Adapting Server Frameworks to Support HTTP/2 in Proxy Settings

Master Thesis

Junyu Pu

Dresden, 1st July 2016

Technische Universität Dresden
Fakultät Informatik
Distributed Systems Engineering
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Supervisor: Dipl.-Inf. Tenshi Hara, Dr.-Ing. Thomas Springer

TECHNISCHE
UNIVERSITÄT
DRESDEN

Declaration of Authorship

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Dresden, 1st July 2016

Contents

1 Introduction

The Hypertext Transfer Protocol (HTTP) is the foundation of data communication for the World Wide Web. Last few decades, the Internet boosted prosperously and provided thousands of contents and services. Nowadays, a web page may contain hundreds of resources and other media from different domains. However, HTTP/1.1 is outdated and designed for less complexity page decades ago and is not optimal for today's Internet. At this very point, we need a new mechanism for faster connection, smother communication and a more data efficient method to transmit data on the World Wide Web.

Google introduced SPDY protocol in 2013 and brought some breakthroughs improvements to HTTP. Since then, the Internet Engineering Task Force (IETF) introduced HTTP/2 [BPT15] in May, 2015. This first draft of HTTP/2 is using SPDY as the working based for its specification drafts and editing. It presents a feasible mechanism to fulfill the requirements of nowadays web technologies. SPDY mainly focused on improving the web page loading time and communication efficiency. The adapting works are promising, most of the Internet services and web server software already updated to support this new protocol. For example, Apache Web server and nginx web server added support for HTTP/2 with recently updates.

The HTTP/2 protocol will renovate the whole World Wide Web during the next decade. Not only the web servers and clients need to support HTTP/2, but also some underneath services need change to embrace the new standard such as video streaming, WebRTC and proxy which discusses mainly in this thesis.

A proxy server is a server that acts as a middle-ware between two ends. The ends can both be clients/servers or one is client and the other is server. The proxy server can be customized to achieve certain purpose. One particular use case is to build crowdsourcing platform.

Crowdsourcing is a new comer in the field of computer science. The basic idea is to gather information provided by the masses, then aggregate and analyze data to produce results. An easy to understand scenario is the concept of crowdsourcing translate platforms shows in Figure 1.1. It gathers translation suggestions from users for a certain word. After rating and analyzing the suggestions in crowdsourcing platform, the results submit to the server to improve the translation quality. In this scenario, a middle-ware between users and servers is responsible for caching and analyzing users' suggestion, finally submits to server.

The translate crowdsourcing platform can be considered as a special configured proxy with additional function. Server Access Network Entity (SANE) is introduced in paper [HSBS13] and uses the similar concept to improve the location accuracy. One proof of this concept is MapBiquitous [Spr11]. MapBiquitous project developed with common and standard web technologies on the server side and also on SANE. Currently, SANE only support HTTP/1.1. In order to improve performance of SANE, one obvious idea is to add HTTP/2 support.

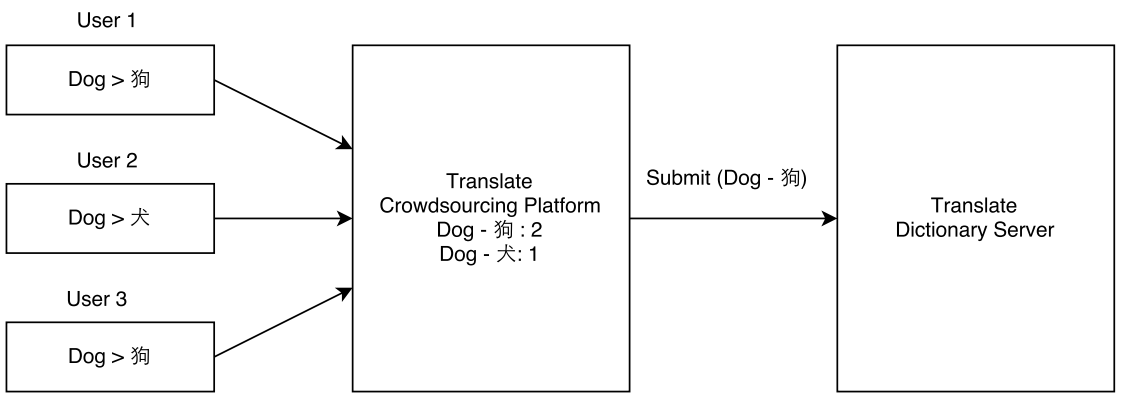


Figure 1.1: Translator Crowdsourcing Platform

In this thesis, the general idea and mechanism of building HTTP/2 supported proxy and research about how to gain best performance are discussed. This thesis also presents the mechanism to adapt exist platform (SANE) to support HTTP/2. One advantage of these methods is that it is compatible to current web technologies. Another perk of these alternatives is that upgrading can be done by adding an extra layer between the frameworks without changing other components. The two main contributions of this thesis are the performance study by introducing HTTP/2 into web proxy with different scenarios and the approach to upgrade current frameworks to support HTTP/2.

The structure of this thesis presents as following:

- In Chapter 2, background and related work are discussed;
- In Chapter 3, the concept and requirement to build a general proxy support HTTP/2 are discussed;
- In Chapter 4, the implementation details of proxy support for different scenario are presented and also the method to adapt SANE frameworks;
- Chapter 5 presents the evaluation of the concept;
- Chapter 6 includes the summarize and conclusion of this thesis, also further work are discussed.

2 Background and Related Work

2.1 HTTP/2 Protocol

The Hypertext Transfer Protocol (HTTP) is wildly used by the Internet nowadays. However, the limitation of HTTP/1.1 is drawing back the evolution of new Internet technologies and has a negative overall effect on performance.

HTTP/2 is the second major version of HTTP and based on SPDY [BP12]. It was developed by the Hypertext Transfer Protocol working group of the Internet Engineering Task Force (IETF), and its specification was published as RFC 7540 [BPT15] in May 2015. Most major browsers added HTTP/2 support by the end of 2015. Simultaneously, the web server technologies, such as Apache, nginx and Tomcat, support HTTP/2 by integrated third-party library or based on previous support for SPDY.

2.1.1 The Limitation of HTTP/1.1

HTTP/1.1 is a plain text request/response protocol which runs over a persistent TCP/IP connection and has been in used by the World Wide Web global information initiative since 1990. Separate TCP connection was established to fetch each URL every time in HTTP/1.0. Since establishing TCP connection increases load time and causes congestion, TCP connections changed to persistent connection in HTTP/1.1 which only terminates at the end of transaction. The left Figure 2.1 describes how HTTP/1.1 persistent connections works: once client and server have established a TCP connection, the client sends `GET` requests to the server for retrieving content without open and close the TCP connection again. Only after the server sends back the answers, client can send new request. The drawback of this mechanism is that messages cannot send simultaneously thus increases content exchanges latency.

After noticing the pitfall, HTTP/1.1 uses *HTTP pipelining* to improve the connection efficiency: Multiple HTTP requests are sent on a single TCP connection without waiting for responses. The answers from server are sent corresponding to the order of `GET` request. Means, latest request will be served last. In pipeline scheme, client does not need to wait for previous `GET` request to be answered. However, there is another problem raised:

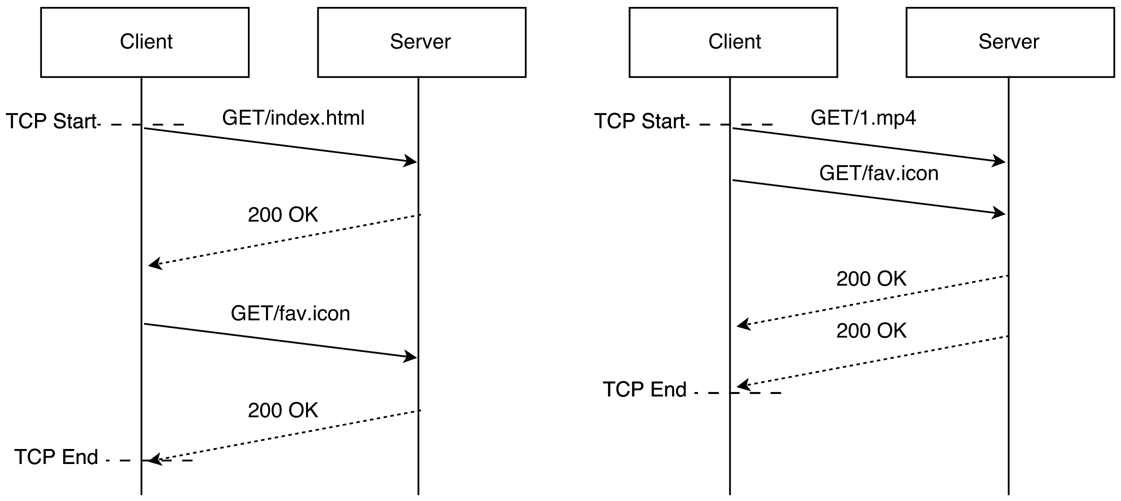


Figure 2.1: HTTP/1.1 communication Scheme. Left: non-pipelined, right: pipelined

Head-of-line Blocking. Assuming `GET/1.mp4` request in the right of Figure 2.1 represents a large file (a video for instance), then the subsequent responses will get delayed since it needs time to finish transmit a large file. As a matter of fact, HTTP/1.1 suffers from slow speed because of its original design: one request at a time [dSOC15]. IETF decided to renovate the HTTP protocol by introduce HTTP/2, one main task is to tackle the Head-of-line Blocking issue.

2.1.2 New Feature of HTTP/2

HTTP/2 protocol provides an optimized transportation mechanism for HTTP semantic. It supports all core concepts of HTTP/1.1 and adds more features to reduce the page loading time during web browsing. To achieve these goals, HTTP/2 introduces few new technologies as follows:

Binary Framing Layer. HTTP/1.1 is a text-based protocol, means the packets exchanges between client and server are human readable. Plain-text based has drawback since it costs a lot of resources during transaction compares to binary code. By contrast, HTTP/2 uses binary framing, it is more informational concentrated and computer-parse friendly. Thus, both client and server must agree this new binary encoding and decoding mechanism in order to understand each other.

Multiplexing. In HTTP/2, all communications are performed within a single TCP connection. A stream is a virtual channel within a connection, which carries bidirectional message. HTTP/2 can contain multiple streams and exchanging messages in parallel within a single TCP connection. This solves the Head-of-line blocking caused by pipeline in HTTP/1.1, since the message can be exchanged without interfering each other in HTTP/2.

Header Compression. In HTTP/1.1, the header meta-data is always sent as plain text and adds overhead per request. In most cases, HTTP headers often carry the same values, it results informational redundancy. To reduce duplication, HTTP/2 cuts the header meta-data expense by enabling HPACK [PR15] compress algorithm.

Server Push. This new feature allows HTTP/2 server to send multiple replies for a single client request. That means instead of having an additional request from client, server could push the additional resources to the client along side with other requests.

Priority. Once an HTTP message can be split into many individual frames, the exact order in which the frames are interleaved and delivered can be optimized to further improve the performance of applications. With priorities, the client and server can apply different strategies to process individual streams, messages, and frames in an optimal order.

2.2 Proxy Server

Web proxy server is a middle-ware setting between client and server [LA94]. A web proxy server can behave as both web server and client. When clients send request to the proxies, the proxy server needs to handle the requests and connections, which is similar to a web server. In the meantime, the proxy itself sends requests to servers, it behaves as a client to the destination server. Moreover, the definition made by Ari Luotonen formally descries a proxy as follows:

> A proxy is a special HTTP server that typically runs on a firewall machine. The proxy waits for a request from inside the firewall, forwards the request to the remote server outside the firewall, reads the response and then sends it back to the client.
>
> Ari Luotonen, Kevin Altis

In Figure 2.2 indicates the process of general proxy communication. Clients connect to the proxy server, and request certain service or resource, such as a file, connection, web page, or other resource available from different servers. The proxy server evaluates the request as a way to simplify and control its complexity. The server responses these requests back to clients via proxy server.

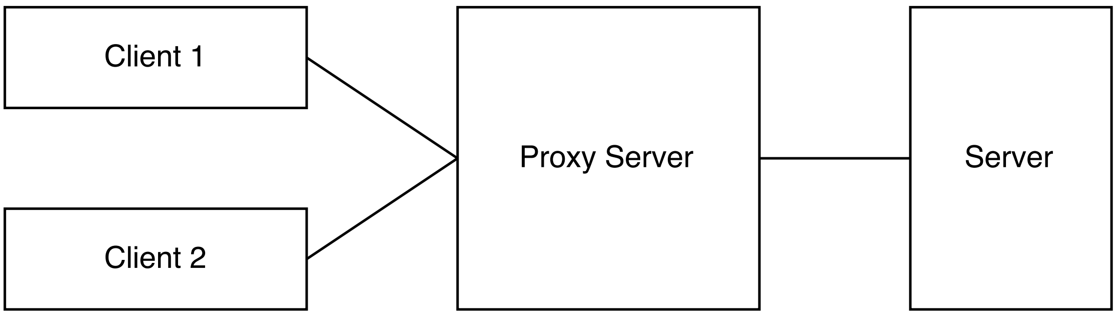


Figure 2.2: Proxy Server Architecture

2.2.1 Proxy Types

FORWARD PROXY

A forward proxy is the most common form of a proxy server and is generally used to pass requests from an isolated, private network to the Internet through a firewall. Using a forward proxy, requests from an isolated network, or intranet, can be rejected or allowed to pass through a firewall. Requests may also be fulfilled by serving from cache rather than passing through the Internet. This allows a level of network security and lessens network traffic.

A forward proxy server will first check to make sure a request is valid. If a request is not valid, or not allowed (specified by configuration file), it will reject the request resulting in the client receiving an error or a redirect. If a request is valid, a forward proxy may check if the requested information is cached. If it is, the forward proxy serves the cached information. If it is not, the request is sent through a firewall to an actual content server which serves the information to the forward proxy. The proxy, in turn, relies this information to the client and may also cache it, for future requests.

REVERSE PROXY

A reverse proxy is another common form of a proxy server and is generally used to pass requests from the Internet, through a firewall to isolated, private networks. It is used to prevent Internet clients from having direct, unmonitored access to sensitive data residing on content servers on an isolated network, or intranet. If caching is enabled, a reverse proxy can also lessen network traffic by serving cached information rather than passing all requests to actual content servers. Reverse proxy servers may also balance workload by spreading requests across a number of content servers. One advantage of using a reverse proxy is that Internet clients do not know their requests are being sent to and handled by a reverse proxy server. This allows a reverse proxy to redirect or reject requests without making Internet clients aware of the actual content server on a protected network.

A reverse proxy server will first check to make sure a request is valid. If a request is not valid, or not allowed, it will not continue to process the request resulting in the client receiving an error or a redirect. If a request is valid, a reverse proxy may check if the requested information is cached. If it is, the reverse proxy serves the cached information. If it is not, the reverse proxy will request the information from the content server and serve it to the requesting client. It also caches the information for future requests.

The main difference between reverse proxy and forward proxy is that the client does not know the traffic from which destination server directly in reverse proxy. In forward proxy, the client's request sends to the proxy, then the proxy server fetchs the resource from destination server and forwards the response to the client. In reverse proxy, the request from client can be balanced or changed to another destination server. The reverse proxy server fetches the resources based on the rules and behaved as a destination server to client.

2.2.2 Proxy Server Usage

The fundamental operation of the proxy sever is to receive client requests. After the proxy server analyzes the requests and then sends to the target servers. During this communication, proxy server can have several different functionalities [GT02]:

Security. A proxy is a special HTTP server that typically runs on a firewall machine. The primary usage [LA94] of proxy is to control accessibility to the web within a firewall. A proxy can mask the internal network from the outside, and creates isolation between two networks.

Improve Performance. A web cache proxy server can be used for the temporary storage of web documents, such as HTML pages and images, to reduce bandwidth usage, server load and perceived lag. A cache proxy server places between clients and servers. Requests send by clients can be pre-fetched by proxy server and store on proxy server. Another common usage of proxy for performance improvement is data compression. A data compression proxy compresses the original data to smaller size or excludes redundant information. By using these methods, the page load time and size can be massively reduced.

Anonymous. An anonymous proxy server can anonymize the client's web surfing information. Since the destination server receives requests from anonymous proxy server, the anonymous proxy server can create new pseudonym of client, and thus server does not gain any information of original client.

Crowdsourcing Platform. In this scenario, proxy behaves as a middle-ware between crowdfunder and crowdsourcer. In addition, crowdsourcing platform proxy can performance the function which processes the data from client then produces result for server to use.

In summary, Table 2.1 shows the proxy usage corresponding its types.

Type	Use Case Example
Forward Proxy	Content-control
	Anonymizer
	Transcoder, e.g. convert GIT images into JPEG
Reverse Proxy	Bypass, e.g. provide access to sensitive information
	Web cache

Table 2.1: Proxy Type and Usage

2.3 Data Compression Proxy

2.3.1 SPDY Protocol

In HTTP/1.1, the deliver of web pages is using multiple, persistent TCP connections for last decades. To make the Web faster, Google proposed and developed a new transport protocol for HTTP, called SPDY [BP12]. The goal of SPDY is to speed up the web by adapting several page loading time critical features:

Multiplexed Streams. This means SPDY works by opening on TCP connection per domain, then allows for unlimited concurrent streams over this single TCP connection. The efficiency of TCP is much higher since fewer network connections need to be made. Based on research [WBKW14], SPDY improves page load time significantly that track the benefits of using a single TCP connection.

Request Prioritization. SPDY allows the client to specify a priority level for each object. For example, the JavaScript or cascading style sheets (CSS) may more important than a large image for browser to present correctly web page and provide functions. SPDY will load higher priority resources first thereby preventing the connection from being congested with non-critical resources.

HTTP Header Compression. Most sites already use compression when downloading textual content as it provides a significant performance benefit. However, HTTP/1.1 does not support the HTTP header compression. SPDY uses the general purpose `DEFLATE` algorithm for header compression. This results in fewer packets and fewer bytes transmitted.

Server Push. SPDY experiments with an option for servers to push data to clients via the X-Associated-Content header. This header informs the client that the server is pushing a resource to the client before the client has asked for it. For initial-page downloads (e.g. the first time a user visits a site), this can vastly enhance the user experience.

2.3.2 Flywheel Data Compression Proxy

Flywheel [ABC+15] is an HTTP proxy service that compresses responses in-flight between origin servers and client browsers. The primary goal is to reduce mobile data usage for web traffic.

Flywheel is based on SPDY protocol. When Google start uses SPDY in 2013, there are not so many sites support SPDY. With an SPDY support proxy, we could leverage the benefits from more advanced communication between client and proxy without the needs to upgrade the entire World Wide Web.

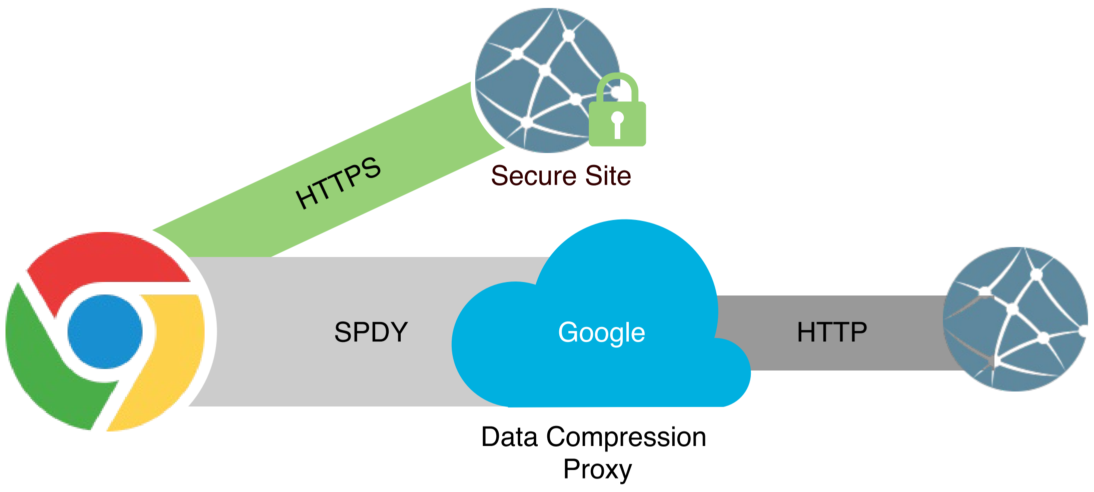


Figure 2.3: Google Flywheel (SPDY) Proxy Communication Scheme

Client Support

Similar to any other proxy, Flywheel acts as an additional middle box in the data transaction path. It creates a tunnel between the browser and the proxy, and carries the HTTP sessions within the SPDY tunnel, extending support for SPDY over the access link for all destinations when requested by a supported browser.

