



毕业设计（论文）外文资料

原文及译文

专 业 软件工程

班 级 171042Y1

学 号 171042Y119

姓 名 吴新龙

指 导 教 师 朱栩

|  |  |
| --- | --- |
| **原文出处：** | I. Fette and A. Melnikov, The WebSocket Protocol, IETF RFC6 455, December 2011; www.rfc-editor.org/rfc/rfc6455.txt |

**The WebSocket Protocol**

I. Fette & A. Melnikov

Abstract

The WebSocket Protocol enables two-way communication between a client

running untrusted code in a controlled environment to a remote host

that has opted-in to communications from that code. The security

model used for this is the origin-based security model commonly used

by web browsers. The protocol consists of an opening handshake

followed by basic message framing, layered over TCP. The goal of

this technology is to provide a mechanism for browser-based

applications that need two-way communication with servers that does

not rely on opening multiple HTTP connections (e.g., using

XMLHttpRequest or <iframe>s and long polling).

Status of This Memo

This is an Internet Standards Track document.

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.

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/rfc6455.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the

document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal

Provisions Relating to IETF Documents

(http://trustee.ietf.org/license-info) in effect on the date of

publication of this document. Please review these documents

carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as

described in the Simplified BSD License.

1. Introduction

1.1. Background

\_This section is non-normative.\_

Historically, creating web applications that need bidirectional

communication between a client and a server (e.g., instant messaging

and gaming applications) has required an abuse of HTTP to poll the

server for updates while sending upstream notifications as distinct

HTTP calls [RFC6202].

This results in a variety of problems:

o The server is forced to use a number of different underlying TCP

connections for each client: one for sending information to the

client and a new one for each incoming message.

o The wire protocol has a high overhead, with each client-to-server

message having an HTTP header.

o The client-side script is forced to maintain a mapping from the

outgoing connections to the incoming connection to track replies.

A simpler solution would be to use a single TCP connection for

traffic in both directions. This is what the WebSocket Protocol

provides. Combined with the WebSocket API [WSAPI], it provides an

alternative to HTTP polling for two-way communication from a web page

to a remote server.

The same technique can be used for a variety of web applications:

games, stock tickers, multiuser applications with simultaneous

editing, user interfaces exposing server-side services in real time,

etc.

The WebSocket Protocol is designed to supersede existing

bidirectional communication technologies that use HTTP as a transport

layer to benefit from existing infrastructure (proxies, filtering,

authentication). Such technologies were implemented as trade-offs

between efficiency and reliability because HTTP was not initially

meant to be used for bidirectional communication (see [RFC6202] for

further discussion). The WebSocket Protocol attempts to address the

goals of existing bidirectional HTTP technologies in the context of

the existing HTTP infrastructure; as such, it is designed to work

over HTTP ports 80 and 443 as well as to support HTTP proxies and

intermediaries, even if this implies some complexity specific to the

current environment. However, the design does not limit WebSocket to

HTTP, and future implementations could use a simpler handshake over a

dedicated port without reinventing the entire protocol. This last

point is important because the traffic patterns of interactive

messaging do not closely match standard HTTP traffic and can induce

unusual loads on some components.

1.2. Protocol Overview

\_This section is non-normative.\_

The protocol has two parts: a handshake and the data transfer.

The handshake from the client looks as follows:

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

Origin: http://example.com

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

The handshake from the server looks as follows:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

Sec-WebSocket-Protocol: chat

The leading line from the client follows the Request-Line format.

The leading line from the server follows the Status-Line format. The

Request-Line and Status-Line productions are defined in [RFC2616].

An unordered set of header fields comes after the leading line in

both cases. The meaning of these header fields is specified in

Section 4 of this document. Additional header fields may also be

present, such as cookies [RFC6265]. The format and parsing of

headers is as defined in [RFC2616].

Once the client and server have both sent their handshakes, and if

the handshake was successful, then the data transfer part starts.

This is a two-way communication channel where each side can,

independently from the other, send data at will.

After a successful handshake, clients and servers transfer data back

and forth in conceptual units referred to in this specification as

"messages". On the wire, a message is composed of one or more

frames. The WebSocket message does not necessarily correspond to a

particular network layer framing, as a fragmented message may be

coalesced or split by an intermediary.

A frame has an associated type. Each frame belonging to the same

message contains the same type of data. Broadly speaking, there are

types for textual data (which is interpreted as UTF-8 [RFC3629]

text), binary data (whose interpretation is left up to the

application), and control frames (which are not intended to carry

data for the application but instead for protocol-level signaling,

such as to signal that the connection should be closed). This

version of the protocol defines six frame types and leaves ten

reserved for future use.

