Internet Engineering Task Force (IETF)
Request for Comments: 9113
Obsoletes: 7540, 8740
Category: Standards Track
ISSN: 2070-1721
June 2022

M. Thomson, Ed.
Mozilla
C. Benfield, Ed.
Apple Inc.

HTTP/2

Аннотация

Эта спецификация описывает оптимизированное выражение семантики протокола передачи гипертекста (Hypertext Transfer Protocol или HTTP), называемого HTTP версии 2 (HTTP/2). HTTP/2 обеспечивает более эффективно использование ресурсов сети и сокращение задержек благодаря сжатию полей и возможности множества обменов одновременно через одно соединение.

Этот документ отменяет RFC 7540 и RFC 8740.

Статус документа

Документ относится к категории Internet Standards Track.

Документ является результатом работы IETF[1] и представляет согласованный взгляд сообщества IETF. Документ прошёл открытое обсуждение и был одобрен для публикации IESG[2]. Дополнительную информацию о стандартах Internet можно найти в разделе 2 в RFC 7841.

Информацию о текущем статусе документа, ошибках и способах обратной связи можно найти по ссылке https://www.rfc-editor.org/info/rfc9113.

Авторские права

Copyright (c) 2022. Авторские права принадлежат IETF Trust и лицам, указанным в качестве авторов документа. Все права защищены.

К документу применимы права и ограничения, указанные в BCP 78 и IETF Trust Legal Provisions и относящиеся к документам IETF (http://trustee.ietf.org/license-info), на момент публикации данного документа. Прочтите упомянутые документы внимательно. Фрагменты программного кода, включённые в этот документ, распространяются в соответствии с упрощённой лицензией BSD, как указано в параграфе 4.e документа IETF Trust Legal Provisions, без каких-либо гарантий (как указано в Revised BSD License).

Оглавление

1. Введение.....2
2. Обзор протокола HTTP/2.....3
  2.1. Организация документа.....3
  2.2. Соглашения и термины.....3
3. Начало работы HTTP/2.....4
  3.1. Идентификация версии HTTP/2.....4
  3.2. Начало работы HTTP/2 для URI https.....4
  3.3. Начало работы HTTP/2 с предварительными сведениями.....4
  3.4. Предисловие соединения HTTP/2.....5
4. Кадры HTTP.....5
  4.1. Формат кадра.....5
  4.2. Размер кадра.....5
  4.3. Сжатие и декомпрессия раздела полей.....6
    4.3.1. Состояние сжатия.....6
5. Потоки и мультиплексирование.....6
  5.1. Состояния потока.....7
    5.1.1. Идентификаторы потоков.....9
    5.1.2. Одновременные потоки.....9
  5.2. Управление потоком данных.....9
    5.2.1. Принципы управления потоком данных.....9
    5.2.2. Приемлемое использование управления потоком данных.....10
    5.2.3. Производительность управления потоком данных.....10
  5.3. Приоритизация.....10
    5.3.1. Приоритеты в RFC 7540.....10
    5.3.2. Указание приоритета в этом документе.....10
  5.4. Обработка ошибок.....11
    5.4.1. Обработка ошибок соединения.....11
    5.4.2. Обработка ошибок потока.....11
    5.4.3. Прерывание соединения.....11
  5.5. Расширение HTTP/2.....11
6. Определения кадров.....12
  6.1. DATA.....12

[1]Internet Engineering Task Force - комиссия по решению инженерных задач Internet.
[2]Internet Engineering Steering Group - комиссия по инженерным разработкам Internet.

6.2. HEADERS....13
6.3. PRIORITY....13
6.4. RST_STREAM....14
6.5. SETTINGS....14
6.5.1. Формат SETTINGS....15
6.5.2. Заданные настройки....15
6.5.3. Синхронизация установок....16
6.6. PUSH_PROMISE....16
6.7. PING....17
6.8. GOAWAY....17
6.9. WINDOW_UPDATE....19
6.9.1. Окно управления потоком данных....19
6.9.2. Начальный размер окна управления потоком данных....20
6.9.3. Снижение размера окна для потока....20
6.10. CONTINUATION....20
7. Коды ошибок....20
8. Выражение семантики HTTP в HTTP/2....21
8.1. Кадрирование сообщений HTTP....21
8.1.1. Некорректно сформированные сообщения....22
8.2. Поля HTTP....22
8.2.1. Пригодность поля....22
8.2.2. Связанные с соединением поля заголовка....22
8.2.3. Сжатие поля заголовка Cookie....23
8.3. Данные управления HTTP....23
8.3.1. Поля псевдозаголовка запроса....23
8.3.2. Поля псевдозаголовка отклика....24
8.4. Выталкивание с сервера....24
8.4.1. Выталкивание запроса....24
8.4.2. Выталкивание отклика....25
8.5. Метод CONNECT....25
8.6. Поле заголовка Upgrade....26
8.7. Надёжность запросов....26
8.8. Примеры....26
8.8.1. Простой запрос....26
8.8.2. Простой отклик....26
8.8.3. Составной запрос....26
8.8.4. Отклик с телом....27
8.8.5. Информационный отклик....27
9. Соединения HTTP/2....27
9.1. Управление соединениями....27
9.1.1. Повторное использование соединения....27
9.2. Использование свойств TLS....28
9.2.1. Свойства TLS 1.2....28
9.2.2. Шифры TLS 1.2....28
9.2.3. Свойства TLS 1.3....28
10. Вопросы безопасности....29
10.1. Полномочия сервера....29
10.2. Кросс-протокольные атаки....29
10.3. Атаки на промежуточную инкапсуляцию....29
10.4. Кэшируемость выталкиваемых откликов....29
10.5. DoS-атаки....29
10.5.1. Предельный размер блока полей....30
10.5.2. Проблемы CONNECT....30
10.6. Использование сжатия....30
10.7. Использование заполнения....31
10.8. Вопросы приватности....31
10.9. Удалённые атаки по времени....31
11. Взаимодействие с IANA....31
11.1. Регистрация поля заголовка HTTP2-Settings....31
11.2. Маркер обновления h2c....32
12. Литература....32
12.1. Нормативные документы....32
12.2. Дополнительная литература....32
Приложение A. Запрещённые шифры TLS 1.2....33
Приложение B. Отличия от RFC 7540....35
Благодарности....35
Участники работы....35
Адреса авторов....35

1. Введение

Производительность приложений, использующих протокол HTTP [HTTP] зависит от способа использования базового транспорта каждой версией HTTP и условий работы этого транспорта.

Выполнение нескольких запросов одновременно может снизить задержку и повысить производительность приложения. В HTTP/1.0 разрешается лишь один запрос в каждый момент работы данного соединения TCP [TCP]. В HTTP/1.1 [HTTP/1.1] добавлен конвейер запросов, но это лишь частично решило задачу параллельных запросов и по-прежнему

сталкивается с проблемой блокировки head-of-line. Поэтому клиенты HTTP/1.0 и HTTP/1.1 используют множество соединений с сервером для одновременного выполнения запросов.

Поля HTTP часто повторяются и многословны, что вызывает избыточный сетевой трафик, а также быстрое заполнение начального окна перегрузки TCP. Это может вызывать излишние задержки при выполнении нескольких запросов через новое соединение TCP.

В HTTP/2 эти проблемы решаются путём оптимизированного отображения семантики HTTP на базовое соединение. В частности, можно чередовать сообщения в одном соединении и применять эффективное кодирование полей HTTP. Возможна также приоритизация запросов, позволяющая быстрее выполнять более важные запросы, что дополнительно повышает производительность.

В результате протокол более дружелюбен к сети, поскольку может применяться меньшее число соединений TCP, нежели для HTTP/1.x. Это означает меньшую конкуренцию с другими потоками и долгосрочными соединениями, что, в свою очередь, ведёт к более эффективному использованию доступной пропускной способности сети. Однако, следует отметить, что протокол не решает проблему блокировки head-of-line в TCP.

HTTP/2 также обеспечивает более эффективную обработку сообщений за счёт применения двоичной структуры сообщений.

Этот документ отменяет RFC 7540 и RFC 8740. Важные отличия приведены в Приложении B.

2. Обзор протокола HTTP/2

HTTP/2 обеспечивает оптимизированный транспорт для семантики HTTP и поддерживает все базовые свойства HTTP, стремясь быть более эффективным по сравнению с HTTP/1.1.

HTTP/2 - это основанный на соединениях протокол прикладного уровня, работающий через соединения TCP ([TCP]), инициируемые клиентом.

Базовым элементом протокола HTTP/2 является кадр (frame, параграф 4.1). Каждый тип кадра служит для своих целей. Например, кадры HEADERS и DATA являются основой для запросов и откликов HTTP (параграф 8.1), другие типы, такие как SETTINGS, WINDOW_UPDATE, PUSH_PROMISE используются для поддержки иных функций HTTP/2.

Мультиплексирование запросов обеспечивается за счёт связывания каждого обмена HTTP запрос-отклик со своим потоком (раздел 5). Потоки в значительной мере не зависят друг от друга, поэтому заблокированные и остановленные запросы или отклики не мешают обработке других потоков.

Эффективное применение мультиплексирования зависит от управления потоками данных и приоритизацией. Управление потоком данных (параграф 5.2) обеспечивает возможность эффективного использования мультиплексируемых потоков, ограничивая передачу данных возможностями их обработки получателем. Приоритизация (параграф 5.3) обеспечивает наиболее эффективное использование ограниченных ресурсов. В данной версии HTTP/2 отменена схема сигнализации для приоритетов из [RFC7540].

Поскольку применяемые в соединении поля HTTP могут содержать большой объем избыточных данных, кадры с такими данными сжимаются (параграф 4.3). Это сильнее всего сказывается на размере запросов в общем случае, позволяя сжать многие запросы в один пакет.

В HTTP/2 добавлен новый (необязательный) режим взаимодействия, где сервер может выталкивать (push) отклики клиенту (параграф 8.4). Это позволяет серверу по своему усмотрению отправлять клиенту данные, которые сервер считает нужными для этого клиента, что ведёт к некой добавочной нагрузке на сеть но может сокращать задержки. Для этого сервер синтезирует запрос, который он отправляет в виде кадра PUSH_PROMISE. После этого сервер может передать ответ на синтезированный запрос в отдельном потоке.

2.1. Организация документа

Спецификация HTTP/2 делится на 4 части:

- раздел 3 посвящён инициированию соединений HTTP/2;
- разделы 4 (кадры) и 5 (потоки) описывают структурирование кадров HTTP/2 и формирование мультиплексируемых потоков;
- разделы 6 (кадры) и 7 (ошибки) определяют детали кадров и типы ошибок, применяемые в HTTP/2;
- раздел 8 описывает сопоставления, а раздел 9 - дополнительные требования, связанные с выражением семантики HTTP с использованием кадров и потоков.

Хотя некоторые концепции уровней кадров и потоков отделены от HTTP, спецификация на задаёт базовый уровень кадров. Уровни кадров и потоков адаптированы к потребностям HTTP.

2.2. Соглашения и термины

Ключевые слова **необходимо** (MUST), **недопустимо** (MUST NOT), **требуется** (REQUIRED), **нужно** (SHALL), **не следует** (SHALL NOT), **следует** (SHOULD), **не нужно** (SHOULD NOT), **рекомендуется** (RECOMMENDED), **не рекомендуется** (NOT RECOMMENDED), **возможно** (MAY), **необязательно** (OPTIONAL) в данном документе интерпретируются в соответствии с BCP 14 [RFC2119] [RFC8174] тогда и только тогда, когда они выделены шрифтом, как показано здесь.

Все числовые значения приводятся с сетевым порядком байтов и не включают знак, если явно не указано иное. Литеральные значения указываются в десятичном или шестнадцатеричном формате в зависимости от ситуации. Для шестнадцатеричных литералов используется префикс 0x, отличающий их от десятичных литералов.

Эта спецификация описывает двоичные форматы в соответствии с соглашениями, описанными в формате 1.3 RFC 9000 [QUIC]. Отметим, что в этом формате применяется сетевой порядок байтов и сначала указываются старшие биты.

Ниже приведены используемые в документе термины.

client - клиент
: Конечная точка, инициирующая соединение HTTP/2. Клиент передаёт запросы HTTP и получает отклики HTTP.

connection - соединение
: Соединение транспортного уровня между парой конечных точек.

connection error - ошибка соединения
: Ошибка, влияющая на всё соединение HTTP/2.

endpoint - конечная точка
: Клиент или сервер в соединении.

frame - кадр
: Наименьшая единица коммуникаций в соединении HTTP/2, состоящая из заголовка и последовательности октетов с переменным размером в соответствии с типом кадра.

peer - партнёр
: Конечная точка. При обсуждении конкретной точки партнёром называется удалённая по отношению к ней точка.

receiver - получатель
: Конечная точка, принимающая кадры.

sender - отправитель
: Конечная точка, передающая кадры.

server - сервер
: Конечная точка, воспринимающая соединение HTTP/2. Сервер принимает запросы HTTP и передаёт отклики HTTP.

stream - поток
: Двухсторонний поток кадров в соединении HTTP/2connection.

stream error - ошибка потока
: Ошибка в отдельном потоке HTTP/2.

Термины шлюз *gateway), посредник (intermediary), прокси (proxy) и туннель (tunnel) определены в параграфе 3.7 [HTTP]. Посредники выступают в разные моменты как клиенты и серверы.

Термин содержимое (content) применительно к телу сообщения определён в параграфе 6.4 [HTTP].

3. Начало работы HTTP/2

Реализации, генерирующей запросы HTTP, нужно выяснить, поддерживает ли сервер HTTP/2.

В HTTP/2 применяются схемы URI http и https, определённые в параграфе 4.2 [HTTP], с теми же принятыми по умолчанию номерами портов, что и HTTP/1.1 [HTTP/1.1]. Эти URI не включают индикации версий HTTP, поддерживаемых сервером восходящего направления (прямой партнёр, с которым клиент хочет организовать соединение). Средства определения поддержки HTTP/2 различаются для схем URI http и https, вариант для https описан в параграфе 3.2. Поддержку HTTP/2 для URI http можно обнаружить только внешними (out-of-band) средствами и знать о ней нужно заранее, как описано в параграфе 3.3.

3.1. Идентификация версии HTTP/2

Заданный этим документом протокол имеет два идентификатора и создание соединения на основе любого их них предполагает использование транспорта, кадрирования и семантики сообщений, определённых в этом документе.

- Строка h2 указывает протокол, где HTTP/2 использует защиту транспортного уровня (Transport Layer Security или TLS), см. параграф 9.2. Этот идентификатор применяется в поле расширения согласования протокола транспортного уровня TLS (TLS Application-Layer Protocol Negotiation или ALPN) [TLS-ALPN] и в любом месте, где идентифицирован протокол HTTP/2 через TLS. Строка h2 преобразуется в идентификатор протокола ALPN в форме двухоктетной последовательности 0x68, 0x32.
- Строка h2c ранее применялась как маркер для использования в поле заголовка Upgrade механизма HTTP Upgrade (параграф 7.8 в [HTTP]). Это так и не получило широкого распространения и отменено данным документом. Это же относится к полю заголовка HTTP2-Settings, которое применялось при обновлении до h2c.

3.2. Начало работы HTTP/2 для URI https

Клиент, запрашивающий URI https, использует TLS [TLS13] с расширением ALPN [TLS-ALPN].

HTTP/2 через TLS использует идентификатор протокола h2. Идентификатор протокола h2c **недопустимо** передавать клиенту или выбирать серверу, поскольку этот идентификатор указывает протокол, не использующий TLS.

По завершении согласования TLS клиент и сервер **должны** передать предисловие к соединению (параграф 3.4).

3.3. Начало работы HTTP/2 с предварительными сведениями

Клиент может узнать о поддержке сервером HTTP/2 другими способами, например, это может быть указано в его конфигурации. Клиент, знающий о поддержке сервером HTTP/2, может организовать соединение TCP и передать предисловие (параграф 3.4), а затем - кадры HTTP/2. Серверы могут идентифицировать такие соединения по наличию предисловия. Это влияет лишь на организацию соединений HTTP/2 через «чистый» протокол TCP, в соединениях через TLS **должно** применяться согласование протокола в TLS [TLS-ALPN].

Сервер тоже **должен** передать предисловие к соединению (параграф 3.4).

Без дополнительных сведений нельзя полагать, что сервер будет поддерживать HTTP/2 в будущих соединениях. Например, может измениться конфигурация сервера, а также могут различаться конфигурации разных серверов или условия в сети.

3.4. Предисловие соединения HTTP/2

В HTTP/2 каждая конечная точка должна передать предисловие к соединению как окончательное подтверждение применяемого протокола и установки начальных настроек для соединения HTTP/2. Клиент и сервер передают разные предисловия к соединению. Клиентское предисловие начинается с последовательности из 24 октетов

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

т. е. предисловие начинается со строки PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n. За ней **должен** следовать кадр SETTINGS (параграф 6.5), который **может** быть пустым. Клиент передаёт предисловие к соединению в качестве первых октетов данных приложения в соединении.

> Примечание. Клиентское предисловие к соединению выбрано так, что большинство серверов и посредников HTTP/1.1 и HTTP/1.0 не пытаются обрабатывать последующие кадры. Отметим, что это не решает проблемы, указанные в [TALKING].

Серверное предисловие к соединению состоит из (возможно пустого) кадра SETTINGS (параграф 6.5), который должен быть первым кадром, передаваемым сервером в соединение HTTP/2. Кадры SETTINGS, полученные от партнёра как часть предисловия к соединению, **должны** подтверждаться (параграф 6.5.3) после отправки предисловия.

Чтобы избежать ненужных задержек, клиентам разрешено передавать серверу дополнительные кадры сразу же после отправки им предисловия к соединению без ожидания предисловия от сервера. Однако важно отметить, что кадр SETTINGS в серверном предисловии к соединению может включать настройки, которые должны изменить взаимодействие клиента с сервером. Предполагается, что при получении кадра SETTINGS клиент установит указанные сервером настройки. В некоторых конфигурациях сервер может передавать SETTINGS до отправки клиентом дополнительных кадров, что позволяет избежать этой проблемы.

Клиент и сервер **должны** считать недействительное предисловие к соединению ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. В этом случае кадр GOAWAY (параграф 6.8) **можно** опустить, поскольку недействительное предисловие указывает, что партнёр не использует HTTP/2.

4. Кадры HTTP

После организации соединения HTTP/2 конечные точки могут начинать обмен кадрами.

4.1. Формат кадра

Все кадры начинаются с фиксированного 9-октетного заголовка, за которым следуют данные переменного размера.

```
HTTP Frame {
  Length (24),
  Type (8),

  Flags (8),

  Reserved (1),
  Stream Identifier (31),

  Frame Payload (..),
}
```

Рисунок 1.Схема кадра.