Google uses Flywheel proxy as data saver for Chrome browser, and optimized the data bandwidth on Google servers. When Data Server is enabled on user's browser, the *non-encrypted* HTTP connection between client and destination server is transmitted over one of Google's data compression proxy server. In Figure 2.3 shows the communication scheme of Flywheel. By default, client connections to Data Compression Proxy (synonym for Flywhell) uses the SPDY transport protocol, which is able to multiplex multiple request and response streams in parallel over a *single* TCP connection. SPDY is intended to improve the performance by using speed-related mechanisms discussed before.

The SPDY support is universal for Flywheel users. Each client application maintains a single SPDY connection to the Flywheel proxy over which multiple HTTP requests are multiplexed. Flywheel in turn translates theses ordinary HTTP transactions with origin servers.

PROXY SERVER

Google implemented Flywheel proxy servers all over the world [ABC+15]. Client connections are directed to a nearby data center using DNS-based load balancer. In Flywheel server, there are two main components: Proxy and Fetch router. The proxy behaves as filter and coordinator: the proxy will match all incoming requests against URL pattern. The proxy decides this request should induce a *Safe Browsing* warning or a proxy bypass. If the request matches the pattern, a control response will be sent. Otherwise the request is forwarded via RPC to a separate fetch serve which retrieves the resource from the origin. After fetching the resource from origin sites, the proxy makes a decision about how to compress the response based on its content type.

2.4 Server Access Network Entity

2.4.1 Crowdsourcing Platform

Crowdsourcing is a relatively recent concept in many practices. This leads to the blurring of the limits of crowdsourcing that may be identified virtually with any type of Internet-based collaborative activity, such as co-creation or user innovation. Varying definitions of crowdsourcing exist, and one definition from Arolars and Guevara is:

> Crowdsourcing is a type of participative online activity in which an individual, an institution, a non-profit organization, or company proposes to a group of individuals of varying knowledge, heterogeneity, and number, via a flexible open call, the voluntary undertaking of a task. [...] The user will receive the satisfaction of a given type of need, be it economic, social recognition, self-esteem, or the development of individual skills, while the Crowdsourcer will obtain and utilize to their advantage what the user has brought to the venture, whose form will depend on the type of activity undertaken.
>
> Enrique Estellés-Arolas, Fernando González-Ladrón-de-Guevara

MapBiquitous [Spr11] is an integrated system for indoor and outdoor location-based services which follows the concept of crowdsourcing. It is based on a decentralized infrastructure of building servers providing information about the building geometry, positioning infrastructure and navigation. Building servers are dynamically discovered and provided information is seamlessly integrated into outdoor systems by the MapBiquitous client. The participants which involved in MapBiquitous crowdsourcing service are:

- **Crowdfunder.** The person or entity requiring resources that can be provide from each individual. The MapBiquitous server which gather information from clients and provide the mapping service.
- **Crowdsourcer.** The individual who can proved information, data or knowledge that crowdfunder requires. The MapBiquitous Clients provide geometry information.

2.4.2 SANE

The Server Access Network Entities (SANE) [HSBS13] is a specified web proxy server which used in MapBiquitous. It acts as intermediaries between Crowdsourcing Clients and Crowdfunding Servers. Crowdsourcing Clients provide geography information and send to SANE for further process.

ARCHITECTURE

SANE behaves as a coordinator between crowdsourcing client and crowdsourcing server. It handles the communications and submission of multiple crowdsourcing data to server. Since client does not connect to server directly, crowdsourcing clients can gain anonymity, security and better scalability. Figure 2.4 depicts the modules set in SANE instance.

Crowdsourcing Driver. It provides interfaces for crowdsourcing client, crowdfunding server and interconnect SANE instances. It encapsulates any crowdsourced read and write-access oriented towards the server side.

Security Module. It cooperates with other module to provides encryption ability and manages signatures. The Crypto module is responsible for data encryption. The signature module creates and verify the signature from clients and servers.

Server Manager. It handles the requests related to access to a crowdfunding servers.

Client Manager. All the clients associated requests will be forward by Crowdsourcing Driver to Client Manager. For example, it will store the user model representing the accessing client.

DHT Maintainer. It maintains and organizes a set of SANE proxies to provide scalability and fault tolerance.

User Model. It stores the information related to clients, such as data provided by clients, client signature and device ID.

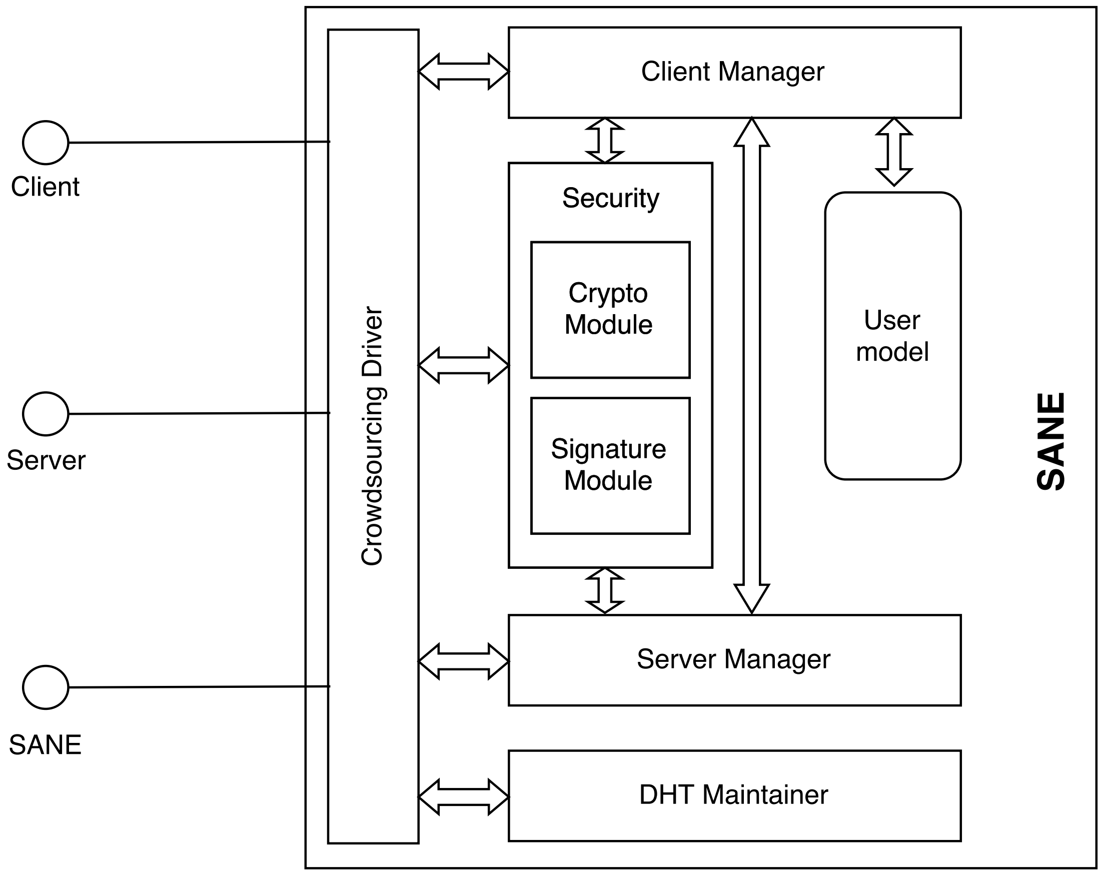


Figure 2.4: Architecture of the SANE

COMMUNICATION SCHEME

The communication in Crowdsourcing Platform only use `POST` and `GET` of standardized Hyper Text Transport Protocol version HTTP/1.1. A example shows in Figure 2.5: every crowdsourced data proposal from clients will be `POST` via HTTPS connection to SANE and attached with its credential. After SANE validates client's credential, it assigns an anonymous submission ID to client for anonymity reason. Crowdsourced data send to server via HTTPS connection with anonymous submission ID and SANE proxy credential. After the server verifies credential of SANE, it sends response to SANE and continue submission processing. SANE forwards the response to the corresponding `POST` request via HTTPS.

ANONYMITY AND SECURITY

Since the SANE functions as a proxy between clients and server, also the communication is via encrypted HTTP connection. The anonymity from outside of SANE is provided naturally. Since the SANE re-signed and re-encrypted the clients' identity, server cannot gain any traceable data from submitter. Hence clients have anonymity inside SANE.

The SANE stores the relation data between client and server which never stores on server side. The permission control is also maintained by SANE. It stores list of granted access

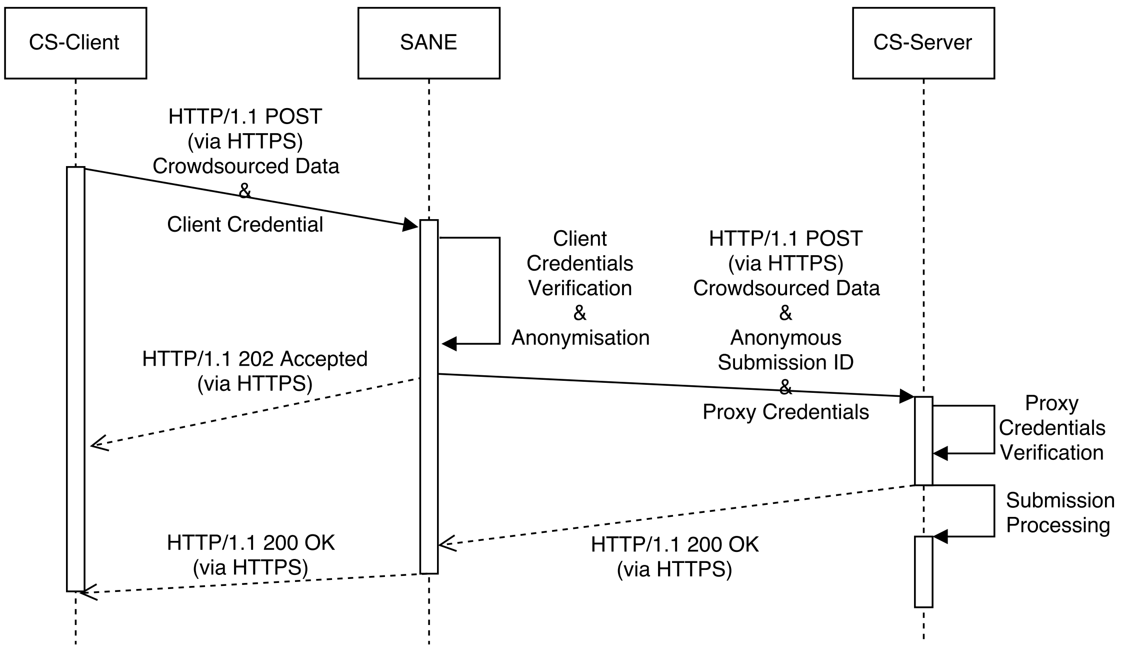

CS-Client
SANE
CS-Server
HTTP/1.1 POST (via HTTPS) Crowdsourced Data & Client Credential
Client Credentials Verification & Anonymisation
HTTP/1.1 POST (via HTTPS) Crowdsourced Data & Anonymous Submission ID & Proxy Credentials
HTTP/1.1 202 Accepted (via HTTPS)
Proxy Credentials Verification
Submission Processing
HTTP/1.1 200 OK (via HTTPS)
HTTP/1.1 200 OK (via HTTPS)

Figure 2.5: Communication in Crowdsourcing Platform

rights to enable any client to gather information on the granted rights centrally.

SCALABILITY AND FAULT TOLERANCE

A set of distributed SANEs formed a scalable, fault tolerated and decentralized proxy services. SANEs use a distributed hash table (DHT) to assign and tolerate the requests. Each SANE is recommended to have one neighbor as backup for each other, thus for each client or server has 3 SANEs to handle their requests at least.

SANE is a distributed, anonymous, scalable, fault tolerated proxy server for crowdsourcing platform. It serves not only for certain client or server, but provides proxy service for any device during crowdsourcing process. Since any proxy is an additional middle-ware between both sides, the performance and latency is big issue for such communication. The SANE only supports HTTP/1.1 right now, it may benefit from the HTTP/2.

2.5 Summary

The background of HTTP/2 is presented in this chapter, and also introduced the proxy server in general. The related works include Flywheel Data Compression Proxy and SANE. The former brings the basic idea of how to construct a proxy with HTTP/2 support. Meanwhile, SANE is one of the test bases of this thesis and the background is provide in this chapter for reader who does not familiar with.

3 Concept

This chapter will discuss the ideas about how to adapt the HTTP/2 in current web server frameworks. It starts with the prerequisites before upgrading to support HTTP/2. Follow the workflow is to upgrade the server to support HTTP/2, and also the client side. The general architectures of proxy are discussed next. Finally, this chapter will bring the concept of adapting SANE to support HTTP/2.

3.1 Before Upgrade to HTTP/2

3.1.1 TLS and ALPN

Transport Layer Security (TLS) is cryptographic protocols designed to provide communications security and privacy over networks. The protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol.

The *TLS Record Protocol* is responsible for securing application data and verifying its integrity and origin. It manages the following:

- Dividing outgoing messages into manageable blocks, and reassembling incoming messages. Compressing outgoing blocks and decompressing incoming blocks;
- Applying a Message Authentication Code (MAC) to outgoing messages, and verifying incoming messages using the MAC;
- Encrypting outgoing messages and decrypting incoming messages;
- When the *Record Protocol* is complete, the outgoing encrypted data is passed down to the Transmission Control Protocol (TCP) layer for transport.

The *TLS Handshake Protocol* is responsible for the authentication and key exchange necessary to establish or resume secure sessions. When establishing a secure session, the Handshake Protocol manages the following:

- Cipher suite negotiation;
- Authentication of the server and optionally, the client;

- Session key information exchange.

HTTPS uses TLS for communication secure and data integrity. It consists of communication over HTTP within a TLS encrypted connection. HTTP/2 does not require connection via a TLS connection compulsorily. However, current shipping browsers *only* support HTTP/2, if server support for HTTP/2 is enabled and TLS is in use.

The client and the server may support multiple protocols for HTTP connection, and they need to agree on one of the protocols to use. RFC 7301 [FLP14] describes *Application-Layer Protocol Negotiation (ALPN)* to solve above problem.

The ALPN is a TLS extension stands for application-layer protocol negotiation. It allows the application-layer to negotiate which protocol should be performed over a secure connection in a manner which avoids additional round trips. By using ALPN, the client sends which application protocols it supports to the server, ordered by priority. The server selects the protocol to use which depends on the protocols it supports and the client's priority. Listing 3.1 shows an example of protocol negotiation between server and client.

```
* Connected to 46.101.99.160 port 443
* ALPN, offering h2
* ALPN, offering http/1.1
* Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL
  :!LOW:!RC4:@STRENGTH
* successfully set certificate verify locations:
* CAfile: /etc/ssl/certs/ca-certificates.crt
* CApath: none
* TLSv1.2 (OUT), TLS header, Certificate Status (22):
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
```

Listing 3.1: ALPN Negotiation Process Example

3.1.2 HTTP Request/Response

HTTP is a request-response protocol between a client and server. There are two types of HTTP messages, request messages and response messages. For example, a web browser may use `GET` method to request data from a specified resource, and an application can use `POST` method to submit the data of specified URI.

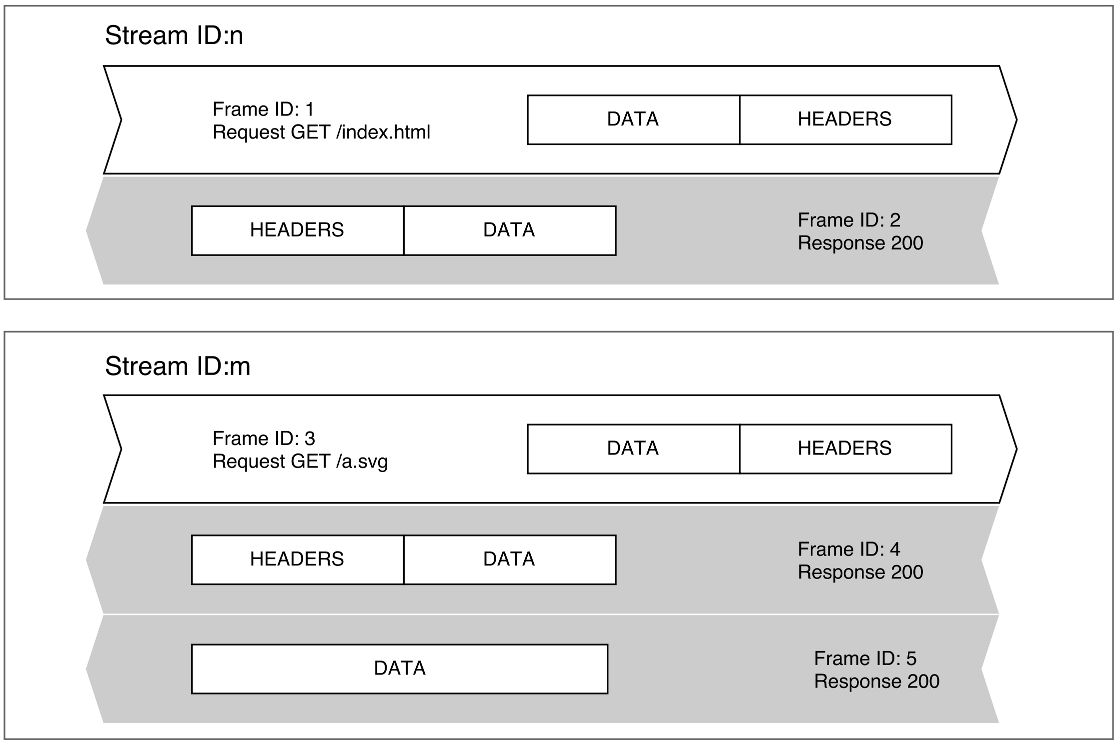


Figure 3.1: HTTP/2 Request/Response Frame Structure and Example

In HTTP/1.1, a request message from a client to a server includes: the method needs to be applied to the resource, the identifier of the resource, and the protocol version in use. The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with `CR` and `LF` [FGM+06]. The request header fields allow the client to pass additional information about the request. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method invocation.

In HTTP/2, the structure of HTTP messages are changed to frames: a client sends each HTTP request starts with a new stream; a server replies the HTTP response on the same stream as the request. An example can be found in Figure 3.1: request `GET/index.html` sends to server with `steam ID: n`. Server responses this request by setting the same steam ID (`n`) then sends header and data in `Frame ID:2`. Another request sends to server will initiate a new steam ID (`m`), which is different from previous one. This allows client and server transferring data multiplexed and concurrently without interfering each other.

This change means the communication process and frame of raw data information, and previous version of HTTP connection header is not suitable for HTTP/2. Developing a client to support HTTP/2 header requires more efforts because the binary nature and HPACK algorithm in HTTP/2 protocol.

3.1.3 Header Compression

HPACK [PR15] is a specified compression format for efficiently representing HTTP header fields in HTTP/2. HPACK is defined in RFC 7541 [PR15] and designed to be simple and inflexible intentionally. Once HTTP/2 enabled, the header compression takes benefits automatically.

STRUCTURE OF HTTP/2 HEADER

HPACK uses two tables for associating header fields to indexes. The *Static Table* is predefined and contains common header fields. The *Dynamic Table* is dynamic and can be used by the encoder to index header fields repeated in the encode header lists. These two tables are combined into a single address space for defining index values.

Static Table. It consists of a predefined and unchangeable list of header fields. The static table contains most frequent header fields used by popular web sites, with the addition of HTTP/2-specific pseudo-header fields. Other undefined header fields will be added to dynamic table.

