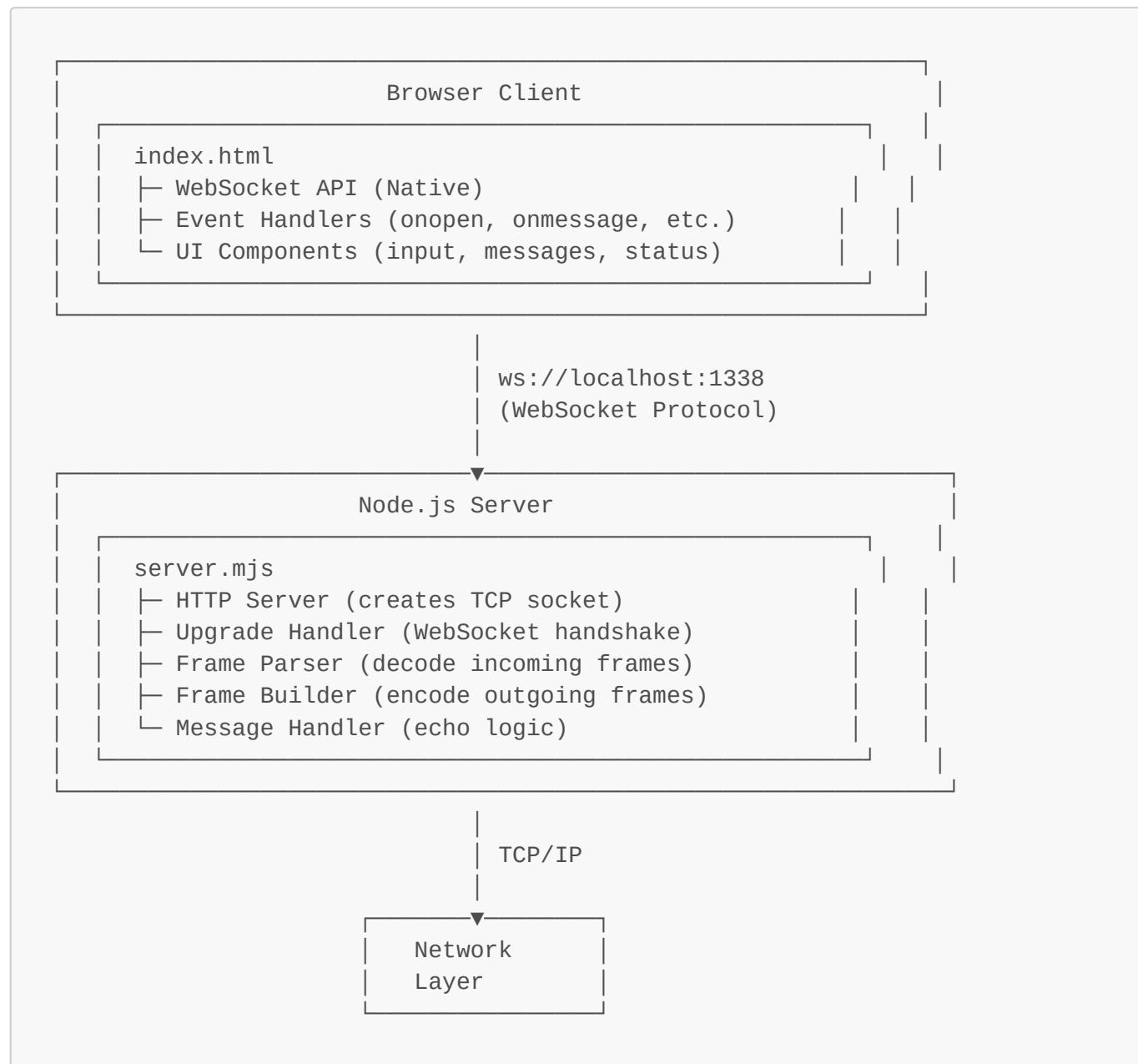


Architecture Documentation

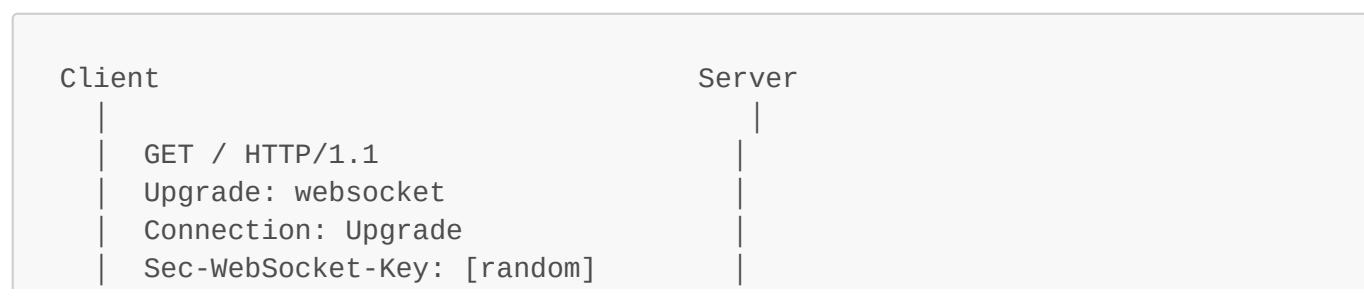
This document provides a deep dive into the WebSocket implementation architecture.

System Architecture



Connection Lifecycle

1. Initial HTTP Request

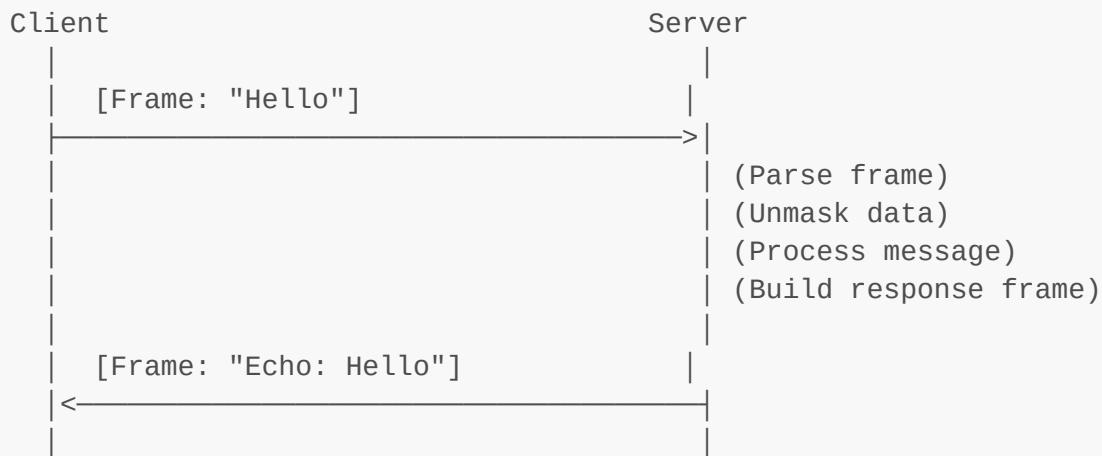




