Project Valiant: Design Document for an Educational TCP/IP Stack

Overview

This document outlines the design for 'Project Valiant,' a user-space implementation of a TCP/IP network stack. The key architectural challenge is building a reliable, layered data communication system with intricate state management and protocol logic, while providing an educational scaffold for developers to understand core networking principles from the ground up.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

1. Context and Problem Statement

> **Milestone(s):** All (Foundational Context)

The TCP/IP protocol suite is the fundamental communication framework of the Internet, yet its inner workings remain a "black box" for most developers. We interact with it daily through web browsers, APIs, and cloud services, but the intricate dance of packets, handshakes, and state machines happens deep within the operating system kernel, hidden from view. **Project Valiant** is an educational journey to open that box. By constructing a functional, user-space TCP/IP network stack from the ground up, developers transform from passive users of networking abstractions into architects who understand the principles, trade-offs, and clever engineering that make reliable communication over an unreliable medium possible.

The core value proposition is **deep, experiential learning**. Reading specifications and diagrams provides one level of understanding; the act of implementing those specifications, debugging why a packet is dropped, or tracing why a connection hangs provides a profoundly deeper, lasting comprehension. This project is not about building a production-grade competitor to the Linux kernel's networking stack. It is a pedagogical tool—a meticulously crafted simulation that isolates and illuminates the key protocols (Ethernet, ARP, IP, ICMP, TCP) and their interactions, providing a controlled environment to experiment with and internalize networking fundamentals.

Analogy: The Postal Service of Computing

To build intuition before diving into technical specifications, consider the TCP/IP stack as a **highly organized, international postal service**. Sending data across a network is analogous to sending a physical letter from your home in one city to a friend's home in another.

1. **Application Layer (The Letter's Content):** You write your message on a piece of paper. This is the application data—the email body, the HTTP request, the file bytes. You don't worry about how it gets there; you just care about the content.
2. **Transport Layer (The Envelope with Tracking & Delivery Confirmation - TCP):** You place your letter in an envelope. The Transport Layer (TCP) adds crucial delivery-control information. It writes a **sequence number** on the envelope (like "Page 1 of 3") so the recipient can reassemble pages in order if they arrive out of sequence. It also requests a **return receipt** (an acknowledgment, or ACK) to confirm delivery. If the receipt isn't received in time, TCP will re-send the letter. This layer establishes a reliable, **connection-oriented** channel, like registered mail.
3. **Network Layer (The Postal Address and Routing - IP):** You address the envelope with the destination's **IP Address** (e.g., `192.168.1.10`) and your return IP address. The postal service (the Network Layer) doesn't deliver directly to the doorstep. It looks at the destination city/zip code (the network portion of the IP), consults its **routing tables** (like mail sorting facilities), and decides the next hop—maybe to a local post office, then a regional hub, then another city's hub. Each hop decrements the **Time-To-Live (TTL)** stamp, and if it reaches zero, the letter is discarded to prevent infinite loops. This is **best-effort, connectionless** delivery.
4. **Link Layer (The Local Mail Carrier and Street Address - Ethernet & ARP):** The envelope finally arrives at the destination city's post office. The mail carrier now needs the final **street address (MAC Address)** to deliver it to the specific house. If they don't know which house corresponds to the IP address, they shout out (broadcasts) to the whole neighborhood, "Who has IP address `192.168.1.10`?" This is the **Address Resolution Protocol (ARP)**. The correct house replies with its MAC address (`aa:bb:cc:dd:ee:ff`). The carrier then writes this physical address on the envelope and delivers it to the exact network interface (the mailbox).
5. **Physical Layer (The Roads and Trucks):** The physical letter is carried by trucks, planes, or mail carriers—the electrical signals on a wire, light pulses in a fiber, or radio waves for Wi-Fi.

This layered model is powerful because each layer has a single, clear responsibility and communicates only with the layers directly above and below it. The letter writer doesn't need to know about interstate highways, and the mail carrier doesn't need to understand the letter's content. Similarly, your web browser doesn't need to understand Ethernet frames, and your network card doesn't parse HTTP headers. This **separation of concerns** is the architectural bedrock of the Internet.

The Core Technical Challenge

Building a TCP/IP stack is the challenge of constructing a **reliable, ordered, stream-based communication channel over an inherently unreliable, packet-switched, best-effort network**. The problem space is defined by a series of adversarial conditions that the stack must systematically overcome:

Adversarial Condition	Consequence	Layer(s) Responsible for Mitigation
Unreliable Medium (e.g., Wi-Fi interference, cable unplugged)	Packets can be lost, corrupted, or arrive out of order.	Transport (TCP): Sequence numbers, acknowledgments, retransmission timers, checksums.
Limited Bandwidth & Contested Resources	Sending too fast can overwhelm network links or the receiver, causing congestion collapse.	Transport (TCP): Flow control (receiver window) and congestion control (congestion window, slow start).
Asynchronous, Concurrent Communication	Multiple applications on the same host need to communicate simultaneously.	Transport (TCP): Port numbers to demultiplex connections; state machines to manage each connection's lifecycle independently.
Dynamic, Decentralized Routing	There is no central map of the network; paths can change mid-connection.	Network (IP): Stateless, destination-based forwarding; routing tables updated by external protocols; ICMP for error reporting.
Dual Addressing Scheme	Applications think in terms of IP addresses, but hardware delivers frames to MAC addresses.	Link (ARP): A distributed, cache-backed protocol to map IP addresses to physical MAC addresses.
Complex, Stateful Interactions	A single logical connection (e.g., a web session) involves dozens of packets with precise timing and state dependencies.	Transport (TCP): A detailed state machine (11+ states) governing connection setup, data transfer, and teardown.

> The critical insight is that **reliability is not a property of the network, but a service built on top of it.** The network layer (IP) provides only "best-effort" datagram delivery. It is the transport layer's (TCP) responsibility to synthesize reliability from this unreliability using algorithms for sequencing, acknowledgment, retransmission, and flow control.

The architectural challenge for the implementer is to correctly orchestrate these layered mitigations within a single system. This involves:

1. **Implementing Protocol Logic:** Translating RFC specifications (notably RFC 793 for TCP) into precise code for parsing and constructing binary packet headers.
2. **Managing Complex State:** Maintaining dozens of variables per TCP connection—sequence numbers, window sizes, timers, and state—and ensuring correct transitions in response to internal events (application `connect`) and external events (incoming `SYN` packet).
3. **Handling Concurrency and Asynchrony:** Processing incoming packets from the network interface, application write requests, and timer expirations, all potentially simultaneously, without corrupting shared state.
4. **Designing Efficient Data Structures:** Creating buffers, caches (ARP), and tables (routing) that support fast lookup and insertion while respecting memory limits.
5. **Validating and Securing the System:** Defensively parsing untrusted network data, verifying checksums, handling malformed packets gracefully, and avoiding common vulnerabilities like ARP cache poisoning.

The ultimate measure of success for Project Valiant is not raw performance, but **correctness and clarity**—a stack that reliably passes data between two endpoints while serving as a transparent, understandable map of Internet plumbing.

Existing Approaches

Understanding the landscape of network stack implementations clarifies the unique position of Project Valiant. There are two primary categories: production-grade kernel stacks and educational user-space stacks.

Production Kernel Stacks (e.g., Linux, FreeBSD, Windows)

- **Location:** Deep within the operating system kernel.
- **Primary Goal: Maximum performance, security, and feature-completeness.** They are optimized with zero-copy buffers, interrupt coalescing, sophisticated queueing disciplines (QDisc), and hardware offloading.
- **Complexity:** Immense. The Linux networking subsystem comprises millions of lines of code, supporting dozens of protocols, virtual interfaces, firewalling (Netfilter), and quality-of-service mechanisms. This complexity is a barrier to understanding core concepts.
- **Accessibility:** Difficult to modify or experiment with. Developing within the kernel requires specialized knowledge and carries risk of system instability.

Educational User-Space Stacks (Project Valiant's Category)

- **Location:** Runs as an ordinary user process.
- **Primary Goal: Clarity, simplicity, and pedagogical value.** Performance is secondary. The focus is on implementing the core RFC-mandated behaviors in a readable, well-structured manner.
- **Complexity:** Deliberately limited. It implements only IPv4, TCP, ICMP, and basic ARP. It ignores fragmentation, TCP options (like SACK), and advanced congestion algorithms (like CUBIC) in the core milestones.
- **Accessibility:** Easy to run, debug, and modify. Developers can use standard tools like `gdb`, `valgrind`, and `printf` debugging. The stack interacts with the real network through a virtual TAP device, creating a safe, isolated sandbox.

Approach	Pros	Cons	Best For
Production Kernel Stack	Blazing fast, secure, full-featured, robust.	Opaque, extremely complex, dangerous to modify, hard to debug.	Building the actual Internet.
Educational User-Space Stack	Transparent, modular, safe to crash, excellent for learning and experimentation.	Slow, incomplete, not suitable for real-world load.	**Understanding how the Internet works.**

Project Valiant is inspired by and aligns with the tradition of exemplary educational stacks like those detailed in "Let's code a TCP/IP stack" and the book "TCP/IP Illustrated, Volume 2: The Implementation". It makes a key architectural choice (detailed in Section 5) to use a **TAP device**, which provides a user-space process with a virtual network interface, allowing it to send and receive raw Ethernet frames as if it were a kernel driver. This choice perfectly balances realism (real Ethernet frames) with safety and simplicity (no kernel modules required).

> The design philosophy is **"First, make it correct. Then, make it clear. Then, and only then, consider making it fast."** This project prioritizes the first two, ensuring the implementation is a faithful and comprehensible reference model.

Implementation Guidance

While the primary focus of this section is conceptual, establishing a solid development foundation is critical. The following guidance will set up your environment for success across all milestones.

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option (For Later)
Development Language	**C (ISO C99)**. Offers direct memory access, essential for manipulating packet headers, and provides the foundational experience upon which many systems are built.	Rust (for memory safety) or Go (for concurrency primitives).
Link Layer Interface	**Linux TAP device**. Creates a virtual network interface; the kernel routes packets to/from your userspace process. Simple and realistic.	Raw sockets (`AF_PACKET` , `SOCK_RAW`). More complex, provides direct access to the physical interface.
Build System	**Make**. Simple, ubiquitous, and sufficient for a project of this size.	CMake or Meson for better cross-platform support.
Debugging & Inspection	**Wireshark/tcpdump** on the TAP interface (`tap0`). `printf` **logging** with hex dumps.	Integrated logging with ring buffers and log levels; integration with `gdb` scripts.
Testing	**Manual testing** with `ping` , `arping` , and `netcat` . **Unit tests** for parsers and state machines using a test harness.	Automated integration tests using a network namespace to create isolated virtual networks.

B. Recommended File/Module Structure

Establish this directory structure from the start. It mirrors the layered architecture and keeps code organized.

```
project_valiant/
├── Makefile                    # Build definitions
├── README.md
├── include/                    # Public header files
│   ├── stack/
│   │   ├── netif.h            # Network interface (TAP) abstraction
│   │   ├── ethernet.h         # Ethernet frame definitions & functions
│   │   ├── arp.h              # ARP packet definitions & cache
│   │   ├── ipv4.h             # IPv4 packet definitions & routing
│   │   ├── icmp.h             # ICMP packet definitions
│   │   └── tcp.h              # TCP segment definitions & state machine
│   └── utilities/
│       ├── checksum.h         # Internet checksum utilities
│       ├── timer.h            # Timer wheel for retransmissions
│       └── buffer.h           // Generic buffer management
├── src/                        # Implementation source files
│   ├── main.c                 // Application entry point, event loop
│   ├── netif/
│   │   └── tap.c              // TAP device setup, read/write frames
│   ├── link/
│   │   ├── ethernet.c         // Ethernet frame parsing/construction
│   │   └── arp.c              // ARP protocol implementation & cache
│   ├── network/
│   │   ├── ipv4.c             // IPv4 packet processing, routing
│   │   └── icmp.c             // ICMP protocol (echo reply)
│   ├── transport/
│   │   ├── tcp.c              // TCP core: input, state machine, output
│   │   ├── tcp_timer.c        // Retransmission timer management
│   │   ├── tcp_sendbuf.c      // Send buffer & sliding window
│   │   └── tcp_recvbuf.c      // Receive buffer & reassembly
│   └── utilities/
│       ├── checksum.c
│       ├── timer.c
│       └── buffer.c
└── tests/                      // Unit and integration tests
    ├── test_ethernet.c
    ├── test_arp.c
    └── harness.c
```

C. Infrastructure Starter Code: TAP Device Setup

The TAP device is the "hardware" for your stack. This is a prerequisite component that you can copy and use without modification initially. It creates a virtual network interface and provides functions to read and write Ethernet frames.

File: `src/netif/tap.c`

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <errno.h>

#include "stack/netif.h"

/**
 * Opens a TAP device with the given name.
 * @param dev The name of the device (e.g., "tap0").
 * @return File descriptor for the TAP device, or -1 on error.
 */
int tap_open(char *dev) {
    struct ifreq ifr;
    int fd;

    // Open the TUN/TAP device clone file
    if ((fd = open("/dev/net/tun", O_RDWR)) < 0) {
        perror("open /dev/net/tun");
        return -1;
    }

    memset(&ifr, 0, sizeof(ifr));

    // Set flags: IFF_TAP (layer 2) + IFF_NO_PI (no packet info prefix)
    ifr.ifr_flags = IFF_TAP | IFF_NO_PI;

    // Copy the device name if provided
    if (dev) {
        strncpy(ifr.ifr_name, dev, IFNAMSIZ);
    }

    // Create the TAP device
    if (ioctl(fd, TUNSETIFF, (void *)&ifr) < 0) {
        perror("ioctl TUNSETIFF");
        close(fd);
        return -1;
    }
```

```c
    // Return the file descriptor
    return fd;
}

/**
 * Reads an Ethernet frame from the TAP device.
 * @param fd The TAP file descriptor.
 * @param buf Buffer to store the frame.
 * @param len Size of the buffer.
 * @return Number of bytes read, or -1 on error.
 */
int tap_read(int fd, unsigned char *buf, int len) {
    return read(fd, buf, len);
}

/**
 * Writes an Ethernet frame to the TAP device.
 * @param fd The TAP file descriptor.
 * @param buf Buffer containing the frame.
 * @param len Length of the frame.
 * @return Number of bytes written, or -1 on error.
 */
int tap_write(int fd, unsigned char *buf, int len) {
    return write(fd, buf, len);
}
```

File: `include/stack/netif.h`

```c
#ifndef NETIF_H
#define NETIF_H

int tap_open(char *dev);
int tap_read(int fd, unsigned char *buf, int len);
int tap_write(int fd, unsigned char *buf, int len);

#endif // NETIF_H
```

D. Core Logic Skeleton: Main Event Loop

The main event loop is the heart of the user-space stack. It uses a simple `poll()` or `select()` to wait for activity on the TAP device (incoming packets) and potentially other file descriptors (like timers or application sockets in the future).

File: `src/main.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <poll.h>

#include "stack/netif.h"

#define MAX_FRAME_SIZE 1600 // MTU + Ethernet header

int main(int argc, char *argv[]) {
    int tap_fd;
    unsigned char frame_buffer[MAX_FRAME_SIZE];
    int frame_len;

    // TODO 1: Parse command line arguments (e.g., interface name, IP address)
    // Hint: Use getopt().

    // TODO 2: Open the TAP device (e.g., "tap0")
    tap_fd = tap_open("tap0");
    if (tap_fd < 0) {
        fprintf(stderr, "Failed to open TAP device.\n");
        exit(1);
    }

    printf("Project Valiant stack started on TAP device. FD: %d\n", tap_fd);

    // TODO 3: Initialize stack components (ARP cache, routing table, TCP state table)
    // This will be filled in during Milestones 1-4.

    // Main event loop
    while (1) {
        struct pollfd fds[1];
        fds[0].fd = tap_fd;
        fds[0].events = POLLIN;

        // Wait for a frame to arrive (blocking)
        int ret = poll(fds, 1, -1); // No timeout, wait forever
        if (ret < 0) {
            perror("poll");
            break;
        }

        if (fds[0].revents & POLLIN) {
            // Frame received from TAP
            frame_len = tap_read(tap_fd, frame_buffer, sizeof(frame_buffer));
```

```
            if (frame_len < 0) {
                perror("tap_read");
                break;
            }

            // TODO 4: Process the incoming Ethernet frame
            // This is the top of the receive path. In Milestone 1, you will
            // call ethernet_handle_frame() here.
            printf("Received frame of length %d bytes.\n", frame_len);
            // TODO: Print hex dump of frame for debugging (remove later).
        }

        // TODO 5: Handle expired timers (e.g., ARP cache timeouts, TCP retransmissions)
        // This will be added in Milestones 1 and 4.
    }

    // TODO 6: Cleanup on exit
    close(tap_fd);
    return 0;
}
```

E. Language-Specific Hints (C)

- **Network Byte Order:** Always use `ntohs()`, `ntohl()`, `htons()`, `htonl()` when reading or writing multi-byte integer fields in packet headers. The network byte order is **big-endian**. Your host CPU is likely **little-endian**.
- **Memory Alignment:** When defining `struct`s for packet headers, be aware that the compiler may insert padding. Use `__attribute__((packed))` (GCC/Clang) or `#pragma pack(1)` to ensure the struct layout matches the on-wire format exactly.
- **Zero-Copy Ideal:** Aim to pass pointers to offsets within the packet buffer rather than copying payloads excessively. For example, an IP packet's payload is just a pointer to `ip_header + ip_header_length`.
- **Error Handling:** In networking, the correct response to many errors is to **silently drop the packet**. Log for debugging, but don't crash. Use `perror()` and `fprintf(stderr, ...)` liberally during development.

F. Milestone Checkpoint: Environment Verification

Before writing any protocol code, verify your TAP setup works.

1. **Compile and Run:**

```
# Compile the starter code
gcc -o valiant src/main.c src/netif/tap.c -I./include

# Run (requires root/capabilities for TAP)
sudo ./valiant
```

BASH

`Expected output: `Project Valiant stack started on TAP device. FD: 3`

2. **In a separate terminal, configure the TAP interface:**

```bash
# Find the TAP interface name (likely tap0)
ip link show
# Bring it up and assign an IP
sudo ip addr add 10.0.0.1/24 dev tap0
sudo ip link set tap0 up
```

3. **Test Basic Connectivity (from another terminal):**

```bash
# Ping the stack's IP (you won't get a reply until Milestone 2)
ping 10.0.0.1
```

```
*   **Expected at this stage:** Ping will hang or show "Destination Host Unreachable." **This is normal.** Check your stack's consc
*   **If you see no output:** Verify TAP device permissions. You might need to run the stack with `sudo` or set the `CAP_NET_ADMIN`
```

2. Goals and Non-Goals

> **Milestone(s):** All (Foundational Scope Definition)

This section defines the boundaries of Project Valiant. Building a full-featured, production-grade network stack is a monumental task spanning years of development. To make this educational project achievable and focused on core learning outcomes, we must explicitly define what is included (**goals**) and what is intentionally excluded (**non-goals**). This clarity ensures developers concentrate on understanding fundamental networking principles without being overwhelmed by edge cases, performance optimizations, or security features that belong in commercial implementations.

Goals (Must-Haves)

The primary goal of Project Valiant is to build a **minimal, correct, and comprehensible** user-space TCP/IP stack that demonstrates the core protocols and data flows of internet communication. Success is measured by the developer's understanding of packet processing, stateful connection management, and reliable data transfer—not by performance benchmarks or feature completeness.

The following table outlines the mandatory implementation targets, aligned with the four project milestones. Each goal includes the **protocol or component** to implement, the **key learning outcome** it provides, and the **concrete deliverable** that demonstrates completion.

Goal	Protocol/Component	Key Learning Outcome	Concrete Deliverable (What You Will Build)
Physical & Link Layer Interaction	Ethernet Frame I/O & ARP	Understanding how software interacts with a virtual network interface at the frame level and how logical IP addresses are resolved to physical MAC addresses.	A TAP device driver that sends/receives raw Ethernet frames (`tap_open` , `tap_read` , `tap_write`) and an ARP protocol handler that resolves IPs to MACs and maintains a cache.
Network Layer Routing & Diagnostics	IPv4 & ICMP (Echo)	Understanding packet structure, header checksums, basic routing decisions, and the role of control messages for network diagnostics.	An IPv4 packet parser/builder that validates checksums and forwards packets, a simple routing table, and an ICMP echo responder (ping reply).
Connection-Oriented Transport Setup/Teardown	TCP (State Machine)	Understanding stateful protocol design, the three-way handshake for reliable connection establishment, and the four-way handshake for graceful connection termination.	A TCP segment parser and a connection state machine that correctly transitions through `LISTEN` , `SYN_SENT` , `ESTABLISHED` , and closing states in response to segment flags.
Reliable Data Transfer & Flow Control	TCP (Sliding Window, ACKs, Timers)	Understanding how sequence numbers, acknowledgments, retransmission timers, and window advertisements combine to provide reliable, in-order data delivery over an unreliable medium.	A sliding window send/receive mechanism, a retransmission timer for lost segments, and logic to respect the receiver's advertised window (flow control) and a basic congestion window (congestion control).
Educational Codebase & Documentation	Clean Architecture & Comments	Developing skills in structuring a layered system, writing clear code, and documenting design decisions for future maintainers (including oneself).	A well-organized codebase with separate modules per layer (e.g., `net/` , `tcp/`), comprehensive comments explaining non-obvious logic, and this design document as a guide.
Interactive Testing & Validation	Integration with OS Network Tools	Gaining practical experience in validating network behavior using standard tools like `ping` , `arp` , `tcpdump` , and `netcat` .	The stack can respond to `ping` from another machine, complete a TCP connection with `netcat` , and transfer simple data, observable via packet captures.

> The critical insight for the goals is **pedagogy over performance**. We prioritize clear, readable implementations of RFC-specified behaviors over optimized algorithms. For example, a simple linear search in the routing table is acceptable if it correctly demonstrates the lookup concept.

Non-Goals (Explicitly Out of Scope)

To prevent scope creep and maintain focus on foundational concepts, the following areas are explicitly **not** part of Project Valiant. This does not mean they are unimportant; rather, they represent advanced topics or concerns that would distract from the core learning objectives. Developers are encouraged to explore these as **future extensions** (see Section 13) once the base stack is functional.

Area	Specific Exclusions	Rationale for Exclusion
Performance & Scalability	High-speed packet processing (e.g., zero-copy, DMA, kernel bypass), support for millions of concurrent connections, or low-latency optimizations.	These require deep system-level optimizations that obscure the clarity of protocol logic. Our stack runs in user-space on a TAP device, which is inherently not performance-oriented.
Security Hardening	Protection against spoofing, replay, or denial-of-service attacks (e.g., SYN flood protection), encryption (IPsec, TLS), or comprehensive input sanitization beyond basic validation.	Security is a vast, specialized domain. Adding it would multiply complexity and divert focus from understanding the base protocols. Our stack is intended for isolated, trusted lab environments.
Protocol Completeness	IPv6, IP fragmentation/reassembly, TCP options (SACK, Window Scaling, Timestamps), UDP, IGMP, DNS, DHCP, or routing protocols (OSPF, BGP).	Each additional protocol is a substantial project in itself. We implement the minimal set (IPv4, ICMP Echo, TCP basics) required to establish a functional data channel.
Production Robustness	Adherence to all RFC edge cases, exhaustive error recovery (e.g., from all possible corrupted packet states), or compatibility with all real-world network middleboxes (NATs, firewalls).	While we handle major RFC-specified behaviors, a production stack must handle countless obscure edge cases. We focus on the "happy path" and major error modes (e.g., checksum failure, no route).
Advanced TCP Features	Fast retransmit/recovery, selective acknowledgments (SACK), explicit congestion notification (ECN), keep-alive probes, or the `TIME_WAIT` state handling intricacies.	These are optimizations and refinements built atop the core reliable delivery mechanism. We implement basic Reno-style congestion control (slow start, congestion avoidance) as a representative example.
Kernel Integration	Loading the stack as a kernel module, providing a standard BSD socket API to other applications, or interfacing with physical NIC hardware.	Kernel programming introduces significant complexity and risk (crashes). Our user-space TAP approach is safer and easier to debug, while still demonstrating all protocol layers.
Cross-Platform Support	Running on operating systems other than Linux (which provides TAP devices), or on architectures with different endianness without conditional compilation.	We standardize on Linux/x86_64 (little-endian) to simplify development. The concepts are portable, but the low-level I/O (TAP) is platform-specific.

> **Architectural Decision: Pedagogical Scope Limitation**
>
> - **Context:** The team (a single developer or small learning group) has limited time and wants to understand core networking concepts without being sidetracked.
> - **Options Considered:**
>   1. **Implement a minimal, pedagogical stack** (Chosen).
>   2. **Aim for a production-ready feature set**.
>   3. **Implement only one layer deeply** (e.g., just TCP).
> - **Decision:** Build a minimal, pedagogical stack covering Ethernet, ARP, IP, ICMP, and basic TCP from the link layer up to a simple socket-like interface.
> - **Rationale:** Option 1 provides the most holistic understanding of how layers interact, which is the primary educational objective. Option 2 is infeasible given time constraints. Option 3 would miss the critical learning of inter-layer dependencies (e.g., how TCP triggers ARP resolution).
> - **Consequences:** The resulting stack is not suitable for real-world deployment but perfectly serves as a learning tool and a foundation for future exploration of the excluded areas.

By adhering to these goals and non-goals, you will construct a working TCP/IP stack that demystifies internet communication while keeping the project manageable. The satisfaction of seeing a `ping` reply from your own code or transferring a file via a TCP connection built from scratch is immense and forms a solid foundation for any future work in networking systems.

---

3. High-Level Architecture

> **Milestone(s):** All (Foundational System Structure)

This section presents the architectural blueprint for Project Valiant—the conceptual model of how the software components are organized and how data flows between them. Think of this as the floor plan for building our digital post office, showing how mail moves from the street (the network wire) to the recipient's mailbox (the application).

Component Overview & Layered Model

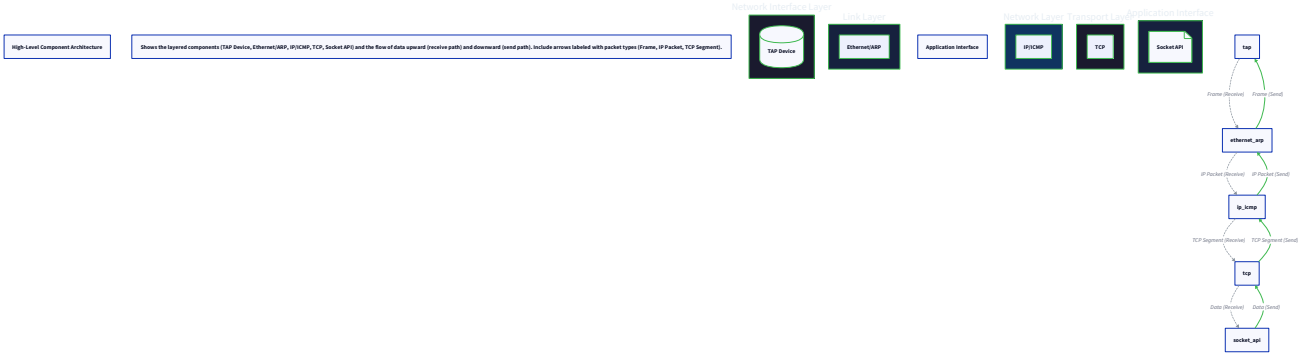
The TCP/IP stack follows a **layered architecture**, where each layer has a specific responsibility and communicates only with the layers immediately above and below it. This separation of concerns is the single most important design principle—it allows each layer to evolve independently and makes the entire system more comprehensible, testable, and debuggable.

> **Architectural Insight:** The layered model is like a factory assembly line. Each station (layer) performs a specific operation on the product (packet) without needing to understand the entire manufacturing process. The Ethernet station just handles local delivery labels, the IP station handles city-wide routing, and the TCP station ensures the product arrives intact and in order. This modularity is why the Internet can scale and innovate incrementally.

Our educational implementation consists of five logical layers, each with corresponding software components:

Layer	Primary Responsibility	Software Components	Analogy (Postal Service)
Link (Layer 2)	Direct communication between devices on the same physical network using MAC addresses.	TAP Device Interface, Ethernet Frame Parser, ARP Protocol Handler	The local mail carrier who physically moves letters between houses on your street, using street addresses (MAC addresses) to know which mailbox to use.
Network (Layer 3)	Logical addressing (IP) and routing between different networks, including error reporting.	IPv4 Packet Handler, ICMP Protocol, Routing Table	The postal sorting facility that reads zip codes (IP addresses), decides which truck/plane to use, and handles "return to sender" notifications (ICMP errors) for undeliverable mail.
Transport (Layer 4)	End-to-end communication between applications, providing reliability, ordering, and flow control.	TCP Segment Parser, TCP State Machine, Sliding Window, Retransmission Timers	The registered mail service that tracks each letter, requests delivery confirmations (ACKs), resends if lost, and ensures the recipient isn't overwhelmed (flow control).
Socket API (Layer 4/5)	Programming interface that allows applications to use the network stack without understanding protocol details.	Socket Interface (Simplified)	The standardized envelope and mailbox system—applications just write a letter and drop it in the mailbox without knowing about sorting facilities or carrier routes.
Application (Layer 7)	User programs that generate and consume network data (e.g., a simple echo server, HTTP client).	Example Applications (Test Clients/Servers)	The person writing or reading the letter—the ultimate producer and consumer of the communication.

These layers interact through well-defined interfaces. Data flows **downward** through the stack when sending (from application to network) and **upward** when receiving (from network to application). This vertical data flow is complemented by **horizontal communication** between peer layers on different machines—when our TCP layer sends a segment, it's ultimately communicating with the TCP layer on the remote host, even though the data physically passes through all lower layers on both ends.



The diagram above illustrates the major components and their relationships. The **TAP Device** serves as our virtual network interface, providing raw Ethernet frame access. The **Ethernet/ARP** component handles local delivery and address resolution. The **IP/ICMP** component manages routing and error reporting. The **TCP** component ensures reliable, ordered data streams. Finally, a simplified **Socket API** exposes network functionality to applications.

Data Flow Through the Layers

Let's trace the lifecycle of a single TCP data segment to understand how components interact:

Sending Path (Application → Network Wire):

1. An application calls `socket_send()` with data to transmit

2. The TCP layer receives the data, encapsulates it in a TCP segment with sequence numbers, flags, and checksum
3. The IP layer receives the TCP segment, adds an IP header with source/destination addresses and its own checksum
4. The ARP subsystem resolves the next-hop IP address to a MAC address (consulting cache or sending ARP request)
5. The Ethernet layer frames the IP packet with source/destination MAC addresses and EtherType
6. The TAP device writes the complete Ethernet frame to the virtual network interface

Receiving Path (Network Wire → Application):

1. The TAP device reads an incoming Ethernet frame from the virtual network interface
2. The Ethernet layer validates the frame and checks the destination MAC (accepting if it matches our address or is broadcast)
3. Based on EtherType, the frame is dispatched—ARP frames go to the ARP handler, IP frames go to the IP layer
4. The IP layer validates the IP header (checksum, version), decrements TTL, and checks destination IP
5. Based on IP protocol field, the packet is dispatched—ICMP goes to ICMP handler, TCP goes to TCP layer
6. The TCP layer finds the matching connection (TCP Control Block), processes the segment according to the state machine, and delivers data to the application's receive buffer
7. The application reads the data via `socket_recv()`

> **Key Design Principle: Layered Encapsulation** Each layer adds its own header (and sometimes trailer) to the data from the layer above, creating a nested structure like Russian dolls. When receiving, each layer strips its own header before passing the payload upward. This encapsulation is why we can swap out physical network technologies (Ethernet, Wi-Fi, cellular) without changing the IP layer above—each link layer just provides a standard way to carry IP packets.

Recommended File/Module Structure

A clean, organized codebase is essential for managing the complexity of a network stack. We recommend a directory structure that mirrors the layered architecture, separating concerns logically and making dependencies explicit. This structure also facilitates incremental development—you can complete and test one layer before moving to the next.

```
project_valiant/
│
├── docs/                          # Project documentation
│   ├── design/                    # Design documents (this document)
│   └── diagrams/                  # Architecture diagrams
│
├── include/                       # Public header files (if creating a library)
│   └── valiant/                   # Namespaced headers
│       ├── ethernet.h
│       ├── arp.h
│       ├── ipv4.h
│       ├── icmp.h
│       ├── tcp.h
│       └── socket.h
│
├── src/                           # Main source code
│   ├── main.c                     # Application entry point, TAP setup, main loop
│   │
│   ├── netif/                     # Network interface abstraction
│   │   ├── tap.c                  # TAP device I/O implementation
│   │   └── netif.h                # Interface abstraction (tap_open, tap_read, tap_write)
│   │
│   ├── link/                      # Link Layer (Milestone 1)
│   │   ├── ethernet.c             # Ethernet frame parsing/construction
│   │   ├── arp.c                  # ARP protocol implementation
│   │   ├── arp_cache.c            # ARP cache management with timeout
│   │   └── link.h                 # Link layer API (netif_rx_packet, etc.)
│   │
│   ├── network/                   # Network Layer (Milestone 2)
│   │   ├── ipv4.c                 # IPv4 packet processing, checksum
│   │   ├── icmp.c                 # ICMP protocol (echo request/reply)
│   │   ├── route.c                # Routing table implementation
│   │   └── network.h              # Network layer API (ipv4_input, ipv4_output)
│   │
│   ├── transport/                 # Transport Layer (Milestones 3 & 4)
│   │   ├── tcp/                   # TCP implementation
│   │   │   ├── tcp.c              # Main TCP input/output, demultiplexing
│   │   │   ├── tcp_state.c        # TCP state machine implementation
│   │   │   ├── tcp_timer.c        # Retransmission and keepalive timers
│   │   │   ├── tcp_window.c       # Sliding window and flow control
│   │   │   └── tcp_cc.c           # Congestion control (slow start, avoidance)
│   │   ├── tcp.h                  # TCP public API and structures
│   │   └── tcp_internal.h         # TCP internal structures and helpers
│   │
│   ├── socket/                    # Socket API (simplified)
│   │   ├── socket.c               # Socket creation, bind, listen, accept, connect
│   │   └── socket.h               # Socket API definitions
│   │
│   ├── utils/                     # Cross-cutting utilities
│   │   ├── checksum.c             # Internet checksum calculation
│   │   ├── buffer.c               # Packet buffer management
│   │   ├── list.c                 # Linked list implementation
│   │   ├── timer.c                # Timer wheel for timeouts
│   │   └── utils.h
│   │
│   └── apps/                      # Example applications using the stack
│       ├── ping.c                 # Simple ping utility
│       ├── tcp_echo_server.c      # TCP echo server
│       └── tcp_echo_client.c      # TCP echo client
│
├── tests/                         # Test suite
│   ├── unit/                      # Unit tests per module
│   ├── integration/               # Integration tests
│   └── scripts/                   # Test automation scripts
│
└── Makefile                       # Build configuration
```

This structure enforces several important design principles:

1. **Separation of Concerns:** Each directory corresponds to a logical layer or subsystem. The `link/` directory knows nothing about TCP, and the `transport/` directory doesn't directly interact with TAP devices.
2. **Clear Dependencies:** Header files define clean interfaces between layers. For example, `network/network.h` declares functions like `ipv4_output()` that the TCP layer calls to send packets, but TCP doesn't need to include Ethernet headers.
3. **Incremental Development:** You can build and test each milestone in isolation. Milestone 1 lives in `link/`, Milestone 2 in `network/`, etc.
4. **Reusable Utilities:** Common functionality (checksums, buffers, timers) is extracted to `utils/` to avoid duplication and ensure consistency.

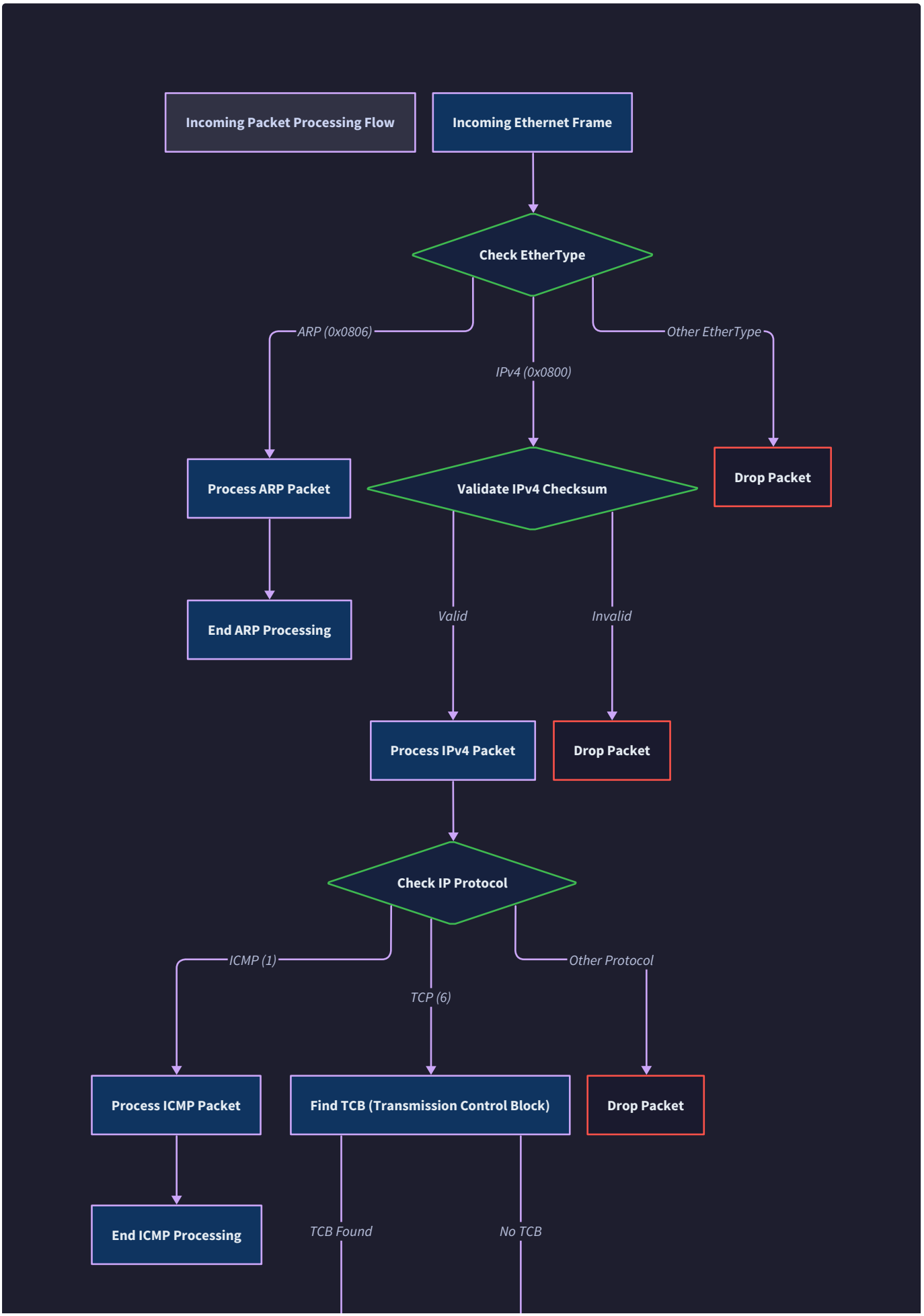
5. **Testability:** The `tests/` directory mirrors the source structure, making it easy to add unit tests for each component.

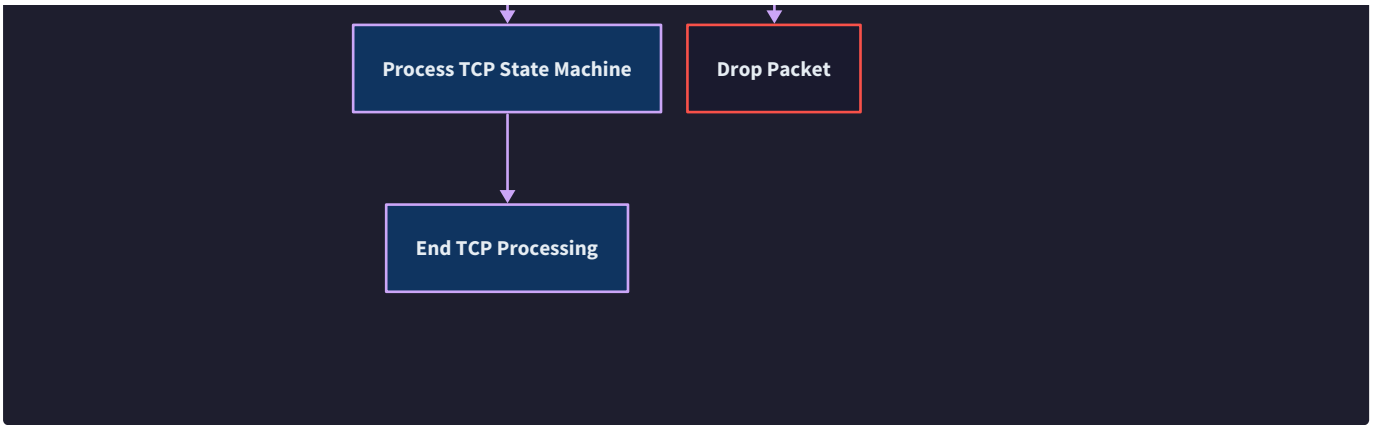
> **Implementation Strategy:** Start with the bottom (`netif/` and `link/`) and work your way up the stack. Each layer should be fully functional and tested before implementing the layer above. This bottom-up approach ensures you have a solid foundation—you can't build reliable TCP connections on top of a broken IP layer.

Module Interaction Patterns

The components interact through three primary patterns:

1. **Layer Callbacks (Upward):** Lower layers call upward into higher layers via function pointers or direct function calls registered during initialization. For example, the IP layer registers a handler with the Ethernet layer for EtherType 0x0800 (IPv4), and the TCP layer registers with the IP layer for protocol 6.
2. **Service Requests (Downward):** Higher layers request services from lower layers via exported APIs. For example, TCP calls `ipv4_output()` to send a segment, which eventually calls `tap_write()` to output the frame.
3. **Shared Data Structures:** Certain structures like the ARP cache and routing table are accessed by multiple layers. These should be protected by appropriate synchronization (mutexes) in a multi-threaded implementation, though our initial implementation can be single-threaded for simplicity.


Incoming Packet Processing Flow
Incoming Ethernet Frame
Check EtherType
ARP (0x0806)
IPv4 (0x0800)
Other EtherType
Process ARP Packet
Validate IPv4 Checksum
Drop Packet
End ARP Processing
Valid
Invalid
Process IPv4 Packet
Drop Packet
Check IP Protocol
ICMP (1)
TCP (6)
Other Protocol
Process ICMP Packet
Find TCB (Transmission Control Block)
Drop Packet
End ICMP Processing
TCB Found
No TCB



The flowchart above details the precise decision logic for processing incoming packets, showing how each layer dispatches to the appropriate next handler based on protocol identifiers in the headers.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option (For Further Exploration)
Link Layer I/O	Linux TAP device (`/dev/net/tun`)	Raw sockets with BPF filtering, DPDK for high performance
Timer Management	Single periodic timer wheel scanning all connections	Per-connection kernel timers (`timerfd`), event-driven with callback queues
Buffer Management	Fixed-size preallocated buffers with reference counting	Dynamic buffer chains with scatter-gather I/O support
Concurrency	Single-threaded event loop (simple, deterministic)	Multi-threaded with per-CPU queues, lock-free data structures

For our educational implementation, we recommend the **Simple Option** for each component. The goal is understanding protocol logic, not building a production-grade stack.