Dynamic Table. The *Dynamic Table* consists of a list of header fields maintained in first-in, first-out order. The first and newest entry in a dynamic table is at the lowest index, and the oldest entry of a dynamic table is at the highest index. The encoder maintains the dynamic table and control behave of adjustment. By using dynamic table, encoder can omit duplicated header fields which requested before.

ENCODE PROCESS

An encoded header field can be represented either as an index or as a literal. For example, an encode code `82` will be decode as `:method: GET`. For string literal, it represents with the *Huffman Code*. Figure3.2 shows HPACK in practical example. In the first request`:method:GET, :path:/, :scheme:https:` has already defined, only need encode as index to: `2,4,7`. Since `:accept-encoding:` has customized value, the value needs to be encode with Huffman code and added into dynamic table. The second request has slightly different from the first request. Since the customized value of `:accept-encoding:` has already been added in dynamic table, there is no need to encode it again.

DECODE PROCESS

A decoder processes a header block sequentially to reconstruct the original header list. The order of header field is corresponding to encoded header order. In order to successfully decode a header, there are few addition rules to follow based on [PR15] Section 3.2. After finishing the header representing, the decoder passes the resulting header fields to the application.

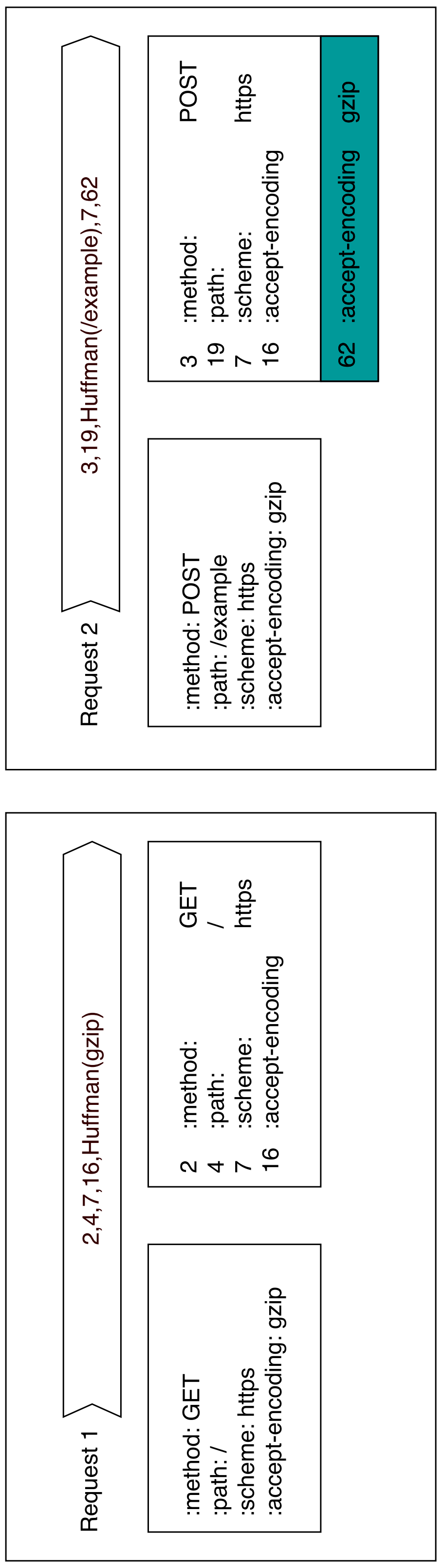


Figure 3.2: HTTP/2 Request/Response Header Compression

3.1.4 Server Push

Server push allows the server to preemptively send multiple responses for a single client request. All server push streams are initiated via `PUSH_PROMIS` frames and notify the client that the server intend to sends accompanying resources with this request. Client can decide to accept or decline push from server.

The most frequent use case is sending resources such as images, CSS and JavaScript along with the page that includes them. For example, to optimize the web page display result, in-line resource are normally used in HTTP/1.1. However, this in-line resource are force pushed to client. The client has no options to decline or control resource individually. In contrast, using server push can give client a fine granularity control on each resource it requested.

Server push feature benefits most from client-side when server needs to send multiple resource along with original request. However, most of the communications between two participants in SANE is finalized with one request and one reply [HSBS13]. The performance improvement gains from server push is limited, thus the evaluation will be focused on header compression and only the implementation is represented in next chapter.

3.2 Server and Client Support

3.2.1 Server Support

The Internet is built by thousands of web servers. Based on the Netcarft's [net16] Web Server Survey, nearly 90% of web server market shared by Apache, nginx and Microsoft. As in February 2016, 45.65% of top million busiest sites are using Apache, by contrast 24.86% of them use nginx. To adapt current widely used web server to support HTTP/2 is a better solution than creating a new web server software. The methods and obstacles of adapting Apache and nginx will be discussed in this section.

APACHE WEB SERVER SUPPORT

Apache is one of the most used web server software. It supports a variety of features, has many implemented features as compiled modules which extend the core functionality. These can range from server-side programming language support to authentication schemes. Some common language interfaces are supported in Apache such as Perl, Python, and PHP.

Apache supports HTTP/2 since version 2.3.14 by implemented the `mod_http2` module. This module relies on `nghttp2` [Tsu15] which is an open source C library implementation of HTTP/2. This module will be default enabled and as a core module after Apache 2.4.17. Listing 3.2 shows the example of enable HTTP/2 on Apache test web server.

```
<IfModule mod_ssl.c>
<VirtualHost *:443>
Protocols h2 http/1.1
SSLCertificateFile /path/to/certfile
```

Listing 3.2: Enable HTTP/2 on Apache Web Server

As mentioned before, client and server need negotiation protocols by ALPN. We need to use 443 port for HTTPS connection and indicate the path to certificate and key be used.

nginx Web Server Support

nginx is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. nginx is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption. Since nginx 1.9.5 [Mem15] a new `ngx_http_v2_module` introduced and replaced the former SPDY support module. The newest version provides support for HTTP/2 and includes some features like HPACK head compression and multiplexing.

To enjoy the perks of HTTP/2 on nginx server, only needs change the configure file. The nginx web server supports two HTTP/2 protocol same configuration syntax in Apache: h2 and h2c. `h2` is a identifier for HTTP/2 over TLS, `h2c` stands for using HTTP/2 over pain text TCP.

nginx web server provides two negotiation methods, ALPN and NPN. Next Protocol Negotiation (NPN) is a predecessor of ALPN which be used for negotiation SPDY. nginx support NPN for legacy reason, since nginx HTTP/2 support derives on SPDY and limitation of old OpenSSL version.

```
server {
  listen 443 ssl http2;
  ssl_certificate /path/to/certfile;
  ssl_certificate_key /path/to/keyfile;
}
```

Listing 3.3: Enable HTTP/2 on nginx Web Server

3.2.2 Client Support

The client benefits more performance improvements from HTTP/2 than web server. Not only the on shipping browsers but also the mobile platforms have support HTTP/2 since last few months. The web browser, such as Firefox, Chrome, Opera and Safari, supports HTTP/2 as default and does not need users to configure. Safari for iOS after iOS 9.1 native support HTTP/2. By contrast, Chrome for Android after version 46 supports HTTP/2 as default. For some system leveled web tools, such as curl, also support HTTP/2 by integrating third-party library.

Browsers Support

Since the announce of HTTP/2, most of the browsers software both on desktop and mobile operating system are adapted to support HTTP/2. The full list of HTTP/2 supported browsers from research of [can15] shows in Table 3.1. By default, all listed browsers only support HTTP/2 over TLS.

IE	Edge	Firefox	Chrome	Safari	Opera	iOS for Safari	Chrome for Android
11	13	43	45	9	35	9.2	49

Table 3.1: The Versions of HTTP/2 Supported Browsers

iOS Support

iOS is a mobile operating system created and developed by Apple Inc. and distributed exclusively on Apple hardware. It allows developer to create apps based on this operating system. Since iOS 9 launched in September 2015, it has native support for HTTP/2 and provide network API for HTTP/2 connection.

The NSURLSession [nsu13] class natively supports the data, file, ftp, http, and https URL schemes, with transparent support for proxy servers and SOCKS gateways, as configured in the user's system preferences. Adapting the client on iOS to support HTTP/2 could be achieved by using NSURLSession API.

Android Support

Android is a mobile operating system developed by Google, based on the Linux kernel and designed primarily for touchscreen mobile devices such as smart phones and tablets. Since the HTTP/2 specification derives from Google's SPDY protocol, the support for HTTP/2 is introduce in very early stage.Android HTTP/2 request support can be achieved through third-party frameworks, such as OKHTTP[1] for Android.

System Library Support

For the application on web server, the HTTP request and response are handled via system library. `libcurl` [cur16] is a client-side URL transfer library, recently add the support for HTTP/2 request and response. The `libcurl` library is portable and can be integrated into many web server friendly scripts, such as PHP. It also provides a command line tool to debug the HTTP/2 communication.

[1]http://square.github.io/okhttp/

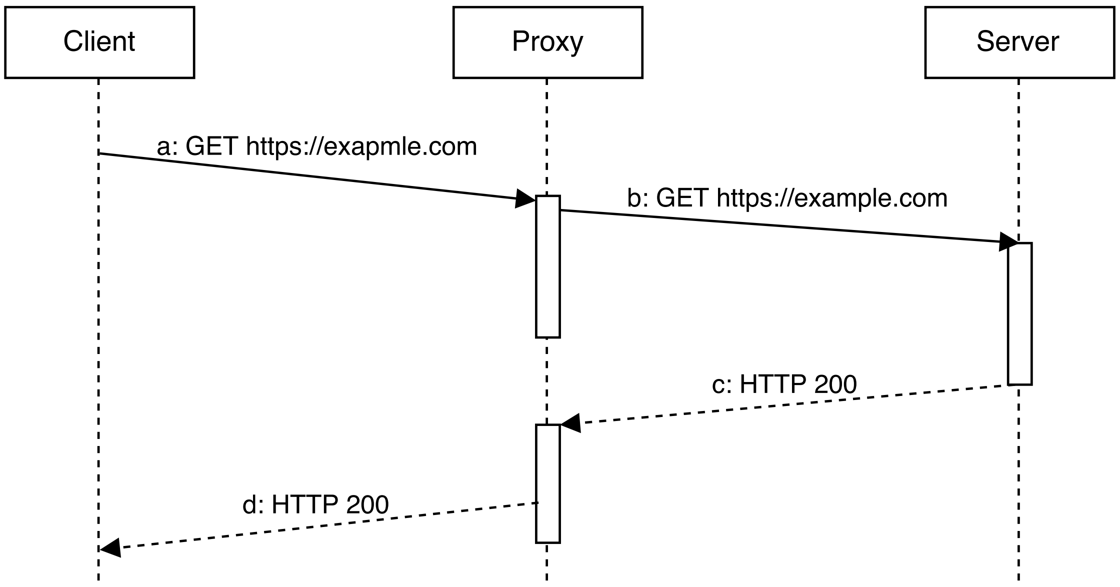


Figure 3.3: Proxy Communication with HTTP Scheme

3.3 Proxy Server Support

Web proxy server is a middle-ware setting between client and server, and the proxy has to maintain two roles to cooperate the information transmission between client and server. For client side, the proxy is a web server. It handles the requests and send the fetched resource back to the client. From server perspective, the proxy behaves as a client. It forward the request from the client, sends to the server and caches the response.

In figure 3.3 shows an example of proxy communication. The client sends `GET` request to proxy server, dedicates as `a`. The proxy server behaves as a web server for client, this request is inside proxy server and forwards to destination server, shows as connection `b`. After the destination processes by server (connection `c`), the proxy sends the responses back to client (connection `d`). In summary, proxy server behaves as a web server in connection `a` and `d`; at the same time proxy server behaves as a client in connection `b` and `c`.

The idea of adapting a proxy to fully support HTTP/2 is forward: the proxy should upgrade the communication from HTTP/1.1 to HTTP/2 on all the connections mentioned above.

There are also substitute solutions for HTTP/2 support in proxy setting by compromises some connections still maintain in HTTP/1.1. We call a proxy is cross-protocol proxy when HTTP/2 and HTTP/1.1 are both used at the same time in a proxy. More details will be discussed in the next section.

3.4 Implement Feasible Proxy with HTTP/2 Setting

3.4.1 General Architecture

Despite the fact that many web server companies are developing their products to adapt HTTP/2 standard, it still needs a long time to dominate the whole Internet. During this phase, here are three architectures to implement proxy server with HTTP/2. Since HTTP/2 maintains high-level compatibility with HTTP/1.1, we assume that if a client or server supports HTTP/2, it also works fine with HTTP/1.1. We also omit the situation of client and server both only supports HTTP/1.1, since it not the purpose of this thesis. After analyzing all the possible combination of HTTP/2 and HTTP/1.1 used in proxy communication, the architecture of proxy communication with HTTP/2 represents in Table 3.2.

Client	Proxy	Server
Support HTTP/1.1	HTTP/1.1 Proxy Server with HTTP/2 Back-end support	Support HTTP/2
Support HTTP/2	HTTP/2 Proxy Server	Support HTTP/2
Support HTTP/2	HTTP/2 Proxy Server with HTTP/1.1 Back-end Support	Support HTTP/1.1

Table 3.2: General Architecture with HTTP/2 Proxy Setting

Upgrade Proxy. When a proxy between a client which supports only HTTP/1.1 and destination server supports HTTP/2, we call this proxy is an *Upgrade Proxy*. One function of this type of proxy is to translation the former HTTP/1.1 protocol to newer HTTP/2 protocol. It can bring most benefits to server side, since the client cannot support HTTP/2, the performance improvement is not significant when proxy and server are tightly coupled. In order to build the bridge between client and server, Upgrade Proxy has two main components. First component is to handle the requests from client-side, second component is responsible for translating the requests from HTTP/1.1 to HTTP/2.

Straightforward Proxy. When client and server are both support HTTP/2, the architecture of the proxy setting is less complex. The proxy behaves as a middle-ware and without to implement other translation for HTTP requests. In this setting, both client and server can benefit. However, this scenario is rare for nowadays Internet environment.

Downgrade Proxy. We call a proxy forward HTTP/2 requests from client to server which only support HTTP/1.1 is *Downgrade Proxy* when the communication between client and proxy is using HTTP/2. By using this setting, server does not need to upgrade and can maintain the same and client may gain most profits.

3.4.2 Requirements of HTTP/2 Proxy

To build a proxy with HTTP/2 protocol support, there are few common components need to be integrated into the current server frameworks. At the meantime, there are few difference considering the proxy architecture variation.

COMMON COMPONENTS

The following components are needed to form a proxy which involved different HTTP protocol within the communication:

Header Translator Module. The header in HTTP/1.1 requests is totally different from HTTP/2. As mentioned in Section 3.1, HTTP/2 header requires HPACK compression, different syntax for variables, and most importantly its binary data rather than plain text in HTTP/1.1. In order to communicate in a cross-protocol proxy, a module to translate the header from HTTP/1.1 to HTTP/2 is needed.

HTTP Proxy Module. After the request translated by *Header Translator Module*, it will be forwarded by HTTP Proxy Module to the destination server. The HTTP Proxy Module gets the resources from web server similar to a cache normal proxy.

DIFFERENT ARCHITECTURE

The common components is need when building a cross-protocol proxy. However, since the architecture variation discussed in Section 3.4.1. We need to configure the building blocks for different type of proxies.

Requirements of Upgrade Proxy

The Upgrade Proxy can support general HTTP request and can improve the performance by enabling HTTP/2 support between server and proxy. The top Figure 3.4 demonstrates the architecture of Upgrade Proxy. In this type of proxy, we do not need the client to support HTTP/2. However, the proxy server and destination server must upgrade to support HTTP/2.

Requirements of Straightforward Proxy

Straightforward Proxy will be the ideal and final proxy type in next decade. The major proxy server software companies and organizations are adapting their products to support this architecture. However, all the available products are in beta version, we need to build this from scratch.

The architecture of Straightforward Proxy is simply like the normal proxy tool which already existed in HTTP/1.1. The idea of to build a proxy support both back and end proxy is to build a proxy server on application level.

Application proxy is easy to maintain and can also gain benefit from the HTTP/2. The transport layer of the communication handled by HTTP/2 server. In side the server, appli-

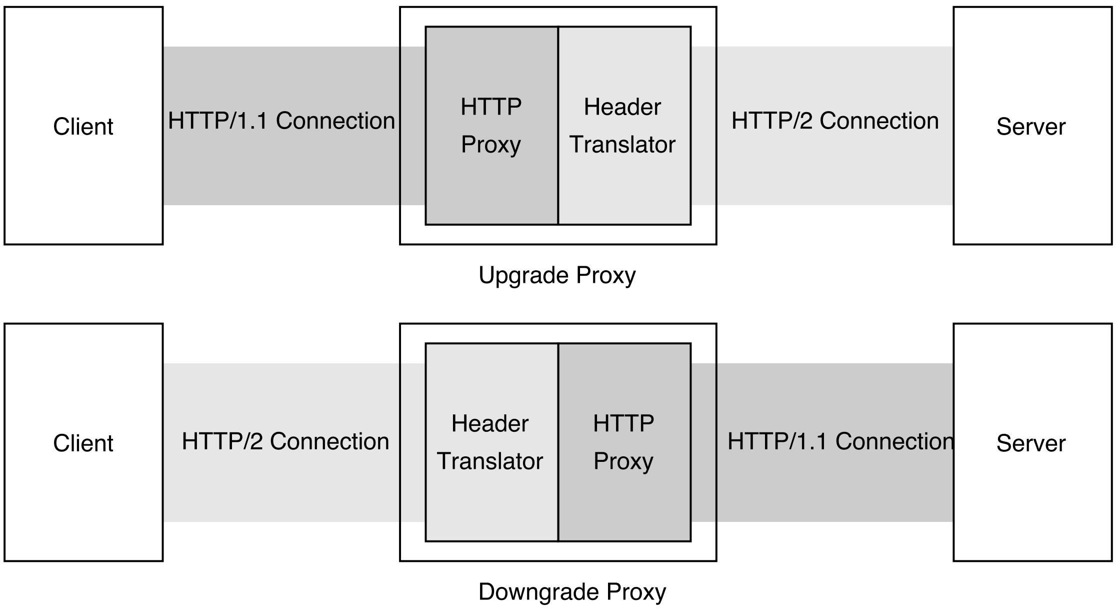


Figure 3.4: Upgrade Server Framework to Support HTTP/2.

cation communicates through CGI (Common Gateway Interface).

In this architecture, proxy application is considered as a client of destination server which forward and control client's requests. So the proxy application should be also support HTTP/2.

Requirements of Downgrade Proxy

A Downgrade Proxy suits majority situation for nowadays Internet environment. Google's Flywheel Data Compression Proxy, it proves that by using this architecture, clients can improve 50% performance of page loading time. Google's Flywheel also employs some addition features to reduce the data consumption. The general concept of this type of proxy is: using proxy server to handle the HTTP/2 requests between client and proxy server. The back-end connection between proxy and server is maintain the same.

One idea to adapt current server and client setting is by adding a new layer to the current proxy server. The implementation is similar to Upgrade Proxy. Instead adding Header Translator Module behind the HTTP Proxy Module, we need to implement it in front HTTP Proxy Module to enable HTTP/2 connection between client and proxy server.

3.4.3 Additional Features Support

General proxy programs can add extra features to support more specific purpose. These features can enhance the user experience of proxy and help clients to save data or achieve goals. The following lists few features are frequently added to a proxy:

Cache. Cache on proxy server can store frequently requested resources. For example, proxy stores the picture in cache after first client requested an image from destination server. When later same requests arrive, the same image sends to client without addition fetching from destination server.

Data Reduction. The response from the fetch server can be compressed on proxy server. Since images consume huge amount of data traffic, trans-coding the image to smaller type can dramatically improve the page loading time. Another benefit in Downgrade Proxy is header compress which may not notable. For responses, header data is often much smaller than the accompanying data, so the benefit of header compression is small. HTML, CSS and JavaScript can be compressed too by GZip.

Anonymity. Proxy server use pseudo identifier of client to communicate with server to anonymized client's sensitive data. For example, client's IP address can be covered by proxy server. Proxy can modify and replace client's cookies which can be used as track information for most of online services.

