System Integration and Networking Lecture Notes

Compiled from Project Documents

September 7, 2025

Contents

1 Course Outline: An Introduction to System Integration

This document outlines a short course designed to introduce our software team to the fundamental concepts of system integration, from single-computer architecture to networked systems.

1.1 Chapter 1: The Heart of the Machine - Computer Architecture Fundamentals

1.1.1 Objective

Understand the primary components of a single computer and how they interact.

1.2 Chapter 3: Connecting the Dots - Network Architectures and Layers

1.2.1 Objective

Understand how different computer systems communicate with each other over a network.

1.2.2 Topics

- **The Need for Layers:** Simplifying complexity through abstraction.
- **The TCP/IP Model (A 5-Layer View):** Application, Transport, Network, Data Link, Physical.

1.2.3 Visualizations & Diagrams

```
graph TD
    subgraph Your Computer
        A[Application Layer - HTTP] --> B(Transport Layer - TCP);
        B --> C(Network Layer - IP);
        C --> D(Data Link Layer - WiFi/Ethernet);
        D --> E(Physical Layer - Radio Waves/Cables);
    end
    subgraph Web Server
        F[Application Layer - HTTP] --> G(Transport Layer - TCP);
        G --> H(Network Layer - IP);
        H --> I(Data Link Layer - Ethernet);
        I --> J(Physical Layer - Cables);
    end
    E -- The Internet --> J;
```

1.2.4 Real-World Application

Ordering a Pizza Online:

- **Application:** You use a web browser (HTTP) to place an order.
- **Transport:** TCP breaks your order into numbered packets to ensure the restaurant gets all the details correctly.
- **Network:** IP puts the restaurant's address on each packet.
- **Data Link:** Your Wi-Fi sends the packets to your router.
- **Physical:** The actual radio waves carry the data.

The restaurant's server receives the data in reverse order.

1.2.5 References & Further Reading

- **Video:** The OSI Model Explained: https://www.youtube.com/watch?v=vv4y_uOneC0 (Closely related to TCP/IP).
- **Article:** "How TCP/IP Works" by HowStuffWorks: https://computer.howstuffworks.com/tcp-ip.htm

1.3 Chapter 4: Tying It All Together - A Practical Example

1.3.1 Objective

Trace a single, common action from start to finish to see how all the concepts interact.

1.3.2 Scenario

"Loading `google.com` in a web browser."

1.3.3 Visualizations & Diagrams

```
sequenceDiagram
   participant User
   participant Browser
   participant OS
   participant Router
   participant DNS_Server
   participant Google_Server

   User->>Browser: Enters "google.com"
   Browser->>OS: Need IP for "google.com"
   OS->>Router: DNS Query
   Router->>DNS_Server: Where is "google.com"?
   DNS_Server-->>Router: IP is 142.250.190.78
   Router-->>OS: Here is the IP
   OS-->>Browser: Here is the IP
   Browser->>Google_Server: HTTP GET request to 142.250.190.78
   Google_Server-->>Browser: HTTP 200 OK (sends webpage)
   Browser->>User: Renders the webpage
```

1.3.4 References & Further Reading

- **Article:** "What happens when you type google.com into your browser's address box and press enter?" - A classic, detailed explanation: https://github.com/alex/what-happens-when
- **Tool:** Visual Trace Route tools like `mtr` (command line) or online versions can show the network path packets take.

1.4 Chapter 5: Network Communication Patterns

1.4.1 Objective

Explore common methods and patterns for communication between systems.

1.4.2 Topics

- **HTTP Communication:** Request/Response model.
- **Sockets:** Low-level, bidirectional communication.
- **Web Servers:** The role of servers like Nginx.
- **Message Queues:** Decoupling systems.
- **Publish-Subscribe Pattern:** Scalable messaging.

