Introduction to Networking

A. Scemama

21/11/2024

Laboratoire de Chimie et Physique Quantiques, Univ. Toulouse/CNRS

Introduction to Networking

Networking

- Networking: the practice of connecting computers and other devices to share resources, exchange information, and enable communication.
- Many different communication protocols (TCP/IP, Bluetooth, MIDI, . . .)
- Local Area Network (LAN): Small geographical area (office, home, university campus).
- Wide Area Network (WAN): Large geographical area, composed of multiple LANs (company offices in different countries, internet)

The OSI Model

- OSI: Open Systems Interconnection model
- Theoretical framework to standardize network communication
- Network communications are divided into 7 layers
- Each layer serves a specific function and communicates only with the layers directly above or below it

1. **Physical Layer**
   - Handles the physical connection between devices to transmit raw binary data
   - Example: Ethernet cables, wireless signals, switches, network cards

The OSI Model

2. **Data Link Layer**
    - Manages the direct connections between network devices and handles data transfer within a local network.
    - Responsible for MAC (Media Access Control) addressing, which uniquely identifies each device on a network.
    - Corrects errors from the physical layer
    - Example: Network switches using MAC addresses to forward data.
3. **Network Layer**
    - Manages IP (Internet Protocol) addressing and routing, determining the best path for data packets across networks.
    - Devices at this layer use IP addresses to forward data to its destination.
    - Example: Routers function at this layer by directing data to other networks.

The OSI Model

4. **Transport Layer**
   - Ensures reliable data transfer between devices, managing packet delivery, end-to-end error checking, and flow control.
   - Key Protocols: TCP (Transmission Control Protocol) for *reliable* communication and UDP (User Datagram Protocol) for faster, less reliable communication.
   - Example: Online video streaming uses UDP, while file downloads use TCP

5. **Session Layer**
   - Manages sessions between applications, establishing, maintaining, and terminating connections
   - Allows multiple connections to exist independently and can resume broken connections
   - Example: Session management in applications like remote desktop.

The OSI Model

6. **Presentation Layer**
    - Transforms data between network and application formats, handling encryption, compression, and translation.
    - Ensures data is readable by the application layer, supporting data encoding, encryption, and compression.
    - Example: Data encryption in HTTPS.
7. **Application Layer**
    - The top layer where applications communicate over the network using protocols suited to their specific needs.
    - Examples: HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol), SSH (Secure SHell), DNS (Domain Name System).

Sending a letter

- 7: Application: Alice writes a letter with her message for Bob
- 6: Presentation: Alice decides to use a specific language that Bob understands
- 5: Session: Alice confirms that Bob's address is correct and arranges to send the letter, expecting it will reach him directly
- 4: Transport: If Alice's letter is too long for one envelope, she divides it into multiple parts, numbering each piece so Bob can reassemble it in the correct order
- 3: Network: Alice writes Bob's address on the envelope, indicating where the letter should go.
- 2: Data Link: Alice places the letter in an envelope with a local postal code, enabling the local post office to send it to the correct regional office.
- 1: Physical: The letter is physically transported by truck, plane, or postal carrier to reach Bob's mailbox.

Receiving a letter

- 1: Physical: The letter arrives at Bob's mailbox
- 2: Data Link: The postal service verifies it was sent to the correct local address
- 3: Network: Nothing
- 4: Transport: Bob puts the parts of the letter in the correct order, if it was split
- 5: Session: Bob verifies the letter was meant for him and acknowledges it
- 6: Presentation: Bob decrypts or interprets the letter's content.
- 7: Application: Bob reads Alice's message

The TCP/IP Model

- Simpler, four-layer model developed for the internet, grouping some OSI layers into broader categories
- Used in internet-based communications
  1. Link Layer (Network Access): Combines OSI layers 1 and 2, covering physical network connection and data link functions.
  2. Internet Layer: Equivalent to OSI Layer 3, handling IP addressing and packet routing.
  3. Transport Layer: Mirrors OSI Layer 4, managing data transfer with protocols like TCP and UDP.
  4. Application Layer: Combines OSI layers 5, 6, and 7, dealing directly with network applications like HTTP and FTP.

