BUILDING LEAN, STANDALONE WEB SERVERS AND ROUTING ENGINES

Advisor: Dr. Chris Pollett

Committee: Dr. Navrati Saxena

Committee: Dr. Genya Ishigaki

MASTER'S DEFENSE

BY AJITA SHRIVASTAVA

CONTENTS

INTRODUCTION

- **Project Objective:** Develop lightweight, single-file PHP servers for web and email operations.
- **Motivation:** Simplify server solutions for scalability, educational use, and embedded use cases.
- **Existing Components:** Atto servers - WebSite, MailSite and GopherSite.
- **Existing Features:**
  - Handle HTTP requests using a micro-framework design.
  - Asynchronous I/O, PHP superglobals, and file caching.
- **My Contributions:**
  - Secure handling, HTTP/2 support, framing layer, header compression.
  - Protocol commands support and functionality enhancements in email server.

BACKGROUND – WEB SERVERS

- **Purpose**: Web servers handle HTTP requests, serving web pages, APIs, or static content to clients (usually browsers).
- **Common Examples**:
  - **Nginx**: Known for high performance, load balancing, and reverse proxying.
  - **Apache**: A robust, widely used web server offering extensive configurability.
  - **Node.js**: A runtime enabling event-driven, non-blocking I/O for real-time applications.
- **Challenges**:
  - Resource-heavy for small-scale applications.
  - Complexity increases with modular add-ons and on-demand configurations.

BACKGROUND – EMAIL SERVERS

- **Purpose**: Email servers manage the sending, receiving, and storage of emails using standardized protocols.
- **Common Examples**:
  - **Postfix**: A mail transfer agent (MTA) focusing on email delivery using SMTP.
  - **Dovecot**: A mail delivery agent (MDA) for handling mailbox storage and retrieval with IMAP and POP3.
- **Challenges**:
  - High resource demands for full-featured implementations.
  - Complexity in securing protocols and managing user authentication.

BACKGROUND

- **Related Work**: Apache, Nginx, Postfix, and Dovecot dominate the field but are complex, resource-heavy, and lack modularity.
- **Challenges and Improvements**: Addressed inefficiencies by developing lightweight, single-file servers with modular, core protocol-focused functionality.
- **Prior Work**: Initial implementation offered HTTP support and asynchronous handling providing a foundation for the enhancements I worked on.

IMPLEMENTATION – PRELIMINARY WORK

- **MailSite – HELP Command Implementation**
  - Added the parsing for Help command which will display usage of the various supported commands.
  - Helped to grasp the architecture and core functionality of the existing server.
- **Implementation of Routes with WebSite Server**:
  - Developed routing system for web traffic.
  - Set up index.php for controlling request handling and route management.
  - Resulted in a fully functional routing system allowing navigation to different web pages.
- **HTTP/2 Detection**:
  - Developed method to detect whether incoming requests used HTTP/1.1 or HTTP/2.
  - Inspected encrypted traffic to determine protocol and routed accordingly for proper handling of each protocol.

IMPLEMENTATION – PRELIMINARY WORK

- **Conversion of HTTP/1.1 to HTTP/2**:
  - Implemented HTTP/1.1 to HTTP/2 frame conversion, including Frame, HeaderFrame, and DataFrame classes.
  - Focused on decoding frame headers and handling payload using binary-to-hex conversion.
  - Established a foundation for complex HTTP/2 features like framing layer and header compression.

```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Content-Length: 35

bookId=12345&author=paulo+Coehlo
```

```
[Frame 1] => 00 00 87 01 20 00 00 49 FF EB 6E 5E 9A 37 E4 83 D8 23 C5 A9
5D 73 C7 A3 2E DC B0 7B 04 8A 95 28 5F 8D D8 D7 5C 30 F4 D1 07 B0 5F 93
7E 3B FA 65 E7 D2 ED 1D 36 EC A9 1B F9 F4 83 D8 3B F7 EF 77 4C BC FA
63 86 37 63 DF B6 A0 E3 C7 97 87 48 3D 83 4E DC 39 F2 DF 9F 4E 6B 20 D3
B7 0E 7C B7 EA E1 97 3D 90 55 7D 55 0F 7E EE 99 77 74 B6 66 5D D9 FA 68
83 D8 33 D6 02

[Frame 2] => 00 00 20 00 A0 00 00 49 FF C5 BF 7E B9 39 3D C7 2C F4 D6 6C
3D 7A 68 DF CB DE 18 7A EC DF 5C 3D F9 74 6C 6F
```

IMPLEMENTATION – PRELIMINARY WORK

```
[headerFrame] => Array
(
    [length] => 135
    [type] => 0x1
    [flags] => Array
        (
            [end_stream_flag] => 0
            [end_header_flag] => 1
            [padded_flag] => 0
            [prior_flag] => 0
        )

    [R] => 0
    [streamID] => 18943
    [padLength] => not set
    [E] => not set
    [StreamDepenedency] => not set
    [weight] => not set
    [requestHeaders] => :method = GET
                        :scheme = HTTP/1.1
                        :path = /doc/test.html
                        Host = www.test101.com
                        Accept = image/gif, image/jpeg, */*
                        Content-Length = 35

    [padding] => not set
)
```

```
[dataFrame] => Array
(
    [length] => 32
    [type] => 0x0
    [flags] => Array
        (
            [end_stream_flag] => 1
            [end_header_flag] => 1
            [padded_flag] => 0
            [prior_flag] => 0
        )

    [R] => 0
    [streamID] => 18943
    [padLength] => not set
    [data] => bookId=12345&author=paulo+Coehlo
    [padding] => not set
)
```