3.4.4 Adapt Squid Cache Framework to SPDY alike Proxy

Squid[2] is a popular caching proxy and used by hundreds of Internet providers world-wide to bring the best web experience. It is an open sourced program and has great customization abilities. Currently, it is not support HTTP/2 protocol. The reason to adapt Squid to support HTTP/2 is a measurable method to compare the beneficent of HTTP/2 in proxy.

The solution is to form a Downgrade Proxy by adding Header Translator Module in front of Squid cache proxy. The Header Translator Module can upgrade the HTTP/1.1 requests and responses to HTTP/2 for communication between proxy and server. One thing need to be noticed is that Squid supports non-encrypted HTTP connection by default. However, the HTTPS connection needs SSL injection. Since we want to build a SPDY alike proxy, and only study the performance improvement from HTTP/2. HTTPS proxy does not need to be supported. SPDY proxy filters the HTTPS connection from normal HTTP communication, this means we do not need to implement the SSL injection in Squid.

The architecture of a SPDY alike proxy is similar to the Figure 3.4: *Header Translator* is responded for converting HTTP/1.1 to HTTP/2; *HTTP Proxy* replaced by Squid cache proxy, it handles the requests from Header Translator and cache the request from client.

[2]http://www.squid-cache.org

3.5 Adapt SANE to Support HTTP/2

SANE [HSBS13] behaves as a proxy server between Crowdsourcing Client and Crowdsourcing Server. The general architecture of Crowdsourcing Platform with SANE setting is given in Figure 3.5. Current version of Crowdsourcing Platform is still using HTTP/1.1, one goal of this thesis is to adapt SANE to support HTTP/2 in order to gain a better performance.

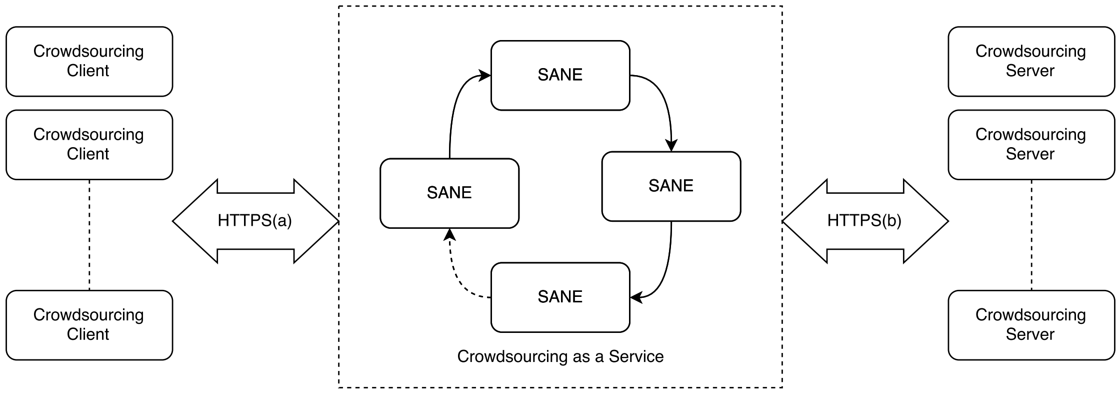


Figure 3.5: Architecture of the Crowdsourcing Platform

One specific project of using SANE is MapBiquitous [Spr11], a project developed in TU Dresden. MapBiquitous is an integrated system for indoor and outdoor location-based services. It is based on a decentralized infrastructure of building servers providing information about the building geometry, positioning infrastructure and navigation. Building servers are dynamically discovered and provided information is seamlessly integrated into outdoor systems by the MapBiquitous client.

In this section, the methods of upgrading MapBiquitous will be presented. Since the main focus of this thesis is on the HTTP/2 protocol. The adapting concept of other components in MapBiquitous is listed here for reference and implementation will be omitted.

3.5.1 MapBiquitous Framework Support

MAPBIQUITOUS CLIENT SUPPORT

MapBiquitous has implemented two different clients. Desktop client is based on Java Standard Edition. For client to support HTTP/2, as we discussed before, we need to upgrade the request protocol. One solution is using Jetty [jet16]. Jetty Java library supports HTTP/2 recently. By using `org.eclipse.jetty.http2.client.HTTP2Client` package we could enable HTTP/2 support seamlessly.

On mobile platform, MapBiquitous has an Android client. The method is mentioned in Section 3.2.2.

MapBiquitous Server Support

The functionality of MapBiquitous Server has been implemented based on Apache Tomcat, MySQL and JSP. HTTP requests are processed by a JSP servlet, which parses the XML requests, extracts the geo-window, queries the database to get all available building servers in that area and generates the response messages.

With the Tomcat 8.5.0 release [tom16], one big change of this version is added support for HTTP/2 and TLS virtual hosting. Since JSP servlet is on the Tomcat, the response is handled for HTTP/2 connection by Tomcat automatically.

3.5.2 Concept of SANE Support

The adaption of SANE separate into two main parts. Firstly, we need to allow SANE support HTTP/2 when it communicates with client. This relies on underneath HTTP server support. From the [HSBS13] we know that SANE is an PHP application and running on Apache. To run PHP scripts Apache HTTP server has two widely used methods. One is using Common Gateway Interface (CGI) and another method is running inside Apache HTTP server as a extend module (`mod_php`). We need to investigate the idea that PHP running on HTTP/2 enable Apache server also can support HTTP/2 communication. Secondly, we need SANE to be a HTTP/2 supported client when it communicates with the destination server or other SANEs.

Apache PHP Module

Using `mod_php`[3] to execute PHP scripts on a web server is one of the most popular methods in nowadays web server.

When using `mod_php` the PHP interpreter is embedded in each Apache process that's spawned on the server. This way every Apache worker is able to handle and execute PHP scripts itself removing the need to deal with any external processes; unlike CGI or FastCGI. This makes it very useful for sites that are PHP heavy where lots of requests are likely to contain PHP code (such as WordPress, Drupal, Joomla, etc.) because all the requests can be handled by Apache.

As the interpreter is started along with Apache, it allows it to run very quickly as it can cache certain information and does not need to repeat the same tasks each time a script is executed.

The downside to this is that the footprint for each Apache process is larger as it requires

[3]https://wiki.apache.org/httpd/PHP-FPM

more system resources with the PHP interpreter embedded. Even when serving static content such as images, text and style sheets where no PHP code needs to be executed, the process still contains the PHP interpreter.

Since the HTTP/2 handled by another module (`mod_http2`) using the same mechanism as `mod_php`. After the requests handled and translate by `mod_http2`, the `mod_php` execute the PHP scripts and send the response through `mod_http2` back to client. Figure 3.6 shows the process of HTTP/2 communication with PHP support.

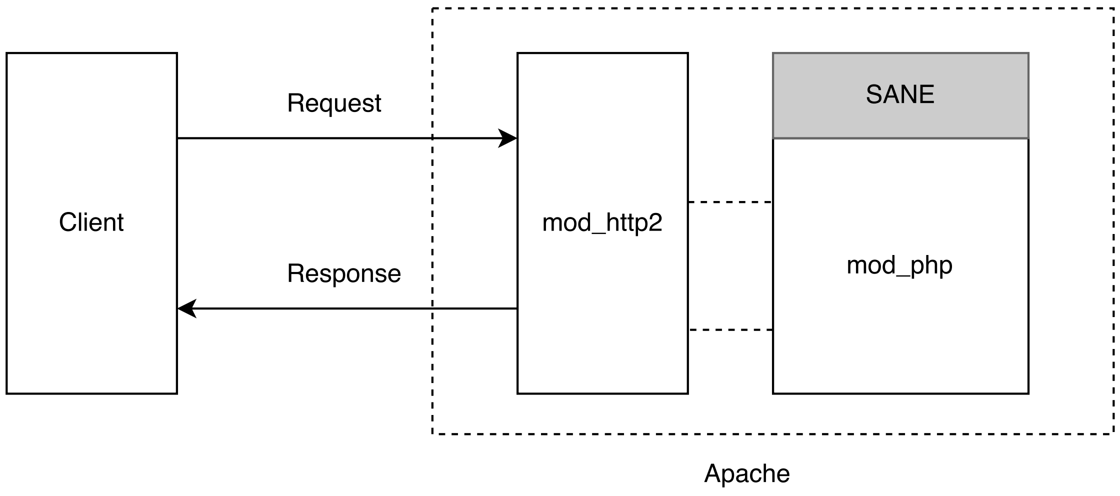


Figure 3.6: PHP Application Executed with mod_php

Apache FastCGI

When executing PHP scripts with FastCGI each request is passed from the web server to FastCGI via a communication socket. To serve an incoming request, the web server sends environment information and the page request itself to a FastCGI process over a socket or TCP connection. After PHP scripts executed, responses are returned from the process to the web server over the same connection, and the web server subsequently delivers that response to the end user. The connection may be closed at the end of a response, but both the web server and the FastCGI service processes persist. In Figure 3.7 shows the executing process of PHP working with FastCGI.

Since the requests and responses are handled by Apache server by using the `mod_http2` module, either choose FastCGI or `mod_php` can lead to the same result of HTTP/2 support.

Communication

Request and response from outside of SANE are handled by *Crowdsourcing Driver*. The requests origin has three different sources: Crowdsourcing Client, Crowdsourcing Server and DHT linked SANEs. In Figure 3.8 shows request from client indicates as (a); request from server indicates as (b) and communication with other SANEs indicates as (c).

Connection (a). This connection is between Crowdsourcing Client and SANE. Without

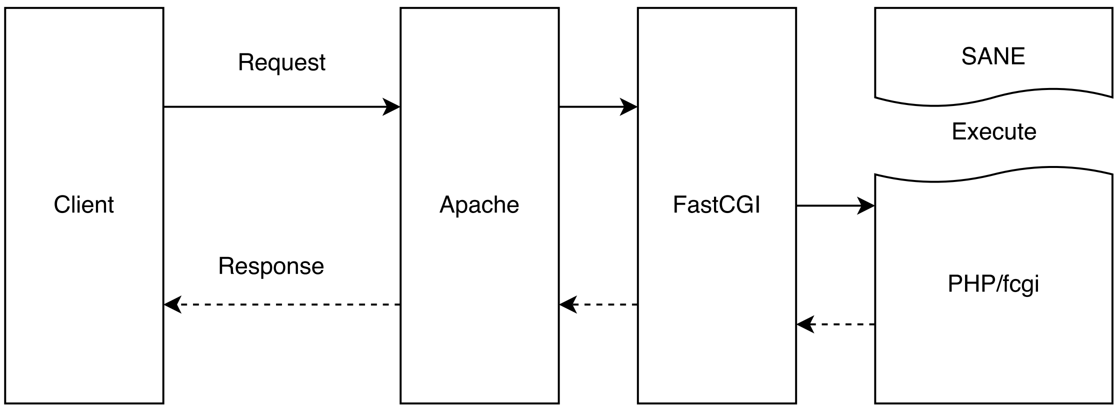


Figure 3.7: PHP Application Communicate with Apache Server Though FastCGI.

considering other associations, SANE behaves as a server for client. Client's requests include GET and POST. To upgrade to support HTTP/2, we only need upgrade Apache server for HTTP communication. Another way is to translate client's HTTP/2 request to HTTP/1.1 request, by this way, we just need add a new layer to the current SANE server.

Connection (b). It supports the communication between SANE and Server. Without other connection, SANE behaves as a client of server. To build a Straightforward Proxy, the HTTP requests send from SANE to Server need to be update. To build a Upgrade Proxy, the Header Translator Module should be added between client and server or adapt the current request to support HTTP/2 natively.

Connection (c). This is connection between SANEs. The situation and condition are similar as Connection (b), because it acts as client to other SANEs.

3.6 Summary

The adapting of HTTP/2 is promising even it only announced a year ago. Most of clients and server frameworks are already embraced this new protocol.

In this chapter, the research and concept for HTTP/2 support are presented. At the same time, the challenges for HTTP/2 supporting are listed in this chapter.

The methodologies of HTTP/2 supporting for current infrastructures are straightforward and easy. In order to adapt the current frameworks to support HTTP/2, this chapter came up three different architectures of HTTP/2 involved proxy. SPDY alike proxy introduced in this chapter follows the idea of adding translation layer in front of a cache proxy. It achieves the idea of enabling HTTP/2 connection between client and server, and also overcomes the trouble that Squid does not support HTTP/2 at this moment.

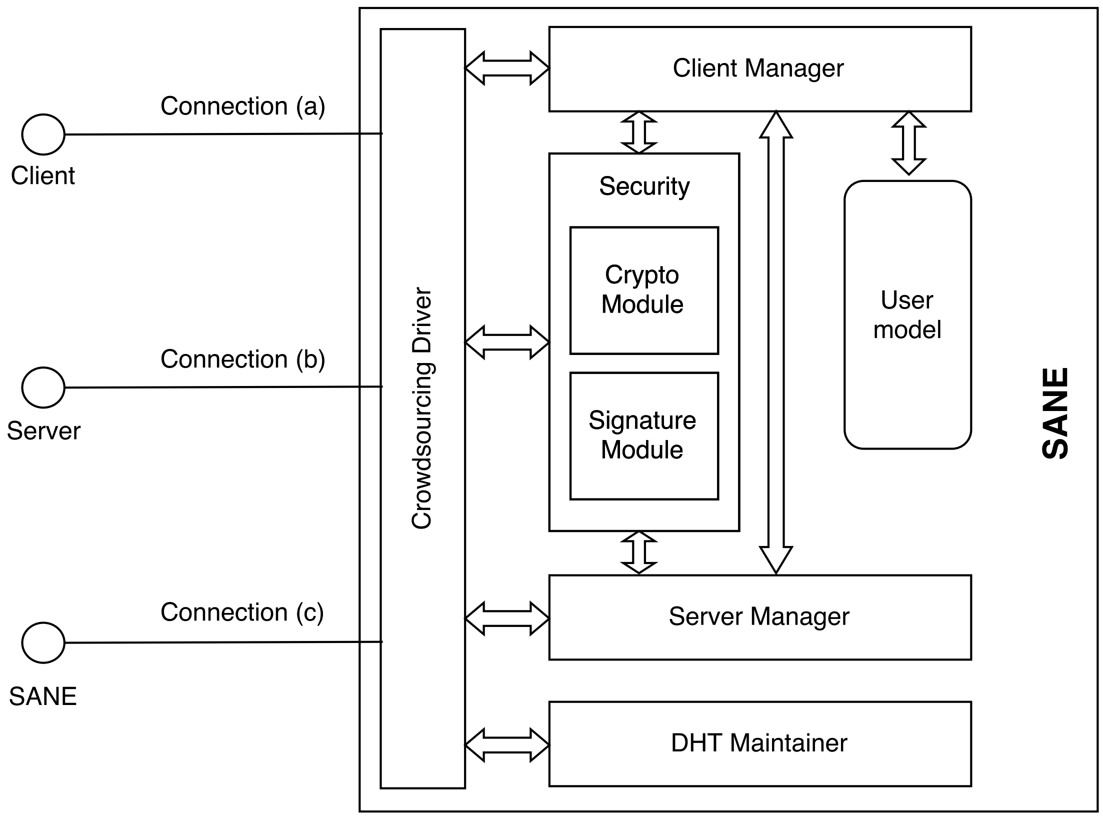


Figure 3.8: SANE Connection Type

To support SANE in HTTP/2, this chapter analyzed the components inside SANE and proposed the concept of adaption.

4 Implementation

Based on the concepts and requirements discussed in Chapter 3, this chapter will apply these methods and implementations. Start from implementing the header compression which is one of the most important features and biggest changes in HTTP/2 protocol. After discussion of header compression implementation, the client and server framework tools upgrade to support for HTTP/2 will be represented. With the fundamental building tools support, building or adapting current frameworks to support HTTP/2 can be possible. The implementation includes building three different types of proxy, SPDY alike proxy and adapting SANE to support HTTP/2.

4.1 Upgrading to HTTP/2

Switching to HTTP/2 cannot happen overnight: millions of servers must be updated to use the new binary framing, and billions of clients must similarly update their networking libraries, browsers, and other applications.

The good thing is, all modern browsers have committed to supporting HTTP/2, and most modern browsers use efficient background update mechanisms, which have already enabled HTTP/2 support with minimal intervention for a large proportion of existing users.

Before adding proxy support HTTP/2 into server frameworks, the current clients and server frameworks should be also updated.

4.1.1 Header Compression with HPACK

In specification RFC7451 defines HPACK [PR15], a new compressor that eliminates redundant header fields, limits vulnerability to known security attacks, and has a bounded memory requirement for use in constrained environments.

The HPACK format is intentionally simple and inflexible. It contains a list of header fields as an ordered collection of name-value pairs that can include duplicate pairs. Encoding in HPACK has two types: unsigned variable-length integers and stings of octets.

INTEGER REPRESENTATION

Integers are used to represent name indexes, header field indexes, or length of strings. Since the integer representation can start at anywhere within an octet, an integer is represented in two parts: a prefix that fills the current octet and an optional list of octets when the integer values does not fit within the prefix.

We assume the prefix length is N, and an integer I:

If $I < 2^N - 1$, then encode I within the N-bit prefix,

If $I \geq 2^N - 1$, set the N-bit prefixes to 1, and the value decreased by $2^N - 1$ is encoded using a list of one or more octets. The most significant bit of each octet is used as a continuation flags: its values is set to 1 except for the last octet in the list. The remaining bits of the octets are used to encode the decreased value. The pseudo code for encoding integers is listed in Listing 4.1.

```
if I < 2^N - 1, encode I on N bits
else
  encode (2^N - 1) on N bits
  I = I - (2^N - 1)
  while I >= 128
    encode (I % 128 + 128) on 8 bits
    I = I / 128
  encode I on 8 bits
```

Listing 4.1: Pseudocode to Represent an Integer I

To understand the processing better, an example is given: The encoding integer is 1239 and uses a 5-Bit prefix. The first step is to compare the integer with the largest number ($2^5 - 1 = 31$) which can be represented in prefix. The remain number will be represent in next octets. In next octet, represent the remainder after division of remain number ($1293 - 31 = 1208$) by 128 with most significant number is 1. The last octet represented the result of division. The diagram of encoding process corresponds to the pseudo code shows in Listing 4.2.

```
1239 is greater than 31,
// 5-Bit prefix can only represent 31 maxmium
  I = 1239 - 31 = 1208;
  1208 >= 128
    1208 % 128 = 56;
    encode (56 + 128) -> 10111000;
    1208 / 128 = 9;
  encode (9) -> 00001001
// The result of encoding 1239 is:
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| X | X | X | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
+---+---+---+---+---+---+---+---+
```

Listing 4.2: Example: Encoding 1239 Using a 5-Bit Prefix

Decoding the integer value from the list of octets stars by reversing the order of the octets in the list. Then, for each octet, its most significant bit is removed. The remaining bits of the octets are concatenated, and the resulting values is increased by $2^N - 1$ to obtain the integer value. The pseudo code for decoding HPACK compressed integer shows in Listing 4.3.

```
decode I from the next N bits
   if I < 2^N - 1, return I
   else
       M = 0
       repeat
           B = next octet
           I = I + (B & 127) * 2^M
           M = M + 7
       while B & 128 == 128
       return I
```

Listing 4.3: Pseudocode to Decode an Integer I

STRING REPRESENTATION

Header field names and values can be represented as strings. A string literal is encode as a sequence of octets in two ways: directly encoding the string literal octets or by using a static Huffman code. In HPACK specification generates a static Huffman code table ([PR15] Appendix B) from statistics obtained on a large sample of HTTP headers. It also add few changes to make that no symbol has a unique code length.

IMPLEMENT HPACK ALGORITHM

Based on the algorithm, the implementation ueses the API[4] provide by nghttp2 [Tsu15]. There are also other implementations that follow the same specifications. However, the nghttp2 HPACK API is more mature and used by most other applications including Apache and curl.

nghttp2 provides deflating tool for header encoding and inflating tool for header decoding. The deflating tool will be used to analyzed the header compressed ratio in the next chapter.

[4]https://nghttp2.org/documentation/tutorial-hpack.html

4.1.2 Update curl to Support HTTP/2

curl[cur16] is an open source command line tool and library for transferring data with URL syntax. libcurl[5] is a client-side URL transfer library and supports multiple HTTP methods, such as `POST`, `GET` and `PUT`. It can be easily integrated into other languages and programs by using the libcurl API.

