RFC 7230

Hypertext Transfer Protocol (HTTP/1.1) : Message Syntax and Routing

문서 최종 수정일	2020-05-26
원문 복사일	2020-04-06
번역 및 정리	이병록(roka88)
이메일	roka88.dev@gmail.com

PROPOSED STANDARD
Errata Exist
Updated By: 8615

Internet Engineering Task Force (IETF)
Request for Comments: 7230
Obsoletes: 2145, 2616
Updates: 2817, 2818
Category: Standards Track
ISSN: 2070-1721

R. Fielding, Ed.
Adobe
J. Reschke, Ed.
greenbytes
June 2014

Hypertext Transfer Protocol (HTTP/1.1) : Message Syntax and Routing

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document provides an overview of HTTP architecture and its associated terminology, defines the "http" and "https" Uniform Resource Identifier (URI) schemes, defines the HTTP/1.1 message syntax and parsing requirements, and describes related security concerns for implementations.

하이퍼텍스트 전송 프로토콜 (HTTP)는 분산, 협업, 하이퍼텍스트 정보 시스템에 대한 상태 없는 애플리케이션-레벨의 프로토콜이다. 이 문서에서는 HTTP 아키텍처 및 관련 용어에 대한 개요를 제공하고,"http" 및 "https" Uniform Resource Identifier(URI) scheme을 정의하며, HTTP/1.1 메시지 구문 및 구문 분석 요구 사항을 정의하고, 구현과 관련된 보안 문제를 설명한다.

Status of This Memo

This is an Internet Standards Track document.

이것은 인터넷 표준 추적 문서이다.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

이 문서는 Internet Engineering Task Force(IETF)의 제품이다. 문서는 IETF 공동체의 합의를 나타낸다. 문서는 공개 검토를 받아왔으며 Internet Engineering Starting Group (IESG)에 의해 발행 승인를 받았다. 인터넷 표준의 추가 정보는 RFC 5741 Section 2에서 확인할 수 있다.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7230.

이 문서에 대한 현재 상태 정보는 정오표와 피드백을 어떻게 제공하는 방법은 http://www.rfc-editor.org/info/rfc7230 에서 얻을 수 있다.

Copyright Notice

Table of Contents

1. Introduction

The Hypertext Transfer Protocol (HTTP) is a stateless application-level request/response protocol that uses extensible semantics and self-descriptive message payloads for flexible interaction with network-based hypertext information systems. This document is the first in a series of documents that collectively form the HTTP/1.1 specification:

1. “Message Syntax and Routing” (해당 문서)

2. “Semantics and Content” [RFC7231]

3. “Conditional Requests” [RFC7232]

4. “Range Requests” [RFC7233]

5. “Caching” [RFC7234]

6. “Authentication” [RFC7235]

하이퍼텍스트 전송 프로토콜 (HTTP)는 네트워크 기반 하이퍼텍스트 정보시스템과 유연한 상호 작용을 위한 확장 가능한 의미론 및 자체-설명 메시지 페이 로드를 사용하는 상태 없는 애플리케이션-레벨 요청/응답 프로토콜이다. 이 문서는 HTTP/1.1의 구성하는 문서 시리즈 중 첫 번째이다.

This HTTP/1.1 specification obsoletes RFC 2616 and RFC 2145 (on HTTP versioning). This specification also updates the use of CONNECT to establish a tunnel, previously defined in RFC 2817, and defines the "https" URI scheme that was described informally in RFC 2818.

이 HTTP/1.1 명세는 RFC 2616과 RFC 2145를 더 이상 사용하지 않는다. (HTTP 버저닝에서) 이 명세는 또한 이전에 RFC 2817에 정의된 터널 설립에 CONNECT 사용을 업데이트하고, RFC 2818에 비공식적으로 설명된 “https” URI scheme을 정의한다.

HTTP is a generic interface protocol for information systems. It is designed to hide the details of how a service is implemented by presenting a uniform interface to clients that is independent of the types of resources provided. Likewise, servers do not need to be aware of each client's purpose: an HTTP request can be considered in isolation rather than being associated with a specific type of client or a predetermined sequence of application steps. The result is a protocol that can be used effectively in many different contexts and for which implementations can evolve independently over time.

HTTP는 정보 시스템을 위한 일반적인 인터페이스 프로토콜이다. HTTP는 제공된 리소스 유형에서 독립인 클라이언트에게 어떻게 균일한 서비스가 구현 되는지 자세한 내용이 숨겨지도록 설계되었다. 마찬가지로, 서버는 각 클라이언트의 목적을 인식할 필요가 없다: HTTP 요청은 특정 클라이언트 또는 미리 결정된 일련의 애플리케이션 단계 순서와 관련되어 별개로 고려될 수 있다. 여러 다른 맥락에서 효과적으로 사용될 수 있고 구현은 시간이 지남에 따라 독립적으로 발전 할 수 있는 프로토콜이다.

HTTP is also designed for use as an intermediation protocol for translating communication to and from non-HTTP information systems. HTTP proxies and gateways can provide access to alternative information services by translating their diverse protocols into a hypertext format that can be viewed and manipulated by clients in the same way as HTTP services.

HTTP는 비 HTTP 정보 시스템간에 통신 변환을 위한 중개 프로토콜으로써 사용하기 위해 설계되었다. HTTP 프락시 및 게이트웨이는 다양한 프로토콜을 HTTP 서비스와 동일한 방법으로 클라이언트가 보고 조작할 수 있는 하이퍼텍스트 형식으로 변환하여 대체 정보 서비스에 대한 접근을 제공할 수 있다.

One consequence of this flexibility is that the protocol cannot be defined in terms of what occurs behind the interface. Instead, we are limited to defining the syntax of communication, the intent of received communication, and the expected behavior of recipients. If the communication is considered in isolation, then successful actions ought to be reflected in corresponding changes to the observable interface provided by servers. However, since multiple clients might act in parallel and perhaps at cross-purposes, we cannot require that such changes be observable beyond the scope of a single response.

이러한 유연성의 결과 중 하나는 프로토콜이 인터페이스 뒤에서 일어나는 일들의 관점에서 정의될 수 없다는 것이다. 대신, 우리는 수신된 통신 및 수신인의 예상된 동작, 통신 구문을 정의하는 것으로 제한된다. 만약에 통신이 독립적으로 고려되고, 성공적인 조치들이 서버들에 의해 제공된 식별할 수 있는 인터페이스의 해당 변경에 반영되어야 한다. 그러나, 여러 클라이언

트들이 서로 다른 목적으로 동시에 또는 서로 다른 용도로 행동할 수 있고, 우리가 그러한 변경들을 단일 응답의 범위를 넘어서 관찰 할 수 있도록 요구 할 수 없다.

This document describes the architectural elements that are used or referred to in HTTP, defines the "http" and "https" URI schemes, describes overall network operation and connection management, and defines HTTP message framing and forwarding requirements. Our goal is to define all of the mechanisms necessary for HTTP message handling that are independent of message semantics, thereby defining the complete set of requirements for message parsers and message-forwarding intermediaries.

이 문서는 HTTP에서 사용되거나 참조되는 아키텍처 요소를 설명하고,"http" 및 "https"URI 구문을 정의하며, 전체 네트워크 운영 및 커넥션 관리를 설명하고, HTTP 메시지 프레임 및 전달 요구 사항을 정의한다. 우리의 목표는 메시지 의미론과 독립적인 HTTP 메시지 처리에 필요한 모든 메커니즘을 정의하여 메시지 분석기 및 메시지 전달 중개자에 대한 전체 요구 사항을 정의하는 것이다.

1.1 요구사항 표기

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

이 문서의 해당 키워드 “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” 들은 RFC2119 에 설명되어 있다.

Conformance criteria and considerations regarding error handling are defined in Section 2.5.

불일치 기준 및 오류 처리와 관련된 고려 사항은 Section 2.5에 정의되어 있다.

1.2 Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7, that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix B shows the collected grammar with all list operators expanded to standard ABNF notation.

이 규격은 Section 7에 정의된 '#' 연산자를 사용하여 쉼표로 구분된 리스트를 콤팩트하게 정의할 수 있는 ('*' 연산자가 반복을 나타내는 방식과 유사) 목록 확장자가 있는 [RFC5234] Augmented Backus_Naur Form (ABNF) 표기법을 사용한다. 부록 B는 표준 ABNF 표기법으로 확장된 모든 목록 연산자와 함께 수집된 문법을 보여준다.

The following core rules are included by reference, as defined in [RFC5234], Appendix B.1: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), HTAB (horizontal tab), LF (line feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any visible [USASCII] character).

[RFC5234], Appendix B.1: ALPHA(letters), CR(carriage return), CRLF(CR LF), CTL(controls), DIGIT(십진수 0-9), DQUote(double quote), HEXDIG(십육진법 최소 0-9/AF-), HTAB (horizontal tab), LF (line feed), OCTET (모든 8-bit 연속 데이터), SP(space), VCHAR([USASCII] 문자)에서 정의한 대로 참조에 의해 다음과 같은 핵심 규칙이 포함된다.

As a convention, ABNF rule names prefixed with "obs-" denote "obsolete" grammar rules that appear for historical reasons.

관례로, “obs-“ 이 앞에 붙은 ABNF 규칙 이름은 역사적 이유로 나타나는 “obsolete” 문법 규칙을 나타낸다.

2. Architecture

HTTP was created for the World Wide Web (WWW) architecture and has evolved over time to support the scalability needs of a worldwide hypertext system. Much of that architecture is reflected in the terminology and syntax productions used to define HTTP.

HTTP는 World Wide Web (WWW) 아키텍처를 위해 만들어 졌고, 시간이 지남에 따라 월드와이드 하이퍼텍스트 시스템의 확장성에 대한 요구들을 지원하기 위해 발전했다. 아키텍처의 대부분이 용어와 HTTP 정의를 위해 사용되는 구문들에 반영되었다.

2.1 Client/Server Messaging

HTTP is a stateless request/response protocol that operates by exchanging messages (Section 3) across a reliable transport- or session-layer "connection" (Section 6). An HTTP "client" is a program that establishes a connection to a server for the purpose of sending one or more HTTP requests. An HTTP "server" is a program that accepts connections in order to service HTTP requests by sending HTTP responses.

HTTP는 신뢰성 있는 전송 계층 또는 세션 계층 “connection”의 (Section 6) 메시지 교환 (Section 3)에 의해 작동되는 상태없는 요청/응답 프로토콜이다. HTTP “client(이하 클라이언트)” 는 하나 이상의 HTTP 요청들을 전송하기 위한 목적으로 커넥션을 서버에 설립하는 프로그램이다. HTTP “server(이하 서버)”는 HTTP 요청을 처리하고 HTTP 응답을 보내기 위해 커넥션을 허용하는 프로그램이다.

The terms "client" and "server" refer only to the roles that these programs perform for a particular connection. The same program might act as a client on some connections and a server on others. The term "user agent" refers to any of the various client programs that initiate a request, including (but not limited to) browsers, spiders (web-based robots), command-line tools, custom applications, and mobile apps. The term "origin server" refers to the program that can originate authoritative responses for a given target resource. The terms "sender" and "recipient" refer to any implementation that sends or receives a given message, respectively.

“client”와 “server”라는 용어들은 특정 커넥션에 대한 프로그램 수행의 역할만 의미한다. 동일한 프로그램이 일부 커넥션에서는 클라이언트 역할을 하고 다른 커넥션에서는 서버 역할을 할 수 있다. “user agent(이하 사용자 에이전트)” 용어는 브라우저, 스파이더 (웹 기반 로봇), 커맨드 라인 툴, 커스텀 응용 프로그램, 그리고 모바일 앱을 포함해서 요청을 시작하는 다양한 클라이언트 프로그램을 말한다. “origin server(이하 원서버)” 용어는 특정 대상 리소스에 대한 권한 있는 응답을 생성할 수 있는 프로그램이다.
“sender(이하 발신자)” 와 “recipient(이하 수신자)” 각각 용어는 주어진 메시지를 전송하고 수신하는 모든 구현된 것을 의미한다.

HTTP relies upon the Uniform Resource Identifier (URI) standard [RFC3986] to indicate the target resource (Section 5.1) and relationships between resources. Messages are passed in a format similar to that used by Internet mail [RFC5322] and the Multipurpose Internet Mail Extensions (MIME) [RFC2045] (see Appendix A of

[RFC7231] for the differences between HTTP and MIME messages).

HTTP는 대상 리소스와 리소스들간의 관계를 나타내기 위해서 URI 표준 [RFC3986]에 의존한다. 인터넷 메일 [RFC 5322] 과 Multipurpose Internet Mail Extensions (MIME) [RFC2045]에 의해 사용된 유사한 포멧으로 메시지는 전송된다. (HTTP와 MIME 메시지 사이의 차이점을 알고 싶다면 Appendix A of [RFC7231]를 참조)

Most HTTP communication consists of a retrieval request (GET) for a representation of some resource identified by a URI. In the simplest case, this might be accomplished via a single bidirectional connection (===) between the user agent (UA) and the origin server (O).

URI에 의해 식별되는 일부 리소스 묘사를 위해 대부분의 HTTP 통신은 검색 요청 (GET)으로 구성 되어있다. 가장 간단한 경우로 사용자 에이전트 (UA)와 원서버 (O) 사이에 단일 양방향 커넥션을 통해 이 작업을 수행할 수 있다.

```
     request >
UA ======================= O
                    < response
```

A client sends an HTTP request to a server in the form of a request message, beginning with a request-line that includes a method, URI, and protocol version (Section 3.1.1), followed by header fields containing request modifiers, client information, and representation metadata (Section 3.2), an empty line to indicate the end of the header section, and finally a message body containing the payload body (if any, Section 3.3).

클라이언트는 HTTP 요청을 서버에 요청 메시지 형식, method, URI와 프로토콜 버전 (Section 3.1.1) 를 포함한 request-line을 시작으로, 헤더 필드들, 요청 수정자, 클라이언트 정보, 표현 메타데이터 (Section 3.2), 헤더 부분의 끝을 나타내는 빈 줄을 포함하여, 마지막으로 페이로드 본문을 포함하는 메시지 본문(만약에 있다면, Section 3.3)을 보낸다.

A server responds to a client's request by sending one or more HTTP response messages, each beginning with a status line that includes the protocol version, a success or error code, and textual reason phrase (Section 3.1.2), possibly followed by header fields containing server information, resource metadata, and representation metadata (Section 3.2), an empty line to indicate the end of the

header section, and finally a message body containing the payload body (if any, Section 3.3).

서버는 클라이언트의 요청에 하나 또는 다수의 HTTP 응답 메시지를 각각의 프로토콜 버전, 성공 또는 에러 코드, 원문으로된 상태 코드 (Section 3.1.2) 를 포함한 status-line을 시작으로, 가능하다면 서버 정보, 리소스 메타데이터와 표현 메타데이터 (Section 3.2)를 포함하는 헤더 필드 다음에 헤더 부분의 끝을 나타내는 빈 줄을 포함하여, 마지막으로 페이로드 body를 포함한 메시지 본문(만약에 있다면, Section 3.3)을 통해 응답한다.

A connection might be used for multiple request/response exchanges, as defined in Section 6.3.

Section 6.3에 정의된 대로, connection은 다수의 요청/응답 교환을 위해 사용될 수 있다.

The following example illustrates a typical message exchange for a GET request (Section 4.3.1 of [RFC7231]) on the URI "http://www.example.com/hello.txt":

다음 예시는 URI 에서 GET 요청을 위한 일반적인 메시지 교환을 보여준다.

"http://www.example.com/hello.txt":

클라이언트 요청:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Lanuage: en, mi
```

서버 응답:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

2.2 Implementation Diversity

When considering the design of HTTP, it is easy to fall into a trap of thinking that all user agents are general-purpose browsers and all origin servers are large public websites. That is not the case in practice. Common HTTP user agents include household appliances, stereos, scales, firmware update scripts, command-line programs, mobile apps, and communication devices in a multitude of shapes and sizes. Likewise, common HTTP origin servers include home automation units, configurable networking components, office machines, autonomous robots, news feeds, traffic cameras, ad selectors, and video-delivery platforms.

HTTP 설계를 고려할 때, 모든 사용자 에이전트들이 범용 브라우저들이고 모든 원 서버들이 큰 규모의 공용 웹사이트라는 생각의 함정에 빠지는 것이 쉽다. 실제론 아니다. 보통의 HTTP 사용자 에이전트는 가정 애플리케이션, 음향기기, 체중계, 스크립트를 갱신하는 firmware, command-line 프로그램, 모바일 앱, 그리고 다양한 모양과 크기의 통신 기기를 포함한다. 마찬가지로, 보통의 HTTP 원 서버는 홈 자동화 장치, 네트워킹을 구성하는 부품, 사무용 기계, 자율 로봇, 뉴스 피드, 트래픽 카메라, 광고 선택자, 그리고 비디오-배송 플랫폼을 포함한다.

The term "user agent" does not imply that there is a human user directly interacting with the software agent at the time of a request. In many cases, a user agent is installed or configured to run in the background and save its results for later inspection (or save only a subset of those results that might be interesting or erroneous). Spiders, for example, are typically given a start URI and configured to follow certain behavior while crawling the Web as a hypertext graph.

“user agent(이하 사용자 에이전트)” 용어는 요청시 소프트웨어 에이전트와 직접적으로 상호작용하는 사용자를 의미하지 않는다. 대부분의 경우, 사용자 에이전트는 백그라운드에서 운영되어 나중에 검사할 수 있도록 설치되거나 구성 되어있다.(또는 에러또는 흥미로울지 모르는 결과들의 부분집합을 위해 결과들을 저장). 예시로 스파이더는 일반적으로 시작 URI가 주어지고, 웹을 하이퍼텍스트 그래프로 크롤링하는 동안 특정 행동을 따르도록 구성되었다.

The implementation diversity of HTTP means that not all user agents can make interactive suggestions to their user or provide adequate warning for security or privacy concerns. In the few cases where this specification requires reporting of errors to the user, it is acceptable for such reporting to only be observable in an error console or log file. Likewise, requirements that an automated action be

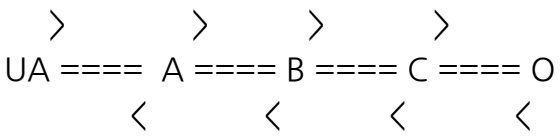
confirmed by the user before proceeding might be met via advance configuration choices, run-time options, or simple avoidance of the unsafe action; confirmation does not imply any specific user interface or interruption of normal processing if the user has already made that choice.

HTTP의 구현 다향성은 모든 사용자 에이전트가 사용자에게 대화형 제안을 하거나 보안 또는 개인 정보 보호 문제에 대한 적절한 경고를 제공할 수 있는 것은 아님을 의미한다. 이 명세가 오류를 보고하는 것은 클라이언트에게 요구하는 몇 안되는 경우이고, 그런 보고는 에러 콘솔 또는 로그 파일에서만 관찰할 수 있다. 다시말해, 사용자가 작업을 진행하기 전에 자동화된 작업을 확인해야 하는 요구 사항은 사전 구성 선택, 런타임 옵션 또는 안전하지 않은 작업을 간단하게 방지하는 것으로 충족될 수 있다; 확인은 사용자가 이미 해당 작업을 선택한 경우 특정 사용자 인터페이스 또는 정상적인 처리 중단을 의미하지는 않는다.

2.3 Intermediaries

HTTP enables the use of intermediaries to satisfy requests through a chain of connections. There are three common forms of HTTP intermediary: proxy, gateway, and tunnel. In some cases, a single intermediary might act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.

HTTP는 중개자를 이용하여 커넥션 체인을 통해 요청들을 충족할 수 있다. HTTP 중개자로 세 가지의 일반적인 형태가 있다: 프락시, 게이트웨이, 터널.
경우에 따라, 단일 중개자는 원서버, 프락시, 게이트웨이 또는 터널, 각 요청의 특성에 따라 전환 동작으로 행동할 수도 있다.



The figure above shows three intermediaries (A, B, and C) between the user agent and origin server. A request or response message that travels the whole chain will pass through four separate connections. Some HTTP communication options might apply only to the connection with the nearest, non-tunnel neighbor, only to the endpoints of the chain, or to all connections along the chain. Although the diagram is linear, each participant might be engaged in multiple, simultaneous communications. For example, B might be receiving requests from many clients other than A, and/or forwarding requests to servers other than C, at the same time that it is handling A's request. Likewise, later requests might be sent through a

different path of connections, often based on dynamic configuration for load balancing.

위의 그림은 사용자 에이전트와 원서버 사이에 3개의 중개자(A, B와 C)를 보여준다. 전체 체인을 이동하는 요청 또는 응답 메시지는 4개의 분리된 커넥션을 통해 전송된다. 일부 HTTP 통신 옵션들은 가장 가까운 터널이 아닌 인접 네트워크와의 커넥션에만 적용되거나 체인의 끝점에만 적용되거나 체인을 따라 있는 모든 커넥션에 적용될 수 있다. 다이어그램은 선형이지만, 각 참여자는 여러개의 동시 통신에 참여할 수 있다. 예를들어 B는 A외에 여러 클라이언트로부터 요청을 받을 수 있거나 또는 같은 시간에 A의 요청을 다루면서 C이외에 서버로 요청을 전송할 수 있다. 마찬가지로 나중의 요청들은 로드 밸런싱을 통한 동적인 구성에 기반하여 커넥션의 다른 경로를 통해 전송질 것이다.

The terms "upstream" and "downstream" are used to describe directional requirements in relation to the message flow: all messages flow from upstream to downstream. The terms "inbound" and "outbound" are used to describe directional requirements in relation to the request route: "inbound" means toward the origin server and "outbound" means toward the user agent.

“upstream(이하 업스트림)”과 “downstream(이하 다운스트림)”의 용어들은 메시지 흐름의 관계에서 방향 요구사항들을 설명하기 위해 사용된다: 모든 메시지들은 "upstream"으로 부터 "downstream"으로 흐른다. "inbound"와 "outbound"의 용어는 요청 라우트의 관계에서 방향 요구사항들을 설명하기 위해 사용된다: "inbound"는 원서버로 향하는 것을 의미하고, "outbound"는 사용자 에이전트로 향하는 것을 의미한다.

A "proxy" is a message-forwarding agent that is selected by the client, usually via local configuration rules, to receive requests for some type(s) of absolute URI and attempt to satisfy those requests via translation through the HTTP interface. Some translations are minimal, such as for proxy requests for "http" URIs, whereas other requests might require translation to and from entirely different application-level protocols. Proxies are often used to group an organization's HTTP requests through a common intermediary for the sake of security, annotation services, or shared caching. Some proxies are designed to apply transformations to selected messages or payloads while they are being forwarded, as described in Section 5.7.2.

“proxy(이하 프락시)“는 클라이언트에 선택되고 일반적으로 로컬 구성 규칙들을 통해 일부 absolute URI의 유형을 위해 요청들을 받기 위한 메시지-전송 에이전트이다. 그리고 HTTP 인터페이스를 통하여 변환을 통해 요청을 충족을 시도한다. 프락시를 위한 “http” URI에 대한 프락시 요청과 같은 일부 변환은 최소 이며, 반면에 다른 요청에는 완전히 다른 애플리케이션-레벨 프로토콜 간에 변환할 수 있다. 프락시들은 종종 보안적 이익, 주석 서비스, 또는 공유 캐시를 위한 공통 중개자로 조직의 HTTP 요청들을 그룹화 하는데 자주 사용된다. 일부 프락시들

은 선택된 메시지 또는 페이로드가 전송되는 동안 변환을 적용하기 위해 설계되었다. Section 5.7.2에 설명되어있다.

A "gateway" (a.k.a. "reverse proxy") is an intermediary that acts as an origin server for the outbound connection but translates received requests and forwards them inbound to another server or servers.
Gateways are often used to encapsulate legacy or untrusted information services, to improve server performance through "accelerator" caching, and to enable partitioning or load balancing of HTTP services across multiple machines.

“gateway(이하 게이트웨이)”는 (“reverse proxy”로 알려진) 아웃바운드 커넥션을 위한 원서버로써 역할을 하지만, 수신된 요청을 다른 서버로 변환하여 인바운드로 전송하는 중간 서버 역할을 한다. 게이트웨이는 오래된 또는 신뢰 할 수 없는 정보 서비스들을 캡슐화하는데 사용되며, 서버의 성능을 향상시키기 위해 “accelerator” 캐싱, 그리고 파티셔닝 또는 여러 시스템 간의 HTTP 서비스의 로드 밸런싱을 할 수 있게 사용된다.

All HTTP requirements applicable to an origin server also apply to the outbound communication of a gateway. A gateway communicates with inbound servers using any protocol that it desires, including private extensions to HTTP that are outside the scope of this specification. However, an HTTP-to-HTTP gateway that wishes to interoperate with third-party HTTP servers ought to conform to user agent requirements on the gateway's inbound connection.

원서버에 적당한 모든 HTTP 요구사항은 게이트웨이의 아웃바운드 통신에도 적용된다. 게이트웨이는 private extensions(개인적으로 프로토콜을 확장한 것들)을 이 명세의 범위 밖의 HTTP에 포함하여 원하는 프로토콜을 사용하여 인바운드 서버와 통신한다. 그러나, 서드파티 HTTP 서버들과 상호작용하는 HTTP-to-HTTP 게이트웨이는 게이트웨이의 인바운드 커넥션에 대한 사용자 에이전트의 요구사항에 준수해야 한다.

A "tunnel" acts as a blind relay between two connections without changing the messages. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel might have been initiated by an HTTP request. A tunnel ceases to exist when both ends of the relayed connection are closed. Tunnels are used to extend a virtual connection through an intermediary, such as when Transport Layer Security (TLS, [RFC5246]) is used to establish confidential communication through a shared firewall proxy.

“tunnel(이하 터널)”은 메시지 변경없이 2개의 커넥션에 숨겨진 중개자로써 역할을 한다. 활성되면, HTTP 요청에 의해 시작되었을 수 있지만, 터널은 HTTP 통신의 당사자로 고려되지 않는다. 두 끝점에서 전달하는 커넥션이 종료되면 터널은 더 이상 존재하지 않는다. 공유 방화

벽 프락시를 통해 기밀 통신을 설정하는 데 전송층 보안(TLS, [RFC5246])이 사용되는 경우와 같이 중계를 통해 가상 커넥션을 확장하는 데 터널이 사용된다.

The above categories for intermediary only consider those acting as participants in the HTTP communication. There are also intermediaries that can act on lower layers of the network protocol stack, filtering or redirecting HTTP traffic without the knowledge or permission of message senders. Network intermediaries are indistinguishable (at a protocol level) from a man-in-the-middle attack, often introducing security flaws or interoperability problems due to mistakenly violating HTTP semantics.

중개자를 위한 범주는 HTTP 통신의 참가자로써 역할만 고려된다. 또한 네트워크 프로토콜 스택의 하위 계층에서 행동하여 메시지 보낸 사람의 알려진 것이나 허가 없이 HTTP 트래픽을 필터링 하거나 리다이렉트 할 수 있는 중개자도 있다. 네트워크 중개자들은 중간자 공격(프로토콜 단계에서)을 구분할 수 없으며, 실수로 HTTP 의미론을 위반하여 보안 결함 또는 상호 운용성 문제를 야기하는 경우가 많다.

For example, an "interception proxy" [RFC3040] (also commonly known as a "transparent proxy" [RFC1919] or "captive portal") differs from an HTTP proxy because it is not selected by the client. Instead, an interception proxy filters or redirects outgoing TCP port 80 packets (and occasionally other common port traffic). Interception proxies are commonly found on public network access points, as a means of enforcing account subscription prior to allowing use of non-local Internet services, and within corporate firewalls to enforce network usage policies.

예를 들어, “interception proxy” [RFC3040] (마찬가지로 일반적으로 “transparent proxy”로 알려진 [RFC1919] 또는 “captive portal”)는 클라이언트에 의해서 선택되지 않았기 때문에 HTTP 프락시와 다르다. 대신에, 가로채기 프락시는 외부의 TCP 포트 80 패킷(그리고 가끔 다른 일반적인 포트 트래픽)을 거르거나 또는 리다이렉트 한다.
가로채기 프락시는 일반적으로 로컬이 아닌 인터넷 서비스 사용을 허용하기 전에 계정 가입을 적용하는 수단으로 공용 네트워크 액세스 지점에서, 그리고 네트워크 사용 정책을 시행하기 위해 회사 방화벽 내에서 발견된다.

HTTP is defined as a stateless protocol, meaning that each request message can be understood in isolation. Many implementations depend on HTTP's stateless design in order to reuse proxied connections or dynamically load balance requests across multiple servers. Hence, a server MUST NOT assume that two requests on the same connection are from the same user agent unless the connection is secured and specific to that agent. Some non-standard HTTP extensions (e.g., [RFC4559]) have

been known to violate this requirement, resulting in security and interoperability problems.

HTTP는 상태 없는 프로토콜로 정의되었고. 각 요청 메시지는 개별적으로 이해될 수 있다는 것을 의미한다. 대부분의 구현은 프락시된 커넥션을 재사용하거나 또는 여러 서버간에 동적으로 로드 밸런스 요청을 하기 위해 HTTP의 상태 없는 설계에 의존한다. 따라서 서버는 커넥션이 보안되어 있고 해당 에이전트에 특정되어 있지 않는한 동일한 커넥션에 대한 두개의 요청이 동일한 사용자 에이전트에서 온 것으로 가정해서는 안 된다.(MUST NOT) 일부 비표준 HTTP 확장들은 (e.g., [RFC4559]) 이 요구사항을 위반하는 것으로 알려져있고, 보안 및 상호작용 문제를 발생시킨다.

2.4 Caches

A "cache" is a local store of previous response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server MAY employ a cache, though a cache cannot be used by a server while it is acting as a tunnel.

“cache(이하 캐시)”는 이전 응답 메시지의 로컬 보관소이고 메시지의 저장, 검색, 삭제를 관리하는 서브 시스템이다. 캐시는 캐시 가능한 응답을 저장하여 향후 동일한 요청에 대한 응답 시간과 네트워크 대역폭 사용을 줄인다. 캐시는 서버가 터널 역할을 하는 동안에는 사용될 수 없지만, 어느 클라이언트 또는 서버는 캐시를 사용할 수 있다.(MAY)

The effect of a cache is that the request/response chain is shortened if one of the participants along the chain has a cached response applicable to that request. The following illustrates the resulting chain if B has a cached copy of an earlier response from O (via C) for a request that has not been cached by UA or A.

캐시의 효과는 체인의 참여자 중 하나가 해당 요청에 적용할 수 있는 캐시된 응답을 가지고 있는 경우 요청/응답 체인이 단축되는 것이다. 다음은 B가 UA 또는 A에 의해 캐시되지 않은 요청에 대한 O(경유 C)의 이전 응답의 캐시된 사본을 가지고 있는 경우의 결과 체인을 보여준다.

```
   >           >
UA ====  A ==== B - - - - C - - - -  O
         <           <
```

A response is "cacheable" if a cache is allowed to store a copy of the response message for use in answering subsequent requests. Even when a response is cacheable, there might be additional constraints placed by the client or by the origin server on when that cached response can be used for a particular request. HTTP requirements for cache behavior and cacheable responses are defined in Section 2 of [RFC7234].

캐시가 후속 요청에 응답하기 위해 응답 메시지의 복사본을 저장할 수 있는 경우 응답은 "cacheable" 이다. 응답을 캐시할 수 있는 경우에도 캐시된 응답을 특정 요청에 사용할 수 있는 경우 클라이언트 또는 원서버에 의해 추가 제약이 있을 수 있다. 캐시 동작 및 캐시 가능한 응답에 대한 HTTP 요건은 [RFC7234]의 Section 2에 정의되어 있다.

There is a wide variety of architectures and configurations of caches deployed across the World Wide Web and inside large organizations. These include national hierarchies of proxy caches to save transoceanic bandwidth, collaborative systems that broadcast or multicast cache entries, archives of pre-fetched cache entries for use in off-line or high-latency environments, and so on.

월드 와일드 웹 과 대규모 조직에 걸쳐 배치된 캐시의 설계와 구성은 매우 다양하다. 대양을 횡단하는 대역폭을 아끼기 위한 프락시 캐시의 국가 계층, 브로드캐스트 또는 멀티캐스트 캐시 항목을 저장하는 협업 시스템, 오프라인 또는 high-latency 환경들에서 사용하기 위한 pre-fetched 캐시 항목 보관소 등이 포함된다.

2.5 Conformance and Error Handling

This specification targets conformance criteria according to the role of a participant in HTTP communication. Hence, HTTP requirements are placed on senders, recipients, clients, servers, user agents, intermediaries, origin servers, proxies, gateways, or caches, depending on what behavior is being constrained by the requirement. Additional (social) requirements are placed on implementations, resource owners, and protocol element registrations when they apply beyond the scope of a single communication.

이 명세는 HTTP 통신의 참여자의 규칙에 따라 적합성 기준을 대상으로 한다. 따라서, HTTP 요구사항은 어떤 행동이 요구사항에 의해 제한되고 있는지에 따라서 발신자, 수신자, 클라이언트, 서버, 사용자 에이전트, 중개자, 원서버, 프락시, 게이트웨이, 또는 캐시가 배치되어 있다. 구현, 리소스 소유자 및 프로토콜 요소 등록이 단일 통신 범위를 벗어나 적용되는 경우 추가 (관행적) 요구 사항이 적용된다.

The verb "generate" is used instead of "send" where a requirement differentiates between creating a protocol element and merely forwarding a received element downstream.

동사 “generate”는 프로토콜 요소를 생성하는 것과 단지 수신된 요소를 다운스트림에서 전달하는 것 사이에서 요구사항이 구별되는 “send” 대신 사용된다.

An implementation is considered conformant if it complies with all of the requirements associated with the roles it partakes in HTTP.

구현은 HTTP에 맞는 규칙들과 관련된 모든 요구사항들을 따른다면, 적합한 것으로 간주된다.

Conformance includes both the syntax and semantics of protocol elements. A sender MUST NOT generate protocol elements that convey a meaning that is known by that sender to be false. A sender MUST NOT generate protocol elements that do not match the grammar defined by the corresponding ABNF rules. Within a given message, a sender MUST NOT generate protocol elements or syntax alternatives that are only allowed to be generated by participants in other roles (i.e., a role that the sender does not have for that message).

적합성은 구문과 프로토콜 요소의 의미론을 둘 다 포함한다. 발신자는 발신자가 거짓이라는 알려진 의미를 전달하는 프로토콜 요소를 생성하면 안 된다.(MUST NOT) 발신자는 ABNF 규칙에 대응하는 정의된 문법에 맞지 않는 프로토콜 요소를 생성하면 안 된다.(MUST NOT) 지정된 메시지 내에서, 발신자는 다른 역할(i.e., 해당 메시지에 발신자에게 없는 역할)의 참가자만 생성할 수 있는 프로토콜 요소나 구문 대안을 생성하면 안 된다.(MUST NOT)

When a received protocol element is parsed, the recipient MUST be able to parse any value of reasonable length that is applicable to the recipient's role and that matches the grammar defined by the corresponding ABNF rules. Note, however, that some received protocol elements might not be parsed. For example, an intermediary forwarding a message might parse a header-field into generic field-name and field-value components, but then forward the header field without further parsing inside the field-value.

수신된 프로토콜 요소가 분석되었을 때, 수신자는 수신자의 역할에 해당하는 적정한 길이의 값과 분석할 수 있어야 하며 ABNF 규칙들에 대응하는 정의된 문법에 일치해야 한다.(MUST) 참고로, 그러나 일부 수신된 프로토콜 요소들은 분석이 되지 않을 수 있다. 예를들어 메시지를 전송하는 중개자는 일반 field-name과 field-value 성분들의 header-field를 분석할 수 있고, field-value의 추가적인 분석 없이 헤더 필드를 전송하는 것도 있다.

HTTP does not have specific length limitations for many of its protocol elements because the lengths that might be appropriate will vary widely, depending on the deployment context and purpose of the implementation. Hence, interoperability between senders and recipients depends on shared expectations regarding what is a reasonable length for each protocol element. Furthermore, what is commonly understood to be a reasonable length for some protocol elements has changed over the course of the past two decades of HTTP use and is expected to continue changing in the future.

HTTP는 구현의 문맥 배치와 목적에 따라 적절한 길이가 매우 다양하기 때문에 대부분의 프로토콜 요소에 대해 특정한 길이 제한을 가지고 있지 않는다. 따라서, 발신자와 수신자 사이의 상호작용은 각 프로토콜 요소를 위한 적정한 길이가 무엇인지에 관한 공유된 예상에 의존한다. 게다가 일부 프로토콜 요소에 대해 일반적으로 적절한 길이로 이해되는 것은 지난 20년 동안 HTTP 사용 과정에서 바뀌었으며 앞으로도 계속 변화하는 것이 예상된다.

At a minimum, a recipient MUST be able to parse and process protocol element lengths that are at least as long as the values that it generates for those same protocol elements in other messages. For example, an origin server that publishes very long URI references to its own resources needs to be able to parse and process those same references when received as a request target.

최소한, 수신자는 다른 메시지의 동일한 프로토콜 요소에 대해 생성한 값만큼 프로토콜 요소 길이를 구문 분석하고 처리할 수 있어야 한다.(MUST) 예를 들어 자체 리소스에 매우 긴 URI 참조를 게시하는 원서버는 요청 대상으로 수신될 때 동일한 참조를 구문 분석하고 처리할 수 있어야 한다.

A recipient MUST interpret a received protocol element according to the semantics defined for it by this specification, including extensions to this specification, unless the recipient has determined (through experience or configuration) that the sender incorrectly implements what is implied by those semantics. For example, an origin server might disregard the contents of a received Accept-Encoding header field if inspection of the User-Agent header field indicates a specific implementation version that is known to fail on receipt of certain content codings.

수신자는 수신자가 해당 의미론에 의해 암시된 것을 잘못 구현한다고 (경험이나 구성을 통해) 판단하지 않는 한, 수신자는 이 명세에 정의된 의미론에 따라 수신된 프로토콜 요소를 해석해야 한다.(MUST) 예를 들어, “User-Agent” 헤더 필드 검사에서 특정 콘텐츠 코딩을 수신할 때 실패한 것으로 알려진 특정 구현 버전을 나타내는 경우 원서버는 수신된 Accept-Encoding 헤더 필드의 내용을 무시할 수 있다.

Unless noted otherwise, a recipient MAY attempt to recover a usable protocol element from an invalid construct. HTTP does not define specific error handling mechanisms except when they have a direct impact on security, since different applications of the protocol require different error handling strategies. For example, a Web browser might wish to transparently recover from a response where the Location header field doesn't parse according to the ABNF, whereas a systems control client might consider any form of error recovery to be dangerous.

달리 명시되지 않은 한, 수신자는 잘못된 구조에서 사용 가능한 프로토콜 요소를 복구하려고 시도할 수 있다.(MAY) HTTP는 보안에 직접적인 영향을 미치는 경우를 제외하고 특정 오류 처리 메커니즘을 정의하지 않는데, 프로토콜의 서로 다른 응용 프로그램에는 다른 오류 처리 전략이 필요하기 때문이다. 예를 들어, 웹 브라우저는 Location 헤더 필드가 ABNF에 따라 구문 분석되지 않는 응답에서 투명하게 복구하기를 원하는 반면, 시스템 제어 클라이언트는 어떤 형태의 오류 복구도 위험하다고 간주할 수 있다.

2.6 Protocol Versioning

HTTP uses a "<major>.<minor>" numbering scheme to indicate versions of the protocol. This specification defines version "1.1". The protocol version as a whole indicates the sender's conformance with the set of requirements laid out in that version's corresponding specification of HTTP.

HTTP는 “<major>.<minor>”를 프로토콜의 버전의 번호를 표시하기 위해 사용한다. 이 규격은 버전 "1.1"을 정의한다. 프로토콜 버전 전체는 발신자가 해당 버전의 HTTP사양에 명시된 요건 집합을 준수하고 있음을 나타낸다.

The version of an HTTP message is indicated by an HTTP-version field in the first line of the message. HTTP-version is case-sensitive.

HTTP 메시지 버전은 메시지의 첫번째 행에 HTTP-version 필드로 표시된다. HTTP-version은 대 소문자를 구분한다.