1.3. Opening Handshake

\_This section is non-normative.\_

The opening handshake is intended to be compatible with HTTP-based

server-side software and intermediaries, so that a single port can be

used by both HTTP clients talking to that server and WebSocket

clients talking to that server. To this end, the WebSocket client's

handshake is an HTTP Upgrade request:

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

Origin: http://example.com

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

In compliance with [RFC2616], header fields in the handshake may be

sent by the client in any order, so the order in which different

header fields are received is not significant.

The "Request-URI" of the GET method [RFC2616] is used to identify the

endpoint of the WebSocket connection, both to allow multiple domains

to be served from one IP address and to allow multiple WebSocket

endpoints to be served by a single server.

The client includes the hostname in the |Host| header field of its

handshake as per [RFC2616], so that both the client and the server

can verify that they agree on which host is in use.

Additional header fields are used to select options in the WebSocket

Protocol. Typical options available in this version are the

subprotocol selector (|Sec-WebSocket-Protocol|), list of extensions

support by the client (|Sec-WebSocket-Extensions|), |Origin| header

field, etc. The |Sec-WebSocket-Protocol| request-header field can be

used to indicate what subprotocols (application-level protocols

layered over the WebSocket Protocol) are acceptable to the client.

The server selects one or none of the acceptable protocols and echoes

that value in its handshake to indicate that it has selected that

protocol.

Sec-WebSocket-Protocol: chat

The |Origin| header field [RFC6454] is used to protect against

unauthorized cross-origin use of a WebSocket server by scripts using

the WebSocket API in a web browser. The server is informed of the

script origin generating the WebSocket connection request. If the

server does not wish to accept connections from this origin, it can

choose to reject the connection by sending an appropriate HTTP error

code. This header field is sent by browser clients; for non-browser

clients, this header field may be sent if it makes sense in the

context of those clients.

Finally, the server has to prove to the client that it received the

client's WebSocket handshake, so that the server doesn't accept

connections that are not WebSocket connections. This prevents an

attacker from tricking a WebSocket server by sending it carefully

crafted packets using XMLHttpRequest [XMLHttpRequest] or a form

submission.

To prove that the handshake was received, the server has to take two

pieces of information and combine them to form a response. The first

piece of information comes from the |Sec-WebSocket-Key| header field

in the client handshake:

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

For this header field, the server has to take the value (as present

in the header field, e.g., the base64-encoded [RFC4648] version minus

any leading and trailing whitespace) and concatenate this with the

Globally Unique Identifier (GUID, [RFC4122]) "258EAFA5-E914-47DA-

95CA-C5AB0DC85B11" in string form, which is unlikely to be used by

network endpoints that do not understand the WebSocket Protocol. A

SHA-1 hash (160 bits) [FIPS.180-3], base64-encoded (see Section 4 of

[RFC4648]), of this concatenation is then returned in the server's

handshake.

Concretely, if as in the example above, the |Sec-WebSocket-Key|

header field had the value "dGhlIHNhbXBsZSBub25jZQ==", the server

would concatenate the string "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"

to form the string "dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-

C5AB0DC85B11". The server would then take the SHA-1 hash of this,

giving the value 0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16 0x90 0xf6

0x46 0x06 0xcf 0x38 0x59 0x45 0xb2 0xbe 0xc4 0xea. This value is

then base64-encoded (see Section 4 of [RFC4648]), to give the value

"s3pPLMBiTxaQ9kYGzzhZRbK+xOo=". This value would then be echoed in

the |Sec-WebSocket-Accept| header field.

The handshake from the server is much simpler than the client

handshake. The first line is an HTTP Status-Line, with the status

code 101:

HTTP/1.1 101 Switching Protocols

Any status code other than 101 indicates that the WebSocket handshake

has not completed and that the semantics of HTTP still apply. The

headers follow the status code.

The |Connection| and |Upgrade| header fields complete the HTTP

Upgrade. The |Sec-WebSocket-Accept| header field indicates whether

the server is willing to accept the connection. If present, this

header field must include a hash of the client's nonce sent in

|Sec-WebSocket-Key| along with a predefined GUID. Any other value

must not be interpreted as an acceptance of the connection by the

server.

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

These fields are checked by the WebSocket client for scripted pages.

If the |Sec-WebSocket-Accept| value does not match the expected

value, if the header field is missing, or if the HTTP status code is

not 101, the connection will not be established, and WebSocket frames

will not be sent.

Option fields can also be included. In this version of the protocol,

the main option field is |Sec-WebSocket-Protocol|, which indicates

the subprotocol that the server has selected. WebSocket clients

verify that the server included one of the values that was specified

in the WebSocket client's handshake. A server that speaks multiple

subprotocols has to make sure it selects one based on the client's

handshake and specifies it in its handshake.