Length

24-битовое целое число без знака, указывающее размер кадра в октетах без учёта 9 октетов заголовка кадра. **Недопустимо** передавать значения больше 2^{14} (16384), если получатель не указал большее значение для SETTINGS_MAX_FRAME_SIZE.

Type

8-битовое значение типа, определяющее формат и семантику кадра. Определения типов приведены в разделе 6. Реализация **должна** игнорировать и отбрасывать кадры неизвестных типов.

Flags

8-битовое резервное поле флагов, связанных с типом кадра. Семантика флагов определяется типом кадра, не имеющие семантики флаги не используются. Они **должны** игнорироваться при получении и сбрасываться (0x00) при отправке.

Reserved

Резервный бит без определённой семантики. Бит **должен** оставаться сброшенным (0x00) при передаче и игнорироваться при получении.

Stream Identifier

Идентификатор потока (параграф 5.1.1), указанный 31-битовым целым числом без знака. Значение 0x00 зарезервировано для кадров, связанных с соединением в целом, а не с отдельными потоками.

Структура и содержимое данных (payload) полностью зависят от типа кадра.

4.2. Размер кадра

Размер данных кадра ограничен максимумом, заданным получателем в SETTINGS_MAX_FRAME_SIZE. Этот параметр может принимать значение от 2^{14} (16384) до 2^{24}-1 (16777215), включительно.

Все реализации **должны** уметь принимать и минимально обрабатывать кадры размером до 214 октетов + 9 октетов заголовка кадра (параграф 4.1). Размер заголовка не учитывается при описании размеров кадров.

> Примечание. Для некоторых кадров, например, PING (параграф 6.7), имеются дополнительные ограничения размера данных.

Конечная точка **должна** передавать код ошибки FRAME_SIZE_ERROR, если размер кадра превышает SETTINGS_MAX_FRAME_SIZE, ограничение, заданное для типа кадра, или кадр слишком мал для включения обязательных данных. Ошибка размера в кадре, способном изменить состояние соединения в целом, **должна**

считаться ошибкой соединения (параграф 5.4.1). Это относится к любым кадрам с блоком полей (параграф 4.3) (HEADERS, PUSH_PROMISE, CONTINUATION), кадрам SETTINGS, а также кадрам с идентификатором потока 0.

Конечные точки не обязаны использовать все доступное в кадре пространство. Отзывчивость можно улучшить используя кадры размером меньше разрешённого максимума. Большой размер может приводить к задержкам при отправке чувствительных ко времени кадров (например, RST_STREAM, WINDOW_UPDATE, PRIORITY), блокировка которых передачей большого кадра может влиять на производительность.

4.3. Сжатие и декомпрессия раздела полей

Сжатие раздела полей представляет собой компрессию набора строк полей (параграф 5.2 в [HTTP]) для формирования блока полей. Декомпрессия раздела полей - это декодирование блока полей в набор строк полей. Детали сжатия и декомпрессии раздела полей HTTP/2 приведены в [COMPRESSION], где в силу исторических причин они называются сжатием и декомпрессией заголовков.

Каждый блок полей содержит все сжатые строки полей из одного раздела полей. Разделы полей включают также данные управления,связанные с сообщением в форме полей псевдозаголовка (параграф 8.3) с тем же форматом, что и строка поля.

Примечание. В RFC 7540 [RFC7540] применяется термин блок заголовка (header block), а не блок полей (field block).

Блоки полей содержат данные управления и разделы заголовков для запросов, откликов, обещанных запросов и выталкиваемых откликов (см. параграф 8.4). Все эти сообщения, за исключением промежуточных откликов и запросов, содержащихся в кадрах PUSH_PROMISE (параграф 6.6), могут включать блок полей с разделом трейлеров.

Раздел полей - это набор строк полей и каждая строка в блоке передаёт одно значение. Сериализованный блок полей после этого делится на последовательности октетов, называемыми фрагментами блока полей. Первый фрагмент блока передаётся в данных кадра HEADERS (параграф 6.2) или PUSH_PROMISE (параграф 6.6), за каждым из которых могут следовать кадры CONTINUATION (параграф 6.10) с последующими фрагментами блока.

Поле заголовка Cookie [COOKIE] особо обрабатывается отображением HTTP (см. параграф 8.2.3).

Принимающая конечная точка собирает блок полей путём конкатенации фрагментов и затем применяет декомпрессию для восстановления раздела полей. Полный раздел полей содержит одно из двух:

- один кадр HEADERS или PUSH_PROMISE с установленным флагом END_HEADERS;
- кадр HEADERS или PUSH_PROMISE со сброшенным флагом END_HEADERS и один или несколько кадров CONTINUATION, в последнем из которых установлен флаг END_HEADERS.

Каждый блок полей обрабатывается как дискретная единица. Блоки полей **должны** передаваться как непрерывная последовательность кадров без промежуточных кадров другого типа или из другого потока. В последнем кадре последовательности HEADERS или CONTINUATION, а также последовательности PUSH_PROMISE или CONTINUATION установлен флаг END_HEADERS. Это позволяет блоку полей быть эквивалентным одному кадру.

Фрагменты блока полей могут передаваться лишь как данные (payload) кадров HEADERS, PUSH_PROMISE или CONTINUATION, поскольку эти кадры переносят сведения, которые могут менять контекст сжатия, используемый получателем. Конечная точка, получающая кадры HEADERS, PUSH_PROMISE или CONTINUATION должна собрать блоки полей и выполнить декомпрессию, даже если кадры будут отброшены. Получатель **должен** прерывать соединение с ошибкой (параграф 5.4.1) типа COMPRESSION_ERROR, если блок полей не распакован (decompress).

Ошибка декодирования блока полей **должна** считаться ошибкой соединения (параграф 5.4.1) типа COMPRESSION_ERROR.

4.3.1. Состояние сжатия

Сжатие полей выполняется с поддержкой состояния. У каждой конечной точки имеется контекст кодирования и декодирования HPACK, которые используются для кодирования и декодирования всех блоков полей в соединении. В разделе 4 [COMPRESSION] определена динамическая таблица, которая является основным состоянием для каждого контекста. Динамическая таблица имеет максимальный размер, устанавливаемый декодером HPACK. Конечная точка передаёт размер, выбранный её контекстом декодера HPACK, с помощью установки SETTINGS_HEADER_TABLE_SIZE (см. параграф 6.5.2). При организации соединения для динамических таблиц кодера и декодера HPACK на обеих конечных точках устанавливается размер 4096 байтов - исходное значение SETTINGS_HEADER_TABLE_SIZE. Любая смена максимального значения, заданного SETTINGS_HEADER_TABLE_SIZE, вступает в силу, когда конечная точка подтверждает установки (параграф 6.5.3). Кодер HPACK в конечной точке может установить для динамической таблицы любой размер, вплоть до максимума, заданного декодером. Кодер HPACK объявляет размер динамической таблицы инструкцией Dynamic Table Size Update (параграф 6.3 в [COMPRESSION]).

Когда конечная точка подтверждает смену SETTINGS_HEADER_TABLE_SIZE, снижающую максимум до значения меньше текущего размера динамической таблицы, кодер HPACK в конечной точке **должен** начать следующий блок полей с инструкцией Dynamic Table Size Update, устанавливающей для динамической таблицы размер не больше сниженного максимума (см. параграф 4.2 в [COMPRESSION]). Конечная точка **должна** считать блок полей, следующий за подтверждением сокращения максимального размера динамической таблицы, ошибкой соединения (параграф 5.4.1) типа COMPRESSION_ERROR, если он не начинается с инструкции Dynamic Table Size Update.

Разработчикам следует учитывать, что снижение SETTINGS_HEADER_TABLE_SIZE не обеспечивает широкой совместимости. Применение предисловия соединения для снижения значения ниже исходных 4096 байтов иногда поддерживается лучше, но может вызывать отказы некоторых реализаций.

5. Потоки и мультиплексирование

Поток (stream) - это независимая двухсторонняя последовательность кадров, передаваемых между клиентом и сервером в соединении HTTP/2. Потоки имеют несколько важных характеристик.

- В одном соединении HTTP/2 может быть множество одновременных открытых потоков, при этом конечные точки могут чередовать кадры из нескольких потоков.
- Потоки могут создаваться и использоваться каждой точкой в одностороннем порядке или совместно.
- Поток может закрыть любая из конечных точек.
- Порядок передачи кадров является важным. Получатель обрабатывают кадры в порядке их приёма. В частности, порядок кадров HEADERS и DATA значим семантически.
- Потоки указываются целочисленными идентификаторами, которые назначаются конечной точкой, инициировавшей поток.

5.1. Состояния потока

Жизненный цикл потока показан на рисунке 2.

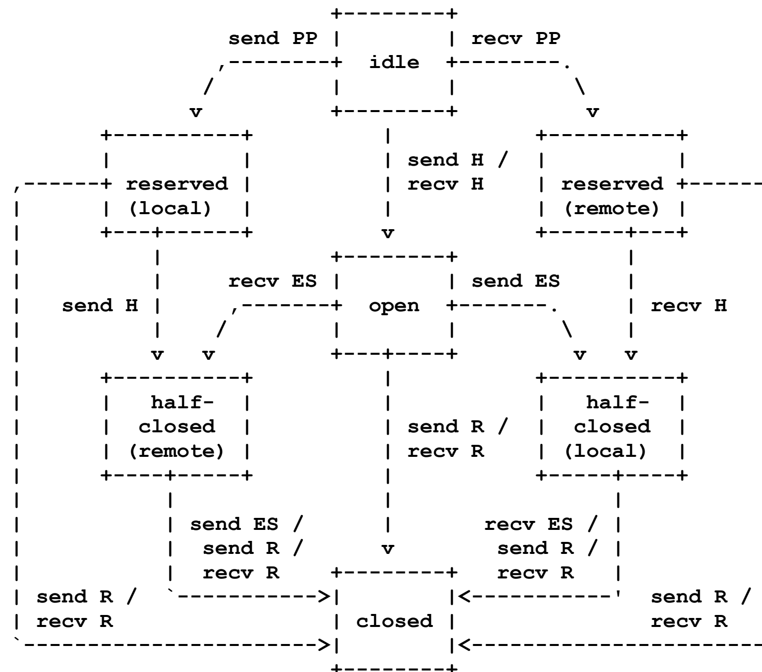

Рисунок 2.Состояния потока.

send
: Конечная точка передаёт этот кадр.

recv
: Конечная точка получает этот кадр.

H
: Кадр HEADERS (с подразумеваемыми кадрами CONTINUATION).

ES
: Флаг END_STREAM.

R
: Кадр RST_STREAM.

PP
: Кадр PUSH_PROMISE (с подразумеваемыми кадрами CONTINUATION), переходы состояний указаны для обещанного потока.

Отметим, что на рисунке показаны только смены состояния потока, а также кадры и флаги, влияющие на эти переходы. Кадры CONTINUATION не меняют состояния, они фактически являются частью HEADERS или PUSH_PROMISE, за которыми следуют. Применительно к смене состояний флаг END_STREAM обрабатывается как отдельное событие для кадра, где он установлен - кадр HEADERS с флагом END_STREAM может вызвать две смены состояния.

Обе конечные точки имеют субъективное представление о состоянии потока, которое может отличаться, когда кадры находятся в пути (in transit). Конечные точки не координируют создание потоков, они создаются любой из них в одностороннем порядке. Негативные последствия такого рассогласования ограничиваются состоянием closed после отправки RST_STREAM, когда кадры могут приниматься в течение некоторого времени после закрытия.

Возможные состояния потока показаны ниже

idle

Начальное состояние каждого потока, из которого возможны указанные ниже переходы.

- Передача кадра HEADERS в качестве клиента или приём HEADERS в качестве сервера вызывает переход потока в состояние open. Идентификатор потока выбирается в соответствии с параграфом 5.1.1. Тот же кадр HEADERS может вызвать немедленный переход потока в состояние half-closed.
- Отправка кадра PUSH_PROMISE в другой поток резервирует бездействующий (idle) поток для последующего использования, меняя его состояние на reserved (local). Кадры PUSH_PROMISE может передавать только сервер.
- Приём кадра PUSH_PROMISE в другом потоке резервирует бездействующий поток для последующего использования, переводя его в состояние reserved (remote). Кадры PUSH_PROMISE может получать только клиент.
- Отметим, что кадр PUSH_PROMISE не передаётся в бездействующий поток, а ссылается на новый зарезервированный поток в поле Promised Stream ID.

- Открытие потока с большим идентификатором вызывает незамедлительный переход потока в состояние closed (это не показано на рисунке).

Приём любого кадра, отличного от HEADERS и PRIORITY, в бездействующем потоке **должен** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Если поток инициирован сервером, как описано в параграфе 5.1.1, получение кадра HEADERS **должно** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

reserved (local)

Поток в состоянии reserved (local) - это поток, обещанный передачей кадра PUSH_PROMISE, который резервирует бездействующий поток, связывая его с открытым потоком, инициированным удаленным партнёром (параграф 8.4). Из этого состояния возможны лишь указанные ниже переходы.

- Конечная точка может передать кадр HEADERS, переводящий поток в состояние half-closed (remote).
- Любая из конечных точек может передать кадр RST_STREAM для перехода потока в состояние closed с освобождением выделенных для него резервов.

В этом состоянии конечной точке **недопустимо** передавать кадры кроме HEADERS, RST_STREAM, PRIORITY.

В этом состоянии **может** быть получен кадр PRIORITY или WINDOW_UPDATE. Получение любого кадра, отличного от RST_STREAM, PRIORITY и WINDOW_UPDATE потоком в этом состоянии **должно** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

reserved (remote)

Поток в состоянии reserved (remote) зарезервирован удаленным партнёром и возможны лишь указанные ниже переходы.

- Получение кадра HEADERS переводит поток в состояние half-closed (local).
- Любая из конечных точек может передать кадр RST_STREAM для перехода потока в состояние closed с освобождением выделенных для него резервов.

Конечной точке в этом состоянии **недопустимо** передавать кадры кроме RST_STREAM, WINDOW_UPDATE, PRIORITY. Получение любого кадра, отличного от HEADERS, RST_STREAM и PRIORITY потоком в этом состоянии **должно** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

open

Поток в состоянии open оба партнёра могут использовать для передачи кадров любого типа. В этом состоянии передающие партнёры соблюдают анонсированные ограничения управления потоком данных, анонсированные на уровне потока (параграф 5.2).

В этом состоянии любая из конечных точек может передать кадр с установленным флагом END_STREAM, вызывающий переход потока в состояние half-closed. Конечная точка, передавшая флаг END_STREAM, переводит поток в состояние half-closed (local), а получившая его - в состояние half-closed (remote).

Любая из конечных точек может передать кадр RST_STREAM для перевода потока в состояние closed.

half-closed (local)

Поток в состоянии half-closed (local) не может использоваться для передачи кадров кроме WINDOW_UPDATE, PRIORITY и RST_STREAM.

Поток переходит из этого состояния в closed при получении кадра с флагом END_STREAM или отправке любым из партнёров кадра RST_STREAM.

Конечная точка в этом состоянии может получить кадр любого типа. Необходимо получение кредита управления потоком данных с использованием кадров WINDOW_UPDATE для продолжения получения кадров с управлением потоком данных. В этом состоянии получатель может игнорировать кадры WINDOW_UPDATE, которые могут приходить в течение короткого интервала после отправки кадра с флагом END_STREAM.

В этом состоянии могут приниматься кадры PRIORITY.

half-closed (remote)

Поток в состоянии half-closed (remote) не может больше использоваться партнёрами для передачи кадров. В этом состоянии конечная точка больше не обязана поддерживать приёмное окно управления потоком данных.

Если конечная точка получает для потока в этом состоянии дополнительные кадры кроме WINDOW_UPDATE, PRIORITY, RST_STREAM, она **должна** отвечать ошибкой потока (параграф 5.4.2) типа STREAM_CLOSED.

Поток в состоянии half-closed (remote) может использоваться конечной точкой для передачи кадров любого типа. Конечная точка продолжает соблюдать анонсированные ограничения управления потоком данных (параграф 5.2).

Поток может перейти из этого состояния в closed, передав кадр с флагом END_STREAM или получив от партнёра кадр RST_STREAM.

closed

Состояние closed является терминальным. Поток переходит в это состояние после передачи и приёма кадра с установленным флагом END_STREAM. Поток переходит также в состояние closed после отправки или приёма конечной точкой кадра RST_STREAM.

Конечной точке **недопустимо** передавать в закрытый поток кадры, отличные от PRIORITY. Конечная точка **может** считать получение кадров любого другого типа в закрытом соединении ошибкой соединения (параграф 5.4.1) типа STREAM_CLOSED, за исключением описанных ниже случаев.

Конечная точка, передавшая кадр с флагом END_STREAM или кадр RST_STREAM, может получить от своего партнёра кадр WINDOW_UPDATE или RST_STREAM до того, как партнёр получит и обработает кадр, закрывающий поток. Конечная точка, передавшая кадр RST_STREAM в поток с состоянием open или half-closed (local), может получить кадр любого типа. Партнёр может передать или поместить эти кадры в очередь передачи до получения кадра RST_STREAM.

Конечная точка **должна** по минимуму обрабатывать и отбрасывать любые кадры, полученные в этом состоянии. Это означает обновление состояния сжатия заголовков в кадрах HEADERS и nd PUSH_PROMISE. Получение кадра PUSH_PROMISE также вызывает переход обещанного потока в состояние reserved (remote), даже при получении кадра PUSH_PROMISE в закрытом потоке. Содержимое кадров DATA учитывается в окне управления потоком данных соединения. Конечная точка может выполнять эту минимальную обработку для всех потоков, находящихся в состоянии closed. Конечные точки **могут** использовать другие сигналы для обнаружения приёма партнёром кадров, переводящих поток в состояние closed, и считать получения любого кадра кроме PRIORITY ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Конечные точки могут использовать для этого кадры, указывающие получение партнёром сигнала закрытия. Конечным точкам **не следует** применять для этого таймеры. Например, конечная точка, передавшая кадр SETTINGS после закрытия потока, может безопасно считать ошибкой приём в потоке кадра DATA после получения подтверждения установок. Могут также применяться кадры PING, получение данных или отклики на запросы в потоках, открытых после закрытия потока.

В отсутствие более конкретных правил реализациям **следует** считать получение кадра, не разрешённого явно в описании состояния, ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Кадры PRIORITY могут передаваться и приниматься в любом состоянии потока.

