Linux Networking Exercises

Easy Exercises (1-2 hours each)

Network Configuration Basics

1. **interface_info** - Write a bash script that uses `ip` command to list all network interfaces, their IP addresses, MAC addresses, and status (up/down). Format output as a readable table.
2. **route_display** - Create a script that displays routing table in human-readable format, highlighting default gateway and interface for specific destination IP.
3. **dns_resolver** - Write a script that queries DNS for A, AAAA, MX, and TXT records of a given domain using `dig` or `host`. Display results in organized format.
4. **netmask_calculator** - Implement a script that takes CIDR notation (e.g., 192.168.1.0/24) and calculates network address, broadcast address, first/last usable IP, and total hosts.
5. **arp_cache_viewer** - Create a script that displays ARP cache with hostnames resolved (using reverse DNS) and time since last update.

Socket Programming - Basic

6. **tcp_echo_server** - Write a simple TCP echo server in C that listens on port 8888, accepts connections, echoes back received data. Test with `telnet` or `nc`.
7. **udp_echo_server** - Implement UDP echo server in C on port 9999. Test with `nc -u`.
8. **tcp_client** - Write TCP client in C that connects to a server, sends message, receives response, and displays it.
9. **port_scanner_basic** - Create a simple port scanner script (bash + nc or Python) that tests if common ports (22, 80, 443, 3306) are open on localhost.

Packet Analysis - Basic

10. **pcap_reader** - Write a script that uses `tcpdump` to capture 100 packets, save to file, and display summary (protocols, source/dest IPs, packet counts).
11. **http_sniffer** - Use `tcpdump` or `tshark` to capture HTTP requests on local interface. Extract and display URLs being accessed.
12. **packet_counter** - Create a script that monitors interface and counts packets by protocol (TCP, UDP, ICMP, ARP) in real-time for 60 seconds.

Network Tools Usage

13. **ping_sweep** - Write a script that pings all addresses in 192.168.1.0/24 subnet and reports which hosts are alive.
14. **traceroute_analyzer** - Create a script that runs traceroute to multiple destinations and identifies common hops/bottlenecks.
15. **bandwidth_monitor** - Use `ifstat`, `iftop`, or parse `/proc/net/dev` to create a simple bandwidth monitoring script showing TX/RX rates.

Network Services

16. **http_server_minimal** - Write minimal HTTP server in Python or C that serves static HTML file on port 8080.
17. **dns_query_tool** - Implement simple DNS query tool in C using raw UDP sockets that queries Google DNS (8.8.8.8) for A record.
18. **whois_parser** - Create a script that performs whois lookup and extracts registrar, creation date, and expiration date.

Netfilter/iptables Basics

19. **firewall_status** - Write a script that displays current iptables rules in organized format by chain and policy.
20. **connection_tracker** - Create a script that monitors `/proc/net/nf_conntrack` and displays active connections grouped by state.

Unix Sockets

21. **unix_socket_ipc** - Create client and server programs communicating via Unix domain socket. Server performs simple calculation requested by client.
22. **socket_pair_demo** - Implement program using `socketpair()` for bidirectional IPC between parent and child processes.

Network Debugging

23. **mtu_finder** - Write a script that determines MTU to remote host using ping with different packet sizes and DF flag.
24. **latency_tester** - Create a tool that measures round-trip latency to host using ICMP echo requests. Display min/max/avg/jitter.
25. **socket_stats** - Parse `/proc/net/tcp` and `/proc/net/udp` to show listening ports and established connections with process names (using `/proc/[pid]/fd/`).

Medium Exercises (3-5 hours each)

Advanced Socket Programming

1. **multi_client_server** - Implement TCP server in C handling multiple clients simultaneously using `select()` or `poll()`. Server broadcasts messages from any client to all connected clients (simple chat server).
2. **http_proxy** - Create a simple HTTP proxy server in C or Python that forwards requests to destination servers and returns responses. Support basic HTTP/1.0.
3. **udp_reliable** - Implement reliable UDP transfer with acknowledgments, retransmission, and sequence numbers. Transfer a file between client and server.

Packet Crafting

4. **raw_socket_ping** - Implement ping utility using raw sockets (SOCK_RAW). Craft ICMP echo request packets manually, send them, and parse replies. Requires root/CAP_NET_RAW.
5. **tcp_syn_scanner** - Write SYN scanner using raw sockets that sends SYN packets and analyzes responses (SYN-ACK = open, RST = closed) to scan ports. Much faster than full connect scan.
6. **arp_request** - Craft and send ARP request using raw sockets. Parse ARP replies to build IP-to-MAC mapping for local subnet.

