Network Programming

BESE-VI – Pokhara University

Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np

Chapter 6:
Network utilities, Current Trends and Emerging Technologies in Network Programming
(5 hrs)

Outline

1. Network Utilities and Applications
   - Introduction to ping, telnet, ip/ifconfig, iperf, netstat, remote login
2. Real-Time Communication Protocols
   - WebSockets: Full-duplex communication over a single TCP connection
   - gRPC: High-performance RPC framework
3. Security in Network Programming
   - TLS/SSL: Introduction to secure socket programming
4. Software-Defined Networking (SDN)
   - Overview of SDN concepts
   - Introduction to OpenFlow protocol and SDN controllers
5. Introduction to P4 and frenetic programming

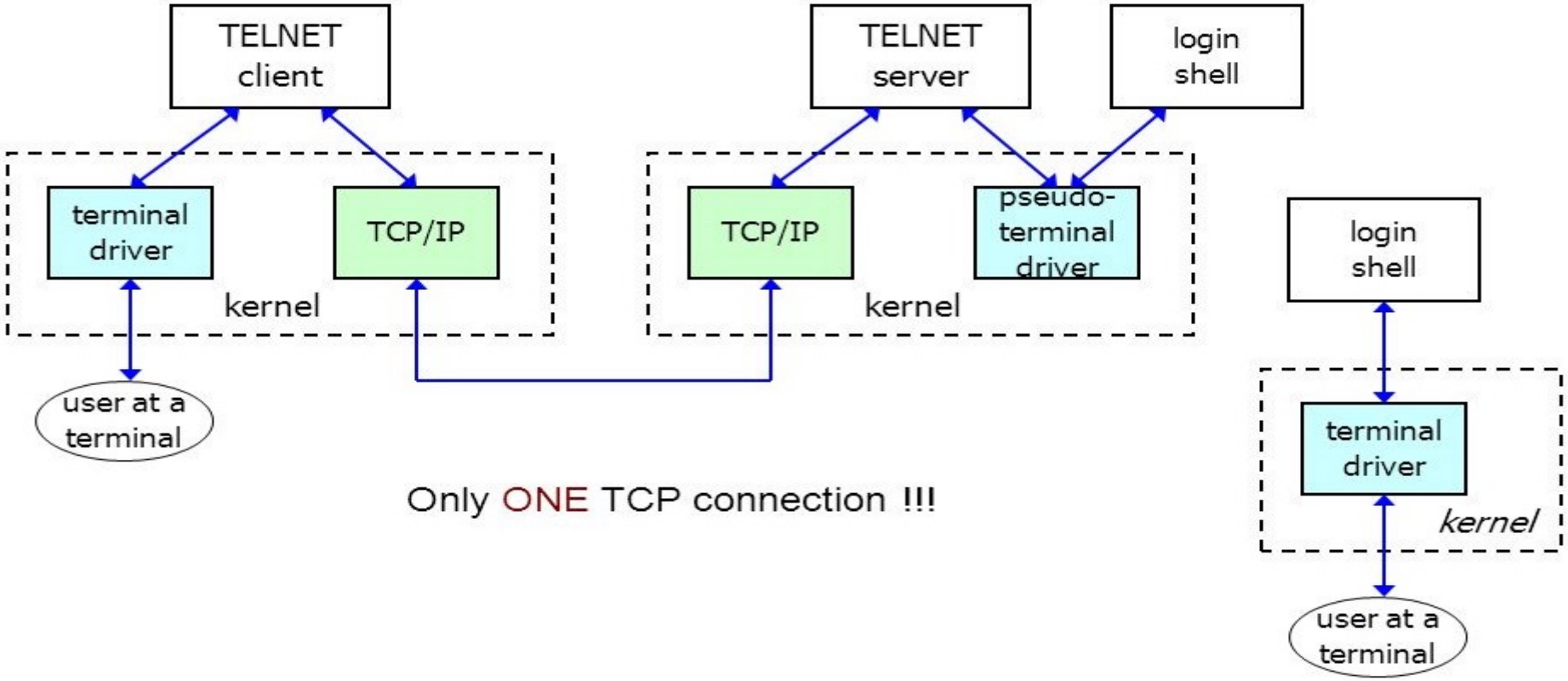
Network Utilities and Applications

Telnet and Rlogin: Remote Login

- Remote login is one of the most popular Internet applications. Instead of having a hardwired terminal on each host, we can login to one host and then remote login across the network to any other host (that we have an account on, of course).
- Two popular applications provide remote login across TCP/IP internets.