Правила этого раздела применимы лишь к кадрам, определенным в этом документе. Получение кадров с неизвестной семантикой не может считаться ошибкой, поскольку условия передачи и получения таких кадров тоже не известны (см. параграф 5.5). Пример смены состояния для обменов запрос-отклик в HTTP приведены в параграфе 8.8, примеры смены состояния для выталкивания (push) сервером даны в параграфах 8.4.1 и 8.4.2.

5.1.1. Идентификаторы потоков

Потоки указываются 31-битовым целым числом без знака. Инициированные клиентом потоки **должны** иметь нечётные идентификаторы потока, инициированные сервером - чётные. Поток с идентификатором 0x00 служит для сообщений управления соединением и этот идентификатор не может применяться для создания нового потока.

Идентификатор вновь созданного потока **должен** быть численно больше идентификаторов всех потоков, созданных или зарезервированных инициирующей конечной точкой. Это относится к потокам, создаваемым с помощью кадра HEADERS и резервируемым с помощью PUSH_PROMISE. Конечная точка, получившая неожиданный идентификатор потока, **должна** отвечать ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадр HEADERS переводит инициированный клиентом поток, указанный идентификатором в заголовке кадра, из состояния idle в состояние open. Кадр PUSH_PROMISE переводит инициированный сервером поток, указанный полем Promised Stream ID в данных (payload) кадра из состояния idle в состояние reserved (local) или reserved (remote). Когда поток выходит из состояния idle, все бездействующие потоки, которые могли быть созданы партнёром с меньшим значением идентификатора потока, незамедлительно переходят в состояние closed, т. е. конечная точка может пропускать идентификаторы потоков и пропущенные потоки будут сразу же закрываться.

Идентификаторы потоков не могут применяться неоднократно. В продолжительных соединениях запас идентификаторов потоков может заканчиваться. Клиент, не способный задать новый идентификатор потока, может организовать для него новое соединение, сервер в такой ситуации может передать кадр GOAWAY, чтобы заставить клиента создать новое соединение для новых потоков.

5.1.2. Одновременные потоки

Партнёр может ограничить число одновременно активных потоков с помощью параметра SETTINGS_MAX_CONCURRENT_STREAMS (параграф 6.5.2) в кадре SETTINGS. Этот параметр связан с конкретной конечной точкой и применяется лишь партнёром, получившим его. Т. е. клиент задаёт максимальное число одновременных потоков, которые может инициировать сервер, а сервер - число одновременных потоков, которые может инициировать клиент. Применительно в SETTINGS_MAX_CONCURRENT_STREAMS учитываются потоки в состоянии open и любом из состояний half-closed, а потоки в состояниях reserved не считаются.

Конечной точке **недопустимо** превышать заданный партнёром предел. Получив кадр HEADERS, ведущий к превышению установленного конечной точкой предельного числа одновременных потоков, эта точка **должна** считать кадр ошибкой потока (параграф 5.4.2) типа PROTOCOL_ERROR или REFUSED_STREAM. Выбор типа ошибки определяет возможность автоматического повторения (см. параграф 8.7).

Конечная точка, решившая снизить значение SETTINGS_MAX_CONCURRENT_STREAMS ниже текущего числа открытых потоков, может закрыть лишние потоки или дождаться их завершения.

5.2. Управление потоком данных

Мультиплексирование с помощью потоков вызывает борьбу за использование ресурсов соединения TCP, ведущую к блокировке потоков. Схема управления потоком данных позволяет предотвратить разрушительное влияние потоков одного соединения друг на друга. Управление потоком данных применяется для отдельных потоков и соединения в целом. В HTTP/2 управление потоками данных выполняется с помощью кадров WINDOW_UPDATE (параграф 6.9).

5.2.1. Принципы управления потоком данных

Управление потоком данных в потоке HTTP/2 нацелено на использование различных алгоритмов управления потоком данных без необходимости внесения изменений в протокол. Характеристики управления потоком данных HTTP/2 приведены ниже.

1. Управление потоком данных HTTP/2 связано с соединением и осуществляется между конечными точками одного интервала пересылки, а не на сквозном пути.
2. Управление осуществляется с помощью кадров WINDOW_UPDATE. Получатель анонсирует число октетов, которые он готов принять в потоке и соединении в целом. Эта схема основана на кредитах.
3. Управление потоком данных является направленным, а общий контроль выполняет получатель. Получатель **может** установить любой желаемый размер окна для каждого потока и соединения в целом. Отправитель **должен** соблюдать заданные получателем ограничения. Клиенты, серверы и посредники независимо анонсируют размер своего окна управления потоком данных в качестве получателя и соблюдают ограничения, заданные партнёром.
4. Начальный размер окна управления потоком данных для нового потока и всего соединения - 65535 октетов.
5. Тип кадра определяет применение к нему управления потоком данных. Из заданных в этом документе типов кадров управление потоком данных применяется только к кадрам DATA, а прочие не занимают места в анонсированном окне управления потоком данных. Это гарантирует, что важные кадры управления не будут блокироваться алгоритмом управления потоком данных.
6. Конечная точка может отключить своё управление потоком данных, но не может игнорировать сигналы управления потоком данных от партнёра.

7. HTTP/2 задаёт только формат и семантику кадров WINDOW_UPDATE (параграф 6.9). Этот документ не указывает, как получатель решает, когда отправить кадр, или какое значение передать, а также не задаёт способ отправки пакетов. Реализации могут выбрать любой алгоритм, соответствующий их потребностям.

Реализации отвечают за приоритизацию отправки запросов и откликов, выбирая способ предотвращения блокировки head-of-line для запросов и управляя созданием новых потоков. Алгоритмы решения этих задач могут взаимодействовать с любым алгоритмом управления потоком данных.

5.2.2. Приемлемое использование управления потоком данных

Управление потоком данных определяется для защиты конечных точек, работающих в условиях нехватки ресурсов. Например, прокси нужна общая память для многочисленных соединений и у него может быть медленное восходящее соединение и быстрое нисходящее. Управление потоком данных решает проблему, когда получатель не способен обработать данные в одном потоке, но хочет продолжать обработку других потоков в том же соединении.

Там, где не требуются такие возможности, можно анонсировать окно управления потоком данных максимального размера (2^{31}-1) и поддерживать его, передавая кадр WINDOW_UPDATE при получении любых данных. Это фактически отключает управление потоком данных для данного получателя. Отправитель всегда соблюдает заданное получателем окно управления потоком данных.

В системах с ограниченными ресурсами (например, памятью) управление потоком данных может применяться для ограничения расхода дефицитных ресурсов. Однако следует отметить, что это может приводить к неоптимальному использованию доступных сетевых ресурсов, если управление потоком данных включить без знания произведения пропускной способности и задержки (bandwidth*delay, см. [RFC7323]). Даже при полной осведомлённости о текущем значении bandwidth*delay реализация управления потоком данных может оказаться сложной. Конечные точки **должны** считывать и обрабатывать кадры HTTP/2 из приёмного буфера TCP, как только данные становятся доступными. Несвоевременное считывание может заводить в тупик, когда важные кадры, такие как WINDOW_UPDATE, не будут вовремя прочитаны и обработаны. Оперативное считывание кадров не открывает конечные точки для атак на истощения ресурсов, поскольку управление потоком данных в HTTP/2 ограничивает расход ресурсов.

5.2.3. Производительность управления потоком данных

Если конечная точка не может гарантировать, что у её партнёра всегда имеется место в окне управления потоком данных, превышающее значение bandwidth*delay для этого соединения, её пропускная способность на приёме будет ограничиваться управлением потоком данных HTTP/2, что приведёт к снижению производительности. Своевременная передача кадров WINDOW_UPDATE может повышать производительность.

Конечным точкам нужен баланс между повышением пропускной способности на приёме и предотвращением истощения ресурсов, поэтому следует внимательно изучить замечания в параграфе 10.5 при определении стратегии управления размером окна.

5.3. Приоритизация

В мультиплексируемых протоколах, таких как HTTP/2, приоритизация выделения потокам пропускной способности и вычислительных ресурсов может иметь решающее значение для достижения высокой производительности. Неудачная схема приоритизации может приводить в низкой производительности HTTP/2. При отсутствии распараллеливания на уровне TCP производительность может оказаться ниже, чем у HTTP/1.1.

Хорошая схема приоритизации выигрывает от применения сведений о контексте, таких как содержимое ресурсов, их взаимосвязь и способы использования этих ресурсов партнёром. В частности, клиенты могут иметь сведения о приоритете запросов, которые связаны с приоритизацией сервера. Предоставление клиентам сведений о приоритетах может повышать производительность.

5.3.1. Приоритеты в RFC 7540

В RFC 7540 определён богатый набор средств указания приоритета запросов. Однако система оказалась сложной и не получила широкого распространения.

Гибкая схема позволяла клиентам указывать приоритет самыми разными способами, при этом подходы не отличались согласованностью. Для серверов реализация поддержки этой схемы оказалась сложной. Реализация приоритетов в клиентах и серверах была несогласованной. Многие серверы игнорировали сигналы клиентов при определении приоритета обработки запросов.

Таким образом, сигнализация приоритета RFC 7540 [RFC7540] не достигла успеха.

5.3.2. Указание приоритета в этом документе

Это обновление HTTP/2 отменяет сигнализацию приоритетов, заданную в RFC 7540 [RFC7540]. Большая часть текста, связанного с указанием приоритета не включена в этот документ. Сохранены описания полей кадров и некоторая обязательная обработка для обеспечения совместимости реализаций этой спецификации с реализациями, применяющими сигналы приоритета из RFC 7540.

Подробное описание схемы приоритетов RFC 7540 дано в параграфе 5.3 [RFC7540].

Сведения о приоритете нужны во многих случаях для достижения высокой производительности. Там, где сведения о приоритете важны, конечным точкам рекомендуется использовать альтернативную схему, например, [HTTP-PRIORITY].

Хотя сигнализация приоритета из RFC 7540 не получила широкого распространения, предоставляемые ею сведений могут быть полезны при отсутствии лучших сигналов. Конечные точки, получающие сигналы приоритета в кадрах HEADERS или PRIORITY могут выиграть от использования этих сведений. В частности, реализации, воспринимающие такие сигналы, не выиграют от их запрета при отсутствии альтернативы.

Серверам **следует** использовать другие данные контекста при определении приоритета запросов в отсутствие каких-либо сигналов. Серверы **могут** считать полное отсутствие сигналов приоритета указанием на то, что клиент не

поддерживает приоритизацию. Принятые по умолчанию значения (параграф 5.3.5 в [RFC7540]) не обеспечивают высокой производительности в большинстве случаев и их использованием вряд ли имеет смысл.

5.4. Обработка ошибок

Кадрирование HTTP/2 допускает два класса ошибок:

- ошибки, делающие соединение непригодны целиком (ошибки соединения);
- ошибки в отдельном потоке (ошибки потока).

Список кодов ошибок приведён в разделе 7.

Конечная точка может сталкиваться с кадрами, вызывающими несколько ошибок. Реализации **могут** обнаруживать несколько ошибок в процессе обработки, но им **следует** сообщать в результате не более одной ошибки потока и одной ошибки соединения. Первая ошибка потока, указанная для данного потока, предотвращает передачу других ошибок этого потока. Протокол допускает несколько кадров GOAWAY, однако конечной точке **следует** сообщать лишь об одном типе ошибки соединения, если не происходит ошибки в процессе аккуратного разрыва соединения. Если такая ошибка возникает, конечная точка **может** передать дополнительный кадр GOAWAY с новым кодом ошибки в дополнение к предшествующему GOAWAY с NO_ERROR.

Если конечная точка обнаруживает несколько разных ошибок, она **может** сообщить о любой из них. Если кадр вызывает ошибку соединения, эта ошибка **должна** указываться. Кроме того, конечная точка **может** использовать любой подходящий код при обнаружении ошибки, а базовые коды (такие как PROTOCOL_ERROR и INTERNAL_ERROR) всегда можно применять вместо более конкретных кодов.

5.4.1. Обработка ошибок соединения

Ошибкой соединения считается любая ошибка, препятствующая дальнейшей обработке уровня кадров или повреждающая состояние соединения.

Конечной точке, столкнувшейся с ошибкой соединения, **следует** сначала передать кадр GOAWAY (параграф 6.8) с идентификатором последнего потока, успешно принятого от партнёра. Кадр GOAWAY включает код ошибки (раздел 7), указывающий причину разрыва соединения. После отправки GOAWAY, конечная точка **должна** закрыть соединение TCP.

Возможно, что кадр GOAWAY не будет получен партнёром. В случае ошибки соединения кадр GOAWAY является лишь попыткой информировать партнёра о прерывании соединения.

Конечная точка может завершить соединение в любой момент. В частности, она **может** счесть ошибку потока ошибкой соединения. Конечным точкам **следует** передавать кадр GOAWAY при завершении соединения, если это возможно.

5.4.2. Обработка ошибок потока

Ошибкой потока считается ошибка, связанная с конкретным потоком и не влияющая на обработку других потоков.

Конечная точка, столкнувшаяся с ошибкой потока, передаёт кадр RST_STREAM (параграф 6.4) с идентификатором потока, где произошла ошибка, включающий код, который указывает тип ошибки.

RST_STREAM является последним кадром, передаваемым конечной точкой в этот поток, и эта точка **должна** быть готова принять любые кадры, которые были переданы или помещены удаленным партнёром в очередь передачи. Эти кадры можно игнорировать, за исключением случаев, когда они меняют состояние соединения, например, статус, поддерживаемый для сжатия полей заголовка (параграф 4.3) или управление потоком данных.

Обычно конечной точке **не следует** передавать более одного пакета RST_STREAM для любого потока. Однако **можно** передать дополнительные кадры RST_STREAM, если конечная точка получает кадры через закрытый поток в течение времени больше интервала кругового обхода. Это разрешено для работы с некорректными реализациями.

Для предотвращения петель конечной точке **недопустимо** передавать кадр RST_STREAM в ответ на RST_STREAM.

5.4.3. Прерывание соединения

Если соединение TCP закрывается или сбрасывается при наличии в нем потоков в состоянии open или half-closed, затронутые этим потоки не могут быть перезапущены автоматически (см. параграф 8.7).

5.5. Расширение HTTP/2

HTTP/2 допускает расширение протокола. В рамках заданных в этом параграфе ограничений расширения протокола можно применять для предоставления дополнительных услуг и изменения любых аспектов протокола. Расширения действуют только в рамках одного соединения HTTP/2. Это относится к элементам протокола, заданным в этом документе и не влияет на имеющиеся опции расширения HTTP, такие как определение новых методов, кодов статуса и полей (см. раздел 16 в [HTTP]).

Расширениям разрешено применять новые типы кадров (параграф 4.1), установки (параграф 6.5) и коды ошибок (раздел 7). Реестры для управления этими точками расширения определены в разделе 11 [RFC7540].

Реализации **должны** игнорировать любые неизвестные или неподдерживаемые значения во всех расширяемых элементах протокола, а также отбрасывать кадры неизвестных или неподдерживаемых типов. Это значит, что точки расширения можно безопасно использовать в расширениях без предварительного согласования или переговоров. Однако кадры расширения в середине блока полей (параграф 4.3) не разрешены и **должны** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Расширениям **следует** избегать изменения элементов протокола, заданных в этом документе, и элементов, не имеющих механизма расширения. Это включает изменение схем кадров, добавления или изменения способов компоновки кадров в сообщения HTTP (параграф 8.1), определение полей псевдозаголовков, а также изменения

любых элементов протокола, которые соответствующая спецификации конечная точка может счесть ошибкой соединения (параграф 5.4.1).

Расширение, меняющее имеющиеся элементы протокола или состояние, **должно** быть согласовано до использования. Например, расширение, меняющее схему кадров HEADERS, не может применяться, пока партнёр не укажет его приемлемость. В этом случае может также потребоваться согласование момента вступления новой схемы в силу. Например, трактовка отличных от DATA кадров как подверженных управлению потоком данных требует изменения их семантику, которую должны понимать обе конечные точки, поэтому может быть принята только после согласования.

Этот документ не задаёт конкретного метода согласования использования расширений, но отмечает возможность использования настроек (параграф 6.5.2) для решения этой задачи. Если оба партнёра установили значение, указывающее готовность применять расширение, это расширение можно использовать. Если для согласования расширения применяется настройка, исходное значение **должно** быть задано так, чтобы расширение было отключено.

6. Определения кадров

Эта спецификация задаёт ряд типов кадров, идентифицируемых уникальным 8-битовым кодом. Каждый тип кадра служит для своих целей при организации и поддержке соединения в целом или отдельных потоков.

Передача определённого типа кадров может менять состояние соединения. Если конечные точки не могут синхронизировать состояние соединения, последующие взаимодействия через это соединение будут невозможны. Поэтому важно, чтобы конечные точки имели общее представление о влиянии использования того или иного кадра на состояние соединения.

6.1. DATA

Кадры DATA (type=0x00) переносят произвольные последовательности октетов переменного размера, связанные с потоком. Например, один или несколько кадров DATA могут служить для передачи содержимого запроса или отклика HTTP. Кадры DATA могут включать заполнение, применяемое для сокрытия размера сообщений. Это защитное свойство описано в параграфе 10.7.

```
DATA Frame {
  Length (24),
  Type (8) = 0x00,

  Unused Flags (4),
  PADDED Flag (1),
  Unused Flags (2),
  END_STREAM Flag (1),

  Reserved (1),
  Stream Identifier (31),

  [Pad Length (8)],
  Data (..),
  Padding (..2040),
}
```

Рисунок 3.Формат кадра DATA.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4. Поля данных кадра DATA приведены ниже.

Pad Length

8-битовое значение размера заполнения кадра в октетах, присутствующее лишь при наличии флага PADDED.

Data

Данные приложения. Объем данных - это часть содержимого кадра (payload) за вычетом размера других полей.

Padding

Октеты заполнения, не имеющие семантического смысла. При передаче в октетах заполнения **должно** указываться значение 0. Получатель не обязан проверять заполнение, но **может** считать отличное от 0 заполнение ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Флаги кадров DATA указаны ниже.

PADDED (0x08)

Установленный флаг указывает присутствие поля Pad Length и октетов заполнения.

END_STREAM (0x01)

Установленный флаг указывает, что это последний кадр, который конечная точка передаёт в указанный поток. Наличие флага указывает, что поток переходит в состояние half-closed или closed (параграф 5.1).

Примечание. Конечная точка, узнавшая о закрытии потока после отправки всех данных, может закрыть этот поток отправкой кадра DATA[1] с пустым полем Data и установленным флагом END_STREAM. Это возможно лишь в том случае, когда конечная точка не передаёт трейлеров, поскольку в этом случае флаг END_STREAM устанавливается в кадре HEADERS (см. параграф 8.1).

