sockfd,
          struct sockaddr *myaddr,
          socklen_t addrlen);
```

Assigns a socket an address.

ῃ sockfd is socket descriptor from `socket(…)`

ῃ myaddr is a pointer to address struct with:

ᾶ port number and IP address

ᾶ if port is 0, then host picks port (>1023)

ᾶ IP address = INADDR_ANY

Especially when rerunning the server in development.

ῃ addrlen is length of structure

ῃ returns 0 on success, -1 on failure and sets errno

ᾶ Common is EADDRINUSE (' *Address already in use* ˜)

Example

```
struct addrinfo hints, *res;
int sockfd;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags =     AI_PASSIVE   ;

getaddrinfo(NULL, "    3490  ", &hints, &res);

sockfd = socket(res->ai_family, res->ai_socktype, res-
    >ai_protocol);

bind (sockfd, res->ai_addr, res->ai_addrlen);
```

Load up address structs with getaddrinfo():

Fill in my IP for me

Bind socket to port we passed to `getaddrinfo`

bind(`H) is used by the server

WeẠd better do error checking
error = bind(`H)

Client connecting to server

```
int connect(    int sockfd,
                struct sockaddr *serv_addr,
                int addrlen);
```

Contains destination IP address / port

- Failures indicated by the returning of -1 and the setting of errno
- E.g., bind(`H) was not called!
- Setting local port is not important
- The destination IP/port matters
- OS selects local port & conveys to remote site

connect(`H) is used by the client

Example

```
struct addrinfo hints, *res; int sockfd;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.ex ... .com", "3490", &hints,
   &res);

sockfd = socket(          res->ai_family, res-
   >ai_socktype,
                         res->ai_protocol);

connect   (sockfd, res->ai_addr, res-
   >ai_addrlen);
```

listen(…)

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

ῃ backlog is maximum number of queued up connections (often about 20)

ῃ Pattern of us

```
getaddrinfo(`H);
socket(`H);
bind(`H);
listen(`H);
/* accept(`H) goes here */
```

ῃ DonẠt ignore any failures an

listen(`H) is used by the **server**

Accepting connections

```
int accept(     int sockfd,
                struct sockaddr cliaddr,
                socklen_t *addrlen);
```

ñ Return value & arguments are similar to the above

ñ Returns brand new descriptor created by OS

ñ Former descriptor continues to listen for new connections

ñ New descriptor is used to send and receive data

accept(`H) is used by the server

Sending data

client & server

```
int send( int sockfd,
          const void *msg,
          int len, int flags);
```

-1 is returned on error, and *errno* is set to the error number.

- ñ msg is a pointer to the data to send
- ñ len is the length of the data to send
- ñ flag controls specifics, set to 0 for us
- ñ send(`H) returns the number of bytes sent, which might be less than what was requested

```
char *msg = ’    ECE297 is great fun, we learn useful stuff
int len, bytes_sent;
. . .
len = strlen(msg);
bytes_sent      = send(sockfd, msg, len, 0);
```

serẠs responsibility to send the rest

!

Receiving data

client & server

```
int recv( int sockfd,
          void *buf,
          int len, int flags);
```

ñ buf is a buffer to read data into

ñ len is the maximum length of the buffer

ñ flag controls specifics, set to 0 for us

ñ recv(`H) returns the number of bytes read

ñ recv(`H) may return 0; remoteẠs side closed connection on us