Recommended File Structure Implementation

Here's a concrete starting point for the directory structure and key header files:

`src/main.c` - Application Entry Point and Main Loop:

```c
#include "netif/netif.h"
#include "link/link.h"
#include "network/network.h"
#include "transport/tcp.h"
#include "utils/timer.h"

int main(int argc, char *argv[]) {
    // 1. Initialize TAP device
    int tap_fd = tap_open("valiant0");
    if (tap_fd < 0) {
        fprintf(stderr, "Failed to open TAP device\n");
        return 1;
    }

    // 2. Initialize stack components (bottom-up)
    arp_cache_init();
    route_table_init();
    tcp_init();

    // 3. Main event loop
    uint8_t frame_buffer[MAX_FRAME_SIZE];
    while (1) {
        // Check for incoming frames
        ssize_t len = tap_read(tap_fd, frame_buffer, MAX_FRAME_SIZE);
        if (len > 0) {
            // Process received frame (starts at link layer)
            netif_rx_packet(frame_buffer, len);
        }

        // Process timers (TCP retransmissions, ARP cache expiry, etc.)
        timer_process();

        // Handle pending socket operations (simplified - in real impl would use select/poll)
        socket_process_pending();
    }

    return 0;
}
```

`src/netif/netif.h` - Network Interface Abstraction:

```c
#ifndef VALIANT_NETIF_H

#define VALIANT_NETIF_H


#include <stdint.h>

#include <stddef.h>


#define MAX_FRAME_SIZE 1600  // MTU (1500) + Ethernet header (14) + some margin


// Open a TAP device with the given name

// Returns file descriptor on success, negative on error

int tap_open(const char *dev_name);


// Read an Ethernet frame from TAP device

// Returns number of bytes read, 0 on timeout, negative on error

ssize_t tap_read(int fd, void *buf, size_t len);


// Write an Ethernet frame to TAP device

// Returns number of bytes written, negative on error

ssize_t tap_write(int fd, const void *buf, size_t len);


// Called by main loop to process received packet

void netif_rx_packet(const uint8_t *frame, size_t len);


#endif // VALIANT_NETIF_H
```

`src/utils/utils.h` - Common Utilities:

```c
#ifndef VALIANT_UTILS_H

#define VALIANT_UTILS_H


#include <stdint.h>

#include <stddef.h>


// Convert 16-bit and 32-bit values between host and network byte order

uint16_t htons(uint16_t hostshort);

uint16_t ntohs(uint16_t netshort);

uint32_t htonl(uint32_t hostlong);

uint32_t ntohl(uint32_t netlong);


// Calculate Internet checksum (RFC 1071)

uint16_t checksum(const void *data, size_t len, uint32_t initial);


// Wrapper for IP/TCP checksum calculation

uint16_t ip_checksum(const void *hdr, size_t len);


#endif // VALIANT_UTILS_H
```

Core Logic Skeleton

`src/link/ethernet.c` - Ethernet Frame Parser (Skeleton):

```c
#include "link/link.h"
#include "utils/utils.h"
#include <string.h>

// TODO: Define Ethernet header structure (see Data Model section)
// struct eth_hdr { ... };

void ethernet_process_frame(const uint8_t *frame, size_t len) {
    // TODO 1: Check frame length (minimum 64 bytes, maximum MAX_FRAME_SIZE)
    // TODO 2: Parse Ethernet header (destination MAC, source MAC, EtherType)
    // TODO 3: Validate destination MAC (accept if broadcast, multicast, or our MAC)
    // TODO 4: Dispatch based on EtherType:
    //   - 0x0800 (IPv4): Call ipv4_input() with IP packet (frame + header size)
    //   - 0x0806 (ARP): Call arp_process_packet() with ARP packet
    //   - Otherwise: Drop frame (unsupported protocol)
}

void ethernet_build_frame(uint8_t *frame, const uint8_t *dst_mac,
                          const uint8_t *src_mac, uint16_t ethertype,
                          const uint8_t *payload, size_t payload_len) {
    // TODO 1: Fill Ethernet header fields (destination, source, EtherType)
    // TODO 2: Copy payload after header
    // TODO 3: Note: This function will be called by ARP and IP layers to send frames
}
```

Language-Specific Hints (C)

1. **Byte Order Conversion:** Use `htons()`, `ntohs()`, `htonl()`, `ntohl()` from `<arpa/inet.h>` or implement your own versions in `utils.c`. Network byte order is **big-endian** (most significant byte first).
2. **Structure Packing:** Use `__attribute__((packed))` in GCC or `#pragma pack(1)` to ensure protocol header structures match the on-wire layout without compiler padding.
3. **Memory Management:** For simplicity, use stack-allocated buffers or preallocated pools. Avoid dynamic allocation in performance-critical paths.
4. **Timer Implementation:** A simple timer wheel can be implemented using a sorted linked list of timer events checked in the main loop. Use `gettimeofday()` or `clock_gettime()` for timestamps.

Milestone Checkpoint

After implementing the high-level architecture with the file structure and basic main loop:

Verification Steps:

1. **Compile the skeleton:** Run `make` (with a simple Makefile) to ensure no syntax errors.
2. **Check TAP device creation:** Run the program with sudo: `sudo ./valiant`. It should create a TAP device `valiant0`.
3. **Verify structure:** Use `ls -la src/` to confirm all directories and skeleton files are in place.
4. **Basic connectivity:** In another terminal, bring up the TAP interface: `sudo ip link set valiant0 up` and assign an IP: `sudo ip addr add 10.0.0.1/24 dev valiant0`.

Expected Output:

- Program starts without errors

- TAP device appears in `ip link show`
- No crashes in main loop (will just wait for packets)

Troubleshooting:

- **"Permission denied" on `/dev/net/tun`:** Ensure you're running as root or have appropriate permissions.
- **"No such file or directory" for headers:** Check include paths in your Makefile.
- **Compiler warnings about packed structures:** Verify you're using the correct attribute syntax for your compiler.

This architectural foundation sets the stage for implementing each protocol layer. The clean separation and clear interfaces will make the subsequent milestones more manageable and the entire system more understandable.

4. Data Model

> **Milestone(s):** 1, 2, 3, 4 (Core Foundation)

At the heart of any TCP/IP stack lies a collection of data structures that faithfully represent the protocol messages and connection state. Think of this data model as the **blueprints and filing system** for our digital post office. The blueprints (`eth_hdr`, `ip_hdr`, `tcp_hdr`) define exactly how to read and construct protocol messages—what each byte means, where addresses go, how flags are set. The filing system (`arp_entry`, `route_entry`, `tcb`) keeps track of everything our stack needs to remember: who we've talked to, where to send things next, and the complete state of every ongoing conversation.

This section defines these fundamental structures with precision. Getting them right is critical because every byte in network protocols has specific meaning, and every field in our state structures must accurately track the complex dance of connection management and data transfer. We'll define these structures in C, paying careful attention to byte ordering, alignment, and memory layout to ensure they match exactly what travels on the wire.

Packet & Header Structures

Network protocols communicate through precisely formatted headers—structured metadata that precedes actual data. Each header is like a **standardized envelope** with specific fields in fixed positions. When our stack receives a raw stream of bytes from the `TAP Device`, it must interpret these bytes according to these standardized formats. Conversely, when sending data, it must construct headers that other stacks can parse correctly.

All multi-byte integer fields in these headers (like `ethertype`, `total_len`, `seq_num`) use **Network Byte Order** (big-endian). This means the most significant byte comes first in memory, regardless of the host system's native byte order (which on modern x86 systems is little-endian). We must use conversion functions like `htons()` and `ntohs()` when reading from or writing to these structures.

The following tables define the exact memory layout for each protocol header. Note the use of compiler directives like `__attribute__((packed))` in C (or `#[repr(packed)]` in Rust) to prevent the compiler from inserting padding bytes between fields, which would break the wire format alignment.

Ethernet Frame Header (`eth_hdr`)

The Ethernet header is the **outermost envelope** in our postal analogy. It contains the hardware (MAC) addresses for the immediate sender and recipient on the local network segment, plus a type field indicating what kind of payload follows.

Field Name	Type	Description
`dst_mac`	`uint8_t[6]`	Destination MAC address (6 bytes). For example, `{0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}` is the broadcast address.
`src_mac`	`uint8_t[6]`	Source MAC address (6 bytes). Identifies the sender's network interface card.
`ethertype`	`uint16_t`	Protocol type of the payload, in **Network Byte Order**. Common values: `0x0800` for IPv4, `0x0806` for ARP.

> **Design Insight:** The Ethernet header uses fixed-size byte arrays for MAC addresses rather than a 48-bit integer because MAC addresses are typically displayed and manipulated as six colon-separated hexadecimal bytes (like `AA:BB:CC:DD:EE:FF`). Storing them as a byte array makes formatting and comparison straightforward.

IPv4 Header (`ip_hdr`)

The IP header functions as the **routing label** on our postal envelope. It contains logical (IP) addresses for the ultimate source and destination, which may be many network hops apart, along with control information for fragmentation, time-to-live, and the protocol of the encapsulated payload.

Field Name	Type	Description
`version_ihl`	`uint8_t`	Combined field: 4 bits for IP version (must be `4`), 4 bits for Internet Header Length (IHL) measured in 32-bit words. Minimum value is `0x45` (version 4, 5 words = 20 bytes).
`tos`	`uint8_t`	Type of Service (historically for QoS, often set to `0`).
`total_len`	`uint16_t`	Total length of the IP packet (header + data) in bytes, in **Network Byte Order**.
`id`	`uint16_t`	Identification field used for fragmentation reassembly, in **Network Byte Order**.
`frag_off`	`uint16_t`	Combined field: 3 bits for flags (MF, DF), 13 bits for fragment offset, in **Network Byte Order**.
`ttl`	`uint8_t`	Time To Live - decremented by each router; packet is discarded when it reaches `0`.
`protocol`	`uint8_t`	Protocol of the payload (e.g., `6` for TCP, `1` for ICMP).
`checksum`	`uint16_t`	Header checksum, in **Network Byte Order**. Calculated over the IP header only.
`src_ip`	`uint32_t`	Source IP address, in **Network Byte Order** (e.g., `0xC0A80001` for `192.168.0.1`).
`dst_ip`	`uint32_t`	Destination IP address, in **Network Byte Order**.
(optional variable-length options)		If IHL > 5, additional option bytes follow. Our implementation will initially ignore options.

> **Important:** The `checksum` field is computed **after** all other header fields are filled. When receiving a packet, we verify this checksum; if it doesn't match, we drop the packet silently.

TCP Header (`tcp_hdr`)

The TCP header is the **conversation control slip** attached to each segment in a reliable stream. It contains port numbers to demultiplex multiple connections between the same IP addresses, sequence and acknowledgment numbers for tracking data flow, control flags for connection management, and window size for flow control.

Field Name	Type	Description
`src_port`	`uint16_t`	Source port number, in **Network Byte Order**.
`dst_port`	`uint16_t`	Destination port number, in **Network Byte Order**.
`seq_num`	`uint32_t`	Sequence number of the first data byte in this segment (or the Initial Sequence Number for SYN segments), in **Network Byte Order**.
`ack_num`	`uint32_t`	Acknowledgment number (valid if ACK flag set), in **Network Byte Order**. This is the next sequence number the sender expects to receive.
`data_offset_reserved_flags`	`uint16_t`	Combined field: 4 bits data offset (header length in 32-bit words), 4 bits reserved (must be `0`), 8 bits for flags (CWR, ECE, URG, ACK, PSH, RST, SYN, FIN). Stored in **Network Byte Order**.
`window`	`uint16_t`	Receive window size (number of bytes the sender of this segment is willing to receive), in **Network Byte Order**.
`checksum`	`uint16_t`	TCP checksum (covers pseudo-header, TCP header, and data), in **Network Byte Order**.
`urg_ptr`	`uint16_t`	Urgent pointer (valid if URG flag set), in **Network Byte Order**. We will not implement urgent data.
(optional variable-length options)		If data offset > 5, TCP options follow (e.g., Maximum Segment Size, Window Scale).

> **Mental Model:** Think of `seq_num` and `ack_num` as two synchronized counters tracking the conversation. `seq_num` says "this is where my data starts," while `ack_num` says "I've received everything up to this point." They allow both sides to detect missing, duplicate, or out-of-order data.

ICMP Header (`icmp_hdr`)

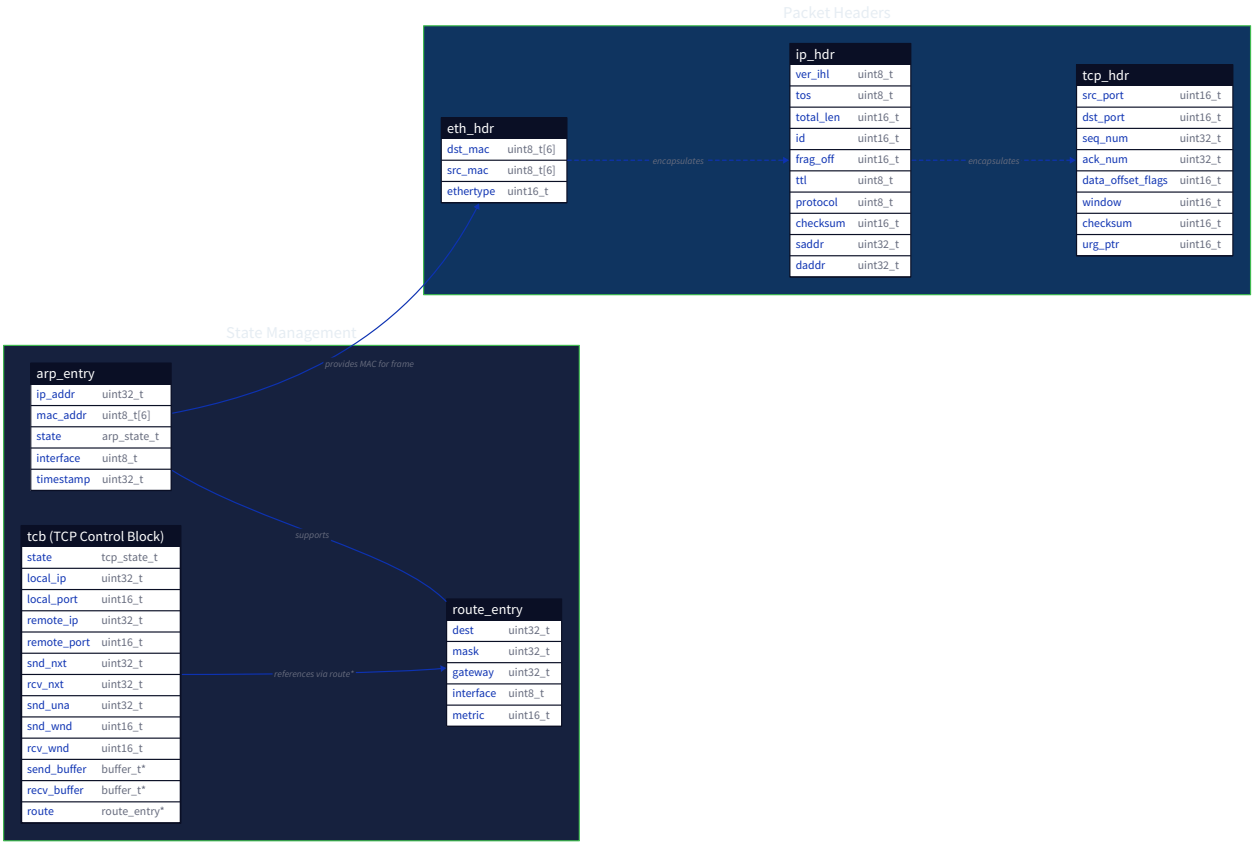
The ICMP header is the **system administrator's note**—used for diagnostic and error messages. For ping (echo request/reply), it has a simple structure identifying the message type and carrying a sequence identifier.

Field Name	Type	Description
`type`	`uint8_t`	ICMP message type (`8` = Echo Request, `0` = Echo Reply).
`code`	`uint8_t`	Subtype (for Echo messages, always `0`).
`checksum`	`uint16_t`	ICMP checksum (covers ICMP header and data), in **Network Byte Order**.
`id`	`uint16_t`	Identifier (used to match requests with replies, often process ID), in **Network Byte Order**.
`seq`	`uint16_t`	Sequence number (used to match requests with replies), in **Network Byte Order**.
(payload data)		For Echo messages, the payload data (variable length) follows.

State Management Structures

While headers represent transient messages, the stack must maintain persistent state to function correctly. These state structures are the **memory and ledgers** of our post office: they remember past interactions, track ongoing conversations, and store pending work.

The relationships between these structures are illustrated in the diagram below, which shows how a `tcb` references routing information, contains buffers, and is indexed by connection quadruples:



ARP Cache Entry (`arp_entry`)

The ARP cache maps IP addresses to MAC addresses on the local network, avoiding the need to broadcast an ARP request for every packet. It's a **neighborhood directory** that remembers "who lives where."

Field Name	Type	Description
`ip_addr`	`uint32_t`	IP address (key), in **Network Byte Order**.
`mac_addr`	`uint8_t[6]`	Corresponding MAC address (value).
`state`	`enum arp_state`	Entry state: `ARP_INCOMPLETE` (request sent, no reply yet), `ARP_RESOLVED` (mapping known), `ARP_STALE` (entry may be outdated).
`last_updated`	`time_t` or `uint64_t`	Timestamp when entry was last updated (seconds since epoch or monotonic clock). Used for expiration.
`retry_count`	`uint8_t`	Number of ARP requests sent without reply. After a limit, entry is removed.

> **ADR: Fixed-Size vs. Dynamic ARP Cache**
>
> **Context:** The stack needs to store IP-to-MAC mappings but memory is finite. We must decide how to structure the cache.
>
> **Options Considered:**
>
> 1. **Dynamic hash table:** Grow/shrink as needed, no fixed limit.
> 2. **Fixed-size array with LRU:** Simple array, evict least recently used entries when full.
> 3. **Fixed-size array with timeout only:** Simple array, entries expire only by timeout, new entries fail when full.
>
> **Decision:** Fixed-size array with timeout-based expiration (option 3).
>
> **Rationale:** Simplicity for educational implementation. A fixed array is easier to implement correctly in C than dynamic structures, and timeout-based expiration matches typical ARP behavior (entries become stale after 15-20 minutes). While LRU is more efficient, the educational focus is on protocol logic, not cache algorithms.
>
> **Consequences:** The cache has a maximum size (e.g., 64 entries). If full, new unresolved IPs cannot be added until old entries expire. This could cause temporary failures in high-traffic scenarios but is acceptable for learning.

Option	Pros	Cons	Chosen?
Dynamic hash table	No size limits, O(1) average lookup	Complex implementation, memory management overhead	No
Fixed array + LRU	Bounded memory, better utilization than timeout-only	LRU adds complexity, need to track access times	No
Fixed array + timeout only	Extremely simple, matches RFC behavior	May fill up and reject new mappings	**Yes**

Routing Table Entry (`route_entry`)

The routing table determines where to send packets based on their destination IP address. It's the **post office's routing guide** that says "for this destination, use this next-hop gateway and interface."

Field Name	Type	Description
`dest_net`	`uint32_t`	Destination network address (with mask applied), in **Network Byte Order**.
`dest_mask`	`uint32_t`	Network mask, in **Network Byte Order** (e.g., `0xFFFFFF00` for /24).
`gateway`	`uint32_t`	Next-hop gateway IP address (or `0` if destination is directly reachable), in **Network Byte Order**.
`iface_index`	`uint8_t`	Index of the network interface to use (for multi-homed hosts). We'll use `0` for our single TAP interface.
`metric`	`uint16_t`	Route cost (lower is preferred). Used when multiple routes match.

> **Mental Model:** The routing table is like a decision tree. For each outgoing packet, we find the most specific match (longest prefix match) between the destination IP and `dest_net` / `dest_mask` combinations. The matching entry tells us whether the destination is on the local network (send directly via ARP) or requires a gateway (send to gateway's MAC).

TCP Control Block (`tcb`)

The TCP Control Block is the **master file for a single conversation**. It contains all state necessary to manage a TCP connection: sequence numbers, buffers, timers, and the current state machine position. Each active connection has exactly one `tcb`.

Field Name	Type	Description
`local_ip`	`uint32_t`	Local IP address, in **Network Byte Order**.
`local_port`	`uint16_t`	Local port number, in **Network Byte Order**.
`remote_ip`	`uint32_t`	Remote IP address, in **Network Byte Order**.
`remote_port`	`uint16_t`	Remote port number, in **Network Byte Order**.
`state`	`enum tcp_state`	Current connection state (`LISTEN` , `SYN_SENT` , `ESTABLISHED` , etc.).
`send`	`struct tcp_send_vars`	Send-side variables (see sub-structure below).
`recv`	`struct tcp_recv_vars`	Receive-side variables (see sub-structure below).
`timer`	`struct tcp_timers`	Timer states (retransmission, persistence, keepalive).
`rx_buffer`	`struct tcp_buffer*`	Pointer to receive buffer (ring buffer for incoming data).
`tx_buffer`	`struct tcp_buffer*`	Pointer to send buffer (ring buffer for unacknowledged data).
`congestion`	`struct tcp_congestion`	Congestion control state (cwnd, ssthresh, etc.).

Sub-structure: `tcp_send_vars` (Send-side state)

Field Name	Type	Description
`nxt`	`uint32_t`	Next sequence number to send.
`una`	`uint32_t`	Oldest unacknowledged sequence number.
`wnd`	`uint16_t`	Receive window size advertised by remote peer.
`up`	`uint32_t`	Urgent pointer (unused in our implementation).
`wl1`	`uint32_t`	Segment sequence number used for last window update.
`wl2`	`uint32_t`	Segment acknowledgment number used for last window update.
`iss`	`uint32_t`	Initial send sequence number.

Sub-structure: `tcp_recv_vars` (Receive-side state)

Field Name	Type	Description
`nxt`	`uint32_t`	Next sequence number expected.
`wnd`	`uint16_t`	Receive window we advertise to peer.
`up`	`uint32_t`	Urgent pointer (unused).
`irs`	`uint32_t`	Initial receive sequence number (from peer's SYN).

Sub-structure: `tcp_timers` (Timer management)

Field Name	Type	Description
`rto`	`uint32_t`	Retransmission timeout value (milliseconds).
`rxt_time`	`uint64_t`	Timestamp when oldest unacknowledged segment was sent.
`rxt_count`	`uint8_t`	Number of retransmissions for current segment.
`persist_time`	`uint64_t`	Timestamp for persist timer (zero window probing).
`keepalive_time`	`uint64_t`	Timestamp for keepalive timer (we may not implement).

Sub-structure: `tcp_congestion` (Congestion control)

Field Name	Type	Description
`cwnd`	`uint32_t`	Congestion window (bytes allowed in flight).
`ssthresh`	`uint32_t`	Slow start threshold (bytes).
`dupack_count`	`uint8_t`	Count of duplicate ACKs (for fast retransmit).

> **ADR: Centralized TCB Table vs. Per-Socket State**
>
> **Context:** We need to track state for multiple TCP connections simultaneously. Where should this state live?
>
> **Options Considered:**
>
> 1. **Centralized TCB table:** Global array or hash table of `tcb` structures, indexed by connection quadruple (local/remote IP/port).
> 2. **Per-socket state:** Each socket file descriptor holds its connection state directly.
> 3. **Hybrid:** Socket holds pointer to TCB in central table.
>
> **Decision:** Centralized TCB table (option 1).
>
> **Rationale:** This follows the traditional BSD stack design and cleanly separates connection management from the socket API. It allows the TCP layer to operate independently, looking up TCBs based on incoming packet headers. This design also simplifies implementing features that affect multiple sockets (like routing changes) and matches the mental model of TCP as a connection-oriented protocol managed by the stack itself.
>
> **Consequences:** The TCP input routine must efficiently find the correct TCB for each incoming segment. We'll implement a simple hash table or linear search for small-scale educational use.

Option	Pros	Cons	Chosen?
Centralized TCB table	Clean separation, matches RFC model, efficient for packet processing	Requires lookup on every packet, global synchronization	**Yes**
Per-socket state	Simple for single-connection apps, no lookup needed	Hard to handle incoming packets for listening sockets, mixes layers	No
Hybrid	Combines benefits of both	More complex, still requires some lookup	No

TCP Buffer (`tcp_buffer`)

TCP buffers hold data that is either waiting to be sent (send buffer) or has been received but not yet read by the application (receive buffer). They're the **holding areas** in our post office where packages wait for pickup or delivery.

Field Name	Type	Description
`data`	`uint8_t*`	Pointer to dynamically allocated buffer memory.
`size`	`size_t`	Total capacity of buffer (bytes).
`head`	`size_t`	Index of first valid byte (read pointer).
`tail`	`size_t`	Index where next byte should be written (write pointer).
`readable`	`size_t`	Number of bytes currently available to read (`(tail - head) mod size`).

> **Implementation Note:** We implement this as a circular buffer (ring buffer). When `head` reaches `size` , it wraps to `0` . The buffer is empty when `head == tail` , and full when `((tail + 1) % size) == head` (leaving one slot unused to distinguish empty from full).

Common Pitfalls in Data Model Design

⚠️ Pitfall: Forgetting Network Byte Order

- **Description:** Storing or comparing header fields without converting between host and network byte order.
- **Why it's wrong:** On little-endian systems (x86), the value `0x0001` stored in memory as `01 00` will be interpreted as `0x0100` if read directly as a 16-bit integer, breaking all protocol logic.
- **Fix:** Always use `ntohs()` , `ntohl()` , `htons()` , and `htonl()` when reading from or writing to header structures. Create helper functions like `ip_hdr_ntoh()` that convert all fields at once.

⚠️ Pitfall: Structure Padding Breaking Wire Format

- **Description:** The C compiler inserts padding bytes between structure fields for alignment, causing the in-memory layout to differ from the on-wire layout.
- **Why it's wrong:** A `sizeof(ip_hdr)` might be 24 bytes instead of 20 due to padding, causing serialization/deserialization to misalign fields.

- **Fix:** Use `__attribute__((packed))` (GCC/Clang) or `#pragma pack(1)` (MSVC) to force byte-aligned packing. Always verify structure size with `sizeof()` matches the RFC specification.

⚠️ Pitfall: Integer Overflow in Sequence Number Arithmetic

- **Description:** Treating 32-bit TCP sequence numbers as regular integers without handling wraparound from `0xFFFFFFFF` to `0`.
- **Why it's wrong:** TCP sequence numbers wrap after 4GB of data. Comparisons like `seq1 < seq2` fail when crossing the wrap boundary.
- **Fix:** Use modular arithmetic with helper functions: `seq_lt(a, b)` returns true if `a` is less than `b` modulo 2^32. Implement as `((int32_t)(a - b) < 0)`.

⚠️ Pitfall: Not Zeroing Structures Before Use

- **Description:** Allocating a `tcb` or header structure without initializing all fields, leaving garbage in reserved or padding areas.
- **Why it's wrong:** Uninitialized fields may contain random bytes that affect checksums or cause protocol violations. Reserved bits must be zero.
- **Fix:** Always use `memset(ptr, 0, sizeof(struct))` or calloc() before filling structure fields. For headers, explicitly set reserved bits to zero.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option (Educational)	Advanced Option (Production)
Data Structures	Plain C structs with packed attributes	Type-safe wrappers with validation methods
Buffer Management	Fixed-size circular buffers	Dynamic buffer pools with slab allocation
Timer Management	Single periodic timer wheel	High-resolution timerfd or timer threads

B. Recommended File/Module Structure:

```
project-valiant/
├── include/
│   ├── stack/
│   │   ├── eth.h          ← Ethernet & ARP structures/functions
│   │   ├── ip.h           ← IPv4 structures/functions
│   │   ├── icmp.h         ← ICMP structures/functions
│   │   ├── tcp.h          ← TCP structures/functions
│   │   ├── types.h        ← Common types (byte order helpers)
│   │   └── buffer.h       ← Buffer management
│   └── arch/
│       └── byteorder.h    ← Architecture-specific byte order handling
├── src/
│   ├── net/
│   │   ├── eth.c          ← Ethernet implementation
│   │   ├── arp.c          ← ARP implementation
│   │   ├── ip.c           ← IPv4 implementation
│   │   ├── icmp.c         ← ICMP implementation
│   │   ├── tcp.c          ← TCP implementation
│   │   ├── tcp_timer.c    ← TCP timer management
│   │   ├── tcp_buffer.c   ← TCP buffer management
│   │   └── route.c        ← Routing table
│   └── platform/
│       ├── tap.c          ← TAP device interface
│       └── time.c         ← Timer utilities
└── tests/
    ├── unit/
    │   ├── test_headers.c ← Header structure validation
    │   └── test_buffer.c  ← Buffer operations
    └── integration/
        └── packet_test.c  ← End-to-end packet tests
```

C. Infrastructure Starter Code:

```c
/* include/stack/types.h - Common type definitions and utilities */

#ifndef STACK_TYPES_H

#define STACK_TYPES_H


#include <stdint.h>

#include <stddef.h>

#include <string.h>


/* Compiler-specific packed attribute */

#ifdef __GNUC__

#define PACKED __attribute__((packed))

#else

#define PACKED

#pragma pack(push, 1)

#endif


/* Network byte order conversion helpers */

#ifndef htons

#define htons(x) ((((x) & 0xFF00) >> 8) | (((x) & 0x00FF) << 8))

#endif


#ifndef ntohs

#define ntohs htons

#endif


#ifndef htonl

#define htonl(x) ((((x) & 0xFF000000) >> 24) | \

                  (((x) & 0x00FF0000) >> 8)  | \

                  (((x) & 0x0000FF00) << 8)  | \

                  (((x) & 0x000000FF) << 24))

#endif


#ifndef ntohl

#define ntohl htonl

#endif


/* TCP sequence number comparison helpers */

static inline int32_t seq_diff(uint32_t a, uint32_t b) {

    return (int32_t)(a - b);

}


static inline int seq_lt(uint32_t a, uint32_t b) {

    return seq_diff(a, b) < 0;

}
```

```c
static inline int seq_le(uint32_t a, uint32_t b) {
    return seq_diff(a, b) <= 0;
}

static inline int seq_gt(uint32_t a, uint32_t b) {
    return seq_diff(a, b) > 0;
}

static inline int seq_ge(uint32_t a, uint32_t b) {
    return seq_diff(a, b) >= 0;
}

/* IP address string conversion helpers */
uint32_t ip_str_to_int(const char *str);
void ip_int_to_str(uint32_t ip, char *buf, size_t buf_len);

#endif /* STACK_TYPES_H */
```

D. Core Logic Skeleton Code:

```c
/* include/stack/eth.h - Ethernet and ARP structures */

#ifndef STACK_ETH_H

#define STACK_ETH_H


#include "types.h"


#define ETH_ALEN 6               /* MAC address length */

#define ETH_HDR_LEN 14           /* Ethernet header length without VLAN */

#define ETHERTYPE_IP  0x0800     /* IPv4 protocol */

#define ETHERTYPE_ARP 0x0806     /* ARP protocol */

#define MAX_FRAME_SIZE 1600      /* MTU + Ethernet header */


/* Ethernet header structure */

typedef struct eth_hdr {

    uint8_t dst_mac[ETH_ALEN];

    uint8_t src_mac[ETH_ALEN];

    uint16_t ethertype;

} PACKED eth_hdr_t;


/* ARP header structure (Ethernet + IPv4 specific) */

typedef struct arp_hdr {

    uint16_t hw_type;      /* Hardware type (1 = Ethernet) */

    uint16_t proto_type;   /* Protocol type (0x0800 = IPv4) */

    uint8_t hw_len;        /* Hardware address length (6) */

    uint8_t proto_len;     /* Protocol address length (4) */

    uint16_t opcode;       /* Operation (1 = request, 2 = reply) */

    uint8_t sender_mac[ETH_ALEN];

    uint32_t sender_ip;

    uint8_t target_mac[ETH_ALEN];

    uint32_t target_ip;

} PACKED arp_hdr_t;


/* ARP cache entry */

typedef struct arp_entry {

    uint32_t ip_addr;           /* Key: IP address in network byte order */

    uint8_t mac_addr[ETH_ALEN]; /* Value: MAC address */

    uint8_t state;              /* ARP_INCOMPLETE, ARP_RESOLVED, ARP_STALE */

    uint64_t last_updated;      /* Monotonic timestamp */

    uint8_t retry_count;        /* Number of retries */

} arp_entry_t;


/* Function declarations */

int eth_process_frame(uint8_t *frame, size_t len);
```

```c
int arp_process_packet(arp_hdr_t *arp, size_t len);

int arp_resolve(uint32_t ip, uint8_t *mac_out);

void arp_cache_update(uint32_t ip, uint8_t *mac);


#endif /* STACK_ETH_H */
```

```c
/* include/stack/ip.h - IPv4 structures */

#ifndef STACK_IP_H

#define STACK_IP_H


#include "types.h"


#define IP_HDR_LEN 20           /* Minimum IP header length */

#define IPPROTO_ICMP 1

#define IPPROTO_TCP 6

#define IPPROTO_UDP 17


/* IPv4 header structure */

typedef struct ip_hdr {

    uint8_t version_ihl;        /* Version (4 bits) + IHL (4 bits) */

    uint8_t tos;

    uint16_t total_len;

    uint16_t id;

    uint16_t frag_off;

    uint8_t ttl;

    uint8_t protocol;

    uint16_t checksum;

    uint32_t src_ip;

    uint32_t dst_ip;

    /* Options follow if IHL > 5 */

} PACKED ip_hdr_t;


/* Function to compute IP checksum */

static inline uint16_t ip_checksum(ip_hdr_t *hdr) {

    uint32_t sum = 0;

    uint16_t *words = (uint16_t *)hdr;



    // TODO 1: Sum all 16-bit words in the header (IHL * 2 words)

    // TODO 2: Add carry bits to lower 16 bits

    // TODO 3: Take one's complement of the result

    // TODO 4: Return the checksum (0xFFFF if calculation results in 0)



    return 0; /* Placeholder */

}


/* Routing table entry */

typedef struct route_entry {

    uint32_t dest_net;

    uint32_t dest_mask;
```

```
    uint32_t gateway;
    uint8_t iface_index;
    uint16_t metric;
} route_entry_t;

/* Function declarations */
int ip_process_packet(uint8_t *packet, size_t len);
int ip_send_packet(uint32_t dst_ip, uint8_t protocol, uint8_t *payload, size_t payload_len);
route_entry_t *route_lookup(uint32_t dst_ip);

#endif /* STACK_IP_H */
```

```c
/* include/stack/tcp.h - TCP structures */

#ifndef STACK_TCP_H

#define STACK_TCP_H


#include "types.h"

#include "buffer.h"


#define TCP_HDR_LEN 20           /* Minimum TCP header length */


/* TCP header structure */

typedef struct tcp_hdr {

    uint16_t src_port;

    uint16_t dst_port;

    uint32_t seq_num;

    uint32_t ack_num;

    uint16_t data_offset_reserved_flags;

    uint16_t window;

    uint16_t checksum;

    uint16_t urg_ptr;

    /* Options follow if data_offset > 5 */

} PACKED tcp_hdr_t;


/* TCP flag masks (shifted to correct position in data_offset_reserved_flags) */

#define TCP_FIN (1 << 0)

#define TCP_SYN (1 << 1)

#define TCP_RST (1 << 2)

#define TCP_PSH (1 << 3)

#define TCP_ACK (1 << 4)

#define TCP_URG (1 << 5)


/* TCP connection states */

typedef enum tcp_state {

    TCP_CLOSED = 0,

    TCP_LISTEN,

    TCP_SYN_SENT,

    TCP_SYN_RECEIVED,

    TCP_ESTABLISHED,

    TCP_FIN_WAIT_1,

    TCP_FIN_WAIT_2,

    TCP_CLOSE_WAIT,

    TCP_CLOSING,

    TCP_LAST_ACK,

    TCP_TIME_WAIT
```

```c
} tcp_state_t;

/* TCP Control Block (TCB) */
typedef struct tcb {
    /* Connection identifier */
    uint32_t local_ip;
    uint16_t local_port;
    uint32_t remote_ip;
    uint16_t remote_port;

    /* Connection state */
    tcp_state_t state;

    /* Send variables */
    struct {
        uint32_t nxt;    /* Next sequence to send */
        uint32_t una;    /* Oldest unacknowledged */
        uint16_t wnd;    /* Receive window from peer */
        uint32_t iss;    /* Initial send sequence */
    } send;

    /* Receive variables */
    struct {
        uint32_t nxt;    /* Next sequence expected */
        uint16_t wnd;    /* Receive window to advertise */
        uint32_t irs;    /* Initial receive sequence */
    } recv;

    /* Buffers */
    tcp_buffer_t *rx_buffer;
    tcp_buffer_t *tx_buffer;

    /* Timers */
    struct {
        uint32_t rto;        /* Retransmission timeout (ms) */
        uint64_t rxt_time;   /* When oldest segment was sent */
        uint8_t rxt_count;   /* Retransmission count */
    } timer;

    /* Congestion control */
    struct {
```

```
        uint32_t cwnd;       /* Congestion window */
        uint32_t ssthresh;   /* Slow start threshold */
        uint8_t dupack_count; /* Duplicate ACK count */
    } congestion;

    /* Next TCB in hash chain (for collision resolution) */
    struct tcb *next;
} tcb_t;

/* Function declarations */
int tcp_process_segment(ip_hdr_t *ip, tcp_hdr_t *tcp, uint8_t *payload, size_t payload_len);
tcb_t *tcb_lookup(uint32_t local_ip, uint16_t local_port,
                  uint32_t remote_ip, uint16_t remote_port);
tcb_t *tcp_connect(uint32_t remote_ip, uint16_t remote_port);
int tcp_listen(uint16_t port);
int tcp_send(tcb_t *tcb, uint8_t *data, size_t len);
size_t tcp_receive(tcb_t *tcb, uint8_t *buf, size_t buf_len);

#endif /* STACK_TCP_H */
```

E. Language-Specific Hints:

1. **Byte Ordering:** Always include `#include <arpa/inet.h>` for `htons/ntohs` etc. If you're on Windows, use `#include <winsock2.h>` and `WSAHtons/WSANtohs`.
2. **Structure Packing:** Use `#pragma pack(push, 1)` before structure definitions and `#pragma pack(pop)` after on MSVC. For GCC/Clang, use `__attribute__((packed))` as shown.
3. **Alignment Warnings:** Accessing fields in packed structures may generate unaligned access warnings. You can suppress these with compiler flags or use memcpy() to copy multi-byte fields:

```c
uint32_t get_seq_num(tcp_hdr_t *tcp) {
    uint32_t val;
    memcpy(&val, &tcp->seq_num, sizeof(val));
    return ntohl(val);
}
```

4. **Memory Management:** Allocate TCBs with `calloc()` instead of `malloc()` to ensure all fields start as zero. Remember to free both the TCB and its buffers on connection close.

F. Milestone Checkpoint for Data Structures:

After implementing these structures (but before implementing protocol logic), you should be able to:

1. **Compile without errors:**

```bash
gcc -c -I./include src/net/*.c -Wall -Wextra -Wno-unused-parameter
```

2. **Verify structure sizes match RFC specifications:**

```c
/* In a test program: */
printf("sizeof(eth_hdr_t) = %lu (expected 14)\n", sizeof(eth_hdr_t));
printf("sizeof(ip_hdr_t) = %lu (expected 20)\n", sizeof(ip_hdr_t));
printf("sizeof(tcp_hdr_t) = %lu (expected 20)\n", sizeof(tcp_hdr_t));
printf("sizeof(arp_hdr_t) = %lu (expected 28)\n", sizeof(arp_hdr_t));
```

3. **Test byte order helpers:**

```c
uint16_t test = 0x1234;
printf("htons(0x%04x) = 0x%04x\n", test, htons(test));
assert(htons(test) == 0x3412); /* On little-endian systems */
```

If structures are larger than expected, check packing directives. If byte order functions don't work, check your includes or implement the fallback macros shown above.

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
"Structure size wrong"	Compiler padding inserted	Print `sizeof(struct)` and compare with RFC	Add proper packing attributes
"Checksums never match"	Byte order wrong	Dump raw bytes and compare with Wireshark capture	Use `htons/ntohs` consistently
"TCP sequence numbers behave strangely near 4GB"	Integer overflow in comparisons	Test with sequence numbers near `0xFFFFFFFF`	Use `seq_lt/seq_gt` helper functions
"Buffer corruption"	Circular buffer indices wrap incorrectly	Add assertions after each buffer operation	Ensure modulo arithmetic: `index % size`
"Accessing packed struct fields crashes on ARM"	Unaligned memory access	Check compiler warnings; run on x86 first	Use memcpy() for multi-byte field access

5. Component Design: Link Layer (Milestone 1)

> **Milestone(s):** 1 (Ethernet & ARP)

The **Link Layer** is the foundational component of Project Valiant, acting as the gateway between our software stack and the physical (or virtual) network medium. This layer has three critical responsibilities: interfacing with network hardware via a TAP device, parsing and constructing Ethernet frames, and resolving IP addresses to MAC addresses using the Address Resolution Protocol (ARP).

Responsibility and Scope

The Link Layer component is responsible for:

1. **Raw Frame I/O:** Reading Ethernet frames from and writing frames to a TAP device, providing the illusion of direct access to the network medium.
2. **Frame Processing:** Parsing incoming Ethernet frames to extract the source/destination MAC addresses and EtherType, then constructing outgoing frames with correct headers.
3. **Address Resolution:** Implementing the ARP protocol to map 32-bit IPv4 addresses to 48-bit MAC addresses, maintaining a cache of recent mappings, and responding to ARP requests for our IP address.

The component **does not** handle higher-layer logic (IP routing, TCP connections) nor does it manage physical network characteristics like signal encoding. Its scope is strictly limited to Layer 2 of the OSI model.

Mental Model: The Neighborhood Mail Carrier

> Imagine you're sending a letter within your neighborhood. You write the recipient's **house number (IP address)** on the envelope, but the mail carrier needs the actual **street address (MAC address)** to deliver it. ARP is like asking your neighbors, "Who lives at house number 192.168.1.5?" The resident responds with their street address, which you note down for future letters. The Ethernet frame is the physical envelope itself, with the sender's and recipient's street addresses written on the outside.

This analogy clarifies the distinction between logical (IP) and physical (MAC) addressing, and why ARP is necessary before any IP communication can occur on a local network segment.

