Lecture 27: Sockets, Application Protocols

David Hovemeyer

April 12, 2023

601.229 Computer Systems Fundamentals

Example code

Today's example code is on course web page in `sockets.zip`

Unix sockets

Unix sockets

Unix sockets: API to allow programs to communicate over networks

Designed to work with many underlying protocols

Socket = "communications endpoint", appears to process as a file descriptor

Several important kinds of sockets:

- *Server socket*: used by server to accept connections from clients (not used for actual exchange of data)
- *Client socket*: used to exchange data between client and server systems

Socket system calls

Important socket system calls:

`socket`: create an unconnected socket

`bind`: associate a socket with a network interface identified by a network address

`listen`: make a socket a server socket (to allow incoming connections)

`accept`: wait for an incoming connection

`connect`: initiate a connection to a remote system

Socket addresses

Socket API designed to work with many underlying network technologies

`struct sockaddr`: "supertype" for all network addresses

- A "type" field is at beginning of struct to distinguish variants
- E.g. if type field contains `AF_INET`, it's an IP address

`struct sockaddr_in`: "subtype" for IP addresses

Create server socket

```
int create_server_socket(int port) {
  struct sockaddr_in serveraddr = {0};
  int ssock_fd = socket(AF_INET, SOCK_STREAM, 0);
  if (ssock_fd < 0)
    fatal("socket failed");

  serveraddr.sin_family = AF_INET;
  serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
  serveraddr.sin_port = htons((unsigned short)port);
  if (bind(ssock_fd, (struct sockaddr *) &serveraddr,
           sizeof(serveraddr)) < 0)
    fatal("bind failed");

  if (listen(ssock_fd, 5) < 0) fatal("listen failed");

  return ssock_fd;
}
```

Wait for incoming connection

```
int accept_connection(int ssock_fd, struct sockaddr_in clientaddr) {
  unsigned clientlen = sizeof(clientaddr);
  int childfd = accept(ssock_fd,
                       (struct sockaddr *) &clientaddr,
                       &clientlen);
  if (childfd < 0)
    fatal("accept failed");
  return childfd;
}
```

Server loop

```
int main(int argc, char **argv) {
  char buf[256];
  int port = atoi(argv[1]);
  int ssock_fd = create_server_socket(port);

  while (1) {
    struct sockaddr_in clientaddr;
    int clientfd = accept_connection(ssock_fd, &clientaddr);
    ssize_t rc = read(clientfd, buf, sizeof(buf));
    if (rc > 0) {
      write(clientfd, buf, rc);
    }
    close(clientfd);
  }
}
```

Testing the server

Run the server:

```
$ gcc -Wall -o server server.c
$ ./server 30000
```

Test using `telnet` program:

```
$ telnet localhost 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hey there!
hey there!
Connection closed by foreign host.
```

Implementation issues

- Reading from socket can return fewer bytes than requested (generally need to call `read` in a loop)
- Network connections can be broken (need to check result of `read` and `write`, error often indicates that the connection no longer exists)

Hostnames

DNS: Domain Name Service

Assign meaningful names (such as `ugradx.cs.jhu.edu`) to network addresses (such as `128.220.224.100`)

`getaddrinfo`: look up network address for hostname

csapp.h/csapp.c

The textbook (*Computer Systems: A Programmer's Perspective*) includes a library of convenient functions for writing network applications

`open_listenfd`: open a server socket given port name as string

`open_clientfd`: simplified interface for connecting to a server by specifying host name (or address) and port

`rio_` functions: Robust I/O routines, handle looping for short reads/writes and interruptions from signals automatically

- `rio_t`: data type wrapping a file descriptor and allowing buffered input
- `rio_readnb`: read n bytes from a `rio_t`
- `rio_readlineb`: read a line of input from a `rio_t`

Using these routines can significantly reduce the complexity of implementing network applications in C and C++

Application protocols

Application protocols

Application protocol: determines how data is exchanged by instances of an application program

- Usually: a server and a client
- Another possibility: peer to peer (P2P) applications

Example: HTTP, HyperText Transport Protocol

- Used by web browsers and web servers

Application protocols in 1 minute

Synchronous: The connected peers take turns talking

- Asynchronous protocols: possible, but significantly more complicated to implement

Client/server protocol: client sends request, server sends response

- Repeat as necessary

Message format: both peers must be able to determine where each message starts and ends

- Also, each peer must be able to determine the meaning of each received message

Text-based protocols are common because they are easy to debug and reason about

HTTP

A synchronous client/server protocol used by web browsers, web servers, web clients, and web services

- HTTP 1.1: `https://tools.ietf.org/html/rfc2616`

Client sends request to server, server sends back a response

- Each client request specifies a *verb* (GET, POST, PUT, etc.) and the name of a *resource*

