UNIX Network Programming with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX Network
Programming with TCP/IP

Short Course Notes

Alan Dix

http://www.hiraeth.com/alan/tutorials

UNIX

Network Programming
with TCP/IP

Course Outline

Alan Dix
http://www.hcibook.com/alan

Session 1	Internet Basics
Session 2	First Code
Session 3	Standard Applications
Session 4	Building Clients
Session 5	Servers I
Session 6	Servers II
Session 7	Security

Three interrelated aspects:

- TCP/IP protocol suite
- standard Internet applications
- coding using UNIX sockets API

Reading

Books:

1. W. Richard Stevens, "TCP/IP Illustrated. Vol. 1: The protocols", Addison Wesley, 1994, (ISBN 0-201-63346-9).

   Explains the protocols using network monitoring tools without programming.

2. Douglas E. Comer and David L. Stevens, "Internetworking with TCP/IP. Vol.3: Client-server programming and applications BSD socket version", Prentice Hall, 1993, (ISBN 0-13-020272-X).

   Good book about principles of client/server design. Assumes you have some knowledge or at least some other reference for actual programming.

3. Michael Santifaller , translated by Stephen S. Wilson, "TCP/IP and ONC/NFS internetworking in a UNIX environment", 2nd Edition, Addison Wesley, 1994, (ISBN 0-201-42275-1).

   Covers more ground less deeply. Translation from German seems good.

4. W. Richard Stevens, "UNIX Network Programming", Prentice Hall, 1990, (ISBN 0-13-949876-1).

   A programming book. I'm waiting for a copy, but Stevens is a good writer and this book is recommended by other authors.

See also:

- your local manual pages (`man 2`)
- RFCs

Requests for comments (RFCs)

- these are <u>the</u> definition of the Internet protocols
- obtain via anonymous ftp from `sun.doc.ic.ac.uk` (`193.63.255.1`)
  login as `anonymous`
  give your email address as password
  cd to `rfc`

UNIX Network Programming with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

Session 1

Internet Basics

Session 1

Alan Dix
http://www.hcibook.com/alan

- origins
- internets and *the* Internet
- protocol layers
- addressing
- common applications

☞ using them

- TCP and UDP
- port numbers
- APIs

☞ information calls

Origins

Development of Internet & TCP/IP

1968	First proposal for ARPANET – military & gov't research Contracted to Bolt, Beranek & Newman
1971	ARPANET enters regular use
1973/4	redesign of lower level protocols leads to TCP/IP
1983	Berkeley TCP/IP implementation for 4.2BSD public domain code
1980s	rapid growth of NSFNET – broad academic use
1990s	WWW and public access to the Internet

The Internet Now

- growing commercialisation of the Internet
- 50,000 networks
- 6 million hosts
- 30 million users
- WWW dominating Internet growth

internets and *the* Internet

an internet is

a collection of

- interconnected networks
- (possibly different)

e.g. X25, AppleTalk

the Internet is

a particular internet which

- uses the TCP/IP protocols
- is global
- is hardware and network independent
- is non-proprietary

in addition

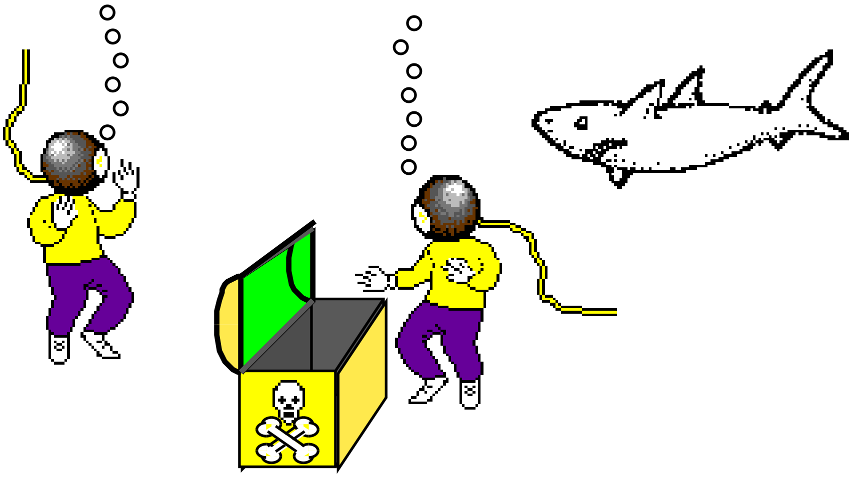
- supports commonly used applications
- publicly available standards (RFCs)

the Internet is not (just) the web !

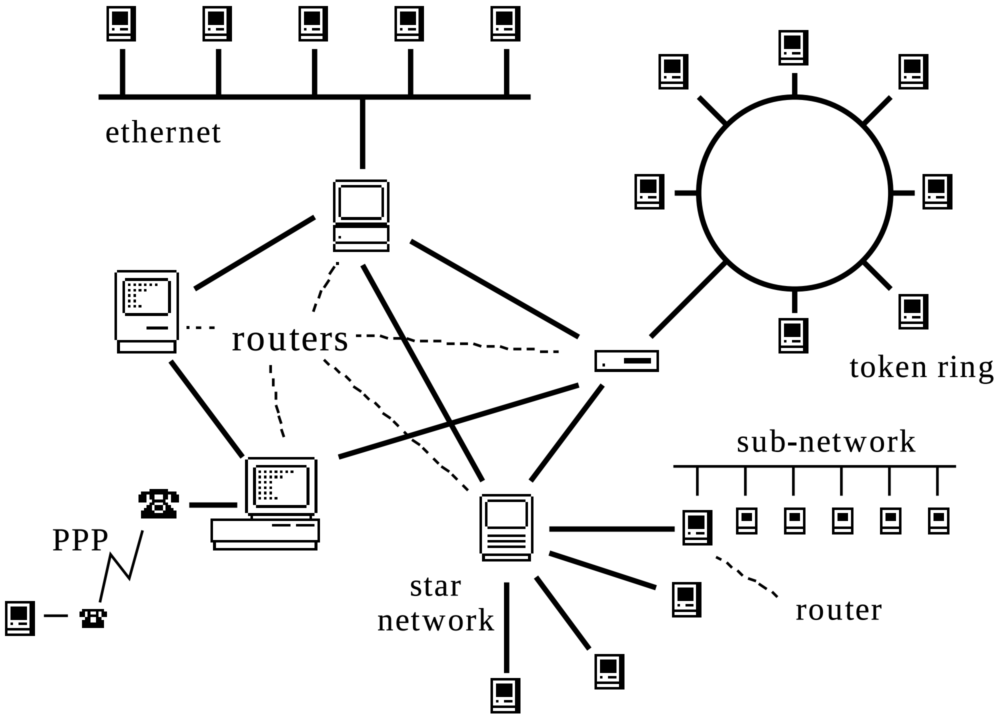
Characteristics of the Internet

To communicate you need:

- continuous connection
- common language
- means of addressing



Global Connectivity



lots of networks:

- ethernet, FDDI, token ring
- AppleTalk (itself an internet!)
- etc. etc. etc.

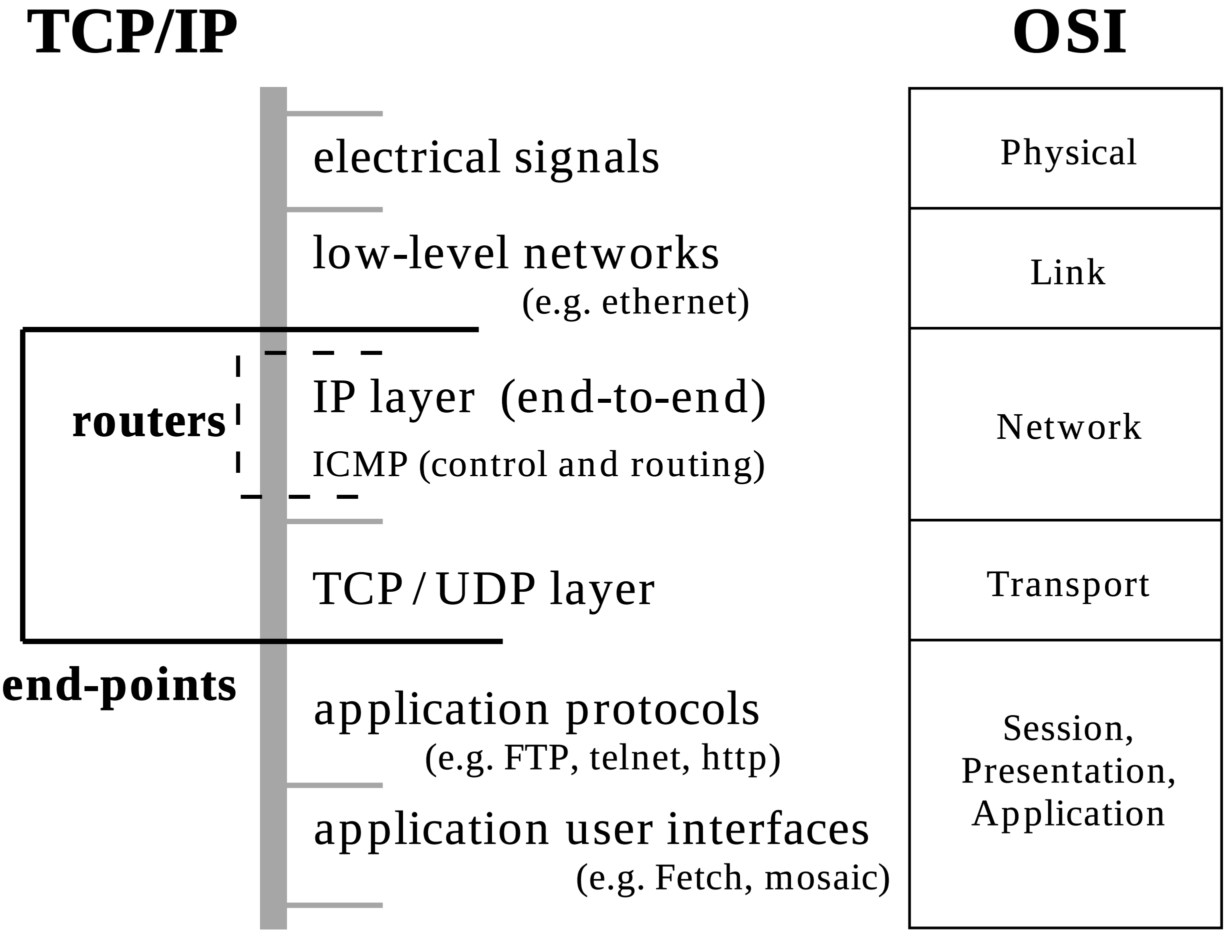
connected (possibly indirectly)

- to each other
- to the central 'ARPAnet' backbone in the US

protocols can be used in isolation

? but is it the Internet

Protocols – the Language of the Internet



TCP/IP

electrical signals

low-level networks
(e.g. ethernet)

routers

IP layer (end-to-end)
ICMP (control and routing)

TCP / UDP layer

end-points

application protocols
(e.g. FTP, telnet, http)

application user interfaces
(e.g. Fetch, mosaic)

OSI

Physical

Link

Network

Transport

Session,
Presentation,
Application

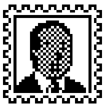
Standardisation:

- RFC (request for comments) and DoD MIL

RFCs also include (defined but not required):

- PPP, ethernet packaging, etc.
- FTP and other protocols

Addressing



J. C. T. Jennings,
Linbury Court School,
Dunhambury,
Sussex,
England,
Europe,
Eastern Hemisphere,
Earth,
near Moon,
Solar System,
Space,
near More Space[†]

Without addresses can only broadcast

Four types of address:

① location independent e.g. personal names
② physical location e.g. letter addresses
③ logical location e.g. organisational hierarchy
④ route based e.g. old email addresses

Two kinds of Internet address:

IP address – type ② (sort of)
e.g. 161.12.188.167

domain name – type ③
e.g. zeus.hud.ac.uk

[†] extract from Jennings Goes to School, Anthony Buckeridge, Collins, 1950.

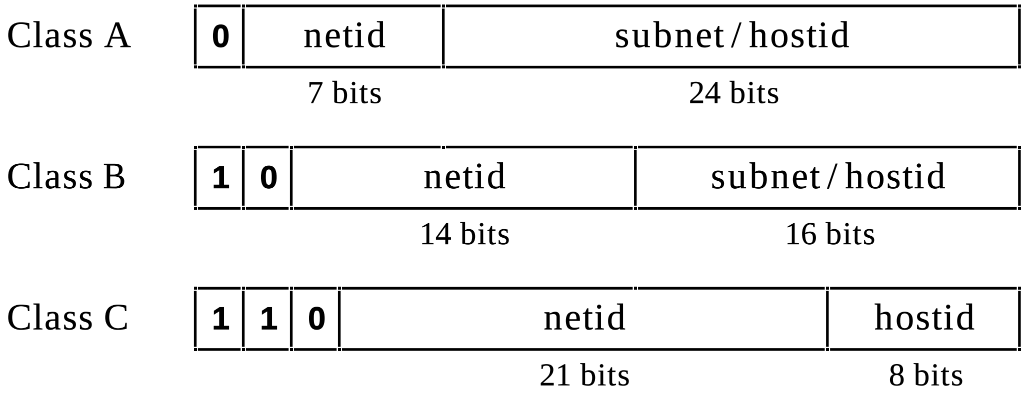
IP addresses

- 32 bit integer – 2701966503
- Often represented as 4 octets – 161.12.188.167
- General structure:

  net id { *sub-net id* } host id

- N.B. octets do not map simply onto components

Five classes of IP address:



Class D & Class E – experimental

- hostids may divided using subnet mask
  - ❍ different for each major network (netid)

    needs to be set for each machine on network

Domain names

- human readable names

  ..... or at least ASCII !

- Heirarchical (roughly organisational)

  `zeus.hud.ac.uk`

  `uk` – United Kingdom

  `ac` – academic

  `hud` – huddersfield

  `zeus` – local machine

  N.B. USA is implicit – `cs.washington.edu`

- Decentralised administration

- Mapping

  from name to IP address

  – domain name servers

  also reverse mapping

- C API :

  gethostbyname – name → IP address

  gethostbyaddr – IP address → name

Common applications

- FTP (file transfer protocol)
- SMTP (simple mail transfer protocol)
- telnet (remote logins)
- rlogin (simple remote login between UNIX machines)
- World Wide Web (built on http)
- NFS (network filing system – originally for SUNs)
- TFTP (trivial file transfer protocol – used for booting)
- SNMP (simple network management protocol)

❋ In each case protocols are defined

❋ User interfaces depend on platform
(where relevant)

Hands on

☞ connect to zeus using telenet:

```
%  telnet zeus.hud.ac.uk
login: c5
```

... etc.

☞ what happens if you just say “`telnet zeus`”?

☞ what is zeus’ IP address?

☞ try “`telnet aa.bb.cc.dd`”
(where ‘aa.bb.cc.dd’ is zeus’ IP address)

☞ connect to zeus using ftp:

```
%  ftp zeus.hud.ac.uk
```

connect as yourself and then as anonymous

Read between the lines

Network communications

Communication can be:

- Connectionless
    - ❍ address every message

      ❋ like letters
- Connection based
    - ❍ use address to establish a fixed link
    - ❍ send each message using the link

      ❋ like telephone

N.B. both need an address

some sort of system address book

or, publicly known addresses

Network communications – 2

Other issues:

- Reliability

  Do all messages arrive?

  Do they arrive in the right order?

- Buffering

  effects responsiveness

  hides potential deadlock

- Messages or byte-stream

  sent:

  write 1 (len=26): “abcde....vwxyz”

  write 2 (len=10): “0123456789”

  received:

  read 1 (len=20): “abcde....qrst”

  read 2 (len=16): “uvwxyz012...89”

  fixed length messages or prefix with length

IP – the fundamental Internet protocol

point to point

- between machines
- addressed using IP address

message (packet) based

unreliable

- network failures
- router buffers fill up

dynamic routing

order may be lost

heterogeneous intermediate networks

fragmentation

TCP & UDP

Both

- built on top of IP
- addressed using port numbers

⇒ process to process

(on UNIX platforms)

TCP

- connection based
- reliable
- byte stream

used in: FTP, telnet, http, SMTP

UDP

- connectionless
- unreliable
- datagram (packet based)

used in: NFS, TFTP

Port numbers

- 16 bit integers
- unique within a machine
- to connect need IP address + port no

TCP

- connection defined by

  IP address & port of server
  \+ IP address & port of client

UNIX

- port < 1023 – root only
- used for authentication

  (e.g. rlogin)

How do you find them?

✔ well known port numbers

Well known port numbers

Service	Port no	Protocol	
echo	7	UDP / TCP	sends back what it receives
discard	9	UDP / TCP	throws away input
daytime	13	UDP / TCP	returns ASCII time
chargen	19	UDP / TCP	returns characters
ftp	21	TCP	file transfer
telnet	23	TCP	remote login
smtp	25	TCP	email
daytime	37	UDP / TCP	returns binary time
tftp	69	UDP	trivial file transfer
finger	79	TCP	info on users
http	80	TCP	World Wide Web
login	513	TCP	remote login
who	513	UDP	different info on users
Xserver	6000	TCP	X windows (N.B. >1023)

N.B. different ‘name’ spaces for TCP & UDP

API – the language of the programmer

Application Programmer Interfaces

Not part of the Internet standard – but very important!