Кадры DATA должны связываться с потоком. При получении кадра DATA с Stream Identifier = 0x00 получатель **должен** возвращать ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

К кадрам DATA применяется управление потоком данных и они могут передаваться лишь в потоках со статусом open и half-closed (remote). Управление потоком данных применяется ко всему содержимому (payload) кадра, включая поля Pad Length и Padding (при наличии). При получении кадра DATA в потоке с состоянием, отличным от open и half-closed (local), получатель **должен** возвращать ошибку потока (параграф 5.4.2) типа STREAM_CLOSED.

Число октетов заполнения определяется значением поля Pad Length. Если размер заполнения не меньше размера содержимого кадра, получатель **должен** считать это ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Примечание. Размер кадра может увеличиваться на 1 октет включением поля Pad Length со значением 0.

[1]В оригинале ошибочно сказано STREAM, см. https://www.rfc-editor.org/errata/eid7013. *Прим. перев.*

6.2. HEADERS

Кадр HEADERS (type=0x01) служит для открытия потока (параграф 5.1), а также передаёт фрагмент блока полей. Несмотря на имя, кадры HEADERS могут содержать разделы заголовков и трейлеров. Кадры могут передаваться в потоках со статусом idle, reserved (local), open, half-closed (remote).

```
HEADERS Frame {
  Length (24),
  Type (8) = 0x01,

  Unused Flags (2),
  PRIORITY Flag (1),
  Unused Flag (1),
  PADDED Flag (1),
  END_HEADERS Flag (1),
  Unused Flag (1),
  END_STREAM Flag (1),

  Reserved (1),
  Stream Identifier (31),

  [Pad Length (8)],
  [Exclusive (1)],
  [Stream Dependency (31)],
  [Weight (8)],
  Field Block Fragment (..),
  Padding (..2040),
}
```

Рисунок 4.Формат кадра HEADERS.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4. Поля содержимого кадра HEADERS приведены ниже.

Pad Length
: 8-битовое значение размера заполнения кадра в октетах, присутствующее лишь при наличии флага PADDED.

Exclusive
: Флаг, присутствующий лишь при наличии флага PRIORITY. Сигнализация приоритета в кадрах HEADERS отменена (см. параграф 5.3.2).

Stream Dependency
: 31-битовый идентификатор потока, используемый лишь при наличии флага PRIORITY.

Weight
: 8-битовое целое число без знака, присутствующее лишь при наличии флага PRIORITY.

Field Block Fragment
: Фрагмент блока полей (параграф 4.3).

Padding
: Октеты заполнения, не имеющие семантического смысла. При передаче в октетах заполнения **должно** указываться значение 0. Получатель не обязан проверять заполнение, но **может** считать отличное от 0 заполнение ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Флаги кадров HEADERS приведены ниже.

PRIORITY (0x20)
: Установленный флаг указывает наличие полей Exclusive, Stream Dependency, Weight.

PADDED (0x08)
: Установленный флаг указывает присутствие поля Pad Length и октетов заполнения.

END_HEADERS (0x04)
: Установленный флаг указывает наличие в кадре полного блока полей (параграф 4.3) и отсутствие последующих кадров CONTINUATION. За кадром HEADERS без флага END_HEADERS в том же потоке **должен** следовать кадр CONTINUATION. Получатель **должен** считать получение кадра иного типа или приём кадра продолжения в другом потоке ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

END_STREAM (0x01)
: Установленный флаг указывает, что блок полей (параграф 4.3) является последним, передаваемым конечной точкой для указанного потока. Кадр HEADERS с флагом END_STREAM говорит о завершении потока, однако за ним в том же потоке может следовать кадр CONTINUATION. Логически CONTINUATION служит частью HEADERS.

Содержимым (payload) кадра HEADERS сявляется фрагмент блока полей (параграф 4.3). Если блок не помещается в кадр HEADERS, он продолжается в кадре CONTINUATION (параграф 6.10).

Кадры HEADERS должны связываться с потоком. При получении кадра HEADERS с Stream Identifier = 0x00 получатель **должен** возвращать ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадр HEADERS меняет состояние соединения, как описано в параграфе 4.3.

Число октетов заполнения определяется значением поля Pad Length. Если размер заполнения не меньше размера содержимого кадра, получатель **должен** считать это ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Примечание. Размер кадра может увеличиваться на 1 октет включением поля Pad Length со значением 0.

6.3. PRIORITY

Кадры PRIORITY (type=0x02) устарели (см. параграф 5.3.2). Такой кадр может передаваться в потоке с любым состоянием, включая idle и closed.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4. Поля содержимого кадра PRIORITY приведены ниже.

```
PRIORITY Frame {
  Length (24) = 0x05,
  Type (8) = 0x02,

  Unused Flags (8),

  Reserved (1),
  Stream Identifier (31),

  Exclusive (1),
  Stream Dependency (31),
  Weight (8),
}
```

Рисунок 5.Формат кадра PRIORITY.

Exclusive
: Однобитовый флаг.

Stream Dependency
: 31-битовый идентификатор потока.

Weight
: 8-битовое целое число без знака.

Для кадров PRIORITY не задано флагов.

Кадр PRIORITY всегда идентифицирует поток. При получении кадра PRIORITY с Stream Identifier = 0x00 получатель **должен** возвращать ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Приём или передача PRIORITY не влияет на состояния потоков (параграф 5.1). Кадр PRIORITY можно передавать в поток с любым статусом, включая idle и closed. Кадр PRIORITY не может передаваться между парой кадров, переносящих один блок полей (параграф 4.3).

Кадр PRIORITY с размером, отличным от 5 октетов, **должен** считаться ошибкой потока (параграф 5.4.2) типа FRAME_SIZE_ERROR.

6.4. RST_STREAM

Кадр RST_STREAM (type=0x03) позволяет сразу же прервать поток и передаётся для запроса прерывания потока или указания обнаруженной ошибки.

```
RST_STREAM Frame {
  Length (24) = 0x04,
  Type (8) = 0x03,

  Unused Flags (8),

  Reserved (1),
  Stream Identifier (31),

  Error Code (32),
}
```

Рисунок 6.Формат кадра RST_STREAM.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4. Кроме того, кадр RST_STREAM содержит одно 32-битовое целочисленное значение без знака, указывающее код ошибки (раздел 7) с причиной прерывания потока.

Для кадров RST_STREAM не задано флагов.

Кадр RST_STREAM полностью прерывает указанный поток и переводит его в состояние closed. После приёма в потоке кадра RST_STREAM получателю **недопустимо** передавать в этот поток кадры, за исключением PRIORITY. Однако после отправки RST_STREAM конечная точка **должна** быть готова принять в этом потоке кадры, которые партнёр мог передать до прибытия к нему кадра RST_STREAM.

Кадр RST_STREAM **должен** быть связан с потоком. При получении кадра RST_STREAM с Stream Identifier = 0x00 получатель **должен** возвращать ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадры RST_STREAM **недопустимо** передавать в потоки с состоянием idle. При получении кадра RST_STREAM, указывающего бездействующий поток, получатель **должен** возвращать ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадр RST_STREAM с размером, отличным от 4 октетов, **должен** считаться ошибкой соединения (параграф 5.4.1) типа FRAME_SIZE_ERROR.

6.5. SETTINGS

Кадр SETTINGS (type=0x04) передаёт параметры конфигурацииЮ влияющие на взаимодействие конечных точек, такие как предпочтения или ограничения для поведения партнёра, а также для подтверждения приёма таких установок. Параметры из кадра SETTINGS называются установками, настройками или параметрами (setting). Настройки не согласуются, они описывают характеристики передающего партнёра и используются принимающим. Кадлый из партнёров может анонсировать своё значение настройки. Например, клиент может задать большое начальное окно управления потоком данных, а север - меньшее для экономии ресурсов.

Кадры SETTINGS **должны** передавать обе конечные точки при старте соединения и эти кадры **могут** передаваться в любой момент в течение срока работы соединения каждой из конечных точек. Реализации **должны** поддерживать все настройки, заданные в этой спецификации. Каждый параметр из кадра SETTINGS заменяет имеющееся значение. Настройки обрабатываются в порядке их указания и получатель SETTINGS не обязан поддерживать какое-либо

состояние сверх текущего значения параметра. Поэтому значение параметра в SETTINGS является последним значением у получателя.

Кадры SETTINGS подтверждаются получившим их партнёром. Для этого в SETTINGS применяется флаг ACK.

ACK (0x01)

Установка флага указывает, что этот кадр подтверждает получение и применение кадра SETTINGS от партнёра. При установленном флаге содержимое (payload) кадра SETTINGS **должно** быть пустым. Получение кадра SETTINGS с флагом ACK и ненулевым размером **должно** считаться ошибкой соединения (параграф 5.4.1) типа FRAME_SIZE_ERROR (см. параграф 6.5.3).

Кадр SETTINGS всегда применяется к соединению, а не к отдельному кадру, и идентификатор потока в нем **должен** иметь значение 0x00. Если конечная точка получает кадр SETTINGS с другим значением fStream Identifier, она **должна** возвращать ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадр SETTINGS влияет на статус соединения. Некорректный или неполный кадр SETTINGS **должен** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадр SETTINGS с размером, отличным от 6 октетов, **должен** считаться ошибкой соединения (параграф 5.4.1) типа FRAME_SIZE_ERROR.

6.5.1. Формат SETTINGS

Данные (payload) кадра SETTINGS могут содержать установки, каждая из которых включает 16-битовый идентификатор параметра и его 32-битовое целочисленное значение без знака.

```
SETTINGS Frame {
  Length (24),
  Type (8) = 0x04,

  Unused Flags (7),
  ACK Flag (1),

  Reserved (1),
  Stream Identifier (31) = 0,

  Setting (48) ...,
}

Setting {
  Identifier (16),
  Value (32),
}
```

Рисунок 7.Формат кадра SETTINGS.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4. Данные кадра SETTINGS могут включать любое число полей Setting, описанных ниже.

Identifier

16-битовый идентификатор параметра (параграф 6.5.2).

Value

32-битовое значение параметра.

6.5.2. Заданные настройки

Ниже приведены определённые этим документом настройки (параметры).

SETTINGS_HEADER_TABLE_SIZE (0x01)

Позволяет отправителю информировать удалённую точку о максимальном размере (в октетах) таблицы сжатия, применяемой для декодирования блоков полей. Кодер может выбрать любой размер, не превышающий это значение, используя сигналы, специфичные для формата сжатия в блоке полей (см. [COMPRESSION]). Начальное значение - 4096 октетов.

SETTINGS_ENABLE_PUSH (0x02)

Этот флаг может применяться для включения и отключения выталкивания с сервера (push). Серверу **недопустимо** передавать кадр PUSH_PROMISE, если он получил этот параметр со значением 0 (параграф 8.4). Клиент, передавший значение 0 и получивший подтверждение, **должен** считать получение кадра PUSH_PROMISE ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Исходно устанавливается SETTINGS_ENABLE_PUSH = 1. Для клиента это указывает его желание принимать кадры PUSH_PROMISE, а для сервера не имеет значения и эквивалентно 0. Любое значение кроме 0 и 1 **должно** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Серверу **недопустимо** явно устанавливать для этого параметра значение 1 и он **может** опустить эту настройку при отправке кадра SETTINGS, но если сервер включает этот параметр он **должен** иметь значение 0. Клиент **должен** считать получение кадра PUSH_PROMISE с SETTINGS_ENABLE_PUSH = 1 ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

SETTINGS_MAX_CONCURRENT_STREAMS (0x03)

Указывает максимальное число одновременных потоков, разрешаемых отправителем. Это ограничение однонаправленное и применяется для числа пакетов, которые отправитель позволяет создать получателю. Исходно ограничение не задано и рекомендуется устанавливать значение не меньше 100, чтобы не ограничивать распараллеливание без необходимости.

Значение SETTINGS_MAX_CONCURRENT_STREAMS =0 **не следует** считать особым, оно препятствует созданию новых потоков, однако это может происходить и при достижении предела. Серверу **следует** устанавливать значение 0 лишь на короткое время. Если сервер не хочет воспринимать запросы, лучше закрыть соединение.

SETTINGS_INITIAL_WINDOW_SIZE (0x04)

Указывает исходный размер (в октетах) окна управления потоком данных у отправителя на уровне потока. Исходное значение - 2^{16}-1 (65535) октетов.

Этот параметр влияет на размер окна во всех потоках (см. параграф 6.9.2).

Значения выше максимального размера окна управления потоком данных 2^{31}-1 **должны** считаться ошибкой соединения (параграф 5.4.1) типа FLOW_CONTROL_ERROR.

SETTINGS_MAX_FRAME_SIZE (0x05)

Указывает наибольший размер (в октетах) данных кадра (payload) которые отправитель готов получать.

Исходное значение - 214 (16384) октетов. Анонсируемое конечной точной значение должно быть в интервале между исходным значением и максимальным размером кадра 2^{24}-1 (16777215), включительно. Иные значения **должны** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

SETTINGS_MAX_HEADER_LIST_SIZE (0x06)

Информирует партнёра о максимальном размере (в октетах) раздела полей, который отправитель готов воспринимать. Значение учитывает размеры несжатых полей, включая имя и значение, а также издержки в 32 октета на каждую строку поля.

Для любого данного запроса **может** устанавливаться предел ниже указанного этим значением. Исходно размер не ограничивается.

Конечная точка, получившая кадр SETTINGS с неизвестным или неподдерживаемым идентификатором, **должна** игнорировать соответствующий параметр.

6.5.3. Синхронизация установок

Для большинства значений в SETTINGS требуется или полезно понимание того, что партнёр получил и применил изменённые значения параметров. Для обеспечения таких моментов синхронизации получатель кадра SETTINGS со сброшенным флагом ACK **должен** применить обновлённые установки как можно быстрее. Кадры SETTINGS подтверждаются в порядке их получения.

Значения из кадра SETTINGS **должны** обрабатываться в порядке их указания, а неподдерживаемые значения **должны** игнорироваться. По завершении обработки всех значений получатель **должен** сразу же передать кадр SETTINGS с установленным флагом ACK. При получении такого кадра отправитель изменённых настроек **может** полагаться на то, что применены значения из самого старого неподтвержденного до этого кадра SETTINGS.

Если отправитель кадра SETTINGS не получил подтверждения в течение разумного интервала времени, он может выдать ошибку соединения (параграф 5.4.1) типа SETTINGS_TIMEOUT. При установке тайм-аута нужно учитывать время обработки партнёром, поскольку тайм-аут, учитывающий лишь время передачи по сети может вызывать ложные ошибки.

6.6. PUSH_PROMISE

Кадр PUSH_PROMISE (type=0x05) служит для предварительного уведомления партнёра о потоках, которые отправитель намерен инициировать. PUSH_PROMISE включает 31-битовый идентификатор потока, который конечная точка планирует создать, а также раздел полей с дополнительным контекстом для потока. Подробное описание использования кадров PUSH_PROMISE дано в параграфе 8.4.

```
PUSH_PROMISE Frame {
  Length (24),
  Type (8) = 0x05,

  Unused Flags (4),
  PADDED Flag (1),
  END_HEADERS Flag (1),
  Unused Flags (2),

  Reserved (1),
  Stream Identifier (31),

  [Pad Length (8)],
  Reserved (1),
  Promised Stream ID (31),
  Field Block Fragment (..),
  Padding (..2040),
}
```

Рисунок 8.Формат кадра PUSH_PROMISE.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4. Содержимое кадра PUSH_PROMISE показано ниже.

Pad Length

8-битовое значение размера заполнения кадра в октетах, присутствующее лишь при наличии флага PADDED.

Promised Stream ID

31-битовое целое число, указывающее поток, резервируемый кадром PUSH_PROMISE. Идентификатор обещанного потока **должен** иметь значение, подходящее для следующего потока, передаваемого отправителем (см. параграф 5.1.1).

Field Block Fragment

Фрагмент блока полей (параграф 4.3) с данными управления запросом и разделом заголовков.

Padding

Октеты заполнения, не имеющие семантического смысла. При передаче в октетах заполнения **должно** указываться значение 0. Получатель не обязан проверять заполнение, но **может** считать отличное от 0 заполнение ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Ниже указаны флаги кадров PUSH_PROMISE.

PADDED (0x08)

Установленный флаг указывает присутствие поля Pad Length и октетов заполнения.

END_HEADERS (0x04)

Установленный флаг указывает наличие в кадре полного блока полей (параграф 4.3) и отсутствие последующих кадров CONTINUATION. За кадром PUSH_PROMISE без флага END_HEADERS в том же потоке **должен** следовать кадр CONTINUATION. Получатель **должен** считать получение кадра иного типа или приём кадра продолжения в другом потоке ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадры PUSH_PROMISE **должны** передаваться только через инициированный партнёром поток, находящийся в состоянии open или half-closed (remote). Идентификатор потока в PUSH_PROMISE указывает связанный с кадром поток. Если Stream Identifier = 0x00, получатель кадра **должен** отвечать ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Обещанные потоки не требуется применять в порядке их предложения и кадры PUSH_PROMISE лишь резервируют потоки для использования в будущем.

Кадр PUSH_PROMISE **недопустимо** передавать, если у партнёра установлено SETTINGS_ENABLE_PUSH = 0. Конечная точка с такой установкой, получавшая подтверждение, **должна** считать приём кадра PUSH_PROMISE ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Получатель PUSH_PROMISE может отвергнуть предложенный поток, возвращая кадр RST_STREAM с указанием идентификатора обещанного потока отправителю PUSH_PROMISE.

Кадр PUSH_PROMISE меняет состояние соединения двумя способами. Во-первых, включение блока полей (параграф 4.3) может менять состояние, поддерживаемое для сжатия полей. Во-вторых, PUSH_PROMISE резервирует поток для последующего использования, заставляя обещанный поток принимать состояние reserved (local) или reserved (remote). Отправителю **недопустимо** передавать PUSH_PROMISE в поток, не имеющий состояния open или half-closed (remote), он **должен** убедиться, что обещанный поток имеет подходящий идентификатор (параграф 5.1.1), т. е. обещанный поток **должен** иметь статус idle.

Поскольку PUSH_PROMISE резервирует поток, игнорирование кадра делает состояние этого потока неопределённым. Получатель PUSH_PROMISE в потоке, не находящемся в состоянии open или half-closed (local), **должен** считать это ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Однако конечная точка, передавшая RST_STREAM в соответствующий поток, **должна** обрабатывать кадры PUSH_PROMISE, которые могли быть созданы до получения и обработки RST_STREAM.

