Computer Networking Project

HTTP Webserver in JavaScript

Team Members

Namra Patel (221040011014)
Yash Prajapati (221040011026)

1. Project Overview:

This project demonstrates the implementation of a robust HTTP webserver built from the ground up in JavaScript, utilizing only Node.js's native 'net' module as its core dependency. By eschewing traditional web frameworks and libraries in favor of a bare-metal approach, we've crafted a sophisticated server that implements HTTP protocol handling, rate limiting, IP tracking, persistent connections, and file management capabilities entirely from scratch.

2. Key Features:

2.1 Core Functionality:

- HTTP Methods Implementation (GET, POST, DELETE)
- Custom COMM Command for Server Communication
- File System Operations
- Directory Listing Capability
- Connection Statistics Tracking

2.2 Advanced Features:

- IP-based Rate Limiting

- Persistent Server Operation
- Connection Tracking
- Automatic Server Recovery
- Comprehensive Logging System

3. Technical Implementation:

3.1 Server Architecture:

The project is split into three main components
1. server_v2.js: Core server implementation
2. ip-rate-limiter.js: Rate limiting and IP tracking
3. watcher.js: Server persistence and recovery

3.2 Socket Event Handling:

The server utilizes Node.js's event-driven architecture for socket management:

1. **Data Event: Handles incoming data chunks**
    - Buffers partial requests
    - Parses HTTP headers
    - Processes complete requests
2. **End Event: Handles connection termination**
    - Processes any remaining buffered data
    - Cleans up resources
3. **Error Event: Handles connection errors**
    - ECONNRESET handling for abrupt disconnections
    - Error logging and recovery
4. **Timeout Event: Manages idle connections**
    - 120-second timeout implementation
    - Resource cleanup

3.3 Key Components Detail:

Rate Limiter (Ip-rate-limiter.js)
Configuration:
- Maximum Requests: 100 per window
- Window Size: 60 seconds
- Automatic cleanup every hour

Features:
- IP address tracking and validation

- Request counting and window management
- Connection statistics tracking
- Detailed access logging

Main Server file (server_v2.js)
Implemented HTTP methods
1. GET: File retrieval and transfer
2. POST: File upload and storage
3. DELETE: File removal
4. COMM: Custom communication protocol
5. GET_LIST: Directory listing
6. GET_INFO: Connection statistics

Server Watcher (watcher.js)
- Automatic server monitoring
- Crash detection and recovery
- Output logging and error tracking

4. Security Features:

4.1 Rate Limiting:
- Prevents abuse through request limiting
- Configurable time windows and request limits
- IP-based tracking and blocking

4.2 Path Validation:
- Home directory restriction
- Path normalization
- Access control validation

4.3 Error Handling:
- Comprehensive error catching
- Detailed error logging
- Graceful failure handling

5. Implementation Highlights:

5.1 Connection Management:
Socket handling features:
- Used ‘Net’ socket library of JavaScript
- Timeout management (120s)

- Buffer handling for large files
- Connection tracking
- Automatic cleanup

5.2 File Operations:
- Safe file path validation
- Directory creation as needed
- Atomic file operations
- Directory listing capabilities

6. Testing and Performance:

6.1 Rate Limiter Testing:
- Accurate IP tracking and statistics
- Proper cleanup of stale entries

6.2 Server Stability:
- Successful recovery from crashes
- Proper handling of concurrent connections
- Effective memory management

7. Buffer Management and Request Processing:

- The server implements a sophisticated buffer management system that handles incoming data streams with precision and efficiency. At its core, the implementation uses Node.js's Buffer class to manage binary data, ensuring reliable handling of both text-based and binary file transfers. When data arrives, it's accumulated in a pre-allocated buffer using Buffer.concat(), which dynamically grows to accommodate incoming chunks while maintaining memory efficiency. This approach is particularly crucial for handling large POST requests and file uploads, where the complete request body may arrive in multiple TCP segments.

The request parsing mechanism employs a state machine approach, tracking three critical states: initial buffer allocation, header parsing, and body accumulation. For POST requests, the system first identifies the Content-Length header to determine the expected request size, then accumulates data until the complete request body is received. This careful orchestration prevents memory leaks and ensures data integrity, even when handling concurrent requests from multiple clients.