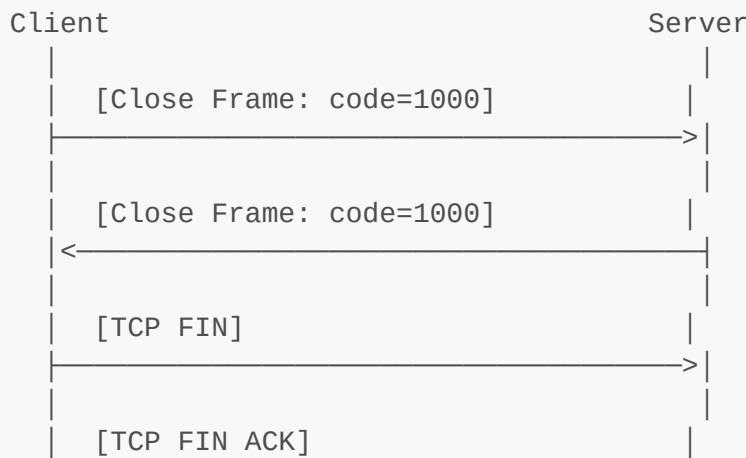
2. Upgrade Response



3. Message Exchange



4. Connection Close





🧩 Component Breakdown

Server Components

1. HTTP Server (`createServer`)

Purpose: Handle initial HTTP requests and serve basic content
Input: HTTP request
Output: HTTP response OR upgrade trigger
State: Listening on PORT **1338**