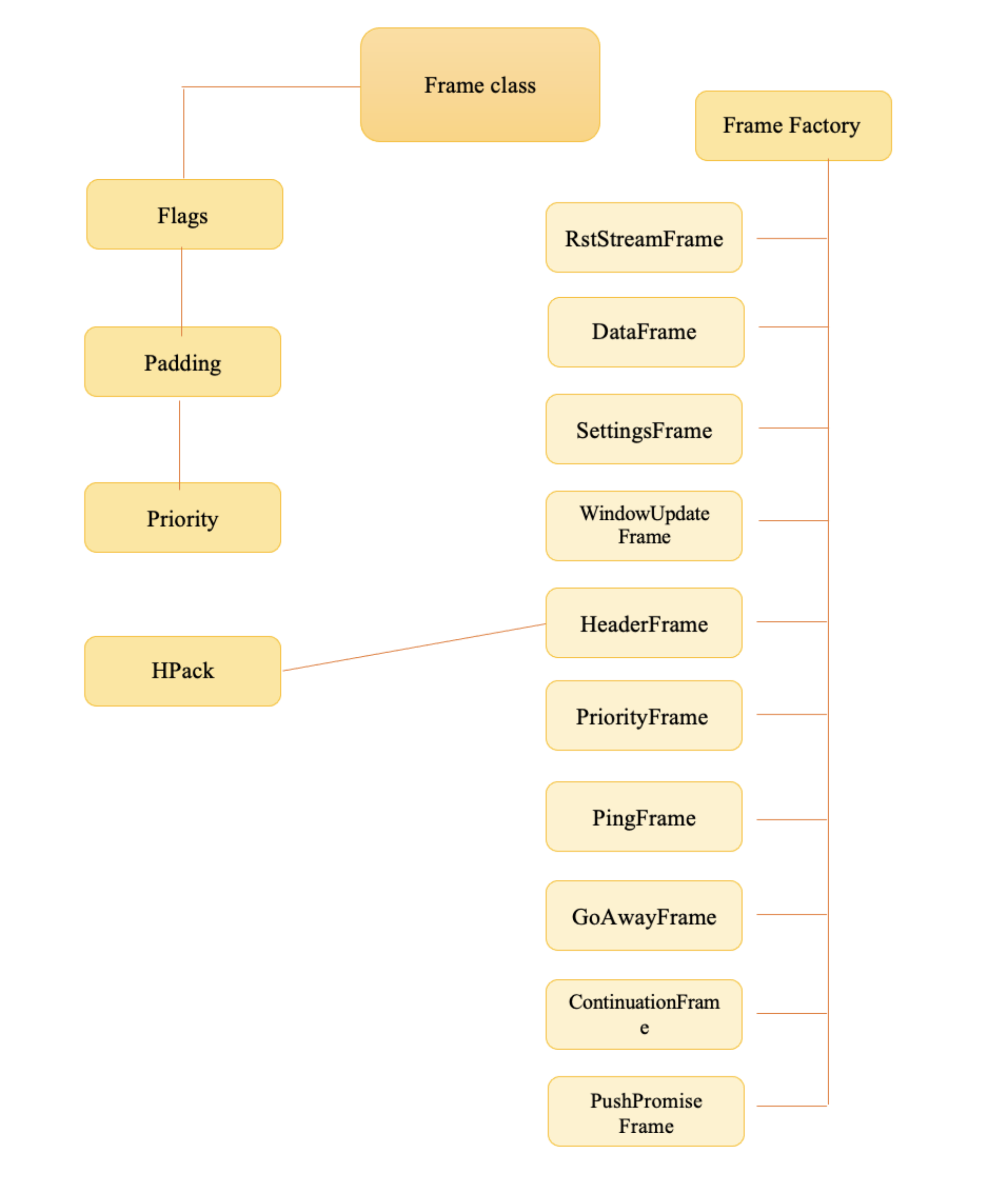
IMPLEMENTATION – LARGE SCALE IMPROVEMENTS

1. Secure handling
2. Binary framing layer
   - Frame class, Padding, Priority, Flags and Frame factory
3. Hpack implementation
   - Static and dynamic table, Indexing schemes, Huffman encoding, Encoding and Decoding
4. WebSite integration
5. Mail server commands implementation

IMPLEMENTATION – SECURE HANDLING

- Establishing secure connections focuses on encryption protocols and certificate management to ensure data privacy and integrity.
- The server is configured using socket programming in PHP to implement secure connections with ALPN (Application-Layer Protocol Negotiation).
- SSL is enabled, and the server context is initialized with certificates, private keys, peer verification settings, and supported ALPN protocols.
- For development, self-signed certificates and self-generated private keys are used, with peer verification disabled as the server runs locally.
- Self-signed certificates and keys are created with the SSL command-line tool and stored in .pem files, ensuring compatibility with secure protocols and platforms.

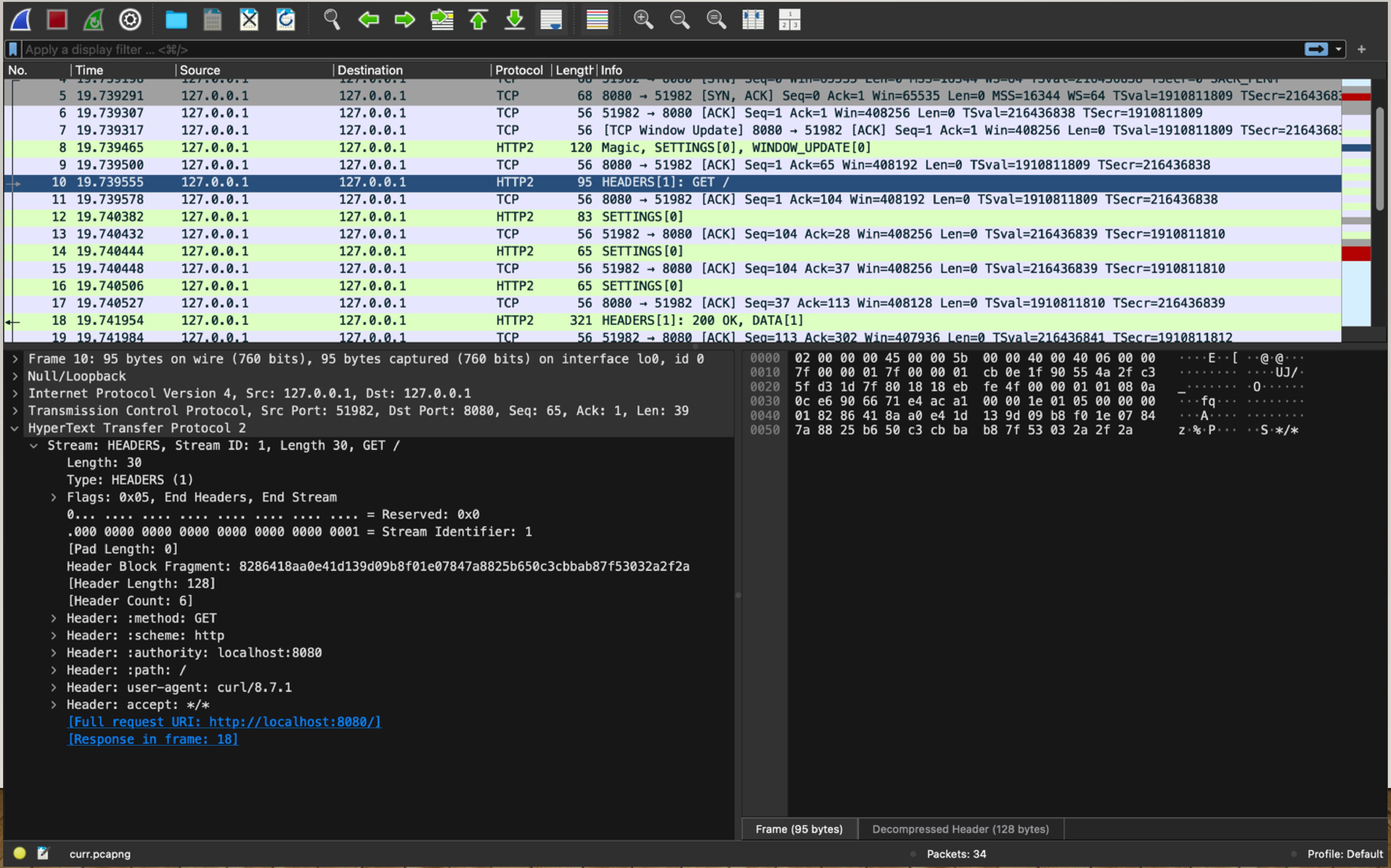
IMPLEMENTATION BINARY FRAMING LAYER



IMPLEMENTATION – FRAME CLASS

Length (24)	
Type (8)	Flags (8)
R	Stream Identifier (31)
Frame Payload	

- The Frame class is the core class in HTTP/2's binary framing layer, acting as the base for all specific frame types.
- It manages the shared structure and functionality common to all frame types, simplifying frame management in the protocol.
- A key responsibility is handling the 9-byte frame header, which contains essential metadata like frame length, type, flags, and stream identifier.