1.4.3 Visualizations & Diagrams

```
graph TD
    subgraph Message Queue
        Producer -- "Message" --> Queue((Queue));
        Queue -- "Message" --> Consumer;
    end
    subgraph Publish-Subscribe
        Publisher -- "Topic A" --> Broker((Topic));
        Broker -- "Topic A" --> Subscriber1;
        Broker -- "Topic A" --> Subscriber2;
    end
```

1.4.4 Real-World Application

- **Message Queue:** An e-commerce site. When you place an order, the web server (Producer) doesn't handle payment, shipping, and email all at once. It just puts an "Order Placed" message onto a **Queue**. Separate services (Consumers) for payment, inventory, and notifications pick up the message and do their work independently. This makes the checkout process fast and reliable.
- **Publish-Subscribe:** A live sports app. A central service (Publisher) publishes score updates to a "game-123" topic. Thousands of users (Subscribers) who are following that game receive the updates in real-time.

1.4.5 References & Further Reading

- **Video:** What is a Message Queue?: https://www.youtube.com/watch?v=rYw_cNJ2YpE
- **Article:** "Understanding Publish/Subscribe Messaging" by AWS: https://aws.amazon.com/pub-sub-messaging/

1.5 Chapter 6: Concurrency and Parallelism

1.5.1 Objective

Understand different models for executing multiple tasks at the same time.

1.5.2 Topics

- **Concurrent Processing (Threads):** Independent execution paths in one process.
- **Parallel Processing:** Simultaneous execution on multiple cores.
- **Asynchronous Processing (Async):** Non-blocking operations.

1.5.3 Visualizations & Diagrams

```
graph TD
    subgraph Concurrency (1 Core)
        direction LR
        Core1 -- "Task A (part 1)" --> CtxSwitch1(Switch);
        CtxSwitch1 -- "Task B (part 1)" --> CtxSwitch2(Switch);
        CtxSwitch2 -- "Task A (part 2)" --> Continue["..."];
    end
    subgraph Parallelism (2 Cores)
        direction LR
        CoreA -- "Task A (all)" --> DoneA;
        CoreB -- "Task B (all)" --> DoneB;
    end
```

1.5.4 Real-World Application

- **Concurrency (Threads):** In a word processor, one thread accepts your typing while another thread in the background is constantly checking your spelling and grammar. It feels simultaneous, but the CPU is rapidly switching between the two tasks.
- **Parallelism:** Rendering a 3D animated movie. The job is split up so that each CPU core (or even different machines) renders a different frame at the exact same time.
- **Asynchronous:** A web server loading data from a database. Instead of freezing while it waits for the database query to return, it starts the query and then immediately goes on to handle other user requests. When the data is ready, it picks up where it left off.

1.5.5 References & Further Reading

- **Video:** Concurrency vs Parallelism: https://www.youtube.com/watch?v=lK4oge36T-s
- **Article:** "Async IO Explained": https://www.ably.com/blog/async-io-explained

1.6 Chapter 7: Operating System Fundamentals

1.6.1 Objective

Gain a foundational understanding of the role of the Operating System.

1.6.2 Topics

- **Core Functions:** Process management, memory management, file systems, I/O.
- **Windows vs. Linux:** High-level comparison.
- **Hyper-Threading:** A single physical core acting as two virtual cores.

1.6.3 Visualizations & Diagrams

```
graph TD
    subgraph OS Kernel
        direction LR
        Scheduler --> P1(Process 1);
        Scheduler --> P2(Process 2);
        MemoryManager -- "allocates" --> P1_Mem[Memory for P1];
        MemoryManager -- "allocates" --> P2_Mem[Memory for P2];
    end
    subgraph Hardware
        CPU & RAM & Disk;
    end
    P1 & P2 -- "run on" --> CPU;
    P1_Mem & P2_Mem -- "reside in" --> RAM;
    OS_Kernel -- "manages" --> Hardware;
```

1.6.4 Real-World Application

Process Management: You can have a web browser, a music player, and a code editor all running at the same time. The OS **Scheduler** rapidly switches the CPU's attention between them, giving each a slice of time so they all appear to run simultaneously. The **Memory Manager** ensures the browser can't accidentally read data from your code editor, providing stability and security.