BUILDING CURL

The reason to choose curl project as a client-side tool and library for HTTP/2 supporting is that it is wildly used and supports most HTTP methods and can be used for testing the web and proxy server in different situations.

curl supports HTTP/2 with the help of nghttp2 library, thus nghttp2 is prerequisite for successful building. At the same time, system needs a supplement SSL library to handle the ALPN and NPN.

To build the curl support HTTP/2 with nghttp2 installed[6] and up to date SSL library shows as following:

```
./configure -with-nghttp2=/usr/local -with-ssl
```

USING CURL WITH HTTP/2

curl can be used in command line or as library for other applications. Internally, curl will convert incoming HTTP/2 headers to HTTP 1.x style headers and provide them to the user, so that they will appear very similar to existing HTTP [Ste].

Using curl as command tool. To use curl command line supports `-http2` option to switch on the libcurl behavior from HTTP/1.1 to HTTP/2.

```
curl -http2 https://http2.akamai.com
```

Using curl in PHP. To enable HTTP/2 support with libcurl, set the `CURLOPT_HTTP_VERSION` to `CURL_HTTP_VERSION_2_0`. Listing 4.4 shows an example of using libcurl in PHP with HTTP/2 enabled. This script `GET` resource from Akamai HTTP/2 test page and print the connection header information.

```php
<?php
  $ch = curl_init("https://http2.akamai.com/");
  curl_setopt($ch, CURLOPT_HTTP_VERSION,
   CURL_HTTP_VERSION_2_0);
  // Specify the version of HTTP
  curl_setopt($ch, CURLOPT_HEADER, 1);
  curl_setopt($ch, CURLOPT_NOBODY, 1);
  curl_setopt($ch, CURLOPT_VERBOSE, true);
```

[5]https://curl.haxx.se/libcurl/
[6]Details of building nghttp2 can be found at: https://github.com/nghttp2/nghttp2

```
    $verbose = fopen('log', 'w+');
    curl_setopt($ch, CURLOPT_STDERR, $verbose);
    $result = curl_exec($ch);
    if ($result === FALSE) {
          printf("cUrl error (#%d): %s<br>\n", curl_errno($ch
     ),
          htmlspecialchars(curl_error($ch)));
        }
    rewind($verbose);
    curl_close($ch)
  ?>
```

Listing 4.4: Testing curl HTTP/2 GET Request in PHP

4.1.3 Apache HTTP Server HTTP/2 Support

Since Apache HTTP Server 2.4.17, it adds support for the HTTP/2 transport layer protocol based on `libnghttp2`[7]. HTTP/2 support in Apache is enabled by specifying `Protocols` in configuration files. The HTTP/2 protocol supports two schemes: `h2` (HTTP/2 over TLS) and `h2c` (HTTP/2 over TCP). Since most of the web browsers are *only* support HTTP/2 over TLS, e.g. Firefox and Google Chrome. As a result, TLS with ALPN negotiation is *de-facto* requirement for enabling HTTP/2 in the browser.

UPDATE THE APACHE TO SUPPORT HTTP/2

The test environment is based on Ubuntu, and currently there are two major LTS release version are wildly used. The newest release Ubuntu 16.04 (Xenial Xerus) has `apache 2.4.18` packages [8] available. This version of apache already supported HTTP/2. Another version of Ubuntu 14.04 (Trusty Tahr) only has `apache 2.4.7` and cannot support HTTP/2 by default. We can build the newer apache version from source file or uses third-party package repository[9]. In Listing 4.5 shows the method to add third-party package repository and upgrade Apache to latest version.

```
  $ apt-get install python-software-properties
  $ add-apt-repository -y ppa:ondrej/apache2
  $ apt-key update
  $ apt-get install apache2
```

Listing 4.5: Install Apache from Third-party Repository

[7] `libnghttp2` is an implementation of HTTP/2 and its header compression algorithm HPACK in C.
[8] http://packages.ubuntu.com/xenial/apache2
[9] https://launchpad.net/ ondrej/+archive/ubuntu/apache2

SELF-SIGNED SSL CERTIFICATE

SSL certificates are required in order to run web server using HTTPS protocol. There are two ways to gain a certificate: the first method is granted and provided by third-party certificate authority (CA). It uses a chain of trust, where each certificate is signed by a higher, more credible certificate. At the top of the chain of trust are root certificates, owned by CA provider. Another way is generate a self-signed certificate. A self-signed certificate is an identity certificate that is signed by the same entity whose identity it certifies.

When user visit a web site over HTTPS, web browser will receive the SSL certificate for the web site. It will examine the contents of the certificate to see that is indeed valid for the domain name it is visiting. After that, it verifies the chain of trust. It will look at who has signed the certificate. If that certificate is a root-certificate, it will compare it against the ones shipped with the operating system. If it is a non-root certificate, it follows the chain of trust up one more level.

For self-signed certificates, there is no chain of trust. The certificate has signed itself. The web browser will then issue a warning, telling that the web site certificate cannot be verified unless client accepts this self-signed certificate.

One can generate a self-signed certificate with OpenSSL. OpenSSL is an open source project that provides a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library. To generate a self-signed certificate using OpenSSL shows in Listing 4.6:

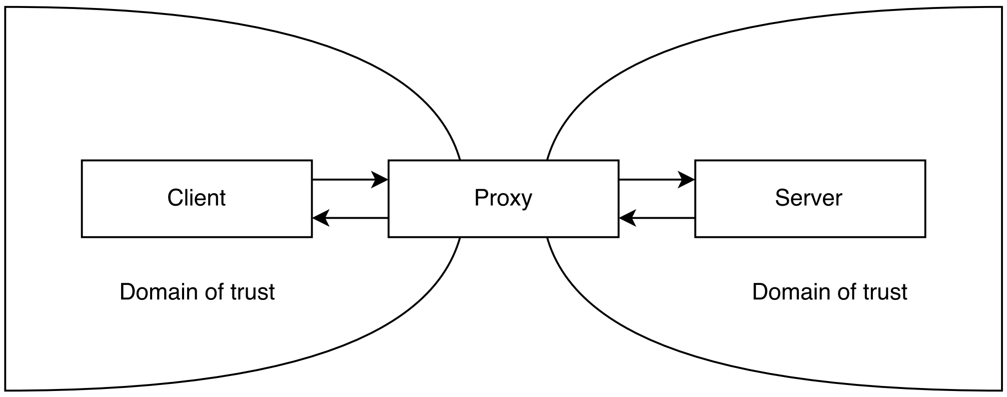
```
openssl
  req -x509 -nodes -days 365 -newkey
  rsa:2048
  -keyout ssl.key
  -out ssl.crt
```

Listing 4.6: Generate Self-signed Certificate with Openssl

With self-signed certificate has acceptance problem on opponent parts. When a communication only has two participants, it easy to solve the trust issue by accepting each other manually. However, when a proxy involved in a communication, there is needs to expand the trust of domain.

In top diagram of Figure 4.1 shows the trust domain between client, proxy and server. Since the client and proxy using the SSL encryption connection, the data integrity and security can be ensured. Also this result applies to the connection between proxy and server. However there is an issue that server and client cannot trust the each other's identity and data integrity. One solution of this problem is to use a full-trust proxy.

A full-trust proxy is a proxy that can be trusted by client and server. Its identity is *explicit* to



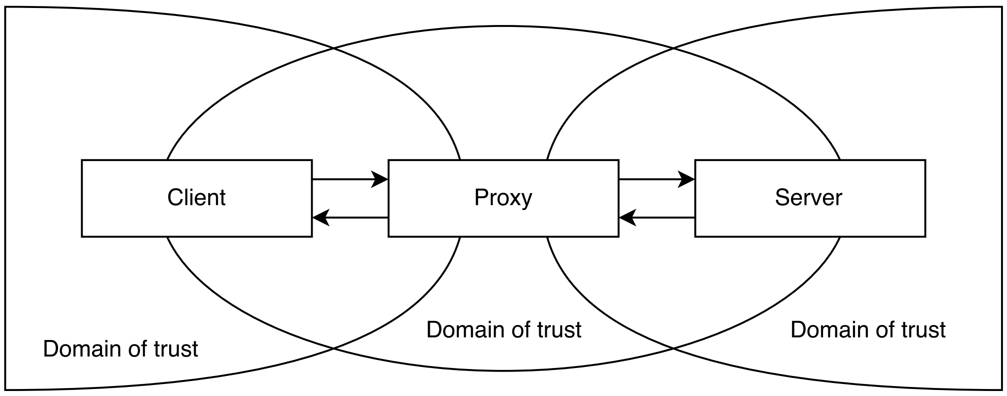


Figure 4.1: Trust Domain of Client, Proxy and Client Switch

both client and server. If the client sends any data through the full-trust proxy, that means it accept the response via proxy is genuine from the server. If the client or server do not trust the proxy, they should never send any information through the proxy. The bottom diagram of Figure 4.1 dedicates the idea of an explicit proxy.

Self-signed certificates has trust issues, and need end-user to valid the certificate by themselves. Introducing a proxy between client and server can be a hazard for man-in-the middle attack (MITMA)[10]. Only the client can accept the identity of explicit proxy, the self-signed certificates can work in a HTTP/2 proxy.

CONFIGURE APACHE TO SUPPORT HTTP/2

In order to support HTTP/2 in Apache, the `mod_http2` module and `mod_ssl` must be enabled. To accept and upgrade HTTP/1.1 to HTTP/2, the Apache configuration file should be modified. In Listing 4.7 gives an example to configure Apache to support HTTP/2. Since Apache enables HTTP/2 via `protocol` directive. The specification of protocol should be added in configuration file. Parameter `h2` enables the HTTP/2 protocol, h2c enables HTTP/2 protocol via plain text. For clients do not support HTTP/2, the HTTP/1.1 also need to be supported. Because of modern browsers need HTTPS for HTTP/2 con-

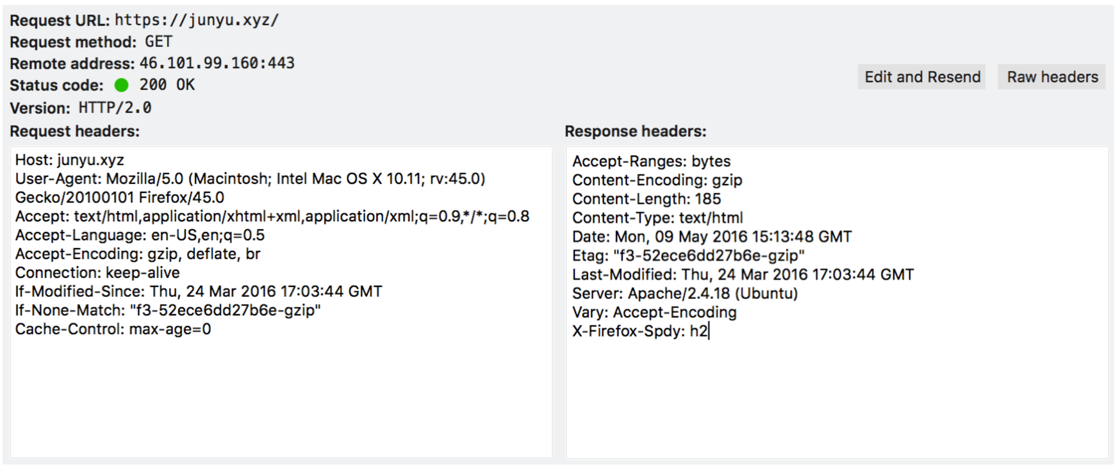
[10]Man-in-the-middle attack is an attack where the attacker secretly relies and possibly alters the communication between two parties who believe they are directly communicating with each other

nection, the self-signed certificate location needs to be inserted in the configuration.

```
# enable apache http2 and ssl modules
$ a2enmod http2 ssl
# modify the default-ssl.conf file to support HTTP/2
   protocol
<IfModule mod_ssl.c>
<VirtualHost *:443>
  # enable http2 protocol on apache
  Protocols h2 h2c http/1.1
  # specify the location to the self-signed certificate
   file and key
  SSLCertificateFile /path/to/certfile
  SSLCertificateKeyFile /path/to/keyfile
  </VirtualHost>
</IfModule>
# enable default-ssl site configuration
$ a2ensite default-ssl.conf
```

Listing 4.7: Enable and configure Apache to support HTTP/2

Once HTTP/2 protocol is enabled on Apache server, the response will be sent over the HTTP/2 connection as showing in Figure 4.2. Additionally, the server push feature is enabled by default.



Request URL: https://junyu.xyz/
Request method: GET
Remote address: 46.101.99.160:443
Status code: ● 200 OK
Version: HTTP/2.0

Request headers:

```
Host: junyu.xyz
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:45.0)
Gecko/20100101 Firefox/45.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
If-Modified-Since: Thu, 24 Mar 2016 17:03:44 GMT
If-None-Match: "f3-52ece6dd27b6e-gzip"
Cache-Control: max-age=0
```

Response headers:

```
Accept-Ranges: bytes
Content-Encoding: gzip
Content-Length: 185
Content-Type: text/html
Date: Mon, 09 May 2016 15:13:48 GMT
Etag: "f3-52ece6dd27b6e-gzip"
Last-Modified: Thu, 24 Mar 2016 17:03:44 GMT
Server: Apache/2.4.18 (Ubuntu)
Vary: Accept-Encoding
X-Firefox-Spdy: h2
```

Figure 4.2: Connection via HTTP/2: Response and Request Header

ENABLE SERVER PUSH

The HTTP/2 protocol allows the server to push resources which specified by server to a client when it asked for a particular one. Apache enables server push function by default. To specify certain resources push to a client, the configuration can be done by editing Apache `H2Push` directive or dedicating in applications. Since the `H2Push` directive settings can be found on Apache HTTP Server documentation [11], the example is omitted here. The convenient way to enable server push is through application. The `Link` headers of response specified in RFC5988 are responsible for notifying the server which resource should be pushed. When a `Link` has `rel=preload` attribute, it will be treaded as a resource to push. In Listing 4.8 shows an example of enabling server push by using `header()` function in PHP to set the resource to be pushed in `index.php` page. When a client sends the request for `index.php`, the CSS file `example.css` will be pushed to the client without the request of CSS reach to the server.

```
<?php
  header('Link:<css/example.css>; rel=preload');
?>
<html>
...
</html>
```

Listing 4.8: Enable Server Push Header Function in PHP

4.2 Build Proxy with HTTP/2 C Library

nghttp2 [Tsu15] is an implementation of HTTP/2 and its header compression algorithm HPACK in C. It provides a reusable C library for enabling HTTP/2 and has been used in many frameworks through API, e.g. Apache HTTP Server, curl tools. nghttp2 library also provides a protocols translation tool called `nghttpx`. It can act as an essential module for building a proxy service with complicated network setup.

In Section 3.4.1 we discussed three different architecture proxies. The communication between client and proxy can be either HTTP/2 or HTTP/1.1, the same between server and proxy. Because of the encryption of HTTP/2 header and hard to debug a binary encode header values, we need a mature tool to handle the translation process. The `nghttpx` from nghttp2 can act as a proxy translating protocols between HTTP/2 and other protocols, such as HTTP/1.1 and SPDY. It has several functional modes and with additional support we can build the proxies we need.

This section implement the reverse proxy with `nghttpx` to prove the concept mentioned in Chapter 3. HTTP/2 connection can be established via TLS or TCP. In Table 4.1 lists 8 situations of HTTP/2 connection with proxy setting.

Alice The client sends request and receive response from proxy (Claire). It can use HTTP scheme or HTTPS scheme for secure connection;

[11]https://httpd.apache.org/docs/trunk/mod/mod_http2.html#h2push

Bob The connection between Alice and Claire. It has three different protocol: HTTP/1.1, HTTP/2 via TLS (h2) or HTTP/2 via TCP plain text (h2c);

Clare The proxy server, it can has three types: Upgrade, Downgrade and Straightforward;

David The connection between Claire and server (Eason), similar as Bob;

Eason The destination server, it can support scheme: HTTP and HTTPS.

Alice	Bob	Claire	David	Eason
HTTP	HTTP/1.1	Upgrade	h2c	HTTP
HTTP	HTTP/1.1	Upgrade	h2	HTTPS
HTTPS	HTTP/1.1	Upgrade	h2c	HTTP
HTTPS	HTTP/1.1	Upgrade	h2	HTTPS
HTTP	h2c	Downgrade	HTTP/1.1	HTTP
HTTP	h2c	Downgrade	HTTP/1.1	HTTPS
HTTPS	h2c	Downgrade	HTTP/1.1	HTTP
HTTPS	h2	Downgrade	HTTP/1.1	HTTPS
HTTP	h2c	Straightforward	h2c	HTTP
HTTPS	h2	Straightforward	h2c	HTTP
HTTP	h2c	Straightforward	h2	HTTPS
HTTPS	h2c	Straightforward	h2	HTTPS

Table 4.1: Proxy with HTTP/2 Connection

To build proxy with different architectures, `nghttpx` is in uses. In order to simplify the discussion and showcase of implementation, the building methods as following only present when Alice, Claire and Eason are all using secure connection.

The default mode of `nghttpx` works as reverse proxy for both HTTP/2 and HTTP/1 clients to back-end server. The front-end connection can be either HTTP/1 or HTTP/2 by the specification in a program.

BUILD DOWNGRADE PROXY

In Downgrade Proxy mode, the client communicates with proxy server by using HTTP/2 connection and the connection between proxy and server is in HTTP/1.1. This is a very common mode in nowadays Internet service. By using `nghttpx` we can easily achieve this goal. nghttpx module can be used as a proxy server or a translation module in front of other web proxy. To build a Downgrade Proxy with other proxy will be discuss in next section as SPDY alike proxy.

Listing 4.9 gives an example to enable a Downgrade Proxy with `nghttpx`:

```
frontend=0.0.0.0,3000
backend=127.0.0.1,8080;tls
private-key-file=/path/to/server.key
certificate-file=/path/to/server.crt
```

Listing 4.9: Configuration for Downgrade Proxy in nghttpx

The first line specifies the `PORT` and allows nghttpx to accept connection from all address. The second line defines the back-end server address. HTTP/2 connection between server and proxy need to be encrypted, `nghttpx` needs to knows the SSL key and certification files location which addressed in line 4 and 5.

BUILD STRAIGHTFORWARD PROXY

The Straightforward Proxy mode assumes that all the communication among client, proxy and server are HTTP/2 connection. `nghttpx` supports this mode by configure the back-end connection to server.

Based on the trust of domain expanding which discussed in last section, when using a self-signed certificate, we need to expand the trust domain to includes proxy, client and server. In another words, the client has to trust that the proxy will not connect a untrusted server to obtain resources and also proxy connects to a trust server.

```
-k
frontend=0.0.0.0,3000
backend=127.0.0.1,8080;proto=h2;tls
private-key-file=/path/to/server.key
certificate-file=/path/to/server.crt
```

Listing 4.10: Configuration for Straightforward Proxy in nghttpx

In Listing 4.10 shows the configuration of building a Straightforward Proxy with `nghttpx`. The first line `-k` specifies that proxy would not verify the identity of destination server and trust it unconditionally.

BUILD UPGRADE PROXY

The Upgrade Proxy mode is similar to the Straightforward Proxy mode with slightly difference. The connection between client and proxy is using HTTP/1.1 instead of HTTP/2. The configuration of `nghttpx` is the same, but we need to specify the client to use a HTTP/1.1 connection. When a non-encrypted connection between client and proxy server is in use, even the proxy server support HTTP/2 will be rolled back to HTTP/1.1 connection.

4.3 Build General SPDY alike Proxy

Squid is a popular caching proxy and has similar behavior as SPDY Accelerator: the HTTP request will be cached and passed through the Squid proxy; encrypted connection (HTTPS) will be filtered out from the Squid. Squid does not support HTTP/2 yet, in this implementation we can adapt the Squid to support front-end HTTP/2 connection between client and proxy and analyze the improvement.