1. **Telnet** is a standard application that almost every TCP/IP implementation provides. It works between hosts that use different operating systems. Telnet uses option negotiation between the client and server to determine what features each end can provide.
2. **Rlogin** is from Berkeley Unix and was developed to work between Unix systems only, but it has been ported to other operating systems also.

- Remote login uses the client-server paradigm.
- Fig: Overview of Telnet client-server.



1. The Telnet client interacts with both the user at the terminal and the TCP/IP protocols. Normally everything we type is sent across the TCP connection, and everything received from the connection is output to our terminal.
2. The Telnet server often deals with what's called a *pseudo-terminal* device, at least under Unix systems. This makes it appear to the login shell that's invoked on the server, and to any programs run by the login shell, that they're talking to a terminal device.
3. Only a single TCP connection is used. Since there are times when the Telnet client must talk to the Telnet server (and vice versa) there needs to be some way to delineate commands that are sent across the connection, versus user data.
4. In dashed boxes in Figure to note that the terminal and pseudo terminal drivers, along with the TCP/IP implementation, are normally part of the operating system kernel. The Telnet client and server, however, are often user applications.
5. The login shell on the server host to reiterate that we have to login to the server. We must have an account on that system to login to it, using either Telnet or Rlogin.

- Remote login is not a high-volume data transfer application. Lots of small packets are normally exchanged between the two end systems. It is found that the ratio of bytes sent by the client (the user typing at the terminal) to the number of bytes sent back by the server is about 1:20. This is because we type short commands that often generate lots of output.

Rlogin Protocol

- Rlogin appeared with 4.2BSD and was intended for remote login only between Unix hosts. This makes it a simpler protocol than Telnet, since option negotiation is not required when the operating system on the client and server are known in advance.
- **Application Startup:**

❖ Rlogin uses a single TCP connection between the client and server. After the normal TCP connection establishment is complete, the following application protocol takes place between the client and server.

1. The client writes four strings to the server; (a) a byte of 0, (b) the login name of the user on the client host terminated by a byte of 0, (c) the login name of the user on the server host, terminated by a byte of 0, (d) the name of the user's terminal type, followed by a slash, followed by the terminal speed, terminated by a byte of 0. Two login names are required because users aren't required to have the same login name on each system. The terminal type is passed from the client to the server because many full-screen applications need to know it. The terminal speed is passed because some applications operate differently depending on the speed. For example, the vi editor works with a smaller window when operating at slower speeds, so it doesn't take forever to redraw the window.
2. The server responds with a byte of 0.

Rlogin Protocol

3. The server has the option of asking the user to enter a password. This is handled as normal data exchange across the Rlogin connection-there is no special protocol. The server sends a string to the client (which the client displays on the terminal), often Password:. If the client does not enter a password within some time limit (often 60 seconds), the server closes the connection. We can create a file in our home directory on the server (named .rhosts) with lines containing a hostname and our username. If we login from the specified host with that username, we are not prompted for a password. If we are prompted by the server for a password, what we type is sent to the server as *cleartext.* Each character of the password that we type is sent as is. Anyone who can read the raw network packets can read the characters of our password. Newer implementations of the Rlogin client, such as 4.4BSD, first try to use Kerberos, which avoids sending cleartext passwords across the network. This requires a compatible server that also supports Kerberos.
4. The server normally sends a request to the client asking for the terminal's window size

Rlogin Protocol

- **Flow Control**
- By default, flow control is done by the Rlogin client. The client recognizes the ASCII STOP and START characters (Control-S and Control-Q) typed by the user, and stops or starts the terminal output.
- **Client Interrupt**
- It is rare for the flow of data from the client to the server to be stopped by flow control. This direction contains only characters that we type. Therefore it is not necessary for these special input characters (Control-S or interrupt) to be sent from the client to the server using TCP's urgent mode.
- **Window Size Changes**
- With remote login, however, the change in the window size occurs on the client, but the application that needs to be told is running on the server. Some form of notification is required for the Rlogin client to tell the server that the window size has changed, and what the new size is.

- **Server to Client Commands :** the four commands that the Rlogin server can send to the client across the TCPconnection. The problem is that only a single TCP connection is used, so the server needs to mark these command bytes so the client knows to interpret them as commands, and not display the bytes on the terminal.

