TCP and Connections

TCP provides reliability by tracking and retrying lost packets

⇒ needs an explicit \textit{connection} between processes

⇒ needs explicit notions of client and server

Server side has two sockets:

A \textit{listener} to receive clients

A per-client socket to communicate with the client

Each client has a single socket
Listening for Connections

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

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

In the background, allow TCP connections via `socket`

`socket` must be bound to an address already

Clients connect to that address

`backlog` indicates how many not-yet-accepted connections to allow in waiting
Accepting Connections

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

int accept(int socket,
            struct sockaddr *addr, socklen_t *addr_len);
```

Accepts a connection from `socket` and returns it as a new socket

`socket` must be previously passed to `listen`

`addr` and `addr_len` are filled with the client’s address

... but a server doesn’t usually care
Making Connections

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

int connect(int socket,
            struct sockaddr *addr, socklen_t addr_len);
```

Binds `socket` to a TCP connection as a client

`addr` and `addr_len` are the server’s address

The server sees the connection when it calls `accept`
Using TCP
Using TCP

**Client**
- `getaddrinfo`
- `socket`
- `connect`
- `read`
- `write`
- `close`

**Server**
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `read`
- `write`
- `close`

Use `shutdown` to leave `read` half open.

Using TCP

client

getaddrinfo

socket

connect

read

write

close

EOF

server

getaddrinfo

socket

bind

listen

accept

read

write

close

EOF

open_clientfd

open_listenfd

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Using TCP
memset(&hints, 0, sizeof(struct addrinfo));

hints.ai_family = AF_INET;        /* Request IPv4 */
hints.ai_socktype = SOCK_STREAM;  /* Accept TCP connections */
hints.ai_flags = AI_PASSIVE;      /* ... on any IP address */
Getaddrinfo(NULL, portno, &hints, &addrs);

ls = Socket(addrs->ai_family, addrs->ai_socktype, addrs->ai_protocol);
Bind(ls, addrs->ai_addr, addrs->ai_addrlen);
Freeaddrinfo(addrs);
Listen(ls, 5);

while (1) {
    ....
s = Accept(ls, (struct sockaddr *)&addr, &len);
amt = Read(s, buffer, MAXBUF);
write(1, buffer, amt);
write(1, "\n", 1);
write(s, &amt, sizeof(amt));
Close(s);
}
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET; /* Request IPv4 */
hints.ai_socktype = SOCK_STREAM; /* TCP connection */
Getaddrinfo(hostname, portno, &hints, &addrs);

s = Socket(addrs->ai_family, addrs->ai_socktype, addrs->ai_protocol);
Connect(s, addrs->ai_addr, addrs->ai_addrlen);
Freeaddrinfo(addrs);

copies = ((argc == 5) ? atoi(argv[4]) : 1);
len = strlen(argv[3]);
while (copies--) {
    amt = Write(s, argv[3], len);
    if (amt != len) app_error("incomplete write");
}

got = 0;
amt = Read(s, &got, sizeof(got));
printf("server got %ld\n", got);
Close(s);
....
Robust I/O

Provided by `csapp.c`:

```c
ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Like `read` and `write`, but loops as needed for short counts and signal interruptions
Revised TCP Server

Robust I/O, wait for an EOF from clients:

```c
s = Accept(ls, (struct sockaddr *)&addr, &len);

while (1) {
    char buffer[MAXBUF];
    size_t amt = Rio_readn(s, buffer, MAXBUF);
    if (amt == 0) {
        printf("client is done\n");
        Rio_writen(s, &total_amt, sizeof(total_amt));
        break;
    } else {
        Rio_writen(1, buffer, amt);
        Rio_writen(1, "\n", 1);
        total_amt += amt;
    }
}

Close(s);
....
```
while (copies--)
    
    Rio_writen(s, argv[3], len);

Shutdown(s, SHUT_WR);

got = 0;
amt = Rio_readn(s, &got, sizeof(got));
if (amt != sizeof(got))
    app_error("response truncated");
printf("server got %ld\n", got);

Close(s);

....
Re-Revised TCP Server

Client can declare amount it will send

```c
....
s = Accept(ls, (struct sockaddr *)&addr, &len);

amt = Rio_readn(s, &total_amt, sizeof(total_amt));
if (amt != sizeof(total_amt))
    app_error("amount truncated");

buffer = malloc(total_amt);
amt = Rio_readn(s, buffer, total_amt);