A story about DOS

TCP/IP stacks supplied by different vendors

different device drivers

different APIs

chaos

APIs depend on platform:

UNIX	–	sockets (original Berkley system calls)
	–	TLI (transport layer interface)
Apple Mac	–	MacTCP
MS Windows	–	WinSock (similar to sockets)

- UNIX TCP/IP API are kernel system calls
- Mac & Windows are extensions/drivers (+DLL)

Hands on

☞ copy `skeleton.c` from `tcp` directory

☞ edit to make two programs:

`getid.c` – returns IP address of machine

`getname.c` – returns name of machine

☞ use the following C calls:

`gethostid()`

returns (lon unsigned) integer result

`gethostname(buff,len)`

returns error code

puts name into buff (maximum len bytes)

☞ if you have time, play with telnet on different ports

```
%  telnet zeus.hud.ac.uk port_no
```

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

Session 2

First Code

Session 2

Alan Dix
http://www.hcibook.com/alan

- features of sockets API
- establishing TCP connections
- simple client / server program

☞ use it

- read & write with sockets
- wrapper functions
- what they do

☞ an echo server

Sockets

- developed for Berkeley UNIX
  - ❍ recall early Berkeley TCP/IP implementation
  - ❍ first delivered with BSD 2.1

- central features
  - ❍ central abstraction - the socket - an end-point
    like an electrical connector
  - ❍ not TCP/IP specific (e.g. UNIX named pipes)
  - ❍ uses normal read/write system calls
  - ❍ sockets associated with UNIX file descriptors
    but some not for normal I/O
  - ❍ some extra system calls

- sits more comfortably with TCP than with UDP
  because of byte-stream nature of UNIX I/O

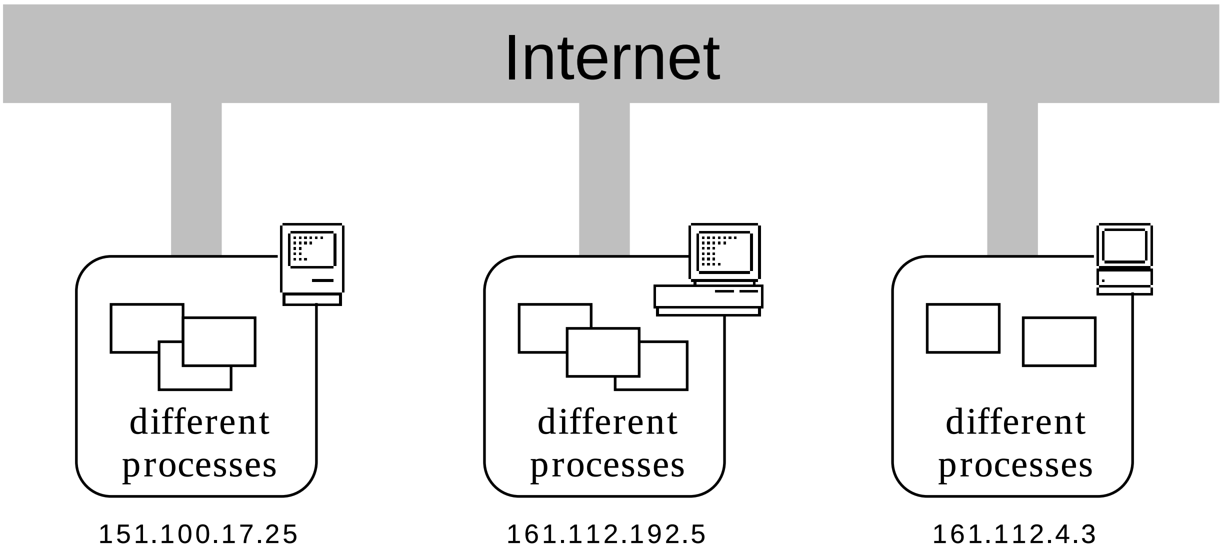
- special UDP functions

  e.g., `recv(...)` – accepts a UDP datagram

- additional non-socket functions

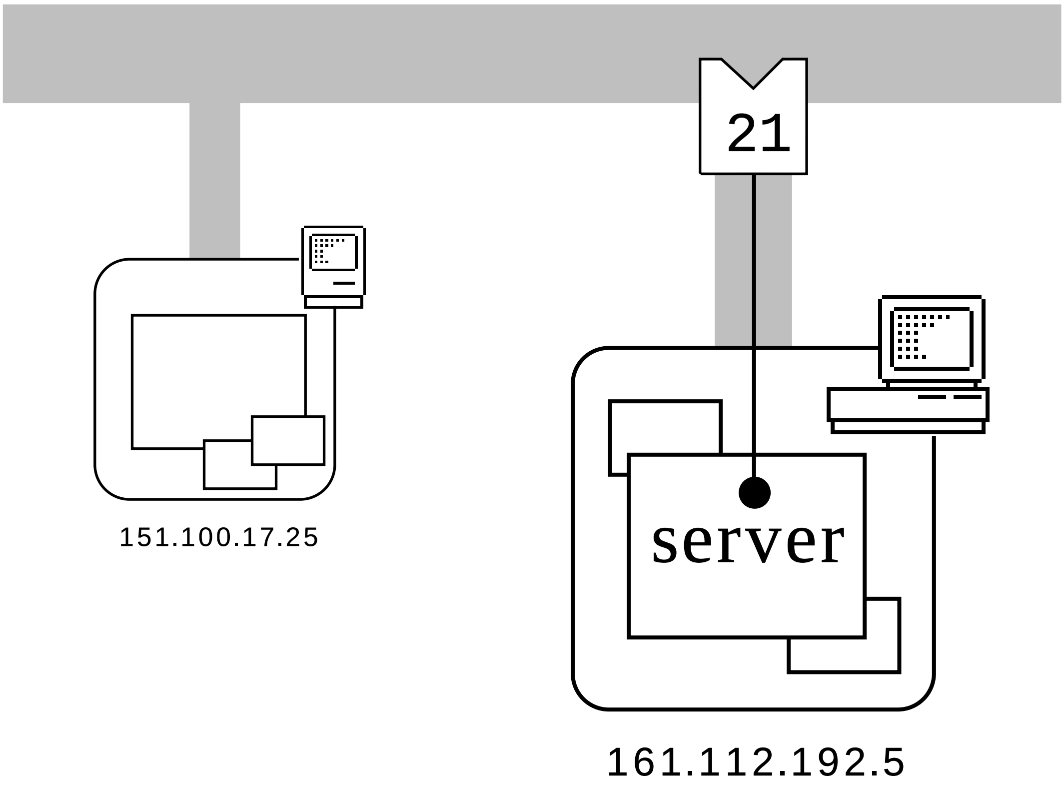
  e.g., `gethostbyname(...)` – domain name server

Establishing a TCP Connection
Initial State



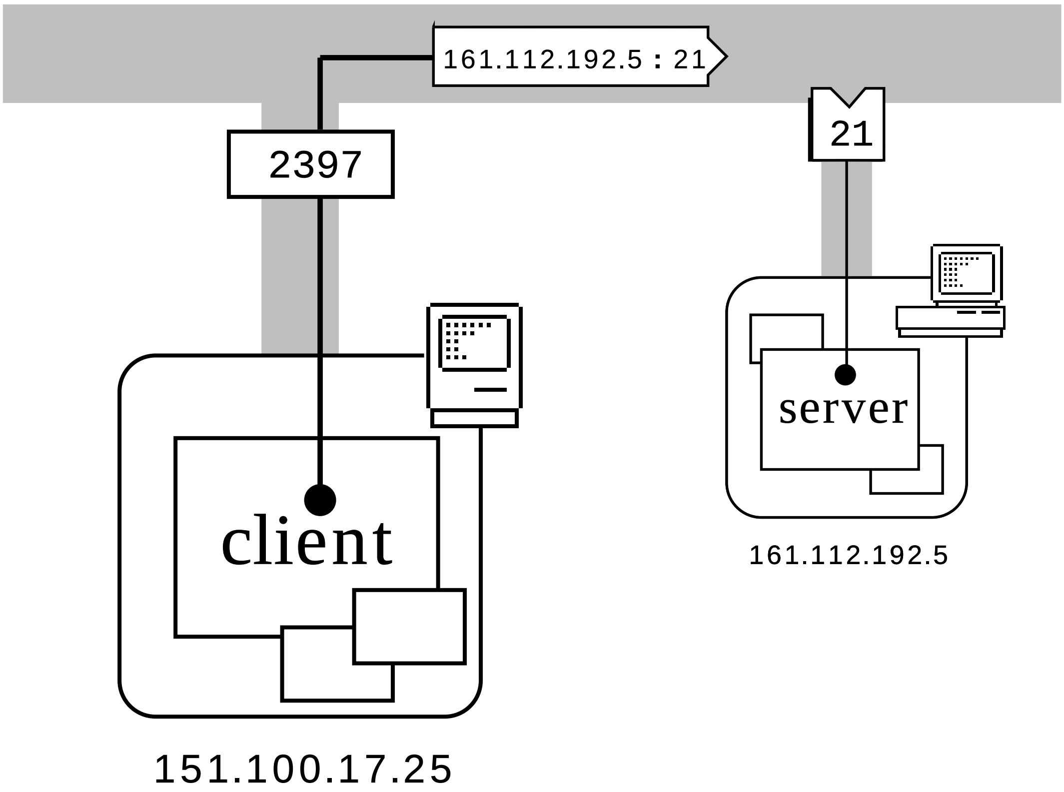
- TCP is connection based
  … establishing it is a complex multistage process
- initially all machines are the same
- no special 'server' machines
- the difference is all in the software

Establishing a TCP Connection
Passive Open



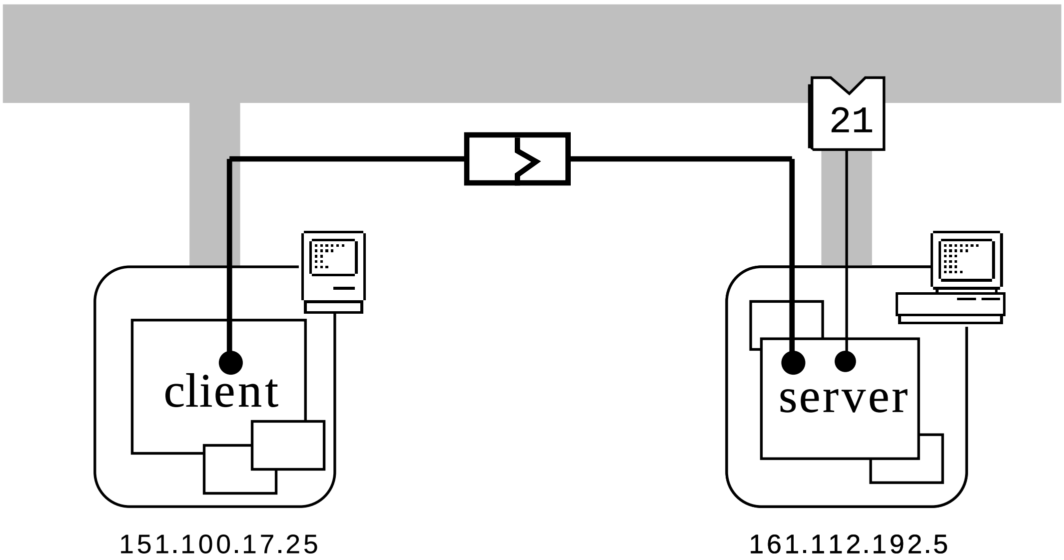
- server process does a ‘passive’ open on a port
- it waits for a client to connect
- at this stage there is no Internet network traffic
- tells the TCP layer which process to connect to

Establishing a TCP Connection
Active Open



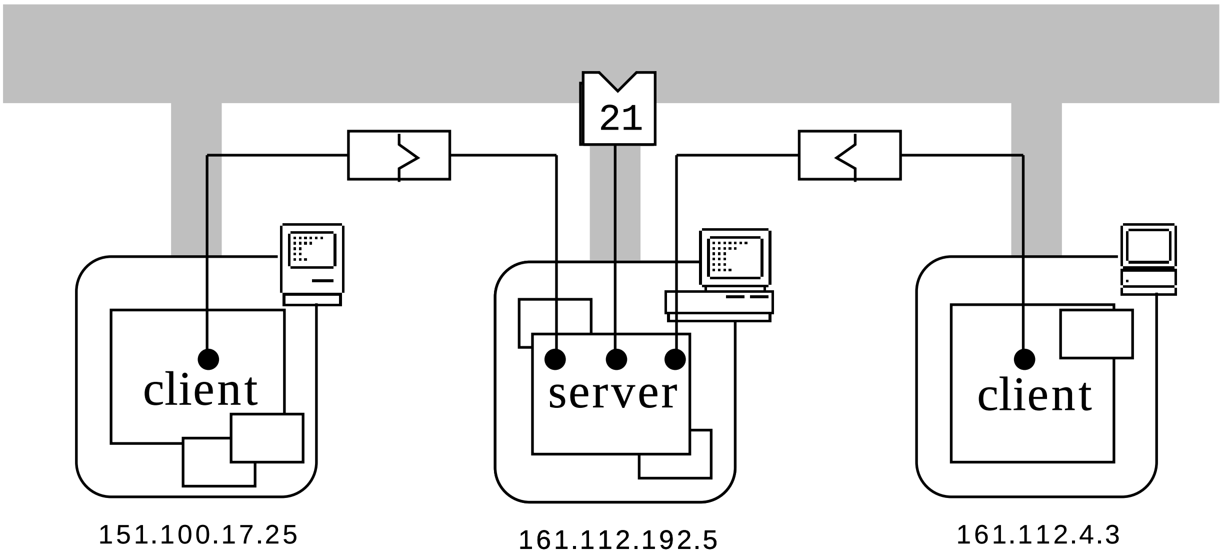
- client process usually on a different machine
- performs an ‘active’ open on the port
- port number at the client end is needed
  usually automatic (e.g., 2397)
  but can be chosen
- network message → server machine
  requests connection

Establishing a TCP Connection Rendezvous



- server side accepts and TCP connection established
- a bi-directional reliable byte-stream
- connection identified by both host / port numbers
  e.g. <151.10017.25:2397/161.112.192.5:21>
- server port is not consumed
  can stay ‘passive’ open for more connections
- like telephone call desk: one number many lines

Establishing a TCP Connection and more ...



- other clients can connect to the same port
- state for connections in the client / server only
- no information needed in the network
  not like old style relay-based exchanges
- server can restrict access to specified host or port
- server can find out connected host / port

Passive & Active Open

passive – patient but lazy
active – industrious but impatient

passive	**active**
waits for request for connection	sends out request for connection
waits for ever	times out

- normally server does passive open
  – waiting for client
- but not always (e.g. ftp)
- active opens can rendezvous …
  … but may miss due to time-outs
- either can specify local port
  but if not specified, allocated automatically

Simple client/server ‘talk’

- uses simplified calls
- server handles only one client
- strict turntaking

user 1

```
zeus:  simple-server
start up complete



client says: hi there
speak: nice day isn't it


client says: bit cold here
speak: ^D (EOF)  bye bye
zeus:
```

user 2

```

io:  simple-client -host zeus
You can send now
speak: hi there


server says: nice day isn't it
speak: bit cold here



server finished the conversation
io:
```

Server Code

establish port

```
port_sk = tcp_passive_open(port)
    /*   only done once   */
```

wait for client to connect

```
client_sk = tcp_accept(port_sk)
    /*   repeated for multiple clients   */
```

then talk to client

```
for(;;) {

    /*   wait for client’s message   */
    len = read(client_sk,buff,buf_len);
    buff[len] = '\0';
    printf("client says: %s\n",buff);

    /*   now it’s our turn   */
    printf("speak: ");
    gets(buff);
    write(client_sk,buff,strlen(buff));
    }
```

N.B. strict turn taking: client–server–client–server ...

Client Code

request connection to server

```
   serv_sk = tcp_active_open(host,port)
       /*   waits for server to accept      */
       /*   returns negative on failure     */
       /*   host  is server's machine       */
```

then talk to server

```
   for(;;) {

       /*   our turn first   */
       printf("speak: ");
       gets(buff);
       write(serv_sk,buff,strlen(buff));

       /*   wait for server's message    */
       len = read(serv_sk,buff,buf_len);
       buff[len] = '\0';
       printf("server says: %s\n",buff);
       }
```

N.B. ① opposite turn order

② no error checking!

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

☞ copy `simple-client.c` from `tcp/session2` directory

- `simple-client.c`
- `simple-server.c`
- `makefile`

☞ compile and run the programs:

- `make simple` – compiles them both
- on one machine type:

  ```
  simple-server
  ```
- on another type:

  `simple-client` *machine-name*

  where *machine-name* is the name of the first

☞ what happens if you re-run the server straight after it finishes?

☞ use the `-port` option

```
zeus:  simple-server -port 3865
io:    simple-client -host zeus -port 3865
```

☞ try a port less than 1024!

read & write

Reminder:

`ret = read(fd,buff,len)`

`int fd` – a file descriptor (int), open for reading
`char *buff` – buffer in which to put chars
`int len` – maximum number of bytes to read
`int ret` – returns actual number read

- `ret` is `0` at end of file, negative for error
- `buff` is not NULL terminated
  leave room if you need to add '\0'!