0x02	Flush output. The client discards all the data received from the server, up through the command byte (the last byte of urgent data). The client also discards any pending terminal output that may be buffered. The server sends this command when it receives the interrupt key from the client.
0x10	The client stops performing flow control.
0x20	The client resumes flow control processing.
0x80	The client responds immediately by sending the current window size to the server, and notifies the server in the future if the window size changes. This command is normally sent by the server immediately after the connection is established.

- **Client to Server Commands**
- Only one command from the client to the server is currently defined: sending the current window size to the server. Window size changes from the client are not sent to the server unless the client receives the command 0x80 from the server.

Telnet Protocol

- Telnet was designed to work between any host (i.e., any operating system) and any terminal. Its specification defines the lowest common denominator terminal, called the *network virtual terminal* (NVT). The NVT is an imaginary device from which both ends of the connection, the client and server, map their real terminal to and from. That is, the client operating system must map whatever type of terminal the user is on to the NVT. The server must then map the NVT into whatever terminal type the server supports.
- The NVT is a character device with a keyboard and printer. Data typed by the user on the keyboard is sent to the server, and data received from the server is output to the printer. By default the client echoes what the user types to the printer.
- Telnet uses TCP to transmit data and telnet control information. The default port for telnet is TCP port 23. Telnet, however, predates TCP/IP and was originally run over Network Control Program (NCP) protocols.

Telnet Commands

- Telnet uses in-band signaling in both directions. The byte 0xff (255 decimal) is called IAC, for "interpret as command." The next byte is the command byte. To send the data byte 255, two consecutive bytes of 255 are sent.

Name	Code (decimal)	Description
EOF	236	end-of-file
SUSP	237	suspend current process (job control)
ABORT	238	abort process
EOR	239	end of record
SE	240	suboption end
NOP	241	no operation
DM	242	data mark
BRK	243	break
IP	244	interrupt process
AO	245	abort output
AYT	246	are you there?
EC	247	escape character
EL	248	erase line
GA	249	go ahead
SB	250	suboption begin
WILL	251	option negotiation (Figure 26.9)
WONT	252	option negotiation
IX)	253	option negotiation
DONT	254	option negotiation
IAC	255	data byte 255

- **Option Negotiation**
  - Although Telnet starts with both sides assuming an NVT, the first exchange that normally takes place across a Telnet connection is option negotiation. The option negotiation is symmetric - either side can send a request to the other. Either side can send one of four different requests for any given option.
- 1. **WILL**. The sender wants to enable the option itself.
- 2. **DO**. The sender wants the receiver to enable the option.
- 3. **WONT**. The sender wants to disable the option itself.
- 4. **DONT**. The sender wants the receiver to disable the option.
- Since the rules of Telnet allow a side to either accept or reject a request to enable an option (cases 1 and 2 above), but require a side to always honor a request to disable an option (cases 3 and 4 above), these four cases lead to the six scenarios shown as follows.

	Sender		Receiver	Description
1.	WILL	-> <-	DO	**sender** wants to **enable** option receiver says OK
2.	WILL	-> <-	DONT	**sender** wants to **enable** option receiver says NO
3.	DO	-> <-	WILL	sender wants **receiver** to **enable** option receiver says OK
4.	DO	-> <-	WONT	sender wants **receiver** to **enable** option receiver says NO
5.	WONT	-> <-	DONT	**sender** wants to **disable** option receiver must say OK
6.	DONT	-> <-	WONT	sender wants **receiver** to **disable** option receiver must say OK

- Table: Six scenarios for Telnet option negotiation.

netstat

- It means network statistics. The netstat is a command-line network utility tool to display the information about network connections. It can
- **1) display the routing table**
- **netstat –r** : shows routing table i.e. destination, gateway, genmask, flags, network interface, etc.
- **2) display interface statistics**
- **netstat –i** : displays statistics for the network interfaces currently configured. If the –a option is also given, it prints all interfaces present in the kernel, not only those that have been configured currently.
- **3) display network connections**
- **netstat –p tcp:** displays the information about TCP connections. The netstat provides statistics for the following: proto, local address, foreign address, TCP states.