Rio_writen(1, buffer, amt);
Rio_writen(1, "\n", 1);
free(buffer);

write(s, &amt, sizeof(amt));
Close(s);
....
```
Re-Revised TCP Client

ten_client3.c

....

len = strlen(argv[3]);

amt = copies * len;
Rio_writen(s, &amt, sizeof(amt));

while (copies--)
    Rio_writen(s, argv[3], len);

got = 0;
amt = Rio_readn(s, &got, sizeof(got));
if (amt != sizeof(got))
    app_error("response truncated");

printf("server got %ld\n", got);

Close(s);

....
Echo Server

Another example: a server that echoes back every line

server

\[\text{lnr} = \text{Open	extunderscore listenfd}(....);\]
\[\text{fd} = \text{Accept(lnr, ....);}\]
\[\text{fd} = \text{Open	extunderscore clientfd}(....);\]
Echo Server

Another example: a server that echoes back every line

server

"Hi\n"

client

\[\text{lncr} = \text{Open\_listenfd}(\ldots)\];
\[\text{fd} = \text{Accept}(\text{lncr}, \ldots)\];
\[\text{fd} = \text{Open\_clientfd}(\ldots)\];
\[\text{Rio\_writen} (\text{fd}, \text{"Hi\n"}, 3)\];
Echo Server

Another example: a server that echoes back every line

```
lnr = Open_listenfd(...);
fd = Accept(lnr, ....);

fd = Open_clientfd(...);
Rio_writen(fd, "Hi\n", 3);
Rio_writen(fd, ....);
```
Echo Server

Another example: a server that echoes back every line

```
lnr = Open_listenfd(...);
fd = Accept(lnr, ...);
while (readline(fd, ...))
    Rio_writen(fd, "Hi
", 3);
    Rio_writen(fd, ....);
...
```
Echo Server

Another example: a server that echoes back every line

server

"Hi\n"
"Hi\n"

client

Reading one byte at a time would be slow

lnr = Open_listenfd(....);
fd = Accept(lnr, ....);

while (readline(fd, ....))
    Rio_writen(fd, "Hi\n", 3);
    Rio_writen(fd, ....);
    ...

fd = Open_clientfd(....);
Robust Buffered Reading

```c
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);
ssize_t rio_readlineb(rio_t *rp, void *buf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *buf, size_t n);
```

**rio_initb** initializes a buffer to hold bytes that are read but not yet consumed

**rio_readlineb** fills the buffer as needed and consumes bytes that make a line

**rio_readnb** is like **rio_readn**, but keeps using the buffer
```c
int main(int argc, char **argv) {
    int listenfd, connfd;
    char client_hostname[MAXLINE], client_port[MAXLINE];
    struct sockaddr_storage clientaddr; /* Enough room for any addr */

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        socklen_t clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);

        Getnameinfo((SA *) &clientaddr, clientlen,
                    client_hostname, MAXLINE, client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);

        echo(connfd);

        Close(connfd);
    }
}
.....
```
void echo(int connfd) {
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);

    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %ld bytes\n", n);
        Rio_writen(connfd, buf, n);
    }
}
Telnet

Instead of implementing an echo client, we can just use `telnet`

If the echo server is at `localhost` on `4567`:

```
$ telnet localhost 4567
```

Use `Ctrl-]` and then `quit` to stop
HTTP

http://www.eng.utah.edu/~cs4400/

browser client
HTTP

www.eng.utah.edu:80

web server

browser client

http://www.eng.utah.edu/~cs4400/
HTTP

www.eng.utah.edu:80

web server

browser client

http://www.eng.utah.edu/~cs4400/
HTTP

http://www.eng.utah.edu/~cs4400/
HTTP

GET /~cs4400/ HTTP/1.0
<html>...</html>

http://www.eng.utah.edu/~cs4400/
URLs

scheme : //host : port/path?query#fragment

scheme — protocol to use

http ⇒ HTTP
  TCP
  IP

https ⇒ HTTP
  SSL
  TCP
  IP

ftp ⇒ FTP
  TCP
  IP

Assume http for the rest
URLs

scheme: //host: port/path?query#fragment

Each color is optional

scheme — protocol to use

host — the server’s host

www.eng.utah.edu

google.com

127.0.0.1
URLs