```
int close(int sockfd);
```

ñ attempts to send any unsent data

ñ closes socket for sending and receiving

ñ returns -1 if error

-1 is returned on error, and `errno` is set to the error number.

Protocols

- ñ Set of rules that specifies data transfer between computing end-points, often including
  - ã Connection establishment & tear-down
  - ã Communication
  - ã Data representation
- ñ Often based on standards, de facto standards, open specifications, `H

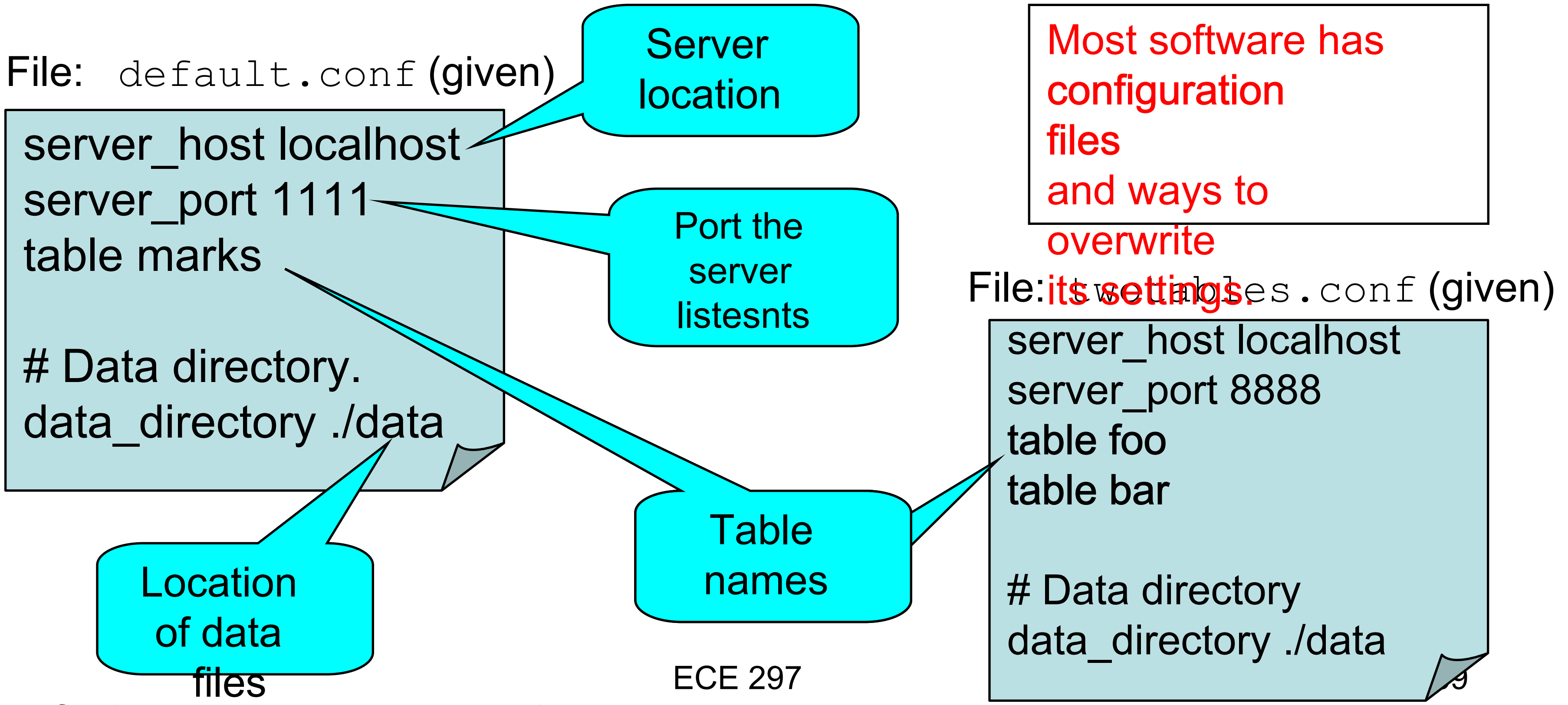
Common protocols

- IP (Internet Protocol
- UDP (User Datagram Protocol)
- TCP (Transmission Control Protocol)
- DHCP (Dynamic Host Configuration Protocol)
- HTTP (Hypertext Transfer Protocol)
- FTP (File Transfer Protocol)
- Telnet (Telnet Remote Protocol)
- SSH (Secure Shell Remote Protocol)
- SIP (Session Initiation Protocol)
- POP3 (Post Office Protocol 3)