Requests and responses may have a *body* containing data

- The body's *content type* specifies what kind of data the body contains

HTTP request example

Command:

```
curl -v http://placekitten.com/1024/768 -o kitten.jpg
```

Request sent by `curl` program:

```
GET /1024/768 HTTP/1.1
Host: placekitten.com
User-Agent: curl/7.58.0
Accept: */*
```

Request is sent via a TCP connection to port 80

HTTP response example

Response sent by `placekitten.com`:

```
HTTP/1.1 200 OK
Date: Wed, 13 Nov 2019 12:33:20 GMT
Content-Type: image/jpeg
Transfer-Encoding: chunked
Connection: keep-alive
Set-Cookie: __cfduid=de2a22cdd3ed939398e0a56f41ce0e4a31573648400; expires=T
Access-Control-Allow-Origin: *
Cache-Control: public, max-age=86400
Expires: Thu, 31 Dec 2020 20:00:00 GMT
CF-Cache-Status: HIT
Age: 51062
Server: cloudflare
CF-RAY: 5350c608682a957e-IAD
```

Headers were followed by a body containing 40,473 bytes of binary data

Kitten

Slightly more complete example

A simple client/server implementation

- Limitations of previous `server.c` example:
  - Only echoes back client message
  - No mechanism to request server to shut down
  - Uses raw system calls, code is somewhat complicated
- "Addition server":
  - Reads integer values, computes the sum, sends sum back to client
  - Client can sent `quit` message
  - Implemented using `csapp` functions: code is less complicated, more robust
  - Better starting point for your own clients and servers

Server main function

```
int main(int argc, char *argv[]) {
  if (argc != 2) { fatal("Usage: ./arithserver <port>"); }

  int server_fd = open_listenfd(argv[1]);
  if (server_fd < 0) { fatal("Couldn't open server socket\n"); }

  int keep_going = 1;
  while (keep_going) {
    int client_fd = Accept(server_fd, NULL, NULL);
    if (client_fd > 0) {
      keep_going = chat_with_client(client_fd);
      close(client_fd); // close the connection
    }
  }
  close(server_fd); // close server socket

  return 0;
}
```

Explanation of server main

- Uses `open_listenfd` to create server socket (`csapp` function)
- In main loop:
  - Call `Accept` to wait for client to connect (`csapp` function)
  - Call `chat_with_client` to read and decode request, do computation, send response back to client
  - `chat_with_client` can return 0 to end loop and shut down server

Server chat_with_client function

```
int chat_with_client(int client_fd) {
  rio_t rio; int sum = 0, val;
  rio_readinitb(&rio, client_fd);

  // Read line from client
  char buf[1024];
  ssize_t rc = rio_readlineb(&rio, buf, sizeof(buf));
  if (rc < 0) { return 1; } // error reading data from client

  if (strcmp(buf, "quit\n") == 0 || strcmp(buf, "quit\r\n") == 0) {
    return 0;
  } else {
    FILE *in = fmemopen(buf, (size_t) rc, "r");
    while (fscanf(in, "%d", &val) == 1) { sum += val; }
    fclose(in);
    snprintf(buf, sizeof(buf), "Sum is %d\n", sum);
    rio_writen(client_fd, buf, strlen(buf));
    return 1;
  }
}
```

Explanation of chat_with_client

- Use a `rio_t` object and `rio` functions for I/O
  - `rio` = "robust I/O"
  - Unbuffered reads/writes, ensures all data is read/written
  - Buffered reads (e.g., `rio_readlineb` to read a complete input line)
  - More suitable for network communication than C standard I/O: thread safe, buffered reads can be freely mixed with unbuffered reads
- Read message from client, scan for integer values, send sum as response
  - A more realistic implementation would have a loop to allow client to send multiple requests

Testing server using telnet

Running the server:

```
$ ./arithserver 40000
```

Testing using `telnet`:

```
$ telnet localhost 40000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
1 2 3
Sum is 6
Connection closed by foreign host.
```

`telnet` is quite useful for connecting to servers which support a plaintext-based protocol

Client implementation

```
int main(int argc, char *argv[]) {
  if (argc != 4) { fatal("Usage: ./arithclient <hostname> <port> <message>"); }

  int fd = open_clientfd(argv[1], argv[2]);
  if (fd < 0) { fatal("Couldn't connect to server"); }

  rio_writen(fd, argv[3], strlen(argv[3])); // send message to server
  rio_writen(fd, "\n", 1);

  rio_t rio;                                // read response from server
  rio_readinitb(&rio, fd);
  char buf[1000];
  ssize_t n = rio_readlineb(&rio, buf, sizeof(buf));

  if (n > 0) {                              // print response
    printf("Received from server: %s", buf);
  }
  close(fd);
  return 0;
}
```

Client implementation explanation

- Use `open_clientfd` to connect to server (`csapp` function)
- `rio_writen` to send data to server
- `rio_readlineb` to receive response from server
  - Note that received data is *not* NUL-terminated