2. Upgrade Handler (`onSocketUpgrade`)

Purpose: Perform WebSocket handshake
Input: request, socket, head
Output: Handshake response headers
Flow:
1. Extract Sec-WebSocket-Key
2. Compute accept key (SHA-1 + Base64)
3. Send **101** response
4. Attach readable event listener

3. Frame Parser (`onSocketReadable`)

Purpose: Decode incoming WebSocket frames
Input: Raw bytes `from` socket
Output: Decoded message string
Algorithm:
1. Read byte **1**: FIN + Opcode
2. Read byte **2**: MASK + Length indicator
3. Read extended length (`if` needed)
4. Read **4**-byte masking key
5. Read payload
6. XOR unmask payload
7. Convert to **UTF-8**

4. Frame Builder (`sendMessage`)

Purpose: Encode outgoing WebSocket frames
Input: Message string, socket

Output: Raw frame bytes
Algorithm:

1. Convert message to Buffer
2. Determine length encoding
3. Build frame header
4. Write header + payload to socket

5. Accept Key Generator (`createSocketAccept`)

Purpose: Generate handshake accept key
Input: Client's Sec-WebSocket-Key
Output: Base64-encoded SHA-1 hash
Formula: Base64(SHA1(key + MAGIC_STRING))

Client Components

1. WebSocket API Wrapper

Purpose: Manage WebSocket connection
Responsibilities:

- Create connection
- Handle events
- Send messages
- Close connection

2. UI Manager

Purpose: Handle user interactions
Components:

- Message input field
- Send button
- Message log display
- Connection status indicator