Interface

The Link Layer exposes the following key functions to upper layers:

Method Name	Parameters	Returns	Description
`netif_rx_packet`	`uint8_t* frame`, `size_t len`	`void`	Main entry point for processing an incoming Ethernet frame from the TAP device. Dispatches based on EtherType.
`netif_tx_packet`	`uint8_t* frame`, `size_t len`	`int` (0 on success, -1 on error)	Transmits a complete Ethernet frame to the TAP device. Called by upper layers with a fully-formed frame.
`arp_resolve`	`uint32_t ip_addr`	`int` (0 on success, -1 on error)	Initiates ARP resolution for the given IP address. Returns immediately; resolution is asynchronous. Checks cache first, sends ARP request if needed.
`arp_cache_lookup`	`uint32_t ip_addr`, `uint8_t* mac_out`	`int` (1 if found, 0 if not)	Synchronously checks the ARP cache for an existing IP→MAC mapping. Copies MAC to `mac_out` if found.
`arp_cache_insert`	`uint32_t ip_addr`, `uint8_t* mac_addr`	`void`	Inserts or updates an entry in the ARP cache with the provided mapping and current timestamp.
`arp_cache_remove`	`uint32_t ip_addr`	`void`	Removes the entry for the given IP address from the ARP cache (e.g., on timeout or explicit invalidation).
`eth_build_frame`	`uint8_t* dst_mac`, `uint16_t ethertype`, `uint8_t* payload`, `size_t payload_len`, `uint8_t* frame_out`	`size_t` (total frame length)	Constructs a complete Ethernet frame: writes header (destination MAC, source MAC, EtherType) and copies payload. Returns total size for transmission.

> The **separation of concerns** is critical: upper layers (IP) should not need to understand Ethernet framing details. They call `netif_tx_packet` with a payload and destination IP, and the Link Layer handles ARP resolution and frame construction internally.

Internal Behavior

Processing an Incoming Ethernet Frame

When a frame arrives from the TAP device, the following sequence occurs:

1. **Frame Validation:** Check that the received length is at least `ETH_HDR_LEN` (14 bytes) and not exceeding `MAX_FRAME_SIZE`. Drop the frame silently if invalid.
2. **Header Parsing:** Cast the frame buffer to an `eth_hdr` pointer. Extract the destination MAC, source MAC, and EtherType fields. Note: EtherType is stored in **network byte order** (big-endian).
3. **Destination Check:** Compare the destination MAC against:
   - Our interface's MAC address (unicast to us)
   - The broadcast MAC address (`FF:FF:FF:FF:FF:FF`)
   - (Optional) Multicast addresses if supported If none match, silently drop the frame—we are not the intended recipient.
4. **EtherType Dispatch:** Based on the EtherType field:
   - If `ETHERTYPE_ARP` (0x0806): Pass the frame payload (after Ethernet header) to `arp_process_packet`.
   - If `ETHERTYPE_IP` (0x0800): Pass the frame payload to the Network Layer's `ipv4_input` function.
   - Otherwise: Drop the frame (unknown protocol).

> This **dispatch logic** is the bridge between layers. The Link Layer acts as a traffic director, routing frames to the appropriate protocol handler based on the EtherType.

Handling an ARP Request/Reply

The ARP subsystem operates as follows:

Receiving an ARP Request (Somebody asks for our MAC):

1. **Parse ARP Header:** Validate the ARP packet length and cast to `arp_hdr`. Confirm it's for Ethernet (`hw_type = 1`) and IPv4 (`proto_type = ETHERTYPE_IP`).

2. **Target IP Check:** Compare `target_ip` against our configured interface IP address. If it doesn't match, ignore the packet (not for us).
3. **Cache Update:** Insert the sender's `(sender_ip, sender_mac)` mapping into the ARP cache, updating the timestamp. This is **gratuitous ARP learning**—we learn mappings even from requests not directed to us.
4. **Send ARP Reply:** Construct an ARP reply packet:
   - Swap sender/target fields: set `sender_mac` to our MAC, `sender_ip` to our IP.
   - Set `target_mac` to the requester's MAC (from their request), `target_ip` to their IP.
   - Set `opcode` to `ARP_OP_REPLY` (2).
   - Encapsulate in an Ethernet frame with the requester's MAC as destination and `ETHERTYPE_ARP`.
   - Transmit via `netif_tx_packet`.

Receiving an ARP Reply (Response to our earlier request):

1. **Parse ARP Header:** Same validation as above.
2. **Cache Update:** Insert the mapping `(sender_ip, sender_mac)` into the ARP cache. This resolves any pending resolution for that IP.
3. **Wake Waiting Senders:** If any upper-layer packets were queued waiting for this ARP resolution, they can now be sent with the known MAC address.

Initiating ARP Resolution (We need someone's MAC):

1. **Cache Lookup:** Check if the IP is already in the ARP cache with a valid (non-expired) entry. If found, return success immediately.
2. **Queue Pending Packet:** If no cache entry exists, store the packet (from the upper layer) in a **pending queue** associated with the target IP. This prevents flooding the network with duplicate ARP requests.
3. **Send ARP Request:** Construct an ARP request:
   - Set `sender_mac` to our MAC, `sender_ip` to our IP.
   - Set `target_mac` to the broadcast MAC (`00:00:00:00:00:00`), `target_ip` to the IP we're resolving.
   - Set `opcode` to `ARP_OP_REQUEST` (1).
   - Encapsulate in an Ethernet frame with broadcast destination and `ETHERTYPE_ARP`.
   - Transmit via `netif_tx_packet`.
4. **Start Retransmission Timer:** If no reply is received within a timeout (typically 1 second), retransmit the ARP request up to a maximum number of attempts (e.g., 3). After exhausting retries, remove the pending queue and notify upper layers of failure.

ADR: Choosing TAP over Raw Sockets

> **Decision: Use TAP Device for Link Layer Abstraction**
>
> - **Context:** We need a way to send and receive raw Ethernet frames from userspace without kernel protocol processing. The project requires isolation from the host's network stack to avoid interference.
> - **Options Considered:**
>   1. **Raw Sockets** (`AF_PACKET` on Linux, `BPF` on BSD): Direct access to network interface, requiring root privileges and careful filtering to avoid processing packets intended for the host stack.
>   2. **TAP Device:** Virtual network kernel interface that presents as a character device, providing clean Ethernet frame boundaries and complete isolation from physical interfaces.
> - **Decision:** Use a TAP device as the primary network interface for the stack.
> - **Rationale:** TAP devices offer a simpler, more controlled environment. They appear as a standard file descriptor, making I/O straightforward with `read()` / `write()` calls. More importantly, they are completely isolated—frames sent to the TAP device don't leak to the physical network unless explicitly routed, which is ideal for testing and learning. Raw sockets, while powerful, require complex filtering to avoid processing the host's own traffic and present privilege management challenges.
> - **Consequences:** The stack will be limited to virtual network environments unless bridged to a physical interface. However, this simplifies development and testing, as we can create self-contained virtual networks between multiple TAP instances. Performance may be lower than raw sockets but remains acceptable for educational purposes.

Option	Pros	Cons	Chosen?
Raw Sockets	Direct hardware access; can use physical interfaces; potentially higher performance	Requires root privileges; complex filtering needed; host stack interference risk; platform-specific APIs	No
TAP Device	Userspace-friendly file descriptor API; complete isolation from host stack; consistent cross-platform behavior (Linux, BSD, macOS via tun/tap)	Virtual-only; requires bridging to physical network; slightly lower performance due to kernel/userspace copying	**Yes**

Common Pitfalls

⚠️ Pitfall: Byte-Order Confusion in EtherType and ARP Fields

- **Description:** Forgetting to convert multi-byte fields (EtherType, ARP opcode, IP addresses in ARP payload) between host and network byte order.
- **Why It's Wrong:** Network protocols use **big-endian** (network byte order), while most modern CPUs are little-endian. Incorrect byte ordering causes frames to be misinterpreted—an EtherType of 0x0800 (IPv4) stored as 0x0008 in host order will be sent as 0x0008, causing the receiver to treat it as an unknown protocol.
- **Fix:** Use `htons()` and `ntohs()` for 16-bit fields, `htonl()` and `ntohl()` for 32-bit fields when reading from/writing to headers. For the `eth_hdr.ethertype` field, store and compare values using the `ETHERTYPE_*` constants (which are defined in network byte order).

⚠️ Pitfall: ARP Cache Poisoning Vulnerability

- **Description:** Blindly accepting any ARP reply and updating the cache without validation, allowing an attacker to redirect traffic.
- **Why It's Wrong:** In a production environment, this enables **ARP spoofing** attacks where a malicious host claims to own another IP address, intercepting traffic. While our educational stack isn't focused on security, understanding the vulnerability is important.
- **Fix:** Implement simple validation: only update the cache for an IP if we previously sent an ARP request for that IP (pending queue exists). For educational purposes, we may accept gratuitous ARP updates but should log them.

⚠️ Pitfall: Mishandling Broadcast Frames

- **Description:** Failing to properly identify and handle broadcast MAC addresses (`FF:FF:FF:FF:FF:FF`) in destination checks.
- **Why It's Wrong:** ARP requests are sent to the broadcast address. If we drop frames not explicitly addressed to our MAC, we'll never receive ARP requests, breaking address resolution entirely.
- **Fix:** In the destination check step, always accept frames where the destination MAC equals the broadcast address. Additionally, when sending ARP requests, use the broadcast MAC as the destination.

⚠️ Pitfall: Not Caching ARP Replies from Requests Not Directed to Us

- **Description:** Ignoring ARP requests where the `target_ip` isn't our IP address, missing an opportunity to learn mappings.
- **Why It's Wrong:** When Host A asks for Host B's MAC, we (Host C) overhear the request and learn Host A's IP→MAC mapping from the request's `sender` fields. Not caching this wastes future resolution time.
- **Fix:** Implement **gratuitous ARP learning**: whenever we see any valid ARP packet (request or reply), update our cache with the `sender_ip` → `sender_mac` mapping.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
TAP Device Interface	Use `open()` , `read()` , `write()` system calls with `/dev/net/tun`	Use `libpcap` for portable packet capture (though heavier)
ARP Cache	Simple hash table with linear probing, periodic cleanup via timer	LRU cache with aging, integrated with timer wheel for precise expiration
Byte Ordering	Manual `htons` / `ntohs` calls	Use compiler attributes (`__attribute__((packed))`) for structs and byte-order conversion macros

B. Recommended File/Module Structure

```
project-valiant/
├── include/
│   ├── net/
│   │   ├── eth.h          ← Ethernet header struct and constants
│   │   ├── arp.h          ← ARP header struct, cache, and functions
│   │   └── netif.h        ← TAP interface and frame I/O functions
├── src/
│   ├── net/
│   │   ├── eth.c          ← Ethernet frame parsing/construction
│   │   ├── arp.c          ← ARP protocol logic and cache management
│   │   ├── netif.c        ← TAP device setup and I/O loop
│   │   └── packet.c       ← Shared packet buffer utilities
│   └── main.c             ← Main event loop and initialization
└── tests/
    └── net/
        ├── test_eth.c
        └── test_arp.c
```

C. Infrastructure Starter Code

Complete TAP device setup (`src/net/netif.c`):

```c
#include <fcntl.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include "net/netif.h"

#define TAP_DEVICE "/dev/net/tun"

int tap_open(const char* dev_name) {
    struct ifreq ifr;
    int fd;

    // Open the TAP device file
    if ((fd = open(TAP_DEVICE, O_RDWR)) < 0) {
        perror("Failed to open " TAP_DEVICE);
        return -1;
    }

    memset(&ifr, 0, sizeof(ifr));
    ifr.ifr_flags = IFF_TAP | IFF_NO_PI;  // TAP device, no packet info
    strncpy(ifr.ifr_name, dev_name, IFNAMSIZ);

    // Create the device
    if (ioctl(fd, TUNSETIFF, (void*)&ifr) < 0) {
        perror("Failed to create TAP device");
        close(fd);
        return -1;
    }

    // Bring interface up
    int sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        close(fd);
        return -1;
    }
    ifr.ifr_flags |= IFF_UP | IFF_RUNNING;
    if (ioctl(sock, SIOCSIFFLAGS, &ifr) < 0) {
        perror("Failed to bring interface up");
        close(sock);
        close(fd);
        return -1;
```

```
    }

    close(sock);

    return fd;  // File descriptor for reading/writing frames
}

ssize_t tap_read(int fd, uint8_t* buf, size_t len) {
    return read(fd, buf, len);
}

ssize_t tap_write(int fd, uint8_t* buf, size_t len) {
    return write(fd, buf, len);
}
```

D. Core Logic Skeleton Code

Ethernet frame parser (`src/net/eth.c`):

```c
#include "net/eth.h"
#include "net/arp.h"
#include "net/ipv4.h"
#include <string.h>

void netif_rx_packet(uint8_t* frame, size_t len) {
    // TODO 1: Validate minimum frame length (at least ETH_HDR_LEN)
    // TODO 2: Cast frame to eth_hdr* pointer
    // TODO 3: Extract destination MAC, compare against our MAC and broadcast
    //         If neither matches, silently return (drop frame)
    // TODO 4: Extract EtherType (convert from network byte order with ntohs)
    // TODO 5: Dispatch based on EtherType:
    //   - If ETHERTYPE_ARP: call arp_process_packet(frame + ETH_HDR_LEN, len - ETH_HDR_LEN)
    //   - If ETHERTYPE_IP: call ipv4_input(frame + ETH_HDR_LEN, len - ETH_HDR_LEN)
    //   - Otherwise: drop (unknown protocol)
}

size_t eth_build_frame(uint8_t* dst_mac, uint16_t ethertype,
                       uint8_t* payload, size_t payload_len,
                       uint8_t* frame_out) {
    // TODO 1: Cast frame_out to eth_hdr* pointer
    // TODO 2: Copy dst_mac to eth_hdr.dst_mac[6]
    // TODO 3: Copy our interface MAC to eth_hdr.src_mac[6]
    // TODO 4: Set eth_hdr.ethertype to ethertype (convert to network byte order with htons)
    // TODO 5: Copy payload data after the Ethernet header (frame_out + ETH_HDR_LEN)
    // TODO 6: Return total frame length: ETH_HDR_LEN + payload_len
}
```

ARP cache and processor (`src/net/arp.c`):

```c
#include "net/arp.h"

#include "net/netif.h"

#include <string.h>

#include <time.h>


#define ARP_CACHE_SIZE 64

#define ARP_ENTRY_TIMEOUT_MS 30000  // 30 seconds


static arp_entry arp_cache[ARP_CACHE_SIZE];


void arp_cache_init(void) {

    memset(arp_cache, 0, sizeof(arp_cache));

}


int arp_cache_lookup(uint32_t ip_addr, uint8_t* mac_out) {

    // TODO 1: Get current timestamp (use clock_gettime or similar)

    // TODO 2: Iterate through arp_cache array

    // TODO 3: For each entry with state != ARP_ENTRY_FREE:

    //   - Compare entry.ip_addr with ip_addr

    //   - Check if entry is not expired (current_time - entry.last_updated < ARP_ENTRY_TIMEOUT_MS)

    //   - If found and valid, copy entry.mac_addr to mac_out and return 1

    // TODO 4: If not found, return 0

}


void arp_cache_insert(uint32_t ip_addr, uint8_t* mac_addr) {

    // TODO 1: Find existing entry for this IP or first free entry (state == ARP_ENTRY_FREE)

    // TODO 2: Update entry.ip_addr, copy mac_addr to entry.mac_addr

    // TODO 3: Set entry.state = ARP_ENTRY_RESOLVED

    // TODO 4: Update entry.last_updated to current timestamp

}


void arp_process_packet(uint8_t* packet, size_t len) {

    // TODO 1: Validate minimum ARP packet length (sizeof(arp_hdr))

    // TODO 2: Cast packet to arp_hdr* pointer

    // TODO 3: Verify hardware type is Ethernet (1) and protocol type is IPv4 (ETHERTYPE_IP)

    // TODO 4: Extract opcode (convert from network byte order with ntohs)

    // TODO 5: Always update cache with sender's mapping (arp_cache_insert)

    // TODO 6: If opcode is ARP_OP_REQUEST and target_ip is our interface IP:

    //   - Build ARP reply (swap sender/target, set our MAC/IP as sender)

    //   - Build Ethernet frame with requester's MAC as destination, ETHERTYPE_ARP

    //   - Transmit via netif_tx_packet

}
```

E. Language-Specific Hints

- Use `#pragma pack(push, 1)` and `#pragma pack(pop)` around your protocol struct definitions to ensure correct field alignment without padding bytes.
- For byte-order conversion, include `<arpa/inet.h>` for `htons`, `ntohs`, `htonl`, `ntohl`.
- Store MAC addresses as `uint8_t[6]` arrays. Use `memcmp` for comparison, `memcpy` for copying.
- When reading from the TAP device, use a static buffer sized to `MAX_FRAME_SIZE` (1600 bytes).

F. Milestone 1 Checkpoint

Expected Behavior: The stack can receive Ethernet frames, parse ARP requests, and send ARP replies. Another host on the same virtual network (e.g., another TAP device or the host itself) should be able to ARP ping the stack's IP address and receive a valid reply.

Verification Commands:

1. **Setup TAP device and assign IP:**

```bash
sudo ip tuntap add mode tap user $USER name tap0

sudo ip addr add 192.168.1.100/24 dev tap0

sudo ip link set tap0 up
```

2. **Run the stack with our TAP device:**

```bash
./valiant --interface tap0 --ip 192.168.1.100
```

3. **From another terminal, send an ARP request:**

```bash
sudo arping -I tap0 192.168.1.100
```

4. **Expected Output:** The stack should log "Received ARP request for our IP" and "Sending ARP reply". The `arping` command should receive a reply showing the stack's MAC address.

Signs of Trouble:

- **"No response received"** from arping: Check that frames are being read from the TAP device (add logging to `tap_read`). Verify destination MAC checking includes broadcast address.
- **Incorrect MAC in reply:** Check byte ordering in ARP header fields. Use Wireshark to capture packets on `tap0` and inspect the raw bytes.
- **Stack crashes on receiving frame:** Validate buffer lengths before casting to structs. Ensure protocol structs are packed correctly (no padding).

Debugging Tip: Enable verbose logging of every frame sent/received in hex format. Compare against Wireshark captures to identify byte-order or field placement errors.

---

6. Component Design: Network Layer (Milestone 2)

> **Milestone(s):** 2 (IP & ICMP)

The **Network Layer** is the internetwork routing and delivery system of Project Valiant. It sits directly above the Link Layer, receiving its payloads (Ethernet frames) and responsible for delivering them across potentially multiple networks to their final destination. This layer introduces logical **IP addresses**, which are hierarchical and routable, unlike the flat MAC addresses of the link layer. Its primary responsibilities are packet forwarding, error reporting via ICMP, and making routing decisions.

Responsibility and Scope

This component's domain is defined by the IPv4 protocol (RFC 791) and the Internet Control Message Protocol (ICMP, RFC 792). Its core responsibilities are:

1. **Packet Validation & Parsing:** Receiving IPv4 packets from the Link Layer, verifying their integrity (checksum), and extracting header fields and payload for further processing.
2. **Packet Construction & Forwarding:** Building valid IPv4 headers for outbound data from upper layers, computing the correct checksum, and determining the next hop (either the final destination or a gateway router) before handing the packet down to the Link Layer for transmission.
3. **Error Reporting & Diagnostics:** Implementing ICMP to handle network-level error conditions (e.g., unreachable host, time exceeded) and to respond to diagnostic requests (echo request/ping).
4. **Routing Decisions:** Maintaining a simple routing table that maps destination IP address ranges to the appropriate next-hop IP address and network interface.

Its scope is strictly Layer 3. It does not handle reliability, flow control, or application data streams—those are Transport Layer concerns. It also does not perform complex routing protocols (like OSPF or BGP); the routing table is static and administratively configured.

Mental Model: The Sorting Facility & System Monitor

Imagine the Network Layer as a massive, automated postal sorting facility.

- **IP Packet as a Parcel:** Each parcel (IP packet) has a destination ZIP code (destination IP address) and a return address (source IP address). The sorting facility doesn't care about the parcel's contents (the Transport Layer payload), only the addressing on the label.
- **Routing Table as Sorting Rules:** The facility has a set of rules (the routing table). For local ZIP codes (the local network), parcels are sent directly to the local delivery truck (the Link Layer). For distant ZIP codes, they are sorted onto specific outbound trucks (gateways) that promise to get them closer to their final destination. The **TTL (Time To Live)** field is like a "handling stamp" counter; each time the parcel passes through a sorting facility (router), a stamp is removed. If the counter reaches zero, the parcel is discarded to prevent infinite loops.
- **ICMP as the Facility's PA System:** ICMP is the facility's internal communication and monitoring system. If a parcel is undeliverable (e.g., invalid address, destination closed), the facility uses the PA system (ICMP) to send a message back to the return address explaining the problem. The "ping" (ICMP Echo Request) is like sending a test parcel with a "please sign and return this receipt" slip—it's a way to check if the path to a destination and back is functional.

This mental model separates the *logical routing* (IP) from the *physical delivery* (Ethernet/ARP) and the *reliable conversation* (TCP).

Interface

The Network Layer's API consists of functions called by the layer above (Transport) and below (Link), as well as internal management functions.

Method	Parameters	Returns	Description
`ipv4_input`	`uint8_t *packet_data`, `size_t len`, `uint32_t from_ip`	`void`	**The main ingress point.** Called by the Link Layer (`netif_rx_packet`) when an Ethernet frame with `ETHERTYPE_IP` is received. Validates the IP header, processes options (if any), checks the destination IP, and either forwards the packet or passes the payload to the appropriate upper-layer protocol handler (e.g., `icmp_input` or `tcp_input`).
`ipv4_output`	`uint32_t dst_ip`, `uint8_t protocol`, `uint8_t *payload`, `size_t payload_len`	`int` (0 on success)	**The main egress point.** Called by the Transport Layer (or ICMP) to send data. It performs a route lookup, constructs a complete IP header (with correct checksum), and calls `netif_tx_packet` via the Link Layer to send the frame. Handles fragmentation if the packet size exceeds the MTU (a non-goal for Milestone 2, but the interface should allow for it).
`icmp_input`	`uint8_t *icmp_data`, `size_t len`, `uint32_t src_ip`, `uint32_t dst_ip`	`void`	Called by `ipv4_input` when the IP protocol field is `IPPROTO_ICMP`. Parses the ICMP header and dispatches to specific handlers (e.g., `icmp_echo_reply` for Type 8).
`icmp_echo_reply`	`uint8_t *request_data`, `size_t len`, `uint32_t src_ip`, `uint32_t dst_ip`	`void`	Handles an ICMP Echo Request (ping). It constructs an Echo Reply packet (swapping source/destination IPs, setting Type to 0) and calls `ipv4_output` to send it back.
`route_lookup`	`uint32_t dest_ip`	`struct route_entry*` (or NULL)	Consults the routing table to find the best matching route for the given destination IP. The lookup involves finding the route with the longest prefix match (most specific network). Returns a pointer to the `route_entry` which contains the next-hop gateway IP and the egress interface.
`route_add`	`uint32_t dest_net`, `uint32_t netmask`, `uint32_t gateway`, `uint8_t iface`	`int` (0 on success)	Adds a static route to the routing table. Used for initial configuration (e.g., default route `0.0.0.0/0` via a gateway).
`ip_checksum`	`const void *data`, `size_t len`	`uint16_t`	**Helper function.** Computes the Internet Checksum (one's complement of the one's complement sum) over a block of data. Used for IP and higher-layer checksums.

Internal Behavior

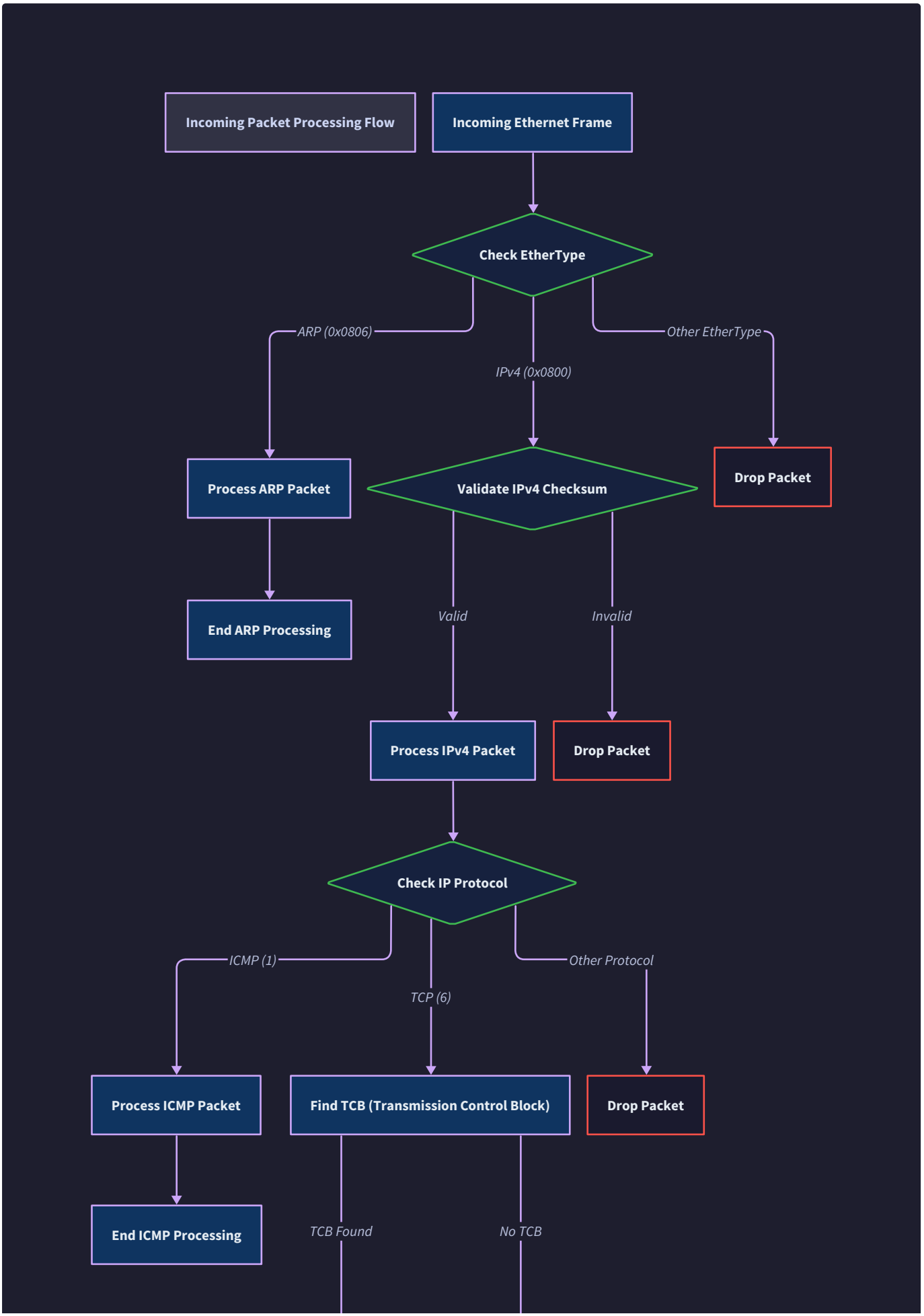
1. Processing an Incoming IP Packet (`ipv4_input`)

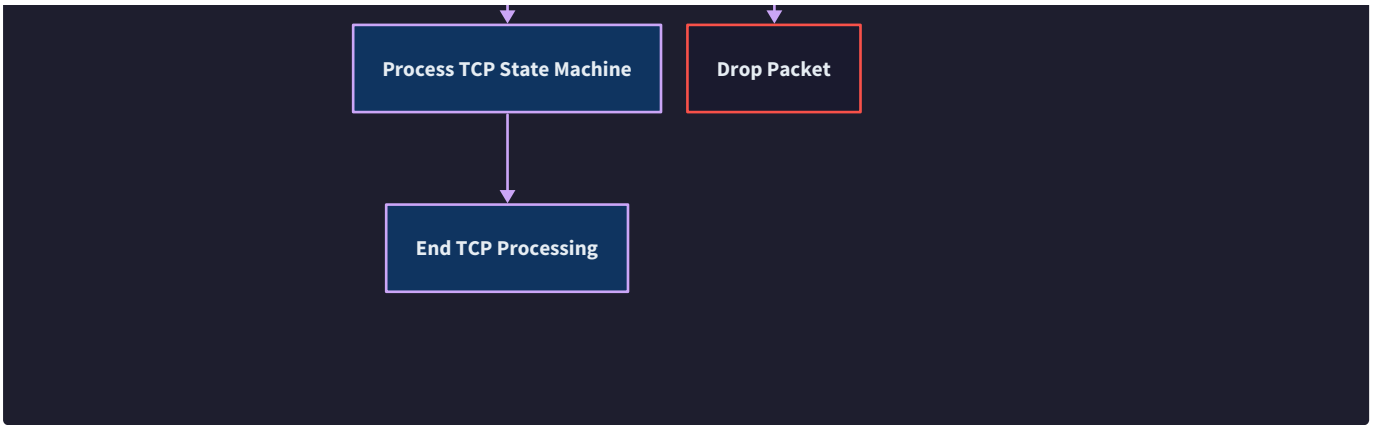
The following steps occur when an IP packet is received from the network:

1. **Length & Version Check:** Verify the packet length is at least `IP_HDR_LEN` (20 bytes) and the IP version field is 4. Drop the packet if not.
2. **Header Length Validation:** Extract the Internet Header Length (IHL) from the `version_ihl` field. Ensure `(IHL * 4) >= IP_HDR_LEN` and that the total packet length reported in the header is not less than the header length. Drop malformed packets.
3. **Checksum Verification:** Calculate the checksum over the IP header (first `IHL * 4` bytes) using `ip_checksum`. The result must be `0xFFFF` (or 0 if using the common comparison trick). Drop the packet if the checksum fails.
4. **Destination Check:** Compare the `dst_ip` field with the IP addresses assigned to the stack's network interfaces and the broadcast address. If it doesn't match any, the packet is not for us. For a *non-goal* but important conceptual step: a router would forward it; our host stack should drop it silently.

5. **Protocol Demultiplexing:** Examine the `protocol` field. If it is `IPPROTO_ICMP`, call `icmp_input`. If it is `IPPROTO_TCP`, call `tcp_input`. For any other protocol (e.g., UDP), drop the packet (or log it as unimplemented).
6. **Payload Delivery:** Pass the pointer to the start of the transport layer payload (byte offset `IHL * 4` into the packet) and its length (`total_len - (IHL * 4)`) to the appropriate upper-layer handler.

This process is visualized in the incoming packet flowchart:


Incoming Packet Processing Flow
Incoming Ethernet Frame
Check EtherType
ARP (0x0806)
IPv4 (0x0800)
Other EtherType
Process ARP Packet
Validate IPv4 Checksum
Drop Packet
End ARP Processing
Valid
Invalid
Process IPv4 Packet
Drop Packet
Check IP Protocol
ICMP (1)
TCP (6)
Other Protocol
Process ICMP Packet
Find TCB (Transmission Control Block)
Drop Packet
End ICMP Processing
TCB Found
No TCB



2. Handling an ICMP Echo Request

When `icmp_input` receives a packet with Type 8 (Echo Request):

1. **Validate ICMP Checksum:** Compute the ICMP checksum over the entire ICMP message (header + data). Drop if invalid.
2. **Construct Reply Header:** Create a new ICMP header. Set Type to 0 (Echo Reply) and Code to 0. Copy the Identifier and Sequence Number fields from the request.
3. **Copy Payload:** Copy the entire data payload from the request.
4. **Compute ICMP Checksum:** Calculate the checksum over the new ICMP header and payload.
5. **Send via IP Layer:** Call `ipv4_output` with the destination IP set to the source IP of the request, protocol `IPPROTO_ICMP`, and the newly built ICMP message as the payload.

3. Routing Table Lookup (`route_lookup`)

The routing table is a list of `route_entry` structures. The lookup algorithm is a **longest prefix match**:

1. **Iterate:** For each entry in the routing table:
    1. Apply the entry's `dest_mask` (netmask) to both the destination IP (`dest_ip`) and the entry's `dest_net` .
    2. If the results are equal, the destination IP is within this route's network.
2. **Select Best Match:** Among all matching entries, select the one with the largest population count (number of `1` bits) in its `dest_mask` . This is the most specific route.
3. **Determine Next Hop:** If the selected entry's `gateway` field is `0.0.0.0` , the next hop is the `dest_ip` itself (direct delivery). Otherwise, the next hop is the `gateway` IP.
4. **Return Result:** Return a pointer to the winning `route_entry` . If no match is found (e.g., no default route), return `NULL` .

ADR: On-the-Wire vs. Recalculated Checksums

> **Decision: Validate Received Checksums, Always Recalculate for Outgoing Packets**
>
> - **Context:** The IPv4 header contains a checksum field intended to detect corruption in the header between routers. RFC 791 states the checksum must be verified by every receiving node and recalculated at every hop because the TTL field changes. Modern hardware often offloads checksum calculation to the NIC. In our educational user-space stack, we control the entire processing path.
> - **Options Considered:**
>   1. **Validate and Recalculate:** Verify the checksum on every received packet. Recompute it from scratch for every outgoing packet.
>   2. **Incremental Update (Optimized):** For forwarded packets where only the TTL is decremented, use an incremental checksum update algorithm instead of a full recalculation.
>   3. **Skip Validation:** Assume lower layers (TAP) or the host kernel provide error-free packets and skip checksum validation for performance.
> - **Decision:** We will implement Option 1: full validation and recalculation.
> - **Rationale:** This is an educational project where correctness, clarity, and understanding the RFC are paramount. Implementing the full checksum algorithm reinforces the protocol specification. Incremental update is an important optimization but adds complexity that distracts from core networking concepts. Skipping validation teaches bad habits and would mask bugs in our own packet construction.
> - **Consequences:** Our stack will be less performant than a production stack for forwarded packets, as we will fully recompute the checksum even for a simple TTL decrement. However, it will be unequivocally correct according to the RFC, and the checksum code will serve as a reusable helper for TCP and ICMP checksums as well.

Option	Pros	Cons	Chosen?
Validate and Recalculate	Simple, clear, RFC-compliant, reusable helper function.	Slightly less efficient for packet forwarding.	**YES**
Incremental Update	More efficient for routers, teaches an important optimization.	More complex algorithm, obscures the basic checksum concept.	No
Skip Validation	Fastest, simplifies code.	Non-compliant, hides bugs, poor educational value.	No

Common Pitfalls

⚡ Pitfall: Incorrect Checksum Calculation

- **Description:** The Internet Checksum algorithm is often implemented incorrectly—forgetting one's complement, mishandling odd-length data, or incorrectly swapping bytes.
- **Why it's Wrong:** A wrong checksum will cause valid packets to be dropped by receivers or cause receivers to accept corrupted packets. Other hosts on the network will silently discard our packets.
- **Fix:** Write a dedicated, well-tested `ip_checksum` function. Use known test vectors from RFCs (e.g., RFC 1071) to verify it. Use this same function for ICMP and TCP checksums.

⚡ Pitfall: Forgetting to Decrement TTL

- **Description:** When forwarding a packet (or even when sending a generated packet like an ICMP reply), failing to decrement the Time-To-Live (TTL) field before transmission.
- **Why it's Wrong:** The TTL must be decremented by at least one at every IP layer hop. Not doing so violates RFC 791 and can cause packets to loop forever in the presence of routing loops. It also breaks `traceroute`.
- **Fix:** Always set the TTL field in the `ip_hdr` when constructing an outgoing packet. For a host stack, a common initial value is 64. For a forwarded packet, you must decrement the received TTL. If the TTL reaches 0, you must discard the packet and may send an ICMP Time Exceeded message.

⚡ Pitfall: Ignoring IP Fragmentation

- **Description:** Assuming all IP packets will be smaller than the MTU and ignoring the fragmentation fields (`id` , `frag_off` , `flags`).
- **Why it's Wrong:** While we define handling fragmentation as a non-goal, completely ignoring the fields can lead to parsing errors. A real stack must at least recognize a fragmented packet and know it cannot deliver it to a higher-layer protocol that doesn't support reassembly (like TCP, which uses Path MTU Discovery).
- **Fix:** Read the `frag_off` field. If the "More Fragments" (MF) flag is set or the "Fragment Offset" is non-zero, the packet is a fragment. For Milestone 2, you can log and drop fragmented packets. A more complete implementation would include a reassembly queue.

⚡ Pitfall: Incorrect Byte Order in IP Addresses

- **Description:** Storing or comparing IP addresses from the packet (`ip_hdr.src_ip` , `ip_hdr.dst_ip`) in host byte order (little-endian on x86) instead of treating them as network byte order (big-endian) integers.
- **Why it's Wrong:** The routing table lookup and comparison with local interface addresses will produce incorrect matches, breaking all communication.
- **Fix:** Consistently store IP addresses in `uint32_t` variables in **network byte order**. Use `ntohl()` to convert to host order only for display or arithmetic (like adding to a subnet). Use `htonl()` to convert back. Our helper functions `ip_str_to_int` and `ip_int_to_str` assume network byte order.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option	Advanced Option (Out of Scope)
IP Packet Processing	Linear scan routing table, full checksum recalculation.	Hash-table for routes, incremental checksum updates.
ICMP	Handle only Echo Request/Reply (ping).	Implement Destination Unreachable, Time Exceeded, Redirect messages.
Routing	Static routing table configured at startup.	Dynamic routing via a user-space daemon (e.g., implementing RIP).

B. Recommended File/Module Structure

Add the following files for the Network Layer:

```
project_valiant/
├── include/
│   ├── stack/
│   │   ├── netif.h      # Link Layer interface (existing)
│   │   ├── ip.h         # IP and ICMP public API
│   │   ├── route.h      # Routing table API
│   │   └── types.h      # Common types (ip_hdr, icmp_hdr, etc.)
├── src/
│   ├── link/            # Milestone 1 code
│   ├── net/
│   │   ├── ip.c         # Implementation of ipv4_input, ipv4_output
│   │   ├── icmp.c       # Implementation of icmp_input, icmp_echo_reply
│   │   ├── route.c      # Implementation of route_lookup, route_add
│   │   └── checksum.c   # Implementation of ip_checksum helper
│   └── main.c
```

C. Infrastructure Starter Code

`include/stack/types.h` (Additions)

```c
#ifndef STACK_TYPES_H
#define STACK_TYPES_H

#include <stdint.h>

// IPv4 Header (20 bytes + options)
struct ip_hdr {
    uint8_t  version_ihl;    // Version (4 bits) + IHL (4 bits)
    uint8_t  tos;            // Type of Service
    uint16_t total_len;      // Total length (header + data)
    uint16_t id;             // Identification
    uint16_t frag_off;       // Flags (3 bits) + Fragment Offset (13 bits)
    uint8_t  ttl;            // Time to Live
    uint8_t  protocol;       // Protocol (ICMP=1, TCP=6, UDP=17)
    uint16_t checksum;       // Header checksum
    uint32_t src_ip;         // Source address (network byte order)
    uint32_t dst_ip;         // Destination address (network byte order)
} __attribute__((packed));

// ICMP Header (Base 8 bytes + variable data)
struct icmp_hdr {
    uint8_t  type;           // ICMP message type (8=Echo Request, 0=Echo Reply)
    uint8_t  code;           // Sub-code (0 for echo)
    uint16_t checksum;       // ICMP checksum (covers header+data)
    uint16_t identifier;     // Echo identifier
    uint16_t sequence;       // Echo sequence number
    // Variable data follows...
} __attribute__((packed));

// Routing Table Entry
struct route_entry {
    uint32_t dest_net;       // Network address (network byte order)
    uint32_t dest_mask;      // Network mask (network byte order)
    uint32_t gateway;        // Next-hop gateway (0.0.0.0 for direct)
    uint8_t  iface_index;    // Index of the network interface to use
    uint16_t metric;         // Administrative distance/metric (lower is better)
    struct route_entry *next;
};

#endif // STACK_TYPES_H
```

src/net/checksum.c

```c
#include <stddef.h>
#include <stdint.h>
#include "stack/types.h"

// Compute Internet Checksum (RFC 1071) for 'len' bytes starting at 'data'.
// Returns the checksum in network byte order.
uint16_t ip_checksum(const void *data, size_t len) {
    const uint16_t *words = (const uint16_t *)data;
    uint32_t sum = 0;

    // Sum all 16-bit words
    for (size_t i = 0; i < len / 2; i++) {
        sum += ntohs(words[i]); // Convert from network to host order for addition
    }

    // Handle odd byte if present
    if (len % 2) {
        uint16_t last_word = 0;
        uint8_t *last_byte = (uint8_t *)data + len - 1;
        last_word = *last_byte << 8; // Pad the last byte to make a 16-bit word
        sum += last_word;
    }

    // Fold 32-bit sum to 16 bits, adding carries
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }

    // Return one's complement in network byte order
    return htons(~(uint16_t)sum);
}
```

D. Core Logic Skeleton Code

`src/net/ip.c`

```c
#include "stack/ip.h"

#include "stack/netif.h"

#include "stack/route.h"

#include "stack/types.h"

#include <string.h>

#include <stdio.h>


// Our local IP address (configured)

static uint32_t g_local_ip;


void ipv4_input(uint8_t *packet_data, size_t len, uint32_t from_ip) {

    // TODO 1: Verify minimum length (>= IP_HDR_LEN). Drop if too short.

    // TODO 2: Cast packet_data to `struct ip_hdr*` for easier access.

    // TODO 3: Verify IP version is 4. Drop if not.

    // TODO 4: Calculate IHL (header length in bytes) = (hdr->version_ihl & 0x0F) * 4.

    // TODO 5: Verify packet total length (from header) matches received `len`. Drop if mismatch.

    // TODO 6: Validate IP header checksum using `ip_checksum`. Drop if invalid.

    // TODO 7: Check destination IP (hdr->dst_ip). If not our local_ip and not broadcast, drop.

    // TODO 8: Demultiplex based on hdr->protocol:

    //     - If IPPROTO_ICMP: call icmp_input(packet_data + IHL, len - IHL, hdr->src_ip, hdr->dst_ip)

    //     - If IPPROTO_TCP: call tcp_input(...) (to be implemented in Milestone 3)

    //     - Else: drop/log "Unhandled protocol"

}


int ipv4_output(uint32_t dst_ip, uint8_t protocol, uint8_t *payload, size_t payload_len) {

    // TODO 1: Perform route lookup for dst_ip via `route_lookup`. If none, return -1 (no route).

    // TODO 2: Determine next-hop IP: if route->gateway != 0, use it; else use dst_ip.

    // TODO 3: Allocate buffer for full packet: header + payload.

    // TODO 4: Build IP header (`struct ip_hdr`):

    //     - version_ihl = (4 << 4) | (IP_HDR_LEN / 4)   // Version=4, IHL=5 (20 bytes)

    //     - tos = 0

    //     - total_len = htons(IP_HDR_LEN + payload_len)

    //     - id = htons(unique_id++)  // Simple global counter

    //     - frag_off = 0  // No fragmentation

    //     - ttl = 64

    //     - protocol = protocol

    //     - checksum = 0  // Placeholder

    //     - src_ip = g_local_ip

    //     - dst_ip = dst_ip (original destination, not next-hop)

    // TODO 5: Compute checksum over the header using `ip_checksum`, store in hdr->checksum.

    // TODO 6: Copy payload data after the header.

    // TODO 7: Resolve next-hop IP to MAC address using `arp_resolve`/`arp_cache_lookup`.
```

```
    // TODO 8: Build Ethernet frame with `eth_build_frame` (dest MAC from ARP, EtherType=ETHERTYPE_IP).
    // TODO 9: Transmit frame using `netif_tx_packet`.
    // TODO 10: Return 0 on success, -1 on any failure (e.g., ARP resolution failed).
    return 0;
}
```

src/net/icmp.c

```c
#include "stack/ip.h"
#include "stack/types.h"
#include <string.h>

void icmp_input(uint8_t *icmp_data, size_t len, uint32_t src_ip, uint32_t dst_ip) {
    // TODO 1: Cast icmp_data to `struct icmp_hdr*`.
    // TODO 2: Verify length >= sizeof(struct icmp_hdr).
    // TODO 3: Compute ICMP checksum over entire `len` bytes. Drop if invalid.
    // TODO 4: Switch on hdr->type:
    //     - Case 8 (ECHO REQUEST): call icmp_echo_reply(icmp_data, len, src_ip, dst_ip)
    //     - Default: drop/log "Unhandled ICMP type"
}

void icmp_echo_reply(uint8_t *request_data, size_t len, uint32_t src_ip, uint32_t dst_ip) {
    // Allocate buffer for reply (same size as request)
    uint8_t reply_buf[MAX_FRAME_SIZE];
    struct icmp_hdr *req_hdr = (struct icmp_hdr *)request_data;
    struct icmp_hdr *rep_hdr = (struct icmp_hdr *)reply_buf;

    // TODO 1: Set reply header fields:
    //     - type = 0 (ECHO REPLY)
    //     - code = 0
    //     - identifier = req_hdr->identifier
    //     - sequence = req_hdr->sequence
    // TODO 2: Copy the ICMP data payload (the part after the 8-byte header).
    //         Hint: The data starts at (request_data + 8), length is (len - 8).
    // TODO 3: Calculate ICMP checksum over the entire reply message (header + data).
    // TODO 4: Send the reply via `ipv4_output`:
    //     - dst_ip = src_ip (the original sender)
    //     - protocol = IPPROTO_ICMP
    //     - payload = reply_buf
    //     - payload_len = len
}
```

src/net/route.c

```c
#include "stack/route.h"

#include "stack/types.h"

#include <stdlib.h>


static struct route_entry *g_route_table = NULL;


struct route_entry* route_lookup(uint32_t dest_ip) {

    struct route_entry *best_route = NULL;

    uint32_t best_mask = 0;


    // TODO 1: Iterate through linked list `g_route_table`.

    // TODO 2: For each entry, check if dest_ip is within the network:

    //         if ((dest_ip & entry->dest_mask) == entry->dest_net)

    // TODO 3: If it matches, compare the mask (entry->dest_mask) with `best_mask`.

    //         A larger mask (more 1 bits) means a more specific route.

    // TODO 4: Keep track of the most specific matching route.

    // TODO 5: After loop, return `best_route` (could be NULL if no match).

    return best_route;

}


int route_add(uint32_t dest_net, uint32_t netmask, uint32_t gateway, uint8_t iface) {

    // TODO 1: Allocate a new `struct route_entry`.

    // TODO 2: Populate its fields with the provided arguments.

    // TODO 3: Set metric to a default (e.g., 1).

    // TODO 4: Insert at the head of the linked list `g_route_table`.

    // TODO 5: Return 0 on success, -1 on allocation failure.

    return 0;

}
```

E. Language-Specific Hints (C)

- **Network Byte Order:** Always use `ntohl()`, `htonl()`, `ntohs()`, `htons()` when reading/writing multi-byte integer fields from/to the wire. The fields in `struct ip_hdr` are defined as they appear on the wire (network order).
- **Struct Packing:** Use `__attribute__((packed))` (GCC/Clang) or `#pragma pack` (MSVC) for all header structures to prevent the compiler from inserting padding bytes, which would break the alignment expected by the protocol.
- **Checksum Function:** The `ip_checksum` function works on any byte-aligned data. For odd lengths, the RFC specifies padding the last byte with a zero byte on the right (most significant side) to make a 16-bit word. Our implementation does this by shifting the last byte left by 8 bits.
- **Pointer Arithmetic:** When parsing packets, be careful with pointer offsets. `(uint8_t*)hdr + (IHL * 4)` correctly points to the transport payload.