`ret = write(fd,buff,len)`

`int fd` – a file descriptor (int), open for writing
`char *buff` – buffer from which to get chars
`int len` – number of bytes to write
`int ret` – returns actual number written

- `ret` is negative for error, 0 means "end of file"
  `ret` may be less than `len` e.g. if OS buffers full
  * should really check and repeat until all gone *
- `buff` need not be NULL terminated
  if `buff` is a C string, use `strlen` to get its length

N.B. Both may return negative after interrupt (signal)

read & write with sockets

- similar to normal UNIX pipes
- bi-directional byte stream
    - ❍ read and write to same file descriptor
    - ✘ difficult to close one direction
    - ✔ special socket call `shutdown(sock,dir)`
- reading may block
    - ❍ reading from a file either:
        - (i) succeeds
        - (ii) gets end of file (`ret = 0`)
    - ❍ reading from a socket waits until
        - (i) network data received (`ret > 0`)
        - (ii) connection closed (`ret = 0`)
        - (iii) network error (`ret < 0`)
- writing may block
    - ❍ writing to a socket may
        - (i) send to the network (`ret > 0`)
        - (ii) find connection is closed (`ret = 0`)
        - (iii) network error (`ret < 0`)
    - ❍ it may return instantly
    - ❍ but may block if buffers are full

✘ BEWARE – may work during testing
then fail in use

Wrapper Functions (1)

- not real socket functions
- simplified versions for examples

```
ret = parse_network_args( &argc, argv,
                          &host, &port, &errmess )
```

scan command arguments for network options

```
port_sk  =  tcp_passive_open(port)
```

server performs passive open

```
serv_sk  =  tcp_active_open(host,port)
```

client performs active open

```
client_sk =  tcp_accept(port_sk)
```

server accepts client connection

❍ `parse_network_args` does not use socket calls

❍ the rest package one or more socket calls

Wrapper Functions (2)

```
ret = parse_network_args( &argc, argv,
                           &host, &port, &errmess )
```

- scans and edits argument list
- looks for options: `-host name` `-port nos`
- removes them from argument list
- sets the arguments `host` and `port` if options found
- set either `host` or `port` to `NULL` to disable options
- returns `0` for success
  non-zero failed – `errmess` set to appropriate message

```
port_sk  =  tcp_passive_open(port)
```

`int port` – port number to use
`int port_sk` – file descriptor of socket

① creates Internet TCP socket

```
port_sk = socket( AF_INET, SOCK_STREAM, 0 );
```

② 'binds' socket with right port and address 0.0.0.0
(special address means "this machine")

```
bind( port_sk, &bind_addr, addr_len );
```

N.B. `port_sk` is not used for normal reading and writing

Wrapper Functions (3)

`serv_sk  =  tcp_active_open(hostname,port)`

`char *hostname` – name of server's machine
`int  port` – port number to use
`int  serv_sk` – file descriptor of socket

① finds IP address of host

```
hostIP = gethostbyname(hostname);
```

② creates Internet TCP socket

```
serv_sk = socket( AF_INET, SOCK_STREAM, 0 );
```

③ 'connects' socket to appropriate port and host

```
connect( serv_sk, &bind_addr, addr_len );
```

- rendezvous with the server happens at ③
  socket `serv_sk` can then be used to talk to the server

`client_sk =  tcp_accept(port_sk)`

`int  port_sk` – file descriptor of socket

① performs raw accept call

```
client_sk = accept(port_sk, &bind_addr, &len);
```

- waits for rendezvous at ①
  when it returns `client_sk` can be used to talk to client

Special IP addresses

- `bind` call in `tcp_passive_open` uses IP address 0.0.0.0

One of several special IP addresses

0.0.0.0

- source only
- default IP address – ‘local machine’
- filled in by socket API call

127.0.0.0

- loopback address,
- also means ‘the local machine’
- usually used as recipient for local server
- doesn’t normally hit network
- N.B. can also connect to own IP address

255.255.255.255

- limited broadcast (doesn’t pass routers)

```
any netid – subnetid/hostid =  –1
any netid & any subnetid –hostid =  –1
```

- broadcast to specified net or subnet
- N.B. need to know subnet mask

Hands on

build an echo server

☞ copy `simple-server.c` and call it `echo-server.c`

☞ alter code so that instead of asking the user for input (`gets`) it simply uses the last message from the client (in `buff`)

☞ you will need to add to the makefile:

```
echo-server: echo-server.o $(MYLIBS)
	cc $(CFLAGS) -o echo-server echo-server.o $(MYLIBS)
```

↑ N.B. this must be a tab

☞ compile and run your code

☞ does your server echo everything once or twice to its terminal?

☞ the server exits after it has finished echoing
make it continue to wait for additional clients
(don’t try for two at once!)

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

Session 3

Application Protocols

Session 3

Alan Dix
http://www.hcibook.com/alan

Standard Applications

- trusted login – rlogin
- negotiating options – telnet
- world wide web– http

☞ peeking

- file transfer – ftp
- standard response codes
- electronic mail – SMTP

☞ drive it by hand

- argc , argv & makefiles

☞ build your own mail client

Types of Protocol

user character stream

- used by remote terminal/login applications
  (rlogin & telnet)
- most of the traffic is uninterpretted data
- some embedded control sequences

ascii turn-taking protocols

- includes ftp, SMTP, http
- human readable client & server messages
- usually line oriented
- turn-taking typically: client command
  server response
  . . .

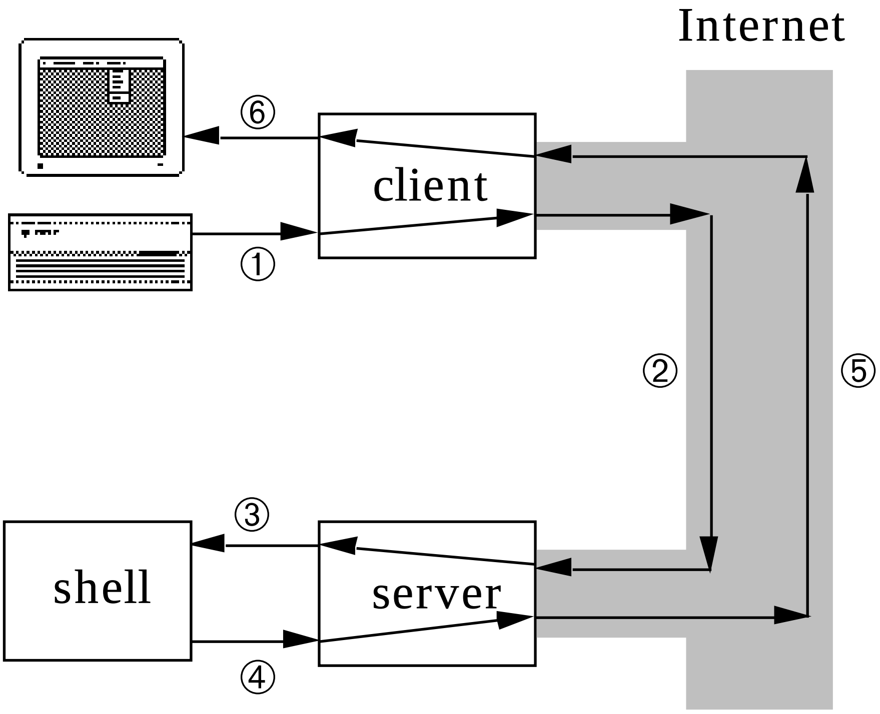
  but roles may reverse
- bulk data may be embedded (SMTP, http)
  or use separate connection (ftp)

binary protocols

- used for low level protocols:
  TCP/IP itself!
  SNMP – simple network management protocol
  NFS (built on top of RPC – remote procedure call)
- issues such as byte order important

Remote Terminal Access: rlogin and telnet

- one of the earliest Internet application areas
- the client end – interacts with the user
- the server end – shell or command interpreter



basic pattern:

① user types characters
② the client sends them to the server
③ the server passes them on to the shell
④ shell generates output
⑤ server passes output to client
⑥ client puts output on user's screen

Remote Terminals – Issues

- initialisation and authentication
  - ① how does the server know who you are?
  - ② how do you know the server is official?

  answer to ②:
  - ❍ the server is on a reserved port (<1024)
  - **N.B.** only works for UNIX servers!

- how to deal with special characters
  … including end-of-line !

- which end performs different things:
  - ❍ user flow control (crtl-S, ctrl-Q)
  - ❍ line editing
  - ❍ echoing

- how do the client and server communicate:
  - ❍ user interrupts
  - ❍ window size changes
  - ❍ who does what

- if embedded control characters are used
  what happens if the user types them?

rlogin

- simple stable protocol
- designed for UNIX–UNIX logins

  can make more assumptions

  (terminal handling, interrupts, etc.)
- authentication by ‘trusted’ hosts
  - ❍ no password required if:

    client uses port <1024

    and

    client host is in ‘`.rhosts`’ file
  - ❍ means that client must be `setuid` to `root`
- responsibility
  - ❍ echoing – server
  - ❍ flow-control – client on server request
- client–server communication
  - ❍ client → server initialisation string
  - ❍ client → server window size change:

    ctrl chars – 2 bytes of 255
    followed by window size in 2 bytes

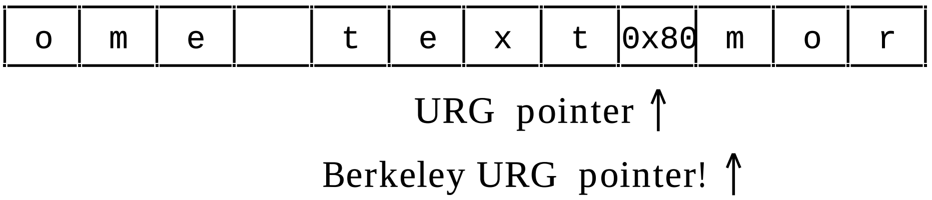
    no protection against user typing it!
  - ❍ server → client requests:

    special characters (bytes `x02,x10,x20,x80`)

    marked by URG (urgent) pointer

Urgent Data

- sometimes called out-of-band data
  - . . . but it’s not!
- data sent in normal TCP stream
- special URG pointer set
  - ❍ officially to the last byte of urgent data
  - ❍ BSD set it one beyond!



o	m	e		t	e	x	t	0x80	m	o	r

URG pointer ↑

Berkeley URG pointer! ↑

- client should:
  - ① read until urgent data reached
  - ② if necessary discard intervening data
    (e.g. if insufficient buffer space to store it)

  problem with ①
  - ❍ URG pointer says where it ends . . .
    . . . but how do you know where it <u>starts</u>?
  - ❍ have to have special codes again
- with UNIX sockets
  - ❍ send urgent data with ‘`send`’ system call
  - ❍ recipient gets a `SIGURG` signal

telnet

- cross platform more complex
- many downward-compatible options
- can be used to connect to non-login services
- client authentication
  - ❍ not in protocol – application specific
    e.g. `getty`
- responsibility
  - ❍ client may handle echoing, line editing etc.
    subject to <u>option</u> <u>negotiation</u>
- NVT character set
  - ❍ needed because cross-platform
  - ❍ 7 bit US ASCII
  - ❍ end-of-line sent as “`\r\n`” (carriage return, line feed)
  - ❍ carriage return sent as “`\r\0`”
  - ❍ also used by SMTP, ftp, finger etc.

✔ high bit free for control characters!

telnet – 2

control codes

- introduced by byte `255`
  - ❍ called: IAC – interpret as command
- following byte is actual control code

  examples:

  `255` – the actual byte `255` (needed for binary mode)
  `236` – end of file
  `241` – no op
  `243` – break

  option negotiation control codes:

  `251` – WILL
  `252` – WONT
  `253` – DO
  `254` – DONT
  `250` – sub-option begin
  `240` – sub-option end

option negotiation

- many different options:
  - ❍ echoing
  - ❍ line editing,
  - ❍ flow control
  - ❍ window size information
- client and server play “will you / wont you” to determine common protocol
- just like fax machines and modems

http

- the World Wide Web protocol

- protocol:
  - ❍ ASCII control messages
  - ❍ standard data formats for pages / images

- uses single step transactions
  - ① establish TCP connection
  - ② client sends request
  - ③ server sends reply + page
  - ④ connection closed

- why transaction based?
  - ❍ client end – many different servers
    (hypertext links to different sites)
  - ❍ server end – many clients
  - ❍ load time < interaction time (ideally!)

- why use TCP?
  - ✘ high cost of establishing connection
  - ❍ wide area, large messages & simple clients
  - ✔ reliable communication needed

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

peeking

☞ use the program `proxy` in `tcp/session3`

☞ it sits between client and server

☞ use it to see how http works:

① run: `proxy www.hud.ac.uk 80 -port 8800`
② start up Netscape using background menu
③ go to the url:
`http://www.hud.ac.uk/schools/comp+maths/private/alan/alandix.html`
④ now edit the host name in the url field
if your machine is `io`
change `//www.hud.ac.uk` to `//io.hud.ac.uk:8800`
the `8800` is to set the port number used by `proxy`
⑤ hit return and watch the `proxy` window

☞ you can do the same with telnet:

① run: `proxy zeus.hud.ac.uk 23 -port 2300`
① then: `telnet io 2300`

N.B. cannot be used for protected ports (ftp, mail etc.)

☞ try using the `-v` option of `ftp`
type:

```
ftp -v prometheus.hud.ac.uk
```

File Transfer Protocol
FTP

- used to transfer files and list directory contents

- uses two types of connection:
  control – for commands and responses
  data – for files and listings

- protocol for <u>control</u> is ascii turn-taking
  client command, server response, ...

- client commands nearly user level, including:

`USER`	user name for connection often 'anonymous' is accepted
`PASS`	password, email address for anonymous
`GET`	receive a file from remote machine
`PUT`	send file to remote machine
`CWD`	change remote directory
`LIST`	change remote directory
`PORT`	tell server what data port to use
`HELP`	info about commands supported
`QUIT`	finish session

FTP - 2
control and data

control connection

- server waits (passive open) on port 21
- client establishes connection (active open)
- client sends ascii commands – one per line
- server responds: single or multi-line response
- when required a data connection is established

data connection

- client performs a passive open on some port
  (may leave OS to determine port number)
- client tells server using control connection
  `PORT  161.112.192.5.9.93`
  port `2397` (=9*256+93) on host `161.112.192.5`

when data transfer is required

- client sends appropriate command
  e.g. `GET simple-client.c`
  then waits listening for connection
- server performs an active open on port
  then sends data
- server tells client when transfer is complete
  e.g. `226 Transfer complete.`
  then both sides (usually) close the data port

standard response codes

- ftp server replies with lines such as:

```
200 PORT command successful
```

- SMTP and some other protocols use similar codes

- three digit codes – type given by first digit:

  `1yz` — expect further reply from server
  `2yz` — all OK
  `3yz` — more required from client
  `4yz` — temporary failure (try again)
  `5yz` — error or permanent failure

- single-line response general format

```
999 a text message
```

   ↑ space here

- multi-line response

  either:

   ↓ hyphen means ‘more to come’

```
999-first line
999-one or more further lines
999 the last line
```

   ↑ space here on last line

  or

```
999-first line
  lots of lines all starting with
  at least one space
999 the last line
```

Simple Mail Transfer Protocol
SMTP

- allows:
  - ❍ mail client (user interface) to send via server
  - ❍ servers to talk to one another
    (one server takes ‘client’ role)
- note:
  - not used by user interface for receipt
  - `sendmail` is common SMTP server under UNIX
- client commands:

`HELO`	client tells server who it is
`MAIL`	initiates message and sets sender
`RCPT`	sets one of the recipients
`DATA`	says actual message content follows
`VRFY`	check that recipient exists (no mail sent)
`EXPN`	expand mail alias (no mail sent)
`RSET`	start from scratch
`EHLO`	see if server handles advanced features
`QUIT`	finish session

SMTP – 2

- authentication, servers typically:
  - do not trust `HELO`
    use reverse name mapping instead
  - do trust sender name (`From:`)
    how could they verify it?

- SMTP specifies delivery not content

- other standards used for content:
  - non-ASCII characters in headers
    `=?ISO-8859-1?Q?Alan=20Dix?=`
  - MIME for multi-part mixed content messages

- simple mail message is just:
  - header

```
From:  alan@zeus.hud.uk
To:  R.Beale@cs.bham.uk.ac
Subject:  HCI book 2E
```

  - blank line
  - body

```
Russell,
   have you heard from Prentice Hall
yet concerning the web pages?
     Alan
```

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

see what it does

☞ we want to send a mail message using raw SMTP!

☞ first of all see how ‘`mail`’ does it
cannot use `proxy` as SMTP is at port 25 (protected)

☞ instead try the `-v` option of `mail`
type:

`mail -v c3` – or whoever you want to send mail to!

see the messages from the server and the client
N.B. not all messages are shown

☞ when does mail establish the connection?
why?

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

drive it by hand

☞ use telnet to send a message
type:

```
telnet zeus.hud.ac.uk 25
```

☞ you are connected to the SMTP server on `zeus`

☞ say hello! which machine you are on

```
HELO walt.disney.com
```

did it believe you?
how does it know?

☞ now say who the message is from and who it is to

```
MAIL From:<Donald_Duck>
RCPT To:<c3@zeus.hud.ac.uk>
```

☞ next send the message

```
DATA
first line of message
..dotty
shear quackery
.
```

☞ finally say goodbye

```
QUIT
```

☞ run `mail` to see if any celebrity has sent you any

argc & argv

- recall: `int main( int argc, char **argv ) ...`
  or: `int main( int argc, char *argv[ ] ) ...`

- one of the ways to get information into a C program

- in UNIX you type:

  ```
  myprog a "b c" d
  ```

  the program gets:

  `argc = 4` – length of `argv`

  `argv[0] = "myprog"` – program name
  `argv[1] = "a"`
  `argv[2] = "b c"` – single second argument
  `argv[3] = "d"`
  `argv[4] = NULL` – terminator

N.B.
- ❍ DOS is identical (except `argv[0]` is `NULL` early versions)
- ❍ `argc` is one less than the number of arguments!