3. Event Handlers

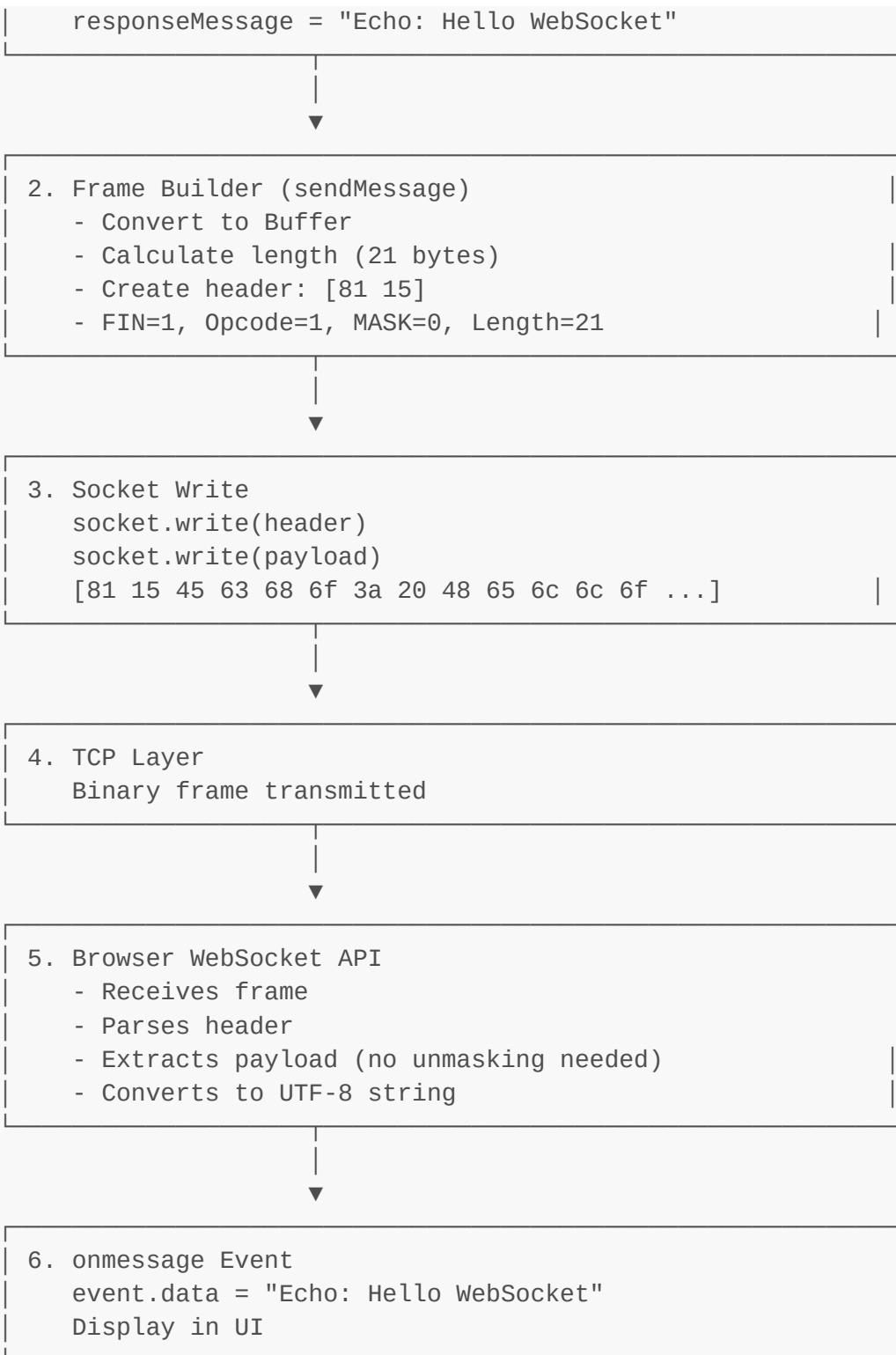
onopen: Connection established
onmessage: Data received
onerror: Error occurred
onclose: Connection closed

Client → Server Message



Server → Client Message





Frame Parsing Algorithm

Detailed Flow

```

function onSocketReadable(socket) {
  // Step 1: Read first byte
  const [firstByte] = socket.read(1);
  // Bit 7: FIN (1 = final fragment)
  // Bits 6-4: RSV1-3 (reserved, must be 0)
}
  