Apply a display filter ... <⌘/>
No. | Time | Source | Destination | Protocol | Length | Info
5 19.739291 127.0.0.1 127.0.0.1 TCP 68 8080 → 51982 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=1910811809 TSecr=2164368
6 19.739307 127.0.0.1 127.0.0.1 TCP 56 51982 → 8080 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=216436838 TSecr=1910811809
7 19.739317 127.0.0.1 127.0.0.1 TCP 56 [TCP Window Update] 8080 → 51982 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1910811809 TSecr=2164368
8 19.739465 127.0.0.1 127.0.0.1 HTTP2 120 Magic, SETTINGS[0], WINDOW_UPDATE[0]
9 19.739500 127.0.0.1 127.0.0.1 TCP 56 8080 → 51982 [ACK] Seq=1 Ack=65 Win=408192 Len=0 TSval=1910811809 TSecr=216436838
10 19.739555 127.0.0.1 127.0.0.1 HTTP2 95 HEADERS[1]: GET /
11 19.739578 127.0.0.1 127.0.0.1 TCP 56 8080 → 51982 [ACK] Seq=1 Ack=104 Win=408192 Len=0 TSval=1910811809 TSecr=216436838
12 19.740382 127.0.0.1 127.0.0.1 HTTP2 83 SETTINGS[0]
13 19.740432 127.0.0.1 127.0.0.1 TCP 56 51982 → 8080 [ACK] Seq=104 Ack=28 Win=408256 Len=0 TSval=216436839 TSecr=1910811810
14 19.740444 127.0.0.1 127.0.0.1 HTTP2 65 SETTINGS[0]
15 19.740448 127.0.0.1 127.0.0.1 TCP 56 51982 → 8080 [ACK] Seq=104 Ack=37 Win=408256 Len=0 TSval=216436839 TSecr=1910811810
16 19.740506 127.0.0.1 127.0.0.1 HTTP2 65 SETTINGS[0]
17 19.740527 127.0.0.1 127.0.0.1 TCP 56 8080 → 51982 [ACK] Seq=37 Ack=113 Win=408128 Len=0 TSval=1910811810 TSecr=216436839
18 19.741954 127.0.0.1 127.0.0.1 HTTP2 321 HEADERS[1]: 200 OK, DATA[1]
19 19.741984 127.0.0.1 127.0.0.1 TCP 56 51982 → 8080 [ACK] Seq=113 Ack=302 Win=407936 Len=0 TSval=216436841 TSecr=1910811812
Frame 10: 95 bytes on wire (760 bits), 95 bytes captured (760 bits) on interface lo0, id 0
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 51982, Dst Port: 8080, Seq: 65, Ack: 1, Len: 39
HyperText Transfer Protocol 2
Stream: HEADERS, Stream ID: 1, Length 30, GET /
Length: 30
Type: HEADERS (1)
Flags: 0x05, End Headers, End Stream
0... .... = Reserved: 0x0
.000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
[Pad Length: 0]
Header Block Fragment: 8286418aa0e41d139d09b8f01e07847a8825b650c3cbbab87f53032a2f2a
[Header Length: 128]
[Header Count: 6]
Header: :method: GET
Header: :scheme: http
Header: :authority: localhost:8080
Header: :path: /
Header: user-agent: curl/8.7.1
Header: accept: */*
[Full request URI: http://localhost:8080/]
[Response in frame: 18]
0000 02 00 00 00 45 00 00 5b 00 00 40 00 40 06 00 00
0010 7f 00 00 01 7f 00 00 01 cb 0e 1f 90 55 4a 2f c3
0020 5f d3 1d 7f 80 18 18 eb fe 4f 00 00 01 01 08 0a
0030 0c e6 90 66 71 e4 ac a1 00 00 1e 01 05 00 00 00
0040 01 82 86 41 8a a0 e4 1d 13 9d 09 b8 f0 1e 07 84
0050 7a 88 25 b6 50 c3 cb ba b8 7f 53 03 2a 2f 2a
Frame (95 bytes)
Decompressed Header (128 bytes)
curr.pcapng
Packets: 34
Profile: Default

IMPLEMENTATION – PADDING, PRIORITY AND FLAGS

- **Flag Class**: Represents individual flags in HTTP/2 frames, encapsulating the flag's name and bit value. It provides a constructor for initializing flags, ensuring clear and consistent flag usage in the protocol.
- **Flags Class**: Manages a collection of flags for HTTP/2 frames, enabling addition, removal, and validation. It includes methods like add(), discard(), and contains(), along with __toString() for a human-readable representation of active flags.
- **Padding Trait**: Handles padding in HTTP/2 frames, ensuring correct frame size alignment. It includes methods like serializePaddingData() and parsePaddingData() to add or extract padding based on the 'PADDED' flag, crucial for proper frame structure.
- **Priority Class**: Manages the HTTP/2 PRIORITY frame, dealing with stream dependencies, weight, and exclusivity. It provides methods to serialize and parse priority data, helping optimize resource allocation and data flow for improved performance.

IMPLEMENTATION – FRAME FACTORY

- FrameFactory uses the Factory pattern to dynamically create frame objects based on their type number, enhancing flexibility and extensibility.
- It maps frame type numbers to corresponding frame classes, such as DataFrame, HeaderFrame, and PriorityFrame, simplifying frame management.
- The factory decouples frame creation from other parts of the system, making it easier to maintain and extend, including adding custom frames.
- It supports various HTTP/2 frame types, including DataFrame, HeaderFrame, SettingsFrame, and others, each serving specific roles in the protocol.

```php
class FrameFactory {
    public static $frames = [
        0x0 => DataFrame::class,
        0x1 => HeaderFrame::class,
        0x2 => PriorityFrame::class,
        0x3 => RstStreamFrame::class,
        0x4 => SettingsFrame::class,
        0x5 => PushPromiseFrame::class,
        0x6 => PingFrame::class,
        0x7 => GoAwayFrame::class,
        0x8 => WindowUpdateFrame::class,
        0x9 => ContinuationFrame::class,
    ];
}
```

IMPLEMENTATION – FRAMING LAYER CLASSES

Frame

HeaderFrame	Encodes headers of the request in compressed format.
DataFrame	Encodes the actual payload.
SettingsFrame	Encodes all parameters for config.
WindowUpdate	Increases stream / window capacity.
RstStream	Used to terminate a stream abruptly in case of errors.
Priority	Adjusts stream priorities.
Ping	Measures latency of the connection
GoAway	Used to gracefully close the connection.
Continuation	Continues large header blocks.
PushPromise	Promises server push resources

IMPLEMENTATION – HPACK IMPLEMENTATION

Key	Value
Host	www.test.com
Accept	image/gif, image/jpeg, */*
Content-Length	35

- HPACK is the compression format used in HTTP/2 for headers.
- Its primary purpose is to reduce the size of HTTP header data, improving speed and efficiency in communication.
- HPACK uses two main techniques:
  - Huffman coding for literal strings and
  - indexing for common headers.
- The implementation involves static and dynamic tables for header indexing, alongside Huffman encoding to minimize overhead.