Sec-WebSocket-Protocol: chat

The server can also set cookie-related option fields to \_set\_

cookies, as described in [RFC6265].

1.4. Closing Handshake

\_This section is non-normative.\_

The closing handshake is far simpler than the opening handshake.

Either peer can send a control frame with data containing a specified

control sequence to begin the closing handshake (detailed in

Section 5.5.1). Upon receiving such a frame, the other peer sends a

Close frame in response, if it hasn't already sent one. Upon

receiving \_that\_ control frame, the first peer then closes the

connection, safe in the knowledge that no further data is

forthcoming.

After sending a control frame indicating the connection should be

closed, a peer does not send any further data; after receiving a

control frame indicating the connection should be closed, a peer

discards any further data received.

It is safe for both peers to initiate this handshake simultaneously.

The closing handshake is intended to complement the TCP closing

handshake (FIN/ACK), on the basis that the TCP closing handshake is

not always reliable end-to-end, especially in the presence of

intercepting proxies and other intermediaries.

By sending a Close frame and waiting for a Close frame in response,

certain cases are avoided where data may be unnecessarily lost. For

instance, on some platforms, if a socket is closed with data in the

receive queue, a RST packet is sent, which will then cause recv() to

fail for the party that received the RST, even if there was data

waiting to be read.

1.5. Design Philosophy

\_This section is non-normative.\_

The WebSocket Protocol is designed on the principle that there should

be minimal framing (the only framing that exists is to make the

protocol frame-based instead of stream-based and to support a

distinction between Unicode text and binary frames). It is expected

that metadata would be layered on top of WebSocket by the application

layer, in the same way that metadata is layered on top of TCP by the

application layer (e.g., HTTP).

Conceptually, WebSocket is really just a layer on top of TCP that

does the following:

o adds a web origin-based security model for browsers

o adds an addressing and protocol naming mechanism to support

multiple services on one port and multiple host names on one IP

address

o layers a framing mechanism on top of TCP to get back to the IP

packet mechanism that TCP is built on, but without length limits

o includes an additional closing handshake in-band that is designed

to work in the presence of proxies and other intermediaries

Other than that, WebSocket adds nothing. Basically it is intended to

be as close to just exposing raw TCP to script as possible given the

constraints of the Web. It's also designed in such a way that its

servers can share a port with HTTP servers, by having its handshake

be a valid HTTP Upgrade request. One could conceptually use other

protocols to establish client-server messaging, but the intent of

WebSockets is to provide a relatively simple protocol that can

coexist with HTTP and deployed HTTP infrastructure (such as proxies)

and that is as close to TCP as is safe for use with such

infrastructure given security considerations, with targeted additions

to simplify usage and keep simple things simple (such as the addition

of message semantics).

The protocol is intended to be extensible; future versions will

likely introduce additional concepts such as multiplexing.

1.6. Security Model

\_This section is non-normative.\_

The WebSocket Protocol uses the origin model used by web browsers to

restrict which web pages can contact a WebSocket server when the

WebSocket Protocol is used from a web page. Naturally, when the

WebSocket Protocol is used by a dedicated client directly (i.e., not

from a web page through a web browser), the origin model is not

useful, as the client can provide any arbitrary origin string.

This protocol is intended to fail to establish a connection with

servers of pre-existing protocols like SMTP [RFC5321] and HTTP, while

allowing HTTP servers to opt-in to supporting this protocol if

desired. This is achieved by having a strict and elaborate handshake

and by limiting the data that can be inserted into the connection

before the handshake is finished (thus limiting how much the server

can be influenced).

It is similarly intended to fail to establish a connection when data

from other protocols, especially HTTP, is sent to a WebSocket server,

for example, as might happen if an HTML "form" were submitted to a

WebSocket server. This is primarily achieved by requiring that the

server prove that it read the handshake, which it can only do if the

handshake contains the appropriate parts, which can only be sent by a

WebSocket client. In particular, at the time of writing of this

specification, fields starting with |Sec-| cannot be set by an

attacker from a web browser using only HTML and JavaScript APIs such

as XMLHttpRequest [XMLHttpRequest].

1.7. Relationship to TCP and HTTP

\_This section is non-normative.\_

The WebSocket Protocol is an independent TCP-based protocol. Its

only relationship to HTTP is that its handshake is interpreted by

HTTP servers as an Upgrade request.

By default, the WebSocket Protocol uses port 80 for regular WebSocket

connections and port 443 for WebSocket connections tunneled over

Transport Layer Security (TLS) [RFC2818].