```
scheme: //host:port/path?query#fragment
```

each color is optional

- **scheme** — protocol to use
- **host** — the server’s host
- **port** — the server’s port

defaults to 80

**http://localhost/**  ⇒  port 80 on localhost

**http://localhost:8090/**  ⇒  port 8090 on localhost
URLs

scheme: //host: port/path?query#fragment

each color is optional

scheme — protocol to use
host — the server’s host
port — the server’s port
path — item to get from the server, defaults to empty

Meaning of path is up to the server:

• could be a file
• could be a request to compute
• could be a request to change
URLs

scheme: //host: port/path?query#fragment

- scheme — protocol to use
- host — the server’s host
- port — the server’s port
- path — item to get from the server, defaults to empty

http://www.eng.utah.edu/~cs4400/network.pdf
⇒ gets a file from the CADE filesystem

https://goo.gl/om5FJq
⇒ looks up redirection URL
URLs

scheme: //host: port/path?query#fragment

each color is optional

scheme — protocol to use

host — the server’s host

port — the server’s port

path — item to get from the server, defaults to empty

query — options related to path

Sequence of key=value separated by & or ;

http://www.youtube.com/watch?v=KFyuGneWhJ4

http://twitter.com/search?q=utah&src=typd

Meaning of query is also up to the server
URLs

scheme: //host:port/path?query#fragment

scheme — protocol to use
host — the server’s host
port — the server’s port
path — item to get from the server, defaults to empty
query — options related to path
fragment — more options related to path

Use of fragment is up to the client — not sent to the server

http://www.eng.utah.edu/~cs4400/#(part._staff)
Encoding

scheme: //host: port/path?query#fragment

What if a path includes ?
What if a query includes #?
What if a value in a query includes &?

Percent encoding:
• Replace a character with %XX
  ○ Each X is a hex digit
  ○ 0xXX is the character’s value
• Replace a space with +

Access lost+found with user as me&you:

http://server.com/lost%2bfound?user=me%26you
HTTP

http://host:port/path?query#fragment

Client connects to host: port with TCP and sends...

Still more choices!

• **GET** mode — the default

• **POST** mode — better for sending data

• ...
HTTP

http://host:port/path?query#fragment

Client connects to host:port with TCP and sends...

GET /path?query HTTP/1.0

field: value ← zero or more as header

CRLF

where CRLF is a two-byte sequence:

• carriage return (CR) character, which is ASCII 13
• a linefeed (LF) character, which is ASCII 10
HTTP

\texttt{http://host:port/path?query#fragment}

Client connects to \texttt{host:port} and sends...

\texttt{GET /path?query HTTP/1.0 CRLF}

\texttt{field: value CRLF} \leftarrow \text{zero or more as header}

\texttt{CRLF}

where \texttt{CRLF} is a two-byte sequence:

- carriage return (CR) character, which is ASCII 13
- a linefeed (LF) character, which is ASCII 10
HTTP

http://host:port/path?query#fragment

Client connects to host: port with TCP and sends...

```
GET /path?query HTTP/1.0 CRLF
field: value CRLF ← zero or more as header CRLF
```

where CRLF is a two-byte sequence:

- carriage return (CR) character, which is ASCII 13
- a linefeed (LF) character, which is ASCII 10
HTTP

http://host:port/path?query#fragment

Client connects to host: port with TCP and sends...

GET /path?query HTTP/1.0 CRLF
field: value CRLF ← zero or more as header CRLF

Adjust echo.c to print received lines and point a web browser at it
HTTP

http://host:port/path?query#fragment

Client connects to host: port with TCP and sends...

GET /path?query HTTP/1.0 CRLF
field: value CRLF ← zero or more as header CRLF

Server replies with

HTTP/1.0 status status-message CRLF
field: value CRLF ← zero or more as header CRLF
data
HTTP

http://host:port/path?query#fragment

Client connects to host: port with TCP and sends...

GET /path?query HTTP/1.0 CRLF

field: value CRLF ← zero or more as header CRLF

Server replies with

HTTP/1.0 status status-message CRLF

field: value CRLF ← zero or more as header CRLF

data
HTTP

http://host:port/path?query#fragment

Client connects to host: port with TCP and sends...

GET /path?query HTTP/1.0
field: value
HTTP/1.0 status status-message
field: value
200 OK
301 Moved permanently
400 Bad request
501 Not implemented
HTTP

```
http://host:port/path?query#fragment
```

Client connects to `host:port` with TCP and sends...