- SMTP (Simple Mail Transfer Protocol)
- IMAP (Internet Message Access Protocol)

Configuration files

ñThe configuration file sets *properties* for the storage serve
ñIt is read at startup to initialize the storage server.
ñIs not available to client , which may reside across the netv

Most software has configuration files and ways to overwrite its settings.



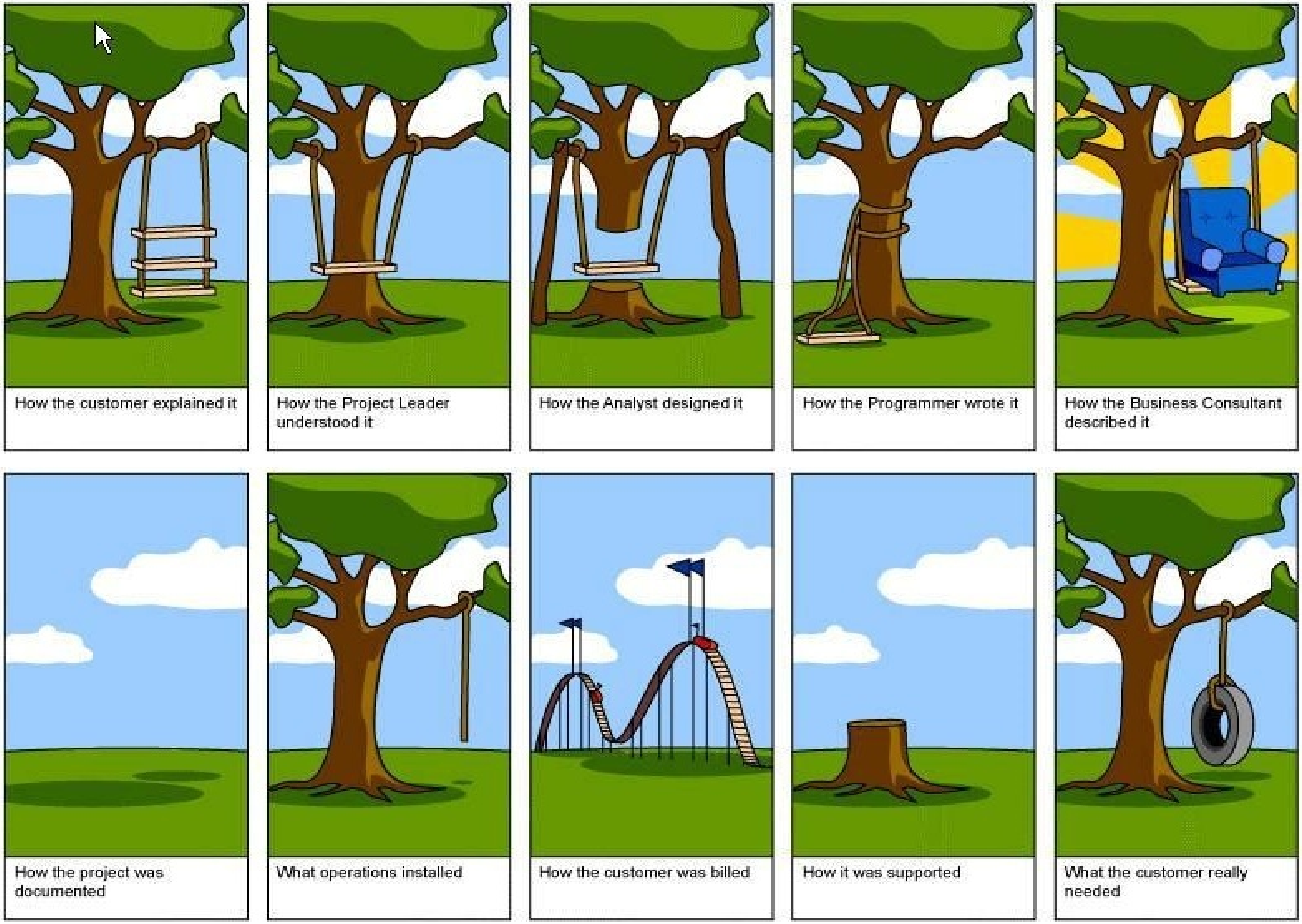
File: `default.conf` (given)

```
server_host localhost
server_port 1111
table marks

# Data directory.
data_directory ./data
```

File: `twotables.conf` (given)

```
server_host localhost
server_port 8888
table foo
table bar

# Data directory
data_directory ./data
```

For a further example see onetable.
f


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

Source: http://onproductmanagement.files.wordpress.com/2007/07/treecomicbig.jpg