1.8. Establishing a Connection

\_This section is non-normative.\_

When a connection is to be made to a port that is shared by an HTTP

server (a situation that is quite likely to occur with traffic to

ports 80 and 443), the connection will appear to the HTTP server to

be a regular GET request with an Upgrade offer. In relatively simple

setups with just one IP address and a single server for all traffic

to a single hostname, this might allow a practical way for systems

based on the WebSocket Protocol to be deployed. In more elaborate

setups (e.g., with load balancers and multiple servers), a dedicated

set of hosts for WebSocket connections separate from the HTTP servers

is probably easier to manage. At the time of writing of this

specification, it should be noted that connections on ports 80 and

443 have significantly different success rates, with connections on

port 443 being significantly more likely to succeed, though this may

change with time.

1.9. Subprotocols Using the WebSocket Protocol

\_This section is non-normative.\_

The client can request that the server use a specific subprotocol by

including the |Sec-WebSocket-Protocol| field in its handshake. If it

is specified, the server needs to include the same field and one of

the selected subprotocol values in its response for the connection to

be established.

These subprotocol names should be registered as per Section 11.5. To

avoid potential collisions, it is recommended to use names that

contain the ASCII version of the domain name of the subprotocol's

originator. For example, if Example Corporation were to create a

Chat subprotocol to be implemented by many servers around the Web,

they could name it "chat.example.com". If the Example Organization

called their competing subprotocol "chat.example.org", then the two

subprotocols could be implemented by servers simultaneously, with the

server dynamically selecting which subprotocol to use based on the

value sent by the client.

Subprotocols can be versioned in backward-incompatible ways by

changing the subprotocol name, e.g., going from

"bookings.example.net" to "v2.bookings.example.net". These

subprotocols would be considered completely separate by WebSocket

clients. Backward-compatible versioning can be implemented by

reusing the same subprotocol string but carefully designing the

actual subprotocol to support this kind of extensibility.

2. Conformance Requirements

All diagrams, examples, and notes in this specification are non-

normative, as are all sections explicitly marked non-normative.

Everything else in this specification is normative.

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].

Requirements phrased in the imperative as part of algorithms (such as

"strip any leading space characters" or "return false and abort these

steps") are to be interpreted with the meaning of the key word

("MUST", "SHOULD", "MAY", etc.) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps MAY

be implemented in any manner, so long as the end result is

equivalent. (In particular, the algorithms defined in this

specification are intended to be easy to follow and not intended to

be performant.)

2.1. Terminology and Other Conventions

\_ASCII\_ shall mean the character-encoding scheme defined in

[ANSI.X3-4.1986].

This document makes reference to UTF-8 values and uses UTF-8

notational formats as defined in STD 63 [RFC3629].

Key terms such as named algorithms or definitions are indicated like

\_this\_.

Names of header fields or variables are indicated like |this|.

Variable values are indicated like /this/.

This document references the procedure to \_Fail the WebSocket

Connection\_. This procedure is defined in Section 7.1.7.

\_Converting a string to ASCII lowercase\_ means replacing all

characters in the range U+0041 to U+005A (i.e., LATIN CAPITAL LETTER

A to LATIN CAPITAL LETTER Z) with the corresponding characters in the

range U+0061 to U+007A (i.e., LATIN SMALL LETTER A to LATIN SMALL

LETTER Z).

Comparing two strings in an \_ASCII case-insensitive\_ manner means

comparing them exactly, code point for code point, except that the

characters in the range U+0041 to U+005A (i.e., LATIN CAPITAL LETTER

A to LATIN CAPITAL LETTER Z) and the corresponding characters in the

range U+0061 to U+007A (i.e., LATIN SMALL LETTER A to LATIN SMALL

LETTER Z) are considered to also match.

The term "URI" is used in this document as defined in [RFC3986].

When an implementation is required to \_send\_ data as part of the

WebSocket Protocol, the implementation MAY delay the actual

transmission arbitrarily, e.g., buffering data so as to send fewer IP

packets.

Note that this document uses both [RFC5234] and [RFC2616] variants of

ABNF in different sections.

3. WebSocket URIs

This specification defines two URI schemes, using the ABNF syntax

defined in RFC 5234 [RFC5234], and terminology and ABNF productions

defined by the URI specification RFC 3986 [RFC3986].

ws-URI = "ws:" "//" host [ ":" port ] path [ "?" query ]

wss-URI = "wss:" "//" host [ ":" port ] path [ "?" query ]

host = <host, defined in [RFC3986], Section 3.2.2>

port = <port, defined in [RFC3986], Section 3.2.3>

path = <path-abempty, defined in [RFC3986], Section 3.3>

query = <query, defined in [RFC3986], Section 3.4>

The port component is OPTIONAL; the default for "ws" is port 80,

while the default for "wss" is port 443.

The URI is called "secure" (and it is said that "the secure flag is

set") if the scheme component matches "wss" case-insensitively.