- other ways to get information in (UNIX & DOS):
  - ❍ configuration file (known name)
  - ❍ standard input
  - ❍ environment variables using `getenv()`
    or (UNIX only) third arg to main:

    ```
    main(int argc, char **argv, char **envp)
    ```

Make

‘`make`’ is a UNIX† command which:

- automates program construction and linking
- tracks dependencies
- keeps things up-to-date after changes

to use it:

- ❍ construct a file with rules in it

  you can call it anything, but ‘`makefile`’ is the default
- ❍ run ‘make’ itself

```
make  target
```

– (uses the default `makefile`)

```
make -f myfile  target
```

– (uses the rule file `myfile`)

either rebuilds the program ‘`target`’ if necessary

- each makefile consists of:
  - ❍ definitions
  - ❍ rules
- rules say how one thing depends on another
  they are either:
  - ❍ specific – e.g. to make `mail-client` do this ...
  - ❍ generic – e.g. to make any ‘`.o`’ from its ‘`.c`’ ...

† `make` is also available in many other programming environments

Makefile format

Definitions

- general form:

  variable = *value*

- example:

```
SDIR  =  tcp
MYLIBS  =  $(SDIR)/lib
```

N.B. one variable used in another's definition

- make variables are referred to later using `$`

  e.g. `$(SDIR), $(MYLIBS)`

- expanded like `#defines` or shell variables

  (some versions of make will expand shell variables also)

Rules (just specific rules)

- general form:

  target : *dependent1 dependent2 ...*
  command-line

  ↑ N.B. this must be a tab

- example:

```
myprog: myprog.o another.o
     cc -o myprog myprog.o another.o $(MYLIBS)
```

this says:

to make `myprog` you need `myprog.o` and `another.o`
if either of them is newer than `myprog` rebuild it using the
then rebuild it using the command: “`cc -o myprog` ...”

Helper Functions
standard response lines

- to make life easier!
- my own helper functions
    - ❍ to read standard response lines
      `#include "protocol.h"`
    - ❍ to interact with SMTP server
      `#include "mail-helper.h"`

```
int get_response_fd( int server_fd, int echo_fd,
                     char *buff,  int len     );
```

- reads from `server_fd`
- parses a single or multi-line response
- returns the response code (of last line)
- echoes full response to `echo_fd`
- also copies it into `buff` if non-NULL

```
int get_response_fp( FILE *server_fp, FILE *echo_fp,
                     char *buff,  int len     );
```

- similar with `stdio` files

Helper Functions – 2 for sending mail

```
int do_mail_init(int serv_fd);
```

- awaits first response and does ‘`HELO`’
- checks response and returns `0` if OK

```
int do_mail_from(int serv_fd, char *from);
int do_mail_to(int serv_fd, char *to);
```

- sends ‘`MAIL`’ and ‘`RCPT`’ commands respectively
- sender (`from`) and recipient (`to`) are C strings

```
int do_mail_data_fp(int serv_fd, FILE *user_fp);
int do_mail_data_buff(int serv_fd, char *buff, int len);
```

- send ‘`DATA`’ command and send message copied from `user_fp` or `buff` respectively

```
int do_mail_quit(int serv_fd);
```

- does ‘`QUIT`’ command

All optionally echo all exchanges to a file (or terminal) set by:

```
FILE *do_mail_set_echo_fp(FILE *new_echo_fp)
```

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

build your own mail client

☞ copy `simple-client.c` and call it `mail-client.c`

☞ copy the following from `tcp/session3`:

```
mail-helper.c
make3
```

the makefile is ready to compile your mail client
you can type (when ready!):

```
make -f make3  mail-client
```

N.B. ① SMTP obeys strict turn-taking:
server–client–server–client–server
② server starts with a return code
③ but client 'in control'

☞ modify the client code
① set default host (`zeus`) and port (`25`)
② to and from addresses:
either read in or use argv
③ message: initially read a single line
④ 'unwrap' loop to give fixed turns

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

mail client – 2

☞ resulting program structure:

(a) `read (parse) to/from addresses from user`
(b) `read message from user (gets or scanf)`
(c) `open tcp connection to mail server on correct port`
(d) `wait for server response line(s)`
(e) `say hello to server`
(f) `wait for server response line(s)`
(g) `say who the mail is from`
(h) `wait for server response line(s)`
(i) `say who the mail is to`
(j) `wait for server response line(s)`
(k) `say that data is coming`
(l) `wait for server response line(s)`
(m) `send one line message`
(n) `send line with just full stop`
(o) `wait for server response line(s)`
(p) `say goodbye`
(q) `wait for server response line(s)`
(r) `close connection`

☞ compile and run your code!

☞ if you have time modify it to send longer messages

either: change step (b) and (m) to accept long messages
or: remove step (b) and
make (m) read from user before sending each line
or: whatever you like ...

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

Session 4

Concurrent Clients

Session 4

Alan Dix
`http://www.hcibook.com/alan`

- sequential and concurrent clients
- techniques for concurrency
- call-backs
- knowing what you're doing
- callback–based client

☞ using it

Sequential Clients

e.g. FTP

1. client waits for user input
2. user types “`DIR`”
3. client performs passive open on data port (`2397`)
4. client sends “`PORT   161.112.192.5.9.93`” to server
5. client waits for standard ‘`200`’ reply line
6. if not OK then fail
7. client sends “`LIST`” to server
8. client waits for standard ‘`150`’ reply line
9. if not OK then fail
10. client reads from data port
11. client waits for standard ‘`226`’ reply line
12. if not OK then fail
13. report success to user

- client is in control

- next client action depends on:
    - ❍ what happened last

      e.g. what commend the user types
    - ❍ NOT on when it happens

Naturally Concurrent Clients

e.g. telnet

- at <u>any</u> moment either
  user may type something
  or
  output may come from server end

- client must respond whichever happens

- program a bit like:
  - ❍ when user types
    then send to the server
  - ❍ when server sends message
    then print on terminal

Concurrency for Usability

e.g. Netscape – WWW client

- basic protocol transaction based

✘. but response can be slow

✓ interaction allowed during transaction
    ➥ scrolling
    ➥ '**STOP**' button

client has to listen to
    server – more data
    user – mouse and keyboard

Programming Concurrency

Problem

- doing more than one thing at once

  listening user terminal & TCP server port

Solutions

- polling
  - ❍ use non-blocking I/O
  - ✘ keeps processor busy
- threads
  - ❍ needs built-in support (language or OS)
  - ❍ program written as several sequential parts
  - ❍ all executed at the same time
  - ❍ communicate using shared data

    (also semaphores etc.)
- event driven programming
  - ❍ low-level – e.g. UNIX select
  - ❍ event-loop – e.g., raw X and Mac
  - ❍ program paradigm – e.g. Visual Basic, HyperCard
  - ❍ call-backs – e.g., Windows, X Motif

Event Loop

Typical program structure

```
for (; ; )  {    /*  loop forever  */
     struct event_st event;
     read_event(&event);
     if ( event.type == BUTTON
             && event.target = quit_button)
         return OK;
     else if ( event.type == KEYPRESS )
         insert_char(event.char);

     else if ( event.type == INPUT_READY )
         do_network_task(event.buff);
         .  .  .
     }
```

✓ programmer in control

✘ related code gets spread out in if/case statements

- often written with sub-loops   e.g. for dialogue boxes

  unforeseen events (e.g. network I/O)
  may be delayed or even ignored!

Event–Based Languages

program = collection of event handlers

e.g. HyperCard

```
on mouseUp
      set cursor to watch
      put getServerAddress() into serverAddr
      put getUserName() into userName
      put cd fld "ToOrFrom" into toName
      put cd fld "Message" into theMess
      send "toServerSendMail"  ¬
           && quote & toName & quote & comma ¬
           && quote & userName & quote & comma ¬
           && quote & theMess & quote ¬
           to program serverAddr
end mouseUp

on AppleEvent class, id, sender
      answer "AppleEvent" && class && “from” && sender
           -- dialogue box for user
end AppleEvent
```

- ✓ concurrency naturally part of language
- ✘ network I/O not always treated uniformly

Call-backs

used in many toolkits and window mgrs:

e.g.:

- ❍ WinSock (TCP/IP under Windows)
- ❍ X Motif

General pattern

Program

① define a function

② tell toolkit to attach it to event

③ give control to the toolkit

Toolkit

❊ when event happens
call user defined function

Example – X Motif Call-backs

```
XtAddCallback( widget, callback-type, func, my-data )
```

widget	–	a widget such as a button
type	–	a callback resource name: which type of event to respond to e.g., XmNactivateCallback
func	–	pointer to C function defined by you e.g., quit_func
my-data	–	an integer or pointer to your data passed on to your callback

The callback function definition:

```
void quit_func( widget, my-data, event-data )
```

widget	–	where the event occurred
my-data	–	the integer or pointer passed in the call to XtAddCallback
event-data	–	the X event structure which caused the callback

What’s going on?

Sequential Programs

```
for ( ; ; ) {    /*  N.B. pseudo-C !!!  */
     gets(command);
     if ( . . . )
     if (  command is "quit"  )  {
         char response[MAX_LINE_SIZE+1];    ← ②
         write(serv_sd,"QUIT\n",5);
① ☞      read(serv_fd,response,MAX_LINE_SIZE);
         if ( response[0] != '2' )  . . .
         printf("session complete\n");
         exit(0);
         }
     if ( . . . )
     }
```

features for free

① program counter (☞)
  – what you are doing

② local variables
  – what you are doing it to

What’s going on? - 2

sequential → concurrent

implicit → explicit

- local variables

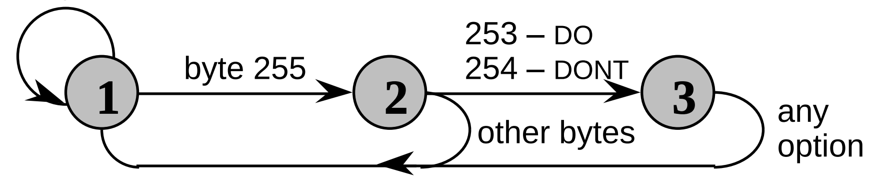
  → global variables
  or dynamic data structures

  e.g. partial line of user input

- program counter

  → mode variable
  or finite state machines!

  e.g. TELNET command sequences
  server output modes:
  - ① normal echoing
  - ② waiting for command
  - ③ waiting for option



Callback based client – 1

① Initialisation

```
main(...) {
        /*   request connection to server    */
    sd = tcp_active_open(host,port)
        /*   set-up callback for server    */
    inform_input(sd,read_socket,NULL);
        /*   set-up call-backs for interface    */
    ...
        /*   give control to toolkit    */
    inform_loop();
}
```

② When server sends a message ...
... `read_socket` is called

```
read_socket( int sd, ... ) {
        /*   read server’s message    */
    len = read(sd,buff,buf_len);
        /*   process message             */
        /*   probably update interface   */
}
```

Callback based client – 2

③ When user does something ...
... appropriate function is called

```
term_line( int fd, void *id, char *buff ) {
        /*   process interface event   */
    mess("sending {%s}\n",buff);
        /*   possibly send message to server   */
    write(sd,buff,strlen(buff));
}
```

step ① once at initialisation
steps ② & ③ any number of times in any order

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

an electronic conference

☞ copy the following from `tcp/session4`:

```
client.c
server.c
make4
```

the makefile is ready to compile, type :

```
make -f make4  conf
```

☞ one person run the server:

```
io:  server
```

☞ two or more others run the client:

```
other:  client -host io
```

N.B. you cannot participate from the server
to join in launch a client in another
window of the server’s machine

UNIX

Network Programming with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

Session 5

Server Design

Session 5

Alan Dix
http://www.hcibook.com/alan

- types of server
- handling server concurrency
- server state
- stateless servers
- when things go wrong!
- survival – the 3 Rs
- callback–based server

☞ modify server

Servers

Kinds of server

① transaction based

e.g. database: 1 query → 1 result

② strict turn-taking

e.g. ftp

③ inherent concurrency

e.g. electronic conferences, MUDs

for lots of clients either:

- serve one at a time in turn
  - ① may be slow
  - ② may take forever!
- serve several at the same time

both require concurrency

Server Concurrency

- similar solutions to client
  - ❍ polling
    - ✓ acceptable if machine dedicated to server
  - ❍ threads
  - ❍ UNIX select
  - ❍ event driven
    - ✘ less likely to run in event-based system
    - ✓ some web based servers do

- in addition:
  - ❍ when no intrinsic concurrency
  - ❍ can use UNIX fork
    - ✓ launch separate process to serve each client so each is simpler
    - ✓ uses standard UNIX process concurrency
    - ✘ can be expensive (process creation) especially with lots of small transactions

Server State

- concurrent server needs to remember
  - ❍ how many clients
  - ❍ state of their connection
  - ❍ state of each transaction / protocol

    etc. etc. etc.

✘. many clients   large state

✘. disaster scenarios
  - ❍ client establishes connection
  - ❍ client crashes
  - ❍ client restarts
  - ❍ client establishes a new connection
  - ❍ it crashes again ...

✓ solution – no state

Stateless Servers

stateless = no per client state

- for transaction based services
  - ❍ client makes request
  - ❍ server performs action
  - ❍ server returns result

- really only possible with UDP

e.g. http – transaction based, but uses TCP

may need several reads for request
need to store partially filled buffer ...

N.B. in general, buffers part of the per client state

✘ not all plain sailing ...

- ❍ clients have to maintain more state
- ❍ requests more complex (no context)
- ❍ unreliable protocol

transactions must be idempotent
time-outs for lots transactions ...

When things go wrong

PC crash	one sad user
server crash	lots of angry users

- take special care with servers!

- probability of failure:

  clients – prob. of failure = p
  server – prob. of failure = q
  n clients and only 1 server, so:
  probability of some failure ≈ np+q

- good news!
  - ❍ server failure less likely (or is it?)

- bad news!
  - ❍ servers are more complex (q > p)
  - ❍ what if client brings server down?

Causes of failure

① hardware failures
② programming errors
③ unforeseen sequences of events
④ system does not scale

Large number of components

① more frequent

Complexity of algorithms

② more likely

Interleaving and delays

③ difficult to debug

Limited testing conditions

④ unexercised

Survival

Network or server failure
standard solutions

Client fails — three **R**s for server

- **robust**

  server should survive
  - never wait for response from client
  - non-blocking network I/O
- **reconfigure**

  detect and respond to failure
  - time-out or failure of I/O operations
  - reset internal data structures
  - inform other clients
- **resynchronise**

  catch up when client restarts
  - similar to new client
  - N.B. client may not know (network)

Software faults

Defensive programming

- inconsistent client/server data structures

Use simple algorithms

- fixed sized structures – but check bounds!
- may conflict with scaleability – document

Verify

- close hand checks
- for production code – formal methods

Unforeseen sequences of events

- deadlock – never use blocking I/O
- never assume particular orders of events
- back-to-back messages

  network packet ≠ logical message

Debugging and testing

- logging – to reproduce failure
- random data – at interface or network
- ask your friends

Callback based server – 1

① Initialisation

```
main(...) {
        /*   establish port    */
     pd = tcp_passive_open(port)
        /*   set-up callback for port    */
     inform_input(pd,accept_client,NULL);
        /*   give control to notifier    */
     inform_loop();
}
```

② When client requests connection ...
... notifier calls `accept_client`

```
accept_client(...) {
        /*   accept client’s connection    */
     fd = tcp_accept(port_fd);
        /*   record connection details    */
     client_fd[count] = fd;
        /*   set-up callback for client    */
     inform_input(fd,read_client,count);
        /*   keep track of number of clients    */
     count = count+1;
        /*   probably tell other clients also    */
}
```

Callback based server – 2

③ When client sends message ...
... notifier calls `read_client`

```
read_client( c_fd, id ) {
        /*   read client's message   */
     len = read(c_fd,buff,buf_len);
        /*   broadcast to other clients   */
     for( c=0; c<client_count; c++) {
        if ( client_fd == c_fd ) {
          /*   special reply for sender   */
        }
        else {
          /*   relay message to other clients   */
        }
}
```

N.B. step ① performed <u>once</u> at initialisation
steps ② & ③ happen any number of times ...
... in any order

- similar to client code, but with extra ‘accept’ stage.

My window-less callbacks – 1

- so you can experience the pain of callbacks without the added pain of windows ...

```
#include "inform.h"
```

```
int inform_input( int fd,  inform_fun f,
                           inform_id id   );
```

  - function `f` is your callback
  - `f` is called when a buffer can be read from `fd` ... without blocking
  - the identifier `id` is also passed to `f`

```
int inform_output( int fd,  inform_fun f,
                            inform_id id   );
```

  - similar to `inform_input` but for output
  - `f` is called when a buffer can be written to `fd`

```
int inform_loop( );
```

  - gives control to the 'notifier' which performs callbacks for you

My window-less callbacks – 2

```
#include "line_by_line.h"

int inform_line_by_line( int fd,  line_fun line_f,
                         eof_fun eof_f, id_type id );
```

- the file `fd` is monitored by notifier
- two callbacks: `line_f` and `eof_f`
- `line_f` is called when a complete line is read
- `eof_f` is called when the end of file is reached

```
#include "monitor.h"

struct mon_tab_struct monitor_tab[] = {
    { 0, "command", callback, "description" },
    { 0, 0, 0, 0 }
};

int perform_line( char *buff );
```

- helper for simple command interface
- you make `monitor_tab` with suitable functions
- the first word in `buff` is regarded as a command
- it is looked up in `monitor_tab`

  . . . and the relevant callback is run

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

❇ the conference server is not very friendly
it refers to everyone by number
you are going to make this better!