IMPLEMENTATION – STATIC AND DYNAMIC TABLES

- **Static Table**: Predefined set of common HTTP header fields.
- **Dynamic Table**: Stores headers added during runtime for efficient reuse.
- This implementation defines a 'headers_table':
  - Combination of both tables
  - First 61 entries are populated with the static table in the constructor
  - 'max_table_size' to tune memory utilization
  - FIFO eviction policy

Index	Header Name	Header Value
1	:authority	(empty)
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
• • •		
61	www-authenticate	(empty)
62	--	--
63		
64		
• • •		

IMPLEMENTATION – INDEXING SCHEMES

- There are two types of indexing schemes:
  - **Indexed**
  - **Literal**
- These schemes utilize the headers_table to compress and decompress header fields.
- The table serves as the core data structure, with each indexing scheme modifying the table in different ways.
- Each header field (key, value pair) is encoded into binary with the starting bits representing its bit pattern.

IMPLEMENTATION – INDEXED HEADER FIELDS

- Identifies an entry in the table.
- An indexed header field starts with a 1-bit pattern, followed by the index of the matching header field

0	1	2	3	4	5	6	7
1	Index						

IMPLEMENTATION – LITERAL HEADER FIELDS

- Literal header fields consist of key-value pairs where the value is explicitly provided (literal).
- The key can either be retrieved from an index or encoded as a literal string.
- Literal strings are encoded using Huffman encoding or ASCII-hex representation.
- There are three forms of literal header field representations defined:
  - **With indexing**: The header field is added to the dynamic table for future reference.
  - **Without indexing**: The header field is transmitted without being added to the dynamic table.
  - **Never indexed**: The header field is explicitly marked to never be added to the dynamic table, ensuring privacy or security.

IMPLEMENTATION – LHF WITH INCREMENTAL INDEXING

INDEXED NAME

0	1	2	3	4	5	6	7
0	1	Index					
H	Value Length						
Value string							

NEW NAME

0	1	2	3	4	5	6	7
0	1	0					
H	Name length						
Name String (Length octets)							
H	Value Length						
Value String (Length octets)							

IMPLEMENTATION – LHF WITHOUT INDEXING

INDEXED NAME

0	1	2	3	4	5	6	7
0	0	0	0	Index			
H	Value Length						
Value string							

NEW NAME

0	1	2	3	4	5	6	7
0	0	0	0	0			
H	Name length						
Name String (Length octets)							
H	Value Length						
Value String (Length octets)							

IMPLEMENTATION – LHF NEVER INDEX

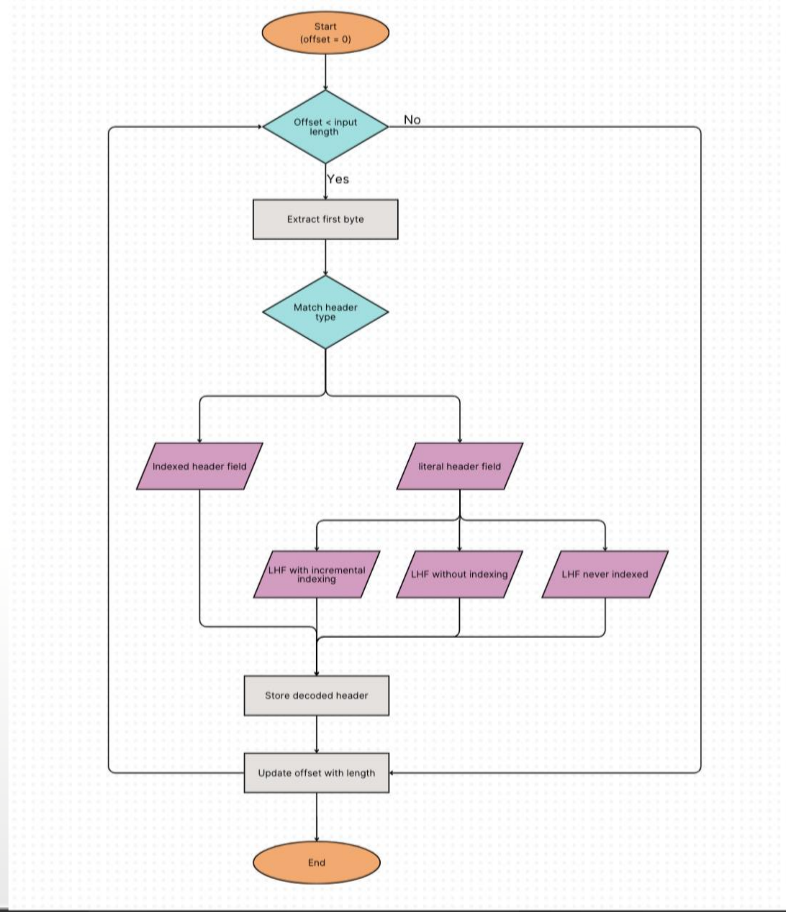
INDEXED NAME

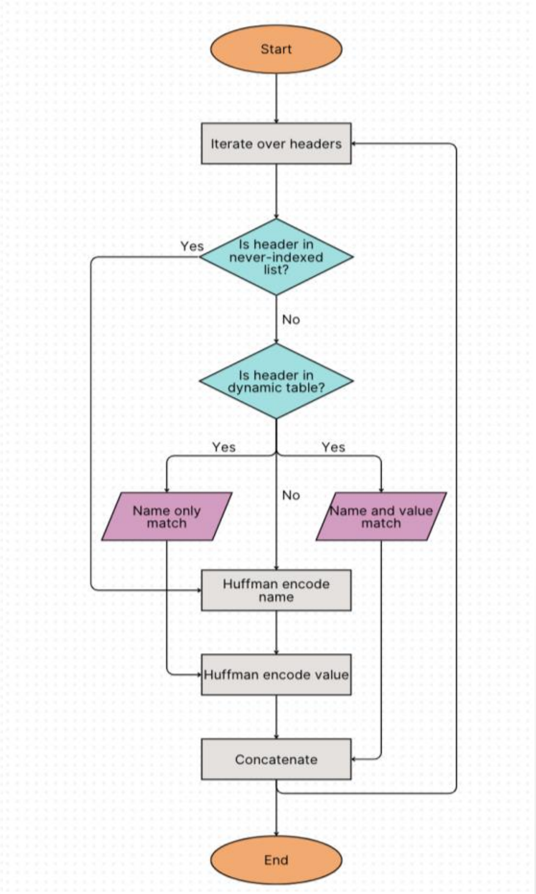
<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="4">Index</td></tr>
<tr><td>H</td><td colspan="7">Value Length</td></tr>
<tr><td colspan="8">Value string</td></tr>
</table>

