Networking: Application-to-application communication; Sockets

COSC 208, Introduction to Computer Systems, 2021-11-29

Announcements

- Project 3 ?
- Attend faculty candidate research talks
    - 11:20am Tues, Nov 30; Thurs, Dec 2; Tues, Dec 7; Wed, Dec 15
    - BeyondCS replacement

Outline

- Warm-up
- Application-to-application communication
- HyperText Transfer Protocol (HTTP)
- Sockets
- Client application

Warm-up

Q1: *List at least five types of applications that communicate over a network*

- Web browser
- Video conferencing (e.g., Zoom)
- Games
- Video streaming (e.g., Netflix)
- Audio streaming (e.g., Spotify)
- Smart assistant (e.g., Siri, Alexa, Cortana, Google Assistant)
- Cloud storage (e.g., Dropbox, iCloud)
- Email
- Maps

Trajectory

- Individual applications — C, assembly language, program optimization
- Running multiple applications concurrently on the same hardware — enabled by operating systems, especially the process abstraction
- Running multiple applications concurrently on different machines — enabled by networking

Application-to-application communication

- *Assume you are running such applications on your device. Who/what are the applications communicating with?*
    - Servers — web server, video/audio streaming servers, email server, etc.
    - Other users' devices — possibly in the case of gaming or video conferencing
- Models
    - Client/server
        - Client: Application that initiates communication; typically sends "requests"; e.g., web browser asks for web page
        - Server: Application that waits for communication; typically sends "replies"; e.g., web server provides web page
    - Peer-to-peer
        - Peer: Application that both initiates and waits for communication; e.g., gaming application sends updates to peers and receives updates from peers
- Client/server communication with humans

    - *I'm going to say something and you should respond with whatever seems natural. I'm also going to display what I expected you would say; we'll see how close my prediction was.*
    - Greeting
        - Hi, I'm Aaron.
        - Hi, I'm NAME.
        - Where are you from?
        - I'm from LOCATION.
        - Were you born there?
        - Yes. ~OR~ No, I was born in LOCATION.
    - Joke
        - Knock knock.
        - Who's there?
        - Spell.
        - Spell who?
        - Okay, fine. W-H-O.
        - [Laughs or Groans]
    - Greeting 2
        - Hi, I'm Aaron.
        - Hi, I'm NAME.
        - Are you taking a computer science course next semester?
        - Yes. ~OR~ No.
        - Which course?
        - COURSE NAME ~OR~ COURSE NUMBER ~OR~ ???
    - Strange greeting
        - Goodbye, I'm Aaron.
        - ???
    - *What did you observe?*
        - Predictions were mostly correct
        - Not sure how to respond to unexpected message
    - *How does this relate to applications?* — applications need to agree on how they are going to communicate
- Protocol: a set of rules that govern how applications communicate
    - Client and server must follow the same protocol
    - Widely used protocols are standardized — RFCs (Requests for Comment) published by the Internet Engineering Task Force (IETF)
        - HyperText Transfer Protocol (HTTP)
        - Simple Mail Transfer Protocol (SMTP)
        - Secure SHell (SSH)
- Aside: networking is acronym soup!

HyperText Transfer Protocol (HTTP)

- Widely used application protocol — why?
    - Simple — only a few pieces of information must be included in requests/responses
    - Flexible — any type of data can be put in a response (e.g., HyperText Markup Language (HTML), image, video, Portable Document Format (PDF), etc.)
    - Plain-text protocol — contains "human-readable" words instead of numeric codes
- RFC 2616
- HTTP request

```
GET / HTTP/1.1\r\n
Host: www.example.com\r\n
\r\n
```

    - First line
        - Method — `GET`, `POST`, etc.