Получатель **должен** считать приём PUSH_PROMISE и непригодным идентификатором потока (параграф 5.1.1) ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Непригодным считается идентификатор потока, который не имеет в данный момент состояния idle.

Число октетов заполнения определяется значением поля Pad Length. Если размер заполнения не меньше размера содержимого кадра, получатель **должен** считать это ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Примечание. Размер кадра может увеличиваться на 1 октет включением поля Pad Length со значением 0.

6.7. PING

Кадры PING (type=0x06) служат механизмом измерения минимального времени кругового обхода от отправителя, а также проверки функционирования бездействующих соединений. PING может передавать любая конечная точка.

```
PING Frame {
  Length (24) = 0x08,
  Type (8) = 0x06,

  Unused Flags (7),
  ACK Flag (1),

  Reserved (1),
  Stream Identifier (31) = 0,

  Opaque Data (64),
}
```

Рисунок 9.Формат кадра PING.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4.

В дополнение к заголовку кадры PING **должны** включать 8 октетов неинтерпретируемых (opaque) данных в содержимом кадра (payload). Отправитель может указать любое значение и применять его по своему усмотрению.

Получатель кадра PING со сброшенным флагом ACK **должен** передать в ответ PING с установленным флагом ACK и идентичным содержимым. Откликам PING **следует** предоставлять приоритет перед кадрами иных типов.

Ниже описан флаг кадров PING.

ACK (0x01)

Установка флага указывает, что этот кадр является откликом на PING. Конечная точка **должна** устанавливать этот флаг в откликах PING. На кадры PING с этим флагом **недопустимо** отвечать.

Кадры PING не связаны с каким-либо отдельным потоком. При получении PING с отличным от 0 значением Stream Identifier получатель **должен** отвечать ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадр PING с размером, отличным от 8 октетов, **должен** считаться ошибкой соединения (параграф 5.4.1) типа FRAME_SIZE_ERROR.

6.8. GOAWAY

Кадры GOAWAY (type=0x07) применяются для инициирования закрытия (shutdown) соединения или сигнализации о серьёзных ошибках. GOAWAY позволяет конечной точке аккуратно прекратить восприятие новых потоков, сохраняя обработку созданных ранее. Это позволяет выполнять административные операции, такие как обслуживание сервера.

Между конечной точкой, запускающей новые потоки, и её партнёром, передающим кадр GOAWAY возникает соперничество. Для решения проблемы в кадр GOAWAY включается идентификатор потока, который был или может быть обработан передающей кадр конечной точкой в этом соединении. Например при передаче кадра GOAWAY сервером указываться будет созданный клиентом поток с наибольшим номером.

После передачи кадра GOAWAY отправитель игнорирует кадры от партнёра, если идентификатор потока в них превышает указанный в кадре. Получателю кадра GOAWAY **недопустимо** открывать в соединении дополнительные потоки, но можно создавать для таких потоков новое соединение. Если получатель GOAWAY передал данные в потоке с идентификатором больше указанного в кадре GOAWAY, эти кадры не будут обработаны. Получатель кадра GOAWAY может считать такие потоки не созданными, что позволяет повторить их в новом соединении.

Конечным точкам **следует** всегда передавать кадр GOAWAY перед закрытием соединения, чтобы удалённый партнёр мог знать, был ли поток частично обработан. Например, если клиент HTTP отправил POST в то же время, когда сервер закрывал соединение, клиент не будет знать, обработан ли запрос POST сервером, если тот не передаст кадр GOAWAY для указания потоков, которые он мог обработать.

Конечная точка может закрыть соединение без отправки кадра GOAWAY, если партнёр ведёт себя некорректно.

Кадр GOAWAY может не предшествовать немедленному закрытию соединения и получателю GOAWAY, который больше не использует соединение, **следует** передать кадр GOAWAY перед завершением соединения.

```
GOAWAY Frame {
  Length (24),
  Type (8) = 0x07,

  Unused Flags (8),

  Reserved (1),
  Stream Identifier (31) = 0,

  Reserved (1),
  Last-Stream-ID (31),
  Error Code (32),
  Additional Debug Data (..),
}
```

Рисунок 10.Формат кадра GOAWAY.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4.

Для кадров GOAWAY не задано флагов.

Кадр GOAWAY применяется к соединению, а не к отдельному потоку. Конечная точка **должна** считать кадр GOAWAY с ненулевым идентификатором потока ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Идентификатор последнего потока в кадре GOAWAY содержит наибольший номер потока, для которого отправитель GOAWAY выполнил или может выполнить какие-либо действия. Все потоки до указанного, включительно, могли быть как-то обработаны. Значение 0 для последнего идентификатора указывает, что потоки не обрабатывались.

> Примечание. В этом контексте обработка означает, что некоторые данные из потока переданы на вышележащий уровень программе, которая могла выполнить какие-то действия.

Если соединение прерывается без кадра GOAWAY, последним идентификатором потока фактически становится максимальный возможный номер.

В потоках с идентификаторами не больше указанного в кадре значения, которые не были полностью закрыты до завершения соединения, попытки повтора запросов, транзакции или иные операции протокола невозможны, за исключением идемпотентных действий, таких как HTTP GET, PUT, DELETE. Любые действия протокола, использующие потоки с большими идентификаторами, можно безопасно повторить в новом соединении. Действия в потоках с номерами не больше указанного в кадре могут завершиться успехом. Отправитель кадра GOAWAY может аккуратно завершить соединение, сохраняя его в состоянии open до завершения всех обрабатываемых потоков.

При изменении ситуации конечная точка **может** передать несколько кадров GOAWAY. Например, после отправки GOAWAY с NO_ERROR в процессе аккуратного завершения может возникнуть ситуация, требующая немедленного разрыва соединения. Идентификатор последнего потока из последнего принятого кадра GOAWAY указывает, для каких потоков могли быть выполнены действия. Конечной точке **недопустимо** увеличивать значение последнего идентификатора потока, поскольку партнёры уже могли передать необработанные запросы в новом соединении.

Клиент, не способный повторять запросы, при закрытии соединения сервером теряет все находящиеся в пути запросы. Это особенно важно для посредников, не способных обслуживать клиентов, использующих HTTP/2. Серверу, пытающемуся аккуратно завершить соединение, **следует** передать первый кадр GOAWAY с максимальным идентификатором 2^{31}-1 и кодом NO_ERROR. Оно укажет клиенту неизбежность разрыва и запрет отправки новых запросов. По истечении времени для создания находящихся в пути потоков (не менее времени кругового обхода) сервер **может** передать другой кадр GOAWAY с обновлённым последним номером. Это обеспечит завершение соединения без потери запросов.

После передачи GOAWAY отправитель может отбрасывать кадры потоков, созданных получателем с идентификаторами больше указанного в кадре. Однако кадры, меняющие состояние соединения, нельзя игнорировать полностью. Например, кадры HEADERS, PUSH_PROMISE, CONTINUATION **должны** обрабатываться по минимуму, чтобы обеспечить согласованность состояния сжатия полей (параграф 4.3), кадры DATA **должны** учитываться в окне управления потоком данных соединения. Отказ от обработки этих кадров вызовет несогласованность управления потоком данных или состояния сжатия раздела полей.

Кадр GOAWAY содержит 32-битовый код ошибки (раздел 7), указывающий причину закрытия соединения.

Конечная точка **может** добавлять необрабатываемые (opaque) данные в конце содержимого (payload) любого кадра GOAWAY. Дополнительные отладочные данные служат лишь для диагностики и не имеют семантического смысла.

Отладочные данные могут быть приватными или конфиденциальными. При их регистрации или ином постоянном хранении **должны** обеспечиваться адекватные меры защиты от несанкционированного доступа.

6.9. WINDOW_UPDATE

Кадры WINDOW_UPDATE (type=0x08) служат для управления потоком данных (см. параграф 5.2). Управление потоком данных работает на двух уровнях - для потока и для соединения в целом. Оба типа управления работают поэтапно (hop by hop), т. е. лишь между парой конечных точек. Посредники не пересылают кадры WINDOW_UPDATE между связанными соединениями. Однако дросселирование передачи данных любым получателем может косвенно приводить к распространению сведений управления потоком данных в направлении исходного отправителя.

Управление потоком данных применяется лишь к кадрам, которые указаны как подлежащие такому управлению. Из заданных этим документом кадров к таковым относятся лишь кадры DATA. Кадры, для которых не применяется управление потоком данных, **должны** восприниматься и обрабатываться, пока получатель может выделить ресурсы для их обслуживания. Получатель **может** отвечать ошибкой потока (параграф 5.4.2) или соединения (параграф 5.4.1) типа FLOW_CONTROL_ERROR, если он не способен воспринять кадр.

```
WINDOW_UPDATE Frame {
  Length (24) = 0x04,
  Type (8) = 0x08,

  Unused Flags (8),

  Reserved (1),
  Stream Identifier (31),

  Reserved (1),
  Window Size Increment (31),
}
```

Рисунок 11.Формат кадра WINDOW_UPDATE.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4. Содержимым (payload) кадра WINDOW_UPDATE является зарезервированный бит и 31-битовое целое число, указывающее количество октетов, которые отправитель может передать с дополнение к имеющемуся окну управления потоком данных. Диапазон разрешённых размеров увеличения окна управления потоком данных составляет 1 - 2^{31}-1 (2147483647) октетов.

Для кадров WINDOW_UPDATE не задано флагов.

Кадр WINDOW_UPDATE может относиться к отдельному потоку или соединению в целом. В первом случае в кадре указывается идентификатор соответствующего потока, в последнем - 0.

Получатель **должен** считать приём кадра WINDOW_UPDATE с инкрементом окна управления потоком данных 0 ошибкой потока (параграф 5.4.2) типа PROTOCOL_ERROR, ошибки для окна управления потоком данных в соединении **должны** считаться ошибками соединения (параграф 5.4.1).

Кадр WINDOW_UPDATE может быть передан партнёром, отправившим кадр с флагом END_STREAM. Это означает, что получатель примет кадр WINDOW_UPDATE в потоке со статусом half-closed (remote) или closed. **Недопустимо** считать это ошибкой (см. параграф 5.1).

Получатель кадра, подлежащего управлению потоком данных, всегда **должен** учитывать вклад этого кадра в окно управления потоком данных соединения, если он не рассматривает это как ошибку соединения (параграф 5.4.1). Это требуется даже для кадров с ошибками. Отправитель учитывает кадр в окне управления потоком данных, но если этого не делает получатель, размер окна у отправителя и получателя становится разным.

Кадр WINDOW_UPDATE с размером, отличным от 4 октетов, **должен** считаться ошибкой соединения (параграф 5.4.1) типа FRAME_SIZE_ERROR.

6.9.1. Окно управления потоком данных

Управление потоком данных в HTTP/2 реализовано с применением окна, хранящегося у каждого отправителя в каждом потоке. Размер окна указывается целым числом, которое говорит, сколько октетов данных разрешается передать отправителю, размер окна определяется буферной ёмкостью получателя.

Применяется два окна - на уровне потока и на уровне соединения в целом. Отправителю **недопустимо** передавать кадры, подверженные управлению потоком данных, размер которых превышает пространство, доступное в любом из анонсированных получателем окон. Кадры нулевого размера с установленным флагом END_STREAM (пустые кадры DATA) **можно** передавать даже при отсутствии пространства в окнах управления потоком данных. При расчётах для управления потоком данных 9-октетные заголовки кадров не учитываются.

После отправки кадра с управлением потоком данных отправитель уменьшает доступное пространство в обоих окнах на размер переданного кадра. Получатель передаёт кадр WINDOW_UPDATE при прочтении данных и освобождении пространства в окнах управления потоком данных. Для окон потока и соединения передаются отдельные кадры WINDOW_UPDATE. Получателям рекомендуется иметь механизм предотвращения отправки кадров WINDOW_UPDATE с очень малым приращением (см. параграф 4.2.3.3 в [RFC1122]).

Отправитель, получив кадр WINDOW_UPDATE, обновляет соответствующее окно, добавляя значение из кадра.

Отправителю **недопустимо** разрешать окно управления потоком данных размером больше 2^{31}-1 октетов. Если отправитель получает кадр WINDOW_UPDATE, вызывающий такое превышение, он **должен** прервать поток или соединение (что подходит). Для потоков отправитель передаёт кадр RST_STREAM с кодом ошибки FLOW_CONTROL_ERROR, для соединения - GOAWAY с кодом ошибки FLOW_CONTROL_ERROR.

Кадры с управлением потоком данных от отправителя и кадры WINDOW_UPDATE от получателя не синхронизированы между собой. Это позволяет получателю активно обновлять размер окна у отправителя для предотвращения замирания потоков.

6.9.2. Начальный размер окна управления потоком данных

При организации соединения HTTP/2 новые потоки создаются с начальным размером окна управления потоком данных в 65535 октетов, такой же размер окна устанавливается для соединения. Обе конечные точки могут настроить размер начального окна для новых потоков, включив значение SETTINGS_INITIAL_WINDOW_SIZE в кадр SETTINGS. Размер начального окна для соединения изменяется с помощью кадра WINDOW_UPDATE. Перед отправкой кадра SETTINGS со значением SETTINGS_INITIAL_WINDOW_SIZE конечная точка может использовать лишь принятый по умолчанию начальный размер окна при передаче кадров с управлением потоком данных. Окно для соединения устанавливается на основе принятого по умолчанию размера окна, пока не будет получен кадр WINDOW_UPDATE. Помимо изменения размера окна управления потоком данных для ещё не активных потоков, кадр SETTINGS может менять начальный размер окна для потоков с активным окно управления потоком данных (потоков со статусом open и half-closed (remote)). При смене значения SETTINGS_INITIAL_WINDOW_SIZE получатель **должен** изменить размер окна управления потоком данных во всех потоках, которые он поддерживает на разность между новым и старым значением. Изменение SETTINGS_INITIAL_WINDOW_SIZE может приводить к отрицательному размеру доступного в окне управления потоком данных пространства. Отправитель **должен** отслеживать такие случаи и ему **недопустимо** передавать новые кадры с управлением потоком данных, пока не будет принят кадр WINDOW_UPDATE, делающий размер окна положительным. Например, если клиент передал 60 Кбайт сразу после организации соединения, а сервер установил начальный размер окна в 16 Кбайт, клиент будет иметь размер доступного окна -44 Кбайт после приёма кадра SETTINGS. Клиент будет сохранять отрицательный размер окна, пока кадры WINDOW_UPDATE не восстановят положительное значение, после чего клиент может продолжить отправку.

Кадр SETTINGS не может менять окно управления потоком данных.

Конечная точка **должна** считать установку SETTINGS_INITIAL_WINDOW_SIZE, вызывающую превышение любым окном управления потоком данных максимального размера, ошибкой соединения (параграф 5.4.1) типа FLOW_CONTROL_ERROR.

6.9.3. Снижение размера окна для потока

Получатель, желающий использовать меньшее окно управления потоком данных, чем сейчас, может передать новый кадр SETTINGS. Однако он **должен** быть готов к получению большего объёма данных, поскольку отправитель мог передать эти данные до получения кадра SETTINGS.

После отправки кадра SETTINGS, сокращающего окно управления потоком данных, получатель **может** продолжать обработку потоков, превышающих ограничение. Это не позволяет получателю моментально сократить пространство, используемое для окно управления потоком данных. Продвижение этих потоков **может** застопориться, поскольку нужны кадры WINDOW_UPDATE, позволяющие отправителю возобновить передачу. Получатель может взамен передать для затронутых потоков RST_STREAM с кодом ошибки FLOW_CONTROL_ERROR.

6.10. CONTINUATION

Кадр CONTINUATION (type=0x09) служит для продолжения последовательности фрагментов блока полей (параграф 4.3). Можно передать любое число кадров CONTINUATION, пока предыдущий кадр передаётся в том же потоке и имеет тип HEADERS, PUSH_PROMISE или CONTINUATION без флага END_HEADERS.

```
CONTINUATION Frame {
  Length (24),
  Type (8) = 0x09,

  Unused Flags (5),
  END_HEADERS Flag (1),
  Unused Flags (2),

  Reserved (1),
  Stream Identifier (31),

  Field Block Fragment (..),
}
```

Рисунок 12.Формат кадра CONTINUATION.

Поля Length, Type, Unused Flag(s), Reserved, Stream Identifier описаны в разделе 4. Содержимым (payload) кадра CONTINUATION является фрагмент блока полей (параграф 4.3).

Флаг для кадра CONTINUATION описан ниже.

END_HEADERS (0x04)

Установленный флаг указывает, что этот кадр является последним для блока полей (параграф 4.3), а при сброшенном флаге в том же потоке **должен** следовать другой кадр CONTINUATION. Получатель должен считать получение кадра иного типа или приём кадра продолжения в другом потоке ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадр CONTINUATION меняет статус соединения, как указано в параграфе 4.3.

Кадр CONTINUATION **должен** быть связан с потоком, а при получении такого кадра с Stream Identifier = 0x00 получатель **должен** отвечать ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадру CONTINUATION **должен** предшествовать кадр HEADERS, PUSH_PROMISE или CONTINUATION без флага END_HEADERS. При нарушении этого получатель **должен** отвечать ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

7. Коды ошибок

Коды ошибок передаются 32-битовым полем в кадрах RST_STREAM и GOAWAY для указания причины ошибки.

Для кодов ошибок используется общее пространство. Некоторые коды применимы к потокам или всему соединению и не имеют семантики в ином контексте. Идентификаторы и значения кодов ошибок приведены ниже.

NO_ERROR (0x00)

Связанное с кодом условие не является результатом ошибки. Например, кадр GOAWAY может включать этот код для указания аккуратного завершения соединения.

PROTOCOL_ERROR (0x01)

Конечная точка обнаружила не указанную ошибку протокола. Этот код применяется при отсутствии более конкретного.

INTERNAL_ERROR (0x02)

Конечная точка столкнулась с неожиданной внутренней ошибкой.

FLOW_CONTROL_ERROR (0x03)

Конечная точка обнаруженила нарушение партнёром протокола управления потоком данных.

SETTINGS_TIMEOUT (0x04)

Конечная точка передала кадр SETTINGS и не получила своевременного отклика (см. параграф 6.5.3).

STREAM_CLOSED (0x05)

Конечная точка получила кадр в полузакрытом потоке.

FRAME_SIZE_ERROR (0x06)

Конечная точка получила кадр с недействительным размером.

REFUSED_STREAM (0x07)

Конечная точка отвергла поток до выполнения прикладной обработки (см. параграф 8.7).

CANCEL (0x08)

Конечная точка использует этот код для укащания того, что поток больше не нужен.