1.6.5 References & Further Reading

- **Video:** Crash Course Computer Science - Operating Systems: https://www.youtube.com/watch?v=26QPDBe-NB8
- **Article:** "What is an Operating System?" by FreeCodeCamp: https://www.freecodecamp.org/news/what-is-an-operating-system-definition-for-beginners/

1.7 Chapter 8: Virtualization and Isolation

1.7.1 Objective

Understand how we create virtual environments to run software.

1.7.2 Topics

- **Virtual Machines (VMs):** Emulating an entire computer system.
- **Containers:** OS-level virtualization (e.g., Docker).
- **VMs vs. Containers:** Key differences.

1.7.3 Visualizations & Diagrams

```
graph TD
    subgraph Physical Server
        direction TB
        HW[Hardware] --> HostOS[Host OS];

        subgraph VM Approach
            HostOS --> Hypervisor;
            Hypervisor --> GuestOS_A[Guest OS A];
            Hypervisor --> GuestOS_B[Guest OS B];
            GuestOS_A --> App_A;
            GuestOS_B --> App_B;
        end

        subgraph Container Approach
            HostOS --> ContainerEngine[Container Engine];
            ContainerEngine --> App_C;
            ContainerEngine --> App_D;
        end
    end
```

1.7.4 Real-World Application

- **Virtual Machine:** A developer on a Mac needs to test their website on Internet Explorer. They can run a **Windows VM** on their Mac. This VM contains a full, separate copy of the Windows operating system, allowing them to run IE as if they were on a native Windows PC.
- **Container:** A team builds a microservice that requires Python 3.9 and a specific database library. They package the service and its dependencies into a **Docker container**. Now, any developer can run that container on their machine (Windows, Mac, or Linux) and it will work identically, because the container provides the exact environment the application needs, without needing a whole separate guest OS.

1.7.5 References & Further Reading

- **Video:** Containers vs VMs: What's the Difference?: https://www.youtube.com/watch?v=cjXI-A-4854
- **Article:** "What is a Container?" by Docker: https://www.docker.com/resources/what-container/

2 Networking Essentials for System Design

This document provides a summary of key networking concepts that are essential for system design interviews, based on the provided video transcript.

2.1 The OSI Model and Networking Basics

Networking is often conceptualized as a layered cake, where each layer provides a level of abstraction and builds upon the one below it. For system design interviews, we are primarily concerned with three of these layers:

- **Layer 3: The Network Layer:** This is where protocols like **IP** (Internet Protocol) live. It's responsible for addressing and routing packets across the network.
- **Layer 4: The Transport Layer:** This layer includes protocols like **TCP** and **UDP**, which provide services like guaranteed delivery and ordering.
- **Layer 7: The Application Layer:** This is the top layer, featuring protocols that developers interact with directly, such as **HTTP**, **WebSockets**, and **gRPC**.

These layers work together. For example, an HTTP request (Layer 7) is transmitted over a TCP connection (Layer 4), which in turn uses IP addresses (Layer 3) to route the data. This layering creates overhead and latency, especially during connection setup (e.g., the TCP three-way handshake), which is an important consideration in system design.

2.2 Layer 3: The Network Layer (IP)

The **Internet Protocol (IP)** is responsible for giving usable names (addresses) to nodes on a network and allowing for routing.

- **IPv4 vs. IPv6:**
  - **IPv4:** 4-byte addresses. We have run out of these.
  - **IPv6:** 16-byte addresses. More modern and plentiful.
  - Typically, you'll use IPv4 for external-facing services for compatibility and IPv6 internally.
- **Public vs. Private IP Addresses:**
  - **Public IPs:** Globally unique and routable on the public internet. Assigned by a central authority. Used for your API gateways, load balancers, and other externally facing components.
  - **Private IPs:** Used within a private network (e.g., your home network or a VPC in the cloud). Not routable on the public internet. Used for all your internal microservices and hosts.

