Client-Server Computing, Node.js, and Event-Driven Programming

1. Introduction

This lecture covers foundational topics in distributed computing, focusing on:

- Client-server models
- Introduction to Node.js
- Event-driven programming
- Performance and reliability in networked systems
- Comparative discussion of distributed computing architectures
- Historical context: from circuit switching to packet switching

---

2. Client-Server Architecture

Overview

A client-server model is a fundamental pattern in distributed systems where:

- A **server** hosts resources or services (e.g., databases)
- A **client** requests access to those services

Key Properties

- The application logic is split between client and server
- Runs over a network (often visualized as a line or cloud)
- The server maintains the authoritative state of the system
- The client acts as an interface, with little or no authoritative state

Real-World Example: Course Enrollment

- The server (e.g., registrar's database) stores the true state (who is enrolled)
- The client (e.g., your laptop/browser) only requests changes or queries state
- Only the server can authoritatively say if you are enrolled

Multiple Clients, Server as Mediator

- Many clients can exist simultaneously
- Clients communicate only with the server, not directly with each other
- Example: Two students try to enroll at the same time
    - The server must resolve the conflict (pick a winner, reject both, or—if buggy—accept both)

```
Client1
   \
    \
    [Network] — [Server]
    /
   /
Client2
```

What Can Go Wrong?

- New types of bugs: race conditions, state conflicts, server bottlenecks
- Server is a single point of failure
- Scaling issues as the number of clients grows
- **Network partition:** Clients may be unable to reach the server, causing split-brain or stale state
- **Server crash:** All clients lose access to the service

Failure Scenario Diagram

```
Client1   Client2
   \      /
    [Network]   [Server] (down)
         X
```

Model Comparison Table

Model	Centralized?	Fault Tolerance	Scalability	Consistency	Example Use Case
Client-Server	Yes	Low	Moderate	Strong	Web apps, databases
P2P	No	High	High	Weak/Varies	File sharing, BitTorrent
Master-Worker	Partial	Moderate	High	Strong	ML training, MapReduce

---

3. Event-Driven Programming

Definition

A programming paradigm where code responds to asynchronous events (e.g., web requests, file completions, database responses).

Synchronous vs. Event-Driven

Synchronous Program	Event-Driven Program
Linear execution	Loops + Handlers
Blocking I/O	Non-blocking I/O

The Event Loop

A core concept in environments like Node.js:

```
while (true) {
  event = getNextEvent();
  handleEvent(event);
}
```

Step-by-Step Event Loop (ASCII)

```
[Event Queue] -> [Event Loop] -> [Handler] -> [Done]
     ^                                |
     |________________________________|
```

Real-World Analogy: Restaurant Order System

- The chef (event loop) handles one order (event) at a time, quickly moving to the next as soon as possible. If a dish takes too long, the kitchen (server) gets backed up.

```
[Order Queue] -> [Chef] -> [Serve Dish]
     ^                        |
     |________________________|
```

Constraints of Event Handlers

- Must be **fast** (microseconds to a few milliseconds)
- **Never block** (no synchronous file or network reads/writes)
- Blocking causes delays, queued requests, and unresponsiveness

Bad Example (Blocking):

```
function handleEvent() {
  const result = readBigFileSync(); // BAD: blocks event loop!
}
```

Good Strategy (Non-blocking):

- Split long tasks into short, fast handlers
- Schedule continuation via next event trigger

Example:

```
function doOne(i) {
   doHeavyTask(i);
   scheduleNext(() => doOne(i+1));
}
```

Callback Hell Example

```
readFile('a.txt', (err, dataA) => {
  if (err) throw err;
  readFile('b.txt', (err, dataB) => {
    if (err) throw err;
    readFile('c.txt', (err, dataC) => {
      // ...
    });
  });
});
```

Solution: Promises/Async-Await

```
async function readFiles() {
  const dataA = await readFileAsync('a.txt');
  const dataB = await readFileAsync('b.txt');
  const dataC = await readFileAsync('c.txt');
}
```

Problems in Event-Driven Code

- Higher overhead per action (event dispatch, handler setup)
- Error handling is more complex (no exceptions across events)
- Loss of execution order (events may be handled out of order)
- Debugging and reasoning can be difficult

Multi-threading vs. Event-Driven

Feature	Event-Driven	Multi-threading
Concurrency Model	Single-threaded, non-blocking	Multiple threads, context switching
Shared State	No (single thread = no races)	Yes (needs locks to avoid races)
Performance Potential	Limited CPU parallelism	High parallel performance
Safety	No race conditions	Race conditions possible
Complexity	Moderate	High (synchronization)

Key Benefit of Event-Driven

- No need for locks
- Less prone to race conditions

Limitations

- No true parallelism (one core at a time)
- If an event handler crashes or loops, the whole system halts
- Not suitable for adversarial or untrusted code

Other Languages

- Ruby: EventMachine
- Python: Twisted, asyncio (non-blocking I/O)

Error Handling in Event-Driven Code

- Error handling is more complex than in synchronous code.
- No exception propagation across events: if an event handler fails, you must explicitly record the error state for future handlers to check.
- You cannot rely on try/catch to propagate errors between events.
- Engineering mindset: always ask "what can go wrong?" and plan for error cases.
- In practice, much of robust software engineering is about anticipating and handling errors.