NEW NAME

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="4">0</td></tr>
<tr><td>H</td><td colspan="7">Name length</td></tr>
<tr><td colspan="8">Name String (Length octets)</td></tr>
<tr><td>H</td><td colspan="7">Value Length</td></tr>
<tr><td colspan="8">Value String (Length octets)</td></tr>
</table>

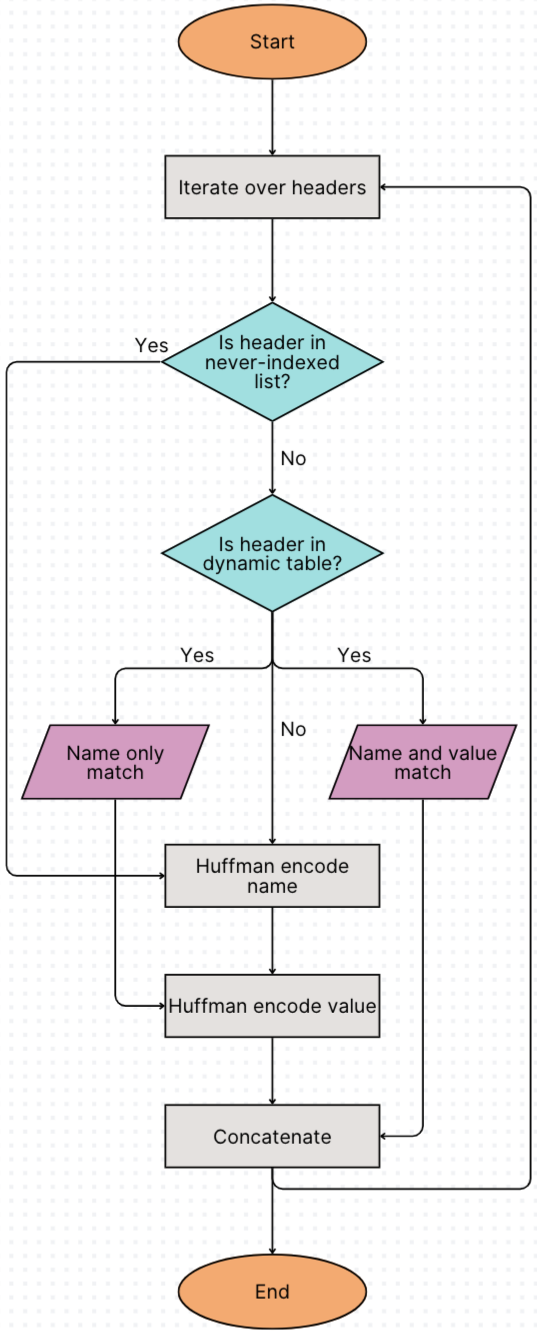
IMPLEMENTATION – HPACK COMPRESSION WORKFLOWS





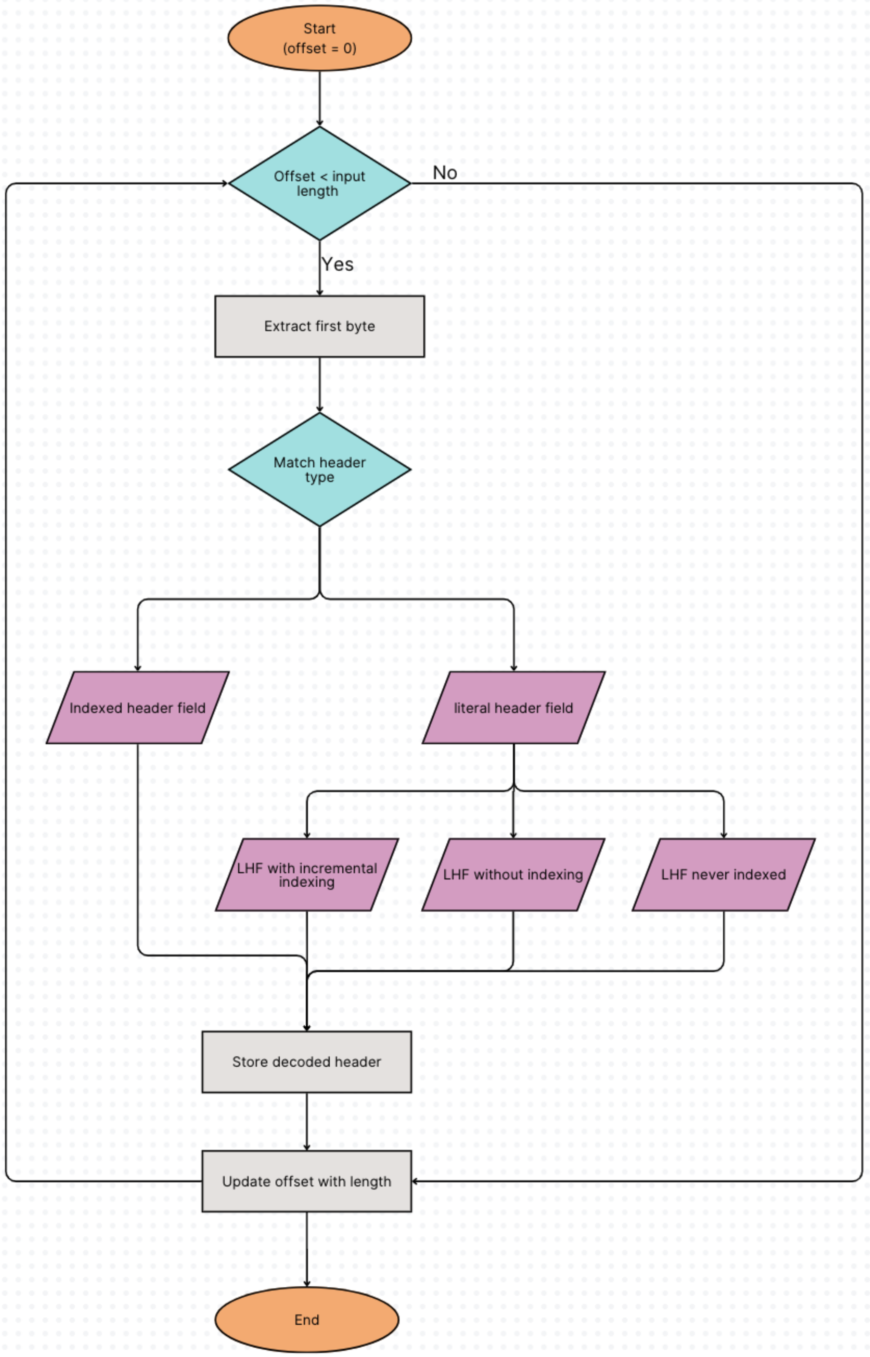
IMPLEMENTATION

HPACK ENCODING WORKFLOW



IMPLEMENTATION

HPACK DECODING WORKFLOW



IMPLEMENTATION – HUFFMAN ENCODING

- The Huffman code is used for encoding string literals, specifically in HTTP headers.
- Generated from statistical analysis of a large sample of HTTP headers.
- Canonical form of Huffman code with modifications to ensure that no symbol has a unique code length.
- Utilizes buffer matching at every bit during the encoding process.