F. Milestone 2 Checkpoint

After implementing the Network Layer, you should be able to respond to pings.

1. **Configure the Stack:** In your `main.c`, set the local IP address and add a default route.

```c
    g_local_ip = ip_str_to_int("192.168.1.100");

    route_add(ip_str_to_int("192.168.1.0"), ip_str_to_int("255.255.255.0"), 0, 0); // Local network

    route_add(0, 0, ip_str_to_int("192.168.1.1"), 0); // Default route via gateway
```

2. **Test with Ping:**

- Start your stack.
- From another machine on the same network (or from the host OS if using a TAP device bridged to your host), ping your stack's IP address.

```
    ping 192.168.1.100
```

3. **Expected Result:** You should see successful ping replies. Use Wireshark/tcpdump on the TAP interface to verify that ICMP Echo Request and Reply packets are being sent and received correctly, with valid IP and ICMP checksums.
4. **Signs of Trouble:**
   - **No reply:** Check if `ipv4_input` is being called (add logging). Verify the destination IP check. Double-check your checksum calculation—a single mistake here will cause silent packet drops.
   - **"Destination Host Unreachable" from ping:** Your host's routing table may not know how to reach `192.168.1.100`. Ensure the TAP device is configured with an IP and is up.
   - **Checksum errors in Wireshark:** Wireshark will highlight bad checksums. Compare your `ip_checksum` output with Wireshark's calculation. Remember that for received packets, the checksum field itself is part of the data being checksummed; the result should be `0xFFFF`.

7. Component Design: TCP Connection Management (Milestone 3)

> **Milestone(s):** 3 (TCP Connection Management)

The **Transport Layer** is the heart of reliable communication in Project Valiant. While the Network Layer (IP) provides best-effort packet delivery, the Transport Layer—specifically TCP—transforms this into a reliable, ordered, and flow-controlled byte stream between applications. This component implements the core connection management logic defined in RFC 793, handling the intricate state machine that governs every TCP connection's lifecycle.

Responsibility and Scope

The TCP Connection Management component is responsible for **managing the lifecycle of TCP connections** from establishment through teardown. Its scope includes:

1. **Segment Parsing**: Extracting and validating TCP header fields (ports, sequence numbers, flags, etc.) from IP payloads.
2. **Connection State Management**: Maintaining a finite-state machine for each connection, tracking its progression through the standard TCP states (LISTEN, SYN_SENT, ESTABLISHED, etc.).
3. **Handshake Execution**: Implementing the three-way handshake (SYN → SYN-ACK → ACK) for connection establishment and the four-way FIN handshake for graceful connection termination.
4. **Connection Demultiplexing**: Mapping incoming TCP segments to the correct connection control block based on the 4-tuple (source IP, source port, destination IP, destination port).
5. **Basic Reliability**: Acknowledging received data and retransmitting unacknowledged segments (though detailed sliding window and congestion control are in Milestone 4).

This component **owns**:

- The global list of active and listening TCP connections
- The TCP Control Block (TCB) structures containing all connection state
- The state machine transition logic
- The segment parser and validator

It **does not** own:

- The IP layer routing and delivery (delegates to Network Layer via `ipv4_output`)
- The application data buffers (though it manages pointers to them)
- The detailed retransmission timer algorithms (Milestone 4)
- The sliding window calculations for data transfer (Milestone 4)

Mental Model: A Telephone Call

Understanding TCP's connection management is easiest through the analogy of **a telephone call between two people**. Imagine Alice wants to call Bob:

1. **SYN (Synchronize) → "Hello, are you there?"**: Alice initiates by dialing Bob's number (destination port) and saying "Hello?" This is the SYN packet, containing her initial sequence number (like a conversation starter code).
2. **SYN-ACK (Synchronize-Acknowledge) → "Yes, I'm here and I hear you!"**: If Bob is available (listening on that port), he picks up and says "Hello, I can hear you!" while also acknowledging he heard Alice's "Hello?" This is the SYN-ACK packet, containing his own initial sequence number and an acknowledgment of hers.

3. **ACK (Acknowledge) → "Great, let's talk!"**: Alice hears Bob's response and says "Great, let's talk now," acknowledging she heard his "Hello." This is the ACK packet. The connection is now **ESTABLISHED**—both parties know the other is present and ready.
4. **Data Exchange → Conversation**: Both parties exchange messages (data segments), each acknowledging receipt of the other's previous messages (ACKs). They might pause if the other is talking too fast (flow control).
5. **FIN (Finish) → "Goodbye, I have nothing more to say."**: When Alice is done, she says "Goodbye" (FIN). Bob acknowledges with "Okay, goodbye" (ACK for the FIN). Bob may then also say his own "Goodbye" (FIN) when he's finished, which Alice acknowledges (ACK). This four-step exchange cleanly terminates the conversation.

This mental model highlights several key TCP concepts: **bidirectional communication** (each side has its own sequence number space), **reliable setup/teardown** (explicit handshakes), and **graceful closure** (each side independently signals completion). The TCP state machine formalizes these conversational states and transitions.

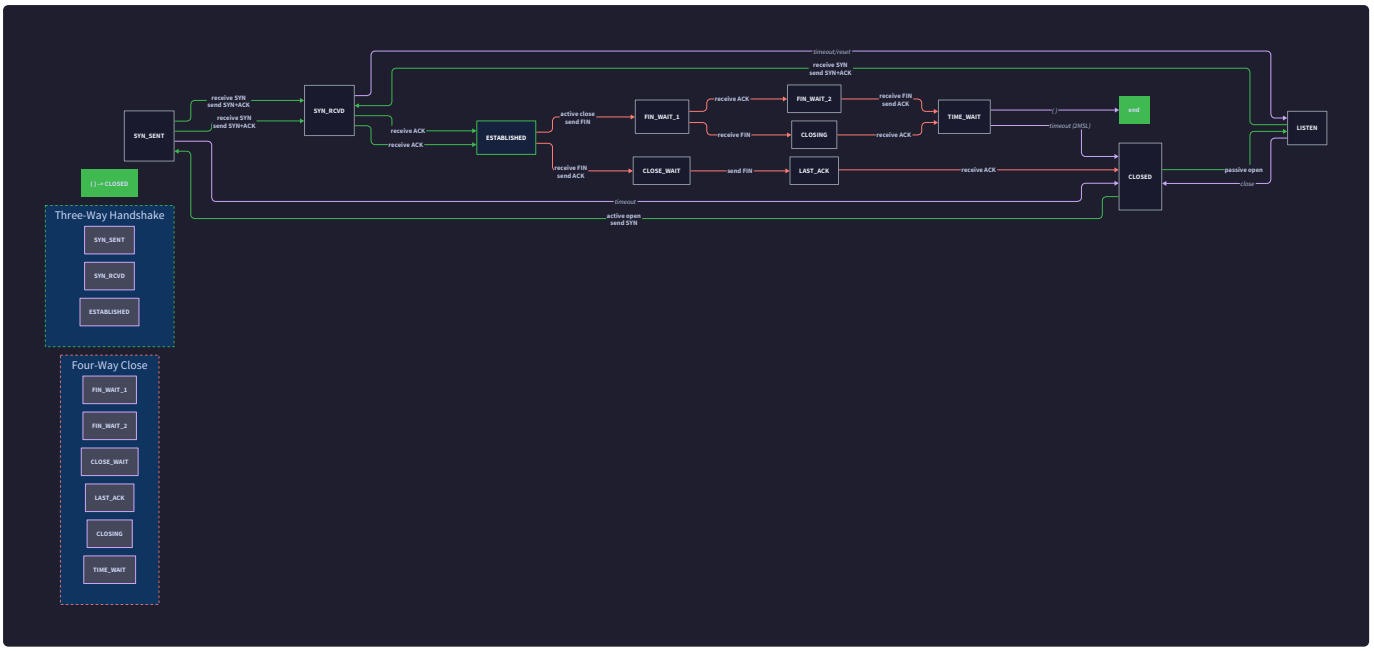
TCP State Machine

TCP's operation is governed by a **finite-state machine** where each connection exists in exactly one state at any time. Events (receiving a segment, application actions, timeouts) trigger transitions between states, often accompanied by specific actions (sending segments, updating variables). The following table documents the essential states and transitions for Milestone 3, based on RFC 793:

Current State	Event / Trigger	Next State	Actions Taken
CLOSED	Application calls `tcp_listen()` on a port	**LISTEN**	Create a TCB in listening state, wait for incoming SYN.
LISTEN	Receive valid SYN segment (to this port)	**SYN_RCVD**	Allocate new TCB for this connection, send SYN-ACK (with our initial sequence number, ACK their SYN), initialize sequence variables.
SYN_RCVD	Receive ACK for our SYN-ACK	**ESTABLISHED**	Connection is fully established. Signal application that connection is ready.
LISTEN	Application calls `tcp_connect()` (active open)	**SYN_SENT**	Create TCB, send SYN to remote host, start retransmission timer for SYN.
SYN_SENT	Receive SYN-ACK (ACK field valid, SYN flag set)	**ESTABLISHED**	Send ACK for their SYN, connection established. Cancel SYN retransmission timer.
SYN_SENT	Receive SYN (simultaneous open)	**SYN_RCVD**	Send SYN-ACK, remain in SYN_RCVD (both sides sent SYN).
ESTABLISHED	Application calls `tcp_close()` (active close)	**FIN_WAIT_1**	Send FIN segment, wait for ACK (and possibly their FIN).
ESTABLISHED	Receive FIN from remote (passive close)	**CLOSE_WAIT**	ACK the FIN, inform application that remote side has closed. Application may now call `tcp_close()` to initiate its own close.
FIN_WAIT_1	Receive ACK for our FIN	**FIN_WAIT_2**	Wait for remote's FIN (they may still have data to send).
FIN_WAIT_1	Receive FIN+ACK (remote also closing)	**TIME_WAIT**	ACK the FIN, move to TIME_WAIT to handle late segments.
FIN_WAIT_2	Receive FIN from remote	**TIME_WAIT**	ACK the FIN, start 2MSL timer.
CLOSE_WAIT	Application calls `tcp_close()` (after passive close)	**LAST_ACK**	Send our FIN, wait for ACK.
LAST_ACK	Receive ACK for our FIN	**CLOSED**	Free TCB and resources.
TIME_WAIT	2MSL (Maximum Segment Lifetime) timer expires	**CLOSED**	Free TCB and resources.
Any state	Receive RST segment	**CLOSED**	Abruptly terminate connection, free resources, signal error to application.
Any state	Retransmission timeout for SYN or FIN	*State-specific*	Retransmit SYN or FIN, restart timer (exponential backoff). If max retries exceeded, transition to CLOSED.

> **Design Insight:** The state machine is the single source of truth for connection behavior. Every incoming segment must be processed in the context of the connection's current state—the same segment (e.g., a FIN) triggers different actions in ESTABLISHED vs. SYN_RCVD. Implementing the state machine as a clear `switch(current_state)` block with well-defined transition functions ensures correctness.

For a visual representation of the complete state machine, including less common transitions like simultaneous open/close, refer to:



Interface

The TCP Connection Management component exposes a set of functions to the layer above (the Socket API/Application) and is invoked by the layer below (IP) for incoming segments. The following table defines the key interface functions:

Method Name	Parameters	Returns	Description
`tcp_input`	`const uint8_t* packet_data`, `size_t len`, `uint32_t src_ip`, `uint32_t dst_ip`	`void`	**Main ingress point for TCP segments.** Called by `ipv4_input` when IP protocol is `IPPROTO_TCP`. Parses TCP header, finds matching TCB (or creates new for SYN in LISTEN), and processes segment according to current state.
`tcp_output`	`struct tcb* connection`, `uint8_t flags`, `const uint8_t* payload`, `size_t payload_len`	`int`	**Constructs and sends a TCP segment.** Builds TCP header with current sequence numbers, computes checksum (pseudo-header included), and calls `ipv4_output` with `IPPROTO_TCP`. Used by state machine to send SYN, ACK, FIN, etc.
`tcp_connect`	`uint32_t remote_ip`, `uint16_t remote_port`, `uint16_t local_port`	`struct tcb*`	**Initiates an active open (client side).** Creates a TCB, selects an ephemeral local port if 0, sends SYN, transitions to SYN_SENT. Returns pointer to TCB (or NULL on error).
`tcp_listen`	`uint16_t local_port`	`struct tcb*`	**Creates a listening endpoint (server side).** Creates a TCB in LISTEN state bound to the specified port. Returns pointer to listening TCB.
`tcp_accept`	`struct tcb* listen_tcb`	`struct tcb*`	**Accepts an established connection.** Called by application after a SYN has been received and the connection is in ESTABLISHED state. Returns a new TCB for the data connection (or NULL if none pending).
`tcp_close`	`struct tcb* connection`	`int`	**Initiates graceful connection close.** If in ESTABLISHED, sends FIN and transitions to FIN_WAIT_1. If in CLOSE_WAIT (after remote FIN), sends FIN and transitions to LAST_ACK. Returns 0 on success.
`tcp_find_connection`	`uint32_t local_ip`, `uint16_t local_port`, `uint32_t remote_ip`, `uint16_t remote_port`	`struct tcb*`	**Demultiplexes incoming segment to correct TCB.** Searches global TCB list for matching 4-tuple. For LISTENING TCBs, remote IP/port may be wildcard (0).
`tcp_send_data`	`struct tcb* connection`, `const uint8_t* data`, `size_t len`	`int`	**Queues application data for transmission.** In Milestone 3, may simply send immediately if window allows. In Milestone 4, will manage sliding window. Returns bytes queued.
`tcp_receive_data`	`struct tcb* connection`, `uint8_t* buffer`, `size_t buffer_len`	`size_t`	**Reads received, in-order data from connection buffer.** Copies data from TCB's receive buffer to application buffer, advances receive window.
`tcp_process`	`struct tcb* connection`, `struct tcp_hdr* hdr`, `const uint8_t* payload`, `size_t payload_len`	`void`	**Core state machine processor.** Called by `tcp_input` after finding TCB. Executes state-specific logic based on segment flags and sequence numbers.

ADR: Centralized TCB vs. Per-Socket State

> **Decision: Centralized TCP Control Block (TCB) Management**
>
> - **Context**: We need to maintain complete state for each TCP connection, including sequence numbers, window sizes, buffers, timers, and the current state. The design must support efficient lookup of TCBs for incoming segments and provide clear ownership of connection resources.
> - **Options Considered**:
>   1. **Centralized TCB List**: A global linked list (or hash table) of `tcb` structures, managed entirely by the TCP layer. The Socket API holds only a handle (e.g., file descriptor integer) that maps to a TCB.
>   2. **Per-Socket State Embedding**: Each socket structure (owned by the Socket API layer) contains the TCP connection state directly as an embedded struct. The TCP layer operates on socket objects.
> - **Decision**: Centralized TCB List managed by the TCP layer.
> - **Rationale**:
>   - **Separation of Concerns**: The TCP protocol logic is complex and self-contained. Keeping all TCP state in a dedicated TCB structure isolates protocol details from the socket abstraction, making both layers easier to test and reason about.
>   - **Lookup Efficiency**: Incoming segments must find the correct TCB quickly. A centralized list allows the TCP layer to implement an optimized lookup (e.g., hash on 4-tuple) without exposing internal data structures to the socket layer.
>   - **Resource Management**: The TCP layer is responsible for connection lifetimes (including TIME_WAIT cleanup). Central ownership simplifies resource cleanup when connections close—the TCP layer can free TCBs without coordinating with socket layer garbage collection.
>   - **Educational Clarity**: For learners, seeing a clear `tcb` struct that mirrors RFC 793's "Transmission Control Block" concept reinforces protocol specification mapping to implementation.
> - **Consequences**:
>   - **Positive**: Clean layer separation; TCP implementation can evolve independently of socket API; efficient demultiplexing possible.
>   - **Negative**: Requires a mapping mechanism (e.g., file descriptor to TCB pointer) in the socket layer; slightly more indirection when application calls socket functions.

The following table compares the two options:

Option	Pros	Cons	Why Not Chosen
Centralized TCB List	Protocol logic encapsulated; efficient centralized lookup; clear resource ownership; matches RFC terminology.	Requires mapping between socket handles and TCBs; additional indirection for socket operations.	**CHOSEN** - Best for separation of concerns and educational alignment with RFC.
Per-Socket State Embedding	Direct access from socket operations; no mapping overhead.	TCP logic spills into socket layer; harder to optimize lookup (must scan all sockets); conflates abstraction layers.	Rejected - Blurs layer boundaries and makes TCP logic harder to isolate for testing.

Common Pitfalls

Implementing TCP connection management is fraught with subtle bugs that can cause connections to hang, fail, or behave unreliably. Below are the most common pitfalls for learners:

⚡ Pitfall 1: Incorrect State Transitions

- **Description**: Transitioning to the wrong state based on an event, e.g., moving from SYN_RCVD to ESTABLISHED upon receiving any ACK (instead of only the ACK for our SYN-ACK).
- **Why It's Wrong**: Violates TCP specification, causing connections to become "half-open" or fail to close properly. The state machine ensures all protocol guarantees (reliability, ordered delivery) are maintained.
- **How to Avoid**: Implement the state machine as a strict `switch` statement with explicit conditions for each transition. Refer to RFC 793's state diagram and the transition table above. Log state changes for debugging.

⚡ Pitfall 2: Sequence Number Initialization and Wrap-Around

- **Description**: Using incorrect initial sequence numbers (ISN) or failing to handle 32-bit sequence number arithmetic (wrap-around after 2^32-1).
- **Why It's Wrong**: Sequence numbers provide reliability and order. If ISNs are predictable or reused, it can cause segment misordering. Wrap-around without proper handling leads to incorrect "future" vs. "past" judgments.
- **How to Avoid**: Generate ISNs using a moderately secure random increment (e.g., based on clock). Use the provided `seq_lt(a, b)` and `seq_diff(a, b)` helper functions for all sequence comparisons and arithmetic. These functions handle wrap-around correctly.

⚡ Pitfall 3: Ignoring RST Packets

- **Description**: Not processing the RST (Reset) flag or not transitioning to CLOSED when a valid RST is received.

- **Why It's Wrong**: RST is a hard reset signal. Ignoring it leaves connections in a zombie state, consuming resources. RFC requires that RST be accepted in any state (except if it doesn't match sequence window).
- **How to Avoid**: Always check for the RST flag after parsing headers. If set and the sequence number is within the acceptable window (or the connection is not yet synchronized), immediately free the TCB and transition to CLOSED.

⚡ **Pitfall 4: Mishandling Simultaneous Open/Close**

- **Description**: Not handling the case where both sides send SYN simultaneously (leading to SYN_RCVD state on both) or both send FIN simultaneously.
- **Why It's Wrong**: Real networks exhibit these scenarios. Failing to handle them can cause connections to deadlock or close improperly.
- **How to Avoid**: Follow RFC 793 precisely: if in SYN_SENT and receive a SYN (without ACK), transition to SYN_RCVD and send SYN-ACK. For simultaneous close, handle the transition from FIN_WAIT_1 to CLOSING upon receiving FIN+ACK.

⚡ **Pitfall 5: Not Validating Segment Sequence Numbers**

- **Description**: Accepting segments whose sequence numbers are outside the receive window (either too old—already acknowledged—or too far in the future).
- **Why It's Wrong**: Security and reliability: accepting out-of-window data can corrupt the stream or enable blind attacks. TCP must only process data that is within the current receive window.
- **How to Avoid**: Before processing payload or flags (except RST), check that the segment's sequence number is within `[rcv.nxt, rcv.nxt + rcv.wnd)` for data, or that it is exactly `rcv.nxt` for SYN/FIN (which consume one sequence number). Use `seq_lt` and `seq_diff` for these checks.

Implementation Guidance

This section provides concrete starter code and structure for implementing TCP Connection Management in C, following the naming conventions.

A. Recommended File/Module Structure

Add TCP-specific files under a `tcp/` directory:

```
project_valiant/
├── src/
│   ├── main.c
│   ├── net/
│   │   ├── ethernet.c
│   │   ├── arp.c
│   │   ├── ipv4.c
│   │   ├── icmp.c
│   │   ├── route.c
│   │   └── tcp/              # New TCP module directory
│   │       ├── tcp.c         # Core state machine, tcp_input, tcp_output
│   │       ├── tcb.c         # TCB management, lookup, allocation
│   │       ├── tcp_timer.c   # Retransmission and TIME_WAIT timers (Milestone 4)
│   │       ├── tcp_buffer.c  # Send/receive buffer management (Milestone 4)
│   │       └── tcp.h         # Public interface and structures
│   └── util/
│       ├── checksum.c
│       └── helpers.c
└── include/valiant/
    ├── net.h
    └── tcp.h                 # Public TCP API for socket layer
```

B. Infrastructure Starter Code

File: `src/net/tcp/tcp.h` - Core data structures and constants.

```c
#ifndef VALIANT_TCP_H

#define VALIANT_TCP_H


#include <stdint.h>

#include <stddef.h>

#include <stdbool.h>


/* TCP header flags (bits in data_offset_reserved_flags) */

#define TCP_FIN  (1 << 0)

#define TCP_SYN  (1 << 1)

#define TCP_RST  (1 << 2)

#define TCP_PSH  (1 << 3)

#define TCP_ACK  (1 << 4)

#define TCP_URG  (1 << 5)


/* TCP header length without options */

#define TCP_HDR_LEN 20


/* TCP states (simplified for Milestone 3) */

typedef enum {

    TCP_STATE_CLOSED = 0,

    TCP_STATE_LISTEN,

    TCP_STATE_SYN_SENT,

    TCP_STATE_SYN_RCVD,

    TCP_STATE_ESTABLISHED,

    TCP_STATE_FIN_WAIT_1,

    TCP_STATE_FIN_WAIT_2,

    TCP_STATE_CLOSE_WAIT,

    TCP_STATE_CLOSING,

    TCP_STATE_LAST_ACK,

    TCP_STATE_TIME_WAIT

} tcp_state_t;


/* TCP Control Block (TCB) - core connection state */

typedef struct tcb {

    /* Connection identifiers */

    uint32_t local_ip;

    uint16_t local_port;

    uint32_t remote_ip;

    uint16_t remote_port;



    /* State and sequence numbers */

    tcp_state_t state;
```

```
    uint32_t snd_nxt;      /* Next sequence number to send */
    uint32_t snd_una;      /* Oldest unacknowledged sequence number */
    uint32_t rcv_nxt;      /* Next sequence number expected to receive */

    /* Window sizes */
    uint16_t snd_wnd;      /* Send window (from remote's advertised window) */
    uint16_t rcv_wnd;      /* Receive window (we advertise to remote) */

    /* Buffers (stubs for Milestone 3, expanded in Milestone 4) */
    void* rx_buffer;       /* Receive buffer (circular) */
    void* tx_buffer;       /* Send buffer (circular) */

    /* Timers (stubs) */
    void* retransmit_timer;
    void* timewait_timer;

    /* Application callbacks/pointers (for socket layer integration) */
    void* app_data;

    /* Linked list next pointer */
    struct tcb* next;
} tcb_t;

/* TCP header structure (network byte order) */
typedef struct __attribute__((packed)) tcp_hdr {
    uint16_t src_port;
    uint16_t dst_port;
    uint32_t seq_num;
    uint32_t ack_num;
    uint16_t data_offset_reserved_flags; /* 4 bits data offset, 6 bits reserved, 6 bits flags */
    uint16_t window;
    uint16_t checksum;
    uint16_t urg_ptr;
} tcp_hdr_t;

/* Public API functions */
void tcp_input(const uint8_t* packet_data, size_t len, uint32_t src_ip, uint32_t dst_ip);
int tcp_connect(uint32_t remote_ip, uint16_t remote_port, uint16_t local_port);
tcb_t* tcp_listen(uint16_t local_port);
int tcp_close(tcb_t* connection);

/* Helper functions */
```

```
uint16_t tcp_checksum(const tcp_hdr_t* hdr, const uint8_t* payload, size_t payload_len,
                      uint32_t src_ip, uint32_t dst_ip);
tcb_t* tcp_find_connection(uint32_t local_ip, uint16_t local_port,
                           uint32_t remote_ip, uint16_t remote_port);

#endif /* VALIANT_TCP_H */
```

C. Core Logic Skeleton Code

File: `src/net/tcp/tcp.c` - Main state machine and processing.

```c
#include "tcp.h"
#include "../ipv4.h"
#include "../../util/checksum.h"
#include "../../util/helpers.h"
#include <string.h>
#include <stdlib.h>

/* Global list of TCBs */
static tcb_t* tcb_list = NULL;

/* Internal function prototypes */
static void tcp_process(tcb_t* tcb, tcp_hdr_t* hdr, const uint8_t* payload, size_t payload_len);
static void tcp_send_segment(tcb_t* tcb, uint8_t flags, const uint8_t* payload, size_t payload_len);
static void tcp_handle_listen(tcb_t* tcb, tcp_hdr_t* hdr, const uint8_t* payload, size_t payload_len);
static void tcp_handle_syn_sent(tcb_t* tcb, tcp_hdr_t* hdr, const uint8_t* payload, size_t payload_len);
static void tcp_handle_established(tcb_t* tcb, tcp_hdr_t* hdr, const uint8_t* payload, size_t payload_len);
/* ... other state handlers ... */

void tcp_input(const uint8_t* packet_data, size_t len, uint32_t src_ip, uint32_t dst_ip) {
    /* TODO 1: Verify minimum length (at least TCP header size) */

    /* TODO 2: Parse TCP header (cast to tcp_hdr_t), ensure it's in network byte order */

    /* TODO 3: Verify TCP checksum (using tcp_checksum helper) - drop if invalid */

    /* TODO 4: Extract source/dest ports from header */

    /* TODO 5: Find matching TCB using tcp_find_connection() with the 4-tuple */

    /* TODO 6: If no TCB found but segment has SYN flag and we have a listening socket on dst_port,
                create new TCB for this connection, initialize sequence numbers, transition to SYN_RCVD,
                and send SYN-ACK. */

    /* TODO 7: If TCB found, call tcp_process(tcb, hdr, payload, payload_len) */

    /* TODO 8: If no matching TCB and not a SYN to a listening port, send RST (unless segment itself is RST) */
}

static void tcp_process(tcb_t* tcb, tcp_hdr_t* hdr, const uint8_t* payload, size_t payload_len) {
    /* TODO 1: Validate segment sequence number against current receive window.
               For ESTABLISHED state: data must be within [rcv_nxt, rcv_nxt+rcv_wnd).
               For SYN_RCVD/FIN_WAIT_1 etc: SYN/FIN must have exact sequence rcv_nxt.
```

```
            If invalid, send ACK with current rcv_nxt (unless RST flag set). */

    /* TODO 2: Check for RST flag - if set and sequence valid, reset connection and return. */

    /* TODO 3: Dispatch to state-specific handler based on tcb->state */
    switch (tcb->state) {
        case TCP_STATE_LISTEN:
            tcp_handle_listen(tcb, hdr, payload, payload_len);
            break;
        case TCP_STATE_SYN_SENT:
            tcp_handle_syn_sent(tcb, hdr, payload, payload_len);
            break;
        case TCP_STATE_SYN_RCVD:
            /* TODO: Implement SYN_RCVD handler */
            break;
        case TCP_STATE_ESTABLISHED:
            tcp_handle_established(tcb, hdr, payload, payload_len);
            break;
        /* TODO: Handle other states */
        default:
            /* Unexpected state - possibly send RST? */
            break;
    }

    /* TODO 4: If ACK flag is set and valid, update snd_una (oldest unacknowledged) */

    /* TODO 5: If segment contains data (payload_len > 0) and sequence is valid,
               deliver to receive buffer (for Milestone 3, just store in simple buffer) */
}

static void tcp_handle_listen(tcb_t* tcb, tcp_hdr_t* hdr, const uint8_t* payload, size_t payload_len) {
    /* TODO 1: In LISTEN state, we only accept SYN segments (no ACK) */

    /* TODO 2: Verify SYN flag is set and ACK flag is NOT set */

    /* TODO 3: Create a new TCB for this incoming connection (copy local IP/port,
               set remote IP/port from segment, initialize sequence numbers) */

    /* TODO 4: Set new TCB state to SYN_RCVD */

    /* TODO 5: Send SYN-ACK: our sequence number = new ISN, ACK number = their seq+1 */
```

```
    /* TODO 6: Insert new TCB into global list */
}

static void tcp_handle_syn_sent(tcb_t* tcb, tcp_hdr_t* hdr, const uint8_t* payload, size_t payload_len) {
    /* TODO 1: Check for valid SYN-ACK (both SYN and ACK flags, ack_num == our snd_nxt+1) */

    /* TODO 2: If valid SYN-ACK: transition to ESTABLISHED, update rcv_nxt (their seq+1),
               send ACK for their SYN, cancel retransmission timer */

    /* TODO 3: If SYN without ACK (simultaneous open): transition to SYN_RCVD,
               send SYN-ACK */

    /* TODO 4: If bad ACK (acknowledges something not sent): send RST */
}

static void tcp_handle_established(tcb_t* tcb, tcp_hdr_t* hdr, const uint8_t* payload, size_t payload_len) {
    /* TODO 1: Check for FIN flag - if set: ACK it, transition to CLOSE_WAIT (passive close),
               notify application that remote side closed */

    /* TODO 2: If data present and sequence valid: store in receive buffer,
               advance rcv_nxt, send ACK for received data */

    /* TODO 3: If ACK flag set (for our data): update snd_una, cancel retransmission timer
               for acknowledged data */
}

int tcp_connect(uint32_t remote_ip, uint16_t remote_port, uint16_t local_port) {
    /* TODO 1: Allocate new TCB, initialize fields */

    /* TODO 2: If local_port is 0, assign an ephemeral port (e.g., between 49152-65535) */

    /* TODO 3: Generate initial sequence number (ISN) - use a random value */

    /* TODO 4: Set state to SYN_SENT */

    /* TODO 5: Send SYN segment (tcp_send_segment with SYN flag) */

    /* TODO 6: Start retransmission timer for SYN */

    /* TODO 7: Insert TCB into global list */
```

```
    /* TODO 8: Return TCB pointer (or NULL on error) */
    return 0;
}

static void tcp_send_segment(tcb_t* tcb, uint8_t flags, const uint8_t* payload, size_t payload_len) {
    /* TODO 1: Allocate buffer for full TCP segment (header + payload) */

    /* TODO 2: Fill TCP header:
        - src_port, dst_port from TCB
        - seq_num: use tcb->snd_nxt (increment after if SYN/FIN/data)
        - ack_num: use tcb->rcv_nxt if ACK flag set, else 0
        - flags: set appropriate bits
        - window: advertise our current receive window (tcb->rcv_wnd)
        - urg_ptr: 0 (unless URG flag set)
    */

    /* TODO 3: Copy payload if any */

    /* TODO 4: Compute TCP checksum (pseudo-header + TCP header + payload) */

    /* TODO 5: Call ipv4_output(tcb->remote_ip, IPPROTO_TCP, segment, total_len) */

    /* TODO 6: If sending SYN/FIN or data, update snd_nxt (increment by 1 for SYN/FIN, payload_len for data) */

    /* TODO 7: If sending SYN/FIN or data, start retransmission timer for this segment */
}
```

D. Milestone 3 Checkpoint

After implementing TCP Connection Management, you should be able to:

1. **Establish a TCP connection**:
   - Start your stack with a listening socket on port 8080.
   - From another machine (or local loopback), use `netcat` to connect:

```
nc -v 192.168.1.100 8080
```

   - Observe in your stack logs: SYN received → SYN-ACK sent → ACK received → state transition to ESTABLISHED.
   - The `netcat` connection should establish successfully (though no data can be exchanged yet).
2. **Close a TCP connection gracefully**:
   - After connection establishment, type `Ctrl+C` in `netcat` to initiate active close.
   - Observe: FIN from client → ACK from stack → stack may send its own FIN (if application calls `tcp_close`) → final ACK from client.
   - Connection should transition through FIN_WAIT_1/2, TIME_WAIT, then CLOSED.
3. **Verify state machine transitions** using a test harness that sends crafted packets:
   - Send SYN to listening port → expect SYN-ACK.
   - Send ACK for the SYN-ACK → connection established.
   - Send FIN → expect ACK.

- Send RST → connection should immediately close.

Expected Debug Output:

```
[TCP] Listening on port 8080
[TCP] Received SYN from 192.168.1.50:54321 to 192.168.1.100:8080
[TCP] Sending SYN-ACK, new TCB state SYN_RCVD
[TCP] Received ACK for SYN-ACK, connection established
[TCP] Received FIN from 192.168.1.50:54321
[TCP] Sending ACK for FIN, state CLOSE_WAIT
[TCP] Application called close, sending FIN, state LAST_ACK
[TCP] Received ACK for FIN, connection closed
```

If connections hang or fail:

- Use Wireshark to capture packets between your stack and the client. Verify that SYN, SYN-ACK, ACK sequence is correct.
- Check that sequence numbers and ACK numbers are properly incremented (SYN/FIN consume one sequence number).
- Add extensive logging of state transitions and segment processing to identify where the state machine deviates from expectations.
- Ensure you're handling all TCP flags correctly and validating sequence numbers against the window.

---

8. Component Design: TCP Data Transfer (Milestone 4)

> **Milestone(s):** 4 (TCP Data Transfer & Flow Control)

This component transforms the TCP connection established in Milestone 3 into a reliable, bidirectional data pipe. While connection management sets up the communication channel, data transfer ensures every byte sent arrives correctly, in order, without overwhelming the receiver or the network. This is where TCP earns its reputation as a **reliable, connection-oriented protocol**—a promise that requires sophisticated coordination between sliding windows, retransmission timers, flow control, and congestion control mechanisms working in concert.

Responsibility and Scope

The **TCP Data Transfer** component is responsible for implementing **reliable, in-order data delivery** over the established TCP connections. This encompasses four core mechanisms:

1. **Sliding Window Protocol:** Manages the sequence number space for outstanding, unacknowledged data. The sender can transmit multiple segments up to the window size without waiting for individual acknowledgments, dramatically improving throughput over a simple stop-and-wait protocol.
2. **Retransmission and Acknowledgment:** Implements reliability through positive acknowledgments (ACKs). Any segment not acknowledged within a timeout period (Retransmission Timeout - RTO) is resent. The component must handle duplicate ACKs, which can signal segment loss and trigger fast retransmit.
3. **Flow Control:** Protects the receiver's finite buffer space. The receiver advertises its available capacity in the `window` field of every TCP header (`rcv_wnd`). The sender must never transmit data beyond the receiver's advertised window, ensuring the receiver is never forced to discard data due to overflow.
4. **Congestion Control:** Protects the network from collapse due to overload. The sender maintains a **congestion window (`cwnd`)** that limits the amount of in-flight data based on perceived network conditions. This component implements the core algorithms of **Slow Start** and **Congestion Avoidance**, dynamically adjusting `cwnd` in response to ACK arrivals and loss events (timeouts or duplicate ACKs).

Key Data Ownership: This component owns and manages the per-connection `tcb` 's send and receive buffers (`tx_buffer` , `rx_buffer`), the retransmission timer, and the congestion control state variables (`ssthresh` , `cwnd`). It processes incoming ACKs, updates window state, triggers retransmissions, and decides when new data can be sent.

Out of Scope: Advanced TCP extensions like Selective Acknowledgments (SACK), Explicit Congestion Notification (ECN), or sophisticated congestion control variants (CUBIC, BBR). These are valuable extensions but beyond the educational core.

Mental Model: A Windowed Conveyor Belt with Traffic Control

Imagine a **loading dock with a sliding window conveyor belt** that delivers packages (data segments) from a warehouse (sender application) to a store (receiver application).

- **The Sliding Window:** The conveyor belt has a movable section—the "window"—that can hold a certain number of packages ready for transit. This window's size is determined by two factors: 1) the store's current storage capacity (the **receiver's advertised window**), and 2) traffic conditions on the delivery route (the **congestion window**). The sender can only load packages within this active window onto the belt. When the store confirms receipt of the first package (sends an **ACK**), the window slides forward, allowing the next package in line to be loaded.
- **Retransmission:** For every package sent, the warehouse starts a timer. If the confirmation receipt isn't received before the timer expires, the warehouse assumes the package was lost and loads an identical copy onto the conveyor belt again. If the warehouse receives *three duplicate confirmations* for the same package (e.g., the store keeps asking for package #5), it takes this as a strong hint that package #5 was lost and immediately resends it—this is **fast retransmit**.
- **Flow Control:** The store periodically signals how much empty shelf space it has. The warehouse's loading window must never exceed this available space. If the store's shelves are full (window size = 0), the warehouse stops loading packages entirely until the store clears some space and sends an update.

- **Congestion Control:** The delivery route has variable traffic. Initially, the warehouse doesn't know the road's capacity, so it starts cautiously, sending just one package and waiting for its confirmation (**Slow Start**). Each successful round-trip confirmation allows it to *double* the number of packages in transit (exponential growth) until it hits a threshold or detects traffic (loss). After that, it switches to **Congestion Avoidance**, increasing the window by just one package per successful round-trip (linear growth) to gently probe for more available bandwidth.

This mental model separates the concerns: the sliding window is the *mechanism*, flow control respects the *receiver's limits*, and congestion control respects the *network's limits*.

Interface

The TCP Data Transfer functionality is accessed through a combination of application-facing calls and internal processing routines. The table below details the key functions.

Method Name	Parameters	Returns	Description
`tcp_send_data`	`tcb_t* conn`, `const void* data`, `size_t len`	`int` (bytes queued or error)	Application-level call to queue data for transmission. Copies `data` into the connection's send buffer (`tx_buffer`). Does not necessarily transmit immediately; transmission is governed by the sliding window and congestion control. Returns number of bytes successfully queued.
`tcp_receive_data`	`tcb_t* conn`, `void* buffer`, `size_t buffer_len`	`size_t` (bytes copied)	Application-level call to read received, in-order data from the connection's receive buffer (`rx_buffer`). Copies up to `buffer_len` bytes into `buffer` and removes them from the buffer, advancing `rcv_nxt`.
`tcp_input` (extends Milestone 3)	`const void* packet_data`, `size_t len`, `uint32_t src_ip`, `uint32_t dst_ip`	`void`	Main ingress point from IP layer. Now, in the `ESTABLISHED` state, it also: 1) Processes payload data (placing it in `rx_buffer`), 2) Updates `rcv_nxt` and `rcv_wnd`, 3) Generates ACKs for received data, 4) Processes incoming ACKs to update `snd_una` and handle retransmissions.
`tcp_output` (extends Milestone 3)	`tcb_t* conn`, `uint16_t flags`, `const void* payload`, `size_t payload_len`	`int` (0 on success)	Constructs and sends a TCP segment. Now includes logic for: 1) Packaging data from `tx_buffer` according to the current window (`snd_nxt`, `snd_una`, `cwnd`, `snd_wnd`), 2) Starting the retransmission timer for the segment if it contains new data, 3) Updating `snd_nxt`.
`tcp_ack_received` (internal)	`tcb_t* conn`, `uint32_t ack_num`, `uint16_t window`	`void`	Core internal subroutine called by `tcp_input` when a valid ACK is processed. Updates `snd_una` (sliding the send window), cancels the retransmission timer for all acknowledged data, updates `snd_wnd`, and invokes congestion control logic to potentially increase `cwnd`. Also handles duplicate ACK counting for fast retransmit.
`retransmit_timer_callback`	`void* arg` (typically `tcb_t*`)	`void`	Callback function invoked when the retransmission timer for a connection expires. Identifies the oldest unacknowledged segment (`snd_una`) and retransmits it. Also invokes congestion control logic to respond to the loss (e.g., set `ssthresh`, reset `cwnd`).
`tcp_send_window_available` (internal)	`const tcb_t* conn`	`size_t`	Calculates the current **usable window**: the minimum of (`cwnd`, `snd_wnd`) minus the amount of outstanding data (`snd_nxt - snd_una`). Returns 0 if no window is available. This function drives the decision of whether `tcp_output` can send more data.