```

```
// Bits 3-0: Opcode (frame type)

const fin = !(firstByte & 0x80); // 10000000
const opcode = firstByte & 0x0f; // 00001111

// Step 2: Read second byte
const [secondByte] = socket.read(1);
// Bit 7: MASK (1 for client→server)
// Bits 6-0: Payload length indicator

const masked = !(secondByte & 0x80); // 10000000
const lengthIndicator = secondByte & 0x7f; // 01111111

// Step 3: Determine actual payload length
let payloadLength;

if (lengthIndicator <= 125) {
    // Length fits in 7 bits
    payloadLength = lengthIndicator;
} else if (lengthIndicator === 126) {
    // Next 2 bytes contain length
    const lengthBuffer = socket.read(2);
    payloadLength = lengthBuffer.readUInt16BE(0);
} else if (lengthIndicator === 127) {
    // Next 8 bytes contain length
    const lengthBuffer = socket.read(8);
    // Read as two 32-bit integers (JavaScript safe)
    const high = lengthBuffer.readUInt32BE(0);
    const low = lengthBuffer.readUInt32BE(4);
    payloadLength = high * 0x10000000 + low;
}

// Step 4: Read masking key (always 4 bytes for client messages)
const maskingKey = socket.read(4);

// Step 5: Read payload
const maskedPayload = socket.read(payloadLength);

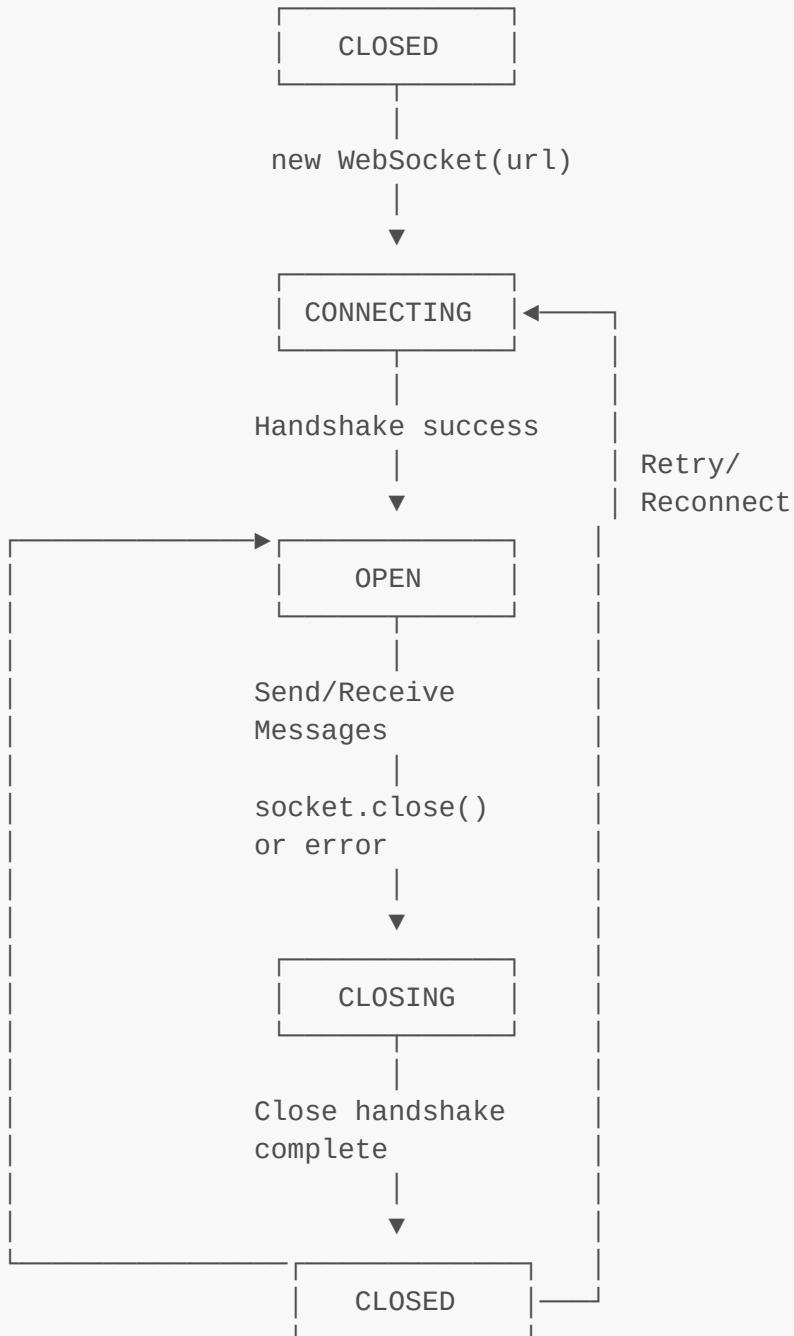
// Step 6: Unmask payload
const payload = Buffer.alloc(payloadLength);
for (let i = 0; i < payloadLength; i++) {
    payload[i] = maskedPayload[i] ^ maskingKey[i % 4];
}

// Step 7: Convert to string (for text frames)
const message = payload.toString("utf8");

// Step 8: Process message based on opcode
if (opcode === 0x1) {
    // Text
    handleTextMessage(message);
} else if (opcode === 0x8) {
    // Close
    handleCloseFrame(payload);
```

```
} else if (opcode === 0x9) {  
    // Ping  
    sendPong(payload);  
}  
}
```