```
HTTP-Version = HTTP-name “/“ DIGIT “.” DIGIT
HTTP-name = %x48.54.54.50 ; "HTTP", case-sensitive
```

The HTTP version number consists of two decimal digits separated by a "." (period or decimal point). The first digit ("major version") indicates the HTTP messaging syntax, whereas the second digit ("minor version") indicates the highest minor version within that major version to which the sender is conformant and able to understand for future communication. The minor version advertises the sender's communication capabilities even when the sender is only using a backwards-compatible subset of the protocol, thereby letting the recipient know that more advanced features can be used in response (by servers) or in future requests (by clients).

HTTP 버전 번호는 "."(마침표 또는 소수 점)로 구분된 두개의 10진수로 구성돈다. 첫번째 숫자("major 버전")는 HTTP 메시징 구문을 나타내는 반면, 두번째 숫자("minor 버전")는 발신자에 적합하고 이후 통신을 이해할 수 있는 해당 major 버전 내에서 가장 높은 minor 버전을 나타낸다. minor 버전은 발신자가 역으로 호환되는 프로토콜의 하위 집합만 사용하는 경우에도 발신자의 통신 기능을 알리므로 수신자는 더 많은 고급 기능을 응답(서버에 의해)또는 향후 요청(클라이언트에 의해)에 사용할 수 있음을 알 수 있다.

When an HTTP/1.1 message is sent to an HTTP/1.0 recipient [RFC1945] or a recipient whose version is unknown, the HTTP/1.1 message is constructed such that it can be interpreted as a valid HTTP/1.0 message if all of the newer features are ignored. This specification places recipient-version requirements on some new features so that a conformant sender will only use compatible features until it has determined, through configuration or the receipt of a message, that the recipient supports HTTP/1.1.

HTTP/1.1 메시지가 HTTP/1.0 수신자 [RFC1945] 또는 버전을 알 수 없는 수신자에게 전송될 때, HTTP/1.1 메시지는 새로운 기능이 모두 무시될 경우 유효한 HTTP/1.0 메시지로 해석될 수 있도록 구성된다. 이 명세는 수신자가 HTTP/1.1을 지원하는 것을 구성 또는 메시지 수신을 통해 결정할 때까지, 준수 발신자가 호환되는 기능만 사용할 수 있도록 recipient-version 요구 사항을 일부 새로운 기능에 배치한다.

The interpretation of a header field does not change between minor versions of the same major HTTP version, though the default behavior of a recipient in the absence of such a field can change. Unless specified otherwise, header fields defined in HTTP/1.1 are defined for all versions of HTTP/1.x. In particular, the Host and Connection header fields ought to be implemented by all HTTP/1.x implementations whether or not they advertise conformance with HTTP/1.1.

헤더 필드의 해석은 동일한 major HTTP 버전과 minor 버전 간에 변경되지 않지만 이러한 필드가 없는 경우 수신자의 기본 동작은 변경될 수 있다. 별도로 지정하지 않는 한, HTTP/1.1에 정의된 헤더 필드는 모든 버전의 HTTP/1.x에 대해 정의된다. 특히, Host와 Connection 헤더 필드는 HTTP/1.1준수 여부에 관계 없이 모든 HTTP/1.x구현에 의해 구현되어야 한다.

New header fields can be introduced without changing the protocol version if their defined semantics allow them to be safely ignored by recipients that do not recognize them. Header field extensibility is discussed in Section 3.2.1.

새로운 헤더 필드는 프로토콜 버전을 변경하지 않고 도입될 수 있다. 새로운 헤더 필드의 정의된 의미론은 그것들을 인식하지 못하는 수신자들에 의해 새로운 헤더 필드를 안전하게 무시할 수 있게 해 준다. 헤더 필드 확장성은 Section 3.2.1에 설명되어 있다.

Intermediaries that process HTTP messages (i.e., all intermediaries other than those acting as tunnels) MUST send their own HTTP-version in forwarded messages. In other words, they are not allowed to blindly forward the first line of an HTTP message without ensuring that the protocol version in that message matches a version to which that intermediary is conformant for both the receiving and sending of messages. Forwarding an HTTP message without rewriting the HTTP-version might result in communication errors when downstream recipients use the message sender's version to determine what features are safe to use for later communication with that sender.

HTTP 메시지를 처리하는 중개자(즉, 터널 역할을 하는 중개자를 제외한 모든 중개자)는 전달된 메시지로 자신의 HTTP-version을 전송해야 한다.(MUST) 다시 말해서, 그들은 메시지의 프로토콜 버전이 메시지 수신과 발신 모두에 적합한 버전과 일치하는지 확인하지 않고 HTTP메시지의 첫번째 줄(request-line, response-line)을 맹목적으로 전달하는 것이 허용되지 않는다. HTTP-version은 다운 스트림 수신자가 메시지 보낸 사람의 버전을 사용하여 나중에 해당 발신자의 통신에 사용할 수 있는 기능을 결정할 때 통신 오류를 초래할 수 있다.

A client SHOULD send a request version equal to the highest version to which the client is conformant and whose major version is no higher than the highest version supported by the server, if this is known. A client MUST NOT send a version to which it is not conformant.

클라이언트의 major 버전이 서버에 의해 지원하는 가장 높은 버전보다 높지 않고 클라이언트가 가장 높은 버전을 호환하는 것으로 알려져 있다면 클라이언트는 가장 높은 버전과 동일한 요청버전을 전송해야 한다.(SHOULD) 클라이언트는 적합하지 않은 버전을 보내면 안된다.(MUST NOT)

A client MAY send a lower request version if it is known that the server incorrectly implements the HTTP specification, but only after the client has attempted at least one normal request and determined from the response status code or header fields (e.g., Server) that the server improperly handles higher request versions.

서버가 HTTP 규격을 잘못 구현한 것으로 알려진 경우, 그러나 클라이언트가 적어도 하나의 정상적인 요청을 시도하고 서버가 상위 요청 버전을 잘못 처리하는 응답 상태 코드 또는 헤더 필드(e.g., 서버)를 통해 확인한 후에만 클라이언트가 하위 요청 버전을 보낼 수 있다.(MAY)

A server SHOULD send a response version equal to the highest version to which the server is conformant that has a major version less than or equal to the one received in the request. A server MUST NOT send a version to which it is not conformant. A server can send a 505 (HTTP Version Not Supported) response if it wishes, for any reason, to refuse service of the client's major protocol version.

서버는 서버가 구성된 가장 높은 버전과 동일한 응답 버전을 전송해야 하며 major 버전은 요청에 수신된 버전보다 작거나 같아야 한다.(SHOULD) 서버는 호환되지 않는 버전을 보내면 안 된다.(MUST NOT) 어떤 이유로든 서버는 클라이언트의 주요 프로토콜 버전 서비스를 거부하고자 하는 경우 505(HTTP Version Not Supported)의 응답을 보낼 수 있다.

A server MAY send an HTTP/1.0 response to a request if it is known or suspected that the client incorrectly implements the HTTP specification and is incapable of correctly processing later version responses, such as when a client fails to parse the version number correctly or when an intermediary is known to blindly forward the HTTP-version even when it doesn't conform to the given minor version of the protocol. Such protocol downgrades SHOULD NOT be performed unless triggered by specific client attributes, such as when one or more of the request header fields (e.g., User-Agent) uniquely match the values sent by a client known to be in error.

클라이언트가 HTTP 사양을 잘못 구현한 것으로 알려지거나 의심되는 경우(예:클라이언트가 버전 번호를 올바르게 구문 분석하지 못하거나 중개자가 프로토콜의 minor 버전이 주어진 것을 따르지 않을 때 HTTP 버전을 맹목적으로 전달하는 것으로 알려진 경우) 서버가 요청에 대해 HTTP/ 1.0 응답을 보낼 수 있다.(MAY) 하나 이상의 요청 헤더 필드(e.g., User-Agent)가 오류가 있는 것으로 알려진 클라이언트가 전송한 값과 고유하게 일치하는 경우와 같은 특정 클라이언트 속성에 의해 트리거 되지 않는 한 이러한 프로토콜 다운그레이드를 수행해서는 안 된다.(SHOULD NOT)

The intention of HTTP's versioning design is that the major number will only be incremented if an incompatible message syntax is introduced, and that the minor number will only be incremented when changes made to the protocol have the

effect of adding to the message semantics or implying additional capabilities of the sender. However, the minor version was not incremented for the changes introduced between [RFC2068] and [RFC2616], and this revision has specifically avoided any such changes to the protocol.

HTTP의 버전 관리 설계의 목적은 호환되지 않는 메시지 구문이 도입될 때만 큰 숫자가 증가하며, 프로토콜 변경이 메시지 의미론을 추가하거나 발신자의 추가 기능을 의미할 때만 작은 숫자가 증가한다는 것이다. 단, [RFC2068]과 [RFC2616]사이에 도입된 변경 사항에 대해서는 minor 버전이 증가되지 않았으며, 본 개정은 프로토콜에 대한 그러한 변경을 특별히 피했다.

When an HTTP message is received with a major version number that the recipient implements, but a higher minor version number than what the recipient implements, the recipient SHOULD process the message as if it were in the highest minor version within that major version to which the recipient is conformant. A recipient can assume that a message with a higher minor version, when sent to a recipient that has not yet indicated support for that higher version, is sufficiently backwards-compatible to be safely processed by any implementation of the same major version.

HTTP메시지가 수신자가 구현한 major 버전 번호로 수신되지만 수신자가 구현한 버전보다 높은 minor 버전 번호로 수신되면 수신자는 해당 메시지를 수신자가 확인할 수 있는 major 버전 내에서 가장 높은 major 버전에 있는 것처럼 처리해야 한다.(SHOULD) 수신자는 메시지가 해당 상위 버전에 대한 지원을 아직 표시하지 않은 수신자에게 전송되면 동일한 major 버전의 구현에서 안전하게 처리할 수 있을 만큼 충분히 역 호환성이다.

2.7 Uniform Resource Identifiers

Uniform Resource Identifiers (URIs) [RFC3986] are used throughout HTTP as the means for identifying resources (Section 2 of [RFC7231]). URI references are used to target requests, indicate redirects, and define relationships.

Uniform Resource Identifiers (URIs) [RFC3986]은 리소스를 식별하는 수단으로 HTTP 전체에서 사용된다.([RFCC7231]의 Section 2) URI 참조는 요청 대상 지정, 리다이렉트 표시 및 관계 정의에 사용된다.

The definitions of "URI-reference", "absolute-URI", “relative-part", "scheme", "authority", "port", "host", "path-abempty", “segment", "query", and "fragment" are adopted from the URI generic syntax. An "absolute-path" rule is defined for

protocol elements that can contain a non-empty path component. (This rule differs slightly from the path-abempty rule of RFC 3986, which allows for an empty path to be used in references, and path-absolute rule, which does not allow paths that begin with "//".) A "partial-URI" rule is defined for protocol elements that can contain a relative URI but not a fragment component.

"URI-reference", "absolute-URI", “relative-part", "scheme", "authority", "port", "host", "path-abempty", “segment", "query", 와 “fragment"는 정의는 URI 일반 구문에서 채택되었다. “absolute-path”의 규칙은 non-empty 경로 구성을 포함할 수 있는 프로토콜 요소를 위해 정의 되었다. (참조에서 사용되는 empty 경로를 허가하는 RFC 3986의 path-abempty 규칙과 “//“를 시작하는 경로에 허가하지 않는 path-absolute 규칙은 약간 다르다.) “partial-URI” 규칙은 relative URI를 포함할 수 있으나 부분은 포함할 수 없는 프로토콜 요소를 위해 정의되었다.

```
URI-reference = <URI-reference, see [RFC3986], Section 4.1>
absolute-URI  = <absolute-URI, see [RFC3986], Section 4.3>
relative-part = <relative-part, see [RFC3986], Section 4.2>
scheme        = <scheme, see [RFC3986], Section 3.1>
authority     = <authority, see [RFC3986], Section 3.2>
uri-host      = <host, see [RFC3986], Section 3.2.2>
port          = <port, see [RFC3986], Section 3.2.3>
path-abempty  = <path-abempty, see [RFC3986], Section 3.3>
segment       = <segment, see [RFC3986], Section 3.3>
query         = <query, see [RFC3986], Section 3.4>
fragment      = <fragment, see [RFC3986], Section 3.5>

absolute-path = 1*( "/" segment )
partial-URI   = relative-part [ "?" query ]
```

Each protocol element in HTTP that allows a URI reference will indicate in its ABNF production whether the element allows any form of reference (URI-reference), only a URI in absolute form (absolute-URI), only the path and optional query components, or some combination of the above. Unless otherwise indicated, URI references are parsed relative to the effective request URI (Section 5.5).

URI 참조를 허용하는 HTTP의 각 프로토콜 요소는 ABNF 생산에 요소가 유일한 절대 형태의 URI (absolute-URI), 유일한 경로 및 선택적 쿼리 구성 요소, 또는 일부 위의 조합들 같은 어

떤 형태의 참조(URI-reference)든 허용되는지 여부를 명시한다. 달리 명시되지 않은 경우, URI 참조는 유효한 요청 URI (Section 5.5) 비교하여 분석된다.

2.7.1 http URI scheme

The "http" URI scheme is hereby defined for the purpose of minting identifiers according to their association with the hierarchical namespace governed by a potential HTTP origin server listening for TCP ([RFC0793]) connections on a given port.

지정된 포트에서 TCP ([RFC0793]) 커넥션을 청취하는 잠재적 HTTP 원서버에 의해 관리되는 계층 네임 스페이스에 따라 식별자를 해석하기 위한 목적으로 “http” URI scheme이 정의되어 있다.

```
http-URI = "http:" "//" authority path-abempty [ "?" query ]
           [ "#" fragment ]
```

The origin server for an "http" URI is identified by the authority component, which includes a host identifier and optional TCP port ([RFC3986], Section 3.2.2). The hierarchical path component and optional query component serve as an identifier for a potential target resource within that origin server's name space. The optional fragment component allows for indirect identification of a secondary resource, independent of the URI scheme, as defined in Section 3.5 of [RFC3986].

“http" URI를 위한 원서버는 호스트 식별자와 선택적 TCP 포트를 포함하는 권한 구성 요소로 식별된다. ([RFC3986], Section 3.2.2). 계층 경로 구성 요소 및 선택적 쿼리 구성 요소는 해당 원서버의 네임 스페이스 내에서 잠재적인 대상 리소스의 식별자 역할을 한다. 선택적 단편화 구성 요소를 사용하면 [RFC3986]의 Section 3.5에 정의된 URI scheme과 독립적으로 보조 리소스를 간접적으로 식별할 수 있다.

A sender MUST NOT generate an "http" URI with an empty host identifier. A recipient that processes such a URI reference MUST reject it as invalid.

발신자는 호스트 식별자가 비어있는 상태로 “http" URI를 생성해서는 안 된다.(MUST NOT)이러한 URI 참조를 처리하는 수신자는 유효하지 않은 것으로 거부해야 한다.(MUST)

If the host identifier is provided as an IP address, the origin server is the listener (if any) on the indicated TCP port at that IP address. If host is a registered name, the registered name is an indirect identifier for use with a name resolution service, such as DNS, to find an address for that origin server. If the port subcomponent is empty or not given, TCP port 80 (the reserved port for WWW services) is the default.

호스트 식별자가 IP 주소로 제공되는 경우 원서버는 해당 IP주소의 지정된 TCP 포트에서 청취자(있는 경우)가 된다. 호스트가 등록된 이름인 경우 등록된 이름은 해당 원서버의 주소를 찾기 위해 DNS와 같은 이름 확인 서비스와 함께 사용할 간접 식별자이다. 포트 하위 구성 요소가 비어 있거나 지정되지 않은 경우 TCP 포트 80(WWW 서비스용 예약 포트)이 기본 값이다.

Note that the presence of a URI with a given authority component does not imply that there is always an HTTP server listening for connections on that host and port. Anyone can mint a URI. What the authority component determines is who has the right to respond authoritatively to requests that target the identified resource. The delegated nature of registered names and IP addresses creates a federated namespace, based on control over the indicated host and port, whether or not an HTTP server is present. See Section 9.1 for security considerations related to establishing authority.

참고로 주어진 권한 구성 요소와 함께 URI가 있다고 해서 해당 호스트 및 포트에서 항상 HTTP 서버가 커넥션을 청취한다는 의미는 아니다. 누구나 URI 를 만들 수 있다. 권한 구성 요소는 누가 식별된 리소스를 대상으로 하는 요청에 대해 인증적으로 응답할 권한이 있는지를 결정한다. 등록된 이름 및 IP주소의 위임된 특성은 HTTP 서버의 존재 여부에 관계 없이 지정된 호스트 및 포트에 대한 제어를 기반으로 결합 네임 스페이스를 생성한다. 권한 설정과 관련된 보안 고려 사항은 Section 9.1을 참조해라.

When an "http" URI is used within a context that calls for access to the indicated resource, a client MAY attempt access by resolving the host to an IP address, establishing a TCP connection to that address on the indicated port, and sending an HTTP request message (Section 3) containing the URI's identifying data (Section 5) to the server. If the server responds to that request with a non-interim HTTP response message, as described in Section 6 of [RFC7231], then that response is considered an authoritative answer to the client’s request.

지정된 리소스에 대한 접근을 요구하는 컨텍스트 내에서 “http”URI를 사용하는 경우 클라이언트는 IP주소에 대한 호스트를 확인하고 지정된 포트의 해당 주소에 TCP 커넥션을 설정하고

URI의 식별 데이터(Section 5)가 포함된 HTTP요청 메시지(Section 3)를 서버에 보내 접근 시도할 수 있다. 서버가 Section 6 [RFC7231]에 설명된 non-interim HTTP 응답 메시지와 요청을 응답 한다면, 그 응답은 클라이언트 요청에 대한 신뢰할 수 있는 답변으로 간주된다.

Although HTTP is independent of the transport protocol, the “http" scheme is specific to TCP-based services because the name delegation process depends on TCP for establishing authority. An HTTP service based on some other underlying connection protocol would presumably be identified using a different URI scheme, just as the “https" scheme (below) is used for resources that require an end-to-end secured connection. Other protocols might also be used to provide access to "http" identified resources -- it is only the authoritative interface that is specific to TCP.

HTTP는 전송 프로토콜과 독립적이지만, 이름 위임 프로세스는 권한을 설정하는 TCP에 의존하기 때문에 “http” scheme은 TCP기반 서비스에만 한정된다. “https” scheme(아래)이 종단간 보안 커넥션이 필요한 리소스에 사용되는 것처럼, 일부 다른 기본 케넥션 프로토콜에 기반한 HTTP서비스도 다른 URI scheme을 사용하여 식별될 수 있다. “http”로 식별된 리소스에 대한 접근을 제공하기 위해 다른 프로토콜을 사용할 수도 있다. — 이것은 TCP에만 해당하는 권한 있는 인터페이스이다.

The URI generic syntax for authority also includes a deprecated userinfo subcomponent ([RFC3986], Section 3.2.1) for including user authentication information in the URI. Some implementations make use of the userinfo component for internal configuration of authentication information, such as within command invocation options, configuration files, or bookmark lists, even though such usage might expose a user identifier or password. A sender MUST NOT generate the userinfo subcomponent (and its "@" delimiter) when an "http" URI reference is generated within a message as a request target or header field value. Before making use of an "http" URI reference received from an untrusted source, a recipient SHOULD parse for userinfo and treat its presence as an error; it is likely being used to obscure the authority for the sake of phishing attacks.

권한에 대한 URI일반 구문에는 URI에 사용자 인증 정보를 포함하기 위해 더 이상 사용되지 않는 userinfo 하위 구성 요소([RFC3986], Section 3.2.1)도 포함된다. 일부 구현에서는 명령 호출 옵션, 구성 파일, 등의 인증 정보의 내부 구성에 userinfo 구성 요소를 사용한다. 또는 북마크 목록을 표시할 수도 있다. 이러한 사용으로 인해 사용자 ID또는 암호가 노출될 수도 있다. 메시지 내에서 “http” URI 참조가 요청 대상 또는 헤더 필드 값으로 생성되는 경우 보낸 사람은 userinfo 하위 구성 요소(및 해당"@"구분 기호)를 생성해서는 안 된다.(MUST NOT) 신뢰할 수 없는 소스에서 수신한 “http"URI 참조를 사용하기 전에 수신자는 userinfo에 대해 구문 분석하고 이 참조의 존재를 오류로 간주해야 한다.(SHOULD); 피싱 공격을 위해 권한을 숨기는 데 사용될 가능성이 높다.

2.7.2 https URI Scheme

The "https" URI scheme is hereby defined for the purpose of minting identifiers according to their association with the hierarchical namespace governed by a potential HTTP origin server listening to a given TCP port for TLS-secured connections ([RFC5246]).

“https” URI scheme는 잠재적으로 HTTP 원서버가 주어진 TLS-보안 커넥션을 위한 TCP 포트를 청취하는 것으로 계층적으로 관리된 네임 스페이스의 연계에 따라 식별자를 주조하는 목적으로 정의 된다.

All of the requirements listed above for the "http" scheme are also requirements for the "https" scheme, except that TCP port 443 is the default if the port subcomponent is empty or not given, and the user agent MUST ensure that its connection to the origin server is secured through the use of strong encryption, end-to-end, prior to sending the first HTTP request.

위의 "http" scheme에 대한 모든 요구 사항도 "https" scheme에 대한 요구 사항이다. TCP 포트 443이 포트 하위 구성 요소가 비어 있거나 지정되지 않은 경우를 제외하고, 기본값이며, 그리고 사용자 에이전트는 첫 번째 HTTP 요청을 보내기 전에 강력한 암호화 사용, end-to-end, 통해 원서버에 대한 커넥션이 안전한지 확인해야 한다.(MUST)

```
https-URI = "https:" "//" authority path-abempty [ "?" query ]
                    [ "#" fragment ]
```

Note that the "https" URI scheme depends on both TLS and TCP for establishing authority. Resources made available via the “https" scheme have no shared identity with the "http" scheme even if their resource identifiers indicate the same authority (the same host listening to the same TCP port). They are distinct namespaces and are considered to be distinct origin servers. However, an extension to HTTP that is defined to apply to entire host domains, such as the Cookie protocol [RFC6265], can allow information set by one service to impact communication with other services within a matching group of host domains.

참고로 “https” URI scheme는 권한을 설립하기 위해 TLS와 TCP 모두에 따라 달라진다. “https” scheme을 통해 사용 가능한 리소스는 리소스 식별자가 동일한 권한(동일한 TCP 포트를 수신하는 동일한 호스트)을 나타내더라도 “http” scheme와 identity을 공유하지 않도록 만든다. 그것들은 별개의 네임스페이스이고, 별개의 원서버로 간주된다. 그러나 Cookie 프로토콜 [RFC6265]과 같은 전체 호스트 도메인에 적용하도록 정의된 HTTP에 확장은 호스트 도메인 그룹내의 다른 서비스와의 통신에 영향을 주기 위해 하나의 서비스에 의해 정보 집합을 허용할 수 있다

The process for authoritative access to an "https" identified resource is defined in [RFC2818].

"https"에 대한 권한 있는 접근 절차 리소스는 [RFC2818]에 정의되어 있다.

2.7.3 http and https URI Normalization and Comparison

Since the "http" and "https" schemes conform to the URI generic syntax, such URIs are normalized and compared according to the algorithm defined in Section 6 of [RFC3986], using the defaults described above for each scheme.

“http” 및 “https” scheme은 URI 일반 구문을 준수하기 때문에 이러한 URI들은 각 scheme의 위해 설명된 기본 값을 사용하며, [RFC3986]의 Section 6에 정의된 알고리즘을 따라 정규화되고 비교된다.

If the port is equal to the default port for a scheme, the normal form is to omit the port subcomponent. When not being used in absolute form as the request target of an OPTIONS request, an empty path component is equivalent to an absolute path of "/", so the normal form is to provide a path of "/" instead. The scheme and host are case-insensitive and normally provided in lowercase; all other components are compared in a case-sensitive manner. Characters other than those in the "reserved" set are equivalent to their percent-encoded octets: the normal form is to not encode them (see Sections 2.1 and 2.2 of [RFC3986]).

포트가 scheme의 기본포트와 같다면, 일반적인 형태는 포트 하위 구성을 생략하는 것이다. OPTIONS 요청의 요청 대상으로 absolute 형식으로 사용하지 않을 경우, 빈 경로 구성 요소는 "/"의 절대 경로와 같으므로, 일반적인 형식은 “/" 경로를 대신 제공하는 것이다. scheme과 호스트는 대 소문자를 구분하지 않으며 일반적으로 소문자로 제공된다. 다른 모든 구성 요소는 대 소문자를 구분하여 비교된다. “reserved" 집합에 있는 문자 이외의 문자는 백분율로

인코딩된 octets와 동일하다. 일반적인 형식은 해당 문자를 인코딩하지 않는다.([RFC3986]의 Section 2.1 및 2.2 참조)

For example, the following three URIs are equivalent:
예시로, 다음의 세개 동등한 URI들이 있다.

http://example.com:80/~smith/home.html
http://EXAMPLE.com/%7Esmith/home.html
http://EXAMPLE.com:/%7esmith/home.html

3. Message Format

All HTTP/1.1 messages consist of a start-line followed by a sequence of octets in a format similar to the Internet Message Format [RFC5322]: zero or more header fields (collectively referred to as the "headers" or the "header section"), an empty line indicating the end of the header section, and an optional message body.

모든 HTTP/1.1 메시지는 인터넷 메시지 형식 [RFC5232]과 유사한 형식의 일련의 octets 시퀀스 뒤에 start-line으로 구성한다: 0 또는 여러 헤더 필드(집합적으로 "헤더"또는"헤더 부문"이라고 함), 헤더 부문의 끝을 나타내는 빈 줄 및 선택적 메시지 본문으로 구성된다.

```
HTTP-message   = start-line
                 *( header-field CRLF )
                 CRLF
                 [ message-body ]
```

The normal procedure for parsing an HTTP message is to read the start-line into a structure, read each header field into a hash table by field name until the empty line, and then use the parsed data to determine if a message body is expected. If a message body has been indicated, then it is read as a stream until an amount of octets equal to the message body length is read or the connection is closed.

HTTP 메시지를 분석하는 일반적인 절차는 start-line을 읽고, 각 헤더 필드를 빈 행까지 필드 이름으로 해시 테이블로 읽은 다음 분석된 데이터를 사용하여 메시지 본문이 필요한지 여부를 확인하는 것이다. 메시지 본문이 표시된 경우, 메시지 본문 길이와 동일한 octet의 양을 읽거나 커넥션을 닫을 때까지 스트림으로 읽는다.

A recipient MUST parse an HTTP message as a sequence of octets in an encoding that is a superset of US-ASCII [USASCII]. Parsing an HTTP message as a stream of Unicode characters, without regard for the specific encoding, creates security vulnerabilities due to the varying ways that string processing libraries handle invalid multibyte character sequences that contain the octet LF (%x0A). String-based parsers can only be safely used within protocol elements after the element has been extracted from the message, such as within a header field-value after message parsing has delineated the individual fields.

수신자는 반드시 HTTP 메시지를 US-ASCII [USASCII]의 상위 집합인 인코딩에서 octet로 구문 분석해야 한다.(MUST) 특정 인코딩에 관계 없이 HTTP 메시지를 유니코드 문자의 스트림으로 구문 분석하면 octet LF(%x0A)를 포함하는 유효하지 않은 다중 바이트 문자 시퀀스를 처리하는 문자열 처리 라이브러리의 다양한 방식 때문에 보안 취약성이 발생한다. 문자열 기반 분석기는 메시지 분석이 개별 필드를 분석 후 헤더 field-value과 같이 메시지에서 요소를 추출한 후에만 프로토콜 요소 내에서 안전하게 사용할 수 있다.

An HTTP message can be parsed as a stream for incremental processing or forwarding downstream. However, recipients cannot rely on incremental delivery of partial messages, since some implementations will buffer or delay message forwarding for the sake of network efficiency, security checks, or payload transformations.

HTTP 메시지는 스트림으로 구문 분석하여 증분 처리하거나 다운 스트림으로 전송할 수 있습니다. 그러나 일부 구현에서는 네트워크 효율성, 보안 검사 또는 페이 로드 변환을 위해 메시지 전달을 버퍼링 하거나 지연시키기 때문에 수신자는 부분 메시지의 점진적 전달에 의존할 수 없다.

A sender MUST NOT send whitespace between the start-line and the first header field. A recipient that receives whitespace between the start-line and the first header field MUST either reject the message as invalid or consume each whitespace-preceded line without further processing of it (i.e., ignore the entire line, along with any subsequent lines preceded by whitespace, until a properly formed header field is received or the header section is terminated).

발신자는 start-line과 첫 번째 헤더 필드 사이에 공백을 보내면 안 된다.(MUST NOT) start-line과 첫 번째 헤더 필드 사이의 공백을 수신하는 수신자는 메시지를 유효하지 않은 것으로 거부하거나 메시지를 더 이상 처리하지 않고 각 공백이 지정된 줄을 소비해야 한다.(MUST) (i.e., 공백이 오는 모든 후속 줄과 함께, 올바르게 형성된 헤더 필드가 수신되거나 헤더 부문이 끝날 때까지, 전체 라인을 무시한다)

The presence of such whitespace in a request might be an attempt to trick a server into ignoring that field or processing the line after it as a new request, either of which might result in a security vulnerability if other implementations within the request chain interpret the same message differently. Likewise, the presence of such whitespace in a response might be ignored by some clients or cause others to cease parsing.

요청에 이러한 공백이 있으면 서버가 해당 필드를 무시하도록 속이거나 새 요청으로 처리하려고 시도한 것일 수 있으며, 요청 체인의 다른 구현에서 동일한 메시지를 다르게 해석할 경우 보안 취약성이 발생할 수 있다. 마찬가지로, 응답에 이러한 공백이 있으면 일부 클라이언트가 무시하거나 다른 사용자가 구문 분석을 중지할 수 있다.

3.1 Start Line

An HTTP message can be either a request from client to server or a response from server to client. Syntactically, the two types of message differ only in the start-line, which is either a request-line (for requests) or a status-line (for responses), and in the algorithm for determining the length of the message body (Section 3.3).

HTTP 메시지는 클라이언트에서 서버로 요청하거나 서버에서 클라이언트로 응답하는 것일 수 있다. 동기적으로, 두 유형의 메시지는 start-line(요청의 경우)또는 status-line(응답의 경우)과 메시지 본문의 길이를 결정하기 위한 알고리즘(Section 3.3)에서만 다르다.

In theory, a client could receive requests and a server could receive responses, distinguishing them by their different start-line formats, but, in practice, servers are implemented to only expect a request (a response is interpreted as an unknown or invalid request method) and clients are implemented to only expect a response.

이론적으로 클라이언트는 요청을 수신할 수 있고 서버는 응답을 수신할 수 있으며, 이를 서로 다른 start-line 형식으로 구분할 수 있지만, 실제로, 서버가 요청만 예상하도록 구현된다.(응

답은 알 수 없거나 잘못된 요청 메서드로 해석됨). 클라이언트는 응답만을 예상하도록 구현된다.

```
start-line     = request-line / status-line
```

3.1.1 Request Line

A request-line begins with a method token, followed by a single space (SP), the request-target, another single space (SP), the protocol version, and ends with CRLF.

request-line은 method 토큰을 시작으로, 공백 (SP), request-target, 공백 (SP), 프로토콜 버전, 그리고 CRLF를 끝으로 한다.

```
request-line   = method SP request-target SP HTTP-version CRLF
```

The method token indicates the request method to be performed on the target resource. The request method is case-sensitive.

method 토큰은 대상 리소스를 실행하기 위한 요청 메서드를 표시한다. 요청 메서드는 대 소문자를 구분한다.