```
GET /path?query HTTP/1.0 CRLF
field: value CRLF ← zero or more as header CRLF
```

Server replies with

```
HTTP/1.0 status status-message CRLF
field: value CRLF ← zero or more as header CRLF
data
telnet to a web server
```
HTTP

http://host:port/path?query#fragment

Client connects to host: port with TCP and sends...

POST /path?query HTTP/1.0 CRLF
field: value CRLF ← zero or more as header CRLF
data

Server reply is the same as for GET
Common Header Fields

Content-Length — length of response data or POST data

Content-Type — type response data or POST data
  • text/html: HTML
  • text/html; charset=utf-8: HTML with UTF-8 content
  • application/x-www-form-urlencoded: POST data that uses the query format

Connection — close to get a single response

Field names are case-insensitive
TINY Web Server

The **TINY Web Server** is a useful web server in 250 lines of code

For each `/path?query` request:

If `/path` does not start `/cgi-bin`:
- Assume that `path` refers to a file
- Report `Content-Length` as file size
- Infer `Content-Type` from file extension
- Send file content as reply data

If `/path` starts `/cgi-bin`:
- Assume that `path` refers to an executable
- Set `QUERY_STRING` environment variable to `query`
- Run executable to generate result
TINY Web Server — Main

```c
int main(int argc, char **argv) {
    int listenfd, connfd;
    char hostname[MAXLINE], port[MAXLINE];
    struct sockaddr_storage clientaddr;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        socklen_t clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

        Getnameinfo((SA *)&clientaddr, clientlen, hostname, MAXLINE,
                    port, MAXLINE, 0);
        printf("Accepted connection from (%s, %s)\n", hostname, port);

        doit(connfd);

        Close(connfd);
    }
}
....
```
void doit(int fd) {
    int is_static;
    char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
    char filename[MAXLINE], cgiargs[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, fd);

    if (!Rio_readlineb(&rio, buf, MAXLINE))
        return;
    sscanf(buf, "%s %s %s", method, uri, version);

    read_requesthdrs(&rio);

    is_static = parse_uri(uri, filename, cgiargs);

    ....
}
TINY Web Server — Reading Headers

```c
void read_requesthdrs(rio_t *rp) {
   char buf[MAXLINE];

   Rio_readlineb(rp, buf, MAXLINE);
   while (strcmp(buf, "\r\n"))
      Rio_readlineb(rp, buf, MAXLINE);
}
```
int parse_uri(char *uri, char *filename, char *cgiargs) {
    ....
    if (!strstr(uri, "cgi-bin")) {
        ....
        if (uri[strlen(uri)-1] == '/')
            strcat(filename, "home.html");
        ....
        return 1; /* => static content */
    } else {
        ....
        return 0; /* => dynamic content */
    }
}
void doit(int fd) {
  struct stat sbuf;

  ....
  if (stat(filename, &sbuf) < 0) {
    clienterror(fd, filename, "404", "Not found",
                ....);
    return;
  }

  if (is_static) {
    ....
    serve_static(fd, filename, sbuf.st_size);
  } else {
    ....
    serve_dynamic(fd, filename, cgiargs);
  }
}
# Tiny Web Server — Serving Files

```c
void serve_static(int fd, char *filename, int filesize) {
    int srcfd;
    char *srcp, filetype[MAXLINE], buf[MAXBUF];

    get_filetype(filename, filetype);
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
    sprintf(buf, "%sConnection: close\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
    Rio_writen(fd, buf, strlen(buf));

    srcfd = Open(filename, O_RDONLY, 0);
    srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
    Close(srcfd);
    Rio_writen(fd, srcp, filesize);
    Munmap(srcp, filesize);
}
```
TINY Web Server — Inferring a File Type

```c
void get_filetype(char *filename, char *filetype) {
    if (strstr(filename, "html"))
        strcpy(filetype, "text/html");
    else if (strstr(filename, "gif"))
        strcpy(filetype, "image/gif");
    else if (strstr(filename, "png"))
        strcpy(filetype, "image/png");
    else if (strstr(filename, "jpg"))
        strcpy(filetype, "image/jpeg");
    else
        strcpy(filetype, "text/plain");
}
```
void serve_dynamic(int fd, char *filename, char *cgiargs) {
    char buf[MAXLINE], *emptylist[] = { NULL };

    /* Return first part of HTTP response */
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Server: Tiny Web Server\r\n");
    Rio_writen(fd, buf, strlen(buf));

    if (Fork() == 0) {
        setenv("QUERY_STRING", cgiargs, 1);
        Dup2(fd, STDOUT_FILENO); /* Redirect stdout to client */
        Execve(filename, emptylist, environ);
    }
    Wait(NULL);
}