IP Addressing

IP Addresses

- An IP (Internet Protocol) address is a unique identifier assigned to each device on a network, allowing it to communicate with other devices. (An IP address is like a postal address for your device)
- IPv4: The most common form of IP addressing, consisting of four numbers separated by periods (e.g., 192.168.1.1). It uses a 32-bit format, supporting about 4.3 billion unique addresses.
- IPv6: A newer IP format developed due to the limitations of IPv4 address space. IPv6 uses a 128-bit format, providing a significantly larger address space.

Public *vs* Private IP Addresses

- Public: Unique and accessible from outside the local network. Used for devices on the internet.
- Private: Used within private networks (e.g., home or office), not routable on the internet. Examples include addresses in ranges like `192.168.0.0` to `192.168.255.255`
- `127.0.0.1` is called localhost. It is a loopback address, telling the machine to refer to itself

IPv4 Address structure

- Format: IPv4 addresses consist of four octets (e.g., `192.168.1.1`), each representing 8 bits, totaling 32 bits.
- Class A: Large networks, IP range `1.0.0.0` to `126.0.0.0`
- Class B: Medium-sized networks, IP range `128.0.0.0` to `191.255.0.0`
- Class C: Small networks, IP range `192.0.0.0` to `223.255.255.0`.
- Network Portion: Identifies the network to which the device belongs.
- Host Portion: Identifies the specific device within that network.

Subnetting

- Method for dividing large networks into smaller, more manageable subnetworks
- A subnet mask is a 32-bit number that helps identify the network and host portions of an IP address.

A	255.0.0.0	11111111.00000000.0000000.00000000
B	255.255.0.0	11111111.11111111.0000000.00000000
C	255.255.255.0	11111111.11111111.1111111.00000000

- CIDR Notation: Classless Inter-Domain Routing. Indicates the number of network bits: `192.168.1.0/24`, where /24 indicates 24 bits for the network portion

Subnetting Example

- Consider a network with the IP address range `192.168.1.0/24`, which has 256 addresses.
- Task: Divide this range into four smaller networks
    - `192.168.1.0/26` (addresses 0–63)
    - `192.168.1.64/26` (addresses 64–127)
    - `192.168.1.128/26` (addresses 128–191)
    - `192.168.1.192/26` (addresses 192–255)
- Each subnet now has 64 addresses, suitable for smaller segments like office floors or departments.

Packet Flow in Networks

Data packets

- A packet is a small, formatted unit of data transmitted over a network.
- Large files are divided into packets to make data transmission more manageable, efficient, and resilient to network issues
- Structure:
  - Header: Contains metadata like source and destination IP addresses, protocol information, and error-checking data.
  - Payload: The actual data being transmitted, which could be part of an email, file, or other information.

Packet creation and sending process

- Application Layer: The data starts at the application layer, where it's generated by an application (e.g., an email client or web browser).
- Encapsulation: Each OSI layer from the transport layer down to the physical layer adds information (like IP addresses, port numbers, and error-checking codes) as the data moves downward.
- Segmentation and Packetization: The transport layer breaks the data into segments, adds headers (e.g., TCP/UDP), and encapsulates each segment into packets.
- Packet is Ready for Transmission: After packetization, the packet is passed to the physical layer for transmission.

Packet Flow Through Network Devices

- Switches (Data Link Layer): When the packet arrives at a switch, the switch checks its MAC address table to determine the correct path to the destination within the same local network.
- Routers (Network Layer): Directs packets between different networks based on IP addresses. examines the destination IP address in the packet header to determine the next best path toward the destination network. Routers use routing tables to make these decisions.
- Firewalls: Security devices that monitor and control network traffic based on predetermined security rules. When a packet passes through a firewall, it checks the packet's headers against its rules to allow, block, or redirect the packet as needed.