4.3.1 Architecture

Figure 4.3 shows the general architecture of the SPDY alike Squid proxy which can support HTTP/2 connection. There are two main components in this architecture:

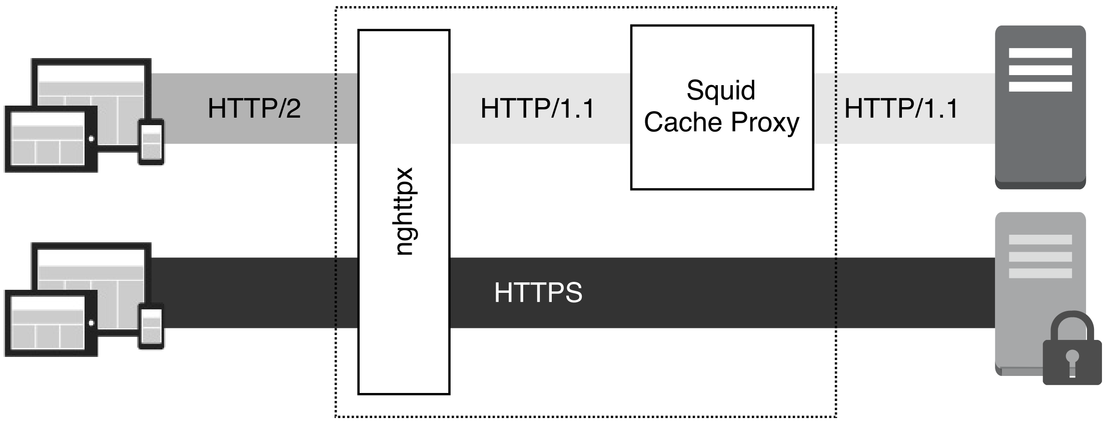


Figure 4.3: Architecture of SPDY alike Squid Proxy

nghttpx. It handles the protocol translation process. All the connection via `nghttpx` will be switched to HTTP/1.1 for Squid proxy to handle the cache process.

Squid Cache Proxy. The main task of this component is to cache the resource from remote server. With additional cache control mechanism, such as Least Frequently Used Replacement (LFUR), the most frequently resources can be cached in Squid Cache proxy to reduce the load time.

HTTP and HTTPS connection are handled separately and differently in this proxy implementation.

HTTP Request When a client request a resource from a no-encrypted web server. The request will be switched to an encrypted connection between proxy and client. After switch to HTTPS, proxy and client can negotiate which protocol to use next. The connection can be either HTTP/2 if the client support or fall back to HTTP/1.1 if the client can not support. After this process the `nghttpx` will forward or translate further request to HTTP/1.1 for Squid cache proxy. Only after the translation, Squid proxy can understand the request and continue.

HTTPS Request When client's request sends to an encrypted web server. The `nghttpx` will forward the request to Squid without modifying the header section. Clients behind an explicit proxy use the `CONNECT` HTTP method. The first connection to the proxy port uses HTTP and specifies the destination server (often termed the Origin Content Server, or OCS). The Squid proxy simply acts as a tunnel, and blindly proxies the connection without inspecting the traffic. The connection between client and proxy is either HTTP/2 or HTTP/1.1 which depends on the destination server.

4.3.2 Integrate Squid and nghttpx

NGHTTPX CONFIGURATION

In order to integrate `nghttpx` with Squid cache proxy we need to specify that the `nghttpx` works in HTTP/2 proxy mode. In this mode, `nghttpx` can handle and translate the communication protocol automatically. However, it is not working as the default proxy mode that cache and server the resource. It acts like forward proxy and assumes the back-end is HTTP proxy server.

By default, front-end connection is encrypted. It means if clients support HTTP/2 protocol the connection can be upgrade to HTTP/2, otherwise the connection maintains in HTTP/1.1.

This mode is invoked with `-http2-proxy` parameter and should set the back-end port to the Squid listening port.

SQUID CONFIGURATION

In order to make the Squid works properly with `nghttpx`, we need to customize the Squid directives [Sai11]. The configuration file of Squid is located at `/etc/squid3 /squid.conf` in Ubuntu.

HTTP port directive is used to specify the port where Squid will listen for client connections. The address and port should be the same as the back-end server setting in `nghttpx` configuration file.

`http_port 3128`; set the listen port to 3128;

HTTP access control directive is used to grant access to perform HTTP transactions through Squid. For security reason we should only allows the Squid to accept the connection from `localhost`.

`http_access allow localhost`;

Memory cache mode directive is for setting the memory cache behavior.

`memory_cache_mode always`; the mode `always` is used to keep all the most recently fetched objects that can fit in the available space;

Cache Replacement Policy. The web documents cached in the main memory or RAM can be served faster compared than the hard disks with mechanical parts. However, the space available in RAM for caching is limited. Only most popular objects or the documents with a high probability of being requested again. The Least Frequently Used (`lru`) cache replacement policy helps Squid to keep the most popular objects in the cache, and recycle the space by removing the oldest (since the last HIT).

`memory_replacement_policy lru`; enable cache replacement policy.

4.3.3 Configure Client Support

To make Firefox or Chromium use nghttpx front-end SPDY alike proxy, user has to use proxy.pac script file to automatically change the proxy address. In Listing 4.11 shows the example script file. The `SERVERADDR` and `PORT` is the hostname/address and port of the machine `nghttpx` is running. Since both Firefox and Chromium require valid certificate for secure proxy, client needs to accept the self-signed certification.

```
function FindProxyForURL(url, host) {
    return "HTTPS SERVERADDR:PORT";
}
```

Listing 4.11: Proxy.pac File Example

To support HTTP/1.1 clients use this proxy, one method is to maintain a HTTP header translator locally at clients side. This translator converts client's HTTP/1.1 request to HTTP/2 request and sends to SPDY alike proxy. The further connection is similar to the client which supports HTTP/2 natively.

4.4 Adapting SANE to Support HTTP/2

SANE is a highly specialized Web proxy application which behaves not the same as normal proxy. Unlike other proxies forward client's requests and responses, SANE acts as a fully functional application between client and server to handle the data procedure. On client's point of view, SANE is an application running on web server. From server's perspective, SANE behaves as a client who sends request and get responses. The role of SANE is more independent compare to regular proxy.

This section will introduce how to adapt SANE which currently *only* supports HTTP/1.1 to support HTTP/2 in different architectures.

4.4.1 Adapting SANE to Downgrade Proxy

A Downgrade Proxy with SANE means it will support HTTP/2 connection between client and proxy, but will not upgrade the connection between proxy and server. Because the feature of decoupling client from server in SANE, the adapt methods are easily to let SANE application act as a proxy web server with HTTP enabled. This section introduces two mechanisms to achieve this goal.

INTEGRATE NGHTTPX WITH SANE TO SUPPORT HTTP/2

An intuitive way to support HTTP/2 in SANE is by adding a translation module. The responsibility of translation module is to help SANE application deal with the HTTP/2 request. Since the complex and binary features of HTTP/2 connection, nghttpx, a more maturer component for translation HTTP/2 request is introduced to handle the process.

As showing in Figure 4.4, `nghttpx` is placed in front of SANE to help translating the HTTP connection. When clients try to connect the SANE server, `nghttpx` will manage this request and response for negotiation the HTTP protocol between client and SANE. If the client accepted the HTTP/2 protocol the further transaction will be carried under HTTP/2. If the client does not support HTTP/2, the connection will fall back to HTTP/1.1 (which is not showing in Figure 4.4). After `nghttpx` translates the POST request from HTTP/2 to HTTP/1.1, the SANE can continue the rest processing as regular SANE. The responses also follow the same path and workflow.

By adding `nghttpx` is intuitive and can adapt the current frameworks to support HTTP/2 in a Downgrade Proxy without changing other components. SANE and related server will maintain the same. However, this addition layer can cost few more cycles for messages to transact.

UPGRADE APACHE SERVER TO SUPPORT HTTP/2 IN SANE

Additional translation layer is easy to adapt current frameworks to support HTTP/2 but may affect the performance of the application behind the translation module. Since the Apache Web Server supports HTTP/2 and based on the research from Section 3.2.1, we can upgrade the Apache Web server to enable front-end HTTP/2 connection in SANE.

Once SANE is running on a Web server which supports HTTP/2, the connection between

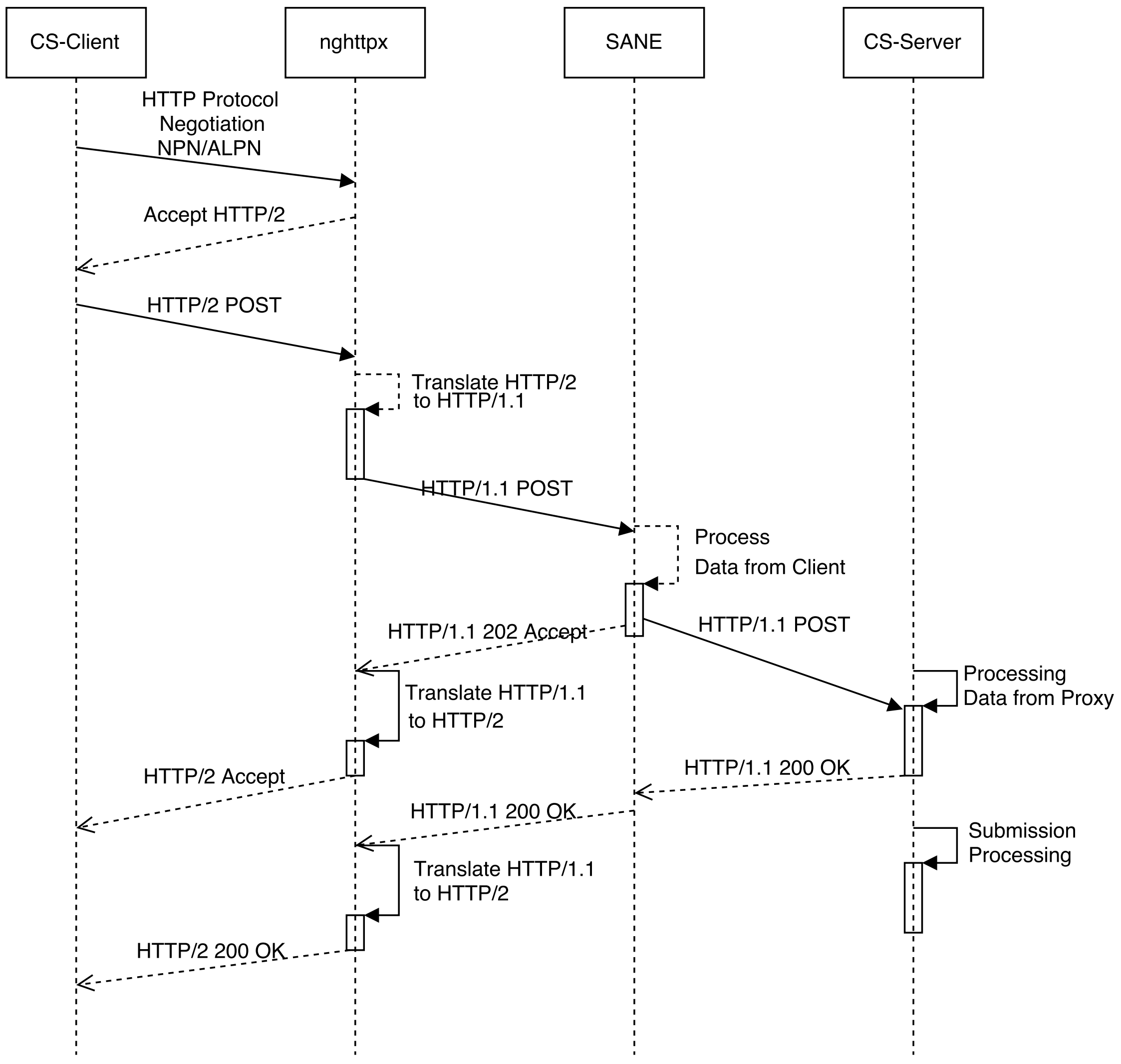


Figure 4.4: Exemplary HTTP/2 Communication Support with nghttpx in SANE

client and proxy is upgraded to HTTP/2 automatically. The communication sequence of SANE maintains the same from proxy and server, since the SANE application is not upgraded to support HTTP/2 as a client yet.

4.4.2 Adapting SANE to Straightforward Proxy

HTTP Request in SANE

In a Straightforward Proxy, all connections should be under HTTP/2. The front-end connection can be upgraded through the methods mentioned in previous section. In order to reduce the latency of communication, adapting the front-end communication with Apache upgrade is a better choice. It avoids the process time in `nghttpx` and provide a more straightforward communication regardless the server's setup. From SANE to the destination server, the upgrading should be in the SANE since the SANE behaves as a client of the destination server which controls the final connection mechanism.

SANE communicates with other components using HTTP `POST` and `GET` methods. It sends the parameters in HTTP request body. The HTTP library locates at `SANE/libs/HTTP.php` is responsible for all the further communication outside SANE no matter the destination server is another SANE or Crowdsourcing server.

When `post_request` or `get_request` invoked by other SANE methods, the request data, request URL and a boolean for enabling HTTPS connection are passed to the functions. Then HTTP library assembles the data and sends to the request server with HTTP or HTTPS connection based on the boolean. For `get_request` it reconfigures the request URL based on parameters.

INTEGRATE CURL IN SANE TO SUPPORT HTTP/2

The reason to use curl instead of to upgrade the header field directly is that HTTP/2 protocol uses HPACK encoding and also needs other components to handle the APNL protocol. This kind of work is deep down to the implementation of HTTP/2 protocol and is not the purpose of using HTTP/2 in proxy communication. curl provides an easy to use API in PHP, hide HTTP/2 binary nature and convert received HTTP/2 traffic to headers in HTTP/1.1 style. This allows application works in both HTTP/2 or HTTP/1.1 connection.

A new method is implemented in SANE to test the memory modification type proxy works as expected. This method allows user to POST a message via SANE, then SANE hashes the message and POST this hashed message to server.

```php
<?php
$data = hash("sha256", $method_values["message"]);
include_once(getcwd()."/libs/HTTP.php");
include_once(getcwd()."/libs/Database.php");
global $_CONFIG;
$theQuery = "INSERT INTO `Messages` (`message`) VALUES (:
    message);";
$theValues = array("message" => $data);
db($theQuery, $theValues, $_CONFIG["dbPersistent"]);
$post_to_send = array(
  "method"=>"viaSANE",
  "message"=> $data,
  );
$response = post_request2($post_to_send,"127.0.0.1:8080",
    true);
?>
```

Listing 4.12: Method to Store Message in SANE and Send to Server

This method uses upgraded HTTP request to post data. The code of HTTP post function is listed in Listing 4.13. When secure connection enabled, the post request will try HTTP/2 connection by default. If the destination server does not support HTTP/2 the post will be sent via HTTP/1.1. When the secure connection is disabled and the server supports HTTP/2 via plain text, the connection will be upgraded to HTTP/2 with `h2c`.

This method only emulates the basic communication mechanism in SANE: data sent by crowdfunder to SANE will be modified and reconstructed before being sent to the destination server. It omit the encryption and assertion function of SANE. The reason to implement method without concern of other components is that the evaluation only focus on the performance *directly* related to HTTP/2 connection and it is easier to investigate the message flow. It is easy to be integrated in further work. Since this function is implemented in lower layer of the system as a library, any other methods related and invoke HTTP library will perform the same course.

```
function post_request($data, $url="127.0.0.1:8080", $secure
   =true){
  global $_CONFIG;
  if (!defined('CURL_HTTP_VERSION_2_0')) {
        define('CURL_HTTP_VERSION_2_0', 3);
  }
  $params = http_build_query($data);
  $connect = 'http'.($secure?"s":"").'://'.$url;
  $ch = curl_init();
  curl_setopt($ch, CURLOPT_HTTP_VERSION,
   CURL_HTTP_VERSION_2_0);
  curl_setopt($ch, CURLOPT_URL,$connect);
  curl_setopt($ch, CURLOPT_POST, 1);
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1 );
  curl_setopt($ch, CURLOPT_POSTFIELDS, $params);
  curl_setopt($ch, CURLOPT_HEADER, 1);
  curl_setopt($ch, CURLINFO_HEADER_OUT, true);
  $result = curl_exec($ch);
  curl_close($ch);
  header("HTTP/2 200 OK");
  print_r("Send success");
}
```

Listing 4.13: Upgrade SANE to Support HTTP/2 with curl Library

4.5 Summary

This chapter presents the fundamental methods to adapt system frameworks to support HTTP/2. Firstly, the implementation of compressing and encoding HTTP/1.1 header to a binary header with HPACK provides a clear view of HTTP/2 header structure and helps evaluate the benefits from header compression. Then curl library and command tool introduced to adapt SANE to support HTTP/2. The configuration and implementation of Apache server demonstrates the common way to support HTTP/2 in current web server infrastructure.

Secondly, three different architecture proxies with HTTP/2 are implemented to prove the idea derived from Chapter 3. The feasibility and performance of each architecture will be given in the next chapter. To prove the idea in a more realistic situation, a SPDY alike proxy is implemented, this help us gain more information of the performance from HTTP/2.

Finally, a new function method is introduced into SANE to study the benefit from adapting HTTP/2 in SANE. This function allows us to study the performance without involving other factors, such as system encryption overhead and memory query latency.

5 Evaluation

This chapter represents the evolution of introducing HTTP/2 with proxy setting. Since the different architectures of system with HTTP involved, to analyze the performance and prove the concept of bringing HTTP/2 proxy into the current system framework can improve the performance, the evaluation will be separated into three main parts. Firstly, the compression ratio of header with HPACK will be represented. Header compression is a significant feature of HTTP/2 and fundamental change from HTTP/1.1. By analyzing the performance leads to a comprehensive understanding about the performance related to the header compression. Secondly, the three different architecture proxies will be evaluated to prove the feasibility and performance will also be given accordingly. A SPDY alike proxy will be evaluated in this chapter as a supplement to prove the concept that HTTP/2 can be used in proxy setting. This gives a more common usage scenario of proxy when web browsing. Finally, the adapted SANE, a specialized memory modifies proxy, with HTTP/2 support will be tested and investigated.

5.1 Header Compression Ratio

Header compression is a new feature brought into HTTP/2 protocol. The algorithm and implementation are discussed in Chapter 3 and 4. For header compression performance analyze, two methodologies are represented in this section. The evolution of header compression gives an overall view about HTTP/2 header compression mechanism effects on the performance.

Homologous Requests

The testbed website contains CSS file, picture files and other pages. All the requests are homologous which means all the requests query the resource from the same website domain. The header request are compressed in different number each time. Listing5.1 shows the first two original requests of the testbed website and the compressed HTTP/2 header (the value of `wire` key). The calculation of $CompressedRatio(\phi_H)$ is based on the Equation 5.1:

$$CompressedRatio(\phi_H) = \frac{OutputLength}{InputLength} \tag{5.1}$$

The difference between two requests are only the `:path` values and other information is the same, we call this kind of request is *Homologous Requests*. However, the compressed ratio has a significant disparity. The first request ϕ_H is 0.625 and the second request ϕ_H is 0.114. The reason is that HPACK uses *Dynamic Table* to store the redundant data to save the header space. This provides a significant reduce of data during header transaction. Figure 5.1 shows the number of request and corresponding ratio. As can be seen, the ratio is decreasing, which means the performance benefit from HPACK is increasing with the increased number of requests from the same server domain.

```
//First request
"headers": [
    {":method": "GET"},
    {":path": "/guideline/css/pure-min.css"},
    {":authority": "junyu.xyz"},
    {"user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X
   10.11; rv:46.0) Gecko/20100101 Firefox/46.0"},
    {"accept": "text/css,*/*;q=0.1"},
    {"accept-language": "en-US,en;q=0.5"},
    {"accept-encoding": "gzip, deflate, br"},
    {"referer": "https://junyu.xyz/guideline/"},
    {"connection": "keep-alive"},
    {"pragma": "no-cache"},
    {"cache-control": "no-cache"}
  ],
"wire": "3fe1...9cbf" //325 Byte
//Second Request
"headers": [
  ...
  {":path": "/guideline/css/grids-responsive-min.css"}
  ...
  ],
"wire": "82..bf" //60 Byte
```

Listing 5.1: Header Compression Test Set

General Requests

To obtain a more general idea of the average compressed ratio, the same number (10 request each site) of requests for different sites is compressed. The request headers are varied from each other and contains different methods and values of the same key.