The ifconfig/ipconfig

- **ipconfig** – Windows
- **ifconfig** – Unix and Unix-like systems
- The ifconfig stands for "interface configuration". It is used to assign an address to a network interface and/or configure network interface parameters.

- **Examples**
- **List interfaces (only active)**
  - ifconfig
- **List all interfaces**
  - ifconfig –a
- **Display the configuration of device eth0 only**
  - ifconfig eth0
- **Enable and disable an interface**
  - sudo ifconfig eth1 up
  - sudo ifconfig eth1 down
- **Configure an interface**
  - sudo ifconfig eth0 inet 192.168.0.10 netmask 255.255.255.0

iperf

- *iperf* is a commonly used **network testing tool** that measures **network bandwidth and performance** between two hosts. It's useful for diagnosing issues, benchmarking throughput, or stress-testing network infrastructure.

Key Features of iperf:

- Measures **TCP** and **UDP** bandwidth.
- Supports **IPv4** and **IPv6**.
- Reports **jitter**, **packet loss**, and **retransmissions**.
- Allows **server-client** mode.
- Supports **multiple parallel streams**.
- Works cross-platform (Linux, Windows, macOS).

iperf

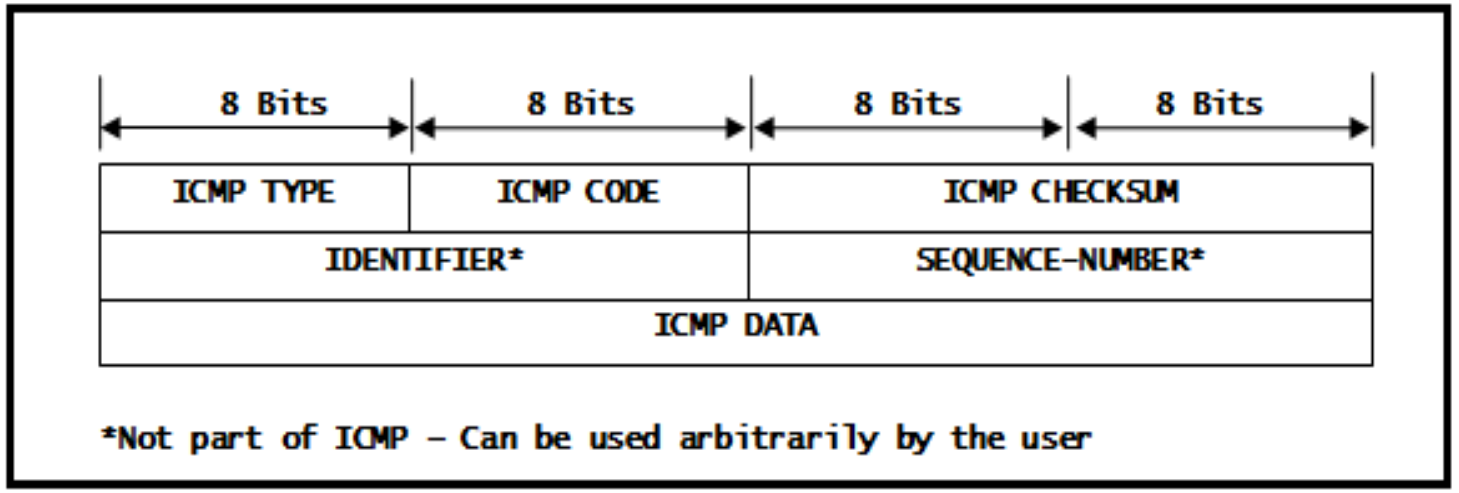
Common Options:

Option	Description
-s	Run in server mode
-c	Run in client mode (followed by server IP)
-u	Use UDP instead of TCP
-t	Duration of test in seconds (default: 10 sec)
-p	Specify port number
-b	Bandwidth to send at (UDP only, e.g., -b 100M)
-P	Number of parallel client streams
-R	Reverse test (server sends to client)

The ping

- Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network.
- It measures the round-trip time for messages sent from the originating host to a destination computer and echoed back to the source.
- Ping operates by sending Internet Control Message Protocol (ICMP) Echo Request packets to the target host and waiting for an ICMP Echo Reply.
- The results of the test usually include a statistical summary of the results, including the minimum, maximum, the mean, round-trip time, and usually standard deviation of the mean.
- Example: ping –c 4 www.google.com // c = packet counts

