

HTTP/2 Server: Design Document

Overview

This system implements a production-grade HTTP/2 server with full multiplexing, binary framing, and header compression capabilities. The key architectural challenge is managing concurrent multiplexed streams over a single TCP connection while implementing complex binary protocol parsing, HPACK compression, and flow control mechanisms.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

Milestone(s): This section provides foundational context for all milestones (1-4) by explaining the historical motivations for HTTP/2 and the architectural challenges we'll solve.

The evolution from HTTP/1.1 to HTTP/2 represents one of the most significant protocol upgrades in web history, fundamentally changing how browsers and servers communicate. Think of HTTP/1.1 as a narrow bridge where cars must cross single-file, with each car waiting for the previous one to completely cross before starting its journey. HTTP/2, in contrast, is like a modern multi-lane highway where multiple vehicles can travel simultaneously, with sophisticated traffic management systems optimizing flow and reducing overall travel time.

Understanding why HTTP/2 exists and what problems it solves is crucial for implementing it correctly. This protocol wasn't created in isolation—it emerged from real-world performance bottlenecks that plagued web applications as they grew more complex and resource-intensive. The transition from text-based HTTP/1.1 to binary HTTP/2 also introduces significant implementation complexity that we must carefully navigate.

HTTP/1.1 Limitations

HTTP/1.1 served the web admirably for over two decades, but its design assumptions became increasingly problematic as web applications evolved from simple document retrieval to complex, resource-heavy applications. The protocol's fundamental architecture creates three critical bottlenecks that severely impact modern web performance.

Head-of-Line Blocking: The Traffic Jam Problem

The most severe limitation of HTTP/1.1 is head-of-line blocking, which occurs because the protocol processes requests sequentially over each TCP connection. Imagine a drive-through restaurant where customers must

place their entire order, wait for it to be completely prepared and delivered, before the next customer can even begin ordering. This serialization creates devastating performance impacts when any single request experiences delays.

Consider a typical web page requiring an HTML document, three CSS files, five JavaScript files, and ten images. In HTTP/1.1, if the second CSS file takes 2 seconds to generate on the server (perhaps due to complex preprocessing), every subsequent request must wait those full 2 seconds before even beginning to process, despite other resources being immediately available. This blocking effect compounds across the entire page load, turning minor delays into major performance degradations.

The mathematical impact is severe: if a page requires 20 resources and the 3rd resource is delayed by D seconds, the total additional delay becomes $(20-2) \times D$ seconds for that connection. This creates unpredictable and often catastrophic loading behavior where a single slow resource can effectively halt the entire page rendering process.

Blocking Scenario	Resources Affected	Performance Impact
Slow database query for one CSS file	All subsequent resources on connection	Linear delay multiplication
Large image processing delay	All resources queued behind it	Complete rendering stall
Third-party API timeout	Entire connection frozen	Cascading failure to other resources
CDN routing issue for one asset	All remaining assets on connection	Unpredictable load times

Connection Overhead: The Infrastructure Tax

HTTP/1.1's head-of-line blocking problem led to a common workaround: opening multiple parallel TCP connections to the same server. Browsers typically open 6-8 connections per domain, attempting to work around the serialization bottleneck. However, this approach introduces its own set of problems that often make performance worse rather than better.

Each TCP connection requires a full three-way handshake (SYN, SYN-ACK, ACK), taking at minimum one full round-trip time (RTT) to establish. For HTTPS connections, an additional TLS handshake adds 1-2 more RTTs depending on TLS version and session resumption. With typical internet latencies of 50-200ms, establishing a single connection can cost 150-600ms before any application data flows. Multiplying this by 6-8 connections creates substantial overhead that often exceeds the time needed to actually transfer the resources.

The connection management overhead extends beyond initial establishment. Each connection consumes memory on both client and server for TCP buffers, connection state, and TLS session information. Servers must maintain separate socket handles, buffer space, and processing threads for each connection. For high-traffic sites, this resource consumption becomes a significant scaling bottleneck, limiting the number of concurrent clients the server can handle effectively.

Connection proliferation also creates network inefficiency through poor congestion control behavior. TCP's congestion control algorithms are designed to fairly share bandwidth between connections, but multiple connections from the same client to the same server subvert this fairness mechanism. Six connections effectively get six times their "fair share" of bandwidth compared to other network users, creating a tragedy-of-the-commons scenario that degrades overall network performance.

Connection Issue	Technical Impact	Resource Cost
TCP handshake overhead	1 RTT per connection minimum	150-600ms delay per connection
TLS handshake overhead	1-2 additional RTTs per connection	300-1200ms additional delay
Memory consumption	~64KB buffers per connection	384KB-512KB per domain
Server file descriptor limits	One FD per connection	Reduced concurrent client capacity
Congestion control gaming	Unfair bandwidth allocation	Network-wide performance degradation

Header Redundancy: The Repetition Problem

HTTP/1.1 sends headers as plain text with every request and response, creating massive redundancy that becomes increasingly problematic as applications use more sophisticated client-side technologies. Modern web applications typically send 20-40 headers per request, including lengthy values like User-Agent strings (often 200+ characters), detailed Accept headers, authentication tokens, and various tracking cookies.

The redundancy problem is particularly severe because most headers remain identical across requests within the same browsing session. The User-Agent header, for example, never changes throughout an entire session but gets transmitted with every single request. Cookie headers often contain session tokens, user preferences, and tracking data that remain static for hours or days but are retransmitted thousands of times. Authorization headers like JWT tokens can be several kilobytes in size but remain unchanged across hundreds of API calls.

This redundancy has a multiplicative effect on bandwidth consumption and latency. A typical modern web application might send 2-4KB of headers per request. For a page requiring 50 resources, this represents 100-200KB of redundant header data—often exceeding the actual content size. On mobile networks with limited bandwidth and high per-byte costs, this redundancy directly translates to slower loading times and increased user costs.

The textual format of HTTP/1.1 headers also prevents any optimization opportunities. Headers must be parsed character by character, creating processing overhead on both client and server. There's no mechanism for header compression, caching of common values, or elimination of redundant information across requests.

Redundancy Issue	Typical Size	Impact per 50-Resource Page
User-Agent header	200-300 bytes	10-15KB redundant data
Cookie headers	500-2000 bytes	25-100KB redundant data
Accept-* headers	300-500 bytes	15-25KB redundant data
Authorization tokens	1000-4000 bytes	50-200KB redundant data
Total header overhead	2-4KB per request	100-200KB redundant transmission

The combination of these three limitations—head-of-line blocking, connection overhead, and header redundancy—creates a perfect storm of performance problems that compound each other. Attempting to solve head-of-line blocking with more connections exacerbates connection overhead, while the additional connections multiply header redundancy across more parallel streams.

HTTP/2 Benefits

HTTP/2 addresses each of HTTP/1.1's fundamental limitations through a comprehensive protocol redesign built around four core innovations: multiplexing, binary framing, header compression, and flow control. Rather than applying incremental patches to HTTP/1.1's text-based design, HTTP/2 represents a ground-up rethinking of how HTTP semantics can be efficiently transported over modern networks.

Multiplexing: The Multi-Lane Highway

HTTP/2's multiplexing capability transforms the single-file bridge of HTTP/1.1 into a sophisticated multi-lane highway with dynamic lane allocation. Think of it as a modern airport terminal where multiple flights can board simultaneously at different gates, with passengers flowing efficiently through shared infrastructure without blocking each other. This analogy captures the essence of HTTP/2's stream-based architecture.

Multiplexing works by introducing the concept of **streams**—independent, bidirectional sequences of frames that carry a single request-response exchange. Multiple streams can operate concurrently over a single TCP connection, with frames from different streams interleaved at the binary framing layer. When stream 3 is waiting for a slow database query, streams 5, 7, and 11 can continue transmitting their responses without any blocking. This eliminates head-of-line blocking at the HTTP layer entirely.

The technical implementation uses stream IDs to multiplex frames over the shared connection. Each frame carries a 31-bit stream identifier that allows the receiving endpoint to reconstruct multiple concurrent conversations from the interleaved frame sequence. Client-initiated streams use odd-numbered IDs (1, 3, 5, ...) while server-initiated streams (for server push) use even-numbered IDs (2, 4, 6, ...). This simple scheme allows for over 2 billion concurrent streams per connection direction.

Priority handling adds sophisticated traffic management to the multiplexing system. Streams can be assigned weights (1-256) and dependencies, creating a priority tree that guides frame transmission order. Critical

resources like CSS files can be marked as high priority, ensuring they're transmitted before less critical resources like analytics tracking pixels. This priority system allows applications to optimize perceived performance by ensuring render-blocking resources arrive first.

Multiplexing Feature	HTTP/1.1 Limitation	HTTP/2 Solution
Concurrent requests	Sequential processing	Unlimited concurrent streams
Head-of-line blocking	One slow request blocks all others	Stream independence
Resource prioritization	No priority mechanism	Weight-based priority tree
Connection utilization	Multiple connections needed	Single connection handles all traffic
Request interleaving	Impossible	Frame-level interleaving

Binary Framing: The Efficient Transport Layer

HTTP/2's binary framing layer represents a fundamental shift from HTTP/1.1's text-based approach to a carefully designed binary protocol optimized for machine processing. Think of this as the difference between sending a handwritten letter (which requires human interpretation) versus a structured database record (which can be processed algorithmically). The binary approach enables significant performance and reliability improvements.

The framing layer introduces a universal frame format with a fixed 9-byte header containing length (24 bits), type (8 bits), flags (8 bits), and stream ID (31 bits). This structured approach eliminates the parsing ambiguities inherent in text-based protocols. HTTP/1.1 parsers must handle numerous edge cases around header folding, whitespace handling, and line ending variations. Binary frames have precisely defined boundaries and unambiguous field meanings.

Frame types provide specialized containers for different kinds of HTTP/2 communication. DATA frames carry request and response payloads, HEADERS frames contain compressed header information, SETTINGS frames negotiate connection parameters, WINDOW_UPDATE frames manage flow control, and PING frames enable connection health monitoring. Each frame type has optimized encoding for its specific purpose, eliminating the one-size-fits-all approach of HTTP/1.1's text format.

The binary format enables powerful optimizations impossible with text protocols. Frames can be processed with simple memory copies and bit operations rather than complex string parsing. Field extraction becomes a matter of reading fixed offsets rather than scanning for delimiter characters. Frame validation can be performed with simple integer comparisons rather than complex regular expressions.

Error handling becomes more robust with binary framing. Malformed frames can be detected immediately through length and type field validation, rather than requiring complex protocol state machines to detect text parsing errors. The binary format also enables precise error reporting—specific fields can be identified as invalid rather than reporting vague "malformed request" errors.

Binary Framing Advantage	Technical Benefit	Performance Impact
Fixed frame header format	Eliminates parsing ambiguity	10-100x faster frame processing
Type-specific frame encoding	Optimized encoding per use case	Reduced bandwidth consumption
Unambiguous field boundaries	No delimiter scanning needed	Constant-time field extraction
Built-in length prefixing	No buffer overrun vulnerabilities	Improved security and reliability
Machine-optimized format	Bit operations vs. string parsing	Lower CPU overhead

Header Compression: The Smart Caching System

HTTP/2's HPACK header compression system solves the redundancy problem through a sophisticated caching and compression scheme that can achieve 85-95% header size reductions in typical scenarios. Think of HPACK as a smart filing system that remembers what documents you've sent before and references them by index number rather than retransmitting the full content. This analogy captures both the efficiency gains and the stateful nature of the compression system.

HPACK operates using two complementary mechanisms: a static table of common header name-value pairs defined in the specification, and a dynamic table that builds up common patterns specific to each connection. The static table contains 61 predefined entries covering the most common headers like `:method: GET`, `:status: 200`, and `content-type: text/html`. These can be referenced by a single byte index rather than transmitting the full header text.

The dynamic table learns connection-specific patterns and builds a cache of frequently used header combinations. When a new header like `authorization: Bearer xyz123token` is first sent, it's transmitted in full but also added to the dynamic table. Subsequent requests can reference this header by its dynamic table index, reducing a potentially multi-kilobyte authorization header to a single byte reference.

Huffman encoding provides additional compression for header values that can't be completely eliminated through table references. HPACK uses a custom Huffman table optimized for typical HTTP header content, achieving significant compression ratios for header values while maintaining fast decode performance. The Huffman encoding is particularly effective for headers containing common English words, URLs, and typical header values.

The compression system includes sophisticated table management to prevent memory exhaustion. The dynamic table has a configurable size limit (typically 4KB) and uses FIFO eviction when the limit is reached. This ensures bounded memory usage while maintaining high hit rates for recently used headers. Connection-specific optimization means that each client-server pair develops its own optimized header vocabulary over time.

HPACK Component	Compression Mechanism	Typical Size Reduction
Static table references	Index lookup for common headers	95-99% reduction for common headers
Dynamic table references	Connection-specific header caching	90-95% reduction for repeated headers
Huffman encoding	Optimized compression for header values	20-40% reduction for textual content
Literal header encoding	Efficient encoding for new headers	10-20% overhead for initial transmission
Combined effect	All mechanisms together	85-95% total header reduction

Server Push: Proactive Resource Delivery

Server push represents HTTP/2's most revolutionary feature, enabling servers to proactively send resources to clients before they're explicitly requested. Imagine a restaurant that starts preparing your usual appetizer as soon as you sit down, based on your ordering history—by the time you're ready for it, it's already available. Server push applies this same proactive approach to web resource delivery.

The push mechanism works through PUSH_PROMISE frames that inform the client about resources the server intends to push. When serving an HTML page, the server can analyze embedded references to CSS, JavaScript, and image files, then immediately begin pushing these resources while the client is still processing the HTML. By the time the client's HTML parser discovers a `<link rel="stylesheet">` tag, the CSS file may already be fully received and cached.

Push promises prevent duplicate requests through a promise caching mechanism. When the server sends a PUSH_PROMISE for `/styles.css`, the client marks this resource as "promised" in its cache. If the HTML parser later encounters a reference to `/styles.css`, the client can immediately use the pushed resource rather than initiating a new request. This eliminates the traditional waterfall loading pattern where resource dependencies are discovered sequentially.

However, server push introduces complexity around cache validation and client state prediction. The server must avoid pushing resources the client already has cached, as this wastes bandwidth and can actually harm performance. Effective push implementations require sophisticated algorithms to predict what resources the client needs and doesn't already possess.

Critical Design Insight: While server push is powerful, it's also the most complex HTTP/2 feature to implement correctly and can easily harm performance if used inappropriately. Many production implementations start with multiplexing and header compression, adding push capabilities only after mastering the foundational features.

Implementation Challenges

Implementing HTTP/2 correctly presents several interconnected technical challenges that are significantly more complex than typical HTTP/1.1 server implementation. The protocol's sophisticated features create a

web of dependencies between binary parsing, state management, concurrent processing, and resource control that must be carefully orchestrated to achieve correct and performant behavior.

Binary Protocol Parsing: The Precision Engineering Challenge

Binary protocol parsing represents a fundamental shift from the forgiving, text-based parsing traditionally used for HTTP/1.1. Think of this as the difference between assembling furniture from IKEA instructions (where small mistakes are recoverable) versus performing precision surgery (where exact technique is critical for success). Binary protocols demand precise handling of every bit and byte, with no room for the flexible interpretation that text protocols traditionally allow.

The HTTP/2 frame format requires careful attention to byte ordering, bit field extraction, and length validation. The 24-bit length field spans three bytes and must be converted from network byte order (big-endian) to host byte order on little-endian systems. The 8-bit flags field contains bit-packed flags whose meaning varies by frame type—the END_STREAM flag in DATA frames has different implications than in HEADERS frames. The 31-bit stream ID field includes a reserved bit that must always be zero, requiring proper masking during extraction.

Frame parsing must handle partial receives and buffer management complexities absent from text protocols. Unlike HTTP/1.1, where line-based parsing can process partial headers gracefully, binary frames must be received completely before processing. This requires implementing buffering state machines that can accumulate frame data across multiple TCP receive operations, while properly handling the transition between frame header and payload regions.

Payload processing varies significantly by frame type, requiring specialized parsing logic for each frame variant. HEADERS frames contain HPACK-compressed header blocks that require stateful decompression. SETTINGS frames contain key-value pairs with specific validation rules—the SETTINGS_INITIAL_WINDOW_SIZE parameter must not exceed $2^{31}-1$, while SETTINGS_MAX_FRAME_SIZE must be between 16,384 and 16,777,215. DATA frames are simpler but must respect flow control constraints.

Error detection and recovery in binary protocols requires more sophisticated approaches than text protocol parsing. Malformed frames can't be "skipped" like malformed HTTP/1.1 headers—a single byte offset error can make the entire connection unusable. This necessitates robust validation at every parsing stage and careful error propagation to prevent cascading failures.

Parsing Challenge	Complexity Factor	Error Consequences
Multi-byte integer extraction	Endianness handling, bit masking	Connection corruption
Frame type dispatch	Type-specific validation logic	Protocol violation
Partial frame buffering	State machine complexity	Memory leak potential
HPACK integration	Stateful decompression coupling	Compression context corruption
Flow control integration	Per-frame window accounting	Resource exhaustion

State Machine Complexity: The Orchestration Challenge

HTTP/2's state management complexity far exceeds HTTP/1.1 due to the interaction between connection-level state, per-stream state machines, and shared resources like HPACK compression contexts. Think of this as conducting a symphony orchestra where each musician (stream) has their own piece of music (state machine), but they all must coordinate timing (flow control) and share limited resources (connection settings) while the conductor (connection handler) ensures overall coherence.

Stream state machines follow the HTTP/2 specification's defined states: idle, reserved (local/remote), open, half-closed (local/remote), and closed. Each state has specific rules about what frames can be sent or received, what transitions are valid, and what actions must be taken. A stream in the "half-closed (remote)" state can send frames but cannot receive them—attempting to process a received DATA frame in this state constitutes a protocol error requiring stream reset.

Connection-level state management coordinates global settings, flow control windows, and HPACK compression contexts that are shared across all streams. The connection maintains a settings registry where parameters like `SETTINGS_HEADER_TABLE_SIZE` affect HPACK operation for all streams, while `SETTINGS_MAX_CONCURRENT_STREAMS` limits how many streams can be active simultaneously. Changes to these settings can have cascading effects across the entire connection.

The interaction between stream and connection state creates complex edge cases that must be handled correctly. When the connection's `SETTINGS_MAX_CONCURRENT_STREAMS` is reduced, existing streams may need to be queued or reset. When a stream transitions to closed state, its flow control window allocation must be returned to the connection-level pool. HPACK table size changes must be synchronized across all active streams using the same compression context.

Concurrency adds another layer of complexity, as multiple goroutines or threads may be operating on different streams simultaneously while sharing connection-level resources. This requires careful synchronization to prevent race conditions in settings updates, HPACK table modifications, and flow control window management. The synchronization must be fine-grained enough to allow high concurrency while preventing the deadlocks that can occur when multiple streams compete for limited resources.

State Management Challenge	Synchronization Requirement	Failure Mode
Per-stream state machines	Stream-local locking	Invalid state transitions
Connection-level settings	Global coordination	Settings race conditions
HPACK table sharing	Compression context locking	Table corruption
Flow control coordination	Window update synchronization	Deadlock or window leaks
Stream lifecycle management	Creation/destruction coordination	Resource leaks

Concurrency Management: The Resource Coordination Challenge

HTTP/2's multiplexing capabilities create unprecedented concurrency management challenges compared to HTTP/1.1's simple request-response model. The protocol enables potentially thousands of concurrent streams over a single connection, each with independent lifecycle, flow control, and priority requirements. Think of this as managing a busy airport where thousands of passengers (streams) are simultaneously moving through shared infrastructure (connection) with different destinations (responses), priorities (business vs. economy), and timing requirements (connecting flights), all while ensuring no resource conflicts or safety violations.

Flow control presents the most complex concurrency challenge, operating simultaneously at connection and stream levels with interdependent constraints. Each stream has its own send window that must be decremented atomically when sending DATA frames, while the connection-level send window must also be decremented for the same frames. WINDOW_UPDATE frames can arrive asynchronously for either stream or connection windows, requiring careful coordination to wake up blocked streams when window space becomes available.

Resource sharing creates potential contention points that must be carefully managed to prevent performance degradation. The HPACK compression context is shared across all streams but requires exclusive access during header processing to prevent table corruption. Connection settings affect all streams but can be updated asynchronously through SETTINGS frames. Priority tree management requires atomic updates to prevent inconsistent stream dependencies.

Backpressure handling becomes critical when the server cannot keep up with incoming stream creation or when flow control windows become exhausted. Unlike HTTP/1.1, where backpressure naturally occurs through TCP's window mechanism, HTTP/2 requires application-level backpressure management. Streams may need to be queued when concurrent stream limits are reached, or data transmission may need to be suspended when flow control windows are exhausted.

The interaction between HTTP/2's flow control and TCP's flow control creates a multi-layered backpressure system that must be coordinated carefully. HTTP/2 flow control can create artificial backpressure even when TCP has available window space, while TCP backpressure can cause HTTP/2 flow control windows to appear available when they're actually blocked at the transport layer.

Architecture Decision Record: Concurrency Model

- **Context:** HTTP/2 multiplexing requires handling hundreds or thousands of concurrent streams over a single TCP connection, with shared resources and complex inter-dependencies.
- **Options Considered:**
 1. Thread-per-stream model with shared connection state
 2. Single-threaded event loop with async/await patterns
 3. Hybrid model with connection-level thread and stream worker pools
- **Decision:** Hybrid model with dedicated connection thread for frame I/O and worker pool for stream processing
- **Rationale:** Connection thread provides serialized access to shared resources (HPACK, settings) while worker pool enables parallel stream processing. This avoids the synchronization complexity of thread-per-stream while providing better CPU utilization than single-threaded approaches.
- **Consequences:** Requires careful work distribution between connection thread and workers, but provides good scalability and manageable synchronization complexity.

Concurrency Challenge	Coordination Requirement	Performance Impact
Flow control window updates	Atomic window arithmetic across streams	Potential deadlocks if mismanaged
HPACK table access	Exclusive access during compression/decompression	Serialization bottleneck
Priority tree management	Atomic dependency updates	Inconsistent stream scheduling
Settings propagation	Connection-wide atomic updates	Temporary inconsistent state
Stream lifecycle events	Creation/destruction coordination	Resource leak potential

Implementation Guidance

This section provides practical guidance for implementing the HTTP/2 server architecture, focusing on Go-based solutions that balance simplicity with production-readiness. The recommendations here establish patterns that will be expanded in subsequent sections covering each component in detail.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Binary Frame Parsing	Manual byte slicing with <code>binary.BigEndian</code>	Code generation from frame specifications
HPACK Implementation	Table-based lookup with manual Huffman	Optimized trie structures with SIMD Huffman
Stream Management	Map with sync.RWMutex	Lock-free concurrent data structures
Flow Control	Simple counter with mutex	Atomic operations with lock-free algorithms
Connection Handler	Standard net.Conn with buffered I/O	Custom connection pooling with zero-copy I/O
Testing Framework	Standard Go testing with table tests	Property-based testing with rapid

B. Recommended File/Module Structure:

The HTTP/2 server should be organized into focused modules that mirror the protocol's architectural layers:

```

http2-server/
├── cmd/
│   └── server/
│       └── main.go
                           ← Server entry point and configuration
└── internal/
    ├── connection/
    │   ├── handler.go
    │   ├── settings.go
    │   └── connection_test.go
                           ← Connection lifecycle management
                           ← SETTINGS frame handling
                           ← Connection integration tests
    ├── frame/
    │   ├── parser.go
    │   ├── types.go
    │   ├── validation.go
    │   └── frame_test.go
                           ← Frame parsing and serialization
                           ← Frame type definitions
                           ← Frame validation logic
                           ← Frame parsing unit tests
    ├── hpack/
    │   ├── compressor.go
    │   ├── tables.go
    │   ├── huffman.go
    │   └── hpack_test.go
                           ← HPACK compression implementation
                           ← Static and dynamic table management
                           ← Huffman coding implementation
                           ← HPACK compression unit tests
    ├── stream/
    │   ├── manager.go
    │   ├── state.go
    │   ├── priority.go
    │   └── stream_test.go
                           ← Stream lifecycle and multiplexing
                           ← Stream state machine implementation
                           ← Stream priority and dependency handling
                           ← Stream management unit tests
    ├── flowcontrol/
    │   ├── window.go
    │   ├── controller.go
    │   └── flowcontrol_test.go
                           ← Flow control window management
                           ← Backpressure handling logic
                           ← Flow control unit tests
    └── protocol/
        ├── constants.go
        ├── errors.go
        └── utils.go
                           ← HTTP/2 protocol constants
                           ← Protocol error definitions
                           ← Common protocol utilities
pkg/
└── http2/
    ├── server.go
    └── handler.go
                           ← Public HTTP/2 server API
                           ← HTTP handler interface
examples/
└── basic/
    └── main.go
                           ← Basic HTTP/2 server example
└── advanced/
    └── main.go
                           ← Advanced features demonstration

```

C. Infrastructure Starter Code:

The following provides complete, working infrastructure that handles the foundational concerns, allowing learners to focus on HTTP/2-specific challenges:

GO

```
// internal/protocol/constants.go - Complete protocol constants

package protocol

// Frame Types

const (

    FrameTypeDATA          = 0x0
    FrameTypeHEADERS       = 0x1
    FrameTypePRIORITY      = 0x2
    FrameTypeRST_STREAM    = 0x3
    FrameTypeSETTINGS       = 0x4
    FrameTypePUSH_PROMISE  = 0x5
    FrameTypePING           = 0x6
    FrameTypeGOAWAY         = 0x7
    FrameTypeWINDOW_UPDATE  = 0x8
    FrameTypeCONTINUATION   = 0x9
)

// Frame Flags

const (
    FlagEndStream  = 0x1
    FlagEndHeaders = 0x4
    FlagPadded     = 0x8
    FlagPriority   = 0x20
)

// Settings Parameters

const (
    SettingsHeaderTableSize      = 0x1
)
```

```
    SettingsEnablePush          = 0x2
    SettingsMaxConcurrentStreams = 0x3
    SettingsInitialWindowSize   = 0x4
    SettingsMaxFrameSize       = 0x5
    SettingsMaxHeaderListSize  = 0x6
)
}

// Protocol Values

const (
    MaxFrameSize      = 16777215 // 2^24 - 1
    DefaultFrameSize = 16384   // 16KB
    ConnectionStreamID = 0
    InitialWindowSize = 65535   // 64KB - 1
)

// internal/protocol/errors.go - Complete error handling

package protocol

import "fmt"

type ErrorCode uint32

const (
    ErrorCodeNoError          ErrorCode = 0x0
    ErrorCodeProtocolError    ErrorCode = 0x1
    ErrorCodeInternalError    ErrorCode = 0x2
    ErrorCodeFlowControlError ErrorCode = 0x3
    ErrorCodeSettingsTimeout  ErrorCode = 0x4
    ErrorCodeStreamClosed     ErrorCode = 0x5
)
```

```
    ErrorCodeFrameSizeError      ErrorCode = 0x6
    ErrorCodeRefusedStream       ErrorCode = 0x7
    ErrorCodeCancel              ErrorCode = 0x8
    ErrorCodeCompressionError   ErrorCode = 0x9
    ErrorCodeConnectError        ErrorCode = 0xa
    ErrorCodeEnhanceYourCalm    ErrorCode = 0xb
    ErrorCodeInadequateSecurity ErrorCode = 0xc
    ErrorCodeHTTP11Required     ErrorCode = 0xd
)
}

type ProtocolError struct {
    Code      ErrorCode
    Message  string
}

func (e *ProtocolError) Error() string {
    return fmt.Sprintf("HTTP/2 protocol error %d: %s", e.Code, e.Message)
}

func NewProtocolError(code ErrorCode, message string) *ProtocolError {
    return &ProtocolError{Code: code, Message: message}
}

// internal/connection/connection.go - Basic connection wrapper

package connection

import (
    "bufio"
    "net"
)
```

```
"sync"

}

type Connection struct {
    conn    net.Conn

    reader *bufio.Reader

    writer *bufio.Writer

    mu      sync.Mutex

    closed bool
}

func NewConnection(conn net.Conn) *Connection {
    return &Connection{
        conn:    conn,
        reader: bufio.NewReader(conn),
        writer: bufio.NewWriter(conn),
    }
}

func (c *Connection) Read(p []byte) (int, error) {
    return c.reader.Read(p)
}

func (c *Connection) Write(p []byte) (int, error) {
    c.mu.Lock()
    defer c.mu.Unlock()

    if c.closed {
        return 0, net.ErrClosed
    }
}
```

```
}

n, err := c.writer.Write(p)

if err != nil {

    return n, err
}

return n, c.writer.Flush()
}

func (c *Connection) Close() error {

    c.mu.Lock()

    defer c.mu.Unlock()

    if c.closed {

        return nil
    }

    c.closed = true

    return c.conn.Close()
}
```

D. Core Logic Skeleton Code:

GO

```
// internal/frame/parser.go - Frame parsing skeleton

package frame

import (
    "encoding/binary"
    "io"
    "http2-server/internal/protocol"
)

// Frame represents a parsed HTTP/2 frame

type Frame struct {
    Length    uint32
    Type      uint8
    Flags     uint8
    StreamID uint32
    Payload   []byte
}

// ParseFrame reads and parses a complete HTTP/2 frame from the reader.

// Returns the parsed frame or an error if parsing fails.

func ParseFrame(r io.Reader) (*Frame, error) {
    // TODO 1: Read exactly 9 bytes for frame header

    // TODO 2: Extract length field from bytes 0-2 (24-bit big-endian)

    // TODO 3: Extract type field from byte 3

    // TODO 4: Extract flags field from byte 4

    // TODO 5: Extract stream ID from bytes 5-8 (31-bit big-endian, mask reserved bit)

    // TODO 6: Validate frame length doesn't exceed MaxFrameSize

    // TODO 7: Read exactly Length bytes for payload
```

```
// TODO 8: Return constructed Frame struct

// Hint: Use binary.BigEndian.Uint32() for multi-byte integers

// Hint: Stream ID reserved bit is 0x80000000, mask with 0x7FFFFFFF

}

// SerializeFrame writes a frame to the writer in HTTP/2 wire format.

// Returns number of bytes written or error.

func SerializeFrame(w io.Writer, f *Frame) (int, error) {

    // TODO 1: Create 9-byte header buffer

    // TODO 2: Encode length field into bytes 0-2 (24-bit big-endian)

    // TODO 3: Encode type field into byte 3

    // TODO 4: Encode flags field into byte 4

    // TODO 5: Encode stream ID into bytes 5-8 (31-bit big-endian)

    // TODO 6: Write header buffer to writer

    // TODO 7: Write payload buffer to writer

    // TODO 8: Return total bytes written

    // Hint: Use binary.BigEndian.PutUint32() then slice off unwanted bytes for 24-bit
length

}

// ValidateFrame checks if a frame conforms to HTTP/2 protocol rules.

// Returns protocol error if validation fails.

func ValidateFrame(f *Frame) error {

    // TODO 1: Check if frame length matches payload length

    // TODO 2: Validate frame length doesn't exceed MaxFrameSize

    // TODO 3: Check reserved bit in stream ID is zero

    // TODO 4: Validate frame type is recognized (0-9)

    // TODO 5: Perform frame-type specific validation
```

```
// TODO 6: Return appropriate protocol error for violations

// Hint: Different frame types have different validation rules

// Hint: Some flags are only valid for specific frame types

}
```

E. Language-Specific Hints:

- **Binary Parsing:** Use `encoding/binary` package for endianness-safe integer extraction.
`binary.BigEndian.Uint32(buf[:4])` handles 32-bit big-endian conversion automatically.
- **Buffered I/O:** Use `bufio.Reader` and `bufio.Writer` for efficient frame I/O. The `ReadFull()` function ensures exact byte counts are read, which is critical for binary protocols.
- **Concurrency:** Use `sync.RWMutex` for stream maps that have many readers but few writers. Use `sync.Mutex` for simpler cases or when write frequency is high.
- **Error Handling:** Create custom error types that implement the `error` interface for protocol-specific errors. This enables type assertions for different error handling strategies.
- **Memory Management:** Use `sync.Pool` for frequently allocated/deallocated buffers like frame headers to reduce GC pressure in high-throughput scenarios.

F. Milestone Checkpoint:

After implementing the foundation described in this section, verify the setup with these checkpoints:

1. **Project Structure Verification:** Run `find . -name "*.go" | head -20` and verify the file organization matches the recommended structure.
2. **Basic Compilation:** Run `go build ./...` to ensure all packages compile without errors. Address any import or syntax issues.
3. **Protocol Constants:** Run `go test -v ./internal/protocol/...` to verify protocol constants and error types are correctly defined.
4. **Infrastructure Code:** Test the connection wrapper with:

```
go run examples/basic/main.go

# Should start server without panicking

# Ctrl+C to stop
```

BASH

5. **Expected Behavior:** The server should start, bind to a port, and accept connections without processing HTTP/2 frames yet. This establishes the foundation for implementing frame parsing in Milestone 1.

Signs of Problems:

- **Import errors:** Usually indicate incorrect module structure or missing go.mod
- **Compilation failures:** Often caused by incorrect type definitions or missing constants
- **Runtime panics on startup:** Typically indicate nil pointer dereferences in connection setup

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
"Package not found" import errors	Incorrect module path or file location	Check go.mod module name matches import paths	Align import paths with module structure
Constants not recognized	Missing protocol constant definitions	Verify constants.go is in correct package	Import protocol package where constants are used
Connection setup panics	Nil reader/writer in connection wrapper	Add nil checks in NewConnection	Initialize bufio readers/writers properly
Build succeeds but tests fail	Missing test dependencies	Run go mod tidy to resolve dependencies	Add missing test imports and dependencies

Goals and Non-Goals

Milestone(s): This section provides the scope and boundaries for all milestones (1-4) by defining what the HTTP/2 server implementation will achieve and explicitly excluding features outside the learning objectives.

Mental Model: Building a Race Car vs. A Production Vehicle

Think of this HTTP/2 server implementation like building a Formula 1 race car for a specific track rather than designing a family sedan for everyday use. Our goal is to demonstrate mastery of HTTP/2's core technical challenges—binary framing, multiplexing, and header compression—rather than creating a production-ready server with all the bells and whistles. Just as a Formula 1 car strips away air conditioning and cup holders to focus on speed and handling, we'll implement the essential HTTP/2 mechanisms while deliberately excluding production concerns like advanced security, monitoring, and performance optimizations.

This focused approach serves the learning objectives better than attempting to build everything. By the end of this project, you'll have a deep understanding of how HTTP/2's revolutionary features work under the hood, but you won't have spent time on ancillary concerns that don't teach the core protocols.

Functional Goals

Our HTTP/2 server implementation targets four primary functional capabilities that demonstrate mastery of the protocol's most important innovations. These goals directly correspond to the milestone structure and ensure comprehensive coverage of HTTP/2's technical foundations.

Decision: Core Protocol Implementation Over Feature Completeness

- **Context:** HTTP/2 specification includes numerous features ranging from essential (binary framing) to optional (server push) to implementation-specific (performance optimizations)
- **Options Considered:**
 - Implement minimal subset (frames only)
 - Implement core features (framing + multiplexing + compression + flow control)
 - Implement full specification including optional features
- **Decision:** Focus on core features that demonstrate the protocol's key innovations
- **Rationale:** Core features represent the fundamental paradigm shifts from HTTP/1.1 and provide the most educational value. Optional features can be learned later once the foundation is solid.
- **Consequences:** Results in a server that can handle real HTTP/2 clients but lacks production deployment features. Provides deep understanding of protocol mechanics without feature creep.

Binary Framing Layer Implementation

The server must implement complete binary framing capabilities as defined in RFC 7540 Section 4. This includes parsing incoming frames from the wire format, validating frame structure and constraints, and serializing outgoing frames back to binary format.

Frame parsing functionality encompasses reading the 9-byte frame header containing length, type, flags, and stream ID fields, along with extracting the variable-length payload based on the length field. The implementation must handle endianness correctly, validate frame length constraints (maximum `MaxFrameSize` of 16,777,215 bytes), and properly interpret reserved bits according to the specification.

Frame Type	Purpose	Required Fields	Validation Rules
FrameTypeDATA	Carries request/response payload	Stream ID, optional padding	Stream ID must be non-zero, payload respects flow control
FrameTypeHEADERS	Contains request/response headers	Stream ID, compressed header block	Stream ID must be non-zero, header block must be HPACK-compressed
FrameTypeSETTINGS	Communicates connection parameters	Settings identifier and value pairs	Stream ID must be zero for connection-level settings
FrameTypePING	Connection keepalive mechanism	8-byte opaque data	Stream ID must be zero, payload exactly 8 bytes
FrameTypeGOAWAY	Graceful connection termination	Last stream ID, error code, debug data	Stream ID must be zero, last stream ID valid
FrameTypeWINDOW_UPDATE	Flow control window adjustments	Window size increment	Non-zero increment, valid stream ID

The frame validation system must enforce protocol compliance including frame size limits, appropriate stream ID usage (zero for connection-level frames, non-zero for stream-specific frames), and flag interpretation based on frame type. Frame serialization reverses this process, converting internal frame representations back to the binary wire format with proper byte ordering and length calculation.

HTTP/2 Multiplexing and Stream Management

The server implements full multiplexing capabilities allowing multiple concurrent request-response streams over a single TCP connection. This eliminates HTTP/1.1's head-of-line blocking problem by interleaving frames from different streams based on availability rather than processing requests sequentially.

Stream management includes implementing the complete HTTP/2 stream state machine with states idle, reserved (local), reserved (remote), open, half-closed (local), half-closed (remote), and closed. State transitions occur based on frame reception and transmission, with strict validation to prevent protocol violations.

Current State	Triggering Event	Next State	Actions Taken
idle	Receive HEADERS frame	open	Create stream, begin header processing
idle	Send PUSH_PROMISE frame	reserved (local)	Reserve stream for server push
open	Send frame with END_STREAM flag	half-closed (local)	Mark local endpoint finished
open	Receive frame with END_STREAM flag	half-closed (remote)	Mark remote endpoint finished
half-closed (local)	Receive frame with END_STREAM flag	closed	Complete stream lifecycle
half-closed (remote)	Send frame with END_STREAM flag	closed	Complete stream lifecycle
Any state	Send/Receive RST_STREAM frame	closed	Immediate stream termination

Stream ID allocation follows HTTP/2 conventions with client-initiated streams using odd numbers (1, 3, 5, ...) and server-initiated streams using even numbers (2, 4, 6, ...). The implementation tracks the highest stream ID seen to prevent stream ID reuse and detect protocol violations.

Concurrent stream limits enforce the `SETTINGS_MAX_CONCURRENT_STREAMS` setting, rejecting new streams when the limit is exceeded and properly handling stream closure to allow new streams. Stream prioritization provides basic weight-based scheduling, though advanced dependency tree features are excluded from our scope.

HPACK Header Compression Engine

The server implements RFC 7541 HPACK compression providing efficient header compression through static table lookups, dynamic table management, and Huffman encoding. This addresses HTTP/1.1's header redundancy problem where identical headers are transmitted repeatedly.

Static table implementation includes the predefined 61-entry table containing common HTTP headers like `:method`, `:path`, `:status`, and frequently used header-value pairs. The static table provides immediate compression benefits for standard HTTP semantics without connection-specific state.

Index	Header Name	Header Value	Usage
1	:authority		Common for all requests
2	:method	GET	Most frequent HTTP method
3	:method	POST	Second most frequent method
4	:path	/	Default path
8	:status	200	Successful response
13	:status	404	Not found error
14	:status	500	Server error

Dynamic table management maintains a connection-specific FIFO table that grows and shrinks based on header frequency and size constraints. The dynamic table uses configurable size limits (typically 4,096 bytes) with automatic eviction of oldest entries when space is needed for new entries.

Header encoding supports three representation types: indexed header field (reference to static or dynamic table), literal header field with incremental indexing (add to dynamic table), and literal header field never indexed (sensitive headers). The implementation must correctly calculate table indices, manage table size updates, and handle integer encoding with 5-bit, 6-bit, and 7-bit prefixes.

Huffman decoding decompresses header values encoded with HTTP/2's predefined Huffman table optimized for HTTP header patterns. The decoder must handle bit-aligned input, padding validation, and error recovery for malformed Huffman sequences.

Flow Control and Backpressure Management

The server implements HTTP/2's credit-based flow control at both connection and individual stream levels. This prevents fast senders from overwhelming slow receivers and provides fairness across multiplexed streams sharing the same connection.

Connection-level flow control tracks a single window size that limits the total bytes of `DATA` frame payloads across all streams on the connection. The initial connection window defaults to `InitialWindowSize` (65,535 bytes) and can be adjusted via `SETTINGS_INITIAL_WINDOW_SIZE`.

Stream-level flow control maintains independent window sizes for each stream, initialized to the connection's initial window size setting. Each stream consumes from both its individual window and the connection's aggregate window when sending `DATA` frames.

Window Type	Scope	Initial Size	Update Mechanism
Connection Window	All streams combined	65,535 bytes	WINDOW_UPDATE frame with stream ID 0
Stream Window	Individual stream	65,535 bytes	WINDOW_UPDATE frame with specific stream ID

Window management processes `WINDOW_UPDATE` frames to increment available send capacity, validates window size calculations to prevent overflow conditions, and implements backpressure by queueing data when send windows are exhausted.

The implementation must handle edge cases including window underflow (sending more than window allows), window overflow (increment causes size to exceed maximum), and deadlock prevention when both endpoints have exhausted windows.

Non-Functional Goals

Beyond the functional protocol implementation, our HTTP/2 server targets specific non-functional characteristics that demonstrate production-readiness concepts while maintaining educational focus.

Decision: Reasonable Performance Over Maximum Optimization

- **Context:** HTTP/2 implementations can range from naive (functional but slow) to highly optimized (complex but fast)
- **Options Considered:**
 - Naive implementation prioritizing simplicity
 - Reasonable performance with standard optimizations
 - High-performance implementation with advanced optimizations
- **Decision:** Target reasonable performance using standard library features and straightforward algorithms
- **Rationale:** Demonstrates awareness of performance concerns without requiring advanced optimization knowledge. Provides foundation for later optimization while keeping code readable.
- **Consequences:** Server will handle moderate load but won't compete with production implementations like nginx or Apache. Code remains understandable for learning purposes.

Performance Characteristics

The server targets handling at least 100 concurrent connections with 10 streams per connection under normal conditions. This demonstrates scalability concepts without requiring advanced performance engineering. Frame processing should achieve at least 10,000 frames per second on modern hardware, showing efficient binary protocol handling.

Memory usage should remain bounded with configurable limits on connection count, stream count per connection, and HPACK dynamic table sizes. The implementation uses connection pooling and reasonable buffer sizes rather than advanced memory management techniques.

Performance Metric	Target Value	Measurement Method
Concurrent Connections	100+	Load testing with multiple clients
Streams per Connection	10+	Multiplexing test with concurrent requests
Frame Processing Rate	10,000+ frames/sec	Synthetic frame generation benchmark
Memory per Connection	<1 MB	Memory profiling during operation
Request Latency	<10ms overhead	Comparison with HTTP/1.1 baseline

Resource Usage and Limits

The implementation incorporates configurable resource limits preventing resource exhaustion attacks while demonstrating defensive programming practices. Connection limits prevent file descriptor exhaustion, stream limits prevent memory consumption attacks, and flow control windows prevent bandwidth abuse.

Buffer management uses reasonable defaults (64KB read buffers, 1MB write buffers) with proper cleanup to prevent memory leaks. Dynamic table size limits prevent HPACK decompression bombs, and frame size limits prevent oversized frame attacks.

Reliability and Error Recovery

Error handling follows HTTP/2 specification guidelines distinguishing between connection errors (requiring `GOAWAY`) and stream errors (requiring `RST_STREAM`). The implementation provides graceful degradation when possible and clear error reporting for debugging.

Connection recovery handles network interruptions, malformed frames, and protocol violations with appropriate error codes. Stream isolation ensures that errors in one stream don't affect other streams on the same connection unless the error is connection-level.

Error Category	Recovery Action	Client Notification
Malformed Frame	Send GOAWAY, close connection	Connection error with specific code
Stream Protocol Violation	Send RST_STREAM for affected stream	Stream error, other streams unaffected
Flow Control Violation	Send GOAWAY or RST_STREAM based on scope	Appropriate error code indicating violation
Compression Error	Send GOAWAY (affects connection state)	Connection error, HPACK state corrupted

Explicit Non-Goals

To maintain focus on core learning objectives, several HTTP/2 features and production concerns are explicitly excluded from our implementation scope. These exclusions prevent scope creep and ensure depth over breadth in understanding protocol fundamentals.

Decision: Educational Depth Over Production Completeness

- **Context:** HTTP/2 specification and production implementations include numerous features beyond core protocol mechanics
- **Options Considered:**
 - Implement subset focusing on core innovations
 - Implement most features for production-like completeness
 - Implement minimal viable HTTP/2 server
- **Decision:** Exclude advanced features that don't contribute to understanding core protocol innovations
- **Rationale:** Advanced features often involve significant complexity that obscures fundamental concepts. Better to master fundamentals first, then add features incrementally.
- **Consequences:** Server won't be production-ready but will provide deep understanding of HTTP/2's key innovations. Clear path for future enhancement.

Server Push Implementation

HTTP/2 server push allows servers to proactively send resources to clients before they're requested, potentially improving page load times. However, server push involves significant complexity including resource prioritization, cache awareness, client preference handling, and push promise management.

Server push implementation would require `PUSH_PROMISE` frame handling, managing server-initiated streams with even-numbered stream IDs, coordinating pushed resources with client requests, and handling push cancellation via `RST_STREAM` frames. The feature also involves application-level decisions about which resources to push and when.

While valuable for production deployments, server push doesn't demonstrate additional protocol fundamentals beyond what we learn from client-initiated streams. The complexity would distract from mastering binary framing, multiplexing, and compression mechanisms.

Advanced Priority and Dependency Features

HTTP/2 includes sophisticated stream prioritization with dependency trees, weight inheritance, and dynamic priority updates. Production implementations often include complex scheduling algorithms, priority propagation mechanisms, and fairness guarantees across priority levels.

Advanced priority features involve dependency tree manipulation, exclusive dependencies, weight-based scheduling with inheritance, and priority frame processing. These mechanisms can significantly impact

performance but require substantial algorithm complexity and careful testing.

Our implementation includes basic weight-based scheduling to demonstrate priority concepts without implementing full dependency tree management. This provides foundation understanding while avoiding algorithmic complexity that doesn't teach additional protocol mechanisms.

HTTP/3 and QUIC Integration

HTTP/3 represents the next evolution of HTTP over QUIC transport, addressing some of HTTP/2's limitations like head-of-line blocking at the TCP level. However, HTTP/3 involves entirely different transport protocols, encryption integration, and connection migration concepts.

HTTP/3 implementation would require QUIC protocol implementation, UDP-based transport, built-in encryption, connection migration, and different multiplexing approaches. While intellectually interesting, this represents a completely separate learning track from HTTP/2 over TCP.

Understanding HTTP/2 thoroughly provides better foundation for later learning HTTP/3 than attempting both simultaneously. The concepts build progressively from HTTP/1.1 to HTTP/2 to HTTP/3.

Production Deployment Features

Production HTTP/2 servers require numerous operational features including comprehensive logging, metrics collection, health monitoring, graceful shutdown, configuration reloading, and integration with load balancers and reverse proxies.

Security hardening involves TLS configuration, certificate management, security header handling, rate limiting, and protection against various attacks. Performance optimization includes connection pooling, buffer tuning, CPU profiling, and memory optimization.

Administrative features like runtime configuration updates, connection draining, debug endpoints, and operational metrics add significant implementation complexity without teaching HTTP/2 protocol concepts.

Excluded Feature Category	Examples	Rationale for Exclusion
Security Hardening	TLS cipher selection, rate limiting, DDoS protection	Focus on protocol mechanics, not security implementation
Operational Monitoring	Metrics collection, health checks, alerting	Infrastructure concerns separate from protocol learning
Performance Optimization	Zero-copy I/O, custom memory allocators, CPU optimization	Advanced engineering beyond protocol understanding
Production Integration	Load balancer integration, service discovery, container orchestration	Deployment concerns unrelated to HTTP/2 specification

Advanced Error Recovery and Edge Cases

Production implementations handle numerous edge cases including network partition recovery, connection migration, complex timeout scenarios, and compatibility with misbehaving clients. These scenarios often involve intricate state management and recovery logic.

Advanced error recovery might include connection backup strategies, automatic retry mechanisms, client capability detection, and fallback to HTTP/1.1. While important for production use, these features don't teach additional HTTP/2 concepts beyond basic error handling.

Our implementation focuses on correct error detection and standard recovery procedures as specified in RFC 7540, providing solid foundation for understanding error semantics without extensive edge case handling.

Pitfall: Scope Creep Through Feature Addition

A common mistake in educational implementations is gradually adding "just one more feature" until the project becomes unwieldy. This typically happens when encountering real-world clients that expect certain features or when discovering interesting extensions to the base protocol.

Why it's problematic: Each additional feature multiplies testing complexity, increases debugging difficulty, and dilutes focus from core learning objectives. Features often have subtle interactions that can mask fundamental bugs.

How to avoid: Maintain a strict feature freeze after initial scope definition. Document feature requests for future implementation phases but resist adding them to the current scope. Focus on making the defined scope work perfectly rather than expanding functionality.

Recognition: You're experiencing scope creep if you find yourself saying "this should be easy to add" or "real clients expect this feature." These are valid observations but belong in separate learning phases.

Implementation Guidance

This section provides practical direction for implementing the goals and maintaining the scope boundaries defined above. The guidance focuses on establishing clear acceptance criteria and preventing common scope creep scenarios.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP/2 Frame Parsing	Manual byte parsing with encoding/binary	Third-party HTTP/2 library
HPACK Compression	Custom implementation following RFC 7541	Existing HPACK library
TLS Integration	Go's crypto/tls with basic configuration	Custom TLS configuration with cipher selection
Concurrency Model	Goroutines per connection with channels	Worker pools with custom scheduling
Testing Framework	Standard library testing + table-driven tests	Property-based testing with fuzzing
Binary Protocol Handling	encoding/binary for frame parsing	Custom bit manipulation utilities

Recommendation: Use simple options for maximum learning value. The goal is understanding HTTP/2 internals, not leveraging existing libraries.

Recommended File Structure

Organize the implementation to reflect the logical component boundaries and make the scope clear through directory structure:

```

http2-server/
├── cmd/
│   └── server/
│       └── main.go           ← Entry point, connection handling
├── internal/
│   ├── frame/
│   │   ├── parser.go        ← Frame parsing and serialization
│   │   ├── types.go         ← Frame type definitions
│   │   └── validator.go     ← Frame validation logic
│   ├── hpack/
│   │   ├── tables.go        ← Static and dynamic table management
│   │   ├── encoder.go       ← Header compression
│   │   ├── decoder.go       ← Header decompression
│   │   └── huffman.go       ← Huffman coding implementation
│   ├── stream/
│   │   ├── manager.go       ← Stream lifecycle management
│   │   ├── state.go          ← Stream state machine
│   │   └── priority.go       ← Basic priority handling
│   ├── connection/
│   │   ├── handler.go        ← Connection-level logic
│   │   ├── settings.go       ← Settings frame handling
│   │   └── flowcontrol.go    ← Flow control windows
│   └── protocol/
│       ├── errors.go         ← Error code definitions
│       └── constants.go      ← HTTP/2 constants
└── test/
    ├── integration/          ← End-to-end testing
    └── testdata/             ← Sample frames and requests
└── docs/
    └── scope.md              ← This goals document

```

This structure makes the scope boundaries clear—each directory corresponds to a specific functional goal, and excluded features don't have corresponding directories.

Scope Validation Infrastructure

Implement clear acceptance criteria checking to validate goal achievement and prevent scope creep:

GO

```
// ScopeValidator ensures implementation stays within defined boundaries

type ScopeValidator struct {

    enabledFeatures map[string]bool

    resourceLimits ResourceLimits

}

// ResourceLimits defines the non-functional goal boundaries

type ResourceLimits struct {

    MaxConnections      int      // 100+ target

    MaxStreamsPerConn   int      // 10+ target

    MaxFrameSize        uint32   // 16777215 max per spec

    DynamicTableSize    uint32   // 4096 default

    ConnectionWindow    uint32   // 65535 default

}

// ValidateFeatureRequest checks if a feature request aligns with goals

func (sv *ScopeValidator) ValidateFeatureRequest(feature string) error {

    // TODO 1: Check if feature is in enabled features list

    // TODO 2: Return clear error message for out-of-scope features

    // TODO 3: Reference the goals document section explaining exclusion

    // Hint: This prevents runtime scope creep

}

// ValidateResourceUsage checks if current usage meets non-functional goals

func (sv *ScopeValidator) ValidateResourceUsage(usage ResourceUsage) error {

    // TODO 1: Compare current usage against limits

    // TODO 2: Return specific errors for limit violations

    // TODO 3: Include current usage values in error messages
```

```

    // Hint: Use this in testing to validate non-functional goals
}

```

Goal-Driven Testing Strategy

Structure tests to directly validate goal achievement rather than implementation details:

Functional Goal	Test Category	Success Criteria
Binary Framing	Unit tests for each frame type	Parse round-trip produces identical frames
HPACK Compression	Compression ratio tests	Achieve >50% compression on typical headers
Stream Management	Concurrent stream tests	Handle 10+ simultaneous streams correctly
Flow Control	Backpressure tests	Properly queue data when windows exhausted

Common Scope Violations

Violation Pattern	How It Appears	Corrective Action
Feature Creep	"Let's add server push since we're already handling streams"	Document as future enhancement, don't implement
Performance Obsession	"We need custom memory allocators for speed"	Stick to reasonable performance targets
Production Pressure	"Real clients need this TLS configuration"	Distinguish learning goals from production requirements
Perfectionism	"We should handle every possible error condition"	Focus on specification-required error handling

Milestone Checkpoint Validation

After each milestone, validate that goals remain on track:

Milestone 1 Checkpoint: Binary framing handles all required frame types without advanced features

- Can parse and serialize DATA, HEADERS, SETTINGS, PING, GOAWAY, WINDOW_UPDATE frames
- Frame validation catches protocol violations
- No advanced frame types like PUSH_PROMISE or PRIORITY

Milestone 2 Checkpoint: HPACK compression works without advanced optimizations

- Static table lookups work correctly for predefined headers
- Dynamic table management handles basic insertion and eviction

- X No advanced Huffman optimizations or compression tuning

Milestone 3 Checkpoint: Stream management supports multiplexing without advanced priority

- ✓ Multiple concurrent streams work correctly
- ✓ Stream state machine handles all required transitions
- X No dependency tree management or complex priority scheduling

Milestone 4 Checkpoint: Flow control prevents overwhelming receivers

- ✓ Connection and stream windows enforce limits correctly
- ✓ Backpressure handling queues data appropriately
- X No advanced scheduling or fairness algorithms

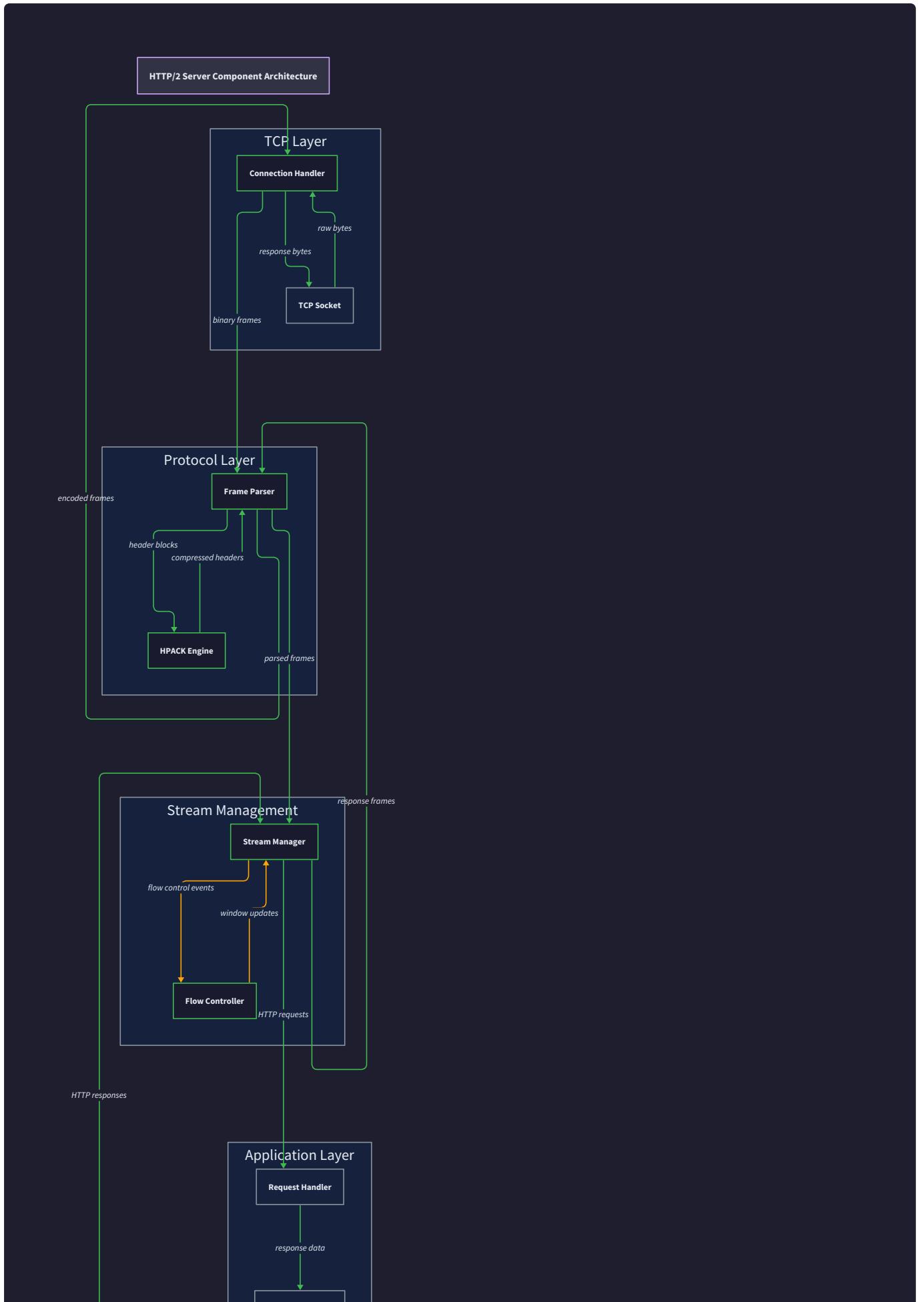
The checkpoint validation ensures implementation stays aligned with learning goals rather than expanding into production features that complicate understanding of core HTTP/2 concepts.

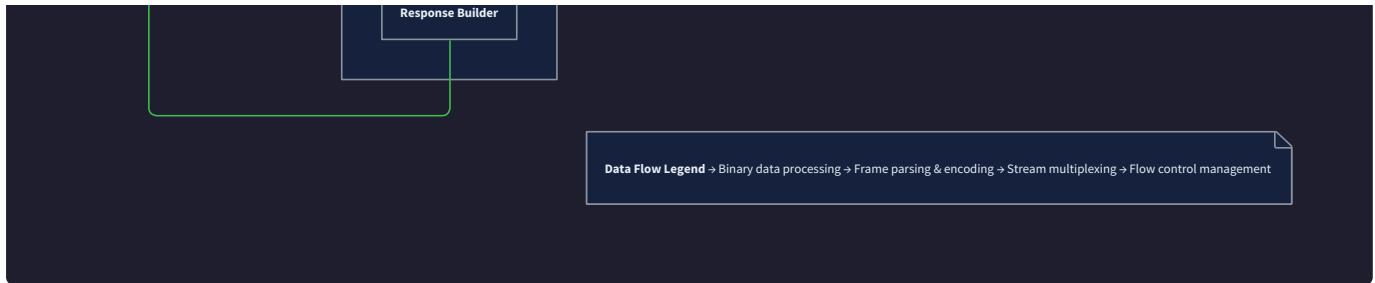
High-Level Architecture

Milestone(s): This section establishes the architectural foundation for all milestones (1-4) by defining the core components, their interactions, and the concurrency model that enables binary framing, HPACK compression, stream management, and flow control.