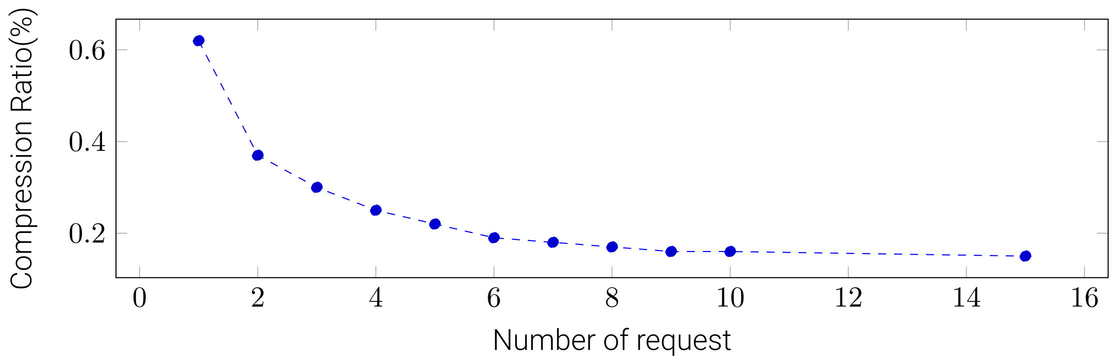


Figure 5.1: Ratio of Homologous Requests

Based on the Figure 5.2, the average size of header among 10 different sites is 3219 Byte, and the average ϕ_H is 0.201, the saving from HPACK is approximately 80% which is 2572 Byte of average header size.

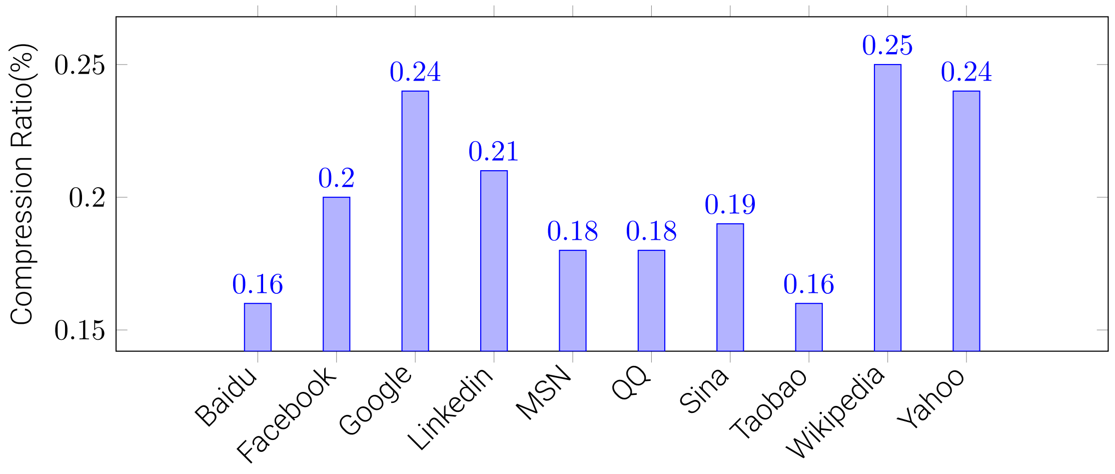


Figure 5.2: Ratio of Heterologous Requests

Overall. This shows that HPACK can dramatically reduce the header size and decrease the traffic from header transaction.

Dynamic Table contributes most in header compression. HPACK uses *Static Table* to represent common header fields, that reduces the header size. However, the *Dynamic Table* reuse mechanism shows a more advantage of header ration decreasing. As can be ascertained from the test of *Homologous Requests*, start from the second request, the encoded header size drops dramatically.

Header Compression is related to header complexity and redundancy. The results from *Heterologous Request* shows the ϕ_H is varied from different site. After analyzing the requests of different sites, ϕ_H depends on the header fields complexity and redundancy.

HPACK has security vulnerability. HPACK uses Static Table to represent frequently used header field. It means the same header field value computes the same encoded header field. For example, `:method: GET` always encoded as `82`. Since the Dynamic Table uses Huffman Code, there is likewise the possibility for hacker to guess the header fields by comparing the encode results. Relying on HPACK binary feature for security is not wise enough for nowadays Internet environment. It also means HTTPS is necessary when the application demands a more secure setting.

5.2 Evaluation on Proxy with HTTP/2 Setting

In this section the evaluation of general proxy will be demonstrated. First discussion is about the different architecture of proxies that influence the functioning. Second part of this section is giving a more realistic scenario of using a proxy in web browsing and the performance will be analyzed.

5.2.1 Architecture Performance Influence

From the concept of Chapter 3, three different architectures are:

1. Straightforward Proxy which has HTTP/2 connection on both side;
2. Upgrade Proxy which the connection on client side is HTTP/2 and server side is HTTP/1.1;
3. Downgrade Proxy whose connection setup is opposite of Upgrade Proxy.

These three different architecture proxies are implemented on Ubuntu 14.04 server, which is located in Frankfurt, Germany. The test server is running Apache and has a single page for responding the requests. For each architecture, two different variables are given: n and c. n means that the total number of requests are sent to the destination server. c means the number of concurrent clients. For example, $-n100 - c10$ represents 100 requests are sent concurrently by 10 clients. In Figure 5.3 shows the total requests finishing time when $n = 100; c = 10$, $n = 50; c = 5$ and $n = 1; c = 1$. The upper figure shows the result which is running on a local machine and bottom figure shows the mean request finishing time when requests are sent from the remote client. For remote client request, each setup runs three times to calculate the mean finish time to smooth the network speed fluctuation.

Overall. The result above shows the different architecture of proxies will affect the performance.

HTTP/2 enabled on the client side is preferable than server side. Based on the results, different architecture proxies will affect the performance. The calculation of mean per-

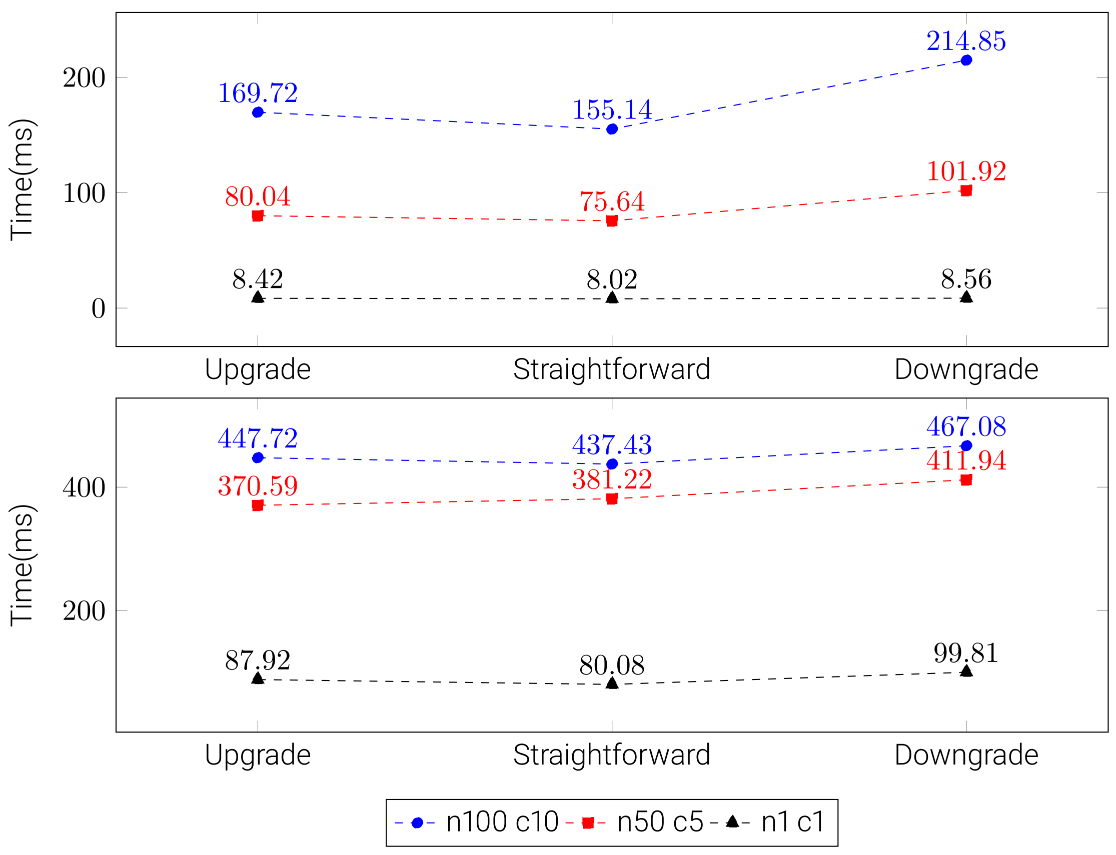


Figure 5.3: Different Architecture of Proxy with HTTP/2

formance fluctuation between two proxies is $\Delta = (SUM(Time_A) - SUM(Time_B))/3$. The mean performance fluctuation is $\Delta_{USLocal} = 6.46$ and $\Delta_{USLocal} = 3.84$. The mean fluctuation between Downgrade Proxy and Straightforward Proxy is $\Delta_{DSRemote} = 28.84$ and $\Delta_{DSRemote} = 26.7$. If Δ is more close to 0 means the performance is more close to a native HTTP/2 proxy. Since $\Delta_{DSLocal} > \Delta_{USLocal}$ and $\Delta_{DSRemote} > \Delta_{USRemote}$, we can assume that the enabling HTTP/2 on the client side is more efficient and preferred. The reason is that the HTTP/2 proxy is running on the server side, and behaves as a protocol translator before the server responses to the request. The abstract distance between server and proxy is smaller than client and proxy.

Straightforward Proxy has most performance improvement. When HTTP/2 is enabled on both sides, the header traffic reduced thus decreas the request time. With the increase of requests number, the benefit is more noticeable.

Header Compression is not the only factor affecting the performance. The result of request -n50 -c5 from remote server shows the Upgrade Proxy uses less time than Straightforward. The transaction latency, additional ACK rounds for HTTP/2 protocol and other factors should be considered when estimates the performance when the test is running in a realistic state of affairs. In other words, for real-time request should use HTTP/2 carefully.

5.2.2 SPDY alike Proxy Performance Evaluation

This evaluation is based on the implementation of SPDY alike proxy presented in Section 4.3. The implementation has two major parts: the protocol translator which converts the HTTP/1.1 style header to a HTTP/2 header for clients; the Squid proxy which behaves as cache proxy and caches the requested resources. The test server is running on a virtual cloud host based in Frankfurt, Germany. In order to analyze the performance improvements, the test sample selected 10 popular websites from Alexa[12] that supports non-HTTPS connection and has a consistent request number per web page. The website information and the request number are listed in Table 5.1. From the feature of SPDY alike proxy we know that only the HTTP requests will pass through a cache proxy. For HTTPS request, the connections will maintain the same protocol and would not be affected by cache proxy. HTTPS request number is also listed in Table 5.1 since it is a huge factor that affects the performance of HTTP/2 SPDY alike proxy.

Website	HTTP Request #	HTTPS Request #	Page Size
Baidu	19	0	579.95 KB
t.co	2	0	3.59 KB
Onclickads	6	9	256.71 KB
Imgurl	94	6	3252.13 KB
FC2	33	12	1130.52 KB
IMDB	115	72	4443.19 KB
Aliexpress	44	4	2085.60 KB
Diply	128	8	10368.14 KB
Nicovideo	76	26	2514.64 KB
Alibaba	63	10	2944.88 KB

Table 5.1: Test Website Information.

In Figure 5.5 shows the page loading time of the websites which are listed in the table above. The gray bar presents the mean page loading time using direct HTTP/1.1 squid proxy; the red bar dedicates the mean page loading time using HTTP/2 squid proxy. The blue line with triangle shows the deviation between two connections.

Overall, the HTTP/2 Squid proxy does not show a very good improvements. The performance is about the same and sometimes are slower than HTTP/1.1 connection.

A HTTP/2 based SPDY alike proxy can be constructed. The implementation based on the concept from Chapter 3 is an example of proxy using HTTP/2. With more fundamental

[12]http://www.alexa.com/topsites

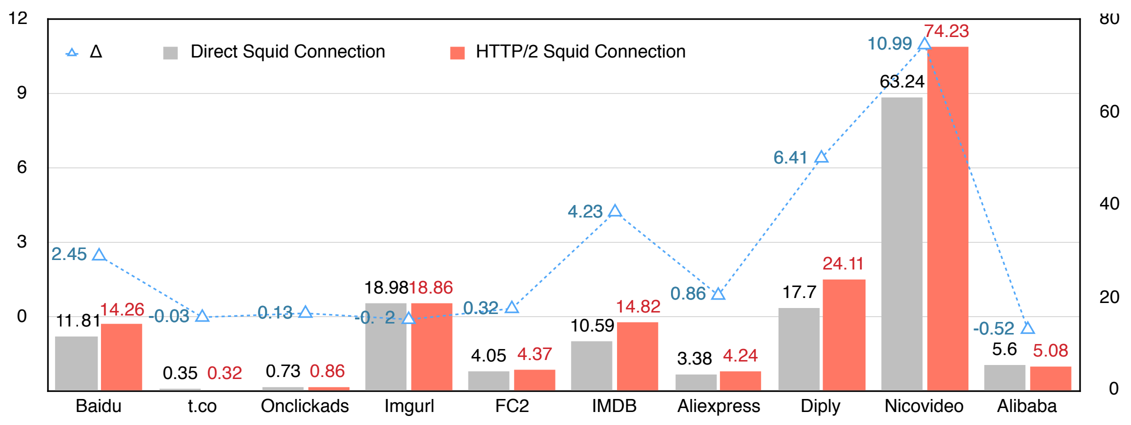


Figure 5.4: SPDY alike Proxy Performance with Different Sites

tools support in next few years, a native HTTP/2 cache proxy will finally launch in the market. However, we need a better solution for HTTPS proxy, because the proxy needs to inject the SSL protocol and decrypt the header information for further communication. A plain text based HTTP/2 proxy is not accepted by nowadays browser by default.

HTTP/2 proxy improvement is not universal. One of the reasons is the over header on the `nghttpx` translator module. The nghttpx adds an extra layer before Squid proxy and increased the message transaction time. Another reason is that the HTTPS connection will be redirected and break the HTTP/2 stream. The web pages contain other resource or HTTPS connection can be the trigger to break down the current HTTP/2 connection. From observation, most of the HTTPS connection loads at last after the HTTP/2 connection is completed. The last reason is third-party website resources, these resource does not get huge benefits from HTTP/2 because these requests are sent to third parties which connections need to be reestablished and *Dynamic Table* cannot be used in this scenario. For small web pages, enabling HTTP/2 may increase the latency. The similar result shows in [ABC+15] that Google's Flywheel gets benefits mostly from the data reducing, which is not the main study area of this thesis.

Single TCP connection may hurt the performance of the proxy. HTTP/2 only uses a single TCP connection. The performance relies on the reliability of that connection. Compare to the HTTP/1.1 connection which has multiple TCP connections, one suffers from poor connection will not affects the whole communication. The similar result has also been found in [dSOC15].

5.3 Evaluation of SANE with HTTP/2 Setting

SANE is a specified proxy program which processes clients' requests before sending to the destination server. Based on the methods mentioned in Section 4.4, viaSANE methods is created for the evaluation. The study of the performance improved by HTTP/2 in SANE separate in two tests: the first test is running on the local virtual machine to avoid the network latency, the second is functioning on remote server which simulate a more realistic surroundings. Finally, the security and anonymity consideration will be discussed in this section.

5.3.1 Feasibility Proof

The function of viaSANE is to simulate the basic function of SANE and simplify the data processing to mainly focus on HTTP/2 connection related factors. Adding HTTP/2 support in SANE is a fundamental which all the upper layers are based on this. The feasibility proof is formed in two parts: we need to prove that running SANE in a HTTP/2 supported framework can allow the client to communicate with SANE by using HTTP/2. Second part is to prove that SANE can communicate with Crowdsourcing Server using HTTP/2.

To debug the HTTP/2 communication and show a more readable proof, the communication between client and SANE is using HTTPS connection with SSL key injection. However, the communication between SANE and server is using HTTP plain text. The reason is that Wireshark cannot inject SSL key in cURL or other server library which is used in the implementation. To prove the concept and show the result, the destination server is configured to accept HTTP/2 protocol via plain text, which allows HTTP/1.1 upgrade to HTTP/2 protocol.

Listing 5.2 shows the HTTP/2 header between client and SANE (abbreviation H_{CS}), the *HyperText Transfer Protocol 2* indicates the protocol is using. The Listing only shows the data frame and the compressed data are repressed as commentary in the Listing.

```
HyperText Transfer Protocol 2
Stream: DATA, Stream ID: 13, Length 27
Length: 27
Type: DATA (0)
Flags: 0x01
  0... .... .... .... .... .... .... .... = Reserved: 0
   x00000000
  .000 0000 0000 0000 0000 0000 0000 1101 = Stream
   Identifier: 13
  [Pad Length: 0]
Data: 6d6574686f643d76696153414e45266d6573736167653d61
  //method: viaSANE; message: abcd
Padding: <MISSING>
```

Listing 5.2: Header of Client Sends POST Request to SANE

In Listing 5.3 exhibits the header information in HTTP/1.1 with upgrading between SANE and server (abbreviation H_{SS}). In H_{CS}, the post data is: `method: viaSANE`; `message: abcd` and it has been compressed with HPACK. After SANE accepted the post data, it hashes the incoming message, then stores it in the database.

After processing the `POST` request from client, SANE forwards the data to server using h2c protocol. The line `POST /request.php HTTP/1.1` is still showing `HTTP/1.1` since the connection is using h2c. The `POST` sends to the destination by HTTP/1.1 at first, after the client finds the server support h2c, the upgrading process continues. However, this process will be changed when using HTTPS connection, cause protocol negotiation process can be done during key exchange.

```
Hypertext Transfer Protocol
POST /request.php HTTP/1.1
...
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: AAMAAABkAAQAAP__
Content-Length: 87
Content-Type: application/x-www-form-urlencoded
  [Full request URI: http://127.0.0.1/request.php]
  [HTTP request 1/1]
  [Response in frame: 34]
HTML Form URL Encoded: application/x-www-form-urlencoded
  Form item: "method" = "viaSANE"
  Form item: "message" = "88d4266fd4e6338d13b845fcf289579d2
   09c897823b9217da3e161936f031589"
```

Listing 5.3: Header of SANE Forwards POST Request to Server

From the above Listings, the idea of HTTP/2 support in SANE is feasible. The PHP programs that running on HTTP/2 supported Apache can upgrade the connection between client and SANE to the newer protocol. From SANE to server, libcurl helps handle the HTTP/2 communications and can be easily integrated into current system frameworks.

5.3.2 Performance Evaluation

To study the performance of SANE enabling HTTP/2, two separate testbeds are set up: The local test environment operates client, SANE and server in the local virtual machine. The physical connection (T_P) approximates to 0 (which $T_P \approx 0$); For the remote performance test where the $T_P \neq 0$, the setup locates SANE in the local virtual machine and destination server in the remote host server.

For each test we run the similar test as Section 5.2.1, post data to the SANE though viaSANE method. After SANE forward the data successfully, SANE sends Header `200` code to notice the client request finished.

Local Performance

The test runs locally and sends POST request with following parameters: -n100 -c10, sends 100 requests with 10 concurrently clients; -n50 -c5, sends 50 requests with 5 concurrently clients; -n1 -c1, sends 1 request with 1 client. In Figure 5.5 shows the performance with different setups. The blue line with round dots dedicates the performance for 100 requests with 10 clients, the red line with square dots means the total finishing request time for 50 requests with 5 clients concurrently. The black line with triangle dots presents only 1 request. In the x axis coordinates, the variable before slash means the protocol, which using for front-end communication, the parameter after slash means the protocol that using for back-end communication. For example, H2/H2 means the connection between client and SANE is using HTTP/2 and same for the connection between SANE and server. This indicator also applies to Table 5.1.

