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

1. Introduction

This lecture covers foundational topics from distributed computing, specifically: - Client-server models - Introduction to Node.js - Event-driven programming - Performance and reliability concerns in networked systems - Comparative discussion of different distributed computing architectures

---

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

Client-Server Communication

- Runs on a network
- Typically modeled visually with clients and servers connected via a line or “cloud”
- The application logic is shared between client and server

Server Responsibilities

- Maintains the authoritative “state of the system”.
- Handles all significant data (e.g., enrollment records at a university).
- Enforces rules of the system (e.g., capacity limits).

Client Responsibilities

- Sends requests to the server.
- Has little or no authoritative state information.
- Acts primarily as an interface or interaction mechanism.

Real-world Example: Course Enrollment

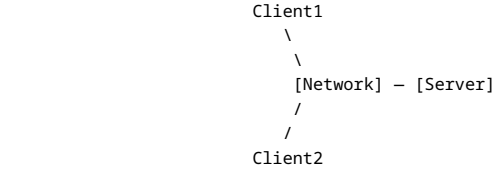
- Students attempt to enroll in a class via a client (browser)
- The server determines whether a student is successfully enrolled
- Only servers possess the true, authoritative state

Multiple Clients, Server Mediator Model

- Many clients can exist simultaneously
- Clients communicate only with the server, not directly with each other
- Race conditions can occur (e.g., two clients attempt to enroll at the same time)

Server resolves such conflicts by: - Picking a winner - Rejecting both (error) - Or mistakenly accepting both (undesired)

Diagram:



---

3. Event-Driven Programming

Definition

A different programming paradigm useful in environments where events happen spontaneously (like web servers).

Asynchronous Events

- Events arrive from the external environment.
- May include:
  - Web requests.
  - Results from external databases.
  - File read completions.

> “Events arrive, and code is run in response.”

Key Concepts

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

The Event Loop

A core concept underpinning environments like Node:

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

Constraints of Event Handlers

- Should be FAST (e.g., microseconds to a few milliseconds)
- Avoid blocking functions (e.g., file.read(), network.read())

Blocking I/O causes: - Delay in serving new requests - Queued inputs - Unresponsive systems

Example of bad event handler:

```
function handleEvent() {
  const result = readBigFileSync(); // This blocks - BAD!
}
```

Proper Strategy

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

Example:

From:

```
for (let i = 0; i < n; ++i) {
   doHeavyTask(i);
}
```

To:

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

Problems in Event-Driven Code

Potential Issues: - Higher overhead per action. - Error handling becomes complex. - Loss of execution order. - Difficult debugging and reasoning.

Multi-threading vs Event-Driven

Feature	Event-Driven	Multi-threading
Concurrency Model	Single-threaded, non-blocking	Multiple threads with 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 due to 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 takes too long, the whole system halts.

---

4. Introduction to Node.js

Overview

- JavaScript runtime for event-driven applications
- Runs JavaScript outside the browser (e.g., on servers)
- Allows asynchronous I/O using callbacks/event loop
- Built on Chrome's V8 JavaScript engine

The Node Runtime

- Operates using the event loop.
- Non-blocking I/O.
- Built on the V8 JavaScript engine.

JavaScript in Node

- Originally a browser language.
- Node allows JavaScript to run on servers (no GUI required).
- Async nature fits web servers well.

  "Node is for web servers that juggle many clients."

---

5. Writing a Basic Node.js Web Server

Source 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.

Special JS Feature:

```
console.log(`Server running at http://${ip}:${port}/`);
```

- Example of a **template literal** (uses backticks ``).
- Allows expression embedding: `${variable}`.

Demo

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

---

6. Distributed Computing Models

1. Client-Server Model

- Explained earlier.
- Centralized control and data.
- Problems:
  - Scalability.
  - Single point of failure.

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.

3. Primary-Secondary (Master-Worker)

- One **primary/master** controls the computation.
- Many **secondary/worker nodes** perform tasks.
- Used in:
  - ML Training.
  - Distributed DBs.

Model	Use Case	Pros	Cons
Client-Server	Small to medium apps	Simple	Scalability
P2P	Decentralized systems	Scalable	Complexity
Master-Worker	Parallel computation	Control	Bottleneck at primary

---

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
  - Use faster protocols
  - Reduce payload sizes

Latency Mitigation: Browser Cache Example

- Caches previously fetched results
- Avoids round trips to server

B. Correctness Problems

Serialization

- Ensure multiple events result in consistent output
- Even if order of execution differs
- Software must be able to “simulate” serial logic

Example:

```
Requests A, B, C → Executed in order C, A, B
But result same as A, B, C → OK (Serializable)
```

Cache Staleness

- Client caches may become stale.
- Solutions:
  1. Validate cache by comparing server state identifier (UUID)
  2. Accept that cache is stale, if app can tolerate (e.g., video games)

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

---

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

Problems: - Inefficient - Poor scalability - Prone to failure if any intermediate node fails - Motivated by Cold War defense needs

Diagram:

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

---

9. Packet Switching (Modern Networks)

Overview

The backbone of the modern Internet.

Mechanism: - Data is split into **packets** (~1000 bytes). - Each packet routes independently via routers.

Advantages: - Resilient: if one node fails, packets find alternative routes. - Efficient: no need to hold a full path open. - Allows for re-routing around network failures

Key Terminology:

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

Routing independence: Each packet can take a different path, allowing flexible and dynamic routing.

Conclusion & Key Takeaways

- Event-driven programming requires different thinking – design has to focus on performance AND correctness
- Node.js makes it easy to write scalable, single-threaded network apps
- Client-server is the dominant model, but alternatives should be considered
- Performance tuning (latency and throughput) involves caching, async logic, or distributing load
- Older models like circuit-switching help understand today’s infrastructure evolution

---

Appendix: Requests and Status Codes Table

Code	Meaning
200	OK
404	Not Found
500	Internal Server Error

Summary

This lecture explores foundational concepts in client-server computing and networking, emphasizing the transition from traditional single-machine applications to distributed systems involving multiple interacting programs. It introduces the client-server model, detailing how clients interact with servers over networks, often relying on the server to store and manage application state, and discusses common pitfalls such as race conditions and request timing conflicts. The discussion includes introductory coverage of technologies like HTML, CSS, JavaScript, and particularly Node.js, highlighting its event-driven, asynchronous programming model that contrasts with synchronous, sequential paradigms taught in introductory programming courses. It delves into the structure and benefits of event-driven programs, including reduced complexity from avoiding multithreading and avoiding resource locking, while also discussing their limitations, including lack of true parallelism and potential vulnerabilities if event handlers are mishandled. The lecture further explores distributed systems challenges such as latency, throughput, and correctness, examining strategies like caching, serialization, and maintaining consistency. Lastly, it contrasts modern packet switching with historic circuit switching models.