The "resource-name" (also known as /resource name/ in Section 4.1)

can be constructed by concatenating the following:

o "/" if the path component is empty

o the path component

o "?" if the query component is non-empty

o the query component

Fragment identifiers are meaningless in the context of WebSocket URIs

and MUST NOT be used on these URIs. As with any URI scheme, the

character "#", when not indicating the start of a fragment, MUST be

escaped as %23.

4. Opening Handshake

4.1. Client Requirements

To \_Establish a WebSocket Connection\_, a client opens a connection

and sends a handshake as defined in this section. A connection is

defined to initially be in a CONNECTING state. A client will need to

supply a /host/, /port/, /resource name/, and a /secure/ flag, which

are the components of a WebSocket URI as discussed in Section 3,

along with a list of /protocols/ and /extensions/ to be used.

Additionally, if the client is a web browser, it supplies /origin/.

|  |
| --- |
| **译文：** |

**WebSocket协议**

I. Fette & A. Melnikov

**摘要**

WebSocket协议使在控制环境下运行不受信任代码的客户端和能够选择与那些代码通信的远程主机之间能够双向通信。用于这个的安全模型是以origin为基础的安全模型，一般被浏览器使用。协议包含打开握手，其次是基本消息框架，在TCP之上。这项技术的目的是为基于浏览器的、需要与服务器双向通信的应用程序提供一种不依赖于打开多个HTTP连接的机制（例如，使用XMLHttpRequest 或 <iframe> 和长轮询）。

**本备忘录的状态**

这是一个Internet标准跟踪文件。

这个文档是因特网工程师任务组（IETF）的一个产品。它代表了IETF社区的共识。它已接受公众审查，因特网工程指导组（IESG）证明可出版。关于互联网标准的进一步信息在RFC5741的第2章节。

关于本文档当前状态的信息、勘误表和如何提供反馈，可以在<http://www.rfc-editor.org/info/rfc6455> 找到。

**版权声明**

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

1. **介绍**
   1. 背景

此章节为非规范章节。

在历史上，创建一个客户端和服务端的双向数据Web应用（例如IM应用和游戏应用）需要向服务端频繁发送不同于一般HTTP请求的HTTP轮询请求来从服务端上游更新数据。

这导致各种问题：

* 服务端被迫使用大量的的潜在的TCP连接与客户端进行交互：一部分是用来发送数据，而另一部分是用来接收数据。
* 通信（wire）协议具有很高的开销，因为每个客户端到服务器的消息有HTTP头。
* 服务端被迫使用大量的的潜在的TCP连接与客户端进行交互：一部分是用来发送数据，而另一部分是用来接收数据。

一个简单的解决方案是使用一个简单的TCP链接来进行双向数据传输。这就是WebSocket提供的能力。结合WebSocket的API，它能够提供一个可以替代HTTP轮询的方法来满足Web页面和远端服务器的双向数据通信。

相同的技术可以被用到许多的Web应用：游戏、股票应用、多人协作应用、与后端服务实时交互的用户接口等。。

WebSocket协议设计的原因是取代已经存在的使用HTTP作为传输层的双向通信技术，从而使得已经存在的基础服务（如代理、过滤器、认证服务）能够受益。这种技术是基于效率和可靠性权衡后来进行实现的，而HTTP协议最初也不是用来做双向数据通信的。WebSocket协议尝试实现基于现有的HTTP基础服务来实现在现有环境中双向通信技术的目标；所以，即使这意味着在现有环境中会有一些复杂性，它在设计中仍然使用了HTTP的80和443端口，以及支持HTTP代理。然而，这个设计并没有限制WebSocket只能使用HTTP端口，在以后的实现中也可以使用一个简单的握手方式来使用特定的端口而不需要改动整个协议。最后一点很重要，因为双向消息的通信方式不是很符合标准HTTP的模式，可能导致在某些组件中出现异常的负载。

* 1. 协议概览

此节为非规范章节。

这个协议有两部分：握手和数据传输。

来自客户端的握手数据如下所示：

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

Origin: http://example.com

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

服务端的握手响应如下所示：

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

Sec-WebSocket-Protocol: chat

客户端请求的第一行（leading line）遵循了HTTP请求行的格式。

服务端的第一行（leading line）遵循了HTTP状态行的格式。

HTTP请求行和状态行的规范定义在[RFC2616][4]。

在两个协议中，第一行header下面是一组无序的header字段。这些header字段包含的内容在本文的[第四节][5]。另外的header字段如[cookies][6]，也有可能存在。格式和解析头信息被定义在了[RFC2616][7]。

当客户端和服务端都发送了他们的握手协议，并且当握手已经成功，那么数据传输就开始了。这是一个双方都可以独立发送任意数据的双向通信渠道。

在握手成功以后，客户端和服务端传输的数据来回传输的数据单位，我们在规范中称为消息（messages）。在传输中，一条消息有一个或者多个帧组成。WebSocket中的消息不需要对应特定网络层中的帧，一条零散的消息可能由中间人合并或者拆分成网络层的帧。