---

4. Web Technologies: HTML, CSS, JavaScript, and the DOM

HTML (HyperText Markup Language)

- Describes the structure and content of web pages (declarative).
- Elements are organized in a tree structure (the DOM: Document Object Model).
- Elements can have attributes (name-value pairs) and can be normal, raw text, or void (self-closing).
- HTML is forgiving: browsers try to render as much as possible even with errors.

DOM Tree Example

```
<html>
  <body>
    <h1>Hello</h1>
    <p>World</p>
  </body>
</html>
```

CSS (Cascading Style Sheets)

- Describes the presentation (style) of HTML elements.
- Styles can be inherited and cascade through the DOM tree.
- Separation of content (HTML) and style (CSS) improves accessibility and maintainability.
- Multiple sources of styles: browser defaults, user preferences, author styles; priority rules determine which wins.

JavaScript

- Imperative programming language for dynamic behavior in web pages.
- Can manipulate the DOM, respond to events, and update content dynamically.
- Can be included inline or as external files.
- Modern web apps often use JavaScript frameworks (e.g., React, which uses JSX as a declarative extension for UI).

Declarative vs. Imperative

- HTML and CSS are declarative: describe what you want, not how to do it.
- JavaScript is imperative: describes how to perform actions.
- Good web design separates content, style, and behavior.

Data Exchange: JSON vs. XML

Feature	JSON	XML
Syntax	Lightweight, concise	Verbose, tag-based
Readability	Easy for humans/machines	Harder for humans
Data Types	Objects, arrays, numbers	Text, attributes, elements
Use Cases	Web APIs, config, storage	Legacy systems, documents
Comments	Not supported	Supported

- JSON (JavaScript Object Notation) is now the most common format for client-server data exchange (tree-structured, easy for JavaScript).
- XML is more verbose and less common in modern web apps.

Security Issues in Web Technologies

- **XSS (Cross-Site Scripting):** Malicious scripts injected into web pages.
- **CORS (Cross-Origin Resource Sharing):** Controls which domains can access resources.
- **CSRF (Cross-Site Request Forgery):** Tricks users into submitting unwanted actions.

---

5. Writing a Basic Node.js Web Server

Example Code

```
const http = require('http');
const ip = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello CS35L World\n');
});

server.listen(port, ip, () => {
  console.log(`Server running at http://${ip}:${port}/`);
});
```

Explanation

Line	Purpose
1	Load the HTTP module
2	Set the local IP address
3	Define the port number
5–9	Set up request handler (callback)
10–12	Start the server and define startup callback

- `require('http')`: Loads the HTTP module (can be named anything, but conventionally matches the module name)
- `ip = '127.0.0.1'`: Localhost IP; server only accessible from the same machine
- `port = 3000`: Port number (no distinction between integers and floats in JS)
- `http.createServer(...)`: Sets up a server with a callback for each request
- Callback parameters: `req` (request object), `res` (response object)
- `res.statusCode = 200`: HTTP status code (200 = OK)
- `res.setHeader('Content-Type', 'text/plain')`: Sets response header
- `res.end('Hello CS35L World\n')`: Ends response, sends data
- `server.listen(...)`: Starts server, with a callback for when it is ready
- Template literals (backticks) allow variable interpolation: `` `Server running at http://${ip}:${port}/` ``

Request Flow Diagram

```
[Client] -> [Request] -> [Node.js Server] -> [Handler] -> [Response] -> [Client]
```

What if the Server Blocks?

- If a synchronous/blocking call is made in the handler, all incoming requests are delayed until it finishes.
- This can make the server unresponsive and is a common pitfall in Node.js.