☞ copy `server.c` call it `new-server.c`

☞ edit the makefile `make4` so that you can compile `new-server.c` by using:

```
make -f make4  new-conf
```

☞ locate the place where the server first establishes contact with the client.

☞ make the server wait for a line (or buffer) of input from the client (the clients name)

☞ modify the notification message it sends to all the clients to make it name the user

☞ compile and run (use the same client)
run several clients, do you notice delays?

❇ Harder bits

☞ add the user name to the per-client data structure

☞ alter the server so that all messages use the name rather than client number

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

UNIX
Network Programming
with TCP/IP

Session 6

Forking Servers & more TCP/IP

Session 6

Alan Dix
http://www.hcibook.com/alan

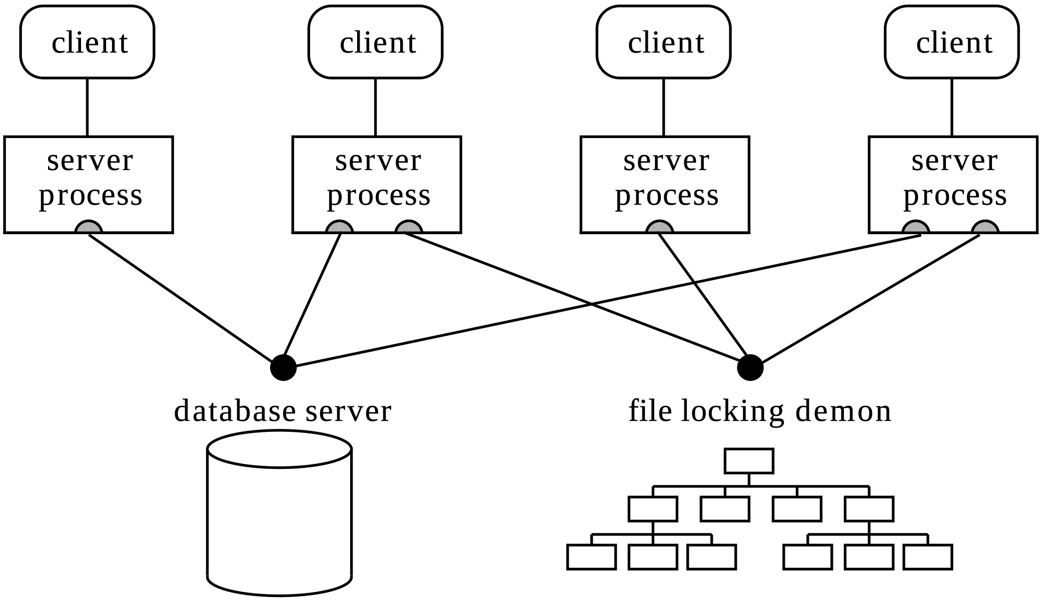
Forking Servers & TCP/IP behaviour

- UNIX processes and fork
- forking servers
- fork system call
- example code
- dup, exec and wait
- ☞ remote shell
- inet demon and remote login
- ☞ another echo server
- IP fragmentation
- TCP flow control

Loosely coupled services

- closely coupled:
  strong client interaction
  e.g. electronic conference
- loosely coupled:
  little or no client interaction
  e.g. WWW
- no interaction at all
  separate process to serve each client
- weak interaction
  need locking, database server etc.

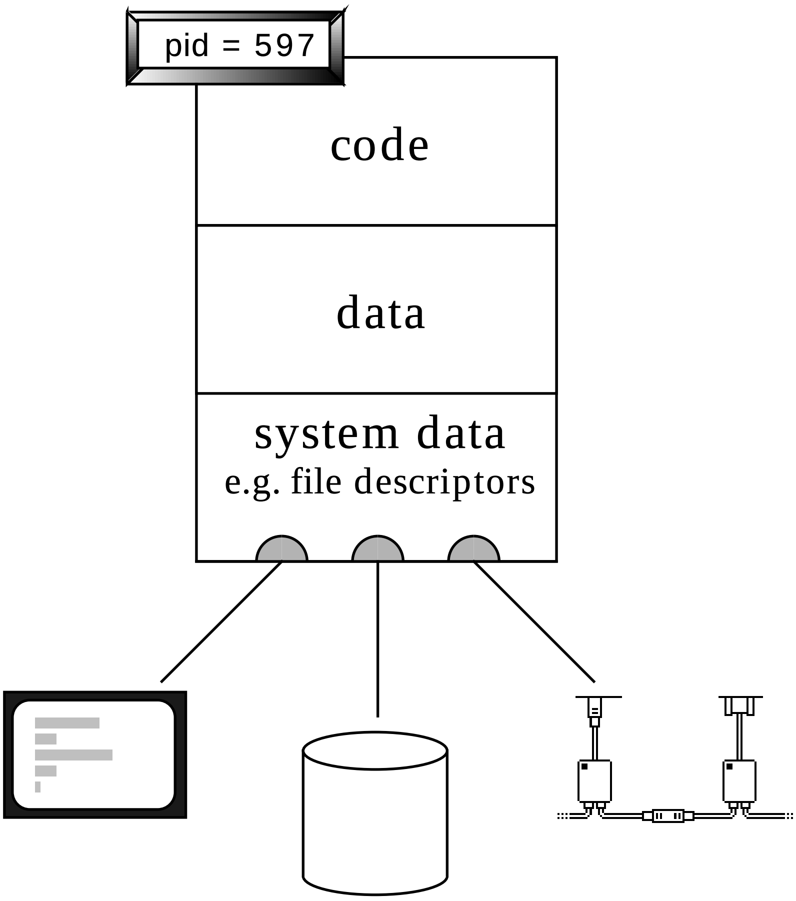
  i.e. some central point of control



A UNIX process

UNIX process:

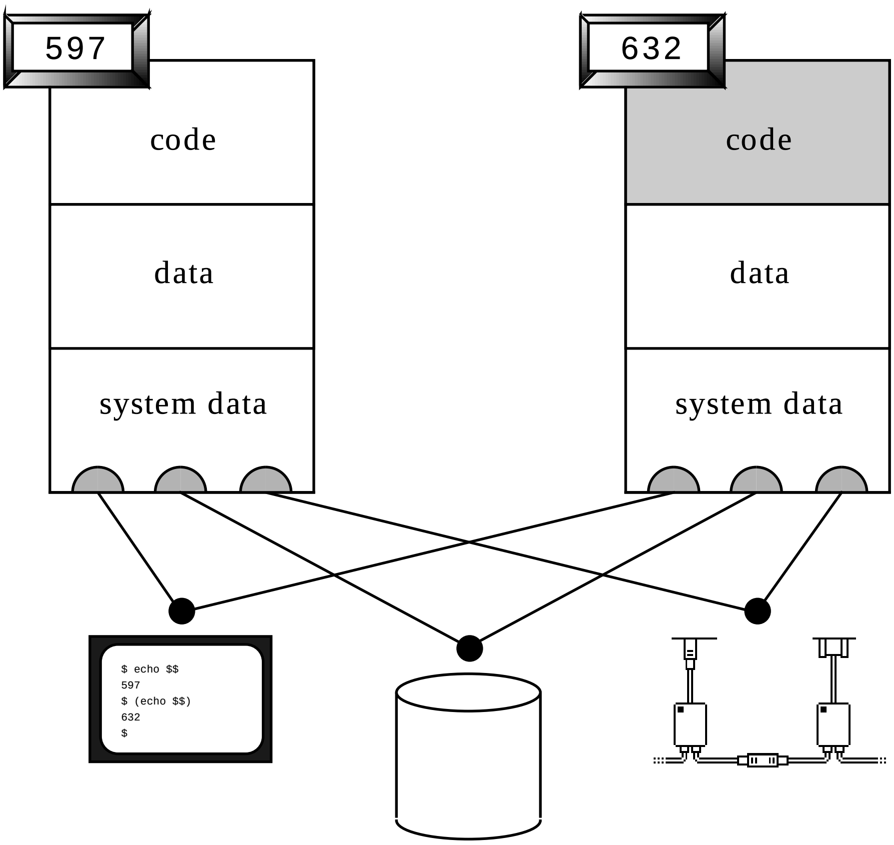
- identified by process id (pid)
- process includes:
    - ❍ program code
    - ❍ application data
    - ❍ system data
        - ❋ including file descriptors



Forking

UNIX 'fork' duplicates process:

- copies complete process state:
  - ❍ program data + system data
  - ❍ including file descriptors
- code immutable – shared



```
$ echo $$
597
$ (echo $$)
632
$
```

Forking – 2

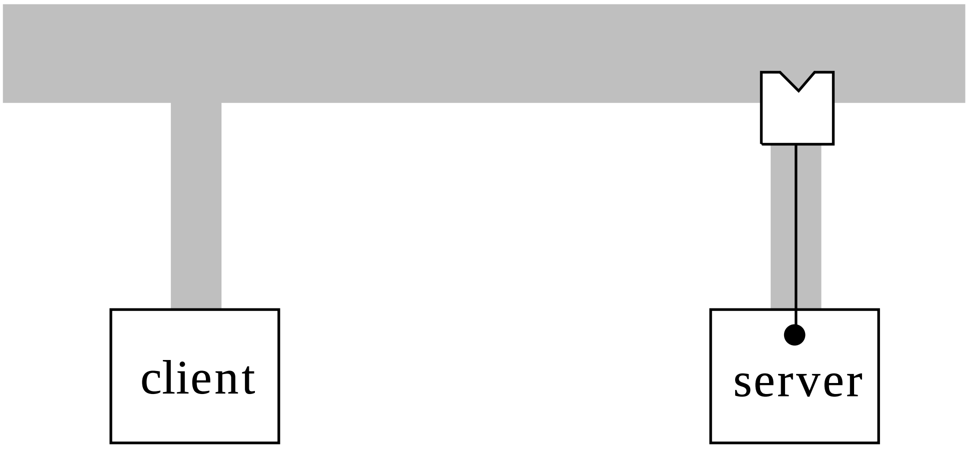
- old process called the parent
- new process called the child
- process ids
  - ❍ allocated sequentially
  - ❍ so effectively unique
    (but do wrap after a very long time)
- finding process ids
  - ❍ at the shell prompt:
    use '**ps**'
  - ❍ in a C program:
    use '**int p = getpid();**'
  - ❍ in a shell script:
    use '**$$**'

N.B. useful for naming temporary files:

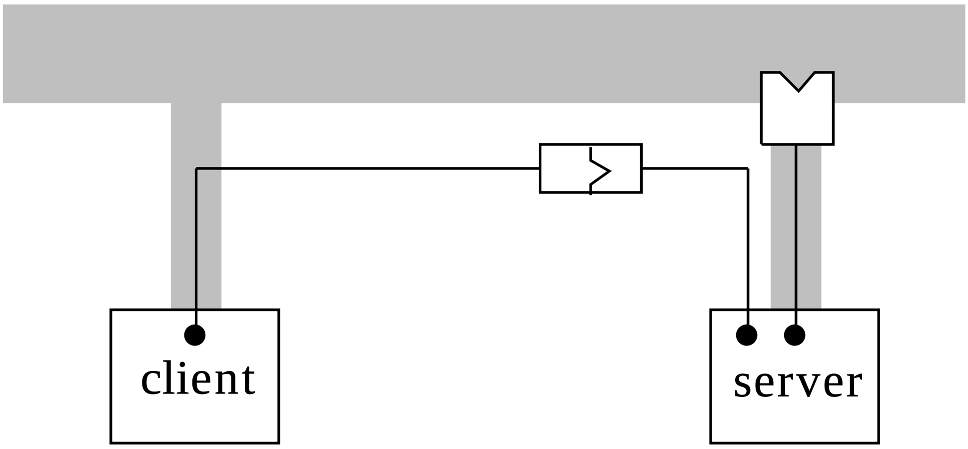
```
tmpfile = "/tmp/myfile$$"
```

Use in servers

① the server passive opens a port
and waits for a client

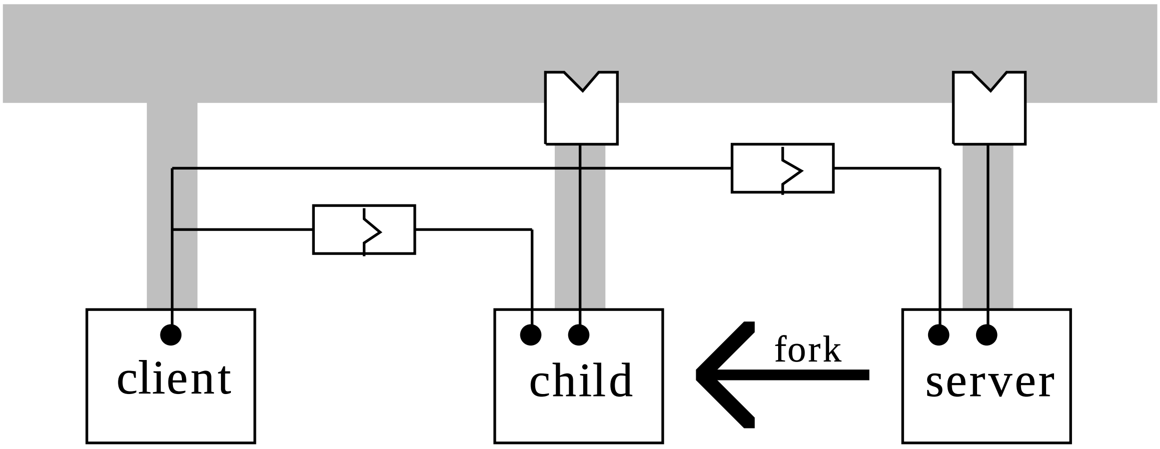


② the client performs an active open
a connection is established



Use in servers – 2

③ the server forks a child



- child is a copy of the server

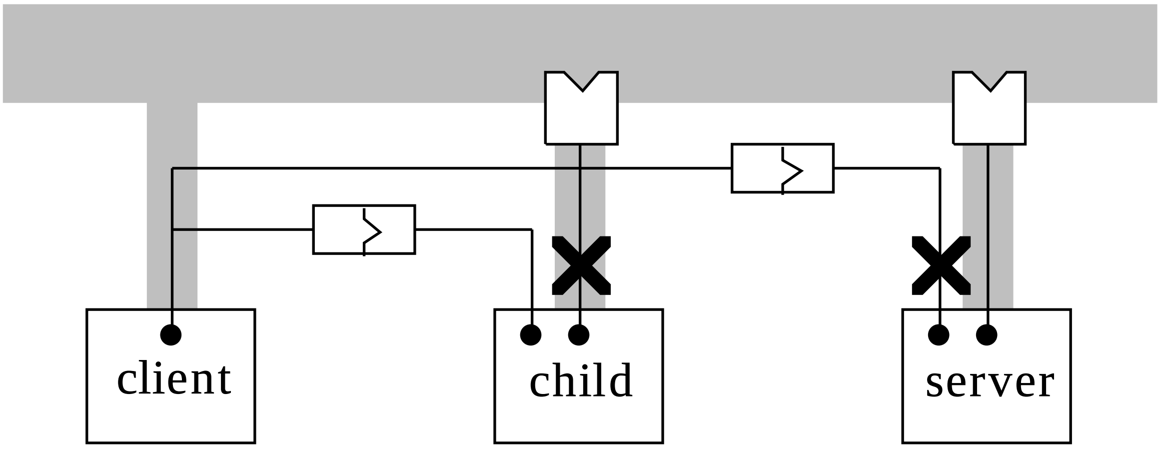
- both socket connections are duplicated

server waiting on port . . .
. . . and child waiting on port

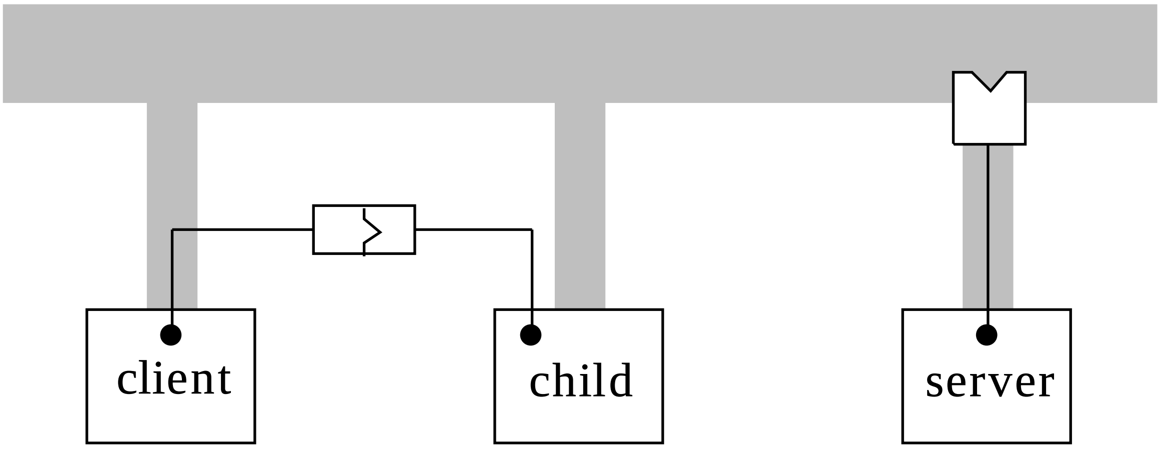
child connected to client . . .
. . . and server connected to client

Use in servers – 3

④ server closes the connection
child closes the passive port



⑤ server waits for further connections
child talks to client



Fork system call

```
pid_t  p  =  fork();
```

(`pid_t ≈ int`)

- if successful
  - ❍ process
  - ❍ successful fork returns:

    0 – to child process

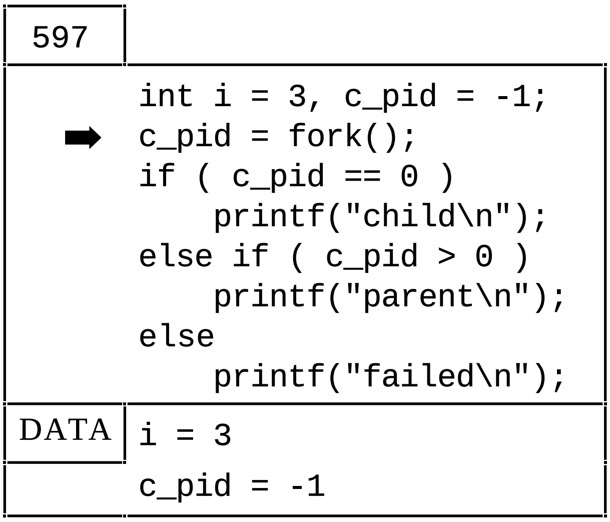
    child pid – to parent process

    parent and child are different!