Packet arrival at the destination

- Physical Layer: The physical layer receives the incoming bits and passes them up to the data link layer.
- Data Link Layer: Checks for errors and passes the frames up to the network layer.
- Network Layer: Verifies the IP address to confirm it matches the destination device's IP, then passes the data up to the transport layer.
- Transport Layer: Reassembles segmented data using sequence numbers (for protocols like TCP) to ensure packets are in the correct order and handles any retransmissions if packets are missing.
- Application Layer: Delivers the fully reassembled and decapsulated data to the receiving application.

Error checking

- Checksums and Cyclic Redundancy Checks (CRC): Each layer adds error-checking data to verify the packet's integrity at the receiving end.
- Retransmission Requests: If errors are detected (e.g., missing or corrupted packets), the transport layer (TCP) requests retransmission of those packets to complete the data correctly.

Network Protocols

Network Protocols

- Transport layer protocols: TCP, UDP
- Internet Layer Protocols:
  - IP
  - ICMP: Internet Control Message Protocol. Used for error messages and network diagnostics. `ping` and `traceroute`
  - ARP: Address Resolution Protocol. Maps IP addresses to MAC addresses.

Application Layer Protocols

- HTTP/HTTPS: The protocol used for transmitting web pages. It's a request-response protocol where the client requests resources, and the server responds.
- FTP/SFTP: Used for transferring files between a client and server
- SMTP :Simple Mail Transfer Protocol. Used to send emails
- POP3/IMAP: Used to retrieve emails
- DNS: Domain Name System. Translates human-readable domain names (e.g., www.example.com) into IP addresses that computers use. `nslookup`

Network port

- Port: logical endpoint used by the OS to direct network traffic to the correct application
- 16-bit number (ranging from 0 to 65535) used in conjunction with an IP address to identify specific processes or services that a computer is running on a network
- Example: `192.168.2.10:80` is port 80 on host `192.168.2.10` (refers to the HTTP server)
- 0-1023: Reserved for core services and protocols widely used across the internet (HTTP, FTP, SSH, DNS, . . .)
- 1024-49151: Assigned to specific services or applications (MySQL, Docker, . . .)
- 49152–65535: Ephemeral ports used for temporary, short-lived connections.

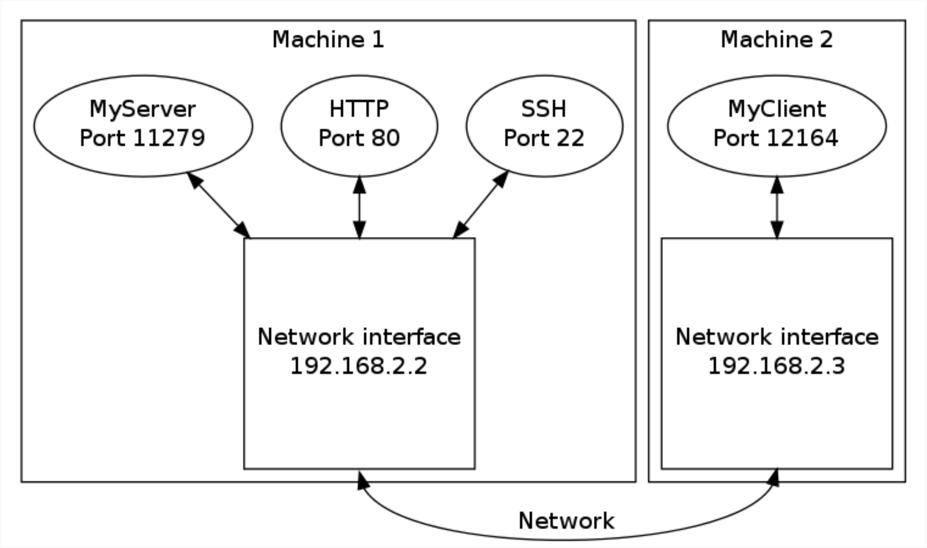
Browsing a web page

When a user types a URL into a browser:

1. DNS (Domain Name System): The browser uses DNS to resolve the domain name to an IP address.
2. TCP Connection Establishment: The browser initiates a TCP connection to the server IP on port 80 (HTTP) or 443 (HTTPS).
3. HTTP/HTTPS Request: The browser sends an HTTP or HTTPS request to retrieve the page.
4. Data Transmission: The server responds with the requested data, which is packetized and sent back to the client.
5. Decapsulation and Display: The client receives, decapsulates, and reassembles the data, displaying the web page to the user.