IMPLEMENTATION – HUFFMAN ENCODING EXAMPLE

Input → Huffman codes per character → Concatenation and padding → Chunks to hex

text/html

IMPLEMENTATION – HUFFMAN ENCODING EXAMPLE

Input → Huffman codes per character → Concatenation and padding → Chunks to hex

Character	Huffman Code
t	01001
e	00101
x	1111001
/	011000
h	100111
m	101001
l	101000

IMPLEMENTATION – HUFFMAN ENCODING EXAMPLE

Input → Huffman codes per character → **Concatenation and padding** → Chunks to hex

010010010111110010100101100010011101001101001101000

01001001 01111100 10100101 10001001 11010011 01001101 00011111

IMPLEMENTATION – HUFFMAN ENCODING EXAMPLE

Input → Huffman codes per character → Concatenation and padding → **Chunks to hex**

01001001 01111100 10100101 10001001 11010011 01001101 00011111

49	7C	A5	89	D3	4D	1F

IMPLEMENTATION – HUFFMAN ENCODING EXAMPLE

Input → Huffman codes per character → Concatenation and padding → Chunks to hex

Metric	Hex Encoding	Huffman Encoding
Final Encoded Output	74 65 78 74 2F 68 74 6D 6C	49 7C A5 89 D3 4D 1F
Compression Length	9 bytes	7 bytes

Result: Huffman encoding chosen due to higher compression efficiency.

IMPLEMENTATION – HUFFMAN DECODING EXAMPLE

Input → Hex to binary → Buffer matching → Result

25b650c3cbbab87f

IMPLEMENTATION – HUFFMAN DECODING EXAMPLE

Input → Hex to binary → Buffer matching → Result

25	b6	50	c3	cb	ba	b8	7f
0010 0101	1011 0110	0101 0000	1100 0011	1100 1011	1011 1010	1011 1000	0111 1111

IMPLEMENTATION – HUFFMAN DECODING EXAMPLE

Input → Hex to binary → **Buffer matching** → Result

Current Buffer	Match Found?
0	No
00	No
001	No
0010	No
00100	Yes

Decoded character: c (ASCII: 99)

Buffer	Decoded Character	ASCII
00100	c	99
101101	u	117
101100	r	114
101000	l	108
011000	/	47
011110	8	56
010111	.	46
011101	7	55
010111	.	46
00001	1	49

IMPLEMENTATION – HUFFMAN DECODING EXAMPLE

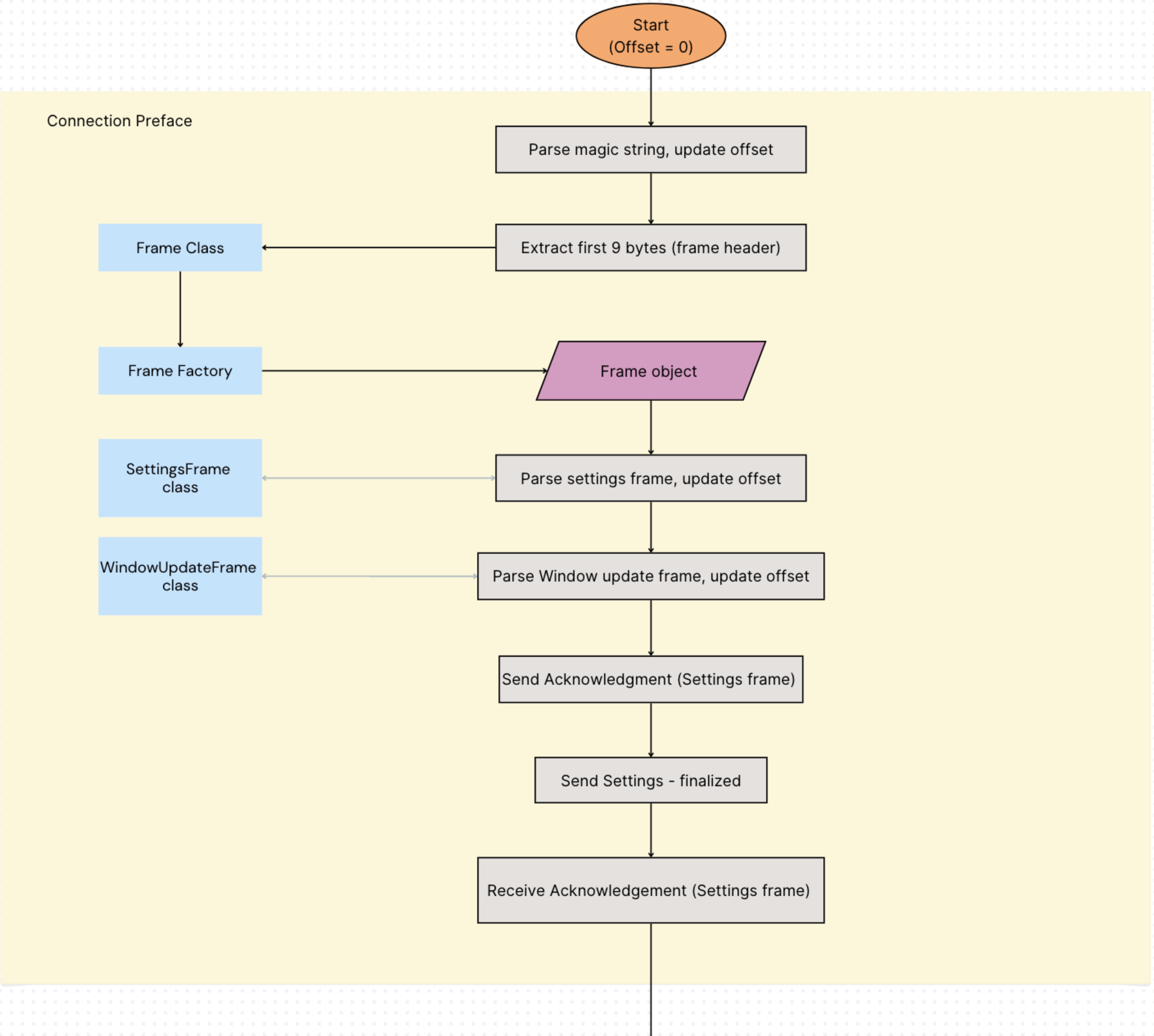
Input → Hex to binary → Buffer matching → **Result**