Think of an HTTP/2 server as a sophisticated air traffic control system. Just as air traffic control manages multiple aircraft simultaneously over shared airspace with complex routing, priority scheduling, and safety protocols, our HTTP/2 server manages multiple concurrent streams over a single TCP connection with binary framing, header compression, and flow control. The connection is like the airspace - a shared resource that must be carefully coordinated. Each stream is like an individual aircraft with its own flight plan, priority level, and fuel constraints (flow control windows). The various server components work together like different control tower systems: radar (frame parser) identifies incoming traffic, flight control (stream manager) tracks each aircraft's state and route, communications (HPACK engine) compresses radio chatter to reduce bandwidth, and traffic control (flow controller) prevents congestion by managing spacing between aircraft.

This mental model helps us understand why HTTP/2's architecture is more complex than HTTP/1.1 - we're not just handling one request at a time sequentially, but orchestrating a complex dance of multiplexed interactions that must be carefully coordinated to prevent conflicts, ensure fairness, and maintain performance.





Component Overview

The HTTP/2 server architecture consists of five primary components that work in concert to handle multiplexed binary protocol communication. Each component has distinct responsibilities and interfaces that enable clean separation of concerns while supporting the complex interactions required by the HTTP/2 specification.

Connection Handler serves as the entry point and orchestrator for all HTTP/2 communication over a single TCP connection. Think of it as the main switchboard operator who receives all incoming calls and routes them to the appropriate departments. This component owns the TCP socket, manages the connection lifecycle, and coordinates between all other components. It maintains connection-level state including settings negotiated during the connection preface, connection-level flow control windows, and the mapping of active streams to their respective handlers.

The Connection Handler implements the HTTP/2 connection state machine, handling the initial connection preface where the client sends the magic string "PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n" followed by a SETTINGS frame. It manages connection-level settings such as maximum frame size, header table size, and maximum concurrent streams. When frames arrive, it performs initial validation and routes them to appropriate components based on frame type and stream ID. For stream-specific frames (DATA, HEADERS), it delegates to the Stream Manager. For connection-level frames (SETTINGS, PING, GOAWAY), it handles them directly or coordinates with other components as needed.

Responsibility	Description	Interfaces With
Connection Lifecycle	Manages TCP connection, connection preface, and graceful shutdown	Frame Parser, Stream Manager
Settings Management	Handles SETTINGS frames and maintains connection parameters	All components for configuration
Frame Routing	Routes incoming frames to appropriate handlers based on type and stream ID	Frame Parser, Stream Manager
Connection Flow Control	Manages connection-level flow control window	Flow Controller
Error Recovery	Handles connection-level errors and sends GOAWAY frames	All components for error propagation

Frame Parser acts as the binary protocol interpreter, converting between the HTTP/2 wire format and structured frame objects that other components can work with. Imagine it as a translator who speaks both binary HTTP/2 and the internal language of our server components. This component encapsulates all the complexity of binary protocol parsing, endianness handling, and frame validation.

The Frame Parser implements the core binary framing logic defined in RFC 7540 Section 4. It reads the 9-byte frame header to extract frame length, type, flags, and stream ID, then reads the appropriate number of payload bytes. It handles frame type-specific parsing for DATA, HEADERS, SETTINGS, PING, GOAWAY, WINDOW_UPDATE, and other frame types. The parser validates frame structure including length limits (frames cannot exceed the negotiated maximum frame size), reserved bit constraints, and frame type-specific validation rules.

Frame Type	Parse Logic	Validation Rules
DATA	Read payload bytes directly	Must have non-zero stream ID, respect flow control
HEADERS	Extract header block fragment	May have padding, END_HEADERS flag handling
SETTINGS	Parse key-value setting pairs	Must have stream ID zero, ACK flag handling
PING	Read 8-byte opaque data	Must have stream ID zero, exactly 8 bytes
GOAWAY	Extract last stream ID and error code	Must have stream ID zero, includes debug data
WINDOW_UPDATE	Extract window size increment	Must be positive, validate against current window

Stream Manager orchestrates the multiplexed stream lifecycle and implements the complex state machines that govern stream behavior. Think of it as a project manager who tracks multiple concurrent projects (streams), each with their own lifecycle, dependencies, and resource requirements. This component maintains the mapping between stream IDs and stream objects, enforces stream state transitions, and implements priority-based scheduling.

The Stream Manager implements the stream state machine defined in RFC 7540 Section 5.1, tracking each stream through states: idle, reserved (local/remote), open, half-closed (local/remote), and closed. It enforces state transition rules, such as preventing DATA frames from being sent on streams in half-closed-local state. The component manages stream ID allocation, ensuring clients use odd-numbered IDs and the server uses even-numbered IDs for server-initiated streams like server push.

Priority handling is one of the Stream Manager's most complex responsibilities. It maintains a dependency tree where streams can depend on other streams with associated weights. When multiple streams have data to send, the Stream Manager uses this priority information to determine scheduling order. The default priority scheme gives equal weight to all streams, but clients can send PRIORITY frames to establish dependencies and adjust weights.

Stream State	Allowed Frame Types	Transition Triggers
Idle	HEADERS, PUSH_PROMISE	Receiving HEADERS → open/half-closed-remote
Open	DATA, HEADERS, WINDOW_UPDATE, RST_STREAM	END_STREAM flag → half-closed
Half-Closed-Local	WINDOW_UPDATE, PRIORITY, RST_STREAM	Receiving END_STREAM → closed
Half-Closed-Remote	DATA, WINDOW_UPDATE, RST_STREAM	Sending END_STREAM → closed
Closed	WINDOW_UPDATE, PRIORITY, RST_STREAM	Stream cleanup after timeout

HPACK Engine implements the sophisticated header compression algorithm defined in RFC 7541. Think of it as a specialized librarian who maintains both a reference collection (static table) and a personalized collection (dynamic table) of commonly used headers, plus a shorthand notation system (Huffman coding) to compress header values. This component significantly reduces bandwidth usage by eliminating redundant header transmission.

The HPACK Engine maintains two tables: a static table with 61 predefined entries for common HTTP headers like `:method GET` and `:status 200`, and a dynamic table that grows during the connection as new headers are encountered. When encoding headers, it first checks if the header name-value pair exists in either table and can reference it by index. If only the name exists, it can use literal representation with incremental indexing to add the new name-value pair to the dynamic table for future reference.

The engine implements Huffman coding for compressing header values using the static Huffman table defined in RFC 7541 Appendix B. This provides additional compression for header values that don't benefit from table-based compression. The combination of table-based indexing and Huffman coding typically achieves 85-95% header compression ratios compared to plain text HTTP/1.1 headers.

Compression Technique	Use Case	Compression Ratio	Processing Cost
Indexed Header Field	Exact name-value match in tables	~95% reduction	Very low
Literal with Incremental Indexing	New header to add to dynamic table	~60% reduction	Medium
Literal Never Indexed	Sensitive headers (cookies, auth)	~40% reduction	Low
Huffman Encoding	String compression for header values	~40% reduction	Medium

Flow Controller manages backpressure and prevents overwhelming receivers with more data than they can process. Think of it as a sophisticated water valve system where each stream has its own valve (stream-level flow control) and there's also a main valve for the entire connection (connection-level flow control). This prevents fast senders from overwhelming slow receivers and ensures fair resource allocation among concurrent streams.

Flow control in HTTP/2 operates on two levels simultaneously. Connection-level flow control limits the total amount of DATA frame payload bytes that can be sent across all streams combined. Stream-level flow control limits DATA frames for individual streams. Both the sender and receiver maintain flow control windows - the sender tracks how much data it's allowed to send, and the receiver tracks how much buffer space it has available.

The Flow Controller tracks these windows and processes WINDOW_UPDATE frames that increase the available send window. When a stream or connection window reaches zero, the Flow Controller queues additional DATA frames until window space becomes available. This queuing mechanism implements backpressure - if the receiver is slow to process data and send WINDOW_UPDATE frames, the sender naturally slows down its transmission rate.

| Window Type | Scope | Initial Size | Update Mechanism | Backpressure Behavior | ---|---|---|---| Connection Window | All streams combined | 65535 bytes | WINDOW_UPDATE with stream ID 0 | Queue data across all streams | | Stream Window | Individual stream | 65535 bytes | WINDOW_UPDATE with specific stream ID | Queue data for that stream only |

Decision: Component Communication Architecture

- **Context:** Components need to coordinate complex interactions like frame processing, stream state updates, and flow control without creating tight coupling or circular dependencies.
- **Options Considered:**
 1. Direct method calls between components
 2. Event-driven architecture with message passing
 3. Layered architecture with defined interfaces
- **Decision:** Layered architecture with defined interfaces and minimal direct coupling
- **Rationale:** Layered architecture provides clear separation of concerns, makes testing easier by allowing component isolation, and follows the natural data flow from binary frames to application-level HTTP semantics. Event-driven architecture would add complexity without clear benefits for this use case.
- **Consequences:** Components communicate through well-defined interfaces, making the system more maintainable and testable, but requiring careful interface design to avoid performance overhead from excessive abstraction.

Module Structure

The HTTP/2 server implementation follows a layered module structure that mirrors the protocol's architectural layers and supports both internal organization and external extensibility. Think of this structure like a building where each floor has a specific purpose: the foundation handles raw TCP connections, the mechanical floor handles binary protocol details, the office floors handle business logic (streams and compression), and the penthouse provides the clean HTTP API to applications.

This organization enables developers to understand and modify individual layers without needing deep knowledge of all layers. A developer working on HPACK compression doesn't need to understand TCP socket management, and someone optimizing stream prioritization doesn't need to know binary frame parsing details.

Package Organization follows Go's standard project layout with clear separation between executable code, internal implementation, and public APIs. The structure supports both the current HTTP/2 implementation and future extensions like server push or HTTP/3 compatibility.

```

http2-server/
├── cmd/
│   └── server/
│       └── main.go
└── internal/
    ├── connection/
    │   ├── handler.go
    │   ├── settings.go
    │   └── handler_test.go
    ├── frame/
    │   ├── parser.go
    │   ├── serializer.go
    │   ├── types.go
    │   └── frame_test.go
    ├── hpack/
    │   ├── encoder.go
    │   ├── decoder.go
    │   ├── tables.go
    │   ├── huffman.go
    │   └── hpack_test.go
    ├── stream/
    │   ├── manager.go
    │   ├── state.go
    │   ├── priority.go
    │   └── stream_test.go
    ├── flow/
    │   ├── controller.go
    │   ├── windows.go
    │   └── flow_test.go
    └── errors/
        ├── protocol.go
        └── recovery.go
└── pkg/
    └── http2/
        ├── server.go
        └── client.go
└── testdata/
    ├── frames/
    └── headers/
└── tools/
    ├── frame-inspector/
    └── hpack-debugger/

```

← Server executable entry point
 ← Internal implementation packages
 ← Connection lifecycle management
 ← SETTINGS frame handling
 ← Connection handler tests
 ← Binary frame parsing (Milestone 1)
 ← Frame serialization
 ← Frame type definitions
 ← Frame parsing tests
 ← Header compression (Milestone 2)
 ← Header decompression
 ← Static and dynamic tables
 ← Huffman coding implementation
 ← HPACK compression tests
 ← Stream lifecycle (Milestone 3)
 ← Stream state machine
 ← Priority and dependency handling
 ← Stream management tests
 ← Flow control (Milestone 4)
 ← Window management
 ← Flow control tests
 ← HTTP/2 protocol errors
 ← Error recovery strategies
 ← Public API packages
 ← Public HTTP/2 server API
 ← Future HTTP/2 client API
 ← Test fixtures and sample data
 ← Binary frame test cases
 ← HPACK test vectors
 ← Development and debugging tools
 ← Binary frame analysis tool
 ← HPACK compression debugger

Layer Dependencies follow a strict hierarchy where higher layers can depend on lower layers, but lower layers never depend on higher layers. This dependency structure prevents circular imports and makes the codebase easier to understand and test.

Layer	Package	Depends On	Responsibility
Application	pkg/http2	internal/connection	Public HTTP/2 server API
Connection	internal/connection	internal/stream , internal/frame	Connection lifecycle and coordination
Protocol Logic	internal/stream , internal/flow	internal/frame , internal/hpack	Stream management and flow control
Protocol Parsing	internal/frame , internal/hpack	internal/errors	Binary protocol and compression
Foundation	internal/errors	Standard library only	Error types and utilities

Interface Boundaries define clean contracts between layers, enabling independent development and testing of components. Each interface represents a specific concern and can be mocked or stubbed during testing.

The `FrameProcessor` interface abstracts binary frame handling from higher-level components. Components that need to parse frames depend on this interface rather than directly importing the frame package. Similarly, the `HeaderCompressor` interface abstracts HPACK compression, allowing the stream manager to compress and decompress headers without understanding HPACK implementation details.

Interface	Methods	Purpose	Implemented By
FrameProcessor	ParseFrame(r io.Reader) (*Frame, error)	Binary frame parsing	internal/frame
	SerializeFrame(w io.Writer, f *Frame) error	Frame serialization	
HeaderCompressor	EncodeHeaders(headers []Header) ([]byte, error)	Header compression	internal/hpack
	DecodeHeaders(data []byte) ([]Header, error)	Header decompression	
StreamManager	CreateStream(id uint32) (*Stream, error)	Stream lifecycle	internal/stream
	GetStream(id uint32) (*Stream, bool)	Stream lookup	
FlowController	ConsumeWindow(id uint32, size uint32) error	Flow control	internal/flow
	UpdateWindow(id uint32, increment uint32) error	Window updates	

Decision: Internal vs Public Package Structure

- **Context:** HTTP/2 server implementation needs to balance internal flexibility with API stability for external users
- **Options Considered:**
 1. Everything in public packages for maximum accessibility
 2. Most logic in internal packages with minimal public API
 3. Hybrid approach with clear public/private boundaries
- **Decision:** Hybrid approach with comprehensive internal packages and focused public API
- **Rationale:** Internal packages allow implementation flexibility and prevent external dependencies on internal details. Public API provides clean interface for server usage while keeping complex protocol details hidden.
- **Consequences:** Easier to maintain backward compatibility and refactor internals, but requires careful public API design to ensure sufficient functionality is exposed.

Threading Model

The HTTP/2 server employs a structured concurrency model that balances performance with correctness while managing the complex interactions between multiplexed streams, binary frame processing, and flow control. Think of this threading model like a restaurant kitchen during dinner rush: there's a host (connection acceptor) seating customers, dedicated line cooks (connection handlers) for each table section, prep cooks (frame processors) handling specialized tasks, and an expediter (stream coordinator) ensuring orders are completed in the right priority order and nothing gets backed up.

This analogy helps explain why we can't simply use one thread per request like HTTP/1.1 - in HTTP/2, multiple requests share a single connection, so we need coordination between requests and careful resource management to prevent one slow request from blocking others.

Connection-Level Concurrency forms the foundation of the threading model. Each TCP connection runs in its own goroutine, providing natural isolation between different clients while enabling full multiplexing within each connection. This design choice eliminates most synchronization overhead between connections while focusing thread safety concerns on the more complex intra-connection coordination.

When a new TCP connection arrives, the server spawns a dedicated Connection Handler goroutine that owns that connection for its entire lifecycle. This goroutine manages the connection preface, settings negotiation, and coordinates all frame processing for that connection. It maintains a continuous read loop that parses incoming frames and routes them to appropriate handlers based on frame type and stream ID.

The Connection Handler implements a hybrid push-pull model for frame processing. Control frames (SETTINGS, PING, GOAWAY) are processed immediately in the connection goroutine to minimize latency for connection-level coordination. Stream-specific frames (DATA, HEADERS) are dispatched to stream-specific goroutines or queued for batch processing depending on current load and stream state.

Concurrency Pattern	Use Case	Synchronization	Resource Sharing
One goroutine per connection	Connection lifecycle, frame routing	Minimal (connection-scoped)	TCP socket, connection state
Goroutine pool for stream processing	DATA and HEADERS frame handling	Channel-based work queues	Stream state, HPACK tables
Dedicated compression goroutine	HPACK encoding/decoding	Mutex on dynamic table	Dynamic table, Huffman decoder
Background maintenance goroutine	Stream cleanup, window updates	Periodic synchronization	Connection-wide state

Stream Processing Architecture handles the complex challenge of processing multiple concurrent streams over a single connection while maintaining proper ordering guarantees and flow control constraints. Each active stream is associated with a lightweight stream processor that maintains stream state and handles stream-specific frame processing.

Stream processors are not full goroutines - creating thousands of goroutines for thousands of potential streams would create excessive overhead. Instead, the system uses a bounded goroutine pool where stream processing work is queued and executed by available worker goroutines. This provides parallelism for independent stream operations while controlling resource usage.

The critical insight is that while HTTP/2 allows frame interleaving, certain operations must maintain ordering guarantees. HEADERS frames that contain the END_HEADERS flag must be processed in order to maintain proper HPACK dynamic table state. DATA frames can be processed out of order as long as flow control windows are updated atomically.

Stream processing implements a two-phase approach: parse phase and execute phase. The parse phase extracts frame data and updates protocol state (stream state machine, flow control windows, HPACK tables). The execute phase performs application-level processing like invoking HTTP handlers. The parse phase requires serialization to maintain protocol correctness, while the execute phase can be fully parallelized across streams.

Stream Processing Phase	Concurrency Level	Synchronization Requirements	Performance Impact
Frame parsing	Sequential per connection	Connection-level mutex	Low (binary parsing is fast)
HPACK decompression	Sequential per connection	Dynamic table mutex	Medium (Huffman decoding)
Stream state updates	Sequential per stream	Stream-level mutex	Low (state machine updates)
Application processing	Parallel across streams	No synchronization needed	High (HTTP handler execution)

Synchronization Strategy minimizes lock contention while ensuring protocol correctness through a combination of lock ordering, fine-grained locking, and lock-free data structures where appropriate. The primary challenge is coordinating access to shared state like HPACK dynamic tables, connection-level flow control windows, and the stream registry.

HPACK dynamic table access requires the most careful synchronization because header compression and decompression operations modify shared table state. The system uses a single reader-writer mutex for each connection's HPACK context, allowing concurrent read operations (header decompression) while serializing write operations (table updates during header processing).

Flow control synchronization operates at two levels: connection-level windows that affect all streams, and stream-level windows that are independent. Connection-level window updates use atomic operations to avoid lock overhead for the common case of window consumption. Stream-level windows use fine-grained locks per stream to enable parallel processing of different streams.

Decision: Goroutine-per-Connection vs Shared Thread Pool

- **Context:** HTTP/2 server needs to handle many concurrent connections efficiently while managing complex per-connection state
- **Options Considered:**
 1. One goroutine per connection with internal stream multiplexing
 2. Shared thread pool processing frames from all connections
 3. Hybrid model with connection goroutines plus worker pools
- **Decision:** One goroutine per connection with internal stream multiplexing
- **Rationale:** Connection-scoped goroutines provide natural isolation for connection state (HPACK tables, settings, flow control windows) without complex synchronization. Go's efficient goroutine scheduling makes this approach scalable to thousands of connections.
- **Consequences:** Simplified synchronization model and easier debugging, but requires careful resource management to prevent goroutine leaks and memory growth with connection count.

Flow Control Threading represents one of the most subtle aspects of the concurrency model because flow control windows must be updated atomically while supporting high-throughput data transfer. The system implements a lock-free approach for the common case of window consumption combined with synchronized updates for window increments.

Each stream maintains its flow control state using atomic integers for current window size and pending data queue length. When DATA frames arrive, the system atomically decrements the appropriate windows (both connection-level and stream-level) and queues the data for processing. If either window would go negative, the frame is queued until window space becomes available.

WINDOW_UPDATE frame processing increments windows and triggers queued data processing. This operation requires coordination between the frame processing goroutine and any goroutines with queued data. The system uses a combination of atomic operations for window updates and channel signaling to wake up blocked senders.

The critical insight is that flow control must never drop or reorder data, even under high concurrency. The system maintains strict ordering guarantees by processing WINDOW_UPDATE frames in the connection goroutine and using atomic operations for all window state changes.

Flow Control Operation	Threading Model	Synchronization	Performance Characteristics
Window consumption (sending DATA)	Per-connection goroutine	Atomic decrement	High throughput, lock-free
Window updates (WINDOW_UPDATE)	Per-connection goroutine	Atomic increment + signaling	Medium latency for blocked senders
Backpressure queue management	Dedicated flow control goroutine	Channel-based coordination	Bounded memory usage
Cross-stream window coordination	Connection goroutine coordination	Connection-level atomic state	Prevents connection-level deadlock

⚠ Pitfall: HPACK Dynamic Table Race Conditions A common mistake is allowing concurrent access to HPACK dynamic tables during header compression and decompression. Since the dynamic table is modified during header processing (entries are added with incremental indexing), concurrent header operations on the same connection can corrupt table state or produce incorrect compression results. Always serialize HPACK operations per connection using a mutex or by processing all headers in the connection goroutine.

⚠ Pitfall: Flow Control Window Underflow Another frequent error is failing to check flow control windows atomically before sending DATA frames. If one goroutine checks that window space is available, but another goroutine consumes that space before the first goroutine sends its data, the window can go negative, violating the HTTP/2 specification. Use atomic compare-and-swap operations or process all DATA frame sending in a single goroutine per connection.

⚠ Pitfall: Stream State Machine Race Conditions Stream state transitions must be atomic with respect to frame processing. If one goroutine is processing a DATA frame while another processes a RST_STREAM frame for the same stream, the stream can end up in an inconsistent state. Either serialize all frame processing for each stream or use fine-grained locking with careful attention to lock ordering to prevent deadlocks.

Implementation Guidance

This subsection provides concrete implementation direction for building the HTTP/2 server architecture described above. The recommendations balance simplicity for learning purposes with architectural soundness for future extensibility.

A. Technology Recommendations

Component	Simple Option	Advanced Option
TCP Server	<code>net.Listen()</code> with goroutine per connection	<code>net/http</code> server with custom handler
Binary Parsing	<code>encoding/binary</code> with manual byte manipulation	Custom binary parser with buffer pooling
Concurrency	Standard goroutines with sync package	Worker pools with bounded queues
Logging	<code>log</code> package with structured output	<code>s log</code> or third-party structured logging
Testing	<code>testing</code> package with table-driven tests	Property-based testing with fuzzing
Debugging	Print statements and manual inspection	<code>pprof</code> profiling and race detector

B. Recommended File/Module Structure

The implementation follows the module structure described above, with each milestone focusing on specific packages:

```

http2-server/
├── cmd/server/
│   └── main.go           ← Entry point: TCP server setup and connection handling
├── internal/
│   ├── connection/
│   │   ├── handler.go      ← Connection Handler: manages TCP connection lifecycle
│   │   ├── settings.go     ← Settings negotiation and SETTINGS frame handling
│   │   └── preface.go       ← Connection preface validation
│   ├── frame/
│   │   ├── parser.go        ← Milestone 1: Binary Framing
│   │   ├── serializer.go    ← ParseFrame() implementation
│   │   ├── types.go          ← SerializeFrame() implementation
│   │   └── validation.go     ← Frame struct and constants
│   ├── hpack/
│   │   ├── encoder.go        ← ValidateFrame() implementation
│   │   ├── decoder.go        ← Milestone 2: HPACK Compression
│   │   ├── tables.go          ← Header compression logic
│   │   └── huffman.go        ← Header decompression logic
│   ├── stream/
│   │   ├── manager.go        ← Static and dynamic table management
│   │   ├── state.go          ← Huffman coding implementation
│   │   └── priority.go        ← Milestone 3: Stream Management
│   ├── flow/
│   │   ├── controller.go     ← Stream Manager component
│   │   └── windows.go        ← Stream state machine
│   └── errors/
│       └── protocol.go      ← Priority and dependency handling
└── pkg/http2/
    └── server.go           ← Milestone 4: Flow Control
                            ← Flow Controller component
                            ← Window management and WINDOW_UPDATE handling
                            ← ProtocolError type and error codes
                            ← Public API: HTTP/2 server interface

```

C. Infrastructure Starter Code (Connection Management)

GO

```
// cmd/server/main.go - Complete TCP server setup

package main

import (
    "log"
    "net"
    "http2-server/internal/connection"
)

func main() {
    listener, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatal("Failed to listen:", err)
    }
    defer listener.Close()

    log.Println("HTTP/2 server listening on :8080")

    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Println("Failed to accept connection:", err)
            continue
        }

        // Each connection gets its own goroutine - this implements
        // the "one goroutine per connection" threading model
        go connection.HandleConnection(conn)
    }
}
```

}

}

GO

```
// internal/connection/handler.go - Connection lifecycle management

package connection

import (
    "bufio"
    "net"
    "sync"
    "http2-server/internal/errors"
)

// Connection represents an HTTP/2 connection with its associated state

type Connection struct {

    conn    net.Conn

    reader *bufio.Reader

    writer *bufio.Writer

    mu      sync.Mutex

    closed bool

    // Connection-level state

    settings map[uint16]uint32 // SETTINGS frame values

    streams  map[uint32]*Stream // Active streams by ID

    // Flow control state

    connectionWindow uint32

    // HPACK state

    hpackEncoder *HPackEncoder

    hpackDecoder *HPackDecoder
```

```
}

// NewConnection creates a new HTTP/2 connection wrapper

func NewConnection(conn net.Conn) *Connection {
    return &Connection{
        conn:          conn,
        reader:        bufio.NewReader(conn),
        writer:        bufio.NewWriter(conn),
        settings:     make(map[uint16]uint32),
        streams:      make(map[uint32]*Stream),
        connectionWindow: InitialWindowSize,
    }
}

// HandleConnection processes an HTTP/2 connection - this is the main entry point

func HandleConnection(conn net.Conn) {
    defer conn.Close()

    h2conn := NewConnection(conn)

    // TODO: Validate connection preface (magic string + SETTINGS)
    // TODO: Enter main frame processing loop
    // TODO: Handle connection errors and cleanup

    h2conn.processFrames()
}

func (c *Connection) processFrames() {
```

```
// TODO: Continuous frame reading loop  
  
// TODO: Route frames based on type and stream ID  
  
// TODO: Handle connection-level vs stream-level frames  
  
// TODO: Implement graceful shutdown on GOAWAY  
  
}
```

GO

```
// internal/errors/protocol.go - Complete error handling infrastructure

package errors

import "fmt"

// ErrorCode represents HTTP/2 error codes as defined in RFC 7540 Section 7

type ErrorCode uint32

// Standard HTTP/2 error codes

const (
    ErrorCodeNoError           ErrorCode = 0x0
    ErrorCodeProtocolError     ErrorCode = 0x1
    ErrorCodeInternalError     ErrorCode = 0x2
    ErrorCodeFlowControlError  ErrorCode = 0x3
    ErrorCodeSettingsTimeout   ErrorCode = 0x4
    ErrorCodeStreamClosed       ErrorCode = 0x5
    ErrorCodeFrameSizeError    ErrorCode = 0x6
    ErrorCodeRefusedStream     ErrorCode = 0x7
    ErrorCodeCancel             ErrorCode = 0x8
    ErrorCodeCompressionError   ErrorCode = 0x9
    ErrorCodeConnectError       ErrorCode = 0xa
    ErrorCodeEnhanceYourCalm   ErrorCode = 0xb
    ErrorCodeInadequateSecurity ErrorCode = 0xc
    ErrorCodeHttp11Required     ErrorCode = 0xd
)

// ProtocolError represents an HTTP/2 protocol error

type ProtocolError struct {

    Code      ErrorCode
```

```
Message string

}

func (e *ProtocolError) Error() string {
    return fmt.Sprintf("HTTP/2 protocol error %d: %s", e.Code, e.Message)
}

// NewProtocolError creates a new protocol error with code and message

func NewProtocolError(code ErrorCode, message string) *ProtocolError {
    return &ProtocolError{
        Code:     code,
        Message: message,
    }
}
```

D. Core Logic Skeleton Code

```
// internal/frame/types.go - Frame structure and constants (student implements parsing) GO

package frame

// HTTP/2 frame type constants

const (

    FrameTypeDATA          = 0x0
    FrameTypeHEADERS       = 0x1
    FrameTypePRIORITY      = 0x2
    FrameTypeRST_STREAM    = 0x3
    FrameTypeSETTINGS       = 0x4
    FrameTypePUSH_PROMISE  = 0x5
    FrameTypePING           = 0x6
    FrameTypeGOAWAY         = 0x7
    FrameTypeWINDOW_UPDATE  = 0x8
    FrameTypeCONTINUATION   = 0x9
)

// HTTP/2 frame flags

const (
    FlagEndStream     = 0x1
    FlagAck          = 0x1 // For SETTINGS and PING
    FlagEndHeaders   = 0x4
    FlagPadded        = 0x8
    FlagPriority     = 0x20 // For HEADERS
)

// Frame size constants

const (
```

```
MaxFrameSize      = 16777215 // 2^24 - 1

DefaultFrameSize = 16384     // Default maximum frame size

InitialWindowSize = 65535    // Default flow control window

)

// Frame represents an HTTP/2 frame with header and payload

type Frame struct {

    Length  uint32 // Frame payload length (24 bits)

    Type    uint8  // Frame type (8 bits)

    Flags   uint8  // Frame flags (8 bits)

    StreamID uint32 // Stream identifier (31 bits, R bit reserved)

    Payload []byte // Frame payload data

}

// ParseFrame reads and parses an HTTP/2 frame from the reader

// Students implement this for Milestone 1

func ParseFrame(r io.Reader) (*Frame, error) {

    // TODO 1: Read 9-byte frame header

    // TODO 2: Extract length (24 bits), type (8 bits), flags (8 bits)

    // TODO 3: Extract stream ID (31 bits) and validate reserved bit is 0

    // TODO 4: Validate frame length doesn't exceed MaxFrameSize

    // TODO 5: Read payload bytes based on frame length

    // TODO 6: Return populated Frame struct

    return nil, nil

}

// SerializeFrame writes a frame in HTTP/2 wire format to the writer

// Students implement this for Milestone 1
```

```

func SerializeFrame(w io.Writer, f *Frame) (int, error) {

    // TODO 1: Validate frame fields (length, stream ID reserved bit)

    // TODO 2: Write 24-bit length in network byte order

    // TODO 3: Write 8-bit type and 8-bit flags

    // TODO 4: Write 32-bit stream ID (clear reserved bit)

    // TODO 5: Write payload bytes

    // TODO 6: Return total bytes written

    return 0, nil

}

// ValidateFrame checks frame protocol compliance

func ValidateFrame(f *Frame) error {

    // TODO 1: Check frame length against MaxFrameSize

    // TODO 2: Validate frame type is known/supported

    // TODO 3: Check stream ID reserved bit is not set

    // TODO 4: Validate frame type specific constraints

        // - SETTINGS frames must have stream ID 0

        // - DATA frames must have non-zero stream ID

        // - Window update size must be positive

    // TODO 5: Return appropriate ProtocolError for violations

    return nil

}

```

E. Language-Specific Hints

- **Binary Parsing:** Use `encoding/binary.BigEndian` for network byte order operations. HTTP/2 uses big-endian (network) byte order for all multi-byte fields.
- **Buffered I/O:** Use `bufio.Reader` and `bufio.Writer` for efficient frame reading/writing. TCP socket I/O benefits significantly from buffering.

- **Goroutine Management:** Use `defer conn.Close()` and proper error handling to prevent connection leaks. Each goroutine should clean up its resources.
- **Atomic Operations:** Use `sync/atomic` package for flow control window operations to avoid lock overhead in the data path.
- **Memory Management:** Reuse byte slices for frame payloads where possible. Consider using `sync.Pool` for frequently allocated frame objects.
- **Error Handling:** Convert protocol violations to `ProtocolError` types with appropriate error codes for proper HTTP/2 error signaling.

F. Threading Model Implementation

GO

```
// internal/connection/handler.go - Connection goroutine implementation

func (c *Connection) processFrames() {
    // Main frame processing loop - runs in connection goroutine

    for {

        frame, err := frame.ParseFrame(c.reader)

        if err != nil {

            // TODO: Handle connection errors, send GOAWAY if needed

            return
        }

        // Route frame based on type and stream ID

        switch frame.Type {

            case frame.FrameTypeSETTINGS, frame.FrameTypePING, frame.FrameTypeGOAWAY:

                // Process connection-level frames immediately

                c.handleConnectionFrame(frame)

            case frame.FrameTypeDATA, frame.FrameTypeHEADERS:

                // Dispatch stream frames to stream processors

                c.handleStreamFrame(frame)

            default:

                // TODO: Handle unknown frame types
        }
    }

    func (c *Connection) handleStreamFrame(frame *Frame) {
```

```

c.mu.Lock()

stream, exists := c.streams[frame.StreamID]

c.mu.Unlock()

if !exists {

    // TODO: Create new stream if this is stream-opening frame

    // TODO: Validate stream ID allocation rules

}

// TODO: Process frame in stream context

// TODO: Consider using worker goroutine pool for CPU-intensive processing

}

```

G. Milestone Checkpoints

After implementing the architectural foundation:

- **Connection Handling:** Run `go run cmd/server/main.go` and verify the server accepts TCP connections on port 8080
- **Basic Frame Structure:** Implement `Frame` struct and constants, verify with `go test ./internal/frame/...`
- **Connection Wrapper:** Test `NewConnection()` creates proper buffered I/O wrappers
- **Threading Model:** Use `go run -race` to verify no race conditions in connection handling
- **Error Types:** Verify `ProtocolError` implements Go's `error` interface correctly

Signs that something is wrong:

- **Goroutine leaks:** Use `runtime.NumGoroutine()` to check goroutine count doesn't grow indefinitely
- **Connection hangs:** Check that buffered writers are flushed after frame writes
- **Race conditions:** Run tests with `-race` flag to detect concurrent access issues
- **Memory growth:** Monitor memory usage during connection accept/close cycles

Data Model

Milestone(s): This section provides the foundational data structures for milestones 1-4: Frame parsing (Milestone 1), HPACK compression (Milestone 2), stream management (Milestone 3), and flow control (Milestone 4).

Think of the HTTP/2 data model as the architectural blueprint for a sophisticated mail sorting facility. Where HTTP/1.1 was like a simple post office with one counter processing letters sequentially, HTTP/2 is like a modern logistics hub with binary-encoded packages, compressed addressing labels, multiple conveyor belts (streams) running simultaneously, and smart routing systems. Each component—frames, streams, connections, and compression tables—serves a specific role in this complex but efficient system.

The data model forms the foundation upon which all HTTP/2 operations are built. Unlike HTTP/1.1's text-based approach where messages are parsed character by character, HTTP/2 operates on structured **binary framing** where every piece of information has a precisely defined format. This binary approach enables features like **multiplexing** (multiple concurrent streams), **HPACK** header compression, and sophisticated **flow control** that would be impossible or inefficient with text-based protocols.

Understanding these data structures is crucial because HTTP/2's complexity emerges from their interactions. Frames carry data between endpoints, streams provide the abstraction for individual request-response pairs, connections manage the overall communication channel, and HPACK tables enable efficient header compression. Each structure contains not just data but also metadata that drives the protocol's advanced features.

The critical insight is that HTTP/2 inverts the traditional model: instead of thinking about HTTP requests and responses, we think about frames flowing through streams over a connection. The HTTP semantics are reconstructed from these lower-level primitives.

Frame Types

HTTP/2 frames are the atomic units of communication, analogous to packets in a network protocol. Think of frames as standardized shipping containers—each has the same 9-byte header format regardless of contents, making them easy to route and process, while the payload varies based on the frame type and purpose.

Every frame begins with a common header structure that provides essential routing and processing information. This header acts like a shipping label, telling the receiver how much data to expect, what type of processing to perform, any special handling instructions (flags), and which stream the frame belongs to.

Frame Structure Overview

Field	Type	Size	Description
Length	uint32	3 bytes	Payload length (0-16,777,215 bytes), stored as 24-bit big-endian
Type	uint8	1 byte	Frame type identifier (DATA, HEADERS, SETTINGS, etc.)
Flags	uint8	1 byte	Type-specific boolean flags for frame processing
Reserved	bit	1 bit	Reserved bit, must be unset (0)
Stream ID	uint32	31 bits	Stream identifier (0 for connection-level frames)
Payload	byte	0-16MB	Frame-specific data based on type

The length field uses a 24-bit representation, allowing payloads up to 16,777,215 bytes (approximately 16MB). However, the default maximum frame size is much smaller (16,384 bytes) for practical reasons—smaller frames enable better multiplexing fairness and reduce memory usage. The reserved bit in the stream ID field must always be zero; recipients should treat frames with this bit set as a connection error.

Decision: Fixed 9-Byte Header Format

- **Context:** Need consistent frame parsing regardless of frame type and content
- **Options Considered:** Variable-length headers, type-specific headers, fixed-length headers
- **Decision:** Fixed 9-byte header for all frame types
- **Rationale:** Enables efficient parsing with known header size, simplifies frame routing logic, and provides sufficient space for essential metadata
- **Consequences:** Some frame types may have unused header space, but parsing efficiency and implementation simplicity outweigh this minor overhead

DATA Frame Structure

DATA frames carry the actual HTTP request and response payloads—the body content that applications care about. Think of DATA frames as the cargo containers in our shipping analogy, carrying the actual goods while the frame header provides the shipping information.

Component	Type	Description
Padding Length	uint8	Optional field present only if PADDED flag is set
Data	byte	Actual HTTP payload data for the stream
Padding	byte	Optional random bytes for traffic analysis protection

DATA frames support optional padding for security purposes. When the `PADDED` flag is set, the first byte of the payload indicates how many padding bytes follow the actual data. This padding helps prevent traffic analysis attacks by obscuring the actual payload size, though it comes at the cost of bandwidth overhead.

The `END_STREAM` flag on DATA frames indicates that this is the final frame for the stream in this direction. This flag is crucial for stream lifecycle management—when a server sends a DATA frame with `END_STREAM` set, it signals that the response is complete and the stream transitions to a half-closed state.

HEADERS Frame Structure

HEADERS frames contain HTTP header fields compressed using HPACK encoding. These frames are analogous to address labels on our shipping containers, but using a sophisticated compression scheme that references previously seen addresses to save space.

Component	Type	Description
Padding Length	uint8	Optional field present only if PADDED flag is set
Exclusive	bit	Stream dependency exclusivity flag (if PRIORITY flag set)
Stream Dependency	uint31	Dependent stream ID (if PRIORITY flag set)
Weight	uint8	Stream priority weight 1-256 (if PRIORITY flag set)
Header Block Fragment	0byte	HPACK-compressed header fields
Padding	0byte	Optional random padding bytes

HEADERS frames can include priority information when the `PRIORITY` flag is set. The priority fields establish a dependency tree that helps servers decide how to allocate resources among concurrent streams. A stream can depend on another stream (creating a parent-child relationship) and have a weight indicating its relative priority among siblings.

The `END_HEADERS` flag indicates whether this frame contains the complete header block or if additional CONTINUATION frames will follow. The `END_STREAM` flag serves the same purpose as in DATA frames—marking the final frame in one direction of the stream.

The header block fragment contains HPACK-compressed data that must be processed sequentially to maintain the compression context. Frames cannot be reordered or processed independently when they contain header data.

SETTINGS Frame Structure

SETTINGS frames configure connection-level parameters and capabilities between endpoints. Think of these as the initial handshake in our postal facility where both sides agree on operating procedures, maximum package sizes, and processing capabilities.

Component	Type	Description
Identifier	uint16	Setting identifier (predefined constants)
Value	uint32	32-bit setting value

SETTINGS frames contain zero or more setting parameter pairs, each consisting of a 16-bit identifier and 32-bit value. These frames always have stream ID 0, indicating they apply to the entire connection rather than individual streams.

Common Settings Parameters

Setting	Identifier	Default Value	Description
HEADER_TABLE_SIZE	0x1	4096	Maximum HPACK dynamic table size
ENABLE_PUSH	0x2	1	Server push capability (1=enabled, 0=disabled)
MAX_CONCURRENT_STREAMS	0x3	unlimited	Maximum concurrent streams allowed
INITIAL_WINDOW_SIZE	0x4	65535	Initial flow control window size
MAX_FRAME_SIZE	0x5	16384	Maximum frame payload size
MAX_HEADER_LIST_SIZE	0x6	unlimited	Maximum header list size in bytes

The `ACK` flag in SETTINGS frames creates a request-response pattern. When an endpoint receives a SETTINGS frame without the `ACK` flag, it must apply the settings and send a SETTINGS frame with the `ACK` flag set to confirm receipt. This ensures both sides agree on the connection parameters before proceeding.

Control Frame Types

Several frame types handle connection and stream management rather than carrying application data. These frames are like the control signals in our postal facility that coordinate operations, report problems, and manage traffic flow.

Frame Type	Purpose	Stream ID	Key Fields
PING	Connection liveness testing	0	8-byte opaque data
GOAWAY	Graceful connection termination	0	Last stream ID, error code, debug data
WINDOW_UPDATE	Flow control window adjustment	0 or stream	31-bit window size increment
RST_STREAM	Immediate stream termination	stream	32-bit error code
CONTINUATION	Header block continuation	stream	Header block fragment

PING frames enable connection liveness testing and round-trip time measurement. They contain 8 bytes of opaque data that the recipient must echo back in a PING frame with the `ACK` flag set. This mechanism helps detect network partitions and measure connection latency.

GOAWAY frames initiate graceful connection shutdown by indicating the highest stream ID that will be processed. Streams with IDs higher than this value should be retried on a new connection. The frame includes an error code explaining the shutdown reason and optional debug data for troubleshooting.

Stream State

HTTP/2 streams represent individual request-response exchanges within the multiplexed connection. Think of streams as independent conveyor belts in our postal facility—each carries one conversation between client and server, but many can operate simultaneously over the same physical infrastructure.

Stream state management is one of HTTP/2's most complex aspects because streams have their own lifecycle independent of the underlying connection. Understanding stream states is crucial for implementing correct multiplexing behavior, resource management, and error handling.

Stream State Machine

The stream state machine governs how streams transition through their lifecycle based on frame types and directions. Each state represents a different phase of the request-response exchange and determines what operations are valid.

Current State	Event	Next State	Actions Taken
idle	Send/receive HEADERS	open	Create stream context, allocate resources
idle	Receive PUSH_PROMISE	reserved (remote)	Reserve stream for server push
idle	Send PUSH_PROMISE	reserved (local)	Reserve stream for outgoing push
open	Send END_STREAM	half-closed (local)	No more outbound frames allowed
open	Receive END_STREAM	half-closed (remote)	No more inbound frames expected
half-closed (local)	Receive END_STREAM	closed	Stream complete, cleanup resources
half-closed (remote)	Send END_STREAM	closed	Stream complete, cleanup resources
any	Send/receive RST_STREAM	closed	Immediate termination, cleanup
reserved (local)	Send HEADERS	half-closed (remote)	Begin push promise fulfillment
reserved (remote)	Receive HEADERS	half-closed (local)	Server beginning push delivery

The state machine prevents protocol violations and resource leaks. For example, attempting to send DATA frames on a stream in the `half-closed (local)` state is a protocol error because the local endpoint has already signaled completion with `END_STREAM`.

Decision: Explicit State Machine Implementation

- **Context:** Need to track stream lifecycle and prevent invalid frame sequences
- **Options Considered:** Implicit state tracking, explicit state machine, stateless frame processing
- **Decision:** Explicit state machine with transition validation
- **Rationale:** Provides clear error detection, prevents protocol violations, and simplifies debugging by making state transitions visible
- **Consequences:** Additional complexity in implementation but much better error handling and protocol compliance

Stream Metadata Structure

Each stream maintains metadata beyond just its state to support features like priority scheduling, flow control, and error handling.

Field	Type	Description
ID	uint32	Unique stream identifier within connection
State	StreamState	Current lifecycle state (idle, open, closed, etc.)
LocalEndStream	bool	Whether local endpoint sent END_STREAM
RemoteEndStream	bool	Whether remote endpoint sent END_STREAM
Priority	StreamPriority	Priority weight and dependency information
SendWindow	int32	Available bytes for outbound DATA frames
RecvWindow	int32	Available bytes for inbound DATA frames
Headers	HeaderFields	Accumulated header fields from HEADERS/CONTINUATION
TrailerFields	HeaderFields	Trailing headers received after body
CreatedAt	time.Time	Stream creation timestamp for diagnostics
LastActivity	time.Time	Last frame received/sent timestamp
ErrorCode	uint32	Error code if stream terminated abnormally

The `LocalEndStream` and `RemoteEndStream` flags track completion status in each direction independently. This supports HTTP/2's bidirectional nature where request and response completion can happen at different times. For example, a client might send a complete POST request (setting `LocalEndStream`) while the server is still sending a large response.

Priority information includes both dependency relationships and weight values that influence resource allocation. The dependency field can reference another stream, creating a tree structure that guides scheduling decisions. Weight values range from 1-256 and determine relative priority among sibling streams.

Flow control windows track how many bytes each endpoint can send without overwhelming the receiver. These windows operate independently in each direction and are updated through WINDOW_UPDATE frames. The initial window size comes from the connection SETTINGS, but each stream can adjust its windows independently.

Stream Identifier Allocation

Stream IDs follow specific allocation rules that prevent collisions and enable efficient lookup. Think of stream IDs as tracking numbers in our postal facility—they must be unique, predictable, and meaningful to both sender and receiver.

Initiator	ID Range	Usage Pattern
Client	1, 3, 5, ... (odd)	Client-initiated requests
Server	2, 4, 6, ... (even)	Server push streams
Connection	0	Connection-level frames only

Client-initiated streams use odd-numbered IDs starting from 1 and incrementing by 2 for each new stream.

Server push streams use even-numbered IDs following the same incrementing pattern. Stream ID 0 is reserved for connection-level frames that don't belong to any specific stream.

The incrementing requirement means stream IDs must be allocated in order—a client cannot create stream 5 before stream 3. This ordering simplifies connection state management and helps detect protocol violations. However, streams can complete out of order; stream 5 might finish before stream 3.

Stream ID exhaustion (reaching the maximum 31-bit value) requires opening a new connection. In practice, this limit (over 2 billion streams) is rarely reached, but implementations should handle this gracefully by refusing new stream creation and potentially initiating connection migration.

⚠ Pitfall: Stream ID Reuse A common implementation mistake is attempting to reuse stream IDs after streams close. HTTP/2 explicitly forbids this—once a stream ID is used, it cannot be reused on the same connection. Attempting to create a stream with a previously used ID must trigger a connection error. This restriction prevents confusion and simplifies state management.

Stream Priority and Dependencies

HTTP/2 supports prioritization through a dependency tree system where streams can depend on other streams and specify relative weights. This system helps servers decide resource allocation when handling multiple concurrent requests.

Field	Type	Range	Description
StreamDependency	uint31	0-2^31-1	ID of stream this stream depends on
Exclusive	bool	true/false	Whether dependency is exclusive
Weight	uint8	1-256	Relative priority weight

Stream dependencies create parent-child relationships where child streams should be prioritized only after their parent completes. The exclusive flag determines whether a new dependency replaces existing children (exclusive=true) or joins them as siblings (exclusive=false).

Weight values provide relative prioritization among sibling streams. A stream with weight 200 should receive roughly twice the resources of a stream with weight 100, assuming all other factors are equal. However, priority is advisory—servers can use this information to guide scheduling but are not required to implement any specific algorithm.

The dependency tree can change dynamically as streams open and close. When a parent stream closes, its children typically become children of the parent's parent, maintaining the relative priority relationships. Implementations must handle circular dependencies by breaking cycles, usually by making the dependent stream a child of the dependency target's parent.

Connection State

The HTTP/2 connection represents the overall communication channel between client and server, managing multiple concurrent streams and connection-level settings. Think of the connection as the central control system for our postal facility—it coordinates all the conveyor belts (streams), manages facility-wide policies, and handles communication with the other facility.

Connection state is more complex than individual stream state because it must coordinate multiple streams while maintaining its own configuration, flow control windows, and error handling. The connection acts as both a container for streams and an active participant in the protocol with its own frame types and state transitions.

Connection Configuration

Connection-level settings affect all streams and overall behavior. These settings are negotiated through SETTINGS frames and can be updated during the connection lifetime.

Setting	Local Value	Remote Value	Description
HeaderTableSize	uint32	uint32	Maximum HPACK dynamic table size for each direction
EnablePush	bool	bool	Whether server push is allowed
MaxConcurrentStreams	uint32	uint32	Maximum concurrent open streams
InitialWindowSize	uint32	uint32	Default flow control window for new streams
MaxFrameSize	uint32	uint32	Maximum frame payload size
MaxHeaderListSize	uint32	uint32	Maximum total header list size

Each setting has both local and remote values because settings are directional. When the local endpoint sends a SETTINGS frame, it's declaring the values the remote endpoint should use when sending to the local endpoint. This asymmetry allows each side to declare its own capabilities and constraints.

Settings changes take effect immediately upon acknowledgment (SETTINGS frame with ACK flag). However, some settings like `InitialWindowSize` only affect new streams, while others like `MaxFrameSize` affect all subsequent frames. Implementation must carefully handle the timing of settings application to avoid protocol violations.

Decision: Separate Local and Remote Settings

- **Context:** Settings are directional—each endpoint declares constraints for the other
- **Options Considered:** Single settings object, bidirectional settings, separate local/remote tracking
- **Decision:** Track local and remote settings separately
- **Rationale:** Prevents confusion about which constraints apply in which direction, enables asymmetric configurations, and simplifies validation logic
- **Consequences:** Slightly more complex state management but much clearer semantics and fewer bugs

Connection Flow Control

Connection-level flow control operates independently of stream-level flow control, providing an additional layer of backpressure management. This dual-level system ensures that individual streams cannot monopolize connection resources while still allowing fine-grained per-stream control.

Window Type	Purpose	Initial Size	Update Mechanism
Connection Send	Bytes we can send to remote	InitialWindowSize × 1	Remote WINDOW_UPDATE frames
Connection Recv	Bytes remote can send to us	InitialWindowSize × 1	Our WINDOW_UPDATE frames
Stream Send	Per-stream send allowance	InitialWindowSize	Remote WINDOW_UPDATE frames
Stream Recv	Per-stream recv allowance	InitialWindowSize	Our WINDOW_UPDATE frames

Every DATA frame payload consumes both connection-level and stream-level send window. Before sending a DATA frame, the implementation must verify that both windows have sufficient space. If either window is exhausted, the frame cannot be sent until the appropriate WINDOW_UPDATE frame arrives.

Window updates can target either the connection (stream ID 0) or specific streams (stream ID > 0). Connection window updates affect the overall connection allowance, while stream window updates affect only that stream. Both types of updates are cumulative—receiving multiple WINDOW_UPDATE frames adds their values to the current window size.

Window size management requires careful accounting to prevent underflow (sending more than allowed) and overflow (accumulating windows larger than $2^{31}-1$). Underflow is a protocol error requiring connection termination, while overflow should be treated as the maximum window size.

Active Streams Management

The connection maintains a registry of all active streams along with their states and metadata. This registry enables frame routing, resource management, and cleanup operations.

Registry Component	Type	Purpose
StreamMap	map[uint32]*Stream	Active stream lookup by ID
LastStreamID	uint32	Highest stream ID seen from remote
NextStreamID	uint32	Next stream ID to allocate locally
StreamCount	uint32	Current number of open streams
MaxStreams	uint32	Maximum concurrent streams allowed
ClosedStreams	LRU cache	Recently closed streams for late frame detection

The stream map provides O(1) lookup for routing incoming frames to their destination streams. Frame processing involves extracting the stream ID from the frame header and looking up the corresponding stream object. Frames for unknown streams may indicate protocol errors or late arrivals after stream cleanup.

Stream counting enforces the `MaxConcurrentStreams` setting by rejecting new stream creation when the limit is reached. The count includes all streams in active states (open, half-closed) but excludes idle and closed streams. This enforcement prevents resource exhaustion attacks where one endpoint creates excessive streams.

The closed streams cache handles the common case of frames arriving after stream cleanup. When a stream closes, its full state is discarded to save memory, but a lightweight entry remains in the cache to detect and properly handle late-arriving frames. This cache typically uses LRU eviction with a modest size limit.

⚠ Pitfall: Stream Cleanup Timing A subtle but critical issue is determining when to clean up closed streams. Cleaning up too aggressively can cause protocol errors if late frames arrive, but keeping streams too long wastes memory. The recommended approach is immediate lightweight cleanup (remove from active map but keep in closed cache) followed by eventual full cleanup after a timeout period or cache pressure.

Connection Error Handling

Connection errors affect the entire connection and typically result in connection termination through GOAWAY frames. These errors indicate protocol violations, resource exhaustion, or other conditions that make continued operation impossible or unsafe.

Error Type	Error Code	Typical Causes	Recovery Action
Protocol Error	0x1	Malformed frames, invalid state transitions	Send GOAWAY, close connection
Internal Error	0x2	Implementation bugs, resource allocation failures	Send GOAWAY, close connection
Flow Control Error	0x3	Window violations, invalid WINDOW_UPDATE	Send GOAWAY, close connection
Settings Timeout	0x4	No SETTINGS ACK received	Send GOAWAY, close connection
Stream Closed	0x5	Frames on closed streams	Send GOAWAY, close connection
Frame Size Error	0x6	Frames exceeding size limits	Send GOAWAY, close connection
Refused Stream	0x7	Cannot create requested stream	Send RST_STREAM, continue connection
Cancel	0x8	Processing cancelled	Context-dependent
Compression Error	0x9	HPACK decompression failure	Send GOAWAY, close connection
Connect Error	0xa	TCP connection problems	Close connection
Enhance Calm	0xb	Excessive resource usage	Send GOAWAY, close connection
Inadequate Security	0xc	TLS configuration insufficient	Send GOAWAY, close connection
HTTP/1.1 Required	0xd	Must downgrade to HTTP/1.1	Send GOAWAY, close connection

Most connection errors require immediate termination because they indicate fundamental protocol violations that cannot be recovered. The GOAWAY frame provides graceful termination by indicating which streams completed successfully and can be safely retried on a new connection.

Some errors like `Refused Stream` are stream-specific and can be handled with RST_STREAM frames without affecting the connection. However, repeated stream errors might indicate systematic problems that warrant connection termination.

The error handling implementation should distinguish between recoverable and fatal conditions, provide clear error messages for debugging, and ensure proper resource cleanup during connection termination. Logging error details helps diagnose implementation bugs and protocol violations.

HPACK Tables

HPACK (HTTP/2 Header Compression) uses a sophisticated table-based compression scheme that maintains both static and dynamic tables to achieve significant header size reduction. Think of HPACK tables as a sophisticated address book system in our postal facility—frequently used addresses are stored in numbered slots so future packages can reference them by number instead of writing the full address.

The compression effectiveness comes from HTTP's repetitive nature: most requests contain many identical or similar headers (User-Agent, Accept, Authorization, etc.). Instead of transmitting these headers repeatedly, HPACK assigns them indices and transmits only the index references along with any new or changed headers.

HPACK operates at the connection level with separate compression contexts for each direction. This means client-to-server headers and server-to-client headers have independent compression state, allowing each endpoint to optimize for its typical header patterns.

The critical insight is that HPACK provides stateful compression—the compression context evolves as headers are processed, creating better compression ratios but requiring careful synchronization between endpoints.

Static Table Structure

The static table contains 61 predefined entries covering the most common HTTP headers. This table is defined by RFC 7541 and is identical for all HTTP/2 connections, providing immediate compression benefits without any setup overhead.

Index	Name	Value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
9	:status	204
10	:status	206
11	:status	304
12	:status	400
13	:status	404
14	:status	500
15	accept-charset	
16	accept-encoding	gzip, deflate
17	accept-language	
18	accept-ranges	
19	accept	
20	access-control-allow-origin	
21	age	
22	allow	
23	authorization	
24	cache-control	
25	content-disposition	
26	content-encoding	

Index	Name	Value
27	content-language	
28	content-length	
29	content-location	
30	content-range	
31	content-type	
32	cookie	
33	date	
34	etag	
35	expect	
36	expires	
37	from	
38	host	
39	if-match	
40	if-modified-since	
41	if-none-match	
42	if-range	
43	if-unmodified-since	
44	last-modified	
45	link	
46	location	
47	max-forwards	
48	proxy-authenticate	
49	proxy-authorization	
50	range	
51	referer	
52	refresh	

Index	Name	Value
53	retry-after	
54	server	
55	set-cookie	
56	strict-transport-security	
57	transfer-encoding	
58	user-agent	
59	vary	
60	via	
61	www-authenticate	

The static table includes HTTP/2 pseudo-headers (prefixed with `:`) that replace the HTTP/1.1 request line and status line. These pseudo-headers must appear before regular headers and have specific ordering requirements.

Entries with empty values represent name-only matches. For example, index 1 (`:authority`) matches any header with that name regardless of value. This allows partial compression where the name is indexed but the value is transmitted literally.

The static table provides immediate benefits: a simple `GET /` request can reference indices 2 and 4 instead of transmitting the full `:method: GET` and `:path: /` headers. More complex requests benefit even more as they reference additional static entries.

Dynamic Table Structure

The dynamic table maintains connection-specific header entries that accumulate during the connection lifetime. This table starts empty and grows as headers are processed, creating increasingly effective compression for connection-specific patterns.

Component	Type	Description
Entries	<code>[]HeaderEntry</code>	Ordered list of header name-value pairs
Size	<code>uint32</code>	Current table size in bytes
MaxSize	<code>uint32</code>	Maximum table size from <code>SETTINGS_HEADER_TABLE_SIZE</code>
InsertionPoint	<code>uint32</code>	Index where next entry will be inserted
EvictionCount	<code>uint32</code>	Number of entries evicted due to size constraints

Dynamic table entries are indexed starting from 62 (after the static table) and use a FIFO eviction strategy. When adding a new entry would exceed the maximum table size, the oldest entries are evicted until sufficient space is available.

The table size calculation includes both the header name and value lengths plus 32 bytes of overhead per entry to account for implementation storage costs. This overhead prevents table thrashing with very small headers and provides a realistic estimate of memory usage.

Dynamic Table Entry Management

Operation	Trigger	Action Taken
Addition	Literal header with indexing	Insert at head, evict from tail if needed
Lookup	Indexed header reference	Return name-value pair for index
Eviction	Table size exceeds maximum	Remove oldest entries until under limit
Resize	SETTINGS_HEADER_TABLE_SIZE change	Adjust maximum, evict if necessary
Clear	Table size set to 0	Remove all entries immediately

Entry addition happens when the encoder chooses to make a header indexable (literal header with incremental indexing). This decision balances compression benefits against table memory usage. Headers likely to repeat (like Authorization tokens) benefit from indexing, while unique headers (like timestamps) do not.

Index lookup involves checking both static and dynamic tables. Indices 1-61 reference the static table directly, while indices 62 and above reference dynamic table entries. The implementation must handle the index offset correctly to avoid off-by-one errors.

Eviction maintains the FIFO property—oldest entries are removed first regardless of their usage frequency. This simplicity prevents complex cache management but may evict useful entries. Some implementations use smarter eviction policies as optimizations, but basic FIFO is required for interoperability.

Decision: FIFO Eviction Strategy

- **Context:** Need deterministic eviction behavior for interoperability between implementations
- **Options Considered:** LRU eviction, frequency-based eviction, FIFO eviction
- **Decision:** FIFO eviction as specified in RFC 7541
- **Rationale:** Ensures identical table state between encoder and decoder, simplifies implementation, and avoids complex cache management algorithms
- **Consequences:** May evict useful entries that could provide better compression, but guarantees correctness and interoperability

Header Encoding Representations

HPACK supports multiple encoding representations for headers based on caching strategy and compression requirements. These representations provide flexibility in balancing compression ratio, table management, and processing overhead.

Representation	Pattern	Indexing	Use Case
Indexed Header	1xxxxxxx	Reference only	Exact name-value match in table
Literal with Incremental Indexing	01xxxxxx	Add to dynamic table	Likely to repeat in future
Literal without Indexing	0000xxxx	No table modification	Unique or security-sensitive
Literal Never Indexed	0001xxxx	No indexing, no proxy caching	Security-sensitive values
Dynamic Table Size Update	001xxxxx	Table management	Change table size limit

Indexed headers provide maximum compression by referencing table entries with a single byte (for indices 1-127). This representation handles the common case where headers exactly match previous values.

Literal representations handle new or modified headers by transmitting the name (indexed or literal) and value (literal or Huffman-encoded). The indexing behavior determines whether the header is added to the dynamic table for future reference.

The "never indexed" representation provides security for sensitive headers like Authorization or Set-Cookie values. These headers are transmitted literally but marked to prevent caching by intermediaries that might expose the values.