Exercises

Client-server program in Python and C



Server program in Python i

```
#!/usr/bin/env python
import socket, datetime          # socket is used for network communication
now = datetime.datetime.now      # Function that gets the current date and time

def main():
   HOSTNAME = socket.gethostname()      # Get the hostname of the server machine
   PORT     = 11279                     # Define the port number for the server to listen on
   print(now(), "I am the server : %s:%d"%(HOSTNAME,PORT))

   s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Create an IPv4 TCP socket
   s.bind( (HOSTNAME, PORT) )  # Bind the socket to the hostname and port
   s.listen()                  # Set the socket to listen for incoming connections
   conn, addr = s.accept()     # Block until a client connects, then return a new socket conn
   print(now(), "Connected by", addr)
```

Server program in Python ii

```python
    data = ""                # Initialize an empty string to store the received data
    while True:              # Start a loop to read incoming data from the client
        message = conn.recv(8).decode('utf-8')      # Read up to 8 bytes
        print(now(), "Buffer : "+message)
        data += message            # Append the message to the data buffer
        if len(message) < 8:       # If received data is less than 8 bytes, break (end of connection)
           break
    print(now(), "Received data : ", data)

    print(now(), "Sending thank you...")
    conn.send("Thank you".encode())         # Send a response back to the client
    conn.recv(1)                            # Wait until the client closes the connection
    conn.close()                            # Close the connection to the client

if __name__ == "__main__": main()
```

Client program in Python i

```python
#!/usr/bin/env python
import socket, datetime   # socket is used for network communication
import sys, os            # handle command-line arguments and system operations

now = datetime.datetime.now      # Function that gets the current date and time

def main():
   HOSTNAME = sys.argv[1]       # Get the server hostname from the command line argument
   PORT     = int(sys.argv[2]) # Get the server port from the command line argument
   print(now(), "The target server is : %s:%d"%(HOSTNAME,PORT))

   s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Create an IPv4 TCP socket
   s.connect((HOSTNAME, PORT))    # Connect to the specified server at HOSTNAME and PORT
   message = "Hello, world!!!!!!!"
   print(now(), "Sending : "+message)
```

Client program in Python ii

```
s.send(message.encode()) # Encode the message as bytes and send it over the connection

data = s.recv(1024)  # Receive up to 1024 bytes from the server's response
s.close()            # Close the socket connection after receiving the server response
print(now(), 'Received: ', data.decode('utf-8'))

if __name__ == "__main__": main()
```

Server program in C i

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <time.h>

#define BUFFER_SIZE 8

void print_current_time(const char *message) {
    // Function to print the current time with a message
    time_t now = time(NULL);
    struct tm *t = localtime(&now);
    printf("%04d-%02d-%02d %02d:%02d:%02d %s\n",
```

Server program in C ii

```
            t->tm_year + 1900, t->tm_mon + 1, t->tm_mday,
            t->tm_hour, t->tm_min, t->tm_sec, message);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {  // Ensure a port number is provided
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        return 1;
    }

    int port = atoi(argv[1]);  // Get the port number from the command line
    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_size = sizeof(client_addr);

    // Step 1: Create a socket
```

Server program in C iii

```c
server_sock = socket(AF_INET, SOCK_STREAM, 0);  // IPv4, TCP socket
if (server_sock == -1) {
    perror("Socket creation failed");
    return 1;
}

// Step 2: Bind the socket to the specified port
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;                // IPv4
server_addr.sin_addr.s_addr = INADDR_ANY;        // Bind to any available network interface
server_addr.sin_port = htons(port);              // Convert port to network byte order

if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("Bind failed");
    close(server_sock);
    return 1;
```

Server program in C iv

```
}

// Step 3: Listen for incoming connections
if (listen(server_sock, 5) < 0) {  // Allow up to 5 queued connections
    perror("Listen failed");
    close(server_sock);
    return 1;
}

print_current_time("Server is listening...");