Demo

- Run: `node app.js`
- Access: `http://localhost:3000`
- Response: "Hello CS35L World"

---

6. Distributed Computing Models

1. Client-Server Model

- Centralized control and data
- Problems: scalability, single point of failure
- Example: Traditional web applications

2. Peer-to-Peer (P2P)

- Each peer has its own data and control

- No centralized server
- Pros: scalable, redundancy
- Cons: complex consistency, data propagation challenges
- Example: BitTorrent, blockchain

3. Primary-Secondary (Master-Worker)

- One primary/master controls computation
- Many secondary/worker nodes perform tasks
- Used in ML training, distributed databases
- Example: MapReduce, parameter servers in ML

Model	Use Case	Pros	Cons	Fault Tolerance	Consistency
Client-Server	Web apps, DBs	Simple	Scalability, SPOF	Low	Strong
P2P	File sharing, crypto	Scalable	Complexity	High	Weak/Varies
Master-Worker	ML, batch jobs	Control	Bottleneck at primary	Moderate	Strong

ASCII Diagrams

Client-Server:

```
Client1   Client2   Client3
   \        |        /
    \       |       /
     [Network]---[Server]
```

Peer-to-Peer:

```
[Peer1]---[Peer2]
   |        |
[Peer3]---[Peer4]
```

Primary-Secondary:

```
         [Primary]
        /   |   \
   [Sec1] [Sec2] [Sec3]
```

---

7. Distributed Systems Challenges

A. Performance Problems

Throughput

- Number of operations per second
- Optimization methods:
  - Handle events out-of-order
  - Prioritize cached contents
  - Balance workload

Latency

- Time between request and response
- Optimization methods:
  - Cache on client (or server)
  - Use faster protocols
  - Reduce payload sizes

Example: Browser Cache

- Caches previously fetched results
- Avoids round trips to server (especially important for distant servers)

B. Correctness Problems

Serialization (Expanded)

- Serialization means the system's behavior can be explained as if all actions happened in some serial order, even if they were actually interleaved or out of order.
- This is not a cure-all: the resulting computation must still be valid (e.g., not exceeding enrollment limits).
- Serialization is a way to justify out-of-order or parallel execution as long as the end result matches some valid serial order.
- If your application is buggy, serialization won't save you; it's a tool for reasoning about correctness, not a fix for logic errors.
- Example: In a bank, if you deposit and then withdraw, the system must not reorder these in a way that causes an overdraft fee if you had enough money.
- For a single client, actions should appear in the order they were issued; for multiple clients, the system can justify the result as if some serial order occurred.

Cache Staleness

- Client caches may become stale
- Solutions:
  1. **Validate cache**: Compare a unique server state ID (e.g., UUID) with the client cache; if mismatched, update
  2. **Tolerate staleness**: If the application can handle slightly outdated data (e.g., video games, weather apps)

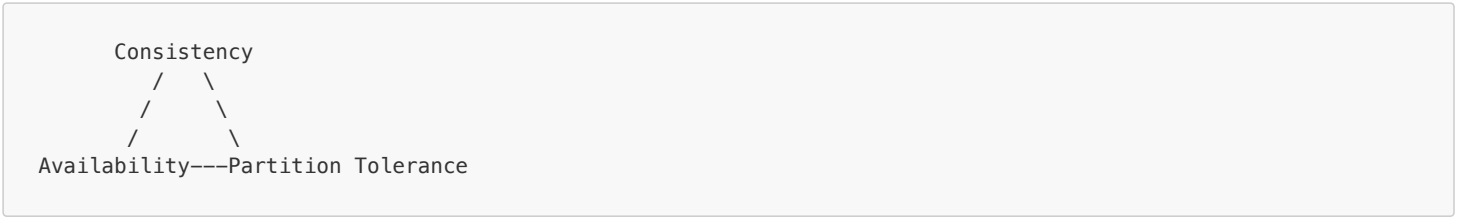
Condition	Strategy
Cache critical	Validate with server
Cache negligible (e.g., news)	Ignore freshness slightly

C. CAP Theorem

- States that a distributed system can only guarantee two out of three: **Consistency**, **Availability**, **Partition Tolerance**.
- **Consistency:** Every read receives the most recent write or an error.
- **Availability:** Every request receives a (non-error) response, without guarantee that it contains the most recent write.
- **Partition Tolerance:** The system continues to operate despite network partitions.

Property	Description
Consistency	All nodes see the same data at the same time
Availability	Every request gets a response (no error)
Partition Tolerance	System works even if network is split

CAP Theorem Diagram



- In practice, systems must choose which two to prioritize during a partition.

D. Consistency Models

- **Strong Consistency:** All clients see the same data at the same time after a write.
- **Eventual Consistency:** All clients will eventually see the same data, but not immediately.
- **Causal Consistency:** Writes that are causally related are seen by all processes in the same order.