COMPRESSION_ERROR (0x09)

Коенчная точка не может поддерживать контекст сжатия полей для соединения.

CONNECT_ERROR (0x0a)

Соединение, организованное по запросу CONNECT (параграф 8.5) сброшено или аварийно закрыто.

ENHANCE_YOUR_CALM (0x0b)

Конечная точка обнаружила, что поведение партнёра может вызывать избыточную нагрузку.

INADEQUATE_SECURITY (0x0c)

Базовый транспорт не соответствует минимальным требованиям безопасности (см. параграф 9.2).

HTTP_1_1_REQUIRED (0x0d)

Конечная точка требует использовать HTTP/1.1 вместо HTTP/2.

Неизвестным или неподдерживаемым кодам ошибок **недопустимо** вызывать особое поведение. Реализация **может** считать такие коды эквивалентом INTERNAL_ERROR.

8. Выражение семантики HTTP в HTTP/2

HTTP/2 является реализацией абстракции сообщений HTTP (раздел 6 в [HTTP]).

8.1. Кадрирование сообщений HTTP

Клиент передаёт запрос HTTP в новый поток, используя незанятый идентификатор потока (параграф 5.1.1), а севео возвращает отклик HTTP в том же потоке. Сообщение HTTP (запрос или отклик) состоит из::

1. одного кадра HEADERS (за которым могут следовать кадры CONTINUATION), содержащего раздел заголовков (см. параграф 6.3 в [HTTP]);
2. необязательных кадров DATA с содержимым сообщения (см. параграф 6.4 в [HTTP]);
3. необязательного кадра HEADERS (за которым могут следовать кадры CONTINUATION), содержащего раздел трейлеров, если он имеется (см. параграф 6.5 в [HTTP]).

При ответе сервер **может** передать любое число промежуточных (interim) откликов до кадра HEADERS с финальным откликом. Промежуточный отклик ACK состоит из кадра HEADERS (за которым могут следовать кадры CONTINUATION) с данными управления и разделом заголовков промежуточного (1xx) отклика HTTP (см. раздел 15 в [HTTP]). Кадр HEADERS с установленным флагом END_STREAM, содержащий информационный (1xx) код статуса, является некорректно сформированным (параграф 8.1.1). В последнем кадре последовательности устанавливается флаг END_STREAM, но за ним могут следовать кадры CONTINUATION с остальными фрагментами блока полей. Другие кадры (из любого потока) **недопустимо** включать между кадром HEADERS и следующими за ним кадрами CONTINUATION.

HTTP/2 использует кадры DATA для передачи содержимого сообщения. Блочное (chunked) транспортное кодирование, определённое в параграфе 7.1 [HTTP/1.1], неприменимо в HTTP/2 (см. параграф 8.2.2).

Поля трейлера передаются в блоке полей, который завершает поток. Т. е. поля трейлера образуют последовательность, начинающуюся с кадра HEADERS с установленным флагом END_STREAM, за которым могут следовать кадры CONTINUATION. В трейлер **недопустимо** включать поля псевдозаголовка (параграф 8.3). Конечная точка, получившая такие поля в трейлере, **должна** считать запрос или отклик неверно сформированным (параграф 8.1.1). Конечная точка, получившая кадр HEADERS без флага END_STREAM после приёма открывающего запрос кадра HEADERS или после финального (не информационного) кода статуса, **должна** считать запрос или отклик неверно сформированным (параграф 8.1.1).

Обмен HTTP запрос-отклик полностью расходует один поток. Запрос начинается с кадра HEADERS, переводящего поток в состояние open, и завершается кадром с флагом END_STREAM, который вызывает переход потока в состояние half-closed (local) для клиента и half-closed (remote) для сервера. Поток отклика начинается с необязательный промежуточных откликов в кадрах HEADERS, за которыми следует кадр HEADERS с финальным кодом статуса.

Запрос HTTP завершается после того, как сервер отправит или клиент получит кадр с флагом END_STREAM (включая кадры CONTINUATION для завершения блока полей). Сервер может передать полный отклик до того, как клиент отправит весь запрос, если отклик не зависит от ещё не полученной части запроса. В таком случае сервер **может** попросить клиента прервать передачу запроса без ошибки, передав RST_STREAM с кодом NO_ERROR после отправки полного отклика (кадр с флагом END_STREAM). Клиенту **недопустимо** отбрасывать отклик при получении такого RST_STREAM, хотя он всегда может отбросить отклик по своему усмотрению в силу иных причин.

8.1.1. Некорректно сформированные сообщения

Некорректно сформированным считается запрос или отклик, непригодный из-за наличия лишних кадров, запрещённых полей или псевдозаголовков, отсутствия обязательных полей псевдозаголовка, наличия заглавных букв в именах полей, недействительных имён или значений полей (в некоторых случаях, см. параграф 8.2), а в остальном являющийся корректной последовательностью кадров HTTP/2.

Запрос или отклик с содержимым может включать поле заголовка content-length, но будет считаться некорректно сформированным, если значение поля content-length не равно сумме размеров данных (payload) в кадрах DATA, образующих содержимое (если только сообщение не определено как не имеющее содержимого). Например, отклики 204 и 304 не имеют содержимого, как и отклик на запрос HEAD. Отклик, заданный как не имеющий содержимого (см. параграф 6.4.1 в [HTTP]), **может** включать отличное от 0 поле content-length, даже при отсутствии содержимого в кадрах DATA.

Посредникам, обрабатывающим запросы или отклики HTTP (т. е. не являющимся туннелями), **недопустимо** пересылать некорректно сформированные запросы и отклики, а получение их **должно** считаться ошибкой потока (параграф 5.4.2) типа PROTOCOL_ERROR.

Для некорректно сформированных запросов сервер **может** передавать отклик HTTP до закрытия или сброса потока. Клиенту **недопустимо** воспринимать некорректно сформированные откликиaccept a malformed response.

Конечные точки, обрабатывающие сообщения постепенно могут выполнить часть обработки до обнаружения непригодной формы запроса или отклика. Например, можно сгенерировать информационный код статуса или код 404 без получения запроса целиком. Посредник может переслать неполное сообщение до обнаружения ошибки. Сервер **может** сгенерировать окончательный отклик до получения всего запроса, когда отклик не зависит от корректности оставшейся части запроса.

Эти требования предназначены для защиты от некоторых типов атак на HTTP и они намеренно строги, поскольку попустительство может сделать реализацию уязвимой.

8.2. Поля HTTP

Поля HTTP (параграф 5 в [HTTP]) передаются в кадрах HTTP/2 HEADERS, CONTINUATION, PUSH_PROMISE со сжатием HPACK [COMPRESSION].

При создании сообщения HTTP/2 имена полей должны приводиться к нижнему регистру.

8.2.1. Пригодность поля

Определения имён и значений полей HTTP запрещают некоторые символы, которые может передавать HPACK. реализациям HTTP/2 **следует** проверять соответствие имён и значений полей их определениям в параграфах 5.1 и 5.5 в HTTP] и считать некорректно сформированными (параграф 8.1.1) поля с запретными символами.

Отказы при проверке полей могут использоваться для атак с контрабандой запросов. В частности, непроверенные поля могут применяться для атак при пересылке сообщений, использующих HTTP/1.1 [HTTP/1.1], где символы возврата каретки (CR), перевода строки (LF) и двоеточие (COLON) служат разделителями. Реализации **должны** выполнять минимальные проверки имён и значений полей, как указано ниже.

- В имена полей **недопустимо** включать символы из диапазонов 0x00-0x20, 0x41-0x5a, 0x7f-0xff (включительно). Это запрещает все невидимые символы ASCII, ASCII SP (0x20) и символы верхнего регистра (A-Z, ASCII 0x41 - 0x5a).
- За исключением полей псевдозаголовка (параграф 8.3), начинающихся с двоеточия (:), в имена полей **недопустимо** включать символ двоеточния (ASCII COLON, 0x3a).
- В значения полей **недопустимо** включать символ NUL (ASCII NUL, 0x00), а также символ перевода строки (ASCII LF, 0x0a) и возвраат каретки (ASCII CR, 0x0d).
- Значение поля **недопустимо** начинать или заканчивать пробельным символом ASCII (ASCII SP или HTAB, 0x20 или 0x09).

  Примечание. Реализации, проверяющей поля в соответствии с определениями параграфов 5.1 и 5.5 в [HTTP] нужна лишь проверка отсутствия в именах полей заглавных букв.

Запрос или отклик, включающий поле, которое нарушает приведённые выше требования, **должен** считаться некорректно сформированным (параграф 8.1.1). В частности, посреднику, не обрабатывающему поля при пересылке, **недопустимо** пересылать поля, значения которых указаны выше как запрещенные.

При нарушении запросом любого из указанных выше требований реализации **следует** генерировать код статуса 400 (Bad Request) (см. параграф 15.5.1 в [HTTP]), если нет более подходящего кода или код статуса невозможно передать (например, при ошибке в поле трейлера).

> Примечание. Имена полей, недействительные в соответствии с определением, не вызывают признания запроса некорретно сформированным. Приведённые выше требования применяются лишь к базовому синтаксису полей, заданному в разделе 5 [HTTP].

8.2.2. Связанные с соединением поля заголовка

В HTTP/2 не применяется поле заголовка Connection (параграф 7.6.1 в [HTTP]) для указания специфичных для соединения полей. В этом протоколе специфичные для соединения метаданные передаются иначе. Конечной точке **недопустимо** генерировать сообщения HTTP/2 со специфичными для соединения полями заголовка. Это включает Connection и поля, указанные как специфичные для соединения в параграфе 7.6.1 [HTTP] (Proxy-Connection, Keep-Alive, Transfer-Encoding, Upgrade). Сообщения со специфичными для соединения полями заголовка **должны** считаться некорректно сформированными (параграф 8.1.1). Едиственным исключением является поле заголовка TE, которое может присутствовать в запросе HTTP/2, этому полю **недопустимо** иметь значение, отличное от trailers.

Посредник, преобразующий сообщение HTTP/1.x в HTTP/2, **должен** удалять специфичные для соединения поля, как описано в параграфе 7.6.1 [HTTP], иначе такие сообщения другие конечные точки HTTP/2 будут считать некорректно сформированными (параграф 8.1.1).

> Примечание. В HTTP/2 целенаправленно не поддерживается переход на другой протокол (обновление). Методы согласования, описанные в разделе 3, считаются достаточными для согласования дополнительных протоколов.

8.2.3. Сжатие поля заголовка Cookie

В поле заголовка Cookie [COOKIE] применяется точка с запятой (;) для разделения cookie-пар (crumb). В этом поле содержится несколько значений, но запятая (, COMMA) не применяется как разделитель, предотвращая передачу cookie-пар в нескольких строках поля (см. параграф 5.2 в [HTTP]). Это может значительно снизить эффективность сжатия, поскольку при обновлении отдельных cookie-пар все записи в таблице HPACK становятся недействительными.

Для повышения эффективности сжатия поле заголовка Cookie **можно** разделить на отдельные поля, каждое из которых содержит одну или несколько cookie-пар. Если после декомпресии остаётся несколько полей Cookie, они **должны** объединяться в одну строку октетов с использованием двухоктетного разделителя 0x3b, 0x20 (строка ASCII string "; ") до передачи в контекст, не относящийся к HTTP/2, например, в соединение HTTP/1.1 или обычное серверное приложение HTTP. С учётом этого два приведённых ниже списка полей заголовка Cookie семантически эквивалентны.

```
cookie: a=b; c=d; e=f

cookie: a=b
cookie: c=d
cookie: e=f
```

8.3. Данные управления HTTP

В HTTP/2 применяются специальные поля псевдозаголовка, начинающиеся с символа двоеточия (:, ASCII 0x3a), для передачи данных управления (см. параграф 6.2 в [HTTP]). Поля псевдозаголовков не являются полями заголовка HTTP. Конечным точкам **недопустимо** генерировать поля псевдозаголовка, кроме заданных в этом документе, однако расширения могут задавать дополнительные поля (см. параграф 5.5).

Поля псевдозаголовка действительны лишь в контексте, где они были определены, поля псевдозаголовка откликов **недопустимо** включать в запросы и наоборот, а также **недопустимо** включать поля псевдозаголовка в трейлеры. Конечные точки **должны** считать запрос или отклик с неопределённым или недействительным полем псевдозаголовка некорректно сформированными (параграф 8.1.1).

Все поля псевдозаголовка **должны** включаться в блок полей до строй обычных полей. Любой запрос или отклик с полями псевдозаголовка в блоке полей после обычных полей заголовка **должен** считаться некорректно сформированным (параграф 8.1.1).

Одно и то же поле заголовка **недопустимо** включать в блок полей более одного раза. Блок полей для запроса или отклика HTTP с повторяющимся полем заголовка **должен** считаться некорректно сформированным (параграф 8.1.1).

8.3.1. Поля псевдозаголовка запроса

Ниже указаны поля псевдозаголовка, определённые для запросов HTTP/2.

- :method указывает метод HTTP method (параграф 9 в [HTTP]).
- :scheme включает схему из цели запроса. Схема берётся из URI цели (параграф 3.1 в [RFC3986]) при генерации запроса напрямую или из схемы транслированного запроса (см., например, параграф 3.3 в [HTTP/1.1]). В запросах CONNECT (параграф 8.5) схема не указывается.

  В поле :scheme можно указывать не только http и https. Прокси и шлюзы могут транслировать запросы в другие (не HTTP) схемы, обеспечивая взаимодействие HTTP с другими типами служб.
- :authority содержит компонент полномочий (параграф 3.2 в [RFC3986]) из URI цели (параграф 7.1 of [HTTP]). Получателю запроса HTTP/2 **недопустимо** использовать поле заголовка Host для указания URI цели при наличии :authority.

  Клиенты, создающие запросы HTTP/2 напрямую, **должны** использовать поле :authority для передачи сведений о полномочиях, если эти сведения нужно передавать (в ином случае создавать :authority **недопустимо**).

  Клиентам **недопустимо** создавать запросы с полем заголовка Host, отличающимчя от поля псевдозаголовка :authority. Серверу **следует** считать такие запросы некорректно сформированными, если поле Host указывает сущность, отличающуюся от указанной в :authority. Значение полей при сравнении должны нормализоваться (см. параграф 6.2 bf [RFC3986]). Сервер-источник может применять любой метод нормализации, а другие серверы **должны** использовать нормализацию на основе схемы (см. параграф 6.2.3 в [RFC3986]) из двух полей.

  Посредник, пересылающий запрос по HTTP/2, **должен** создавать поле псевдозаголовка :authority и использованием сведений о полномочиях из данных управления исходного запроса, если она имеется в URI цели (в ином случае создавать :authority **недопустимо**). Отметим, что поле заголовка Host не является единственным источником таких сведений (см. параграф 7.2 в [HTTP]). Посредник, которому нужно создавать поле заголовка Host (это может требоваться для создания запроса HTTP/1.1), **должен** использовать значение :authority в качестве значения поля Host, если только он не меняет цель запроса. Это заменяет имеющееся значение поля Host для предотвращения возможных уязвимостей маршрутизации HTTP. Посредник, пересылающий запрос через HTTP/2, **может** сохранить поле заголовка Host.

  Отметим, что цель запроса для CONNECT или asterisk-form в OPTIONS не включает сведений о полномочиях (см. параграфы 7.1 и 7.2 в [HTTP].

  В :authority **недопустимо** включать устаревший субкомпонент userinfo для URI со схемой http или https.

- :path включает путь и части запроса URI цели (absolute-path и необязательный символ ?, за которым следует запрос (query), см. параграф 4.1 в [HTTP]). Запрос в форме * (для OPTIONS) включает значение * для поля псевдозаголовка :path.

  Этому полю псевдозаголовка **недопустимо** быть пустым для URI http и https, которые при отсутствии компонента пути **должны** включать значение /. Исключения из этого правила приведены ниже.

  - Запрос OPTIONS для URI http или https без компонента пути **должен** включать поле псевдозаголовка :path со значением * (см. параграф 7.1 в [HTTP]).
  - Запросы CONNECT (параграф 8.5), где поле псевдозаголовка :path отсутствует.

Все запросы HTTP/2, кроме CONNECT (параграф 8.5), **должны** включать в точности одно значение для полей псевдозаголовка :method, :scheme, :path. Запрос HTTP без обязательных полей псевдозаголовка считается некорректно сформированным (параграф 8.1.1).

В отдельных запросах HTTP/2 нет явного индикатора версии протокола. Все запросы HTTP/2 неявно имеют версию протокола 2.0 (см. параграф 6.2 в [HTTP]).

8.3.2. Поля псевдозаголовка отклика

Для откликов HTTP/2 определено единственное поле псевдозаголовка :status, передающее код статуса (см. раздел 15 в [HTTP]). Это поле **должно** включаться во все отклики, в том числе, промежуточные. В противном случае отклик будет считаться сформированным некорректно (параграф 8.1.1).

Отклик HTTP/2 неявно имеют версию протокола 2.0.

8.4. Выталкивание с сервера

HTTP/2 позволяет серверу заранее передавать (выталкивать - push) отклики, основанные на предшествующем запросе клиента, вместе с соответствующими «предсказанными» (promised) запросами. Выталкивание с сервера было разработано для того, чтобы позволить серверу повысить воспринимаемую клиентом производительность путём предсказания запросов, которые последуют за полученными, что позволяло исключить один интервал кругового обхода. Например, за запросом страницы HTML часто следуют запросы таблиц стилей и сценариев, на которые эта страница ссылается. При выталкивании таких запросов клиенту не ждать получения ссылок на них в HTML и отправлять отдельные запросы.

На практике сложно эффективно использовать выталкивание с сервера, поскольку для этого требуется конкретно предсказать дополнительные запросы, которые клиент может сделать, учитывая такие факторы, как кэширование, согласование содержимого и поведение пользователя. Ошибки при прогнозировании ведут к снижению производительности из-за дополнительных затрат на передачу данных. В частности, выталкивание значительного объёма данных может приводить к проблемам из-за соперничества с откликами, которое более важны. Клиент может запросить запрет выталкивания с сервера, но это нужно независимо согласовывать на каждом интервале (hop). Для отключения выталкивания с сервера можно передать SETTINGS_ENABLE_PUSH = 0.

Предсказанные запросы **должны** быть безопасными (см. параграф 9.2.1 в [HTTP]) и кэшируемыми (параграф 9.2.3 в [HTTP]), а также не могут включать содержимого и раздела трейлеров. Получение клиентом предсказанного запроса, который не является кэшируемым, для которого неизвестно о безопасности и в котором указано наличие содержимого, **должно** сбрасывать promised-поток с ошибкой потока (параграф 5.4.2) типа PROTOCOL_ERROR. Отметим, что это может приводить к сбросу promised-потока, если клиент не считает вновь определённый метод безопасным.