// Step 4: Accept an incoming connection
client_sock = accept(server_sock, (struct sockaddr *)&client_addr, &client_addr_size);
if (client_sock < 0) {
    perror("Accept failed");
    close(server_sock);
```

Server program in C v

```c
        return 1;
    }
    print_current_time("Client connected");

    // Step 5: Receive data from the client
    char buffer[BUFFER_SIZE + 1];  // Buffer for incoming data (+1 for null-terminator)
    char data[1024] = "";          // Buffer to store all received data
    int bytes_received;

    while ((bytes_received = recv(client_sock, buffer, BUFFER_SIZE, 0)) > 0) {
        buffer[bytes_received] = '\0';  // Null-terminate the received chunk
        print_current_time("Buffer received:");
        printf("%s\n", buffer);
        strncat(data, buffer, bytes_received);  // Append buffer to data
        if (bytes_received < BUFFER_SIZE) break;  // End of transmission
    }
```

Server program in C vi

```
if (bytes_received < 0) {
    perror("Receive failed");
    close(client_sock);
    close(server_sock);
    return 1;
}
print_current_time("Complete message received:");
printf("%s\n", data);

// Step 6: Send a response to the client
const char *response = "Thank you";
print_current_time("Sending thank you message...");
send(client_sock, response, strlen(response), 0);

// Step 7: Close the connections
```

Server program in C vii

```
    bytes_received = recv(client_sock, buffer, 1, 0);  // Ensure the client has disconnected
    close(client_sock);
    close(server_sock);
    print_current_time("Server shut down");

    return 0;
}
```

Client program in C i

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <time.h>

void print_current_time(const char *message) {
    // Function to print the current time with a message
    time_t now = time(NULL);
    struct tm *t = localtime(&now);
    printf("%04d-%02d-%02d %02d:%02d:%02d %s\n",
           t->tm_year + 1900, t->tm_mon + 1, t->tm_mday,
           t->tm_hour, t->tm_min, t->tm_sec, message);
```

Client program in C ii

```
}

int main(int argc, char *argv[]) {
    if (argc != 3) {  // Ensure hostname and port are provided
        fprintf(stderr, "Usage: %s <hostname> <port>\n", argv[0]);
        return 1;
    }

    const char *hostname = argv[1];       // Server hostname from command line
    int port = atoi(argv[2]);             // Server port from command line

    // Step 1: Create the socket
    int sock = socket(AF_INET, SOCK_STREAM, 0);  // IPv4, TCP socket
    if (sock == -1) {
        perror("Socket creation failed");
        return 1;
```

Client program in C iii

```
}

// Step 2: Define the server address structure
struct sockaddr_in server_addr;
memset(&server_addr, 0, sizeof(server_addr));      // Zero out the structure
server_addr.sin_family = AF_INET;                  // IPv4
server_addr.sin_port = htons(port);                // Convert port to network byte order

// Step 3: Convert hostname to IP address and set it in server_addr
if (inet_pton(AF_INET, hostname, &server_addr.sin_addr) <= 0) {
    perror("Invalid address or address not supported");
    close(sock);
    return 1;
}

// Step 4: Connect to the server
```

Client program in C iv

```
if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("Connection failed");
    close(sock);
    return 1;
}

print_current_time("Connected to the server");

// Step 5: Send a message to the server
const char *message = "Hello, world!!!!!!!";
print_current_time("Sending message: Hello, world!!!!!!!");
if (send(sock, message, strlen(message), 0) < 0) {
    perror("Send failed");
    close(sock);
    return 1;
}
```

Client program in C v

```c
// Step 6: Receive a response from the server
char buffer[1024];
int bytes_received = recv(sock, buffer, sizeof(buffer) - 1, 0);
if (bytes_received < 0) {
    perror("Receive failed");
    close(sock);
    return 1;
}

buffer[bytes_received] = '\0';  // Null-terminate the received data
print_current_time("Received message:");
printf("%s\n", buffer);         // Print the received message

// Step 7: Close the socket
close(sock);
```

Client program in C vi

```
    print_current_time("Connection closed");

    return 0;
}
```