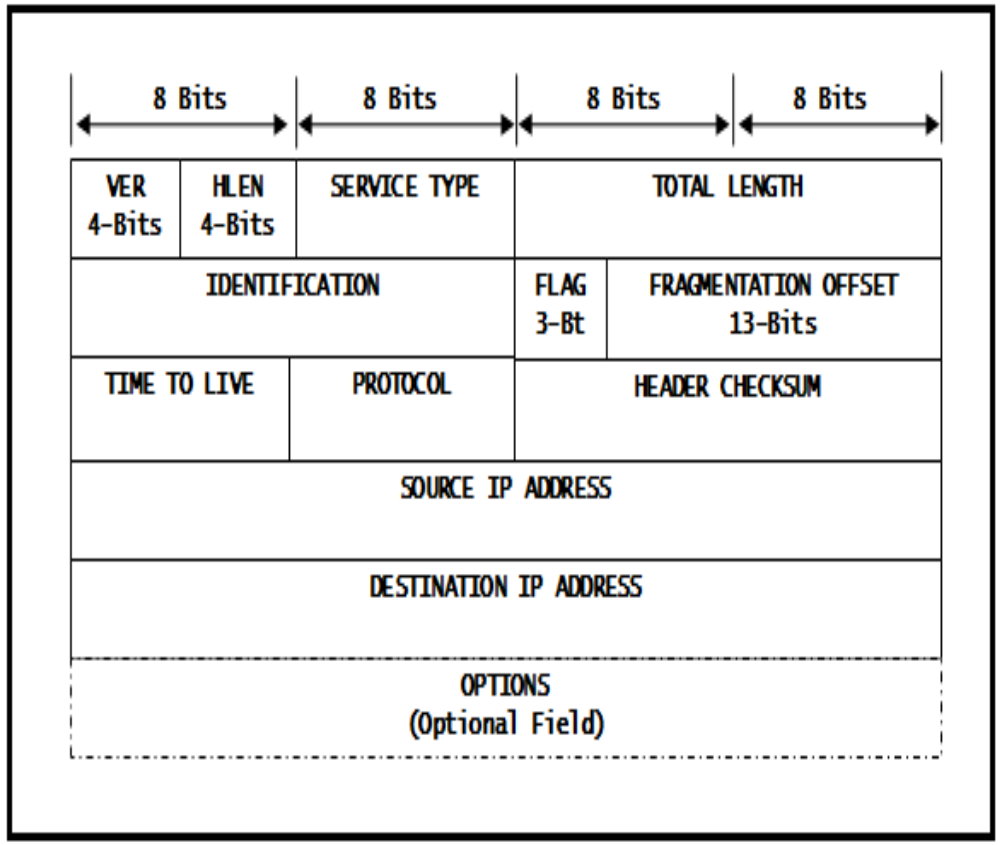
Frame format – ICMP



1. **ICMP TYPE** shall be set to **0x08** since this is an 'Echo-Request' message
2. **ICMP CODE** shall always be **0x00** for PING message
3. **ICMP CHECKSUM** is for header and data and is **'0xA5, 0x51'** for our message
4. **ICMP DATA** is "PING data to be sent" defined above

- Before sending the ICMP encapsulated PING command to MAC layer, the messages should be encapsulated into IP datagrams. Below is the frame-format of IPv4 packet:

Frame format - IP



1. **VER** shall be set to **4** since it's a IPv4 packet
2. **HLEN** is header length in 'lwords' - It shall be set to 5 here since the header length is 20Bytes
3. **SERVICE TYPE** shall be set to **0x00** since ICMP is a normal service
4. **TOTAL LENGTH** shall be set as **'0x00 0x54'** bytes, which includes header and data length
5. **IDENTIFICATION** is the unique identity for all datagrams sent from this source IP – Let's set it as **'0x96, 0xA1'** for our packet
6. **FLAG** and **FRAGMENTATION OFFSET** is set to **0x00, 0x00** since we don't intent to fragment the packet. **Reason**: The packet that we are sending here is smaller than a WLAN frame size
7. **TIME TO LIVE** shall be set to **0x40** since we want to discard the packet after 64 hops
8. **PROTOCOL** is set to **0x01** since it's an ICMP packet
9. Next 2Bytes is **HEADER CHECKSUM** which shall be set to **0x57, 0xFA** in our case
10. **SOURCE IP ADDRESS** is set to **0xc0, 0xa8, 0x01, 0x64** (which is 192.168.1.100)
11. **DESTINATION IP ADDRESS** is set to **0xc0, 0xa8, 0x01, 0x65** (which is 192.168.1.101)
12. **OPTIONS** field is left blank