```
method         = token
```

The request methods defined by this specification can be found in Section 4 of [RFC7231], along with information regarding the HTTP method registry and considerations for defining new methods.

이 명세에 정의된 요청 메서드는 HTTP 메서드 레지스트리 및 새 메서드 정의에 대한 고려 사항과 함께 [RFC7431]의 Section 4에서 확인할 수 있다.

The request-target identifies the target resource upon which to apply the request, as defined in Section 5.3.

request-target은 Section 5.3에 정의된 대로 요청을 적용할 대상 리소스를 식별한다.

Recipients typically parse the request-line into its component parts by splitting on whitespace (see Section 3.5), since no whitespace is allowed in the three components. Unfortunately, some user agents fail to properly encode or exclude whitespace found in hypertext references, resulting in those disallowed characters being sent in a request-target.

세 구성 요소에는 공백이 허용되지 않기 때문에 일반적으로 수신자는 공백을 분할하여 request-line을 구성 요소 부분으로 분할한다(Section 3.5 참조). 불행하게도, 일부 사용자 에이전트는 하이퍼 텍스트 참조에서 발견된 공백을 제대로 인코딩하거나 제외하지 못하여 허용되지 않는 문자가 request-target으로 전송된다.

Recipients of an invalid request-line SHOULD respond with either a 400 (Bad Request) error or a 301 (Moved Permanently) redirect with the request-target properly encoded. A recipient SHOULD NOT attempt to autocorrect and then process the request without a redirect, since the invalid request-line might be deliberately crafted to bypass security filters along the request chain.

유효하지 않은 request-line의 수신자는 400(Bad Request) 오류 또는 301(Moved Permanently) 리다이렉트로 응답해야 하며, request-target은 적절히 인코딩되어야 한다.(SHOULD) 유효하지 않은 request-line은 요청 체인을 따라 보안 필터를 우회하도록 의도적으로 조작될 수 있으므로 수신자는 리다이렉트 없이 요청을 자동 수정하고 처리해서는 안 된다.(SHOULD NOT)

HTTP does not place a predefined limit on the length of a request-line, as described in Section 2.5. A server that receives a method longer than any that it implements SHOULD respond with a 501 (Not Implemented) status code. A server that receives a request-target longer than any URI it wishes to parse MUST respond with a 414 (URI Too Long) status code (see Section 6.5.12 of [RFC7231]).

HTTP는 Section 2.5에서 설명한 것처럼 request-line의 길이에 대해 미리 정의된 제한을 두지 않는다. 구현한 것보다 더 긴 메서드를 수신한 서버는 501(Not Implemented)상태 코드로 응답해야 한다.(SHOULD) request-target을 구문 분석하려는 URI보다 긴 414(URI Too Long)상태 코드로 응답해야 한다.(MUST) ([RFC95231]의 Section 6.5.12 참조)

Various ad hoc limitations on request-line length are found in practice. It is RECOMMENDED that all HTTP senders and recipients support, at a minimum, request-line lengths of 8000 octets.

실제로 request-line의 길이에 대한 다양하고 특별한 목적의 제한 사항이 발견된다. 모든 HTTP 발신자와 수신자는, 최소 8000 octet 길이의 request-line을, 지원하는 것을 권장한다.(RECOMMENDED)

3.1.2 Status Line

The first line of a response message is the status-line, consisting of the protocol version, a space (SP), the status code, another space, a possibly empty textual phrase describing the status code, and ending with CRLF.

응답 메시지의 첫 번째 줄은 status-line 이며, 프로토콜 버전, 공백(SP), status-code, 공백(SP) status-code를 설명하는 빈 텍스트 구문, 그리고 끝으로 CRLF이다.

```
status-line = HTTP-version SP status-code SP reason-phrase CRLF
```

The status-code element is a 3-digit integer code describing the result of the server's attempt to understand and satisfy the client's corresponding request. The rest of the response message is to be interpreted in light of the semantics defined for that status code. See Section 6 of [RFC7231] for information about the semantics of status codes, including the classes of status code (indicated by the first digit), the status codes defined by this specification, considerations for the definition of new status codes, and the IANA registry.

status-code 요소는 서버가 클라이언트의 해당 요청을 이해하고 충족하려고 시도한 결과를 설명하는 세 자리의 정수 코드이다. 응답 메시지의 나머지 부분은 해당 상태 코드에 대해 정의된 의미론을 고려하여 해석된다. 상태 코드 등급(첫번째 숫자로 표시), 이 명세에서 정의한 상태 코드, 새 상태 코드 정의를 위한 고려 사항, IANA 레지스트리 등 상태 코드의 의미론에 대한 정보는[RFC7431]의 Section 6을 참조한다.

status-code = 3DIGIT

The reason-phrase element exists for the sole purpose of providing a textual description associated with the numeric status code, mostly out of deference to earlier Internet application protocols that were more frequently used with interactive text clients. A client SHOULD ignore the reason-phrase content.

reason-phrase 요소는 숫자 상태 코드와 관련된 텍스트 설명을 제공하는 유일한 목적으로 존재하며, 대부분 대화형 텍스트 클라이언트에서 더 자주 사용되었던 이전의 인터넷 애플리케이션 프로토콜에 대한 경의를 배제한다. 클라이언트는 reason-phase 내용을 무시해야 한다. (SHOULD)

reason-phrase = *(HTAB / SP / VCHAR / obs-text)

3.2 Header Fields

Each header field consists of a case-insensitive field name followed by a colon (":"), optional leading whitespace, the field value, and optional trailing whitespace.

각 헤더 필드는 대소문자를 구분하지 않는 필드 이름 뒤에 콜론(":"), 선택적 앞의 공백(OWS), 필드 값,선택적 뒤의 공백(OWS)으로 구성된다.

header-field = field-name ":" OWS field-value OWS

field-name = token
field-value = *(field-content / obs-fold)
field-content = field-vchar [1*(SP / HTAB) field-vchar]
field-vchar = VCHAR / obs-text

```
obs-fold       = CRLF 1*( SP / HTAB )
               ; obsolete line folding
               ; see Section 3.2.4
```

The field-name token labels the corresponding field-value as having the semantics defined by that header field. For example, the Date header field is defined in Section 7.1.1.2 of [RFC7231] as containing the origination timestamp for the message in which it appears.

field-name 토큰은 해당 field-value를 헤더 필드에서 정의한 의미론을 갖는 것으로 레이블을 지정한다. 예를 들어 Date 헤더 필드는 [RFC7231]의 Section 7.1.1.2에 있는 메시지의 시작 타임 스탬프를 포함하는 필드로 정의된다.

3.2.1 Field Extensibility

Header fields are fully extensible: there is no limit on the introduction of new field names, each presumably defining new semantics, nor on the number of header fields used in a given message. Existing fields are defined in each part of this specification and in many other specifications outside this document set.

헤더 필드는 완전히 확장 가능하다: 새 필드 이름의 도입, 각각 새로운 의미론을 정의하는 데 제한이 없고. 또한 지정된 메시지에 사용되는 헤더 필드의 개수에도 제한이 없다. 기존 필드는 이 명세의 각 부분과 이 문서 모음 외 다른 많은 명세에 정의되어 있다.

New header fields can be defined such that, when they are understood by a recipient, they might override or enhance the interpretation of previously defined header fields, define preconditions on request evaluation, or refine the meaning of responses.

새로운 헤더 필드는 수신자가 이해할 때 이전에 정의된 헤더 필드의 해석을 재정의하거나 향상시키거나, 요청 평가에 대한 전제 조건을 정의하거나, 응답의 의미르 구체화하도록 정의할 수 있다.

A proxy MUST forward unrecognized header fields unless the field-name is listed in the Connection header field (Section 6.1) or the proxy is specifically configured to block, or otherwise transform, such fields. Other recipients SHOULD ignore

unrecognized header fields. These requirements allow HTTP's functionality to be enhanced without requiring prior update of deployed intermediaries.

field-name이 Connection 헤더 필드(Section 6.1)에 나열되어 있지 않거나 프락시가 이러한 필드를 차단하거나 변환하도록 특별히 구성되어 있지 않는 경우, 프락시는 인식하지 못하는 헤더 필드를 전달해야 한다.(MUST) 다른 수신자는 인식할 수 없는 헤더 필드를 무시해야 한다.(SHOULD) 이러한 요구 사항을 통해 배치된 중개자의 사전 업데이트를 요구하지 않고도 HTTP의 기능을 강화할 수 있다.

All defined header fields ought to be registered with IANA in the "Message Headers" registry, as described in Section 8.3 of [RFC7231].

정의된 모든 헤더 필드는 [RFC7131] Section 8.3에 설명된 대로 “Message Headers” 레지스트리의 IANA에 등록되어야 한다.

3.2.2 Field Order

The order in which header fields with differing field names are received is not significant. However, it is good practice to send header fields that contain control data first, such as Host on requests and Date on responses, so that implementations can decide when not to handle a message as early as possible. A server MUST NOT apply a request to the target resource until the entire request header section is received, since later header fields might include conditionals, authentication credentials, or deliberately misleading duplicate header fields that would impact request processing. A sender MUST NOT generate multiple header fields with the same field name in a message unless either the entire field value for that header field is defined as a comma-separated list [i.e., #(values)] or the header field is a well-known exception (as noted below).

다른 필드 이름과 헤더 필드가 수신하는 순서는 중요하지 않다. 그러나 요청 시 Host, 응답 시 Date 등 제어 데이터가 포함된 헤더 필드를 먼저 보내 구현 시 메시지를 처리하지 않을 시기를 결정하는 것이 좋다. 서버는 헤더 필드에는 조건, 인증 자격 증명 또는 요청 처리에 영향을 미칠 수 있는 의도적으로 잘못된 중복 헤더 필드가 포함될 수 있으므로 전체 요청 헤더 부문을 받을 때 까지 대상 리소스에 요청을 적용하면 안 된다.(MUST NOT) 발신자는 헤더 필드의 전체 필드 값이 쉼표로 구분된 목록 [i.e., #(values)]으로 정의되어 있거나 헤더 필드가 잘 알려 진 예외(아래 설명 참조)가 아닌 한, 메시지에 동일한 필드 이름을 가진 헤더 필드를 여러개 생성해서는 안 된다.(MUST NOT)

A recipient MAY combine multiple header fields with the same field name into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field value to the combined field value in order, separated by a comma. The order in which header fields with the same field name are received is therefore significant to the interpretation of the combined field value; a proxy MUST NOT change the order of these field values when forwarding a message.

수신자는 각 후속 필드 값을 쉼표로 구분하여 결합된 필드 값에 순서대로 추가하여 메시지의 의미론을 변경하지 않고 동일한 필드 이름을 가진 여러 헤더 필드를 하나의 “field-name: field-value“ 쌍으로 결합할 수 있다.(MAY) 따라서 동일한 필드 이름의 헤더 필드를 수신하는 순서는 결합된 필드 값을 해석하는 데 중요하다; 메시지를 전달할 때 프락시는 이러한 필드 값의 순서를 변경해서는 안 된다.(MUST NOT)

> Note: In practice, the "Set-Cookie" header field ([RFC6265]) often appears multiple times in a response message and does not use the list syntax, violating the above requirements on multiple header fields with the same name. Since it cannot be combined into a single field-value, recipients ought to handle "Set-Cookie" as a special case while processing header fields. (See Appendix A.2.3 of [Kri2001] for details.)
>
> 참고: 실제로 “Set-Cookie” 헤더 필드([RFC6065])는 응답 메시지에 여러번 나타나며 목록 구문을 사용하지 않아 동일한 이름의 여러 헤더 필드에서 위의 요구 사항을 위반한다. 필드 값을 하나의 필드 값으로 결합할 수 없으므로, 수신자는 헤더 필드를 처리하는 동안 “Set-Cookie”를 특별한 경우로 처리해야 한다. (자세한 내용은 [Kri2001]의 Appendix A.2.3을 봐라.)

3.2.3 Whitespace

This specification uses three rules to denote the use of linear whitespace: OWS (optional whitespace), RWS (required whitespace), and BWS ("bad" whitespace).

이 명세는 OWS(선택적 공백), RWD(필수 공백)및 BWS(“불량" 공백)의 사용을 나타내는 세 가지 규칙을 사용한다.

The OWS rule is used where zero or more linear whitespace octets might appear. For protocol elements where optional whitespace is preferred to improve readability, a sender SHOULD generate the optional whitespace as a single SP; otherwise, a sender SHOULD NOT generate optional whitespace except as needed to white out invalid or unwanted protocol elements during in-place message filtering.

OWS 규칙은 0 이상의 선형 공백 octets이 나타나는 곳에 사용된다. 가독성을 향상시키기 위해 선택적인 공백(OWS)을 선호하는 프로토콜 요소의 경우, 발신자는 단일 SP로서 선택적인 공백(OWS)를 생성해야 한다. (SHOULD) 그렇지 않은경우, 발신자는 내부 메시지 필터링 중에 유효하지 않거나 원하지 않는 프로토콜 요소를 화이트아웃하는 데 필요한 경우를 제외하고 선택적 공백을 생성해서는 안 된다.(SHOULD NOT)

The RWS rule is used when at least one linear whitespace octet is required to separate field tokens. A sender SHOULD generate RWS as a single SP.

RWS 규칙은 필드 토큰을 분리하기 위해 하나 이상의 선형 공백 octets이 필요한 경우에 사용된다. 발신자는 단일 SP로서 RWS를 생성해야 한다.(SHOULD)

The BWS rule is used where the grammar allows optional whitespace only for historical reasons. A sender MUST NOT generate BWS in messages. A recipient MUST parse for such bad whitespace and remove it before interpreting the protocol element.

BWS 규칙은 오직 역사적인 이유로 선택적인 공백(OWS)을 허락하는 문법에서 사용된다. 발신자는 메시지에서 BWS를 생성해서는 안된다.(MUST NOT) 수신자는 이러한 잘못된 공백을 구문 분석한 후 프로토콜 요소를 해석하기 전에 제거해야 한다.(MUST)

```
OWS         = *( SP / HTAB )
              ; optional whitespace
RWS         = 1*( SP / HTAB )
              ; required whitespace
BWS         = OWS
              ; "bad" whitespace
```

3.2.4 Field Parsing

Messages are parsed using a generic algorithm, independent of the individual header field names. The contents within a given field value are not parsed until a later stage of message interpretation (usually after the message's entire header section has been processed). Consequently, this specification does not use ABNF rules to define each "Field-Name: Field Value" pair, as was done in previous editions. Instead, this specification uses ABNF rules that are named according to each registered field name, wherein the rule defines the valid grammar for that field's corresponding field values (i.e., after the field-value has been extracted from the header section by a generic field parser).

메시지는 개별 헤더 필드 이름과 독립적으로 일반적인 알고리즘을 사용하여 구문 분석된다. 주어진 필드 값 내의 내용은 메시지 해석의 이후 단계(일반적으로 메시지의 전체 헤더 부문이 처리된 후)까지 구문 분석되지 않는다. 따라서 이 규격은 각 “Field-Name: Field Value” 쌍은 이전 버전에서와 같다. 대신, 이 명세는 등록된 각 필드 이름에 따라 명명된 ABNF 규칙을 사용한다. 여기서 규칙은 해당 필드의 해당 필드 값에 대한 유효한 문법을 정의한다(즉, 일반적인 필드 구문 분석기에 의해 헤더 부문에서 field-value를 추출한 후).

No whitespace is allowed between the header field-name and colon. In the past, differences in the handling of such whitespace have led to security vulnerabilities in request routing and response handling. A server MUST reject any received request message that contains whitespace between a header field-name and colon with a response code of 400 (Bad Request). A proxy MUST remove any such whitespace from a response message before forwarding the message downstream.

헤더 field-name과 콜론 사이에는 공백이 허용되지 않는다. 과거에는 이러한 공백 처리에 차이가 있어 요청 라우팅 및 응답 처리 시 보안 취약성이 발생했다. 서버는 응답 코드가 400(Bad Request)과 함께 헤더 필드 이름과 콜론 사이에 공백이 포함된 수신된 요청 메시지를 거부해야 한다.(MUST) 프락시는 메시지를 다운 스트림으로 전달하기 전에 응답 메시지에서 이러한 공백을 모두 제거해야 한다.(MUST)

A field value might be preceded and/or followed by optional whitespace (OWS); a single SP preceding the field-value is preferred for consistent readability by humans. The field value does not include any leading or trailing whitespace: OWS occurring before the first non-whitespace octet of the field value or after the last non-whitespace octet of the field value ought to be excluded by parsers when extracting the field value from a header field.

필드 값은 OWS 앞에 있거나 뒤에 있을 수 있으며, 필드 값 앞에 있는 단일 SP는 사람이 일관되게 읽을 수 있도록 선호한다. 필드 값에는 선행 또는 후행 공백이 포함되어 있지 않다. 필드 값의 첫번째 공백이 아닌 octet 이전 또는 필드 값의 마지막 공백이 아닌 octets 이후에 발생하는 OWS는 헤더 필드에서 필드 값을 추출할 때 파서에 의해 제외되어야 한다.

Historically, HTTP header field values could be extended over multiple lines by preceding each extra line with at least one space or horizontal tab (obs-fold). This specification deprecates such line folding except within the message/http media type (Section 8.3.1). A sender MUST NOT generate a message that includes line folding (i.e., that has any field-value that contains a match to the obs-fold rule) unless the message is intended for packaging within the message/http media type.

역사적으로, HTTP 헤더 필드 값을 각 여러 줄 앞에 최소 하나 이상의 공백 또는 수평 탭(obs-fold)을 두어 여러줄로 확장할 수 있었다. 이 명세는 message/http 미디어 타입(Section 8.3.1)을 제외하고 이러한 라인 폴딩을 제거한다. 발신자는 메시지가 message/http 미디어 타입 내에서 패키징 되도록 의도되지 않은 한 라인 폴딩(즉, obs-fold 규칙과 일치하는 field-value 포함)을 포함하는 메시지를 생성해서는 안 된다.(MUST NOT)

A server that receives an obs-fold in a request message that is not within a message/http container MUST either reject the message by sending a 400 (Bad Request), preferably with a representation explaining that obsolete line folding is unacceptable, or replace each received obs-fold with one or more SP octets prior to interpreting the field value or forwarding the message downstream.

message/http 컨테이너 내에 없는 요청 메시지에서 obs-fold를 수신하는 서버는 400(Bad Request)을 발송하여 메시지를 거부해야 하며,(MUST) 가급적이면 사용되지 않는 라인 폴딩이 허용되지 않는다는 것을 설명하는 표현으로, 필드 값을 해석하거나 또는 메세지를 다운스트림으로 전달하기 전에, 수신된 각 obs-fold를 하나 이상의 SP octet으로 대체해야 한다.

A proxy or gateway that receives an obs-fold in a response message that is not within a message/http container MUST either discard the message and replace it with a 502 (Bad Gateway) response, preferably with a representation explaining that unacceptable line folding was received, or replace each received obs-fold with one or more SP octets prior to interpreting the field value or forwarding the message downstream.

message/http 컨테이너 내에 없는 응답 메시지에서 obs-fold를 수신하는 프락시 또는 게이트 웨이는 메시지를 삭제하고 502(Bad Gateway) 응답으로 대체해야 하고,(MUST) 가급적이면 사용되지 않는 라인 폴딩이 허용되지 않는 다는 것을 설명하는 표현으로, 필드 값을 해석하거

나 또는 메시지를 다운 스트림으로 전달하기 전에, 수신된 각 obs-fold를 하나 또는 더 많은 SP octets 으로 대체해야 한다.

A user agent that receives an obs-fold in a response message that is not within a message/http container MUST replace each received obs-fold with one or more SP octets prior to interpreting the field value.

message/http 컨테이너 내에 없는 응답 메시지에서 obs-fold를 수신하는 사용자 에이전트는 필드 값을 해석하기 전에, 수신된 각 obs-fold를 하나 또는 더 많은 SP octets 으로 대체해야 한다.(MUST)

Historically, HTTP has allowed field content with text in the ISO-8859-1 charset [ISO-8859-1], supporting other charsets only through use of [RFC2047] encoding. In practice, most HTTP header field values use only a subset of the US-ASCII charset [USASCII]. Newly defined header fields SHOULD limit their field values to US-ASCII octets. A recipient SHOULD treat other octets in field content (obs-text) as opaque data.

역사적으로, HTTP는 ISO-8859-1charset [ISO-8859-1]의 텍스트를 포함한 필드 콘텐츠를 허용해 왔으며, [RFC2047] 인코딩의 사용을 통해서만 다른 문자 집합을 지원해왔다. 실제로 대부분의 HTTP 헤더 필드 값은 US-ASCII 문자 집합 [USASCII]의 하위 집합만 사용한다. 새로 정의된 헤더 필드는 필드 값을 US-ASCII octet으로 제한해야 한다.(SHOULD) 수신자는 필드 내용(obs-text)의 다른 octet을 불투명 데이터로 처리해야 한다.(SHOULD)

3.2.5 Field Limits

HTTP does not place a predefined limit on the length of each header field or on the length of the header section as a whole, as described in Section 2.5. Various ad hoc limitations on individual header field length are found in practice, often depending on the specific field semantics.

HTTP는 Section 2.5에서 설명한 대로 각 헤더 필드의 길이 또는 헤더 부문의 전체 길이에 미리 정의된 제한을 두지 않는다. 실제로 개별 헤더 필드 길이에 대한 다양한 특별한 목적의 제한사항이 발견되며, 종종 특정 필드 의미론에 따라 달라진다.

A server that receives a request header field, or set of fields, larger than it wishes to process MUST respond with an appropriate 4xx (Client Error) status code. Ignoring such header fields would increase the server's vulnerability to request smuggling attacks (Section 9.5).

처리하려는 것보다 큰 요청 헤더 필드 또는 필드 집합을 수신하는 서버는 적절한 4xx(Client Error)상태 코드로 응답해야 한다.(MUST) 이러한 헤더 필드를 무시하면 서버의 smuggling 공격 취약성이 증가한다(Section 9.5).

A client MAY discard or truncate received header fields that are larger than the client wishes to process if the field semantics are such that the dropped value(s) can be safely ignored without changing the message framing or response semantics.

필드 의미론이 메시지 프레임 또는 응답 의미론을 변경하지 않고 삭제된 값을 안전하게 무시할 수 있는 경우 클라이언트가 처리하고자 하는 것보다 큰 수신 헤더 필드를 삭제하거나 잘라낼 수 있다.(MAY)

3.2.6 Field Value Components

Most HTTP header field values are defined using common syntax components (token, quoted-string, and comment) separated by whitespace or specific delimiting characters. Delimiters are chosen from the set of US-ASCII visual characters not allowed in a token (DQUOTE and "(),/:;<=>?@[₩]{}").

대부분의 HTTP 헤더 필드 값은 공백 또는 특정 구분 문자로 구분된 일반 구문 구성 요소(token, quoted-string 및 comment)를 사용하여 정의된다. 구분 기호는 토큰 (DQUOTE and “(),/:;<=>?@[₩]{}")에 허용되지 않는 US-ASCII 시각적 문자 집합에서 선택된다.

```
token       = 1*tchar

tchar       = "!" / "#" / "$" / "%" / "&" / "'" / "*"
              / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~"
              / DIGIT / ALPHA
              ; any VCHAR, except delimiters
```

A string of text is parsed as a single value if it is quoted using double-quote marks.

텍스트 문자는 double-quote로 묶여 있으면 하나의 값으로 분석된다.

```
quoted-string  = DQUOTE *( qdtext / quoted-pair ) DQUOTE
qdtext         = HTAB / SP /%x21 / %x23-5B / %x5D-7E / obs-text
obs-text       = %x80-FF
```

Comments can be included in some HTTP header fields by surrounding the comment text with parentheses. Comments are only allowed in fields containing "comment" as part of their field value definition.

주석은 괄호로 둘러싸인 텍스트를 둘러싼 일부 HTTP 헤더 필드에 포함될 수 있다. 주석은 필드 값 정의의 일부로 "comment"를 포함하는 필드에서만 허용된다.

```
comment        = "(" *( ctext / quoted-pair / comment ) ")"
ctext          = HTAB / SP / %x21-27 / %x2A-5B / %x5D-7E / obs-text
```

The backslash octet ("₩") can be used as a single-octet quoting mechanism within quoted-string and comment constructs. Recipients that process the value of a quoted-string MUST handle a quoted-pair as if it were replaced by the octet following the backslash.

backslash octet (“₩”)는 quoted-string 메커니즘 및 주석 구조 내에서 single-octet으로 사용할 수 있다. quoted-string 값을 처리하는 수신자는 quoted-pair를 백슬래시로 octet에 의해 대체된 것 처럼 처리해야 한다.(MUST)

```
quoted-pair    = "₩" ( HTAB / SP / VCHAR / obs-text )
```

A sender SHOULD NOT generate a quoted-pair in a quoted-string except where necessary to quote DQUOTE and backslash octets occurring within that string. A sender SHOULD NOT generate a quoted-pair in a comment except where necessary to quote parentheses ["(" and ")"] and backslash octets occurring within that comment.

발신자는 DQUOTE와 백슬래시 octets을 해당 문자에 인용해야 하는 경우를 제외하고 quoted-string에서 quoted-pair를 생성면 안 된다.(SHOULD NOT)
발신자는 괄호[“(“and”)”]와 backslash octets를 해당 주석에 인용해야 하는 경우를 제외하고 comment 내에 quoted-pair를 생성하면 안 된다.(SHOULD NOT)

3.3 Message body

The message body (if any) of an HTTP message is used to carry the payload body of that request or response. The message body is identical to the payload body unless a transfer coding has been applied, as described in Section 3.3.1.

HTTP 메시지의 메시지 본문(있는 경우)은 해당 요청 또는 응답의 페이로드 본문을 전달하는 데 사용된다. 메시지 본문은 Section 3.3.1에 설명되어 있는 전송 코딩이 적용되어있지 않는 한 페이로드 본문과 동일하다.

```
message-body = *OCTET
```

The rules for when a message body is allowed in a message differ for requests and responses.

메시지에서 메시지 본문이 허용되는 시기에 대한 규칙은 요청과 응답에 따라 다르다.

The presence of a message body in a request is signaled by a Content-Length or Transfer-Encoding header field. Request message framing is independent of method semantics, even if the method does not define any use for a message body.

요청에서 메시지 본문이 있으면 Content-Length 또는 Transfer-Encoding 헤더 필드로 표시된다. 요청 메시지 프레임은 메서드가 메시지 본문에 대한 사용을 정의하지 않더라도 메서드 의미론과 독립적이다.

The presence of a message body in a response depends on both the request method to which it is responding and the response status code (Section 3.1.2). Responses to the HEAD request method (Section 4.3.2 of [RFC7231]) never include a message body because the associated response header fields (e.g., Transfer-Encoding, Content-Length, etc.), if present, indicate only what their values would have been if the request method had been GET (Section 4.3.1 of [RFC7231]). 2xx (Successful) responses to a CONNECT request method (Section 4.3.6 of [RFC7231]) switch to tunnel mode instead of having a message body. All 1xx (Informational), 204 (No Content), and 304 (Not Modified) responses do not include a message body. All other responses do include a message body, although the body might be of zero length.

응답에서 메시지 본문의 존재 여부는 응답하는 요청 메서드와 응답 상태 코드(Section 3.1.2)에 따라 달라진다. HEAD 요청 메서드([RFC7231]의 Section 4.3.2)에 대한 응답은 관련 응답 헤더 필드(예:Transfer-Encoding, Content-Length, 등)가 있는 경우 요청 메서드가 GET([RFC7231]의 Section 4.3.1) 되었을 때의 값만 나타내므로 메시지 본문을 포함하지 않는다. CONNECT 요청 메서드([RFC7231]의 Section 4.3.6)에 2xx (Successful) 응답은 메시지 본문 대신 터널 모드로 전환한다. 모든 1xx (informational), 204 (No Content) 및 304 (Not Modified) 응답에는 메시지 본문이 포함되지 않는다. 다른 모든 응답에는 메시지 본문이 포함되지만, 본문의 길이는 0일 수 있다.

3.3.1 Transfer-Encoding

The Transfer-Encoding header field lists the transfer coding names corresponding to the sequence of transfer codings that have been (or will be) applied to the payload body in order to form the message body. Transfer codings are defined in Section 4.

Transfer-Encoding 헤더 필드에는 메시지 본문을 형성하기 위해 페이 로드 본문에 적용된(또는 적용될) 전송 코딩 순서에 해당하는 전송 코딩 이름을 나열한다. 전송 코딩은 Section 4에 정의되어 있다.

```
Transfer-Encoding = 1#transfer-coding
```

Transfer-Encoding is analogous to the Content-Transfer-Encoding field of MIME, which was designed to enable safe transport of binary data over a 7-bit transport service ([RFC2045], Section 6). However, safe transport has a different focus for an 8bit-clean transfer protocol. In HTTP's case, Transfer-Encoding is primarily intended to accurately delimit a dynamically generated payload and to distinguish payload encodings that are only applied for transport efficiency or security from those that are characteristics of the selected resource.

Transfer-Encoding은 MIME의 Content-Transfer-Encoding 필드와 유사하며, 7비트 전송 서비스([RFC2045], Section 6)를 통해 바이너리 데이터를 안전하게 전송할 수 있도록 설계되었다. 그러나, 안전한 전송은 8bit-clean 전송 프로토콜로 다른 초점을 두고 있다. HTTP의 경우, Transfer-Encoding은 주로 동적으로 생성되는 페이 로드를 정확하게 지정하고 전송 효율성 또는 보안을 위해서만 적용되는 페이 로드 인코딩과, 선택된 리소스의 특징인 페이 로드로 구분하기 위한 것이다.

A recipient MUST be able to parse the chunked transfer coding (Section 4.1) because it plays a crucial role in framing messages when the payload body size is not known in advance. A sender MUST NOT apply chunked more than once to a message body (i.e., chunking an already chunked message is not allowed). If any transfer coding other than chunked is applied to a request payload body, the sender MUST apply chunked as the final transfer coding to ensure that the message is properly framed. If any transfer coding other than chunked is applied to a response payload body, the sender MUST either apply chunked as the final transfer coding or terminate the message by closing the connection.

페이 로드 본문 크기를 미리 알지 못할 때 메시지 프레임에 중요한 역할을 하기 때문에 수신자는 청크 전송 코딩(Section 4.1)을 구문 분석할 수 있어야 한다. (MUST) 발신자는 메시지 본문에 두번 이상 청크 분할된 메시지를 적용해서는 안 된다.(MUST NOT) (i.e., 이미 청크 분할된 메시지는 허용되지 않는다). 청크 분할 이외의 전송 코딩이 요청 페이 로드 본문에 적용되는 경우, 발신자는 메시지가 올바르게 프레임 되었는지 확인하기 위해 최종 전송 코딩으로 청크 분할을 적용해야 한다.(MUST) 청크 분할 이외의 전송 코딩이 응답 페이 로드 본문에 적용되는 경우 발신자는 최종 전송 코딩으로 청크 분할을 적용하거나 커넥션을 닫아 메시지를 종료해야 한다.

For example,

예를들어,

```
Transfer-Encoding: gzip, chunked
```

indicates that the payload body has been compressed using the gzip coding and then chunked using the chunked coding while forming the message body.

는 메시지 본문을 구성하는 동안 페이 로드 본문이 gzip 코딩을 사용하여 압축된 다음 chunked 코드를 사용하여 분할된 것을 나타낸다.

Unlike Content-Encoding (Section 3.1.2.1 of [RFC7231]), Transfer-Encoding is a property of the message, not of the representation, and any recipient along the request/response chain MAY decode the received transfer coding(s) or apply additional transfer coding(s) to the message body, assuming that corresponding changes are made to the Transfer-Encoding field-value. Additional information about the encoding parameters can be provided by other header fields not defined by this specification.

Content-Encoding [RFC7231]의 Section 3.1.2.1과 달리, Transfer-Encoding은 메시지의 속성이며, 표현이 아니며, 요청/응답 체인에 있는 모든 수신자는 Transfer-Encoding field-value에 변경사항이 있다면 수신된 전송 코딩을 디코딩하거나 추가 전송 코딩을 본문에 적용할 수 있다.(MAY) 추가적인 인코딩 매개 변수에 대한 정보는 이 명세에 정의되지 않은 다른 헤더 필드로 제공되어진다.

Transfer-Encoding MAY be sent in a response to a HEAD request or in a 304 (Not Modified) response (Section 4.1 of [RFC7232]) to a GET request, neither of which includes a message body, to indicate that the origin server would have applied a transfer coding to the message body if the request had been an unconditional GET. This indication is not required, however, because any recipient on the response chain (including the origin server) can remove transfer codings when they are not needed.

HEAD 요청에 응답 하거나 또는 304(Not Modified) 응답 ([RFC7232]의 Section 4.1)을 통해 GET 요청에 응답 하거나, 메시지 본문을 포함하거나 조건 없는 GET 요청인 경우 원서버가 메시지 본문에 전송 코딩을 적용 했을거라고 표시하기 위해 Transfer-Encoding이 보내질 수 있다.(MAY) 그러나 응답 체인(원서버 포함해서)에 있는 모든 수신자는 전송 코딩이 필요하지 않을 때 제거할 수 있기 때문에 이러한 표시는 필요하지 않다.

A server MUST NOT send a Transfer-Encoding header field in any response with a status code of 1xx (Informational) or 204 (No Content). A server MUST NOT send a

Transfer-Encoding header field in any 2xx (Successful) response to a CONNECT request
(Section 4.3.6 of [RFC7231]).

서버는 상태 코드가 1xx (Informational) 또는 204 (No Content) 응답에서 Transfer-Encoding 헤더 필드를 보내면 안 된다.(MUST NOT)
서버는 CONNECT 요청에 2xx (Successful) 응답에서 Transfer-Encoding 헤더 필드를 보내면 안 된다.(MUST NOT) ([RFC7131]의 Section 4.3.6).

Transfer-Encoding was added in HTTP/1.1. It is generally assumed that implementations advertising only HTTP/1.0 support will not understand how to process a transfer-encoded payload. A client MUST NOT send a request containing Transfer-Encoding unless it knows the server will handle HTTP/1.1 (or later) requests; such knowledge might be in the form of specific user configuration or by remembering the version of a prior received response. A server MUST NOT send a response containing Transfer-Encoding unless the corresponding request indicates HTTP/1.1 (or later).

HTTP/1.1에 Transfer-Encoding이 추가되었다. 일반적으로, 단지 HTTP/1.0 지원을 알리는 용도로 구현하는 것은 transfer-encoded 페이로드를 처리하기 위한 방법을 이해하지 못할 것으로 가정한다. 클라이언트는 서버가 HTTP/1.1(이상)요청을 처리 할 수 있을 것이라고 알지 못하는 한 Transfer-Encoding을 포함하는 요청을 보내면 안 된다.(MUST NOT) 이러한 인지는 특정 사용자 구성 형태이거나 이전에 수신한 응답 버전을 기억함으로써 가능하다. 해당 요청이 HTTP/1.1(또는 그 이상)을 나타내지 않는 한 서버는 Transfer-Encoding을 포함하는 응답을 보내면 안 된다.(MUST NOT)

A server that receives a request message with a transfer coding it does not understand SHOULD respond with 501 (Not Implemented).

이해하지 못하는 전송 코딩과 함께 요청 메시지를 받는 서버는 501 (Not Implemented)로 응답해야 한다.(SHOULD)

3.3.2 Content-Length

When a message does not have a Transfer-Encoding header field, a Content-Length header field can provide the anticipated size, as a decimal number of octets, for a

potential payload body. For messages that do include a payload body, the Content-Length field-value provides the framing information necessary for determining where the body (and message) ends. For messages that do not include a payload body, the Content-Length indicates the size of the selected representation (Section 3 of [RFC7231]).

메시지에 Transfer-Encoding 헤더 필드가 없는 경우, Content-Length 헤더 필드는 잠재적 페이 로드 본문에 대해 예상되는 크기를 10진수로 제공할 수 있다. 페이 로드 본문을 포함하는 메시지의 경우, Content-Length 필드 값은 본문(및 메시지)이 끝나는 위치를 결정하는 데 필요한 프레임 정보를 제공한다. 페이 로드 본문이 포함되지 않은 메시지의 경우, Content-Length는 선택된 표현의 크기를 나타낸다. ([RFC7131]의 Section 3)