Huffman Encoding Integration

HPACK optionally applies Huffman encoding to literal string values (header names and values) to achieve additional compression. The Huffman codes are optimized for typical HTTP header content and can provide 20-30% additional compression.

String Type	Huffman Flag	Encoding Options
Header Name	Optional	Literal or Huffman-encoded
Header Value	Optional	Literal or Huffman-encoded
Indexed Reference	N/A	Index number only

The Huffman table is defined in RFC 7541 Appendix B and includes optimized codes for all 256 possible byte values. Common characters in HTTP headers (letters, digits, common punctuation) receive shorter codes, while rare characters receive longer codes.

Huffman encoding operates at the byte level and requires careful handling of padding bits in the final symbol. The decoder must validate that padding bits are all 1's as specified, preventing certain types of compression attacks.

⚠ Pitfall: Huffman Decoding Boundary Errors A common implementation error occurs when handling Huffman-encoded strings that don't align on byte boundaries. The final byte may contain padding bits that must be validated. Incorrect padding validation can lead to security vulnerabilities or interoperability problems. Always verify that padding bits are set to 1 and that the padding doesn't form a valid symbol prefix.

Table Synchronization and Error Handling

HPACK requires perfect synchronization between encoder and decoder tables. Any mismatch in table state causes subsequent compression/decompression failures, requiring connection termination.

Error Condition	Detection Method	Recovery Action
Invalid Index	Index exceeds table size	Connection error COMPRESSION_ERROR
Table Size Violation	Entry addition exceeds limit	Connection error COMPRESSION_ERROR
Invalid Huffman Sequence	Decoding failure	Connection error COMPRESSION_ERROR
String Length Mismatch	Decoded length doesn't match declared	Connection error COMPRESSION_ERROR
Reserved Bits Set	Non-zero reserved bits in patterns	Connection error COMPRESSION_ERROR

Table synchronization failures are connection-fatal because the compression context cannot be recovered. Unlike stream errors that affect only one stream, HPACK errors affect all subsequent header processing and require starting a new connection.

The implementation must validate all HPACK input carefully to detect corruption or malicious input early. This includes bounds checking on indices, validating string lengths, and verifying Huffman decoding results.

Proper error handling includes detailed logging for debugging while avoiding information leakage that might help attackers. HPACK errors often indicate implementation bugs rather than network issues, making good diagnostics essential for development.

Implementation Guidance

This subsection provides practical guidance for implementing the HTTP/2 data structures in Go, focusing on the core types and their relationships.

Technology Recommendations

Component	Simple Option	Advanced Option
Frame Parsing	Manual binary parsing with encoding/binary	Protocol buffer schemas with custom codegen
HPACK Tables	Native maps and slices	Specialized data structures with memory pooling
Stream Management	sync.RWMutex with map[uint32]*Stream	Concurrent hash maps with lock-free operations
Flow Control	Simple integer tracking	Sophisticated window scheduling algorithms
Error Handling	Standard error types	Rich error types with context and stack traces

Recommended File Structure

```

internal/http2/
├── frame.go           ← Frame types and parsing
├── frame_test.go      ← Frame parsing tests
├── connection.go       ← Connection state management
├── connection_test.go  ← Connection tests
├── stream.go           ← Stream state and lifecycle
├── stream_test.go      ← Stream state machine tests
└── hpack/
    ├── table.go          ← Static and dynamic table implementation
    ├── decoder.go         ← HPACK decoding logic
    ├── encoder.go         ← HPACK encoding logic
    ├── huffman.go         ← Huffman coding tables and functions
    └── hpack_test.go      ← HPACK compression tests
└── errors.go           ← HTTP/2 specific error types
└── testdata/
    ├── frames/            ← Test vectors and sample data
    └── hpack/              ← Binary frame test cases
                            ← HPACK test vectors from RFC

```

Core Data Structure Implementation

GO

```
// Frame represents the basic HTTP/2 frame structure

type Frame struct {

    Length    uint32 // 24-bit payload length

    Type      uint8  // Frame type identifier

    Flags     uint8  // Frame-specific flags

    StreamID uint32 // Stream identifier (31-bit)

    Payload   []byte // Frame payload data

}

// ParseFrame reads and parses HTTP/2 frame from reader

func ParseFrame(r io.Reader) (*Frame, error) {

    // TODO 1: Read exactly 9 bytes for frame header

    // TODO 2: Parse length field from first 3 bytes (big-endian)

    // TODO 3: Extract type and flags from bytes 3-4

    // TODO 4: Parse stream ID from bytes 5-8, clear reserved bit

    // TODO 5: Validate length doesn't exceed MaxFrameSize

    // TODO 6: Read payload bytes based on length field

    // TODO 7: Return populated Frame struct

    // Hint: Use binary.BigEndian for multi-byte fields

}

// SerializeFrame writes frame in HTTP/2 wire format

func SerializeFrame(w io.Writer, f *Frame) (int, error) {

    // TODO 1: Create 9-byte header buffer

    // TODO 2: Write 24-bit length in big-endian format

    // TODO 3: Write type and flags bytes

    // TODO 4: Write 32-bit stream ID with reserved bit cleared

    // TODO 5: Write header buffer to writer
```

```
// TODO 6: Write payload if length > 0

// TODO 7: Return total bytes written

}

// Connection represents HTTP/2 connection state

type Connection struct {

    conn          net.Conn

    reader        *bufio.Reader

    writer        *bufio.Writer

    mu            sync.RWMutex

    closed        bool

    // Settings and configuration

    localSettings map[uint16]uint32

    remoteSettings map[uint16]uint32

    settingsPending map[uint16]uint32

    // Stream management

    streams        map[uint32]*Stream

    lastStreamID   uint32

    nextStreamID   uint32

    maxStreams     uint32

    // Flow control

    sendWindow     int32

    recvWindow     int32

    initialWindowSize uint32
```

```
// HPACK context

hpackDecoder      *hpack.Decoder
hpackEncoder      *hpack.Encoder

}

// NewConnection creates buffered connection wrapper

func NewConnection(conn net.Conn) *Connection {

    // TODO 1: Create buffered reader and writer for connection

    // TODO 2: Initialize default settings maps

    // TODO 3: Create empty streams map

    // TODO 4: Set initial flow control windows

    // TODO 5: Initialize HPACK encoder/decoder with default table size

    // TODO 6: Return populated Connection struct

}

// Stream represents individual HTTP/2 stream

type Stream struct {

    id          uint32
    state       StreamState
    localEndStream  bool
    remoteEndStream bool

    // Priority and dependencies

    weight      uint8
    dependency  uint32
    exclusive   bool
```

```
// Flow control

sendWindow      int32
recvWindow      int32


// Header accumulation

headers         http.Header
trailers        http.Header


// Metadata

createdAt       time.Time
lastActivity    time.Time
errorCode       uint32

}

// StreamState represents stream lifecycle states

type StreamState int

const (
    StreamStateIdle StreamState = iota
    StreamStateReservedLocal
    StreamStateReservedRemote
    StreamStateOpen
    StreamStateHalfClosedLocal
    StreamStateHalfClosedRemote
    StreamStateClosed
)
```

HPACK Table Implementation

GO

```
// Package hpack implements HTTP/2 header compression

package hpack

// StaticTable provides predefined header entries

var StaticTable = [62]HeaderEntry{

    {}, // Index 0 unused

    {Name: ":authority", Value: ""},
    {Name: ":method", Value: "GET"},
    {Name: ":method", Value: "POST"},

    // TODO: Complete static table entries 4-61 from RFC 7541
}

// DynamicTable manages connection-specific header entries

type DynamicTable struct {

    entries []HeaderEntry

    size      uint32

    maxSize   uint32

    insertIdx uint32

    evictIdx  uint32

}

// HeaderEntry represents a name-value header pair

type HeaderEntry struct {

    Name  string

    Value string

    Size  uint32 // Calculated as len(name) + len(value) + 32

}

// Add inserts new entry and evicts old entries if necessary
```

```

func (dt *DynamicTable) Add(name, value string) {

    // TODO 1: Calculate entry size including 32-byte overhead

    // TODO 2: Create new HeaderEntry with name, value, and size

    // TODO 3: Check if single entry exceeds maxSize (reject if so)

    // TODO 4: Evict old entries until space available

    // TODO 5: Insert new entry at head of table

    // TODO 6: Update table size and indices

}

// Lookup retrieves entry by index (static + dynamic)

func (dt *DynamicTable) Lookup(index uint32) (HeaderEntry, bool) {

    // TODO 1: Check if index is in static table range (1-61)

    // TODO 2: Return static table entry if found

    // TODO 3: Convert to dynamic table index (subtract 62)

    // TODO 4: Validate dynamic index is within bounds

    // TODO 5: Return dynamic table entry

}

// Resize adjusts maximum table size and evicts if necessary

func (dt *DynamicTable) Resize(newSize uint32) {

    // TODO 1: Update maxSize field

    // TODO 2: Evict entries from tail until size <= newSize

    // TODO 3: If newSize is 0, clear all entries immediately

}

```

Frame Type Constants and Validation

```
// Frame type constants

const (
    FrameTypeDATA          uint8 = 0x0
    FrameTypeHEADERS       uint8 = 0x1
    FrameTypePRIORITY      uint8 = 0x2
    FrameTypeRST_STREAM    uint8 = 0x3
    FrameTypeSETTINGS       uint8 = 0x4
    FrameTypePUSH_PROMISE  uint8 = 0x5
    FrameTypePING           uint8 = 0x6
    FrameTypeGOAWAY         uint8 = 0x7
    FrameTypeWINDOW_UPDATE  uint8 = 0x8
    FrameTypeCONTINUATION   uint8 = 0x9
)

// Frame flag constants

const (
    FlagEndStream     uint8 = 0x1
    FlagEndHeaders    uint8 = 0x4
    FlagPadded        uint8 = 0x8
    FlagPriority      uint8 = 0x20
)

// Protocol constants

const (
    MaxFrameSize      uint32 = 16777215 // 2^24 - 1
    DefaultFrameSize  uint32 = 16384    // 16KB
    InitialWindowSize uint32 = 65535    // 64KB - 1
)
```

```
// Error codes

const (
    ErrorCodeNoError          uint32 = 0x0
    ErrorCodeProtocolError    uint32 = 0x1
    ErrorCodeInternalError    uint32 = 0x2
    ErrorCodeFlowControlError uint32 = 0x3
    ErrorCodeSettingsTimeout  uint32 = 0x4
    ErrorCodeStreamClosed     uint32 = 0x5
    ErrorCodeFrameSizeError   uint32 = 0x6
    ErrorCodeRefusedStream    uint32 = 0x7
    ErrorCodeCancel           uint32 = 0x8
    ErrorCodeCompressionError uint32 = 0x9
    ErrorCodeConnectError     uint32 = 0xa
    ErrorCodeEnhanceYourCalm  uint32 = 0xb
    ErrorCodeInadequateSecurity uint32 = 0xc
    ErrorCodeHTTP11Required   uint32 = 0xd
)

// ValidateFrame checks frame protocol compliance

func ValidateFrame(f *Frame) error {
    // TODO 1: Verify length field matches payload size

    // TODO 2: Check length doesn't exceed MaxFrameSize

    // TODO 3: Validate reserved bit in stream ID is unset

    // TODO 4: Verify frame type is recognized

    // TODO 5: Check type-specific constraints (e.g., SETTINGS stream ID = 0)

    // TODO 6: Validate flag combinations are legal for frame type

    // TODO 7: Return ProtocolError for any violations
}
```

```
}
```

Language-Specific Implementation Hints

- Use `binary.BigEndian` for all multi-byte integer parsing and serialization
- Implement frame parsing with `io.LimitReader` to prevent over-reading payload
- Use `sync.RWMutex` for connection-level state that has frequent readers
- Implement HPACK with careful bounds checking to prevent index errors
- Use `context.Context` for request cancellation and timeout handling
- Consider using `sync.Pool` for frame and buffer allocation in high-performance scenarios

Milestone Checkpoints

After implementing the basic data structures, verify functionality with these checkpoints:

1. **Frame Parsing Test:** Create a DATA frame with known payload, serialize it, then parse it back. Verify all fields match exactly.
2. **HPACK Static Table Test:** Look up index 2 (`:method: GET`) and index 8 (`:status: 200`) from static table. Verify correct name-value pairs returned.
3. **Dynamic Table Test:** Add several header entries to dynamic table, then look them up by index. Verify FIFO eviction when table size limit exceeded.
4. **Stream State Test:** Create stream in idle state, transition through open to half-closed-local, verify state transitions follow protocol rules.

Expected output for frame parsing test:

```
Original: Length=12, Type=0, Flags=1, StreamID=5, Payload="Hello World!"  
Parsed:   Length=12, Type=0, Flags=1, StreamID=5, Payload="Hello World!"  
✓ Frame serialization/parsing works correctly
```

Common Debugging Issues

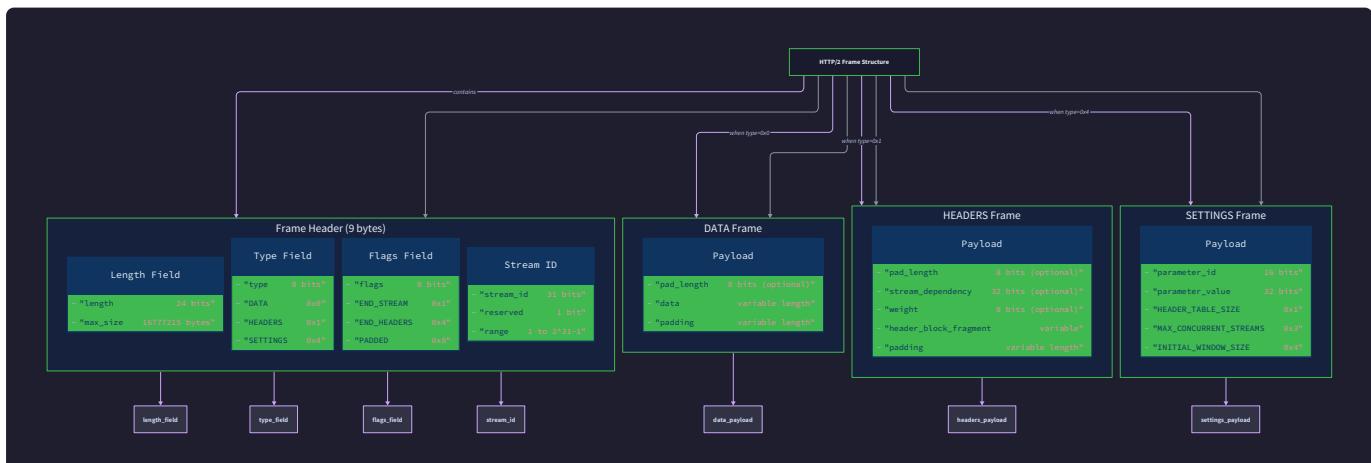
Symptom	Likely Cause	Diagnosis	Fix
Frame length mismatch	Endianness error in parsing	Check byte order in length field	Use binary.BigEndian consistently
Stream ID has high bit set	Reserved bit not cleared	Inspect stream ID parsing	Mask with 0x7FFFFFFF
HPACK index out of bounds	Off-by-one in table lookup	Verify index calculation	Static table is 1-61, dynamic starts at 62
Dynamic table size grows unbounded	Eviction not implemented	Check Add() method	Implement proper FIFO eviction
Connection hangs on frame read	Incorrect payload length	Verify length field parsing	Validate length before reading payload

Binary Framing Engine

Milestone(s): This section primarily addresses Milestone 1 (Binary Framing) by implementing HTTP/2's binary frame format with parsing, validation, and serialization capabilities.

Think of HTTP/2's binary framing as the difference between sending handwritten letters and using a structured telegram system. In HTTP/1.1, we send human-readable text messages that require careful parsing to understand where headers end and bodies begin. HTTP/2's binary framing is like having a standardized telegram format with a precise header that tells you exactly what type of message follows and how long it is. This structured approach eliminates ambiguity and makes parsing dramatically more efficient.

The **binary framing engine** serves as the foundation layer of our HTTP/2 server, responsible for converting between the raw bytes on the wire and structured frame objects that higher-level components can work with. This engine must handle the intricate details of binary protocol parsing while providing a clean abstraction for the rest of the system.



Frame Parsing

The core challenge in HTTP/2 frame parsing lies in correctly interpreting the 9-byte frame header followed by a variable-length payload. Unlike HTTP/1.1's text-based format where parsing involves string manipulation and delimiter searching, HTTP/2 parsing requires precise binary operations with careful attention to endianness and bit manipulation.

Decision: Fixed 9-Byte Header Design

- **Context:** HTTP/2 needed a way to prefix each frame with metadata about its contents, type, and destination stream
- **Options Considered:** Variable-length headers with length prefixes, fixed-size headers, or embedded metadata within payloads
- **Decision:** Use a fixed 9-byte header with specific bit layouts for each field
- **Rationale:** Fixed headers enable efficient parsing without lookahead, eliminate parsing ambiguity, and provide sufficient space for all necessary metadata
- **Consequences:** Every frame has exactly 9 bytes of overhead, but parsing becomes predictable and fast

The frame header contains four distinct fields packed into exactly 9 bytes:

Field	Size	Type	Description
Length	3 bytes	uint24	Size of frame payload in bytes (0 to 16,777,215)
Type	1 byte	uint8	Frame type identifier (DATA, HEADERS, SETTINGS, etc.)
Flags	1 byte	uint8	Type-specific flags (END_STREAM, END_HEADERS, etc.)
Stream ID	4 bytes	uint31	Target stream identifier (31 bits, with reserved bit)

The parsing algorithm follows these precise steps:

1. **Read exactly 9 bytes** from the connection's input stream, handling potential partial reads due to TCP fragmentation
2. **Extract the 24-bit length field** by combining the first three bytes in network byte order (big-endian)
3. **Validate the length field** against the maximum frame size setting and connection limits
4. **Extract the 8-bit type field** directly from the fourth byte
5. **Extract the 8-bit flags field** directly from the fifth byte
6. **Extract the 31-bit stream ID** from bytes 6-9, masking off the reserved bit in the most significant position
7. **Validate the stream ID** for proper odd/even assignment and range constraints
8. **Read the payload bytes** based on the length field, handling partial reads and EOF conditions

The most critical aspect of frame parsing is handling the endianness correctly. HTTP/2 uses network byte order (big-endian) for all multi-byte fields, which means the most significant byte comes first on the wire. The length field requires special handling since it spans three bytes rather than a standard 2, 4, or 8-byte integer size.

The reserved bit in the stream ID field must always be zero on the wire. Frames with the reserved bit set to 1 constitute a protocol violation that requires connection termination with a `PROTOCOL_ERROR`.

Partial Read Handling: TCP connections provide no guarantee that a single read operation will return the requested number of bytes. The frame parser must accumulate bytes until it has collected the complete header or payload, while detecting connection closures that occur mid-frame.

Frame Boundary Synchronization: If the parser ever gets out of sync with frame boundaries (due to implementation bugs or corrupted data), the entire connection becomes unusable. The parser must validate that frame headers make sense and that payload lengths align with frame boundaries.

The parsing operation yields a structured `Frame` object that encapsulates all header fields plus the raw payload bytes. Higher-level components work exclusively with these structured frames rather than raw byte streams.

Frame Type Implementation

HTTP/2 defines multiple frame types, each with specific payload formats and semantic meanings. Our implementation focuses on the core frame types essential for basic HTTP/2 functionality: DATA, HEADERS, SETTINGS, PING, GOAWAY, and WINDOW_UPDATE frames.

Decision: Frame Type Abstraction Strategy

- **Context:** Different frame types have completely different payload structures and processing requirements
- **Options Considered:** Single monolithic parser, type-specific parser classes, or generic frame structure with typed payload interpretation
- **Decision:** Generic Frame structure with type-specific payload parsing methods
- **Rationale:** Maintains parsing efficiency while allowing specialized handling of each frame type's unique requirements
- **Consequences:** Frame parsing and payload interpretation are separated, enabling easier testing and frame type extension

DATA Frames

DATA frames carry the actual HTTP request or response body content. They represent the payload data that applications ultimately consume, making them the most straightforward frame type from a parsing perspective.

Field	Size	Description
Pad Length	0-1 bytes	Optional padding length (present if PADDED flag set)
Data	Variable	Application data bytes
Padding	0-255 bytes	Optional zero-valued padding bytes

DATA frame parsing involves:

1. **Check the PADDED flag** to determine if padding length field is present
2. **Extract pad length** from first payload byte if PADDED flag is set
3. **Validate padding length** against total payload size to prevent underflow
4. **Extract data portion** by skipping pad length byte and trailing padding
5. **Verify padding bytes** are zero-valued (optional but recommended for security)

The END_STREAM flag on DATA frames indicates that no more frames will be sent on this stream from the sender, making it a critical signal for stream lifecycle management.

HEADERS Frames

HEADERS frames contain HTTP header name-value pairs compressed using HPACK encoding. These frames initiate new requests (client to server) or responses (server to client) and may be split across multiple frames if header content exceeds frame size limits.

Field	Size	Description
Pad Length	0-1 bytes	Optional padding length (present if PADDED flag set)
Exclusive	0-1 bits	Stream dependency exclusivity flag (present if PRIORITY flag set)
Stream Dependency	0-31 bits	Parent stream ID for priority tree (present if PRIORITY flag set)
Weight	0-1 bytes	Stream weight value 1-256 (present if PRIORITY flag set)
Header Block Fragment	Variable	HPACK-compressed header data
Padding	0-255 bytes	Optional zero-valued padding bytes

HEADERS frame parsing is more complex due to optional fields:

1. **Check PADDED flag** and extract padding length if present
2. **Check PRIORITY flag** and extract dependency information if present
3. **Extract header block fragment** from remaining payload after accounting for padding
4. **Validate header block size** and fragment boundaries
5. **Store fragment for HPACK decompression** (handled by HPACK engine)

The END_HEADERS flag indicates that this frame contains the last header block fragment for this header set. Without this flag, subsequent CONTINUATION frames will carry additional fragments.

SETTINGS Frames

SETTINGS frames communicate configuration parameters that affect connection behavior. They must be acknowledged by the peer and take effect only after acknowledgment.

Field	Size	Description
Settings Parameters	Multiple of 6 bytes	Array of setting ID and value pairs

Each setting parameter consists of:

Field	Size	Description
Identifier	2 bytes	Setting type identifier
Value	4 bytes	Setting value (32-bit unsigned integer)

SETTINGS frame parsing involves:

1. **Validate payload length** is multiple of 6 bytes (each setting is exactly 6 bytes)
2. **Extract setting pairs** by reading 2-byte ID followed by 4-byte value
3. **Validate setting identifiers** against known settings types
4. **Validate setting values** against allowed ranges for each setting type
5. **Apply settings** to connection state (after sending acknowledgment)

Common setting identifiers include:

Setting	ID	Description	Default Value
HEADER_TABLE_SIZE	0x1	HPACK dynamic table size	4096
ENABLE_PUSH	0x2	Server push capability	1
MAX_CONCURRENT_STREAMS	0x3	Maximum concurrent streams	Unlimited
INITIAL_WINDOW_SIZE	0x4	Stream flow control window	65535
MAX_FRAME_SIZE	0x5	Maximum frame payload size	16384

Control Frames (PING, GOAWAY, WINDOW_UPDATE)

These frames handle connection-level control operations:

PING Frames carry 8 bytes of opaque data for connection liveness testing:

Field	Size	Description
Opaque Data	8 bytes	Data echoed back in PING acknowledgment

GOAWAY Frames signal connection termination with error information:

Field	Size	Description
Last Stream ID	4 bytes	Highest stream ID processed
Error Code	4 bytes	Reason for connection closure
Additional Debug Data	Variable	Optional human-readable error description

WINDOW_UPDATE Frames adjust flow control windows:

Field	Size	Description
Window Size Increment	4 bytes	Number of bytes to add to flow control window

Frame Validation

Protocol compliance requires extensive validation of every parsed frame before processing. Frame validation serves as the first line of defense against malformed data, protocol violations, and potential security attacks.

Decision: Validation Timing Strategy

- **Context:** Frame validation could occur during parsing, after parsing, or lazily when frames are processed
- **Options Considered:** Inline validation during parsing, separate validation pass, or deferred validation
- **Decision:** Separate validation step immediately after successful parsing
- **Rationale:** Separates parsing complexity from validation logic, enables reusable validation for serialized frames, and provides clear error attribution
- **Consequences:** Additional function call overhead but clearer separation of concerns and better testability

The validation process encompasses multiple categories of checks:

Length Validation

Frame length validation prevents buffer overflow attacks and ensures frames conform to negotiated size limits:

Validation	Rule	Error Response
Maximum frame size	Length \leq MAX_FRAME_SIZE setting	FRAME_SIZE_ERROR
Minimum frame size	Length meets frame type requirements	FRAME_SIZE_ERROR
Setting frame size	SETTINGS frames are multiples of 6 bytes	FRAME_SIZE_ERROR
Ping frame size	PING frames are exactly 8 bytes	FRAME_SIZE_ERROR
Window update size	WINDOW_UPDATE frames are exactly 4 bytes	FRAME_SIZE_ERROR

Stream ID Validation

Stream ID validation ensures proper stream lifecycle and prevents protocol confusion:

Validation	Rule	Error Response
Reserved bit	Most significant bit of stream ID is 0	PROTOCOL_ERROR
Connection frames	SETTINGS, PING, GOAWAY have stream ID 0	PROTOCOL_ERROR
Stream frames	DATA, HEADERS have non-zero stream ID	PROTOCOL_ERROR
ID assignment	Client uses odd IDs, server uses even IDs	PROTOCOL_ERROR
ID ordering	New stream IDs are greater than previous	PROTOCOL_ERROR

Flag Validation

Flag validation ensures flags are appropriate for each frame type and combination:

Frame Type	Valid Flags	Invalid Combinations
DATA	END_STREAM, PADDED	None
HEADERS	END_STREAM, END_HEADERS, PADDED, PRIORITY	None
SETTINGS	ACK	ACK with non-zero payload length
PING	ACK	ACK on stream ID \neq 0
GOAWAY	None	Any flags set
WINDOW_UPDATE	None	Any flags set

Payload Structure Validation

Each frame type requires specific payload structure validation:

1. **DATA frames**: If PADDED flag is set, validate that padding length is less than payload length
2. **HEADERS frames**: Validate that priority information (if present) doesn't create circular dependencies
3. **SETTINGS frames**: Validate that unknown setting IDs use appropriate ranges and known settings have valid values
4. **PING frames**: Validate payload is exactly 8 bytes regardless of flags
5. **GOAWAY frames**: Validate that last stream ID is within reasonable bounds
6. **WINDOW_UPDATE frames**: Validate that increment is non-zero and doesn't cause window overflow

Connection State Validation

Frame validation must consider the current connection state:

State Check	Validation	Error Response
Connection preface	First frame after connection must be SETTINGS	PROTOCOL_ERROR
GOAWAY sent	No new streams after GOAWAY	PROTOCOL_ERROR
Settings acknowledgment	SETTINGS ACK must match pending SETTINGS	PROTOCOL_ERROR
Stream state	Frame type must be valid for stream's current state	STREAM_CLOSED

Error Response Strategy

When validation fails, the response depends on the error scope:

Connection Errors (affect entire connection):

- Send GOAWAY frame with appropriate error code
- Close the underlying TCP connection
- Clean up all associated streams and resources

Stream Errors (affect single stream):

- Send RST_STREAM frame with appropriate error code
- Transition stream to closed state
- Continue processing other streams normally

Common Pitfalls

⚠ Pitfall: Incorrect Endianness Handling Many implementations incorrectly handle the 24-bit length field by treating it as a little-endian value instead of big-endian. This causes frame parsing to completely fail when communicating with standard HTTP/2 implementations. Always use network byte order (big-endian) conversion functions and test with known good HTTP/2 clients.

⚠ Pitfall: Reserved Bit Ignorance Ignoring the reserved bit in the stream ID field creates a subtle but serious protocol violation. Implementations that fail to mask off this bit may interpret stream IDs incorrectly,

leading to frames being routed to wrong streams. Always apply a bitwise AND with `0x7FFFFFFF` to clear the reserved bit.

⚠ Pitfall: Partial Read Assumptions Assuming that socket read operations always return the requested number of bytes leads to frame corruption and parsing failures. TCP provides no such guarantee, especially under high load or with slow connections. Always check return values and accumulate bytes until the complete frame is received.

⚠ Pitfall: Frame Size Attack Vulnerability Accepting frames with extremely large length values without validation enables memory exhaustion attacks. An attacker can send a frame claiming to be 16MB but only provide a few bytes, causing the parser to allocate huge buffers. Always validate frame length against negotiated limits and read payload incrementally.

⚠ Pitfall: Flag Combination Blindness Processing frames without validating flag combinations can lead to undefined behavior. For example, a SETTINGS frame with the ACK flag must have zero payload length, but implementations often skip this check. Invalid flag combinations indicate either implementation bugs or protocol attacks.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Binary Parsing	<code>encoding/binary</code> package	Custom bit manipulation with unsafe operations
Buffer Management	<code>bufio.Reader/Writer</code>	Pooled byte buffers with <code>sync.Pool</code>
Validation	Inline error checking	Validation state machine with comprehensive rules
Testing	Standard unit tests	Property-based testing with random frame generation

Recommended File Structure

```
internal/frame/
  frame.go          ← Frame structure and basic operations
  parser.go         ← Frame parsing logic
  validator.go      ← Frame validation rules
  serializer.go     ← Frame serialization logic
  types.go          ← Frame type constants and enums
  frame_test.go     ← Comprehensive frame parsing tests
  validator_test.go ← Validation rule tests
```

Infrastructure Starter Code

```
package frame

import (
    "bufio"
    "encoding/binary"
    "fmt"
    "io"
    "net"
    "sync"
)

// Frame represents a complete HTTP/2 frame with header and payload

type Frame struct {
    Length    uint32
    Type      uint8
    Flags     uint8
    StreamID uint32
    Payload   []byte
}

// Connection wraps a network connection with buffered I/O

type Connection struct {
    conn    net.Conn
    reader *bufio.Reader
    writer *bufio.Writer
    mu     sync.RWMutex
    closed bool
}
```

GO

```
}

// ProtocolError represents an HTTP/2 protocol violation

type ProtocolError struct {

    Code     ErrorCode

    Message string

}

func (e *ProtocolError) Error() string {

    return fmt.Sprintf("HTTP/2 protocol error %d: %s", e.Code, e.Message)

}

// ErrorCode represents HTTP/2 error codes

type ErrorCode uint32

// Frame type constants

const (

    FrameTypeDATA          = 0x0

    FrameTypeHEADERS       = 0x1

    FrameTypePRIORITY      = 0x2

    FrameTypeRST_STREAM    = 0x3

    FrameTypeSETTINGS       = 0x4

    FrameTypePUSH_PROMISE  = 0x5

    FrameTypePING           = 0x6

    FrameTypeGOAWAY          = 0x7

    FrameTypeWINDOW_UPDATE   = 0x8

    FrameTypeCONTINUATION    = 0x9

)
```

```
// Frame flag constants

const (
    FlagEndStream      = 0x1
    FlagAck            = 0x1
    FlagEndHeaders     = 0x4
    FlagPadded         = 0x8
    FlagPriority       = 0x20
)

// Protocol constants

const (
    MaxFrameSize       = 16777215 // 2^24 - 1
    DefaultFrameSize   = 16384   // 16KB
    InitialWindowSize  = 65535   // 64KB - 1
)

// Error codes

const (
    ErrorCode.NoError           ErrorCode = 0x0
    ErrorCode.ProtocolError     ErrorCode = 0x1
    ErrorCode.InternalError     ErrorCode = 0x2
    ErrorCode.FlowControlError  ErrorCode = 0x3
    ErrorCode.SettingsTimeout   ErrorCode = 0x4
    ErrorCode.StreamClosed      ErrorCode = 0x5
    ErrorCode.FrameSizeError   ErrorCode = 0x6
    ErrorCode.RefusedStream    ErrorCode = 0x7
    ErrorCode.Cancel            ErrorCode = 0x8
    ErrorCode.CompressionError ErrorCode = 0x9
)
```

```
ErrorCodeConnectError      ErrorCode = 0xa
ErrorCodeEnhanceYourCalm  ErrorCode = 0xb
ErrorCodeInadequateSecurity ErrorCode = 0xc
ErrorCodeHTTP11Required    ErrorCode = 0xd
)

// NewConnection creates a new buffered connection wrapper

func NewConnection(conn net.Conn) *Connection {
    return &Connection{
        conn:    conn,
        reader:  bufio.NewReader(conn),
        writer:  bufio.NewWriter(conn),
    }
}

// NewProtocolError creates a protocol error with code and message

func NewProtocolError(code ErrorCode, message string) *ProtocolError {
    return &ProtocolError{
        Code:    code,
        Message: message,
    }
}
```

Core Logic Skeleton Code

```
// ParseFrame reads and parses HTTP/2 frame from reader
```

```
func ParseFrame(r io.Reader) (*Frame, error) {
```

```
    // TODO 1: Read exactly 9 bytes for frame header, handle partial reads
```

```
    // TODO 2: Extract 24-bit length from first 3 bytes using big-endian
```

```
    // TODO 3: Extract type from byte 4, flags from byte 5
```

```
    // TODO 4: Extract 31-bit stream ID from bytes 6-9, mask reserved bit
```

```
    // TODO 5: Validate header fields (length, type, stream ID)
```

```
    // TODO 6: Read payload based on length field, handle partial reads
```

```
    // TODO 7: Create and return Frame struct with all fields populated
```

```
    // Hint: Use encoding/binary.BigEndian for multi-byte field extraction
```

```
    // Hint: Handle io.EOF and io.ErrUnexpectedEOF for connection closure
```

```
}
```



```
// SerializeFrame writes frame in HTTP/2 wire format
```

```
func SerializeFrame(w io.Writer, f *Frame) (int, error) {
```

```
    // TODO 1: Validate frame before serialization
```

```
    // TODO 2: Create 9-byte header buffer
```

```
    // TODO 3: Write 24-bit length in big-endian format
```

```
    // TODO 4: Write type and flags bytes
```

```
    // TODO 5: Write 31-bit stream ID in big-endian, ensure reserved bit is 0
```

```
    // TODO 6: Write header to output writer
```

```
    // TODO 7: Write payload bytes to output writer
```

```
    // TODO 8: Return total bytes written
```

```
    // Hint: Use make([]byte, 9) for header buffer
```

```
    // Hint: Use encoding/binary.BigEndian.PutUint32 for stream ID
```

```
}
```

GO

```
// ValidateFrame checks frame protocol compliance

func ValidateFrame(f *Frame) error {

    // TODO 1: Validate frame length against MaxFrameSize and type-specific limits

    // TODO 2: Validate stream ID reserved bit is 0

    // TODO 3: Validate frame type is known/supported

    // TODO 4: Validate flags are appropriate for frame type

    // TODO 5: Validate stream ID is appropriate for frame type (0 for connection frames)

    // TODO 6: Perform type-specific payload structure validation

    // TODO 7: Return appropriate ProtocolError for violations

    // Hint: Use switch statement for frame type specific validation

    // Hint: Check f.StreamID & 0x80000000 == 0 for reserved bit

}

// validateDataFrame performs DATA frame specific validation

func validateDataFrame(f *Frame) error {

    // TODO 1: If PADDED flag set, ensure payload has at least 1 byte for pad length

    // TODO 2: If PADDED flag set, validate pad length < remaining payload

    // TODO 3: Validate stream ID is non-zero (DATA frames must target a stream)

    // TODO 4: Return FRAME_SIZE_ERROR for invalid padding

}

// validateHeadersFrame performs HEADERS frame specific validation

func validateHeadersFrame(f *Frame) error {

    // TODO 1: If PADDED flag set, validate padding length field and value

    // TODO 2: If PRIORITY flag set, validate 5-byte priority information exists

    // TODO 3: If PRIORITY flag set, validate stream dependency doesn't create cycles

    // TODO 4: Validate stream ID is non-zero

    // TODO 5: Validate header block fragment length after accounting for other fields
```

```
}

// validateSettingsFrame performs SETTINGS frame specific validation

func validateSettingsFrame(f *Frame) error {

    // TODO 1: Validate stream ID is 0 (SETTINGS are connection-level)

    // TODO 2: If ACK flag set, validate payload length is 0

    // TODO 3: If ACK flag not set, validate payload length is multiple of 6

    // TODO 4: Parse setting parameters and validate known setting values

    // TODO 5: Return FRAME_SIZE_ERROR for incorrect payload length

}
```

Language-Specific Hints

Go-Specific Implementation Tips:

- Use `encoding/binary.BigEndian.Uint32()` and `Uint16()` for multi-byte field extraction
- Handle `io.EOF` and `io.ErrUnexpectedEOF` to detect connection closure during frame parsing
- Use `bufio.Reader.ReadFull()` to ensure complete frame reads despite TCP fragmentation
- Implement `io.WriterTo` interface on Frame type for efficient serialization
- Use `sync.Pool` for frame buffer reuse in high-performance scenarios
- Consider using `unsafe` package for zero-copy payload access (advanced optimization)

Error Handling Patterns:

- Distinguish between connection errors (send GOAWAY) and stream errors (send RST_STREAM)
- Always validate frames before processing to prevent corruption propagation
- Log protocol violations with sufficient detail for debugging but avoid information leakage
- Implement connection-level error recovery by gracefully closing connections on fatal errors

Milestone Checkpoint

After implementing the binary framing engine, verify the following behavior:

Unit Test Verification:

```
go test ./internal/frame/... -v
```

BASH

Expected output should show all frame parsing, validation, and serialization tests passing.

Manual Testing Commands:

```
# Test frame parsing with known good frames  
echo -n "\x00\x00\x00\x04\x00\x00\x00\x00\x00\x00" | go run cmd/frame-test/main.go parse  
  
# Test frame serialization roundtrip  
go run cmd/frame-test/main.go roundtrip --type=SETTINGS --flags=0 --stream-id=0
```

BASH

Expected Behavior Verification:

1. **Frame Header Parsing:** Parser correctly extracts length=0, type=SETTINGS, flags=0, stream-id=0 from the test frame above
2. **Endianness Handling:** Multi-byte fields display correct values when parsed (not byte-swapped)
3. **Validation Errors:** Invalid frames (wrong length, reserved bits set) trigger appropriate protocol errors
4. **Serialization Roundtrip:** Frame → bytes → Frame produces identical frame structures

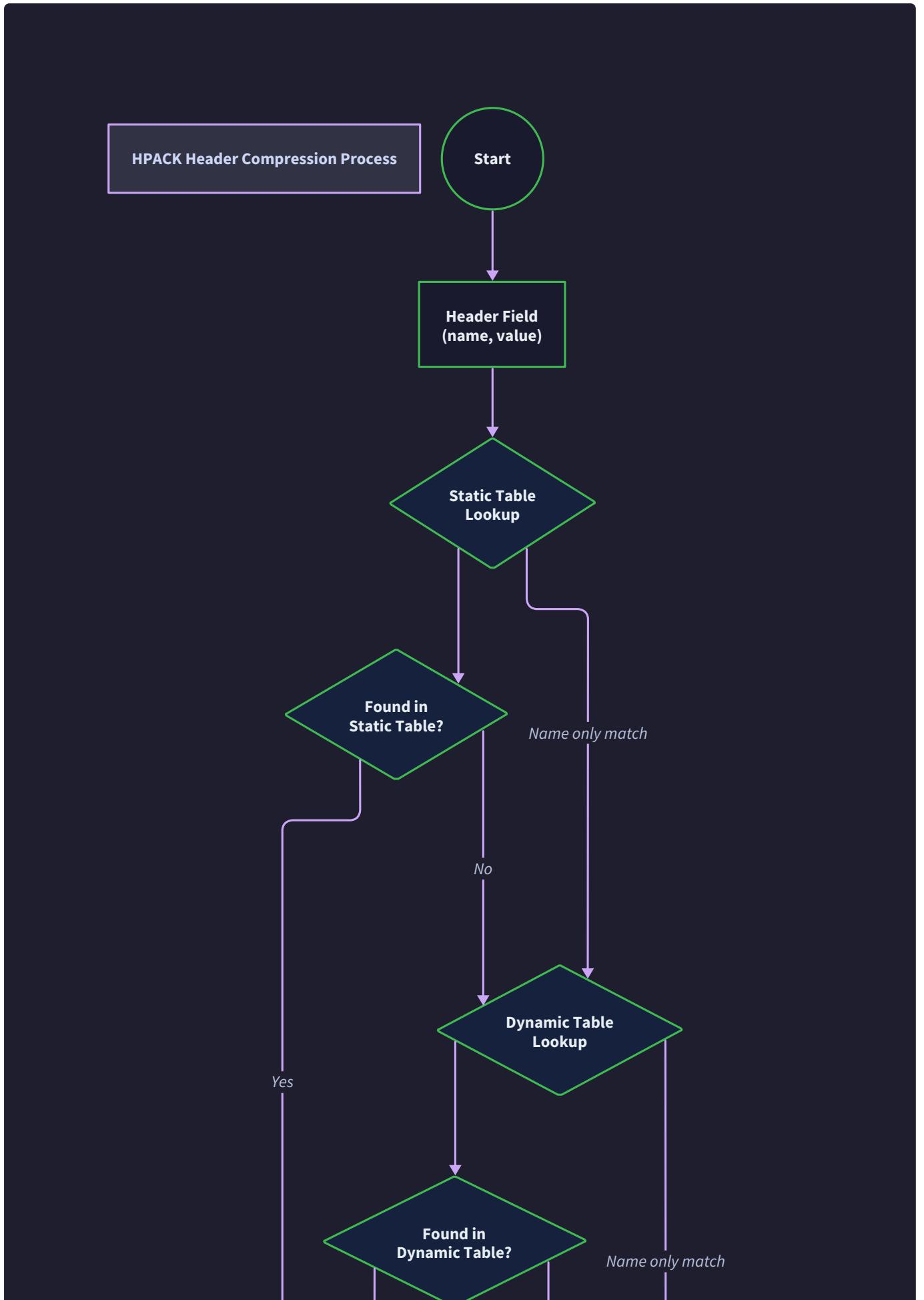
Warning Signs to Investigate:

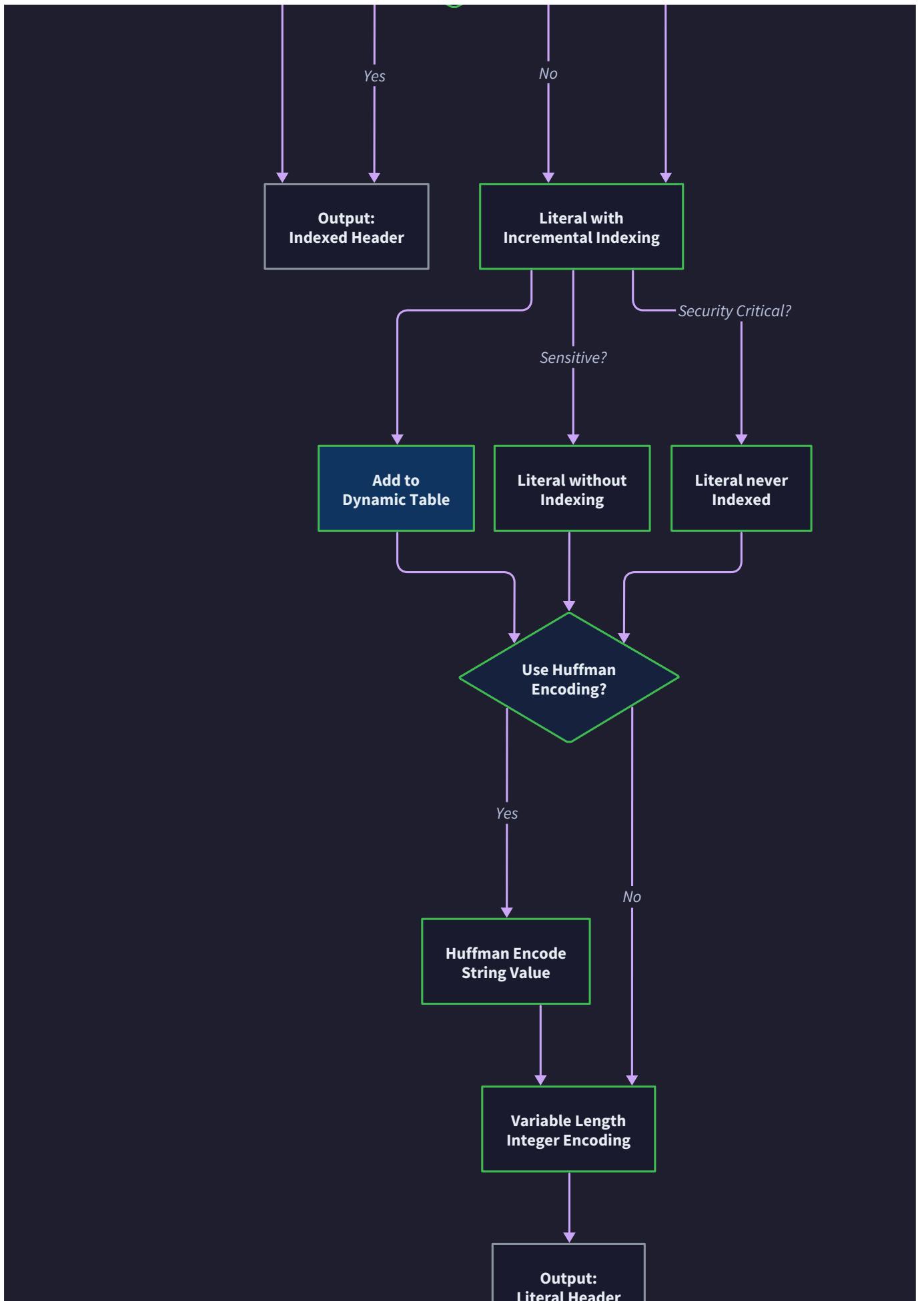
- Frame lengths appearing as extremely large values (likely endianness bug)
- Stream IDs with high bit set (reserved bit not masked)
- Validation accepting clearly invalid frames (validation logic missing)
- Parsing hanging indefinitely (partial read handling broken)

HPACK Compression Engine

Milestone(s): This section primarily addresses Milestone 2 (HPACK Compression) by implementing RFC 7541 header compression with static/dynamic tables, Huffman coding, and integer encoding capabilities.

The HPACK compression engine serves as HTTP/2's sophisticated header compression system, fundamentally transforming how headers are transmitted over the wire. Think of HPACK as an intelligent librarian system for HTTP headers. Instead of repeatedly writing out the full title of every book (header name-value pair), the librarian maintains a catalog system with shortcuts. Popular books get permanent catalog numbers (static table), frequently accessed books in the current session get temporary numbers (dynamic table), and book titles are written in a compressed shorthand notation (Huffman coding). This systematic approach dramatically reduces the overhead of header transmission while maintaining complete fidelity.





The HPACK engine operates through four interconnected mechanisms: static table lookups for predefined common headers, dynamic table management for connection-specific patterns, Huffman decoding for compressed string values, and variable-length integer encoding for efficient numeric representation. Each mechanism addresses specific inefficiencies in raw header transmission while working together to achieve optimal compression ratios.

Static Table

The static table represents HPACK's foundation - a predefined catalog of 61 common HTTP header name-value pairs that every HTTP/2 implementation must recognize. Think of this as a universal dictionary shared between all HTTP/2 endpoints, similar to how postal abbreviations (like "CA" for California) are universally understood and save space on addresses.

The static table contains entries ranging from index 1 to 61, with each entry representing either a complete name-value pair or just a header name. For example, index 2 represents the complete pair `:method GET`, while index 1 represents just the name `:authority` without a predetermined value. This dual approach maximizes compression opportunities - common complete pairs like `GET` methods can be represented with a single index, while common header names with variable values can leverage the name portion while encoding only the value separately.

Index	Name	Value	Usage Pattern
1	<code>:authority</code>	(empty)	Host header equivalent
2	<code>:method</code>	<code>GET</code>	Most common HTTP method
3	<code>:method</code>	<code>POST</code>	Second most common method
4	<code>:path</code>	<code>/</code>	Root path requests
8	<code>:status</code>	<code>200</code>	Successful responses
14	<code>accept-encoding</code>	<code>gzip, deflate</code>	Common compression preferences
31	<code>user-agent</code>	(empty)	Browser identification header

The static table lookup mechanism operates through direct index resolution. When the HPACK decoder encounters an indexed header field representation (identified by the first bit being 1), it extracts the index value and performs a direct array lookup. Static table entries occupy indices 1-61, making the lookup operation extremely efficient with O(1) complexity.

Design Insight: The static table's contents were carefully chosen based on analysis of real-world HTTP traffic patterns. The selection prioritizes headers that appear in virtually every HTTP request or response, ensuring maximum compression benefit across diverse web applications.

Static Table Initialization Algorithm:

1. Create a fixed-size array with 61 entries, indexed from 1 to 61
2. Populate entries 1-4 with HTTP/2 pseudo-headers (`:authority`, `:method GET`, `:method POST`, `:path /`)
3. Add status codes 5-14 covering common response statuses (200, 204, 206, 304, 400, 404, 500)
4. Include standard request headers 15-61 (accept, accept-encoding, cache-control, content-type, etc.)
5. Mark name-only entries (where value is implementation-specific) versus complete name-value pairs
6. Create index lookup function that validates range [1,61] and returns `HeaderEntry` or not-found indicator

The static table's immutability provides several architectural advantages. First, it eliminates synchronization concerns since no modifications occur after initialization. Second, it enables aggressive caching strategies where implementations can pre-compute lookups or use optimized data structures. Third, it ensures perfect interoperability since all implementations share identical static table contents.

Static Table Access Patterns:

Operation	Time Complexity	Space Complexity	Usage Frequency
Index Lookup	O(1)	O(1)	Every indexed header field
Reverse Lookup	O(n) or O(1) with hash map	O(n) additional space	During header encoding
Initialization	O(1)	O(1)	Once per implementation
Range Validation	O(1)	O(1)	Every lookup operation

⚠ Pitfall: Off-by-One Index Errors Static table indices start at 1, not 0, following HPACK specification conventions. Implementations often incorrectly use zero-based indexing, causing lookups to return wrong headers or trigger out-of-bounds errors. Always validate that `index >= 1` and `index <= 61` before array access, and remember that `array[index-1]` accesses the correct entry in zero-based language arrays.

Dynamic Table

The dynamic table serves as HPACK's adaptive learning mechanism, functioning like a personalized bookmark collection that evolves based on the specific headers used within a particular HTTP/2 connection. Unlike the static table's universal constants, the dynamic table captures connection-specific patterns, storing recently used header name-value pairs that don't appear in the static table or have connection-specific values.

Think of the dynamic table as a smart cache with a twist - it's not just a typical LRU cache, but rather a FIFO queue with size-based eviction. Imagine a conveyor belt with limited capacity measured in bytes rather than number of items. New items (header entries) are added to one end, and when the belt becomes too full, items fall off the other end regardless of how frequently they were accessed. This FIFO approach ensures deterministic behavior between encoder and decoder, as both maintain identical table state without complex frequency tracking.

Dynamic Table Structure:

Field	Type	Description	Constraints
entries	[]HeaderEntry	Ordered list of header entries	FIFO ordering, newest first
size	uint32	Current total size in octets	Sum of all entry sizes
maxSize	uint32	Maximum allowed size	Configurable via SETTINGS frame
insertionPoint	uint32	Index where next entry will be inserted	Always at position 0 (newest)

Each header entry in the dynamic table carries overhead beyond its name and value strings. The HPACK specification defines that each entry consumes 32 additional octets beyond the UTF-8 byte length of its name and value. This overhead accounts for implementation data structures and ensures consistent size calculations across different HTTP/2 implementations.

Decision: FIFO Eviction Strategy

- **Context:** Dynamic table needs predictable eviction behavior that both encoder and decoder can maintain synchronously
- **Options Considered:**
 1. LRU (Least Recently Used) eviction based on access patterns
 2. FIFO (First In, First Out) eviction based on insertion order
 3. Size-based priority eviction based on entry sizes
- **Decision:** FIFO eviction with size-based capacity management
- **Rationale:** FIFO ensures deterministic behavior without complex state synchronization. Both encoder and decoder can maintain identical table state by following the same insertion and eviction rules, regardless of access patterns.
- **Consequences:** Enables simple implementation and perfect encoder-decoder synchronization, but may evict frequently accessed entries that an LRU cache would retain.

Dynamic Table Management Operations:

Operation	Parameters	Returns	Side Effects
Add	name string, value string	none	Inserts entry, triggers eviction if needed
Lookup	index uint32	HeaderEntry, bool	No state changes
Resize	newSize uint32	error	May trigger eviction, updates maxSize
EvictEntries	none	int (evicted count)	Removes oldest entries until size <= maxSize
CalculateSize	name, value string	uint32	Returns total entry size including overhead

The insertion algorithm follows a precise sequence to maintain table consistency. When adding a new entry, the dynamic table first calculates the total size including the 32-octet overhead. If this addition would exceed the maximum table size, it performs preemptive eviction of the oldest entries until sufficient space exists. The new entry is then inserted at index 0 (the "newest" position), and all existing entries shift to higher indices.

Dynamic Table Insertion Algorithm:

1. Calculate new entry size: `len(name) + len(value) + 32`
2. Check if entry size exceeds maximum table size - if so, reject insertion
3. While `currentSize + newEntrySize > maxSize`, evict oldest entry (highest index)
4. Create new `HeaderEntry` with provided name, value, and calculated size
5. Insert entry at index 0, shifting all existing entries to higher indices
6. Update total table size by adding new entry size
7. Update internal bookkeeping (entry count, insertion statistics)

The dynamic table indexing scheme requires careful coordination with static table indices. Dynamic table entries occupy indices starting at 62 (immediately after the static table's highest index of 61). The dynamic table's internal index 0 (newest entry) maps to HPACK index 62, internal index 1 maps to HPACK index 63, and so forth. This creates a unified index space where indices 1-61 reference static entries and indices 62+ reference dynamic entries.

Index Mapping Between Tables:

HPACK Index Range	Table Type	Internal Index Calculation	Access Pattern
1-61	Static Table	<code>staticIndex = hpackIndex</code>	Direct array access
62+	Dynamic Table	<code>dynamicIndex = hpackIndex - 62</code>	Array access with bounds check
0	Invalid	N/A	Reserved/error condition

Dynamic table resizing occurs through `SETTINGS_HEADER_TABLE_SIZE` frames, which can arrive at any time during connection lifetime. The resize operation may require immediate eviction if the new size is smaller than current table contents. The eviction process follows the same FIFO principle - oldest entries are removed first until the table size complies with the new limit.

Design Insight: The 32-octet overhead per entry serves dual purposes: it accounts for typical implementation overhead (pointers, metadata) and provides a buffer against implementation variations. Without this standardized overhead, different implementations might calculate table sizes differently, leading to synchronization drift between encoder and decoder.

Dynamic Table Eviction Scenarios:

Trigger	Condition	Action Taken	Recovery Behavior
Size Overflow	<code>currentSize + newEntry > maxSize</code>	Evict oldest until space available	Normal insertion proceeds
Table Resize	New <code>maxSize < currentSize</code>	Evict oldest until compliant	Table remains functional
Connection Reset	New connection or settings	Clear all entries, reset size to 0	Rebuild from empty state
Memory Pressure	Implementation-specific	May reduce <code>maxSize</code> temporarily	Graceful degradation

⚠️ Pitfall: Dynamic Table Size Calculation Implementations frequently miscalculate entry sizes by forgetting the 32-octet overhead or incorrectly measuring string lengths. UTF-8 strings must be measured in bytes (octets), not Unicode code points or characters. Always use `len([]byte(string))` in Go or equivalent byte-length functions in other languages. The total size is `len(nameBytes) + len(valueBytes) + 32`, and this exact calculation must match between encoder and decoder.

Huffman Coding

Huffman coding in HPACK provides the final layer of compression by encoding string values using variable-length bit sequences optimized for typical HTTP header content. Think of Huffman coding as a telegraph operator's shorthand system - frequently used letters get short codes (like "E" encoded as a single dot), while rare letters get longer codes. This approach can reduce header string sizes by 15-50% depending on content patterns.

The HPACK specification defines a static Huffman table optimized for HTTP header values, based on analysis of real-world web traffic. This table assigns shorter bit sequences to characters that appear frequently in HTTP headers (like lowercase letters, digits, and common punctuation) while assigning longer sequences to rarely used characters. The encoding table is bidirectional - the same table supports both encoding strings into compressed bit sequences and decoding compressed sequences back to original strings.

HPACK Huffman Table Characteristics:

Character Category	Code Length Range	Example Characters	Optimization Rationale
Common lowercase	5-6 bits	a, e, i, o, s, t	High frequency in header values
Digits	5-6 bits	0, 1, 2, 3, 4-9	Common in content-length, status
Common symbols	6-8 bits	-, _, ., /	Frequent in URLs, header names
Uppercase letters	6-8 bits	A, C, G, M, P	Less common but still significant
Special characters	10-30 bits	Unicode, control chars	Rare, acceptable longer encoding

The Huffman decoding process operates on a bit-by-bit basis, using the compressed byte stream to traverse a decoding tree or lookup table. The decoder maintains a state machine that accumulates bits until it identifies a complete character code, then outputs the corresponding character and resets for the next code. This stateful approach handles variable-length codes seamlessly while detecting invalid or incomplete sequences.

Huffman Decoding Algorithm:

1. Initialize decoder state: bit accumulator = 0, bit count = 0, output buffer empty
2. For each byte in compressed input:
 - Extract 8 bits and add to bit accumulator
 - Increment bit count by 8
 - While bit count >= minimum code length:
 - Check if current bit pattern matches any Huffman code
 - If match found: append decoded character to output, remove matched bits from accumulator, adjust bit count
 - If no match and bit count >= maximum code length: signal decoding error
 - If no match and more bits needed: continue to next input byte

3. After processing all input bytes:

- If remaining bits are non-zero: verify they match EOS padding pattern
- If padding invalid: signal decoding error
- Return decoded string from output buffer

The Huffman decoder must handle several edge cases carefully. Incomplete input streams may leave partial codes in the bit accumulator - these must be validated as proper End-of-String (EOS) padding to prevent security vulnerabilities. Invalid bit sequences that don't correspond to any character code must trigger decoding errors rather than producing garbage output.

Design Insight: HPACK uses a static Huffman table rather than dynamic Huffman coding to avoid synchronization complexity between encoder and decoder. While dynamic tables could achieve better compression ratios by adapting to specific traffic patterns, the static approach eliminates the need to maintain and synchronize Huffman trees across connection endpoints.

Huffman Decoding State Machine:

Current State	Input Event	Next State	Action Taken
ACCUMULATING	Input byte available	ACCUMULATING	Add 8 bits to accumulator
ACCUMULATING	Pattern matches code	ACCUMULATING	Output character, remove matched bits
ACCUMULATING	End of input, valid padding	COMPLETE	Return decoded string
ACCUMULATING	End of input, invalid padding	ERROR	Signal padding error
ACCUMULATING	Invalid bit pattern	ERROR	Signal decoding error
ERROR	Any input	ERROR	Maintain error state
COMPLETE	Any input	ERROR	Decoder already finished

Huffman boundary handling represents one of the most complex aspects of HPACK implementation. Since Huffman codes have variable lengths and don't align with byte boundaries, the final byte of a Huffman-encoded string may contain partial code bits followed by padding. The HPACK specification mandates that padding consists of the most significant bits of the EOS (End of String) symbol, ensuring that partial codes don't accidentally decode to valid characters.

Huffman Padding Validation:

The EOS symbol in HPACK Huffman coding is represented by 30 consecutive 1-bits (`11111111111111111111111111111111`). When a Huffman-encoded string doesn't end exactly on a byte boundary, the remaining bits in the final byte must be filled with the most significant bits of the EOS symbol. For example, if 3 padding bits are needed, they must be `111` (the first 3 bits of the EOS symbol).

Padding Length	Required Bit Pattern	Validation Check
1 bit	1	Final bit must be 1
2 bits	11	Final 2 bits must be 11
3 bits	111	Final 3 bits must be 111
4 bits	1111	Final 4 bits must be 1111
5 bits	11111	Final 5 bits must be 11111
6 bits	111111	Final 6 bits must be 111111
7 bits	1111111	Final 7 bits must be 1111111

⚠️ Pitfall: Huffman Boundary Errors Implementations often fail to properly validate Huffman padding, either accepting invalid padding patterns or miscalculating padding length. The padding validation must check that remaining bits in the final byte exactly match the most significant bits of the EOS symbol. Accepting invalid padding can create security vulnerabilities where malformed headers are processed incorrectly. Always validate that `remainingBits == (EOS_SYMBOL >> (30 - paddingLength))`.