Network Protocol Implementation

7. **tftp_client** - Implement TFTP (Trivial File Transfer Protocol) client supporting read requests (RRQ). Test against standard tftpd server.
8. **dhcp_discover** - Create program that sends DHCP DISCOVER packet and parses DHCP OFFER responses to discover DHCP servers on network.
9. **ntp_client** - Implement simple NTP (Network Time Protocol) client that queries NTP server and displays offset between local and server time.

Traffic Analysis

10. **protocol_analyzer** - Write a tool using libpcap that captures packets and generates statistics: top talkers (IP pairs by bytes), protocol distribution, packet size distribution.
11. **tcp_stream_reassembly** - Parse pcap file and reassemble TCP streams. Extract and save complete HTTP responses or file transfers.
12. **dns_traffic_analyzer** - Analyze DNS queries in pcap file. Identify most queried domains, query types, response codes, DNS servers used.

Network Performance

13. **bandwidth_tester** - Implement iperf-like tool with TCP client/server that measures throughput by sending data stream for configurable duration and reports bandwidth.
14. **concurrent_connections** - Write a tool that opens N concurrent TCP connections to a server and measures how connection establishment time scales with N.

Advanced Firewall/NAT

15. **connection_logger** - Use libnetfilter_conntrack to monitor new connections in real-time and log them with timestamps, protocols, addresses.

Hard Exercises (Multiple evenings)

Advanced Protocol Implementation

1. **tcp_stack_userspace** - Implement basic TCP state machine in userspace using raw IP sockets. Support connection establishment (3-way handshake), data transfer, and graceful close (FIN). No need for retransmission/flow control, just basic state management.
2. **socks5_proxy** - Implement complete SOCKS5 proxy server supporting TCP connections, authentication, and both CONNECT and BIND commands. Test with curl and browsers.
3. **vpn_tunnel_simple** - Create simple VPN using TUN/TAP interfaces. Program reads packets from TUN interface, encrypts them (simple XOR or AES), sends via UDP to remote endpoint, which decrypts and injects to its TUN interface.

Traffic Manipulation

4. **packet_injector** - Using raw sockets and libpcap, create a tool that sniffs packets matching filter, modifies them (e.g., change HTTP headers, redirect DNS responses), and reinjects modified packets. Requires understanding of checksums and sequence numbers.
5. **mitm_arp_poison** - Implement ARP poisoning tool that redirects traffic between two hosts through your machine. Capture and display forwarded traffic. Include cleanup to restore normal operation. **Test only on isolated network you control.**
6. **tcp_connection_hijack** - Demonstrate TCP sequence number prediction and connection hijacking on local network. Inject data into existing TCP stream. **Educational only, isolated test environment.**

Load Balancing / High Availability

7. **layer4_load_balancer** - Implement TCP load balancer that accepts connections and distributes them across backend servers using round-robin or least-connections algorithm. Handle backend failures gracefully.
8. **health_check_system** - Create system that monitors multiple services (HTTP, TCP ports) on multiple hosts, detects failures, and updates routing/load balancing configuration automatically.

Network Namespace & Virtualization

9. **network_namespace_lab** - Write a tool that creates isolated network namespaces, sets up virtual ethernet pairs (veth), configures routing/NAT between namespaces and host. Create a mini virtual network with 3+ isolated "hosts".
10. **container_network_basic** - Implement basic container networking setup: create network namespace for container, set up veth pair, configure bridge on host, set up NAT for internet access. Similar to Docker's default bridge network.

Testing Environment Setup

Safe Testing Infrastructure

Isolated Network Options:

1. **Virtual machines** with isolated network (QEMU/VirtualBox internal network)
2. **Network namespaces** for isolation on single machine
3. **Docker containers** with custom networks
4. **Separate physical VLAN** (if available)

Recommended Lab Setup:

```
# Create isolated network namespace
ip netns add testns1
ip netns add testns2

# Create veth pairs connecting namespaces
ip link add veth0 type veth peer name veth1
ip link set veth1 netns testns1

# Configure addresses
ip netns exec testns1 ip addr add 10.0.0.1/24 dev veth1
ip netns exec testns1 ip link set veth1 up
```

Tools You'll Need

Network Configuration:

- `ip`, `ifconfig` (interface management)
- `route`, `ip route` (routing)
- `brctl`, `bridge` (bridging)
- `iptables`, `nftables` (firewalling)

Analysis Tools:

- `tcpdump`, `tshark`, `wireshark` (packet capture)
- `netstat`, `ss` (socket statistics)

- `lsof` (open files/sockets)
- `nmap` (network scanning)
- `iperf3` (bandwidth testing - for comparison)

Development:

- `gcc`, `make` (C compilation)
- `python3` (scripting)
- `libpcap-dev`, `libnetfilter-conntrack-dev` (development libraries)
- `strace`, `ltrace` (system call tracing)
- `gdb`, `valgrind` (debugging)

Network Utilities:

- `nc` (netcat)
- `socat` (advanced relay)
- `curl`, `wget` (HTTP clients)
- `dig`, `nslookup`, `host` (DNS tools)
- `ping`, `traceroute`, `mtr`

Exercise Validation

Easy exercises:

- Script produces correct output for given inputs
- Server accepts connections and responds correctly
- Captured data shows expected protocol information

Medium exercises:

- Client-server communication works as specified
- Protocol implementation interoperates with standard tools
- Performance measurements are reasonable
- Packet crafting produces valid packets (verify with Wireshark)

Hard exercises:

- Complex protocols handle multiple states correctly
- Load balancer distributes connections evenly
- Network namespace isolation works properly
- VPN tunnel successfully routes traffic

Safety Guidelines

DO:

- Test on localhost (127.0.0.1) or local network you control
- Use network namespaces for isolation
- Use VMs for potentially disruptive experiments
- Document what your code does
- Include cleanup code to restore system state

DON'T:

- Scan networks you don't own
- Intercept traffic on production networks
- Test ARP poisoning on shared networks
- Flood networks with traffic
- Use exploits against systems without permission

Requires Root/Capabilities:

- Raw socket operations: `CAP_NET_RAW`
- Network configuration: `CAP_NET_ADMIN`
- Packet capture: `CAP_NET_RAW` or setcap on tcpdump

Grant capabilities instead of full root:

```
# Allow binary to use raw sockets without root
sudo setcap cap_net_raw+ep ./my_program
```

Testing Progression

Phase 1 (Easy):

- Learn standard tools (ip, tcpdump, ss)
- Basic socket programming
- Reading existing traffic

Phase 2 (Medium):

- Multi-client servers
- Protocol implementation
- Packet crafting with raw sockets
- Traffic analysis with libpcap

Phase 3 (Hard):

- Stateful protocols (TCP)
- Traffic manipulation
- Network virtualization
- Complex distributed systems

Useful Test Services

Run locally for testing:

```
# Simple HTTP server
python3 -m http.server 8000

# Echo server
nc -l -p 7777 -k -c 'xargs -n1 echo'

# TFTP server
sudo in.tftpd -l -s /srv/tftp

# DNS server (dnsmasq)
sudo dnsmasq --no-daemon --log-queries
```

Sample Test Scenarios

For servers:

1. Single client connection
2. Multiple concurrent clients
3. Rapid connect/disconnect
4. Large data transfer
5. Client timeout/abnormal disconnect

For packet crafting:

1. Verify with Wireshark that packet structure is correct
2. Test with standard tools (does Linux respond to your packets?)
3. Verify checksums are correct

For traffic analysis:

1. Generate known traffic with curl/wget
2. Capture with your tool
3. Compare with tcpdump output
4. Test with real-world pcap files (available online)

Performance Benchmarking

Compare your implementations with standard tools:

- Your HTTP server vs nginx (requests/sec)
- Your port scanner vs nmap (time to scan)
- Your bandwidth tester vs iperf3
- Your packet capture vs tcpdump (packets/sec, CPU usage)

Learning Resources

Protocol RFCs (for implementation exercises):

- RFC 792 (ICMP)
- RFC 793 (TCP)
- RFC 768 (UDP)
- RFC 826 (ARP)
- RFC 1350 (TFTP)
- RFC 2131 (DHCP)
- RFC 1928 (SOCKS5)

Test Data:

- Wireshark sample captures (wiki.wireshark.org/SampleCaptures)
- PCAP files from CTF challenges
- Generate your own with curl, wget, nc

This exercise list covers essential Linux networking skills from basic configuration and monitoring through advanced protocol implementation and network virtualization, all safely testable on your Debian system.