```
Content-Length = 1*DIGIT
```

An example is

예를들어

```
Content-Length: 3495
```

A sender MUST NOT send a Content-Length header field in any message that contains a Transfer-Encoding header field.

발신자는 Transfer-Encoding 헤더 필드를 포함하는 메시지에서 Content-Length 헤더 필드를 보내면 안 된다.(MUST NOT)

A user agent SHOULD send a Content-Length in a request message when no Transfer-Encoding is sent and the request method defines a meaning for an enclosed payload body. For example, a Content-Length header field is normally sent in a POST request even when the value is 0 (indicating an empty payload body). A user agent SHOULD NOT send a Content-Length header field when the request message does not contain a payload body and the method semantics do not anticipate such a body.

Transfer-Encoding 없이 전송하고 요청 메서드가 동봉된 페이 로드 본문에 대한 의미를 정의할 때 사용자 에이전트는 요청 메시지에 Content-Length를 보내야 한다.(SHOULD) 예를 들어, 그 값이 0 (빈 페이 로드 본문을 나타냄)인 경우에도 Content-Length 헤더 필드는 POST 요청에서 일반적으로 전송된다. 요청 메시지에 페이 로드 본문이 포함되어 있지 않고 메서드 의미론이 이러한 본문을 예상하지 않는 경우 사용자 에이전트는 Content-length 헤더 필드를 발송해서는 안 된다.(SHOULD NOT)

A server MAY send a Content-Length header field in a response to a HEAD request (Section 4.3.2 of [RFC7231]); a server MUST NOT send Content-Length in such a response unless its field-value equals the decimal number of octets that would have been sent in the payload body of a response if the same request had used the GET method.

서버는 HEAD 요청에 대한 응답으로 Content-Length 헤더 필드를 전송할 수 있다. (MAY) ([RFC7231]의 Section 4.3.2); 서버는 동일한 요청에서 GET 메소드를 사용한 경우 응답의 페이로드 본문에 전송 된 10진수와 Content-Length의 field-value가 같지 않는 한, 서버는 응답으로 Content-Length를 보내서는 안 된다.(MUST NOT)

A server MAY send a Content-Length header field in a 304 (Not Modified) response to a conditional GET request (Section 4.1 of [RFC7232]); a server MUST NOT send Content-Length in such a response unless its field-value equals the decimal number of octets that would have been sent in the payload body of a 200 (OK) response to the same request.

서버는 조건부 GET 요청에 대해 304 (Not Modified) 응답의 Content-Length 헤더 필드를 보낼 수 있다. ([RFC7232]의 Section 4.1); 필드 값이 동일한 요청에 대한 200 (OK) 응답의 페이로드 본문에서 전송된 10진수와 Content-Length의 field-value가 같지 않는 한, 서버는 응답으로 Content-Length를 보내서는 안 된다.(MUST NOT)

A server MUST NOT send a Content-Length header field in any response with a status code of 1xx (Informational) or 204 (No Content). A server MUST NOT send a Content-Length header field in any 2xx (Successful) response to a CONNECT request (Section 4.3.6 of [RFC7231]).

서버는 상태 코드가 1xx (Informational) 또는 204 (No Content)인 응답에서 Content-Length헤더 필드를 보내면 안 된다.(MUST NOT)
서버는 CONNECT 요청에 대한 2xx (Successful) 응답에서 Content-Length 헤더 필드를 보내서는 안 된다.(MUST NOT) ([RFC7231]의 Section 4.3.6)

Aside from the cases defined above, in the absence of Transfer-Encoding, an origin server SHOULD send a Content-Length header field when the payload body size is known prior to sending the complete header section. This will allow downstream recipients to measure transfer progress, know when a received message is complete, and potentially reuse the connection for additional requests.

위에 정의된 경우와 별도로, Transfer-Encoding이 없는 경우, 전체 헤더 부문을 보내기 전에 페이 로드 본문 크기를 알고 있는 원서버는 Content-Length 헤더 필드를 보내야 한다.(SHOULD) 이를 통해 다운스트림 수신자는 전송 진행률을 측정하고, 수신된 메시지가 완료된 시점을 알 수 있으며, 잠재적으로 추가 요청을 위해 커넥션을 재사용할 수 있다.

Any Content-Length field value greater than or equal to zero is valid. Since there is no predefined limit to the length of a payload, a recipient MUST anticipate potentially large decimal numerals and prevent parsing errors due to integer conversion overflows (Section 9.3).

Content-Length 필드 값이 0보다 크거나 같으면 유효하다. 페이 로드 길이에 대한 사전 정의된 제한이 없으므로 수신자는 잠재적으로 큰 10진수 숫자를 예측하고 정수 변환 오버 플로우로 인한 구문 분석 오류를 방지해야 한다.(MUST) (Section 9.3).

If a message is received that has multiple Content-Length header fields with field-values consisting of the same decimal value, or a single Content-Length header field with a field value containing a list of identical decimal values (e.g., "Content-Length: 42, 42"), indicating that duplicate Content-Length header fields have been generated or combined by an upstream message processor, then the recipient MUST either reject the message as invalid or replace the duplicated field-values with a single valid Content-Length field containing that decimal value prior to determining the message body length or forwarding the message.

여러 Content-Length 헤더 필드와 함께 동일한 10진수로 구성하는 field-value를 가지고 있거나, 또는 단독 Content-Length 헤더 필드와 함께 동일한 10진수의 리스트로 구성하는 field-value를 가지고 있거나(예: “Content-length:42, 42”), 중복된 Content-Length 헤더 필드가 생성되거나 업스트림 메시지 처리기에 의해 결합된 것으로 나타나는 메시지를 받은 경우, 그때 수신자는 유효하지 않은 메시지를 거부하거나, 메시지 본문 길이를 결정하는것 또는 전송되기 전에 10진수를 포함하는 유효한 단독 Content-Length와 함께 중복된 field-value를 대체해야 한다.(MUST)

Note: HTTP's use of Content-Length for message framing differs significantly from the same field's use in MIME, where it is an optional field used only within the “message/external-body" media-type.

참고: 메시지 프레임에 HTTP의 Content-Length 사용은 “message/exexternal-body" media-type 내에서만 사용되는 선택적 필드이며, MIME에서 동일한 필드를 사용하는 것과 크게 다르다.

3.3.3 Message body Length

The length of a message body is determined by one of the following (in order of precedence):

메시지 본문의 길이는 다음 중 하나에 의해 결정된다(우선 순위 순서대로):

1. Any response to a HEAD request and any response with a 1xx (Informational), 204 (No Content), or 304 (Not Modified) status code is always terminated by the first empty line after the header fields, regardless of the header fields present in the message, and thus cannot contain a message body.

1. HEAD 요청에 대한 응답과 1xx(Informational), 204(No Content) 또는 304(Not Modified) 상태 코드 응답은 메시지에 있는 헤더 필드에 관계 없이 헤더 필드 다음의 첫 번째 빈 줄로 항상 종료되므로 메시지 본문을 포함할 수 없다.

2. Any 2xx (Successful) response to a CONNECT request implies that the connection will become a tunnel immediately after the empty line that concludes the header fields. A client MUST ignore any Content-Length or Transfer-Encoding header fields received in such a message.

2. CONNECT 요청에 대한 2xx(Successful) 응답은 헤더 필드를 마치는 빈 줄 직후 커넥션은 터널이 됨을 내포한다. 클라이언트는 이러한 메시지에 수신된 Content-Length 또는 Transfer-Encoding 헤더 필드를 반드시 무시해야 한다. (MUST)

3. If a Transfer-Encoding header field is present and the chunked transfer coding (Section 4.1) is the final encoding, the message body length is determined by reading and decoding the chunked data until the transfer coding indicates the data is complete.

3. Transfer-Encoding 헤더 필드가 있고 청크 전송 코딩(Section 4.1)이 최종 인코딩인 경우, 전송 코딩이 데이터가 완료되었음을 나타낼 때까지 청크 데이터를 읽고 디코딩 하여 메시지 본문 길이를 결정한다.

If a Transfer-Encoding header field is present in a response and the chunked transfer coding is not the final encoding, the message body length is determined by reading the connection until it is closed by the server. If a Transfer-Encoding header field is present in a request and the chunked transfer coding is not the final encoding, the message body length cannot be determined reliably; the server MUST respond with the 400 (Bad Request) status code and then close the connection.

응답에 Transfer-Encoding 헤더 필드가 있고 청크 분할 전송 코딩이 최종 인코딩이 아닌 경우, 서버가 커넥션을 닫을 때까지 커넥션을 읽어 메시지 본문 길이를 결정한다. 요청에 Transfer-Encoding 헤더 필드가 있고 청크 전송 코딩이 최종 인코딩이 아닌 경우 메시지 본문 길이를 신뢰할 수 없다; 서버는 400(Bad Request) 상태 코드로 응답한 다음, 커넥션을 반드시 닫아야 한다.(MUST)

If a message is received with both a Transfer-Encoding and a Content-Length header field, the Transfer-Encoding overrides the Content-Length. Such a message might indicate an attempt to perform request smuggling (Section 9.5) or response splitting (Section 9.4) and ought to be handled as an error. A sender MUST remove the received Content-Length field prior to forwarding such a message downstream.

메시지가 Transfer-Encoding 및 Content-Length 헤더 필드와 함께 수신되면 Transfer-Encoding이 Content-Length를 재정의한다. 그러한 메시지는 요청 smuggling(Section 9.5) 또는 응답 분할(Section 9.4)을 수행하려는 시도를 나타낼 수 있으며 오류로 취급되어야 한다. 발신자는 이러한 메시지를 다운 스트림으로 전달하기 전에 수신된 Content-Length 필드를 반드시 제거해야 한다.(MUST)

4. If a message is received without Transfer-Encoding and with either multiple Content-Length header fields having differing field-values or a single Content-Length header field having an invalid value, then the message framing is invalid and the recipient MUST treat it as an unrecoverable error. If this is a request message, the server MUST respond with a 400 (Bad Request) status code and then close the connection. If this is a response message received by a proxy, the proxy MUST close the connection to the server, discard the received response, and send a 502 (Bad Gateway) response to the client. If this is a response message

received by a user agent, the user agent MUST close the connection to the server and discard the received response.

4. Transfer-Encoding이 없는 메시지를 수신하거나 여러 Content-Length 헤더 필드들의 field-value가 다르거나 단일 Content-length 헤더 필드가 잘못된 경우, 메시지 프레임은 유효하지 않으며 수신자는 이를 복구할 수 없는 오류로 간주 해야 한다.(MUST) 요청 메시지인 경우, 서버는 400 (Bad Request) 상태 코드로 응답한 다음 커넥션을 닫아야 한다. (MUST) 프락시에서 수신한 응답 메시지인 경우, 프락시는 서버에 대한 커넥션을 닫고 수신된 응답을 무시하고, 502 (Bad Gateway)을 클라이언트에게 전송해야 한다.(MUST) 사용자 에이전트가 수신한 응답 메시지인 경우, 사용자 에이전트는 서버에 대한 커넥션을 닫고 수신된 응답을 삭제해야 한다. (MUST)

5. If a valid Content-Length header field is present without Transfer-Encoding, its decimal value defines the expected message body length in octets. If the sender closes the connection or the recipient times out before the indicated number of octets are received, the recipient MUST consider the message to be incomplete and close the connection.

5. Transfer-Encoding이 없는 유효한 Content-Length 헤더 필드가 있는 경우, 10진수 값은 octet으로 예상되는 메시지 본문 길이를 정의한다. 발신자가 커넥션을 닫거나 표시된 octet 수를 수신하기 전에 시간이 초과되면, 수신자는 메시지가 불완전한 것으로 간주하고 커넥션을 반드시 닫아야 한다.(MUST)

6. If this is a request message and none of the above are true, then the message body length is zero (no message body is present).

6. 이 메시지가 요청 메시지이고 위의 메시지 중 하나가 참이 아니면 메시지 본문 길이가 0이다.(메시지 본문이 없음).

7. Otherwise, this is a response message without a declared message body length, so the message body length is determined by the number of octets received prior to the server closing the connection.

7. 그렇지 않으면, 이것은 선언된 메시지 본문 길이가 없는 응답 메시지이므로, 메시지 본문 길이는 서버가 커넥션을 닫기 전에 수신한 octet 수로 결정된다.

Since there is no way to distinguish a successfully completed, close-delimited message from a partially received message interrupted by network failure, a server

SHOULD generate encoding or length-delimited messages whenever possible. The close-delimiting feature exists primarily for backwards compatibility with HTTP/1.0.

성공적으로 완료된 메시지와 네트워크 장애로 인해 중단된 부분적으로 수신된 close-delimited 메시지를 구분할 수 있는 방법이 없으므로, 서버는 가능할 때마다 인코딩 또는 길이가 length-delimited 메시지를 생성해야 한다.(SHOULD) close-delimiting 특징은 주로 HTTP/1.0 하위 호환성을 위해 존재한다.

A server MAY reject a request that contains a message body but not a Content-Length by responding with 411 (Length Required).

서버는 411(Length Required) 로 응답하여 메시지 본문은 포함하지만 Content-Length는 포함하지 않는 요청을 거부할 수 있다.(MAY)

Unless a transfer coding other than chunked has been applied, a client that sends a request containing a message body SHOULD use a valid Content-Length header field if the message body length is known in advance, rather than the chunked transfer coding, since some existing services respond to chunked with a 411 (Length Required) status code even though they understand the chunked transfer coding. This is typically because such services are implemented via a gateway that requires a content-length in advance of being called and the server is unable or unwilling to buffer the entire request before processing.

청크 이외의 전송 코딩이 적용되지 않은 경우, 메시지 본문을 포함하는 요청을 전송하는 클라이언트는 메시지 본문 길이가 청크 전송 코딩 보다 미리 알려진 경우, 유효한 Content-Length 헤더 필드를 사용해야한다.(SHOULD) 그 이유는, 일부 기존 서비스는 청크 전송 코딩을 이해하더라도 411 (Length Required) 상태 코드로 청크에 응답하기 때문이다. 이것은 일반적으로 그러한 서비스가 호출되기 전에 Content-Length가 필요하고 서버가 처리하기 전에 전체 요청을 버퍼링 할 수 없거나 버퍼링 하지 않는 게이트웨이를 통해 구현되기 때문이다.

A user agent that sends a request containing a message body MUST send a valid Content-Length header field if it does not know the server will handle HTTP/1.1 (or later) requests; such knowledge can be in the form of specific user configuration or by remembering the version of a prior received response.

서버가 HTTP/1.1(또는 그 이상)의 요청을 처리할 것인지 모르는 경우 메시지 본문이 포함된 요청을 보내는 사용자 에이전트는 유효한 Content-Length 헤더 필드를 반드시 보내야 합니다.(MUST); 이러한 정보는 특정 사용자 구성 형식이거나 이전에 수신한 응답의 버전을 기억하는 방식일 수 있다.

If the final response to the last request on a connection has been completely received and there remains additional data to read, a user agent MAY discard the remaining data or attempt to determine if that data belongs as part of the prior response body, which might be the case if the prior message's Content-Length value is incorrect. A client MUST NOT process, cache, or forward such extra data as a separate response, since such behavior would be vulnerable to cache poisoning.

커넥션에서 마지막 요청에 최종 응답이 완전하게 수신되고 읽어야 할 추가 데이터가 남아 있는 경우는, 사용자 에이전트가 나머지 데이터를 삭제하거나 또는 그 데이터는 이전 응답 본문의 일부로 결정해야하며,(MAY) 이전 메시지의 Content-Length 값이 잘못된 경우가 이에 해당할 수 있다. 이러한 동작은 캐시 중독에 취약하므로 클라이언트는 별도의 응답으로 추가 데이터를 처리, 캐시 또는 전달해서는 안 된다.(MUST NOT)

3.4 Handling Imcomplete Messages

A server that receives an incomplete request message, usually due to a canceled request or a triggered timeout exception, MAY send an error response prior to closing the connection.

일반적으로 취소된 요청 또는 트리거 된 시간 초과 예외 때문에 발생하는 불완전한 요청 메시지를 수신하는 서버는 커넥션을 닫기 전에 오류 응답을 보낼 수 있다.(MAY)

A client that receives an incomplete response message, which can occur when a connection is closed prematurely or when decoding a supposedly chunked transfer coding fails, MUST record the message as incomplete. Cache requirements for incomplete responses are defined in Section 3 of [RFC7234].

커넥션이 조기에 닫히거나 청크 분할된 전송 코딩을 디코딩 하는 데 실패할 때 발생할 수 있는 불완전한 응답 메시지를 수신하는 클라이언트는 메시지를 반드시 불완전한 메시지로 기록해야 한다.(MUST) 불완전한 응답에 대한 캐시 요구 사항은 [RFC7234]의 Section 3에 정의되어 있다.

If a response terminates in the middle of the header section (before the empty line is received) and the status code might rely on header fields to convey the full meaning of the response, then the client cannot assume that meaning has been conveyed;

the client might need to repeat the request in order to determine what action to take next.

응답이 헤더 부문의 중간에서 종료되고(빈 줄이 수신되기 전에) 상태 코드가 응답의 전체 의미를 전달하기 위해 헤더 필드에 의존할 수 있는 경우, 클라이언트는 의미가 전달되었다고 가정할 수 없다. 클라이언트는 다음 조치를 결정하기 위해 요청을 반복해야 할 수 있다.

A message body that uses the chunked transfer coding is incomplete if the zero-sized chunk that terminates the encoding has not been received. A message that uses a valid Content-Length is incomplete if the size of the message body received (in octets) is less than the value given by Content-Length. A response that has neither chunked transfer coding nor Content-Length is terminated by closure of the connection and, thus, is considered complete regardless of the number of message body octets received, provided that the header section was received intact.

인코딩을 종료하는 zero-sized(0)의 청크가 수신되지 않으면 청크 전송 코딩을 사용하는 메시지 본문은 불완전하다. 수신된 메시지 본문의 크기(in octets)가 Content-Length에서 지정한 값보다 작은 경우 유효한 Content-Length를 사용하는 메시지는 불완전하다. 청크 전송 코딩이 없거나 Content-Length가 없는 응답은 커넥션이 종료되어야 하고, 제공된 헤더 부문이 온전히 수신되었다면 받은 메시지 본문의 수(octet)에 관계 없이 완전한 응답으로 간주된다.

3.5 Message Parsing Robustness

Older HTTP/1.0 user agent implementations might send an extra CRLF after a POST request as a workaround for some early server applications that failed to read message body content that was not terminated by a line-ending. An HTTP/1.1 user agent MUST NOT preface or follow a request with an extra CRLF. If terminating the request message body with a line-ending is desired, then the user agent MUST count the terminating CRLF octets as part of the message body length.

오래된 HTTP/1.0 사용자 에이전트 구현은 line-ending으로 종료되지 않은 메시지 본문 내용을 읽지 못하는 일부 초기 서버 응용 프로그램에 대한 회피 방법으로 POST 요청 후 추가 CRLF를 전송할 수 있다. HTTP/1.1 사용자 에이전트는 추가 CRLF로 시작하거나 요청을 반드시 따르면 안 된다.(MUST NOT) line-ending으로 요청 메시지 본문을 종료해야 하는 경우, 사용자 에이전트는 메시지 본문 길이의 부분으로서 종료 CRLF octet을 세어야 한다.(MUST)

In the interest of robustness, a server that is expecting to receive and parse a request-line SHOULD ignore at least one empty line (CRLF) received prior to the request-line.

엄격함을 위해, request-line을 수신하고 구문 분석할 것으로 예상되는 서버는 request-line 이전에 수신된 적어도 하나의 빈 라인 (CRLF) 을 무시해야 한다.(SHOULD)

Although the line terminator for the start-line and header fields is the sequence CRLF, a recipient MAY recognize a single LF as a line terminator and ignore any preceding CR.

start-line과 헤더 필드의 라인 종료자가 CRLF 순서이지만, 수신자가 단일 LF를 라인으로 인식할 수 있으며 선행 CR을 무시할 수 있다.(MAY)

Although the request-line and status-line grammar rules require that each of the component elements be separated by a single SP octet, recipients MAY instead parse on whitespace-delimited word boundaries and, aside from the CRLF terminator, treat any form of whitespace as the SP separator while ignoring preceding or trailing whitespace; such whitespace includes one or more of the following octets: SP, HTAB, VT (%x0B), FF (%x0C), or 기본적인 CR. However, lenient parsing can result in security vulnerabilities if there are multiple recipients of the message and each has its own unique interpretation of robustness (see Section 9.5).

request-line과 status-line 문법 규칙은 각 구성요소가 단일 SP octet 으로 구분되는 것이 필요하지만, 수신자는 대신에 whitespace-delimited(공백으로 구분된) 단어 경계를 분석하고, CRLF 종료자를 제외하고, 선행 또는 후행 공백을 무시하면서 모든 형식의 공백을 SP 구분자로 처리할 수 있다; 그러한 공백은 하나 또는 더 많은 다음의 octets:SP, HTAB, VT(%x0B), FF (%x0C), or CR 을 포함한다. 그러나 메시지의 여러 수신자와 각각 고유의 엄격함에 대한 해석이 있는 경우 관대한 분석은 보안적인 위험성을 초래할 수 있다. (Section 9.5를 참조)

When a server listening only for HTTP request messages, or processing what appears from the start-line to be an HTTP request message, receives a sequence of octets that does not match the HTTP-message grammar aside from the robustness exceptions listed above, the server SHOULD respond with a 400 (Bad Request) response.

HTTP 요청 메시지만 청취하거나, HTTP 요청 메시지이기 위해 start-line에서 나타나는 내용을 처리하는 서버가 위에 나열된 엄격함 예외 외에도 HTTP-message 문법과 일치하지 않는

octets의 시퀀스를 수신하는 경우, 서버는 400 (Bad Request)응답으로 응답해야 한다. (SHOULD)

4. Transfer codings

Transfer coding names are used to indicate an encoding transformation that has been, can be, or might need to be applied to a payload body in order to ensure "safe transport" through the network. This differs from a content coding in that the transfer coding is a property of the message rather than a property of the representation that is being transferred.

전송 코딩 이름은 네트워크를 통해 “safe transport” 을 보장하기 위해 페이 로드 본문에 적용되었거나 적용되어야 하거나 적용되어야 할 수 있는 인코딩 변환을 나타내는 데 사용된다. 이는 전송 코딩이 전송되는 표현의 속성이 아니라 메시지의 속성이라는 점에서 내용 코딩과 다르다.

```
transfer-coding    = "chunked" ; Section 4.1
                   / "compress" ; Section 4.2.1
                   / "deflate" ; Section 4.2.2
                   / "gzip" ; Section 4.2.3
                   / transfer-extension
transfer-extension = token *( OWS ";" OWS transfer-parameter )
```

Parameters are in the form of a name or name=value pair.

매개변수는 이름 또는 이름=값 쌍의 형식이다.

```
transfer-parameter = token BWS "=" BWS ( token / quoted-string )
```

All transfer-coding names are case-insensitive and ought to be registered within the HTTP Transfer Coding registry, as defined in Section 8.4. They are used in the TE (Section 4.3) and Transfer-Encoding (Section 3.3.1) header fields.

모든 transfer-coding 이름은 대소문자를 구분하지 않으며 Section 8.4에 정의된 HTTP Transfer Coding registry에 등록해야 한다. 전송 코딩은 TE(Section 4.3)및 Transfer-Encoding(Section 3.3.1) 헤더 필드에 사용된다.

4.1 Chunked Transfer Coding

The chunked transfer coding wraps the payload body in order to transfer it as a series of chunks, each with its own size indicator, followed by an OPTIONAL trailer containing header fields. Chunked enables content streams of unknown size to be transferred as a sequence of length-delimited buffers, which enables the sender to retain connection persistence and the recipient to know when it has received the entire message.

청크 전송 코딩은 페이 로드 본체의 각각 자체 크기를 표시하여 전송하며, OPTIONAL 트레일러를 포함하는 헤더 필드 뒤에, 일련의 청크를 순차적으로 전송하기 위해 페이 로드 본체를 감싼다. 청크를 사용하면 알 수 없는 크기의 콘텐츠 스트림을 길이가 제한된 버퍼의 순서로 전송할 수 있으며, 이를 통해 송신자는 커넥션 영속성을 유지하고 수신자는 전체 메시지를 수신한 시점을 알 수 있다.

```
chunked-body   = *chunk
                 last-chunk
                 trailer-part
                 CRLF

chunk          = chunk-size [ chunk-ext ] CRLF
                 chunk-data CRLF
chunk-size     = 1*HEXDIG
last-chunk     = 1*("0") [ chunk-ext ] CRLF

chunk-data     = 1*OCTET ; a sequence of chunk-size octets
```

The chunk-size field is a string of hex digits indicating the size of the chunk-data in octets. The chunked transfer coding is complete when a chunk with a chunk-size of zero is received, possibly followed by a trailer, and finally terminated by an empty line.

chunk-size 필드는 octet 단위의 청크 데이터 크기를 나타내는 16 진수 문자열이다. 청크 전송 코딩은 크기가 0인 청크가 수신되고, 가능하면 트레일러가 뒤따르며, 마지막으로 빈 줄로 끝날 때 완료된다.

A recipient MUST be able to parse and decode the chunked transfer coding.

수신자는 청크 분할 전송 코딩을 반드시 구문 분석하고 디코딩 할 수 있어야 한다.(MUST)

4.1.1 Chunk Extensions

The chunked encoding allows each chunk to include zero or more chunk extensions, immediately following the chunk-size, for the sake of supplying per-chunk metadata (such as a signature or hash), mid-message control information, or randomization of message body size.

청크 인코딩 각 청크가 0 또는 더 많은 청크 확장을 포함하도록 하며, chunk-size 바로 다음, 청크 단위 메타 데이터(서명 또는 해시와 같은), mid-message 제어 정보 또는 메시지 크기의 임의화를 제공한다.

```
chunk-ext      = *( ";" chunk-ext-name [ "=" chunk-ext-val ] )

chunk-ext-name = token
chunk-ext-val  = token / quoted-string
```

The chunked encoding is specific to each connection and is likely to be removed or recoded by each recipient (including intermediaries) before any higher-level application would have a chance to inspect the extensions. Hence, use of chunk extensions is generally limited to specialized HTTP services such as "long

polling" (where client and server can have shared expectations regarding the use of chunk extensions) or for padding within an end-to-end secured connection.

청크 인코딩은 각 커넥션에 따라 다르며 상위 수준의 애플리케이션이 확장자를 검사하기 위한 기회를 가지기 전에 각 수신자(중개자 포함)에 의해 제거되거나 재코딩될 수 있다. 따라서 청크 확장자의 사용은 일반적으로 “long polling”(클라이언트와 서버가 청크 확장자 사용과 관련한 예상을 공유할 수 있다.) 또는 end-to-end 보안 커넥션 내의 padding과 같은 특별한 목적의 HTTP 서비스로 제한된다.

A recipient MUST ignore unrecognized chunk extensions. A server ought to limit the total length of chunk extensions received in a request to an amount reasonable for the services provided, in the same way that it applies length limitations and timeouts for other parts of a message, and generate an appropriate 4xx (Client Error) response if that amount is exceeded.

수신자는 인식할 수 없는 청크 확장자를 무시해야 한다.(MUST) 서버는 요청으로 수신된 청크 확장자의 총 길이를 메시지의 다른 부분에 대해 길이 제한 및 시간 제한을 적용하는 것과 동일한 방식으로 제공된 서비스에 적합한 양으로 제한해야 하며, 이 양을 초과할 경우 적절한 4xx (Client Error) 응답을 생성해야 한다.

4.1.2 Chunked Trailer Part

A trailer allows the sender to include additional fields at the end of a chunked message in order to supply metadata that might be dynamically generated while the message body is sent, such as a message integrity check, digital signature, or post-processing status. The trailer fields are identical to header fields, except they are sent in a chunked trailer instead of the message's header section.

트레일러를 사용하면 발신자가 메시지 무결성 검사, 디지털 서명 또는 후 처리 상태와 같이 메시지 본문이 전송되는 동안 동적으로 생성될 수 있는 메타 데이터를 제공하기 위해 청크 분할된 메시지의 끝에 추가 필드를 포함할 수 있다. 트레일러 필드는 메시지의 헤더 부문 대신 청크된 트레일러로 보내지는 것을 제외하고 헤더 필드와 동일하다.

trailer-part   = *(header-field CRLF)

A sender MUST NOT generate a trailer that contains a field necessary for message framing (e.g., Transfer-Encoding and Content-Length), routing (e.g., Host), request modifiers (e.g., controls and conditionals in Section 5 of [RFC7231]), authentication (e.g., see [RFC7235] and [RFC6265]), response control data (e.g., see Section 7.1 of [RFC7231]), or determining how to process the payload (e.g., Content-Encoding, Content-Type, Content-Range, and Trailer).

발신자는 메시지 프레이밍에 필요한 필드(e.g., Transfer-Encoding 과 Content-Length), 라우팅 (e.g., Host), 요청 수정자 (e.g., [RFC7231]의 Section 5 제어와 조건부), 권한 (e.g., [RFC7235]와 [RFC6265]를 봐라), 응답 제어 데이터 (e.g., [RFC7231]의 Section 7.1) 또는 페이로드 처리를 어떻게 결정하는 지 (e.g., Content-Encoding, Content-Type, Content-Range과 Trailer)를 포함하는 트레일러를 생성해서는 안 된다.(MUST NOT)

When a chunked message containing a non-empty trailer is received, the recipient MAY process the fields (aside from those forbidden above) as if they were appended to the message's header section. A recipient MUST ignore (or consider as an error) any fields that are forbidden to be sent in a trailer, since processing them as if they were present in the header section might bypass external security filters.

트레일러가 비어있지 않는 청크 메시지가 수신되면 수신자는 필드를 메시지의 헤더 부문에 추가된 것처럼 처리한다.(위에서 금지된 필드는 제외) 수신자는 헤더 부문에 있는 것처럼 필드를 처리하면 외부 보안 필터를 우회할 수 있으므로 트레일러로 보낼 수 없는 모든 필드를 무시하거나 오류로 간주해야 한다.(MUST)

Unless the request includes a TE header field indicating “trailers" is acceptable, as described in Section 4.3, a server SHOULD NOT generate trailer fields that it believes are necessary for the user agent to receive. Without a TE containing "trailers", the server ought to assume that the trailer fields might be silently discarded along the path to the user agent. This requirement allows intermediaries to forward a de-chunked message to an HTTP/1.0 recipient without buffering the entire response.

Section 4.3에 설명된 것처럼 요청에 “trailers”를 나타내는 TE 헤더 필드가 포함되지 않는 한, 서버는 사용자 에이전트가 받는 데 필요하다고 생각하는 트레일러 필드를 생성해서는 안 된다.(SHOULD NOT) “trailers”가 포함된 TE가 없는 경우, 서버는 트레일러 필드가 사용자 에이전트의 경로를 따라 자동으로 폐기될 수 있다고 가정해야 한다. 이 요구사항은 중개자가 전체 응답의 버퍼링 없이 de-chunked 메시지를 HTTP/1.0 수신자에게 전달할 수 있게 한다.

4.1.3 Decoding Chunked

A process for decoding the chunked transfer coding can be represented in pseudo-code as:

청크 처리된 전송 코딩을 디코딩하는 프로세스는 다음과 같이 유사 코드로 나타낼 수 있다.

```
length := 0
read chunk-size, chunk-ext (if any), and CRLF
while (chunk-size > 0) {
   read chunk-data and CRLF
   append chunk-data to decoded-body
   length := length + chunk-size
   read chunk-size, chunk-ext (if any), and CRLF
}
read trailer field
while (trailer field is not empty) {
   if (trailer field is allowed to be sent in a trailer) {
      append trailer field to existing header fields
   }
   read trailer-field
}
Content-Length := length
Remove "chunked" from Transfer-Encoding
Remove Trailer from existing header fields
```

4.2 Compression Codings

The codings defined below can be used to compress the payload of a message.

아래에 정의된 코딩은 메시지 페이로드 압축에 사용될 수 있다.

4.2.1 Compress Coding

The "compress" coding is an adaptive Lempel-Ziv-Welch (LZW) coding [Welch] that is commonly produced by the UNIX file compression program "compress". A recipient SHOULD consider "x-compress" to be equivalent to "compress".

“compress” 코딩은 공통적으로 유닉스 파일 압축 프로그램 “compress”로 생산된 Lempel-Ziv-Welch (LZW) 코딩이다. 수신자는 “x-comress”를 “compress”로 동등하게 간주해야 한다.(SHOULD)

4.2.2 Deflate Coding

The "deflate" coding is a "zlib" data format [RFC1950] containing a “deflate" compressed data stream [RFC1951] that uses a combination of the Lempel-Ziv (LZ77) compression algorithm and Huffman coding.

“deflate” 코딩은 Lempel-Ziv (LZ77) 압축 알고리즘과 허프만 코딩을 결합하여 사용하는 “deflate” 압축 데이터 스트림을 포함하는 “zlib” 데이터 포멧 [RFC1950]이다.

Note: Some non-conformant implementations send the “deflate" compressed data without the zlib wrapper.

참고: 일부 부적합한 구현은 zlib 래퍼 없이 “delfate” 압축된 데이터를 전송한다.

4.2.3 Gzip Coding

The "gzip" coding is an LZ77 coding with a 32-bit Cyclic Redundancy Check (CRC) that is commonly produced by the gzip file compression program [RFC1952]. A recipient SHOULD consider "x-gzip" to be equivalent to "gzip".

“gzip” 코딩은 공통적으로 gzip 파일 압축 프로그램 [RFC1952]으로 생산된 LZ77 32비트 Cyclic Redundancy Check (CRC) 코딩 이다. 수신자는 “x-gzip”를 “gzip”로 동등하게 간주해야 한다.(SHOULD)

4.3 TE

The "TE" header field in a request indicates what transfer codings, besides chunked, the client is willing to accept in response, and whether or not the client is willing to accept trailer fields in a chunked transfer coding.

“TE” 헤더 필드가 요청에서 어떤 전송 코딩을 나타내며, 청크 외에, 클라이언트가 응답을 수락할 의향이 있는지 여부와 클라이언트가 청크 전송 코딩의 트레일러 필드를 수용할 의사가 있는지 여부를 나타낸다.

The TE field-value consists of a comma-separated list of transfer coding names, each allowing for optional parameters (as described in Section 4), and/or the keyword "trailers". A client MUST NOT send the chunked transfer coding name in TE; chunked is always acceptable for HTTP/1.1 recipients.

TE field-value는 전송 코드 이름의 쉼표로 구분된 목록으로 구성되며, 각 이름은 (Section 4에서 설명한 대로) 선택적 매개 변수 또는 키워드 “trailers” 를 허용한다. 클라이언트는 TE로 “chunked” 전송 코딩 이름을 전송해서는 안 된다.(MUST NOT); HTTP/1.1 수신자는 항상 청크를 허용한다.

```
TE       = #t-codings
t-codings = "trailers" / ( transfer-coding [ t-ranking ] )
t-ranking = OWS ";" OWS "q=" rank
rank     = ( "0" [ "." 0*3DIGIT ] )
             / ( "1" [ "." 0*3("0") ] )
```

Three examples of TE use are below.

TE 사용의 세 가지 예는 다음과 같다.