帧有关联的类型。同一条消息的每一帧都包含相同类型的数据。通常来说，它可以是文本数据（UTF-8编码）、二进制数据（留给应用解析的数据）和控制帧数据（不是用来传输数据，而是用来作为协议层的特定符号，如关闭连接帧）。当前版本的协议定义了6种控制帧类型并且预留了10个保留类型。

* 1. 打开握手

此节为非规范章节。

开始握手为了与基于HTTP的服务端软件和中介兼容，因此一个独立的端口既能够同时满足HTTP客户端来与服务进行交互，又能够满足WebSocket客户端与服务进行交互。最终，WebSocket客户端的握手是一个基于HTTP的升级请求：

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

Origin: http://example.com

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

遵照RFC2616，客户端在握手过程中发送的header字段可能是乱序的，所以收到的header字段的顺序不同也没有太大影响。

GET方法的请求URI（Request-URI）是用于定义WebSocket连接的终端，允许同一个IP对多个域名提供服务，也允许多个WebSocket终端连接同一个服务器。

客户端在每一个握手的Hostheader里面包含了一个主机域名。所以客户端和服务端都可以校验哪些域名在使用中。

另外的header字段是用来确定WebSocket协议的选项。这个版本中提供的特定选项是子协议选择(Sec-WebSocket-Protocol)、客户端支持的扩展列表（Sec-WebSocket-Extensions）、Originheader字段等。请求header字段Sec-WebSocket-Protocol可以用来标识哪些子协议（基于WebSocket的应用高层协议）是客户端可以支持的。服务端会从中选择零个或者一个支持的协议并且在响应握手中输出它选择的那个协议。

Sec-WebSocket-Protocol: chat

Origin 头域（RFC6454）用于保护WebSocket服务器不被未授权的运行在浏览器的脚本跨源使用WebSocket API。如果服务器不想接受来自这个源的连接，它可以拒绝连接，并发送一个合适的HTTP错误码。这个头域有浏览器客户端发送；对于非浏览器客户端，这个头域可能发送，如果它在客户端上下文环境中有意义。

最后，服务器得向客户端证明它接收到了客户端的WebSocket握手，为使服务器不接受非WebSocket连接。这防止攻击者通过XMLHttpRequest发送或表单提交精心构造的包来欺骗WebSocket服务器。

为了证明握手被接收，服务器把两块信息合并来形成响应。第一块信息来自客户端握手头域Sec-WebSocket-Key，如

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

对于这个头域，服务器取头的值（由于出现在头域，例如，base64编码[RFC4648]后的版本，消除任何前面后面的空白符），以字符串的形式拼接全局唯一的（GUID，[RFC4122]）标识：258EAFA5-E914-47DA-95CA-C5AB0DC85B11，此值不大可能被不明白WebSocket协议的网络终端使用。然后进行SHA-1 hash（160位）编码，再进行base64编码，将结果作为服务器的握手返回。

具体如下：

请求头：Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

取值，字符串拼接后得到："dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11";

SHA-1后得到： 0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16 0x90 0xf6 0x46 0x06 0xcf 0x38 0x59 0x45 0xb2 0xbe 0xc4 0xea

Base64后得到： s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

最后的结果值作为响应头 Sec-WebSocket-Accept 的值。

来自服务器的握手比客户端的简单很多。首先是HTTP 状态行，状态码是101：

HTTP/1.1 101 Switching Protocols

任何非101的状态码表示WebSocket握手还没有完成，HTTP语义仍然生效。

Connection和Upgrade头域完成HTTP升级。Sec-WebSocket-Accept 头表明服务器是否愿意接受连接。如果有，值必须是前面提到的算法得到的值，否则不能解释为服务器接受连接。

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

这些字段由WebSocket客户端为脚本页面检测，如果Sec-WebSocket-Accept的值不符合预期的值，如果头缺失， 或HTTP状态码不是101，连接不会建立，WebSocket帧不会发送。

还可以包含一些可选字段。在协议的这个版本里，主要的可选字段是Sec-WebSocket-Protocol，表示服务器选择的子协议。WebSocket客户端验证服务器选择了一个客户端握手中指定的值。支持多个子协议的服务器必须确保它选择了一个基于客户端握手并在它自己的握手里指定了它。

Sec-WebSocket-Protocol: chat

服务器也可以设置cookie有关的可选头。

* 1. 关闭握手

此节为非规范章节。

结束握手远比连接握手简单。

任何一端都可以发送一个包含特定关闭握手的控制帧数据（详情见5.5.1节）。收到此帧后，另一端在不发送任何数据后会发送一个结束帧作为响应。收到另一端的结束帧后，最开始发送控制帧的端在没有数据需要发送时，就会安全的关闭此连接。