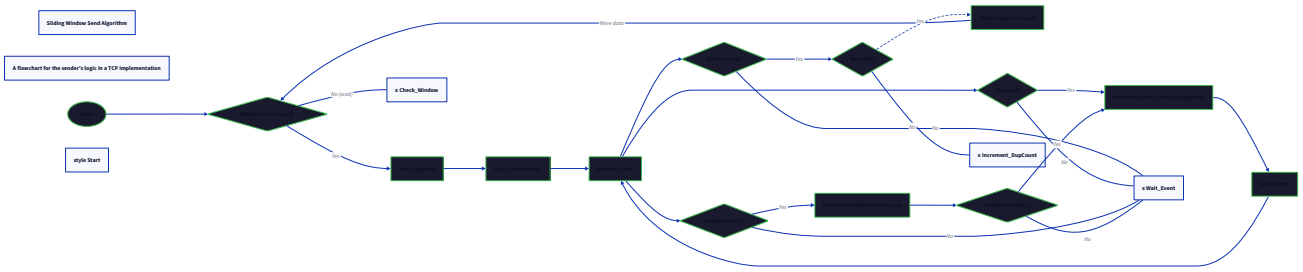
Internal Behavior

The data transfer logic is a complex interplay of events driven by application writes, ACK arrivals, and timer expirations. The following numbered steps describe the core algorithms.

1. Sending Application Data (`tcp_send_data` -> `tcp_output` Path)

1. The application calls `tcp_send_data(conn, data, len)`.
2. The function copies `data` into the connection's `tx_buffer` (a circular buffer managed by the `tcp_buffer` structure).
3. It then attempts to **send any pending data** by calling an internal `tcp_try_send(conn)` routine (not in the interface table, but a logical helper).
4. `tcp_try_send` calculates the **usable window** (see `tcp_send_window_available`). If the window is zero, it returns immediately; the data remains buffered until window space opens.
5. If window space is available, it extracts a chunk of data from `tx_buffer` up to the usable window size (or the Maximum Segment Size - MSS).
6. It calls `tcp_output(conn, TCP_ACK, chunk_data, chunk_len)` to construct and send the segment. The `seq_num` in the header is set to `conn->snd_nxt`.
7. `tcp_output` sends the segment via the IP layer, increments `conn->snd_nxt` by `chunk_len`, and **arms the retransmission timer** for this segment if it is not already running. (If the timer is already running, it's left alone; it covers the oldest unacknowledged data).

8. Steps 4-7 repeat until either the `tx_buffer` is empty or the usable window is exhausted.



2. Processing an Incoming Data Segment & Generating ACKs (`tcp_input` in `ESTABLISHED`)

1. A segment arrives and is demultiplexed to the correct `tcb`.
2. The segment's `seq_num` is validated. It is acceptable if it falls within the range `[rcv_nxt, rcv_nxt + rcv_wnd)`. Segments with older sequence numbers (duplicates) are acknowledged but otherwise ignored.
3. If the segment contains new, in-order data:
   - The payload is copied into the `rx_buffer` at the appropriate offset.
   - `conn->rcv_nxt` is advanced by the payload length.
   - `conn->rcv_wnd` is recalculated based on the remaining free space in `rx_buffer`.
4. **An ACK must be generated.** RFC 793 mandates that an ACK should be sent for every segment containing data, and should not be delayed more than 500ms. For simplicity, our implementation will send an **immediate ACK**:
   - The ACK number is set to `conn->rcv_nxt`.
   - The window field is set to the new `conn->rcv_wnd`.
   - `tcp_output(conn, TCP_ACK, NULL, 0)` is called to send the pure ACK segment.

3. Processing an Incoming ACK (`tcp_ack_received`)

1. Validate the ACK: It must acknowledge new data, i.e., `ack_num` must be > `conn->snd_una` and ≤ `conn->snd_nxt`.
2. **Slide the send window:** Set `conn->snd_una = ack_num`. All data with sequence numbers less than `ack_num` is considered delivered.
3. **Cancel retransmission timer:** If `snd_una` has advanced past the sequence number of the segment that was being timed, the timer is stopped. If unacknowledged data remains, the timer is restarted for the *new* oldest segment.
4. **Update flow control:** Set `conn->snd_wnd` from the received `window` field (converted from network byte order).
5. **Process duplicate ACKs:** If `ack_num` is equal to `snd_una` (it's acknowledging the same data again), increment a duplicate ACK counter. If the counter reaches 3, trigger **fast retransmit**: immediately retransmit the segment starting at `snd_una` and perform a congestion response (set `ssthresh`, reduce `cwnd`).
6. **Invoke Congestion Control:** If this is a new ACK (advancing `snd_una`):
   - In **Slow Start**: Increment `cwnd` by **MSS** for each ACK received (effectively doubling `cwnd` per RTT).
   - In **Congestion Avoidance**: Increment `cwnd` by **(MSS * MSS) / cwnd** for each ACK received (additive increase, roughly +1 MSS per RTT).
7. After processing the ACK, call `tcp_try_send(conn)` again. The newly available window (from sliding `snd_una` forward and possibly increased `cwnd`) may allow more buffered data to be sent.

4. Handling Retransmission Timeout (`retransmit_timer_callback`)

1. The timer callback is invoked with the `tcb` as its argument.
2. **Retransmit:** Call `tcp_output` to retransmit the oldest unacknowledged segment (the one starting at `conn->snd_una`).
3. **Congestion Response:** This is a strong signal of network congestion.
   - Set **slow start threshold**: `ssthresh = max(cwnd / 2, 2 * MSS)`.
   - Reset **congestion window**: `cwnd = 1 * MSS` (enter Slow Start).
4. **Backoff the timer:** Double the Retransmission Timeout (RTO) value for this connection (exponential backoff). `RTO = min(MAX_RTO, RTO * 2)`.
5. Restart the retransmission timer for the just-retransmitted segment.

ADR: Timer Management Strategy

> **Decision: Single Periodic Timer Tick with Per-Connection RTO Tracking**
>
> - **Context:** We need to manage retransmission timeouts (RTOs) for potentially multiple concurrent TCP connections. Each connection's RTO is dynamic, calculated using smoothed round-trip time (SRTT) measurements. Implementing precise, individual timers for each outstanding segment is complex and resource-intensive for a user-space educational stack.
> - **Options Considered:**
>   1. **Per-Segment Kernel Timers:** Using OS timer facilities (e.g., `timerfd` on Linux, `kevent` on BSD) to create a separate timer for each segment sent. This offers high accuracy but introduces significant complexity in timer creation, deletion, and callback management, potentially requiring a large number of timers under high load.
>   2. **Single Periodic Timer Wheel:** Implement a classic "timer wheel" within the stack. A single periodic interrupt (e.g., from `setitimer` or a thread sleeping at a fixed tick interval, like 10ms) fires a callback. This callback iterates through all active connections, decrements their individual RTO counters, and triggers retransmission when a counter hits zero.
> - **Decision:** Use a **Single Periodic Timer Tick** (Option 2).
> - **Rationale:**
>   - **Simplicity:** A single timer source is far easier to implement and reason about in an educational context. It avoids the complexities of managing many dynamic OS timers.
>   - **Control:** It keeps all timing logic within the stack, making the behavior portable and explicit. Learners can step through the timer tick function to see how RTOs are managed.
>   - **Sufficiency:** For the expected scale (tens of connections) and the educational goal, millisecond-level granularity (achievable with a 10ms tick) is perfectly adequate. TCP's RTO is inherently an estimate, not a precision instrument.
>   - **Established Pattern:** This is a common technique in embedded and user-space network stacks.
> - **Consequences:**
>   - **Time Granularity:** Retransmissions can be delayed by up to one timer tick interval (e.g., 10ms). This is an acceptable trade-off.
>   - **CPU Overhead:** The timer tick runs periodically even when there's no network activity, performing list iterations. This is negligible for our use case.
>   - **Scalability:** Iterating over all connections on every tick is O(n). For a very large number of connections (thousands), a hierarchical timer wheel would be more efficient, but this is beyond our non-goal of high performance.

Option	Pros	Cons	Chosen?
Per-Segment Kernel Timers	High precision, offloads scheduling to the efficient OS kernel.	High complexity in lifecycle management, potential for many timers, harder to debug and port.	No
Single Periodic Timer Wheel	Simple, predictable, self-contained, excellent for learning. All timing logic is visible.	Lower time granularity, constant background CPU tick, O(n) scan overhead.	**Yes**

Common Pitfalls

⚠️ Pitfall: Incorrect Usable Window Calculation

- **Description:** Calculating the send window as simply `min(cwnd, snd_wnd)` without subtracting the already-in-flight data (`snd_nxt - snd_una`). This leads to sending more data than allowed, violating flow or congestion control.
- **Why it's wrong:** The congestion window (`cwnd`) and advertised window (`snd_wnd`) are absolute limits. The amount of data already sent but not yet acknowledged (`snd_nxt - snd_una`) consumes part of that limit. The *usable* window is the remaining quota: `min(cwnd, snd_wnd) - (snd_nxt - snd_una)`.
- **Fix:** Always compute the usable window before sending. Implement and use the `tcp_send_window_available` helper function.

⚠️ Pitfall: Mishandling TCP Sequence Number Wraparound

- **Description:** Using simple integer comparisons (`<`, `>`) for 32-bit sequence numbers. Since sequence numbers wrap around after 2^32 - 1, a naive comparison will fail (e.g., 4,294,967,290 < 10 should be true after wraparound, but integer comparison says false).
- **Why it's wrong:** It breaks all sequence number arithmetic: checking if data is within the receive window, calculating ACK validity, and computing outstanding data amounts. This causes connections to lock up or accept corrupt data.
- **Fix:** Never use raw `uint32_t` comparisons. **Always** use the provided helper functions `seq_lt(a, b)` (true if `a < b` modulo 2^32) and `seq_diff(a, b)` (returns the signed 32-bit difference `a - b` with wrap handling). These functions implement RFC 1982 serial number arithmetic.

⚠️ Pitfall: Forgetting to Update `rcv_wnd` in ACKs

- **Description:** Sending ACKs with a static or incorrect receiver window value. The window advertised in the TCP header (`rcv_wnd`) must reflect the *current* amount of free space in the application's receive buffer.
- **Why it's wrong:** If the advertised window is too large, the sender may overflow the receiver's buffer, causing data loss. If it's too small or zero (when space is actually available), it can create a **zero window deadlock**, stalling the connection unnecessarily.

- **Fix:** Recalculate `rcv_wnd` every time an ACK is generated, based on `rx_buffer->size - rx_buffer->readable`. Always convert this value to **network byte order** before placing it in the TCP header.

⚠️ **Pitfall: Not Restarting the Retransmission Timer on New ACKs**

- **Description:** When a new ACK arrives that acknowledges some but not all outstanding data, the developer forgets to restart the retransmission timer for the *new* oldest unacknowledged segment.
- **Why it's wrong:** The timer was originally set for the first segment sent. If that segment is acknowledged, but a later segment is still outstanding, the timer should now be measuring the RTO for that later segment. Failing to restart it means a loss of the later segment might never be detected.
- **Fix:** The rule is: **The retransmission timer should always be running if there is any unacknowledged data in flight.** Whenever `snd_una` advances, check if `snd_nxt > snd_una`. If yes, restart the timer, setting its expiry based on the current RTO.

⚠️ **Pitfall: Congestion Window Inflation During Loss Recovery**

- **Description:** Continuing to increase the congestion window (`cwnd`) in Slow Start or Congestion Avoidance when processing duplicate ACKs or after a timeout, before the loss recovery is complete.
- **Why it's wrong:** Duplicate ACKs and timeouts are signals of congestion. Increasing `cwnd` during this period exacerbates the network overload, contrary to the core "back-off" principle of congestion control.
- **Fix:** Strictly follow the TCP Reno or Tahoe algorithms. On a timeout, set `cwnd = 1*MSS` and enter Slow Start. On three duplicate ACKs (fast retransmit), set `ssthresh = cwnd/2` and `cwnd = ssthresh + 3*MSS` (Fast Recovery). Do not apply standard ACK-based `cwnd` increases while in Fast Recovery.

Implementation Guidance

This section provides concrete starter code and skeletons to implement TCP Data Transfer in C.

A. Technology Recommendations

Component	Simple Option (Educational)	Advanced Option (For Further Study)
Timer Management	Single periodic thread using `usleep()` or `nanosleep()`	Timer wheel with hierarchical buckets (hashed timing wheel)
Buffer Management	Simple circular byte buffer (`tcp_buffer`)	Linked list of packet buffers (mbufs) for zero-copy
Congestion Control	Basic TCP Reno (Slow Start, Congestion Avoidance, Fast Retransmit/Recovery)	TCP New Reno or CUBIC

B. Recommended File/Module Structure

Add the following files to the `src/` directory for TCP data transfer logic:

```
project-root/
  src/
    tcp/
      tcp.c           # Core TCP processing (input/output, state machine) - Extend Milestone 3 file
      tcp.h           # TCP public API and structures
      tcp_timer.c     # Timer tick implementation and retransmission callbacks
      tcp_timer.h
      tcp_buffer.c    # Circular buffer implementation for send/recv queues
      tcp_buffer.h
      tcp_cc.c        # Congestion control algorithms (Reno)
      tcp_cc.h
    main.c            # Main loop, now includes timer tick invocation
```

C. Infrastructure Starter Code

Complete Timer Tick Manager (`tcp_timer.c`):

```c
#include <stdint.h>
#include <time.h>
#include "tcp.h"

#define TIMER_TICK_MS 10  // 10ms tick interval

static struct timespec last_tick;

void tcp_timer_init() {
    clock_gettime(CLOCK_MONOTONIC, &last_tick);
}

void tcp_timer_tick() {
    struct timespec now;
    clock_gettime(CLOCK_MONOTONIC, &now);

    // Calculate elapsed milliseconds
    long elapsed_ms = (now.tv_sec - last_tick.tv_sec) * 1000 +
                      (now.tv_nsec - last_tick.tv_nsec) / 1000000;

    if (elapsed_ms < TIMER_TICK_MS) {
        return; // Not enough time has passed
    }

    last_tick = now;

    // Iterate through all active TCBs
    tcb_t *conn = tcp_get_connection_list(); // Assume a function to get the list head
    while (conn != NULL) {
        if (conn->state == ESTABLISHED || conn->state == FIN_WAIT_1 ||
            conn->state == FIN_WAIT_2 || conn->state == CLOSE_WAIT) {
            // Decrement RTO timer if running
            if (conn->rto_timer > 0) {
                conn->rto_timer -= elapsed_ms;
                if (conn->rto_timer <= 0) {
                    // Timer expired: trigger retransmission
                    retransmit_timer_callback(conn);
                }
            }
            // TODO: Decrement other timers (TIME_WAIT, etc.)
        }
        conn = conn->next;
    }
}
```

Circular Buffer Implementation (`tcp_buffer.c`):

```c
#include <stdlib.h>

#include <string.h>

#include "tcp_buffer.h"


int tcp_buffer_init(tcp_buffer_t *buf, size_t size) {

    buf->data = (uint8_t*)malloc(size);

    if (!buf->data) return -1;

    buf->size = size;

    buf->head = buf->tail = 0;

    buf->readable = 0;

    return 0;

}


void tcp_buffer_free(tcp_buffer_t *buf) {

    if (buf->data) free(buf->data);

    buf->data = NULL;

}


size_t tcp_buffer_write(tcp_buffer_t *buf, const uint8_t *data, size_t len) {

    size_t free_space = buf->size - buf->readable;

    if (len > free_space) len = free_space;

    if (len == 0) return 0;


    // Write in two parts if wrap-around is needed

    size_t to_end = buf->size - buf->tail;

    if (len <= to_end) {

        memcpy(buf->data + buf->tail, data, len);

        buf->tail = (buf->tail + len) % buf->size;

    } else {

        memcpy(buf->data + buf->tail, data, to_end);

        memcpy(buf->data, data + to_end, len - to_end);

        buf->tail = len - to_end;

    }

    buf->readable += len;

    return len;

}


size_t tcp_buffer_read(tcp_buffer_t *buf, uint8_t *out, size_t len) {

    if (len > buf->readable) len = buf->readable;

    if (len == 0) return 0;


    size_t to_end = buf->size - buf->head;

    if (len <= to_end) {
```

```
        memcpy(out, buf->data + buf->head, len);
        buf->head = (buf->head + len) % buf->size;
    } else {
        memcpy(out, buf->data + buf->head, to_end);
        memcpy(out + to_end, buf->data, len - to_end);
        buf->head = len - to_end;
    }
    buf->readable -= len;
    return len;
}
```

D. Core Logic Skeleton Code

Extended `tcb` Structure (`tcp.h`):

```c
typedef struct tcb {
    // ... Milestone 3 fields (local/remote IP/port, state, seq/ack numbers)
    uint32_t snd_nxt;        // Next sequence number to send
    uint32_t snd_una;        // Oldest unacknowledged sequence number
    uint32_t rcv_nxt;        // Next sequence number expected
    uint16_t snd_wnd;        // Send window (from receiver's advertised window)
    uint16_t rcv_wnd;        // Receive window (free space in rx_buffer)

    // Buffers
    tcp_buffer_t *tx_buffer; // Send buffer (holds app data not yet sent/acked)
    tcp_buffer_t *rx_buffer; // Receive buffer (holds in-order data not read by app)

    // Timer
    int32_t rto_timer;       // Milliseconds until RTO expiry (managed by timer tick)
    uint32_t srtt;           // Smoothed Round-Trip Time (microseconds)
    uint32_t rttvar;         // Round-Trip Time Variation
    uint32_t rto;            // Current Retransmission Timeout (microseconds)

    // Congestion Control
    uint32_t cwnd;           // Congestion window (bytes)
    uint32_t ssthresh;       // Slow start threshold (bytes)
    uint8_t dup_acks;        // Count of duplicate ACKs received

    // ... other fields (next pointer, etc.)
} tcb_t;
```

Usable Window Calculation (`tcp.c` - internal helper):

```c
static size_t tcp_send_window_available(const tcb_t *conn) {

    // TODO 1: Calculate the current flight size (bytes in transit)

    //   flight_size = conn->snd_nxt - conn->snd_una (use seq_diff)

    // TODO 2: Calculate the absolute window limit

    //   window_limit = min(conn->cwnd, conn->snd_wnd)

    // TODO 3: If flight_size >= window_limit, return 0

    // TODO 4: Otherwise, return window_limit - flight_size

    // Hint: Use seq_diff for sequence number arithmetic

    return 0;

}
```

Processing ACKs (`tcp.c` - internal function):

```c
static void tcp_ack_received(tcb_t *conn, uint32_t ack_num, uint16_t window) {

    // TODO 1: Validate ACK number (must advance snd_una)

    //   Use seq_lt to check: ack_num > snd_una && ack_num <= snd_nxt

    //   If invalid, return (but count duplicate ACK if ack_num == snd_una)



    // TODO 2: If this is a duplicate ACK (ack_num == snd_una):

    //   Increment conn->dup_acks

    //   If dup_acks == 3, trigger fast retransmit:

    //     - Retransmit segment at snd_una

    //     - Set ssthresh = max(cwnd/2, 2*MSS)

    //     - Set cwnd = ssthresh + 3*MSS (Fast Recovery)

    //   Return



    // TODO 3: New ACK: Slide send window

    //   conn->snd_una = ack_num

    //   Reset dup_acks = 0



    // TODO 4: Update flow control window

    //   conn->snd_wnd = window (converted from network byte order)



    // TODO 5: Update RTO estimator (if we have a valid RTT sample)

    //   (Advanced: Implement Jacobson/Karels algorithm)



    // TODO 6: Congestion control: Increase cwnd

    //   if (cwnd < ssthresh): Slow Start: cwnd += MSS (per ACK)

    //   else: Congestion Avoidance: cwnd += (MSS * MSS) / cwnd (per ACK)



    // TODO 7: Restart retransmission timer if there is still outstanding data

    //   if (snd_nxt > snd_una): restart timer for segment at snd_una



    // TODO 8: Try to send more queued data now that window has moved

    //   tcp_try_send(conn);

}
```

Retransmission Timer Callback (`tcp_timer.c`):

```c
void retransmit_timer_callback(tcb_t *conn) {

    // TODO 1: Exponential backoff: Double the RTO

    //   conn->rto = min(conn->rto * 2, MAX_RTO)



    // TODO 2: Congestion response

    //   conn->ssthresh = max(conn->cwnd / 2, 2 * MSS)

    //   conn->cwnd = 1 * MSS  (Back to Slow Start)



    // TODO 3: Retransmit the oldest unacknowledged segment

    //   Find the segment starting at conn->snd_una in your send buffer

    //   Call tcp_output to retransmit it (same seq_num, payload)



    // TODO 4: Restart the timer with the new RTO value

    //   conn->rto_timer = conn->rto / 1000; // Convert microseconds to milliseconds

}
```

E. Language-Specific Hints (C)

- **Sequence Number Arithmetic:** Always use the provided `seq_lt(a, b)` and `seq_diff(a, b)` functions. Never write `if (seq1 < seq2)` directly.
- **Network Byte Order:** Remember `window` fields in TCP headers are in network byte order. Use `ntohs()` when reading from a received header, `htons()` when writing to a header to send.
- **Timer Granularity:** The `usleep()` function has limited resolution (often ~10ms). For a more consistent tick, consider `clock_nanosleep()` with `CLOCK_MONOTONIC`.
- **Buffer Management:** The `tcp_buffer` starter code handles wrap-around. When calculating `rcv_wnd`, remember it's in *bytes* and should be expressed as a 16-bit value (max 65535). If your buffer is larger, you should scale the advertised window (e.g., advertise actual free space / 2).

F. Milestone 4 Checkpoint

After implementing the data transfer component, you should be able to perform end-to-end data transfer.

Verification Steps:

1. **Build and run** your stack with a simple echo server application.
2. **From another machine (or namespace),** use `netcat` to connect to your stack's IP address and port:

```bash
    echo "Hello TCP Stack" | nc -N 192.168.1.100 8080
```

`(Replace `192.168.1.100` with your stack's IP and `8080` with your listening port).`

3. **Expected Behavior:**
    - The connection should establish (SYN, SYN-ACK, ACK visible in Wireshark).
    - The data "Hello TCP Stack" should be sent in one or more TCP segments.
    - Your stack should receive the data, place it in the connection's `rx_buffer`, and send an ACK back.
    - Your echo server application should read the data from `rx_buffer` via `tcp_receive_data` and send it back via `tcp_send_data`.
    - The reply data should be transmitted, ACK'd by the client (`netcat`), and displayed on the client's terminal.
    - Typing `Ctrl+C` on the `netcat` client should initiate a graceful close (FIN exchange).
4. **Signs of Success:**
    - Complete two-way data transfer.
    - No packets are retransmitted (under good network conditions).
    - Wireshark shows proper sequence/ack number advancement and window updates.
5. **Common Failure Signs & Diagnostics:**
    - **Connection hangs after data is sent:** Check if ACKs are being generated and processed. Verify `tcp_ack_received` logic and window calculations. Use logging to print `snd_una`, `snd_nxt`, `cwnd`, and `snd_wnd`.

- **Data is corrupted or out of order:** Verify `seq_num` validation in `tcp_input`. Ensure `rx_buffer` is being written to at the correct offset (`seq_num - initial_rcv_nxt`).
- **Extremely slow transfer:** The sender may be stuck in a zero-window state. Check that `rcv_wnd` is being calculated and advertised correctly. Also verify that `cwnd` is growing appropriately (log its value).
- **Retransmissions occur on a lossless local network:** Your RTO may be too low. Ensure your RTO calculation has a sensible minimum (e.g., 1 second). Check for duplicate ACKs being mis-counted.

9. Interactions and Data Flow

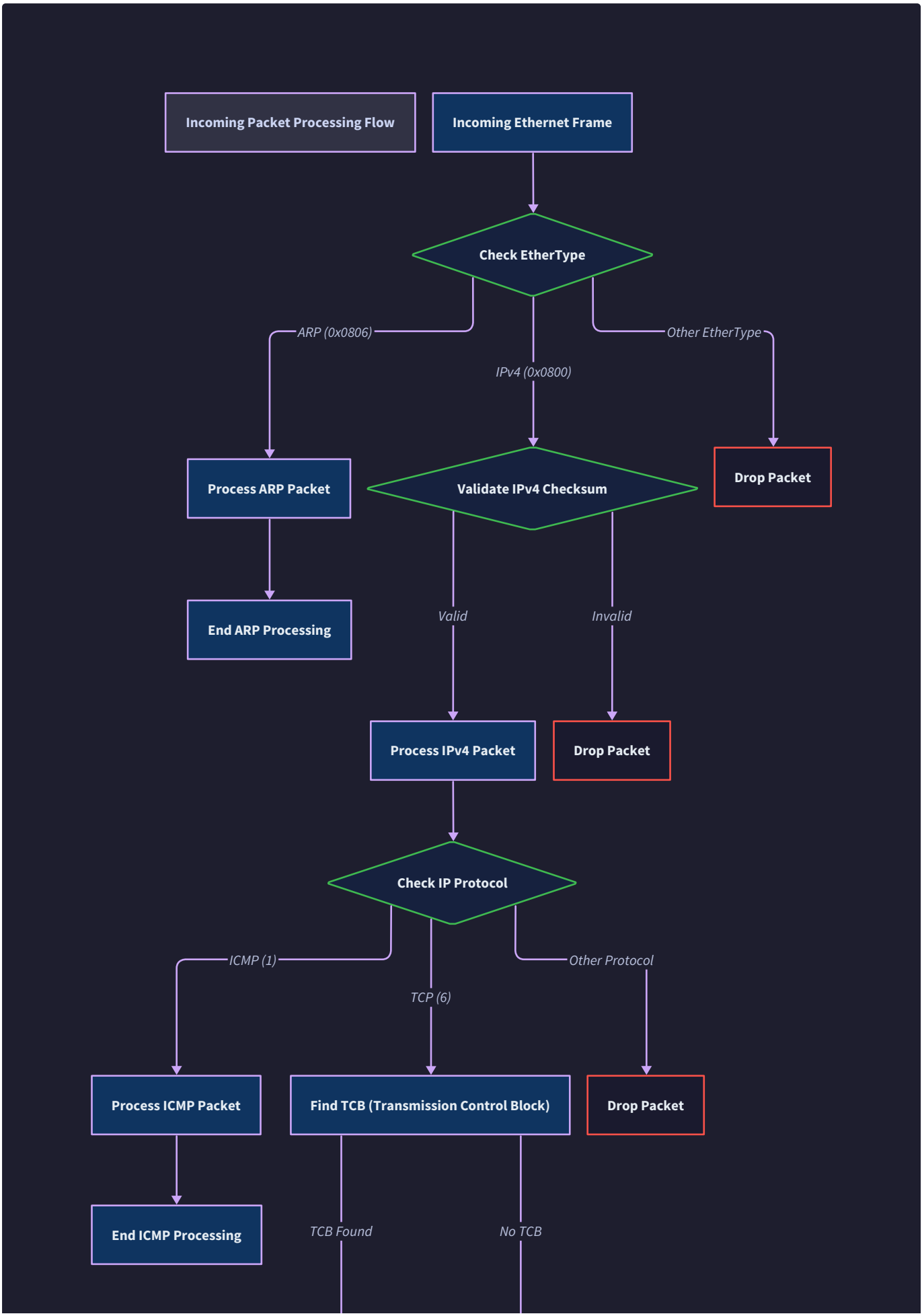
> **Milestone(s):** 1, 2, 3, 4 (Cross-Component Integration)

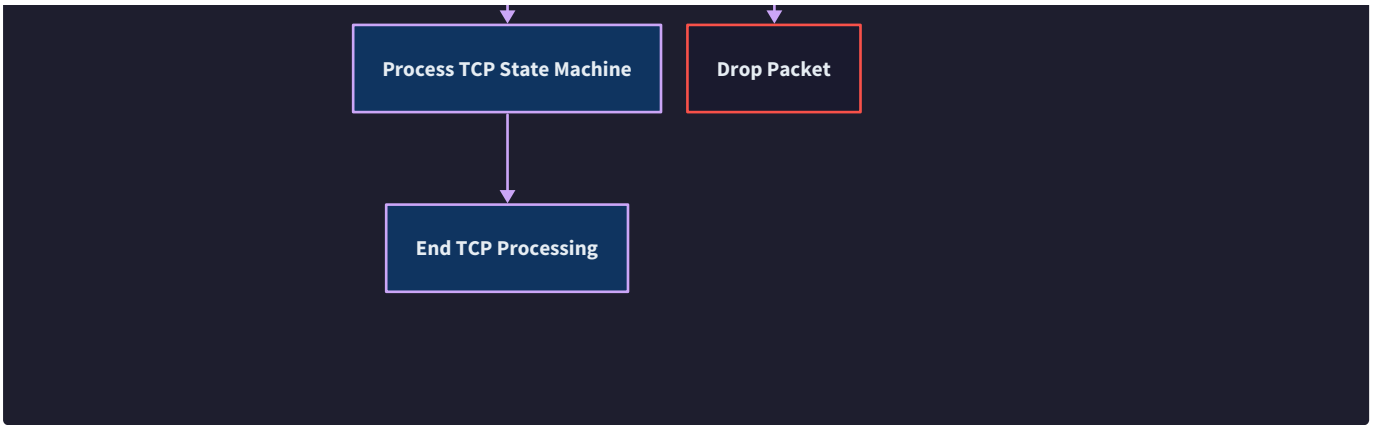
This section describes the dynamic behavior of Project Valiant—how the layered components interact as data flows through the system. Understanding these interactions is critical because the **TCP/IP stack** is fundamentally a data processing pipeline where each layer transforms and passes information to adjacent layers using well-defined interfaces. While previous sections detailed each component in isolation, here we examine the choreography that enables end-to-end communication. The core challenge is maintaining correct **encapsulation** and **decapsulation** while coordinating complex state changes across multiple independent state machines.

9.1 Packet Lifecycle: From Wire to Socket

> **Mental Model: Assembly Line Processing Plant**
>
> Imagine a package arriving at a sophisticated logistics facility. The package moves through successive stations, each staffed by specialists who perform specific tasks: the receiving dock (Link Layer) accepts raw shipments, the routing department (Network Layer) determines destination, the quality control team (Transport Layer) verifies contents and order, and finally the delivery office (Application) hands the item to the intended recipient. Each station removes one layer of wrapping, checks its paperwork, and passes the inner package to the next station. This unidirectional flow—with occasional return trips for acknowledgments or error notifications—mirrors how packets ascend through the stack.

The journey of an incoming **TCP segment** (packet containing application data) from the physical network to the application's receive buffer illustrates the layered processing paradigm. The following numbered steps trace this path, referencing the flowchart in


Incoming Packet Processing Flow
Incoming Ethernet Frame
Check EtherType
ARP (0x0806)
IPv4 (0x0800)
Other EtherType
Process ARP Packet
Validate IPv4 Checksum
Drop Packet
End ARP Processing
Valid
Invalid
Process IPv4 Packet
Drop Packet
Check IP Protocol
ICMP (1)
TCP (6)
Other Protocol
Process ICMP Packet
Find TCB (Transmission Control Block)
Drop Packet
End ICMP Processing
TCB Found
No TCB



.

1. **Physical Reception at TAP Device**: The kernel's virtual network driver delivers a complete **Ethernet frame** to the TAP device file descriptor. Our stack's main event loop calls `tap_read()` which copies up to `MAX_FRAME_SIZE` bytes (typically 1600) into a buffer. The frame arrives in **Network Byte Order** (big-endian).
2. **Link Layer Processing**: The `netif_rx_packet()` function receives the raw frame. It first validates basic structure (minimum 14-byte header). It parses the `eth_hdr` to extract:
   - `dst_mac`: Destination MAC address (6 bytes)
   - `src_mac`: Source MAC address (6 bytes)
   - `ethertype`: Protocol identifier (2 bytes)

   If the `dst_mac` doesn't match our interface's MAC address and isn't the **broadcast MAC** (`FF:FF:FF:FF:FF:FF`), the frame is silently dropped (unless in promiscuous mode). Valid frames proceed to protocol dispatch based on `ethertype`.
3. **EtherType Dispatch**: The `ethertype` field determines the next handler:
   - `ETHERTYPE_ARP` (0x0806): Frame passed to ARP processor (see Section 9.2.1)
   - `ETHERTYPE_IP` (0x0800): Frame passed to `ipv4_input()`
   - Unknown ethertype: Frame dropped
4. **Network Layer Processing**: For IP packets, `ipv4_input()` receives the frame's payload (everything after the 14-byte Ethernet header). It:
   - Parses the `ip_hdr` structure, extracting critical fields: `version_ihl`, `total_len`, `ttl`, `protocol`, `checksum`, `src_ip`, `dst_ip`
   - Validates the header checksum using `ip_checksum()`—if mismatch, packet dropped
   - Checks if `dst_ip` matches any of our interface addresses; if not, packet might be forwarded (simplified stack typically drops)
   - Decrements `ttl` by 1; if `ttl` becomes 0, generates an **ICMP** "Time Exceeded" message and drops packet
   - Checks `protocol` field for next-layer dispatch
5. **IP Protocol Dispatch**: The `protocol` field determines transport handler:
   - `IPPROTO_ICMP` (1): Packet passed to `icmp_input()`
   - `IPPROTO_TCP` (6): Packet passed to `tcp_input()`
   - Unknown protocol: Packet dropped (could send ICMP "Protocol Unreachable")
6. **Transport Layer TCP Processing**: `tcp_input()` receives the IP payload (TCP segment). It:
   - Parses the `tcp_hdr` structure, extracting: `src_port`, `dst_port`, `seq_num`, `ack_num`, flags, `window`, `checksum`
   - Computes the **TCP checksum** using `tcp_checksum()` with the pseudo-header (includes IP addresses); drops segment if invalid
   - Calls `tcp_find_connection()` to locate the matching `tcb` using the 4-tuple (`src_ip`, `src_port`, `dst_ip`, `dst_port`)
   - If no matching TCB exists and the `RST` flag isn't set, behavior depends on state: for `LISTEN` sockets, a new connection is created; otherwise, a `RST` is sent back
   - If TCB exists, calls `tcp_process()` to run the **TCP state machine** with the received segment
7. **TCP State Machine Execution**: `tcp_process()` executes the appropriate actions based on current `tcb->state` and received flags:
   - Updates `rcv_nxt` and `snd_una` based on sequence numbers
   - Handles `SYN`, `ACK`, `FIN`, `RST` flags per **RFC 793** rules
   - For data-carrying segments in `ESTABLISHED` state, places payload into `tcb->rx_buffer` via `tcp_buffer_write()`
   - Adjusts `rcv_wnd` based on buffer availability
   - Triggers ACK transmission if needed (immediate or delayed)
8. **Application Data Delivery**: When the application calls `tcp_receive_data()`, it reads from the `tcb->rx_buffer` (implemented as a `tcp_buffer_t` circular buffer). Data is copied to the application's buffer, `rcv_nxt` is advanced, and window updates may be scheduled.

The reverse path for outgoing data follows symmetric steps in the opposite direction (down the stack), with each layer adding its header via encapsulation:

```
Application → tcp_send_data() → TCP segment construction → ipv4_output() →
IP packet construction → arp_resolve() → Ethernet frame construction → tap_write()
```

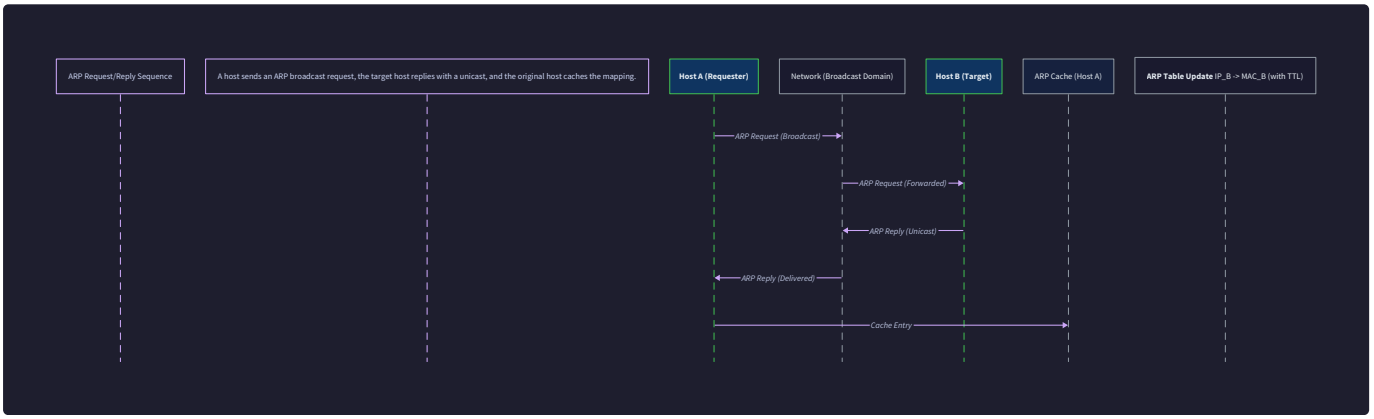
9.2 Key Control Flows

While the packet lifecycle describes the generic processing path, these three specific sequences illustrate critical protocol interactions that span multiple layers and involve bidirectional communication.

9.2.1 ARP Resolution Sequence

> **Mental Model: Asking for Directions in a Neighborhood**
>
> You need to deliver a package to "123 Main Street" (an IP address), but you only know the street address, not the specific house location (MAC address). You shout to the entire neighborhood (broadcast), "Who lives at 123 Main Street?" The resident hears this, recognizes their address, and calls back (unicast), "I'm at 123 Main Street—here's my exact location (MAC address)." You note this mapping in your address book (ARP cache) for future deliveries. This query/response protocol happens entirely at the link layer, using broadcasts that don't cross router boundaries.

ARP (Address Resolution Protocol) resolves IPv4 addresses to MAC addresses for local network delivery. The sequence diagram



illustrates this exchange:

1. **Trigger**: The stack needs to send an **IP packet** to a destination on the local network (determined by routing table). `ipv4_output()` calls `arp_resolve()` with the target IP.
2. **Cache Lookup**: `arp_resolve()` first checks the **ARP cache** via `arp_cache_lookup()`. If a valid entry exists (state `ARP_ENTRY_RESOLVED` and not expired), it immediately returns the MAC address.
3. **Cache Miss - Request Creation**: If no entry exists, `arp_resolve()`:
   - Creates an `arp_entry` with state `ARP_ENTRY_FREE` (or marks existing entry as pending)
   - Constructs an **ARP request** packet: `opcode` = `ARP_OP_REQUEST`, `sender_ip` = our IP, `target_ip` = IP to resolve, `target_mac` = all zeros
   - Builds a complete **Ethernet frame** with `dst_mac` = broadcast MAC, `ethertype` = `ETHERTYPE_ARP`
   - Sends frame via `netif_tx_packet()` → `tap_write()`
   - Starts a retransmission timer (typically 1 second) in case of no response
4. **Request Reception**: All hosts on the local network receive the broadcast frame. Each host's `netif_rx_packet()` sees `ETHERTYPE_ARP` and passes to ARP handler, which:
   - Parses the `arp_hdr`
   - If `opcode` is `ARP_OP_REQUEST` and `target_ip` matches the host's IP address, proceeds to reply
5. **Reply Generation**: The target host:
   - Creates an **ARP reply** packet: `opcode` = `ARP_OP_REPLY`, swaps sender/target fields (sender becomes the original target)
   - Builds an **Ethernet frame** with `dst_mac` = requester's MAC (from request packet), `ethertype` = `ETHERTYPE_ARP`
   - Sends unicast reply via `netif_tx_packet()`
6. **Reply Processing**: Original requester receives the unicast reply, validates it matches a pending request, and:
   - Updates **ARP cache** via `arp_cache_insert()` with IP→MAC mapping
   - Changes entry state to `ARP_ENTRY_RESOLVED`, sets `last_updated` timestamp
   - Completes the pending IP packet transmission using the resolved MAC

7. **Cache Maintenance**: The **ARP cache** periodically ages out entries (default 120 seconds) to handle network changes. Expired entries revert to `ARP_ENTRY_FREE` state and require re-resolution if needed.

Key Design Insight: ARP resolution must complete before any IP packet can be transmitted to a local destination. This creates a natural queueing system where pending packets wait for ARP resolution, with the first packet triggering the request and subsequent packets queueing behind it.

9.2.2 ICMP Ping (Echo Request/Reply)

> **Mental Model: Sonar Ping in a Submarine**
>
> A submarine sends a sound pulse (echo request) through the water and listens for the reflection (echo reply). The time between sending and receiving indicates distance and confirms the target is reachable. ICMP ping operates similarly—it's a diagnostic tool that tests basic IP connectivity without involving higher-layer protocols like TCP.

ICMP (Internet Control Message Protocol) echo request/reply (commonly "ping") provides network diagnostic capabilities. The flow involves both Network and Link layers:

1. **External Trigger**: Another host (or the same host via loopback) sends an **ICMP Echo Request** to our IP address. The packet arrives as an IP packet with `protocol` = `IPPROTO_ICMP`.
2. **Request Processing**: `ipv4_input()` validates the IP header, passes payload to `icmp_input()`, which:
   - Parses the `icmp_hdr` structure
   - If `type` = 8 (Echo Request) and `code` = 0, calls `icmp_echo_reply()`
3. **Reply Generation**: `icmp_echo_reply()`:
   - Swaps `src_ip` and `dst_ip` from the request
   - Sets ICMP `type` = 0 (Echo Reply), preserves `identifier` and `sequence` fields
   - Copies the request's payload data unchanged
   - Computes new ICMP checksum over the entire ICMP message
   - Calls `ipv4_output()` with `protocol` = `IPPROTO_ICMP` to send the reply
4. **Reply Transmission**: `ipv4_output()` performs normal IP processing:
   - Checks routing table via `route_lookup()` for destination IP
   - If local network, resolves MAC via ARP (may trigger ARP resolution as above)
   - Constructs IP header with `ttl` typically 64, computes IP checksum
   - Passes to link layer for transmission
5. **Completion**: The original sender receives the Echo Reply, matching it to the outstanding request based on `identifier` and `sequence`, and calculates round-trip time.