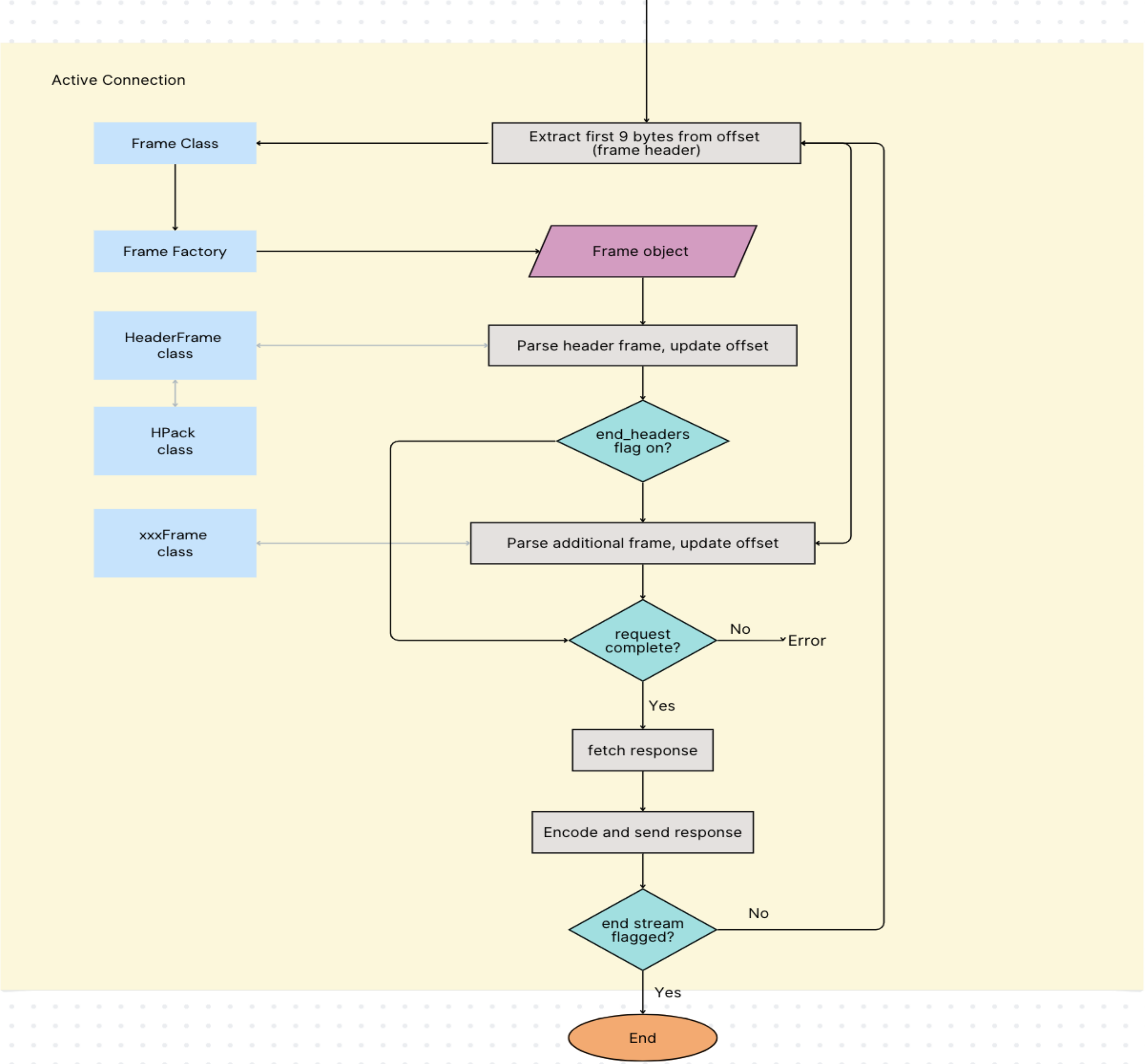
curl/8.7.1

NOTE: Padding detected and removed: 111111

IMPLEMENTATION – INTEGRATION INTO WEBSITE

- Till now, we discussed
  1. Secure handling
  2. Binary framing layer (Frame class, Padding, priority, flags and, Frame factory)
  3. Hpack implementation (Static and dynamic table, Indexing schemes, Huffman encoding, Encoding, Decoding)
- Now, we will go through the integration of these into Website and cover the overall flow for frame parsing.
- After a TCP connection is established with a new client, the frame parsing flow occurs in 2 phases:
  1. Connection Preface
  2. Active Connection
- Until the connection is closed by client.


Start
(Offset = 0)
Connection Preface
Parse magic string, update offset
Frame Class
Extract first 9 bytes (frame header)
Frame Factory
Frame object
SettingsFrame class
Parse settings frame, update offset
WindowUpdateFrame class
Parse Window update frame, update offset
Send Acknowledgment (Settings frame)
Send Settings - finalized
Receive Acknowledgement (Settings frame)



Active Connection
Frame Class
Extract first 9 bytes from offset (frame header)
Frame Factory
Frame object
HeaderFrame class
Parse header frame, update offset
HPack class
end_headers flag on?
xxxFrame class
Parse additional frame, update offset
request complete?
No
Error
Yes
fetch response
Encode and send response
end stream flagged?
No
Yes
End

EXPERIMENTS

- Experiment 1: Analyzing the setup process
- Experiment 2: Measuring Response Time
- Experiment 3: Measuring Total Time for Request
- Experiment 4: Header Compression
- Experiment 5: Apache BenchMark Results

EXPERIMENT - ANALYZING THE SETUP PROCESS

Metric	NGINX	WebSite
Number of Steps Required	10	5
Time Taken for Installation	20 minutes	5 minutes
Space Utilized	2.6 MB	145 KB

EXPERIMENT - MEASURING RESPONSE TIME

- For most web applications and the specific use cases this server was built for, especially those with low traffic or casual use, a 0.02-second difference can be deemed negligible.
- Illustrations on the right depict one of the sample observations. The test was run several times to ensure reliability of results.

Metric	Nginx	WebSite
Response time	0.068991s	0.088829s

EXPERIMENT - MEASURING CONNECTION TIME

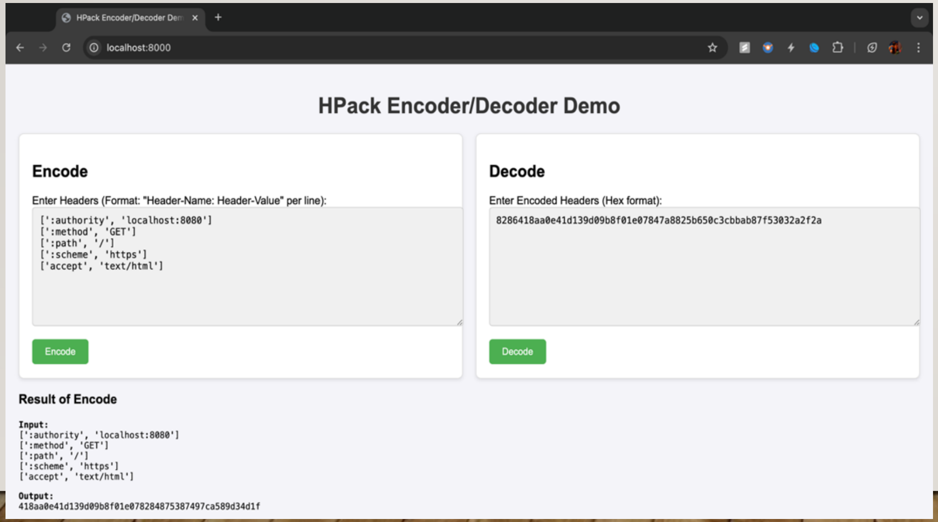
- Results showed minute variations between the connection times of both servers with the average as shown on the right.
- Screenshot shows one of the runs of the command used for measuring the connection time.