- negative result on failure

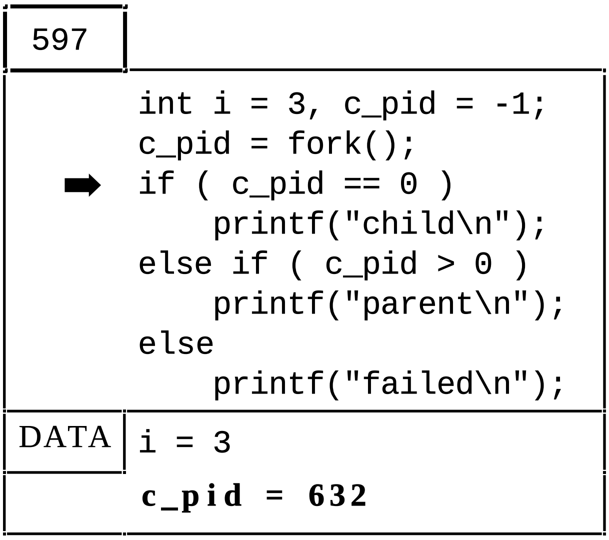
Execution – 1

- parent forks

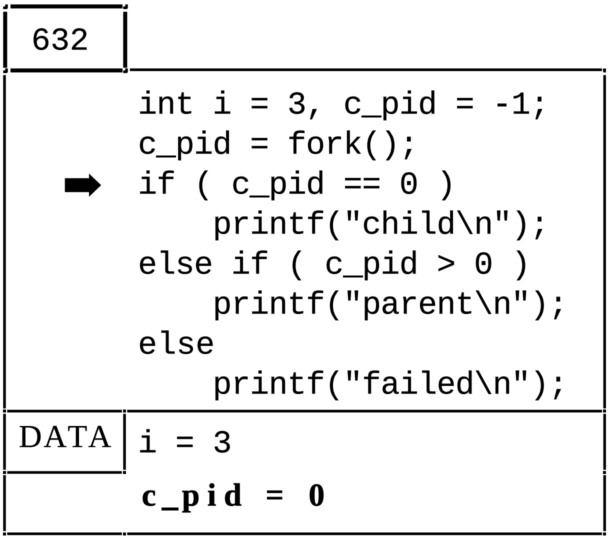


```
597

   ➡  int i = 3, c_pid = -1;
      c_pid = fork();
      if ( c_pid == 0 )
          printf("child\n");
      else if ( c_pid > 0 )
          printf("parent\n");
      else
          printf("failed\n");
DATA  i = 3
      c_pid = -1
```

- after fork parent and child identical



```
597

      int i = 3, c_pid = -1;
      c_pid = fork();
   ➡  if ( c_pid == 0 )
          printf("child\n");
      else if ( c_pid > 0 )
          printf("parent\n");
      else
          printf("failed\n");
DATA  i = 3
      c_pid = 632
```

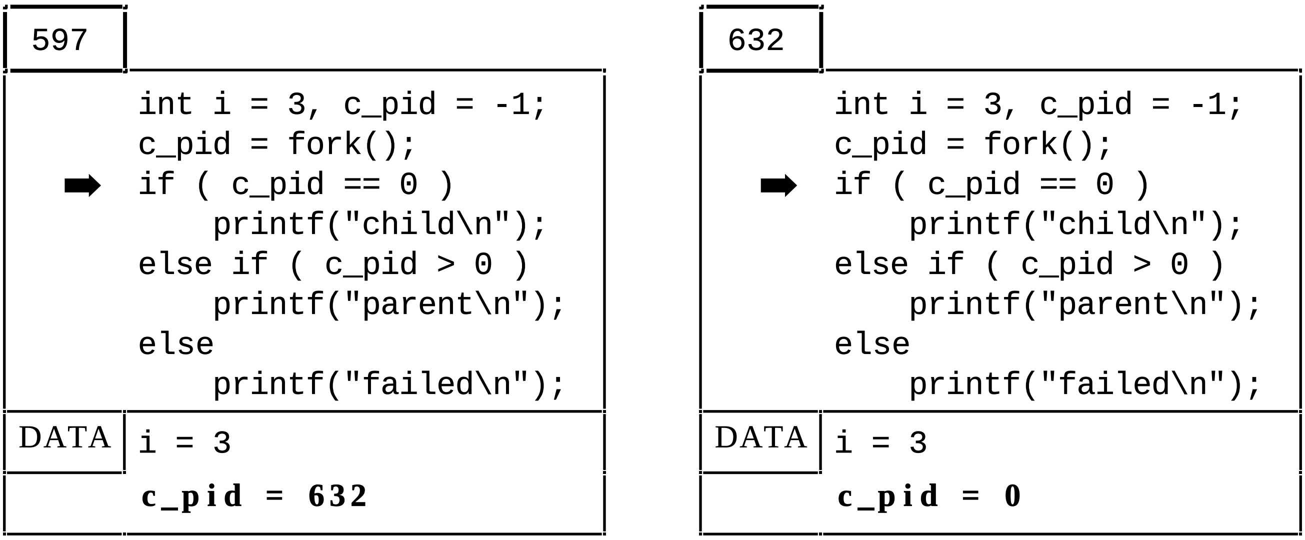


```
632

      int i = 3, c_pid = -1;
      c_pid = fork();
   ➡  if ( c_pid == 0 )
          printf("child\n");
      else if ( c_pid > 0 )
          printf("parent\n");
      else
          printf("failed\n");
DATA  i = 3
      c_pid = 0
```

- except for the return value of fork

Execution – 2

- because data are different



597

```
      int i = 3, c_pid = -1;
      c_pid = fork();
  ➡   if ( c_pid == 0 )
          printf("child\n");
      else if ( c_pid > 0 )
          printf("parent\n");
      else
          printf("failed\n");
```

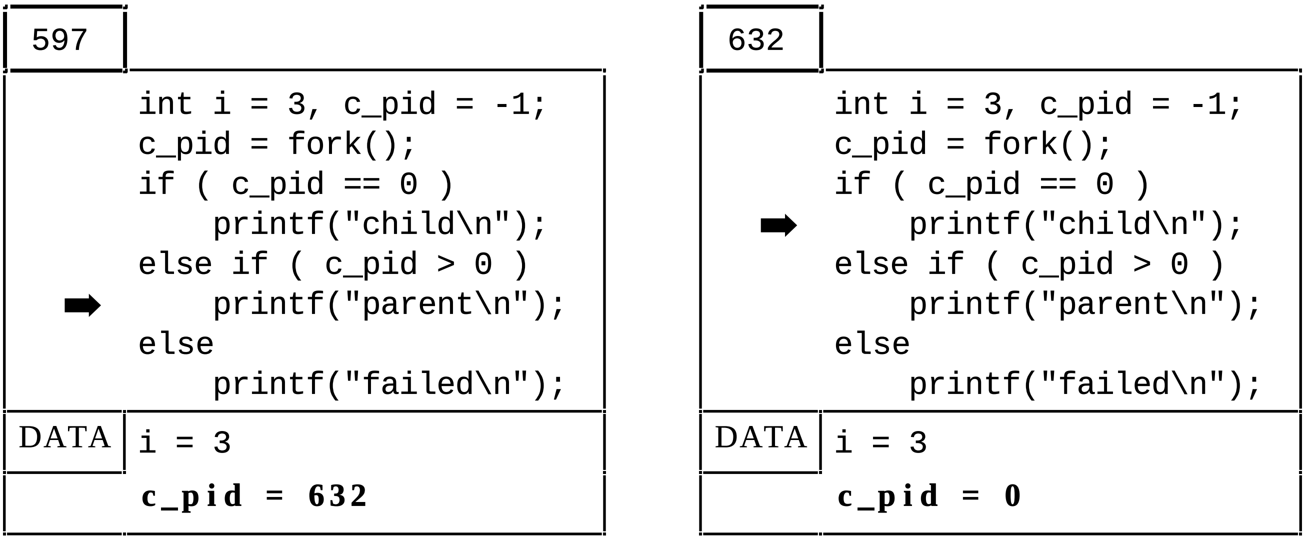
DATA
i = 3
c_pid = 632

632

```
      int i = 3, c_pid = -1;
      c_pid = fork();
  ➡   if ( c_pid == 0 )
          printf("child\n");
      else if ( c_pid > 0 )
          printf("parent\n");
      else
          printf("failed\n");
```

DATA
i = 3
c_pid = 0

- program execution differs



597

```
      int i = 3, c_pid = -1;
      c_pid = fork();
      if ( c_pid == 0 )
          printf("child\n");
      else if ( c_pid > 0 )
  ➡       printf("parent\n");
      else
          printf("failed\n");
```

DATA
i = 3
c_pid = 632

632

```
      int i = 3, c_pid = -1;
      c_pid = fork();
      if ( c_pid == 0 )
  ➡       printf("child\n");
      else if ( c_pid > 0 )
          printf("parent\n");
      else
          printf("failed\n");
```

DATA
i = 3
c_pid = 0

- so parent and child behaviour diverge

fork based shell server – 1

Basic structure:

- establish port
- loop forever
- on each loop:
  - ❍ accept a single client connection
  - ❍ fork a child to manage client
- child execs a copy of the shell

N.B. no login – very insecure !

① Main loop

```
main(...) {
        /*  open port                          */
    port_sk = tcp_passive_open(port)
        /*  loop forever accepting clients  */
    while ( accept_one(port_sk) > 0 );
        /*  on error close and exit         */
    close(port_sk);
    exit(0);
}
```

fork based shell server – 2

② Process each client in turn

```
accept_one( int port_sk ) {
        /*  accept a single connection       */
    client_sk = tcp_accept(port_sk);
        /*  perform fork                     */
    child_pid = fork();
```

- child gets zero return from fork

```
    if ( child_pid == 0 ) {
            /*  child closes passive port                 */
        close(port_sk);
            /*  then starts its own behaviour             */
        exec_a_shell(client_sk);
        }
```

- parent gets child process id returned from fork

```
    else if ( child_pid > 0 ) {
            /*  parent closes client socket               */
        close(client_sk);
            /*  N.B.    child has open descriptor         */
            /*          so client is not cut off          */
            /*  returns child pid to main loop            */
        return child_pid;
        }
```

- negative result on failure

```
    else  return 0;
}
```

fork based shell server – 3

③ Child `exec`s a copy of the shell
N.B. only the child process calls this function

```
int exec_a_shell(int fd)  /* doesn't return */
{
    int tty_fd;;
```

- shell will expect I/O from standard file descriptors
  use '`dup2`' system call to link them to `fd`

```
    dup2(fd,0);     /*   standard input from fd        */
    dup2(fd,1);     /*   standard output to fd         */
    dup2(fd,2);     /*   standard error to fd              */
    close(fd);
    execv("/bin/sh",argv);
```

- exec only returns if it fails

- standard error has been closed
  so need to open `/dev/tty` explicitly

```
    tty_fd = open("/dev/tty",1);
    write(tty_fd,exec_fail_mess);
     _exit(1);
}
```

dup2 system call

```
int  res  =  dup2(old_fd, new_fd);
```

- makes `new_fd` point to same file / stream as `old_fd`
- `new_fd` is closed if already open
- most often used with standard I/O descriptors:

  ```
  dup2(fd,0);
  ```

  – standard input reads from `fd`
- can close the old descriptor
  ... but new descriptor still works

  ```
  dup2(fd,0);
  close(fd):
  n = read(0,buff,buff_len);
  ```
- negative return on failure

exec system call

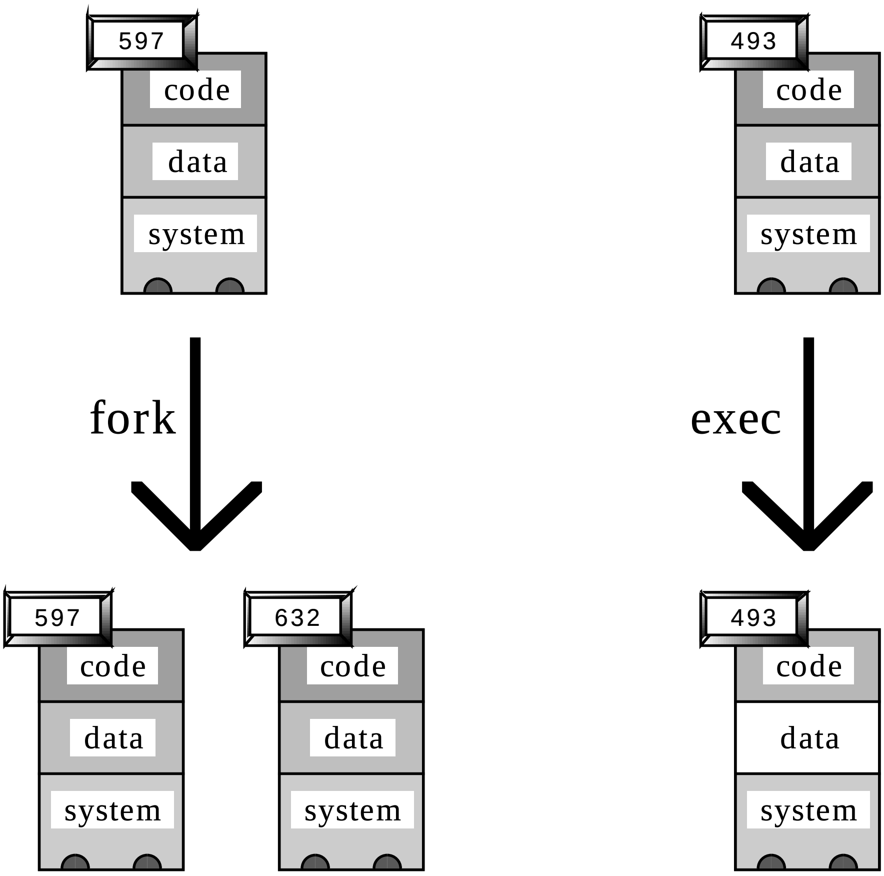
```
execv(char *prog, char **argv);
```

- replaces the current process with `prog`
- never returns except on failure
- `argv` is passed to the '`main`' of `prog`
  N.B. needs at least `argv[0]` set to program name
- new process:
  - ❍ code – replaced by `prog`
  - ❍ data – reinitialised
  - ❍ system data – partly retained

  ❋ file descriptors still open
- several variants (`execl`, `execvp`, . . .)
- often used after `fork` to spawn a fresh program

exec vs. fork

- fork duplicates process
- exec replaces process



- fork child shares open file descriptors
- exec-ed process retains open fds

death of a forked process

- when parent dies
  - ❍ children become orphans !
  - ❍ system init process 'adopts' them

- when child dies
  - ❍ parent (or init) informed by signal (`SIGCHLD`)
  - ❍ child process partly destroyed
  - ❍ rump retained until parent 'reaps'
    - – using `wait` or `wait3` system call
  - ❍ until then child is 'zombie'
    - – `ps` says `<exiting>` or `<defunct>`

N.B. zombie state necessary so parent can discover which child died

SIGCHLD & wait3

- if parent does not reap children
  . . . they stay zombies forever
  . . . system resources may run out

① first catch your signal

```
signal(my_reaper,SIGCHLD);
```

- function 'my_reaper' called when signal arrives

② then reap a child

```
int my_reaper()
{
   union wait status;
   while( wait3(&status,WNOHANG,NULL) >= 0 );
}
```

- use `WNOHANG` so that `wait3` doesn't block
- loop to reap multiple children

fork and I/O

low-level I/O

- open file descriptors shared so:
  - ❍ output is merged
  - ❍ input goes to first read
    - – accept similar
  - ❍ close down may be delayed until all processes close fd

    `close` all unwanted fds

    or use `ioctl` to set `close-on-exec`

high-level I/O

- C stdio is buffered:
  - ❍ duplicated at fork
  - ❍ may get flushed after fork
    duplicate writes
  - ✓ `stderr` OK – unbuffered

    careful with stdio

    use `stderr` or `setbuff(fd,NULL)`

Hands on

☞ copy the following from `tcp/session6`:

```
knife.c
make6
```

☞ compile `knife.c` :

```
make -f make6  knife
```

☞ launch the knife server: `knife.c` :

```
io 3%  knife -port 2345
```

☞ connect to it from a different machine or window

```
klah 7%  telnet io 2345
```

☞ do you get a shell prompt?