2.3 Layer 4: The Transport Layer (TCP, UDP, QUIC)

The transport layer provides important functionality on top of IP.

- **TCP (Transmission Control Protocol):**
  - The **default** choice for most applications.
  - Provides **guaranteed delivery** and **ordering** of packets using sequence numbers and acknowledgements.
  - It's reliable, but this reliability comes at the cost of higher latency and lower throughput, as lost packets must be retransmitted.
- **UDP (User Datagram Protocol):**
  - Offers higher performance (lower latency, higher throughput) by sacrificing the guarantees of TCP.
  - It's a "fire and forget" protocol. Packets can be lost or arrive out of order.

  - **Use cases:** Real-time applications where latency is critical and some data loss is acceptable, such as:
    * Video and audio conferencing
    * Multiplayer gaming
    * Live streaming

- **QUIC:** A more modern transport protocol built on top of UDP that aims to provide the reliability of TCP with lower latency. It's increasingly used for web traffic.

2.4 Layer 7: The Application Layer

This is where most of the application-level logic and protocols reside.

2.4.1 HTTP and REST

- **HTTP (Hypertext Transfer Protocol):** The most popular application layer protocol. It uses a simple text-based request/response model.
  - **Requests** include a method (verb) like `GET`, `POST`, `PUT`, `DELETE`, a URL, and headers.
  - **Responses** include a status code (e.g., `200 OK`, `404 Not Found`), headers, and a body.
  - **Content Negotiation:** Clients and servers use headers to negotiate content types, encodings, etc., which makes HTTP highly flexible and extensible.
- **REST (Representational State Transfer):** The most common architectural style for building APIs on top of HTTP.
  - It's organized around **resources** (identified by URLs) and **verbs** (HTTP methods).
  - Example: To get user 1, you would make a `GET` request to `/users/1`. To create a new user, you would `POST` to `/users`.
  - REST is the default choice for building APIs in system design interviews due to its simplicity, scalability, and wide adoption.

2.4.2 GraphQL

GraphQL is a query language for APIs that provides an alternative to REST.

- **Problem it solves:**
  - **Under-fetching:** When a single API endpoint doesn't provide enough data, requiring the client to make multiple requests.
  - **Over-fetching:** When an endpoint returns more data than the client needs.
- **How it works:** The client sends a single query specifying the exact shape of the data it needs, and the server returns a JSON object matching that shape.
- **Use cases:**
  - When the frontend is changing rapidly or has complex data requirements (e.g., news feeds).
  - When you need to aggregate data from multiple backend services.

2.4.3 gRPC (Google Remote Procedure Call)

gRPC is a high-performance RPC framework.

- **How it works:**
  - Uses **Protocol Buffers (Protobufs)** as its schema definition language and serialization format. Protobufs are a highly efficient binary format.
  - Defines services and messages in `.proto` files, which can be compiled into client and server stubs in many languages.

- **Advantages:**
  - **High performance:** Can be up to 10x faster than REST/JSON due to efficient binary serialization.
  - **Features:** Supports client-side load balancing, streaming, and authentication.
- **Disadvantages:**
  - Not natively supported by web browsers.
  - Binary format is harder to debug than human-readable JSON.
- **Use cases:** Primarily for **internal microservice communication** where performance is critical. A common pattern is to have an external-facing REST API that communicates with internal gRPC services.

2.4.4 Server-Sent Events (SSE)

SSE is a standard that allows a server to push data to a client over a standard HTTP connection.

- **How it works:** The client makes a regular HTTP request, but the server keeps the connection open and sends events as they become available, separated by newlines.
- **Characteristics:**
  - **Unidirectional:** Server-to-client push only.
  - Built on HTTP, so it works with existing infrastructure.
  - Connections can be unreliable and are often terminated by proxies after 30-60 seconds, but clients can automatically reconnect, passing the ID of the last event received.
- **Use cases:**
  - Short-lived updates to a UI (e.g., status of a background job).
  - Streaming AI responses (tokens) back to a user.