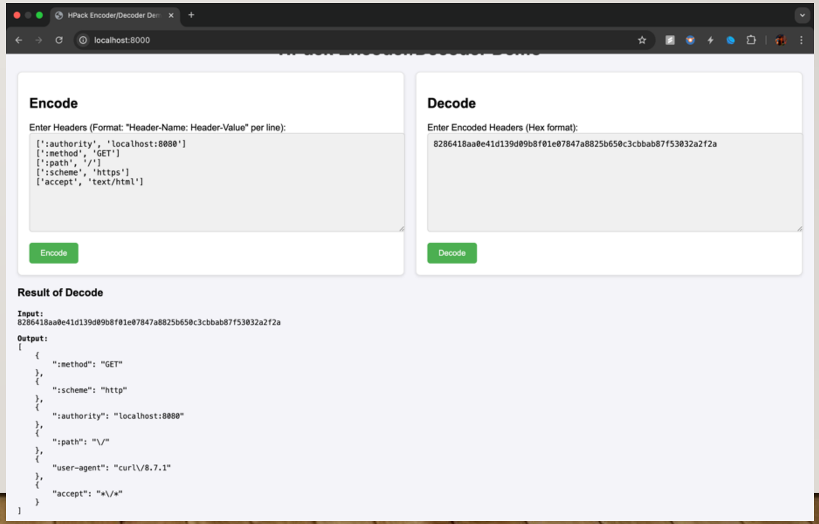
Metric	Nginx	WebSite
Connection time (avg)	0.000413	0.000387

```
ajitashrivastava@Ajitas-MacBook-Pro 01 nignxComparison % curl -o /dev/null -s -w "Connection time: %{time_connect}s\n" https://localhost/index.html

Connection time: 0.000396s
ajitashrivastava@Ajitas-MacBook-Pro 01 nignxComparison % curl -o /dev/null -s -w "Connection time: %{time_connect}s\n" https://localhost/index.html

Connection time: 0.000378s
```

EXPERIMENT - HEADER COMPRESSION

Metric	Original	Compressed
Size of sample headers 1	104 bytes	30 bytes
Size of sample headers 2	255 bytes	169 bytes
Size of sample headers 3	2339 bytes	702 bytes





EXPERIMENT – APACHE BENCHMARKING

Metric	Value
Requests Completed	999 (99.9% success rate)
Failed Requests	0
Total Time Taken	18.628 seconds
Requests per Second (mean)	53.63 requests/sec
Time per Request (mean)	186.467 ms
Transfer Rate	15.03 Kbytes/sec
Connection Times	
Minimum Connection Time	38 ms
Mean Connection Time	78 ms
Maximum Connection Time	97 ms
Median Connection Time	77 ms

- *Test Configuration*
  - Total Requests: 1000
  - Concurrency Level: 10
  - Server: localhost, port 8080
  - SSL/TLS Protocol: TLSv1.2 with ECDHE-RSA-AES128-GCM-SHA256

RESULTS

- **Installation Efficiency:** The project server outperforms Nginx in installation time, steps and space. This highlights the project's simplicity and efficiency.
- **Response Time:** Similar scores are observed. Both servers have low latency, suitable for lightweight applications.
- **Compression Performance:** Hpack implementation shows significant compression efficiency with a marked reduction in byte size (e.g., from 2339 to 702 bytes), demonstrating the server's ability to minimize data transfer for better performance.
- **Request Handling:** The project's performance closely matches Nginx's, but the fewer failed requests suggest that the implementation is reliable and suitable for high-traffic scenarios with minor tweaks.
- **Connection Times:** Connection times for both servers are within a reasonable range, with the project showing a median connection time of 77 ms, comparable to NGINX's 78 ms, indicating efficient network handling and minimal delays.

CONCLUSION

- **Efficiency**: The project proves to be a lightweight and efficient alternative to complex servers, with fast setup, competitive response times, and strong compression rates. Further performance improvements can be done but is reliable for niche applications and educational use.
- **Projects' Strengths**: The project demonstrates its potential as an alternative to traditional, resource-heavy servers like Apache and Nginx. The project's single-file, minimalist design makes it highly suitable for modern web and email traffic handling with simplicity and ease of integration.
- **Contributions and Features**: Key enhancements, including HTTP/2 support, secure handling mechanisms, have significantly improved the project's functionality, security, and efficiency.
- **Improved Communication Efficiency**: The inclusion of HTTP/2 with header compression and Huffman encoding boosted performance, reducing latency and improving overall communication efficiency.
- **Real-World Applicability**: The project confirms its ability to efficiently handle traffic while maintaining a small footprint and minimal dependencies, making it suitable for educational purposes, embedded applications, and real-world scenarios.

FUTURE WORK

- **Scalability and Expansion**: The foundation laid in this project enables future growth, allowing the server framework to scale with evolving communication demands. Such as, customized HTTP/2 frames (allowed as per RFC).
- **Enhanced Protocol Support**: Future work can include the implementation of additional modern protocols, such as QUIC or HTTP/3, to further improve performance and capabilities.
- **Optimization for Production Use**: Further optimization of resource management, load balancing, and fault tolerance to make the project suitable for high-demand production environments.
- **Educational Tools and Documentation**: Expanding the educational use aspect by creating comprehensive documentation and tools to help students and developers understand server operations.

REFERENCES

1. Apache Software Foundation. (n.d.). *Apache HTTP Server Version 2.4 Documentation*. Retrieved December 4, 2024, from https://httpd.apache.org/docs/.
2. Nginx. (n.d.). *Nginx documentation*. Retrieved December 4, 2024, from https://nginx.org/en/docs/
3. Dent, Kyle D. Postfix: The Definitive Guide: A Secure and Easy-to-Use MTA for UNIX. ("Postfix: The Definitive Guide: A Secure and Easy-to-Use MTA for UNIX ...") O'Reilly Media, Inc., 2003. Postfix Overview.
4. Dovecot. (n.d.). *Dovecot documentation*. Retrieved December 4, 2024, from https://doc.dovecot.org/
5. Hildebrandt, Ralf, and Patrick Koetter. The Book of Postfix: State-of-the-Art Message Transport. No Starch Press, 2005.
6. Belshe, M., Peon, R., & Thomson, M. (2015). *Hypertext transfer protocol version 2 (HTTP/2) (RFC 7540)*. Internet Engineering Task Force. Retrieved December 4, 2024, from https://www.rfc-editor.org/rfc/rfc7540
7. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). *Hypertext Transfer Protocol -- HTTP/1.1 (RFC 2616)*. Internet Engineering Task Force (IETF). Retrieved December 4, 2024, from https://datatracker.ietf.org/doc/html/rfc2616
8. Klensin, J. (2008). *Simple Mail Transfer Protocol (RFC 5321)*. Internet Engineering Task Force. Retrieved December 4, 2024, from https://datatracker.ietf.org/doc/html/rfc5321
9. Crispin, M. (2003). *Internet message access protocol - version 4rev1 (RFC 3501)*. Internet Engineering Task Force. Retrieved December 4, 2024, from https://datatracker.ietf.org/doc/html/rfc3501

THANK YOU!