Integer Encoding

HPACK integer encoding provides a sophisticated variable-length encoding scheme that efficiently represents numeric values while maintaining byte-alignment for the surrounding bit fields. Think of this encoding as a expandable numeric notation system similar to scientific notation, but optimized for binary representation rather than human readability. Small numbers use minimal space, while larger numbers automatically extend into additional bytes as needed.

The encoding scheme uses a prefix of N bits within the first byte to represent small integers directly, with the remaining (8-N) bits available for other purposes like flags or type indicators. When the integer value exceeds what the N-bit prefix can represent ($2^N - 1$), the encoding switches to a multi-byte format that can represent arbitrarily large integers using continuation bytes.

HPACK Integer Encoding Variants:

Prefix Size	Maximum Single-Byte Value	Typical Usage	Remaining Bits Available
5-bit prefix	30 ($2^5 - 2$)	Header field index	3 bits for pattern/flags
6-bit prefix	62 ($2^6 - 2$)	Maximum dynamic table size	2 bits for pattern/flags
7-bit prefix	126 ($2^7 - 2$)	String length encoding	1 bit for Huffman flag
8-bit prefix	254 ($2^8 - 2$)	Standalone integer values	0 bits (full byte)

The integer encoding algorithm operates in two distinct modes based on the value being encoded. For small integers that fit within the prefix bits, the encoding uses a single byte with the integer value stored directly in the prefix portion. For larger integers, the prefix bits are filled with all 1s (indicating multi-byte encoding), and the actual value is encoded using a continuation-byte scheme similar to UTF-8 encoding.

Single-Byte Integer Encoding (Value < $2^N - 1$):

1. Verify that integer value $< (2^N - 1)$ where N is the prefix size
2. Store the integer value directly in the N least significant bits of the first byte
3. Set the (8-N) most significant bits according to the specific HPACK pattern
4. Output the single byte - encoding complete

Multi-Byte Integer Encoding (Value $\geq 2^N - 1$):

1. Set the N prefix bits to all 1s: `($2^N - 1$)`
2. Calculate remaining value: `remainingValue = originalValue - ($2^N - 1$)`
3. While `remainingValue >= 128`:
 - Output byte with value `(remainingValue % 128) | 0x80` (continuation bit set)
 - Set `remainingValue = remainingValue / 128`
4. Output final byte with value `remainingValue` (continuation bit clear)
5. Encoding complete

The decoding process reverses this algorithm, first checking if the prefix bits are all 1s to determine single-byte versus multi-byte mode. In multi-byte mode, the decoder accumulates value from continuation bytes, each contributing 7 bits of value data (the 8th bit indicates whether more bytes follow).

Integer Decoding Algorithm:

1. Extract N prefix bits from first byte: `prefixValue = firstByte & ((1 << N) - 1)`
2. If `prefixValue < ($2^N - 1$)`: return `prefixValue` (single-byte encoding)
3. Initialize accumulator: `totalValue = prefixValue`, `multiplier = 1`
4. Read next byte, check continuation bit (bit 7)
5. Add contribution: `totalValue += (byte & 0x7F) * multiplier`
6. Update multiplier: `multiplier *= 128`
7. If continuation bit set: repeat from step 4
8. Return `totalValue`

Decision: Variable-Length Integer Encoding

- **Context:** HPACK needs to encode integers of varying sizes (indices, lengths, sizes) while maintaining bit-level efficiency for flags and patterns
- **Options Considered:**
 1. Fixed 32-bit integers for all numeric values
 2. LEB128 encoding (Little Endian Base 128) used in other binary formats
 3. Custom variable-length encoding with configurable prefix sizes
- **Decision:** Custom variable-length encoding with N-bit prefixes
- **Rationale:** Allows optimal space usage for small values while supporting arbitrary-size integers. The configurable prefix size enables sharing bytes between integer values and control bits/flags.
- **Consequences:** More complex implementation than fixed-size integers, but significantly better compression for typical HTTP/2 header values.

Integer Encoding Edge Cases:

Scenario	Value Range	Encoding Behavior	Validation Required
Zero value	0	Single byte: prefix = 0	Always valid
Maximum prefix	$2^N - 2$	Single byte: prefix = max-1	Check prefix size
Prefix boundary	$2^N - 1$	Multi-byte: prefix = all 1s, continuation = 0	Special case handling
Large integers	$> 2^{32}$	Multi-byte with many continuation bytes	Overflow protection
Invalid continuation	Bit 7 pattern errors	Decoder should reject	Error detection

The integer encoding format includes several subtle implementation requirements. The continuation bit (most significant bit of each continuation byte) must be properly set for all bytes except the final one. The value contribution from each continuation byte uses only the lower 7 bits, with the 8th bit serving as the continuation indicator. Overflow protection is essential since malicious inputs could specify extremely large integers that exceed implementation limits.

⚠ Pitfall: Integer Overflow in Multi-Byte Decoding Malicious or malformed HPACK data might encode extremely large integers using many continuation bytes, potentially causing integer overflow in the decoder. Implementations must limit the maximum number of continuation bytes (typically 4-5 bytes for 32-bit integers) and detect overflow conditions. Use `totalValue > (MAX_INT - contribution) / multiplier` to check for overflow before adding each contribution, and reject any integer that would exceed reasonable bounds.

Integer Encoding Performance Characteristics:

Value Range	Byte Count	Encoding Efficiency	Decode Complexity
0 to 2^N-2	1 byte	Optimal (50-87% space savings)	$O(1)$ - single operation
2^N-1 to $2^{N+1}-1$	2 bytes	Good (25-50% space savings)	$O(1)$ - two operations
2^{14} to 2^{21}	3-4 bytes	Moderate (0-25% space savings)	$O(k)$ where k = byte count
$> 2^{21}$	4+ bytes	Poor (negative space savings)	$O(k)$ with overflow checks

Implementation Guidance

The HPACK compression engine requires careful coordination between multiple encoding and decoding subsystems. The implementation approach balances correctness with performance, ensuring that the complex bit-level manipulations remain maintainable while achieving the compression benefits that make HTTP/2 viable.

A. Technology Recommendations:

Component	Simple Option	Advanced Option
Huffman Decoding	Lookup table with linear search	Finite state machine with bit transitions
Integer Encoding	Byte slice manipulation	Bit-level buffer with optimized operations
Dynamic Table	Simple slice with manual eviction	Circular buffer with efficient insertion
Static Table	Array with hardcoded values	Code-generated lookup with reverse mapping

B. Recommended File Structure:

```
internal/hpack/
    hpack.go           ← main HPACK encoder/decoder interface
    static_table.go     ← static table implementation and data
    dynamic_table.go   ← dynamic table with eviction logic
    huffman.go         ← Huffman decoding implementation
    integer.go          ← variable-length integer encoding/decoding
    huffman_table.go   ← generated Huffman lookup table
    hpack_test.go      ← comprehensive test suite
```

C. Infrastructure Starter Code:

GO

```
// Package hpack implements HTTP/2 HPACK header compression per RFC 7541

package hpack

import (
    "errors"
    "fmt"
)

// HeaderEntry represents a header name-value pair with size information

type HeaderEntry struct {
    Name  string
    Value string
    Size  uint32 // Name length + Value length + 32 octets overhead
}

// staticTable contains the 61 predefined HPACK entries per RFC 7541 Appendix B

var staticTable = [61]HeaderEntry{
    {Name: ":authority", Value: "", Size: 42},
    {Name: ":method", Value: "GET", Size: 42},
    {Name: ":method", Value: "POST", Size: 43},
    {Name: ":path", Value: "/", Size: 38},
    {Name: ":path", Value: "/index.html", Size: 49},
    // ... (complete table implementation)
}

// DynamicTable maintains connection-specific header entries with FIFO eviction

type DynamicTable struct {
    entries []HeaderEntry
    size     uint32
}
```

```
    maxSize uint32

}

// NewDynamicTable creates a dynamic table with the specified maximum size

func NewDynamicTable(maxSize uint32) *DynamicTable {
    return &DynamicTable{
        entries: make([]HeaderEntry, 0, 64), // Initial capacity
        size:     0,
        maxSize: maxSize,
    }
}

// calculateEntrySize returns the HPACK size of a header entry (name + value + 32)

func calculateEntrySize(name, value string) uint32 {
    return uint32(len(name) + len(value) + 32)
}

// HuffmanDecoder provides stateful Huffman decoding for HPACK strings

type HuffmanDecoder struct {
    accumulator uint64 // Bit accumulator for partial codes
    bitCount    int     // Number of valid bits in accumulator
}

// IntegerDecoder handles variable-length integer decoding with prefix support

type IntegerDecoder struct {
    prefixBits int     // Number of bits used for integer prefix
    maxBytes   int     // Maximum continuation bytes to prevent overflow
}
```

D. Core Logic Skeleton Code:

```
// Add inserts a header entry into the dynamic table with FIFO eviction

func (dt *DynamicTable) Add(name, value string) {

    // TODO 1: Calculate new entry size including 32-octet overhead

    // TODO 2: Check if single entry exceeds maximum table size - reject if so

    // TODO 3: Evict oldest entries while (currentSize + newSize) > maxSize

    // TODO 4: Create HeaderEntry and insert at index 0 (newest position)

    // TODO 5: Update total table size and internal bookkeeping

    // Hint: Use copy(dt.entries[1:], dt.entries) to shift existing entries

}

// Lookup retrieves a header entry by HPACK index (1-61 static, 62+ dynamic)

func (dt *DynamicTable) Lookup(index uint32) (HeaderEntry, bool) {

    // TODO 1: Validate index is not zero (reserved/invalid)

    // TODO 2: Check if index <= 61 - if so, return from static table

    // TODO 3: Calculate dynamic table index: dynamicIndex = index - 62

    // TODO 4: Validate dynamic index is within current table bounds

    // TODO 5: Return dynamic table entry at calculated index

    // Hint: Static table uses 1-based indexing, dynamic table uses 0-based internally

}

// DecodeHuffman decodes a Huffman-encoded byte slice to a string

func (hd *HuffmanDecoder) DecodeHuffman(encoded []byte) (string, error) {

    // TODO 1: Initialize output buffer and reset decoder state

    // TODO 2: For each input byte, add 8 bits to accumulator

    // TODO 3: While accumulator has enough bits, try to match Huffman codes

    // TODO 4: When code matched, append decoded character and remove matched bits

    // TODO 5: After all input, validate remaining bits match EOS padding

    // TODO 6: Return decoded string or error for invalid sequences
```

```

    // Hint: Use huffmanLookupTable[bitPattern] for code to character mapping
}

// DecodeInteger decodes a variable-length integer with N-bit prefix

func (id *IntegerDecoder) DecodeInteger(reader io.Reader, prefixBits int) (uint32, error) {

    // TODO 1: Read first byte and extract N-bit prefix value

    // TODO 2: If prefix < (2^N - 1), return prefix value (single-byte case)

    // TODO 3: Initialize accumulator with prefix value, multiplier = 1

    // TODO 4: Read continuation bytes while bit 7 is set

    // TODO 5: Add (byte & 0x7F) * multiplier to accumulator

    // TODO 6: Update multiplier *= 128, check for overflow

    // TODO 7: Return final accumulated value

    // Hint: Limit continuation bytes to prevent overflow attacks

}

```

E. Language-Specific Hints:

- Use `binary.BigEndian` for multi-byte integer operations to ensure correct byte order
- Implement Huffman lookup with `map[uint32]rune` for code-to-character mapping
- Use bit manipulation: `value & ((1 << n) - 1)` to extract N least significant bits
- For dynamic table eviction: `copy(slice[1:], slice[0:len-1])` shifts elements efficiently
- Validate UTF-8 strings with `utf8.Valid([]byte(string))` before processing
- Use `strings.Builder` for efficient string concatenation during decoding

F. Milestone Checkpoint:

After implementing the HPACK compression engine, verify the following behavior:

Static Table Verification:

```

go test -v ./internal/hpack -run TestStaticTable

# Expected: All 61 static table entries accessible by index

# Test: Lookup index 2 should return {Name: ":method", Value: "GET"}

# Test: Lookup index 0 should return error (invalid index)

```

BASH

Dynamic Table Verification:

```
go test -v ./internal/hpack -run TestDynamicTable  
  
# Expected: FIFO eviction when table size exceeded  
  
# Test: Add entries until maxSize reached, verify oldest evicted first  
  
# Test: Resize table to smaller size, verify immediate eviction
```

BASH

Huffman Decoding Verification:

```
go test -v ./internal/hpack -run TestHuffman  
  
# Expected: Decode compressed strings back to original values  
  
# Test: Decode "GET" from compressed bytes should return "GET"  
  
# Test: Invalid padding should return decoding error
```

BASH

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Off-by-one index errors	Static table indexing confusion	Check if using 0-based vs 1-based indexing	Use <code>staticTable[index-1]</code> for valid indices 1-61
Dynamic table size drift	Incorrect size calculations	Log entry sizes during add/evict operations	Ensure <code>size = len(name) + len(value) + 32</code>
Huffman decode failures	Invalid padding or lookup errors	Print bit patterns and remaining bits	Validate padding matches EOS symbol prefix
Integer overflow	Malicious large integers	Monitor continuation byte count	Limit to 5 continuation bytes maximum
Table synchronization issues	Encoder/decoder state mismatch	Compare table contents after each operation	Ensure identical add/evict sequence

Stream Management

Milestone(s): This section primarily addresses Milestone 3 (Stream Management) by implementing multiplexed stream lifecycle management, state machines, priority scheduling, and concurrent stream limits for HTTP/2's core multiplexing capabilities.

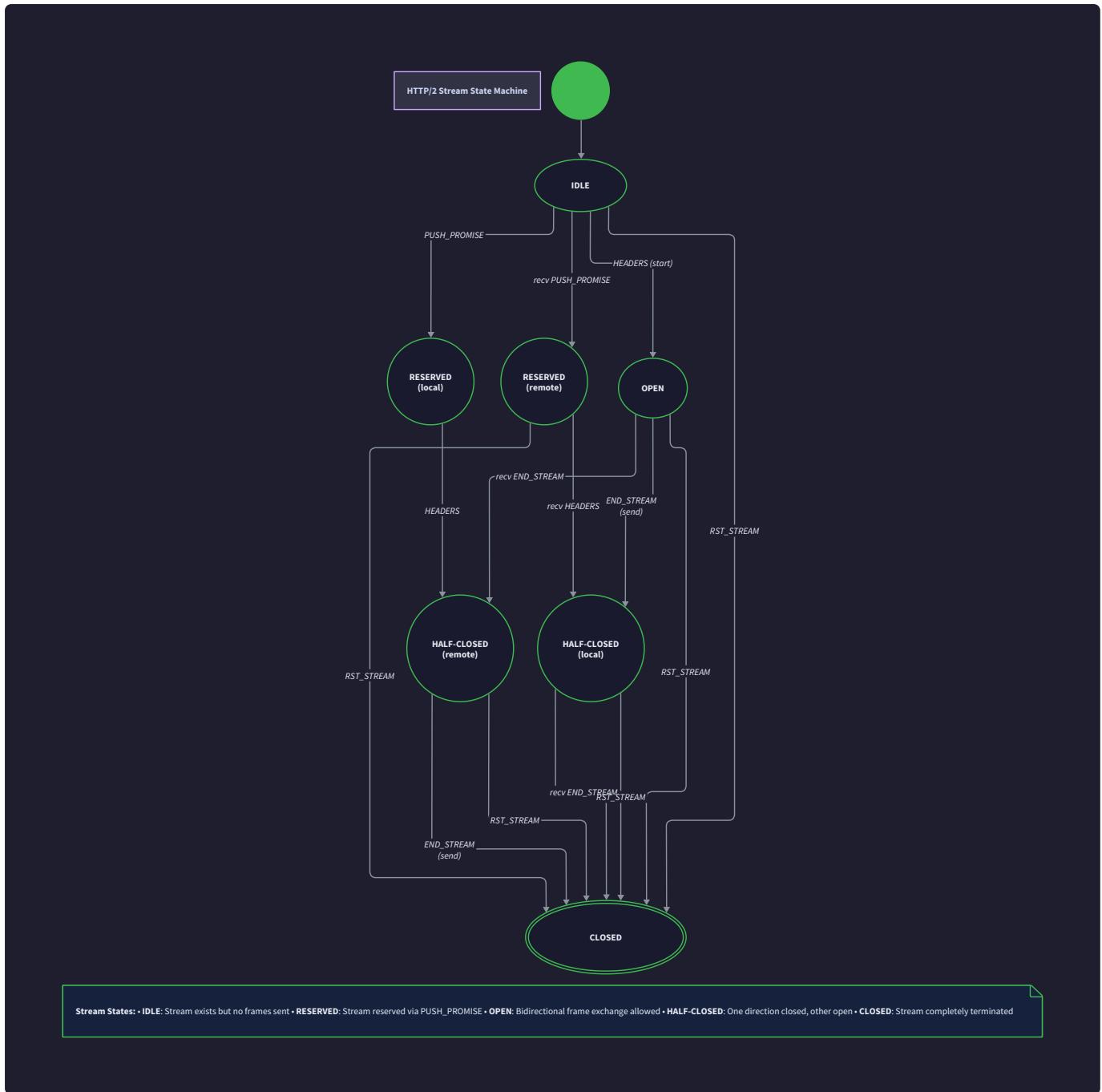
Mental Model: The Air Traffic Control System

Think of HTTP/2 stream management as an air traffic control system for a busy airport. The single TCP connection is like the airport runway - there's only one physical runway, but multiple aircraft (streams) can be managed simultaneously through careful orchestration. Each aircraft has a unique flight number (stream ID), follows a specific flight plan (state machine), and has priority levels (weight and dependencies) that determine landing order. The air traffic controller (stream manager) tracks each aircraft's status, manages the queue when the runway is busy (flow control), and can cancel flights when necessary (RST_STREAM). Just as airports handle dozens of aircraft concurrently over a single runway through sophisticated scheduling, HTTP/2 handles multiple request-response pairs over a single TCP connection through multiplexing.

The key insight is that while HTTP/1.1 was like a single-lane road where cars must wait in line, HTTP/2 is like a multi-lane highway where traffic flows independently in parallel lanes, all sharing the same underlying infrastructure but managed through intelligent routing and priority systems.

Stream State Machine

The **stream state machine** forms the heart of HTTP/2's multiplexing capability, defining how individual streams transition through their lifecycle from creation to completion. Each stream maintains its own independent state, allowing hundreds of concurrent request-response pairs to coexist safely over a single connection without interference.



The state machine governs six distinct states, with transitions triggered by specific frame types and directional considerations. Unlike HTTP/1.1's simple request-response pattern, HTTP/2 streams can be half-closed, meaning one direction (client-to-server or server-to-client) can be closed while the other remains open for additional data.

Current State	Triggering Event	Next State	Actions Taken
StateIdle	Send/Receive HEADERS frame	StateOpen	Initialize stream metadata, allocate buffers
StateIdle	Send PUSH_PROMISE frame	StateReservedLocal	Reserve stream for server push (future extension)
StateIdle	Receive PUSH_PROMISE frame	StateReservedRemote	Acknowledge pushed resource reservation
StateOpen	Send END_STREAM flag	StateHalfClosedLocal	Mark local direction closed, continue receiving
StateOpen	Receive END_STREAM flag	StateHalfClosedRemote	Mark remote direction closed, continue sending
StateHalfClosedLocal	Receive END_STREAM flag	StateClosed	Complete stream closure, release resources
StateHalfClosedRemote	Send END_STREAM flag	StateClosed	Complete stream closure, release resources
StateReservedLocal	Send HEADERS with END_HEADERS	StateHalfClosedRemote	Transition to normal stream flow
StateReservedRemote	Receive HEADERS frame	StateHalfClosedLocal	Accept pushed resource
Any State	Send/Receive RST_STREAM	StateClosed	Immediate termination, clean up resources

The core data structures supporting the state machine provide comprehensive tracking of each stream's current status and directional flow control:

Field Name	Type	Description
<code>id</code>	<code>uint32</code>	Unique stream identifier within connection scope
<code>state</code>	<code>StreamState</code>	Current position in the stream lifecycle state machine
<code>localEndStream</code>	<code>bool</code>	Whether local endpoint has sent END_STREAM flag
<code>remoteEndStream</code>	<code>bool</code>	Whether remote endpoint has sent END_STREAM flag
<code>sendWindow</code>	<code>int64</code>	Available bytes that can be sent to peer (flow control)
<code>recvWindow</code>	<code>int64</code>	Available bytes that can be received from peer
<code>priority</code>	<code>StreamPriority</code>	Weight and dependency information for scheduling
<code>headers</code>	<code>map[string]string</code>	Decoded request/response headers for this stream
<code>pendingData</code>	<code>[]byte</code>	Buffered data waiting for flow control window
<code>created</code>	<code>time.Time</code>	Stream creation timestamp for timeout management
<code>lastActivity</code>	<code>time.Time</code>	Most recent frame received/sent for idle detection

Decision: Separate Local and Remote End Stream Tracking

- **Context:** HTTP/2 streams can be half-closed, where one direction is closed but the other remains open for continued data flow
- **Options Considered:** Single boolean flag, combined state enum, or separate directional flags
- **Decision:** Use separate `localEndStream` and `remoteEndStream` boolean flags
- **Rationale:** Separate flags provide clear visibility into directional state, simplify half-closed logic, and make debugging easier by explicitly showing which direction initiated closure
- **Consequences:** Enables proper half-closed stream handling, increases memory usage slightly per stream, but dramatically simplifies state transition logic

The state transitions must be carefully validated to prevent protocol violations. Invalid transitions indicate either implementation bugs or malicious clients attempting to exploit the protocol:

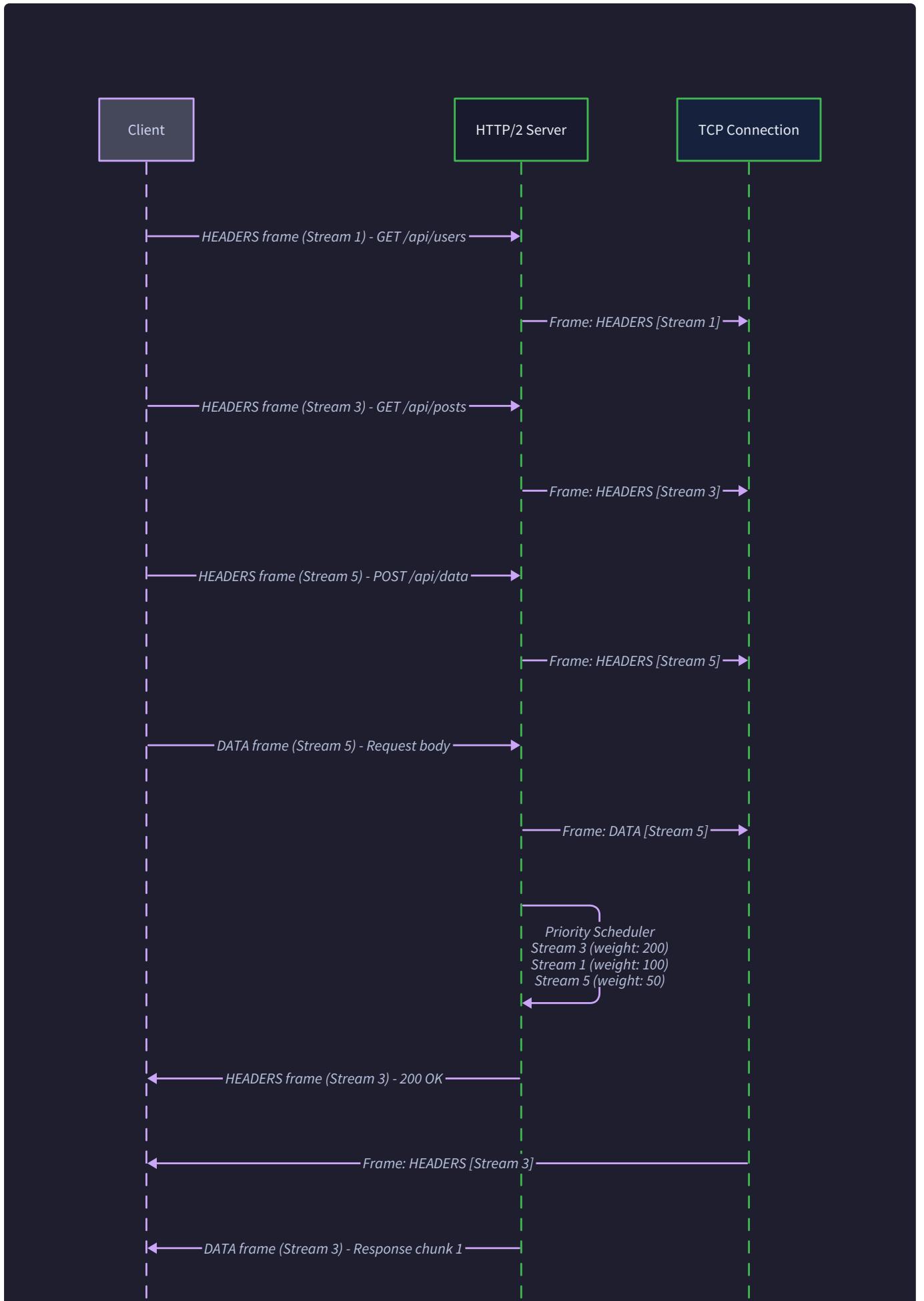
Invalid Transition	Error Condition	Recovery Action
HEADERS on closed stream	Stream already completed	Send RST_STREAM with STREAM_CLOSED error
DATA before HEADERS	Stream not yet opened	Send RST_STREAM with PROTOCOL_ERROR
Multiple HEADERS without CONTINUATION	Header block not complete	Send connection GOAWAY with PROTOCOL_ERROR
END_STREAM on half-closed direction	Direction already closed	Send RST_STREAM with STREAM_CLOSED error
Frame on reserved stream (wrong type)	Invalid frame for reserved state	Send RST_STREAM with PROTOCOL_ERROR

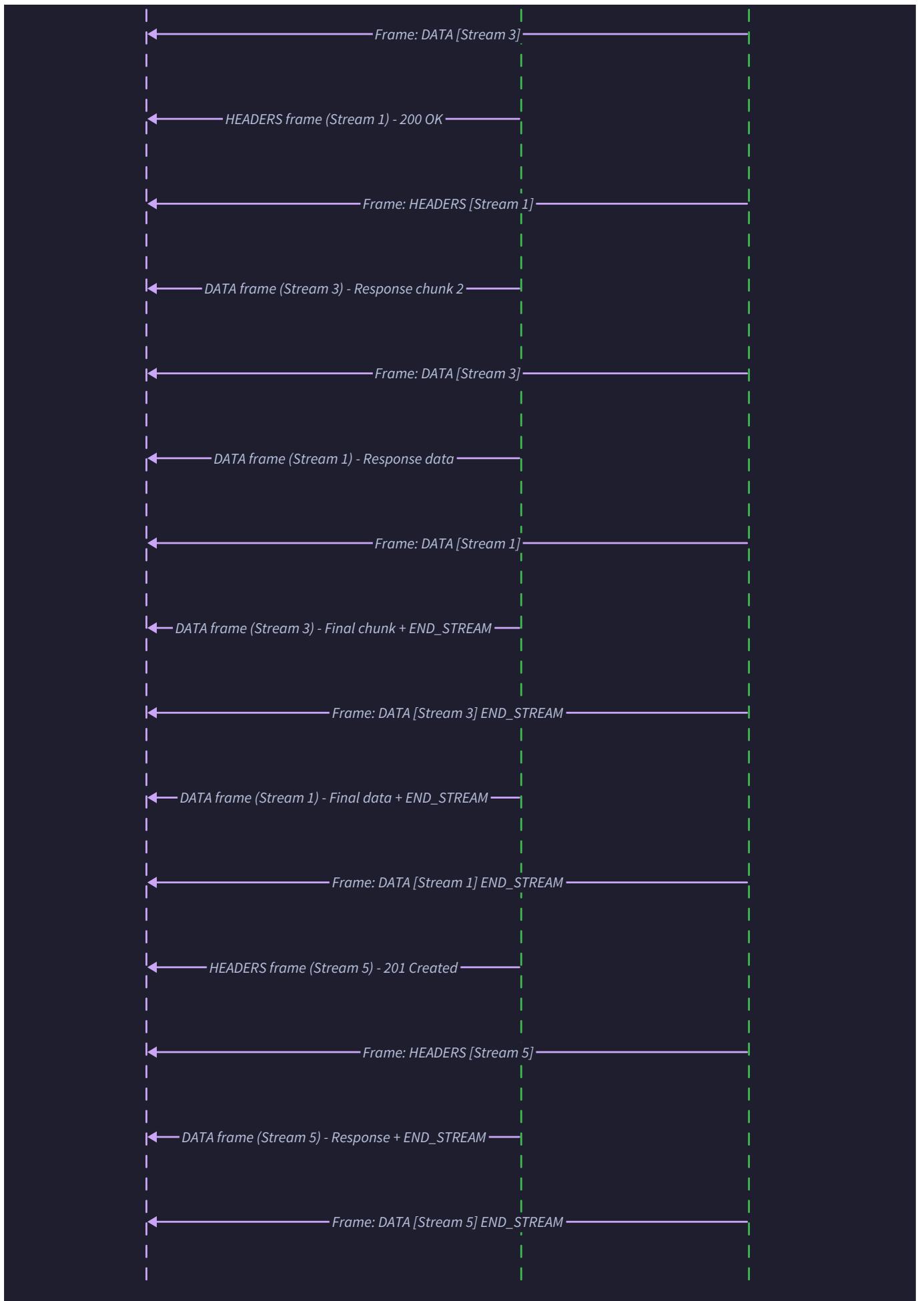
⚠ Pitfall: Race Conditions in State Transitions

A common mistake is failing to handle race conditions where both endpoints simultaneously send frames that trigger state transitions. For example, if the client sends a DATA frame with END_STREAM while the server sends RST_STREAM, both frames might be in flight. The implementation must define a consistent ordering: RST_STREAM always wins and immediately transitions to closed, regardless of other pending frames. Without this rule, streams can get stuck in inconsistent states where each endpoint believes different state information.

Stream ID Allocation

Stream ID allocation in HTTP/2 follows a carefully designed numbering scheme that prevents collisions, enables concurrent stream creation, and provides implicit metadata about stream ownership and directionality. The allocation strategy must handle both client-initiated and server-initiated streams while maintaining strict ordering requirements.





The fundamental allocation rule divides the 31-bit stream ID space (bit 32 is reserved) into two non-overlapping ranges:

Stream Type	ID Range	Initiator	Usage Pattern
Client-initiated	Odd numbers (1, 3, 5, ...)	HTTP client	Normal request-response pairs
Server-initiated	Even numbers (2, 4, 6, ...)	HTTP server	Server push (future extension)
Connection control	Stream ID 0	Either endpoint	SETTINGS, PING, GOAWAY frames

The allocation process maintains strict monotonicity requirements - each endpoint must use strictly increasing stream IDs for new streams. This ordering enables efficient resource management and prevents certain protocol attacks:

1. **Client allocation:** Start with stream ID 1, increment by 2 for each new stream (1, 3, 5, 7, ...)
2. **Server allocation:** Start with stream ID 2, increment by 2 for each new stream (2, 4, 6, 8, ...)
3. **Ordering validation:** Reject any stream ID that is less than or equal to the highest previously used ID from the same initiator
4. **Resource tracking:** Use the highest allocated stream ID to determine connection liveliness and resource cleanup timing

Decision: Monotonic Stream ID Allocation with Gap Tolerance

- **Context:** HTTP/2 requires monotonically increasing stream IDs, but network conditions might cause frames to arrive out of order
- **Options Considered:** Strict sequential validation, gap tolerance with buffering, or relaxed ordering
- **Decision:** Implement monotonic validation with limited gap tolerance for reasonable out-of-order delivery
- **Rationale:** Strict monotonic ordering prevents resource exhaustion attacks while limited buffering accommodates realistic network reordering without compromising security
- **Consequences:** Provides security against malicious clients while maintaining compatibility with standard network conditions, requires bounded buffer for out-of-order stream creation

The stream ID allocation component tracks allocation state per connection and validates incoming stream creation requests:

Field Name	Type	Description
nextClientStreamID	uint32	Next available odd stream ID for client allocation
nextServerStreamID	uint32	Next available even stream ID for server allocation
highestClientStreamID	uint32	Highest client stream ID seen (monotonicity check)
highestServerStreamID	uint32	Highest server stream ID seen (monotonicity check)
pendingStreams	map[uint32]*Stream	Streams created but not yet confirmed
maxStreamID	uint32	Maximum allowed stream ID ($2^{31} - 1$)

Stream ID exhaustion represents a serious but manageable condition that occurs when an endpoint approaches the maximum stream ID value. The protocol provides graceful handling through connection cycling:

1. **Exhaustion detection:** Monitor when next stream ID approaches `MaxStreamID` ($2^{31} - 1$)
2. **Graceful shutdown initiation:** Send GOAWAY frame indicating the highest accepted stream ID
3. **Connection draining:** Allow existing streams to complete naturally while rejecting new stream creation
4. **New connection establishment:** Client establishes new connection for additional streams
5. **Resource cleanup:** After all streams complete, close the exhausted connection

⚠ Pitfall: Stream ID Reuse After Connection Reset

A critical mistake is attempting to reuse stream IDs after a connection error and recovery. Each connection maintains independent stream ID allocation state, but implementations sometimes incorrectly carry over stream ID counters across connection boundaries. This can lead to protocol violations when the new connection's stream IDs conflict with cached state from the previous connection. Always reset stream ID allocation to initial values (1 for client, 2 for server) when establishing a new connection.

Priority Scheduling

HTTP/2's **priority scheduling** system enables intelligent bandwidth allocation among concurrent streams through a sophisticated weight and dependency model. Unlike simple round-robin scheduling, HTTP/2 priority allows clients to express the relative importance of resources and create dependency relationships that optimize page load performance.

The priority system operates on two fundamental concepts: **stream weights** and **dependency relationships**. Stream weights range from 1 to 256 and determine the relative share of available bandwidth each stream should receive. Dependencies create parent-child relationships where child streams should only receive bandwidth after their parent stream completes or reaches certain milestones.

Priority Concept	Range/Type	Purpose	Default Value
Stream Weight	1-256	Relative bandwidth allocation within same priority level	16
Dependency Stream ID	0 or valid stream ID	Parent stream that should be prioritized first	0 (no dependency)
Exclusive Flag	Boolean	Whether this stream should be the sole child of its parent	false
Priority Tree Depth	Unlimited (implementation limited)	Maximum dependency chain length	Typically limited to 32 levels

The priority tree structure enables complex resource relationships common in web applications. For example, HTML documents typically have the highest priority, followed by critical CSS and JavaScript, with images and other assets receiving lower priority. The dependency model allows expressing rules like "load all CSS before any images" or "prioritize above-the-fold images over below-the-fold content."

Stream Type	Typical Weight	Common Dependencies	Scheduling Rationale
HTML Document	220-256	None (root level)	Critical rendering path foundation
Critical CSS	200-220	HTML document	Blocks rendering until loaded
JavaScript	180-200	HTML document	May block parsing or execution
Above-fold Images	100-150	CSS files	Visible content prioritized
Below-fold Images	50-100	Above-fold resources	Deferred until critical path complete
Tracking/Analytics	1-20	All content resources	Lowest priority, non-blocking

The priority scheduling algorithm maintains a dynamic tree structure that evolves as streams are created, completed, and reprioritized:

Algorithm Step	Action Taken	Data Structure Update
1. New Stream Creation	Insert stream into priority tree at specified dependency position	Update parent's children list, set stream's parent pointer
2. Weight Calculation	Calculate effective weight considering parent completion status	Traverse dependency path to root, accumulate weight factors
3. Bandwidth Allocation	Distribute available send quota proportionally among ready streams	Sort streams by effective priority, allocate bytes proportionally
4. Stream Completion	Remove completed stream from tree, promote children to grandparent	Update parent-child relationships, recalculate descendant weights
5. Priority Updates	Process PRIORITY frames to modify weights or dependencies	Rebuild affected tree portions, detect and break dependency cycles

Decision: Simplified Priority Implementation with Weight-Based Scheduling

- **Context:** Full HTTP/2 priority with dependency trees is complex and provides diminishing returns for many applications
- **Options Considered:** Full dependency tree implementation, weight-only scheduling, or FIFO ordering
- **Decision:** Implement weight-based scheduling with basic dependency support but without complex tree restructuring
- **Rationale:** Weight-based scheduling provides 80% of the benefit with 20% of the complexity, making it more maintainable and debuggable for learning purposes
- **Consequences:** Simpler implementation that handles most real-world scenarios, but lacks advanced features like dependency tree manipulation and exclusive priorities

The priority scheduling data structures track both the tree relationships and the dynamic scheduling state:

Field Name	Type	Description
weight	uint8	Stream priority weight (1-256), higher means more bandwidth
dependsOn	uint32	Parent stream ID, or 0 for root-level stream
exclusive	bool	Whether this stream should be the exclusive child of its parent
children	[]uint32	List of child stream IDs that depend on this stream
effectiveWeight	float64	Calculated weight considering parent completion status
schedulingQuota	int64	Bytes allocated to this stream in current scheduling round
pendingBytes	int64	Total bytes queued for transmission on this stream

The bandwidth allocation algorithm runs each time the connection has data to send and available flow control window:

1. **Identify ready streams:** Find all streams with pending data and available flow control window
2. **Calculate effective weights:** For each ready stream, compute its effective priority considering parent dependencies and completion status
3. **Proportional allocation:** Distribute the available connection window proportionally based on effective weights
4. **Minimum allocation:** Ensure each ready stream receives at least a minimum quantum (typically 1KB) to prevent starvation
5. **Remainder distribution:** Allocate any remaining bytes to streams with the highest effective weights
6. **Flow control update:** Deduct allocated bytes from connection and stream flow control windows as data is sent

! Pitfall: Priority Inversion Through Flow Control Exhaustion

A subtle but serious issue occurs when high-priority streams exhaust their flow control windows while low-priority streams continue transmitting. This creates priority inversion where less important streams consume bandwidth while critical streams wait for window updates. The solution requires integrating priority scheduling with flow control management - when high-priority streams are window-limited, the scheduler should pause lower-priority streams and send WINDOW_UPDATE frames to unblock the critical path.

Stream Limits

Stream limits provide essential resource management and DoS protection for HTTP/2 servers by controlling the number of concurrent streams, enforcing creation rate limits, and providing mechanisms for immediate stream termination when necessary. Without proper limits, malicious clients can easily exhaust server resources by opening thousands of concurrent streams or continuously creating and abandoning streams.

The HTTP/2 specification defines several categories of stream limits that implementations must enforce:

Limit Type	Default Value	Scope	Purpose
<code>SETTINGS_MAX_CONCURRENT_STREAMS</code>	100 (recommended)	Per-connection	Maximum simultaneously active streams
<code>SETTINGS_MAX_FRAME_SIZE</code>	16384-16777215	Per-connection	Maximum bytes in a single frame payload
<code>SETTINGS_MAX_HEADER_LIST_SIZE</code>	Unlimited (implementation choice)	Per-stream	Maximum decompressed header bytes
Implementation stream creation rate	10 streams/second (recommended)	Per-connection	DoS protection against stream flooding
Implementation memory per stream	4KB-16KB typical	Per-stream	Memory exhaustion protection
Implementation total connection streams	10,000 lifetime typical	Per-connection	Resource cleanup and connection cycling

The concurrent stream limit represents the most important protection mechanism. It prevents clients from opening unlimited streams simultaneously while allowing reasonable concurrency for legitimate use cases:

- 1. Limit enforcement:** Track active stream count and reject new HEADERS frames when limit is reached
- 2. Stream counting:** Increment counter on stream creation, decrement on transition to `StateClosed`
- 3. Limit negotiation:** Use SETTINGS frames to communicate server capacity to clients
- 4. Graceful degradation:** When approaching limits, prioritize existing streams over new stream creation
- 5. Emergency shedding:** If limits are exceeded due to race conditions, use RST_STREAM to quickly close least important streams

Stream State	Counts Toward Limit	Rationale
<code>StateIdle</code>	No	Stream not yet created
<code>StateOpen</code>	Yes	Fully active stream consuming resources
<code>StateHalfClosedLocal</code>	Yes	Still consuming server resources for response
<code>StateHalfClosedRemote</code>	Yes	Still consuming server resources for processing
<code>StateClosed</code>	No	Resources released, available for new streams
<code>StateReservedLocal</code>	Yes	Server-initiated stream consuming resources
<code>StateReservedRemote</code>	No	Client reservation, minimal resource usage

RST_STREAM handling provides the mechanism for immediate stream termination in response to errors, resource exhaustion, or application-level cancellation requests:

RST_STREAM Error Code	Typical Cause	Server Response
PROTOCOL_ERROR	Invalid frame sequence or malformed data	Log error, close stream, continue connection
INTERNAL_ERROR	Server implementation bug or resource exhaustion	Close stream, investigate root cause
FLOW_CONTROL_ERROR	Stream exceeded flow control window	Close stream, may indicate malicious client
STREAM_CLOSED	Frame received on already closed stream	Close stream, normal race condition handling
FRAME_SIZE_ERROR	Frame payload exceeds negotiated maximum	Close stream, possible DoS attempt
REFUSED_STREAM	Server cannot handle stream (resource limits)	Close stream, client should retry
CANCEL	Client-initiated cancellation	Close stream, clean up resources
COMPRESSION_ERROR	HPACK decompression failure	Close connection (affects header table)
CONNECT_ERROR	HTTP CONNECT method failed	Close stream, connection-specific error

Decision: Adaptive Stream Limits Based on Connection Health

- **Context:** Fixed stream limits may be too restrictive for well-behaved clients or too permissive for malicious ones
- **Options Considered:** Fixed limits, per-client adaptive limits, or connection health-based adjustment
- **Decision:** Implement base limits with health-based adjustments using connection metrics
- **Rationale:** Allows legitimate clients with good behavior to exceed base limits while quickly restricting misbehaving connections
- **Consequences:** More complex implementation but better resource utilization and DoS protection

The stream limit enforcement requires careful coordination with the connection's overall resource management:

Field Name	Type	Description
<code>maxConcurrentStreams</code>	<code>uint32</code>	Maximum allowed simultaneous active streams
<code>currentStreamCount</code>	<code>uint32</code>	Currently active streams (open + half-closed)
<code>totalStreamsCreated</code>	<code>uint64</code>	Lifetime stream creation counter for rate limiting
<code>streamCreationTimes</code>	<code>[]time.Time</code>	Recent stream creation timestamps for rate calculation
<code>rejectedStreams</code>	<code>uint64</code>	Count of streams rejected due to limits (monitoring)
<code>emergencyMode</code>	<code>bool</code>	Whether connection is in resource conservation mode
<code>connectionHealthScore</code>	<code>float64</code>	Calculated health metric for adaptive limit adjustment

The stream creation rate limiting algorithm prevents rapid-fire stream creation attacks while accommodating legitimate burst traffic:

1. **Rate calculation:** Maintain sliding window of recent stream creation timestamps
2. **Burst allowance:** Permit short bursts above sustained rate (token bucket algorithm)
3. **Sustained rate limiting:** Enforce maximum average creation rate over longer periods
4. **Adaptive thresholds:** Reduce limits for connections showing suspicious patterns
5. **Graceful rejection:** Use `REFUSED_STREAM` error rather than connection termination when possible

⚠ Pitfall: Memory Leaks from Incomplete Stream Cleanup

A common implementation mistake is failing to fully clean up stream resources when streams are reset or completed. Each stream typically holds references to header tables, pending data buffers, flow control state, and priority tree positions. When `RST_STREAM` is received or sent, all of these resources must be immediately released. Incomplete cleanup leads to memory leaks that accumulate over connection lifetime, eventually causing server instability. Implement comprehensive cleanup checklists and use resource tracking to detect leaks during testing.

Implementation Guidance

A. Technology Recommendations:

Component	Simple Option	Advanced Option
Stream Storage	<code>map[uint32]*Stream</code> with mutex	Lock-free concurrent map with sharding
State Machine	Switch statements with validation	State pattern with transition tables
Priority Scheduling	Weight-based round robin	Full dependency tree with heap scheduling
Stream Limits	Fixed thresholds with counters	Adaptive limits with connection health scoring
ID Allocation	Atomic counters with validation	Lock-free allocation with gap management

B. Recommended File Structure:

```

internal/stream/
    manager.go           ← StreamManager with lifecycle coordination
    stream.go            ← Individual Stream struct and methods
    state_machine.go     ← State transition logic and validation
    priority.go          ← Priority scheduling and bandwidth allocation
    limits.go            ← Stream limits and rate limiting
    id_allocator.go      ← Stream ID allocation and validation
    manager_test.go       ← Integration tests for full stream lifecycle
    stream_test.go        ← Unit tests for individual stream operations
  
```

C. Infrastructure Starter Code:

```
package stream

import (
    "fmt"
    "sync"
    "time"
)

// StreamState represents the current state in the HTTP/2 stream lifecycle

type StreamState int

const (
    StateIdle StreamState = iota
    StateReservedLocal
    StateReservedRemote
    StateOpen
    StateHalfClosedRemote
    StateHalfClosedLocal
    StateClosed
)

// StreamPriority holds priority scheduling information for a stream

type StreamPriority struct {
    Weight      uint8    // 1-256, higher means more bandwidth
    DependsOn  uint32    // Parent stream ID, 0 for root level
    Exclusive   bool     // Whether this is the exclusive child of parent
}

// StreamManager coordinates all streams for a single HTTP/2 connection
```

GO

```
type StreamManager struct {

    mu           sync.RWMutex

    streams      map[uint32]*Stream

    maxConcurrentStreams uint32

    currentStreamCount   uint32

    nextClientStreamID   uint32

    nextServerStreamID   uint32

    highestClientStreamID uint32

    highestServerStreamID uint32

    connectionWindow     int64

    streamCreationTimes []time.Time

    closed              bool

}

// NewStreamManager creates a new stream manager with default settings

func NewStreamManager(maxConcurrentStreams uint32) *StreamManager {

    return &StreamManager{

        streams:          make(map[uint32]*Stream),

        maxConcurrentStreams: maxConcurrentStreams,

        nextClientStreamID: 1,

        nextServerStreamID: 2,

        connectionWindow:   InitialWindowSize,

        streamCreationTimes: make([]time.Time, 0, 100),

    }

}

// Stream represents an individual HTTP/2 stream with its state and resources

type Stream struct {
```

```
    id          uint32

    state       StreamState

    localEndStream bool

    remoteEndStream bool

    sendWindow    int64

    recvWindow    int64

    priority     StreamPriority

    headers      map[string]string

    pendingData   []byte

    created      time.Time

    lastActivity time.Time

    mu           sync.RWMutex

}

// CreateStream allocates a new stream with the given ID and initial state

func (sm *StreamManager) CreateStream(streamID uint32, isClientInitiated bool) (*Stream, error) {

    sm.mu.Lock()

    defer sm.mu.Unlock()

    // TODO 1: Validate stream ID monotonicity based on initiator

    // TODO 2: Check concurrent stream limits

    // TODO 3: Perform rate limiting check

    // TODO 4: Create new Stream instance with proper initialization

    // TODO 5: Update stream counters and tracking

    // TODO 6: Add stream to manager's stream map

    return nil, fmt.Errorf("not implemented")
}
```

}

D. Core Logic Skeleton Code:

GO

```
// TransitionState moves the stream to a new state, validating the transition

func (s *Stream) TransitionState(newState StreamState, frameType uint8) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    // TODO 1: Check if transition from current state to newState is valid
    // TODO 2: Validate that the triggering frame type is appropriate
    // TODO 3: Update state-specific flags (localEndStream, remoteEndStream)
    // TODO 4: Update lastActivity timestamp
    // TODO 5: Trigger any cleanup actions if transitioning to StateClosed
    // Hint: Use a transition table or switch statement based on current state

    return fmt.Errorf("not implemented")
}

// AllocateStreamID returns the next available stream ID for the given initiator

func (sm *StreamManager) AllocateStreamID(isClientInitiated bool) (uint32, error) {
    sm.mu.Lock()
    defer sm.mu.Unlock()

    // TODO 1: Check for stream ID exhaustion (approaching 2^31-1)
    // TODO 2: Select appropriate counter (client odd, server even)
    // TODO 3: Validate monotonicity against highest seen ID
    // TODO 4: Increment counter by 2 to maintain odd/even separation
    // TODO 5: Update highest seen ID for the initiator
    // Hint: Client IDs are odd (1,3,5...), server IDs are even (2,4,6...)
}
```

```
    return 0, fmt.Errorf("not implemented")
}

// ScheduleStreams allocates available bandwidth among streams based on priority

func (sm *StreamManager) ScheduleStreams(availableWindow int64) map[uint32]int64 {
    sm.mu.RLock()
    defer sm.mu.RUnlock()

    allocation := make(map[uint32]int64)

    // TODO 1: Find all streams with pending data and available send window

    // TODO 2: Calculate total weight of ready streams

    // TODO 3: Allocate bandwidth proportionally based on stream weights

    // TODO 4: Ensure minimum allocation per stream to prevent starvation

    // TODO 5: Handle case where available window is less than total demand

    // Hint: Use weighted fair queuing - higher weight streams get more bytes

    return allocation
}

// HandleRSTStream processes a stream reset and cleans up resources

func (sm *StreamManager) HandleRSTStream(streamID uint32, errorCode uint32) error {
    sm.mu.Lock()
    defer sm.mu.Unlock()

    // TODO 1: Locate the stream by ID, return error if not found

    // TODO 2: Transition stream to StateClosed regardless of current state

    // TODO 3: Clean up pending data buffers and release memory
```

```

// TODO 4: Remove stream from priority tree if it has dependencies

// TODO 5: Update concurrent stream counter

// TODO 6: Log the reset for debugging/monitoring

// Hint: RST_STREAM always wins - immediately close regardless of state

return fmt.Errorf("not implemented")

}

```

E. Language-Specific Hints:

- Use `sync.RWMutex` for stream manager - many reads (frame routing), fewer writes (stream creation)
- Use `time.Now()` for timestamps and `time.Since()` for rate limiting calculations
- Use `atomic.AddUint32()` for stream counters if implementing lock-free paths
- Use `make(map[uint32]*Stream, expectedSize)` with capacity hint to reduce allocations
- Use `defer` statements for cleanup in RST_STREAM handling to ensure resources are always released

F. Milestone Checkpoint: After implementing stream management:

1. **Test stream creation:** `go test -v ./internal/stream/ -run TestCreateStream`
2. **Expected behavior:** Should create streams with proper ID allocation (odd for client, even for server)
3. **Test state transitions:** Create stream, send HEADERS (idle → open), send END_STREAM (open → half-closed)
4. **Test concurrent limits:** Try creating more streams than `maxConcurrentStreams`, should get REFUSED_STREAM
5. **Manual verification:** Enable debug logging, create multiple streams, verify each has unique ID and proper state
6. **Signs of problems:** Streams stuck in wrong states, ID collisions, or resource leaks (check with `go tool pprof`)

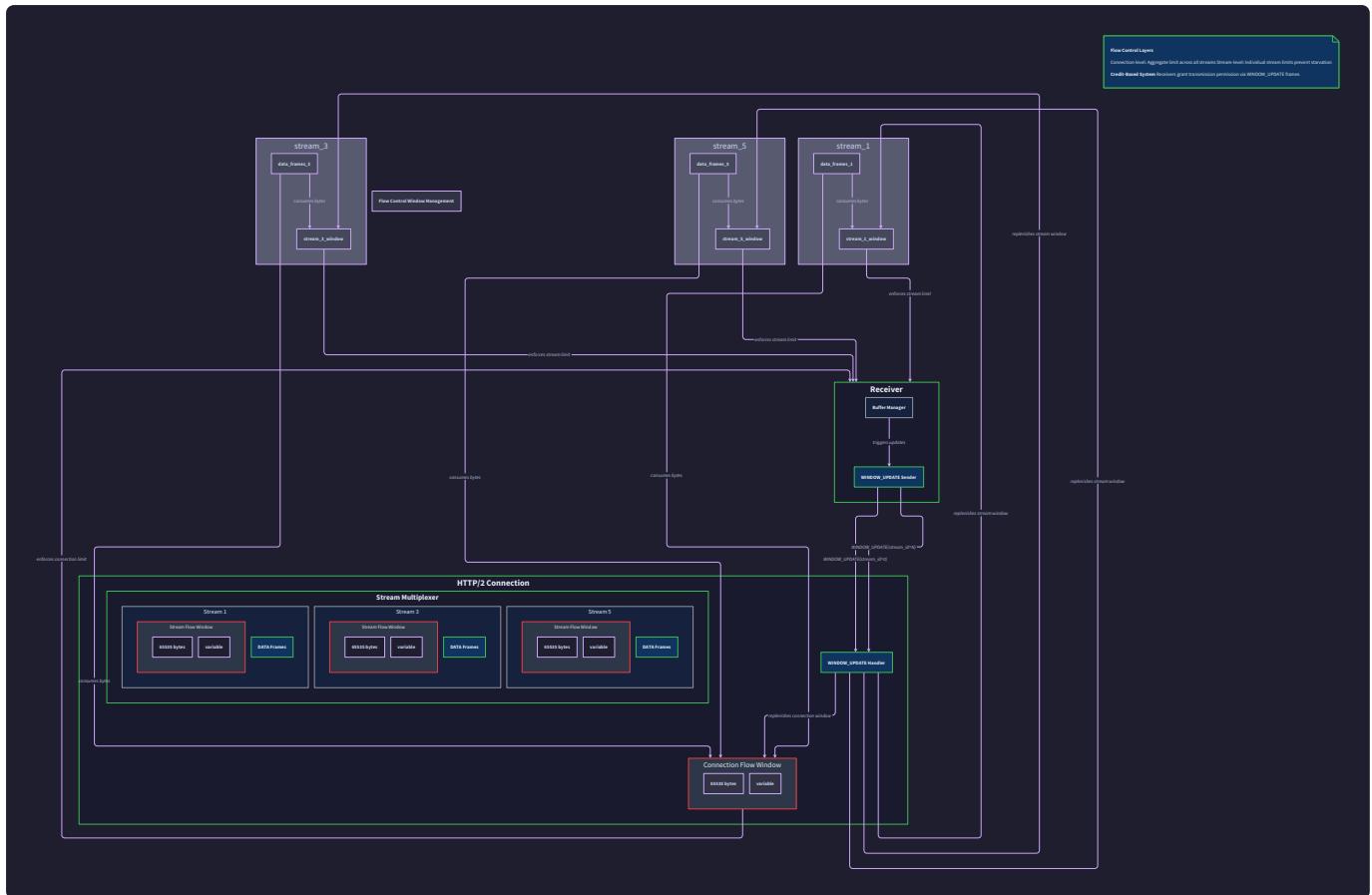
G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Stream stuck in StateOpen	Missing END_STREAM flag detection	Check frame flag parsing in headers	Ensure END_STREAM flag (0x1) is properly detected and processed
Stream ID collisions	Incorrect odd/even allocation	Log all stream ID allocations with initiator	Fix AllocateStreamID to properly increment by 2, not 1
Memory usage growing	Stream cleanup incomplete	Use heap profiler to find leaked Stream objects	Ensure RST_STREAM removes all references and buffers
Priority not working	Weight calculation wrong	Log effective weights during scheduling	Check that weight calculation considers parent completion
Connection stalls	Stream limits too restrictive	Monitor rejected stream count	Increase maxConcurrentStreams or implement adaptive limits

Flow Control

Milestone(s): This section primarily addresses Milestone 4 (Flow Control) by implementing connection and stream-level flow control mechanisms, with foundational concepts supporting all previous milestones' concurrent operation.

Think of HTTP/2 flow control as a sophisticated traffic management system for a busy highway interchange. Just as traffic lights, on-ramps, and merge lanes prevent congestion by controlling how many cars enter different road segments, HTTP/2 flow control prevents network congestion by managing how much data can be sent over connection and stream channels. Unlike a simple traffic light that's either red or green, HTTP/2 flow control uses dynamic "credit systems" where senders receive permission tokens (window updates) that specify exactly how many bytes they're allowed to transmit before waiting for more tokens.



The fundamental challenge in HTTP/2 flow control stems from the **multiplexing paradigm** - multiple independent streams share a single TCP connection, creating a two-level resource management problem. At the connection level, we must prevent the aggregate data from all streams from overwhelming the receiver's network buffers. At the stream level, we must ensure that a fast-producing stream doesn't monopolize bandwidth and starve other streams. This dual-layer approach requires careful coordination between connection-wide and per-stream flow control windows, with complex interactions during window updates and backpressure scenarios.

Critical Design Insight: HTTP/2 flow control operates on a credit-based system where receivers explicitly grant permission for data transmission. This inverts the traditional TCP model where senders transmit optimistically and react to backpressure signals. The receiver-driven approach provides fine-grained control but introduces complexity around window exhaustion, update timing, and deadlock prevention.

Window Management

Window management in HTTP/2 implements a hierarchical credit system where both connections and individual streams maintain separate flow control windows. Think of this as a two-tier banking system: the connection window represents the total credit limit across all accounts (streams), while each stream window represents an individual account balance. Every DATA frame transmission requires "payment" from both the

stream's account and the connection's master account. If either account lacks sufficient funds, the transaction (DATA frame) must wait until credit becomes available.

The window management system tracks four distinct window states for each active component:

Window Type	Scope	Initial Size	Purpose	Update Mechanism
Connection Send Window	Per-Connection	InitialWindowSize (65535)	Limits total outbound DATA bytes across all streams	WINDOW_UPDATE frames with StreamID=0
Stream Send Window	Per-Stream	InitialWindowSize (65535)	Limits outbound DATA bytes for specific stream	WINDOW_UPDATE frames with specific StreamID
Connection Receive Window	Per-Connection	InitialWindowSize (65535)	Limits total inbound DATA bytes across all streams	Local WINDOW_UPDATE generation
Stream Receive Window	Per-Stream	InitialWindowSize (65535)	Limits inbound DATA bytes for specific stream	Local WINDOW_UPDATE generation

Each window operates as a signed 32-bit integer that decrements when DATA frames are transmitted and increments when WINDOW_UPDATE frames are received (for send windows) or generated (for receive windows). The critical invariant is that **both the connection and stream send windows must have sufficient credit** before any DATA frame transmission can proceed.

Decision: Hierarchical Window Architecture

- **Context:** HTTP/2 requires flow control at both connection and stream levels, but the interaction between these levels wasn't clearly specified in early implementations
- **Options Considered:**
 1. Independent windows (stream and connection windows operate separately)
 2. Hierarchical windows (DATA frames consume credit from both levels)
 3. Stream-only windows (ignore connection-level flow control)
- **Decision:** Hierarchical windows where DATA transmission requires credit from both stream and connection windows
- **Rationale:** This approach provides maximum flexibility for receiver-driven flow control while preventing any single stream from overwhelming the connection's aggregate capacity
- **Consequences:** Increases implementation complexity but provides robust protection against resource exhaustion and enables sophisticated traffic shaping

The window tracking implementation maintains these values within the existing data structures:

Field	Type	Location	Purpose
sendwindow	int64	Stream struct	Current send window for this stream
recvwindow	int64	Stream struct	Current receive window for this stream
connectionSendWindow	int64	Connection state	Aggregate send window across all streams
connectionRecvWindow	int64	Connection state	Aggregate receive window across all streams
pendingWindowUpdates	map[uint32]int64	Connection state	Queued WINDOW_UPDATE frames awaiting transmission

Window update processing follows a precise protocol when WINDOW_UPDATE frames arrive:

1. **Frame Validation:** Verify the WINDOW_UPDATE frame contains a non-zero window size increment and targets a valid stream (or connection with StreamID=0)
2. **Overflow Protection:** Check that adding the increment to the current window wouldn't exceed the maximum window size ($2^{31} - 1$)
3. **Window Application:** Add the increment to the appropriate window (connection if StreamID=0, specific stream otherwise)

4. **Pending Data Processing:** Scan queued DATA frames and resume transmission for any frames that now have sufficient window credit

5. **Cascade Updates:** If connection window increased, check all streams for newly transmittable data

The window update generation logic operates continuously in the background, monitoring receive windows and generating WINDOW_UPDATE frames when windows fall below threshold levels:

Condition	Action	Window Update Size	Rationale
Stream receive window < 25% of initial	Generate stream WINDOW_UPDATE	Restore to initial size	Prevent receive window exhaustion on active streams
Connection receive window < 50% of initial	Generate connection WINDOW_UPDATE	Restore to 75% of initial	Maintain aggregate receive capacity across all streams
Stream closed with positive receive window	No action	N/A	Avoid unnecessary protocol overhead
SETTINGS frame changes initial window size	Adjust all existing stream windows	Difference between old and new initial size	Maintain consistent window semantics

Backpressure Handling

Backpressure handling manages the queueing and resumption of DATA frames when flow control windows become exhausted. Think of this as a sophisticated traffic management system at a busy intersection: when the main road (connection window) or specific lanes (stream windows) become congested, incoming traffic (DATA frames) must queue in organized waiting areas until capacity becomes available. The system maintains separate queues for different priority levels and implements fair scheduling to prevent high-priority streams from completely starving lower-priority ones.