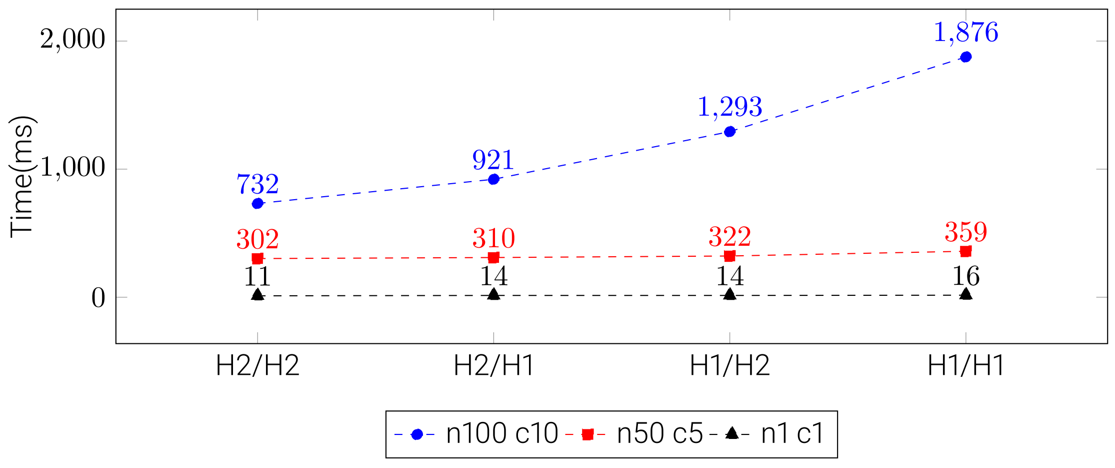


Figure 5.5: POST Request Performance in SANE with HTTP/2 Enabled

The figure above shows a noticeable improvement where using HTTP/2 connection in SANE compare to pure HTTP/1.1 connection. As can be seen, the pure HTTP/2 connection is over 2 times faster than the pure HTTP/1.1 communication when the 10 clients send 100 requests in total. In Table 5.2 lists the mean time consumption of 100 requests in each step during the communication. The definition of each index as following:

- **Connect.** The time taken to connect the server;
- **Request.** The time taken to receive full response after connection was opened;
- **1st Byte.** The time taken to get 1st byte from a server.

Time(ms)	H2/H2	H2/H1	H1/H2	H1/H1
Connect	17.80	13.33	14.84	18.09
Request	68.34	87.15	115.65	168.49
1st Byte	339.38	537.44	114.65	167.53

Table 5.2: Mean Time Consumption with -n100 -c10

From the table we notice that the connection time in different setup is almost the same, however the request time and 1st byte arrive time has larger difference[13].

In general, HTTP/1.1 performances much better than HTTP/2 in 1st Byte. When HTTP/2 is in use, one additional round cycle time for negotiation protocol adds to the 1st byte arrive time. As we can see from the table, H1/H2 surpass H1/H1, the reason is that the back-end connection is finished faster when using HTTP/2 on the back-end.

However, time to the first byte does not represent the overall performance. HTTP/1.1 starts a faster communication with SANE but the request finish time is much worse than HTTP/2. On the one hand, HTTP/2 reduces the header size, for example, the header is been compressed (reduced $\approx$ 89% from original request) on each request. On the other hand, HTTP/2 optimize the communication by enabling multiplexing. It avoids congestion and increases throughputs when large quantity requests occur.

REMOTE PERFORMANCE

To adapt a more realistic situation where $T_P \neq 0$, the following experiment locates the back-end server on remote server. In this situation, the connection time consumption between client and SANE is approximate to zero and the connection latency is different which depends on network connection speed. The test parameters are same as previous tests. However, in order to reduce the affects from network connection speed, each test setup runs 3 times and calculated the mean finish time as final results.

Overall. Figure 5.6 shows that HTTP/2 can improve the performance in SANE. H2/H2 connection still performance the best either in locally or remotely. Bringing HTTP/2 connection either in front-end side or back-end side can also improve the throughput and reduce the request time.

Enable HTTP/2 on slower connection side can bring more benefit. The result in Figure 5.6 shows a slightly different performance than the evaluation on local machine. The

[13]In the case of SANE, time to first byte (1st Byte) measure the time from SANE to the client.

H2/H1 connection has the worst performance than H1/H2. Since the connection between SANE and server has larger latency than the locally front-end connection. HTTP/2 protocol reduces the data traffic and allow faster concurrency connection. Enabling HTTP/2 on more time-consuming side can provide a better result.

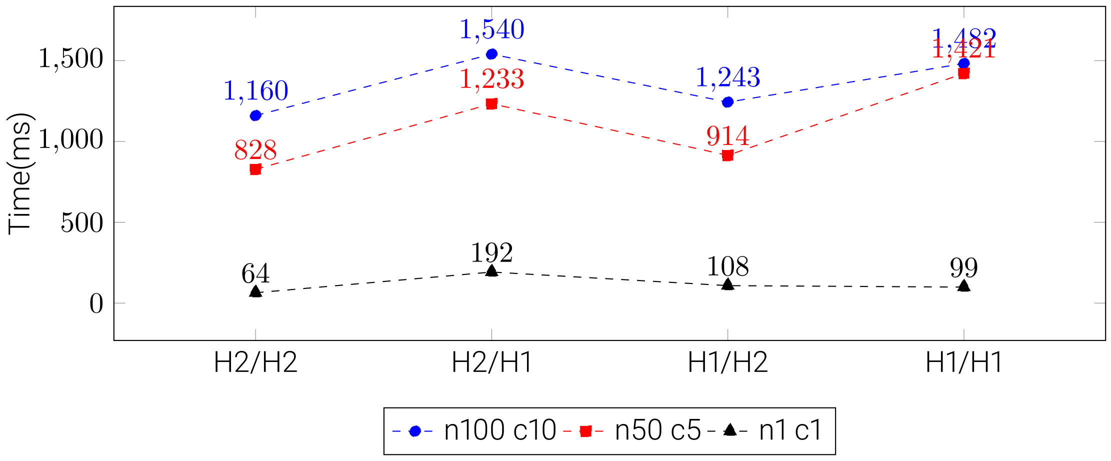


Figure 5.6: POST Request Performance in SANE with Remote Server

5.3.3 Security Consideration

The HTTP/2 accelerated the communication process. As a matter of fact, one of the most important features of SANE is to add security to the current crowdsourcing platform. After evaluating the mechanism of using the implementation the following security consideration should be noticed.

Anonymity and security maintain the same when using HTTP/2. SANE provides an anonymous and secure environment for Crowdfunder and Crowdsourcer. To enable this feature, SANE encrypt the data from crowdfunder and generate a new signature assign to the Crowdfunder. By using this mechanism, the security level depends on the strength of encryption algorithm and the safety of private key. By introducing the HTTP/2 communication into SANE will not effect these functionalities.

Enabling HTTP/2 can add more secure due to its nature. The viaSANE function only use HASH function to mask the incoming message from Crowdfunder, the same message has the same hashed value, this allows the attacker to recover incoming message using brute force. The HTTPS is a stronger and essential protocol to ensure the security. As specified in RFC7540 Section 9.2, implementations of HTTP/2 MUST use TLS version 1.2 or higher for HTTP/2 over TLS [BPT15]. A higher version of TLS protocol enforces security by replacing weaker algorithm in previous protocol and adds new mode to implement stronger encryption.

Enabling HTTP/2 may increase the risk of denial-of-service attack. An HTTP/2 connection can demand more resources to operate than an HTTP/1.1 connection. The header compression and other features of HTTP/2 can cause system resources consumption increasing dramatically with huge amounts of requests. Without monitoring the requests process, using HTTP/2 is more easily to cross over the capacity of system.

5.4 Summary

In this chapter, the hypothesis of header compression performance influence has been verified. The study on the compression ratio gives us a clearer idea how HTTP/2 leverage the size of header fields. The homologous requests from the same site can help to reduce large amount header overhead. The heterogeneous request shows an average header compression ratio, this proof the idea about using HTTP/2 can give benefit to the general web browsing behaves.

With proxy in HTTP/2 enabling, the test shows the different architecture can affect the performance improvements. Combining the experiments in SANE, it shows pure HTTP/2 proxy has the best performance, and in a cross-protocol environment replace the connection in the slower connection can give a better performance improvement. The test with Squid proxy, proof the feasibility of building a HTTP/2 proxy with current frameworks. However, the performance is suffered from the complexity of the Internet environments. It also proves the results mentioned in [WBKW14]: With HTTP/1.1 and HTTP/2 request at the same time, it will hurt the performance. Another consideration is HTTPS connection in proxy setting. Since the security nature of HTTP/2, it is hard to apply some proxy that can be fully trusted by clients and servers. One solution is explicit proxy described in [Peo12]. However, it is still a draft and the implementation is close to the protocol standard level.

The experiment in SANE shows how HTTP/2 can give us benefits on application-layer. The single TCP connection increases the throughput and the header compression decreases the overhead. The security can maintain the same lever without further changes.

6 Conclusion

After more than two decades, the HTTP/2, a brand new protocol for World Wide Web, is finally standardized and will change the future browsing experience during the next few years. However, it still needs several years to fully replace the network infrastructures across the Internet. The progress of HTTP/2 adaption is promising and accelerating with new software supports HTTP/2 every other day.

HTTP/2 protocol is a fundamental protocol for most Internet service infrastructures. Because the complexity of fully implementations, the development of HTTP/2 are dominated by main participants of HTTP/2 standard maker. Fortunately, upper layer applications that are based on HTTP/2 can benefit from the new standard without making many changes.

Since the change of HTTP/2 is dramatic from HTTP/1.1, there are only few proxy software programs that support HTTP/2 natively currently. One reason is the complexity of HTTP/2 standard, another is the encryption scheme needs a better solution with proxy setting which is suggested in [Peo12] and [JU12]. In order to support HTTP/2 in proxy at this moment, further adaption on both client-side and server-side are still needed. In most cases, the adaption process is seamless and compatible with most frameworks.

Since the new protocol has more complicity than its predecessor, it results most adaption mechanisms are based on the early developing library and third-party software. Adapting current frameworks to support HTTP/2 in proxy setting is possible either by adding extra translate layer or integrate with third-party tools.

The evaluation shows by enabling HTTP/2 can improve the communication efficiency and the profit is vast enough to make the efforts on this adaption.

6.1 HTTP/2 and Proxy

Header compression contributes most in performance improvement. Header compression is one of the most significant features of HTTP/2. Enabling header compression can reduce the header fields size, thus reduces the traffic bandwidth. HTTP/2 enabled header compression by default. The compressed header field is binary and the scheme is different from previous HTTP protocol. Due to the binary nature of HTTP/2 header, it is a barrier for application to migrate seamlessly and adds complexity to build a native HTTP/2 application. One way to handle this is by integrating third-party HTTP/2 supported library. The evaluation shows the performance improvement surpasses the performance degradation caused by introducing third-party library. Another assumption can be made from evaluation is that avoiding heterologous (requests from a different origin) can optimize the header compression.

HTTP/2 support is easy to adapt and integrate. HTTP/2 does not modify the application semantics of HTTP in any way. All the core concepts, such as HTTP methods, status codes, URIs, and header fields, remain the same as HTTP/1.1 [Gri13]. Currently, most software or third-party library already support HTTP/2 with limitation or near finish. It is easy to adapt system frameworks to support HTTP/2 either by upgrading themselves or integrated with third-party library. By choosing a mature and well-developed third-party library can limit the overall overhead.

Application supports HTTP/2 relies on underneath infrastructure. Since the core concepts are still the same in HTTP/2. It eases the burden for developer and keeps the developing process straightforward without considering system compatibility. For example, to support HTTP/2 in iOS system, developer can easily switch to an API which supports HTTP/2 by default.

Cross-protocol proxy is feasible and performance influenced by different architectures. Supporting HTTP/2 in proxy setting is a new topic when the release of HTTP/2 only a year ago. There is no full solution at this stage. One feasible way to adapt system frameworks to support HTTP/2 is to change the system architectures with cross-protocol. The solutions are:

- **Upgrade Proxy**. It supports HTTP/2 connection between client and proxy by adding HTTP Translate Module in between.
- **Downgrade Proxy.** In this architecture, in order to support HTTP/2 between proxy and server, the system framework integrates the HTTP Header Translator Module in between.

HTTP/2 enabled proxy gains performance improvement due to the header compression, multiplex streaming and other features brought by the new protocol. In cross-protocol proxy, the performance improvement is larger when using HTTP/2 on slower (has observable network latency) connection side. Although cross-protocol proxies compromise some features in HTTP/2, it still shows potential performance enhancement compare to pure HTTP/1.1 proxy. The research from Dropbox [dro16] uses HTTP/2 enabled Nginx to terminate SSL connection and perform load balancing. The results show the ingress traffic bandwidth was reduced nearly 50% when HTTP/2 is enabled on canary machines. This practical example proves that by using HTTP/2 in proxy can reduce the bandwidth and improve the performance.

Straightforward Proxy performance surpasses the cross-protocol proxy. It supports native HTTP/2 among client, server and proxy. Since Straightforward Proxy benefits from HTTP/2 on both client and server side, its performance overcomes the cross-protocol proxy in general. No doubts that the Straightforward Proxy will be the mainstream proxy type during the next few years when a better and fully HTTP/2 implemented software is launched on the market.

SPDY alike proxy benefits mostly from data compression not HTTP/2 in universal test environment. Before the Squid is developing its software to support HTTP/2[Jef] natively. By adapting the communication between client and proxy to support HTTP/2 can turn Squid to an SPDY alike proxy which architecture is similar to a Downgrade Proxy. The SPDY alike proxy adaption is a feasible example of how to make current framework support HTTP/2. In general, the result shows when introduced HTTP/2 connection into a complex network setup cannot bring a huge improvement to the system. One major problem is HTTPS connection handling. Since the SPDY alike proxy filters out all HTTPS connection and creates a tunnel between client and destination server when HTTPS is in use. On the other hand, the performance may also suffer from the single TCP connection feature of HTTP/2 sometimes.

6.2 SANE with HTTP/2 Support

Adapting SANE to support HTTP/2 is feasible. SANE is a memory modification proxy, which behaves differently from forward proxy or reverse proxy. It does not manipulate the requests from clients, instead, it acts as masker of client from its destination server to provide anonymity and robustness. The SANE acts as server from client's point of view and client from server's point of view. Two roles of SANE are separated and do not interfere each other. This characteristic helps the integration of HTTP/2 support process become easier. Adapting SANE to support HTTP/2 in server role can be achieved by upgrading the infrastructure that is running SANE. The client role part gains HTTP/2 support by integrating third-party library. In exception, if PHP supports native HTTP/2 methods, the third-party library can be omitted, then a better performance can be achieved.

SANE with HTTP/2 support can reduce traffic bandwidth and improve throughput. The analysis shows a promising performance benefit from using HTTP/2. It gains nearly two times performance in a Straightforward Proxy when requests concurrence and quantity are large. The same tendency also appears in the cross-protocol SANE. Header compress helps reduce a large amount of header size, speeds the overall request response time. Multiplexing solves the congestion problem of HTTP/1.1 and increases the throughput for concurrent request handling. Single TCP connection also contributes to the improvidence. Comparing to HTTP/1.1 establishes TCP connection on each request, HTTP/2 has fewer connections to maintain and start. Overall, adapting SANE to support HTTP/2 is worthwhile considering the effort puts in.

The security and anonymity of SANE do not jeopardize with HTTP/2 enabled. Since HTTP/2 security nature and does not alter the core concept and communication mechanism of SANE, the security and anonymity still can be guaranteed.

Caveat, for latency sensitive request should be careful about enabling HTTP/2 in the system. HTTP/2 has about one additional round trip time for getting the 1st byte from a server. When the system is sensitive about the response time and requests appear sporadically, enabling HTTP/2 on client-side is not always optimal.

6.3 Recommendations and Future Work

In the author's opinion, it is recommended to upgrade current system frameworks to support HTTP/2. Since the HTTP/2 is approved by IETF, there is no need for developers to worry about the compatibility issues and the trend is to support HTTP/2 for majority software in the future. Also, HTTP/2 protocol improves the network transaction efficiency and reduces the web page loading time, therefore applications and infrastructures will enjoy these perks!

However, due to the fact that most of the software provide HTTP/2 support are still under development and enable only a few HTTP/2 features. Majority web server software or third-party library such as Apache and nghttp2 are still in an experimental stage, performance may suffer from bugs and unknown issues, also some features of HTTP/2 are not fully supported. With a better support HTTP/2 in future, adapting system frameworks could be further developed in a number of ways:

Adapting Cache Proxy support HTTP/2 on back-end. A native HTTP/2 cache proxy are not available yet, and the most popular cache proxy, Squid, is still under developing [Jef]. With more underneath library release, there is a possibility to extend current SPDY alike proxy to support HTTP/2 on back-end. The performance improvement is promising for a straightforward cache proxy based on the study.

Adapting MapBiquios with HTTP/2 support. SANE is the prototype of INSANE which is used in MapBiquios system as the proxy of crowdsourcing platform. Other components are not supported HTTP/2 yet. Following the concept and method of adapting MapBiquious components to support HTTP/2 from Chapter 3 and 4, can bring MapBiquious performance to next level.

List of Figures

List of Tables

Listings

References

[ABC$^+$15] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel: Google's data compression proxy for the mobile web. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 367–380, 2015.

[BP12] Mike Belshe and Roberto Peon. Spdy protocol. 2012.

[BPT15] Mike Belshe, Roberto Peon, and M Thomson. Rfc 7540: hypertext transfer protocol version 2 (http/2), 2015.

[can15] Can i use... support tables for html5, css3, etc. `https://caniuse.com`, 2015.

[cur16] curl and libcurl. `https://curl.haxx.se/`, 2016.

[dro16] Enabling http/2 for dropbox web services: experiences and observations. `https://blogs.dropbox.com/tech/2016/05/enabling-http2-for-dropbox-web-services-experiences-and-observations/`, 2016.

[dSOC15] Hugues de Saxce, Iuniana Oprescu, and Yiping Chen. Is http/2 really faster than http/1.1? In *Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on*, pages 293–299. IEEE, 2015.

[FGM$^+$06] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999. *RFC2616*, 2006.

[FLP14] Stephan Friedl, Adam Langley, and Andrey Popov. Transport layer security (tls) application-layer protocol negotiation extension. *Transport*, 2014.

[Gri13] Ilya Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, Inc., 2013.

[GT02] David Gourley and Brian Totty. *HTTP: the definitive guide*. O'Reilly Media, Inc., 2002.

[HSBS13] Tenshi Hara, Thomas Springer, Gerd Bombach, and Alexander Schill. Decentralised approach for a reusable crowdsourcing platform utilising standard web servers. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 1063–1074. ACM, 2013.

[Jef] Amos Jeffries. Feature: Http/2.0 support. `http://wiki.squid-cache.org/Features/HTTP2`.

[jet16] Jetty - servlet engine and http server. `http://www.eclipse.org/jetty/`, 2016.

[JU12] Jeff Jarmoc and DSCT Unit. Ssl/tls interception proxies and transitive trust. *Black Hat Europe*, 2012.

[LA94] Ari Luotonen and Kevin Altis. World-wide web proxies. *Computer Networks and ISDN systems*, 27(2):147–154, 1994.

[Mem15] Faisal Memon. Open source nginx 1.9.5 released with http/2 support. `https://www.nginx.com/blog/nginx-1-9-5/`, 2015.

[net16] February 2016 web server survey. `http://news.netcraft.com/archives/2016/02/22/february-2016-web-server-survey.html`, 2016.

[nsu13] Nsurlsession class reference. `https://developer.apple.com/library/ios/documentation/Foundation/Reference/NSURLSession_class/`, 2013.

[Peo12] Roberto Peon. Explicit proxies for http/2.0. 2012.

[PR15] R Peon and H Ruellan. Hpack: Header compression for http/2. Technical report, 2015.

[Sai11] Kulbir Saini. *Squid Proxy Server 3.1: Beginner's Guide*. Packt Publishing Ltd, 2011.

[Spr11] Thomas Springer. Mapbiquitous–an approach for integrated indoor/outdoor location-based services. In *Mobile Computing, Applications, and Services*, pages 80–99. Springer, 2011.

[Ste] Daniel Stenberg. http2 explained. `http://http2-explained.haxx.se/content/en/part11.html`.

[tom16] Apache tomcat 8 (8.5.2) - documentation index. `http://tomcat.apache.org/tomcat-8.5-doc/`, 2016.

[Tsu15] Tatsuhiro Tsujikawa. Nghttp2: Http/2 c library - nghttp2.org. `https://nghttp2.org/`, 2015.

[WBKW14] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How speedy is spdy? In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 387–399, 2014.