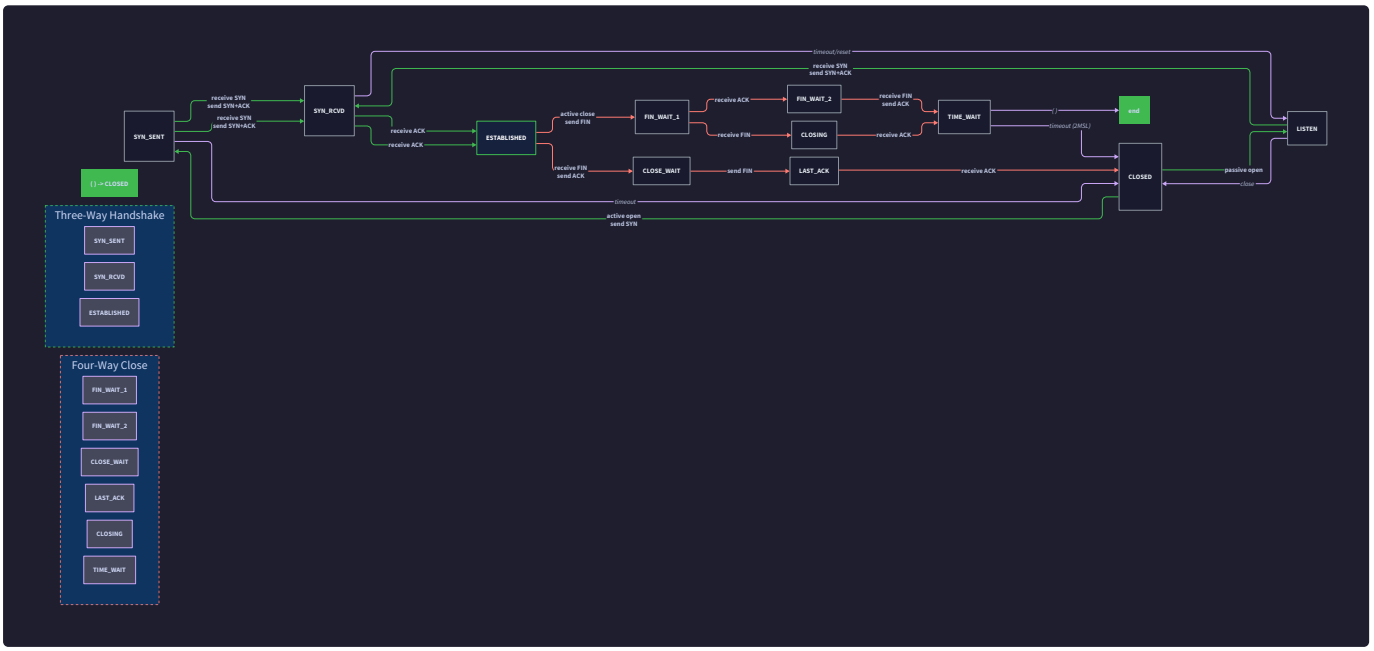
> **Important Note**: ICMP messages are carried directly in IP packets (protocol 1), not over TCP or UDP. Error messages like "Destination Unreachable" follow similar paths but are generated internally by the stack when conditions warrant (e.g., no route, port unreachable).

9.2.3 TCP Connection Establishment, Data Transfer, and Closure

> **Mental Model: Business Meeting Protocol**
>
> 1. **Connection Establishment**: You call a colleague (SYN), they answer and confirm they can talk (SYN-ACK), you acknowledge their confirmation (ACK). The meeting is now officially in session (ESTABLISHED).
> 2. **Data Transfer**: You present information in ordered segments, they acknowledge receipt. You adjust your speaking pace based on their ability to take notes (flow control) and room capacity (congestion control).
> 3. **Connection Closure**: You signal you're done speaking (FIN), they acknowledge and finish their thoughts (ACK + FIN), you acknowledge their closure (ACK). Both parties hang up.

The complete **TCP** lifecycle involves complex state coordination across all layers. The sequence diagram

shows the message exchange, while the state machine



governs internal transitions.

Phase 1: Connection Establishment (Three-Way Handshake)

1. **Active Open (Client)**: Application calls `tcp_connect(remote_ip, remote_port, local_port)`:
    - Creates a new `tcb` with initial state `CLOSED`
    - Selects an initial sequence number (`ISS`) for `snd_nxt`
    - Transitions to `SYN_SENT` state
    - Calls `tcp_output()` with `flags` = `TCP_SYN`
    - `tcp_output()` builds TCP segment with `seq_num` = `ISS`, `ack_num` = 0
    - Segment passes down stack: `ipv4_output()` → (ARP if needed) → `netif_tx_packet()`
2. **Passive Open (Server)**: Application calls `tcp_listen(local_port)`:
    - Creates a `tcb` in `LISTEN` state waiting for incoming SYN
    - When SYN arrives at `tcp_input()`, `tcp_find_connection()` matches the listening socket
    - Creates a new `tcb` for this connection in `SYN_RCVD` state
    - Selects its own initial sequence number (`ISS`)
    - Sends `SYN-ACK` via `tcp_output()` with `flags` = `TCP_SYN | TCP_ACK`, `seq_num` = its `ISS`, `ack_num` = client's `ISS + 1`
3. **Handshake Completion**: Client receives `SYN-ACK`:

- Validates `ack_num` equals its `ISS + 1`
- Transitions to `ESTABLISHED` state
- Sends `ACK` via `tcp_output()` with `flags` = `TCP_ACK`, `seq_num` = `ISS + 1`, `ack_num` = server's `ISS + 1`
- Server receives `ACK`, validates it, transitions to `ESTABLISHED`

Phase 2: Data Transfer with Reliability Mechanisms

4. **Data Sending**: Application calls `tcp_send_data(connection, data, len)`:
   - Data is queued in `tcb->tx_buffer` (circular buffer)
   - `tcp_output()` is called to send available data within the **usable window** (`min(cwnd, rcv_wnd) - bytes_in_flight`)
   - Each segment gets a unique sequence number range, `TCP_ACK` flag set, and includes receiver's `rcv_nxt` as `ack_num`
   - Retransmission timer starts for each unacknowledged segment
5. **Data Reception & Acknowledgment**: Receiver processes incoming data segment:
   - Validates sequence number falls within acceptable window
   - Places data in `tcb->rx_buffer` in correct order (handles out-of-order via queueing)
   - Advances `rcv_nxt` by payload length
   - Schedules `ACK` (immediate for out-of-order, may delay for in-order to enable piggybacking)
   - Updates advertised window (`rcv_wnd`) based on buffer availability
6. **ACK Processing**: Sender receives `ACK`:
   - `tcp_ack_received()` updates `snd_una` to the highest cumulatively acknowledged sequence
   - Slides the **sliding window** forward, freeing buffer space
   - Cancels retransmission timer for acknowledged data
   - Adjusts **congestion window** (`cwnd`) based on **slow start** or **congestion avoidance** algorithm
7. **Loss Recovery**: If retransmission timer expires before ACK:
   - `retransmit_timer_callback()` retransmits oldest unacknowledged segment
   - Sets **ssthresh** to half of current flight size, `cwnd` to 1 MSS (multiplicative decrease)
   - Re-enters **slow start** phase

Phase 3: Connection Termination (Four-Way Handshake)

8. **Active Close (Initiator)**: Application calls `tcp_close(connection)`:
   - If in `ESTABLISHED`, transitions to `FIN_WAIT_1`
   - Sends `FIN` via `tcp_output()` with `flags` = `TCP_FIN | TCP_ACK`
   - Waits for corresponding `ACK`
9. **Passive Close (Responder)**: Receives `FIN`:
   - Transitions to `CLOSE_WAIT`, delivers EOF to application
   - Sends `ACK` for the `FIN`
   - When application calls `tcp_close()`, sends its own `FIN`, transitions to `LAST_ACK`
10. **Handshake Completion**: Initiator receives `ACK` for its `FIN`:
    - Transitions to `FIN_WAIT_2`
    - Receives other side's `FIN`, sends final `ACK`, transitions to `TIME_WAIT`
    - After `TIME_WAIT` timeout (2MSL), transitions to `CLOSED`, frees `tcb`

> **Critical Insight**: The **TCP state machine** coordinates these phases independently for each connection. Events (segment arrival, user calls, timer expiration) trigger state transitions with specific actions. The stack must handle simultaneous close (both sides send FIN independently) and abrupt resets (`RST` flag) that bypass the graceful closure process.

9.3 Cross-Layer Dependencies and Timing Considerations

The control flows reveal several critical cross-layer dependencies:

1. **ARP-TCP Dependency**: A TCP handshake cannot begin until the destination MAC is resolved via ARP. This creates a potential deadlock if ARP requests fail (target offline).
2. **Routing-IP Dependency**: Every outgoing IP packet requires a routing decision via `route_lookup()`. The route determines whether the packet is for the local network (direct ARP) or requires a gateway (ARP for gateway MAC).

3. **Timer Interactions**: Multiple timers operate concurrently:
    - ARP cache expiration (minutes)
    - TCP retransmission timeout (milliseconds to seconds)
    - TCP keepalive (optional, minutes)
    - TIME_WAIT timeout (minutes)
4. **Buffer Management Flow Control**: When `tcb->rx_buffer` fills, `rcv_wnd` shrinks, which flow-controls the sender. If the application doesn't read data, eventually `rcv_wnd` becomes 0, stalling the connection until buffer space frees.
5. **Error Propagation**: Link layer errors (failed transmission) typically result in silent drops. Network layer errors (TTL expiry, no route) generate ICMP messages back to source. Transport layer errors (invalid checksum, no listener) may generate TCP `RST` or ICMP port unreachable.

These interactions create emergent behavior where the stack self-regulates based on network conditions, receiver capacity, and application behavior—all coordinated through the layered design with well-defined interfaces between components.

9.4 Common Pitfalls in Component Interactions

⚠️ Pitfall: Missing ARP Resolution Before IP Transmission

- **Description**: Attempting to send an IP packet without first checking/obtaining the destination MAC address, resulting in dropped packets.
- **Why It's Wrong**: Ethernet frames require valid destination MACs; IP packets alone cannot traverse the link layer.
- **Fix**: Always call `arp_resolve()` (which checks cache, sends request if needed) before `netif_tx_packet()`. Queue packets waiting for ARP resolution.

⚠️ Pitfall: Incorrect Layered Checksum Validation

- **Description**: Validating TCP checksum without including the pseudo-header, or validating IP checksum on a header that includes options incorrectly.
- **Why It's Wrong**: RFC-compliant checksums detect corruption; incorrect validation accepts corrupt packets or rejects valid ones.
- **Fix**: Use `tcp_checksum()` with pseudo-header parameters. Use `ip_checksum()` on header only (not payload).

⚠️ Pitfall: State Machine Transition Race Conditions

- **Description**: Processing events (incoming segments, timer expirations, application calls) in wrong order due to lack of synchronization.
- **Why It's Wrong**: TCP requires strict ordering (e.g., processing old duplicate ACK after new ACK can cause incorrect window advancement).
- **Fix**: Use per-connection locking or ensure single-threaded event processing. Maintain strict event queue ordering.

⚠️ Pitfall: Ignoring ICMP Error Messages

- **Description**: Not processing received ICMP messages (like "Destination Unreachable") that indicate connection problems.
- **Why It's Wrong**: ICMP provides critical network feedback; ignoring it causes connections to hang when routes fail.
- **Fix**: Implement basic ICMP error processing in `tcp_input()` to reset connections upon receipt of hard errors.

⚠️ Pitfall: Buffer Deadlock Between Layers

- **Description**: Application not reading receive data → TCP receive buffer fills → window size shrinks to 0 → sender stops → application expects more data but sender is flow-controlled.
- **Why It's Wrong**: Creates apparent hang where data exists but cannot flow due to backpressure.
- **Fix**: Implement proper application notification (callback/select) when data arrives. Document that applications must read data to keep window open.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Event Loop	Single-threaded `poll()` on TAP fd	Multi-threaded with work queues
Timer Management	Linked list of timers checked each loop	Hierarchical timer wheel
Buffer Management	Fixed-size circular buffers	Dynamic buffer pooling with watermarks

Recommended Integration Points:

The main integration occurs in the event loop that ties all layers together. A typical implementation organizes files as:

```
project-valiant/
  src/
    main.c                    # Event loop, initialization
    net/
      tap.c                   # TAP device I/O (tap_open, tap_read, tap_write)
      ethernet.c              # netif_rx_packet, netif_tx_packet, eth_build_frame
      arp.c                   # ARP cache, arp_resolve, arp_cache_*
    ip/
      ipv4.c                  # ipv4_input, ipv4_output, routing
      icmp.c                  # icmp_input, icmp_echo_reply
    tcp/
      tcp.c                   # tcp_input, tcp_output, state machine
      tcp_timer.c             # Timer management
      tcp_buffer.c            # Circular buffer implementation
    include/
      net.h                   # Common headers, constants
      tcp.h                   # TCB structure, TCP APIs
```

Event Loop Skeleton:

```c
// main.c - Simplified event loop showing integration points

#include "net/tap.h"

#include "net/ethernet.h"

#include "tcp/tcp_timer.h"


#define MAX_EVENTS 10

#define TIMER_TICK_MS 10


int main(int argc, char *argv[]) {

    int tap_fd = tap_open("tap0");

    uint8_t frame_buffer[MAX_FRAME_SIZE];



    // Initialize all components

    arp_cache_init();

    routing_table_init();

    tcp_timer_init();



    struct pollfd fds[1];

    fds[0].fd = tap_fd;

    fds[0].events = POLLIN;



    uint64_t last_timer_tick = get_current_time_ms();



    while (1) {

        int timeout = TIMER_TICK_MS;



        // Calculate next timer tick

        uint64_t now = get_current_time_ms();

        if (now - last_timer_tick >= TIMER_TICK_MS) {

            tcp_timer_tick();  // Process TCP timers

            arp_cache_cleanup(); // Clean expired ARP entries

            last_timer_tick = now;

            timeout = 0;  // Process immediately after timer

        }



        // Wait for packet or timer

        int ret = poll(fds, 1, timeout);



        if (ret > 0 && (fds[0].revents & POLLIN)) {

            // Packet received from TAP

            ssize_t len = tap_read(tap_fd, frame_buffer, MAX_FRAME_SIZE);
```

```
            if (len > 0) {
                // Entry point to Link Layer
                netif_rx_packet(frame_buffer, len);
            }
        }

        // Check for pending transmissions (simplified)
        // In full implementation, check for queued packets waiting for ARP, etc.
    }

    return 0;
}
```

Cross-Layer Helper Functions:

```c
// net/ethernet.c - Integration between ARP and IP layers

int netif_tx_packet(uint8_t *frame, size_t len) {

    // TODO 1: Validate frame length (>= ETH_HDR_LEN, <= MAX_FRAME_SIZE)

    // TODO 2: Call tap_write() with frame and len

    // TODO 3: Return success/failure based on tap_write result

    // TODO 4: Log transmission for debugging (optional)

}


// ip/ipv4.c - Integration between IP and transport layers

int ipv4_output(uint32_t dst_ip, uint8_t protocol, uint8_t *payload, size_t payload_len) {

    // TODO 1: Look up route for dst_ip using route_lookup()

    // TODO 2: Determine next_hop_ip (dst_ip if local, gateway if remote)

    // TODO 3: Resolve next_hop_ip to MAC using arp_resolve()

    //         (this may block/queue if ARP pending)

    // TODO 4: Build IP header with correct fields:

    //         - version_ihl = 0x45 (IPv4, 5*4=20 byte header)

    //         - total_len = IP_HDR_LEN + payload_len

    //         - ttl = 64 (typical default)

    //         - protocol = protocol parameter

    //         - src_ip = our interface IP

    //         - dst_ip = dst_ip parameter

    // TODO 5: Compute checksum using ip_checksum()

    // TODO 6: Build Ethernet frame with:

    //         - dst_mac = resolved MAC from ARP

    //         - src_mac = our MAC

    //         - ethertype = ETHERTYPE_IP

    // TODO 7: Call netif_tx_packet() with complete frame

}
```

Milestone Integration Checkpoints:

After Milestone 1 (ARP/Ethernet):

- Run: `sudo ./valiant &` (starts stack with TAP device)
- From another terminal: `ping -I tap0 10.0.0.2` (if stack IP is 10.0.0.2)
- Expected: Wireshark on `tap0` shows ARP request from ping, ARP reply from stack, then ICMP echo requests and replies.
- Debug: If no ARP reply, check MAC address matching in `netif_rx_packet()`.

After Milestone 2 (IP/ICMP):

- Same ping test should work without ARP issues (cached).
- Run: `tcpdump -i tap0 icmp` should show bidirectional ICMP traffic.
- Test routing: Configure stack with IP 10.0.0.1/24, host with 10.0.0.100/24, ping from host to stack IP.

After Milestone 3 (TCP Connection):

- In one terminal: `./valiant --listen 8080` (implements simple echo server)
- In another: `nc 10.0.0.2 8080` (netcat client)
- Expected: TCP three-way handshake visible in Wireshark, connection establishes.
- Type text in netcat, should echo back (requires basic data transfer).

After Milestone 4 (TCP Data Transfer):

- Use `iperf` or custom test to transfer large file via stack.
- Monitor with Wireshark for sliding window behavior, ACKs, retransmissions if induced.
- Test flow control: Send data faster than application reads, verify window shrinks to 0.

Debugging Interactions:

Symptom	Likely Cause	How to Diagnose	Fix
ARP works, ping doesn't reply	IP checksum error or wrong IP matching	Capture packets, verify IP checksum in received packets matches `ip_checksum()` calculation	Fix `ip_checksum()` algorithm or IP address comparison
TCP SYN sent, no SYN-ACK	No listening socket or wrong demultiplexing	Check `tcp_find_connection()` logic for `LISTEN` sockets. Verify TCP checksum.	Ensure listening TCB created with `tcp_listen()`, proper 4-tuple matching
Connection establishes but no data transfer	Window size = 0 or buffer full	Log `rcv_wnd` in incoming ACKs. Check `tcp_buffer_write()` success.	Ensure application reads data, buffers sized appropriately
Retransmissions but no ACKs	ACKs lost or ACK processing bug	Check if ACKs are being sent (Wireshark). Verify `tcp_ack_received()` updates `snd_una`.	Fix ACK generation or processing logic

---

10. Error Handling and Edge Cases

> **Milestone(s):** 1, 2, 3, 4 (Error handling is integral to every layer and milestone)

A robust network stack must gracefully handle the inherent unreliability of network communication while maintaining stability in the face of malformed inputs and resource constraints. Unlike typical application development where inputs are often trusted or validated at boundaries, a TCP/IP stack operates in a hostile environment where every byte received could be intentionally malformed, accidentally corrupted, or completely unexpected. This section catalogs the expected failure modes across all protocol layers and details the stack's strategies for detection, containment, and recovery.

Failure Modes & Detection

Network communication involves multiple points of potential failure: physical corruption during transmission, routing failures, protocol violations, resource exhaustion, and malicious attacks. Project Valiant adopts a **defensive, fail-safe approach**—when in doubt, drop the packet and minimize state changes. The following table enumerates the primary failure modes across layers, their detection mechanisms, and the prescribed recovery actions.

Failure Mode	Layer(s) Affected	Detection Mechanism	Immediate Action	Recovery/Follow-up
Corrupt Frame (CRC Error)	Link	Hardware/Firmware (TAP device)	Frame dropped before reaching stack	N/A – Physical layer issue
Malformed Ethernet Header	Link	`netif_rx_packet()` validates: length ≥ `ETH_HDR_LEN`, known `ethertype`	Drop frame, increment error counter	Log warning for debugging
Unknown EtherType	Link	`ethertype` not in { `ETHERTYPE_IP`, `ETHERTYPE_ARP` }	Drop frame	Could log for protocol discovery
ARP Cache Poisoning Attempt	Link	`arp_input()` detects unsolicited reply for IP we don't own, or MAC mismatch for existing entry	Ignore or overwrite based on policy	Default: reject unsolicited updates
ARP Request for Non-local IP	Link	`arp_input()` checks `target_ip` against local interface IPs	Drop request (don't reply)	N/A – not our responsibility
IP Header Checksum Mismatch	Network	`ipv4_input()` computes `ip_checksum()` on received header, compares with `checksum` field	Drop packet, increment error counter	N/A – corrupted in transit
IP Version Not 4	Network	`ip_hdr->version_ihl` high nibble ≠ 4	Drop packet	Could support IPv6 in future
IP Header Length Invalid	Network	`(ip_hdr->version_ihl & 0x0F) < 5` (minimum 5 words)	Drop packet	N/A – malformed header
IP Packet Too Short	Network	`total_len < IP_HDR_LEN`	Drop packet	N/A – truncated packet
TTL Expired	Network	`ip_hdr->ttl == 0` after decrement (for forwarded packets)	Drop packet, may send ICMP Time Exceeded (Type 11)	N/A – routing loop prevention
No Route to Destination	Network	`route_lookup()` returns NULL for `dst_ip`	Drop packet, may send ICMP Destination Unreachable (Type 3, Code 0)	N/A – routing table incomplete
IP Fragmentation Needed	Network	Packet size > interface MTU and DF (Don't Fragment) flag set	Drop packet, send ICMP Fragmentation Needed (Type 3, Code 4) with next-hop MTU	Sender may retry with smaller packets
Unknown IP Protocol	Network	`ip_hdr->protocol` not in { `IPPROTO_ICMP`, `IPPROTO_TCP` }	Drop packet	Could log for protocol support
ICMP Checksum Mismatch	Network	`icmp_input()` computes checksum, compares with `icmp_hdr->checksum`	Drop ICMP message	N/A – corrupted in transit
Unexpected ICMP Type/Code	Network	`icmp_hdr->type` not expected (e.g., not Echo Request for ping)	Drop ICMP message	Some types could be logged
TCP Checksum Mismatch	Transport	`tcp_input()` computes `tcp_checksum()` with pseudo-header, compares with `tcp_hdr->checksum`	Drop segment, increment error counter	N/A – corrupted in transit
TCP Segment Length Mismatch	Transport	IP payload length < `TCP_HDR_LEN` or < calculated header length (data offset)	Drop segment	N/A – malformed segment
No Matching TCB (Connection)	Transport	`tcp_find_connection()` returns NULL for incoming segment	If RST flag not set, send RST segment back	Protects against stray packets
Invalid TCP State Transition	Transport	`tcp_process()` state machine: event not allowed in current `tcb->state`	Send RST segment (if incoming segment has ACK), otherwise drop	Reset connection to clean state
TCP RST Received	Transport	`tcp_hdr->flags & TCP_RST` is set	Immediately free TCB, notify application	Connection forcibly terminated
Sequence Number Out of Window	Transport	`seq_diff(seq_num, rcv_nxt)` outside receive window	Send ACK with current `rcv_nxt` (no data acceptance)	Protects against old duplicate segments
Acknowledgment Number Invalid	Transport	`ack_num` not between `snd_una` and `snd_nxt` (not acknowledging sent data)	Send ACK with current `snd_nxt`	May indicate buggy peer
Retransmission Timeout (RTO)	Transport	`retransmit_timer_callback()` fires for unacknowledged data	Retransmit oldest unacknowledged segment, reduce congestion window	Exponential backoff of RTO timer

Failure Mode	Layer(s) Affected	Detection Mechanism	Immediate Action	Recovery/Follow-up
Persistent Timer Expiry	Transport	Keep-alive or time-wait timer expires	Close connection, free TCB	Cleanup of stale connections
Receive Buffer Full	Transport	`tcp_buffer_write()` fails (no space in `rx_buffer`)	Advertise `rcv_wnd = 0` in outgoing ACKs	Application must read data to free space
Send Buffer Full	Transport	`tcp_send_data()` returns error (no space in `tx_buffer`)	Application should wait/retry	Flow control prevents buffer overflow
Resource Exhaustion	All	`malloc()` fails, TCB table full, ARP cache full	Drop packet/connection, return error to caller	Implement graceful degradation

> **Key Insight:** The principle of **"be conservative in what you send, liberal in what you accept"** (Postel's Law) guided early Internet protocol design. However, for security and robustness, modern implementations must balance this with **defensive validation**—accepting only well-formed packets that strictly conform to protocol specifications and expected state.

Building Robustness

Beyond reacting to specific failures, Project Valiant incorporates several architectural patterns and implementation practices to build systemic robustness against both accidental errors and intentional attacks.

Defensive Parsing Strategy

Each protocol layer implements a **validation pipeline** that progresses from basic structural checks to semantic validation, rejecting malformed packets as early as possible:

1. **Length Validation First:** Before accessing any header field, verify the packet contains at least the minimum required bytes for that layer's header. For example, `netif_rx_packet()` must check `len >= ETH_HDR_LEN` before casting to `eth_hdr*`.
2. **Bounds Checking with Pointer Arithmetic:** When calculating header lengths from variable-length fields (like IP header length in 32-bit words), ensure the calculated length doesn't exceed the total packet length and doesn't underflow.
3. **Reserved Field Verification:** Check that reserved bits/fields are zero (as required by RFCs). Non-zero values in reserved fields may indicate a buggy or malicious sender.
4. **Semantic Range Checking:** Validate that numeric values fall within plausible ranges—for example, TCP source/destination ports should be > 0, IP `total_len` should be ≥ header length and ≤ `MAX_FRAME_SIZE - ETH_HDR_LEN`.
5. **State-Dependent Validation:** In TCP, the validity of sequence numbers depends on the connection state and receive window. Use helper functions like `seq_lt()` and `seq_diff()` that handle 32-bit wraparound correctly.

Resource Limits and Graceful Degradation

A user-space stack has finite memory and must impose limits to prevent denial-of-service scenarios:

- **ARP Cache Limits:** Maintain a fixed-size cache (e.g., 256 entries) with LRU (Least Recently Used) eviction. Prevent cache exhaustion by malicious ARP packets.
- **Connection Limits:** Limit the number of simultaneous `tcb` structures (e.g., 1024 connections). In `LISTEN` state, maintain a backlog queue with configurable maximum length.
- **Buffer Size Limits:** Implement configurable maximum sizes for TCP send and receive buffers. When buffers fill, apply backpressure through advertised window size (receive side) or blocking the application (send side).
- **Timer Management Limits:** Use a single timer wheel with fixed slots rather than per-segment timers to bound timer management overhead.

Handling Malicious/Malformed Packets

The stack operates in a potentially adversarial environment. Key defensive measures include:

- **ARP Cache Poisoning Mitigation:** Only update ARP cache entries based on:
  1. Solicited replies (where we sent a request for that IP)
  2. Gratuitous ARPs for IP addresses we own (for duplicate IP detection)
  3. Implement optional "ARP inspection" that validates MAC/IP consistency
- **TCP Sequence Number Validation:** Strictly validate sequence numbers against the receive window to prevent data injection attacks. RFC 793-based "window check" ensures segments are accepted only if they fall within the current window.
- **SYN Flood Protection:** In `tcp_listen()`, limit the rate of SYN arrivals and number of half-open connections (in `SYN_RCVD` state). Options include:

    - SYN cookies (advanced, cryptographic)
    - Simple rate limiting (drop SYNs above threshold)
    - Connection timeout for half-open states
- **Land Attack Prevention:** Drop packets where source IP equals destination IP and source port equals destination port, which can cause loops.
- **Ping of Death Protection:** Validate that ICMP echo request payload length doesn't exceed maximum reassembled IP packet size (65535 bytes).

State Machine Invariant Preservation

The TCP state machine is particularly vulnerable to edge cases. We maintain invariants through:

1. **Atomic State Transitions:** Each `tcb->state` change occurs at a single point in `tcp_process()`, with all necessary side effects (timer starts/stops, buffer clears) performed atomically with the transition.
2. **Timer-Cleanup Pairing:** Every timer start has a corresponding stop condition. For example, when entering `TIME_WAIT`, start the 2MSL timer; when it expires, transition to `CLOSED` and free the TCB.
3. **Sequence Number Arithmetic Safety:** Use the provided `seq_diff()` and `seq_lt()` functions for all sequence number comparisons. Never use raw integer comparisons (`<`, `>`) which fail due to 32-bit wraparound.
4. **Zero Window Probing:** When the receive window is zero, periodically send "window probes" (1-byte segments at `rcv_nxt`) to detect when window opens, preventing deadlock.

Error Reporting to Higher Layers

When the stack cannot handle an error internally, it must report to the application layer:

- **ICMP Error Generation:** For network-layer errors (TTL expired, destination unreachable), generate appropriate ICMP error messages sent back to the source IP, but rate-limit to prevent amplification attacks.
- **Socket Error Reporting:** Map internal errors to standard socket error codes (e.g., `ECONNREFUSED`, `ETIMEDOUT`, `ENOBUFS`). In our simplified stack, this might mean setting an error flag in the TCB for the application to check.
- **Connection Reset Notification:** When a `TCP_RST` is received or sent, immediately notify any waiting application with a connection reset error.

> **Design Philosophy:** A robust network stack should fail **predictably and safely**. When encountering an unexpected condition, the stack should default to dropping the problematic packet while preserving the stability of existing connections. Complex recovery mechanisms should only be implemented when they don't introduce new failure modes themselves.

Common Pitfalls

⚠️ Pitfall: Silent Packet Drops Without Logging
Description: Implementing error checks that drop packets but provide no indication of why packets are disappearing.
Why it's Wrong: Makes debugging nearly impossible—developers waste hours using packet sniffers to see packets arrive but not knowing where in the stack they're dropped.
Fix: Add incremental, toggleable logging at each validation point (e.g., "Dropping IP packet: checksum mismatch 0x%04x != 0x%04x").

⚠️ Pitfall: Not Validating Before Accessing Header Fields
Description: Casting a byte buffer to a protocol header struct without first verifying the buffer is large enough for that struct.
Why it's Wrong: Causes segmentation faults or reads of uninitialized memory when receiving truncated packets.
Fix: Always check `buffer_length >= sizeof(struct_header)` before casting and accessing fields.

⚠️ Pitfall: Ignoring Reserved Field Violations
Description: Not checking that reserved bits in protocol headers are zero as required by RFCs.
Why it's Wrong: Allows non-compliant implementations to interact with your stack, potentially exploiting undefined behavior.
Fix: Add explicit checks: `if (hdr->reserved_bits != 0) { drop_packet(); }`.

⚠️ Pitfall: Integer Overflows in Length Calculations
Description: Calculating packet or buffer sizes without checking for overflow, e.g., `total_len = ip_hdr_len + tcp_hdr_len + data_len`.
Why it's Wrong: Maliciously crafted length fields can cause integer wrap-around, leading to buffer overflows.
Fix: Use safe addition with overflow check: `if (ip_hdr_len > total_len - tcp_hdr_len) { drop_packet(); }`.

⚠️ Pitfall: Not Handling Resource Exhaustion Gracefully
Description: Assuming `malloc()` always succeeds or that system resources are infinite.
Why it's Wrong: Under heavy load or attack, the stack will crash instead of degrading gracefully.
Fix: Check all allocations, implement configurable limits, and have fallback paths (drop new connections when above threshold).

⚠️ Pitfall: Forgetting to Convert Byte Order
Description: Treating network byte order values (in headers) as host byte order without conversion.

Why it's Wrong: Numeric values (ports, IP addresses, lengths) will be incorrect on little-endian systems.
Fix: Use `ntohs()`, `ntohl()`, `htons()`, `htonl()` consistently for all multi-byte header fields.

Implementation Guidance

This implementation guidance provides defensive programming patterns and error handling infrastructure for Project Valiant. The primary language is C, which requires careful manual memory and resource management.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Error Logging	`fprintf(stderr, ...)` with log levels	Syslog integration or structured logging library
Memory Allocation	Standard `malloc()` / `free()` with checks	Arena allocator per connection + garbage collection
Timer Management	Simple periodic tick checking all timers	Hierarchical timer wheel for O(1) operations
Packet Validation	Manual checks before each header access	Generated parser from protocol definitions

B. Recommended File/Module Structure

```
project-valiant/
├── include/
│   ├── errors.h          ← Error codes and logging macros
│   ├── validation.h      ← Packet validation helper functions
│   └── ... (other headers)
├── src/
│   ├── core/
│   │   ├── errors.c      ← Logging implementation
│   │   ├── validation.c  ← Validation helpers implementation
│   │   └── resources.c   ← Resource limit management
│   ├── link/             ← Link layer components
│   ├── network/          ← Network layer components
│   ├── transport/        ← Transport layer components
│   └── utils/            ← Utility functions
└── tests/
    └── test_fuzzing.c    ← Fuzz tests for packet parsing
```

C. Infrastructure Starter Code

Complete Error Logging Module (`include/errors.h`):

```c
#ifndef VALIANT_ERRORS_H
#define VALIANT_ERRORS_H

#include <stdio.h>
#include <stdint.h>
#include <time.h>

// Error severity levels
typedef enum {
    LOG_DEBUG = 0,
    LOG_INFO,
    LOG_WARN,
    LOG_ERROR,
    LOG_FATAL
} log_level_t;

// Global log level configuration
extern log_level_t g_log_level;

// Logging macro that includes file and line
#define LOG(level, fmt, ...) do { \
    if ((level) >= g_log_level) { \
        time_t now = time(NULL); \
        struct tm *tm_info = localtime(&now); \
        char timestamp[20]; \
        strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", tm_info); \
        const char *level_str[] = {"DEBUG", "INFO", "WARN", "ERROR", "FATAL"}; \
        fprintf(stderr, "[%s] %s %s:%d: " fmt "\n", \
                timestamp, level_str[level], __FILE__, __LINE__, ##__VA_ARGS__); \
    } \
} while (0)

// Convenience macros for each level
#define LOG_DEBUG(fmt, ...) LOG(LOG_DEBUG, fmt, ##__VA_ARGS__)
#define LOG_INFO(fmt, ...)  LOG(LOG_INFO, fmt, ##__VA_ARGS__)
#define LOG_WARN(fmt, ...)  LOG(LOG_WARN, fmt, ##__VA_ARGS__)
#define LOG_ERROR(fmt, ...) LOG(LOG_ERROR, fmt, ##__VA_ARGS__)
#define LOG_FATAL(fmt, ...) LOG(LOG_FATAL, fmt, ##__VA_ARGS__)

// Stack error codes (map to standard errno where possible)
typedef enum {
    ERR_OK = 0,
    ERR_NOMEM = -1,           // Memory allocation failure
```

```
    ERR_NO_ROUTE = -2,        // No route to destination
    ERR_TIMEOUT = -3,         // Operation timed out
    ERR_CONN_RESET = -4,      // Connection reset by peer
    ERR_INVALID_PACKET = -5,  // Malformed protocol packet
    ERR_BUFFER_FULL = -6,     // Buffer capacity exceeded
    ERR_NOT_IMPLEMENTED = -7, // Feature not implemented
    ERR_INVALID_STATE = -8,   // Invalid protocol state
    ERR_CHKSUM = -9,          // Checksum mismatch
    ERR_TTL_EXPIRED = -10     // IP TTL reached zero
} stack_error_t;

// Convert stack error to string
const char* error_to_string(stack_error_t err);

#endif // VALIANT_ERRORS_H
```

Implementation (`src/core/errors.c`):

```c
#include "errors.h"
#include <string.h>

log_level_t g_log_level = LOG_INFO; // Default to INFO level

const char* error_to_string(stack_error_t err) {
    switch (err) {
        case ERR_OK: return "Success";
        case ERR_NOMEM: return "Memory allocation failure";
        case ERR_NO_ROUTE: return "No route to destination";
        case ERR_TIMEOUT: return "Operation timed out";
        case ERR_CONN_RESET: return "Connection reset by peer";
        case ERR_INVALID_PACKET: return "Malformed protocol packet";
        case ERR_BUFFER_FULL: return "Buffer capacity exceeded";
        case ERR_NOT_IMPLEMENTED: return "Feature not implemented";
        case ERR_INVALID_STATE: return "Invalid protocol state";
        case ERR_CHKSUM: return "Checksum mismatch";
        case ERR_TTL_EXPIRED: return "IP TTL expired";
        default: return "Unknown error";
    }
}
```

Packet Validation Helper (`include/validation.h`):

```c
#ifndef VALIANT_VALIDATION_H

#define VALIANT_VALIDATION_H


#include <stdint.h>

#include <stddef.h>

#include <stdbool.h>


// Check if buffer has at least 'required' bytes

static inline bool validate_buffer_length(size_t available, size_t required) {

    return available >= required;

}


// Safe pointer addition with overflow check

static inline const void* safe_ptr_add(const void* ptr, size_t offset,

                                       const void* base, size_t total_len) {

    const uint8_t* p = (const uint8_t*)ptr;

    const uint8_t* end = (const uint8_t*)base + total_len;



    // Check for overflow in pointer arithmetic

    if (p > end || offset > (size_t)(end - p)) {

        return NULL;

    }

    return p + offset;

}


// Validate IP header length field

bool validate_ip_header_length(const uint8_t version_ihl, size_t total_packet_len);


// Validate TCP header length (data offset)

bool validate_tcp_header_length(uint8_t data_offset, size_t ip_payload_len);


#endif // VALIANT_VALIDATION_H
```

D. Core Logic Skeleton Code

Defensive Ethernet Frame Parser (`src/link/ethernet.c`):

```c
#include "errors.h"

#include "validation.h"

#include "ethernet.h"


void netif_rx_packet(uint8_t* frame, size_t len) {

    // TODO 1: Validate minimum frame length (at least Ethernet header)

    //   Use: if (!validate_buffer_length(len, ETH_HDR_LEN)) { LOG_WARN(...); return; }



    // TODO 2: Safely cast to eth_hdr pointer after validation

    //   eth_hdr* hdr = (eth_hdr*)frame;



    // TODO 3: Check for known EtherType values

    //   uint16_t ethertype = ntohs(hdr->ethertype);

    //   if (ethertype != ETHERTYPE_IP && ethertype != ETHERTYPE_ARP) {

    //       LOG_DEBUG("Unknown EtherType: 0x%04x", ethertype);

    //       return;

    //   }



    // TODO 4: Dispatch to appropriate handler based on EtherType

    //   if (ethertype == ETHERTYPE_ARP) { arp_input(...); }

    //   else if (ethertype == ETHERTYPE_IP) { ipv4_input(...); }



    // TODO 5: Increment error counter for any validation failures

    //   stats.dropped_frames++;

}
```

Robust IP Packet Handler (`src/network/ipv4.c`):

```c
#include "errors.h"

#include "validation.h"

#include "ipv4.h"


void ipv4_input(uint8_t* packet_data, size_t len, uint32_t from_ip) {

    // TODO 1: Validate minimum IP header length (20 bytes)

    //   if (!validate_buffer_length(len, IP_HDR_LEN)) { ... }



    // TODO 2: Cast to ip_hdr pointer safely

    //   ip_hdr* hdr = (ip_hdr*)packet_data;



    // TODO 3: Verify IP version is 4

    //   if ((hdr->version_ihl >> 4) != 4) { LOG_WARN(...); return; }



    // TODO 4: Validate header length (IHL) field

    //   uint8_t ihl = (hdr->version_ihl & 0x0F) * 4;

    //   if (!validate_ip_header_length(hdr->version_ihl, len)) { ... }



    // TODO 5: Verify total length field consistency

    //   uint16_t total_len = ntohs(hdr->total_len);

    //   if (total_len > len || total_len < ihl) { ... }



    // TODO 6: Verify checksum

    //   uint16_t received_checksum = hdr->checksum;

    //   hdr->checksum = 0;

    //   uint16_t computed = ip_checksum(hdr, ihl);

    //   if (received_checksum != computed) { LOG_WARN(...); return; }



    // TODO 7: Decrement TTL, check for expiration

    //   if (hdr->ttl == 0) { send_icmp_time_exceeded(...); return; }

    //   hdr->ttl--;



    // TODO 8: Recompute checksum after TTL change

    //   hdr->checksum = 0;

    //   hdr->checksum = ip_checksum(hdr, ihl);



    // TODO 9: Validate destination IP matches local interface

    //   if (!is_local_ip(hdr->dst_ip)) { maybe forward packet? }



    // TODO 10: Dispatch to protocol handler based on protocol field

    //   uint8_t protocol = hdr->protocol;
```

```
    //   uint8_t* payload = packet_data + ihl;
    //   size_t payload_len = total_len - ihl;


    // TODO 11: Handle fragmentation if needed (simplify: drop fragmented packets)
    //   uint16_t frag_off = ntohs(hdr->frag_off);
    //   if ((frag_off & 0x3FFF) != 0) { LOG_DEBUG(...); return; }
}
```

TCP Segment Validation with State Checking (`src/transport/tcp_input.c`):

```c
#include "errors.h"
#include "validation.h"
#include "tcp.h"

void tcp_input(uint8_t* segment_data, size_t len, uint32_t src_ip, uint32_t dst_ip) {
    // TODO 1: Validate minimum TCP header length (20 bytes)
    //   if (!validate_buffer_length(len, TCP_HDR_LEN)) { ... }

    // TODO 2: Cast to tcp_hdr pointer
    //   tcp_hdr* hdr = (tcp_hdr*)segment_data;

    // TODO 3: Extract data offset (header length in 32-bit words)
    //   uint8_t data_offset = (hdr->data_offset_reserved_flags >> 12) & 0x0F;
    //   uint16_t tcp_header_len = data_offset * 4;

    // TODO 4: Validate TCP header length
    //   if (!validate_tcp_header_length(data_offset, len)) { ... }

    // TODO 5: Verify TCP checksum with pseudo-header
    //   uint16_t computed = tcp_checksum(hdr, payload, payload_len, src_ip, dst_ip);
    //   if (computed != hdr->checksum) { LOG_WARN(...); return; }

    // TODO 6: Demultiplex to find TCB using 4-tuple
    //   tcb_t* conn = tcp_find_connection(dst_ip, ntohs(hdr->dst_port),
    //                                     src_ip, ntohs(hdr->src_port));

    // TODO 7: Handle case where no TCB found (no matching connection)
    //   if (!conn) {
    //       // If segment has RST flag, ignore
    //       // If segment has SYN flag, create new connection (if in LISTEN)
    //       // Otherwise, send RST segment back
    //   }

    // TODO 8: Validate sequence number is within receive window
    //   uint32_t seq = ntohl(hdr->seq_num);
    //   if (!tcp_validate_sequence(conn, seq, payload_len)) {
    //       // Send ACK with current rcv_nxt (no data acceptance)
    //   }

    // TODO 9: Process through state machine with defensive checks
    //   tcp_process(conn, hdr, payload, payload_len);
```

```
    // TODO 10: Handle urgent pointer if URG flag set (simplify: ignore for now)
    //   if (hdr->data_offset_reserved_flags & TCP_URG) { ... }
}
```

E. Language-Specific Hints

- **Memory Safety in C:** Always validate buffer lengths before accessing memory. Use the `safe_ptr_add()` helper for pointer arithmetic with bounds checking.
- **Error Propagation:** In C, functions should return error codes (negative values) for recoverable errors and use logging for debugging. For unrecoverable errors, log and terminate gracefully.
- **Resource Cleanup:** Use `goto cleanup` pattern for functions with multiple allocation points:

```c
int allocate_resources() {
    void* buf1 = malloc(100);
    if (!buf1) { goto error; }
    void* buf2 = malloc(200);
    if (!buf2) { goto cleanup_buf1; }
    // ... use resources
    free(buf2);
cleanup_buf1:
    free(buf1);
error:
    return ERR_NOMEM;
}
```

- **Atomic Operations:** For statistics counters accessed from multiple threads (if threading is added), use `__atomic_add_fetch()` for increment operations.
- **Network Byte Order:** Create wrapper functions for common operations:

```c
static inline uint32_t get_ip_src(const ip_hdr* hdr) {
    return ntohl(hdr->src_ip);  // Convert to host byte order
}
```

F. Milestone Checkpoints

Milestone 1 Checkpoint - Error Handling:

- Run the stack and send malformed Ethernet frames (e.g., too short, wrong EtherType)
- Verify in logs: "Dropping frame: too short (5 bytes)" or "Unknown EtherType: 0x1234"
- Send ARP request for non-local IP, verify no reply is sent (packet dropped silently)
- Fill ARP cache to capacity, verify new entries evict oldest (LRU behavior)

Milestone 2 Checkpoint - Error Handling:

- Send IP packet with invalid checksum, verify it's dropped and logged
- Send IP packet with TTL=1, verify it's dropped and ICMP Time Exceeded sent back
- Send to non-existent destination IP, verify packet dropped (or ICMP Destination Unreachable)
- Send fragmented IP packet (with MF flag), verify it's dropped (simplified implementation)

Milestone 3 Checkpoint - Error Handling:

- Send TCP SYN to closed port, verify RST segment is sent back
- Send TCP segment with invalid checksum, verify it's dropped
- Initiate connection, then send segment with old sequence number, verify it's ignored (ACK sent with current rcv_nxt)
- Send FIN after connection already closed, verify RST is sent

Milestone 4 Checkpoint - Error Handling:

- Fill receive buffer, verify window size goes to zero in outgoing ACKs
- Don't ACK sent data, verify retransmission after RTO timeout
- Send duplicate ACKs, verify fast retransmit triggers (if implemented)
- Exceed congestion window, verify packets are queued, not sent

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Packets received but no response	Silent drop due to validation failure	Enable debug logging (`g_log_level = LOG_DEBUG`), check for validation warnings	Fix the validation logic, ensure all checks pass
Ping works but TCP doesn't	TCP checksum error or sequence number issue	Use Wireshark to verify checksums, compare sequence numbers in trace	Ensure TCP checksum includes pseudo-header, handle byte order correctly
Connection hangs at SYN_SENT	SYN-ACK not received or validation fails	Capture packets, verify SYN-ACK checksum, sequence numbers	Check TCP state machine transitions, ensure ACK of SYN increments sequence
Data transfer extremely slow	Receive window stuck at zero or congestion window small	Log window sizes in ACKs, check buffer management	Ensure application reads receive buffer, implement proper window scaling
Stack crashes randomly	Buffer overflow or null pointer dereference	Use AddressSanitizer (`-fsanitize=address`), add more length checks	Add bounds checking before all buffer accesses, validate packet lengths
Memory leak over time	TCBs or buffers not freed on connection close	Use Valgrind or `mtrace()` to track allocations	Ensure all code paths free resources, use cleanup functions
ARP cache fills with stale entries	No timeout or eviction implemented	Log ARP cache state periodically	Implement aging timer, remove entries older than timeout

11. Testing Strategy

> **Milestone(s):** 1, 2, 3, 4 (Verification at Every Stage)

Testing a network stack presents unique challenges because it's an interactive system that communicates with external entities using precisely defined protocols. Unlike testing a standalone application, we must verify that our implementation correctly interprets and generates binary protocol formats, maintains complex state across time, and responds appropriately to both normal and abnormal network conditions. This section outlines a **multi-faceted verification strategy** that combines isolated unit testing, controlled integration testing with packet inspection, and system-level validation using standard networking tools. The goal is to build confidence in each layer's correctness before integrating it into the complete stack, while providing clear, actionable checkpoints for learners to validate their progress against each milestone.