```
TE: deflate
TE:
TE: trailers, deflate;q=0.5
```

The presence of the keyword "trailers" indicates that the client is willing to accept trailer fields in a chunked transfer coding, as defined in Section 4.1.2, on behalf of itself and any downstream clients. For requests from an intermediary, this implies that either: (a) all downstream clients are willing to accept trailer fields in the forwarded response; or, (b) the intermediary will attempt to buffer the response on behalf of downstream recipients. Note that HTTP/1.1 does not define any means to limit the size of a chunked response such that an intermediary can be assured of buffering the entire response.

키워드 "trailers"가 있다는 것은 클라이언트는 Section 4.1.2에 정의된 것처럼 청크 전송 코딩의 트레일러 필드를 자신과 다운스트림 클라이언트를 대신하여 기꺼이 수용할 용의가 있음을 나타낸다. 중개자 요청의 경우,
(a) 모든 다운 스트림 클라이언트가 전달된 응답의 트레일러 필드를 기꺼이 수락하거나; 또는,
(b) 중개자는 다운 스트림 수신자를 대신하여 응답을 버퍼링 하려고 시도한다는 것을 의미한다. 참고로 HTTP/1.1는 중개자가 전체 응답을 버퍼링하는 것을 보증할 수 있도록 청크의 크기를 제한하는 수단을 정의하지 않는다.

When multiple transfer codings are acceptable, the client MAY rank the codings by preference using a case-insensitive "q" parameter (similar to the qvalues used in content negotiation fields, Section 5.3.1 of [RFC7231]). The rank value is a real number in the range 0 through 1, where 0.001 is the least preferred and 1 is the most preferred; a value of 0 means "not acceptable".

복수의 전송 코딩이 허용 가능한 경우, 클라이언트는 대소문자 구분하지 않는 “q” 매개 변수([RFC7231]의 Section 5.3.1, 내용 협상 필드에서 q 값을 사용하는 것과 유사)를 사용하여 코딩 순위를 선호도에 따라 지정할 수 있다. 순위 값은 0~1범위의 실제 숫자이며, 여기서 0.001은 가장 선호되지 않으며 1은 가장 선호된다. 0의 값은 “not acceptable"을 의미한다.

If the TE field-value is empty or if no TE field is present, the only acceptable transfer coding is chunked. A message with no transfer coding is always acceptable.

TE field-value가 비어 있거나 TE 필드가 없는 경우, 허용 가능한 전송 코딩만 청크된다. 전송 코딩이 없는 메시지는 항상 허용된다.

Since the TE header field only applies to the immediate connection, a sender of TE MUST also send a "TE" connection option within the Connection header field (Section 6.1) in order to prevent the TE field from being forwarded by intermediaries that do not support its semantics.

TE 헤더 필드는 바로 옆 커넥션에만 적용되므로, TE의 발신자는 TE 필드가 의미론을 지원하지 않는 중개자에 의해 전송 되는 것을 방지하기 위해 Connection 헤더 필드(Section 6.1) 내에서 “TE” 커넥션 옵션을 반드시 보내야 한다. (MUST)

4.4 Trailer

When a message includes a message body encoded with the chunked transfer coding and the sender desires to send metadata in the form of trailer fields at the end of the message, the sender SHOULD generate a Trailer header field before the message body to indicate which fields will be present in the trailers. This allows the recipient to prepare for receipt of that metadata before it starts processing the body, which is useful if the message is being streamed and the recipient wishes to confirm an integrity check on the fly.

메시지가 청크 전송 코딩과 인코딩된 메시지 본문을 포함하거나 발신자는 메시지 끝에 있는 트레일러 필드 형태의 메타 데이터를 전송하고자 할 때, 발신자는 메시지 본문 앞에 Trailer 헤더 필드를 생성하여 트레일러에 어떤 필드가 존재하는지 표시해야 한다. (SHOULD) 이렇게 하면 수신자가 본문을 처리하기 전에 해당 메타 데이터를 수신할 준비를 할 수 있다. 메시지가 스트리밍 되고 수신자가 즉시 무결성 검사를 확인하고자 할 때 유용하다.

```
Trailer = 1#field-name
```

5. Message Routing

HTTP request message routing is determined by each client based on the target resource, the client's proxy configuration, and establishment or reuse of an inbound connection. The corresponding response routing follows the same connection chain back to the client.

HTTP 요청 메시지 라우팅은 대상 리소스, 클라이언트의 프락시 구성 및 인바운드 커넥션의 설정 또는 재사용을 기준으로 각 클라이언트에 의해 결정된다. 해당 응답 라우팅은 클라이언트 뒤에서 동일한 커넥션 체인을 따른다.

5.1 Identifying a Target Resource

HTTP is used in a wide variety of applications, ranging from general-purpose computers to home appliances. In some cases, communication options are hard-coded in a client's configuration. However, most HTTP clients rely on the same resource identification mechanism and configuration techniques as general-purpose Web browsers.

HTTP는 범용 컴퓨터에서 가전 제품에 이르는 다양한 애플리케이션에서 사용된다. 경우에 따라 클라이언트 구성에서 통신 옵션이 하드 코딩되기도 한다. 그러나 대부분의 HTTP 클라이언트는 범용 웹 브라우저와 동일한 리소스 식별 메커니즘과 구성 기술에 의존한다.

HTTP communication is initiated by a user agent for some purpose. The purpose is a combination of request semantics, which are defined in [RFC7231], and a target resource upon which to apply those semantics. A URI reference (Section 2.7) is typically used as an identifier for the "target resource", which a user agent would resolve to its absolute form in order to obtain the "target URI". The target URI excludes the reference's fragment component, if any, since fragment identifiers are reserved for client-side processing ([RFC3986], Section 3.5).

HTTP 통신은 어떤 목적을 위해 사용자 에이전트에 시작된다. 목적은 [RFC7231]에 정의된 요청 의미론과 이러한 의미론을 적용할 대상 리소스의 조합이다. URI 참조 (Section 2.7)는 일반적으로 “target resource”를 위해 식별자로 사용되며, 사용자 에이전트는 “target URI” 를 얻기 위해 식별자의 절대 형식으로 결정했을 것이다. 대상 URI는 클라이언트 측 처리를 위해 단편 식별자가 예약되어 있기 때문에 참조의 단편 구성을 제외한다. ([RFC3986]의 Section 3.5)

5.2 Connecting Inbound

Once the target URI is determined, a client needs to decide whether a network request is necessary to accomplish the desired semantics and, if so, where that request is to be directed.

대상 URI가 결정되면 클라이언트는 바람직한 의미론을 달성하기 위해 네트워크 요청이 필요한지 여부를 결정해야 하며, 그렇다면 요청이 필요한 경우 어디에 요청되는 위치를 결정해야 한다.

If the client has a cache [RFC7234] and the request can be satisfied by it, then the request is usually directed there first.

클라이언트에 캐시 [RFC7234] 가 있고 요청이 이에 의해 충족될 수 있는 경우, 일반적으로 요청은 먼저 해당 캐시로 전달된다.

If the request is not satisfied by a cache, then a typical client will check its configuration to determine whether a proxy is to be used to satisfy the request. Proxy configuration is implementation-dependent, but is often based on URI prefix matching, selective authority matching, or both, and the proxy itself is usually identified by an "http" or "https" URI. If a proxy is applicable, the client connects inbound by establishing (or reusing) a connection to that proxy.

요청이 캐시에 의해 충족되지 않으면 일반 클라이언트는 해당 구성을 확인하여 요청을 충족하기 위해 프락시를 사용할지 여부를 결정한다. 프락시 구성은 구현에 따라 다르지만 URI 접두사 일치, 선택적 권한 일치 또는 둘 다에 기반하는 경우가 많으며, 프락시 자체는 대개 "http" 또는 "https" URI로 식별된다. 프락시가 적용 가능한 경우 클라이언트는 해당 프락시에 대한 커넥션을 설립(또는 재사용) 하여 인바운드를 연결한다.

If no proxy is applicable, a typical client will invoke a handler routine, usually specific to the target URI's scheme, to connect directly to an authority for the target resource. How that is accomplished is dependent on the target URI scheme and defined by its associated specification, similar to how this specification defines origin server access for resolution of the "http" (Section 2.7.1) and "https" (Section 2.7.2) schemes.

프락시가 적용되지 않는 경우, 일반적인 클라이언트는 대상 URI 의 scheme과 관련된 핸들러 루틴을 호출하여 대상 리소스에 대한 권한에 직접 연결한다. 이 명세는 “http”(Section 2.7.1) 및 "https" (Section 2.7.2) scheme의 해결을 위해 원서버 접근을 정의하는 방식과 유사하게 특정 URI scheme에 따라 수행되는 방법이 달라진다.

HTTP requirements regarding connection management are defined in Section 6.

커넥션 관리와 관련된 HTTP 요구사항은 Section 6에 정의되어 있다.

5.3 Request Target

Once an inbound connection is obtained, the client sends an HTTP request message (Section 3) with a request-target derived from the target URI. There are four distinct formats for the request-target, depending on both the method being requested and whether the request is to a proxy.

한번 인바운드 커넥션을 얻으면, 클라이언트는 대상 URI에서 파생된 request-target과 함께 HTTP 요청 메시지(Section 3)를 보낸다. 요청하는 방법과 프락시에 대한 요청인지 여부에 따라 request-target 형식은 네 가지로 구분된다.

```
request-target = origin-form
               / absolute-form
               / authority-form
               / asterisk-form
```

5.3.1 origin-form

The most common form of request-target is the origin-form.

대부분의 request-target의 공통적 양식은 origin-form이다.

```
origin-form    = absolute-path [ "?" query ]
```

When making a request directly to an origin server, other than a CONNECT or server-wide OPTIONS request (as detailed below), a client MUST send only the absolute path and query components of the target URI as the request-target. If the target URI's path component is empty, the client MUST send "/" as the path within the origin-form of request-target. A Host header field is also sent, as defined in Section 5.4.

CONNECT 또는 서버 전체 OPTIONS 요청 외에 원서버에 직접 요청할 때(아래에 자세히 설명된 대로) 클라이언트는 대상 URI의 절대 경로 및 쿼리 구성 요소만 request-target으로 보내야 한다.(MUST) 대상 URI의 경로 구성 요소가 비어 있으면 클라이언트가 request-target의 origin-form 내에 경로로 “/" 를 반드시 보내야 한다.(MUST) Section 5.4에 정의된 대로 Host 헤더 필드도 전송된다.

For example, a client wishing to retrieve a representation of the resource identified as

예를 들어, 클라이언트는 http://www.example.org/where?q=now 같은 식별된 리소스의 표시로 검색하는 것을 바랄 것이다.

```
http://www.example.org/where?q=now
```

directly from the origin server would open (or reuse) a TCP connection to port 80 of the host "www.example.org" and send the lines:

원서버로부터 직접 포트 80에 대한 TCP 커넥션을 열고(또는 재사용하고) 호스트 "www.example.org"과 다음줄을 보냈을 것이다.

```
GET /where?q=now HTTP/1.1
Host: www.example.org
```

followed by the remainder of the request message.

요청 메시지의 나머지 부분이 뒤따른다.

5.3.2 absolute-form

When making a request to a proxy, other than a CONNECT or server-wide OPTIONS request (as detailed below), a client MUST send the target URI in absolute-form as the request-target.

프락시에 요청을 할 때, CONNECT 또는 서버 측 OPTIONS 요청을 제외하고 (아래에 자세히 설명된 대로), 클라이언트는 absolute-form을 request-target으로 대상 URI를 보내야 한다.

```
absolute-form  = absolute-URI
```

The proxy is requested to either service that request from a valid cache, if possible, or make the same request on the client's behalf to either the next inbound proxy server or directly to the origin server indicated by the request-target. Requirements on such "forwarding" of messages are defined in Section 5.7.

유효한 캐시로 요청하는 서비스 중 하나에서 프락시가 요청될 때, 가능하다면 또는 클라이언트를 위해 다음 인바운드 프락시 서버 중 하나 또는 request-target 에서 표시된 원서버에 직접 동일한 요청을 한다. 그러한 메시지의 “forwarding” 요구사항은 Section 5.7에 정의되어 있다.

An example absolute-form of request-line would be:

예시로 request-line의 absolute-form이 있다:

```
GET http://www.example.org/pub/WWW/TheProject.html HTTP/1.1
```

To allow for transition to the absolute-form for all requests in some future version of HTTP, a server MUST accept the absolute-form in requests, even though HTTP/1.1 clients will only send them in requests to proxies.

일부 미래의 HTTP 버전에서 모든 요청을 위한 absolute-form으로 변환하려면, HTTP/1.1 클라이언트가 프락시에만 요청으로 absolute-form을 전송하더라도, 서버는 요청에서 absolute-form를 수용해야 한다.(MUST)

5.3.3 authority-form

The authority-form of request-target is only used for CONNECT requests (Section 4.3.6 of [RFC7231]).

request-target의 authority-form은 CONNECT 요청에서만 사용된다. ([RFC7231]의 Section 4.3.6)

authority-form = authority

When making a CONNECT request to establish a tunnel through one or more proxies, a client MUST send only the target URI's authority component (excluding any userinfo and its "@" delimiter) as the request-target. For example,

하나 또는 그 이 상의 프락시들을 통해 터널과 설립하기 위해 CONNECT 요청을 할 때, 클라이언트는 반드시 대상 URI의 권한 구성요소만 request-target으로 보내야 한다. (MUST)(어떤 userinfo나 “@“ 기호를 제외하고)

CONNECT www.example.com:80 HTTP/1.1

5.3.4 asterisk-form

The asterisk-form of request-target is only used for a server-wide OPTIONS request (Section 4.3.7 of [RFC7231]).

request-target의 asterisk-form은 서버 전체 OPTIONS 요청에서만 사용된다.

asterisk-form = "*"

When a client wishes to request OPTIONS for the server as a whole, as opposed to a specific named resource of that server, the client MUST send only "*" (%x2A) as the request-target. For example,

클라이언트가 서버 전체에 대해 OPTIONS 요청을 하고자 할 때, 서버의 특정한 명명된 리소스와는 반대로, 클라이언트는 반드시 “*” (%x2A)만 request-target으로 보내야 한다.(MUST)

OPTIONS * HTTP/1.1

If a proxy receives an OPTIONS request with an absolute-form of request-target in which the URI has an empty path and no query component, then the last proxy on the request chain MUST send a request-target of "*" when it forwards the request to the indicated origin server.

프락시가 URI에 빈 경로가 있고 쿼리 구성 요소가 없는 request-target의 absolute-form과 OPTIONS 요청을 수신하는 경우, 요청 체인의 마지막 프락시가 요청을 지정된 원서버로 전달할 때 "*"의 request-target을 반드시 보내야 한다.(MUST)

For example, the request

예를들어, 요청에서

OPTIONS http://www.example.org:8001 HTTP/1.1

would be forwarded by the final proxy as

마지막 프락시에 의해 전송됐을 것이다.

```
OPTIONS * HTTP/1.1
Host: www.example.org:8001
```

after connecting to port 8001 of host "www.example.org".

host “www.example.org"의 8001 포트에 연결된 후

5.4 Host

The "Host" header field in a request provides the host and port information from the target URI, enabling the origin server to distinguish among resources while servicing requests for multiple host names on a single IP address.

요청의 “Host” 헤더 필드는 대상 URI로 부터 호스트 및 포트 정보를 제공하므로 원서버가 단일 IP주소에서 다중 호스트 이름에 대한 요청을 처리하는 동안 리소스를 구분할 수 있다.

```
Host = uri-host [ ":" port ] ; Section 2.7.1
```

A client MUST send a Host header field in all HTTP/1.1 request messages. If the target URI includes an authority component, then a client MUST send a field-value for Host that is identical to that authority component, excluding any userinfo subcomponent and its “@" delimiter (Section 2.7.1). If the authority component is missing or undefined for the target URI, then a client MUST send a Host header field with an empty field-value.

클라이언트는 모든 HTTP/1.1 요청 메시지에서 Host 헤더 필드를 반드시 보내야 한다.(MUST) 대상 URI에 권한 구성 요소가 포함된 경우, 어떤 userinfo나 “@“ 기호를 제외하고, 클라이언트는 해당 권한 구성 요소와 호스트에 대한 field-value를 반드시 보내야 한다. (MUST) 대상 URI에 대한 권한 구성 요소가 없거나 정의되지 않은 경우 클라이언트는 빈 field-value가 포함된 Host 헤더 필드를 반드시 보내야 한다.(MUST)

Since the Host field-value is critical information for handling a request, a user agent SHOULD generate Host as the first header field following the request-line.

Host의 field-value는 요청을 처리하는 데 중요한 정보이므로 사용자 에이전트는 request-line 다음에 첫 번째 헤더 필드로 Host를 생성해야 한다.(SHOULD)

For example, a GET request to the origin server for
<http://www.example.org/pub/WWW/> would begin with:

예를 들어, 원서버에 대한 <http://www.example.org/pub/WWW/>의 GET 요청은 다음과 같이 시작된다.

```
GET /pub/WWW/ HTTP/1.1
Host: www.example.org
```

A client MUST send a Host header field in an HTTP/1.1 request even if the request-target is in the absolute-form, since this allows the Host information to be forwarded through ancient HTTP/1.0 proxies that might not have implemented Host.

request-target이 absoluted-form인 경우라도 클라이언트는 HTTP/1.1 요청에서 Host 헤더 필드를 반드시 전송해야 한다.(MUST) Host를 구현하지 않았을 지도 모르는 오래된 HTTP/1.0 프락시를 통해 Host 정보가 전달할 수 있기 때문이다.

When a proxy receives a request with an absolute-form of request-target, the proxy MUST ignore the received Host header field (if any) and instead replace it with the host information of the request-target. A proxy that forwards such a request MUST generate a new Host field-value based on the received request-target rather than forward the received Host field-value.

프락시가 request-target의 absolute-form을 가진 요청을 수신하는 경우, 프락시는 수신된 Host 헤더 필드(있는 경우)를 무시하고 request-target의 호스트 정보로 대체해야 한다.(MUST) 이러한 요청을 전달하는 프록시는 수신된 Host field-value를 전달하는 것이 아니라 수신된 요청 대상을 기반으로 새 Host field-value를 생성해야 한다.

Since the Host header field acts as an application-level routing mechanism, it is a frequent target for malware seeking to poison a shared cache or redirect a request

to an unintended server. An interception proxy is particularly vulnerable if it relies on the Host field-value for redirecting requests to internal servers, or for use as a cache key in a shared cache, without first verifying that the intercepted connection is targeting a valid IP address for that host.

Host 헤더 필드는 애플리케이션-레벨 라우팅 메커니즘의 역할을 하므로 공유 캐시를 못쓰게 하거나 요청을 의도하지 않은 서버로 리다이렉트 하려는 맬웨어의 빈번한 대상이 된다. 가로채기 프락시는 먼저 차단된 커넥션이 해당 호스트의 유효한 IP 주소를 대상으로 하는지 확인하지 않고 Host의 field-value 사용하여 요청을 내부 서버로 리다이렉트 하거나 공유 캐시에서 캐시 키로 사용하는 경우 특히 취약하다.

A server MUST respond with a 400 (Bad Request) status code to any HTTP/1.1 request message that lacks a Host header field and to any request message that contains more than one Host header field or a Host header field with an invalid field-value.

서버는 Host 헤더 필드가 없는 HTTP/1.1 요청 메시지와 Host 헤더 필드의 유효하지 않은 field-value 또는 두 개 이상의 Host 헤더 필드를 포함하는 요청 메시지에 400 (Bad Request) 상태 코드로 반드시 응답해야 한다.(MUST)

5.5 Effective Request URI

Since the request-target often contains only part of the user agent’s target URI, a server reconstructs the intended target as an "effective request URI" to properly service the request. This reconstruction involves both the server's local configuration and information communicated in the request-target, Host header field, and connection context.

request-target이 종종 사용자 에이전트의 대상 URI의 일부만 포함한 이래로, 서버는 요청을 적절하게 처리하기 위해 원하는 대상을 “effcetive request URI”로 재구성한다. 재구성에는 서버의 로컬 구성과 request-target에 있는 통신 정보, Host 헤더 필드 및 커넥션 컨텍스트에서 전달되는 정보를 모두 포함한다.

For a user agent, the effective request URI is the target URI.

사용자 에이전트의 경우 유효한 요청 URI가 대상 URI이다.

If the request-target is in absolute-form, the effective request URI is the same as the request-target. Otherwise, the effective request URI is constructed as follows:

request-target이 absolute-form이면 유효한 요청 URI는 request-target과 동일하다. 그렇지 않으면 유효한 요청 URI가 다음과 같이 구성된다:

If the server's configuration (or outbound gateway) provides a fixed URI scheme, that scheme is used for the effective request URI. Otherwise, if the request is received over a TLS-secured TCP connection, the effective request URI's scheme is "https"; if not, the scheme is “http".

서버의 구성(또는 아웃바운드 게이트 웨이)이 고정된 URI scheme을 제공하는 경우 해당 scheme이 효과적인 요청 URI에 사용된다. 그렇지 않으면, 요청이 TLS-secured TCP 커넥션을 통해 수신되면 유효한 요청 URI의 scheme은 "https"이고, 그렇지 않은 경우에는 "http"이다.

If the server's configuration (or outbound gateway) provides a fixed URI authority component, that authority is used for the effective request URI. If not, then if the request-target is in authority-form, the effective request URI's authority component is the same as the request-target. If not, then if a Host header field is supplied with a non-empty field-value, the authority component is the same as the Host field-value. Otherwise, the authority component is assigned the default name configured for the server and, if the connection's incoming TCP port number differs from the default port for the effective request URI’s scheme, then a colon (":") and the incoming port number (in decimal form) are appended to the authority component.

서버의 구성(또는 아웃바운드 게이트 웨이)에서 고정된 URI 권한 구성 요소를 제공하면 해당 권한이 유효한 요청 URI에 사용된다. 그렇지 않으면 request-target이 authority-form인 경우 유효한 요청 URI의 권한 구성 요소가 request-target과 동일하다. 그렇지 않으면, Host 헤더 필드에 비어 있지 않은 field-value가 제공된 경우 권한 구성 요소는 Host field-value과 동일 하다. 그렇지 않으면, 권한 구성 요소에 서버에 대해 구성된 기본 이름이 할당되고 커넥션의 TCP포트 번호가 유효한 요청 URI 구성의 기본 포트와 다르면 콜론(":")과 들어오는 포트 번호(십진수 형식)가 권한 구성 요소에 추가된다.

If the request-target is in authority-form or asterisk-form, the effective request URI's combined path and query component is empty. Otherwise, the combined path and query component is the same as the request-target.

request-target이 authority-form 또는 asterisk-form인 경우 유효한 요청 URI의 결합된 경로 및 쿼리 구성 요소가 비어 있다. 그렇지 않으면 결합된 경로 및 쿼리 구성 요소가 request-target과 동일하다.

The components of the effective request URI, once determined as above, can be combined into absolute-URI form by concatenating the scheme, "://", authority, and combined path and query component.

유효한 요청 URI의 구성 요소는 일단 위에서 결정되면 scheme, “://“, 권한, 결합된 경로 및 쿼리 구성요소로 이어지는 absolute-URI 형식으로 결합될 수 있다.

Example 1: the following message received over an insecure TCP connection

예시 1: 다음의 메시지는 보안되지 않는 TCP 커넥션에서 수신했다.

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.example.org:8080
```

has an effective request URI of

유효한 요청 URI을 가졌다.

```
http://www.example.org:8080/pub/WWW/TheProject.html
```

Example 2: the following message received over a TLS-secured TCP connection

예시 2: 다음의 메시지는 TLS-secured TCP 커넥션을 통해 수신했다.

```
OPTIONS * HTTP/1.1
Host: www.example.org
```

has an effective request URI of

유효한 요청 URI을 가졌다.

https://www.example.org

Recipients of an HTTP/1.0 request that lacks a Host header field might need to use heuristics (e.g., examination of the URI path for something unique to a particular host) in order to guess the effective request URI's authority component.

Host 헤더 필드가 없는 HTTP/1.0의 요청 수신자는 유효한 요청 권한 구성 요소를 추측하기 위해 휴리스틱스(e.g., 특정 호스트에 고유한 항목에 대한 URI 경로 검사)를 사용해야 할 수도 있다.

Once the effective request URI has been constructed, an origin server needs to decide whether or not to provide service for that URI via the connection in which the request was received. For example, the request might have been misdirected, deliberately or accidentally, such that the information within a received request-target or Host header field differs from the host or port upon which the connection has been made. If the connection is from a trusted gateway, that inconsistency might be expected; otherwise, it might indicate an attempt to bypass security filters, trick the server into delivering non-public content, or poison a cache. See Section 9 for security considerations regarding message routing.

일단 유효한 요청 URI가 구성되면, 원서버는 요청을 수신한 커넥션을 통해 해당 URI에 대한 서비스를 제공할지 여부를 결정해야 한다. 예를 들어, 수신된 request-target 또는 Host 헤더 필드 내의 정보가 커넥션이 만들어진 호스트 또는 포트와 다르도록 요청이 의도적으로 또는 실수로 잘못 전달되었을 수 있다. 커넥션이 신뢰할 수 있는 게이트 웨이에서 생성된 경우에는 이러한 불일치가 예상될 수 있다. 그렇지 않으면 보안 필터를 무시하거나 서버를 속여 공개되지 않은 콘텐츠를 제공하거나 캐시를 손상시킬 수 있다. 메시지 라우팅과 관련된 보안 고려 사항은 Section 9를 참조해라.

5.6 Associating a Response to a Request

HTTP does not include a request identifier for associating a given request message with its corresponding one or more response messages. Hence, it relies on the order of response arrival to correspond exactly to the order in which requests are made on the same connection. More than one response message per request only occurs when one or more informational responses (1xx, see Section 6.2 of [RFC7231]) precede a final response to the same request.

HTTP는 지정된 요청 메시지를 하나 이상의 응답 메시지와 연결하기 위한 요청 식별자가 포함되어 있지 않다. 따라서, 응답 도착 순서에 따라 동일한 커넥션에서 요청이 이루어지는 순서에 정확히 일치한다. 동일한 요청에 대한 최종 응답에 앞서 하나 이상의 정보 응답(1xx, [RFC7231]의 Section 6.2 참조)이 있을 경우에만 요청당 두개 이상의 응답 메시지가 발생한다.

A client that has more than one outstanding request on a connection MUST maintain a list of outstanding requests in the order sent and MUST associate each received response message on that connection to the highest ordered request that has not yet received a final (non-1xx) response.

커넥션에 대해 두개 이상의 결정되지 못한 요청이 있는 클라이언트는 보낸 순서대로 미결정 요청 목록을 반드시 유지해야 하며(MUST) 해당 커넥션에 대해 수신된 각 응답 메시지를 아직 최종 (non-1xx) 응답을 받지 않은 가장 우선순위가 높은 요청과 반드시 연결해야 한다.(MUST)

5.7 Message Forwarding

As described in Section 2.3, intermediaries can serve a variety of roles in the processing of HTTP requests and responses. Some intermediaries are used to improve performance or availability. Others are used for access control or to filter content. Since an HTTP stream has characteristics similar to a pipe-and-filter architecture, there are no inherent limits to the extent an intermediary can enhance (or interfere) with either direction of the stream.

Section 2.3에서 설명한 바와 같이, 중개자는 HTTP 요청 및 응답 처리에서 다양한 역할을 수행할 수 있다. 일부 중개자들은 성능이나 가용성을 향상시키기 위해 사용된다. 나머지 중개자는 액세스 제어 또는 콘텐츠 필터링에 사용된다. HTTP 스트림은 pipe-and-filter 아키텍처와 유사한 특성을 가지고 있기 때문에, 중개자가 스트림의 어느 방향으로든 향상(또는 interface)할 수 있는 범위에는 본질적인 제한이 없다.

An intermediary not acting as a tunnel MUST implement the Connection header field, as specified in Section 6.1, and exclude fields from being forwarded that are only intended for the incoming connection.

터널 역할을 하지 않는 중개자는 Section 6.1에 지정된 Connection 헤더 필드를 반드시 구현해야 하며, 들어오는 커넥션을 위해 전달되는 필드는 반드시 제외해야 한다.(MUST)

An intermediary MUST NOT forward a message to itself unless it is protected from an infinite request loop. In general, an intermediary ought to recognize its own server names, including any aliases, local variations, or literal IP addresses, and respond to such requests directly.

메시지가 무한 요청 루프로부터 보호되지 않는 한, 중개자는 메시지를 자기 자신에게 전달해서는 안 된다.(MUST NOT) 일반적으로, 중개자는 자신의 서버 이름(별칭, 로컬 변형 또는 리터럴 IP 주소 포함하는)을 인식하고 이러한 요청에 직접 응답해야 한다.

5.7.1 Via

The "Via" header field indicates the presence of intermediate protocols and recipients between the user agent and the server (on requests) or between the origin server and the client (on responses), similar to the "Received" header field in email (Section 3.6.7 of [RFC5322]). Via can be used for tracking message forwards, avoiding request loops, and identifying the protocol capabilities of senders along the request/response chain.

“Via” 헤더 필드는 이메일의 “Received" 헤더 필드와 유사하게 사용자 에이전트와 서버(요청 시)또는 원서버와 클라이언트(응답 시)사이에 중개자 프로토콜과 수신자가 있음을 나타낸다.([RFC5322]의 Section 3.6.7). Via 는 메시지 전송을 추적하여 요청 루프를 방지하고 요청/응답 체인을 따라 보낸 사람의 프로토콜의 역량을 식별하는 데 사용할 수 있다.

```
Via = 1#( received-protocol RWS received-by [ RWS comment ] )

received-protocol = [ protocol-name "/" ] protocol-version
                        ; see Section 6.7
received-by       = ( uri-host [ ":" port ] ) / pseudonym
pseudonym         = token
```

Multiple Via field values represent each proxy or gateway that has forwarded the message. Each intermediary appends its own information about how the message was received, such that the end result is ordered according to the sequence of forwarding recipients.

다중 Via 필드 값은 메시지를 전송한 각 프락시 또는 게이트웨이를 알려준다. 각 중개자는 최종 결과가 전달 수신자의 순서에 따라 정렬되도록, 메시지가 어떻게 수신되었는지에 대한 자신의 정보를 추가한다.

A proxy MUST send an appropriate Via header field, as described below, in each message that it forwards. An HTTP-to-HTTP gateway MUST send an appropriate Via header field in each inbound request message and MAY send a Via header field in forwarded response messages.

프락시는 전달되는 각 메시지에서 아래 설명된 대로 적절한 Via 헤더 필드를 보내야 한다.(MUST) HTTP-to-HTTP 게이트 웨이는 각 인바운드 요청 메시지에서 적절한 Via 헤더 필드를 보내야 하며(MUST) 전달된 응답 메시지로 Via 헤더 필드를 보낼 수 있다.(MAY)

For each intermediary, the received-protocol indicates the protocol and protocol version used by the upstream sender of the message. Hence, the Via field value records the advertised protocol capabilities of the request/response chain such that they remain visible to downstream recipients; this can be useful for determining what backwards-incompatible features might be safe to use in response, or within a later request, as described in Section 2.6. For brevity, the protocol-name is omitted when the received protocol is HTTP.

각 중개자에 대해서, received-protocol은 메시지의 업스트림 발신자가 사용하는 프로토콜과 프로토콜 버전을 나타낸다. 따라서 Via 필드 값은 요청/응답 체인의 알려진 프로토콜 능력을 기록하여 다운스트림 수신자가 계속해서 볼 수 있게 한다; Via 필드 값은 Section 2.6에서 설명한 것처럼 하위 호환하지 않는 어떤 기능을 안전하게 사용할 수 있는지를 결정하는 데 유용하다. 간결하게 하기 위해, 수신된 프로토콜이 HTTP일 때, protocol-name은 생략된다.

The received-by portion of the field value is normally the host and optional port number of a recipient server or client that subsequently forwarded the message. However, if the real host is considered to be sensitive information, a sender MAY replace it with a pseudonym. If a port is not provided, a recipient MAY interpret

that as meaning it was received on the default TCP port, if any, for the received-protocol.

일반적으로 필드 값의 received-by 부분은 그 뒤에 메시지를 전달한 수신자 서버 또는 클라이언트의 호스트 및 선택적 포트 번호이다. 그러나 실제 호스트가 중요한 정보로 간주되는 경우 보내는 사람이 이를 가명으로 대체할 수 있다. 포트가 제공되지 않은 경우 수신자는 해당 포트를 received-protocol의 기본 TCP 포트(있는 경우)에서 수신된 것으로 해석할 수 있다.

A sender MAY generate comments in the Via header field to identify the software of each recipient, analogous to the User-Agent and Server header fields. However, all comments in the Via field are optional, and a recipient MAY remove them prior to forwarding the message.

발신자는 Via 헤더 필드에 User-Agent 및 Server 헤더 필드와 유사한 주석을 생성하여 각 수신자의 소프트웨어를 식별할 수 있다.(MAY) 그러나 Via 필드의 모든 주석은 선택 사항이며, 수신자는 메시지를 전달하기 전에 해당 주석을 제거할 수 있다.(MAY)

For example, a request message could be sent from an HTTP/1.0 user agent to an internal proxy code-named "fred", which uses HTTP/1.1 to forward the request to a public proxy at p.example.net, which completes the request by forwarding it to the origin server at www.example.com. The request received by www.example.com would then have the following Via header field:

예를 들어, 요청 메시지를 HTTP 1.0 사용자 에이전트에서 내부 프락시 code-named “fred”로 보낼 수 있다. 이 프락시는 HTTP/1.1을 사용하여 공용 프락시인 p.example.net로 요청을 전달하고, www.example.com인 원서버로 요청을 전달함으로써 완료한다. www.example.com에서 수신한 요청에는 다음 Via 헤더 필드가 포함된다:

```
Via: 1.0 fred, 1.1 p.example.net
```

An intermediary used as a portal through a network firewall SHOULD NOT forward the names and ports of hosts within the firewall region unless it is explicitly enabled to do so. If not enabled, such an intermediary SHOULD replace each received-by host of any host behind the firewall by an appropriate pseudonym for that host.

네트워크 방화벽을 통해 포털로 사용되는 중개자는 명시적으로 설정되지 않은 한, 방화벽 영역 내에서 호스트의 이름과 포트를 전달해서는 안 된다.(SHOULD NOT) 사용하도록 설정되지 않은

경우, 그러한 중개자는 방화벽 뒤에 있는 모든 호스트의 각 received-by 호스트를 해당 호스트에 대한 적절한 필명으로 대체해야 한다.(SHOULD)

An intermediary MAY combine an ordered subsequence of Via header field entries into a single such entry if the entries have identical received-protocol values. For example,

항목이 동일한 received-protocol 값을 갖는 경우 중개자는 Via 헤더 필드 항목이 정렬된 하위 시퀀스를 단일 항목으로 결합 할 수 있다.(MAY) 예를들어

Via: 1.0 ricky, 1.1 ethel, 1.1 fred, 1.0 lucy

could be collapsed to

Via: 1.0 ricky, 1.1 mertz, 1.0 lucy

A sender SHOULD NOT combine multiple entries unless they are all under the same organizational control and the hosts have already been replaced by pseudonyms. A sender MUST NOT combine entries that have different received-protocol values.

발신자는 모두 동일한 조직 통제 하에 있고 호스트가 이미 익명으로 대체되지 않는 한 여러 항목을 결합해서는 안 된다.(SHOULD NOT) 발신자는 received-protocol 값이 다른 항목을 결합해서는 안 된다.(MUST) (e.g,. 1.0과 1.1을 결합하면 안 된다)

5.7.2 Transformations

Some intermediaries include features for transforming messages and their payloads. A proxy might, for example, convert between image formats in order to save cache space or to reduce the amount of traffic on a slow link. However, operational problems might occur when these transformations are applied to payloads intended for critical applications, such as medical imaging or scientific data analysis, particularly when integrity checks or digital signatures are used to ensure that the payload received is identical to the original.

일부 중개자는 메시지와 페이 로드를 변환하기 위한 기능을 포함한다. 예를 들어 프락시는 캐시 공간을 절약하거나 느린 링크의 트래픽 양을 줄이기 위해 이미지 형식 간에 변환할 수 있다. 그러나 이러한 변환이 의료 이미징 또는 과학 데이터 분석과 같은 중요한 애플리케이션을 위한 페이 로드에 적용될 때, 특히 수신된 페이 로드가 원래 페이 로드와 동일한지 확인하기 위해 무결성 검사 또는 디지털 서명을 사용할 때 작동 문제가 발생할 수 있다.

An HTTP-to-HTTP proxy is called a "transforming proxy" if it is designed or configured to modify messages in a semantically meaningful way (i.e., modifications, beyond those required by normal HTTP processing, that change the message in a way that would be significant to the original sender or potentially significant to downstream recipients). For example, a transforming proxy might be acting as a shared annotation server (modifying responses to include references to a local annotation database), a malware filter, a format transcoder, or a privacy filter. Such transformations are presumed to be desired by whichever client (or client organization) selected the proxy.

의미론적으로 의미 있는 방식으로 메시지를 수정하도록 설계되거나 구성된 HTTP-to-HTTP 프락시를 “transforming proxy"라고 한다. (i.e., 정상적인 HTTP 처리에 필요한 수정을 넘어 메시지가 원래 보낸 사람에게 중요하거나 다운스트림 수신자에게 중요한 방식으로 변경되는 경우). 예를 들어 변환 프락시는 공유된 주석 서버(로컬 주석 데이터베이스에 대한 참조를 포함하도록 응답을 수정하는 것), 맬웨어 필터, 포맷 변환기 또는 개인 정보 필터로 작동할 수 있다. 이러한 변환은 프락시를 선택한 클라이언트(또는 클라이언트 조직)가 바라는 것으로 가정한다.