        - Uniform resource locator (URL), excluding domain name
        - Version — `HTTP/1.1`
    - Metdata
        - Host — domain name portion of url
        - User-Agent — web browser (or client application) name/version
        - Cookie — information used to identify a specific user
        - ...
    - Blank line
- HTTP reply

    ```
    HTTP/1.1 200 OK\r\n
    Content-Type: text/html\r\n
    \r\n
    <html>
    ...
    </html>
    ```

    - First line
        - Version — `HTTP/1.1`
        - Status — `200 OK`, `403 Forbidden`, `404 Not found`, `301 Moved permanently`, `418 I'm a teapot`, etc.
    - Metdata
        - Content-Type — type of data
        - Content-Length — size of data
        - ...
    - Blank line
    - Data

Sockets

- Application programming interface (API) exposed by the operating system
- Similar to file API
    - `open` ~ `socket` && (`listen` || `connect`)
    - `read` ~ `recv`
    - `write` ~ `send`
    - `close` ~ `close`
- Abstraction — low-level details are hidden from the application
    - Some is handled by the operating system — network stack
    - Some is handled by the network interface card (NIC) — hardware
    - Some is handled by wireless access points and routers
- Makes it easier to write an application — application developer only needs to know how the API works (to interact with the operating system)

Client application (live coding)

1. Create a socket

    ```
                                                      ↱ 0
    int sock = socket(int domain, int type, int protocol);
                    AF_INET ↲           ↳ SOCK_STREAM
    ```

    - Check for return values that indicate an error has occurred — usually `-1`
        - See the `RETURN VALUES` section of the man page
        - Use `perror` to print the error — e.g., `perror("socket failed");`
2. Establish a communication channel (i.e., connect) to the server

```
int connect(int socket, struct sockaddr *address, int addr_len);
              sock ↲          &server_addr ↲             ↳ sizeof(serveraddr)
```

- Address has two parts
    - Internet Protocol (IP) address ~ street address
        - IPv4 addresses consist of four decimal numbers, each in the range 0 to 255, separated by periods
        - E.g., `75.101.200.152` is `cosc208.cs.colgate.edu`'s IP address
        - `127.0.0.1` is a special address a machine uses to refer to itself
    - Port number ~ apartment number
        - Range 1 to 65536 — 16-bits, short
        - E.g., `80` is HyperText Transfer Protocol (HTTP)
- Address encoded in a `struct sockaddr_in`

```
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET:
                           ↳ meaning: Internet Protocol (IP) version 4
server_addr.sin_port = htons(80);
                              ↳ meaning: HyperText Transfer Protocol
(HTTP)
server_addr.sin_addr.s_addr = htonl(0x7F000001);
                                        ↳ meaning: 127.0.0.1 (localhost)
```

- Numbers must be expressed in network byte order (i.e., big endian)
    - Convert from host (little endian for ARM) to network byte order using `htons` for `short` and `htonl` for `int` or `long`
    - Convert from network to host byte order using `ntohs` for `short` and `ntohl` for `int` or `long`

3. Send/receive messages

```
int send(int socket, void *buffer, int buf_len, int flags);
int recv(int socket, void *buffer, int buf_len, int flags);
          sock ↲                                       ↳ 0
```

4. Close communication channel

```
int close(int socket);
           sock ↲
```

Full program

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define HTTP_PORT 80
#define WWW_EXAMPLE_COM 0x5DB8D822 // 93.184.216.34

int main() {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("socket failed");
        return 1;
    }

    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(HTTP_PORT);
    server_addr.sin_addr.s_addr = htonl(WWW_EXAMPLE_COM);

    int conn = connect(sock, (const struct sockaddr *) &server_addr,
sizeof(server_addr));
    if (conn < 0) {
        perror("connect failed");
        return 1;
    }

    char buf[1024];
    snprintf(buf, 1024, "GET / HTTP/1.1\r\nHost: www.example.com\r\n\r\n");
    int sent = send(sock, buf, strlen(buf), 0);
    if (sent < 0) {
        perror("send failed");
        return 1;
    }

    int received = recv(sock, buf, 1024, 0);
    if (received < 0) {
        perror("recv failed");
        return 1;
    }

    printf("%s\n", buf);

    close(sock);
}
```