Testing Approaches

A successful testing strategy for Project Valiant employs three complementary approaches, each serving a different purpose in the verification hierarchy:

Unit Testing: Protocol Logic in Isolation

Unit testing focuses on the smallest testable components—individual functions and data structure manipulations—in complete isolation from the rest of the system. This approach is particularly valuable for verifying **protocol parsing logic, checksum calculations, state machine transitions, and algorithmic correctness** without the complexity of real network I/O.

> **Mental Model: The Laboratory Microscope** Think of unit testing as examining individual cogs and gears under a microscope. You can verify each gear's teeth are correctly shaped, measure its rotation precisely, and confirm it interacts properly with adjacent gears in a controlled fixture—all without needing to assemble the entire clock. Similarly, unit tests let us verify that our `ip_checksum` function computes correct values for known inputs, that our `seq_lt` function handles 32-bit wraparound correctly, and that our `tcp_process` function transitions states appropriately when given synthetic input segments.

Unit tests should cover:

- **Header parsing and construction**: Verify that `eth_hdr`, `ip_hdr`, `tcp_hdr`, and `arp_hdr` structures correctly interpret raw bytes and generate valid output.
- **Checksum algorithms**: Test `ip_checksum` and `tcp_checksum` against known-correct examples from RFCs or packet captures.
- **Sequence number arithmetic**: Validate `seq_diff`, `seq_lt`, and window calculations with edge cases around 32-bit wraparound.
- **State machine transitions**: Test the TCP state machine with controlled inputs, verifying each state transition and associated actions.
- **Buffer management**: Verify `tcp_buffer` operations (write, read, wraparound) maintain correctness under various patterns.

- **Routing logic**: Test `route_lookup` with various destination addresses and routing table configurations.

For unit testing, we recommend using a simple testing framework (like **Unity** or **Check** for C) that provides assertions, test fixtures, and test runners. Since the stack interacts heavily with the operating system (TAP devices, timers), we'll need to **mock or stub these dependencies**—for example, replacing `tap_read` / `tap_write` with memory buffers, and replacing timer callbacks with manual tick advancement.

Test Category	Example Test Cases	Mocking Strategy
Header Parsing	Parse known Ethernet frame bytes, verify `src_mac`, `dst_mac`, `ethertype` match expected values; parse IP header with options, verify all fields correct	Direct byte arrays as input, no mocking needed
Checksums	Compute IPv4 checksum for header with zero checksum field, verify result; compute TCP checksum with pseudo-header, compare to Wireshark-captured packet	Provide raw header/payload data
State Machines	Create `tcb` in `LISTEN` state, simulate receiving SYN segment, verify transition to `SYN_RCVD` and SYN-ACK sent	Mock `tcp_output` to capture sent segments
Routing	Populate routing table with multiple entries, query for specific destinations, verify longest prefix match works	Isolate routing table module
Buffer Operations	Write data until buffer full, read partial data, verify readable count and pointer positions update correctly	Test `tcp_buffer_t` independently

Integration Testing: Protocol Interactions with Packet Capture

Integration testing verifies that components work together correctly, focusing on the **protocol interactions between layers** and with external systems. The most effective approach for a network stack is to use **packet capture and replay**: generate or capture real network traffic, feed it through the stack, and verify both the internal state changes and the outgoing responses match protocol specifications.

> **Mental Model: The Automotive Test Track** Integration testing is like putting a car's drivetrain on a dynamometer—the engine, transmission, and wheels work together under controlled conditions that simulate real driving, while instruments measure power output, fuel consumption, and emissions. Similarly, we'll feed recorded network traffic into our stack and instrument its responses, verifying that the complete system behaves according to TCP/IP specifications when faced with real-world packet sequences.

Integration testing strategy:

1. **Capture real protocol exchanges** using Wireshark/tcpdump (ARP requests/replies, TCP handshakes, data transfers).
2. **Create test fixtures** that replay these captures through the stack's input functions (`netif_rx_packet`, `ipv4_input`, `tcp_input`).
3. **Intercept outgoing packets** by mocking the `tap_write` function and capturing what would be transmitted.
4. **Compare generated packets** against expected protocol behavior: correct header fields, appropriate flags, valid checksums, proper sequence/acknowledgment numbers.
5. **Verify internal state** after processing each packet: ARP cache entries, TCB state changes, buffer contents, timer states.

This approach is particularly valuable for testing **complex protocol sequences** like the TCP three-way handshake, where multiple packets must be exchanged with precise timing and field values. By replaying a known-good handshake capture, we can verify our stack generates the correct SYN-ACK in response to a SYN, and transitions to `ESTABLISHED` after sending the final ACK.

Integration Test Scenario	Input Packet Sequence	Expected Output & State Changes
ARP Resolution	ARP Request for stack's IP (broadcast)	ARP Reply with correct MAC; ARP cache entry created
ICMP Echo (Ping)	ICMP Echo Request to stack's IP	ICMP Echo Reply with matching identifier/sequence; checksum valid
TCP Passive Open	SYN to listening port	SYN-ACK with correct seq/ack numbers; TCB in `SYN_RCVD`
TCP Active Open	Initiate `tcp_connect`, receive SYN-ACK	Send ACK; TCB transitions to `ESTABLISHED`
TCP Data Transfer	Data segment after connection established	ACK with correct acknowledgment number; data delivered to receive buffer
TCP Fast Retransmit	Send 3 duplicate ACKs for missing segment	Retransmit unacknowledged segment; reduce congestion window
TCP Graceful Close	FIN from remote peer	ACK the FIN; send our own FIN; transition through `CLOSE_WAIT`, `LAST_ACK`

System Testing: End-to-End Validation with Standard Tools

System testing validates the complete stack as a **black box** using standard networking tools that real applications would use. This is the ultimate acceptance test: can our stack interoperate with existing operating systems and tools? This approach tests not only protocol correctness but also **usability, performance characteristics, and robustness** under real network conditions.

> **Mental Model: The Road Test** System testing is the road test—actually driving the car on real roads with traffic, hills, and weather. The car must start reliably, accelerate smoothly, brake effectively, and navigate turns. Similarly, our stack must respond to `ping`, serve data to `netcat`, handle multiple connections, and survive packet loss—just like a production network stack.

System testing methodology:

1. **Deploy the stack** on a test machine with a TAP device connected to a virtual or physical network.
2. **Assign IP and routing configuration** to the stack (matching the test network).
3. **Use standard UNIX networking tools** (`ping`, `netcat`, `curl` if HTTP implemented, `telnet`) to interact with the stack.
4. **Monitor traffic with Wireshark** to verify packets are correctly formed and sequenced.
5. **Test edge cases and failure modes**: kill connections, introduce packet loss (via `tc`), test with jumbo frames, verify timeout handling.

The key advantage of system testing is that it **tests the complete integrated system**, including any bugs in the interaction between layers that might be missed by unit or integration tests. It also provides the most satisfying validation for learners—seeing their stack respond to `ping` or serve a web page proves the entire system works.

Tool	Test Purpose	Expected Behavior
`ping`	Basic IP/ICMP functionality	Stack responds to Echo Requests; replies have correct source IP; round-trip times measured
`arping`	ARP resolution	Stack replies to ARP requests; entries appear in `arp -a` on other hosts
`netcat` (TCP client)	TCP connection establishment and data transfer	Stack accepts connections on listening port; echoes data back; connections close cleanly
`netcat` (TCP server)	Active opening and data sending	Stack initiates connections to remote servers; sends data; receives responses
`tcpdump`/Wireshark	Protocol compliance	Captured packets show correct header fields, flags, sequence numbers; follow RFC specifications
`iptables` + `tc`	Loss and delay simulation	Stack handles packet loss with retransmissions; adjusts congestion window; survives network delay
`ab` (Apache Bench)	Concurrent connections	Stack handles multiple simultaneous connections; manages separate TCBs correctly

Milestone Checkpoints

Each milestone has specific acceptance criteria that should be verifiable through a combination of the testing approaches above. These checkpoints provide concrete, step-by-step validation procedures that learners can execute to confirm their implementation is working correctly.

Milestone 1: Ethernet & ARP

Objective: Verify that the stack correctly receives and transmits Ethernet frames, parses Ethernet and ARP headers, and performs IP-to-MAC address resolution.

Validation Procedure:

1. **TAP Device Setup**:
   - Create and configure a TAP device with IP address `192.168.1.100/24`.
   - Verify the device appears in `ip link show` with state `UP`.
   - Use `tcpdump -i tap0 -n` to monitor raw Ethernet frames.
2. **Ethernet Frame Parsing**:
   - From another host on the same network (or using `ping -I tap0 192.168.1.1`), generate traffic to the TAP device.
   - Verify that `netif_rx_packet` is called with frames, and that `eth_hdr` parsing correctly extracts:
     - Destination MAC (should be TAP device's MAC or broadcast)
     - Source MAC (should be sender's MAC)
     - EtherType (0x0800 for IP, 0x0806 for ARP)
3. **ARP Request Handling**:
   - From another host, send ARP request: `arping -I eth0 192.168.1.100`.
   - Verify the stack:
     - Receives ARP request with `opcode` = `ARP_OP_REQUEST`
     - Validates target IP matches configured IP
     - Sends ARP reply with `opcode` = `ARP_OP_REPLY`
     - Includes correct MAC address in `sender_mac` field

- Check Wireshark capture shows proper ARP exchange.

4. **ARP Cache Functionality**:
   - After ARP resolution, verify `arp_cache_lookup` returns the correct MAC.
   - Wait beyond cache timeout (configurable, e.g., 30 seconds).
   - Verify entry is evicted (next lookup triggers new ARP request).
5. **Broadcast Handling**:
   - Send broadcast Ethernet frame (destination MAC `FF:FF:FF:FF:FF:FF`).
   - Verify stack processes it (for ARP) but ignores other broadcast protocols.

Commands to Run:

```bash
# Terminal 1: Start the stack with TAP device
sudo ./stack --tap tap0 --ip 192.168.1.100 --netmask 255.255.255.0

# Terminal 2: Monitor traffic
sudo tcpdump -i tap0 -n -v

# Terminal 3: Generate ARP request (from another host or namespace)
sudo ip netns exec test-ns arping -c 3 -I veth0 192.168.1.100

# Terminal 4: Check ARP cache (if implemented via CLI)
./stack-cli arp-show
```

Expected Output:

- tcpdump shows: `ARP, Request who-has 192.168.1.100 tell 192.168.1.1` followed by `ARP, Reply 192.168.1.100 is-at <MAC>`.
- Stack logs indicate: `Received ARP request for our IP`, `Sending ARP reply`, `ARP cache updated`.
- ARP cache display shows entry for `192.168.1.1` with MAC address and state `ARP_ENTRY_RESOLVED`.

Milestone 2: IP & ICMP

Objective: Verify IP packet processing, checksum validation, ICMP echo reply (ping), and basic routing.

Validation Procedure:

1. **IP Header Parsing**:
   - Send IP packet (via `ping` or raw socket) to stack's IP.
   - Verify `ipv4_input` correctly extracts:
     - Source and destination IP addresses
     - Protocol field (`IPPROTO_ICMP` for ping)
     - Total length matches received packet size
     - TTL value
2. **IP Checksum Validation**:
   - Send IP packet with correct checksum (should be accepted).
   - Send IP packet with invalid checksum (should be dropped with `ERR_CHKSUM`).
   - Verify outgoing IP packets have valid checksums (compute with `ip_checksum`).
3. **ICMP Echo Request/Reply**:
   - From another host: `ping 192.168.1.100`.
   - Verify stack:
     - Receives ICMP Echo Request (type 8, code 0)
     - Generates ICMP Echo Reply (type 0, code 0)
     - Copies identifier and sequence numbers from request
     - Computes valid ICMP checksum for reply
   - Ping should show successful replies with round-trip times.
4. **Routing Table Lookup**:

- Add route: `route_add(192.168.2.0, 255.255.255.0, 192.168.1.1, tap0)`.
- Query route for `192.168.2.5`: `route_lookup` should return gateway `192.168.1.1`.
- Query route for `192.168.1.50` (directly connected): should return gateway `0.0.0.0`.
- Query route for `10.0.0.1` (no route): should return `NULL`.

5. **TTL Handling**:
   - Send IP packet with TTL=1.
   - Verify stack decrements TTL to 0, drops packet, and sends ICMP Time Exceeded message (type 11, code 0) back to source (optional advanced test).

Commands to Run:

```bash
# Terminal 1: Start stack with routing
sudo ./stack --tap tap0 --ip 192.168.1.100/24 --route "192.168.2.0/24 via 192.168.1.1"

# Terminal 2: Monitor with detailed IP/ICMP filter
sudo tcpdump -i tap0 -n "icmp or arp"

# Terminal 3: Ping the stack
ping -c 4 192.168.1.100

# Terminal 4: Test routing (via separate namespace with different subnet)
sudo ip netns exec test-ns ping -c 2 192.168.2.100  # Should be routed via gateway
```

Expected Output:

- Ping shows: `4 packets transmitted, 4 received, 0% packet loss`.
- tcpdump shows: `IP 192.168.1.1 > 192.168.1.100: ICMP echo request` and `IP 192.168.1.100 > 192.168.1.1: ICMP echo reply`.
- Stack logs show: `IP packet from 192.168.1.1 to 192.168.1.100, proto ICMP`, `ICMP echo request, sending reply`, `Routing: dest 192.168.2.100 via 192.168.1.1`.

Milestone 3: TCP Connection Management

Objective: Verify TCP segment parsing, three-way handshake, state machine transitions, and graceful connection teardown.

Validation Procedure:

1. **TCP Header Parsing**:
   - Send TCP SYN packet to stack's listening port.
   - Verify `tcp_input` correctly extracts:
     - Source/destination ports
     - Sequence number (valid ISN)
     - Acknowledgment number (0 for SYN)
     - Flags (SYN bit set)
     - Window size
2. **Passive Open (Server)**:
   - Call `tcp_listen(8080)` to create listening socket.
   - From another host: `nc -v 192.168.1.100 8080`.
   - Verify stack:
     - Receives SYN, sends SYN-ACK with `seq_num = initial_seq`, `ack_num = received_seq + 1`
     - Transitions TCB from `LISTEN` to `SYN_RCVD`
     - Receives ACK, transitions to `ESTABLISHED`
   - Connection should establish successfully.
3. **Active Open (Client)**:
   - Call `tcp_connect(192.168.1.1, 80, 5000)`.
   - Verify stack:
     - Sends SYN with local port ~5000
     - Receives SYN-ACK, sends ACK

        - Transitions from `SYN_SENT` to `ESTABLISHED`
    - Connection should establish (if real server at destination).
4. **Graceful Close (Both Directions)**:
    - After established connection:
        - Remote sends FIN: stack should ACK, transition to `CLOSE_WAIT`
        - Application calls `tcp_close`: stack sends FIN, transitions to `LAST_ACK`
        - Receives ACK for FIN: transitions to `CLOSED`, cleans up TCB
    - Verify four-way handshake completes cleanly.
5. **Reset Handling**:
    - Send RST packet to established connection.
    - Verify stack transitions to `CLOSED`, cleans up resources, notifies application.

Commands to Run:

```bash
# Terminal 1: Start stack with TCP listening on port 8080
sudo ./stack --tap tap0 --ip 192.168.1.100/24 --tcp-listen 8080

# Terminal 2: Monitor TCP traffic with sequence numbers
sudo tcpdump -i tap0 -n "tcp and port 8080" -S

# Terminal 3: Connect to stack using netcat
nc -v 192.168.1.100 8080
# Type some text, verify it's received by stack
# Press Ctrl+C to send FIN (if netcat does graceful close)

# Terminal 4: Test active open (if real server available)
./stack-cli tcp-connect 93.184.216.34 80  # example.com HTTP
```

Expected Output:

- tcpdump shows classic three-way handshake: `[S]`, `[S.]`, `[.]` flags.
- Stack logs show: `TCP connection request from 192.168.1.1:45678 to 192.168.1.100:8080`, `State transition LISTEN→SYN_RCVD`, `State transition SYN_RCVD→ESTABLISHED`.
- Netcat connection establishes, data can be sent/received.
- On close: `FIN` flags exchanged, state transitions through `FIN_WAIT_1`, `FIN_WAIT_2`, `TIME_WAIT` (or `CLOSE_WAIT`, `LAST_ACK` for passive close).

Milestone 4: TCP Data Transfer & Flow Control

Objective: Verify reliable data transfer with sliding windows, retransmissions, flow control, and basic congestion control.

Validation Procedure:

1. **Sliding Window Transmission**:
    - Send 10KB of data via `tcp_send_data` in chunks smaller than MSS.
    - Verify:
        - Multiple segments sent before waiting for ACKs (up to window size)
        - `snd_nxt` advances as segments sent
        - `snd_una` advances as ACKs received
        - Window slides properly as acknowledgments arrive
2. **Retransmission on Timeout**:
    - Artificially drop ACKs for a segment (by modifying test harness).
    - Verify retransmission timer fires after RTO.
    - Segment is retransmitted with same sequence number.
    - RTO backs off exponentially on subsequent timeouts (up to limit).
3. **Flow Control (Receiver Window)**:

- Set small receive buffer (e.g., 1KB).
- Send data faster than application reads.
- Verify sender's window (`snd_wnd`) shrinks as receiver buffer fills.
- Sender stops sending when window reaches 0.
- When application reads data, window opens and sender resumes.

4. **Congestion Control (Slow Start)**:
   - On new connection, verify `cwnd` starts at 1*MSS.
   - For each ACK received, `cwnd` increases by 1*MSS (exponential growth).
   - After `ssthresh` reached, verify `cwnd` grows linearly (additive increase).
   - On packet loss (timeout), verify `ssthresh = cwnd/2` , `cwnd = 1*MSS` .
5. **Fast Retransmit**:
   - Send segments 1,2,3,4,5.
   - Drop segment 2, but deliver 3,4,5.
   - Verify receiver sends duplicate ACKs for segment 2.
   - After 3 duplicate ACKs, sender retransmits segment 2 without waiting for timeout.
   - `ssthresh` and `cwnd` adjust appropriately.

Commands to Run:

```bash
# Terminal 1: Start stack with TCP echo server on port 8080
sudo ./stack --tap tap0 --ip 192.168.1.100/24 --tcp-echo 8080

# Terminal 2: Monitor with detailed TCP analysis
sudo tcpdump -i tap0 -n "tcp and port 8080" -S -t

# Terminal 3: Send large file to stack (test sliding window)
dd if=/dev/zero bs=1K count=100 | nc -v 192.168.1.100 8080 > /dev/null

# Terminal 4: Test flow control (slow consumer)
# Use custom test client that sends fast but reads slowly
./test_tcp_flowcontrol 192.168.1.100 8080

# Terminal 5: Introduce packet loss (requires root)
sudo tc qdisc add dev tap0 root netem loss 10%  # 10% packet loss
# Observe retransmissions in tcpdump
sudo tc qdisc del dev tap0 root  # Cleanup
```

Expected Output:

- tcpdump shows multiple data segments in flight, ACKs advancing, window size in packets.
- With packet loss: retransmissions appear with same sequence numbers; duplicate ACKs visible.
- Stack logs show: `cwnd=1460` , `cwnd=2920` (exponential growth), `Retransmission timeout for seq=12345` , `Fast retransmit triggered` .
- Data transfer completes reliably despite losses; throughput adapts to network conditions.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (For Extended Projects)
Unit Testing Framework	Custom minimal test harness with assertions	Check framework (C unit testing) or Unity
Mocking Network I/O	Function pointer substitution in test code	CMock or custom mock objects
Packet Capture/Replay	Raw hex dumps in test arrays	libpcap for reading pcap files
System Test Environment	Linux network namespaces + virtual Ethernet pairs	Docker containers with bridged networking
Debugging & Inspection	Extensive logging with `LOG()` macro	Integrated GDB scripting with pretty-printers
Performance Profiling	Simple timestamp logging	perf tools or custom metrics collection

B. Recommended File/Module Structure for Testing

```
project_valiant/
├── src/                          # Main stack implementation
│   ├── link/                     # Milestone 1
│   ├── network/                  # Milestone 2
│   ├── transport/                # Milestones 3 & 4
│   └── utils/                    # Shared utilities
├── tests/                        # All test code
│   ├── unit/                     # Unit tests
│   │   ├── test_link.c           # Ethernet/ARP tests
│   │   ├── test_network.c        # IP/ICMP/Routing tests
│   │   ├── test_tcp_state.c      # TCP state machine tests
│   │   ├── test_tcp_transfer.c   # Sliding window, flow control tests
│   │   └── test_utils.c          # Checksum, buffer, sequence tests
│   ├── integration/              # Integration tests
│   │   ├── arp_integration.c     # ARP request/reply scenarios
│   │   ├── icmp_integration.c    # Ping request/reply scenarios
│   │   ├── tcp_handshake.c       # Three-way handshake test
│   │   └── test_harness.c        # Common test infrastructure
│   ├── system/                   # System tests
│   │   ├── ping_test.sh          # Script to test ping response
│   │   ├── netcat_test.sh        # Script to test TCP echo
│   │   └── packet_loss_test.sh   # Script to test with packet loss
│   └── fixtures/                 # Test data
│       ├── arp_request.bin       # Raw ARP request packet
│       ├── tcp_syn.bin           # Raw TCP SYN packet
│       └── http_get.bin          # Sample HTTP GET request
├── tools/                        # Testing utilities
│   ├── packet_generator.c        # Generate test packets
│   ├── stack_cli.c               # Command-line interface to stack
│   └── tap_setup.sh              # Script to create TAP device
└── Makefile                      # Build with test targets
```

C. Infrastructure Starter Code: Test Harness Framework

```c
/* tests/integration/test_harness.h */

#ifndef TEST_HARNESS_H

#define TEST_HARNESS_H


#include <stdint.h>

#include <stddef.h>

#include <stdbool.h>


/* Mock TAP device implementation for testing */

typedef struct {

    uint8_t frame_buffer[2048];

    size_t frame_length;

    bool frame_ready;

} mock_tap_device_t;


void mock_tap_init(mock_tap_device_t* dev);

ssize_t mock_tap_read(int fd, void* buf, size_t len);  /* Mock version */

ssize_t mock_tap_write(int fd, const void* buf, size_t len); /* Mock version */


/* Packet comparison utilities */

bool compare_ether_header(const uint8_t* expected, const uint8_t* actual, size_t len);

bool compare_ip_header(const uint8_t* expected, const uint8_t* actual, size_t len);

bool compare_tcp_header(const uint8_t* expected, const uint8_t* actual, size_t len);


/* Test assertion macros with better diagnostics */

#define TEST_ASSERT(cond, msg) \

    do { \

        if (!(cond)) { \

            fprintf(stderr, "FAIL: %s at %s:%d\n", msg, __FILE__, __LINE__); \

            return false; \

        } \

    } while(0)


#define TEST_ASSERT_PACKET_EQ(expected, actual, len, msg) \

    do { \

        if (memcmp(expected, actual, len) != 0) { \

            fprintf(stderr, "FAIL: %s - packets differ\n", msg); \

            hex_dump("Expected:", expected, len); \

            hex_dump("Actual:", actual, len); \

            return false; \

        } \

    } while(0)
```

```c
/* Hex dump utility for debugging */

void hex_dump(const char* label, const uint8_t* data, size_t len);


#endif /* TEST_HARNESS_H */
```

```c
/* tests/integration/test_harness.c */

#include "test_harness.h"

#include <string.h>

#include <stdio.h>


mock_tap_device_t g_mock_tap;


void mock_tap_init(mock_tap_device_t* dev) {

    memset(dev, 0, sizeof(*dev));

    dev->frame_ready = false;

}


ssize_t mock_tap_read(int fd, void* buf, size_t len) {

    (void)fd; /* Unused in mock */



    if (!g_mock_tap.frame_ready || len < g_mock_tap.frame_length) {

        return 0; /* Would block */

    }



    memcpy(buf, g_mock_tap.frame_buffer, g_mock_tap.frame_length);

    size_t copied = g_mock_tap.frame_length;

    g_mock_tap.frame_ready = false;

    return copied;

}


ssize_t mock_tap_write(int fd, const void* buf, size_t len) {

    (void)fd; /* Unused in mock */



    /* Store the transmitted frame for verification */

    if (len > sizeof(g_mock_tap.frame_buffer)) {

        return -1; /* Too large */

    }



    memcpy(g_mock_tap.frame_buffer, buf, len);

    g_mock_tap.frame_length = len;

    g_mock_tap.frame_ready = true;



    /* Log the transmission for test verification */

    printf("[TEST] Frame transmitted, %zu bytes, ethertype: 0x%04x\n",

           len, ntohs(*(uint16_t*)((uint8_t*)buf + 12)));



    return len;
```

```
}

void hex_dump(const char* label, const uint8_t* data, size_t len) {
    printf("%s\n", label);
    for (size_t i = 0; i < len; i++) {
        printf("%02x ", data[i]);
        if ((i + 1) % 16 == 0) printf("\n");
    }
    printf("\n");
}
```

D. Core Logic Skeleton Code: Unit Test Example

```c
/* tests/unit/test_tcp_state.c */

#include "../test_harness.h"

#include "../../src/transport/tcp.h"

#include "../../src/network/ip.h"


/* Test: TCP state transition from LISTEN to SYN_RCVD on SYN receipt */

bool test_tcp_listen_to_syn_rcvd(void) {

    printf("Test: TCP LISTEN -> SYN_RCVD transition\n");


    /* Setup */

    tcb_t* listen_tcb = tcp_listen(8080);

    TEST_ASSERT(listen_tcb != NULL, "Failed to create listening TCB");

    TEST_ASSERT(listen_tcb->state == TCP_LISTEN, "Initial state should be LISTEN");


    /* Create a synthetic SYN packet */

    uint8_t syn_packet[60] = {0};

    ip_hdr* ip = (ip_hdr*)syn_packet;

    tcp_hdr* tcp = (tcp_hdr*)(syn_packet + IP_HDR_LEN);


    /* Build IP header */

    ip->version_ihl = (4 << 4) | (IP_HDR_LEN / 4); /* IPv4, 20 bytes */

    ip->total_len = htons(60);

    ip->ttl = 64;

    ip->protocol = IPPROTO_TCP;

    ip->src_ip = htonl(0xC0A80101); /* 192.168.1.1 */

    ip->dst_ip = htonl(0xC0A80164); /* 192.168.1.100 */

    ip->checksum = ip_checksum(ip, IP_HDR_LEN);


    /* Build TCP header */

    tcp->src_port = htons(12345);

    tcp->dst_port = htons(8080);

    tcp->seq_num = htonl(1000); /* Initial sequence number */

    tcp->ack_num = 0;

    tcp->data_offset_reserved_flags = htons((TCP_HDR_LEN / 4) << 12) | TCP_SYN;

    tcp->window = htons(5840);

    tcp->checksum = 0; /* Would be calculated with pseudo-header */


    /* TODO 1: Calculate proper TCP checksum with pseudo-header */

    /* TODO 2: Call tcp_input with the SYN packet */

    /* TODO 3: Verify a new TCB was created (not the listening one) */
```

```
    /* TODO 4: Verify new TCB state is SYN_RCVD */
    /* TODO 5: Verify SYN-ACK was sent (check mock_tap_write was called) */
    /* TODO 6: Verify SYN-ACK has correct seq/ack numbers */
    /* TODO 7: Clean up TCBs */

    return true;
}

/* Test suite runner */
int main(void) {
    int passed = 0;
    int total = 0;

    /* Initialize mock infrastructure */
    mock_tap_init(&g_mock_tap);

    /* Run tests */
    if (test_tcp_listen_to_syn_rcvd()) { passed++; } total++;
    /* Add more test calls here */

    printf("\nTest Summary: %d/%d passed\n", passed, total);
    return (passed == total) ? 0 : 1;
}
```

E. Language-Specific Hints for C Testing

1. **Compilation and Linking**: Compile tests with `-DUNIT_TEST` to conditionally compile out main application code and mock system dependencies.
2. **Function Pointer Injection**: Replace actual I/O functions with test doubles using function pointers in a global struct:

```c
struct net_io_ops {
    ssize_t (*tap_read)(int, void*, size_t);
    ssize_t (*tap_write)(int, const void*, size_t);
};

extern struct net_io_ops io_ops; /* Default to real functions */
/* In tests: io_ops.tap_read = mock_tap_read; */
```

3. **Memory Leak Detection**: Use `valgrind` for memory leak checks: `valgrind --leak-check=full ./test_tcp_state`.
4. **Coverage Analysis**: Compile with `-fprofile-arcs -ftest-coverage`, run tests, then use `gcov` or `lcov` to generate coverage reports.
5. **Byte Order Helpers**: Create helper functions for test data:

```c
uint8_t test_arp_request[] = {
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff, /* dst MAC (broadcast) */
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, /* src MAC */
    0x08, 0x06,                         /* EtherType ARP */
    /* ARP payload... */
};
```

F. Milestone Checkpoint Commands

For each milestone, these commands should succeed:

Milestone 1 Checkpoint:

```bash
make test-link  # Build and run link layer tests
sudo ./tools/tap_setup.sh create  # Setup TAP device
sudo ./stack --tap tap0 --ip 192.168.1.100 &
# In another terminal:
sudo arping -c 2 -I tap0 192.168.1.100
# Should see ARP replies, test passes if stack responds
```

Milestone 2 Checkpoint:

```bash
make test-network
sudo ./stack --tap tap0 --ip 192.168.1.100 &
ping -c 4 192.168.1.100  # Should get 4 replies, 0% loss
./tools/stack_cli route show  # Should show default route
```

Milestone 3 Checkpoint:

```bash
make test-tcp-state
sudo ./stack --tap tap0 --ip 192.168.1.100 --tcp-listen 8080 &
echo "Hello" | nc -w 2 -v 192.168.1.100 8080
# Should connect, send data, and close
```

Milestone 4 Checkpoint:

```bash
make test-tcp-transfer
sudo ./stack --tap tap0 --ip 192.168.1.100 --tcp-echo 8080 &
# Test with packet loss
sudo tc qdisc add dev tap0 root netem loss 20%
dd if=/dev/zero bs=1K count=500 | nc -v 192.168.1.100 8080 > /dev/null
# Should complete despite losses, showing retransmissions in logs
sudo tc qdisc del dev tap0 root
```

G. Debugging Tips for Testing

Symptom	Likely Cause	How to Diagnose	Fix
ARP test fails: no reply	TAP device not configured, wrong IP, or ARP handler not called	Check `ip addr show tap0`; add logging in `netif_rx_packet` and `arp_input`; verify frame reaches parser	Ensure TAP device has IP, ARP handler registered for EtherType 0x0806
Ping test fails: no ICMP reply	IP checksum incorrect, wrong protocol dispatch, ICMP not implemented	Use Wireshark to see if request arrives; log in `ipv4_input` and `icmp_input`; verify checksum calculation	Check `ip_checksum` against known values; ensure `protocol == IPPROTO_ICMP` calls `icmp_input`
TCP connection hangs at SYN_SENT	SYN-ACK not received, ACK not sent, state machine stuck	Check tcpdump for SYN-ACK; log TCB state transitions; verify sequence numbers	Ensure SYN-ACK has correct ack_num (client_seq+1); client should send ACK for SYN-ACK
Data transfer extremely slow	Window size太小, congestion window not growing, retransmission timeout too high	Log `cwnd`, `ssthresh`, `snd_wnd` values; check for packet loss; monitor RTT estimates	Implement proper slow start; ensure ACKs advance `snd_una`; tune initial RTO
Memory leak detected by valgrind	TCBs not freed on close, buffers not released	Use valgrind with trace-children; add allocation/deallocation logging; check close path	Ensure `tcp_free_connection` frees all resources; free buffers in `tcp_buffer_destroy`
Test passes alone but fails in suite	Global state pollution between tests	Check for static/global variables not reset; use test setup/teardown functions	Add `test_setup()` and `test_teardown()` to reset all global state between tests
Packet comparison fails at byte X	Endianness issue, field offset wrong, padding differences	Hex dump both packets, highlight differing bytes; map byte offset to header field	Check `htonl`/`ntohl` usage; verify struct packing with `#pragma pack(1)`; check for compiler padding

12. Debugging Guide

> **Milestone(s):** 1, 2, 3, 4 (Debugging spans all implementation stages)

Building a TCP/IP stack from scratch is a complex endeavor that inevitably involves encountering subtle bugs. This debugging guide serves as a practical manual for diagnosing and fixing common implementation issues. Unlike debugging typical applications where you can rely on a working network stack, here the network stack *itself* is the buggy component, requiring specialized diagnostic approaches.

The core mental model for debugging Project Valiant is **the scientific method applied to network protocols**: observe a symptom, form a hypothesis about which layer or component is malfunctioning, design an experiment (often through packet inspection or targeted logging) to test the hypothesis, and then implement the fix. You are both the developer and the network administrator diagnosing a faulty networking device.

Common Bugs: Symptom → Cause → Fix

The following table catalogs the most frequent symptoms learners encounter when implementing Project Valiant, organized by milestone. Each entry provides a concrete symptom, the likely underlying cause, specific diagnostic steps, and the fix.

Symptom	Likely Milestone	Root Cause	Diagnostic Steps	Fix
"My TAP device won't send or receive any frames"	1	1. TAP device not properly configured with IP address. 2. Incorrect file descriptor handling in main I/O loop. 3. Lack of CAP_NET_ADMIN privileges (Linux).	1. Run `ip addr show` to verify TAP device exists and has an IP. 2. Use `strace` to see if `read()` / `write()` syscalls are being made. 3. Check return values from `tap_open()` and subsequent `read()` / `write()` calls.	1. Configure TAP: `sudo ip addr add 10.0.0.1/24 dev tap0`. 2. Ensure main loop calls `tap_read()` / `tap_write()` correctly. 3. Run stack with sudo or set capabilities: `sudo setcap cap_net_admin+ep ./valiant`.
"Ping doesn't reply — no ARP traffic seen"	1	1. Ethernet frame parser incorrectly reading EtherType. 2. Not handling broadcast MAC (`FF:FF:FF:FF:FF:FF`). 3. Byte-order bug in EtherType comparison.	1. Use `hex_dump()` on received frame. Verify `ethertype` field bytes. 2. Check if `netif_rx_packet()` is called for broadcast frames. 3. Log parsed EtherType value (hex) and compare to `ETHERTYPE_ARP` (0x0806).	1. Ensure `validate_buffer_length()` is used before accessing header fields. 2. Compare destination MAC byte-by-byte against broadcast address. 3. Use `ntohs()` when reading EtherType from wire, compare to host-order constants.
"Ping doesn't reply — ARP works but no ICMP response"	2	1. IP checksum calculation error. 2. Incorrect ICMP type/code for echo reply. 3. Wrong byte order in IP address fields.	1. Capture outgoing ICMP reply with Wireshark, verify checksum. 2. Log ICMP header fields: type should be 0 (reply), code 0. 3. Compare `src_ip` / `dst_ip` in hex dump to expected addresses.	1. Use `ip_checksum()` helper, ensure it handles odd-length data. 2. Set ICMP type to 0 (echo reply), copy identifier/sequence from request. 3. Use network byte order for IP addresses in headers (as `uint32_t`).
"Ping replies, but TCP connections fail"	3	1. TCP checksum error (missing pseudo-header). 2. Not finding matching `tcb` in `tcp_find_connection()`. 3. Initial sequence number (ISN) not randomized.	1. Use Wireshark to check TCP checksum validity. 2. Log 4-tuple (local/remote IP:port) used in lookup vs. incoming segment. 3. Check if SYN segment's SEQ is being ignored or set to fixed value.	1. Implement `tcp_checksum()` with proper pseudo-header (src/dst IP, protocol, TCP length). 2. Ensure listening `tcb` exists for server role, or create new `tcb` for SYN. 3. Generate random ISN (e.g., using `/dev/urandom` or time-based seed).
"TCP handshake hangs at SYN_SENT (client)"	3	1. SYN-ACK not being generated by server. 2. Client not recognizing SYN-ACK (wrong ACK number). 3. Firewall/routing dropping packets.	1. Use Wireshark to see if SYN-ACK is sent from server. 2. Log client's `snd_nxt` (ISN+1) and compare to SYN-ACK's ACK field. 3. Check `route_lookup()` returns valid route for destination.	1. Ensure server's `tcp_input()` creates new `tcb` in SYN_RCVD, sends SYN-ACK. 2. Client should validate ACK number = `snd_nxt` , transition to ESTABLISHED. 3. Add route: `route_add(0, 0, gateway_ip, iface_index)` for default gateway.
"TCP handshake hangs at SYN_RCVD (server)"	3	1. Client's ACK not received or processed. 2. Server's SYN-ACK retransmitted but client ignores (sequence mismatch). 3. `tcb` timeout and cleanup too aggressive.	1. Capture traffic: does client send ACK? Does server receive it? 2. Check SYN-ACK's SEQ number; client's ACK should be SEQ+1. 3. Log server's `tcb` state transitions; is it being cleaned up prematurely?	1. Ensure `tcp_input()` processes ACK flag correctly in SYN_RCVD state. 2. Server's ISN should also be random; ACK validation must use `seq_lt()` / `seq_diff()` . 3. Implement proper SYN-RECEIVED timer (e.g., 75 seconds) before cleanup.
"Data transfer is extremely slow (< 1 KB/s)"	4	1. Stop-and-wait behavior (window size = 1 segment). 2. Retransmission timeout (RTO) too high (seconds). 3. Congestion window (`cwnd`) stuck at 1 MSS.	1. Capture traffic: look for pattern of single packet → ACK → next packet. 2. Log RTO value; should start ~1 second, adjust with RTT measurements. 3. Check `tcp_send_window_available()` returns small value due to `cwnd` .	1. Implement sliding window: allow multiple segments up to `min(cwnd, rcv_wnd)` . 2. Initialize RTO to 3 seconds, implement Jacobson/Karels algorithm. 3. Enable slow start: on connection, `cwnd = 1 MSS` , double per RTT.
"Data transfer loses bytes or corrupts data"	4	1. Sequence number arithmetic errors with wraparound. 2. Buffer management bug (circular buffer indices incorrect). 3. Mishandling of PSH flag or not delivering data to app.	1. Log sequence numbers as 32-bit hex; check for monotonic increase. 2. Use `hex_dump()` on `rx_buffer` before/after writes, check indices. 3. Verify `tcp_receive_data()` copies data from `rx_buffer` correctly.	1. Use `seq_diff()` for all sequence number comparisons, handle 32-bit wrap. 2. Thoroughly test `tcp_buffer_write()` / `tcp_buffer_read()` with edge cases. 3. Deliver data when `PSH` flag set or when `rx_buffer` reaches threshold.
"Connection resets (RST) unexpectedly"	3,4	1. Receiving segment for non-existent connection (no `tcb`). 2. State machine error leading to RST generation.	1. Log all RST segments sent; check which condition triggered it. 2. Trace state machine: current state, event, next state.	1. Only send RST for valid cases (RFC 793): no matching `tcb` for incoming segment. 2. Review state transition table; ensure RST only sent in specific states.

Symptom	Likely Milestone	Root Cause	Diagnostic Steps	Fix
		3. Sequence number outside window (out-of-range).	3. Check window validation: `rcv_nxt ≤ SEQ < rcv_nxt + rcv_wnd`.	3. Implement proper window validation before accepting data.
"Memory usage grows unbounded"	All	1. `tcb` structures not freed after connection close. 2. ARP cache entries never expired. 3. Retransmission queue not cleaned after ACK.	1. Monitor `tcb` allocation count during connection lifecycle. 2. Log ARP cache size; check `last_updated` vs. current time. 3. Check `snd_una` advancement and removal of acknowledged segments.	1. Implement TIME_WAIT timer (2MSL) then free `tcb`. 2. Run periodic ARP cache cleanup based on `last_updated` timestamp. 3. When `snd_una` advances, free buffers for acknowledged data.
"Stack crashes on malformed packet"	All	1. Lack of bounds checking before accessing packet data. 2. Assuming fixed header lengths without checking `version_ihl` or `data_offset`. 3. Null pointer dereference in linked list traversal.	1. Run with `valgrind` to detect invalid reads. 2. Log each validation step; see which check fails before crash. 3. Add assertions before pointer dereferences.	1. Use `validate_buffer_length()` at every layer before parsing. 2. Use `validate_ip_header_length()` and `validate_tcp_header_length()`. 3. Check for NULL in `tcp_find_connection()` and route lookup returns.