在发送了一个表明连接需要被关闭的控制帧后，这个客户端不会再发送任何的数据；在收到一个表明连接需要被关闭的控制帧后，这个客户端会丢弃此后的所有数据。

这样比两边同时发起握手要更加安全。

这个结束握手的目标是来补充TCP结束握手中的一些内容（FIN/ACK），而这是因为TCP结束握手在端与端之间并不一定可靠，尤其是有代理和其他的网络中介时会变得不可靠。

在发送关闭帧等待接受另一端的响应关闭帧时，在某些情况下可以避免数据的不必要丢失。例如，在某些平台中，如果一个socket在接收队列有数据时被关闭，会发送一个RST包，尽管数据还在等待被读取，这也会导致接收到RST的一方数据接收失败。

* 1. 设计哲学

此节为非规范章节。

WebSocket协议设计的原理，将框架最小化，对框架的唯一的约束就是使这个协议是基于帧而不是流并且可以支持Unicode文本和二进制帧两者中的任意一种。在基于WebSocket的应用层中，元数据是应该分层的，就像基于TCP的应用层（例如HTTP）一样。

从概念上来看，WebSocket层是基于TCP实现的，增加了以下的内容：

* 增加了一个基于浏览器的同源策略模型
* 增加了一个地址和协议命名机制用以在同一个端口上支持多个服务，在同一个IP地址自持多个主机名
* 在TCP协议上分层构建框架机制回到TCP使用的IP包机制，但是没有长度限制
* 包含一个设计用于处理有代理和其他网络中介的情况的额外的结束握手协议

除此之外，WebSocket没有增加任何东西。基本上WebSocket的的目标是在约束的条件下向脚本提供尽可能接近原生的TCP的Web服务。它同时考虑了服务器在进行握手和处理有效的HTTP升级请求时，可以和HTTP共用一个服务。大家也可以使用其他协议来建立从客户端到服务端的消息通信，但WebSocket的协议的目的是为了提供一个相对简单的可以和HTTP共存，并且依赖于HTTP基础设施（如代理）的协议。这个非常接近TCP的协议因为基于安全的基础设施和针对性的能够简单使用和让事情变得更简单的补充（例如消息语义的补充），因此可以安全使用。

这个协议具有可扩展性，未来的版本可能会引入一些新的概念如多路复用。

* 1. 安全模型

此节为非规范章节。

当WebSocket协议在web网页中应用时，WebSocket协议在Web页面与WebSocket服务器建立连接时使用基于web浏览器的同源策略模型。所以说，当WebSocket协议在一个特定的客户端（不是web浏览器里面的网页）直接使用时，同源策略模型就不生效了，客户端可以接受任意的源数据。

该协议无法与已经存在的如SMTP（[RFC5421][10]）和HTTP协议的服务器建立连接，如果需要的话，HTTP服务器可以选择支持该协议。该协议还实现了严格约束的握手过程和限制数据不能在握手完成和建立连接之前插入数据进行传输（因此限制了许多被影响的服务器）。

WebSocket服务器同样无法与其他协议尤其是HTTP建立连接。例如，一个HTML“表单”可能会提交给一个WebSocket服务器。WebSocket服务端只能读取包含特定的由WebSocket客户端发送的字段的握手数据。尤其是在编写这个规范时，攻击者不能只使用HTML和JavaScript APIs的Web浏览器来发送以Sec-开头的字段。

* 1. 与TCP和HTTP的关系

此节为非规范章节。

WebSocket协议是独立的基于TCP的协议。他和HTTP的唯一关系是建立连接的握手操作的升级请求是基于HTTP服务器的。

WebSocket默认使用80端口进行连接，而基于TLS（[RFC2818][11]）的WebSocket连接是基于443端口的。

* 1. 建立连接

此节为非规范章节。

当建立了一个和HTTP服务器共享端口的连接时（这种情况很有可能发送在与80和443端口通信上），这个链接将会给HTTP服务器发送一个常规的GET请求来进行升级。在一个IP地址和一个单一的服务器来应对单一主机名的通信这种相对简单的设置上，基于WebSocket协议的系统可以通过一个更加实用的方法来进行部署。在更详细的设置（例如负载均衡和多服务器），与HTTP服务器分开的专属的WebSocket连接集群可能更加易于管理。在编写这个规范时，我们应该知道在80端口和443端口建立WebSocket连接的成功率是不同的，在443端口上面建立的连接很明显更容易成功，尽管这可能随着时间的变化而改变。