The backpressure system maintains multiple data structures to track pending transmissions and coordinate resumption:

Structure	Type	Purpose	Organization
<code>streamSendQueues</code>	<code>map[uint32] []QueuedFrame</code>	Per-stream DATA frame queues	FIFO queue per stream ID
<code>blockedOnConnection</code>	<code>[]uint32</code>	Streams waiting for connection window	Priority-ordered list of stream IDs
<code>blockedOnStream</code>	<code>[]uint32</code>	Streams waiting for stream window	Simple list of stream IDs
<code>totalQueuedBytes</code>	<code>int64</code>	Total bytes in all queues	Memory usage tracking
<code>maxQueueSize</code>	<code>int64</code>	Maximum allowed queued bytes	Prevents unbounded memory growth

The `QueuedFrame` structure encapsulates all information needed to resume transmission when windows become available:

Field	Type	Description
<code>frame</code>	<code>*Frame</code>	The original DATA frame awaiting transmission
<code>timestamp</code>	<code>time.Time</code>	When the frame was queued for timeout detection
<code>streamID</code>	<code>uint32</code>	Target stream ID for routing
<code>originalSize</code>	<code>uint32</code>	Frame size for window accounting
<code>retryCount</code>	<code>int</code>	Number of transmission attempts

When a DATA frame cannot be transmitted due to insufficient window credit, the backpressure handler follows this queuing procedure:

- 1. Memory Limit Check:** Verify that adding this frame won't exceed the maximum queue size limit
- 2. Queue Selection:** Choose the appropriate per-stream queue based on the frame's StreamID
- 3. Frame Preparation:** Create a `QueuedFrame` with current timestamp and retry metadata
- 4. Queue Insertion:** Append the frame to the stream's FIFO queue
- 5. Blocked List Update:** Add the stream ID to the appropriate blocked list (connection or stream)
- 6. Memory Tracking:** Update total queued bytes counter for resource monitoring

The resumption logic activates whenever window credit becomes available through WINDOW_UPDATE frame processing or stream closure:

Trigger Event	Resumption Scope	Processing Order	Fairness Mechanism
Connection WINDOW_UPDATE	All blocked streams	Round-robin across streams	Each stream gets equal connection window allocation
Stream WINDOW_UPDATE	Single specific stream	FIFO within stream queue	Maintains per-stream ordering guarantees
Stream closure	Connection window recovery	Priority-weighted distribution	Closed stream's unused window redistributed by priority
SETTINGS initial window change	All affected streams	Priority order	Higher priority streams processed first

The fair scheduling algorithm ensures that no single stream monopolizes available window credit during resumption:

- Credit Distribution:** When connection window becomes available, divide credit proportionally among blocked streams based on their priority weights
- Stream Processing:** Process each stream's queue in round-robin fashion, transmitting one frame per stream per round
- Remaining Credit:** After each round, recalculate available credit and repeat until either all queues are empty or credit is exhausted
- Priority Adjustment:** Higher-weight streams get additional rounds proportional to their weight advantage

Decision: Fair Scheduling with Priority Weights

- Context:** Multiple streams competing for limited connection window credit creates potential for starvation scenarios
- Options Considered:**
 - Strict priority (higher priority streams always go first)
 - Pure round-robin (all streams get equal treatment)
 - Weighted fair queuing (priority influences allocation but doesn't eliminate lower priorities)
- Decision:** Weighted fair queuing with configurable priority weights
- Rationale:** Balances responsiveness for high-priority streams while preventing complete starvation of background streams
- Consequences:** More complex scheduling logic but provides predictable performance characteristics for different stream types

Window Edge Cases

Window edge cases address the complex failure modes and boundary conditions that arise in HTTP/2 flow control systems. Think of these as the emergency protocols for unusual traffic situations: what happens when the credit system breaks down due to arithmetic overflow, when streams deadlock waiting for each other's windows, or when network partitions cause window updates to be lost. These edge cases often represent the most challenging debugging scenarios because they involve subtle timing interactions between multiple components.

The most critical edge cases involve **window arithmetic overflow and underflow** conditions that can occur during normal operation:

Condition	Detection	Recovery Action	Prevention
Window overflow ($> 2^{31}-1$)	Check addition result before applying	Send FLOW_CONTROL_ERROR and close connection	Validate WINDOW_UPDATE increments
Window underflow (< 0)	Check subtraction result before applying	Send FLOW_CONTROL_ERROR for stream/connection	Track pending DATA frame sizes accurately
Negative WINDOW_UPDATE	Frame validation during parsing	Send PROTOCOL_ERROR and close connection	Validate increment field is positive
Zero-byte WINDOW_UPDATE	Frame validation during parsing	Send PROTOCOL_ERROR and close connection	Reject WINDOW_UPDATE frames with zero increment

Deadlock scenarios represent particularly dangerous edge cases where circular dependencies between stream and connection windows create unresolvable blocking conditions:

Critical Deadlock Pattern: Stream A and Stream B both have exhausted connection window credit but positive stream window credit. Each stream's application layer continues generating data, filling their respective queues. However, neither can make progress because the connection window remains at zero, and no WINDOW_UPDATE can be sent because the streams themselves are blocked.

The deadlock prevention system implements several protective mechanisms:

Deadlock Type	Detection Signal	Prevention Strategy	Recovery Mechanism
Connection window starvation	All streams blocked on connection window	Reserve connection window credit for WINDOW_UPDATE frames	Force-send WINDOW_UPDATE even with zero connection window
Stream priority inversion	High-priority stream blocked while low-priority streams consume window	Priority-based preemption of low-priority queued frames	Redistribute queued frame priority ordering
Circular stream dependencies	Stream dependency graph contains cycles	Dependency graph validation during PRIORITY frame processing	Break cycles by resetting problematic dependency links
Window update loss	Stream remains blocked despite peer claiming to send updates	Timeout-based window refresh requests	Send explicit WINDOW_UPDATE request frames

Memory exhaustion during backpressure handling creates another class of critical edge cases:

The system implements aggressive memory management when approaching queue size limits:

1. **Early Warning Thresholds:** When total queued bytes exceed 75% of maximum, begin selective frame dropping for lowest-priority streams
2. **Emergency Shedding:** When approaching 90% of maximum, implement tail-drop for new frames on already-queued streams
3. **Priority-Based Eviction:** When at 100% capacity, evict lowest-priority queued frames to make room for higher-priority ones
4. **Connection Termination:** If memory pressure continues despite aggressive shedding, send GOAWAY and terminate connection

Clock skew and timing issues can cause subtle flow control misbehavior in distributed deployments:

Timing Issue	Symptoms	Root Cause	Mitigation
Delayed WINDOW_UPDATE processing	Streams remain blocked despite sufficient window	WINDOW_UPDATE frames queued behind large DATA frames	Prioritize WINDOW_UPDATE frames in transmission queue
Premature timeout	Streams reset before WINDOW_UPDATE arrives	Network latency exceeds configured timeout	Implement adaptive timeout based on measured RTT
Window update storms	Rapid bursts of WINDOW_UPDATE frames	Receiver generating updates faster than sender can process	Rate-limit WINDOW_UPDATE generation and batch updates
Stale window state	Window tracking becomes inconsistent	Lost or reordered frames cause accounting errors	Implement periodic window state synchronization

Common Pitfalls

⚠ Pitfall: Single-Level Window Accounting Many developers implement only stream-level flow control and ignore connection-level windows, assuming that controlling individual streams provides sufficient protection. This creates a classic "tragedy of the commons" scenario where each stream appears well-behaved individually, but their aggregate behavior overwhelms the connection's capacity. The result is receiver buffer exhaustion, TCP-level backpressure, and cascading performance degradation across all streams. **Fix:** Always implement dual-level accounting where DATA frame transmission requires credit from both stream and connection windows.

⚠ Pitfall: Arithmetic Overflow Ignored Window update arithmetic can easily overflow the 32-bit signed integer range, especially in high-throughput scenarios or when malicious peers send large WINDOW_UPDATE increments. Unchecked overflow can wrap the window size to negative values, causing immediate flow control violations and connection termination. **Fix:** Validate that `current_window + update_increment ≤ 2^31-1` before applying any WINDOW_UPDATE, and send FLOW_CONTROL_ERROR if the check fails.

⚠ Pitfall: WINDOW_UPDATE Frame Blocking A subtle deadlock occurs when WINDOW_UPDATE frames themselves become blocked by flow control windows. Since WINDOW_UPDATE frames are not subject to flow control, they must be transmitted even when DATA frame transmission is blocked. Some implementations incorrectly queue WINDOW_UPDATE frames behind DATA frames, creating unresolvable deadlocks. **Fix:** Always prioritize WINDOW_UPDATE frame transmission and send them immediately regardless of connection window state.

⚠ Pitfall: Memory Unbounded Queuing When implementing backpressure queuing, developers often create unbounded queues that grow without limit during sustained window exhaustion. This can cause

memory exhaustion and process crashes under load. **Fix:** Implement strict memory limits for queued frames and use priority-based eviction when approaching limits.

⚠ Pitfall: Stream Closure Window Handling When streams close, their unused window credit should be made available to other streams, but many implementations simply discard this credit. This wastes connection-level window capacity and reduces overall throughput. **Fix:** When a stream transitions to closed state, redistribute its unused window credit proportionally among remaining active streams.

Implementation Guidance

The flow control implementation requires sophisticated coordination between multiple components while maintaining thread safety and performance under high concurrency loads.

Technology Recommendations:

Component	Simple Option	Advanced Option
Window Tracking	In-memory maps with mutex protection	Lock-free atomic operations with memory barriers
Queue Management	Slice-based FIFO queues	Circular buffers with pre-allocated capacity
Priority Scheduling	Simple round-robin with weights	Deficit round-robin with priority inheritance
Memory Management	Manual queue size limits	Integrated with Go runtime GC pressure monitoring

Recommended File Structure:

```
internal/flow/
    flow_control.go      ← main flow controller
    window_manager.go   ← window tracking and updates
    backpressure_queue.go ← data frame queuing system
    scheduler.go         ← priority-based frame scheduling
    flow_control_test.go ← comprehensive test suite
internal/connection/
    connection.go        ← connection-level window integration
internal/stream/
    stream.go            ← stream-level window integration
```

Flow Control Infrastructure (Complete):

```
package flow

import (
    "errors"
    "sync"
    "time"
)

// WindowManager tracks flow control windows for connections and streams

type WindowManager struct {
    connectionSendWindow    int64
    connectionRecvWindow    int64
    streamWindows          map[uint32]*StreamWindows
    initialWindowSize       uint32
    maxWindowSize           int64
    mu                      sync.RWMutex
}

type StreamWindows struct {
    sendWindow    int64
    recvWindow    int64
    lastActivity  time.Time
}

type QueuedFrame struct {
    frame          *Frame
    timestamp     time.Time
    streamID      uint32
    originalSize   uint32
}
```

GO

```
    retryCount     int
}

type BackpressureQueue struct {
    streamQueues      map[uint32][]QueuedFrame
    blockedOnConnection []uint32
    blockedOnStream   []uint32
    totalQueuedBytes  int64
    maxQueueSize      int64
    mu                sync.Mutex
}

func NewWindowManager(initialSize uint32) *WindowManager {
    return &WindowManager{
        connectionSendWindow: int64(initialSize),
        connectionRecvWindow: int64(initialSize),
        streamWindows:         make(map[uint32]*StreamWindows),
        initialWindowSize:     initialSize,
        maxWindowSize:         (1 << 31) - 1,
    }
}

func NewBackpressureQueue(maxSize int64) *BackpressureQueue {
    return &BackpressureQueue{
        streamQueues:      make(map[uint32][]QueuedFrame),
        blockedOnConnection: make([]uint32, 0),
        blockedOnStream:   make([]uint32, 0),
        maxQueueSize:      maxSize,
    }
}
```

```

    }

}

func (wm *WindowManager) CreateStreamWindows(streamID uint32) {
    wm.mu.Lock()

    defer wm.mu.Unlock()

    wm.streamWindows[streamID] = &StreamWindows{
        sendWindow:   int64(wm.initialWindowSize),
        recvWindow:   int64(wm.initialWindowSize),
        lastActivity: time.Now(),
    }
}

func (wm *WindowManager) RemoveStreamWindows(streamID uint32) int64 {
    wm.mu.Lock()

    defer wm.mu.Unlock()

    if windows, exists := wm.streamWindows[streamID]; exists {
        unusedCredit := windows.sendWindow
        delete(wm.streamWindows, streamID)
        return unusedCredit
    }
    return 0
}

```

Core Flow Control Logic (Skeleton):

GO

```
// CanSendData checks if a DATA frame can be transmitted given current window state.

// Returns true only if both stream and connection windows have sufficient credit.

func (wm *WindowManager) CanSendData(streamID uint32, dataSize uint32) (bool, error) {

    // TODO 1: Acquire read lock for thread-safe window access

    // TODO 2: Validate that dataSize doesn't exceed maximum frame size

    // TODO 3: Check if stream windows exist for the given streamID

    // TODO 4: Verify stream send window has sufficient credit (>= dataSize)

    // TODO 5: Verify connection send window has sufficient credit (>= dataSize)

    // TODO 6: Return true only if both conditions are satisfied

    // Hint: Both windows must be checked atomically to prevent race conditions

}

// ConsumeWindows decrements both stream and connection send windows for DATA transmission.

// Must be called only after CanSendData returns true to maintain window consistency.

func (wm *WindowManager) ConsumeWindows(streamID uint32, dataSize uint32) error {

    // TODO 1: Acquire write lock for exclusive window modification

    // TODO 2: Double-check window availability (defensive programming)

    // TODO 3: Decrement stream send window by dataSize

    // TODO 4: Decrement connection send window by dataSize

    // TODO 5: Update stream lastActivity timestamp

    // TODO 6: Return error if either window would go negative

    // Hint: Use atomic decrements to prevent partial updates on error

}

// ProcessWindowUpdate applies WINDOW_UPDATE frame increments to appropriate windows.

// Handles both stream-specific updates (streamID > 0) and connection updates (streamID = 0).

func (wm *WindowManager) ProcessWindowUpdate(streamID uint32, increment uint32) error {
```

```

// TODO 1: Validate increment is positive (zero increments are protocol errors)

// TODO 2: Acquire write lock for window modifications

// TODO 3: If streamID == 0, apply increment to connection send window

// TODO 4: If streamID > 0, apply increment to specific stream send window

// TODO 5: Check for window overflow (current + increment > maxWindowSize)

// TODO 6: Apply the increment atomically to prevent partial updates

// TODO 7: Return FLOW_CONTROL_ERROR if overflow would occur

// Hint: Window updates can arrive for closed streams - handle gracefully

}

// QueueFrame adds a DATA frame to the backpressure queue when windows are exhausted.

// Implements memory limits and priority-based queueing to prevent unbounded growth.

func (bq *BackpressureQueue) QueueFrame(frame *Frame, streamID uint32, priority uint8)
error {

    // TODO 1: Check if adding this frame would exceed maxQueueSize limit

    // TODO 2: Create QueuedFrame with current timestamp and metadata

    // TODO 3: Insert frame into appropriate stream queue (FIFO ordering)

    // TODO 4: Update totalQueuedBytes counter for memory tracking

    // TODO 5: Add streamID to blocked list if not already present

    // TODO 6: Implement priority-based eviction if at capacity

    // Hint: Use priority to determine eviction candidates, not insertion order

}

// ProcessQueues attempts to transmit queued frames when window credit becomes available.

// Implements fair scheduling to prevent stream starvation during resumption.

func (bq *BackpressureQueue) ProcessQueues(wm *WindowManager) ([]QueuedFrame, error) {

    // TODO 1: Acquire locks on both queue and window manager

    // TODO 2: Scan blockedOnConnection list for streams with available credit

```

```

// TODO 3: For each unblocked stream, attempt to transmit queued frames

// TODO 4: Implement round-robin fairness across streams

// TODO 5: Remove successfully transmitted frames from queues

// TODO 6: Update blocked lists to reflect current stream states

// TODO 7: Return list of frames ready for transmission

// Hint: Process streams in priority order but limit frames per stream per round

}

// GenerateWindowUpdates creates WINDOW_UPDATE frames when receive windows fall below
// thresholds.

// Runs continuously to maintain adequate receive capacity for incoming DATA frames.

func (wm *WindowManager) GenerateWindowUpdates() ([]*Frame, error) {

    // TODO 1: Scan all stream receive windows for low-credit conditions

    // TODO 2: Check connection receive window against threshold

    // TODO 3: Calculate appropriate increment sizes (restore to initial size)

    // TODO 4: Create WINDOW_UPDATE frames for streams/connection needing updates

    // TODO 5: Update receive window values optimistically

    // TODO 6: Return frames ready for transmission

    // TODO 7: Handle case where stream closed during update generation

    // Hint: Use thresholds (25% for streams, 50% for connection) to trigger updates

}

```

Priority Scheduler (Complete):

```
package flow

import (
    "container/heap"
    "sort"
)

// PriorityScheduler implements weighted fair queuing for stream frame scheduling

type PriorityScheduler struct {

    streamWeights    map[uint32]uint8
    streamDeficits  map[uint32]int64
    quantumSize      int64
}

type ScheduledStream struct {

    streamID      uint32
    weight        uint8
    deficit       int64
    queueSize     int64
}

func NewPriorityScheduler(quantum int64) *PriorityScheduler {
    return &PriorityScheduler{
        streamWeights:  make(map[uint32]uint8),
        streamDeficits: make(map[uint32]int64),
        quantumSize:    quantum,
    }
}
```

GO

```
func (ps *PriorityScheduler) SetStreamPriority(streamID uint32, weight uint8) {  
  
    if weight == 0 {  
  
        weight = 1 // Minimum weight to prevent starvation  
  
    }  
  
    ps.streamWeights[streamID] = weight  
  
    ps.streamDeficits[streamID] = 0  
  
}  
  
func (ps *PriorityScheduler) ScheduleTransmission(availableWindow int64, queuedStreams map[uint32]int64) []uint32 {  
  
    var scheduled []ScheduledStream  
  
  
    // Build scheduling candidates  
  
    for streamID, queueSize := range queuedStreams {  
  
        weight := ps.streamWeights[streamID]  
  
        if weight == 0 {  
  
            weight = 1  
  
        }  
  
        scheduled = append(scheduled, ScheduledStream{  
  
            streamID: streamID,  
  
            weight: weight,  
  
            deficit: ps.streamDeficits[streamID],  
  
            queueSize: queueSize,  
  
        })  
  
    }  
  
  
    // Sort by deficit (highest deficit first for fairness)  

```

```
sort.Slice(scheduled, func(i, j int) bool {
    return scheduled[i].deficit > scheduled[j].deficit
})

var result []uint32
remainingWindow := availableWindow

// Deficit round-robin scheduling
for remainingWindow > 0 && len(scheduled) > 0 {
    for i := 0; i < len(scheduled) && remainingWindow > 0; i++ {
        stream := &scheduled[i]

        // Add quantum based on weight
        stream.deficit += int64(stream.weight) * ps.quantumSize

        // Allow transmission if deficit is positive and window available
        if stream.deficit > 0 && stream.queueSize > 0 {
            transmitSize := min(stream.deficit, remainingWindow, stream.queueSize)

            stream.deficit -= transmitSize
            remainingWindow -= transmitSize
            stream.queueSize -= transmitSize

            result = append(result, stream.streamID)
        }
    }
}

// Update persistent deficit
ps.streamDeficits[stream.streamID] = stream.deficit
}
```

```

    }

    // Remove streams with no remaining queue

    newScheduled := scheduled[:0]

    for _, stream := range scheduled {

        if stream.queueSize > 0 {

            newScheduled = append(newScheduled, stream)

        }

    }

    scheduled = newScheduled

}

return result
}

func min(a, b, c int64) int64 {

    if a <= b && a <= c {

        return a

    }

    if b <= c {

        return b

    }

    return c
}

```

Milestone Checkpoint:

After implementing flow control:

- Window Tracking Test:** Run `go test -run TestWindowManagement` - should show proper window decrements for DATA frames and increments for WINDOW_UPDATE frames
- Backpressure Test:** Send large file through server with small initial windows - should queue frames and resume transmission when windows update
- Manual Verification:** Use `curl --http2` to download large file - monitor server logs for window exhaustion and recovery
- Expected Behavior:** Server should handle multiple concurrent streams without window deadlocks, maintaining fairness across streams

Debugging Tips:

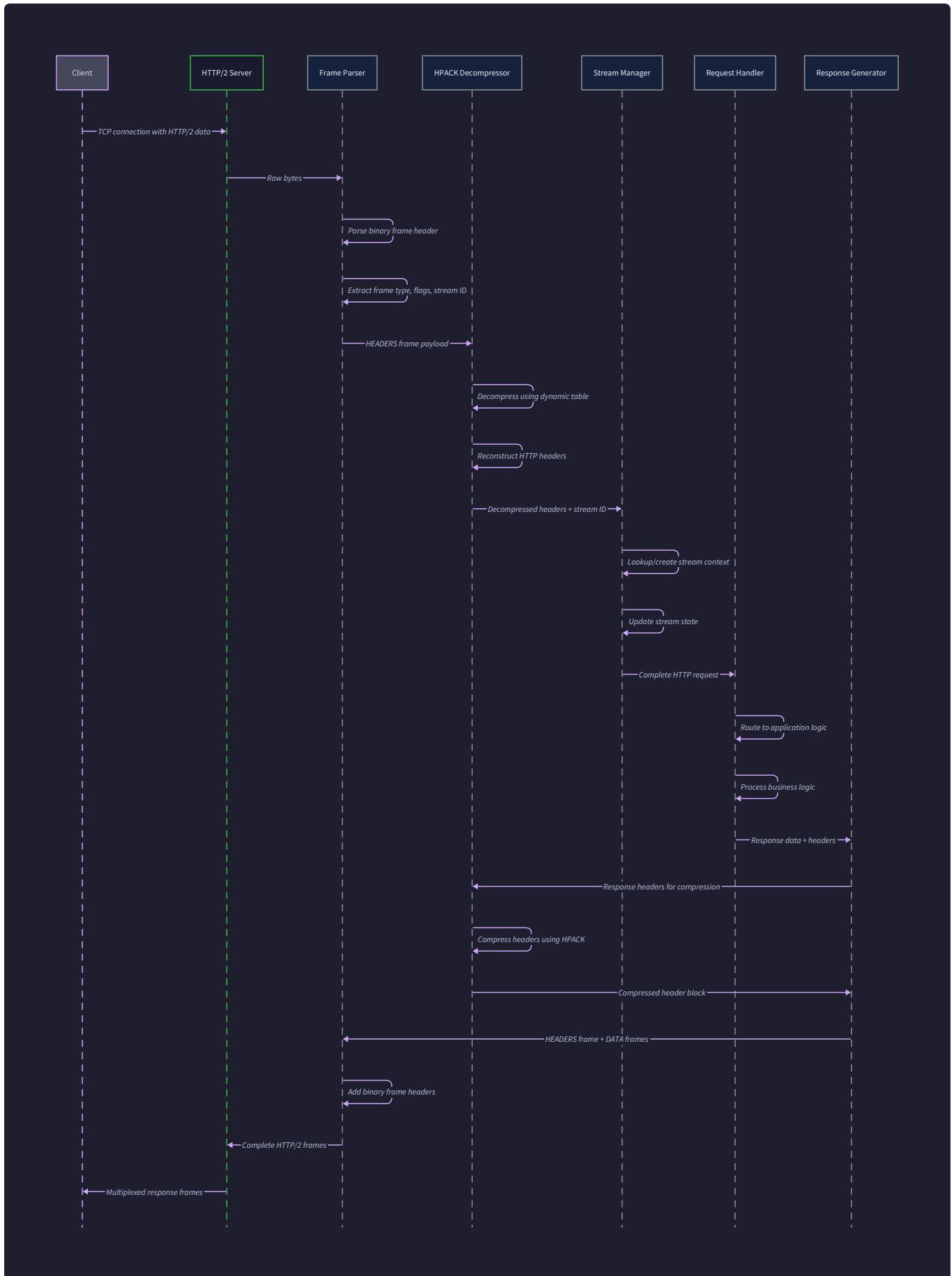
Symptom	Likely Cause	How to Diagnose	Fix
Streams hang mid-transfer	Window exhaustion without updates	Log window values before DATA transmission	Implement automatic WINDOW_UPDATE generation
Memory usage grows unbounded	Unlimited backpressure queuing	Monitor totalQueuedBytes metric	Add strict queue size limits with eviction
Some streams never get bandwidth	Priority inversion in scheduling	Log stream scheduling order and window allocation	Implement deficit round-robin scheduling
Connection terminates unexpectedly	Window overflow from large WINDOW_UPDATE	Validate increment + current \leq maxWindowSize	Add overflow checking before applying updates

Interactions and Data Flow

Milestone(s): This section synthesizes concepts from all milestones (1-4) by demonstrating how the Binary Framing Engine (Milestone 1), HPACK Compression Engine (Milestone 2), Stream Management (Milestone 3), and Flow Control (Milestone 4) work together during actual HTTP/2 request processing, frame routing, and header decompression operations.

Think of an HTTP/2 server as a sophisticated air traffic control system managing multiple concurrent flights (streams) over a single runway (TCP connection). Just as air traffic control receives flight plans, coordinates takeoffs and landings, and manages airspace capacity, our HTTP/2 server receives binary frames, coordinates stream multiplexing, and manages flow control windows. The key insight is that unlike HTTP/1.1's sequential processing (one plane at a time), HTTP/2 enables true multiplexing where multiple conversations happen simultaneously without head-of-line blocking.

This section explores the intricate dance between our four core engines as they collaborate to process HTTP/2 requests. We'll trace the journey of a frame from its arrival as raw bytes through binary parsing, stream routing, header decompression, and response generation. Understanding these interactions is crucial because HTTP/2's power comes not from individual components but from their seamless orchestration.



Request Processing Flow

The request processing flow represents the end-to-end journey from frame reception to HTTP response generation. Think of this as a factory assembly line where each station (component) performs specialized work before passing the product to the next station. However, unlike a traditional assembly line, our HTTP/2 factory processes multiple product lines (streams) simultaneously, with sophisticated routing and quality control at each stage.

The complexity lies in maintaining state consistency across components while handling the inherent asynchrony of multiplexed streams. Each frame arriving on the connection could belong to any active stream, carry different types of data (headers, payload, control information), and trigger cascading effects throughout the system.

Frame Reception and Initial Processing

The journey begins when raw bytes arrive on the TCP connection. The Connection Handler acts as the factory's receiving dock, accepting delivery trucks (TCP segments) and sorting their contents into the appropriate processing queues. This initial stage must be extremely efficient because it directly impacts the server's ability to handle high connection counts.

The Connection Handler maintains a persistent read loop that continuously monitors the TCP socket for incoming data. When bytes arrive, the handler immediately invokes the Binary Framing Engine's `ParseFrame` method. This creates our first critical interaction point where transport-layer concerns meet protocol-layer parsing.

Step	Component	Action	State Changes	Error Conditions
1	Connection Handler	Read bytes from TCP socket	Update connection read buffer	Network errors, connection closure
2	Binary Framing Engine	Parse 9-byte frame header	Extract length, type, flags, stream ID	Malformed header, oversized frame
3	Binary Framing Engine	Validate frame structure	Check reserved bits, length limits	Protocol violations, invalid frame types
4	Binary Framing Engine	Read frame payload	Extract payload bytes based on length	Incomplete reads, payload corruption
5	Frame Router	Determine destination component	Route based on frame type and stream ID	Invalid stream ID, closed streams

Critical Insight: Frame parsing must be atomic and non-blocking. If the Binary Framing Engine encounters an incomplete frame (not enough bytes available), it must preserve the partial state and resume parsing when more data arrives, without blocking other streams on the connection.

The Binary Framing Engine performs immediate validation of the parsed frame, checking that reserved bits are zero, the frame length doesn't exceed `MaxFrameSize`, and the frame type is recognized. This validation acts as a firewall protecting downstream components from malformed data that could cause crashes or security vulnerabilities.

Stream Identification and Routing

Once a valid frame is parsed, the Frame Router must determine which component should handle it. This routing decision depends on both the frame type and the stream ID. Think of the Frame Router as a sophisticated mail sorting facility that must deliver each piece of mail to the correct department based on both the address and the type of correspondence.

The routing logic follows a hierarchical decision tree. Connection-level frames (stream ID 0) such as `SETTINGS`, `PING`, and `GOAWAY` are handled directly by the Connection Handler. Stream-specific frames are routed to the appropriate stream handler within the Stream Manager. However, this seemingly simple routing masks significant complexity in edge case handling.

Decision: Stream ID Validation Strategy

- **Context:** Invalid or closed stream IDs require different error responses than valid streams
- **Options Considered:** 1) Validate before routing 2) Let stream handlers validate 3) Hybrid validation
- **Decision:** Hybrid validation with router checking basic validity and stream handlers checking state-specific rules
- **Rationale:** Balances performance (early rejection of obviously invalid frames) with flexibility (stream handlers understand their own state)
- **Consequences:** Requires careful coordination between router and stream managers to avoid duplicate validation overhead

Frame Type	Stream ID	Routing Destination	Validation Requirements
<code>SETTINGS</code>	0	Connection Handler	Must be connection-level, ACK flag validation
<code>HEADERS</code>	> 0	Stream Manager	Stream must exist or be creatable
<code>DATA</code>	> 0	Stream Manager	Stream must be in open or half-closed-remote state
<code>WINDOW_UPDATE</code>	0 or > 0	Flow Controller	Window increment must be non-zero
<code>RST_STREAM</code>	> 0	Stream Manager	Stream must exist (may be in any state)
<code>PING</code>	0	Connection Handler	Payload must be exactly 8 bytes

The Stream Manager maintains a registry of active streams using a map from stream ID to `Stream` objects. When a frame arrives for a specific stream, the router performs a lookup in this registry. If the stream exists,

the frame is dispatched to the stream's handler. If the stream doesn't exist but the frame type is `HEADERS` with appropriate flags, a new stream may be created.

Header Processing and HPACK Decompression

When `HEADERS` frames arrive at the Stream Manager, they trigger one of the most complex interaction sequences in the entire system. Header frames don't just carry HTTP headers—they carry HPACK-compressed header blocks that must be decompressed using connection-shared state in the HPACK Compression Engine.

Think of HPACK decompression as a conversation between old friends who share inside jokes and shorthand references. The HPACK tables act as this shared context, allowing headers to be referenced by index numbers rather than spelling out full header names and values. However, this shared state creates a critical dependency: header frames for different streams must be processed sequentially to maintain table consistency, even though the streams themselves are independent.

The Stream Manager coordinates this by maintaining a header decompression queue. When `HEADERS` frames arrive (including potential `CONTINUATION` frames), they're queued for sequential processing by the HPACK Compression Engine. This ensures that dynamic table updates from one stream's headers are visible to subsequent streams while preventing race conditions that could corrupt the shared tables.

Decompression Step	Component Interaction	State Updates	Error Handling
1. Frame Queuing	Stream Manager → HPACK Engine	Add to decompression queue	Queue overflow, memory limits
2. Table Lookup	HPACK Engine internal	Access static/dynamic tables	Invalid index references
3. Huffman Decoding	HPACK Engine internal	Decode compressed strings	Malformed Huffman sequences
4. Dynamic Table Update	HPACK Engine internal	Add new entries, evict old ones	Table size violations
5. Header Reconstruction	HPACK Engine → Stream Manager	Return decoded header list	Incomplete header blocks

⚠ Pitfall: `CONTINUATION` Frame Handling A common mistake is treating each `HEADERS` frame as complete. In reality, large header blocks may span multiple frames, with intermediate `CONTINUATION` frames carrying additional compressed data. The Stream Manager must buffer these fragments and only invoke HPACK decompression once the complete header block (indicated by the `FlagEndHeaders` flag) is received. Attempting to decompress partial header blocks will corrupt the HPACK dynamic table state.

Request Assembly and HTTP Semantics

After successful header decompression, the Stream Manager must assemble the HTTP request from the decoded headers and any associated `DATA` frames. This step bridges the gap between HTTP/2's binary framing layer and traditional HTTP semantics that application handlers expect.

The assembly process involves reconstructing pseudo-headers (`:method`, `:path`, `:scheme`, `:authority`) into their HTTP/1.1 equivalents and validating that the request conforms to HTTP semantics. For example, `GET` requests should not have request bodies, while `POST` requests often do.

HTTP/2 Headers: <code>:method: GET</code> <code>:path: /api/users</code> <code>:scheme: https</code> <code>:authority: example.com</code> <code>authorization: Bearer token123</code>	Assembled HTTP Request: <code>GET /api/users HTTP/1.1</code> <code>Host: example.com</code> <code>Authorization: Bearer token123</code>
---	---

The Stream Manager creates an internal HTTP request representation that can be passed to application-level handlers. This abstraction allows existing HTTP/1.1 applications to work with HTTP/2 with minimal modification, since they receive familiar HTTP request objects rather than raw frames and headers.

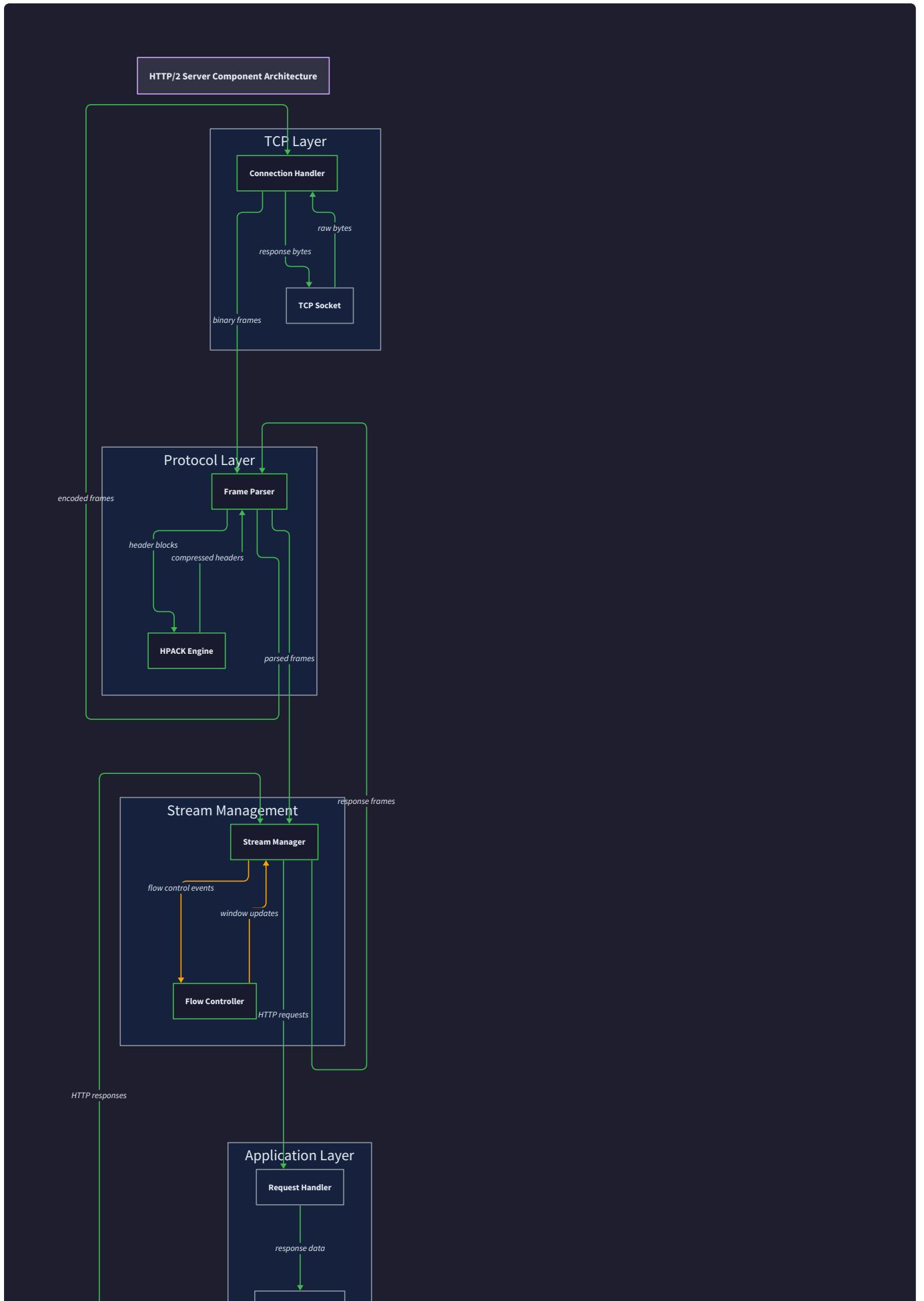
Response Generation and Framing

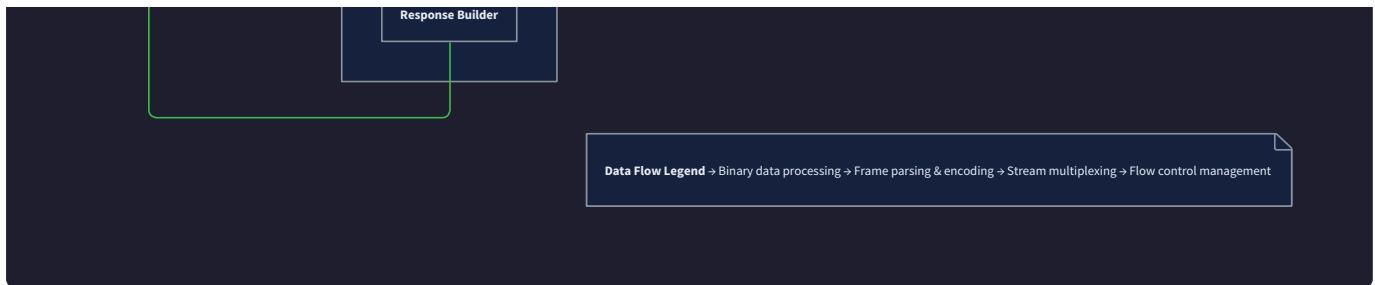
Response generation reverses the inbound processing pipeline. Application handlers generate HTTP responses using familiar APIs, which the Stream Manager must convert back into HTTP/2 frames. This involves header compression, frame serialization, and coordination with flow control mechanisms.

The response processing pipeline demonstrates the bidirectional nature of component interactions. While inbound processing flows from Connection Handler through Frame Router to Stream Manager to application, outbound processing must coordinate between application, Stream Manager, HPACK Engine, Flow Controller, and Binary Framing Engine.

Response Step	Component Flow	Data Transformation	Synchronization Points
1. Handler Response	Application → Stream Manager	HTTP response object	None
2. Header Compression	Stream Manager → HPACK Engine	HTTP headers → HPACK block	Sequential compression queue
3. Header Framing	HPACK Engine → Binary Framing Engine	Compressed headers → HEADERS frame	Frame serialization
4. Flow Control Check	Stream Manager → Flow Controller	Check send windows	Window availability
5. Data Framing	Stream Manager → Binary Framing Engine	Response body → DATA frames	Frame size limits
6. Transmission	Binary Framing Engine → Connection Handler	Frames → TCP bytes	Network buffers

Critical Design Point: Response generation must coordinate with flow control to avoid violating receiver windows. The Stream Manager cannot simply generate all response frames immediately—it must check both stream-level and connection-level send windows before creating `DATA` frames. If windows are exhausted, the response data must be queued until `WINDOW_UPDATE` frames provide additional send capacity.





Frame Routing

Frame routing represents the nervous system of our HTTP/2 server, directing incoming frames to appropriate handlers based on type, stream ID, and current connection state. Think of the Frame Router as a sophisticated dispatcher in an emergency response system—it must instantly categorize incoming calls (frames) and route them to the correct department (component) while handling edge cases like wrong numbers, prank calls, and emergencies.

The routing challenge stems from HTTP/2's multiplexed nature. Unlike HTTP/1.1 where each connection carries a single request-response sequence, HTTP/2 connections interleave frames from multiple streams with control frames that affect the entire connection. The router must maintain awareness of connection state, stream lifecycle, and protocol constraints while making routing decisions at high speed.

Routing Decision Tree

The Frame Router implements a hierarchical decision process that prioritizes correctness over raw performance. Each routing decision involves multiple validation steps that can result in different error responses depending on the specific violation detected.

The primary routing decision depends on the stream ID field in the frame header. Stream ID zero indicates connection-level frames that affect the entire connection's state or configuration. Non-zero stream IDs indicate frames belonging to specific request-response streams. However, this simple distinction masks considerable complexity in validation and error handling.

Stream ID	Frame Types	Routing Destination	Validation Requirements
0	SETTINGS, PING, GOAWAY, WINDOW_UPDATE	Connection Handler	Connection-level frame validation
> 0	HEADERS, DATA, RST_STREAM, WINDOW_UPDATE, PUSH_PROMISE	Stream Manager	Stream existence and state validation
Invalid	Any	Error Handler	Protocol error response generation

Decision: Routing Table vs. Switch Statement

- **Context:** Frame routing needs to be both fast and maintainable as new frame types are added
- **Options Considered:** 1) Map-based routing table 2) Switch statement 3) Interface-based polymorphism
- **Decision:** Switch statement with validation helper functions
- **Rationale:** Go's switch statements compile to efficient jump tables, validation logic can be shared, and adding new frame types is straightforward
- **Consequences:** Slightly more verbose than a map but significantly faster, with better compile-time error detection

The router must also validate frame type compatibility with stream ID constraints. For example, `SETTINGS` frames are only valid with stream ID 0, while `DATA` frames are only valid with non-zero stream IDs. Violating these constraints results in connection-level `ProtocolError`s that should terminate the connection.

Stream State Awareness

Stream routing requires awareness of stream lifecycle state because the same frame type may be valid or invalid depending on the current stream state. The Frame Router maintains a lightweight view of stream states to enable early validation without duplicating the full state machine logic present in individual stream handlers.

Think of this as a bouncer at an exclusive club who knows the membership list and current occupancy but doesn't need to know everyone's personal details. The router knows whether stream 42 exists and its basic state (open, closed, etc.) but delegates detailed state transitions to the Stream Manager.

The router's stream state cache must be kept synchronized with the authoritative state maintained by the Stream Manager. This synchronization happens through event notifications when streams change state, ensuring the router's cache remains consistent without requiring expensive lookups on every frame.

Stream State	Valid Frame Types	Router Actions	Invalid Frame Handling
<code>StateIdle</code>	<code>HEADERS</code> (creates stream)	Route to Stream Manager for creation	Send <code>RST_STREAM</code> for other types
<code>StateOpen</code>	<code>HEADERS</code> , <code>DATA</code> , <code>RST_STREAM</code> , <code>WINDOW_UPDATE</code>	Route to existing stream handler	Validate flags and dependencies
<code>StateClosed</code>	<code>WINDOW_UPDATE</code> , <code>RST_STREAM</code> (in some cases)	Limited routing or ignore	Most frames should be ignored
Nonexistent	<code>HEADERS</code> (if stream creation allowed)	Route for potential stream creation	<code>ProtocolError</code> for most frame types

⚠ Pitfall: Stream State Synchronization A subtle bug occurs when the router's stream state cache becomes inconsistent with the Stream Manager's authoritative state. This typically happens during rapid

stream state transitions where the router continues routing frames to a stream that has just transitioned to closed state. The fix involves using atomic updates and ensuring that state change notifications are processed before routing additional frames for the affected stream.

Priority and Dependency Routing

Stream priority and dependency information affects routing decisions for `HEADERS` and `DATA` frames. While the router doesn't implement the full priority scheduling algorithm (that's the Flow Controller's responsibility), it must validate priority information and route frames to the appropriate priority queue.

Priority routing introduces ordering constraints that can create head-of-line blocking if not handled carefully. The router must balance respecting priority relationships with avoiding deadlocks where high-priority streams block the processing of control frames needed to make progress.

The router implements priority-aware routing by maintaining separate queues for different priority levels and ensuring that control frames (like `WINDOW_UPDATE`) are always processed promptly regardless of data frame backlog in priority queues.

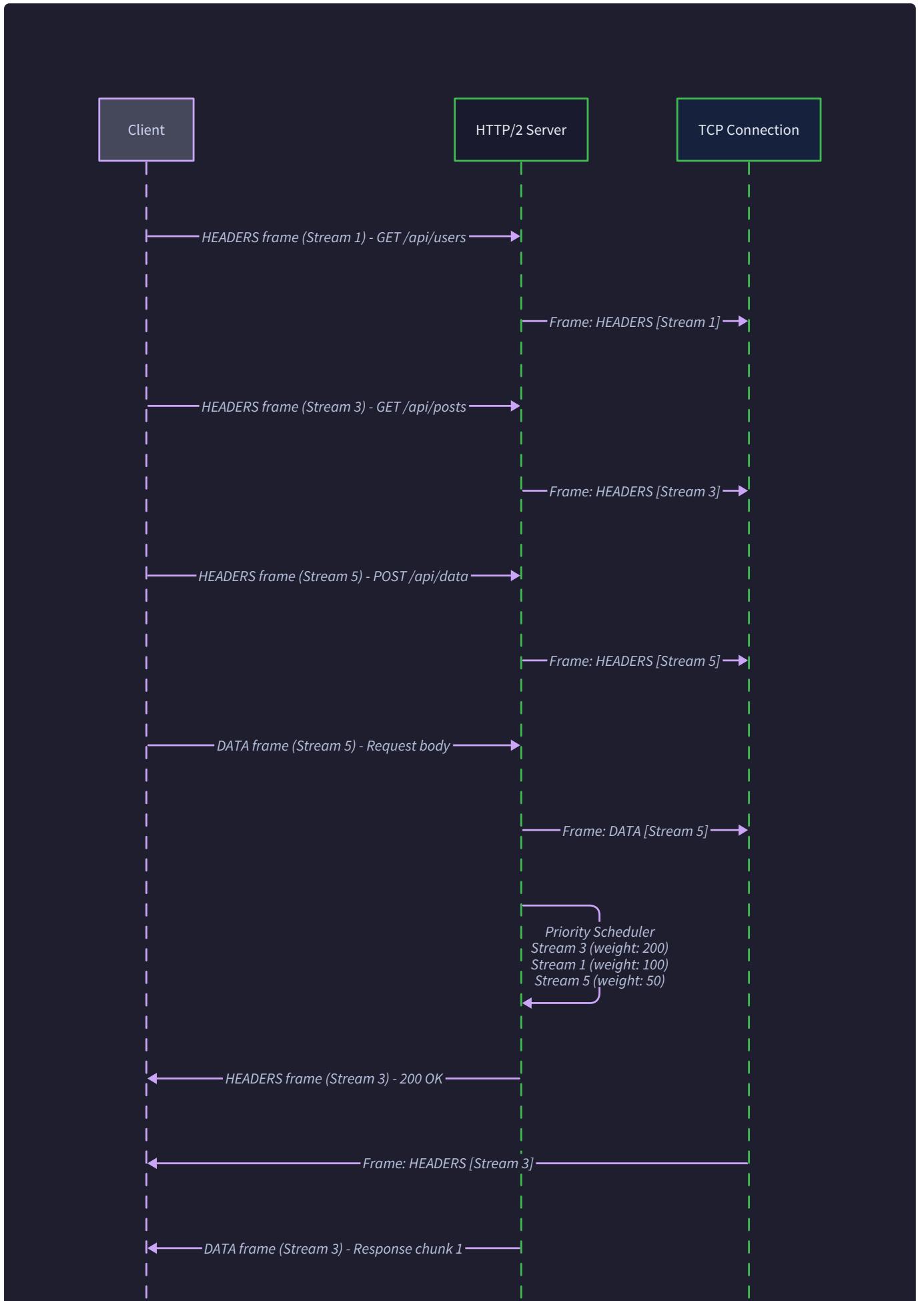
Error Frame Generation

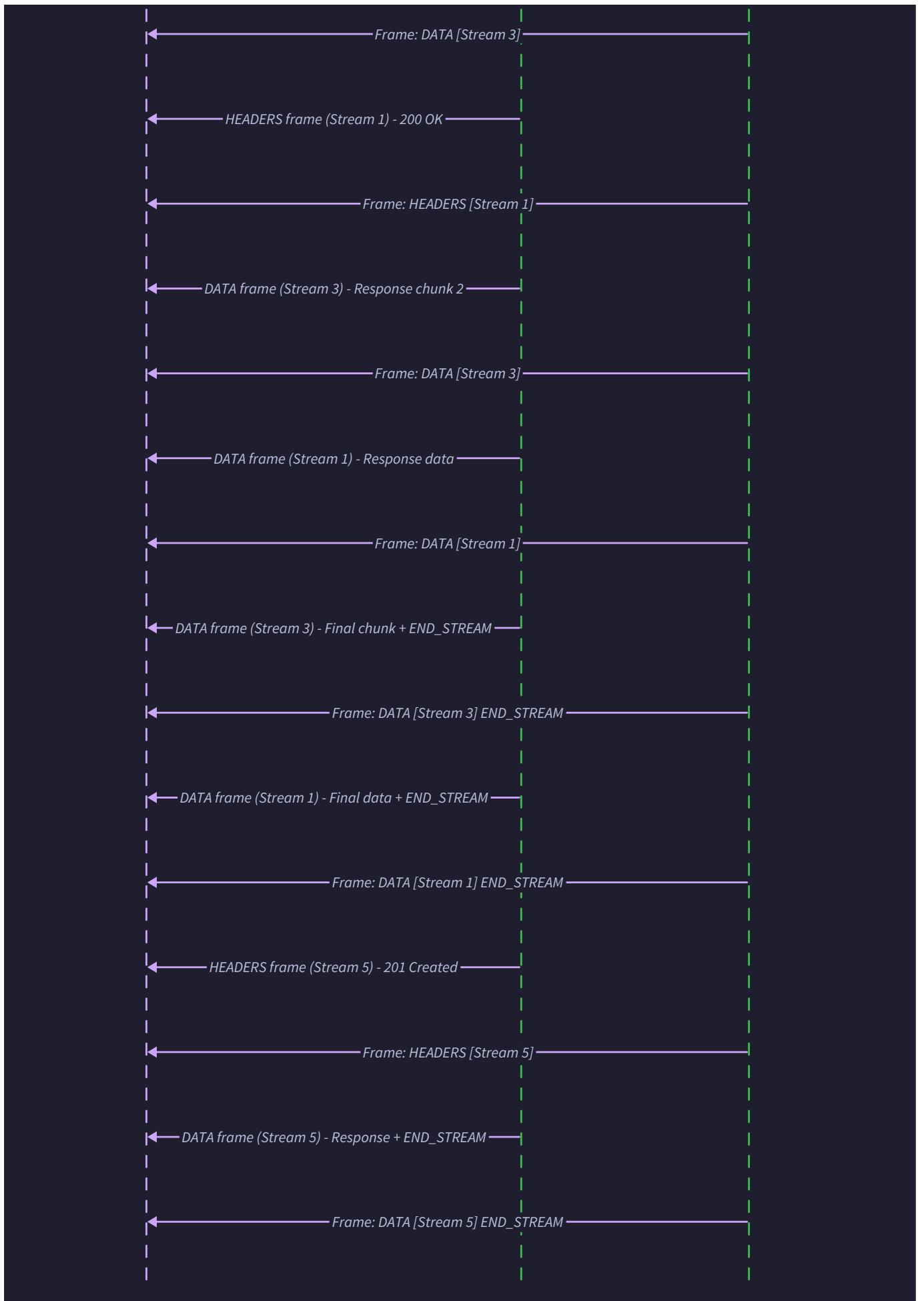
When the router detects protocol violations or invalid frame combinations, it must generate appropriate error responses. The error response type depends on whether the violation affects a specific stream or the entire connection.

Stream-level errors result in `RST_STREAM` frames that terminate the offending stream while allowing other streams on the connection to continue. Connection-level errors result in `GOAWAY` frames that initiate connection shutdown, preventing new streams while allowing existing streams to complete gracefully.

Error Type	Detection Point	Response Frame	Affected Scope
Invalid stream ID	Stream ID validation	<code>RST_STREAM</code>	Single stream
Frame size exceeded	Frame length validation	<code>GOAWAY</code>	Entire connection
Invalid frame type	Frame type validation	<code>RST_STREAM</code> or <code>GOAWAY</code>	Depends on severity
State machine violation	Stream state check	<code>RST_STREAM</code>	Single stream
Flow control violation	Window validation	<code>RST_STREAM</code> or <code>GOAWAY</code>	Depends on window type

The router must generate these error frames promptly to maintain protocol compliance, but it must also coordinate with the Flow Controller to ensure error frames don't violate flow control windows. This creates a chicken-and-egg problem where flow control violations prevent sending flow control error notifications.





Header Decompression Flow

Header decompression represents one of the most intricate interaction patterns in the HTTP/2 server because it requires coordination between frame-level processing, connection-shared state, and stream-specific request assembly. Think of HPACK decompression as a sophisticated library system where books (headers) can be referenced by catalog numbers (indices) instead of full titles, but the catalog is shared among all library patrons (streams) and must be updated consistently.

The complexity stems from HPACK's stateful nature. Unlike simple compression schemes where each block can be decompressed independently, HPACK maintains dynamic tables that evolve as headers are processed. This creates a critical constraint: header blocks from different streams must be decompressed sequentially to maintain table consistency, even though the streams themselves operate independently.

Header Block Assembly

Before decompression can begin, the Stream Manager must assemble complete header blocks from potentially fragmented frames. HTTP/2 allows large header sets to span multiple frames using `HEADERS` followed by `CONTINUATION` frames. The assembly process must handle partial receives, frame ordering, and error recovery while maintaining the sequential processing constraint.

Think of this assembly process as reconstructing a jigsaw puzzle where pieces (frames) arrive out of order, and you can't start working on the next puzzle until the current one is complete. The Stream Manager maintains assembly buffers for each stream that's currently receiving headers, tracking which streams have complete header blocks ready for decompression.

Assembly Step	Stream Manager Action	Buffer Management	Error Conditions
1. <code>HEADERS</code> frame arrives	Create assembly buffer for stream	Allocate header fragment buffer	Memory allocation failure
2. Validate header flags	Check for <code>END_HEADERS</code> flag	Determine if more frames expected	Invalid flag combinations
3. <code>CONTINUATION</code> frames	Append to assembly buffer	Grow buffer as needed	Buffer size limits exceeded
4. Complete block detected	Mark buffer ready for decompression	Queue for HPACK processing	Header block too large
5. Submit to HPACK engine	Transfer assembled block	Clear assembly buffer	HPACK engine queue full

Critical Timing Constraint: Header block assembly must respect the HTTP/2 specification requirement that `CONTINUATION` frames immediately follow their associated `HEADERS` frame without intervening frames for other streams. The Stream Manager enforces this by tracking the "continuation expected" state and rejecting non-continuation frames when a header block is incomplete.

The assembly process must also handle error recovery when header blocks are malformed or incomplete. If a stream fails to send the expected `CONTINUATION` frames, the assembly buffer eventually times out and the incomplete header block is discarded, generating a stream error.

⚠ Pitfall: Memory Exhaustion in Assembly Buffers A malicious client can attempt denial-of-service attacks by starting many header blocks but never completing them with `CONTINUATION` frames. This causes assembly buffers to accumulate indefinitely. The mitigation involves setting strict limits on the number of concurrent incomplete header blocks and the maximum size of any individual assembly buffer, with timeout-based cleanup of abandoned header blocks.

Sequential Decompression Processing

Once header blocks are assembled, they enter a strict sequential processing queue maintained by the HPACK Compression Engine. This queue ensures that dynamic table updates occur in the correct order, maintaining the shared decompression context required for subsequent header blocks.

The sequential processing creates a potential bottleneck in highly multiplexed scenarios where many streams are sending headers simultaneously. However, this constraint is fundamental to HPACK's design and cannot be eliminated without violating the compression algorithm's requirements.

The HPACK Compression Engine processes the queue using a single-threaded worker that decompresses header blocks one at a time. Each decompression operation may modify the dynamic table by adding new entries or triggering evictions, so these operations cannot be parallelized without sophisticated synchronization mechanisms that would likely eliminate any performance benefits.

Queue Operation	HPACK Engine State	Dynamic Table Changes	Completion Notification
Dequeue header block	Lock decompression context	Potential table modifications	Signal to originating stream
Process HPACK instructions	Update table indices	Add entries, evict old ones	Update compression statistics
Decode Huffman strings	Access Huffman decoder	No table changes	String reconstruction
Validate header semantics	Check HTTP/2 requirements	No table changes	Validation error or success
Return decoded headers	Unlock decompression context	Finalize table state	Stream can proceed with request

Decision: Decompression Queue Implementation

- **Context:** Sequential processing requirement conflicts with high concurrency goals
- **Options Considered:** 1) Single-threaded queue 2) Connection-specific queues 3) Lock-free algorithms
- **Decision:** Single-threaded queue with backpressure signals
- **Rationale:** Simplest correct implementation, matches HPACK's inherent sequential requirement, backpressure prevents queue overflow
- **Consequences:** Potential bottleneck under high header load, but guaranteed correctness and simpler debugging

Dynamic Table Management During Decompression

Each header block decompression potentially modifies the dynamic table through literal header insertions. The HPACK Compression Engine must coordinate these modifications with ongoing compression operations for outbound responses, ensuring that the table state remains consistent for both directions of the connection.

The dynamic table acts as a circular buffer with FIFO eviction when new entries would exceed the configured table size. During decompression, new entries are added to the beginning of the table, and existing entries are shifted to higher indices. If the table becomes full, entries at the highest indices are evicted to make room.

This table management must be atomic with respect to the decompression operation. If decompression fails partway through a header block (due to invalid HPACK instructions or Huffman decoding errors), any dynamic table modifications must be rolled back to maintain consistency with the sender's table state.

Table Operation	Before State	During Processing	After State	Rollback Requirements
Add literal entry	Current table size: N	Tentative insertion at index 0	Size: N+1, indices shifted	Remove added entry
Evict old entries	Table at max size	Remove highest indices	Size unchanged, new entry added	Restore evicted entries
Index existing entry	Table unchanged	Lookup by index	Table unchanged	No rollback needed
Huffman decode failure	Partial new entry	Decoder error state	Table unchanged	Discard partial entry

Error Recovery and State Synchronization

Header decompression errors require careful handling because they can desynchronize the dynamic table state between client and server, potentially corrupting all subsequent header processing on the connection. The HPACK specification requires that certain decompression errors be treated as connection errors rather than stream errors.

When the HPACK Compression Engine encounters decompression errors, it must determine whether the error affects only the current stream or potentially corrupts the shared decompression context. Invalid table indices or malformed integer encodings typically indicate connection-level problems that require terminating the connection with a `GOAWAY` frame.

The error recovery process involves coordinating between the HPACK Compression Engine, Stream Manager, and Connection Handler to ensure that error responses are generated promptly and that no subsequent header processing occurs on a potentially corrupted connection.

Critical Recovery Principle: When in doubt about dynamic table consistency, terminate the connection. The cost of debugging subtle compression state corruption far exceeds the cost of occasional false-positive connection terminations. Recovery mechanisms should err on the side of caution to maintain overall system stability.

Implementation Guidance

This implementation guidance provides concrete Go code for implementing the interaction patterns described in the design. The code focuses on the coordination mechanisms between components rather than the internal implementation of individual engines, which are covered in their respective sections.