Model	Guarantees	Example Use Case
Strong Consistency	Immediate, global	Banking, transactions
Eventual Consistency	Eventually, not instant	DNS, social feeds
Causal Consistency	Order of related events	Collaborative editing

---

8. Deep Dive: Legacy Network Technology

Circuit Switching Model

- Used in traditional (1960s–1980s) telephone systems
- Mechanism: A **dedicated communication path** is established for each call
- Strengths: Guaranteed bandwidth, efficient for long stable conversations, high reliability
- Problems: Inefficient (wastes bandwidth), poor scalability, prone to failure if any intermediate node fails, no encryption (security by trust), motivated by Cold War defense needs

```
Phone A → Central Office → [Switch Network] → Central Office B → Phone B
```

- If any wire or node fails, the whole circuit is lost
- Inefficient: most of the time, the line is silent but still reserved
- Not mobile or portable
- Security: only as strong as the trust in the phone company
- Motivated the search for more robust, scalable, and survivable systems (e.g., in case of nuclear war)

---

9. Packet Switching (Modern Networks)

Overview

- Backbone of the modern Internet
- Data is split into **packets** (typically ~1000 bytes)
- Each packet routes independently via routers
- Routers make independent decisions for each packet
- If a node fails, packets are rerouted
- No need to reserve a full path; more efficient and resilient

```
[UCLA]--[Router]--[Phoenix]--[Chicago]--[Boston]
   \_____________________________/
         (alternate path)
```

Key Terminology

Term	Explanation
Packet	A bundle of data (≤1500 bytes)
Router	Device routing packets
IP	Internet Protocol
Hop	Transition to next router

- Packets may arrive out of order; the receiver must reassemble them
- Routers use heartbeats or other mechanisms to detect failed nodes

Protocol Layers (The Internet Stack)

The modern Internet is built on a layered protocol stack:

```
+---------------------+
| Application Layer   |  (e.g., HTTP, Zoom, Email)
+---------------------+
| Transport Layer     |  (e.g., TCP, UDP, QUIC)
+---------------------+
| Internet Layer      |  (e.g., IP)
+---------------------+
| Link Layer          |  (e.g., Ethernet, Wi-Fi)
+---------------------+
```

- **Link Layer:** Handles direct connections between devices (e.g., Ethernet, Wi-Fi).
- **Internet Layer:** Handles routing of packets across networks (IP addresses, best-effort delivery, packets may be lost, duplicated, or reordered).
- **Transport Layer:** Provides end-to-end communication (TCP: reliable, ordered, error-checked; UDP: fast, unreliable, unordered; QUIC: modern, supports streams, used in HTTP/3).
- **Application Layer:** Protocols for specific applications (web, email, video, etc.).

Motivation for Packet Switching

- Developed for resilience (e.g., Cold War, Paul Baran at RAND): if nodes are destroyed, packets can be rerouted.
- No need to reserve a dedicated path; more efficient and robust than circuit switching.
- Each packet is routed independently; packets may arrive out of order and must be reassembled by the receiver.
- Routers use heartbeats to detect failed nodes and reroute traffic.

Real-World Implications

- TCP provides reliability, ordering, and error checking, but can cause "head-of-line blocking" (all subsequent data waits for a lost packet to be retransmitted).
- UDP is used for applications that can tolerate loss (e.g., video, voice, sensor data).
- HTTP/3 and QUIC are designed to avoid head-of-line blocking and improve performance for real-time applications (e.g., video conferencing in browsers).
- Protocols are layered: each layer builds on the services of the one below.

Circuit vs. Packet Switching Table

Feature	Circuit Switching	Packet Switching
Path	Dedicated, reserved	Dynamic, per-packet
Failure Impact	Call dropped	Packets rerouted
Efficiency	Low (idle time wasted)	High (shared bandwidth)
Scalability	Poor	Excellent
Security	Trust-based	Encryption possible

---

10. Conclusion & Key Takeaways

- Event-driven programming requires a different mindset: focus on performance **and** correctness
- Node.js enables scalable, single-threaded network apps
- Client-server is the dominant model, but alternatives (P2P, master-worker) exist and are important
- Performance tuning (latency, throughput) involves caching, async logic, and distributing load
- Serialization and cache validation are key to correctness
- Circuit switching vs. packet switching: understanding the evolution of network infrastructure
- CAP theorem and consistency models are central to distributed system design

---

11. Security and the Engineering Mindset

- Security is a major concern in distributed systems, but is often omitted in basic discussions. Always consider security implications (e.g., trust, encryption, adversaries).
- Engineering mindset: Always ask "what can go wrong?" and plan for failures, scaling, and adversarial conditions. Design for reliability, correctness, and performance.
- "Problems of success": Plan for scaling up if your system becomes popular.

Common Security Threats

Threat	Description	Mitigation
Man-in-the-middle (MITM)	Attacker intercepts/modifies traffic	Use TLS/SSL encryption
DDoS	Overwhelm server with traffic	Rate limiting, filtering, scaling
XSS	Inject malicious scripts into web pages	Input validation, CSP
CSRF	Trick user into unwanted actions	Tokens, same-site cookies
Data Breach	Unauthorized data access	Access controls, encryption

---

Appendix: HTTP Requests and Status Codes

Code	Meaning
200	OK
404	Not Found
500	Internal Server Error