Real-Time Communication Protocols

WebSockets: Full-duplex communication over a single TCP connection

- WebSockets provide a full-duplex, bidirectional communication channel between a client (usually a browser) and a server over a single long-lived TCP connection.
- This is ideal for applications that require real-time updates, such as:
  - Chat applications
  - Online gaming
  - Live sports updates
  - Stock trading dashboards
  - Collaborative editing tools (e.g., Google Docs)
- WebSockets allow:
  - Client and server to send messages independently and simultaneously.
  - Both parties can communicate without waiting for a request or response.

WebSockets: Full-duplex communication over a single TCP connection

How it works? (Connection Establishment)

- **Handshake Process:**

1. **Client sends an HTTP request** with headers:

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

2. **Server replies if supported:**

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

3. After this, the TCP connection stays open and the protocol switches to WebSocket frames.

WebSockets: Full-duplex communication over a single TCP connection

Advantages of WebSockets

- Real-time, low-latency communication
- Less overhead than HTTP polling
- Efficient for apps needing instant updates

Demo:
Start server: node server.js
Open client.html in browser

Limitations /Considerations

- Needs proxy/firewall/WebSocket support
- Not cacheable like HTTP
- Slightly complex to implement securely (use wss://)

Comparison: HTTP vs WebSocket

Feature	HTTP	WebSocket
Communication	Half-duplex (request-response)	Full-duplex (both ways)
Persistent connection	No	Yes
Protocol	Text-based	Binary or text frames
Overhead	High	Low (after upgrade)

gRPC: High-Performance RPC Framework

- *gRPC (Google Remote Procedure Call)* is a modern open-source RPC framework designed for high-performance, low-latency communication between distributed systems and microservices.
- Built on top of HTTP/2 (*HTTP/2 is a major revision of the HTTP network protocol, designed to improve web page load times and efficiency by reducing latency*)
- Uses Protocol Buffers (protobuf) for efficient serialization
- Supports multi-language clients & servers
- Enables streaming, authentication, and load balancing

Term	Description
RPC	Remote Procedure Call – function call between systems over a network.
Protocol Buffers	Efficient binary format to define service contracts.
Stub	Client-side code auto-generated from .proto file that talks to the server.
IDL	Interface Definition Language – .proto file in gRPC.

gRPC: High-Performance RPC Framework

How gRPC Works?

1. We define **service** and **messages** in a .proto file.
2. Use the protoc compiler to generate client and server code.
3. Server implements the service.
4. Client uses the stub to call the server method like a local function.

Types of gRPC Calls

Type	Description
Unary RPC	Single request and response (as above)
Server streaming	Server sends a stream of responses
Client streaming	Client sends a stream of requests
Bidirectional streaming	Both send streams concurrently

Why Use gRPC?

- Fast, due to HTTP/2 and Protobuf
- Full duplex streaming
- Built-in authentication
- Great for microservices

gRPC: High-Performance RPC Framework

HTTP vs gRPC Comparison

Feature	HTTP REST	gRPC
Transport	HTTP/1.1	HTTP/2
Message Format	JSON	Protocol Buffers
Speed	Slower	Faster
Streaming	Hard to implement	Native support
Language Support	Manual SDKs	Auto-generated for many

gRPC: High-Performance RPC Framework

Demo

Setup:

1. npm init -y
2. npm install @grpc/grpc-js @grpc/proto-loader
3. create greet.proto (Contents are in file)
4. create greet_server.js (Contents are in file)
5. create greet_client.js (Contents are in file)
6. run
    - 6.1. node greet_server.js
    - 6.2. node greet_client.js

Security in Network Programming

TLS/SSL: Introduction to secure socket programming

- TLS(Transport Layer Security) and its predecessor SSL(Secure Sockets Layer) are cryptographic protocols that secure communication over a network, ensuring:
  - Confidentiality (data is encrypted)
  - Integrity (data is not tampered with)
  - Authentication (server or client identity is verified)
  - TLS is used in HTTPS, FTPS, SMTPS, and secure socket programming

Demo:
Generate Certificate: openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout key.pem

TLS vs SSL

Feature	SSL (Deprecated)	TLS (Modern)
Versions	SSLv2, SSLv3	TLS 1.0–1.3
Security	Weak	Strong
Current Use	❌ No longer safe	✅ TLS 1.2 or 1.3

Why Use TLS in Socket Programming?