Вытолкнутые отклики, которые являются кэшируемыми (см. раздел 3 в [CACHING]), клиент может сохранять, если он реализует кэширование HTTP. Вытолкнутые отклики считаются подтверждёнными сервером-источником, пока поток, указанный идентификатором promised-потока, остаётся открытым (для случая наличия директивы кэширования откликов no-cache см. параграф 5.2.2.4 в [CACHING]). Некэшируемые вытолкнутые отклики **недопустимо** сохранять в каком-либо кэше HTTP. Они **могут** быть сделаны доступными приложению отдельно.

Сервер **должен** включать значение в поле псевдозаголовка :authority, для которого сервер является полномочным (см. параграф 10.1). Клиент **должен** считать кадр PUSH_PROMISE, для которого сервер не является полномочным, ошибкой потока (параграф 5.4.2) типа PROTOCOL_ERROR.

Посредники могут получать выталкивание (push) от сервера и отказываться от его пересылки клиенту. Иными словами, решение о применении вытолкнутой информации посредник принимает сам. Он также может передавать клиенту дополнительные push-отправки без каких-либо действий со стороны сервера.

Клиент не может применять выталкивание, поэтому получение сервером кадра PUSH_PROMISE **должно** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Сервер не может передавать SETTINGS_ENABLE_PUSH с отличным от 0 значением (см. параграф 6.5.2).

8.4.1. Выталкивание запроса

Выталкивание с сервера семантически эквивалентно отклику на запрос, однако этот запрос также передаётся сервером в форме кадра PUSH_PROMISE. Такой кадр включает блок полей с данными управления и полным набором полей заголовка запроса, приписанных тому серверу. Невозможно вытолкнуть отклик на запрос с содержимым.

Предсказанный запрос всегда связывается с явным запросом от клиента. Кадры PUSH_PROMISE передаются сервером в поток явного запроса и включают идентификатор предсказанного (promised) потока, выбранный из доступных серверу идентификаторов потока (см. параграф 5.1.1). Поля заголовка в PUSH_PROMISE и последующих кадрах CONTINUATION **должны** образовывать действительный и полный набор полей заголовка запроса (параграф 8.3.1). Сервер **должен** указывать в поле псевдозаголовка :method безопасный и кэшируемый метод. Если клиент получает кадр PUSH_PROMISE без полного и действительного набора заголовков или поле псевдозаголовка :method указывает небезопасный метод, он **должен** ответить в promised-потоке ошибкой потока (параграф 5.4.2) типа PROTOCOL_ERROR. Серверу **следует** передавать кадры PUSH_PROMISE (параграф 6.6) до отправки любых кадров, ссылающихся на предсказанные отклики. Это позволяет предотвратить состязание при отправке клиентом запроса до

получения PUSH_PROMISE. Например, если сервер получает запрос на документ, содержащий встроенные ссылки на несколько файлов с изображениями, и решает вытолкнуть клиенту эти дополнительные изображения, передавая кадры PUSH_PROMISE до кадров DATA, содержащих ссылки на изображения, чтобы клиент мог видеть, что ресурс вытолкнут до обнаружения встроенных ссылок. Если сервер выталкивает ресурсы, указанные блоком полей (например, поля заголовка Link), отправка PUSH_PROMISE до передачи заголовка гарантирует, что клиенты не запросят эти ресурсы.

Клиенту **недопустимо** передавать кадры PUSH_PROMISE.

Сервер может передавать кадры PUSH_PROMISE в любой поток, инициированный клиентом, но этот поток **должен** иметь статус open или half-closed (remote) применительно к серверу. Кадры PUSH_PROMISE перемежаются с кадрами отклика, не не могут перемежаться с кадрами HEADERS и CONTINUATION, образующими один блок полей.

Передача кадра PUSH_PROMISE создаёт новый поток и помещает его в состояние reserved (local) для сервера и reserved (remote) для клиента.

8.4.2. Выталкивание отклика

После отправки кадра PUSH_PROMISE сервер может начать доставку выталкиваемого отклика (параграф 8.3.2) в инициированном сервером потоке, который использует идентификатор предсказанного (promised) потока. Сервер использует этот поток для передачи отклика HTTP, применяя последовательность кадров, описанную в параграфе 8.1. Этот поток принимает для клиента состояние half-closed (параграф 5.1) после отправки начального кадра HEADERS.

Когда клиент получает кадр PUSH_PROMISE и решает принять вытолкнутый отклик, ему **не следует** отправлять запросов, пока promised-поток не будет закрыт. Если клиент по какой-либо причине отказывается принимать вытолкнутый отклик или сервер слишком долго не начинает передачу предсказанного отклика, клиент может передать кадр RST_STREAM с кодом CANCEL или REFUSED_STREAM и идентификатором выталкиваемого потока.

Клиент может использовать установку SETTINGS_MAX_CONCURRENT_STREAMS для ограничения числа откликов, которые сервер может вытолкнуть одновременно. Анонсирование SETTINGS_MAX_CONCURRENT_STREAMS = 0 не позволяет серверу открыть потоки, требуемые для выталкивания откликов. Однако это не мешает серверу резервировать потоки с помощью кадров PUSH_PROMISE, поскольку такие потоки не учитываются в числе одновременных потоков. Клиенту, не желающему принимать выталкиваемые ресурсы, нужно сбрасывать (reset) любые нежелательные зарезервированные потоки или задать SETTINGS_ENABLE_PUSH = 0.

Клиенты, принимающие вытолкнутые отклики, **должны** проверять, что сервер полномочен для них (параграф 10.1) или прокси, предоставивший такой отклик, настроен для соответствующего запроса. Например, серверу, предлагающему сертификат лишь для DNS-ID example.com (см. [RFC6125]), не разрешается выталкивать отклики для https://www.example.org/doc.

Отклик для потока PUSH_PROMISE начинается с кадра HEADERS, который сразу переводит поток в состояние half-closed (remote) для сервера и half-closed (local) для клиента, а завершается кадром с флагом END_STREAM, переводящим поток в состояние closed.

Примечание. Клиент никогда не передаёт кадр с флагом END_STREAM для выталкивания с сервера.

8.5. Метод CONNECT

Метод CONNECT (параграф 9.3.6 в [HTTP]) служит для преобразования соединения HTTP в туннель к удалённому хосту и применяется в основном HTTP-прокси при организации сессии TLS с сервером-источником для взаимодействия с ресурсами https.

В HTTP/2 метод CONNECT организует туннель через один поток HTTP/2 к удалённому хосту, а не преобразует в туннель соединение целиком. Раздел заголовков CONNECT создаётся, как указано в параграфе 8.3.1 с некоторыми отличиями:

- в поле псевдозаголовка :method устанавливается значение CONNECT;
- поля псевдозаголовка :scheme и :path **должны** быть опущены;
- поле псевдозаголовка :authority содержит хост и порт для организации соединения (эквивалент authority-form для the request-target в запросах CONNECT, см. параграф 3.2.3 в [HTTP/1.1]).

Запросы CONNECT, не соответствующие этим требованиям, считаются ошибочными (параграф 8.1.1).

Прокси, поддерживающий CONNECT, организует соединение TCP [TCP] с хостом и портом, указанными в поле псевдозаголовка :authority. После организации соединения прокси передаёт клиенту кадр HEADERS с кодом 2xx, как указано в параграфе 9.3.6 [HTTP]. После отправки каждым партнёром начального кадра HEADERS все последующие кадры DATA соответствуют данным, переданным через соединение TCP. Содержимое (payload) всех кадров DATA от клиента прокси передаёт серверу TCP, а данные от этого сервера прокси собирает в кадры DATA. Кадры иных типов (не DATA) и кадры управления потоком (RST_STREAM, WINDOW_UPDATE, PRIORITY) **недопустимо** передавать в поток соединения TCP и получение такого кадра **должно** считаться ошибкой потока (параграф 5.4.2).

Соединение TCP может закрыть любой из партнёров. Флаг END_STREAM в кадре DATA считается эквивалентом бита TCP FIN. Предполагается, что клиент передаст кадр DATA с флагом END_STREAM после получения кадра с установленным флагом END_STREAM. Прокси, получивший кадр DATA с флагом END_STREAM, отправляет присоединённые данные с установленным в последнем сегменте TCP битом FIN. Прокси, получивший сегмент TCP с битом FIN, передаёт кадр DATA с установленным флагом END_STREAM. Отметим, что финальный сегмент TCP или кадр DATA может быть пустым.

Ошибки соединения TCP указываются флагом RST_STREAM. Прокси считает любую ошибку соединения TCP, включающую получение сегмента TCP с установленным битом RST, ошибкой потока (параграф 5.4.2) типа CONNECT_ERROR. Соответственно, прокси **должен** передать сегмент TCP с установленным битом RST при обнаружении ошибки в потоке или соединении HTTP/2.

8.6. Поле заголовка Upgrade

HTTP/2 не поддерживает информационный код статуса 101 (Switching Protocols) (параграф 15.2.2 в [HTTP]). Семантика кода 101 (Switching Protocols) не применима к мультиплексируемому протоколу, а такая же функциональность может быть обеспечена расширенным методом CONNECT [RFC8441] и другими протоколами, способными использовать такие же механизмы, как применяются для согласования в HTTP/2 (раздел 3).

8.7. Надёжность запросов

В общем случае клиент HTTP не может повторить неидемпотентный запрос при возникновении ошибки, поскольку у него нет возможности определить природу ошибки (см. параграф 9.2.2 в [HTTP]). Сервер мог выполнить какую-то часть обработки запроса до возникновения ошибки и повторное выполнение запроса может иметь нежелательные последствия. В HTTP/2 имеется два механизма для информирования клиента о том, что запрос не обработан.

- Кадр GOAWAY указывает наибольший номер потока, который мог быть обработан, поэтому запросы из потоков с большими номерами гарантированно безопасны для повторных попыток.
- В кадр RST_STREAM может включаться код ошибки REFUSED_STREAM для индикации закрытия потока до выполнения обработки. Любой запрос, отправленный в закрытый поток, можно повторять без опасений.

Необработанные запросы не вызывают отказа и клиент **может** автоматически повторить запрос даже для неидемпотентного метода.

Серверу **недопустимо** указывать, что поток не был обработан, если сервер не может это гарантировать. Если находящиеся в потоке кадры переданы на уровень приложения из любого потока, использовать REFUSED_STREAM для этого потока **недопустимо**, а кадр GOAWAY **должен** включать идентификатор потока, не меньше идентификатора данного потока.

В дополнение к этим механизмам кадры PING позволяют клиенту легко проверить соединение. Соединения, остающиеся бездействующими (idle), можно разорвать, поскольку некоторые промежуточные устройства (например, трансляторы сетевых адресов и балансировщики нагрузки) молча отбрасывают привязки соединений. Кадр PING позволяет клиенту безопасно проверить сохранение активности соединения без отправки запроса.

8.8. Примеры

В этом параграфе показаны запросы и отклики HTTP/1.1, а также их эквиваленты в HTTP/2.

8.8.1. Простой запрос

Запрос HTTP GET включает данные управления и заголовок запроса без содержимого сообщения, поэтому передаётся как один кадр HEADERS, за которым могут следовать кадры CONTINUATION с сериализованным блоком полей заголовка. В приведённом ниже кадре HEADERS установлены флаги END_HEADERS и END_STREAM, а кадров CONTINUATION не передаётся.

```
GET /resource HTTP/1.1           HEADERS
Host: example.org          ==>     + END_STREAM
Accept: image/jpeg                 + END_HEADERS
                                     :method = GET
                                     :scheme = https
                                     :authority = example.org
                                     :path = /resource
                                     host = example.org
                                     accept = image/jpeg
```

8.8.2. Простой отклик

Отклик, включающий только данные управления и поля заголовка, передаётся как кадр HEADERS (за которым могут следовать кадры CONTINUATION) с сериализованным блоком полей заголовка отклика.

```
HTTP/1.1 304 Not Modified        HEADERS
ETag: "xyzzy"              ==>     + END_STREAM
Expires: Thu, 23 Jan ...           + END_HEADERS
                                     :status = 304
                                     etag = "xyzzy"
                                     expires = Thu, 23 Jan ...
```

8.8.3. Составной запрос

Запрос HTTP POST с данными управления, заголовком и содержимым сообщения передаётся как один кадр HEADERS, за которым могут следовать кадры CONTINUATION с заголовком и кадры DATA с содержимым. Финальный кадр CONTINUATION (или HEADERS) и финальный кадр DATA имеют установленный флаг END_STREAM.

```
POST /resource HTTP/1.1          HEADERS
Host: example.org          ==>     - END_STREAM
Content-Type: image/jpeg           - END_HEADERS
Content-Length: 123                  :method = POST
                                     :authority = example.org
                                     :path = /resource
{двоичные данные}                    :scheme = https

                                 CONTINUATION
                                   + END_HEADERS
                                     content-type = image/jpeg
                                     host = example.org
                                     content-length = 123

                                 DATA
```

```
                                            + END_STREAM
                                          {двоичные данные}
```

Отметим, что данные той или иной строки поля могут оказаться в соседних фрагментах блока полей. Распределение строк полей по кадрам показано выше лишь для иллюстрации.

8.8.4. Отклик с телом

Отклик с данными управления, заголовком и содержимым сообщения передаётся как кадр HEADERS, за которым могут следовать кадры CONTINUATION с заголовком и кадры DATA с содержимым. В последнем кадре DATA устанавливается флаг END_STREAM.

```
HTTP/1.1 200 OK                   HEADERS
Content-Type: image/jpeg   ==>      - END_STREAM
Content-Length: 123                 + END_HEADERS
                                      :status = 200
{двоичные данные}                     content-type = image/jpeg
                                      content-length = 123

                                  DATA
                                    + END_STREAM
                                  {двоичные данные}
```

8.8.5. Информационный отклик

Информационный отклик с кодом статуса 1xx, отличным от 101, передаётся в кадре HEADERS, за которым могут следовать кадры CONTINUATION. Раздел трейлера передаётся как блок полей после передачи блока полей запроса или отклика и всех кадров DATA. В кадре HEADERS, начинающем блок полей с разделом трейлера имеет установленный флаг END_STREAM.

Ниже приведён пример отклика с кодом 100 (Continue) на запрос с маркером 100-continue в поле заголовка Expect и раздел трейлера.

```
HTTP/1.1 100 Continue             HEADERS
Extension-Field: bar       ==>      - END_STREAM
                                    + END_HEADERS
                                      :status = 100
                                      extension-field = bar


HTTP/1.1 200 OK                   HEADERS
Content-Type: image/jpeg   ==>      - END_STREAM
Transfer-Encoding: chunked          + END_HEADERS
Trailer: Foo                          :status = 200
                                      content-type = image/jpeg
123                                   trailer = Foo
{двоичные данные}
0                                 DATA
Foo: bar                            - END_STREAM
                                  {двоичные данные}

                                  HEADERS
                                    + END_STREAM
                                    + END_HEADERS
                                      foo = bar
```

9. Соединения HTTP/2

В этом разделе описаны атрибуты HTTP, улучшающие совместимость, смягчающие известные угрозы безопасности или возможную вариативность реализаций.

9.1. Управление соединениями

Соединения HTTP/2 сохраняются (persistent). В целях повышения производительности ожидается, что клиенты не будут закрывать соединение, пока не будет установлено, что взаимодействие с сервером больше не нужно (например, пользователь ушёл с web-страницы) или пока соединение не закроет сервер.

Клиент **не следует** создавать более одного соединения HTTP/2 с данным хостом и портом, где хост выводится из URI, выбранной альтернативной службы [ALT-SVC] или настроенного прокси.

Клиент может создавать дополнительные соединения на замену, для подготовки замены соединений с заканчивающимся пространством доступных идентификаторов потоков (параграф 5.1.1), для обновления ключевого материала соединения TLS или замены соединений, где возникли ошибки (параграф 5.4.1).

Клиент **может** создать несколько соединений с одним адресом IP и портом TCP, используя разные значения индикации имени сервера (Server Name Indication или SNI) [TLS-EXT] для обеспечения разных сертификатов TLS, но **следует** избегать создания нескольких соединений с одной конфигурацией.

Серверам рекомендуется поддерживать соединения открытыми как можно дольше, но при необходимости разрешается прерывать бездействующие соединения. Любой из конечных точек, закрывающей транспортное соединение TCP, **следует** сначала передать кадр GOAWAY (параграф 6.8), чтобы обе конечных точки могли надёжно определить, были ли обработаны переданные прежде кадры и аккуратно завершить и прервать оставшиеся задачи.

9.1.1. Повторное использование соединения

Соединения, организованные с сервером-источником напрямую или через туннель с использованием метода CONNECT (параграф 8.5), **можно** повторно использовать для запросов с разными компонентами authority в URI. Соединение можно использовать до тех пор, пока сервер остаётся полномочным (параграф 10.1). Для соединений TCP без TLS это зависит от распознавания имени хоста с одним и тем же адресом IP.

Для ресурсов https повторное использование соединения зависит от пригодности сертификата для хоста в URI. Предоставленный сервером сертификат **должен** удовлетворять всем проверкам, которые клиент выполняет для организации соединения TLS с хостом из URI. Один сертификат может служить для подтверждения полномочности нескольких источников. В параграфе 4.3 [HTTP] описано, как клиент проверяет полномочия сервера для URI.

В некоторых случаях повторное использование соединения для разных источников может приводить к передаче запросов не тому серверу-источнику. Например, завершение TLS может происходить на промежуточном устройстве, использующем расширение TLS SNI [TLS-EXT] для выбора сервера-источника. Это означает, что клиенты могут отправлять запросы серверам, которые могут не быть целью запроса, оставаясь в остальном полномочными.

Сервер, не желающий, чтобы клиенты повторно использовали соединение, может указать, что он не уполномочен для запроса, передав код статуса 421 (Misdirected Request) в отклике (см. параграф 15.5.20 в [HTTP]).

Клиент, настроенный на использование прокси по HTTP/2, направляет запросу этому прокси через одно соединение.

9.2. Использование свойств TLS

Реализации HTTP/2 **должны** использовать TLS версии 1.2 [TLS12] или выше при работе HTTP/2 через TLS. **Следует** выполнять общие рекомендации по использованию TLS [TLSBCP] с дополнительными ограничениями, связанными с HTTP/2. Реализация TLS **должна** поддерживать расширение TLS SNI [TLS-EXT]. Если сервер указан доменным именем [DNS-TERMS], клиенты **должны** передавать расширение TLS server_name, если не применяется иной механизм указания целевого хоста.