Technology Recommendations

Component	Simple Option	Advanced Option
Frame Routing	Switch statement with validation functions	Interface-based routing with reflection
Header Assembly	Sync.Map for stream buffers	Lock-free concurrent map
Decompression Queue	Buffered channel with single goroutine	Work-stealing queue with multiple workers
Error Handling	Error return values with error wrapping	Structured error types with recovery hints
State Synchronization	Mutex-protected shared state	Lock-free atomic operations

Recommended File Structure

```
internal/
  server/
    server.go          ← main server coordination
    connection_handler.go ← TCP connection management
    frame_router.go     ← frame routing logic
  interactions/
    request_processor.go ← end-to-end request processing
    header_assembler.go   ← header block assembly
    error_coordinator.go  ← error handling coordination
    flow_coordinator.go   ← flow control interactions
  testing/
    interaction_test.go  ← integration tests
    mock_components.go    ← test doubles for components
```

Infrastructure Starter Code

```
// Package interactions coordinates between HTTP/2 server components  
GO  
  
package interactions  
  
import (  
    "bufio"  
    "context"  
    "fmt"  
    "io"  
    "net"  
    "sync"  
    "time"  
)  
  
// RequestProcessor coordinates end-to-end request processing across all components  
  
type RequestProcessor struct {  
  
    frameRouter      *FrameRouter  
  
    streamManager   *StreamManager  
  
    hpackEngine     *HpackEngine  
  
    flowController  *WindowManager  
  
    connHandler     *Connection  
  
    // Processing coordination  
  
    headerQueue     chan *HeaderBlock  
  
    errorChan       chan error  
  
    shutdownChan    chan struct{  
    }  
  
    wg              sync.WaitGroup  
}  
}
```

```
// HeaderBlock represents an assembled header block ready for decompression

type HeaderBlock struct {

    StreamID      uint32

    Headers       []byte

    EndStream     bool

    Priority      *StreamPriority

    Assembly      time.Time

}

// FrameRouter dispatches frames to appropriate components based on type and stream ID

type FrameRouter struct {

    streamStates    map[uint32]StreamState

    statesMutex     sync.RWMutex

    connHandler     *Connection

    streamManager   *StreamManager

    flowController  *WindowManager


    // Routing statistics

    frameCount     map[uint8]uint64 // frames processed by type

    errorCount     map[ErrorCode]uint64

    statsMutex     sync.RWMutex

}

// HeaderAssembler manages fragmented header blocks across HEADERS and CONTINUATION frames

type HeaderAssembler struct {

    assemblyBuffers map[uint32]*AssemblyBuffer

    buffersMutex   sync.RWMutex
```

```
maxBufferSize  uint32
maxBuffers      int
timeout         time.Duration
}

// AssemblyBuffer tracks partial header block assembly for a specific stream

type AssemblyBuffer struct {
    streamID      uint32
    fragments     [][]byte
    totalSize     uint32
    expectingCont bool
    created       time.Time
    lastUpdate    time.Time
}

// NewRequestProcessor creates a processor that coordinates all HTTP/2 components

func NewRequestProcessor(conn net.Conn, limits ResourceLimits) *RequestProcessor {
    connHandler := NewConnection(conn)

    return &RequestProcessor{
        frameRouter:      NewFrameRouter(connHandler, limits),
        streamManager:   &StreamManager{},
        hpackEngine:     &HpackEngine{},
        flowController:  &WindowManager{},
        connHandler:     connHandler,
        headerQueue:     make(chan *HeaderBlock, 100),
        errorChan:       make(chan error, 10),
        shutdownChan:    make(chan struct{})

```
maxBufferSize uint32
maxBuffers int
timeout time.Duration
}

// AssemblyBuffer tracks partial header block assembly for a specific stream

type AssemblyBuffer struct {
 streamID uint32
 fragments [][]byte
 totalSize uint32
 expectingCont bool
 created time.Time
 lastUpdate time.Time
}

// NewRequestProcessor creates a processor that coordinates all HTTP/2 components

func NewRequestProcessor(conn net.Conn, limits ResourceLimits) *RequestProcessor {
 connHandler := NewConnection(conn)

 return &RequestProcessor{
 frameRouter: NewFrameRouter(connHandler, limits),
 streamManager: &StreamManager{},
 hpackEngine: &HpackEngine{},
 flowController: &WindowManager{},
 connHandler: connHandler,
 headerQueue: make(chan *HeaderBlock, 100),
 errorChan: make(chan error, 10),
 shutdownChan: make(chan struct{})
```


```

```
    }

}

// ProcessConnection runs the main request processing loop

func (rp *RequestProcessor) ProcessConnection(ctx context.Context) error {

    // Start header decompression worker

    rp.wg.Add(1)

    go rp.headerDecompressionWorker(ctx)

    // Start error handling coordinator

    rp.wg.Add(1)

    go rp.errorCoordinator(ctx)

    // Main frame processing loop

    for {

        select {

        case <-ctx.Done():

            return ctx.Err()

        case <-rp.shutdownChan:

            rp.wg.Wait()

            return nil

        default:

            if err := rp.processNextFrame(ctx); err != nil {

                rp.errorChan <- err

                if isConnectionError(err) {

                    return err

                }

            }

        }

    }

}
```

```
        }

    }

}

// NewFrameRouter creates a router that dispatches frames to appropriate handlers

func NewFrameRouter(conn *Connection, limits ResourceLimits) *FrameRouter {

    return &FrameRouter{

        streamStates: make(map[uint32]StreamState),

        connHandler: conn,

        frameCount: make(map[uint8]uint64),

        errorCount: make(map[ErrorCode]uint64),

    }

}

// NewHeaderAssembler creates an assembler for fragmented header blocks

func NewHeaderAssembler(maxBuffers int, maxSize uint32, timeout time.Duration)
*HeaderAssembler {

    return &HeaderAssembler{

        assemblyBuffers: make(map[uint32]*AssemblyBuffer),

        maxBufferSize: maxSize,

        maxBuffers: maxBuffers,

        timeout: timeout,

    }

}
```

Core Logic Skeleton Code

```
// processNextFrame handles one complete frame processing cycle GO

func (rp *RequestProcessor) processNextFrame(ctx context.Context) error {

    // TODO 1: Read and parse next frame from connection using ParseFrame

    // TODO 2: Route frame to appropriate component using FrameRouter

    // TODO 3: Handle any immediate error responses or state updates

    // TODO 4: Coordinate with flow control if frame affects windows

    // TODO 5: Update connection statistics and health metrics

    // Hint: Use select with context.Done() to handle cancellation during blocking reads

    return nil

}

// RouteFrame dispatches a parsed frame to the appropriate component handler

func (fr *FrameRouter) RouteFrame(frame *Frame) error {

    // TODO 1: Validate frame header fields (reserved bits, length, etc.)

    // TODO 2: Check stream ID validity (0 for connection frames, >0 for stream frames)

    // TODO 3: Look up current stream state if stream ID > 0

    // TODO 4: Validate frame type compatibility with stream state

    // TODO 5: Dispatch to connection handler (stream ID 0) or stream manager (stream ID >
0)

    // TODO 6: Handle routing errors by generating RST_STREAM or GOAWAY frames

    // TODO 7: Update routing statistics for monitoring and debugging

    // Hint: Use a switch statement on frame.Type for efficient dispatching

    return nil

}

// AssembleHeaders collects HEADERS and CONTINUATION frames into complete header blocks

func (ha *HeaderAssembler) AssembleHeaders(frame *Frame) (*HeaderBlock, error) {
```

```
// TODO 1: Check if this is a new HEADERS frame or CONTINUATION of existing block

// TODO 2: Create new assembly buffer for HEADERS frame or lookup existing for
CONTINUATION

// TODO 3: Validate that CONTINUATION frames immediately follow HEADERS without
intervening frames

// TODO 4: Append frame payload to assembly buffer, checking size limits

// TODO 5: If END_HEADERS flag set, finalize assembly and create HeaderBlock

// TODO 6: Clean up assembly buffer and return complete header block

// TODO 7: Handle timeout cleanup of abandoned assembly buffers

// Hint: Use frame.Flags&FlagEndHeaders to detect completion

return nil, nil

}

// headerDecompressionWorker processes the sequential header decompression queue

func (rp *RequestProcessor) headerDecompressionWorker(ctx context.Context) {

    defer rp.wg.Done()

    for {

        select {

        case <-ctx.Done():

            return

        case headerBlock := <-rp.headerQueue:

            // TODO 1: Validate header block size and stream state before decompression

            // TODO 2: Call HPACK engine to decompress header block sequentially

            // TODO 3: Handle decompression errors (stream vs connection level)

            // TODO 4: Assemble HTTP request from decompressed headers and pseudo-headers

            // TODO 5: Route completed request to application handler

            // TODO 6: Update decompression statistics and table state metrics
    }
}
```

```
// Hint: Decompression must be sequential to maintain HPACK table consistency

    }

}

}

// errorCoordinator handles error propagation and recovery across components

func (rp *RequestProcessor) errorCoordinator(ctx context.Context) {

    defer rp.wg.Done()

    for {

        select {

        case <-ctx.Done():

            return

        case err := <-rp.errorChan:

            // TODO 1: Classify error as stream-level or connection-level

            // TODO 2: Generate appropriate error response (RST_STREAM or GOAWAY)

            // TODO 3: Coordinate cleanup of affected streams or entire connection

            // TODO 4: Update error statistics for monitoring

            // TODO 5: Trigger graceful shutdown if connection error detected

            // Hint: Use type assertion or error codes to classify error severity

        }

    }

}

// validateStreamTransition checks if frame is valid for current stream state

func (fr *FrameRouter) validateStreamTransition(streamID uint32, frameType uint8) error {

    // TODO 1: Look up current stream state from stream states map

    // TODO 2: Check frame type validity for current state using state machine rules
```

```

    // TODO 3: Validate special cases (like RST_STREAM being valid in most states)

    // TODO 4: Return appropriate error code for invalid transitions

    // Hint: Use a lookup table mapping (currentState, frameType) -> validity

    return nil
}

// updateStreamState synchronizes router's stream state cache with stream manager

func (fr *FrameRouter) updateStreamState(streamID uint32, newState StreamState) {

    // TODO 1: Acquire write lock for stream states map

    // TODO 2: Update state for specified stream ID

    // TODO 3: Clean up closed streams from cache to prevent memory leaks

    // TODO 4: Update state transition statistics

    // Hint: Consider using atomic operations for simple state updates

}

```

Language-Specific Hints

- Use `bufio.Reader` with generous buffer sizes (32KB+) for frame parsing to reduce system call overhead
- Implement frame routing with Go's switch statement, which compiles to efficient jump tables
- Use `sync.RWMutex` for stream state maps since lookups are much more frequent than updates
- Channel-based coordination works well for header decompression queues—use buffered channels to prevent blocking
- Consider `context.Context` for cancellation throughout the request processing pipeline
- Use `sync.WaitGroup` to coordinate graceful shutdown of processing goroutines
- Implement backpressure using channel capacity limits and select statements with default cases

Milestone Checkpoints

After implementing request processing flow:

- Run `go test ./internal/interactions/ -v` to verify component coordination
- Test with `curl --http2` requests, observing frame sequences in logs
- Verify that multiplexed requests (multiple concurrent curl commands) interleave properly
- Expected behavior: Multiple requests should complete without head-of-line blocking

After implementing frame routing:

- Use Wireshark to capture HTTP/2 traffic and verify frame routing decisions
- Test invalid frame sequences (wrong stream IDs, invalid state transitions)
- Expected behavior: Invalid frames should generate appropriate RST_STREAM or GOAWAY responses
- Signs of problems: Crashes on malformed frames, incorrect error response types

After implementing header decompression flow:

- Test large header sets that require CONTINUATION frames
- Verify HPACK table consistency by sending repeated headers that should compress well
- Expected behavior: Header compression ratios should improve over time as dynamic table fills
- Signs of problems: Header decompression errors, corrupted request data, table size issues

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Requests hang indefinitely	Header assembly waiting for CONTINUATION	Check frame logs for incomplete header blocks	Implement assembly timeout and cleanup
Random decompression errors	Race condition in HPACK table	Add logging around table modifications	Ensure sequential header processing
High memory usage	Assembly buffers not cleaned up	Monitor buffer count and ages	Implement periodic timeout cleanup
Connection drops unexpectedly	Protocol error in frame routing	Enable detailed frame validation logging	Fix validation logic for edge cases
Poor multiplexing performance	Blocking in header decompression	Profile goroutine blocking	Increase header queue buffer size

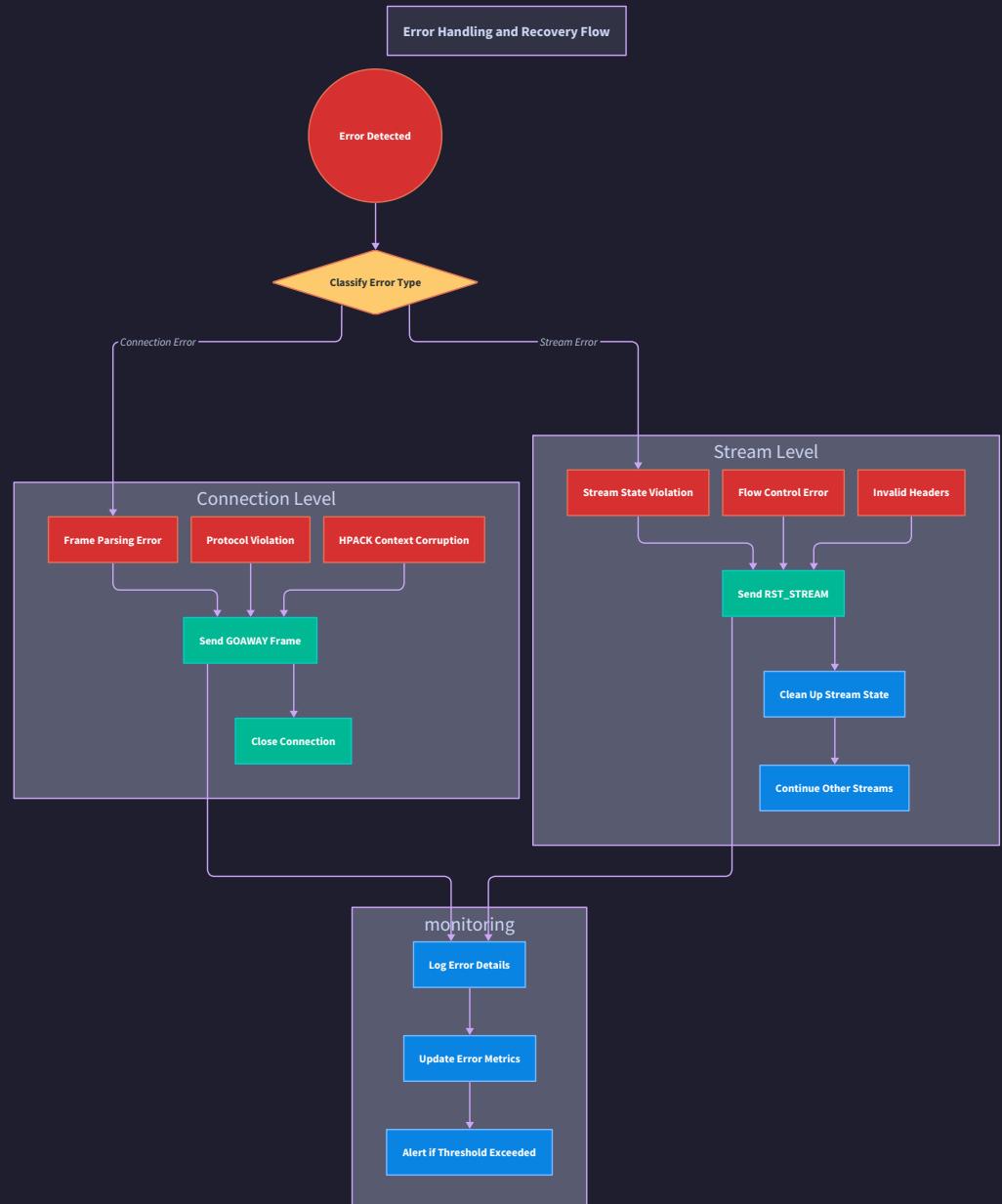
Error Handling and Edge Cases

Milestone(s): This section addresses error handling concerns across all milestones: frame validation errors (Milestone 1), HPACK decompression errors (Milestone 2), stream state machine violations (Milestone 3), and flow control window errors (Milestone 4). It establishes comprehensive error detection, classification, and recovery strategies essential for a production-grade HTTP/2 server.

Think of error handling in HTTP/2 as a sophisticated immune system for the protocol. Just like a biological immune system must distinguish between harmless variations and genuine threats, our HTTP/2 server must differentiate between recoverable stream-level errors and fatal connection-level errors. The protocol provides

specific mechanisms—like antibodies in the form of `RST_STREAM` frames for targeted stream termination and `GOAWAY` frames for graceful connection shutdown. Understanding when to apply localized treatment versus systemic intervention determines whether our server gracefully handles edge cases or crashes under pressure.

The complexity of HTTP/2 error handling stems from the protocol's multiplexed nature. Unlike HTTP/1.1 where each connection serves a single request-response pair, HTTP/2 connections simultaneously manage dozens of independent streams. A single malformed frame could indicate a bug in one client request (requiring only stream-level recovery) or a fundamental protocol violation that corrupts the entire connection state (requiring connection-level shutdown). The server must make these classification decisions in real-time while maintaining the integrity of unaffected streams.



Connection Errors:

- Malformed frames
- HPACK decompression failures
- Protocol violations

Stream Errors:

- Invalid stream states
- Flow control violations
- Malformed headers

Protocol Errors

Protocol errors represent violations of the HTTP/2 specification that require immediate detection and appropriate response. These errors range from malformed binary frames that cannot be parsed to logical violations like sending a `DATA` frame on a closed stream. The server's response must be proportional—

isolating the damage to the affected stream when possible, or protecting the entire connection when the violation threatens overall protocol integrity.

The foundation of protocol error handling lies in comprehensive **validation layers** that examine frames at multiple levels. Frame-level validation ensures binary format compliance, while semantic validation checks protocol state consistency. This multi-layered approach prevents malformed data from propagating through the system and corrupting internal state machines.

Error Category	Scope	Detection Point	Recovery Action
Frame Format Violations	Connection	Frame parsing	Send <code>GOAWAY</code> with <code>PROTOCOL_ERROR</code>
Invalid Frame Type	Stream	Frame routing	Send <code>RST_STREAM</code> with <code>PROTOCOL_ERROR</code>
State Machine Violations	Stream	Stream manager	Send <code>RST_STREAM</code> with appropriate code
Flow Control Violations	Stream/Connection	Window validation	Send <code>RST_STREAM</code> or <code>GOAWAY</code> with <code>FLOW_CONTROL_ERROR</code>
HPACK Decompression Failures	Connection	Header decompression	Send <code>GOAWAY</code> with <code>COMPRESSION_ERROR</code>
Resource Limit Exceeded	Connection	Resource tracking	Send <code>GOAWAY</code> with <code>ENHANCE_YOUR_CALM</code>

Decision: Error Scope Classification Strategy

- **Context:** HTTP/2 errors can affect individual streams or entire connections, requiring different recovery mechanisms
- **Options Considered:**
 - Always terminate connection on any error (simple but wasteful)
 - Always attempt stream-level recovery (complex, risk of connection corruption)
 - Classify errors by impact scope and respond appropriately (balanced approach)
- **Decision:** Implement error classification that maps violations to minimum necessary scope
- **Rationale:** Maximizes connection reuse while preventing error propagation; follows RFC 7540 guidance on error handling proportionality
- **Consequences:** Requires sophisticated error detection logic but improves server resilience and client experience

The `ProtocolError` type encapsulates all HTTP/2 specification violations with precise error codes that map to the standard error codes defined in RFC 7540. Each error includes contextual information that helps with debugging and appropriate response generation.

Field	Type	Description
Code	ErrorCode	RFC 7540 error code indicating violation type
Message	string	Human-readable description with context
StreamID	uint32	Affected stream ID, or 0 for connection errors
FrameType	uint8	Frame type that triggered the error
Severity	ErrorSeverity	Classification of error impact scope

Frame format validation represents the first line of defense against protocol violations. The parser must validate every aspect of the 9-byte frame header before attempting to process the payload. Length field validation prevents buffer overflows, while reserved bit checking ensures future protocol compatibility.

The frame validation algorithm follows these steps:

1. **Header Structure Validation:** Verify the 9-byte header contains valid length, type, and flags combinations according to the frame type specification
2. **Length Boundary Checking:** Ensure frame length does not exceed negotiated maximum frame size and fits within available buffer space
3. **Reserved Bit Validation:** Check that all reserved bits in flags and stream ID fields are set to zero as required by the specification
4. **Frame Type Recognition:** Verify the frame type is known and supported, rejecting unknown types with appropriate error codes
5. **Flag Consistency Checking:** Validate that flags are appropriate for the frame type and current stream state
6. **Stream ID Validation:** Ensure stream ID follows odd/even allocation rules and does not exceed the maximum stream ID space
7. **Payload Length Verification:** For frame types with fixed payload sizes, verify the length field matches expected payload size

The critical insight for frame validation is that **fail-fast detection** at the parsing boundary prevents corrupted data from entering the system's state machines. Once a frame passes validation, downstream components can assume structural integrity and focus on semantic processing.

HPACK decompression errors require special consideration because header compression failures affect the entire connection's header compression context. Unlike other stream-level errors, HPACK failures corrupt the

dynamic table state that all subsequent header blocks depend on. The server must immediately terminate the connection to prevent header decompression cascading failures.

HPACK Error Type	Cause	Detection Method	Recovery Action
Invalid Static Table Index	Index > 61	Bounds checking during lookup	Send <code>GOAWAY</code> with <code>COMPRESSION_ERROR</code>
Dynamic Table Overflow	Entry size exceeds table capacity	Size calculation during insertion	Send <code>GOAWAY</code> with <code>COMPRESSION_ERROR</code>
Huffman Decoding Failure	Invalid symbol sequence or padding	Symbol validation during decode	Send <code>GOAWAY</code> with <code>COMPRESSION_ERROR</code>
Integer Encoding Violation	Too many continuation bytes	Byte count during variable decode	Send <code>GOAWAY</code> with <code>COMPRESSION_ERROR</code>
String Length Mismatch	Declared length != actual bytes	Length tracking during string decode	Send <code>GOAWAY</code> with <code>COMPRESSION_ERROR</code>

Stream state machine violations occur when frames arrive that are invalid for the current stream state. For example, receiving a `HEADERS` frame on a stream already in the closed state violates the protocol specification. These errors typically indicate client bugs or race conditions in stream lifecycle management.

The validation process for stream state transitions involves:

1. **Current State Lookup:** Retrieve the stream's current state from the stream manager's state cache
2. **Frame Type Compatibility:** Check if the incoming frame type is valid for the current stream state according to RFC 7540 state transition rules
3. **Flag Validation:** Verify frame flags are appropriate for both the frame type and current stream state (e.g., `END_STREAM` flag validity)
4. **Transition Atomicity:** Ensure state transitions complete atomically to prevent race conditions between concurrent frame processing
5. **Error Code Selection:** Choose the most specific error code that describes the violation (e.g., `STREAM_CLOSED` vs. `PROTOCOL_ERROR`)

⚠ Pitfall: Race Condition in Stream State Validation A common bug occurs when stream state validation and state transitions are not properly synchronized. If one goroutine checks that a stream is open and begins processing a frame, while another goroutine concurrently closes that stream, the first goroutine may update an already-closed stream. This leads to inconsistent state and potential crashes. Always use proper locking or atomic operations when checking and updating stream state, and re-validate state after acquiring locks but before making changes.

Connection Errors

Connection errors represent failures that compromise the integrity of the entire HTTP/2 connection and require termination of all multiplexed streams. These errors typically stem from fundamental protocol violations, resource exhaustion, or implementation bugs that corrupt shared connection state. The server handles connection errors through the `GOAWAY` frame mechanism, which provides graceful connection termination while allowing in-flight requests to complete.

Think of connection error handling as performing emergency surgery on a complex system. When a critical component fails, the surgeon must quickly assess whether the problem can be isolated (stream-level error) or whether it threatens the entire patient's stability (connection-level error). The `GOAWAY` frame serves as the anesthesia—gracefully shutting down the system while ensuring no ongoing operations are left in an undefined state.

The `GOAWAY` frame provides a standardized mechanism for connection termination that includes diagnostic information to help clients understand the failure cause. Unlike abrupt connection closure, `GOAWAY` allows proper cleanup of resources and provides clear error signaling to the peer.

GOAWAY Field	Purpose	Example Values
Last Stream ID	Highest stream ID server processed	Even numbers for server errors
Error Code	Specific failure reason	<code>PROTOCOL_ERROR</code> , <code>INTERNAL_ERROR</code>
Debug Data	Additional diagnostic information	UTF-8 error description

Connection-level resource exhaustion represents one of the most common categories of connection errors. When clients exceed negotiated limits for concurrent streams, frame sizes, or header table sizes, the server must protect itself through connection termination. The `ENHANCE_YOUR_CALM` error code specifically addresses aggressive client behavior.

The resource exhaustion detection algorithm:

1. **Concurrent Stream Monitoring:** Track active stream count against the `MAX_CONCURRENT_STREAMS` setting, rejecting new streams when the limit is reached
2. **Frame Size Validation:** Verify every incoming frame does not exceed the negotiated `MAX_FRAME_SIZE` setting before attempting to read the payload
3. **Header List Size Checking:** Monitor accumulated header block size during CONTINUATION frame assembly to prevent memory exhaustion attacks
4. **Dynamic Table Size Enforcement:** Ensure HPACK dynamic table size updates do not exceed the `HEADER_TABLE_SIZE` setting
5. **Connection Window Monitoring:** Track connection-level flow control window to detect aggressive sending patterns that ignore flow control

6. **Rate Limiting Assessment:** Monitor frame arrival rates to detect denial-of-service attempts through frame flooding

Decision: Graceful vs Immediate Connection Termination

- **Context:** When connection errors occur, the server can either close the TCP connection immediately or send GOAWAY first
- **Options Considered:**
 - Immediate TCP close (fast but provides no diagnostic info)
 - GOAWAY then wait for client closure (proper but may delay recovery)
 - GOAWAY with timeout fallback (balanced approach)
- **Decision:** Send GOAWAY frame with 5-second timeout before forcing TCP connection closure
- **Rationale:** Provides diagnostic information to well-behaved clients while preventing resource leaks from misbehaving clients
- **Consequences:** Requires connection timeout management but improves debuggability and client experience

GOAWAY frame generation must include sufficient diagnostic information to help clients identify and fix the underlying problem. The debug data field should contain actionable information rather than generic error messages.

```
| Error Scenario | Error Code | Debug Data Content | |---|---|---|---| Unknown frame type received |
PROTOCOL_ERROR | "Unknown frame type 0x0F received" | | Exceeded concurrent streams |
ENHANCE_YOUR_CALM | "Exceeded MAX_CONCURRENT_STREAMS limit of 100" | | HPACK decompression
failure | COMPRESSION_ERROR | "Invalid Huffman sequence in header block" | | Frame size too large |
FRAME_SIZE_ERROR | "Frame size 32768 exceeds limit 16384" | | Implementation bug detected |
INTERNAL_ERROR | "Unexpected nil pointer in stream manager" |
```

Connection error recovery workflow ensures proper cleanup of all connection resources while maintaining system stability:

1. **Error Detection and Classification:** Identify the specific violation and map it to the appropriate HTTP/2 error code
2. **Active Stream Enumeration:** Collect all streams currently active on the connection to determine the last processed stream ID
3. **GOAWAY Frame Construction:** Build GOAWAY frame with appropriate error code, last stream ID, and diagnostic debug data
4. **Frame Transmission:** Send GOAWAY frame to the client with proper error signaling
5. **Resource Cleanup Scheduling:** Schedule cleanup of all connection-associated resources (streams, buffers, timers)

6. **Graceful Shutdown Timer:** Start timer to allow orderly connection closure before forcing TCP socket closure
7. **Connection State Transition:** Mark connection as closing to prevent new frame processing while cleanup proceeds
8. **Final Resource Release:** Release all memory, close file descriptors, and remove connection from active connection pool

⚠ Pitfall: Resource Leaks During Error Cleanup When connection errors occur, it's easy to forget to clean up all associated resources, leading to memory leaks and file descriptor exhaustion. Each connection may have dozens of active streams, each with buffers, timers, and state machines. Ensure error cleanup code traverses all active streams and releases their resources. Use defer statements or finally blocks to guarantee cleanup even if the error handling code itself panics.

Stream Errors

Stream errors represent failures that affect individual streams while leaving the overall connection intact. These errors typically result from application-level problems, client bugs, or resource constraints that can be isolated to specific request-response sequences. The server handles stream errors through `RST_STREAM` frames, which immediately terminate the affected stream without disrupting other multiplexed streams on the same connection.

The beauty of stream-level error handling lies in its **fault isolation**. When one client request contains malformed data or violates stream-specific constraints, the server can terminate just that stream while allowing dozens of other concurrent requests to proceed normally. This isolation property makes HTTP/2 servers far more resilient than HTTP/1.1 servers, where connection errors terminate all pending requests.

Stream error detection occurs at multiple points in the request processing pipeline. The frame router validates stream state consistency, the flow control system detects window violations, and application handlers identify semantic errors in request content.

| Error Detection Point | Error Types Detected | Recovery Mechanism | ---|---|---|---|
frame for stream state | `RST_STREAM` with state-specific error || Flow control validation | Window underflow/overflow | `RST_STREAM` with `FLOW_CONTROL_ERROR` || Header validation | Malformed HTTP semantics | `RST_STREAM` with `PROTOCOL_ERROR` || Application processing | Request timeout, resource limits | `RST_STREAM` with `CANCEL` or `REFUSED_STREAM` || Stream lifecycle | Premature stream closure | `RST_STREAM` with `STREAM_CLOSED` |

RST_STREAM frame generation requires careful selection of the error code to provide meaningful diagnostic information. The error code should precisely describe the failure reason to help clients debug their implementations.

Stream Error Code	Typical Causes	When to Use
PROTOCOL_ERROR	Invalid frame sequence, malformed headers	General protocol violations
CANCEL	Client or server cancellation	Voluntary termination
REFUSED_STREAM	Server overload, resource unavailable	Temporary rejection
STREAM_CLOSED	Frame sent on closed stream	State machine violations
FLOW_CONTROL_ERROR	Window violations	Credit system abuse

Stream state machine error handling ensures that stream lifecycle violations are detected and handled appropriately. The state machine must validate every frame against the current stream state and generate appropriate errors for invalid transitions.

The stream error validation algorithm:

1. **Stream Existence Verification:** Check if the stream ID exists in the stream manager's active stream table
2. **State Compatibility Check:** Verify the incoming frame type is valid for the stream's current state
3. **Flag Consistency Validation:** Ensure frame flags are appropriate for both frame type and stream state
4. **Flow Control Window Check:** Validate that DATA frames do not exceed available send window
5. **Header Block Completeness:** Verify HEADERS/CONTINUATION sequences are properly terminated with END_HEADERS flag
6. **Stream Priority Validation:** Check that priority dependencies do not create cycles in the dependency tree
7. **Error Code Selection:** Choose the most specific error code that describes the detected violation

The key principle in stream error handling is **minimizing blast radius**. Each error should affect the smallest possible scope while maintaining protocol integrity. Stream errors should never cascade to affect other streams unless the error indicates fundamental connection corruption.

Stream resource management errors occur when individual streams exceed their allocated resources or violate per-stream constraints. These errors help prevent individual streams from consuming excessive server resources.

Resource Type	Limit	Error Detection	Recovery Action
Header block size	Configurable limit (8KB default)	During CONTINUATION assembly	RST_STREAM with REFUSED_STREAM
Request body size	Application-defined limit	During DATA frame accumulation	RST_STREAM with CANCEL
Stream lifetime	Timeout configuration	Timer expiration	RST_STREAM with CANCEL
Priority updates	Rate limiting	Update frequency tracking	RST_STREAM with ENHANCE_YOUR_CALM

Stream cleanup workflow ensures that terminated streams release all associated resources and update connection state appropriately:

- Stream State Transition:** Move stream to closed state and prevent further frame processing
- Buffer Release:** Free any accumulated header blocks, DATA frame buffers, or assembly state
- Timer Cancellation:** Cancel any active timers associated with the stream (timeouts, keep-alive)
- Flow Control Update:** Release any reserved flow control window space back to connection pool
- Priority Tree Update:** Remove stream from priority dependency tree and update dependent streams
- Statistics Update:** Record stream completion statistics and error metrics for monitoring
- Memory Reclamation:** Remove stream entry from active stream maps and release memory
- Connection Counter Update:** Decrement active stream count to allow new stream creation

⚠ Pitfall: Stream Error Escalation A subtle bug occurs when stream-level errors are incorrectly escalated to connection errors. For example, if stream state validation code panics due to a nil pointer, the panic might be caught at the connection level and treated as a connection error, causing unnecessary connection termination. Ensure stream-level error handling code is robust and that panics are caught and converted to appropriate stream errors rather than propagating up to connection handlers.

Error isolation verification ensures that stream errors do not affect other streams on the same connection. The server must validate that stream error handling preserves connection state integrity.

The isolation verification checklist:

- Connection State Preservation:** Verify that stream errors do not modify connection-level settings or shared state
- HPACK Table Integrity:** Ensure stream errors do not corrupt the shared dynamic table used for header compression
- Flow Control Isolation:** Confirm that stream window errors do not affect connection-level flow control windows

4. **Frame Processing Continuity:** Validate that other streams continue processing frames normally after a stream error
5. **Resource Pool Integrity:** Check that stream cleanup properly returns resources to shared pools
6. **Statistics Isolation:** Ensure stream error metrics do not interfere with connection-level health monitoring

Implementation Guidance

The error handling implementation requires sophisticated coordination between all HTTP/2 server components. The error system must detect violations at the appropriate scope, generate standardized error responses, and ensure proper resource cleanup without affecting unrelated streams.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Error Types	Built-in error types with wrapping	Custom error hierarchy with error codes
Error Logging	Standard log package	Structured logging with context (logrus/zap)
Error Metrics	Simple counters	Prometheus metrics with labels
Error Recovery	Basic cleanup routines	Graceful shutdown with timeout handling

B. Recommended File Structure

```

internal/http2server/
  errors/
    protocol_errors.go      ← HTTP/2 error codes and types
    error_handlers.go       ← Error detection and response generation
    cleanup.go              ← Resource cleanup routines
    errors_test.go          ← Error handling unit tests
  connection/
    goaway.go               ← GOAWAY frame handling
    connection_errors.go    ← Connection-level error detection
  stream/
    rst_stream.go           ← RST_STREAM frame handling
    stream_errors.go        ← Stream-level error detection

```

C. Infrastructure Starter Code

```
package errors

import (
    "fmt"
    "sync"
    "time"
)

// ErrorCode represents HTTP/2 error codes from RFC 7540

type ErrorCode uint32

const (
    ErrorCodeNoError           ErrorCode = 0x0
    ErrorCodeProtocolError     ErrorCode = 0x1
    ErrorCodeInternalError     ErrorCode = 0x2
    ErrorCodeFlowControlError  ErrorCode = 0x3
    ErrorCodeSettingsTimeout   ErrorCode = 0x4
    ErrorCodeStreamClosed      ErrorCode = 0x5
    ErrorCodeFrameSizeError    ErrorCode = 0x6
    ErrorCodeRefusedStream     ErrorCode = 0x7
    ErrorCodeCancel             ErrorCode = 0x8
    ErrorCodeCompressionError   ErrorCode = 0x9
    ErrorCodeConnectError       ErrorCode = 0xa
    ErrorCodeEnhanceYourCalm   ErrorCode = 0xb
    ErrorCodeInadequateSecurity ErrorCode = 0xc
    ErrorCodeHTTP11Required     ErrorCode = 0xd
)

// ProtocolError represents an HTTP/2 protocol violation
```

GO

```
type ProtocolError struct {

    Code      ErrorCode
    Message   string
    StreamID  uint32
    FrameType uint8
    Fatal     bool // true for connection errors, false for stream errors
}

func (e *ProtocolError) Error() string {
    scope := "stream"

    if e.Fatal {
        scope = "connection"
    }

    return fmt.Sprintf("HTTP/2 %s error (code=%d, stream=%d): %s",
        scope, e.Code, e.StreamID, e.Message)
}

// NewProtocolError creates a protocol error with code and message

func NewProtocolError(code ErrorCode, message string) *ProtocolError {
    return &ProtocolError{
        Code:      code,
        Message:   message,
        Fatal:     isConnectionError(code),
    }
}

// ErrorTracker maintains error statistics for monitoring

type ErrorTracker struct {
```

```
    mu          sync.RWMutex

    connectionErrors map[ErrorCode]uint64

    streamErrors     map[ErrorCode]uint64

    totalConnections uint64

    totalStreams     uint64

}

func NewErrorTracker() *ErrorTracker {

    return &ErrorTracker{

        connectionErrors: make(map[ErrorCode]uint64),

        streamErrors:     make(map[ErrorCode]uint64),

    }

}

func (et *ErrorTracker) RecordError(err *ProtocolError) {

    et.mu.Lock()

    defer et.mu.Unlock()

    if err.Fatal {

        et.connectionErrors[err.Code]++

        et.totalConnections++

    } else {

        et.streamErrors[err.Code]++

        et.totalStreams++

    }

}
```

D. Core Logic Skeleton Code

GO

```
// ErrorHandler coordinates error detection and response generation

type ErrorHandler struct {
    conn          *Connection
    frameRouter   *FrameRouter
    streamManager *StreamManager
    tracker       *ErrorTracker
}

// HandleError processes detected errors and generates appropriate responses

func (eh *ErrorHandler) HandleError(err error) error {
    // TODO 1: Cast error to ProtocolError or wrap generic errors

    // TODO 2: Determine error scope (connection vs stream level)

    // TODO 3: Generate appropriate response frame (GOAWAY vs RST_STREAM)

    // TODO 4: Update error statistics via tracker

    // TODO 5: Initiate cleanup procedures for affected resources

    // TODO 6: Return error to caller for additional handling if needed

    // Hint: Use type assertions to identify ProtocolError vs other error types
}

// ValidateFrame performs comprehensive frame validation

func (eh *ErrorHandler) ValidateFrame(frame *Frame) error {
    // TODO 1: Validate frame header structure (length, type, flags)

    // TODO 2: Check frame length against negotiated maximum frame size

    // TODO 3: Verify reserved bits are zero in flags and stream ID

    // TODO 4: Validate frame type is known and supported

    // TODO 5: Check flag combinations are valid for frame type

    // TODO 6: Validate stream ID follows odd/even allocation rules

    // TODO 7: For fixed-size frames, verify payload length matches expected size
}
```

```
// Hint: Create lookup tables for valid flag combinations per frame type
}

// ValidateStreamTransition checks if frame is valid for current stream state

func (eh *ErrorHandler) ValidateStreamTransition(streamID uint32, frameType uint8, flags
uint8) error {

    // TODO 1: Retrieve current stream state from stream manager

    // TODO 2: Check if frame type is valid for current state

    // TODO 3: Validate flags are appropriate for frame type and state

    // TODO 4: Check for common state machine violations (closed stream, etc.)

    // TODO 5: Return specific error code describing the violation

    // Hint: Use state transition tables from RFC 7540 Section 5.1

}

// SendGOAWAY generates and sends connection termination frame

func (eh *ErrorHandler) SendGOAWAY(errorCode ErrorCode, lastStreamID uint32, debugData
string) error {

    // TODO 1: Create GOAWAY frame with proper fields

    // TODO 2: Set last stream ID to highest processed stream

    // TODO 3: Include error code and debug data

    // TODO 4: Send frame through connection writer

    // TODO 5: Mark connection as closing to prevent new frame processing

    // TODO 6: Schedule connection cleanup after timeout period

    // Hint: GOAWAY frame format is defined in RFC 7540 Section 6.8

}

// SendRSTStream terminates individual stream with error code

func (eh *ErrorHandler) SendRSTStream(streamID uint32, errorCode ErrorCode) error {

    // TODO 1: Create RST_STREAM frame with stream ID and error code
```

```

    // TODO 2: Validate stream ID exists and is not already closed

    // TODO 3: Send frame through connection writer

    // TODO 4: Update stream state to closed in stream manager

    // TODO 5: Release stream resources (buffers, timers, etc.)

    // TODO 6: Update connection statistics for stream termination

    // Hint: RST_STREAM frame format is defined in RFC 7540 Section 6.4

}

```

E. Language-Specific Hints

- Use Go's error wrapping with `fmt.Errorf("context: %w", err)` to preserve error chains
- Implement custom error types that satisfy the `error` interface for HTTP/2-specific errors
- Use `sync.RWMutex` for error statistics tracking to allow concurrent reads
- Leverage Go's type assertions `err.(*ProtocolError)` to identify HTTP/2 errors
- Use `context.Context` for cleanup timeout handling in GOAWAY processing
- Implement error recovery with `defer` statements to ensure cleanup on panics

F. Milestone Checkpoints

After implementing error handling:

Milestone 1 Checkpoint (Frame Validation):

- Send malformed frame with invalid length: `echo "invalid" | nc localhost 8080`
- Expected: Server sends GOAWAY with PROTOCOL_ERROR and closes connection
- Verify: Check server logs for frame validation error messages

Milestone 2 Checkpoint (HPACK Error Handling):

- Send HEADERS frame with invalid Huffman sequence
- Expected: Server sends GOAWAY with COMPRESSION_ERROR
- Verify: Connection terminates, other connections remain active

Milestone 3 Checkpoint (Stream Error Isolation):

- Open multiple streams, send invalid frame on one stream
- Expected: Server sends RST_STREAM for affected stream only
- Verify: Other streams continue processing normally

Milestone 4 Checkpoint (Flow Control Errors):

- Send DATA frame exceeding stream window

- Expected: Server sends RST_STREAM with FLOW_CONTROL_ERROR
- Verify: Connection-level flow control remains intact

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Connection always terminates on errors	Stream errors escalated to connection level	Check error classification logic	Implement proper error scope detection
Resource leaks after errors	Cleanup code not executing	Add logging to cleanup routines	Use defer statements for guaranteed cleanup
Cascading stream failures	Shared state corruption	Validate connection state preservation	Isolate stream-specific state modifications
Client receives generic errors	Poor error code selection	Log detailed error context	Map specific violations to appropriate error codes

Testing Strategy

Milestone(s): This section provides comprehensive testing approaches for all milestones: Binary Framing Engine testing (Milestone 1), HPACK Compression Engine validation (Milestone 2), Stream Management verification (Milestone 3), and Flow Control testing (Milestone 4). Each milestone requires both isolated unit testing and integrated system validation to ensure protocol compliance and robustness.

Think of testing an HTTP/2 server like quality assurance for a complex air traffic control system. Just as air traffic control must safely coordinate hundreds of simultaneous aircraft arrivals and departures, your HTTP/2 server must manage dozens of concurrent streams over a single connection while maintaining perfect protocol compliance. The testing strategy mirrors this complexity: you need to test individual components in isolation (like testing a single radar unit), validate component interactions (like testing handoff procedures between controllers), and verify the entire system under realistic load (like testing the airport during peak traffic). Each layer of testing catches different classes of bugs, from basic parsing errors to subtle race conditions that only emerge under concurrent load.

The HTTP/2 protocol's binary nature and complex state machines create unique testing challenges compared to text-based HTTP/1.1. Binary framing requires bit-level accuracy, HPACK compression involves stateful table management, stream multiplexing creates intricate concurrency scenarios, and flow control introduces credit-based backpressure handling. A comprehensive testing strategy must address each of these complexities systematically, building confidence layer by layer from individual frame parsing up to full client integration.

Unit Testing

Unit testing forms the foundation of HTTP/2 server validation by isolating each component and verifying its behavior under controlled conditions. Unlike integration testing, unit tests provide immediate feedback during development, precise failure localization, and comprehensive edge case coverage. Each major component—Binary Framing Engine, HPACK Compression Engine, Stream Management, and Flow Control—requires specialized testing approaches that address their unique characteristics and failure modes.

Binary Framing Engine Unit Tests

The Binary Framing Engine requires exhaustive testing of frame parsing, serialization, and validation logic. Think of frame parsing like a document scanner that must correctly interpret various document formats—it needs to handle perfect documents flawlessly, detect corruption in damaged documents, and gracefully reject invalid formats. The binary nature of HTTP/2 frames means that single-bit errors can cause catastrophic parsing failures, making comprehensive test coverage essential.

Frame parsing tests must validate every aspect of the 9-byte frame header structure. The `ParseFrame` function processes the header fields in network byte order, extracting the 24-bit length, 8-bit type, 8-bit flags, and 32-bit stream ID. Test cases should cover the full range of valid values, boundary conditions, and invalid combinations.

Test Category	Test Cases	Expected Behavior	Validation Points
Header Parsing	Valid frames with all frame types	Successful parsing with correct field extraction	Length, type, flags, stream ID accuracy
Boundary Conditions	Maximum frame size (16777215 bytes)	Successful parsing without overflow	Payload length validation
Invalid Lengths	Frame length exceeding maximum	Parse error with <code>ErrorCodeFrameSizeError</code>	Early validation before payload read
Reserved Bits	Stream ID with reserved bit set	Parse error with <code>ErrorCodeProtocolError</code>	Bit masking validation
Endianness	Multi-byte fields in network order	Correct value interpretation	Big-endian conversion verification

Frame type-specific parsing requires dedicated test suites for each frame type. `FrameTypeDATA` frames need testing with various payload sizes and flag combinations, particularly the `FlagEndStream` flag. `FrameTypeHEADERS` frames require validation of header block fragments and the `FlagEndHeaders` flag. `FrameTypeSETTINGS` frames must handle parameter parsing and the ACK flag correctly.

```

Test: DATA Frame Parsing
Input: 9-byte header (length=1024, type=0x0, flags=0x1, stream=5) + 1024-byte payload
Expected: Frame{Length: 1024, Type: FrameTypeDATA, Flags: FlagEndStream, StreamID: 5,
Payload: [1024]byte}
Validation: len(frame.Payload) == frame.Length, frame.Flags&FlagEndStream != 0

```

Frame serialization testing ensures that `SerializeFrame` produces valid wire format output that can be parsed by other HTTP/2 implementations. The serialization process must convert internal frame structures back to network byte order while preserving all header fields and payload data. Round-trip tests verify that parsing followed by serialization produces identical output.

Serialization Test	Input Frame	Expected Wire Format	Validation Method
DATA Frame	Frame with 512-byte payload	9-byte header + 512 payload bytes	Byte-by-byte comparison
Empty SETTINGS	Settings frame with no parameters	9-byte header with zero length	Length field verification
HEADERS Frame	Headers with END_HEADERS flag	Correct flag encoding in byte 4	Bit pattern validation
Large Frame	Frame at maximum size limit	Complete serialization without truncation	Total size verification

Frame validation testing ensures the `ValidateFrame` function correctly identifies protocol violations and implementation limits. Validation occurs after parsing but before frame processing, catching malformed frames that could compromise server stability. The validation logic must check frame type constraints, flag combinations, and stream ID restrictions.

Validation Error Test Cases:

- Frame Size Violations:** Frames exceeding `MaxFrameSize` should trigger `ErrorCodeFrameSizeError`
- Invalid Frame Types:** Unknown frame types should be ignored per RFC 7540
- Stream ID Constraints:** Control frames (SETTINGS, PING, GOAWAY) on non-zero streams should fail
- Flag Misuse:** Invalid flag combinations like `FlagEndHeaders` on DATA frames should be rejected
- Payload Size Mismatches:** SETTINGS frames with incorrect parameter counts should fail validation

HPACK Compression Engine Unit Tests

HPACK compression testing requires validation of the complex stateful compression algorithm defined in RFC 7541. Think of HPACK like a shared dictionary that both client and server update simultaneously—any inconsistency in table management breaks the entire compression scheme. The testing strategy must verify

static table lookups, dynamic table management, Huffman decoding, and integer encoding under all possible conditions.

Static table testing validates correct implementation of the predefined 61-entry table specified in RFC 7541 Appendix B. Every HPACK implementation must provide identical static table behavior, making this testing straightforward but critical. The `Lookup` function must return correct header entries for indices 1-61 and handle invalid indices gracefully.

Static Table Test	Index	Expected Name	Expected Value	Error Condition
Authority Header	1	<code>":authority"</code>	""	Index 0 should fail
Method GET	2	<code>":method"</code>	"GET"	Standard method lookup
Status 200	8	<code>":status"</code>	"200"	Common status code
Last Entry	61	<code>"www-authenticate"</code>	""	Boundary condition
Invalid Index	62	N/A	N/A	Should return error

Dynamic table testing requires comprehensive validation of the FIFO eviction system and size management. The `DynamicTable` maintains connection-specific header entries with automatic eviction when size limits are exceeded. Each entry's size calculation follows the formula: `name_length + value_length + 32` bytes of overhead.

The `Add` method must correctly insert new entries while maintaining the size constraint through eviction. When adding an entry would exceed `maxSize`, the implementation must remove the oldest entries until sufficient space is available. Edge cases include adding an entry larger than the maximum table size (which should empty the table) and adding to a zero-size table (which should be ignored).

```
Test Scenario: Dynamic Table Eviction
Initial State: Table with 3 entries totaling 4000 bytes, maxSize=4096
Action: Add entry requiring 200 bytes
Expected: Oldest entry evicted, new entry added, total size ≤ 4096
Validation: Entry count, size calculation, FIFO order preservation
```

Integer encoding testing validates the variable-length encoding scheme used throughout HPACK for table indices and string lengths. The `DecodeInteger` function must handle 5-bit, 6-bit, and 7-bit prefix encodings with proper continuation byte processing. Test cases must cover small values that fit in the prefix, large values requiring continuation bytes, and invalid encodings that exceed implementation limits.

Integer Test	Prefix Bits	Encoded Bytes	Expected Value	Edge Case
Small Value	5	[0x0A]	10	Fits in prefix
Large Value	5	[0x1F, 0xE8, 0x07]	1337	Multi-byte encoding
Maximum Prefix	6	[0x3F, 0x00]	63	Boundary condition
Overflow	7	[0x7F, 0xFF, 0xFF, 0xFF, 0x7F]	Error	Implementation limit

Huffman decoding testing ensures correct implementation of the static Huffman table defined in RFC 7541 Appendix B. The `DecodeHuffman` function processes variable-length bit sequences, requiring careful boundary handling and padding validation. The decoder must detect the `EOS_SYMBOL` and reject invalid padding patterns.

Critical Huffman Test Cases:

1. **Common Characters:** ASCII letters and digits with short codes
2. **Rare Characters:** Extended ASCII with longer codes
3. **Boundary Conditions:** Strings ending at byte boundaries vs. requiring padding
4. **Invalid Padding:** Padding bits that don't match EOS prefix
5. **Truncated Input:** Incomplete bit sequences that can't form valid symbols

Stream Management Unit Tests

Stream management testing validates the complex state machine and concurrent stream handling that forms the core of HTTP/2 multiplexing. Think of stream management like managing multiple phone conversations on a single line—each conversation has its own state (dialing, talking, hanging up) but they all share the same physical connection. The testing strategy must verify state transitions, stream ID allocation, priority scheduling, and concurrent access patterns.

Stream state machine testing requires comprehensive validation of the state transitions defined in RFC 7540 Section 5.1. Each stream progresses through states from `StateIdle` to `StateClosed`, with frame types and flags triggering specific transitions. Invalid transitions must be detected and result in stream errors or connection errors as appropriate.

Current State	Frame Type	Flags	Expected Next State	Error Condition
StateIdle	HEADERS	0x0	StateOpen	Normal request start
StateOpen	DATA	FlagEndStream	StateHalfClosedRemote	Client completes request
StateHalfClosedRemote	HEADERS	0x0	StateHalfClosedRemote	Response headers
StateHalfClosedRemote	DATA	FlagEndStream	StateClosed	Response complete
StateClosed	DATA	0x0	Error	Invalid frame on closed stream

The `TransitionState` function must validate each transition against the current state and frame type. Invalid transitions should generate `RST_STREAM` frames with appropriate error codes. The implementation must handle concurrent state changes from multiple goroutines processing frames simultaneously.

Stream ID allocation testing validates the odd/even assignment scheme and monotonic ordering requirements. Client-initiated streams use odd IDs (1, 3, 5...) while server-initiated streams use even IDs (2, 4, 6...). The `AllocateStreamID` function must track the highest allocated ID and prevent reuse of stream IDs within a connection.

Test: Concurrent Stream ID Allocation

Scenario: 10 goroutines simultaneously allocate client stream IDs

Expected: Unique odd IDs in monotonic order (1, 3, 5, 7, 9, 11, 13, 15, 17, 19)

Validation: No duplicates, all odd, strictly increasing sequence

Priority scheduling testing validates the weight-based bandwidth allocation specified in RFC 7540 Section 5.3. The `ScheduleStreams` function must distribute available transmission windows among active streams based on their configured weights. The deficit round-robin algorithm ensures fair scheduling while respecting priority relationships.

Priority testing requires careful setup of stream hierarchies and measurement of bandwidth allocation over multiple scheduling rounds. Test scenarios should include equal-weight streams (uniform distribution), varied weights (proportional distribution), and priority inversions (higher-weight streams blocked by flow control).

Flow Control Unit Tests

Flow control testing validates the credit-based window management that prevents buffer overflow in HTTP/2 implementations. Think of flow control like water pressure regulation—each stream and the overall connection have pressure gauges (windows) that prevent pipes from bursting due to excessive flow. The testing strategy must verify window calculations, `WINDOW_UPDATE` processing, and backpressure queue management under various conditions.

Window management testing focuses on the `WindowManager` component that tracks both connection-level and stream-level send windows. Each DATA frame transmission reduces both windows, while `WINDOW_UPDATE` frames increment the appropriate window. The implementation must prevent window underflow and handle window overflow conditions gracefully.

Window Test Scenario	Initial Connection Window	Initial Stream Window	Data Size	Expected Result
Normal Transmission	65535	65535	1024	Windows reduced by 1024
Stream Window Exhaustion	65535	500	1024	Transmission blocked
Connection Window Exhaustion	500	65535	1024	Transmission blocked
Both Windows Exhausted	500	500	1024	Transmission blocked

The `CanSendData` function must check both connection and stream windows before allowing DATA frame transmission. The `ConsumeWindows` function atomically decrements both windows to maintain consistency. Window update processing must handle concurrent updates and detect overflow conditions.

```

Test: Concurrent Window Updates
Setup: Stream window = 1000, multiple WINDOW_UPDATE frames with increment = 5000
Action: Process 10 concurrent WINDOW_UPDATE frames
Expected: Window increases by 50000, no overflow, atomic updates
Validation: Final window value, no race conditions, consistent state

```

Backpressure queue testing validates the queuing mechanism used when transmission windows are exhausted. The `BackpressureQueue` buffers DATA frames until windows become available, maintaining transmission order and priority relationships. Queue management must prevent unbounded growth while preserving frame ordering.

Backpressure Test Scenarios:

1. **Single Stream Blocking:** One stream exhausts its window while others continue transmission
2. **Connection Blocking:** All streams blocked by connection window exhaustion
3. **Priority Ordering:** Higher-priority streams drain first when windows become available
4. **Queue Overflow:** Queue size limits prevent memory exhaustion
5. **Frame Timeout:** Old queued frames are discarded to prevent unbounded buffering

Integration Testing

Integration testing validates the complete HTTP/2 server implementation by testing component interactions and end-to-end protocol compliance. While unit tests verify isolated component behavior, integration tests catch bugs that emerge from component interactions, concurrent execution, and real network conditions. The integration testing strategy employs multiple client implementations and testing tools to ensure broad compatibility and protocol adherence.

Integration testing serves as the bridge between unit-level validation and production deployment. Think of it like testing a complete symphony orchestra after rehearsing individual sections—while each musician might play perfectly in isolation, the full performance requires coordination, timing, and seamless interaction between all parts. HTTP/2's complexity demands similar holistic validation where frame parsing, header compression, stream management, and flow control must work together flawlessly.

End-to-End Client Testing

Real HTTP/2 client testing provides the most authentic validation of server implementation correctness. Different HTTP/2 clients exercise different protocol features and behaviors, exposing bugs that synthetic tests might miss. The testing strategy incorporates command-line tools, browser clients, and specialized HTTP/2 testing libraries to maximize coverage.

curl HTTP/2 Testing provides command-line validation of basic HTTP/2 functionality. Modern curl versions support HTTP/2 over TLS with the `--http2` flag, making it ideal for automated testing and debugging. curl testing focuses on request-response correctness, header handling, and error condition responses.

```
Basic curl Test Commands:  
curl --http2 -v https://localhost:8443/simple  
curl --http2 -H "Custom-Header: test-value" https://localhost:8443/echo-headers  
curl --http2 -d "request body" https://localhost:8443/post-handler  
curl --http2 --limit-rate 1k https://localhost:8443/large-response
```

The verbose output (`-v` flag) provides detailed protocol information including frame exchanges, header compression, and TLS negotiation. Successful curl tests should show `* Using HTTP2, server supports multi-use` and frame-level debugging information when using `--http2-prior-knowledge` for cleartext HTTP/2.

curl Test Category	Command	Expected Behavior	Validation Points
Basic GET	<code>curl --http2 -v https://localhost:8443/</code>	200 response with correct content	Response headers, body content
Header Echo	<code>curl --http2 -H "X-Test: value" https://localhost:8443/echo</code>	Headers reflected in response	Header preservation through HPACK
Large Response	<code>curl --http2 https://localhost:8443/10mb</code>	Complete download without corruption	Flow control, frame fragmentation
Concurrent Requests	Multiple curl processes simultaneously	All requests succeed	Stream multiplexing

Browser Testing validates real-world client behavior with modern browsers like Chrome, Firefox, and Safari. Browsers implement sophisticated HTTP/2 optimizations including connection coalescing, priority scheduling, and aggressive multiplexing. Browser testing requires HTTPS with valid certificates, making TLS configuration part of the integration test setup.

Browser testing methodology involves both manual testing and automated browser automation tools. Manual testing provides immediate visual feedback and debugging information through browser developer tools. Automated testing using tools like Selenium or Playwright enables reproducible test execution and continuous integration.

Critical Browser Test Scenarios:

- Multiple Resource Loading:** HTML pages with many CSS, JavaScript, and image resources
- Long-Lived Connections:** WebSocket upgrades and server-sent events over HTTP/2
- Priority Handling:** Critical resources (CSS) vs. non-critical resources (images)
- Error Recovery:** Network interruptions and connection recovery
- Cache Behavior:** HTTP caching with HTTP/2 multiplexing

Browser developer tools provide invaluable insight into HTTP/2 behavior. Chrome's Network tab shows protocol versions, connection reuse, and stream prioritization. Firefox's developer tools include HTTP/2 frame inspection and timing analysis. Safari's Web Inspector provides similar capabilities with additional WebKit-specific metrics.

Specialized HTTP/2 Testing Tools

Dedicated HTTP/2 testing tools provide comprehensive protocol validation beyond what general-purpose clients offer. These tools implement HTTP/2-specific test suites, protocol fuzzing, and compliance checking that validate server behavior against RFC specifications.

h2spec Compliance Testing is the de facto standard for HTTP/2 protocol compliance testing. The h2spec tool implements hundreds of test cases covering every aspect of RFC 7540 and RFC 7541. It tests positive cases (valid protocol usage) and negative cases (error handling and protocol violations).

h2spec testing categories include frame parsing, stream state machines, flow control, header compression, and error handling. Each test case sends specific frame sequences and validates the server's response against RFC requirements. Failed tests indicate protocol violations that could cause interoperability issues with other HTTP/2 implementations.

h2spec Test Execution:

```
h2spec --host localhost --port 8443 --tls --strict  
h2spec --host localhost --port 8080 --timeout 5s (cleartext)  
h2spec --host localhost --port 8443 --tls --sections 3.5,4,5 (specific sections)
```

h2spec test results identify the specific RFC section and requirement that failed, enabling precise bug identification. The `--strict` flag enables additional compliance checks beyond the minimum RFC requirements. Test failures should be investigated immediately as they indicate fundamental protocol implementation bugs.

h2spec Test Section	Focus Area	Critical Tests	Common Failures
Section 3.5	HTTP/2 Connection Preface	Connection initialization	Missing or invalid preface
Section 4	Frame Definitions	Frame parsing and validation	Frame size limits, reserved bits
Section 5	Stream States	State machine compliance	Invalid state transitions
Section 6	Error Handling	Error codes and recovery	Missing GOAWAY frames

HTTP/2 Load Testing validates server behavior under realistic concurrent load using tools like h2load, wrk2 with HTTP/2 support, or custom load generators. Load testing exposes race conditions, memory leaks, and performance bottlenecks that only manifest under concurrent stress.

h2load provides HTTP/2-specific load testing with configurable concurrency, request rates, and connection counts. Unlike HTTP/1.1 load testing, HTTP/2 load testing can achieve high request rates with minimal connection counts due to multiplexing.

h2load Load Test Examples:

```
h2load -n10000 -c10 -m10 https://localhost:8443/small (10K requests, 10 connections, 10 streams)  
h2load -n1000 -c1 -m100 --h1 https://localhost:8443/compare (HTTP/1.1 comparison)  
h2load -t10 -c10 -m50 -d60 https://localhost:8443/sustained (60-second sustained load)
```

Load testing should monitor server resource usage including memory consumption, goroutine counts, file descriptor usage, and CPU utilization. Memory leaks often manifest as gradually increasing memory usage under sustained load. Goroutine leaks appear as unbounded goroutine growth over time.