If a proxy receives a request-target with a host name that is not a fully qualified domain name, it MAY add its own domain to the host name it received when forwarding the request. A proxy MUST NOT change the host name if the request-target contains a fully qualified domain name.

프락시가 fully-qualified 도메인 이름이 아닌 호스트 이름을 가진 request-target을 수신하는 경우 요청을 전달할 때 수신한 호스트 이름에 자신의 도메인을 추가할 수 있다. request-target에 fully-qualified 도메인 이름이 포함된 경우 프락시는 호스트 이름을 변경해서는 안 된다.(MUST NOT)

A proxy MUST NOT modify the "absolute-path" and "query" parts of the received request-target when forwarding it to the next inbound server, except as noted above to replace an empty path with "/" or “*".

프락시는 수신된 request-target을 다음 인바운드 서버로 전달할 때 빈 경로를 “/"또는 “*" 로 바꾸기 위해 위에서 설명한 경우를 제외하고 수신된 request-target의 “absolute-path” 및 “query" 부분을 수정해서는 안 된다.(MUST NOT)

A proxy MAY modify the message body through application or removal of a transfer coding (Section 4).

프락시는 전송 코딩의 적용 또는 제거를 통해 메시지 본문을 수정할 수 있다. (Section 4)

A proxy MUST NOT transform the payload (Section 3.3 of [RFC7231]) of a message that contains a no-transform cache-control directive (Section 5.2 of [RFC7234]).

프락시는 no-trasnform cache-control 지시어를 포함한 메시지의 페이 로드([RFC7231]의 Section 3.3)를 변환해서는 안 된다.(MUST NOT) ([RFC7234]의 Section 5.2).

A proxy MAY transform the payload of a message that does not contain a no-transform cache-control directive. A proxy that transforms a payload MUST add a Warning header field with the warn-code of 214 ("Transformation Applied") if one is not already in the message (see Section 5.5 of [RFC7234]). A proxy that transforms the payload of a 200 (OK) response can further inform downstream recipients that a transformation has been applied by changing the response status code to 203 (Non-Authoritative Information) (Section 6.3.4 of [RFC7231]).

프락시는 no-trasnform cache-control 지시어를 포함하지 않는 메시지의 페이 로드를 변환할 수 있다.(MAY) 페이 로드를 변환하는 프락시는 214 warn-code (“Transformation Applied”)가 아직 메시지에 없는 경우 Warning 헤더 필드에 추가해야 한다.(MUST) ([RFC7434]의 Section 5.5 참조). 200 (OK) 응답의 페이 로드를 변환하는 프락시는 응답 상태 코드를 203 (Non-Authritative Information)로 변경하여 다운스트림 수신자에게 변환이 적용되었음을 추가로 알릴 수 있다. ([RFC7131]의 Section 6.3.4).

A proxy SHOULD NOT modify header fields that provide information about the endpoints of the communication chain, the resource state, or the selected representation (other than the payload) unless the field's definition specifically allows such modification or the modification is deemed necessary for privacy or security.

프락시는 필드의 정의가 이러한 수정을 특별히 허용하거나 수정이 개인 정보 또는 보안을 위해 필요한 것으로 간주되지 않는 한 통신 체인의 엔드 포인트, 리소스 상태 또는 선택된 표현(페이 로드 제외)에 대한 정보를 제공하는 헤더 필드를 수정하면 안 된다.(SHOULD NOT)

6. Connection Management

HTTP messaging is independent of the underlying transport- or session-layer connection protocol(s). HTTP only presumes a reliable transport with in-order delivery of requests and the corresponding in-order delivery of responses. The mapping of HTTP request and response structures onto the data units of an underlying transport protocol is outside the scope of this specification.

HTTP 메시징은 기반을 이루는 전송 또는 세션 계층 커넥션 프로토콜과 독립적이다. HTTP는 in-order 요청 전송과 그에 상응하는 in-order 응답 전송이 있는 신뢰할 수 있는 전송만 가정한다. HTTP 요청 및 응답 구조의 기반을 이루는 전송 프로토콜의 데이터 단위에 매핑 하는 작업은 이 명세의 범위를 벗어난다.

As described in Section 5.2, the specific connection protocols to be used for an HTTP interaction are determined by client configuration and the target URI. For example, the "http" URI scheme (Section 2.7.1) indicates a default connection of TCP over IP, with a default TCP port of 80, but the client might be configured to use a proxy via some other connection, port, or protocol.

Section 5.2에서 설명한 대로, HTTP 상호 작용에 사용되는 특정 커넥션 프로토콜은 클라이언트 구성과 대상 URI에 의해 결정된다. 예를 들어 “http" URI scheme(Section 2.7.1)은 기본 TCP 포트 80을 사용하는 TCP/IP를 통한 기본 커넥션을 나타내지만 클라이언트가 다른 커넥션, 포트 또는 프로토콜을 통해 프락시를 사용하도록 구성될 수 있다.

HTTP implementations are expected to engage in connection management, which includes maintaining the state of current connections, establishing a new connection or reusing an existing connection, processing messages received on a connection, detecting connection failures, and closing each connection. Most clients maintain multiple connections in parallel, including more than one connection per server endpoint. Most servers are designed to maintain thousands of concurrent connections, while controlling request queues to enable fair use and detect denial-of-service attacks.

HTTP 구현은 현재 커넥션 상태 유지, 새 커넥션 설정 또는 기존 커넥션 재사용, 커넥션에서 수신된 메시지 처리, 커넥션 실패 탐지 및 각 커넥션을 닫는 것을 포함하는 커넥션 관리에 책임질 것으로 예상된다. 대부분의 클라이언트는 서버 엔드포인트당 두개 이상의 커넥션을 포함하여 여러 커넥션을 병렬로 유지한다. 대부분의 서버는 수천개의 동시 커넥션을 유지하도록 설

계되어 있으며, 요청 대기 열을 제어하여 공정한 사용을 가능하게 하고 denial-of-service 공격을 탐지한다.

6.1 Connection

The "Connection" header field allows the sender to indicate desired control options for the current connection. In order to avoid confusing downstream recipients, a proxy or gateway MUST remove or replace any received connection options before forwarding the message.

“Connection” 헤더 필드를 통해 발신자는 현재 커넥션에 대해 원하는 제어 옵션을 표시할 수 있다. 메시지를 전달하기 전에, 프락시 또는 게이트웨이가 수신된 커넥션 옵션을 반드시 제거하거나 바꾸어야 한다. (MUST)

When a header field aside from Connection is used to supply control information for or about the current connection, the sender MUST list the corresponding field-name within the Connection header field. A proxy or gateway MUST parse a received Connection header field before a message is forwarded and, for each connection-option in this field, remove any header field(s) from the message with the same name as the connection-option, and then remove the Connection header field itself (or replace it with the intermediary's own connection options for the forwarded message).

현재 커넥션에 대한 제어 정보를 제공하기 위해 Connection 이외의 헤더 필드를 사용할 경우, 발신자는 Connection 헤더 필드에 해당 필드 이름을 반드시 나열해야 한다.(MUST) 프락시 또는 게이트 웨이는 메시지를 전달하기 전에 수신된 Connection 헤더 필드를 분석해야 하며 (MUST), 이 필드의 각 connection-option에 대해 connection-option과 동일한 이름을 가진 메시지에서 헤더 필드를 제거한 다음 Connection 헤더 필드 자체를 제거한다.(또는 전송된 메시지를 위해 중개자의 자체 커넥션 옵션들과 함께 Connection 헤더 필드를 대체한다)

Hence, the Connection header field provides a declarative way of distinguishing header fields that are only intended for the immediate recipient ("hop-by-hop") from those fields that are intended for all recipients on the chain ("end-to-end"), enabling the message to be self-descriptive and allowing future connection-specific extensions to be deployed without fear that they will be blindly forwarded by older intermediaries.

따라서, Connection 헤더 필드는 직접적인 수신자("hop-by-hop")만을 위한 헤더 필드를 체인의 모든 수신자를 위한 필드("end-to-end")와 구분하여 메시지를 자체 설명할 수 있고, 향후 connection-specific 확장을 오래된 중개자들에 의해 맹목적으로 전달될 것이라는 두려움 없이 배포하는것을 허용하는 선언적인 방법을 제공한다.

The Connection header field's value has the following grammar:

Connection 헤더 필드의 값은 다음과 같은 문법을 가진다.

```
Connection        = 1#connection-option
connection-option = token
```

Connection options are case-insensitive.

Connection option은 대소문자를 구분하지 않는다.

A sender MUST NOT send a connection option corresponding to a header field that is intended for all recipients of the payload. For example, Cache-Control is never appropriate as a connection option (Section 5.2 of [RFC7234]).

발신자는 페이 로드의 모든 수신자를 위한 헤더 필드에 해당하는 커넥션 옵션을 보내면 안 된다. (MUST NOT) 예를 들어, Cache-Control은 커넥션 옵션으로는 적합하지 않다([RFC7234]의 Section 5.2).

The connection options do not always correspond to a header field present in the message, since a connection-specific header field might not be needed if there are no parameters associated with a connection option. In contrast, a connection-specific header field that is received without a corresponding connection option usually indicates that the field has been improperly forwarded by an intermediary and ought to be ignored by the recipient.

커넥션 옵션과 관련된 매개 변수가 없는 경우 connection-specific 헤더 필드가 필요하지 않을 수 있으므로 커넥션 옵션이 메시지에 있는 헤더 필드와 항상 일치하지는 않는다. 대조적으로, 해당 커넥션 옵션 없이 수신된 connection-specific 헤더 필드는 일반적으로 중개자에 의해 부적절하게 전달되었음을 나타내므로 수신자는 해당 필드를 무시해야 한다.

When defining new connection options, specification authors ought to survey existing header field names and ensure that the new connection option does not share the same name as an already deployed header field. Defining a new connection option essentially reserves that potential field-name for carrying additional information related to the connection option, since it would be unwise for senders to use that field-name for anything else.

새 커넥션 옵션을 정의할 때 명세 작성자는 기존 헤더 필드 이름을 조사하고 새 커넥션 옵션이 이미 배포된 헤더 필드와 동일한 이름을 공유하지 않는지 확인해야 한다. 새 커넥션 옵션을 정의하면 기본적으로 커넥션 옵션과 관련된 추가 정보를 전달할 수 있는 잠재적인 field-name을 보유한다. 발신자가 해당 field-name을 다른 용도로 사용하는 것은 현명하지 못하기 때문이다.

The "close" connection option is defined for a sender to signal that this connection will be closed after completion of the response. For example,

“close" 커넥션 옵션은 응답 완료 후 이 커넥션이 닫힘을 발신자에게 알리기 위해 정의된다. 예를 들어,

    Connection: close

in either the request or the response header fields indicates that the sender is going to close the connection after the current request/response is complete (Section 6.6).

요청 또는 응답 헤더 필드는 현재 요청/응답이 완료 후 발신자가 커넥션을 닫을 것임을 나타낸다.(Section 6.6)

A client that does not support persistent connections MUST send the "close" connection option in every request message.

영속적 커넥션을 지원하지 않는 클라이언트는 모든 요청 메시지에서 “close" 커넥션 옵션을 보내야 한다.(MUST)

A server that does not support persistent connections MUST send the "close" connection option in every response message that does not have a 1xx (Informational) status code.

영속적 커넥션을 지원하지 않는 서버는 1xx (Informational) 상태 코드가 없는 모든 응답 메시지에서 “close" 커넥션 옵션을 보내야 한다.(MUST)

6.2 Establishment

It is beyond the scope of this specification to describe how connections are established via various transport- or session-layer protocols. Each connection applies to only one transport link.

다양한 전송 또는 세션 계층 프로토콜을 통해 커넥션을 설정하는 방법을 설명하는 것은 이 규격의 범위를 벗어난다. 각 커넥션은 하나의 전송 링크에만 적용된다.

6.3 Persistence

HTTP/1.1 defaults to the use of "persistent connections", allowing multiple requests and responses to be carried over a single connection. The "close" connection option is used to signal that a connection will not persist after the current request/ response. HTTP implementations SHOULD support persistent connections.

HTTP/1.1은 기본적으로 “persistent connections” 을 사용하여 여러 요청과 응답을 단일 커넥션에서 수행할 수 있도록 한다. “close” 커넥션 옵션은 현재 요청/응답 후 커넥션이 지속되지 않음을 알리는 데 사용된다. HTTP 구현은 영속적 커넥션을 지원해야 한다. (SHOULD)

A recipient determines whether a connection is persistent or not based on the most recently received message's protocol version and Connection header field (if any):

수신자는 최근에 수신한 메시지의 프로토콜 버전 및 Connection 헤더 필드(있는 경우)를 기반으로 커넥션이 지속되는지 여부를 확인한다:

o If the "close" connection option is present, the connection will not persist after the current response; else,

“close” 커넥션 옵션이 존재하면, 커넥션은 현재 응답 이후에 영속적이 아닐것이다; 다른,

o If the received protocol is HTTP/1.1 (or later), the connection will persist after the current response; else,

수신된 프로토콜이 HTTP/1.1(또는 이상)이면, 커넥션은 현재 응답 이후에 영속적일 것이다; 다른,

o If the received protocol is HTTP/1.0, the "keep-alive" connection option is present, the recipient is not a proxy, and the recipient wishes to honor the HTTP/1.0 "keep-alive" mechanism, the connection will persist after the current response; otherwise,

수신된 프로토콜이 HTTP/1.0이고, “keep-alive” 커넥션 옵션이 존재하고. 수신자가 프락시가 아니며, 수신자는 HTTP/1.0의 “keep-alive” 메커니즘을 준수하고자 한다면, 커넥션은 현재 응답 이후에 영속적일 것이다. 그게 아니면

o The connection will close after the current response.

커넥션은 현재 응답 이후에 닫을것이다.

A client MAY send additional requests on a persistent connection until it sends or receives a "close" connection option or receives an HTTP/1.0 response without a "keep-alive" connection option.

클라이언트는 영속적 커넥션에서 “close” 커넥션 옵션을 보내거나 받을 때까지 또는 “keep-alive” 커넥션 옵션이 없는 HTTP/1.0 응답을 받을 때 추가 요청을 보낼 수 있다.(MAY)

In order to remain persistent, all messages on a connection need to have a self-defined message length (i.e., one not defined by closure of the connection), as described in Section 3.3. A server MUST read the entire request message body or close the connection after sending its response, since otherwise the remaining data on a persistent connection would be misinterpreted as the next request. Likewise, a client MUST read the entire response message body if it intends to reuse the same connection for a subsequent request.

영속성을 유지하기 위해서는 Section 3.3 에서 설명한 대로, 커넥션의 모든 메시지는 자체 정의된 메시지 길이를 가져야 한다(즉, 커넥션 종료에 의해 정의되지 않은 메시지 길이). 서버는 응답을 보낸 후 요청 메시지 본문 전체를 반드시 읽거나 또는 커넥션을 닫아야 한다.(MUST) 그렇지 않으면 영속적 커넥션의 나머지 데이터가 다음 요청으로 잘못 해석될 수 있기 때문이다. 마찬가지로, 후속 요청에 대해 동일한 커넥션을 재사용하려고 한다면 응답 메시지 본문 전체를 읽어야 한다.(MUST)

A proxy server MUST NOT maintain a persistent connection with an HTTP/1.0 client (see Section 19.7.1 of [RFC2068] for information and discussion of the problems with the Keep-Alive header field implemented by many HTTP/1.0 clients).

프락시 서버는 HTTP/1.0 클라이언트와 영속적 커넥션을 유지해서는 안 된다.(MUST NOT) (많은 HTTP/1.0 클라이언트에 의해 구현된 Keep-Alive 헤더 필드의 문제들에 대한 정보와 논의인 [RFC2068]의 Section 19.7.1를 참조)

See Appendix A.1.2 for more information on backwards compatibility with HTTP/1.0 clients.

HTTP/1.0 클라이언트와 하위 호환성에 관한 더 많은 정보는 Appendix A.1.2 참조.

6.3.1 Retrying Requests

Connections can be closed at any time, with or without intention. Implementations ought to anticipate the need to recover from asynchronous close events.

커넥션은 의도가 있거나 없거나 언제든지 닫힐 수 있다. 비동기적으로 닫히는 상황으로부터 복구 하기 위한 요구를 만족해야 한다.

When an inbound connection is closed prematurely, a client MAY open a new connection and automatically retransmit an aborted sequence of requests if all of those requests have idempotent methods (Section 4.2.2 of [RFC7231]). A proxy MUST NOT automatically retry non-idempotent requests

인바운드 커넥션이 조기에 닫힐 때, 모든 요청이 멱등한 메서드인 경우, 클라이언트는 새 커넥션을 열고 중단된 요청 시퀀스를 자동으로 재전송할 수 있다.(MAY) ([RFC7231]의 Section 4.2.2). 프락시는 멱등하지 않은 아닌 요청을 자동으로 재시도하면 안 된다.(MUST NOT)

A user agent MUST NOT automatically retry a request with a non- idempotent method unless it has some means to know that the request semantics are actually idempotent, regardless of the method, or some means to detect that the original request was never applied. For example, a user agent that knows (through design or configuration) that a POST request to a given resource is safe can repeat that request automatically. Likewise, a user agent designed specifically to operate on a version control repository might be able to recover from partial failure conditions by checking the target resource revision(s) after a failed connection, reverting or fixing any changes that were partially applied, and then automatically retrying the requests that failed.

사용자 에이전트는 메서드에 관계 없이, 요청 의미론이 실제로 멱등한지 알려지 있지 않거나, 또는 원래 요청이 적용되지 않았음을 감지할 수 있는 수단이 있지 않는 한 자동으로 요청을 재시도해서는 안 된다.(MUST NOT) 예를 들어 설계 또는 구성을 통해 특정 리소스에 대한 POST 요청이 안전하다는 것을 알고 있는 사용자 에이전트는 해당 요청을 자동으로 반복할 수 있다. 마찬가지로 버전 제어 저장소에서 작동하도록 특별히 설계된 사용자 에이전트는 커넥션 실패 후, 대상 리소스 수정을 확인하고, 부분적으로 적용된 변경 내용을 되돌리거나 수정한 다음, 실패한 요청을 자동으로 재시도하여 부분적인 장애 조건에서 복구할 수 있다.

A client SHOULD NOT automatically retry a failed automatic retry.

클라이언트는 실패한 자동 재시도를 다시 자동적으로 시도 하면 안 된다.(SHOULD NOT)

6.3.2 Pipelining

A client that supports persistent connections MAY "pipeline" its requests (i.e., send multiple requests without waiting for each response). A server MAY process a sequence of pipelined requests in parallel if they all have safe methods (Section 4.2.1 of [RFC7231]), but it MUST send the corresponding responses in the same order that the requests were received.

영속적 커넥션을 지원하는 클라이언트를 “pipeline”으로 만들 수 있다.(MAY) (즉, 각 응답을 기다리지 않고 여러 요청을 보낼 수 있음). 서버는 요청이 안전한 메서드([RFC7231]의 Section 4.2.1)인 경우 연속의 파이프 라인 요청을 병렬로 처리할 수 있지만, 다만 요청을 받은 순서대로 해당 응답을 전송해야 한다.(MUST)

A client that pipelines requests SHOULD retry unanswered requests if the connection closes before it receives all of the corresponding responses. When retrying pipelined requests after a failed connection (a connection not explicitly closed by the server in its last complete response), a client MUST NOT pipeline immediately after connection establishment, since the first remaining request in the prior pipeline might have caused an error response that can be lost again if multiple requests are sent on a prematurely closed connection (see the TCP reset problem described in Section 6.6).

요청을 파이프라인으로 처리하는 클라이언트는 해당 응답을 모두 수신하기 전에 커넥션이 닫힌 경우, 응답되지 않은 요청을 재시도해야 한다.(SHOULD) 실패된 커넥션 후(마지막의 완전한 응답에서 서버에 의해 명시적으로 닫히지 않은 커넥션) 파이프라인된 요청을 재시도할 때, 일찍 닫혀버린 커넥션 위에서 여러 요청이 보내진 경우, 이전 파이프라인에서 첫번째로 남은 요청은, 다시 손실될 수 있는 잘못된 응답을 야기하기 때문에, 클라이언트는 커넥션 설립 직후 파이프라인을 처리해서는 안 된다.(MUST NOT) (Section 6.6에 설명된 TCP 재설정 문제 참조).

Idempotent methods (Section 4.2.2 of [RFC7231]) are significant to pipelining because they can be automatically retried after a connection failure. A user agent SHOULD NOT pipeline requests after a non-idempotent method, until the final response status code for that method has been received, unless the user agent has a means to detect and recover from partial failure conditions involving the pipelined sequence.

커넥션 실패 후 자동으로 재시도할 수 있기 때문에, 멱등한 메서드([RFC7231]의 Section 4.2.2)는 파이프라이닝에 중요하다. 사용자 에이전트가 파이프라인된 순서와 관련된 부분적 실패 조건을 발견하고 복구 하기 위한 방법을 가지고 있지 않는 한, 멱등하지 않은 메서드의 마지막 응답 코드를 수신할 때 까지, 멱등하지 않은 메서드 후에 요청을 파이프라인 처리 하면 안 된다.(SHOULD NOT)

An intermediary that receives pipelined requests MAY pipeline those requests when forwarding them inbound, since it can rely on the outbound user agent(s) to determine what requests can be safely pipelined. If the inbound connection fails before receiving a response, the pipelining intermediary MAY attempt to retry a sequence of requests that have yet to receive a response if the requests all have idempotent methods; otherwise, the pipelining intermediary SHOULD forward any received responses and then close the corresponding outbound connection(s) so that the outbound user agent(s) can recover accordingly.

파이프라인 요청을 수신하는 중개자는 아웃바운드 사용자 에이전트에 의존하여 안전하게 파이프라인될 수 있는 요청을 결정할 수 있기 때문에 파이프라인 요청을 인바운드 전달 시 해당

요청을 파이프라인으로 전송할 수 있다.(MAY) 요청이 모두 멱등한 메서드라면, 응답을 수신하기 전에 인바운드 커넥션이 실패하는 경우 파이프라이닝은 아직 수신하지 못한 요청의 순서를 즉시 재시도 할 수 있다. 그렇지 않는다면 파이프라이닝은 모든 수신된 응답을 즉시 전송하고 해당 아웃바운드 커넥션을 닫아야 한다.(SHOULD) 그러면 아웃바운드 사용자 에이전트는 그에 맞춰 복구 할 수 있다.

6.4 Concurrency

A client ought to limit the number of simultaneous open connections that it maintains to a given server.

클라이언트는 주어진 서버로 유지하는 동시에 열려 있는 커넥션 수를 제한해야 한다.

Previous revisions of HTTP gave a specific number of connections as a ceiling, but this was found to be impractical for many applications. As a result, this specification does not mandate a particular maximum number of connections but, instead, encourages clients to be conservative when opening multiple connections.

HTTP의 이전 개정 버전에서는 커넥션의 특정 수를 최대 한계로서 지정했지만, 대부분의 애플리케이션에서는 불가능한 것으로 확인되었다. 결과적으로, 이 명세는 지정된 최대 커넥션 수를 의무화하지 않는 대신 다중 커넥션을 열 때 클라이언트에게 보수적이 될 것을 권장한다.

Multiple connections are typically used to avoid the “head-of-line blocking" problem, wherein a request that takes significant server-side processing and/or has a large payload blocks subsequent requests on the same connection. However, each connection consumes server resources. Furthermore, using multiple connections can cause undesirable side effects in congested networks.

동일한 커넥션에서 요청이 서버 측 처리에 상당한 시간이 소요되거나 큰 페이 로드 블록이 있는 경우(한 가지 경우 또는 두 가지 모두), “head-of-line blocking” 문제를 피하기 위해 일반적으로 다중 커넥션이 사용된다. 그러나 각 커넥션에는 서버 리소스를 사용한다. 게다가, 다중 커넥션을 사용하면 혼잡한 네트워크에서 바람직하지 않은 부작용을 일으킬 수 있다.

Note that a server might reject traffic that it deems abusive or characteristic of a denial-of-service attack, such as an excessive number of open connections from a single client.

참고로 서버는 단일 클라이언트에서 열려 있는 커넥션의 과도한 수와 같은 denial-of-service 공격의 특징을 나타내거나 또는 공격으로 판단되는 트래픽을 거부할 수 있다.

6.5 Failures and Timeouts

Servers will usually have some timeout value beyond which they will no longer maintain an inactive connection. Proxy servers might make this a higher value since it is likely that the client will be making more connections through the same proxy server. The use of persistent connections places no requirements on the length (or existence) of this timeout for either the client or the server.

서버는 일반적으로 비활성 커넥션을 더 이상 유지하지 않는 timeout 값을 가지고 있다. 프락시 서버는 클라이언트가 동일한 프락시 서버를 통해 더 많은 커넥션을 만들 가능성이 있으므로 이 값을 더 높게 만들 수 있다. 영속적 커넥션을 사용하면 클라이언트나 서버에 대해 이 timeout의 길이(또는 존재)에 대한 요구사항이 없다.

A client or server that wishes to time out SHOULD issue a graceful close on the connection. Implementations SHOULD constantly monitor open connections for a received closure signal and respond to it as appropriate, since prompt closure of both sides of a connection enables allocated system resources to be reclaimed.

timeout을 원하는 클라이언트나 서버는 커넥션을 적절하게 닫을 수 있어야 한다.(SHOULD) 커넥션의 양쪽을 모두 신속하게 닫으면 할당된 시스템 리소스를 회수할 수 있기 때문에, 구현은 수신된 종료 신호에 대해 열린 커넥션을 지속적으로 모니터링하고 적절하게 응답해야 한다. (SHOULD)

A client, server, or proxy MAY close the transport connection at any time. For example, a client might have started to send a new request at the same time that the server has decided to close the “idle" connection. From the server's point of view, the connection is being closed while it was idle, but from the client's point of view, a request is in progress.

클라이언트, 서버 또는 프락시는 언제든지 전송 커넥션을 닫을 수 있다.(MAY) 예를 들어, 서버가 “idle” 커넥션을 닫기로 결정한 동시에 클라이언트가 새 요청을 전송하기 시작했을 수 있다. 서버의 관점에서 커넥션은 유휴 상태일 때 닫히지만, 클라이언트의 관점에서 요청이 진행 중이다.

A server SHOULD sustain persistent connections, when possible, and allow the underlying transport's flow-control mechanisms to resolve temporary overloads, rather than terminate connections with the expectation that clients will retry. The latter technique can exacerbate network congestion.

서버는 가능한 경우 영속적 커넥션을 유지해야 하며,(SHOULD) 클라이언트가 다시 시도할 것으로 예상하여 커넥션을 종료하기보다는 기본 전송의 흐름 제어 메커니즘이 일시적 과부하를 해결할 수 있도록 해야 한다. 후자의 기술은 네트워크 정체를 악화시킬 수 있다.

A client sending a message body SHOULD monitor the network connection for an error response while it is transmitting the request. If the client sees a response that indicates the server does not wish to receive the message body and is closing the connection, the client SHOULD immediately cease transmitting the body and close its side of the connection.

메시지 본문을 보내는 클라이언트는 요청을 전송하는 동안 네트워크 커넥션을 모니터링하여 오류 응답을 확인해야 한다.(SHOULD) 서버가 메시지 본문을 수신하지 않고 커넥션을 닫고 있음을 나타내는 응답이 클라이언트에 나타나면, 클라이언트는 본문 전송을 즉시 중단하고 해당 커넥션을 닫아야 한다.(SHOULD)

6.6 Tear-down

The Connection header field (Section 6.1) provides a “close" connection option that a sender SHOULD send when it wishes to close the connection after the current request/response pair.

Connection 헤더 필드(Section 6.1)는 현재의 요청/응답 쌍 이후에 커넥션을 닫을 때 발신자가 보내야 하는(SHOULD) “close” 커넥션 옵션을 제공한다.

A client that sends a "close" connection option MUST NOT send further requests on that connection (after the one containing "close") and MUST close the connection after reading the final response message corresponding to this request.

“close” 커넥션 옵션을 보낸 클라이언트는 추가 요청을 보내지 않아야 하며(MUST NOT), (“close”가 포함된 경우) 이 요청에 해당하는 최종 응답 메시지를 읽은 후 커넥션을 반드시 닫아야 한다.(MUST)

A server that receives a "close" connection option MUST initiate a close of the connection (see below) after it sends the final response to the request that contained "close". The server SHOULD send a "close" connection option in its final response on that connection. The server MUST NOT process any further requests received on that connection.

“close” 커넥션 옵션을 수신한 서버는 “close”가 포함된 요청에 대한 최종 응답을 발송한 후 커넥션 종료를 시작해야 한다.(MUST) (아래 참조).
서버는 해당 커넥션에 대한 최종 응답에서 “close" 커넥션 옵션을 보내야 한다.(SHOULD)
서버는 해당 커넥션에 대해 수신된 추가 요청을 처리해서는 안 된다.(MUST NOT)

A server that sends a "close" connection option MUST initiate a close of the connection (see below) after it sends the response containing "close". The server MUST NOT process any further requests received on that connection.

“close” 커넥션 옵션을 보내는 서버는 “close”를 포함하는 응답을 보낸 후 커넥션 종료를 시작해야 한다.(MUST) (아래 참조).
서버는 해당 커넥션에 대해 수신된 추가 요청을 처리해서는 안 된다.(MUST NOT)

A client that receives a "close" connection option MUST cease sending requests on that connection and close the connection after reading the response message containing the "close"; if additional pipelined requests had been sent on the connection, the client SHOULD NOT assume that they will be processed by the server.

“close” 커넥션 옵션을 수신하는 클라이언트는 “close” 가 포함된 응답 메시지를 읽은 후 해당 커넥션에 대한 요청 전송을 중지하고 커넥션을 닫아야 한다.(MUST)
커넥션에 추가적인 파이프라인된 요청이 전송된 경우 클라이언트는 이러한 요청이 서버에 의해 처리될 것으로 가정해서는 안 된다.(SHOULD NOT)

If a server performs an immediate close of a TCP connection, there is a significant risk that the client will not be able to read the last HTTP response. If the server receives additional data from the client on a fully closed connection, such as another request that was sent by the client before receiving the server's response, the server's TCP stack will send a reset packet to the client; unfortunately, the reset packet might erase the client’s unacknowledged input buffers before they can be read and interpreted by the client's HTTP parser.

서버가 TCP 커넥션을 즉시 닫는 경우 클라이언트가 마지막 HTTP 응답을 읽지 못할 위험이 크다. 서버가 클라이언트로부터 서버 응답을 받기 전에 클라이언트가 보낸 다른 요청과 같이 완

전히 닫힌 커넥션에서 추가 데이터를 클라이언트로부터 수신하는 경우, 서버의 TCP 스택은 클라이언트에게 재설정 패킷을 보낸다; 안타깝게도, 재설정 패킷이 클라이언트의 인식되지 않은 입력 버퍼를 클라이언트의 HTTP 파서에 의해 읽어지거나 해석되기 전에 지울 수 있다.

To avoid the TCP reset problem, servers typically close a connection in stages. First, the server performs a half-close by closing only the write side of the read/write connection. The server then continues to read from the connection until it receives a corresponding close by the client, or until the server is reasonably certain that its own TCP stack has received the client's acknowledgement of the packet(s) containing the server's last response. Finally, the server fully closes the connection.

TCP 재설정 문제를 방지하기 위해 서버는 일반적으로 커넥션을 단계적으로 닫는다. 첫 번째로, 서버는 읽기/쓰기 커넥션의 쓰기 측인 절반만 닫는 것을 수행한다. 그런 다음 서버는 클라이언트에 의해 해당하는 close가 수신될 때까지, 또는 서버 자체의 TCP 스택이 서버의 마지막 응답을 포함하는 패킷에 대한 클라이언트의 확인을 수신한 것이 합리적으로 확인될 때까지, 커넥션에서 계속 읽는다. 마지막으로, 서버가 커넥션을 완전히 닫는다.

It is unknown whether the reset problem is exclusive to TCP or might also be found in other transport connection protocols.

재설정 문제가 TCP에만 국한된 것인지 아니면 다른 전송 커넥션 프로토콜에서도 발견될 수 있는지는 알려지지 않았다.

6.7 Upgrade

The "Upgrade" header field is intended to provide a simple mechanism for transitioning from HTTP/1.1 to some other protocol on the same connection. A client MAY send a list of protocols in the Upgrade header field of a request to invite the server to switch to one or more of those protocols, in order of descending preference, before sending the final response. A server MAY ignore a received Upgrade header field if it wishes to continue using the current protocol on that connection. Upgrade cannot be used to insist on a protocol change.

“Upgrade” 헤더 필드는 HTTP/1.1에서 동일한 커넥션의 다른 프로토콜로 전환하기 위한 간단한 메커니즘을 제공하기 위한 것이다. 클라이언트는 최종 응답을 보내기 전에 서버가 하나 이상의 프로토콜로 전환하도록 요구하는 요청의 Upgrade 헤더 필드에 프로토콜 목록을 보낼 수 있다. 서버는 수신된 Upgrade 헤더 필드를 무시할 수 있다. 해당 커넥션에서 현재 프로

토콜을 계속 사용하려면 해당 필드를 무시한다. 프로토콜 변경을 주장하는 데 Upgrade를 사용할 수 없다.

```
Upgrade          = 1#protocol

protocol         = protocol-name ["/" protocol-version]
protocol-name    = token
protocol-version = token
```

A server that sends a 101 (Switching Protocols) response MUST send an Upgrade header field to indicate the new protocol(s) to which the connection is being switched; if multiple protocol layers are being switched, the sender MUST list the protocols in layer-ascending order. A server MUST NOT switch to a protocol that was not indicated by the client in the corresponding request's Upgrade header field. A server MAY choose to ignore the order of preference indicated by the client and select the new protocol(s) based on other factors, such as the nature of the request or the current load on the server.

101 (Switching Protocols) 응답을 전송하는 서버는 반드시 커넥션 전환 대상의 새 프로토콜을 나타내기 위해 Upgrade 헤더 필드를 전송해야 한다.(MUST) 다중 프로토콜 계층이 전환되는 경우 발신자는 계층 구분 순서로 프로토콜을 나열해야 한다.(MUST) 서버는 해당 요청의 Upgrade 헤더 필드에 클라이언트에 의해 표시하지 않은 프로토콜로 전환해서는 안 된다.(MUST NOT) 서버는 클라이언트가 지정한 선호 순서를 무시하고 요청의 성격이나 서버의 현재 부하와 같은 다른 요인에 따라 새로운 프로토콜을 선택할 수 있다.(MAY)

A server that sends a 426 (Upgrade Required) response MUST send an Upgrade header field to indicate the acceptable protocols, in order of descending preference.

426(Upgrade Required) 응답을 전송하는 서버는 내림차순 기본 설정 순서로 허용가능한 프로토콜을 나타내기 위해 Upgrade 헤더 필드를 전송해야 한다.(MUST)

A server MAY send an Upgrade header field in any other response to advertise that it implements support for upgrading to the listed protocols, in order of descending preference, when appropriate for a future request.

서버는 향후 요청에 적합한 경우 내림차순 선호도순으로 나열된 프로토콜로 업그레이드하기 위한 지원을 구현한다고 알리기 위해 다른 응답으로 Upgrade 헤더 필드를 보낼 수 있다. (MAY)

The following is a hypothetical example sent by a client:

다음은 클라이언트가 보낸 가상의 예다.

```
GET /hello.txt HTTP/1.1
Host: www.example.com
Connection: upgrade
Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11
```

The capabilities and nature of the application-level communication after the protocol change is entirely dependent upon the new protocol(s) chosen. However, immediately after sending the 101 (Switching Protocols) response, the server is expected to continue responding to the original request as if it had received its equivalent within the new protocol (i.e., the server still has an outstanding request to satisfy after the protocol has been changed, and is expected to do so without requiring the request to be repeated).