2.4.5 WebSockets

WebSockets provide a **full-duplex (bidirectional)** communication channel over a single, long-lived TCP connection.

- **How it works:** It starts with an HTTP "Upgrade" request and then transitions to a raw TCP-like connection for sending binary or text messages.
- **Characteristics:**
  - Low latency, high frequency, bidirectional communication.
  - **Stateful:** Requires managing persistent connections, which adds complexity for deployments and failure handling.
- **Use cases:**
  - Real-time applications requiring two-way communication:
    - Chat applications
    - Multiplayer games
    - Live collaboration tools

2.4.6 WebRTC (Web Real-Time Communication)

WebRTC is a protocol that enables **peer-to-peer (P2P)** communication directly between browsers, primarily for audio and video.

- **How it works:** It's complex, involving signaling servers (to coordinate connections), STUN servers (to traverse NATs), and TURN servers (as a fallback). It runs over UDP.
- **Use cases:**
  - Audio and video calling.
  - Collaborative editors (often using CRDTs - Conflict-free Replicated Data Types).
- **Interview advice:** Avoid bringing it up unless the problem is explicitly about audio/video calling or collaborative editing.

2.5 Load Balancing

When scaling horizontally (adding more servers), you need a way to distribute traffic among them.

- **Client-Side Load Balancing:**
  - The client is aware of all available servers (e.g., from a service registry) and chooses one to connect to directly.
  - **Pros:** No extra hop, lower latency.
  - **Cons:** Clients need to be updated when servers change; less sophisticated balancing algorithms.
  - **Use cases:** Internal microservices (gRPC supports this natively), DNS.
- **Dedicated Load Balancer (Appliance):**
  - A centralized server (hardware or software) that sits between clients and servers.
  - **Layer 4 (Transport Layer):** Operates at the TCP/UDP level. It forwards packets without inspecting their content. Very high performance. Use for stateful connections like WebSockets.
  - **Layer 7 (Application Layer):** Operates at the HTTP level. It can inspect requests and make routing decisions based on URL, headers, etc. More flexible and feature-rich, but slightly lower performance. This is the default choice for most web applications.
  - **Algorithms:** Round Robin, Random, Least Connections.

2.6 Deep Dives

2.6.1 Regionalization and CDNs

- **Problem:** The speed of light imposes a hard limit on latency for global applications (e.g., 80ms between London and New York).
- **Solution:**
  1. **Partition your system:** If possible, partition users and data by region (e.g., Uber riders and drivers are in the same city).
  2. **Collocate data and processing:** Keep your web servers and databases in the same region to minimize back-and-forth latency.
  3. **Replicate data:** For read-heavy workloads, replicate data across regions so reads are fast, but accept that there will be a replication lag.
  4. **Use a CDN (Content Delivery Network):** A CDN is a network of edge servers distributed globally. It caches static content (images, videos, JS, CSS) and serves it from a location close to the user, dramatically reducing latency.

2.6.2 Failures, Timeouts, and Retries

- **Timeouts:** Always set a sensible timeout on any network request to avoid clients waiting forever.
- **Retries:** When a request fails, it's common to retry. However, a naive retry strategy can make a bad situation worse.
- **The Gold Standard: Retries with exponential backoff and jitter.**
  - **Exponential Backoff:** Increase the delay between retries exponentially (e.g., 1s, 2s, 4s, 8s). This gives a struggling service time to recover.
  - **Jitter:** Add a small, random amount of time to each delay. This prevents a "thundering herd" problem where many clients retry at the exact same time.