8. Rate Limiting and Security Architecture:

- The rate limiting system employs a sliding window algorithm that provides precise control over request rates while maintaining optimal memory usage. Each client IP address is tracked using a sophisticated data structure that stores timestamps of requests within the current window (60 seconds by default). Rather than using a simple counter, this implementation allows for exact tracking of request timing, ensuring that clients cannot circumvent the rate limit by clustering requests at window boundaries.

- The system maintains two key data structures: rateLimits for tracking request timestamps and connectionTracker for maintaining connection statistics. An automated cleanup process runs hourly to remove stale entries, preventing memory leaks from inactive clients. When a request arrives, the system efficiently filters out timestamps outside the current window and checks the remaining count against the configured limit (100 requests per window). This approach provides both accuracy and performance, with O(1) lookup time for IP addresses and O(n) cleanup time where n is the number of requests in the window.

9. Event-Driven Architecture and Error Handling:

- The server's event-driven architecture leverages Node.js's event emitter pattern to handle various socket states and potential error conditions. Socket connections are managed through a series of event listeners that handle data transmission ('data' event), connection termination ('end' event), error conditions ('error' event), and connection timeouts. This architecture ensures robust operation even under adverse conditions.

- Error handling is implemented using a multi-layered approach. At the socket level, error events are caught and handled appropriately, with special attention paid to ECONNRESET errors that occur during abrupt client disconnections. At the request processing level, try-catch blocks wrap critical operations, ensuring that errors in one request don't affect other concurrent connections. All errors are logged with detailed context information, facilitating debugging and system monitoring.

10. File System Operations and Path Security:

- The file system operations are implemented with a strong focus on security and reliability. All file paths are validated through a comprehensive security check that prevents directory traversal attacks and ensures operations remain within authorized directories. The validatePath function normalizes paths and verifies they are contained within the user's home directory, providing a crucial security boundary. File operations are implemented using Node.js's fs module, with careful attention to error handling and atomic operations. For file writes, the system ensures target directories exist, creating them if necessary, and handles concurrent write operations safely. File reads are streamed to minimize memory usage, particularly important when serving large files. The system also maintains careful tracking of file operations through detailed logging, providing an audit trail of all file system changes.

11. Custom Protocol Implementation:

- The COMM protocol extends the server's capabilities beyond standard HTTP operations, providing a lightweight mechanism for direct server communication. This custom protocol implementation demonstrates the flexibility of the underlying TCP socket handling. When a COMM command is received, the system bypasses the standard HTTP parsing pipeline and processes the command directly. This approach allows for efficient handling of simple text-based commands while maintaining the full capabilities of HTTP when needed.

- The protocol's simplicity—requiring only a command keyword and message—makes it ideal for basic server monitoring and control operations. Error handling for COMM commands is streamlined but robust, with specific error codes and messages for common failure conditions. This implementation showcases how custom protocols can be efficiently integrated into an HTTP server while maintaining clean separation of concerns.

12. Connection Management and Resource Optimization:

- The connection management system implements sophisticated tracking and optimization mechanisms. Each connection is monitored

for activity, with a 120-second timeout automatically closing inactive connections to prevent resource exhaustion. The system maintains detailed statistics about each connection, including first seen timestamp, total request count, and current window request count, providing valuable metrics for monitoring and debugging.

- Resource optimization extends to memory management, with careful attention paid to buffer allocation and deallocation. The system implements periodic cleanup routines that remove stale connection data and unused buffers, preventing memory leaks that could otherwise accumulate over long periods of operation. This proactive approach to resource management ensures stable operation even under heavy load or extended runtime.

13. Future Enhancements:

Potential improvements for future versions:
1. HTTPS support implementation
2. Database integration for logging
3. User authentication system
4. Enhanced monitoring capabilities
5. WebSocket support
6. Clustering for load balancing

14. Conclusion:

This minimalist yet powerful implementation not only serves as a practical demonstration of fundamental networking concepts but also delivers production-grade features including multiple HTTP method support, comprehensive security measures, and stable operation—all while maintaining complete control over the underlying TCP connection handling. The server's architecture proves that complex web server functionality can be achieved with minimal external dependencies, offering deep insights into both HTTP protocol internals and network programming principles

This project successfully implements a feature-rich HTTP webserver with advanced capabilities for rate limiting and persistence. The modular design allows for easy maintenance and future enhancements, while the security features ensure safe operation in production environments.

The implementation demonstrates practical application of networking concepts including

- HTTP protocol implementation
- Network security measures
- Connection management
- File system operations
- Error handling and logging

Source Code: https://github.com/nsanamra/HTTP-Server-in-JS-from-Scratch-