프로토콜 변경 후의 애플리케이션-레벨 통신의 기능과 특성은 전적으로 선택한 새로운 프로토콜에 의존한다. 그러나 101(Switching Protocols) 응답을 전송한 직후, 서버는 마치 새로운 프로토콜 내에서 그에 상응하는 것을 받은 것처럼 원래의 요청에 계속 대응할 것으로 예상된다(즉, 프로토콜 변경 후에도 서버가 충족해야 할 미결 요청을 여전히 가지고 있으며, 요구하지 않고 이를 수행할 것으로 예상된다).

For example, if the Upgrade header field is received in a GET request and the server decides to switch protocols, it first responds with a 101 (Switching Protocols) message in HTTP/1.1 and then immediately follows that with the new protocol's equivalent of a response to a GET on the target resource. This allows a connection to be upgraded to protocols with the same semantics as HTTP without the latency cost of an additional round trip. A server MUST NOT switch protocols unless the received message semantics can be honored by the new protocol; an OPTIONS request can be honored by any protocol.

예를 들어, GET 요청으로 Upgrade 헤더 필드가 수신되고 서버가 프로토콜을 전환하기로 결정하면, 먼저 HTTP/1.1의 101 (Switching Protocols) 메시지로 응답한 다음, 대상 리소스의 GET에 대한 응답과 동등한 새로운 프로토콜의 메시지로 즉시 그 뒤를 따른다. 이를 통해 추가적인 round trip의 대기 시간 비용 없이 HTTP와 동일한 의미론의 프로토콜로 커넥션을 업그

레이드할 수 있다. 수신된 메시지 의미론이 새 프로토콜에 의해 준수될 수 있는 경우가 아니면, 서버는 스위치 프로토콜을 사용해서는 안 된다;(MUST NOT) OPTIONS 요청은 어떤 프로토콜에서도 준수될 수 있다.

The following is an example response to the above hypothetical request:

위 가상의 요청에 대한 예시적 대응은 다음과 같다.

```
HTTP/1.1 101 Switching Protocols
Connection: upgrade
Upgrade: HTTP/2.0

[... data stream switches to HTTP/2.0 with an appropriate response
(as defined by new protocol) to the "GET /hello.txt" request ...]
```

When Upgrade is sent, the sender MUST also send a Connection header field (Section 6.1) that contains an "upgrade" connection option, in order to prevent Upgrade from being accidentally forwarded by intermediaries that might not implement the listed protocols. A server MUST ignore an Upgrade header field that is received in an HTTP/1.0 request.

Upgrade 헤더 필드가 전송되면, 발신자는 나열된 프로토콜을 구현하지 않을 수도 있는 중개자에 의해 Upgrade가 실수로 전달되는 것을 방지하기 위해 “upgrade” 커넥션 옵션이 포함된 Connection 헤더 필드(Section 6.1)도 전송해야 한다.(MUST) 서버는 HTTP/1.0 요청으로 수신된 Upgrade 헤더 필드를 무시해야 한다.

A client cannot begin using an upgraded protocol on the connection until it has completely sent the request message (i.e., the client can't change the protocol it is sending in the middle of a message). If a server receives both an Upgrade and an Expect header field with the "100-continue" expectation (Section 5.1.1 of [RFC7231]), the server MUST send a 100 (Continue) response before sending a 101 (Switching Protocols) response.

클라이언트는 요청 메시지를 완전히 보낼 때까지 커넥션에서 업그레이드된 프로토콜을 사용하는 것을 시작할 수 없다.(i.e., 클라이언트가 메시지 중간에 보내는 프로토콜을 변경할 수 없음) 서버가 “100-continue” 예상과 Upgrade 및 Expect 헤더 필드([RFC7231]의 Section

5.1.1)를 모두 수신하는 경우, 서버는 101 (Switching Protocols) 응답을 보내기 전에 반드시 100 (Continue) 응답을 보내야 한다.(MUST)

The Upgrade header field only applies to switching protocols on top of the existing connection; it cannot be used to switch the underlying connection (transport) protocol, nor to switch the existing communication to a different connection. For those purposes, it is more appropriate to use a 3xx (Redirection) response (Section 6.4 of [RFC7231]).

Upgrade 헤더 필드는 기존 커넥션 위에 있는 스위칭 프로토콜에만 적용되며, 기반 커넥션 (transport) 프로토콜을 전환하는 데 사용할 수도 없고 기존 통신을 다른 커넥션로 전환하는 데 사용할 수도 없다. 그러한 목적을 위해서는 3xx (Redirection) 응답 ([RFC7231]의 Section 6.4)을 사용하는 것이 더 적절하다.

This specification only defines the protocol name "HTTP" for use by the family of Hypertext Transfer Protocols, as defined by the HTTP version rules of Section 2.6 and future updates to this specification. Additional tokens ought to be registered with IANA using the registration procedure defined in Section 8.6.

이 규격은 Section 2.6의 HTTP 버전 규칙과 이 규격에 대한 향후 업데이트에 의해 정의된 대로 하이퍼텍스트 전송 프로토콜 제품군에서 사용하기 위한 프로토콜 이름 “HTTP”만 정의한다. 추가 토큰은 Section 8.6에 정의된 등록 절차를 사용하여 IANA에 등록해야 한다.

7. ABNF List Extension: #rule

A #rule extension to the ABNF rules of [RFC5234] is used to improve readability in the definitions of some header field values.

[RFC5234]의 ABNF 규칙에 대한 #rule 확장은 일부 헤더 필드 값의 정의에서 가독성을 개선하기 위해 사용된다.

A construct "#" is defined, similar to "*", for defining comma-delimited lists of elements. The full form is “<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by a single comma (",") and optional whitespace (OWS).

구성 “#”은 쉼표로 구분된 요소 리스트를 정의하기 위해 “*”와 유사하게 정의된다. 전체 형태는 “[n]#[m] 요소”로 최소한 <n>과 기껏해야 <m> 요소를 나타내며, 각각 하나의 쉼표(“,”), 선택적 공백(OWS)으로 구분된다.

In any production that uses the list construct, a sender MUST NOT generate empty list elements. In other words, a sender MUST generate lists that satisfy the following syntax:

목록 구조를 사용하는 생산에서 발신인은 빈 목록 요소를 생성해서는 안 된다.(MUST NOT) 즉, 발신자는 다음 구문을 만족하는 목록을 생성해야 한다.(MUST)

1#element => element *(OWS "," OWS element)

and:

#element => [1#element]

and for n >= 1 and m > 1:

<n>#<m>element => element <n-1>*<m-1>(OWS "," OWS element)

For compatibility with legacy list rules, a recipient MUST parse and ignore a reasonable number of empty list elements: enough to handle common mistakes by senders that merge values, but not so much that they could be used as a denial-of-service mechanism. In other words, a recipient MUST accept lists that satisfy the following syntax:

레거시 목록 규칙과의 호환성을 위해 수신자는 값을 병합하는 발신자의 일반적인 실수를 처리할 수 있을 만큼 충분한 수의 빈 목록 요소를 구문 분석 및 무시해야 하지만,(MUST) 값을 denial-of-service 메커니즘으로 사용할 수 있는 정도는 아니다. 즉, 수신자는 다음 구문을 충족하는 목록을 반드시 수락해야 한다.(MUST)

#element => [("," / element) *(OWS "," [OWS element])]

1#element => *("," OWS) element *(OWS "," [OWS element])

Empty elements do not contribute to the count of elements present. For example, given these ABNF productions:

빈 요소는 존재하는 요소의 수에 기여하지 않는다. 예를 들어, 다음과 같은 ABNF 프로덕션의 경우:

```
example-list      = 1#example-list-elmt
example-list-elmt = token ; see Section 3.2.6
```

Then the following are valid values for example-list (not including the double quotes, which are present for delimitation only):

다음은 예제 목록에 유효한 값이다(구분에만 존재하는 큰 따옴표는 제외).

```
"foo,bar"
"foo ,bar,"
"foo , ,bar,charlie   "
```

In contrast, the following values would be invalid, since at least one non-empty element is required by the example-list production:

반면에, 예시 리스트 제작에 적어도 하나 이상의 비어 있지 않은 요소가 필요하기 때문에 다음 값은 유효하지 않을 것이다.

```
""
","
",   ,"
```

Appendix B shows the collected ABNF for recipients after the list constructs have been expanded.

Appendix B는 목록 구성이 확장된 후 수신자를 위한 편집된 ABNF를 보여준다.

8. IANA Considerations

8.1 Header Field Registrations

HTTP header fields are registered within the "Message Headers" registry maintained at <http://www.iana.org/assignments/message-headers/>.

HTTP 헤더 필드는 <http://www.iana.org/assignments/message-headers/>에서 유지되는 "Message Headers" 레지스트리 내에 등록된다.

This document defines the following HTTP header fields, so the "Permanent Message Header Field Names" registry has been updated accordingly (see [BCP90]).

이 문서는 다음과 같은 HTTP 헤더 필드를 정의하므로, 그에 따라 "Permanent Message Header Field Names" 레지스트리가 업데이트되었다 ([BCP90] 참조).

Header Field Name	Protocol	Status	Reference
Connection	http	standard	Section 6.1
Content-Length	http	standard	Section 3.3.2
Host	http	standard	Section 5.4
TE	http	standard	Section 4.3
Trailer	http	standard	Section 4.4
Transfer-Encoding	http	standard	Section 3.3.1
Upgrade	http	standard	Section 6.7
Via	http	standard	Section 5.7.1

Furthermore, the header field-name "Close" has been registered as "reserved", since using that name as an HTTP header field might conflict with the "close" connection option of the Connection header field (Section 6.1).

또한 HTTP 헤더 필드로 헤더 필드 이름 "Close"가 Connection 헤더 필드의 "close" 커넥션 옵션과 충돌할 수 있으므로 헤더 필드 이름 "Close"가 "reserved"로 등록되었다(Section 6.1).

Header Field Name	Protocol	Status	Reference
Close	http	reserved	Section 8.1

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

8.2 URI Scheme Registration

IANA maintains the registry of URI Schemes [BCP115] at <http://www.iana.org/assignments/uri-schemes/>.

IANA는 URI schemes [BCP115]의 레지스트리를 <http://www.iana.org/assignments/uri-schemes/>에 유지하고 있다.

This document defines the following URI schemes, so the “Permanent URI Schemes” registry has been updated accordingly.

이 문서는 다음과 같은 URI scheme을 정의하므로 “Permanent URI Schemes” 레지스트리는 그에 따라 업데이트되었다.

URI Scheme	Description	Reference
http	Hypertext Transfer Protocol	Section 2.7.1
https	Hypertext Transfer Protocol Secure	Section 2.7.2

8.3 Internet Media Type Restration

IANA maintains the registry of Internet media types [BCP13] at <http://www.iana.org/assignments/media-types>.

IANA는 Internet media types [BCP13]의 레지스트리를 <http://www.iana.org/assignments/media-types>에 유지한다.

This document serves as the specification for the Internet media types "message/http" and "application/http". The following has been registered with IANA.

이 문서는 인터넷 미디어 유형 “message/http” 및 “application/http”의 규격 역할을 한다. 다음은 IANA에 등록되었다.

8.3.1 Internet Media Type message/http

The message/http type can be used to enclose a single HTTP request or response message, provided that it obeys the MIME restrictions for all "message" types regarding line length and encodings.

message/http 유형은 회선 길이 및 인코딩과 관련된 모든 "message" 유형에 대한 MIME 제한을 준수하는 경우 단일 HTTP 요청 또는 응답 메시지를 동봉하는데 사용할 수 있다.

Type name: message

Subtype name: http

Required parameters: N/A

Optional parameters: version, msgtype

version: The HTTP-version number of the enclosed message (e.g., "1.1"). If not present, the version can be determined from the first line of the body.

msgtype: The message type -- "request" or "response". If not present, the type can be determined from the first line of the body.

Encoding considerations: only "7bit", "8bit", or "binary" are permitted

Security considerations: see Section 9

Interoperability considerations: N/A

Published specification: This specification (see Section 8.3.1).

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information:
See Authors' Addresses section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors' Addresses section.

Change controller: IESG

8.3.2 Internet Media Type application/http

The application/http type can be used to enclose a pipeline of one or more HTTP request or response messages (not intermixed).

application/http 유형은 하나 이상의 HTTP 요청 또는 응답 메시지(혼합되지 않음)의 파이프라인을 동봉하는 데 사용할 수 있다.

Type name: application

Subtype name: http

Required parameters: N/A

Optional parameters: version, msgtype

version: The HTTP-version number of the enclosed messages (e.g., "1.1"). If not present, the version can be determined from the first line of the body.

msgtype: The message type -- "request" or "response". If not present, the type can be determined from the first line of the body.

Encoding considerations: HTTP messages enclosed by this type are in "binary" format; use of an appropriate Content-Transfer-Encoding is required when transmitted via email.

Security considerations: see Section 9

Interoperability considerations: N/A

Published specification: This specification (see Section 8.3.2).

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information:
See Authors' Addresses section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors' Addresses section.

Change controller: IESG

8.4 Transfer Coding Registry

The "HTTP Transfer Coding Registry" defines the namespace for transfer coding names. It is maintained at <http://www.iana.org/assignments/http-parameters>.

"HTTP Transfer Coding Registry"는 코딩 이름을 전송하기 위한 네임스페이스를 정의한다. 《http://www.iana.org/assignments/http-parameters》에서 유지되고 있다.

8.4.1 Procedure

Registrations MUST include the following fields:

등록은 반드시 아래의 필드를 포함해야 한다.

- Name
- Description
- Pointer to specification text

Names of transfer codings MUST NOT overlap with names of content codings (Section 3.1.2.1 of [RFC7231]) unless the encoding transformation is identical, as is the case for the compression codings defined in Section 4.2.

Section 4.2에 정의된 압축 코딩의 경우와 같이 인코딩 변환이 동일하지 않는 한, transfer coding의 이름은 content coding의 이름과 중복되지 않아야 한다. (MUST NOT) ([RFC7231]의 Section 3.1.2.1).

Values to be added to this namespace require IETF Review (see Section 4.1 of [RFC5226]), and MUST conform to the purpose of transfer coding defined in this specification.

이 네임스페이스에 추가할 값은 IETF 검토 ([RFC5226]의 Section 4.1 참조)가 필요하며, 반드시 이 규격에서 정의한 transfer coding의 목적에 부합해야 한다. (MUST)

Use of program names for the identification of encoding formats is not desirable and is discouraged for future encodings.

인코딩 형식을 식별하기 위한 프로그램 이름을 사용하는 것은 바람직하지 않으며 향후 인코딩을 위해 권장되지 않는다.

8.4.2 Restration

The "HTTP Transfer Coding Registry" has been updated with the registrations below:

"HTTP Transfer Coding Registry"가 아래 등록과 함께 업데이트되었다.

Name	Description	Reference
chunked	Transfer in a series of chunks	Section 4.1
compress	UNIX "compress" data format [Welch]	Section 4.2.1
deflate	"deflate" compressed data ([RFC1951]) inside the "zlib" data format ([RFC1950])	Section 4.2.2
gzip	GZIP file format [RFC1952]	Section 4.2.3

| x-compress | Deprecated (alias for compress) | Section 4.2.1 |
| x-gzip | Deprecated (alias for gzip) | Section 4.2.3 |

8.5 Content Coding Registration

IANA maintains the "HTTP Content Coding Registry" at <http://www.iana.org/assignments/http-parameters>.

IANA는 "HTTP Content Coding Registry"를 <http://www.iana.org/assignments/http-parameters>에서 유지 관리한다.

The "HTTP Content Coding Registry" has been updated with the registrations below:

"HTTP Content Coding Registry"는 아래 등록과 함께 업데이트되었다.

Name	Description	Reference
compress	UNIX "compress" data format [Welch]	Section 4.2.1
deflate	"deflate" compressed data ([RFC1951]) inside the "zlib" data format ([RFC1950])	Section 4.2.2
gzip	GZIP file format [RFC1952]	Section 4.2.3
x-compress	Deprecated (alias for compress)	Section 4.2.1
x-gzip	Deprecated (alias for gzip)	Section 4.2.3

8.6 Upgrade Token Registry

The "Hypertext Transfer Protocol (HTTP) Upgrade Token Registry" defines the namespace for protocol-name tokens used to identify protocols in the Upgrade

header field. The registry is maintained at <http://www.iana.org/assignments/http-upgrade-tokens>.

"HTTP(Hypertext Transfer Protocol) Upgrade Token Registry"는 Upgrade 헤더 필드에서 프로토콜을 식별하는 데 사용되는 protocol-name 토큰의 네임스페이스를 정의한다. 레지스트리는 <http://www.iana.org/assignments/http-upgrade-tokens>에서 유지되고 있다.

8.6.1 Procedure

Each registered protocol name is associated with contact information and an optional set of specifications that details how the connection will be processed after it has been upgraded.

등록된 각 프로토콜 이름은 접근 정보 및 업그레이드 후 커넥션이 처리되는 방법을 자세히 설명하는 선택 명세의 집합과 연관된다.

Registrations happen on a "First Come First Served" basis (see Section 4.1 of [RFC5226]) and are subject to the following rules:

등록은 “First Come First Served” 기준으로 이루어지며 ([RFC5226]의 Section 4.1 참조) 다음과 같은 규칙에 따른다.

1. A protocol-name token, once registered, stays registered forever.

한번 등록된 protocol-name 토큰은 영원히 등록된다.

2. The registration MUST name a responsible party for the registration.

등록은 반드시 등록의 책임자를 기록해야 한다.(MUST)

3. The registration MUST name a point of contact.

등록은 반드시 연락처를 지정해야 한다.

4. The registration MAY name a set of specifications associated with that token. Such specifications need not be publicly available.

등록은 해당 토큰과 관련된 명세 집합의 이름을 지정할 수 있다. 그러한 규격은 공개적으로 이용할 필요가 없다.(MAY)

5. The registration SHOULD name a set of expected “protocol-version" tokens associated with that token at the time of registration.

등록 시 등록은 해당 토큰과 관련된 예상 “protocol-version”토큰의 이름을 지정해야 한다.(SHOULD)

6. The responsible party MAY change the registration at any time. The IANA will keep a record of all such changes, and make them available upon request.

책임 당사자는 언제든지 등기를 변경할 수 있다. IANA는 그러한 모든 변경사항을 기록하여 요청 시 이용할 수 있도록 할 것이다.(MAY)

7. The IESG MAY reassign responsibility for a protocol token. This will normally only be used in the case when a responsible party cannot be contacted.

IESG는 프로토콜 토큰에 대한 책임을 재할당할 수 있다. 이것은 일반적으로 책임 있는 당사자와 접촉할 수 없는 경우에만 사용된다.(MAY)

This registration procedure for HTTP Upgrade Tokens replaces that previously defined in Section 7.2 of [RFC2817].

HTTP Upgrade 토큰에 대한 이 등록 절차는 [RFC2817]의 Section 7.2에서 이전에 정의한 것을 대체한다.

8.6.2 Upgrade Token Registration

The "HTTP" entry in the upgrade token registry has been updated with the registration below:

Upgrade 토큰 레지스트리의 "HTTP" 항목이 아래 등록으로 업데이트되었다.

Value	Description	Expected Version Tokens	Reference
HTTP	Hypertext Transfer Protocol	any DIGIT.DIGIT (e.g, "2.0")	Section 2.6

The responsible party is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

담당자는 “IETF(iesg@ietf.org) - Internet Engineering Task Force”.이다.

9. Security Considerations

This section is meant to inform developers, information providers, and users of known security considerations relevant to HTTP message syntax, parsing, and routing. Security considerations about HTTP semantics and payloads are addressed in [RFC7231].

이 섹션은 개발자, 정보 제공자에게 정보를 제공하기 위한 것이다. HTTP 메시지와 관련된 알려진 보안 고려사항 사용자 구문, 구문 분석 및 라우팅. HTTP에 대한 보안 고려 사항 의미론 및 페이로드는 [RFC7231]에서 다루었다.

9.1 Establish Authority

HTTP relies on the notion of an authoritative response: a response that has been determined by (or at the direction of) the authority identified within the target URI to be the most appropriate response for that request given the state of the target resource at the time of response message origination. Providing a response from a non-authoritative source, such as a shared cache, is often useful to improve

performance and availability, but only to the extent that the source can be trusted or the distrusted response can be safely used.

HTTP는 권위 있는 응답의 개념에 의존한다. 즉, 대상 URI 내에서 식별된 권한에 의해 (또는 지시로) 결정된 응답은 응답 메시지 작성 시 대상 리소스의 상태를 감안할 때 해당 요청에 가장 적합한 응답이다. 공유 캐시와 같이 권한 없는 소스의 응답을 제공하는 것은 성능과 가용성을 향상시키는 데 유용하지만, 소스가 신뢰할 수 있거나 신뢰할 수 없는 응답을 안전하게 사용할 수 있는 범위까지만 유용하다.

Unfortunately, establishing authority can be difficult. For example, phishing is an attack on the user's perception of authority, where that perception can be misled by presenting similar branding in hypertext, possibly aided by userinfo obfuscating the authority component (see Section 2.7.1). User agents can reduce the impact of phishing attacks by enabling users to easily inspect a target URI prior to making an action, by prominently distinguishing (or rejecting) userinfo when present, and by not sending stored credentials and cookies when the referring document is from an unknown or untrusted source.

불행히도, 권한을 확립하는 것은 어려울 수 있다. 예를 들어, 피싱은 권한에 대한 사용자의 인식에 대한 공격이며, 권한 구성요소를 난독화하는 userinfo의 도움을 받을 수 있는 하이퍼텍스트에 유사한 브랜딩을 표시함으로써 그러한 인식이 오도될 수 있다(Section 2.7.1 참조). 사용자 에이전트는 사용자가 조치를 취하기 전에 대상 URI를 쉽게 검사할 수 있도록 하고, 사용자가 있을 때 userinfo를 눈에 띄게 구별(또는 거부)하며, 참조 문서가 알 수 없거나 신뢰할 수 없는 출처일 때 저장된 자격 증명과 쿠키를 보내지 않음으로써 피싱 공격의 영향을 줄일 수 있다.

When a registered name is used in the authority component, the “http" URI scheme (Section 2.7.1) relies on the user's local name resolution service to determine where it can find authoritative responses. This means that any attack on a user's network host table, cached names, or name resolution libraries becomes an avenue for attack on establishing authority. Likewise, the user's choice of server for Domain Name Service (DNS), and the hierarchy of servers from which it obtains resolution results, could impact the authenticity of address mappings; DNS Security Extensions (DNSSEC, [RFC4033]) are one way to improve authenticity.

등록된 이름이 권한 구성 요소에 사용될 때, "http" URI 구성표(Section 2.7.1)는 사용자의 로컬 이름 확인 서비스에 의존하여 권한 있는 응답을 찾을 수 있는 위치를 결정한다. 이는 사용자의 네트워크 호스트 테이블, 캐시된 이름 또는 이름 확인 라이브러리에 대한 모든 공격이 권한을 설정하기 위한 공격의 수단이 된다는 것을 의미한다. 마찬가지로, 사용자가 DNS(Domain Name Service)를 위해 서버를 선택하고, 확인 결과를 얻는 서버의 계층이 주

소 매핑의 진위 여부에 영향을 미칠 수 있다. DNS 보안 확장(DNSSEC, [RFC4033])은 신뢰성을 향상시키는 한 가지 방법이다.

Furthermore, after an IP address is obtained, establishing authority for an "http" URI is vulnerable to attacks on Internet Protocol routing.

또한 IP 주소를 얻은 후에는 "http" URI에 대한 권한을 설정하는 것이 인터넷 프로토콜 라우팅 공격에 취약하다.

The "https" scheme (Section 2.7.2) is intended to prevent (or at least reveal) many of these potential attacks on establishing authority, provided that the negotiated TLS connection is secured and the client properly verifies that the communicating server's identity matches the target URI's authority component (see [RFC2818]). Correctly implementing such verification can be difficult (see [Georgiev]).

"https" scheme(Section 2.7.2)은 협상된 TLS 커넥션이 보안되고 통신 서버의 ID가 대상 URI의 권한 구성 요소와 일치하는지 클라이언트가 적절하게 검증하는 경우, 권한 확립에 대한 이러한 잠재적 공격의 상당수를 방지하거나 최소한 공개하기 위한 것이다([RFC2818] 참조). 그러한 검증을 올바르게 시행하는 것은 어려울 수 있다([Georgiev] 참조).

9.2 Risks of Intermediaries

By their very nature, HTTP intermediaries are men-in-the-middle and, thus, represent an opportunity for man-in-the-middle attacks. Compromise of the systems on which the intermediaries run can result in serious security and privacy problems. Intermediaries might have access to security-related information, personal information about individual users and organizations, and proprietary information belonging to users and content providers. A compromised intermediary, or an intermediary implemented or configured without regard to security and privacy considerations, might be used in the commission of a wide range of potential attacks.

그것들의 성격상, HTTP 중개자는 men-in-the-middle 공격의 기회를 나타낸다. 중개자가 운영하는 시스템을 손상시키면 심각한 보안과 사생활 문제가 발생할 수 있다. 중개자는 보안 관련 정보, 개별 사용자와 조직에 대한 개인 정보, 그리고 사용자와 콘텐츠 제공자에 속하는 독점 정보에 접근할 수 있다. 보안 및 프라이버시 고려사항 없이 타협된 중개자 또는 구현되거나 구성된 중개자는 광범위한 잠재적 공격의 커미션에 사용될 수 있다.

Intermediaries that contain a shared cache are especially vulnerable to cache poisoning attacks, as described in Section 8 of [RFC7234].

공유 캐시를 포함하는 중개자는 [RFC7234]의 Section 8에 설명된 바와 같이 캐시 중독 공격에 특히 취약하다.

Implementers need to consider the privacy and security implications of their design and coding decisions, and of the configuration options they provide to operators (especially the default configuration).

구현자는 설계 및 코딩 결정과 운영자에게 제공하는 구성 옵션(특히 기본 구성)의 프라이버시 및 보안 영향을 고려해야 한다.

Users need to be aware that intermediaries are no more trustworthy than the people who run them; HTTP itself cannot solve this problem.

사용자들은 중개자들이 중개자를 운영하는 사람들보다 더 신뢰할 수 없다는 것을 알아야 한다; HTTP 자체는 이 문제를 해결할 수 없다.

9.3 Attacks via Protocol Element Length

Because HTTP uses mostly textual, character-delimited fields, parsers are often vulnerable to attacks based on sending very long (or very slow) streams of data, particularly where an implementation is expecting a protocol element with no predefined length.

HTTP는 대부분 텍스트, 문자 구분 필드를 사용하기 때문에, 파서들은 종종 매우 긴(또는 매우 느린) 데이터 스트림을 전송하는 것에 기반한 공격에 취약하며, 특히 구현이 미리 정의된 길이 없이 프로토콜 요소를 예상하는 경우.

To promote interoperability, specific recommendations are made for minimum size limits on request-line (Section 3.1.1) and header fields (Section 3.2). These are minimum recommendations, chosen to be supportable even by implementations with limited resources; it is expected that most implementations will choose substantially higher limits.

상호운용성을 촉진하기 위해 request-line(Section 3.1.1) 및 헤더 필드(Section 3.2)의 최소 크기 제한에 대한 구체적인 권고사항이 제시된다. 이는 리소스가 제한된 구현에서도 지원할 수 있도록 선택한 최소 권장 사항이다. 대부분의 구현이 상당히 높은 제한을 선택할 것으로 예상된다.

A server can reject a message that has a request-target that is too long (Section 6.5.12 of [RFC7231]) or a request payload that is too large (Section 6.5.11 of [RFC7231]). Additional status codes related to capacity limits have been defined by extensions to HTTP [RFC6585].

서버는 요청 대상이 너무 긴 메시지([RFC7231]의 Section 6.5.12) 또는 너무 큰 요청 페이로드([RFC7231]의 Section 6.5.11)를 거부할 수 있다. 용량 제한과 관련된 추가 상태 코드는 HTTP에 대한 확장에 의해 정의되었다[RFC6585].

Recipients ought to carefully limit the extent to which they process other protocol elements, including (but not limited to) request methods, response status phrases, header field-names, numeric values, and body chunks. Failure to limit such processing can result in buffer overflows, arithmetic overflows, or increased vulnerability to denial-of-service attacks.

수신자는 요청 메서드, 응답 상태 구문, 헤더 필드 이름, 숫자 값 및 본문 청크를 포함한 기타 프로토콜 요소를 처리하는 범위를 신중하게 제한해야 한다. 이러한 처리를 제한하지 않을 경우 버퍼 오버플로, 산술 오버플로 또는 denial-of-service 공격에 대한 취약성이 증가할 수 있다.

9.4 Response Splitting

Response splitting (a.k.a, CRLF injection) is a common technique, used in various attacks on Web usage, that exploits the line-based nature of HTTP message framing and the ordered association of requests to responses on persistent connections [Klein]. This technique can be particularly damaging when the requests pass through a shared cache.

응답 분할(a.k.a, CRLF 주입)은 HTTP 메시지 프레임의 line-based 특성과 영속적 커넥션에 대한 응답에 대한 요청의 순서 커넥션을 이용하는 웹 사용에 대한 다양한 공격에 사용되는 일반적인 기술이다[Klein]. 이 기법은 요청이 공유 캐시를 통과할 때 특히 손상될 수 있다.

Response splitting exploits a vulnerability in servers (usually within an application server) where an attacker can send encoded data within some parameter of the request that is later decoded and echoed within any of the response header fields of the response. If the decoded data is crafted to look like the response has ended and a subsequent response has begun, the response has been split and the content within the apparent second response is controlled by the attacker. The attacker can then make any other request on the same persistent connection and trick the recipients (including intermediaries) into believing that the second half of the split is an authoritative answer to the second request.

응답 분할은 공격자가 나중에 디코딩되고 응답의 응답 헤더 필드 중 하나에 반향되는 요청의 일부 매개 변수 내에서 인코딩된 데이터를 전송할 수 있는 서버(일반적으로 애플리케이션 서버 내)의 취약성을 이용한다. 디코딩된 데이터가 응답이 종료되고 후속 응답이 시작된 것처럼 조작된 경우, 응답이 분할되고 명백한 두 번째 응답 내의 내용이 공격자에 의해 제어된다. 그러면 공격자는 동일한 영속적 커넥션에 대해 다른 어떤 요청을 할 수 있으며, 수신자(중간자 포함)를 속여서 분할의 후반부가 두 번째 요청에 대한 권위 있는 답변이라고 믿게 할 수 있다.

For example, a parameter within the request-target might be read by an application server and reused within a redirect, resulting in the same parameter being echoed in the Location header field of the response. If the parameter is decoded by the application and not properly encoded when placed in the response field, the attacker can send encoded CRLF octets and other content that will make the application's single response look like two or more responses.

예를 들어, request-target 내의 매개변수는 애플리케이션 서버가 읽고 리디렉션 내에서 재사용할 수 있으며, 그 결과 동일한 매개변수가 응답의 Location 헤더 필드에 반향될 수 있다. 매개변수가 애플리케이션에 의해 디코딩되고 응답 필드에 배치되었을 때 제대로 인코딩되지 않으면 공격자는 인코딩된 CRLF octet 및 애플리케이션의 단일 응답을 두 개 이상의 응답으로 보이게 하는 기타 콘텐츠를 전송할 수 있다.

A common defense against response splitting is to filter requests for data that looks like encoded CR and LF (e.g., "%0D" and “%0A"). However, that assumes the application server is only performing URI decoding, rather than more obscure data transformations like charset transcoding, XML entity translation, base64 decoding, sprintf reformatting, etc. A more effective mitigation is to prevent anything other than the server's core protocol libraries from sending a CR or LF within the header section, which means restricting the output of header fields to APIs that filter for bad octets and not allowing application servers to write directly to the protocol stream.

응답 분할에 대한 일반적인 방어는 인코딩된 CR 및 LF(예: "%0D" 및 "%0A")처럼 보이는 데이터에 대한 요청을 필터링하는 것이다.그러나 애플리케이션 서버가 Charset 트랜스코딩, XML 엔티티 변환, base64 디코딩, sprintf 재포맷 등과 같은 더 모호한 데이터 변환이 아니라 URI 디코딩만 수행한다고 가정한다. 좀 더 효과적인 완화는 서버의 핵심 프로토콜 라이브러리가 헤더 부문 내에서 CR 또는 LF를 보내는 것을 막는 것인데, 이는 헤더 필드의 출력을 불량 octets을 필터링하는 API로 제한하고 애플리케이션 서버가 프로토콜 스트림에 직접 쓰도록 허용하지 않는다는 것을 의미한다.

9.5 Request Smuggling

Request smuggling ([Linhart]) is a technique that exploits differences in protocol parsing among various recipients to hide additional requests (which might otherwise be blocked or disabled by policy) within an apparently harmless request. Like response splitting, request smuggling can lead to a variety of attacks on HTTP usage.

요청 smuggling([Linhart])은 다양한 수신자 간의 프로토콜 구문 분석의 차이를 이용하여 겉으로 보기에는 무해한 요청 내에서 추가 요청(다른 경우 정책에 의해 차단되거나 비활성화될 수 있음)을 숨기는 기술이다. 대응 분할과 같이, 요청 smuggling은 HTTP 사용에 대한 다양한 공격으로 이어질 수 있다.

This specification has introduced new requirements on request parsing, particularly with regard to message framing in Section 3.3.3, to reduce the effectiveness of request smuggling.

이 규격은 요청 smuggling의 효과를 줄이기 위해 특히 Section 3.3.3의 메시지 프레임과 관련하여 요청 파싱에 대한 새로운 요구사항을 도입하였다.

9.6 Message Integrity

HTTP does not define a specific mechanism for ensuring message integrity, instead relying on the error-detection ability of underlying transport protocols and the use of length or chunk-delimited framing to detect completeness. Additional integrity mechanisms, such as hash functions or digital signatures applied to the content, can be selectively added to messages via extensible metadata header fields. Historically, the lack of a single integrity mechanism has been justified by the informal nature of most HTTP communication. However, the prevalence of HTTP as an information access mechanism has resulted in its increasing use within environments where verification of message integrity is crucial.

HTTP는 메시지 무결성을 보장하기 위한 특정 메커니즘을 정의하지 않고, 대신에 기본 전송 프로토콜의 오류 감지 능력과 완전성을 탐지하기 위한 길이 또는 청크로 구분된 프레임의 사용에 의존한다. 콘텐츠에 적용된 해시함수나 디지털서명과 같은 추가적인 무결성 메커니즘은 확장 가능한 메타데이터 헤더 필드를 통해 메시지에 선택적으로 추가할 수 있다. 역사적으로, 단일 무결성 메커니즘의 부재는 대부분의 HTTP 통신의 비공식적 성격에 의해 정당화되었다. 그러나, 정보 접근 메커니즘으로서의 HTTP의 보급은 메시지 무결성의 검증이 중요한 환경 내에서 HTTP 사용 증가를 가져왔다.

User agents are encouraged to implement configurable means for detecting and reporting failures of message integrity such that those means can be enabled within environments for which integrity is necessary. For example, a browser being used to view medical history or drug interaction information needs to indicate to the user when such information is detected by the protocol to be incomplete, expired, or corrupted during transfer. Such mechanisms might be selectively enabled via user agent extensions or the presence of message integrity metadata in a response. At a minimum, user agents ought to provide some indication that allows a user to distinguish between a complete and incomplete response message (Section 3.4) when such verification is desired.

사용자 에이전트는 무결성이 필요한 환경 내에서 해당 수단이 활성화될 수 있도록 메시지 무결성의 실패를 감지하고 보고하기 위한 구성 가능한 수단을 구현하도록 권장된다. 예를 들어, 의료 기록이나 약물 상호작용 정보를 보기 위해 사용 중인 브라우저는 그러한 정보가 프로토콜에 의해 탐지될 때 사용자에게 전달 중 불완전하거나 만료되거나 손상될 때 표시해야 한다. 이러한 메커니즘은 사용자 에이전트 확장 또는 응답에 메시지 무결성 메타데이터의 존재를 통해 선택적으로 활성화될 수 있다. 최소한 사용자 에이전트는 사용자가 그러한 확인을 원할 때 완전한 응답 메시지와 불완전한 응답 메시지(Section 3.4)를 구별할 수 있는 표시를 제공해야 한다.