* 1. 使用WebSocket协议的子协议

客户端可以通过在握手阶段中的Sec-WebSocket-protocol字段来请求服务端使用指定的子协议。如果指定了这个字段，服务器需要包含相同的字段，并且从子协议的之中选择一个值作为建立连接的响应。

子协议的名称可以按照第11.5节的方法进行注册。为了避免潜在的冲突，推荐使用包含ASCII码的域名名称作为子协议名。例如，Example Corporation创造了在Web上通过多个服务器实现的一个聊天子协议（Chat subprotocol），他们可以叫做chat.example.com。如果Example Organization创造了他们相对的子协议叫做chat.example.org，这两个子协议可以被服务器同时实现，服务器可以根据客户端来动态的选择使用哪一个子协议。

子协议也可以通过修改名字的方式来向后兼容，例如：将bookings.example.net改为v2.bookings.example.net。WebSocket客户端能够完全的区分这些子协议。向后兼容的版本控制可以通过复用相同的子协议字符和小心设计的子协议实现来保证这种扩展性。

1. **一致性要求**

在这篇文档中，所有的图、示例和笔记都是非规范性的，就像标注了非规范性的所有章节一样。在文档中没有指定的其他内容都是规范性的。

在这篇文档中的关键词如“必须（MUST）”、“必须不（MUST NOT）”、“需要（RWQUIRE）”、“应该（SHALL）”、“不应该（SHALL NOT）”、“应该（SHOULD）”、“不应该（SHOULD NOT）”、“推荐（RECOMMENDED）”、“也许（MAY）”和“可选（OPTIONAL）”可以按照[RFC2119

](https://tools.ietf.org/html/rfc2119)所述进行解释。

作为算法的一部分的命令式语句（如“删除任何前导空格”或“返回false并且中止后续步骤”）在介绍算法时应该与关键词一起解释（“必须（MUST）”、“应该（SHOULD）”、“也许（MAY）”等）。

算法或者指定步骤中的符合要求的措辞可以通过任何方式表述，只要最终的结果是等价的。（尤其是在算法定义中，我们的目标是竟可能简单的操作而不是最求完美。）

* 1. 术语和其他公约

\_ASCII\\_表示定义在[ANSI.X3-4.1986][1]的字符编码表。

这个文档参考UTF-8的值，使用在STD 63（[RFX3629][2]）定义的UTF-8标准格式。

如命名算法或者定义关键输入的标识如\\_this\\_。

命名header字段或者变量如|this|。

本文引用了WebSocket连接失败（\\_Fail the WebSocket Connection\\_）这个程序。这个程序位于第7.1.7节。

转换小写字符（\\_Converting a string to ASCII lowercase\\_）意味着替换从U+0041到U+005A的所有字符（拉丁字母大写A到Z）为相对应的U+0061到U+007A的字符（拉丁字母小写A-Z）。

不区分ASCII大小写（\\_ASCII case-insensitive\\_）比较方式意味着通过码点（code point）比较这两个字符，如果这两个字符是U+0041到U+005A（拉丁字母大写A到Z）和相对应的U+0061到U+007A的字符（拉丁字母小写A-Z），那么也认为这两个字符相等。

文档中URI这个词被定sj义在了[RFC3986][3]。

当需要实现WebSocket协议中一部分的\\_send\\_数据时，这个实现是有可能会延迟任意时间来进行数据传输的，例如，使用数据缓冲区来保证发送较少的IP数据包。

这个文档在不同的章节会同时使用[RFC5234][4]和[RFC2616][5]这两个中的扩充巴科斯-瑙尔范式（ABNF）。

1. **WebSocket URI**

本规范定义了两种URI方案，使用在RFC5234定义的ABNF语法、术语和在RFC3986定义的URI规范的ABNF成果。

ws-URI = "ws:" "//" host [ ":" port ] path [ "?" query ]

wss-URI = "wss:" "//" host [ ":" port ] path [ "?" query ]

host = <host, defined in [RFC3986], Section 3.2.2>

port = <port, defined in [RFC3986], Section 3.2.3>

path = <path-abempty, defined in [RFC3986], Section 3.3>

query = <query, defined in [RFC3986], Section 3.4>

端口字段是可选的，默认的"ws"端口是80，而默认的"wss"端口是443。

命中不论大小写的"wss"方案字段就表明这个URI可以被称为安全的（已经设置安全标记）。

"resource-name"字段（在4.1节也被称为/resouce name/字段）可以通过以下的数据串联起来：

* "/"，表示路径（path）字段为空
* 路径（path）字段
* "?"，表示非空的查询参数（query）
* 空查询参数（query）

在WebSocket URIs的里，身份标识片段是没有意义的，而且禁止使用在这些URI里面。与任何的URI方案一样，"#"字符不是表示片段（fragment）开始时，都必须编码为%23。