☞ try something simple like

```
echo hello
```

☞ then try `ps`

☞ what happens?

☞ try typing a # at the end of each line

```
echo hello#
ps #
```

☞ what is happening?

inet demon

- there are many Internet services:
  ftp, telnet, rlogin, echo, etc.
- a server for each is expensive
- `inetd` is a multi-service server
- it does a passive open on lots of ports:
  21 – ftp, 25 – SMTP, etc.
- when a client connects
  it forks the appropriate service
- remote logins somewhat complicated

remote login

First solution . . .

. . . simply fork a shell or getty

✘ no translation of codes

e.g. end of line sequence

✘ no terminal driver at server end

no tty control by application

e.g. editors need tty raw mode

Actual solution . . .

. . . intermediate process

- server-end process
  between client and shell / getty

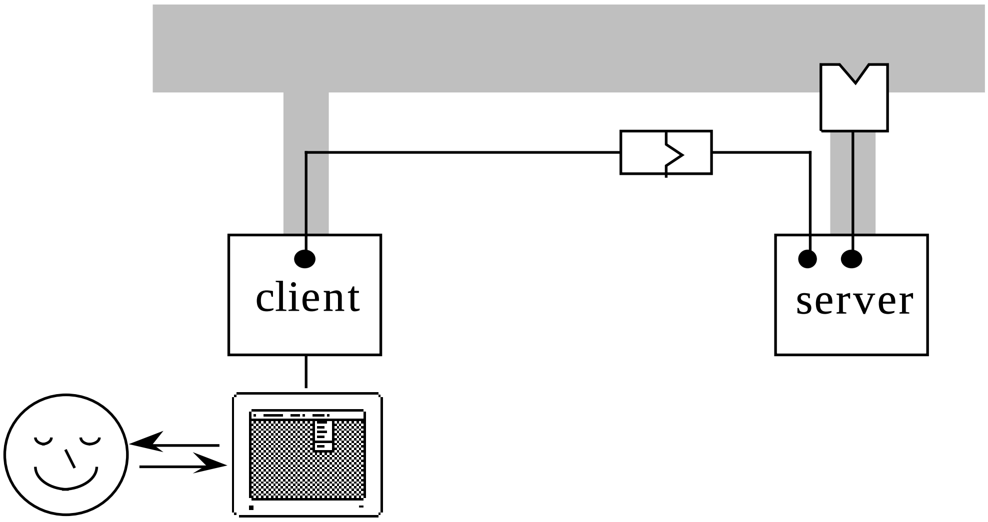
✓ can perform translation

✓ pseudo-tty between it and shell

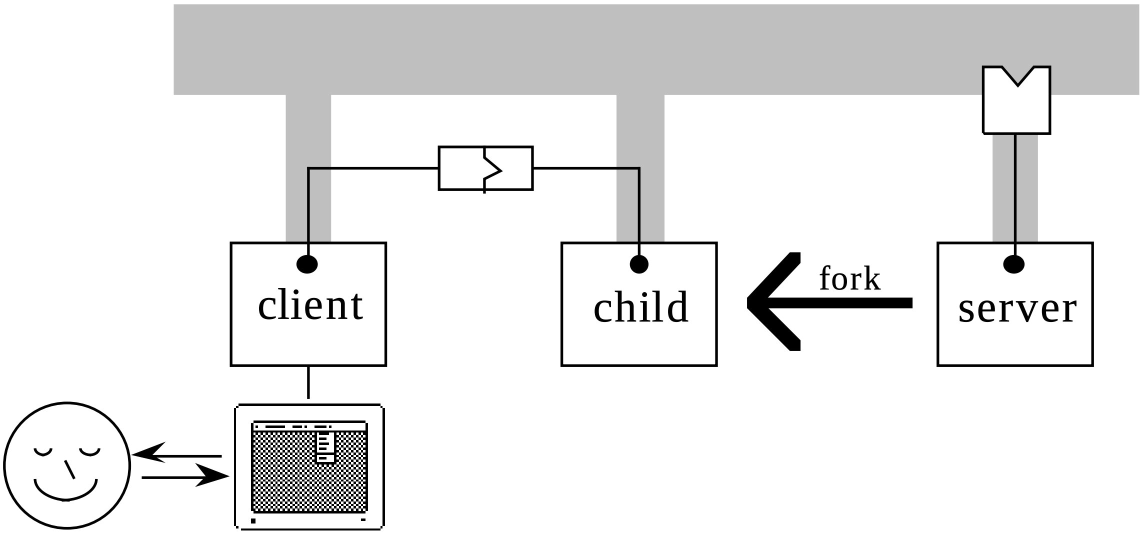
server-end tty control

remote login – 2

① remote login client connects to server

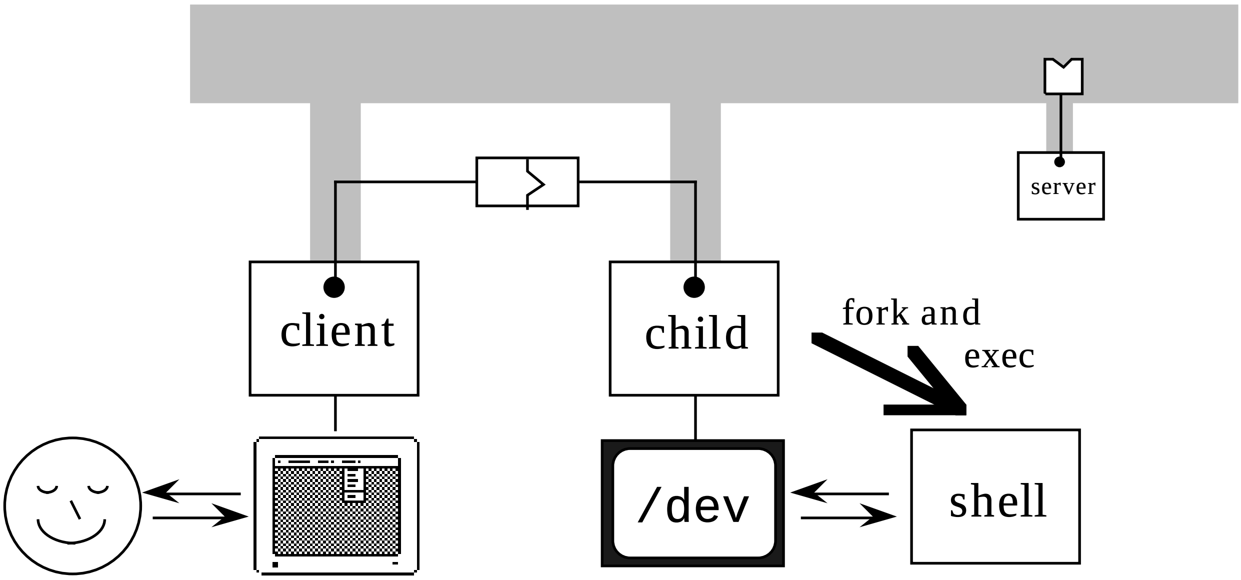


② server forks child to handle login



remote login – 3

③ child then forks another process



④ the new process connects to the child using a pseudo-terminal

⑤ and finally execs a shell (or getty etc.)

- ❋ user is now connected to shell

remote login – 4

- client and server-side child similar
  - ❍ both connected to network
  - ❍ both connected to (pseudo)terminal

- general algorithm:
  - ❍ echo terminal input to network
  - ❍ echo network input to terminal

  N.B. both concurrent

- difference in use of terminal:
  - ❍ where

    client – application end of tty
    child – 'user' end of pseudo-tty
  - ❍ how

    client – tty always in raw mode
    child – pseudo-tty mode set by shell

    only one layer of tty processing

☞ ☞ ☞ 🍴 Hands on 🍴 ✍ ✍ ✍

echo server

☞ modify `knife.c` to make a forking echo server
your previous echo server (session 2) only dealt with one client – this one will deal with any number

☞ copy `knife.c` into `echo-all`

☞ locate the sub-routine where the shell is exec-ed

☞ replace the code duplicating file descriptors and exec-ing the shell – simply have a loop which reads from the socket and writes back to it

☞ compile and run `echo-all`

```
io 15%  make  -f make6  echo-all
io 16%  echo-all  -port 2345
```

☞ an connect to it:

```
klah 23%  telnet io 2345
```

☞ there is an alternative solution which only involves replacing 2 characters of `knife.c`

☞ hint: the answer doesn't involve any dogs

MTUs

- the Internet is heterogeneous
  - ❍ heterogeneous transport layers
    different packet sizes
  - ❍ dynamic routing
    hops on different layers
    unpredictable packet size
- transport layer limit called MTU:
  – maximum transmission unit

transport layer	**MTU in bytes**
Hyperchannel	65535
16Mbps IBM token ring	17914
4Mbps IEEE 802.5 token ring	4464
FDDI	4352
Ethernet	1500
IEEE 802.3/802.2	1492
X.25	576
PPP (performance limit)	296

(from RFC 1191)

IP fragmentation

- what happens when size is too small?
- fragmentation
  - ❍ any intermediate router detects problem
  - ❍ IP datagram broken into pieces
  - ❍ each sent separately (possibly different routes)
  - ❍ reconstructed at further router or destination
- real limit is recipient's buffer size
  - ❍ 576 bytes IP datagram guaranteed
    ... but this includes headers
  - ❍ UDP limit = 512 bytes user data
  - ❍ TCP divides data up for you
    limit is UNIX read / write buffers
- only end points matter

  in a controlled environment . . .
  . . . larger datagrams possible

  e.g. NFS = 8192 bytes

fragmentation considered harmful

- fragmentation

  IP transparent to underlying link layer MTU
  . . . well almost . . .

- IP is not reliable

  some packets (fragments) may be lost

- no re-transmission
  - ❍ IP handles reconstruction . . .
    . . . but not fragment retransmission
  - ❍ fragment lost
    whole IP datagram lost
  - ❍ probability one fragment lost = p
    n fragments
    probability IP datagram lost ≈ n p

- avoiding fragmentation
  - ❍ UDP – most protocols ≤ 512 bytes
  - ❍ TCP – uses local (end-point) MTU
    \+ path MTU discovery algorithm

TCP reliability

- underlying IP unreliable

  TCP must handshake

- stream protocol
  - ❍ sender: this is bytes n–m of the data
  - ❍ recipient: ack m – last byte received

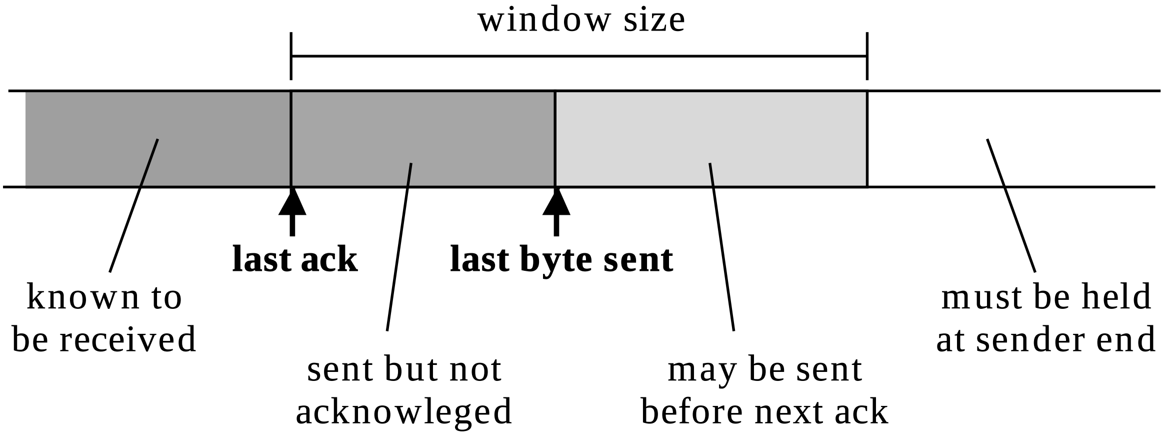
- retransmission
  - ❍ recipient: out of order receipt → repeat ack
  - ❍ timeout or several repeat acks → retransmit

- too many acks
  - ❍ avoid lots of little acknowledgement packets
  - ❍ ack of last packet previous packets arrived
  - ❍ piggyback A→B ack on B→A message
  - ❍ delay acks to allow piggyback
  - ❍ turn off delay for some protocols (e.g. X)

TCP flow control

Cannot send without limits:

- network capacity → packet loss
  - exponential backoff
    rapid resend → nightmare scenario
    long delay before failure (2-9 mins)
  - slow-start algorithm
- link-layer buffer
  - MSS announcement
- TCP buffer
  - window size announcement
    only send to last ack + window size



UNIX
**Network Programming
with TCP/IP**

UNIX
**Network Programming
with TCP/IP**

UNIX
**Network Programming
with TCP/IP**

UNIX
**Network Programming
with TCP/IP**

UNIX
**Network Programming
with TCP/IP**

Session 7

Select and Security

Session 7

Alan Dix
http://www.hcibook.com/alan

Select and Security

- UNIX events
- select system call
- proxy server
- ☞ raw client
- security, secrecy and privacy
- under attack: viruses & worm
- the Internet worm
- levels of security
- encryption and authentication

UNIX Events

Computational programs:

- busy most of the time
- read / write when <u>they</u> are ready

Interactive programs:

- servers & clients
- idle most of the time
- respond to events

UNIX processes – 4 types of event

① signal (interrupt)

② time (alarm)

③ input ready
  read will not block

④ output can accept (more) data
  write will not block

Responding to events

Events:

① signal (interrupt)
② time (alarm)
③ input (read) ready
④ output (write) ready

Responding

- interrupt handler – ①&②
  - use `signal` system call
  - use `setitimer` to send `SIGALRM`
- turntaking – ②,③&④
  - call `read`/`write` when ready
  - use `sleep` for delays
- polling – ②,③&④
  - use non-blocking `read`/`write`
  - use `time` to do things at specific times
- wait for several events
  - use `select` system call
  - timeout or `SIGALRM`

polling in UNIX

```
#include <sys/filio.h>
            ioctl(fd,FIONBIO,1);
```

- call to `ioctl` tells system:
  don’t block on `read`/`write`

- polling therefore possible

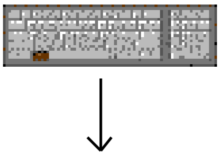
- structure of polling telnet-like client:

```
ioctl(tty_fd,FNBIO,1);
ioctl(net_fd,FNBIO,1);

for(;;) {
        /*  any terminal input?  */
    n = read(tty_fd,buff,buff_len);
    if ( n > 0 ) {  /* yes! do something */  }
        /*  any network input?   */
    n = read(net_fd,buff,buff_len);
    if ( n > 0 ) {  /* yes! do something */  }
}
```

read & write

`read`:

- waits on one file descriptor
- returns when input data is ready
- and reads the data into a buffer

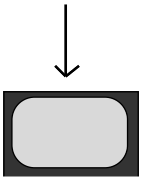


`read(0,buff,len)`

`write`:

- waits on one file descriptor
- returns when output is possible
- and writes the data from the buffer

`write(1,buff,len)`

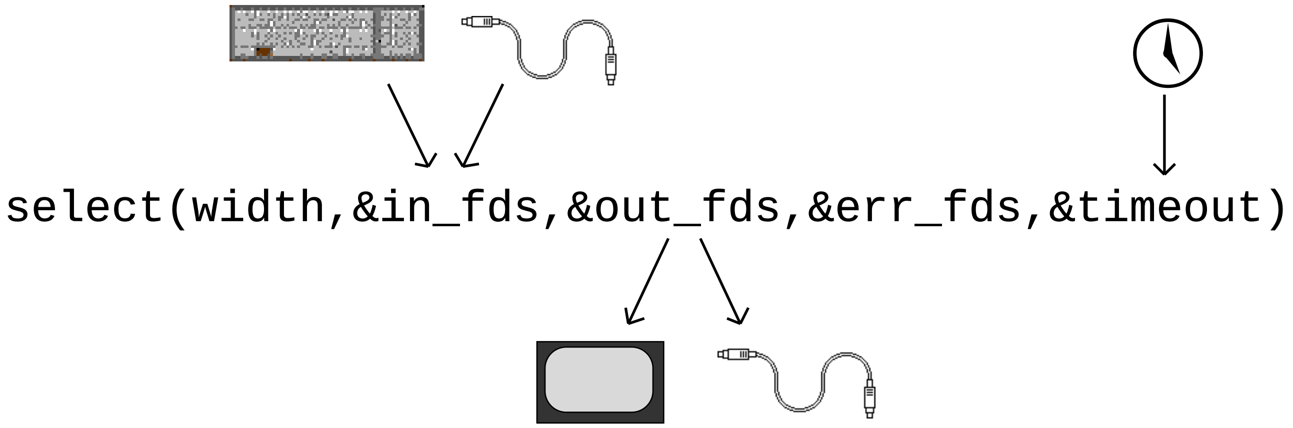


select

`select`:

- waits on <u>many</u> file descriptor
- returns when <u>input</u> or <u>output</u> ready
- but does <u>no</u> actual I/O

\+ also allows timeout



```
select(width,&in_fds,&out_fds,&err_fds,&timeout)
```

select system call – 2

```
int ret =
      select(size,&in_fds,&out_fds,&err_fds,&timeout);
```

- `in_fds, out_fds`: – bitmaps of file descriptors
  - ❍ `in_fds` – wait for input
i.e. read will not block
  - ❍ `out_fds` – wait for output
i.e. write will not block
- `size`: – size of `in_fds, out_fds, err_fds`
- `timeout`: – when to timeout
in seconds and milliseconds

Returns when:

• input ready on one of `in_fds`	(`ret > 0`)
• output ready on one of `out_fds`	(`ret > 0`)
• error occurs on one of `err_fds`	(`ret > 0`)
• timeout expires	(`ret == 0`)
• signal has been caught	(`ret < 0`)
• some other error occurs	(`ret < 0`)

select and I/O

```
#include <sys/types.h>
      fd_set  in_fds, out_fds, err_fds
```

- modified by call:
  call – bit set = wait for file desc
  return – bit set = file desc ready
  return value from select
  = number ready

- long integer in early UNIX systems

```
in_fds = in_fds || ( 1<<fd );
```

limit of 32 file descriptors
. . . but some systems allow more

- now a special `fd_set` structure
  actually an array of integers!
  - ❍ setting:

```
FD_ZERO( &in_fds );
FD_SET( fd, &in_fds );
FD_CLR( fd, &in_fds );
```

  - ❍ testing:

```
if ( FD_ISSET(fd,&in_fds) )  ...
```

select and I/O – 2

- input
  - ❍ terminal / socket
    - – `read` will not block
  - ❍ passive socket
    - – `accept` will not block

- output
  - ❍ terminal / socket
    - – `write` ‘ready’
  - ❍ `write` relies on system resources
  - ❍ change between `select` and `write`?
    - `write` may block
  - ❈ use non-blocking `write`

- can ‘get away’ without `select` on `write`
  . . . but dangerous!

select and timeouts

`#include <sys/time.h>`

```
struct timeval timeout;
```

- `timeout.tv_secs`
  `timeout.tv_ms`
  – maximum time to wait in seconds and ms

- if no I/O ready and no signals in time limit
  then select returns with zero result
  N.B. `in_fds`, `out_fds`, `err_fds` all zero also

- modified by call?
  - ❍ ideally should return time remaining
  - ❍ doesn't now . . .
    . . . but may do one day

  don't rely on `timeout` not being changed
  reset for each call to `select`

select and signals

- signal occurs during system call:

  `read`, `write`, or `select`

- signal not caught . . .

  . . . process aborts!

- signal caught . . .

  ① relevant handler called

  ② systems call returns with ‘error’

- how do you know?
  - ❍ negative return value
  - ❍ `errno` set to `EINTR`

- negative return & `errno` ≠ `EINTR`

  really an error!

care with signals

- signal handlers can run at any time

```
int i = 0

int my_handler()
{
    i = i + 1
}

main()
{
    signal(my_handler,SIGINTR);
    for(;;)
        if ( i > 0 ) {
               do_something();
               i = i - 1;
               }
}
```

- intention:

  execute `do_something` once per interrupt

- what actually happens:

  ① interupt processed (`i=1`)
  ② `do_something` executes
  ③ main calculates `i-1` gets result `0`
  ④ before it stores the result . . .
  . . . another interupt (`i=2`)
  ⑤ `main` stores result (`i=0`)

when to use select

- servers:
  - ❍ where concurrency essential
  - ❍ possibly ftp server
    - – listen to control & data
  - ❍ telnet server
    - – listen to user over network
    - \+ listen to shell / application

- clients
  - ❍ not with most window managers
    - – instead use callback
  - ❍ some event stream WMs
    - – single fd for WM events
    - – listen to WM and network
  - ❍ terminal based clients
    - – not needed for turn-taking
    - – e.g. telnet / rlogin clients

proxy server

- proxy server used in session 3
- structure of code
  ① passive open on own port
  ② wait for client connection
  ③ active open on remote server
  ④ loop forever
  waiting for client or server input:
    ❍ when client data ready
      read it
      send to server
      echo it to terminal
    ❍ when server data ready
      read it
      send to client
      echo it to terminal

proxy code – 1

① Main loop

```
main(...) {
      /*  establish port                            */
   port_sk = tcp_passive_open(port);
      /*  wait for client to connect                */
   client_sk = tcp_accept(port_sk);

      /*  only want one client,                     */
      /*  so close port_sk                          */
    close(port_sk);

      /*  now connect to remote server              */
   serv_sk = tcp_active_open(rem_host,rem_port);

   ret = do_proxy( client_sk, serv_sk );

   exit(0);
}
```

- when `do_proxy` is called both network sockets open

proxy code – 2

② perform proxy loop

```
int do_proxy( int client_sk, int serv_sk )
{
```

- first declare and initialise fd bitmaps

```
   fd_set  read_fds, write_fds, ex_fds;
   FD_ZERO(&read_fds);  FD_ZERO(&write_fds);
   FD_ZERO(&ex_fds);
   FD_SET(client_sk,&read_fds);
   FD_SET(serv_sk ,&read_fds);
```

- then loop forever

```
   for(;;) {
      int      num, len;
```

- copy bitmaps because `select` modifies them

```
      fd_set   read_copy  =  read_fds;
      fd_set   write_copy =  write_fds;
      fd_set   ex_copy    =  ex_fds;
      static struct timeval timeout = {0,0};
```

- then call `select`

```
      num = select(MAX_FD, &read_copy, &write_copy,
                      &ex_copy, &timeout);
```

➥ check return – ③, ④ & ⑤ at this point

```
      }
   return 0;
}
```

proxy code – 3

③ check for signals, errors and timeout

- first check for signals:

  in this case, we are not expecting any so return
  in general, we may need to do some processing following the interrupt
  it is usually better for the interrupt to set some flag and let the main loop do most of the work
  this reduces the risk of stacked interrupts and mistakes in concurrent access to data structures

```
    if (num < 0 && errno == EINTR ) {
                                /* stopped by signal */
        perror("EINTR");  return 1;
        }
```

- if there has been no signal `num < 0` is an error

```
    if (num < 0  ) {            /* not stopped by signal */
        perror("select");  return 1;
        }
```

- if num is zero then a timeout has occurred

  again, in this case no processing
  but in general this is the opportunity for animation or other periodic activity

```
    if ( num == 0 ) continue;   /*  timeout  */
```

proxy code – 4

④ check for client input
client ready if bit is set in `read_copy`

```
if ( FD_ISSET(client_sk,&read_copy) ) {
   int len = read( client_sk, buff, buf_len );
```

- on end of file or error exit the loop

```
   if ( len <= 0 ) {  /*  error or close  */
      close(serv_sk);  return len;
      }
```

- if there is some input data, write it to the server and log it

```
   else {
      write(serv_sk,buff,len);
      log_from_client( buff, len );
      }
   }
```

⑤ server input similar

```
if ( FD_ISSET(serv_sk ,&read_copy) ) {
   int len = read( serv_sk , buff, buf_len );
   if ( len <= 0 ) {  /*  error or close  */
      close(client_sk);
      return len;
      }
   else {
      write(client_sk,buff,len);
      log_from_server( buff, len );
      }
   }
```

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

❇ the proxy server is a bit similar to a telnet client
both open a connection to a remote server
both echo from the user to the server . . .
. . . and from the server to the user
the major difference is that the proxy server
operates on the ‘other end’ of a network connection

☞ you are going make a simple telnet-like client

☞ copy `proxy.c` and `make7` from `tcp/session7`
copy `proxy.c` and call it `raw-client.c`

❇ `proxy.c` reads and writes the client socket
you want to read from standard input (`0`)
and write to standard output (`1`)

☞ proceed as follows:

① remove the code to open the client connection
(passive open and accept)

② remove the parameter to `do_proxy` which corresponds to the client socket

③ modify the `FD_SET` calls so that `select` waits for standard input (`0`) rather than the client

④ change all `read` calls from the client so that they read from standard input (`0`)

⑤ change all `write` calls to the client so that they write to standard output (`1`)

☞ now compile and run your raw client, e.g.:

```
raw-client  hades  25
```

(send mail as in session 3 page 3/17)

Security

- types of security:
  - ❍ information:
    - – secrecy
    - – privacy
  - ❍ resources:
    - – destructive access
    - – virus infection

- linked
  - ❍ information → resources

    e.g. password → login
  - ❍ resources → information

    e.g. modify `/etc/passwd`

- chain reaction
  - ❍ small breach → complete loss

    e.g. root password!
  - ❍ N.B. special problem for computers

who are you afraid of?

- internal
  - selling your secrets
  - personal data
    - payroll, debtor files etc
  - using resources
    - surfing, doom!
  - downloading material
    - indecent, possibly illegal
  - backdoors

```
client_sk = tcp_accept(port_sk);
n= read(client_sk,buff,buff_len);
buff(len) = '\0';
if ( strcmp(buff,"Alan's secret way in") == 0 ) {
   /*  connect client_sk to a root shell  */
   }
/*  normal operation * /
```

- external
  - hackers
  - accidental release
    - e.g. forgotten portable on the train
  - industrial espionage
  - viruses

under attack

- viruses a real risk?
  - ✔ heterogeneous

    cross-infection more difficult
  - ✘ lots of machines just like yours
  - ? interpreted languages?
    - – can be made secure (e.g. JAVA)

- types of attack
  - ❍ virus
    - – embeds itself in another program
  - ❍ Trojan horse
    - – masquerades as another program
  - ❍ worm
    - – independent self-replicating program

N.B. names and definitions differ

viruses on the web?

- explicit download of code
  - ❍ helpers – machine specific code
  - ❍ general software
  - ✘ both risk infection

- implicit download
  - ❍ semi-compiled – JAVA
  - ❍ interpreted – JAVA script
  - ❍ embedded in HTML
  - ✘ you may never know!

✔ the good news
  - ❍ JAVA & JAVA script ‘safe’
  - ❍ cannot read or write to local disk

✘ the bad news
  - ❍ JAVA script can connect remotely
  - ❍ send details of browsing patterns
  - ❍ minor breach of privacy
  - ? the only breach possible?

The Internet Worm

- for 2 days in 1988, the Internet was under siege

November 2nd, 1988

17:00	worm launched from Cornell University
21:00	worm detected at Stanford
22:04	worm detected at Berkeley
23:40	Berkeley discover one means of attack (sendmail)
23:45	infects Dartmouth and Army Ballistics Res. Lab.

November 3rd, 1988

00:21	Princeton University main machine crashes due to load
02:38	email from Berkeley: “We are under attack”
03:15	anonymous warning from foo@bar.arpa
05:54	patches to sendmail distributed
06:45	National Computer Security Centre (NCSC) informed
11:30	Milnet severs itself from Arpanet to prevent infection
16:00	inoculation method found (directory `sh` in `/usr/tmp`)
21:30	Berkeley start to decompile ‘captured’ worm

November 4th, 1988

05:00	MIT finish decompiling worm
11:00	Milnet rejoins Arpanet
17:20	final set of preventative patches mailed
21:30	worm’s author identified – named in the next day’s newspaper as Robert T. Morris son of the NCSC’s chief scientist Robert Morris!

- infections still noted as late as December 1988

What went wrong?

- several means of attack

- between machines:
  - debug mode in `sendmail`
  - buffer overflow in `fingerd`
  - once broken into a user on a machine
    - `rlogin` / `rsh` to other hosts

- within a machine:
  - simple password attacks
    - permutations of user’s own name
    - internal list of 432 common passwords
    - system dictionary

- attempted to prevent repeat infection
  - didn’t always work
  - main damage was excessive load due to repeat infections (often 100s)
  - also how it was detected

sendmail attack

- `sendmail` had a debug mode
  - worm connects to `sendmail`
  - worm sends ‘`debug`’ command
  - `sendmail` will then execute any command!
  - should have been disabled but `sendmail` is complex!

- similar attacks still possible
  - system engineer accounts
  - remote vendor maintenance

- any debug modes on your system?

fingerd attack

- `fingerd` uses `gets` – buffer overflow
  - ❍ worm connects to `fingerd`
  - ❍ worm sends 536 byte line
  - ❍ overflows `fingerd`'s buffer (512 bytes)
    . . . and corrupts stack
  - ❍ extra 24 bytes executed as code!

- lessons:
  - ❍ never use `gets`!
    - – at best may crash
    - - at worst is a loophole
  - ❍ always be careful of buffer lengths

- never again?
  - ❍ a popular www browser . . .
    - – corrected in later versions

physical security

- physical security:
  - ❍ are the machines secure

    can someone reboot, substitute disks etc.?
  - ❍ is the network secure

    can someone link-in their own computer?

- local or global?
  - ① local network and machines
  - ② backbone and routers
  - ③ remote network and machines

- secure?
  - ① possible
  - ② reasonable for non-critical data
  - ③ no way!

N.B. ‘listening in’ easy on many networks

e.g. ethernet

- <u>never</u> trust transport layer

logical security

- secrecy:
  - ❍ TCP/IP packets not secure
    e.g. credit card by email
  - ❍ use encryption
    e.g. Netscape secure sockets layer for WWW

- authentication:
  - ❍ who am I talking to?
  - ❍ is it the real server?
    - ✔ rely on correct routing and protected ports
    - ✘ impostor machine, non-UNIX server host
  - ❍ is it an acceptable client?
    - ✔ user passwords
    - ✘ often sent as plain text! – e.g. telnet

- audit:
  - ❍ risk of detection deters
  - ❍ keeping logs
  - ❍ relies on authentication
    - ✔ SMTP reverse name lookup
    - ✘ can't check `FROM` field – e.g. worm warning

low-level protection – firewalls

- simple measures
  - ❍ isolation
    - – don't connect to the global Internet

      … but lose the benefits too
  - ❍ anonymity
    - – don't publish domain machine names

      … but IP addresses still valid

- firewalls
  - ❍ application independent
  - ❍ act at router / gateway
  - ❍ can only look at IP or TCP headers

- what is possible
  - ❍ only allow friendly IP addresses
    - – N.B. impostors
  - ❍ limited internal routing
    - – protect sensitive machines / data
  - ❍ restrict incoming TCP packets
    - – only allow connection to protected ports

      … but difficult for ftp

high-level protection – ring fences

- rlogin
  - ❍ beware external root logins!
  - ❍ passwords:
    - – if reasonable no ‘equiv’ hosts
    - – certainly no root ‘equiv’ hosts
    - ? means lots of duplicate password files?

- servers
  - ❍ never run as root?
    - – impossible! e.g. inetd, rshd
  - ❍ never unnecessarily run as root?
    - – special login e.g. user ‘`ftp`’
    - – run as user ‘`nobody`’

- the rest of the system – normal measures
  - ❍ backups – damage limitation
  - ❍ permissions – restrict ‘other’ access
  - ❍ setuid – dangerous, no write perm!
  - ❍ `/etc/passwd` – encrypt or restricted read
    – may cause problems

encryption

- one way function:

  `cypher = f(input)` – easy
  `input  = ?(cypher)` – hard

  - used in /etc/passwd
  - brute force attack:
    for each possible input `inp`
    if `f(inp)` is `cypher` – got it!

- single key

  `cypher = code(key,input)`
  `input  = decode(key,cypher)`

  - in DES – `code` = `decode`

- public key encryption

  `cypher = code(key1,input)`
  `input  = decode(key2,cypher)`

  - `key1` – given to everyone – public
    `key2` – kept by you – private
  - anyone can send a message
    only <u>you</u> can decrypt it

session keys and authentication

- public keys good, but:
  - ❍ expensive
  - ❍ the more you use a key
    the easier it is to break
- use public keys to exchange single key
  - ① machine A generates session key K_S
  - ② A encrypts it using B’s public key
    $K_{SB} = \text{code}(K_{B1}, K_S)$
  - ③ A sends K_{SB} to B
  - ④ B decrypts K_{SB} to obtain K_S
    $K_S = \text{decode}(K_{B2}, K_{SB})$
  - ⑤ B generates value X
  - ⑤ B encrypts X and Ks using A’s public key
    $KX_A = \text{code}(K_{A1}, X.K_S)$
  - ⑥ B sends KX_A to A
  - ⑦ A decrypts KX_A
    $X.K_S = \text{decode}(K_{A2}, K_{SA})$
  - ⑧ A encrypts X using B’s public key
    $X_B = \text{code}(K_{B1}, X)$
  - ⑨ and sends it to B
- result:
  - ❍ A and B share a secret key
  - ❍ A and B sure of each other’s identity
- discard key after session or fixed time

authentication servers

- how do you find out B’s public key?
- answers:
  - ① B tells you
  - ② someone else, C, tells you
  - ③ use physical means (post, hand)
- if ① or ②: how do you know it is B / C?
- if ②: why should you believe C?

  ③ ?

✘ no good for broad distribution

✔ use an authentication server

- ❍ trusted machine
- ❍ everyone tells it their public key (using its public key or physical)
- ❍ ask it for other’s public keys
- ❍ or ask it for session keys

don’t panic!

- how secure is a fax?
- credit card number by phone
- hacker ≈ burglar
  - if they want in, you won’t stop them
- main differences
  - rate of loss (Mbytes / sec)
  - hidden loss (electronic copies)
  - automatic attack
- ease of use ≈ ease of access
  - where do <u>you</u> draw the line