9.7 Message Confidentiality

HTTP relies on underlying transport protocols to provide message confidentiality when that is desired. HTTP has been specifically designed to be independent of the transport protocol, such that it can be used over many different forms of encrypted connection, with the selection of such transports being identified by the choice of URI scheme or within user agent configuration.

HTTP는 메시지 기밀성이 필요할 때 메시지 기밀성을 제공하기 위해 기본적인 전송 프로토콜에 의존한다. HTTP는 여러 가지 다른 형태의 암호화된 커넥션에서 사용될 수 있도록 전송 프로토콜과 독립적으로 설계되었으며, 그러한 전송의 선택은 URI 구성의 선택 또는 사용자 에이전트 구성 내에서 식별된다.

The "https" scheme can be used to identify resources that require a confidential connection, as described in Section 2.7.2.

"https" 계획은 Section 2.7.2에 기술된 바와 같이 기밀 커넥션이 필요한 리소스를 식별하는데 사용될 수 있다.

9.8 Privacy of Server Log Information

A server is in the position to save personal data about a user's requests over time, which might identify their reading patterns or subjects of interest. In particular, log information gathered at an intermediary often contains a history of user agent interaction, across a multitude of sites, that can be traced to individual users.

서버는 시간이 지남에 따라 사용자의 요청에 대한 개인 데이터를 저장해야 하는 위치에 있으며, 이는 사용자의 읽기 패턴이나 관심 대상을 식별할 수 있다. 특히, 중개자에게 수집된 로그 정보는 종종 개별 사용자로 추적할 수 있는 다수의 사이트에 걸쳐 사용자 에이전트 상호 작용의 이력을 포함한다.

HTTP log information is confidential in nature; its handling is often constrained by laws and regulations. Log information needs to be securely stored and appropriate guidelines followed for its analysis. Anonymization of personal information within individual entries helps, but it is generally not sufficient to prevent real log traces from being re-identified based on correlation with other access characteristics. As

such, access traces that are keyed to a specific client are unsafe to publish even if the key is pseudonymous.

HTTP 로그 정보는 본질적으로 기밀이며, 취급은 종종 법과 규정에 의해 제한된다. 로그 정보를 안전하게 저장하고 분석을 위해 적절한 지침을 따라야 한다. 개별 항목 내에서의 개인정보 익명화는 도움이 되지만, 일반적으로 다른 접속 특성과의 상관관계에 근거해 실제 로그 흔적이 재식별되는 것을 막기에는 역부족이다. 이와 같이, 특정 클라이언트에 입력된 접근 흔적은 키가 가명인 경우에도 게시하기에 안전하지 않다.

To minimize the risk of theft or accidental publication, log information ought to be purged of personally identifiable information, including user identifiers, IP addresses, and user-provided query parameters, as soon as that information is no longer necessary to support operational needs for security, auditing, or fraud control.

도난 또는 우발적 발행의 위험을 최소화하기 위해 로그 정보는 보안, 감사 또는 부정 행위 통제에 대한 운영상의 요구를 지원하는 데 더 이상 필요하지 않은 즉시 사용자 식별자, IP 주소 및 사용자가 제공한 질의 매개변수를 포함하여 개인 식별 가능한 정보를 제거해야 한다.

10. Acknowledgments

This edition of HTTP/1.1 builds on the many contributions that went into RFC 1945, RFC 2068, RFC 2145, and RFC 2616, including substantial contributions made by the previous authors, editors, and Working Group Chairs: Tim Berners-Lee, Ari Luotonen, Roy T. Fielding, Henrik Frystyk Nielsen, Jim Gettys, Jeffrey C. Mogul, Larry Masinter, and Paul J. Leach. Mark Nottingham oversaw this effort as Working Group Chair.

이 HTTP/1.1 버전은 이전 작가, 편집자 및 작업 그룹 의자의 상당한 기여를 포함하여 RFC 1945, RFC 2068, RFC 2145 및 RFC 2616에 들어간 많은 기여를 기반으로 한다. Tim Berners-Lee, Ari Luotonen, Roy T. Fielding, Henrik Frystyk Nielsen, Jim Gettys, Jeffrey C. Mogul, Larry Masinter, Paul J. Leach. Mark Nottingham은 이 노력을 작업 그룹 의장으로서 감독했다.

Since 1999, the following contributors have helped improve the HTTP specification by reporting bugs, asking smart questions, drafting or reviewing text, and evaluating open issues:

1999년 이후 다음과 같은 기여자들은 버그 보고, 스마트 질문, 초안 작성 또는 텍스트 검토, 공개 이슈 평가 등을 통해 HTTP 사양을 개선하는 데 도움을 주었다.

Adam Barth, Adam Roach, Addison Phillips, Adrian Chadd, Adrian Cole, Adrien W. de Croy, Alan Ford, Alan Ruttenberg, Albert Lunde, Alek Storm, Alex Rousskov, Alexandre Morgaut, Alexey Melnikov, Alisha Smith, Amichai Rothman, Amit Klein, Amos Jeffries, Andreas Maier, Andreas Petersson, Andrei Popov, Anil Sharma, Anne van Kesteren, Anthony Bryan, Asbjorn Ulsberg, Ashok Kumar, Balachander Krishnamurthy, Barry Leiba, Ben Laurie, Benjamin Carlyle, Benjamin Niven-Jenkins, Benoit Claise, Bil Corry, Bill Burke, Bjoern Hoehrmann, Bob Scheifler, Boris Zbarsky, Brett Slatkin, Brian Kell, Brian McBarron, Brian Pane, Brian Raymor, Brian Smith, Bruce Perens, Bryce Nesbitt, Cameron Heavon-Jones, Carl Kugler, Carsten Bormann, Charles Fry, Chris Burdess, Chris Newman, Christian Huitema, Cyrus Daboo, Dale Robert Anderson, Dan Wing, Dan Winship, Daniel Stenberg, Darrel Miller, Dave Cridland, Dave Crocker, Dave Kristol, Dave Thaler, David Booth, David Singer, David W. Morris, Diwakar Shetty, Dmitry Kurochkin, Drummond Reed, Duane Wessels, Edward Lee, Eitan Adler, Eliot Lear, Emile Stephan, Eran Hammer-Lahav, Eric D. Williams, Eric J. Bowman, Eric Lawrence, Eric Rescorla, Erik Aronesty, EungJun Yi, Evan Prodromou, Felix Geisendoerfer, Florian Weimer, Frank Ellermann, Fred Akalin, Fred Bohle, Frederic Kayser, Gabor Molnar, Gabriel Montenegro, Geoffrey Sneddon, Gervase Markham, Gili Tzabari, Grahame Grieve, Greg Slepak, Greg Wilkins, Grzegorz Calkowski, Harald Tveit Alvestrand, Harry Halpin, Helge Hess, Henrik Nordstrom, Henry S. Thompson, Henry Story, Herbert van de Sompel, Herve Ruellan, Howard Melman, Hugo Haas, Ian Fette, Ian Hickson, Ido Safruti, Ilari Liusvaara, Ilya Grigorik, Ingo Struck, J. Ross Nicoll, James Cloos, James H. Manger, James Lacey, James M. Snell, Jamie Lokier, Jan Algermissen, Jari Arkko, Jeff Hodges (who came up with the term 'effective Request-URI'), Jeff Pinner, Jeff Walden, Jim Luther, Jitu Padhye, Joe D. Williams, Joe Gregorio, Joe Orton, Joel Jaeggli, John C. Klensin, John C. Mallery, John Cowan, John Kemp, John Panzer, John Schneider, John Stracke, John Sullivan, Jonas Sicking, Jonathan A. Rees, Jonathan Billington, Jonathan Moore, Jonathan Silvera, Jordi Ros, Joris Dobbelsteen, Josh Cohen, Julien Pierre, Jungshik Shin, Justin Chapweske, Justin Erenkrantz, Justin James, Kalvinder Singh, Karl Dubost, Kathleen Moriarty, Keith Hoffman, Keith Moore, Ken Murchison, Koen Holtman, Konstantin Voronkov, Kris Zyp, Leif Hedstrom, Lionel Morand, Lisa Dusseault, Maciej Stachowiak, Manu Sporny, Marc Schneider, Marc Slemko, Mark Baker, Mark Pauley, Mark Watson, Markus Isomaki, Markus Lanthaler, Martin J. Duerst, Martin Musatov, Martin Nilsson, Martin Thomson, Matt Lynch, Matthew Cox, Matthew Kerwin, Max Clark, Menachem Dodge, Meral Shirazipour, Michael Burrows, Michael Hausenblas, Michael Scharf, Michael Sweet, Michael Tuexen, Michael Welzl, Mike Amundsen, Mike Belshe, Mike Bishop, Mike Kelly, Mike Schinkel, Miles Sabin, Murray S. Kucherawy, Mykyta Yevstifeyev, Nathan Rixham,

Nicholas Shanks, Nico Williams, Nicolas Alvarez, Nicolas Mailhot, Noah Slater, Osama Mazahir, Pablo Castro, Pat Hayes, Patrick R. McManus, Paul E. Jones, Paul Hoffman, Paul Marquess, Pete Resnick, Peter Lepeska, Peter Occil, Peter Saint-Andre, Peter Watkins, Phil Archer, Phil Hunt, Philippe Mougin, Phillip Hallam-Baker, Piotr Dobrogost, Poul-Henning Kamp, Preethi Natarajan, Rajeev Bector, Ray Polk, Reto Bachmann-Gmuer, Richard Barnes, Richard Cyganiak, Rob Trace, Robby Simpson, Robert Brewer, Robert Collins, Robert Mattson, Robert O'Callahan, Robert Olofsson, Robert Sayre, Robert Siemer, Robert de Wilde, Roberto Javier Godoy, Roberto Peon, Roland Zink, Ronny Widjaja, Ryan Hamilton, S. Mike Dierken, Salvatore Loreto, Sam Johnston, Sam Pullara, Sam Ruby, Saurabh Kulkarni, Scott Lawrence (who maintained the original issues list), Sean B. Palmer, Sean Turner, Sebastien Barnoud, Shane McCarron, Shigeki Ohtsu, Simon Yarde, Stefan Eissing, Stefan Tilkov, Stefanos Harhalakis, Stephane Bortzmeyer, Stephen Farrell, Stephen Kent, Stephen Ludin, Stuart Williams, Subbu Allamaraju, Subramanian Moonesamy, Susan Hares, Sylvain Hellegouarch, Tapan Divekar, Tatsuhiro Tsujikawa, Tatsuya Hayashi, Ted Hardie, Ted Lemon, Thomas Broyer, Thomas Fossati, Thomas Maslen, Thomas Nadeau, Thomas Nordin, Thomas Roessler, Tim Bray, Tim Morgan, Tim Olsen, Tom Zhou, Travis Snoozy, Tyler Close, Vincent Murphy, Wenbo Zhu, Werner Baumann, Wilbur Streett, Wilfredo Sanchez Vega, William A. Rowe Jr., William Chan, Willy Tarreau, Xiaoshu Wang, Yaron Goland, Yngve Nysaeter Pettersen, Yoav Nir, Yogesh Bang, Yuchung Cheng, Yutaka Oiwa, Yves Lafon (long-time member of the editor team), Zed A. Shaw, and Zhong Yu.

See Section 16 of [RFC2616] for additional acknowledgements from prior revisions.

이전 개정판으로부터의 추가 인가는 [RFC2616]의 Section 16을 참조한다.

11. Reference

11.1 Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1950] Deutsch, L. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, May 1996.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[RFC1952] Deutsch, P., Gailly, J-L., Adler, M., Deutsch, L., and

G. Randers-Pehrson, "GZIP file format specification version 4.3", RFC 1952, May 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

[RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, June 2014.

[RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, June 2014.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, June 2014.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, June 2014.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[Welch] Welch, T., "A Technique for High-Performance Data Compression", IEEE Computer 17(6), June 1984.

11.2 Informative References

[BCP115] Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI Schemes", BCP 115, RFC 4395, February 2006.

[BCP13] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.

[BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

[Georgiev] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software", In Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12), pp. 38-49, October 2012, <http://doi.acm.org/10.1145/2382196.2382204>.

[ISO-8859-1] International Organization for Standardization, "Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1", ISO/IEC 8859-1:1998, 1998.

[Klein] Klein, A., "Divide and Conquer - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics", March 2004, <http://packetstormsecurity.com/papers/general/whitepaper_httpresponse.pdf>.

[Kri2001] Kristol, D., "HTTP Cookies: Standards, Privacy, and Politics", ACM Transactions on Internet Technology 1(2), November 2001, <http://arxiv.org/abs/cs.SE/0105018>.

[Linhart] Linhart, C., Klein, A., Heled, R., and S. Orrin, "HTTP Request Smuggling", June 2005, <http://www.watchfire.com/news/whitepapers.aspx>.

[RFC1919] Chatel, M., "Classical versus Transparent IP Proxies",

RFC 1919, March 1996.

[RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC2047] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, November 1996.

[RFC2068] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997.

[RFC2145] Mogul, J., Fielding, R., Gettys, J., and H. Nielsen, "Use and Interpretation of HTTP Version Numbers", RFC 2145, May 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2817] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3040] Cooper, I., Melve, I., and G. Tomlinson, "Internet Web Replication and Caching Taxonomy", RFC 3040, January 2001.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, March 2005.

[RFC4559] Jaganathan, K., Zhu, L., and J. Brezak, "SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows", RFC 4559, June 2006.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5322] Resnick, P., "Internet Message Format", RFC 5322, October 2008.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, April 2012.

Appendix A. HTTP Version History

HTTP has been in use since 1990. The first version, later referred to as HTTP/0.9, was a simple protocol for hypertext data transfer across the Internet, using only a single request method (GET) and no metadata. HTTP/1.0, as defined by [RFC1945], added a range of request methods and MIME-like messaging, allowing for metadata to be transferred and modifiers placed on the request/response semantics. However, HTTP/1.0 did not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, or name-based virtual hosts. The proliferation of incompletely implemented applications calling themselves “HTTP/1.0" further necessitated a protocol version change in order for two communicating applications to determine each other's true capabilities.

HTTP는 1990년부터 사용되고 있다. 나중에 HTTP/0.9 라고 불리는 첫 번째 버전은 단일 요청 메서드(GET)만 사용하고 메타데이터는 사용하지 않는 인터넷을 통한 하이퍼텍스트 데이터 전송을 위한 간단한 프로토콜이었다. [RFC1945]에서 정의한 대로 HTTP/1.0은 다양한 요청 메서드와 MIME 유사 메시지를 추가하여 메타데이터를 전송하고 요청/응답 의미론에 수정자를 배치할 수 있도록 했다. 그러나 HTTP/1.0은 계층 프락시, 캐싱, 영속적 커넥션의 필요성 또는 이름 기반 가상 호스트의 영향을 충분히 고려하지 않았다. 불완전하게 구현된 애플리케

이션이 급증함에 따라 "HTTP/1.0"이라는 명칭으로 통신하는 두 애플리케이션이 서로의 진정한 능력을 결정하기 위해서는 프로토콜 버전 변경이 더욱 필요하게 되었다.

HTTP/1.1 remains compatible with HTTP/1.0 by including more stringent requirements that enable reliable implementations, adding only those features that can either be safely ignored by an HTTP/1.0 recipient or only be sent when communicating with a party advertising conformance with HTTP/1.1.

HTTP/1.1은 신뢰할 수 있는 구현을 가능하게 하는 보다 엄격한 요구사항을 포함함으로써 HTTP/1.0과 호환성을 유지하며, HTTP/1.0 수신자가 안전하게 무시하거나 HTTP/1.1과 통신할 때에만 전송할 수 있는 기능만 추가한다.

HTTP/1.1 has been designed to make supporting previous versions easy. A general-purpose HTTP/1.1 server ought to be able to understand any valid request in the format of HTTP/1.0, responding appropriately with an HTTP/1.1 message that only uses features understood (or safely ignored) by HTTP/1.0 clients. Likewise, an HTTP/1.1 client can be expected to understand any valid HTTP/1.0 response.

HTTP/1.1은 이전 버전을 쉽게 지원하도록 설계되었다. 범용 HTTP/1.1 서버는 HTTP/1.0 클라이언트가 이해하거나 안전하게 무시한 기능만 사용하는 HTTP/1.1 메시지로 적절하게 응답하면서 HTTP/1.0 형식의 유효한 요청을 이해할 수 있어야 한다. 마찬가지로 HTTP/1.1 클라이언트는 유효한 HTTP/1.0 응답을 이해할 수 있다.

Since HTTP/0.9 did not support header fields in a request, there is no mechanism for it to support name-based virtual hosts (selection of resource by inspection of the Host header field). Any server that implements name-based virtual hosts ought to disable support for HTTP/0.9. Most requests that appear to be HTTP/0.9 are, in fact, badly constructed HTTP/1.x requests caused by a client failing to properly encode the request-target.

HTTP/0.9는 요청에서 헤더 필드를 지원하지 않았기 때문에 이름 기반 가상 호스트(Host 헤더 필드의 검사에 의한 리소스 선택)를 지원하는 메커니즘이 없다. 이름 기반 가상 호스트를 구현하는 모든 서버는 HTTP/0.9에 대한 지원을 비활성화해야 한다. HTTP/0.9로 보이는 대부분의 요청은 실제로 클라이언트가 요청 대상을 제대로 인코딩하지 못해 발생하는 잘못 구성된 HTTP/1.x 요청이다.

A.1 Changes HTTP/1.0

This section summarizes major differences between versions HTTP/1.0 and HTTP/1.1.

이 섹션은 버전 HTTP/1.0과 HTTP/1.1 사이의 주요 차이점을 요약한다.

A.1.1 Multihomed Web Servers

The requirements that clients and servers support the Host header field (Section 5.4), report an error if it is missing from an HTTP/1.1 request, and accept absolute URIs (Section 5.3) are among the most important changes defined by HTTP/1.1.

클라이언트와 서버가 Host 헤더 필드를 지원하고(Section 5.4), HTTP/1.1 요청에 누락된 경우 오류를 보고하고, 절대 URI를 수락(Section 5.3)하는 요건은 HTTP/1.1에서 정의한 가장 중요한 변경 사항 중 하나이다.

Older HTTP/1.0 clients assumed a one-to-one relationship of IP addresses and servers; there was no other established mechanism for distinguishing the intended server of a request than the IP address to which that request was directed. The Host header field was introduced during the development of HTTP/1.1 and, though it was quickly implemented by most HTTP/1.0 browsers, additional requirements were placed on all HTTP/1.1 requests in order to ensure complete adoption. At the time of this writing, most HTTP-based services are dependent upon the Host header field for targeting requests.

이전 HTTP/1.0 클라이언트는 IP 주소와 서버의 one-to-one 관계를 가정했다. 요청이 지시된 IP 주소 외에 요청의 의도된 서버를 구별하기 위한 다른 확립된 메커니즘은 없었다. Host 헤더 필드는 HTTP/1.1 개발 중에 도입되었으며, 대부분의 HTTP/1.0 브라우저에 의해 신속하게 구현되었지만, 완전한 채택을 보장하기 위해 모든 HTTP/1.1 요청에 추가 요건이 부여되었다. 이 작성 당시에 대부분의 HTTP 기반 서비스는 대상 요청을 위해 Host 헤더 필드에 의존한다.

A.1.2 Keep-Alive Connections

In HTTP/1.0, each connection is established by the client prior to the request and closed by the server after sending the response. However, some implementations implement the explicitly negotiated ("Keep-Alive") version of persistent connections described in Section 19.7.1 of [RFC2068].

HTTP/1.0에서 각 커넥션은 요청 전에 클라이언트에 의해 설정되고 응답 전송 후 서버에 의해 종료된다. 그러나 일부 구현에서는 명시적으로 협상된 [RFC2068] Section 19.7.1에 설명한 영속적 커넥션(“Keep-Alive”) 버전을 구현한다.

Some clients and servers might wish to be compatible with these previous approaches to persistent connections, by explicitly negotiating for them with a "Connection: keep-alive" request header field. However, some experimental implementations of HTTP/1.0 persistent connections are faulty; for example, if an HTTP/1.0 proxy server doesn't understand Connection, it will erroneously forward that header field to the next inbound server, which would result in a hung connection.

일부 클라이언트와 서버는 "Connection: keep-alive" 요청 헤더 필드와 명시적으로 협상함으로써 영속적 커넥션에 대한 이러한 이전 접근방식과 호환되기를 원할 수 있다. 그러나 HTTP/1.0 영속적 커넥션의 일부 실험적 구현은 결함이 있다. 예를 들어 HTTP/1.0 프락시 서버가 Connection을 이해하지 못하면 헤더 필드를 다음 인바운드 서버로 잘못 전달하여 커넥션이 끊어질 수 있다.

One attempted solution was the introduction of a Proxy-Connection header field, targeted specifically at proxies. In practice, this was also unworkable, because proxies are often deployed in multiple layers, bringing about the same problem discussed above.

시도된 해결책 중 하나는 Proxy-Connection 헤더 필드를 도입하는 것으로, 특히 프락시를 대상으로 한다. 실제로 프락시는 여러 계층에 배치되어 위에서 논의한 것과 동일한 문제를 야기하는 경우가 많기 때문에 이 또한 실행 불가능했다.

As a result, clients are encouraged not to send the Proxy-Connection header field in any requests.

결과적으로, 클라이언트는 어떠한 요청에도 Proxy-Connection 헤더 필드를 보내지 않도록 권장된다.

Clients are also encouraged to consider the use of Connection: keep-alive in requests carefully; while they can enable persistent connections with HTTP/1.0 servers, clients using them will need to monitor the connection for "hung" requests (which indicate that the client ought stop sending the header field), and this mechanism ought not be used by clients at all when a proxy is being used.

클라이언트는 또한 요청에서 keep-alive 커넥션을 주의깊게 사용하도록 권장된다: 그들은 HTTP/1.0 서버와의 영속적 커넥션을 가능하게 할 수 있지만, 클라이언트가 “hung” 요청의 커넥션을 감시해야 할 필요가 있을 것이다.(헤더 필드 전송을 중단해야 한다는 것을 나타내는) 그리고 프락시를 사용할 때 이 메커니즘은 클라이언트에 의해 사용되어서는 안 된다.

A.1.3 Introduction of Transfer-Encoding

HTTP/1.1 introduces the Transfer-Encoding header field (Section 3.3.1). Transfer codings need to be decoded prior to forwarding an HTTP message over a MIME-compliant protocol.

HTTP/1.1에는 Transfer-Encoding 헤더 필드가 도입된다(Section 3.3.1). 전송 코딩은 MIME 호환 프로토콜을 통해 HTTP 메시지를 전달하기 전에 해독되어야 한다.

A.2 Changes from RFC 2616

HTTP's approach to error handling has been explained. (Section 2.5)

오류 처리에 대한 HTTP의 접근방식이 설명되었다. (Section 2.5)

The HTTP-version ABNF production has been clarified to be case-sensitive. Additionally, version numbers have been restricted to single digits, due to the fact that implementations are known to handle multi-digit version numbers incorrectly. (Section 2.6)

HTTP-version ABNF은 대소문자를 구분하는 것으로 명확해졌다. 또한 버전 번호는 구현 시 여러 자릿수의 버전 번호를 잘못 취급하는 것으로 알려져 있기 때문에 한 자릿수로 제한되어 왔다. (Section 2.6)

Userinfo (i.e., username and password) are now disallowed in HTTP and HTTPS URIs, because of security issues related to their transmission on the wire. (Section 2.7.1)

Userinfo(즉, 사용자 이름 및 암호)는 현재 유/무선상의 전송과 관련된 보안 문제로 인해 HTTP 및 HTTPS URIs에서 허용되지 않는다. (Section 2.7.1)

The HTTPS URI scheme is now defined by this specification; previously, it was done in Section 2.4 of [RFC2818]. Furthermore, it implies end-to-end security. (Section 2.7.2)

HTTPS URI scheme는 이제 이 명세에 의해 정의된다. 이전에는 [RFC2818]의 Section 2.4에서 수행되었다. 추가로, 그것은 end-to-end 보안을 의미한다. (Section 2.7.2)

HTTP messages can be (and often are) buffered by implementations; despite it sometimes being available as a stream, HTTP is fundamentally a message-oriented protocol. Minimum supported sizes for various protocol elements have been suggested, to improve interoperability. (Section 3)

HTTP 메시지는 구현에 의해 버퍼링될 수 있으며(그리고 종종) 스트림으로 이용 가능함에도 불구하고 HTTP는 근본적으로는 메시지 지향 프로토콜이다. 상호운용성을 개선하기 위해 다양한 프로토콜 요소에 대해 지원되는 최소 크기가 제안되었다. (Section 3)

Invalid whitespace around field-names is now required to be rejected, because accepting it represents a security vulnerability. The ABNF productions defining header fields now only list the field value. (Section 3.2)

field-names 주변의 잘못된 공백을 수락하면 보안 취약성을 나타내기 때문에 잘못된 공백은 거부되어야 한다. 헤더 필드를 정의하는 ABNF은 이제 필드 값만 나열한다. (Section 3.2)

Rules about implicit linear whitespace between certain grammar productions have been removed; now whitespace is only allowed where specifically defined in the ABNF. (Section 3.2.3)

특정 문법 제작 사이의 암시적 선형 공백에 대한 규칙은 제거되었다. 이제 공백은 ABNF에 특별히 정의된 경우에만 허용된다. (Section 3.2.3)

Header fields that span multiple lines ("line folding") are deprecated. (Section 3.2.4)

여러 라인에 걸쳐 있는(“line folding”) 헤더 필드는 더 이상 사용되지 않는다. (Section 3.2.4)

The NUL octet is no longer allowed in comment and quoted-string text, and handling of backslash-escaping in them has been clarified. The quoted-pair rule no longer allows escaping control characters other than HTAB. Non-US-ASCII content in header fields and the reason phrase has been obsoleted and made opaque (the TEXT rule was removed). (Section 3.2.6)

NUL octet은 더 이상 주석과 quoted-string 텍스트에서 허용되지 않으며, 그 안에 있는 backslash-escaping 의 취급이 명확해졌다. quoted-pair 규칙은 더 이상 HTAB 이외의 제어 문자를 이스케이프할 수 없다. 헤더 필드에서 US-ASCII가 아닌 내용 및 이유 구문이 폐기되고 불투명해졌다(TEXT 규칙이 제거됨). (Section 3.2.6)

Bogus Content-Length header fields are now required to be handled as errors by recipients. (Section 3.3.2)

이제 Bogus Content-Length 헤더 필드는 수신자에 의한 오류로 처리되어야 한다. (Section 3.3.2)

The algorithm for determining the message body length has been clarified to indicate all of the special cases (e.g., driven by methods or status codes) that affect it, and that new protocol elements cannot define such special cases. CONNECT is a new, special case in determining message body length. "multipart/byteranges" is no longer a way of determining message body length detection. (Section 3.3.3)

메시지 본문 길이를 결정하는 알고리즘은 그것에 영향을 미치는 모든 특수 케이스(예를 들어 메소드나 상태 코드에 의해 구동됨)를 나타내기 위해 명확히 하고, 새로운 프로토콜 요소로는 그러한 특수 사례를 정의할 수 없다. CONNECT는 메시지 본문 길이를 결정하는 새로운 특별한 경우다. "multipart/byteranges"는 더 이상 메시지 본문 길이 감지를 결정하는 방법이 아니다. (Section 3.3.3)

The "identity" transfer coding token has been removed. (Sections 3.3 and 4)

“identity” 전송 코딩 토큰이 제거되었다. (Section 3.3 및 4)

Chunk length does not include the count of the octets in the chunk header and trailer. Line folding in chunk extensions is disallowed. (Section 4.1)

청크 길이에는 청크 헤더와 트레일러에 있는 octet 개수가 포함되지 않는다. 청크 확장자의 라인 폴딩은 허용되지 않는다. (Section 4.1)

The meaning of the "deflate" content coding has been clarified. (Section 4.2.2)

“deflate” 내용 코딩의 의미가 명확해졌다. (Section 4.2.2)

The segment + query components of RFC 3986 have been used to define the request-target, instead of abs_path from RFC 1808. The asterisk-form of the request-target is only allowed with the OPTIONS method. (Section 5.3)

RFC 3986의 세그먼트 + 쿼리 구성요소는 RFC 1808의 abs_path 대신 request-target을 정의하는 데 사용되어 왔다. request-target의 asterisk-form은 OPTION 메서드에서만 허용된다. (Section 5.3)

The term "Effective Request URI" has been introduced. (Section 5.5)

“Effective Request URI” 라는 용어가 도입되었다. (Section 5.5)

Gateways do not need to generate Via header fields anymore. (Section 5.7.1)

게이트웨이는 더 이상 Via 헤더 필드를 생성할 필요가 없다. (Section 5.7.1)

Exactly when "close" connection options have to be sent has been clarified. Also, "hop-by-hop" header fields are required to appear in the Connection header field; just because they're defined as hop- by-hop in this specification doesn't exempt them. (Section 6.1)

정확히 언제 “close” 커넥션 옵션을 보내야 하는지가 명확해졌다. 또한 “hop-by-hop”헤더 필드는 Connection 헤더 필드에 나타나야 한다. 단지 이 항목에서 hop-by-hop으로 정의된다고 해서 제외되는 것은 아니다. (Section 6.1)

The limit of two connections per server has been removed. An idempotent sequence of requests is no longer required to be retried. The requirement to retry requests under certain circumstances when the server prematurely closes the connection has

been removed. Also, some extraneous requirements about when servers are allowed to close connections prematurely have been removed. (Section 6.3)

서버당 두 개의 커넥션 한도가 제거되었다. 멱등성인 요청 시퀀스는 더 이상 재시도할 필요가 없다. 서버가 커넥션을 조기에 종료할 때 특정 상황에서 요청을 다시 시도해야 하는 요구 사항이 제거됨. 또한, 서버가 언제 조기에 커넥션을 닫을 수 있는지에 대한 일부 관련 요구사항이 제거되었다. (Section 6.3)

The semantics of the Upgrade header field is now defined in responses other than 101 (this was incorporated from [RFC2817]). Furthermore, the ordering in the field value is now significant. (Section 6.7)

Upgrade 헤더 필드의 의미론은 이제 101이 아닌 다른 응답으로 정의된다(이것은 [RFC2817]에서 통합되었다). 게다가, 필드 값의 순서는 이제 중요하다. (Section 6.7)

Empty list elements in list productions (e.g., a list header field containing ", ,") have been deprecated. (Section 7)

목록 제작의 빈 목록 요소(예: “, ,”를 포함하는 목록 헤더 필드)는 더 이상 사용되지 않는다. (Section 7)

Registration of Transfer Codings now requires IETF Review (Section 8.4)

전송 코딩을 등록하려면 IETF 검토가 필요함(Section 8.4)

This specification now defines the Upgrade Token Registry, previously defined in Section 7.2 of [RFC2817]. (Section 8.6)

이 명세는 이전의 [RFC2817]의 Section 7.2에서 정의한 Upgrade Token Registry 를 이제 정의한다. (Section 8.6)

The expectation to support HTTP/0.9 requests has been removed. (Appendix A)

HTTP/0.9 요청을 지원하려는 기대가 제거되었다. (Appendix A)

Issues with the Keep-Alive and Proxy-Connection header fields in requests are pointed out, with use of the latter being discouraged altogether. (Appendix A.1.2)

요청에서 Keep-Alive 및 Proxy-Connection 헤더 필드의 문제가 지적되며, 후자의 사용은 완전히 중단된다. (Appendix.A 1.2)

Appendix B. Collected ABNF

```
BWS = OWS

Connection = *( "," OWS ) connection-option *( OWS "," [ OWS connection-
option ] )

Content-Length = 1*DIGIT

HTTP-message = start-line *( header-field CRLF ) CRLF [ message-body]
HTTP-name = %x48.54.54.50 ; HTTP
HTTP-version = HTTP-name "/" DIGIT "." DIGIT
Host = uri-host [ ":" port ]

OWS = *( SP / HTAB )

RWS = 1*( SP / HTAB )

TE = [ ( "," / t-codings ) *( OWS "," [ OWS t-codings ] ) ]
Trailer = *( "," OWS ) field-name *( OWS "," [ OWS field-name ] )
Transfer-Encoding = *( "," OWS ) transfer-coding *( OWS "," [ OWS transfer-
coding ] )

URI-reference = <URI-reference, see [RFC3986], Section 4.1>
Upgrade = *( "," OWS ) protocol *( OWS "," [ OWS protocol ] )

Via = *( "," OWS ) ( received-protocol RWS received-by [ RWS comment ] ) *( OWS
"," [ OWS ( received-protocol RWS received-by [ RWS comment ] ) ] )

absolute-URI = <absolute-URI, see [RFC3986], Section 4.3>
absolute-form = absolute-URI
absolute-path = 1*( "/" segment )
asterisk-form = "*"
authority = <authority, see [RFC3986], Section 3.2>
```

```
authority-form = authority

chunk = chunk-size [ chunk-ext ] CRLF chunk-data CRLF
chunk-data = 1*OCTET
chunk-ext = *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
chunk-ext-name = token
chunk-ext-val = token / quoted-string
chunk-size = 1*HEXDIG
chunked-body = *chunk last-chunk trailer-part CRLF
comment = "(" *( ctext / quoted-pair / comment ) ")"
connection-option = token
ctext = HTAB / SP / %x21-27 ; '!'-'''
      / %x2A-5B ; '*'-'['
      / %x5D-7E ; ']'-'~'
      / obs-text

field-content = field-vchar [ 1*( SP / HTAB ) field-vchar ]
field-name = token
field-value = *( field-content / obs-fold )
field-vchar = VCHAR / obs-text
fragment = <fragment, see [RFC3986], Section 3.5>

header-field = field-name ":" OWS field-value OWS
http-URI = "http://" authority path-abempty [ "?" query ] [ “#" fragment ]
https-URI = "https://" authority path-abempty [ "?" query ] [ “#" fragment ]

last-chunk = 1*"0" [ chunk-ext ] CRLF

message-body = *OCTET
method = token

obs-fold = CRLF 1*( SP / HTAB )
obs-text = %x80-FF
origin-form = absolute-path [ "?" query ]

partial-URI = relative-part [ "?" query ]
path-abempty = <path-abempty, see [RFC3986], Section 3.3>
port = <port, see [RFC3986], Section 3.2.3>
protocol = protocol-name [ "/" protocol-version ]
protocol-name = token
protocol-version = token
pseudonym = token
```

```
qdtext = HTAB / SP / "!" / %x23-5B ; '#'-'['
      / %x5D-7E ; ']'-'~'
      / obs-text
query = <query, see [RFC3986], Section 3.4>
quoted-pair = "₩" ( HTAB / SP / VCHAR / obs-text )
quoted-string = DQUOTE *( qdtext / quoted-pair ) DQUOTE

rank = ( "0" [ "." *3DIGIT ] ) / ( "1" [ "." *3"0" ] )
reason-phrase = *( HTAB / SP / VCHAR / obs-text )
received-by = ( uri-host [ ":" port ] ) / pseudonym
received-protocol = [ protocol-name "/" ] protocol-version
relative-part = <relative-part, see [RFC3986], Section 4.2>
request-line = method SP request-target SP HTTP-version CRLF
request-target = origin-form / absolute-form / authority-form / asterisk-form

scheme = <scheme, see [RFC3986], Section 3.1>
segment = <segment, see [RFC3986], Section 3.3>
start-line = request-line / status-line
status-code = 3DIGIT
status-line = HTTP-version SP status-code SP reason-phrase CRLF

t-codings = "trailers" / ( transfer-coding [ t-ranking ] )
t-ranking = OWS ";" OWS "q=" rank
tchar = "!" / "#" / "$" / "%" / "&" / "'" / "*" / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~" /
      DIGIT / ALPHA
token = 1*tchar
trailer-part = *( header-field CRLF )
transfer-coding = "chunked" / "compress" / "deflate" / "gzip" / transfer-extension
transfer-extension = token *( OWS ";" OWS transfer-parameter )
transfer-parameter = token BWS "=" BWS ( token / quoted-string )

uri-host = <host, see [RFC3986], Section 3.2.2>
```