📊 State Diagram



🔒 Security Model

Client-Side Masking

Why? Prevents cache poisoning attacks on intermediary proxies

```

Original payload: "Hello"
Masking key: [0x12, 0x34, 0x56, 0x78]

Masked[0] = 'H' ^ 0x12 = 0x48 ^ 0x12 = 0x5A
Masked[1] = 'e' ^ 0x34 = 0x65 ^ 0x34 = 0x51
Masked[2] = 'l' ^ 0x56 = 0x6C ^ 0x56 = 0x3A
Masked[3] = 'l' ^ 0x78 = 0x6C ^ 0x78 = 0x14
Masked[4] = 'o' ^ 0x12 = 0x6F ^ 0x12 = 0x7D

Result: [0x5A, 0x51, 0x3A, 0x14, 0x7D]

```

Handshake Verification

Purpose: Prove server understands WebSocket protocol

Client sends: dGhIHNhbXBsZSBub25jZQ==

Server computes:

1. Concatenate with GUID
2. SHA-1 hash
3. Base64 encode
4. Send as Sec-WebSocket-Accept

If client receives different value → reject connection

⚡ Performance Considerations

Memory Management

```

// Buffer pooling for large messages
const bufferPool = new Map();

function getBuffer(size) {
  if (bufferPool.has(size)) {
    return bufferPool.get(size);
  }
  const buffer = Buffer.alloc(size);
  bufferPool.set(size, buffer);
  return buffer;
}

```

Frame Batching

```

// Batch multiple small messages into fewer TCP packets
const messageQueue = [];

```

```

function queueMessage(msg) {
  messageQueue.push(msg);
  if (messageQueue.length >= 10) {
    flushQueue();
  }
}

function flushQueue() {
  const batch = messageQueue.splice(0);
  // Send all messages
}

```

Backpressure Handling

```

socket.on("drain", () => {
  console.log("Socket drained, resume writing");
  resumeSending();
});

function sendMessage(msg) {
  const canContinue = socket.write(msg);
  if (!canContinue) {
    pauseSending();
  }
}

```

Extension Points

Adding Binary Support

```

function sendBinary(buffer, socket) {
  const opcode = OPCODES.BINARY; // 0x2
  // Same frame construction with different opcode
}

```

Adding Compression

```

import zlib from "zlib";

function compressAndSend(message, socket) {
  zlib.deflate(message, (err, compressed) => {
    // Set RSV1 bit to indicate compression
    const firstByte = 0xc1; // FIN + RSV1 + TEXT
    sendMessage(compressed, socket, firstByte);
  });
}

```

Adding Authentication

```
function onSocketUpgrade(request, socket, head) {
  const token = request.headers["authorization"];
  if (!validateToken(token)) {
    socket.write("HTTP/1.1 401 Unauthorized\r\n\r\n");
    socket.destroy();
    return;
  }
  // Continue with handshake
}
```

Scalability Patterns

Connection Pooling

```
const connections = new Map();

function onSocketUpgrade(request, socket, head) {
  const id = generateId();
  connections.set(id, socket);

  socket.on("close", () => {
    connections.delete(id);
  });
}
```

Broadcast Pattern

```
function broadcast(message) {
  for (const [id, socket] of connections) {
    sendMessage(message, socket);
  }
}
```

Room Pattern

```
const rooms = new Map();

function joinRoom(socketId, roomName) {
  if (!rooms.has(roomName)) {
    rooms.set(roomName, new Set());
  }
  rooms.get(roomName).add(socketId);
```

```
}

function broadcastToRoom(roomName, message) {
  const room = rooms.get(roomName);
  for (const socketId of room) {
    const socket = connections.get(socketId);
    sendMessage(message, socket);
  }
}
```

🎯 Design Decisions

Why Raw Sockets?

Pros:

- Educational value
- Full control over protocol
- No external dependencies
- Deep understanding of WebSocket

Cons:

- More code to maintain
- Potential for bugs
- Missing advanced features

Why Echo Server?

- Simple to understand
- Easy to test
- Demonstrates bidirectional communication
- Foundation for more complex apps

Why Single File?

- Easy to read and understand
- No module complexity
- Quick to deploy
- Focused on protocol details

This architecture is designed for learning and demonstration. For production use, consider libraries like [ws](#), [socket.io](#), or [uWebSockets.js](#).