2.6.3 Cascading Failures and Circuit Breakers

- **Cascading Failure:** A failure in one part of a system (e.g., a slow database) causes failures in upstream services, which in turn cause failures in their upstream services, leading to a system-wide outage. Retries can often exacerbate this.
- **Circuit Breaker Pattern:** A mechanism to prevent cascading failures.
  - **How it works:** An intermediary object monitors for failures. If the failure rate for a downstream service exceeds a threshold, the circuit breaker "trips" or "opens."
  - While the circuit is open, all subsequent calls to the failing service fail immediately without even making a network request. This gives the downstream service a chance to recover.
  - Periodically, the circuit breaker will enter a "half-open" state and allow a single request through. If it succeeds, the circuit closes and normal operation resumes. If it fails, the circuit remains open.

3 API Comparison: REST vs GraphQL

This project demonstrates the practical differences between REST and GraphQL APIs through a complete bookstore example with authors, books, customers, and orders.

3.1 Learning Objectives

- Understand REST API limitations (N+1 queries, over-fetching, multiple requests)
- Experience GraphQL advantages (single queries, precise data fetching, type safety)
- Compare performance and development experience between approaches
- See real-world examples of API design trade-offs

3.2 Project Structure

```
api_comparison/
 shared/                    # Shared components
    models.py              # Data models (Author, Book, Customer, Order)
    database.py            # Mock in-memory database with sample data
 rest_api/                  # Traditional REST API
    app.py                 # Flask server with CRUD endpoints
 graphql_api/               # Modern GraphQL API
    schema.py              # GraphQL schema, types, and resolvers
    app.py                 # FastAPI + Strawberry GraphQL server
 client_examples/           # Demonstration clients
    rest_client.py         # Shows REST API problems
    graphql_client.py      # Shows GraphQL solutions
    compare_apis.py        # Side-by-side comparison
 requirements.txt           # Python dependencies
 start_servers.sh           # Convenient startup script
 README.md                  # This file
```

3.3 API Exploration

3.3.1 REST API (Flask)

- **Base URL**: http://localhost:8000
- **Swagger Docs**: http://localhost:8000/docs
- **Health Check**: http://localhost:8000/health

Example REST Endpoints:

```
GET    /authors                    # List all authors
GET    /authors/1                  # Get specific author
POST   /authors                    # Create author
GET    /books?author_id=1          # Get books by author
GET    /customers/1/orders         # Get customer orders
```

3.3.2 GraphQL API (Strawberry + FastAPI)

- **GraphQL Endpoint**: http://localhost:8001/graphql
- **GraphiQL UI**: http://localhost:8001/graphql (interactive playground)
- **Health Check**: http://localhost:8001/health

Example GraphQL Queries:

```
# Get author with books (single request)
query {
  author(id: 1) {
    name
    email
    books {
      title
      price
      genre
    }
  }
}
```

3.4 Performance Comparison

3.4.1 Problem 1: Multiple Requests

REST: Author + Books = 2 requests

```
GET /authors/1       # Request 1
GET /books?author_id=1   # Request 2
```

GraphQL: Author + Books = 1 request

```
query {
  author(id: 1) {
    name
    books { title price }
  }
}
```

3.4.2 Problem 2: N+1 Query Problem

REST: Customer with order details = 1 + N + M requests

- 1 request for customer
- N requests for each order
- M requests for each book in orders

GraphQL: Customer with order details = 1 request

```
query {
  customer(id: 1) {
    orders {
      items {
        book { title }
      }
    }
  }
}
```

3.4.3 Problem 3: Over-fetching vs Precise Fetching

REST: Always returns all fields

```
{
  "id": 1,
  "title": "Python Programming",
  "isbn": "978-1234567890",
  "price": 29.99,
  "publication_date": "2023-01-15T00:00:00",
```

```
  "author_id": 1,
  "genre": "Programming",
  "description": "A comprehensive guide to Python..."
}
```

GraphQL: Request only needed fields

```
query {
  books {
    title    # Only title
    price    # Only price
  }
}
```