- Prevents eavesdropping(spy),MITM attacks(On-path Attack),data forgery(falsification).
- Essential for financial apps, login systems, APIs.

After certificate:
secure_server.py
secure_client.py

Software Defined Network (SDN)

Software Defined Network (SDN)

Overview of SDN Concepts

- Software-Defined Networking (SDN) is a network architecture approach that separates:
  - Control Plane (decision-making logic)
  - Data Plane (packet forwarding)
- Traditional networks combine both. SDN decouples them for centralized control, programmability, and agility.

Key SDN Concepts:

Component	Description
Controller	Centralized "brain" of the network, makes decisions
Data Plane	Switches and routers that follow instructions from controller
Southbound API	Communication between controller and devices (e.g., OpenFlow)
Northbound API	Apps communicating with controller (e.g., for monitoring, policy enforcement)

Software Defined Network (SDN)

Benefits of SDN

- Centralized Control : The SDN controller manages the entire network from one place, simplifying decision-making and visibility.
- Dynamic Configuration: Network behavior (e.g., routing, policies) can be changed on the fly via software, without manual device reconfiguration.
- Network Automation: Common tasks like provisioning, traffic routing, and security enforcement can be scripted and automated.
- Easier Experimentation: Developers can test new protocols or policies in real-time without changing physical hardware.
- Scalability & Flexibility: Networks can grow or adapt quickly by simply updating the controller logic, not the underlying hardware.

Introduction to OpenFlow & SDN Controllers

- OpenFlow is a standard protocol that allows **SDN** controllers to instruct switches how to handle network packets.
- It defines a flow table on each switch.

Flow Table Entry Example

Match Fields	Actions	Counters
IP Src = A	Forward Port 2	50 packets

How It Works:

1. Packet enters a switch.
2. If it matches a rule in the flow table → apply action (forward, drop, modify).
3. If no match → switch contacts the controller for instructions.

Popular SDN Controllers

Controller	Language	Notes
Ryu	Python	Great for beginners
ONOS	Java	Carrier-grade, scalable
OpenDaylight	Java	Modular, enterprise-ready
Floodlight	Java	Simple, good for labs

Introduction to OpenFlow & SDN Controllers

Demo (try yourself in ubuntu)-my machine doesn't support natively

1. Install Mininet

```
sudo apt update
sudo apt install mininet -y
```

2. Install Ryu

```
sudo apt install python3-pip -y
pip3 install ryu
```

3. Start Ryu Controller (local)

```
ryu-manager ryu.app.simple_switch_13
```

4. Launch Mininet with Remote Controller (localhost)

```
sudo mn --controller=remote,ip=127.0.0.1 --topo=tree,depth=2
```

5. Test Connectivity

```
mininet> pingall
```

Introduction to P4 and Frenetic Programming

P4 Language: Programming Protocol-Independent Packet Processors

- P4 = Programming Protocol-Independent Packet Processors
- Used to program the data plane, not just control logic.
- Focuses on defining how packets are parsed, matched, and processed.
- Can be used with software switches (e.g., BMv2) and programmable hardware (Tofino).

P4 Program Structure:

```
parser MyParser {
  extract(ethernet);
  extract(ipv4);
}
control MyIngress {
  if (ipv4.dst == 10.0.0.1) {
    forward to port 1;
  }
}
```

Frenetic Language:

- Functional programming language for writing SDN applications.
- Higher abstraction than OpenFlow.
- Developed by Cornell/Princeton researchers.

```
# Frenetic-style logic (Python-like pseudocode)
on_packet(packet):
   if packet.dst_ip == "10.0.0.1":
      forward(packet, port=1)
```

P4 Language: Programming Protocol-Independent Packet Processors

Summary Comparison

Technology	Role	Language	Use Case
OpenFlow	Southbound protocol	N/A	Switch-controller link
Ryu	SDN Controller	Python	Manage flow rules
P4	Data Plane language	P4	Define packet processing
Frenetic	High-level SDN lang	OCaml/Py	Write network policies

End of Chapter 6

(End of syllabus of Network Programming)