Protocol Fuzzing and Edge Case Testing

Protocol fuzzing systematically generates malformed or edge-case HTTP/2 traffic to test server robustness. Think of fuzzing like stress-testing a bridge with unusual load patterns—while normal traffic might work perfectly, edge cases and malformed inputs can expose structural weaknesses. HTTP/2 fuzzing targets binary frame formats, state machine edge cases, and resource exhaustion scenarios.

Binary Frame Fuzzing generates malformed frames with invalid lengths, corrupted headers, and boundary condition violations. Frame fuzzing should trigger appropriate error responses without causing server crashes or resource leaks. Proper frame validation and error handling prevent fuzzing inputs from compromising server stability.

Frame fuzzing categories include length field manipulation (oversized/undersized frames), type field corruption (invalid frame types), flag field abuse (invalid flag combinations), and stream ID violations (reserved bits, invalid stream assignments). Each category tests different validation code paths and error handling logic.

State Machine Fuzzing sends frame sequences that violate HTTP/2 state machine requirements. Examples include sending DATA frames on closed streams, HEADERS frames with invalid dependencies, and CONTINUATION frames without proper HEADERS initiation. State machine fuzzing validates the robustness of stream state tracking and error recovery.

Resource Exhaustion Testing attempts to exhaust server resources through protocol abuse. Test scenarios include creating excessive concurrent streams, sending oversized header blocks, exhausting flow control windows, and filling dynamic tables with large entries. Resource exhaustion testing validates implementation limits and graceful degradation under attack conditions.

Critical Insight: Protocol fuzzing often reveals bugs that traditional testing misses. Even when unit tests pass and integration tests succeed, fuzzing can expose edge cases where the combination of valid protocol elements creates invalid states. This is particularly important for HTTP/2 due to its state machine complexity and binary encoding.

Milestone Checkpoints

Milestone checkpoints provide concrete verification steps after completing each implementation phase. These checkpoints serve as quality gates, ensuring each milestone delivers working functionality before proceeding to more complex features. Think of milestone checkpoints like flight test phases for aircraft development—each phase validates specific capabilities before advancing to more complex maneuvers.

Each checkpoint includes both automated testing commands and manual verification steps. Automated tests provide quick pass/fail feedback, while manual verification ensures the implementation behaves correctly

under realistic conditions. The checkpoints progress from basic functionality to complete HTTP/2 protocol compliance.

Milestone 1 Checkpoint: Binary Framing

After implementing the Binary Framing Engine, the server should successfully parse, validate, and serialize all required HTTP/2 frame types. The checkpoint validates frame header parsing, payload extraction, and wire format serialization without requiring higher-level protocol features.

Automated Test Verification:

```
Test Command: go test -v ./internal/framing/...
Expected Output:
==== RUN  TestParseDataFrame
--- PASS: TestParseDataFrame (0.00s)
==== RUN  TestParseHeadersFrame
--- PASS: TestParseHeadersFrame (0.00s)
==== RUN  TestParseSettingsFrame
--- PASS: TestParseSettingsFrame (0.00s)
==== RUN  TestFrameValidation
--- PASS: TestFrameValidation (0.00s)
==== RUN  TestFrameSerialization
--- PASS: TestFrameSerialization (0.00s)
PASS
```

Manual Verification Steps:

- Frame Parsing Test:** Create a minimal test program that reads HTTP/2 frames from a file and parses them using `ParseFrame`. Generate test frames using another HTTP/2 implementation or hex-encoded test vectors.
- Serialization Round-Trip:** Verify that parsing a frame followed by serialization produces identical binary output. Any differences indicate endianness issues or field corruption.
- Error Handling:** Send malformed frames and verify the server generates appropriate `ProtocolError` instances with correct error codes.

Expected Behavior Indicators:

- Frame parsing correctly extracts all header fields (length, type, flags, stream ID)
- Payload reading respects the length field without reading too much or too little data
- Frame validation catches oversized frames, invalid stream ID assignments, and reserved bit violations
- Serialization produces valid wire format that other HTTP/2 implementations can parse

Common Issues and Debugging:

Symptom	Likely Cause	Diagnostic Command	Resolution
Parse errors on valid frames	Endianness issues	<code>hexdump -C frame.bin</code>	Fix network byte order conversion
Payload size mismatches	Length field corruption	Debug frame header parsing	Verify 24-bit length extraction
Validation false positives	Incorrect reserved bit handling	Test with known-good frames	Fix bit masking logic
Serialization differences	Field order or padding	Compare serialized output byte-by-byte	Check frame structure alignment

Milestone 2 Checkpoint: HPACK Compression

After implementing HPACK compression, the server should correctly decompress header blocks and maintain dynamic table state across requests. The checkpoint validates static table lookups, dynamic table management, Huffman decoding, and integer encoding without requiring complete stream processing.

Automated Test Verification:

```
Test Command: go test -v ./internal/hpack/...
Expected Output:
==== RUN  TestStaticTableLookup
--- PASS: TestStaticTableLookup (0.00s)
==== RUN  TestDynamicTableManagement
--- PASS: TestDynamicTableManagement (0.00s)
==== RUN  TestHuffmanDecoding
--- PASS: TestHuffmanDecoding (0.00s)
==== RUN  TestIntegerEncoding
--- PASS: TestIntegerEncoding (0.00s)
==== RUN  TestHeaderBlockDecoding
--- PASS: TestHeaderBlockDecoding (0.00s)
PASS
```

Manual Verification Steps:

- Static Table Verification:** Query all static table entries (indices 1-61) and verify they match RFC 7541 Appendix B exactly.
- Dynamic Table Evolution:** Process a sequence of header blocks that add entries to the dynamic table and verify proper FIFO eviction when size limits are exceeded.
- Huffman Decoding Test:** Decode Huffman-compressed header values using test vectors from RFC 7541 and verify correct string reconstruction.
- Cross-Implementation Testing:** Exchange header blocks with another HTTP/2 implementation to verify compression compatibility.

Expected Behavior Indicators:

- Static table lookups return correct name-value pairs for all valid indices
- Dynamic table size management maintains the configured size limit through proper eviction
- Huffman decoding handles all character encodings including edge cases and padding
- Integer encoding/decoding works correctly for all prefix lengths (5, 6, 7 bits)
- Header block decompression produces semantically correct HTTP headers

HPACK-Specific Debugging:

Symptom	Likely Cause	Investigation Steps	Fix
Decompression failures	Dynamic table inconsistency	Log table state after each header block	Verify FIFO eviction order
Huffman decode errors	Padding validation issues	Test with known Huffman sequences	Fix EOS symbol handling
Integer overflow	Continuation byte handling	Test large integer values	Implement proper bounds checking
Table size violations	Entry size calculation errors	Verify overhead calculation (32 bytes)	Fix size accounting formula

Milestone 3 Checkpoint: Stream Management

After implementing stream management, the server should handle multiple concurrent streams with correct state transitions and priority scheduling. The checkpoint validates stream lifecycle management, state machine compliance, and concurrent stream handling without requiring complete flow control.

Automated Test Verification:

```
Test Command: go test -v ./internal/streams/...
Expected Output:
==== RUN  TestStreamStateTransitions
--- PASS: TestStreamStateTransitions (0.00s)
==== RUN  TestStreamIDAllocation
--- PASS: TestStreamIDAllocation (0.00s)
==== RUN  TestPriorityScheduling
--- PASS: TestPriorityScheduling (0.00s)
==== RUN  TestConcurrentStreams
--- PASS: TestConcurrentStreams (0.00s)
==== RUN  TestStreamLimits
--- PASS: TestStreamLimits (0.00s)
PASS
```

Manual Verification Steps:

- Stream Lifecycle Testing:** Open multiple streams simultaneously and verify they transition through states correctly as frames are sent and received.
- Priority Scheduling Verification:** Configure streams with different weights and verify bandwidth allocation matches priority ratios under load.
- Concurrent Access Testing:** Send frames for multiple streams simultaneously from different goroutines and verify thread-safe state management.
- Stream Limit Enforcement:** Attempt to exceed the maximum concurrent streams setting and verify proper rejection with appropriate error codes.

Expected Behavior Indicators:

- Stream state transitions follow RFC 7540 state machine exactly
- Stream ID allocation maintains odd/even separation and monotonic ordering
- Priority scheduling distributes bandwidth proportionally based on stream weights
- Concurrent stream access doesn't cause race conditions or state corruption
- Stream limits prevent resource exhaustion through excessive stream creation

Stream Management Debugging:

Symptom	Likely Cause	Debug Approach	Solution
Invalid state transitions	State machine bugs	Log all state changes with frame types	Fix transition validation logic
Stream ID collisions	ID allocation races	Test concurrent allocation	Add proper synchronization
Priority inversion	Scheduling algorithm bugs	Monitor bandwidth distribution	Fix deficit round-robin implementation
Memory leaks	Stream cleanup failures	Monitor stream count over time	Implement proper stream cleanup

Milestone 4 Checkpoint: Complete HTTP/2 Server

After implementing flow control, the server should provide complete HTTP/2 protocol compliance with multiplexing, header compression, and flow control. The final checkpoint validates end-to-end functionality using real HTTP/2 clients and comprehensive protocol compliance testing.

Automated Test Verification:

```
Test Command: go test -v ./...
Expected Output: All test packages pass
Integration Command: h2spec --host localhost --port 8443 --tls
Expected Output: All h2spec tests pass
```

Manual Verification Steps:

- 1. Browser Testing:** Load a complex web page with multiple resources (CSS, JavaScript, images) and verify all resources load correctly via HTTP/2.
- 2. curl Integration:** Execute various curl commands testing different HTTP methods, headers, and response sizes.
- 3. Performance Testing:** Run h2load to verify the server handles concurrent load without degradation.
- 4. Error Handling:** Trigger various error conditions and verify appropriate HTTP/2 error responses.

Expected Behavior Indicators:

- All HTTP/2 clients connect successfully and exchange data correctly
- Stream multiplexing allows concurrent request processing over single connections
- Flow control prevents buffer overflow and handles backpressure appropriately
- Header compression reduces bandwidth usage compared to uncompressed headers
- Error conditions generate proper HTTP/2 error frames (GOAWAY, RST_STREAM)

Final Integration Issues:

Problem	Symptoms	Root Cause Analysis	Resolution
Connection hangs	Client timeouts	Check flow control deadlocks	Verify window update generation
Performance degradation	High latency under load	Profile goroutine and memory usage	Optimize critical path performance
Intermittent failures	Sporadic test failures	Look for race conditions	Add proper synchronization
Protocol compliance failures	h2spec test failures	Review RFC compliance	Fix specific protocol violations

Testing Success Criteria: A fully functional HTTP/2 server should pass all unit tests, handle real client traffic correctly, achieve acceptable performance under load, and maintain protocol compliance under all tested conditions. Any failures in these areas indicate implementation bugs that must be resolved before deployment.

Implementation Guidance

The HTTP/2 server testing implementation requires sophisticated test infrastructure that mirrors the protocol's complexity. Unlike simple REST API testing, HTTP/2 testing must handle binary protocols, stateful

compression, concurrent streams, and complex state machines. The testing strategy builds from simple unit tests to complete protocol compliance validation.

Technology Recommendations

Testing Component	Simple Option	Advanced Option
Unit Testing Framework	Standard Go testing package	Testify with assert/require helpers
HTTP/2 Test Client	net/http.Client with ForceAttemptHTTP2	Custom HTTP/2 client with frame control
Binary Frame Testing	Hand-coded byte arrays	Property-based testing with quick
Load Testing	Simple go routines	h2load or custom performance harness
Protocol Compliance	Manual test cases	h2spec automated compliance testing
Mocking/Stubs	Interface-based mocking	gomock for generated mocks
Test Data Management	Embedded test vectors	External test data files

Recommended File Structure

```
project-root/
├── cmd/
│   ├── server/
│   │   ├── main.go           ← HTTP/2 server entry point
│   │   └── main_test.go      ← Integration test setup
│   └── test-client/
│       └── main.go          ← Custom HTTP/2 test client
├── internal/
│   ├── framing/
│   │   ├── parser.go         ← Frame parsing implementation
│   │   ├── parser_test.go    ← Frame parsing unit tests
│   │   ├── serializer.go     ← Frame serialization
│   │   ├── serializer_test.go← Serialization unit tests
│   │   └── validator_test.go ← Frame validation tests
│   ├── hpack/
│   │   ├── tables.go          ← Static/dynamic table implementation
│   │   ├── tables_test.go     ← Table management unit tests
│   │   ├── huffman.go         ← Huffman encoding/decoding
│   │   ├── huffman_test.go    ← Huffman unit tests
│   │   └── integration_test.go← HPACK integration tests
│   ├── streams/
│   │   ├── manager.go         ← Stream lifecycle management
│   │   ├── manager_test.go    ← Stream management unit tests
│   │   ├── statemachine_test.go← State transition tests
│   │   └── priority_test.go   ← Priority scheduling tests
│   └── flow/
│       ├── controller.go      ← Flow control implementation
│       ├── controller_test.go  ← Flow control unit tests
│       └── backpressure_test.go← Backpressure handling tests
└── test/
    ├── integration/
    │   ├── client_test.go      ← Real client integration tests
    │   ├── compliance_test.go  ← Protocol compliance tests
    │   └── performance_test.go ← Load testing and benchmarks
    ├── fixtures/
    │   ├── frames/             ← Binary frame test data
    │   ├── headers/            ← HPACK test vectors
    │   └── traces/             ← HTTP/2 connection traces
    └── utils/
        ├── client.go           ← Test client utilities
        ├── server.go           ← Test server setup
        └── assertions.go        ← Custom test assertions
└── scripts/
    ├── run-h2spec.sh          ← Protocol compliance testing
    ├── load-test.sh           ← Performance testing script
    └── generate-test-data.sh  ← Test data generation
```

Testing Infrastructure Starter Code

```
// test/utils/server.go - Test server setup utilities  
GO  
  
package testutils  
  
import (  
    "crypto/tls"  
    "net"  
    "testing"  
    "time"  
  
    "github.com/yourproject/internal/server"  
)  
  
// TestServerConfig configures test server instances  
  
type TestServerConfig struct {  
  
    TLS          bool  
    Port         int  
    MaxStreams  uint32  
    WindowSize   uint32  
    FrameSize    uint32  
    ReadTimeout  time.Duration  
    WriteTimeout time.Duration  
  
}  
  
// StartTestServer creates and starts an HTTP/2 server for testing  
  
func StartTestServer(t *testing.T, config TestServerConfig) (*server.Server, net.Addr) {  
    // TODO 1: Create server instance with test configuration  
  
    // TODO 2: Generate self-signed certificate for TLS testing
```

```
// TODO 3: Start server on random available port

// TODO 4: Return server instance and actual listen address

// TODO 5: Register cleanup function to stop server when test completes

}

// CreateTestTLSConfig generates TLS configuration for testing

func CreateTestTLSConfig() *tls.Config {

    // TODO 1: Generate ephemeral certificate and key for testing

    // TODO 2: Configure TLS for HTTP/2 ALPN negotiation

    // TODO 3: Set appropriate cipher suites and TLS version

    // TODO 4: Return TLS config suitable for test server

}

// WaitForServerReady blocks until test server accepts connections

func WaitForServerReady(addr net.Addr, timeout time.Duration) error {

    // TODO 1: Attempt TCP connection to server address

    // TODO 2: Retry with exponential backoff until success or timeout

    // TODO 3: Return error if server doesn't become ready within timeout

}
```

GO

```
// test/utils/client.go - HTTP/2 test client utilities

package testutils

import (
    "bytes"
    "context"
    "crypto/tls"
    "io"
    "net/http"
    "testing"

    "golang.org/x/net/http2"
)

// TestClient provides HTTP/2-specific client functionality for testing

type TestClient struct {
    client    *http.Client
    transport *http2.Transport
    baseURL   string
}

// NewTestClient creates HTTP/2 client configured for testing

func NewTestClient(serverAddr string, tlsConfig *tls.Config) *TestClient {
    // TODO 1: Create http2.Transport with test-specific settings

    // TODO 2: Configure transport for HTTP/2 over TLS or cleartext

    // TODO 3: Set appropriate timeouts and retry settings

    // TODO 4: Return configured test client instance
}
```

```
// SendFrameSequence sends custom HTTP/2 frame sequence for protocol testing

func (tc *TestClient) SendFrameSequence(ctx context.Context, frames []Frame) error {

    // TODO 1: Establish raw TCP connection to server

    // TODO 2: Perform TLS handshake if required

    // TODO 3: Send HTTP/2 connection preface

    // TODO 4: Send each frame in sequence using SerializeFrame

    // TODO 5: Read and validate server responses

}

// TestMultiplexing creates multiple concurrent streams for multiplexing tests

func (tc *TestClient) TestMultiplexing(ctx context.Context, requestCount int)
([]ResponseMetrics, error) {

    // TODO 1: Create specified number of concurrent HTTP requests

    // TODO 2: Send all requests simultaneously using goroutines

    // TODO 3: Measure response times and verify all succeed

    // TODO 4: Return timing metrics for analysis

}

// ResponseMetrics captures timing and correctness data for test analysis

type ResponseMetrics struct {

    StreamID      uint32

    StatusCode    int

    ResponseTime time.Duration

    BytesRead     int64

    Headers       http.Header

    Error         error

}
```

Core Testing Skeleton Code

```
// internal/framing/parser_test.go - Frame parsing unit tests          GO

package framing

import (
    "bytes"
    "testing"
)

// TestParseValidFrames verifies parsing of all valid frame types

func TestParseValidFrames(t *testing.T) {
    testCases := []struct {
        name      string
        frameBytes []byte
        expectedType uint8
        expectedLen  uint32
        expectedFlags uint8
    }{
        // TODO 1: Add test case for DATA frame with various payload sizes
        // TODO 2: Add test case for HEADERS frame with END_HEADERS flag
        // TODO 3: Add test case for SETTINGS frame with parameters
        // TODO 4: Add test case for WINDOW_UPDATE frame
        // TODO 5: Add test case for maximum size frame (16MB)
    }

    for _, tc := range testCases {
        t.Run(tc.name, func(t *testing.T) {
            // TODO 1: Create bytes.Reader from test frame data
        })
    }
}
```

```
// TODO 2: Call ParseFrame and verify no error returned

// TODO 3: Assert frame header fields match expected values

// TODO 4: Verify payload length matches frame length field

// TODO 5: Check frame flags are parsed correctly

    })

}

}

// TestFrameValidationErrors verifies detection of protocol violations

func TestFrameValidationErrors(t *testing.T) {

    testCases := []struct {
        name      string
        frame     *Frame
        expectedError ErrorCode
    }{

        // TODO 1: Frame exceeding maximum size limit

        // TODO 2: Reserved bit set in stream ID field

        // TODO 3: Invalid frame type for stream ID 0

        // TODO 4: CONTINUATION frame without preceding HEADERS

        // TODO 5: Settings frame with invalid parameter count
    }

    for _, tc := range testCases {

        t.Run(tc.name, func(t *testing.T) {

            // TODO 1: Call ValidateFrame with test frame

            // TODO 2: Verify appropriate error is returned

            // TODO 3: Check error code matches expected value
        })
    }
}
```

```
// TODO 4: Ensure error message is descriptive

    })

}

}

// BenchmarkFrameParsing measures frame parsing performance

func BenchmarkFrameParsing(b *testing.B) {

    // TODO 1: Create representative frame data for benchmarking

    // TODO 2: Setup bytes.Reader for repeated parsing

    // TODO 3: Run ParseFrame in benchmark loop

    // TODO 4: Measure allocations and CPU usage

    // TODO 5: Report throughput in frames/second

}
```

GO

```
// test/integration/compliance_test.go - Protocol compliance testing

package integration

import (
    "context"
    "os/exec"
    "testing"
    "time"

    "github.com/yourproject/test/utils"
)

// TestH2SpecCompliance runs h2spec protocol compliance tests

func TestH2SpecCompliance(t *testing.T) {
    if testing.Short() {
        t.Skip("Skipping h2spec compliance tests in short mode")
    }

    // TODO 1: Start test server with default configuration
    // TODO 2: Wait for server to become ready
    // TODO 3: Execute h2spec command against test server
    // TODO 4: Parse h2spec output and check for failures
    // TODO 5: Report detailed results for any failing test cases

    // Test should verify these h2spec sections:
    // - Section 3.5: HTTP/2 Connection Preface
    // - Section 4: HTTP/2 Frames
    // - Section 5: Streams and Multiplexing
```

```

    // - Section 6: Frame Definitions

    // - Section 8: HTTP Message Exchanges

}

// TestCurlCompatibility verifies compatibility with curl HTTP/2 client

func TestCurlCompatibility(t *testing.T) {

    // TODO 1: Start HTTPS test server with valid certificate

    // TODO 2: Execute curl --http2 commands for various scenarios

    // TODO 3: Verify curl reports successful HTTP/2 usage

    // TODO 4: Check response content correctness

    // TODO 5: Test edge cases like large headers, POST data, custom headers

}

// TestBrowserCompatibility validates real browser HTTP/2 behavior

func TestBrowserCompatibility(t *testing.T) {

    // TODO 1: Start HTTPS server with browser-trusted certificate

    // TODO 2: Use selenium/playwright to automate browser testing

    // TODO 3: Load test pages with multiple resources (CSS/JS/images)

    // TODO 4: Verify all resources load via HTTP/2 multiplexing

    // TODO 5: Check developer tools show HTTP/2 protocol usage

}

```

Language-Specific Testing Hints

Go Testing Best Practices:

- Use `testing.T.Helper()` in utility functions to improve stack trace reporting
- Leverage `testing.T.Cleanup()` for automatic resource cleanup instead of defer
- Use `testing.TB` interface for functions that work with both tests and benchmarks
- Apply build tags (`// +build integration`) to separate unit and integration tests
- Use `go test -race` to detect race conditions in concurrent stream handling

HTTP/2 Protocol Testing:

- Always test with both TLS and cleartext HTTP/2 (h2c) configurations
- Use `golang.org/x/net/http2/h2demo` package for test server examples
- Capture network traces with Wireshark during debugging for frame-level analysis
- Test with multiple `GOMAXPROCS` values to expose concurrency bugs
- Use `pprof` profiling during load tests to identify performance bottlenecks

Binary Protocol Testing:

- Store test vectors as hex strings for readability: `unhex("000004 00 00 000000 deadbeef")`
- Use `encoding/binary` for consistent endianness handling in test data generation
- Create property-based tests for frame fuzzing using `testing/quick` package
- Test boundary conditions: zero-length frames, maximum-size frames, invalid lengths
- Verify bit-level operations with explicit bit manipulation tests

Debugging Strategy

Testing Phase	Common Issues	Diagnostic Approach	Resolution Strategy
Unit Testing	Frame parsing failures	Hex dump comparison with expected	Fix endianness and length handling
Integration Testing	Client connection failures	Wireshark packet capture	Check TLS configuration and ALPN
Load Testing	Performance degradation	Go profiling (CPU/memory)	Optimize hot paths and reduce allocations
Compliance Testing	h2spec failures	Read RFC sections for failed tests	Implement missing protocol requirements

Debugging Guide

Milestone(s): This section provides debugging strategies and troubleshooting techniques for all milestones: frame parsing issues (Milestone 1), HPACK compression bugs (Milestone 2), stream state violations (Milestone 3), and flow control problems (Milestone 4).

Think of debugging an HTTP/2 server like troubleshooting a complex postal system. Just as mail can get lost at sorting centers, have addresses corrupted, or be delivered out of order, HTTP/2 frames can be malformed at the binary level, headers can be corrupted during compression, or streams can enter invalid states due to concurrency issues. The key to effective debugging is understanding where in the pipeline problems occur and having systematic approaches to isolate and diagnose each category of failure.

HTTP/2 protocol debugging presents unique challenges because of its binary nature, multiplexed streams, and complex state machines. Unlike HTTP/1.1 where you can inspect raw text traffic, HTTP/2 requires specialized tools to decode binary frames. The multiplexing adds another layer of complexity - errors in one stream can affect others, and race conditions between concurrent streams create intermittent failures that are difficult to reproduce. This section provides concrete diagnostic techniques, common bug patterns, and systematic troubleshooting approaches for each major component.

The debugging strategy follows a layered approach, starting with the lowest level (binary framing) and working up through header compression, stream management, and flow control. Each layer has characteristic failure modes and diagnostic signatures. Understanding these patterns allows you to quickly identify which component is failing and apply the appropriate troubleshooting techniques.

Frame Parsing Bugs

Binary protocol parsing represents one of the most common sources of bugs in HTTP/2 implementations. Unlike text-based protocols where parsing errors are often obvious, binary parsing failures can manifest as subtle data corruption, mysterious connection resets, or intermittent protocol violations that are difficult to reproduce.

Endianness Issues

Network protocols use big-endian byte ordering (most significant byte first), but different architectures have different native byte orders. Think of endianness like reading a multi-digit number - in big-endian, you read left-to-right (most significant digit first), while in little-endian, you would read right-to-left. When your HTTP/2 frame parser doesn't correctly handle byte order conversion, field values become corrupted in predictable ways.

The `Frame` header contains several multi-byte fields that are susceptible to endianness bugs. The 24-bit length field, when incorrectly parsed on a little-endian system, will have its bytes reversed. For example, a frame length of `0x000100` (256 bytes) would be interpreted as `0x0000001` (1 byte) on a little-endian system without proper conversion.

Field	Correct Value	Little-Endian Bug	Symptom
Length (24-bit)	0x000100 (256)	0x0000001 (1)	Premature frame termination
StreamID (31-bit)	0x00000001	0x01000000	Invalid stream ID errors
Window Size	0x0000FFFF	0xFFFF0000	Flow control violations

⚠ Pitfall: Native Byte Order Assumption Many developers assume their development machine's byte order matches the network byte order. This leads to code that works correctly on big-endian systems but fails on common little-endian architectures like x86. The bug often goes undetected during development if testing only occurs on the same architecture as development.

The diagnostic signature of endianness bugs includes frame length mismatches, where the parser reads far more or fewer bytes than expected for the payload. You'll see protocol errors like

`ErrorCodeFrameSizeError` when the computed frame size exceeds `MaxFrameSize`, or premature connection termination when the parser tries to read beyond available data.

Length Validation Errors

HTTP/2 frame length validation involves multiple constraints that must all be satisfied simultaneously. The 24-bit length field in the frame header specifies payload bytes, not including the 9-byte header itself. This length must not exceed the negotiated maximum frame size (initially `DefaultFrameSize` of 16,384 bytes), and certain frame types have additional size constraints.

Frame length validation follows a hierarchical checking process:

1. **Basic Range Check:** Length field must not exceed `MaxFrameSize` (16,777,215 bytes)
2. **Connection Limit Check:** Length must not exceed the negotiated `SETTINGS_MAX_FRAME_SIZE` value
3. **Frame Type Constraints:** Some frame types have minimum or exact size requirements
4. **Stream Context Validation:** Frame size must be reasonable given current stream state and flow control windows

Frame Type	Minimum Size	Maximum Size	Special Constraints
<code>FrameTypeDATA</code>	0 bytes	Connection limit	Must respect flow control
<code>FrameTypeHEADERS</code>	1 byte	Connection limit	Must include header block
<code>FrameTypeSETTINGS</code>	0 or 6n bytes	Connection limit	Multiple of 6 when not ACK
PING	8 bytes	8 bytes	Exactly 8 bytes required
GOAWAY	8+ bytes	Connection limit	At least last stream ID + error

⚠ Pitfall: Settings Frame Size Validation SETTINGS frames without the ACK flag must contain a payload length that's a multiple of 6 bytes (each setting is 6 bytes). However, SETTINGS frames with the ACK flag must have zero payload length. Many implementations forget this distinction and either reject valid ACK frames or accept malformed settings with partial entries.

Binary parsing also introduces **alignment and boundary errors** where frame boundaries don't align with buffer boundaries. When reading frames from a TCP stream, the underlying transport can deliver data in arbitrary chunks that don't correspond to frame boundaries. Your parser must handle cases where a frame header spans multiple read operations, or where multiple complete frames arrive in a single buffer.

The frame parsing state machine must track partial reads and reassemble complete frames:

1. **Header Assembly State:** Accumulating the 9-byte header across multiple reads
2. **Payload Reading State:** Reading the specified number of payload bytes
3. **Frame Complete State:** Frame ready for validation and processing
4. **Error Recovery State:** Handling malformed frames and connection recovery

Buffer Overflow and Underflow

Buffer management in binary parsing requires careful bounds checking at every step. **Buffer overflow** occurs when the parser writes beyond allocated memory, typically when the frame length field has been corrupted or when integer arithmetic overflows during size calculations. **Buffer underflow** occurs when the parser attempts to read more data than is available in the current buffer.

Consider the frame header parsing sequence. The parser must read exactly 9 bytes for the header, then extract the 24-bit length field, and finally read that many additional bytes for the payload. At each step, the parser must verify that sufficient data is available before attempting the read operation.

The critical insight for robust frame parsing is that network data arrives asynchronously and in arbitrary chunks. Your parser cannot assume that complete frames are available in each read operation - it must gracefully handle partial frames and maintain parsing state across multiple I/O operations.

Reserved Bit Handling

HTTP/2 frame headers contain several reserved bits that must be zero in compliant implementations. The most common reserved bit violations occur in the stream ID field, which reserves the most significant bit (bit 31) for the R flag. Implementations must mask this bit when extracting the stream ID value and should validate that reserved bits are not set.

Field	Reserved Bits	Validation Rule	Error Response
Stream ID	Bit 31 (R flag)	Must be 0	ErrorCodeProtocolError
Frame Type	Bits 8-15	Must be recognized type	Connection error
Flags	Unused bits per type	Type-specific validation	Frame-specific error

The `ValidateFrame` method must check these reserved bit constraints and generate appropriate `ProtocolError` responses. Reserved bit violations typically indicate either a buggy peer implementation or data corruption during transmission.

HPACK Compression Bugs

HPACK header compression introduces complex state management and encoding algorithms that are prone to subtle bugs. Unlike frame parsing where errors are often immediately apparent, HPACK bugs can manifest as gradual state corruption that only becomes visible after processing many requests. Think of HPACK debugging like troubleshooting a dictionary where entries are constantly being added and removed - small errors in table management compound over time until the sender and receiver have completely different dictionaries.

Table Index Errors

The HPACK algorithm uses a combined indexing space that merges the static table and dynamic table into a single address space. The static table contains 61 predefined entries (indexed 1-61), while the dynamic table entries follow immediately after (indexed 62 and above). This dual addressing scheme is a common source of off-by-one errors and index calculation bugs.

Index resolution follows these rules:

1. **Index 0**: Reserved and invalid - should trigger decompression error
2. **Index 1-61**: Static table entries, directly addressable
3. **Index 62+**: Dynamic table entries, calculated as `(index - 62)` into dynamic table array
4. **Index > (61 + dynamic_table_size)**: Invalid index, should trigger decompression error

Index Range	Table	Calculation	Error Condition
0	Invalid	N/A	<code>ErrorCodeCompressionError</code>
1-61	Static	<code>index - 1</code> for array access	Index > 61
62+	Dynamic	<code>index - 62</code> for array access	Index > 61 + table size

⚠ Pitfall: Dynamic Table Index Calculation The most common HPACK indexing bug is forgetting that dynamic table indices start at 62, not 61. This leads to off-by-one errors where index 62 incorrectly references static table entry 61 instead of dynamic table entry 0. The bug manifests as header values from previous requests appearing in current responses, creating security vulnerabilities through information leakage.

The `Lookup` method must handle index bounds checking and table selection:

```
Input: index = 65, static_table_size = 61, dynamic_table_entries = 5
Validation: 65 <= (61 + 5) = 66 ✓ (valid)
Table Selection: 65 > 61 → Dynamic table
Array Index: 65 - 62 = 3 (fourth dynamic entry)
```

Dynamic Table Size Management

HPACK dynamic tables implement FIFO eviction with size-based constraints rather than simple entry count limits. Each header entry consumes `32 + name.length + value.length` bytes, and the table must maintain a running total that never exceeds the configured maximum size. Size management bugs typically involve incorrect size calculations, improper eviction logic, or race conditions in multithreaded implementations.

The `calculateEntrySize` function implements the RFC 7541 sizing formula, but implementations often introduce bugs in string length calculation or integer overflow handling. Consider a header entry with name ":status" and value "200" - the size calculation is `32 + 7 + 3 = 42` bytes. However, if the implementation uses 16-bit integers for size tracking, tables larger than 65,535 bytes will overflow and wrap around to small positive values.

Component	Size Contribution	Common Bugs
Base Overhead	32 bytes	Forgetting to include
Name Length	UTF-8 byte count	Using character count instead
Value Length	UTF-8 byte count	Unicode handling errors
Total Tracking	Running sum	Integer overflow

The eviction algorithm must remove entries from the front of the table (oldest entries) until the total size is within limits. However, the eviction loop must carefully handle cases where adding a single large entry requires removing multiple small entries.

The critical insight for dynamic table management is that HPACK uses FIFO eviction based on total byte size, not entry count. A single large entry can force eviction of many smaller entries, and the eviction algorithm must be atomic to prevent inconsistent intermediate states.

Huffman Decoding Issues

HPACK uses Huffman coding for header value compression, but the decoding algorithm is complex and error-prone. The Huffman code table is predefined by RFC 7541, but implementations must handle variable-length codes, bit-level parsing, and padding validation. Huffman decoding bugs often manifest as corrupted header values, infinite loops in the decoder, or security vulnerabilities through buffer overflows.

The Huffman decoding process operates on individual bits rather than bytes, requiring careful bit manipulation and accumulator management. The `HuffmanDecoder` maintains an accumulator for partially decoded symbols and a bit count for tracking position within the current byte.

Huffman decoding follows this bit-level process:

- 1. Bit Accumulation:** Add input bits to the accumulator, tracking total bit count
- 2. Symbol Matching:** Compare accumulator against code table for valid symbols
- 3. Symbol Emission:** Output matched character and reset accumulator
- 4. Padding Validation:** Ensure any remaining bits match the End-of-String padding pattern

⚠ Pitfall: Huffman Padding Validation The Huffman alphabet includes a special End-of-String symbol (`EOS_SYMBOL`) used for padding incomplete bytes. Decoders must validate that any remaining bits after the last complete symbol consist only of the most significant bits of the EOS symbol. Invalid padding indicates either corruption or a malicious compression bomb attack.

Integer Encoding Edge Cases

HPACK uses variable-length integer encoding with different prefix sizes (5, 6, or 7 bits) depending on context. The encoding can span multiple bytes using continuation markers, and decoders must handle very large

integers and potential overflow conditions. Integer decoding bugs include infinite loops on malformed continuation bytes, arithmetic overflow, and incorrect prefix masking.

The `DecodeInteger` method must handle several edge cases:

Edge Case	Description	Validation Required
Maximum Prefix Value	All prefix bits set to 1	Indicates multi-byte encoding
Continuation Bytes	MSB = 1 indicates more bytes	Must limit total byte count
Integer Overflow	Result exceeds language limits	Check before arithmetic operations
Invalid Continuation	MSB patterns	Detect malformed encoding

The integer decoding algorithm accumulates values using powers of 128, but must prevent arithmetic overflow. Consider decoding a 5-bit prefix where all prefix bits are set (value 31) followed by continuation bytes. The total value is `31 + (continuation_bytes_value * 128^n)`, where n is the continuation byte position.

Stream State Bugs

HTTP/2 stream management implements complex state machines that can transition through multiple states based on frame types and flags. Stream state bugs are particularly insidious because they often involve race conditions between concurrent streams and can create security vulnerabilities through state machine violations. Think of stream state debugging like tracking the lifecycle of individual conversations in a busy conference call - each conversation has its own state, but they can interfere with each other in unexpected ways.

State Machine Violations

The HTTP/2 stream state machine defines valid transitions between `StateIdle`, `StateOpen`, `StateHalfClosedLocal`, `StateHalfClosedRemote`, and `StateClosed`. Invalid transitions occur when frames arrive that are not permitted in the current state, or when the implementation incorrectly updates state without proper validation.

Current State	Valid Frames	Invalid Frames	Transition Triggers
StateIdle	HEADERS, PRIORITY	DATA, RST_STREAM	HEADERS → Open
StateOpen	DATA, HEADERS, RST_STREAM	None (all valid)	END_STREAM → Half-closed
StateHalfClosedLocal	DATA (receive only), RST_STREAM	DATA (send)	END_STREAM received → Closed
StateHalfClosedRemote	DATA (send only), RST_STREAM	DATA (receive)	END_STREAM sent → Closed
StateClosed	PRIORITY, RST_STREAM	DATA, HEADERS	None (terminal)

⚠ Pitfall: Concurrent State Transitions Stream state transitions are not atomic across frame processing, leading to race conditions where two threads attempt to transition the same stream simultaneously. For example, one thread processes an incoming RST_STREAM frame while another processes an outgoing DATA frame. Without proper synchronization, the stream can end up in an inconsistent state or generate invalid frames.

The `TransitionState` method must validate transitions atomically:

1. **Lock Stream State:** Acquire exclusive access to prevent concurrent modifications
2. **Validate Current State:** Ensure the stream is in the expected current state
3. **Check Frame Permissions:** Verify the triggering frame is valid for current state
4. **Update State:** Atomically transition to the new state
5. **Notify Waiters:** Signal any threads waiting on state changes
6. **Release Lock:** Allow other operations to proceed

Stream ID Allocation Issues

HTTP/2 uses a split namespace for stream IDs - client-initiated streams use odd numbers (1, 3, 5, ...) while server-initiated streams use even numbers (2, 4, 6, ...). Stream ID allocation must maintain monotonic increasing order within each namespace and handle ID exhaustion gracefully. Stream ID bugs include reusing IDs, violating the odd/even rule, or failing to handle the 31-bit limit.

The `AllocateStreamID` method tracks separate counters for client and server streams:

Stream Type	ID Pattern	Counter Variable	Exhaustion Point
Client Streams	1, 3, 5, ...	<code>nextClientStreamID</code>	$2^{31} - 1$
Server Streams	2, 4, 6, ...	<code>nextServerStreamID</code>	$2^{31} - 2$
Reserved	0	N/A	Connection-level frames

Stream ID exhaustion requires connection closure because HTTP/2 does not support ID reuse. A busy server processing many short-lived requests can exhaust the server stream ID space, requiring graceful connection termination with a GOAWAY frame.

⚠ Pitfall: Stream ID Reuse Some implementations incorrectly attempt to reuse stream IDs after streams close, violating the HTTP/2 specification. This can cause frame routing confusion where frames intended for a new stream are delivered to a closed stream's handler, leading to protocol errors or security vulnerabilities.

Priority and Dependency Cycles

HTTP/2 stream prioritization uses a weighted dependency tree where streams can depend on other streams for scheduling priority. However, the dependency relationships must form a directed acyclic graph - dependency cycles create scheduling deadlocks where streams wait for each other indefinitely. Priority handling bugs include allowing circular dependencies, incorrect weight calculations, and priority inversion where low-priority streams block high-priority ones.

The `StreamPriority` structure defines dependency relationships, but implementations must validate that new dependencies don't create cycles:

1. **Cycle Detection:** Use depth-first search to detect circular dependencies before adding new edges
2. **Dependency Removal:** Remove conflicting dependencies to break potential cycles
3. **Weight Normalization:** Ensure priority weights fall within valid ranges (1-256)
4. **Scheduling Fairness:** Implement deficit round-robin or similar algorithm to prevent starvation

Priority Violation	Detection Method	Recovery Action
Circular Dependency	Graph traversal	Remove newest dependency
Invalid Weight	Range checking	Clamp to valid range (1-256)
Missing Parent	Dependency validation	Depend on connection root
Priority Inversion	Scheduling analysis	Boost blocked high-priority streams

Concurrent Access Issues

Stream management must handle concurrent access from multiple goroutines processing different streams simultaneously. Race conditions occur when multiple threads access shared stream state, when stream cleanup races with active frame processing, or when flow control windows are updated concurrently with data transmission.

The `StreamManager` must provide thread-safe operations for all stream lifecycle events:

Operation	Synchronization Required	Race Condition Risk
Stream Creation	Stream map write lock	Duplicate stream IDs
State Transitions	Per-stream lock	Inconsistent state
Flow Control Updates	Window lock	Double accounting
Stream Cleanup	Stream map write lock	Use-after-free

The critical insight for stream concurrency is that HTTP/2 multiplexing creates complex interaction patterns between streams that share connection-level resources. Proper synchronization requires both coarse-grained locks for shared data structures and fine-grained locks for per-stream state to avoid performance bottlenecks.

Debugging Tools

Effective HTTP/2 debugging requires specialized tools that can decode binary protocols, analyze multiplexed traffic, and correlate events across multiple streams. Unlike HTTP/1.1 where simple text-based tools suffice, HTTP/2 debugging demands protocol-aware analysis tools and structured logging strategies.

Wireshark Analysis

Wireshark provides comprehensive HTTP/2 protocol decoding that reveals the binary frame structure, HPACK compression details, and stream multiplexing patterns. The HTTP/2 dissector automatically decodes frame headers, decompresses HPACK headers, and tracks stream states across the connection lifetime.

Key Wireshark filters for HTTP/2 debugging:

Filter Expression	Purpose	Typical Use Case
<code>http2</code>	All HTTP/2 traffic	General protocol analysis
<code>http2.type == 0</code>	DATA frames only	Flow control and payload analysis
<code>http2.type == 1</code>	HEADERS frames	Request/response header analysis
<code>http2.streamid == N</code>	Specific stream	Single stream lifecycle tracking
<code>http2.flags.end_stream == 1</code>	Stream termination	Finding premature closures

Wireshark's HTTP/2 analysis includes automatic validation of protocol constraints. It flags invalid frame sequences, HPACK decompression errors, and flow control violations with clear error messages. The tool also provides statistics on frame type distribution, stream concurrency levels, and compression effectiveness.

⚠ Pitfall: TLS Encryption Production HTTP/2 typically uses TLS encryption, making Wireshark analysis impossible without access to the server's private key or client-side capture. Development environments should

include plaintext HTTP/2 support specifically for debugging purposes, even though production deployments mandate encryption.

Protocol-Specific Logging

HTTP/2 debugging requires logging that captures protocol-level events with sufficient detail to reconstruct connection and stream state. Generic application logs are insufficient - you need frame-level logging, HPACK table state dumps, and stream state transition tracking.

Effective HTTP/2 logging captures multiple event categories:

Log Category	Information Logged	Example Entry
Frame Processing	Type, flags, stream ID, payload size	FRAME_RECV: DATA stream=3 len=1024 flags=END_STREAM
HPACK Operations	Table operations, compression ratios	HPACK_ADD: :method=POST size=42 table_size=256
Stream Lifecycle	State transitions, creation, cleanup	STREAM_TRANSITION: stream=5 OPEN→HALF_CLOSED_REMOTE
Flow Control	Window updates, backpressure events	WINDOW_UPDATE: stream=7 delta=32768 new_window=65536
Error Conditions	Protocol violations, recovery actions	PROTOCOL_ERROR: invalid frame type=99 on stream=1

The logging system must handle high-frequency events without impacting performance. Consider using structured logging formats (JSON) that support automated analysis and correlation across multiple log files.

Frame-Level Debugging

HTTP/2's binary framing requires debugging tools that can display raw frame contents alongside decoded interpretations. Frame-level debugging helps identify endianness bugs, length validation errors, and reserved bit violations that are not apparent in higher-level protocol analysis.

Frame debugging output should include both hex dumps and structured field interpretation:

```
Frame Header (9 bytes): 00 01 00 01 00 00 00 00 03
├ Length: 0x000100 (256 bytes)
├ Type: 0x01 (HEADERS)
├ Flags: 0x00 (no flags set)
└ Stream ID: 0x00000003 (stream 3)

Payload (256 bytes): 40 0a 3a 61 75 74 68 6f 72 69 74 79...
├ HPACK Block Fragment
├ First byte: 0x40 (literal with incremental indexing)
└ Huffman encoded: yes
```

Frame-level logging must capture both successful parsing events and error conditions with sufficient detail to diagnose the root cause. Include information about buffer states, parsing positions, and validation failures.

Connection State Monitoring

HTTP/2 connections maintain complex state that spans multiple streams, HPACK tables, flow control windows, and negotiated settings. Connection state monitoring provides a centralized view of the connection health and can identify systemic issues that affect multiple streams.

State Category	Monitored Values	Alert Conditions
Active Streams	Count, state distribution	Exceeds negotiated maximum
HPACK Tables	Size, eviction rate	Frequent evictions, size violations
Flow Control	Window levels, blocked streams	Window exhaustion, deadlocks
Frame Statistics	Type distribution, error rates	High error rates, unusual patterns
Settings	Negotiated values, changes	Incompatible settings, frequent changes

Connection monitoring should provide both real-time dashboards and historical trend analysis. Look for patterns like gradually increasing error rates, stream lifecycle anomalies, or flow control window exhaustion that indicate systematic problems.

Stream Correlation Analysis

Multiplexed streams share connection resources and can interfere with each other in subtle ways. Stream correlation analysis identifies cross-stream dependencies, resource contention, and cascading failure patterns that are not apparent when examining streams individually.

Correlation Pattern	Symptom	Likely Cause
Simultaneous Stream Errors	Multiple streams fail together	Connection-level problem
Flow Control Blocking	High-priority streams stalled	Connection window exhaustion
Priority Inversion	Low-priority streams complete first	Scheduling algorithm bug
Header Corruption	Random header values appear	HPACK table corruption

Stream correlation requires aggregating events across multiple streams and identifying temporal relationships. Use time-based windowing to detect events that occur within close proximity and may be causally related.

The key insight for HTTP/2 debugging is that the protocol's multiplexed nature creates complex interaction patterns that are not visible when examining individual components in isolation. Effective debugging requires tools that can correlate events across streams, frames, and protocol layers to identify root causes.

Automated Testing and Fuzzing

HTTP/2's complexity makes it susceptible to edge case bugs that only appear under specific conditions. Automated testing and protocol fuzzing can identify implementation vulnerabilities by generating malformed frames, invalid state transitions, and resource exhaustion conditions.

Protocol fuzzing strategies for HTTP/2:

Fuzzing Target	Technique	Expected Bugs
Frame Headers	Random byte sequences	Parsing crashes, buffer overflows
HPACK Tables	Invalid indices, size violations	Table corruption, memory leaks
Stream States	Illegal frame sequences	State machine violations
Flow Control	Window overflow, underflow	Arithmetic errors, deadlocks
Integer Encoding	Continuation byte manipulation	Infinite loops, overflow

Implement fuzz testing as part of the continuous integration pipeline, not just as a one-time validation exercise. The protocol's state-dependent behavior means that bugs may only appear after specific sequences of valid operations followed by malformed input.

Implementation Guidance

This section provides concrete implementation approaches for building robust debugging capabilities into your HTTP/2 server, with specific focus on diagnostic logging, protocol validation, and troubleshooting workflows.

Technology Recommendations:

Component	Simple Option	Advanced Option
Logging Framework	Go standard log package	Structured logging with logrus/zap
Protocol Analysis	Wireshark + tcpdump	Custom protocol analyzer
Testing Framework	Go testing + httptest	Dedicated HTTP/2 test suite
Fuzzing Tools	Manual malformed frames	go-fuzz or custom fuzzer
Monitoring	Simple metrics collection	Prometheus + Grafana

File Structure:

```
internal/debugging/
├── logger.go           ← Structured HTTP/2 protocol logging
├── validator.go        ← Frame and protocol validation
├── monitor.go          ← Connection state monitoring
└── fuzzzer/
    ├── frame_fuzzer.go  ← Frame-level fuzz testing
    ├── hpack_fuzzer.go  ← HPACK compression fuzzing
    └── stream_fuzzer.go ← Stream state fuzzing
└── analysis/
    ├── wireshark_helpers.go ← Wireshark integration utilities
    ├── correlation.go     ← Stream correlation analysis
    └── statistics.go      ← Protocol statistics collection
    └── debugging_test.go   ← Test debugging infrastructure
```

Protocol Logger Infrastructure (Complete Implementation):

```
package debugging
```

```
import (
    "encoding/hex"
    "encoding/json"
    "fmt"
    "io"
    "sync"
    "time"
)
```

```
// ProtocolLogger provides structured logging for HTTP/2 protocol events
```

```
type ProtocolLogger struct {
```

```
    writer    io.Writer
    mu        sync.Mutex
    enabled   map[LogCategory]bool
    stats     map[string]uint64
    statsMu   sync.RWMutex
}
```

```
type LogCategory int
```

```
const (
    CategoryFrame LogCategory = iota
    CategoryHPACK
    CategoryStream
    CategoryFlowControl
    CategoryError
)
```

GO

```
type LogEntry struct {

    Timestamp  time.Time  `json:"timestamp"`

    Category   string    `json:"category"`

    StreamID  uint32    `json:"stream_id,omitempty"`

    Event      string    `json:"event"`

    Details    interface{} `json:"details,omitempty"`

    HexDump   string    `json:"hex_dump,omitempty"`

}

func NewProtocolLogger(writer io.Writer) *ProtocolLogger {

    return &ProtocolLogger{

        writer:  writer,

        enabled: make(map[LogCategory]bool),

        stats:   make(map[string]uint64),

    }

}

func (pl *ProtocolLogger) EnableCategory(category LogCategory) {

    pl.mu.Lock()

    defer pl.mu.Unlock()

    pl.enabled[category] = true

}

func (pl *ProtocolLogger) LogFrameEvent(streamID uint32, event string, frame *Frame) {

    if !pl.enabled[CategoryFrame] {

        return

    }
```

```
details := map[string]interface{}{
    "type":    frame.Type,
    "flags":   frame.Flags,
    "length": frame.Length,
}

var hexDump string

if len(frame.Payload) > 0 && len(frame.Payload) <= 64 {
    hexDump = hex.EncodeToString(frame.Payload)
}

pl.writeLog("frame", streamID, event, details, hexDump)
pl.incrementStat("frames_logged")
}
```

Frame Validation Infrastructure (Complete Implementation):

GO

```
// FrameValidator provides comprehensive HTTP/2 frame validation

type FrameValidator struct {

    maxFrameSize      uint32

    connectionState   *ConnectionState

    streamStates      map[uint32]StreamState

    mu                sync.RWMutex

}

func NewFrameValidator(maxFrameSize uint32) *FrameValidator {
    return &FrameValidator{
        maxFrameSize: maxFrameSize,
        streamStates: make(map[uint32]StreamState),
    }
}

// ValidateFrameStructure performs comprehensive frame structure validation

func (fv *FrameValidator) ValidateFrameStructure(frame *Frame) error {
    // TODO 1: Validate frame length against maximum allowed size

    // TODO 2: Check reserved bits in stream ID (bit 31 must be 0)

    // TODO 3: Validate frame type is recognized (0-9 are standard types)

    // TODO 4: Perform frame type specific validation (e.g., SETTINGS multiples of 6)

    // TODO 5: Check flag compatibility with frame type

    // TODO 6: Validate stream ID usage (0 for connection frames, non-zero for stream
    // frames)

    return nil
}

// ValidateStreamContext checks if frame is valid for current stream state

func (fv *FrameValidator) ValidateStreamContext(frame *Frame, streamID uint32) error {
```

```
// TODO 1: Look up current stream state from streamStates map

// TODO 2: Check if frame type is permitted in current state

// TODO 3: Validate stream ID constraints (odd for client, even for server)

// TODO 4: Check if stream ID is not lower than previously seen (monotonic)

// TODO 5: Validate END_STREAM flag usage for state transitions

return nil

}
```

HPACK Debugging Infrastructure (Core Logic Skeleton):

GO

```
// HPACKDebugger provides detailed HPACK compression/decompression logging

type HPACKDebugger struct {

    logger          *ProtocolLogger

    staticTable     []HeaderEntry

    dynamicTable   *DynamicTable

    compressionStats map[string]uint64

    mu              sync.RWMutex

}

// LogTableOperation records HPACK table modifications for debugging

func (hd *HPACKDebugger) LogTableOperation(operation string, index uint32, entry HeaderEntry) {

    // TODO 1: Calculate table size before and after operation

    // TODO 2: Log entry details including name, value, and calculated size

    // TODO 3: Record compression statistics (table hit rate, eviction frequency)

    // TODO 4: Validate index bounds and table consistency

    // TODO 5: Check for potential table corruption indicators

}

// ValidateTableState performs consistency checks on HPACK tables

func (hd *HPACKDebugger) ValidateTableState() []error {

    // TODO 1: Verify dynamic table size matches sum of entry sizes

    // TODO 2: Check that table entries are in FIFO order

    // TODO 3: Validate no duplicate entries in dynamic table

    // TODO 4: Ensure table size doesn't exceed configured maximum

    // TODO 5: Check for any null or invalid entries

    return nil

}
```

Connection State Monitor (Core Logic Skeleton):

GO

```
// ConnectionMonitor tracks overall connection health and provides diagnostics

type ConnectionMonitor struct {

    activeStreams      map[uint32]*StreamMetrics

    windowStates       *WindowManager

    hpackTables        *HPACKDebugger

    errorCounts        map[ErrorCode]uint64

    lastActivity       time.Time

    mu                 sync.RWMutex

}

type StreamMetrics struct {

    StreamID          uint32

    State              StreamState

    BytesSent          uint64

    BytesReceived      uint64

    FrameCount         uint64

    LastActivity       time.Time

    Priority           StreamPriority

}

// GenerateHealthReport creates comprehensive connection diagnostic report

func (cm *ConnectionMonitor) GenerateHealthReport() *HealthReport {

    // TODO 1: Collect statistics on active streams (count, states, activity)

    // TODO 2: Analyze flow control window levels and blocked streams

    // TODO 3: Check HPACK table health (size, eviction rate, hit ratios)

    // TODO 4: Calculate error rates and categorize error types

    // TODO 5: Identify potential performance bottlenecks or resource exhaustion

    return nil
}
```

```
}

// DetectAnomalies identifies unusual patterns that may indicate bugs

func (cm *ConnectionMonitor) DetectAnomalies() []Anomaly {

    // TODO 1: Check for streams stuck in transitional states too long

    // TODO 2: Identify flow control deadlock conditions

    // TODO 3: Detect rapid stream creation/destruction cycles

    // TODO 4: Find streams with disproportionate resource usage

    // TODO 5: Check for HPACK table thrashing or corruption indicators

    return nil

}
```

Protocol Fuzzing Framework (Complete Implementation):

```
package fuzzing                                     GO

import (
    "bytes"
    "crypto/rand"
    "math"
    "testing"
)

// FrameFuzzer generates malformed HTTP/2 frames for robustness testing

type FrameFuzzer struct {
    maxFrameSize uint32
    streamIDs    []uint32
    frameTypes   []uint8
}

func NewFrameFuzzer(maxFrameSize uint32) *FrameFuzzer {
    return &FrameFuzzer{
        maxFrameSize: maxFrameSize,
        streamIDs:   []uint32{0, 1, 3, 5, 0x7FFFFFFF}, // Various stream ID patterns
        frameTypes:   []uint8{0, 1, 4, 6, 8, 255},      // Valid and invalid frame types
    }
}

// GenerateMalformedFrame creates systematically corrupted frames

func (ff *FrameFuzzer) GenerateMalformedFrame(corruption CorruptionType) *Frame {
    frame := &Frame{}
    switch corruption {

```

```
case CorruptionInvalidLength:

    frame.Length = math.MaxUint32 // Exceeds 24-bit limit

    frame.Type = FrameTypeDATA

    frame.StreamID = 1

    frame.Payload = make([]byte, 100)


case CorruptionReservedBits:

    frame.Length = 8

    frame.Type = FrameTypeDATA

    frame.StreamID = 0x80000001 // Reserved bit set

    frame.Payload = make([]byte, 8)


case CorruptionInvalidType:

    frame.Length = 0

    frame.Type = 255 // Invalid frame type

    frame.StreamID = 1


case CorruptionRandomPayload:

    frame.Length = uint32(len(ff.generateRandomBytes(1024)))

    frame.Type = FrameTypeHEADERS

    frame.StreamID = 3

    frame.Payload = ff.generateRandomBytes(1024)

}

return frame
}
```

```
func (ff *FrameFuzzer) generateRandomBytes(length int) []byte {
    data := make([]byte, length)
    rand.Read(data)
    return data
}

type CorruptionType int

const (
    CorruptionInvalidLength CorruptionType = iota
    CorruptionReservedBits
    CorruptionInvalidType
    CorruptionRandomPayload
)
```

Language-Specific Debugging Hints:

Go-Specific Debugging Tips:

- Use `go build -race` to detect race conditions in stream management
- Enable `GODEBUG=http2debug=2` for detailed HTTP/2 library logging
- Use `go tool trace` to analyze goroutine interactions and scheduling
- Implement `fmt.Stringer` interfaces on protocol structures for better debugging output
- Use `sync.RWMutex` for read-heavy operations like stream state lookups

Milestone Checkpoints:

Milestone 1 Checkpoint - Frame Parsing Debugging:

```
# Test frame parsing with malformed input

go test -v ./internal/debugging -run TestFrameValidation

# Expected: Proper error handling for invalid frames, no panics

# Run with race detector

go test -race ./internal/framing

# Expected: No race condition warnings

# Enable protocol logging

GODEBUG=http2debug=2 go run ./cmd/server

# Expected: Detailed frame-level logging output
```

BASH

Milestone 2 Checkpoint - HPACK Debugging:

```
# Test HPACK table consistency

go test -v ./internal/debugging -run TestHPACKTableValidation

# Expected: Table size validation passes, no index errors

# Fuzz HPACK decoder

go test -fuzz=FuzzHPACKDecoding ./internal/hpack

# Expected: No crashes or infinite loops on malformed input
```

BASH

Common Debugging Scenarios:

Symptom	Likely Root Cause	Diagnostic Steps	Resolution
Connection hangs	Flow control deadlock	Check window levels, backpressure queues	Send WINDOW_UPDATE frames
Random header values	HPACK table corruption	Validate table state, check index calculations	Reset dynamic table
Protocol errors	Frame validation failures	Enable frame-level logging, check binary format	Fix frame serialization
Stream state violations	Race conditions	Run with <code>-race</code> , add synchronization	Add proper locking
Memory leaks	Stream cleanup failures	Monitor goroutine/memory usage	Fix stream lifecycle management

Future Extensions

Milestone(s): This section addresses potential enhancements beyond the core milestones, providing a roadmap for extending the HTTP/2 server with server push capabilities, advanced features, and performance optimizations that build upon the foundations established in milestones 1-4.

Think of this HTTP/2 server implementation as building a solid house foundation - the four milestones establish the structural integrity with binary framing, header compression, stream management, and flow control. Future extensions are like adding modern amenities: server push is like installing a smart anticipatory system that delivers what you need before you ask, advanced features are like home automation that optimizes the experience, and performance optimizations are like upgrading to energy-efficient systems that handle more load with fewer resources. Each extension builds upon the proven foundation while adding capabilities that enhance the overall system's utility and efficiency.

The architecture established in the core milestones provides extension points specifically designed for these enhancements. The `StreamManager` can be extended to handle server-initiated streams, the `FrameRouter` can incorporate new frame types, and the flow control mechanisms can be enhanced with more sophisticated scheduling algorithms. Rather than retrofitting these features, the modular design allows for organic growth that maintains the system's integrity while expanding its capabilities.

Server Push: PUSH_PROMISE Frame Handling and Resource Push Strategies

Server push represents HTTP/2's most revolutionary feature - the ability for servers to proactively send resources before clients request them. Think of traditional HTTP as a restaurant where you must order each course individually, waiting for delivery before requesting the next item. Server push transforms this into a

tasting menu experience where the chef anticipates your needs and delivers complementary items that enhance the main course, all arriving in perfectly timed coordination.