> **Key Insight:** Most networking bugs manifest at a different layer than their cause. A TCP connection failure might be due to an ARP resolution problem. Always verify lower layers are functioning before debugging upper layers—this is the **layered debugging principle**.

Debugging Techniques & Tools

Effective debugging of a network stack requires a multi-faceted approach combining packet-level inspection, strategic logging, and controlled testing environments.

Packet Inspection with Wireshark/tcpdump

Mental Model: Wireshark is your network microscope—it lets you see the exact bytes traveling on the wire, revealing discrepancies between what your stack *thinks* it's sending/receiving and what's actually happening.

Technique	How to Use	Purpose
Live Capture on TAP	`sudo tcpdump -i tap0 -w debug.pcap`	Capture all frames to/from your stack for later analysis.
Filtering by Protocol	`tcpdump -i tap0 arp` or `tcpdump -i tap0 tcp`	Isolate specific protocol traffic to reduce noise.
Hex + ASCII Display	`tcpdump -i tap0 -XX`	See raw bytes alongside interpretation, crucial for spotting byte-order issues.
Wireshark Decoding	Open `.pcap`, right-click → "Decode As..." → Set TCP port to your stack's port.	Force Wireshark to interpret traffic using your protocol (helpful before stack is RFC-compliant).
Checksum Validation	In Wireshark: Preferences → Protocols → TCP/IP → "Validate checksum if possible"	Let Wireshark highlight checksum errors in your outgoing packets.

Diagnostic Workflow:

1. **Reproduce the bug** while capturing packets on the TAP interface.
2. **Filter to the relevant conversation** (e.g., by IP address or port).
3. **Examine key fields** in each layer:
   - Ethernet: Destination/source MAC, EtherType
   - IP: Source/dest IP, TTL, protocol, checksum
   - TCP: Sequence/ACK numbers, flags, window size, checksum
4. **Compare with RFC expectations**: Is the SYN flag set when it should be? Is the ACK number correct (previous SEQ + payload length + flags)?
5. **Follow the TCP stream** in Wireshark to see data transfer patterns and spot missing segments.

Strategic Logging

Since you cannot rely on a working network stack for debugging, logging becomes your primary insight into internal state. However, excessive logging will overwhelm; strategic logging focuses on state changes and decision points.

Recommended Logging Macros:

```c
#define LOG(level, fmt, ...) \
    if (level >= current_log_level) \
        fprintf(stderr, "[%s] %s:%d: " fmt "\n", \
                log_level_string[level], __FILE__, __LINE__, ##__VA_ARGS__)

// Specialized loggers for key components
#define LOG_STATE(conn, old, new) \
    LOG(LOG_INFO, "TCP %p: %s -> %s", conn, tcp_state_str(old), tcp_state_str(new))

#define LOG_PACKET(dir, proto, src, dst) \
    LOG(LOG_DEBUG, "%s %s %s -> %s", dir, proto, src, dst)
```

What to Log at Each Layer:

Layer	Critical Log Points	Sample Log Output
Link	Frame reception, ARP request/reply, cache updates	`[DEBUG] arp.c:45: ARP request for 10.0.0.2`
Network	IP packet received/sent, checksum result, routing decision	`[INFO] ipv4.c:102: Forwarding packet to 10.0.0.2 via gateway 10.0.0.1`
TCP	State transitions, segment arrival (SEQ/ACK/flags), window updates	`[INFO] tcp.c:233: TCP 0x55a1b: ESTABLISHED -> FIN_WAIT_1 (send FIN)`
Buffers	Buffer write/read operations, free space	`[DEBUG] tcp_buffer.c:78: rx_buffer 0x55a1c: wrote 1460 bytes, readable=2920`

Hex Dumping Packets: Always include a `hex_dump()` function to log raw packet data at suspicious points. This is invaluable for spotting byte-order issues, incorrect field offsets, or mangled data.

```c
void hex_dump(const char *label, const void *data, size_t len) {
    const uint8_t *bytes = data;
    printf("%s (%zu bytes):\n", label, len);
    for (size_t i = 0; i < len; i += 16) {
        printf("  %04zx: ", i);
        for (size_t j = 0; j < 16; j++) {
            if (i + j < len) printf("%02x ", bytes[i + j]);
            else printf("   ");
        }
        printf(" ");
        for (size_t j = 0; j < 16; j++) {
            if (i + j < len) {
                uint8_t b = bytes[i + j];
                printf("%c", (b >= 32 && b < 127) ? b : '.');
            }
        }
        printf("\n");
    }
}
```

Logging Strategy:

1. **Start with `LOG_ERROR` only** for initial bring-up.
2. **Enable `LOG_INFO` for state transitions** when debugging connection setup/teardown.
3. **Use `LOG_DEBUG` selectively** with compile-time flags (e.g., `-DDEBUG_ARP`) to avoid performance impact.
4. **Log to stderr with timestamps** to correlate with packet captures (Wireshark timestamps).

Building a Test Harness

A controlled testing environment isolates your stack from unpredictable real networks and provides reproducible test cases.

Components of a Test Harness:

Component	Purpose	Implementation
Mock TAP Device	Replace physical/virtual NIC with in-memory buffer for deterministic I/O.	`mock_tap_device_t` with ring buffer for frames.
Packet Injector	Programmatically send crafted packets to your stack.	Functions to build Ethernet/IP/TCP packets with specific fields.
Response Validator	Verify your stack's responses match expectations.	Compare generated frames against expected ones field-by-field.
State Inspector	Query internal state (ARP cache, routing table, TCBs).	Debug functions that print internal data structures.

Testing Approach:

1. **Unit Testing**: Test parsers/builders in isolation with known-good packet bytes.
2. **Integration Testing**: Use mock TAP to simulate sequences (ARP request→reply, TCP handshake).
3. **Regression Testing**: Save failing packet captures as test cases to prevent re-introduction of bugs.

Example Test Case for ARP:

```c
void test_arp_request_reply() {
    mock_tap_init();
    // Build ARP request for our IP
    uint8_t request_frame[MAX_FRAME_SIZE];
    size_t len = build_arp_request(request_frame, OUR_MAC, OUR_IP, TARGET_IP);
    mock_tap_write(request_frame, len);  // Inject into mock

    // Run stack processing
    netif_process_pending();

    // Check response
    uint8_t response_frame[MAX_FRAME_SIZE];
    ssize_t resp_len = mock_tap_read(response_frame, sizeof(response_frame));
    assert(resp_len > 0);

    // Validate response is ARP reply with correct MAC
    assert(is_arp_reply(response_frame, resp_len));
    assert(arp_target_mac_matches(response_frame, OUR_MAC));
}
```

Network Namespaces for Isolation: On Linux, use network namespaces to create isolated network environments without affecting host configuration:

```bash
# Create namespace
sudo ip netns add valiant-test
# Create veth pair to connect namespace to host
sudo ip link add veth0 type veth peer name veth1 netns valiant-test
# Configure IPs
sudo ip addr add 10.0.1.1/24 dev veth0
sudo ip netns exec valiant-test ip addr add 10.0.1.2/24 dev veth1
# Bring up interfaces
sudo ip link set veth0 up
sudo ip netns exec valiant-test ip link set veth1 up
# Run your stack inside namespace
sudo ip netns exec valiant-test ./valiant --tap tap0
```

Debugging State Machines

TCP's state machine is particularly prone to bugs. Use these techniques:

1. **Visualize State Transitions**: Maintain a table of `(current_state, event, next_state, action)` and log each transition. Compare against RFC 793's state diagram.
2. **Check All Events**: Ensure you handle not just common events (SYN, ACK, FIN) but also RST, unexpected segments, and timeouts.
3. **State Assertions**: At the start of `tcp_process()`, assert that the connection is in a valid state for the received segment.

Memory Debugging

Networking code involves complex buffer management. Use these tools:

Tool	Command	Purpose
valgrind	`valgrind --leak-check=full ./valiant`	Detect memory leaks, invalid reads/writes.
AddressSanitizer	`gcc -fsanitize=address -g ...`	Catch buffer overflows, use-after-free.
gcov	`gcc -fprofile-arcs -ftest-coverage ...`	Measure code coverage to identify untested paths.

> **Pro Tip:** When stuck, implement the **Minimal Working Example** (MWE) approach. Strip your stack down to the bare essentials needed to reproduce the bug (e.g., disable ARP caching, use fixed sequence numbers, disable congestion control). This often reveals the core issue obscured by complexity.

Implementation Guidance

A. Technology Recommendations Table

Debugging Aspect	Simple Option	Advanced Option
Packet Capture	`tcpdump` CLI with basic filters	Wireshark GUI with custom dissection plugins
Logging	`fprintf(stderr, ...)` with manual level control	Structured logging (JSON) with syslog integration
Memory Debugging	Manual allocation tracking with counters	Valgrind + custom heap allocator with guard zones
Unit Testing	Simple test functions with `assert()`	Unity/CMocka framework with mock objects
Network Isolation	Separate physical test network	Linux network namespaces + virtual Ethernet pairs

B. Recommended File/Module Structure

```
project-valiant/
├── src/
│   ├── debug/              # Debugging utilities
│   │   ├── hexdump.c       # hex_dump() implementation
│   │   ├── logger.c        # LOG() macro and log level management
│   │   └── inspector.c     # Functions to dump ARP cache, routing table, TCBs
│   ├── tests/              # Test harness
│   │   ├── mock_tap.c      # mock_tap_device_t implementation
│   │   ├── test_arp.c      # ARP unit tests
│   │   ├── test_tcp_handshake.c # TCP connection tests
│   │   └── test_runner.c   # Main test runner
│   └── ... (other components as in earlier sections)
├── tools/
│   ├── pcap2c.py           # Convert pcap to C array for test cases
│   └── packet_injector.c   # Standalone tool to craft and send packets
└── Makefile
```

C. Infrastructure Starter Code

Complete Hex Dump Utility:

```c
// src/debug/hexdump.c

#include <stdio.h>

#include <stdint.h>

#include <string.h>


void hex_dump(const char *label, const void *data, size_t len) {

    const uint8_t *bytes = (const uint8_t *)data;

    printf("[HEXDUMP] %s (%zu bytes):\n", label, len);



    for (size_t i = 0; i < len; i += 16) {

        printf("  %04zx: ", i);



        // Print hex bytes

        for (size_t j = 0; j < 16; j++) {

            if (i + j < len) {

                printf("%02x ", bytes[i + j]);

            } else {

                printf("   ");

            }

            if (j == 7) printf(" ");

        }



        printf(" |");

        // Print ASCII representation

        for (size_t j = 0; j < 16; j++) {

            if (i + j < len) {

                uint8_t b = bytes[i + j];

                printf("%c", (b >= 32 && b < 127) ? b : '.');

            } else {

                printf(" ");

            }

        }

        printf("|\n");

    }

    printf("\n");

}
```

Configurable Logger:

```c
// src/debug/logger.c

#include <stdio.h>

#include <stdarg.h>

#include <time.h>


static log_level_t current_log_level = LOG_INFO;

static const char *log_level_strings[] = {

    "DEBUG", "INFO", "WARN", "ERROR", "FATAL"

};


void log_set_level(log_level_t level) {

    current_log_level = level;

}


void log_message(log_level_t level, const char *file, int line, const char *fmt, ...) {

    if (level < current_log_level) return;



    time_t now = time(NULL);

    struct tm *tm = localtime(&now);



    fprintf(stderr, "[%02d:%02d:%02d] %-5s %s:%d: ",

            tm->tm_hour, tm->tm_min, tm->tm_sec,

            log_level_strings[level], file, line);



    va_list args;

    va_start(args, fmt);

    vfprintf(stderr, fmt, args);

    va_end(args);



    fprintf(stderr, "\n");

}


// Convenience macro (use this instead of direct function call)

#define LOG(level, ...) log_message(level, __FILE__, __LINE__, __VA_ARGS__)
```

D. Core Logic Skeleton Code

State Transition Logger for TCP:

```c
// src/tcp/tcp_state.c

void tcp_log_state_change(tcb_t *conn, tcp_state_t old_state, tcp_state_t new_state) {

    static const char *state_names[] = {

        "CLOSED", "LISTEN", "SYN_SENT", "SYN_RCVD",

        "ESTABLISHED", "FIN_WAIT_1", "FIN_WAIT_2", "CLOSE_WAIT",

        "CLOSING", "LAST_ACK", "TIME_WAIT"

    };



    char src_ip_str[16], dst_ip_str[16];

    ip_int_to_str(conn->local_ip, src_ip_str, sizeof(src_ip_str));

    ip_int_to_str(conn->remote_ip, dst_ip_str, sizeof(dst_ip_str));



    LOG(LOG_INFO, "TCP %s:%d -> %s:%d: %s -> %s",

        src_ip_str, conn->local_port,

        dst_ip_str, conn->remote_port,

        state_names[old_state], state_names[new_state]);

}


// In tcp_process(), after state change:

void tcp_process(tcb_t *conn, const tcp_hdr *hdr, const void *payload, size_t payload_len) {

    tcp_state_t old_state = conn->state;



    // TODO: Process segment based on current state and flags

    // (Implementation from Milestone 3 & 4)



    if (conn->state != old_state) {

        tcp_log_state_change(conn, old_state, conn->state);

    }

}
```

Packet Validation Debug Helper:

```c
// src/debug/inspector.c

void inspect_incoming_frame(const uint8_t *frame, size_t len) {

    if (len < ETH_HDR_LEN) {

        LOG(LOG_WARN, "Frame too short: %zu bytes", len);

        return;

    }



    const eth_hdr *eth = (const eth_hdr *)frame;

    uint16_t ethertype = ntohs(eth->ethertype);



    LOG(LOG_DEBUG, "Ethernet: dst=%02x:%02x:%02x:%02x:%02x:%02x src=%02x:%02x:%02x:%02x:%02x:%02x type=0x%04x",

        eth->dst_mac[0], eth->dst_mac[1], eth->dst_mac[2],

        eth->dst_mac[3], eth->dst_mac[4], eth->dst_mac[5],

        eth->src_mac[0], eth->src_mac[1], eth->src_mac[2],

        eth->src_mac[3], eth->src_mac[4], eth->src_mac[5],

        ethertype);



    if (ethertype == ETHERTYPE_IP && len >= ETH_HDR_LEN + IP_HDR_LEN) {

        const ip_hdr *ip = (const ip_hdr *)(frame + ETH_HDR_LEN);

        char src_ip[16], dst_ip[16];

        ip_int_to_str(ip->src_ip, src_ip, sizeof(src_ip));

        ip_int_to_str(ip->dst_ip, dst_ip, sizeof(dst_ip));



        LOG(LOG_DEBUG, "IPv4: %s -> %s proto=%d ttl=%d len=%d checksum=0x%04x",

            src_ip, dst_ip, ip->protocol, ip->ttl,

            ntohs(ip->total_len), ntohs(ip->checksum));

    }

}
```

E. Language-Specific Hints

For C Implementation:

- Use `__attribute__((packed))` for protocol structs to prevent compiler padding.
- Enable all warnings: `gcc -Wall -Wextra -Werror -Wno-unused-parameter`
- Compile with debugging symbols: `-g -Og` (optimize for debugging)
- Use `assert()` liberally for invariants but remember they're removed with `-DNDEBUG`.
- Consider using `clang-tidy` for static analysis of common C bugs.

Memory Debugging with Valgrind:

```bash
# Run with full memory checking

valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./valiant


# Generate suppressions for system libraries to reduce noise

valgrind --gen-suppressions=all --leak-check=full ./valiant 2>&1 | grep -A5 "== Conditional jump"
```

F. Milestone Checkpoint

Debugging Verification for Each Milestone:

Milestone	Verification Command	Expected Outcome	Debugging Focus
1	`ping -c 1 10.0.0.2` (from host to TAP IP)	ARP request/reply visible in tcpdump, ping responds	ARP cache, frame parsing, byte order
2	`ping -c 3 10.0.0.2`	ICMP echo request/reply with correct checksums, TTL decremented	IP checksum, ICMP type/code, routing
3	`nc -vz 10.0.0.2 8080`	TCP handshake completes (SYN→SYN-ACK→ACK), connection established	TCP state machine, sequence numbers, demultiplexing
4	`dd if=/dev/zero bs=1M count=10 | nc 10.0.0.2 8080`	Data transferred reliably, no corruption, flow control respected	Retransmission, congestion control, performance

Checkpoint Failure Diagnosis:

1. **No packets seen on wire**: Check TAP device configuration, CAP_NET_ADMIN, main I/O loop.
2. **Packets sent but no reply**: Verify protocol handlers are registered, checksums correct.
3. **Connection hangs**: Use Wireshark to see where handshake stops; add state transition logging.
4. **Data corruption**: Hex dump sent vs received data; check sequence number arithmetic.

G. Debugging Tips

For "Ghost Connections" (TCBs not cleaned up):

- Symptom: Memory leak, cannot re-use ports.
- Diagnosis: Log `tcb` allocation/deallocation; check TIME_WAIT timer fires.
- Fix: Ensure `tcp_close()` properly schedules TIME_WAIT (2MSL) timer.

For "Zero Window Stalls":

- Symptom: Data transfer stops, receiver advertises window = 0.
- Diagnosis: Log `rcv_wnd` updates; check if application reads data from buffer.
- Fix: Ensure `tcp_receive_data()` is called by application, freeing buffer space.

For "Spurious Retransmissions":

- Symptom: Same data sent multiple times despite ACKs received.
- Diagnosis: Log `snd_una`, `snd_nxt`, and ACK processing; check RTO calculation.
- Fix: Verify `tcp_ack_received()` correctly advances `snd_una`; tune RTO constants.

> **Final Advice:** When completely stuck, take a break then **read the RFC again**. TCP/IP's behavior is precisely specified; often the bug is a misunderstanding of the specification rather than a coding error. Use RFC 793 as your ultimate debugging reference.

13. Future Extensions

> **Milestone(s):** Beyond Milestone 4 (Optional Extensions)

Congratulations on completing the four core milestones of Project Valiant! You now have a fully functional TCP/IP stack capable of reliable communication. However, the journey of understanding computer networks doesn't end here—this implementation serves as a solid foundation upon which you can build numerous extensions that deepen your knowledge of networking protocols and systems programming. This section explores potential advanced features you could implement, highlighting how the current architecture accommodates these extensions and what new concepts each would teach you.

Ideas for Further Learning

The extensions below are organized from relatively straightforward additions that build directly on existing components to more complex projects that introduce entirely new protocol layers. Each represents a self-contained learning opportunity that would significantly enhance both your stack and your understanding of real-world networking systems.

Extension	Description	Educational Value	How Current Design Accommodates
UDP Protocol Support	Implement the User Datagram Protocol (RFC 768) as a connectionless, unreliable transport alternative to TCP.	Understanding connectionless vs connection-oriented paradigms, datagram semantics, multiplexing without state management.	The `ipv4_input` function already demultiplexes by protocol field (`IPPROTO_TCP` vs `IPPROTO_UDP`). The transport layer interface can be extended with `udp_input` and `udp_output` functions that mirror TCP's interface but without connection state.
IPv6 Implementation	Add support for Internet Protocol version 6 (RFC 2460), including 128-bit addressing, simplified header structure, and extension headers.	Modern IP addressing, header design evolution, dual-stack operation, and the transition from IPv4.	The network layer can be refactored to handle both IP versions through version-agnostic interfaces. The `ip_hdr` structure would be complemented by `ipv6_hdr` , and `route_lookup` would need to support both address families.
TCP Selective Acknowledgments (SACK)	Implement TCP Selective Acknowledgment option (RFC 2018) to improve performance in lossy networks by acknowledging non-contiguous data ranges.	Advanced TCP performance optimization, handling optional header fields, and sophisticated retransmission strategies.	The `tcb` structure already has fields for tracking unacknowledged data. SACK would add a SACK block list to track which non-contiguous ranges have been received, enhancing `tcp_ack_received` to process SACK information.
TLS/SSL Integration	Add Transport Layer Security (TLS 1.2/1.3) on top of TCP to provide encrypted, authenticated communication channels.	Cryptography fundamentals, secure protocol design, handshake negotiation, and record layer encapsulation.	The `tcb` structure's `app_data` field can hold TLS session state. The `tcp_receive_data` and `tcp_send_data` interfaces would wrap TLS record processing, encrypting/decrypting payloads before handing data to the application.
HTTP/1.1 Server	Build a simple HTTP/1.1 web server that listens on port 80, parses HTTP requests, and serves static files or dynamic responses.	Application-layer protocol design, request/response semantics, MIME types, and the complete protocol stack from physical layer to application.	The socket API abstraction (implicit in the current design) provides the interface for applications. An HTTP server would use `tcp_listen` on port 80, `tcp_accept` connections, and process data received via `tcp_receive_data` .
Dynamic Routing Protocol (RIP)	Implement the Routing Information Protocol (RIP v2, RFC 2453) to exchange routing information between multiple instances of your stack.	Distributed routing algorithms, distance-vector protocols, route advertisement, and convergence behavior.	The `route_entry` structure and `route_add` function provide the foundation. RIP would add periodic routing updates, neighbor discovery, and route metric calculation, integrating with the existing routing table.
Network Address Translation (NAT)	Add NAT functionality to translate private IP addresses to a public IP when packets exit to the external network.	Address conservation, port translation, connection tracking, and stateful packet rewriting.	The `ipv4_output` function would be extended to check NAT rules and modify source IP/port. A NAT table (structurally similar to the TCP connection table) would track mappings and timeouts.
Quality of Service (DiffServ)	Implement Differentiated Services (RFC 2474) to classify and prioritize traffic based on the DS field in the IP header.	Traffic engineering, packet scheduling algorithms, queue management, and service differentiation.	The `ip_hdr` structure's `tos` field provides the DS codepoint. The network layer would need priority queues and a scheduler in `ipv4_output` to handle packets based on their classification.
Packet Filtering Firewall	Add stateful packet filtering capabilities to accept, reject, or drop packets based on configurable rules (source/destination, protocol, connection state).	Security policy implementation, stateful inspection, rule matching algorithms, and access control.	The `ipv4_input` and `tcp_input` functions would consult a firewall rule table before processing. The `tcb` structure helps track connection state for stateful rules.
Multicast Support (IGMP)	Implement Internet Group Management Protocol (IGMP v2, RFC 2236) to handle IP multicast group membership and forwarding.	One-to-many communication, group management protocols, and efficient data distribution.	The IP layer would need to recognize multicast addresses (224.0.0.0/4). IGMP messages would be handled similarly to ICMP, with group membership tracked in a separate table affecting routing decisions.

Detailed Exploration of Selected Extensions

UDP Protocol Support: This is the most natural first extension after completing the TCP implementation. Unlike TCP's complex state machine and reliability guarantees, UDP provides a simple, connectionless datagram service. Implementing UDP would involve:

1. **New Protocol Handler:** Create `udp_input` and `udp_output` functions analogous to their TCP counterparts but without connection state management.
2. **Demultiplexing:** Extend `ipv4_input` to call `udp_input` when the IP protocol field equals `IPPROTO_UDP` (value 17).
3. **Socket Abstraction:** UDP "sockets" would be simpler than TCP sockets—just a local IP/port pair without connection state.
4. **Checksum Computation:** UDP includes an optional checksum (mandatory for IPv6) calculated similarly to TCP's checksum with a pseudo-header.

> **Design Insight:** UDP's simplicity highlights the value trade-off in protocol design: TCP provides reliability at the cost of complexity and overhead, while UDP offers minimal overhead but pushes reliability concerns to the application layer.

TLS/SSL Integration: Adding encryption transforms your educational stack into a platform for studying secure communications. This extension would involve implementing a simplified TLS 1.2 or 1.3 handshake and record layer:

1. **Session State:** Extend the `tcb` structure to include TLS session context (cipher suite, keys, sequence numbers).
2. **Handshake Integration:** Intercept the initial bytes of a TCP connection to perform TLS handshake before exposing the stream to the application.
3. **Record Processing:** Wrap `tcp_send_data` and `tcp_receive_data` to encrypt/decrypt using the negotiated cipher.
4. **Certificate Handling:** For learning purposes, you might implement simple pre-shared key or self-signed certificate validation.

The mental model for TLS integration is a **secure envelope service**: Just as you might place a letter (application data) inside a tamper-evident, locked envelope (TLS record) before handing it to the postal service (TCP), TLS wraps application data in encrypted records before transmission.

HTTP/1.1 Server: Building a web server on top of your stack provides the satisfying experience of seeing the complete protocol stack in action—from Ethernet frames carrying IP packets containing TCP segments transporting HTTP requests. This extension would:

1. **Listen for Connections:** Use `tcp_listen` on port 80 to accept incoming HTTP connections.
2. **Parse HTTP Requests:** Implement an HTTP parser that reads request lines, headers, and body from the TCP stream.
3. **Generate Responses:** Create proper HTTP responses with status codes, headers, and content.
4. **Serve Resources:** Map URL paths to file system resources or generate dynamic content.

> **Architecture Accommodation:** The existing `tcb` structure's `rx_buffer` and `tx_buffer` provide the necessary buffering for HTTP request/response data. The application would register callbacks or use a simple event loop to handle multiple concurrent connections, demonstrating how real servers manage numerous simultaneous clients.

Design Principles for Extensions

When implementing any of these extensions, consider these guiding principles derived from the existing architecture:

1. **Layered Isolation:** Keep protocol layers decoupled. For example, UDP implementation shouldn't modify TCP code, and TLS should sit cleanly between the application and TCP transport.
2. **Configuration Hooks:** The current design intentionally leaves room for configuration (e.g., `route_add` for routing tables). Extensions should follow this pattern with clear initialization and configuration APIs.
3. **State Management Consistency:** Follow the pattern established by the `tcb` structure for connection state. New stateful protocols (like NAT or firewall connection tracking) should use similar timeout and cleanup mechanisms.
4. **Error Handling Continuity:** Use the existing `stack_error_t` enumeration and error reporting mechanisms rather than inventing new error systems.

Implementation Guidance

While full implementation of these extensions is beyond the scope of this document, here are starting points for the most commonly pursued extensions:

UDP Support Skeleton

File Structure Addition:

```
src/
├── udp/
│   ├── udp.c           # UDP protocol implementation
│   ├── udp.h           # UDP header definitions and API
│   └── udp_test.c      # UDP-specific tests
```

Core Data Structure:

```c
/* Add to existing protocol definitions in network.h or similar */
#define IPPROTO_UDP 17

/* UDP header structure (RFC 768) */
typedef struct udp_hdr {
    uint16_t src_port;      /* Source port */
    uint16_t dst_port;      /* Destination port */
    uint16_t length;        /* Length of UDP header + data */
    uint16_t checksum;      /* Optional checksum (0 means none) */
} __attribute__((packed)) udp_hdr_t;
```

Interface Functions:

```c
/* In udp.h */
int udp_output(uint32_t dst_ip, uint16_t dst_port,
               uint16_t src_port, const void *payload,
               size_t payload_len);

void udp_input(const uint8_t *packet_data, size_t len,
               uint32_t src_ip, uint32_t dst_ip);
```

Integration Point in IP Layer:

```c
/* In ipv4_input function, add UDP case: */

void ipv4_input(const uint8_t *packet_data, size_t len, uint32_t from_ip) {

    // ... existing IP header validation ...



    switch (ip_header->protocol) {

        case IPPROTO_TCP:

            tcp_input(packet_data + ip_hlen, len - ip_hlen,

                      ip_header->src_ip, ip_header->dst_ip);

            break;

        case IPPROTO_UDP:

            udp_input(packet_data + ip_hlen, len - ip_hlen,

                      ip_header->src_ip, ip_header->dst_ip);

            break;

        case IPPROTO_ICMP:

            icmp_input(packet_data + ip_hlen, len - ip_hlen,

                       ip_header->src_ip, ip_header->dst_ip);

            break;

        default:

            LOG(LOG_WARN, "Unhandled IP protocol: %d", ip_header->protocol);

            break;

    }

}
```

Simple HTTP Server Skeleton

File Structure:

```
src/
├── http/
│   ├── http_server.c    # HTTP server implementation
│   ├── http_parser.c    # HTTP request parsing
│   └── http_response.c  # HTTP response generation
```

Core Server Loop Concept:

```c
/* Simplified HTTP server using the TCP stack */

int http_server_start(uint16_t port) {

    // TODO 1: Create listening TCP socket using tcp_listen(port)

    // TODO 2: Enter infinite loop: tcp_accept() to get new connections

    // TODO 3: For each connection: read HTTP request via tcp_receive_data()

    // TODO 4: Parse request, generate appropriate HTTP response

    // TODO 5: Send response via tcp_send_data(), close connection

    // TODO 6: Handle errors and clean up resources

    return 0;

}
```

HTTP Integration with Existing Stack:

```c
/* Example of how an HTTP server would use the TCP API */
void handle_http_connection(tcb_t *conn) {
    char request_buffer[4096];
    char response_buffer[4096];

    // Read HTTP request from TCP connection
    ssize_t bytes_read = tcp_receive_data(conn, request_buffer,
                                          sizeof(request_buffer) - 1);
    if (bytes_read > 0) {
        request_buffer[bytes_read] = '\0';

        // Parse HTTP request (simplified)
        if (strncmp(request_buffer, "GET ", 4) == 0) {
            // Generate HTTP response
            const char *response =
                "HTTP/1.1 200 OK\r\n"
                "Content-Type: text/html\r\n"
                "Content-Length: 25\r\n"
                "\r\n"
                "<h1>Hello from Valiant!</h1>";

            tcp_send_data(conn, response, strlen(response));
        }
    }

    // Gracefully close connection
    tcp_close(conn);
}
```

Learning Pathways

Depending on your interests, consider these learning pathways:

1. **Protocol Explorer Pathway:** UDP → IPv6 → Multicast (IGMP) → Dynamic Routing (RIP)
2. **Performance Optimizer Pathway:** TCP SACK → Quality of Service → Advanced Congestion Control
3. **Security Specialist Pathway:** Firewall → NAT → TLS/SSL → HTTPS Server
4. **Application Developer Pathway:** HTTP Server → WebSocket Support → Simple REST API

Each extension not only adds functionality but deepens your understanding of how real-world network stacks evolve to meet new requirements. The modular design of Project Valiant makes these extensions feasible without major architectural overhauls, demonstrating the value of clean layering and well-defined interfaces in systems software design.

---

14. Glossary

> **Milestone(s):** All (Cross-Cutting Terminology Reference)

This glossary provides clear, concise definitions for key technical terms, acronyms, and domain-specific vocabulary used throughout the Project Valiant design document. Think of this as your **technical dictionary**—a reference you can consult when encountering unfamiliar networking concepts or implementation details.

Terminology Reference

The following table lists terms alphabetically, providing their definitions and references to relevant sections for deeper understanding.

Term	Definition	Section Reference
ACK	An abbreviation for **Acknowledgment**. In TCP, a control flag (`TCP_ACK`) and accompanying acknowledgment number (`ack_num`) that informs the sender which sequence number the receiver expects next, confirming successful receipt of data.	7, 8
Active Open	The process by which a client initiates a TCP connection by sending a SYN segment to a server.	7
Advertised Window	A TCP receiver's advertised window size (`rcv_wnd`) communicated via the `window` field in the TCP header, indicating how much buffer space it has available to receive new data.	8
ARP (Address Resolution Protocol)	A link-layer protocol that maps Internet Protocol (IP) addresses to physical MAC addresses on a local network. ARP operates via request/reply messages to discover the hardware address associated with a given IP address.	5
ARP Cache	A table (typically stored as an array of `arp_entry` structs) that maintains recently learned IP-to-MAC address mappings to avoid repeated ARP requests.	4, 5
ARP Cache Poisoning	A network attack where a malicious actor sends forged ARP replies to corrupt the IP-to-MAC mapping table of other hosts, potentially leading to traffic interception or denial of service.	5, 10
Acknowledgment Number	In TCP, a 32-bit field (`ack_num`) that indicates the next sequence number the receiver expects, thereby acknowledging all data with lower sequence numbers.	4, 7, 8
Byte Order	The order in which bytes of a multi-byte numeric value are stored in memory. **Network Byte Order** is big-endian (most significant byte first), used in protocol headers; **Host Byte Order** varies by architecture (little-endian on x86). Functions like `htonl()` and `ntohl()` convert between them.	5, 6
Broadcast MAC	The Ethernet destination address `FF:FF:FF:FF:FF:FF` , indicating a frame intended for all devices on the local network segment.	5
Checksum	A value computed from a block of data (e.g., IP header, TCP segment) used to detect corruption during transmission. Receivers verify checksums and discard corrupted packets.	6, 8
Congestion Control	A mechanism in TCP that prevents a sender from overwhelming the network by dynamically adjusting its sending rate based on inferred network conditions, primarily through the **congestion window (cwnd)**.	8
Congestion Window (cwnd)	A sender-side limit, measured in bytes, representing the maximum amount of unacknowledged data the sender can have in flight based on perceived network capacity. The actual sending window is the minimum of `cwnd` and the receiver's advertised window.	8
Congestion Avoidance	The phase of TCP congestion control that follows **Slow Start**, where `cwnd` increases additively (by one maximum segment size per round-trip time) to probe for available bandwidth without causing congestion.	8
Connection Teardown	The process of gracefully closing a TCP connection, typically involving a **four-way handshake** of FIN and ACK segments.	7
Control Block	See **TCP Control Block (TCB)**.	4, 7
cwnd	See **Congestion Window**.	8
Data Offset	In the TCP header, a 4-bit field indicating the length of the TCP header in 32-bit words, used to locate the start of the payload. Also known as the header length.	4, 7
Defensive Parsing	A validation strategy where every input (e.g., packet headers) is rigorously checked for correct length, range, and structural integrity before processing to prevent crashes or security vulnerabilities.	10
Demultiplexing	The process of directing an incoming network segment to the correct higher-layer entity (e.g., a specific TCP connection) based on header fields like source/destination IP addresses and port numbers.	7, 9
Duplicate ACK	An acknowledgment that repeats the same acknowledgment number as previously sent. Three duplicate ACKs trigger **Fast Retransmit** in TCP.	8
Encapsulation	The process where each networking layer adds its own protocol header (and possibly trailer) to the data unit received from the layer above, creating nested protocol data units (e.g., TCP segment inside IP packet inside Ethernet frame).	3, 9
Ethernet Frame	The data unit at the link layer, consisting of a header (source/destination MAC addresses, EtherType), payload, and optional frame check sequence.	4, 5
EtherType	A 2-byte field in the Ethernet header indicating the protocol of the encapsulated payload (e.g., `ETHERTYPE_IP` for IPv4, `ETHERTYPE_ARP` for ARP).	4, 5
Fast Retransmit	A TCP recovery mechanism where a sender retransmits a segment upon receiving three duplicate ACKs, inferring segment loss without waiting for a retransmission timeout.	8
FIN	A TCP control flag (`TCP_FIN`) used to indicate that the sender has no more data to send, initiating connection teardown.	7

Term	Definition	Section Reference
Flow Control	A mechanism in TCP that prevents a sender from overwhelming a receiver by ensuring the sender does not transmit more data than the receiver's buffer can hold, as indicated by the receiver's advertised window.	8
Four-Way Handshake	The sequence of FIN and ACK segments exchanged to gracefully close a TCP connection from both ends: FIN from one side, ACK from the other, then FIN from the other side, and final ACK.	7
Fragmentation	The process of dividing a large IP packet into smaller fragments to fit the maximum transmission unit (MTU) of a network link. Project Valiant does not implement fragmentation.	6
Gateway	In routing, the IP address of the next-hop router to which packets should be forwarded to reach a destination network.	4, 6
Graceful Degradation	A design principle where the system continues to function at reduced capacity when resources are constrained, rather than failing completely.	10
Header Checksum	In IPv4, a 16-bit checksum computed over the IP header only (not the payload) to verify header integrity.	6
Host Byte Order	The byte order native to the CPU architecture (little-endian on x86). Must be converted to/from network byte order for protocol headers.	5
ICMP (Internet Control Message Protocol)	A supporting network-layer protocol used for diagnostic and error-reporting messages, such as Echo Request/Reply (ping) and Destination Unreachable.	6
IP Packet	The protocol data unit of the Internet Protocol, consisting of an IP header (with source/destination addresses) and a payload (e.g., a TCP segment or ICMP message).	4, 6
IP Routing	The process of forwarding IP packets from one network to another based on destination IP addresses and a routing table.	6
Layered Architecture	A software design pattern where components are organized in hierarchical layers, each with specific responsibilities and communicating only with adjacent layers (e.g., the OSI or TCP/IP model).	3
Link Layer	The networking layer (also known as Layer 2) responsible for communication between devices on the same physical network, using MAC addresses and protocols like Ethernet and ARP.	3, 5
Listen	A TCP server state (`LISTEN`) where a socket is waiting for incoming connection requests (SYN segments).	7
Longest Prefix Match	The algorithm used by IP routers to select the most specific route for a destination address. Among multiple matching routes, the one with the longest subnet mask (smallest network) is chosen.	6
MAC Address	A 6-byte hardware address (e.g., `00:11:22:AA:BB:CC`) uniquely identifying a network interface at the link layer. Also called a physical address.	4, 5
Maximum Segment Size (MSS)	The largest amount of data, in bytes, that TCP will send in a single segment. Typically negotiated during connection establishment.	7, 8
Maximum Transmission Unit (MTU)	The largest size of a data unit (e.g., Ethernet frame) that can be transmitted on a network link without fragmentation.	5, 6
Network Byte Order	Big-endian byte order (most significant byte first) used for multi-byte integers in network protocol headers. Standardized to ensure interoperability across different host architectures.	5, 6
Network Layer	The networking layer (also known as Layer 3) responsible for logical addressing (IP), routing between networks, and packet forwarding.	3, 6
Next-Hop	The immediate next router (gateway) or destination host to which a packet should be sent based on routing table lookup.	6
Passive Open	The process by which a server prepares to accept TCP connections by creating a listening socket and entering the `LISTEN` state.	7
Payload	The actual data carried within a protocol data unit (e.g., the TCP data inside an IP packet, or the IP packet inside an Ethernet frame).	9
Ping	A network diagnostic tool that uses ICMP Echo Request and Echo Reply messages to test reachability and measure round-trip time.	6
Pseudo-Header	A conceptual header used when computing the TCP checksum, containing source/destination IP addresses, protocol number, and TCP length. This ensures the checksum also validates the integrity of critical IP header fields.	8
Raw Socket	A type of socket that provides direct access to lower-layer protocols (e.g., Ethernet frames or IP packets). Project Valiant uses a TAP device instead for simplicity.	5
Reliable Delivery	A transport-layer service guarantee that data sent will be delivered correctly, in order, and without loss or duplication, typically implemented via acknowledgments and retransmissions (as in TCP).	8

Term	Definition	Section Reference
Resource Limits	Configurable constraints on system resources (memory, number of connections, buffer sizes) to prevent exhaustion and ensure graceful degradation under load.	10
Retransmission Timeout (RTO)	A dynamically calculated timer duration after which an unacknowledged TCP segment is retransmitted, assuming loss.	8
RFC 793	The foundational specification document defining the Transmission Control Protocol (TCP), published by the Internet Engineering Task Force (IETF).	Context
Round-Trip Time (RTT)	The time taken for a packet to travel from sender to receiver and for the acknowledgment to return. TCP estimates RTT to adjust retransmission timeouts.	8
Route Entry	A data structure (`route_entry`) representing a single routing rule, containing destination network, netmask, gateway, interface, and metric.	4, 6
Routing Table	A data structure (typically a linked list or trie of `route_entry` structs) that maps destination IP network prefixes to next-hop gateways and interfaces for packet forwarding decisions.	4, 6
RST	A TCP control flag (`TCP_RST`) used to immediately reset (abort) a connection, typically in response to an error or unexpected segment.	7, 10
Segment	The protocol data unit of TCP, consisting of a TCP header and optional data payload.	4, 7
Sequence Number	In TCP, a 32-bit field (`seq_num`) that identifies the byte offset of the first data byte in the segment (except during connection establishment, where it is the Initial Sequence Number).	4, 7, 8
Sliding Window	A protocol mechanism that allows a sender to transmit multiple segments (up to a window size) without waiting for individual acknowledgments, improving throughput. The window "slides" forward as acknowledgments are received.	8
Slow Start	The initial phase of TCP congestion control where the congestion window (`cwnd`) increases exponentially (doubling each round-trip time) to quickly probe for available bandwidth.	8
Socket	An abstraction representing an endpoint for network communication, typically identified by an IP address and port number pair.	3
sscanf	A C standard library function used to parse formatted input from strings, such as converting dotted-decimal IP addresses to integers.	6
State Machine	A finite-state model governing the lifecycle of a TCP connection, with defined states (e.g., `LISTEN` , `ESTABLISHED`) and transitions triggered by events like segment arrival or user calls.	7
SYN	A TCP control flag (`TCP_SYN`) used to initiate a connection during the three-way handshake. The SYN segment carries the Initial Sequence Number (ISN).	7
SYN-ACK	A TCP segment with both SYN and ACK flags set, sent by a server in response to a client's SYN during connection establishment.	7
TAP Device	A virtual network kernel interface that provides Ethernet frame-level access to user-space programs, allowing implementation of a network stack without raw sockets or kernel modifications.	5
TCP (Transmission Control Protocol)	A connection-oriented, reliable, byte-stream transport protocol providing in-order delivery, flow control, and congestion control.	7, 8
TCP Control Block (TCB)	The central data structure (`tcb` or `tcb_t`) containing all state information for a single TCP connection, including sequence numbers, window sizes, buffers, and timers.	4, 7
TCP/IP Stack	A layered software implementation of the TCP/IP protocol suite, handling communication from the physical network up to the application layer.	Context
Three-Way Handshake	The SYN, SYN-ACK, ACK exchange used to establish a TCP connection, synchronizing sequence numbers and negotiating parameters.	7
Time To Live (TTL)	An 8-bit field in the IP header that limits a packet's lifetime (measured in hops) to prevent infinite routing loops. Each router decrements TTL by 1; if it reaches 0, the packet is discarded and an ICMP Time Exceeded message may be sent.	6
Transport Layer	The networking layer (also known as Layer 4) responsible for end-to-end communication between applications, providing services like reliability (TCP) or simple datagram delivery (UDP).	3, 7, 8
TTL	See **Time To Live**.	6
Usable Window	The amount of new data a TCP sender can transmit at a given moment, calculated as the minimum of the congestion window and the receiver's advertised window, minus the amount of data already in flight (sent but not yet acknowledged).	8

Term	Definition	Section Reference
Validation Pipeline	A series of checks applied to incoming packets in a specific order (from structural integrity to semantic validity) to efficiently detect and discard malformed or malicious packets.	10
Window Scale Factor	An optional TCP extension (RFC 1323) that allows the advertised window to be scaled by a power of two, enabling windows larger than 65,535 bytes. Project Valiant does not implement this.	8
Wireshark	A popular network protocol analyzer (packet sniffer) used to capture and inspect network traffic, invaluable for debugging network stack implementations.	12