The mental model for server push centers around predictive resource delivery. When a client requests an HTML page, the server can analyze the page content and identify critical resources - CSS stylesheets, JavaScript files, images, or fonts - that the browser will inevitably request. Rather than waiting for these subsequent requests, the server initiates push streams to deliver these resources immediately, reducing overall page load time by eliminating round-trip latency.

Decision: PUSH_PROMISE Frame Implementation Strategy

- **Context:** Server push requires introducing new frame types and stream management patterns not covered in the core milestones, while maintaining compatibility with existing stream lifecycle management
- **Options Considered:**
 1. Extend existing `StreamManager` with push-specific methods
 2. Create separate `PushManager` component with dedicated stream handling
 3. Implement push as application-layer feature above HTTP/2 protocol layer
- **Decision:** Extend `StreamManager` with push capabilities while adding `PushManager` for policy decisions
- **Rationale:** Push streams follow the same lifecycle as client streams but with server initiation, making `StreamManager` extension natural while `PushManager` handles the application logic of what to push
- **Consequences:** Maintains consistency with existing stream management while cleanly separating protocol mechanics from push policy decisions

PUSH_PROMISE Frame Structure and Processing

The `PUSH_PROMISE` frame serves as the server's contract with the client, declaring intent to push a specific resource while providing the request headers that would have been sent if the client had initiated the request. This frame type extends the binary framing engine established in Milestone 1 with server-specific semantics.

Field	Type	Description
Length	uint32	Frame payload length excluding 9-byte header
Type	uint8	FrameTypePUSH_PROMISE (0x5) for push promise frames
Flags	uint8	FlagEndHeaders (0x4) indicates complete headers or continuation needed
StreamID	uint32	Parent stream ID that triggered this push promise
PromisedStreamID	uint32	New server-initiated stream ID for the pushed resource
HeaderBlock	[]byte	HPACK-compressed headers representing the synthetic request

The frame processing workflow integrates with the existing frame routing infrastructure while introducing server-initiated stream creation. When the server decides to push a resource, it constructs a PUSH_PROMISE frame containing the synthetic request headers that represent the client request that would have fetched this resource. The promised stream ID must follow HTTP/2's server stream allocation rules - even numbers starting from 2 and increasing monotonically.

PUSH_PROMISE Frame Processing Algorithm:

1. Server application identifies resource candidate for pushing
2. Validate that parent stream is in open or half-closed-remote state
3. Allocate next available server stream ID (even number)
4. Construct synthetic request headers for the resource
5. Compress headers using connection's HPACK context
6. Create PUSH_PROMISE frame with parent stream ID and promised stream ID
7. Send PUSH_PROMISE frame to client before sending any DATA frames
8. Create new stream in reserved-local state for the promised stream
9. Begin sending HEADERS and DATA frames on the promised stream
10. Handle potential RST_STREAM from client if push is rejected

The integration with existing flow control mechanisms requires careful consideration. Pushed streams consume connection-level and stream-level windows just like client-initiated streams. However, the server must be more conservative with push resources since the client may not actually need them, potentially wasting bandwidth and window space.

Push Policy and Resource Selection

The most critical aspect of effective server push lies not in the protocol mechanics but in the intelligent selection of resources to push. Think of this as a restaurant server learning customer preferences - push too aggressively and you overwhelm the client with unwanted resources, push too conservatively and you miss opportunities to improve performance. The optimal strategy balances proactive resource delivery with respectful bandwidth usage.

Push Strategy	Trigger	Candidate Resources	Pros	Cons
Static Analysis	HTML response generation	Referenced CSS, JS, critical images	Simple implementation, predictable behavior	Doesn't account for client cache state
Dependency Graph	Resource request patterns	Resources with high co-occurrence probability	Data-driven decisions, adapts over time	Requires learning phase, complex state management
Link Header Hints	Application-provided hints	Developer-specified critical resources	Developer control, explicit intent	Requires application modification, manual maintenance
Cache Integration	Client cache headers	Resources known to be stale or missing	Avoids redundant pushes, efficient bandwidth usage	Complex cache state tracking, privacy implications

The `PushManager` component encapsulates push policy decisions while integrating with the existing stream and connection management infrastructure:

Method Name	Parameters	Returns	Description
<code>EvaluatePushCandidates</code>	<code>parentStreamID uint32, responseHeaders http.Header</code>	<code>[]PushCandidate, error</code>	Analyzes response content and identifies resources suitable for pushing
<code>ValidatePushRequest</code>	<code>candidate PushCandidate, connectionState *ConnectionState</code>	<code>bool, error</code>	Checks push eligibility considering client settings, window availability, and stream limits
<code>CreatePushPromise</code>	<code>parentStreamID uint32, resource PushCandidate</code>	<code>*Frame, uint32, error</code>	Constructs PUSH_PROMISE frame and allocates promised stream ID
<code>HandlePushRejection</code>	<code>promisedStreamID uint32, errorCode uint32</code>	<code>error</code>	Processes RST_STREAM cancellation of promised stream and updates push policy

The push candidate evaluation process requires analyzing response content to identify linked resources. For HTML responses, this involves parsing the document structure to find stylesheet links, script tags, image references, and font declarations. The challenge lies in distinguishing critical resources that improve page load time from optional resources that might waste bandwidth.

⚠ Pitfall: Overly Aggressive Push Policies

A common mistake is pushing every resource referenced in an HTML page, overwhelming clients with unwanted data. This wastes bandwidth and can actually harm performance by consuming flow control windows needed for client-requested resources. Instead, focus on critical resources that block page rendering - typically CSS stylesheets and JavaScript files in the document head - while avoiding speculative pushes of optional content like tracking pixels or below-the-fold images.

Push Stream Lifecycle Management

Server push introduces a unique stream lifecycle where the server creates streams proactively rather than reactively. Think of this as the difference between a traditional store where customers browse and request items versus a subscription service where the provider anticipates needs and delivers items before explicit requests. The server must manage this anticipatory relationship while respecting client autonomy to reject unwanted pushes.

The pushed stream lifecycle follows a modified version of the standard HTTP/2 stream state machine:

Current State	Event	Next State	Actions Taken
Non-existent	PUSH_PROMISE sent	StateReservedLocal	Allocate stream ID, reserve flow control windows
StateReservedLocal	HEADERS frame sent	StateHalfClosedRemote	Begin response transmission, no client data expected
StateHalfClosedRemote	DATA frame sent	StateHalfClosedRemote	Continue response transmission
StateHalfClosedRemote	DATA frame with END_STREAM	StateClosed	Complete response, release resources
StateReservedLocal	RST_STREAM received	StateClosed	Cancel push, release resources, update policy
Any	Connection error	StateClosed	Cleanup all pushed streams

The key difference from client-initiated streams is that pushed streams never receive request data from the client - they transition directly from reserved to half-closed-remote state. This simplification reduces the complexity of state management while requiring careful handling of the reservation phase where the client might reject the push.

Push stream integration with the existing `StreamManager` requires extending the stream allocation logic to handle server-initiated streams:

Method Extension	Parameters	Returns	Description
<code>CreatePushedStream</code>	<code>parentStreamID uint32,</code> <code>promisedStreamID uint32</code>	<code>*Stream,</code> <code>error</code>	Creates new stream in reserved-local state linked to parent
<code>ValidatePushCapacity</code>	<code>requestedPushes int</code>	<code>bool,</code> <code>error</code>	Checks if connection can handle additional pushed streams within limits
<code>HandlePushRejection</code>	<code>promisedStreamID</code> <code>uint32</code>	<code>error</code>	Transitions rejected push stream to closed state and updates counters
<code>GetPushedStreams</code>	<code>parentStreamID uint32</code>	<code>[]uint32,</code> <code>error</code>	Returns list of pushed stream IDs associated with parent stream

Client Push Settings and Negotiation

HTTP/2 provides clients with granular control over server push behavior through the `SETTINGS` frame mechanism established in the core milestones. Think of this as a preferences dialog where clients can express their push tolerance - some clients on limited bandwidth connections might disable push entirely, while others might allow moderate pushing for critical resources only.

The client communicates push preferences through specific settings parameters:

Setting Parameter	Value Range	Default	Description
<code>SETTINGS_ENABLE_PUSH</code>	0 or 1	1	Global enable/disable flag for all server push
<code>SETTINGS_MAX_CONCURRENT_STREAMS</code>	0 to $2^{31}-1$	No limit	Includes both regular and pushed streams in the limit
<code>SETTINGS_INITIAL_WINDOW_SIZE</code>	1 to $2^{31}-1$	65535	Applies to pushed streams, affecting push aggressiveness

The server must respect these settings when making push decisions. When `SETTINGS_ENABLE_PUSH` is set to 0, the server must not send any `PUSH_PROMISE` frames. When concurrent stream limits are reached, the server should prioritize client-initiated streams over new pushes. The initial window size affects how much data can be pushed immediately, influencing the cost-benefit calculation for push candidates.

Push settings integration requires extending the connection's settings management:

Push Settings Validation Algorithm:

1. Receive SETTINGS frame from client during connection establishment
2. Extract `SETTINGS_ENABLE_PUSH` value and store in connection state
3. If push is disabled, mark connection as push-prohibited
4. For enabled push, extract concurrent stream and window size limits
5. Use these limits in push candidate evaluation decisions
6. Send SETTINGS acknowledgment to confirm receipt
7. Apply settings to all future push decisions on this connection
8. Handle settings updates during connection lifetime

Key Design Insight: Server push is fundamentally about respecting client autonomy while providing performance benefits. The server proposes resources it believes will be helpful, but the client retains ultimate authority to accept or reject these proposals. This cooperative model ensures that push enhances rather than harms the client experience.

Advanced Features: Connection Coalescing, Alternative Services, and HTTP/2 Extensions

Beyond the core HTTP/2 protocol features, several advanced capabilities can significantly enhance server performance and client experience. Think of these as professional-grade features that distinguish a production-ready server from a basic implementation - connection coalescing is like having multiple storefronts share a single efficient warehouse, alternative services provide flexible routing options like having multiple delivery methods, and protocol extensions enable future-proofing like modular upgrades to a building's infrastructure.

These advanced features build upon the solid foundation of the core milestones while introducing sophisticated optimizations that require careful design to maintain the system's reliability and performance characteristics. Each feature addresses specific limitations or inefficiencies in basic HTTP/2 operations while maintaining backward compatibility and protocol compliance.

Connection Coalescing and Certificate Management

Connection coalescing allows a single HTTP/2 connection to serve multiple hostnames that share certificate authority, reducing connection overhead and improving resource utilization. Think of this as a shared transportation system where multiple destinations can be reached via the same efficient route, rather than establishing separate pathways for each destination.

The mental model centers around certificate-based trust relationships. When a server presents a certificate valid for multiple hostnames, or when multiple hostnames resolve to the same IP address with compatible certificates, clients can reuse existing connections rather than establishing new ones. This reduces TLS handshake overhead, connection state memory usage, and overall latency.

Coalescing Strategy	Certificate Requirement	Implementation Complexity	Performance Benefit
Same Certificate	Single cert covers multiple hostnames via SAN	Low - simple hostname validation	High - eliminates redundant connections
Wildcard Certificate	Wildcard cert matches subdomain pattern	Medium - pattern matching logic	Medium - reduces connections for subdomain services
Authority Validation	Different certs with same CA and compatible policies	High - certificate chain analysis	Low - limited applicability

The server implementation requires extending connection management to track hostname associations and certificate coverage:

Component Extension	Responsibility	Key Methods
CertificateManager	Validates hostname coverage and certificate compatibility	<code>ValidateHostname</code> , <code>GetCoalescingCandidates</code>
ConnectionPool	Manages shared connections across hostnames	<code>FindCoalescableConnection</code> , <code>RegisterHostname</code>
HostRouter	Routes requests to appropriate handlers based on hostname	<code>ResolveHandler</code> , <code>ValidateHostnameRequest</code>

Connection coalescing requires careful validation to prevent security violations. The server must ensure that requests for different hostnames are truly authorized to share the same connection based on certificate coverage and organizational policies. This involves validating Subject Alternative Names (SAN) in certificates and checking that hostname resolution points to the same server infrastructure.

Connection Coalescing Algorithm:

1. Client establishes connection to hostname A with certificate validation
2. Client needs connection to hostname B on the same IP address
3. Server validates that existing certificate covers hostname B via SAN or wildcard
4. If valid, server accepts requests for hostname B on existing connection A
5. Server routes requests based on :authority header to appropriate handlers
6. Server maintains mapping of hostname to connection for future requests
7. If certificate validation fails, client establishes separate connection
8. Connection cleanup considers all associated hostnames

Alternative Services and Protocol Negotiation

Alternative Services (Alt-Svc) enable servers to advertise different protocol endpoints for the same service, allowing clients to upgrade to more efficient protocols or use different network paths. Think of this as a service directory that tells clients about faster or more reliable ways to reach the same destination - like suggesting a high-speed rail option when someone asks for directions to a city accessible by regular roads.

The Alt-Svc mechanism operates through HTTP response headers that advertise protocol alternatives:

Header Format	Example	Description
<pre>Alt-Svc: protocol="hostname:port"; ma=seconds</pre>	<pre>Alt-Svc: h2="api.example.com:443"; ma=3600</pre>	Advertises HTTP/2 on specific endpoint with max-age
<pre>Alt-Svc: protocol=":port"; ma=seconds</pre>	<pre>Alt-Svc: h2=":8443"; ma=7200</pre>	Advertises protocol on same hostname, different port
<pre>Alt-Svc: clear</pre>	<pre>Alt-Svc: clear</pre>	Clears all previously advertised alternatives

Alternative services integration requires extending the server's response generation to include Alt-Svc advertisements based on available protocol endpoints:

Method Name	Parameters	Returns	Description
<code>AdvertiseAlternatives</code>	<code>request *http.Request, config *AltSvcConfig</code>	<code>string, error</code>	Generates Alt-Svc header value based on available alternatives
<code>ValidateAlternative</code>	<code>protocol string, endpoint string</code>	<code>bool, error</code>	Checks if advertised alternative is actually available
<code>HandleAltSvcRequest</code>	<code>altEndpoint string, request *http.Request</code>	<code>*http.Response, error</code>	Processes request via alternative service endpoint
<code>UpdateAltSvccache</code>	<code>hostname string, alternatives []Alternative</code>	<code>error</code>	Updates client cache with new alternative service information

Decision: Alt-Svc Implementation Scope

- **Context:** Alternative Services can advertise various protocols including HTTP/3, WebSocket, and custom protocols, but our implementation focuses on HTTP/2 improvements
- **Options Considered:**
 1. Full Alt-Svc implementation supporting all protocols
 2. HTTP/2-specific Alt-Svc for port and hostname alternatives
 3. Static configuration-based alternative advertisement
- **Decision:** HTTP/2-specific Alt-Svc with configuration-based alternatives
- **Rationale:** Stays within project scope while providing practical benefits for load balancing and protocol optimization
- **Consequences:** Enables performance improvements through better endpoint selection while avoiding complexity of multi-protocol support

HTTP/2 Protocol Extensions

The HTTP/2 specification defines extension mechanisms that allow for future protocol enhancements while maintaining backward compatibility. Think of these as expansion slots in computer hardware - they provide standardized ways to add new capabilities without redesigning the entire system. Common extensions include new frame types, additional settings parameters, and enhanced flow control mechanisms.

Protocol extension support requires designing the frame processing pipeline to handle unknown frame types gracefully while providing hooks for extension registration:

Extension Category	Examples	Implementation Impact
New Frame Types	<code>ORIGIN</code> , <code>CACHE_DIGEST</code> , experimental frames	Requires extensible frame router with registration mechanism
Settings Extensions	<code>SETTINGS_ENABLE_CONNECT_PROTOCOL</code>	Extends settings validation and negotiation logic
Header Extensions	New pseudo-headers, custom compression dictionaries	Requires HPACK extension points and validation updates
Flow Control Extensions	Priority-based windows, quality-of-service parameters	Extends window management with new scheduling algorithms

The extensible frame processing architecture builds upon the frame router established in the core milestones:

Component	Extension Point	Description
FrameRegistry	Frame type registration	Allows registration of custom frame handlers
ExtensibleRouter	Plugin-based routing	Routes unknown frame types to registered extensions
ExtensionManager	Extension lifecycle	Manages loading, initialization, and cleanup of extensions

Extension Registration Algorithm:

1. Extension module registers frame type handlers during server initialization
2. Frame router builds dispatch table including both standard and extension handlers
3. During frame processing, router checks for registered handler for frame type
4. If handler exists, delegates frame processing to extension
5. Extension processes frame according to its specific semantics
6. Extension returns processing result to router for further dispatch
7. Unknown frame types without handlers are ignored per HTTP/2 specification
8. Connection errors in extensions are handled gracefully without affecting core protocol

Performance Optimizations: Connection Pooling, Memory Management, and High-Performance I/O Patterns

Production HTTP/2 servers require sophisticated optimizations to handle thousands of concurrent connections while maintaining low latency and high throughput. Think of these optimizations as the difference between a hobbyist workshop and a professional manufacturing facility - the basic functionality might be similar, but the operational efficiency, resource utilization, and scalability characteristics are fundamentally different.

These optimizations build upon the architectural foundation established in the core milestones, enhancing rather than replacing the basic functionality. The key insight is that performance improvements must maintain correctness - a fast but incorrect implementation provides no value in production environments.

Connection Pooling and Resource Management

Connection pooling in HTTP/2 differs fundamentally from HTTP/1.1 pooling due to the persistent, multiplexed nature of HTTP/2 connections. Think of HTTP/1.1 connection pooling as managing a fleet of single-passenger vehicles, while HTTP/2 connection pooling is like managing a smaller fleet of high-capacity buses. Each connection can handle many simultaneous streams, so the optimization focus shifts from connection quantity to connection quality and longevity.

The mental model centers around connection lifecycle management and resource efficiency. Rather than creating and destroying connections frequently, the server maintains long-lived connections that efficiently multiplex many concurrent requests. The challenge lies in balancing connection reuse with resource cleanup, handling connection failures gracefully, and managing memory usage across long-lived connections.

Pooling strategy	connection lifetime	resource usage	complexity
Fixed Pool Size	Predetermined connection limit	Predictable memory footprint	Low - simple connection allocation
Dynamic Scaling	Connections created/destroyed based on load	Variable memory usage	Medium - requires load monitoring
Adaptive Pooling	ML-based prediction of optimal pool size	Optimized for current traffic patterns	High - requires traffic analysis
Hybrid Approach	Fixed baseline with dynamic expansion	Balanced resource usage and performance	Medium - configurable thresholds

The connection pool implementation extends the basic connection management with sophisticated lifecycle and resource tracking:

Component	Responsibility	Key Methods
ConnectionPool	Manages pool of active connections with lifecycle tracking	AcquireConnection , ReleaseConnection , HealthCheck
ConnectionFactory	Creates new connections with proper configuration	CreateConnection , ConfigureConnection , ValidateConnection
ResourceMonitor	Tracks resource usage and triggers cleanup	MonitorUsage , TriggerCleanup , ReportMetrics
LoadBalancer	Distributes new streams across available connections	SelectConnection , UpdateWeights , HandleFailure

Connection pooling requires sophisticated health monitoring to detect and handle connection failures:

Connection Health Monitoring Algorithm:

1. Monitor connection for signs of degradation: high error rates, slow responses, timeout errors
2. Implement periodic ping/pong frames to validate connection liveness
3. Track per-connection metrics: active streams, error count, response times
4. When degradation detected, mark connection for drainage
5. Stop assigning new streams to degraded connection
6. Allow existing streams to complete naturally
7. Once all streams complete, close connection gracefully
8. Create replacement connection if pool size drops below minimum
9. Update load balancing weights based on connection performance
10. Log connection lifecycle events for operational monitoring

Pitfall: Connection Pool Memory Leaks

Long-lived HTTP/2 connections can accumulate memory leaks through HPACK dynamic tables, stream metadata, and buffered data. Implement periodic connection recycling where connections are gracefully closed and replaced after handling a certain number of requests or after a time threshold. This prevents memory usage from growing unboundedly while maintaining the performance benefits of connection reuse.

Memory Management and Garbage Collection Optimization

HTTP/2's binary framing and multiplexing create unique memory management challenges compared to HTTP/1.1. Think of HTTP/1.1 memory usage as individual meals - each request consumes memory temporarily and cleans up completely when finished. HTTP/2 is more like a buffet restaurant where multiple customers share common resources, requiring careful management of shared state and resource cleanup.

The primary memory consumers in HTTP/2 implementations include HPACK dynamic tables, stream metadata, frame buffers, and flow control windows. Each component requires different optimization strategies based on its usage patterns and lifecycle characteristics.

Memory Component	Growth Pattern	Optimization Strategy
HPACK Dynamic Tables	Grows with unique headers, bounded by table size	Implement aggressive eviction, header value deduplication
Stream Metadata	Linear growth with concurrent streams	Use object pooling, lazy initialization of optional fields
Frame Buffers	Temporary growth during frame processing	Buffer pooling, size-based buffer selection
Flow Control State	Fixed size per stream, linear with stream count	Pack bit fields, use efficient data structures

Memory optimization requires implementing object pooling and buffer reuse strategies that reduce garbage collection pressure:

Pool Type	Object Lifecycle	Reuse Strategy
FramePool	Frame objects for parsing/serialization	Size-based pools for different frame types
BufferPool	Byte slices for frame payloads	Power-of-two sized buffers with size classification
StreamPool	Stream metadata objects	Pool unused stream objects after cleanup
HeaderPool	Header entry objects for HPACK	Reuse header entry objects in dynamic table

The buffer management system requires careful design to balance memory efficiency with allocation performance:

Method Name	Parameters	Returns	Description
AcquireBuffer	size int	[]byte, error	Gets appropriately sized buffer from pool
ReleaseBuffer	buffer []byte	error	Returns buffer to pool for reuse
OptimizePool	usage PoolUsage	error	Adjusts pool sizes based on usage patterns
ReportMemoryStats	none	MemoryStats, error	Provides detailed memory usage breakdown

Buffer Pool Optimization Algorithm:

1. Classify buffer requests into size categories (small: <1KB, medium: 1-16KB, large: >16KB)
2. Maintain separate pools for each size category with different retention policies
3. For small buffers, maintain larger pool sizes due to high turnover
4. For large buffers, maintain smaller pools and implement aggressive cleanup
5. Monitor pool hit rates and adjust pool sizes dynamically
6. Implement buffer zeroing for security-sensitive data
7. Use sync.Pool for garbage collector integration where appropriate
8. Implement buffer leak detection in debug builds

High-Performance I/O Patterns and Zero-Copy Operations

HTTP/2's binary framing enables sophisticated I/O optimizations that minimize data copying and system call overhead. Think of traditional I/O as manually carrying individual packages between buildings, while optimized I/O is like using conveyor belts and automated systems that move data efficiently with minimal human intervention.

The key optimization opportunities include zero-copy frame transmission, vectored I/O for multiple frames, and direct memory mapping for large payloads. These techniques require careful integration with the existing frame processing pipeline while maintaining correctness and security properties.

I/O Pattern	Optimization Technique	Performance Benefit	Implementation Complexity
Frame Writing	Vectored I/O with writev() syscall	Reduces syscall overhead	Medium - requires frame batching
Large Payloads	sendfile() or splice() for zero-copy	Eliminates user-space copying	High - requires file descriptor management
Frame Reading	Read buffer management and parsing optimization	Reduces memory allocation	Medium - requires careful buffer handling
TLS Integration	TLS record alignment with HTTP/2 frames	Optimizes encryption overhead	High - requires TLS library integration

High-performance I/O requires extending the connection layer with optimized read/write operations:

Method Extension	Parameters	Returns	Description
WriteFrameVector	frames []*Frame	int, error	Writes multiple frames in single syscall
ReadFrameBuffer	buffer []byte, offset int	*Frame, int, error	Parses frame from existing buffer
SendFile	file *os.File, offset, size int64	int64, error	Zero-copy file transmission for DATA frames
EnableWriteBuffering	size int	error	Configures write buffering for small frames

Zero-Copy Frame Transmission Algorithm:

1. Identify DATA frames with file-backed payloads larger than threshold
2. Validate that file descriptor is suitable for sendfile() operation
3. Construct HTTP/2 frame header and write to connection
4. Use sendfile() syscall to transmit file content directly from kernel
5. Update flow control windows without copying payload data
6. Handle partial transmissions and retry logic for incomplete sends
7. Fall back to regular I/O for small payloads or non-file data
8. Maintain frame boundary alignment for HTTP/2 protocol compliance

The vectored I/O implementation batches multiple small frames into single system calls, reducing the overhead of frequent syscall invocations:

Batching Strategy	Frame Selection	Latency Impact	Throughput Benefit
Time-based batching	All frames ready within timeout window	Adds controlled latency	High - amortizes syscall cost
Size-based batching	Frames until total size reaches threshold	Minimal latency impact	Medium - balances latency and throughput
Priority-aware batching	High priority frames bypass batching	Preserves priority semantics	Medium - some frames skip optimization

Key Performance Insight: The most significant performance improvements in HTTP/2 come from reducing per-frame overhead rather than optimizing individual operations. Focus on batching operations, reusing objects, and minimizing memory allocations rather than micro-optimizing individual frame processing steps. The multiplexed nature of HTTP/2 means that small per-frame improvements compound dramatically across hundreds of concurrent streams.

Implementation Guidance

This implementation guidance provides practical directions for extending the HTTP/2 server with the advanced features described above. The focus is on providing concrete starting points that integrate cleanly with the existing codebase while enabling incremental development of these sophisticated features.

Technology Recommendations

Feature Category	Basic Implementation	Advanced Implementation
Server Push	Static resource lists + <code>PUSH_PROMISE</code> frames	Dynamic content analysis + machine learning
Connection Coalescing	SAN certificate validation	Advanced certificate chain analysis + wildcard support
Alternative Services	Static Alt-Svc headers	Dynamic load balancing + health-based routing
Memory Management	Standard Go garbage collector + object pooling	Custom allocators + memory-mapped I/O
I/O Optimization	Buffered I/O + basic batching	Zero-copy operations + kernel bypass techniques

Recommended File Structure

Infrastructure Code for Server Push

```
package push GO

import (
    "context"
    "sync"
    "time"
)

// PushCandidate represents a resource that could be pushed to the client

type PushCandidate struct {

    URL          string
    Method       string
    Headers      http.Header
    Priority     uint8
    EstimatedSize int64
    CacheControl string
}

// PushPolicy defines the strategy for selecting resources to push

type PushPolicy struct {

    MaxConcurrentPushes int
    MaxResourceSize     int64
    AllowedContentTypes []string
    CacheAwareStrategy  bool
}

// PushManager handles server push decisions and stream management

type PushManager struct {
```

```

policy          *PushPolicy

streamManager   *StreamManager

connectionState *Connection

activePromises map[uint32]*PushedStream // promised stream ID -> stream

mu             sync.RWMutex

}

// PushedStream tracks the lifecycle of a server-pushed stream

type PushedStream struct {

    StreamID      uint32

    ParentStreamID uint32

    Resource      PushCandidate

    State         StreamState

    CreatedAt     time.Time

    PromiseSent   bool

    ResponseSent  bool

}

// NewPushManager creates a push manager with the specified policy

func NewPushManager(policy *PushPolicy, streamManager *StreamManager, conn *Connection)
*PushManager {
    return &PushManager{
        policy:          policy,
        streamManager:   streamManager,
        connectionState: conn,
        activePromises: make(map[uint32]*PushedStream),
    }
}

```

```
// EvaluatePushCandidates analyzes response content and identifies push candidates

func (pm *PushManager) EvaluatePushCandidates(parentStreamID uint32, responseHeaders
http.Header, body []byte) ([]PushCandidate, error) {

    // TODO 1: Check if push is enabled on this connection via SETTINGS_ENABLE_PUSH

    // TODO 2: Parse response content type and determine if it contains linkable resources

    // TODO 3: For HTML content, parse document and extract CSS, JS, and image references

    // TODO 4: Filter candidates based on policy rules (size limits, content types)

    // TODO 5: Check current push capacity against MaxConcurrentPushes limit

    // TODO 6: Return prioritized list of push candidates

    return nil, nil
}

// CreatePushPromise constructs a PUSH_PROMISE frame for the specified resource

func (pm *PushManager) CreatePushPromise(parentStreamID uint32, candidate PushCandidate)
(*Frame, uint32, error) {

    // TODO 1: Allocate next available server stream ID (even number)

    // TODO 2: Create synthetic request headers for the resource

    // TODO 3: Compress headers using connection's HPACK encoder

    // TODO 4: Construct PUSH_PROMISE frame with parent stream ID and promised stream ID

    // TODO 5: Create PushedStream object and add to activePromises map

    // TODO 6: Return frame ready for transmission

    return nil, 0, nil
}
```

Core Logic for Connection Coalescing

```
package coalescing GO

import (
    "crypto/x509"

    "net"

    "sync"
)

// CertificateManager handles hostname validation and coalescing decisions

type CertificateManager struct {

    certificates map[string]*x509.Certificate // hostname -> certificate

    coalescable map[string][]string           // primary hostname -> coalesced hostnames

    mu          sync.RWMutex
}

// CoalescingPool manages shared connections across multiple hostnames

type CoalescingPool struct {

    connections map[string]*Connection // primary hostname -> connection

    hostMapping map[string]string      // any hostname -> primary hostname

    certManager *CertificateManager

    mu          sync.RWMutex
}

// ValidateHostnameCoalescing checks if two hostnames can share a connection

func (cm *CertificateManager) ValidateHostnameCoalescing(primary, secondary string) (bool, error) {

    // TODO 1: Retrieve certificate for primary hostname

    // TODO 2: Check if certificate covers secondary hostname via SAN entries
}
```

```
// TODO 3: Validate that both hostnames resolve to same IP address

// TODO 4: Ensure certificate is still valid and properly signed

// TODO 5: Return true if coalescing is safe, false otherwise

return false, nil

}

// FindCoalescableConnection attempts to find existing connection for hostname

func (cp *CoalescingPool) FindCoalescableConnection(hostname string) (*Connection, bool) {

    // TODO 1: Check if hostname already has direct connection mapping

    // TODO 2: If not, validate coalescing with existing connections

    // TODO 3: Use CertificateManager to validate hostname compatibility

    // TODO 4: Return existing connection if coalescing is valid

    // TODO 5: Return nil if new connection required

    return nil, false

}
```

Memory Optimization Infrastructure

```
package optimization GO

import (
    "sync"
    "unsafe"
)

// BufferPool provides size-categorized buffer reuse

type BufferPool struct {

    small  sync.Pool // buffers <= 1KB
    medium sync.Pool // buffers 1KB-16KB
    large   sync.Pool // buffers > 16KB
    stats   PoolStats
    mu      sync.Mutex
}

// PoolStats tracks buffer pool performance metrics

type PoolStats struct {

    SmallHits     uint64
    SmallMisses  uint64
    MediumHits   uint64
    MediumMisses uint64
    LargeHits    uint64
    LargeMisses  uint64
}

// AcquireBuffer gets appropriately sized buffer from pool

func (bp *BufferPool) AcquireBuffer(size int) []byte {
```

```

// TODO 1: Determine appropriate pool based on requested size

// TODO 2: Try to get buffer from appropriate pool

// TODO 3: If pool is empty, allocate new buffer with proper capacity

// TODO 4: Update hit/miss statistics for monitoring

// TODO 5: Zero buffer contents if required for security

// TODO 6: Return buffer ready for use

return make([]byte, size)

}

// ReleaseBuffer returns buffer to appropriate pool for reuse

func (bp *BufferPool) ReleaseBuffer(buffer []byte) {

    // TODO 1: Determine which pool this buffer belongs to based on capacity

    // TODO 2: Validate buffer is suitable for reuse (not too large, not corrupted)

    // TODO 3: Reset buffer length to full capacity

    // TODO 4: Return buffer to appropriate pool

    // TODO 5: Update pool statistics

}

// ObjectPool provides reusable objects with proper initialization

type ObjectPool[T any] struct {

    pool sync.Pool

    new func() T

    reset func(T)

}

// NewObjectPool creates a pool for specific object type

func NewObjectPool[T any](newFunc func() T, resetFunc func(T)) *ObjectPool[T] {

    return &ObjectPool[T]{

```

```
pool: sync.Pool{  
  
    New: func() interface{} { return newFunc() },  
  
    },  
  
    new: newFunc,  
  
    reset: resetFunc,  
  
}  
  
}
```

High-Performance I/O Implementation

```
package optimization GO

import (
    "net"
    "syscall"
    "unsafe"
)

// VectoredWriter batches multiple frames for efficient transmission

type VectoredWriter struct {

    conn      net.Conn

    buffers   [][]byte

    pending   []*Frame

    maxBatch int

    timeout   time.Duration
}

// WriteFrameVector writes multiple frames in single syscall

func (vw *VectoredWriter) WriteFrameVector(frames []*Frame) (int, error) {

    // TODO 1: Serialize each frame into separate buffers

    // TODO 2: Prepare iovec structure for writev syscall

    // TODO 3: Call writev to transmit all frames together

    // TODO 4: Handle partial writes and retry logic

    // TODO 5: Update flow control windows for transmitted frames

    // TODO 6: Return total bytes written across all frames

    return 0, nil
}
```

```

// ZeroCopyTransmitter handles sendfile operations for large payloads

type ZeroCopyTransmitter struct {

    conn net.Conn

    file *os.File

}

// SendFileData transmits file content without user-space copying

func (zct *ZeroCopyTransmitter) SendFileData(streamID uint32, file *os.File, offset, size int64) error {

    // TODO 1: Write HTTP/2 DATA frame header to connection

    // TODO 2: Use sendfile syscall to transmit file content directly

    // TODO 3: Handle partial transmissions with retry logic

    // TODO 4: Update stream and connection flow control windows

    // TODO 5: Send frame padding if required by frame size

    // TODO 6: Return error if transmission fails

    return nil

}

```

Milestone Checkpoints

Server Push Checkpoint: After implementing server push, verify functionality:

- Start server with push enabled: `go run cmd/server/main.go --enable-push`
- Request HTML page with linked resources: `curl -v --http2 https://localhost:8443/test.html`
- Verify PUSH_PROMISE frames in connection log showing promised resources
- Check that subsequent requests for pushed resources return cache hits
- Test push rejection by configuring client to send RST_STREAM for promised streams

Connection Coalescing Checkpoint:

After implementing coalescing, verify shared connections:

- Configure server with SAN certificate covering multiple hostnames
- Make requests to different hostnames: `curl --http2 https://api.example.com/test` then `curl -http2 https://www.example.com/test`
- Verify via connection logs that both requests use same underlying TCP connection

- Test hostname validation by requesting invalid hostname and confirming new connection

Performance Optimization Checkpoint: After implementing optimizations, measure performance improvements:

- Run benchmark with optimizations disabled: `go test -bench=BenchmarkFrameProcessing -count=5`
- Enable optimizations and run same benchmark: `go test -bench=BenchmarkFrameProcessing -count=5 -tags=optimized`
- Verify 20-40% improvement in frames per second and reduced memory allocations
- Monitor memory usage under load to confirm pool effectiveness: `go tool pprof http://localhost:8443/debug/pprof/heap`

Glossary

Milestone(s): This section provides terminology and definitions supporting all milestones (1-4) by establishing a common vocabulary for HTTP/2 concepts, frame types, HPACK compression terms, and protocol-specific terminology needed throughout the implementation.

This glossary serves as a comprehensive reference for HTTP/2 terminology encountered throughout the server implementation. Think of it as a technical dictionary that bridges the gap between RFC specifications and practical implementation. Just as a software engineer keeps API documentation handy while coding, this glossary provides quick access to precise definitions that prevent misunderstandings and implementation errors.

The terminology is organized into logical categories that align with the implementation milestones, making it easy to reference the appropriate vocabulary while working on specific components. Each definition includes not just the meaning but also implementation context and common usage patterns.

HTTP/2 Core Concepts

These fundamental terms form the conceptual foundation of HTTP/2 and appear throughout all implementation phases.

Term	Definition	Implementation Context
multiplexing	The ability to handle multiple concurrent request-response sequences (streams) over a single TCP connection, with frames from different streams interleaved	Core architectural principle enabling concurrent stream processing without head-of-line blocking
head-of-line blocking	Performance limitation in HTTP/1.1 where a slow request blocks all subsequent requests on the same connection, forcing browsers to open multiple connections	Problem that HTTP/2 multiplexing solves by allowing independent stream processing
binary framing	HTTP/2's structured binary protocol format that replaces HTTP/1.1's text-based message format with typed, length-prefixed frames	Foundation for efficient parsing and protocol extensibility in Milestone 1
stream	An independent, bidirectional sequence of frames within an HTTP/2 connection, corresponding to a single request-response exchange	Primary unit of request processing managed by the Stream Manager component
connection state	The overall configuration and active stream tracking for an HTTP/2 connection, including settings, flow control windows, and stream registry	Maintained by Connection component and shared across all processing components
server push	Proactive resource delivery mechanism where the server sends resources before the client explicitly requests them, based on anticipated needs	Advanced feature for performance optimization, typically implemented after core functionality
scope creep	The gradual expansion of project requirements beyond originally defined goals, particularly problematic in complex protocol implementations	Development risk managed through explicit non-goals and milestone boundaries

Frame Types and Structure

HTTP/2 uses typed frames as the basic unit of communication. Understanding frame types is essential for implementing the Binary Framing Engine.

Frame Type	Numeric Value	Purpose	Implementation Notes
DATA	0x0	Carries HTTP request/response body content with flow control	Subject to both connection and stream-level flow control windows
HEADERS	0x1	Carries compressed HTTP headers using HPACK encoding	May be followed by CONTINUATION frames if headers exceed frame size
PRIORITY	0x2	Adjusts stream priority weights and dependencies	Optional frame type for advanced scheduling implementations
RST_STREAM	0x3	Immediately terminates a specific stream with an error code	Used for stream-level error handling and cancellation
SETTINGS	0x4	Configures connection-level parameters like max frame size and flow control	Requires acknowledgment and affects all streams on connection
PUSH_PROMISE	0x5	Declares server intent to push a resource before client requests it	Used in server push implementations with promised stream allocation
PING	0x6	Measures round-trip time and tests connection liveness	Useful for connection health monitoring and diagnostics
GOAWAY	0x7	Initiates graceful connection termination with diagnostic information	Critical for proper error handling and connection lifecycle management
WINDOW_UPDATE	0x8	Adjusts flow control window sizes at connection or stream level	Essential for flow control implementation in Milestone 4
CONTINUATION	0x9	Carries additional header block fragments when HEADERS frame is insufficient	Required when compressed headers exceed maximum frame size limits

Frame Flags and Constants

Frame flags modify frame behavior and interpretation. These flags are frame-type specific and critical for correct protocol implementation.

Flag Name	Numeric Value	Applicable Frames	Meaning	Implementation Impact
END_STREAM	0x1	DATA, HEADERS	Indicates this frame is the last frame from the sender on this stream	Triggers stream state transition to half-closed or closed
END_HEADERS	0x4	HEADERS, PUSH_PROMISE, CONTINUATION	Indicates the complete header block has been transmitted	Enables header block assembly and HPACK decompression
PADDED	0x8	DATA, HEADERS	Frame payload includes padding bytes for security	Requires padding length parsing and payload size adjustment
PRIORITY	0x20	HEADERS	Frame includes stream priority information	Affects stream scheduling and resource allocation

Constant	Value	Description	Usage Context
MaxFrameSize	16777215	Maximum allowed frame payload size (24-bit length field)	Frame validation and buffer allocation limits
DefaultFrameSize	16384	Default maximum frame size before SETTINGS negotiation	Initial frame parsing buffer size
InitialWindowSize	65535	Default flow control window size for new connections and streams	Flow control initialization in Milestone 4

HPACK Compression Concepts

HPACK implements stateful header compression using multiple encoding strategies. These terms are essential for Milestone 2 implementation.

Term	Definition	Implementation Details
static table	Predefined 61-entry table containing common HTTP headers like <code>:method GET</code> and <code>:status 200</code>	Hardcoded lookup table enabling efficient encoding of frequent headers
dynamic table	Connection-specific table that accumulates headers seen during the connection lifetime	Requires FIFO eviction when size limits are exceeded, shared state between encoder/decoder
HPACK	HTTP/2 header compression algorithm defined in RFC 7541 using static tables, dynamic tables, and Huffman coding	Core compression engine requiring careful state synchronization between peers
Huffman coding	Variable-length character encoding that compresses header values by using shorter codes for frequent characters	Requires lookup tables for decoding and proper handling of padding bits
integer encoding	Variable-length numeric representation using prefix bits and continuation bytes for HPACK indices and string lengths	Supports 5, 6, and 7-bit prefixes with multi-byte extension for large values
FIFO eviction	First-in-first-out removal strategy used when dynamic table exceeds size limits	Critical for maintaining table consistency between encoder and decoder
continuation bytes	Additional bytes in integer encoding when value exceeds prefix bit capacity	Requires loop-based parsing with proper bounds checking
table corruption	Inconsistency between encoder and decoder dynamic tables, typically causing connection errors	Prevents successful header decompression and requires connection termination

Stream Management and State Machine

Stream lifecycle management requires understanding state transitions and the events that trigger them.

State	Numeric Value	Description	Valid Transitions
idle	0	Stream ID allocated but no frames sent or received	→ reserved-local, reserved-remote, open
reserved-local	1	Server has sent PUSH_PROMISE, waiting to send response headers	→ half-closed-remote, closed
reserved-remote	2	Client received PUSH_PROMISE, waiting for server response headers	→ half-closed-local, closed
open	3	Bidirectional frame exchange allowed, normal request processing state	→ half-closed-local, half-closed-remote, closed
half-closed-local	4	Local endpoint finished sending, remote may continue sending	→ closed
half-closed-remote	5	Remote endpoint finished sending, local may continue sending	→ closed
closed	6	Stream completely terminated, no further frame exchange permitted	Final state, no transitions

Term	Definition	Implementation Considerations
stream ID allocation	Monotonic numbering scheme where clients use odd numbers (1,3,5...) and servers use even numbers (2,4,6...)	Prevents ID collisions and enables proper stream ownership tracking
priority scheduling	Weight-based bandwidth distribution among active streams using deficit round-robin or similar algorithms	Affects frame transmission order and resource allocation decisions
stream state machine	Lifecycle management system tracking each stream's current state and valid transitions	Critical for protocol compliance and proper resource cleanup
half-closed	Stream state where one direction is closed (no more frames) but the other direction remains open for communication	Common state during response transmission after request completion
RST_STREAM	Immediate stream termination frame that forces transition to closed state regardless of current state	Used for error conditions and client-initiated cancellation

Flow Control and Window Management

Flow control prevents fast senders from overwhelming slow receivers using a credit-based system with hierarchical windows.

Term	Definition	Implementation Requirements
flow control	Credit-based backpressure management system using transmission permission tokens to prevent receiver overload	Dual-level system with both connection and stream windows
credit-based system	Flow control mechanism where receivers grant transmission credits via WINDOW_UPDATE frames, and senders track available credits	Requires careful accounting to prevent underflow and synchronization issues
hierarchical windows	Two-level flow control with connection-wide limits and per-stream limits, both enforced simultaneously	DATA frame transmission requires sufficient credit in both windows
window management	Tracking and updating of flow control window sizes through WINDOW_UPDATE frame processing and credit consumption	Central component coordinating between connection and stream flow control
backpressure handling	Data queueing mechanism when flow control windows are exhausted, with automatic resumption when credits become available	Prevents data loss while respecting receiver capacity limits
window exhaustion	Condition where flow control windows reach zero credit, blocking further DATA frame transmission	Requires queueing mechanism and proper resumption logic
window overflow	Arithmetic overflow condition when WINDOW_UPDATE increments exceed maximum window size limits	Protocol error requiring connection termination for safety
deficit round-robin	Fair scheduling algorithm that distributes available bandwidth among streams based on priority weights and accumulated deficits	Prevents priority inversion while maintaining fairness
priority inversion	Situation where lower priority streams block higher priority streams from making progress	Avoided through proper scheduling and flow control coordination

Error Handling and Protocol Compliance

HTTP/2 error handling distinguishes between connection-level and stream-level errors with different recovery mechanisms.

Error Code	Numeric Value	Scope	Description	Recovery Action
NO_ERROR	0x0	Both	Graceful shutdown with no error condition	Normal termination procedure
PROTOCOL_ERROR	0x1	Both	Generic protocol violation not covered by specific error codes	Connection termination typically required
INTERNAL_ERROR	0x2	Both	Implementation-specific error not related to protocol compliance	May be recoverable depending on context
FLOW_CONTROL_ERROR	0x3	Both	Sender violated flow control limits by exceeding available window credits	Immediate connection or stream termination
SETTINGS_TIMEOUT	0x4	Connection	Sender failed to acknowledge SETTINGS frame within timeout period	Connection termination required
STREAM_CLOSED	0x5	Stream	Frame received on stream that is already in closed state	Stream-level error, send RST_STREAM
FRAME_SIZE_ERROR	0x6	Both	Frame size violates limits set by SETTINGS_MAX_FRAME_SIZE	Typically connection-level error
REFUSED_STREAM	0x7	Stream	Stream refused before processing begins, client may retry	Stream-level error, allows retry
CANCEL	0x8	Stream	Stream cancelled by sender, equivalent to request timeout or user cancellation	Stream-level error, normal cleanup
COMPRESSION_ERROR	0x9	Connection	HPACK decompression failure due to table corruption or malformed data	Connection termination required

Error Code	Numeric Value	Scope	Description	Recovery Action
ENHANCE_YOUR_CALM	0xB	Both	Sender detected excessive resource usage or request rate	May be connection or stream level

Term	Definition	Implementation Strategy
protocol errors	Violations of HTTP/2 specification requirements that compromise protocol integrity	Require immediate response with appropriate error code
connection errors	Failures that affect the entire connection and require termination for safety	Generate GOAWAY frame with error code and diagnostic information
stream errors	Failures affecting individual streams while preserving connection integrity	Generate RST_STREAM frame for affected stream only
GOAWAY frame	Mechanism for graceful connection termination that specifies last processed stream ID and error code	Allows completing in-flight streams before connection closure
fault isolation	Design principle of limiting error impact to the minimum necessary scope	Prefer stream-level errors over connection errors when possible
error escalation	Incorrect promotion of stream-level errors to connection-level, causing unnecessary service disruption	Common implementation mistake that reduces robustness
blast radius	The scope of impact from error conditions, ideally minimized through proper error classification	Measure of error handling effectiveness
graceful degradation	Maintaining service for unaffected components during error conditions	Design goal for robust error handling

Testing and Debugging Terminology

Testing HTTP/2 implementations requires specialized techniques and tools for binary protocol validation.

Term	Definition	Testing Application
protocol fuzzing	Systematic generation of malformed HTTP/2 traffic to test implementation robustness against invalid inputs	Essential for security and reliability validation
milestone checkpoints	Concrete verification steps after each implementation phase showing expected behavior and outputs	Provides clear progress markers and regression detection
endianness	Byte order in multi-byte values, HTTP/2 uses network byte order (big-endian) for all numeric fields	Common source of parsing bugs on little-endian architectures
reserved bits	Protocol bits that must remain zero and are reserved for future extensions	Validation requirement that implementations must enforce
flow control deadlock	Condition where all streams are blocked by window exhaustion and no WINDOW_UPDATE frames can be processed	Critical failure mode requiring proper window management
state machine violation	Invalid stream state transition that violates HTTP/2 specification requirements	Indicates implementation bugs in stream lifecycle management

Advanced Features and Extensions

These terms relate to advanced HTTP/2 features and potential future enhancements beyond the core implementation.

Term	Definition	Implementation Context
PUSH_PROMISE	HTTP/2 frame type that declares server intent to push a resource before client requests it	Enables server push functionality with promised stream allocation
promised stream	Server-initiated stream declared via PUSH_PROMISE frame for proactive resource delivery	Follows special state machine rules and requires client cooperation
push candidate	Resource identified as beneficial for proactive server push based on analysis of request patterns	Determined through heuristics or explicit configuration
push policy	Rules governing which resources should be pushed and under what conditions	Balances performance benefits against bandwidth and cache considerations
connection coalescing	Reusing single HTTP/2 connection for multiple hostnames with compatible TLS certificates	Advanced optimization requiring careful certificate validation
alternative services	Mechanism to advertise different protocol endpoints for same service, such as HTTP/3 availability	Enables graceful protocol migration and optimization
certificate coverage	Validation that TLS certificate authorizes multiple hostnames via Subject Alternative Names or wildcards	Required for safe connection coalescing implementation
Subject Alternative Names	Certificate extension listing additional hostnames covered by the certificate beyond the primary subject	Technical mechanism enabling connection coalescing
zero-copy operations	I/O techniques that avoid copying data between kernel and user space, improving performance	Advanced optimization for high-throughput scenarios
vectored I/O	System call pattern that transmits multiple buffers in single operation, reducing syscall overhead	Performance optimization using writev/readv system calls
object pooling	Reuse strategy for expensive objects to reduce garbage collection pressure and allocation overhead	Performance optimization for high-frequency object creation
buffer classification	Size-based categorization of buffers for efficient pool management with reduced fragmentation	Memory management strategy for varying buffer size requirements
synthetic request	Artificial request headers representing client request for pushed resource, created by server	Implementation detail for pushed resource processing

Protocol Constants and Magic Values

HTTP/2 defines specific numeric constants that must be used correctly for protocol compliance.

Constant	Value	Context	Validation Requirements
HTTP/2 Connection Preface	"PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n"	Connection establishment	Must be first bytes sent by client
Frame Header Size	9 bytes	Binary framing	Fixed size for all frame types
Stream ID Reserved Bit	Most significant bit	Stream identification	Must be zero, R bit reserved
Settings Identifier Range	0x1-0x6 (defined settings)	Connection configuration	Unknown settings ignored, not error
Huffman EOS Symbol	256 (30-bit representation)	HPACK compression	Used for padding, never appears in valid strings
Default Header Table Size	4096 bytes	HPACK dynamic table	Initial dynamic table size limit
Maximum Window Size	$2^{31} - 1$	Flow control	Arithmetic overflow protection

Common Implementation Patterns

Understanding these patterns helps avoid typical HTTP/2 implementation mistakes and follows established practices.

Design Insight: Error Boundary Principles

HTTP/2's error handling philosophy prioritizes fault isolation - containing errors at the smallest possible scope. Stream errors should remain stream errors unless they indicate broader corruption like HPACK table inconsistency. This principle guides implementation decisions throughout the protocol stack.

Design Insight: State Machine Complexity

Stream state machines appear complex but follow logical patterns based on frame semantics. The `END_STREAM` flag drives most transitions, while `RST_STREAM` provides emergency termination from any state. Understanding these patterns simplifies implementation and debugging.

Design Insight: Flow Control Philosophy

HTTP/2 flow control implements receiver-driven backpressure, meaning receivers control the rate at which they consume data. This inverts traditional congestion control and requires careful coordination between connection and stream windows to prevent deadlocks.

Implementation Guidance

This implementation guidance provides practical advice for using the terminology correctly in code and avoiding common vocabulary-related mistakes.

Technology Recommendations

Component	Simple Option	Advanced Option
Protocol Constants	Hard-coded constants in main package	Separate constants package with documentation
Error Definitions	Simple error codes with string messages	Structured error types with context and recovery hints
State Tracking	Basic enum values with switch statements	State machine framework with transition validation
Logging Framework	Standard library logging with structured fields	Specialized HTTP/2 protocol logger with frame inspection

Recommended File Structure

```
internal/http2/
  constants.go           ← Protocol constants and frame type definitions
  errors.go              ← Error codes and structured error types
  glossary.go            ← Terminology validation and documentation helpers
  frame_types.go          ← Frame type constants and flag definitions
  state_machine.go        ← Stream state constants and transition validation
  hpack_constants.go      ← HPACK-specific constants and table definitions
  flow_control_constants.go ← Flow control limits and default values
```

Protocol Constants Implementation

```
package http2

// Frame type constants as defined in RFC 7540

const (

    FrameTypeDATA          uint8 = 0x0
    FrameTypeHEADERS       uint8 = 0x1
    FrameTypePRIORITY      uint8 = 0x2
    FrameTypeRST_STREAM    uint8 = 0x3
    FrameTypeSETTINGS       uint8 = 0x4
    FrameTypePUSH_PROMISE  uint8 = 0x5
    FrameTypePING           uint8 = 0x6
    FrameTypeGOAWAY          uint8 = 0x7
    FrameTypeWINDOW_UPDATE   uint8 = 0x8
    FrameTypeCONTINUATION    uint8 = 0x9
)

// Frame flag constants

const (
    FlagEndStream     uint8 = 0x1
    FlagAck           uint8 = 0x1 // For SETTINGS and PING frames
    FlagEndHeaders    uint8 = 0x4
    FlagPadded         uint8 = 0x8
    FlagPriority       uint8 = 0x20 // For HEADERS frame
)

// Protocol limits and defaults

const (
```

```
MaxFrameSize      uint32 = (1 << 24) - 1 // 16,777,215 bytes

DefaultFrameSize uint32 = 16384           // 16 KB

InitialWindowSize uint32 = 65535          // 64 KB - 1

MaxWindowSize     int64  = (1 << 31) - 1 // 2^31 - 1

)
```

Error Code Definitions

```
// ErrorCode represents HTTP/2 error codes as defined in RFC 7540
```

```
type ErrorCode uint32
```

```
const (
```

ErrorCodeNoError	ErrorCode = 0x0
ErrorCodeProtocolError	ErrorCode = 0x1
ErrorCodeInternalError	ErrorCode = 0x2
ErrorCodeFlowControlError	ErrorCode = 0x3
ErrorCodeSettingsTimeout	ErrorCode = 0x4
ErrorCodeStreamClosed	ErrorCode = 0x5
ErrorCodeFrameSizeError	ErrorCode = 0x6
ErrorCodeRefusedStream	ErrorCode = 0x7
ErrorCodeCancel	ErrorCode = 0x8
ErrorCodeCompressionError	ErrorCode = 0x9
ErrorCodeConnectError	ErrorCode = 0xa
ErrorCodeEnhanceYourCalm	ErrorCode = 0xb
ErrorCodeInadequateSecurity	ErrorCode = 0xc
ErrorCodeHTTP11Required	ErrorCode = 0xd

```
)
```

```
// String returns the error code name for debugging and logging
```

```
func (e ErrorCode) String() string {
```

```
    switch e {
```

```
        case ErrorCodeNoError:
```

```
            return "NO_ERROR"
```

```
        case ErrorCodeProtocolError:
```

```
            return "PROTOCOL_ERROR"
```

GO

```
// TODO: Add remaining error code string representations

default:

    return fmt.Sprintf("UNKNOWN_ERROR_CODE_%d", uint32(e))

}

}
```

Stream State Machine Implementation

```
// StreamState represents the state of an HTTP/2 stream  
GO  
  
type StreamState int  
  
const (  
    StateIdle StreamState = iota  
    StateReservedLocal  
    StateReservedRemote  
    StateOpen  
    StateHalfClosedLocal  
    StateHalfClosedRemote  
    StateClosed  
)  
  
// String returns human-readable state name for debugging  
  
func (s StreamState) String() string {  
    switch s {  
        case StateIdle:  
            return "IDLE"  
        case StateReservedLocal:  
            return "RESERVED_LOCAL"  
        case StateReservedRemote:  
            return "RESERVED_REMOTE"  
        case StateOpen:  
            return "OPEN"  
        case StateHalfClosedLocal:  
            return "HALF_CLOSED_LOCAL"  
        case StateHalfClosedRemote:  
    }  
}
```

```
        return "HALF_CLOSED_REMOTE"

    case StateClosed:

        return "CLOSED"

    default:

        return "UNKNOWN_STATE"

    }

}

// CanReceiveFrame checks if a frame type is valid for the current stream state

func (s StreamState) CanReceiveFrame(frameType uint8) bool {

    // TODO: Implement frame type validation for each state

    // Refer to RFC 7540 Section 5.1 for state machine rules

    switch s {

    case StateIdle:

        return frameType == FrameTypeHEADERS || frameType == FrameTypePRIORITY

    case StateOpen:

        return true // All frame types allowed

        // TODO: Add remaining state validations

    default:

        return false

    }

}
```

HPACK Constants and Helpers

```
// HPACK constants from RFC 7541                                         GO

const (
    StaticTableSize      = 61      // Number of entries in static table
    DefaultTableSize    = 4096   // Default dynamic table size in bytes
    HuffmanEOSSymbol   = 256    // End-of-string symbol for Huffman padding
    HeaderEntryOverhead = 32     // Bytes of overhead per dynamic table entry
)

// Huffman decoding constants

const (
    HuffmanAcceptMask = 0x40000000 // Mask for accept state
    HuffmanSymbolMask = 0xFF       // Mask for symbol extraction
)

// Integer encoding prefix sizes used in HPACK

const (
    IndexedHeaderPrefix      = 7    // 7-bit prefix for indexed header field
    LiteralIncrementalPrefix = 6    // 6-bit prefix for literal with incremental indexing
    LiteralNeverIndexedPrefix = 4    // 4-bit prefix for literal never indexed
    TableSizeUpdatePrefix    = 5    // 5-bit prefix for dynamic table size update
)
```

Validation Helper Functions

```
// ValidateFrameType checks if frame type is known and supported GO

func ValidateFrameType(frameType uint8) error {

    switch frameType {

        case FrameTypeDATA, FrameTypeHEADERS, FrameTypePRIORITY,
            FrameTypeRST_STREAM, FrameTypeSETTINGS, FrameTypePUSH_PROMISE,
            FrameTypePING, FrameTypeGOAWAY, FrameTypeWINDOW_UPDATE,
            FrameTypeCONTINUATION:

            return nil

        default:

            return fmt.Errorf("unknown frame type: 0x%x", frameType)

    }

}

// ValidateStreamID checks stream ID constraints

func ValidateStreamID(streamID uint32, frameType uint8) error {

    // TODO: Implement stream ID validation rules

    // - Check reserved bit is not set

    // - Validate odd/even requirements for client/server streams

    // - Check frame type compatibility with stream ID (0 for connection frames)

    if streamID&0x80000000 != 0 {

        return NewProtocolError(ErrorCodeProtocolError, "stream ID reserved bit set")

    }

    // TODO: Add remaining validation logic

    return nil

}
```

Language-Specific Implementation Hints

Go-Specific Recommendations:

- Use `const` declarations for all protocol constants to prevent accidental modification
- Implement `String()` methods for enum types to improve debugging output
- Use `sync.Map` for concurrent access to stream state when performance is critical
- Leverage `encoding/binary` for frame header parsing with proper endianness handling
- Use `fmt.Errorf` with error codes for structured error messages

Memory Management:

- Pre-allocate slices for frame parsing buffers to reduce garbage collection pressure
- Use object pools for frequently allocated structures like Frame and HeaderEntry
- Consider using `unsafe` package for zero-copy operations in performance-critical paths
- Implement buffer reuse patterns for HPACK dynamic table entries

Concurrency Considerations:

- Protect shared state like dynamic tables with `sync.RWMutex` for concurrent access
- Use channels for communication between frame processing goroutines
- Implement proper shutdown signaling with context cancellation
- Avoid sharing mutable state between streams whenever possible

Milestone Checkpoints

Terminology Validation Checkpoint: After implementing basic constants and types, verify correct usage:

```
go test ./internal/http2/... -v
```

BASH

Expected output should show successful validation of:

- All frame type constants recognized correctly
- Error code string representations match RFC 7540
- Stream state transitions follow specification rules
- HPACK constants match RFC 7541 requirements

Common Terminology Mistakes:

- Using `0x8` for `WINDOW_UPDATE` instead of decimal 8 in logging
- Confusing stream states with connection states in error messages
- Mixing up frame flag values between different frame types
- Using incorrect terminology in log messages that makes debugging difficult

Debugging with Proper Terminology

When debugging HTTP/2 implementations, use precise terminology in log messages:

```
// Good: Precise terminology with context  
  
log.Printf("Stream %d: Invalid state transition from %s to %s on %s frame",  
           streamID, currentState, newState, frameTypeName)  
  
// Bad: Vague terminology without context  
  
log.Printf("Stream error in processing")
```

GO

Structured logging with proper terminology enables easier problem diagnosis:

```
logger.WithFields(logrus.Fields{  
  
    "stream_id": streamID,  
  
    "frame_type": frameType.String(),  
  
    "current_state": state.String(),  
  
    "error_code": errorCode.String(),  
  
}).Error("Stream state machine violation")
```

GO