Требования к развёртываниям HTTP/2, согласующим TLS 1.3 [TLS13], включены в параграф 9.2.3, а системы TLS 1.2 подчиняются требованиям параграфов 9.2.1 и 9.2.2. Реализациям рекомендуется представлять принятые по умолчанию значения, соответствующие этим требованиям, но в конечном итоге за соответствие отвечает само развёртывание.

9.2.1. Свойства TLS 1.2

У этом параграфе указаны ограничения для набора функций TLS 1.2, которые можно использовать с HTTP/2. Из-за ограничений развёртывания **может** оказаться невозможным отказ при согласовании TLS, когда эти ограничения не соблюдаются. Конечная точка может незамедлительно разорвать соединение HTTP/2, где не выполняются указанные здесь ограничения для TLS, передав ошибку соединения (параграф 5.4.1) типа INADEQUATE_SECURITY.

При развёртывании HTTP/2 через TLS 1.2 сжатие **должно** быть отключено, поскольку оно может приводить к раскрытию сведений, которые без этого остались бы скрытыми [RFC3749]. Базовое сжатие не требуется, поскольку в HTTP/2 имеются средства сжатия, лучше учитывающие контекст и поэтому более подходящие в плане производительности, безопасности и других аспектов.

При развёртывании HTTP/2 через TLS 1.2 **должно** быть отключено повторное согласование и конечная точка ACK **должна** считать его ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Отметим, что запрет повторного согласования может приводить к непригодности долгосрочных соединений из-за ограничений на число сообщений, которые разрешено обрабатывать базовому шифру.

Конечная точка **может** использовать повторное согласование для защиты конфиденциальности свидетельств клиента, предоставляемых при согласовании, но любое повторное согласование **должно** выполняться до отправки предисловия к соединению. Серверу **следует** запрашивать сертификат клиента, если он видит запрос на повторное согласование сразу после организации соединения. Это препятствует использованию повторного согласования в отклике на запрос к конкретному защищённому ресурсу. Будущие спецификации могут обеспечить поддержку для таких случаев. Кроме того, сервер может передать ошибку (параграф 5.4) типа HTTP_1_1_REQUIRED для запроса, где клиент применяет протокол, поддерживающий повторное согласование.

Реализации **должны** поддерживать обмен эфемерными ключами размером не менее 2048 битов для шифров, использующих эфемерное конечное поле Diffie-Hellman (DHE) (параграф 8.1.2 и [TLS12]), и 224 битов для шифров, использующих эфемерную эллиптическую кривую Diffie-Hellman (ECDHE) [RFC8422]. Клиенты **должны** воспринимать DHE размером до 4096 битов. Конечные точки **могут** считать согласование ключей размером меньше нижнего предела ошибкой соединения (параграф 5.4.1) типа INADEQUATE_SECURITY.

9.2.2. Шифры TLS 1.2

При развёртывании HTTP/2 через TLS 1.2 **не следует** применять какой-либо из шифров, указанных в Приложении A. Конечная точка **может** возвращать ошибку соединения (параграф 5.4.1) типа INADEQUATE_SECURITY при согласовании любого из запрещённых шифров. Системы, выбравшие запрещённый шифр, рискуют столкнуться с ошибкой соединения, если нет уверенности, что партнёр поддерживает этот шифр. Реализациям **недопустимо** возвращать такую ошибку в ответ на согласование незапрещенного шифра. Поэтому клиенты, предлагающие незапретные шифры, должны быть готовы к использованию этих шифров с HTTP/2.

Список запрещённых шифров включает обязательные для TLS 1.2 шифры, что означает возможность наличия в системах TLS 1.2 непересекающихся наборов разрешённых шифров. Для предотвращения вызываемых этим отказов при согласовании TLS в системах HTTP/2, использующих TLS 1.2, **должен** поддерживаться шифр TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 [TLS-ECDHE] с эллиптической кривой P-256 [RFC8422].

Отметим, что клиенты могут анонсировать поддержку запретных шифров для соединения с серверами, не поддерживающими HTTP/2. Это позволяет серверу выбрать HTTP/1.1 с запрещённым для HTTP/2 шифром, но может приводить к согласованию запрещённого шифра для HTTP/2 при независимом выборе прикладного протокола и шифра.

9.2.3. Свойства TLS 1.3

TLS 1.3 включает много свойств, не доступных в прежних версиях. Использование таких свойств рассматривается здесь.

Серверам HTTP/2 **недопустимо** передавать сообщения TLS 1.3 CertificateRequest после согласования. Клиенты HTTP/2 **должны** считать такое сообщение ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Запрет аутентификации после согласования может анонсироваться даже после расширения TLS post_handshake_auth, а поддержка такой аутентификации может анонсироваться независимо от ALPN [TLS-ALPN]. Клиенты могут предлагать возможность применения других протоколов, но включение расширения не может означать его поддержку в HTTP/2.

В [TLS13] имеются другие сообщения после согласования - NewSessionTicket и KeyUpdate, которые можно использовать, поскольку они напрямую не взаимодействуют с HTTP/2. Если использование нового типа сообщений TLS не зависит от взаимодействия с протоколом прикладного уровня, такое сообщение TLS можно передавать после согласования.

Можно использовать ранние данные TLS для отправки запроса при соблюдении рекомендаций [RFC8470]. Клиенты передают такие запросы в предположении использования исходных значений для всех настроек сервера.

10. Вопросы безопасности

Применение TLS необходимо для обеспечения многих свойств безопасности этого протокола. Многие утверждения этого раздела утрачивают силу, если не применяется TLS в соответствии с параграфом 9.2.

10.1. Полномочия сервера

HTTP/2 полагается на определение полномочий в HTTP для решения вопроса о полномочиях сервера для предоставления данного отклика (см. параграф 4.3 в [HTTP]). Это опирается на локальное распознавание имён для схемы URI http и аутентифицированную идентификацию сервера для схемы https.

10.2. Кросс-протокольные атаки

При кросс-протокольной атаке злоумышленник заставляет клиента инициировать транзакцию с сервером по протоколу, отличному от понимаемого сервером. Злоумышленник может сделать транзакцию воспринимаемой как действительная для второго протокола. В сочетании с возможностями web-контекста это может служить для взаимодействия со слабо защищёнными серверами в частных сетях.

Завершение согласования TLS идентификатором ALPN для HTTP/2 может считаться достаточной защитой от кросс-протокольных атак. ALPN обеспечивает положительную индикацию готовности сервера работать с HTTP/2, что предотвращает атаки на другие протоколы, основанные на TLS.

Шифрование TLS затрудняет атакующим контроль над данными, который может применяться в кросс-протокольных атаках на протокол с открытым текстом. У открытого (cleartext) варианта HTTP/2 имеется лишь минимальная защита от кросс-протокольных атак. Предисловие к соединению (параграф 3.4) содержит строку, призванную запутать серверы HTTP/1.1, но для других протоколов не предусмотрено какой-либо специальной защиты.

10.3. Атаки на промежуточную инкапсуляцию

HPACK позволяет кодировать имена и значения полей, которые в других версиях HTTP могут считаться разделителями. Посредник, транслирующий запрос или отклик HTTP/2, **должен** проверять поля в соответствии с правилами параграфа 8.2 до преобразования сообщения в другую версию HTTP. Трансляция поля с непригодными разделителями может использована для ошибочной интерпретации сообщений получателями, которой может воспользоваться атакующий. В параграфе 8.2 не задано специальных правил для проверки полей псевдозаголовков. При использовании значений таких полей нужна дополнительная проверка. Это особенно важно при объединении полей :scheme, :authority и :path в одну строку URI [RFC3986]. Аналогичные проблемы могут возникать при объединении этого URI или просто :path с полем :method для создания строки запроса (раздел 3 в [HTTP/1.1]). Простая конкатенация не будет защищённой, если не проверяются полностью входные значения.

Посредник может по иным причинам отвергать поля с недействительными имена или значениями - в частности, поля, не соответствующие базовой грамматике HTTP ABNF из раздела 5 в [HTTP]. Посредники, не выполняющие проверки полей сверх минимума, требуемого параграфом 8.2, могут пересылать поля с непригодными именами и значениями.

Посредник, получивший какие-либо поля, которые требуют удаления до пересылки (см. параграф 7.6.1 в [HTTP]), **должен** удалить эти поля заголовка перед пересылкой сообщения. Кроме того, при пересылке сообщений с полыми Content-Length посредникам следует контролировать корректность формы сообщения (параграф 8.1.1). Это гарантирует корректность кадрирования при трансляции сообщений в HTTP/1.1.

10.4. Кэшируемость выталкиваемых откликов

Для вытолкнутых откликов нет явного запроса от клиента, такой запрос представляет сервер в кадре PUSH_PROMISE.

Кэширование выталкиваемых откликов возможно на основе рекомендаций, предоставленных сервером-источником в поле заголовка Cache-Control. Однако это может вызывать проблемы при наличии на сервер более одного арендатора. Например, может предоставлять каждому из пользователей небольшую часть пространства URI. При совместном использовании пространства на сервере несколькими арендаторами этот сервер **должен** предотвратить возможность арендаторам выталкивать представления ресурсов, для которых у них нет полномочий. Отказ от этого позволит арендатору выдать представление, которое будет обслуживаться из кэша, переопределяя актуальное представление от полномочного арендатора.

Вытолкнутые отклики, для которых сервер-источник не является полномочным (параграф 10.1), **недопустимо** использовать для кэширования.

10.5. DoS-атаки

Для работы соединения HTTP/2 может потребоваться больше ресурсов, чем для соединения HTTP/1.1. Сжатие раздела полей и управление потоком данных зависят от большого числа состояний. Настройки этих функций обеспечивают строгое ограничение выделяемой для них памяти.

Число кадров PUSH_PROMISE не ограничивается таким же способом. Клиенту, воспринимающему выталкивание с сервера, **следует** ограничивать число потоков, которые он разрешает держать в состоянии reserved (remote). Подтверждение чрезмерного числа push-потоков от сервера может считаться ошибкой потока (параграф 5.4.2) типа ENHANCE_YOUR_CALM.

В ряде реализаций HTTP/2 обнаружена уязвимость к DoS-атакам [NFLX-2019-002] и ниже приведён список известных способов, которыми реализации могут быть атакованы.

- Неэффективное отслеживание остающихся исходящих кадров может вызывать перегрузку, если атакующий может вызвать постановку с очередь передачи большого числа кадров. Партнёр может воспользоваться несколькими методами для генерации большого числа кадров:
  - предоставление очень мелких приращений для управления потоком данных в кадрах WINDOW_UPDATE, способное заставить отправителя генерировать большое число кадров DATA;
  - требование наличия конечной точки для ответа на кадры PING;
  - требование подтверждать каждый кадр SETTINGS;
  - недействительный запрос (или выталкивание с сервера), способный вызвать передачу кадров RST_STREAM.
- Атакующий может предоставить большой объем кредитов управления потоком данных на уровне HTTP/2, не предоставляя их на уровне TCP, что будет препятствовать передаче кадров. Конечная точка, создающая и запоминающая кадры для отправки без учёта ограничений TCP, может столкнуться с нехваткой ресурсов.
- Можно использовать большое число мелких или пустых кадров, чтобы вынудить партнёра затратить время на обработку их заголовков. Здесь нужна осторожность, поскольку некоторые варианты применения мелких кадров полностью легитимны (например, обработка пустого кадра DATA или CONTINUATION в конце потока).
- Кадрами SETTINGS также можно злоупотреблять для вынуждения партнёра тратить дополнительное время на обработку. Это можно сделать путём бессмысленного изменения настроек, передачи множества неопределённых установок или многократного изменения конкретной настройки в одном кадра.
- Обработка изменения приоритетов с помощью кадров PRIORITY может потребовать значительного времени и приводить к перегрузке, если передаётся много кадров PRIORITY.
- Сжатие раздела полей предоставляет атакующему возможность расхода ресурсов на обработку. Возможные злоупотребления рассмотрены в разделе 7 [COMPRESSION].
- Ограничения в SETTINGS не могут быть снижены мгновенно, что делает конечную точку зависимой от партнёра, который может превышать новые ограничения. В частности, сразу после организации соединения клиент не знает установленных сервером пределов и может превысить их без явного нарушения протокола.

Большинство свойств, которые могут быть использованы для DoS-атак (например, изменения SETTINGS, мелкие кадры, сжатие полей) имеют легитимное применение. Эти свойства становятся обременительными лишь при неоправданном или избыточном их применении. Конечная точка, не отслеживающая использование таких свойств, подвергается риску DoS-атак. реализациям **следует** контролировать использование таких свойств и устанавливать пределы использования. Конечная точка **может** считать подозрительную активность ошибкой соединения (параграф 5.4.1) типа ENHANCE_YOUR_CALM.

10.5.1. Предельный размер блока полей

Большой блок полей (параграф 4.3) может вынудить реализацию зафиксировать большой объем состояния. Важные для маршрутизации строки полей могут размещаться в конце блока, что будет мешать передаче потока полей конечному адресату. Такое упорядочивание и другие причины, например, обеспечение корректности кэша, означают, что конечной точке может потребоваться целиком буферизовать блок полей. Поскольку размер блока полей жёстко не ограничен, некоторым конечным точкам может потребоваться большой объём памяти для блока полей.

Конечная точка может использовать SETTINGS_MAX_HEADER_LIST_SIZE для информирования своих партнёров о возможных ограничениях размера несжатого блока полей. Эта настройка является лишь рекомендацией, поэтому конечные точки **могут** передавать блок полей, размер которого превышает этот предел, рискуя тем, что запрос или отклик будет сочтён некорректно сформированным. Этот параметр относится к соединению и любой запрос или отклик может столкнуться в пути с более низким и неизвестным ограничением. Посредник может попытаться избежать этой проблемы, передавая значения, представленные разными партнёрами, но не обязан делать этого.

Сервер, получивший блок полей, размер которого превышает тот, который сервер готов обработать, может передать код статуса HTTP 431 (Request Header Fields Too Large) [RFC6585]. Клиент может отбрасывать отклики, которые он не может обработать. Блок полей **должен** быть обработан для обеспечения согласованного (consistent) состояния соединения, если только соединение не закрыто.

10.5.2. Проблемы CONNECT

Метод CONNECT можно использовать для создания непропорциональной нагрузки на прокси, поскольку создание потока относительно недорого по сравнению с созданием и поддержкой соединения TCP. Прокси может также поддерживать некоторые ресурсы для соединения TCP после закрытия потока, в котором передан запрос CONNECT, поскольку исходящее соединение TCP остаётся в состоянии TIME_WAIT. Поэтому прокси не может полагаться только на SETTINGS_MAX_CONCURRENT_STREAMS для ограничения ресурсов, потребляемых запросами CONNECT.

10.6. Использование сжатия

Сжатие может позволить атакующему восстановить секретные данные, если они сжаты в одном контексте с данными, контролируемыми им. HTTP/2 разрешает сжатие строк полей (параграф 4.3), а приведённые ниже соображения относятся также к использованию в HTTP кодирования содержимого со сжатием (параграф 8.4.1 в [HTTP]).

Существуют наглядные атаки на сжатие, использующие свойства Web (например, [BREACH]). Атакующий создаёт множество запросов с различными открытыми данными (plaintext), наблюдая для каждого размер полученного шифротекста (ciphertext), который будет более коротким при угаданном секрете.

Реализациям, обменивающимся данными по защищённому каналу, **недопустимо** сжимать содержимое, включающее как конфиденциальные, так и контролируемые злоумышленником данные, если только для каждого источника данных не применяются свои словари. **Недопустимо** сжимать данные, если их источник невозможно определить достоверно. Базовое сжатие потоков, такое как предоставляемое TLS, **недопустимо** использовать с HTTP/2 (см. параграф 9.2).

Дополнительные соображения безопасности применительно к сжатию полей представлены в [COMPRESSION].

10.7. Использование заполнения

Заполнение в HTTP/2 не предназначено для замены заполнения общего назначения, вроде предоставляемого TLS [TLS13]. Избыточное заполнение может быть даже вредным (counterproductive). Корректность применения может зависть от наличия конкретных сведений о данных для дополнения.

Для смягчения атак, основанных на сжатии, запрет компрессии может оказаться полезней, чем заполнение.

Заполнение может служить для сокрытия точного размера содержимого кадра и предназначено для смягчения определённых атак в HTTP, например, атак, где сжатое содержимое включает контролируемый злоумышленником открытый текст и секретные данные (например, [BREACH]).

Использование заполнения может предоставлять более слабую защиту, чем кажется на первый взгляд. В лучшем случае заполнение лишь усложняет для атакующего получение сведений о размере за счёт увеличения числа кадров, которые тому приходится наблюдать. Некорректно реализованные схемы заполнения легко преодолеваются. В частности, случайное заполнение с предсказуемым распределением обеспечивает лишь очень слабую защиту, а заполнение до определённого размера раскрывает сведения о размере, когда кадр превосходит заданный предел, что может быть полезно для злоумышленника, контролирующего открытый текст.

Посредникам **следует** сохранять заполнение в кадрах DATA, но они **могут** исключать его из кадров HEADERS и PUSH_PROMISE. Уважительной причиной изменения размера заполнения посредником является повышение уровня защиты, обеспечиваемой заполнением.

10.8. Вопросы приватности

Некоторые особенности HTTP/2 дают наблюдателю возможность составить временную картину действий клиента или сервера. Это включает значения настроек, манеру поддержки окна управления потоком данных, способ распределения приоритетов между потоками, время реакции на воздействия и работу функций, управляемых настройками. По мере возникновения наблюдаемых различий в поведении эти свойства могут стать основой для «отпечатка» конкретного клиента, как указано в параграфе 3.2 [PRIVACY].

Предпочтение HTTP/2 использовать одно соединение TCP позволяет отслеживать активность пользователя на сайте. Повторное использование соединений для других источников позволяет отслеживать эти источники.

Поскольку кадры PING и SETTINGS запрашивают незамедлительные отклики, любая конечная точка может применять их для измерения задержки между собой и партнёром, что может в некоторых случаях влиять на приватность.

10.9. Удалённые атаки по времени

Удалённые атаки по времени извлекают секреты серверов путём наблюдения за вариациями времени обработки запросов, включающих использование секретов. HTTP/2 позволяет одновременно создавать и обрабатывать запросы, что может дать злоумышленнику возможность точнее контролировать время начала обработки запроса. Несколько запросов HTTP/2 могут быть включены в один пакет IP или запись TLS, поэтому HTTP/2 может снизить эффективность удалённых атак по времени, устраняя вариативность доставки запросов и оставляя лишь их порядок и доставку откликов в качестве источника сведений об изменчивости.

Независимость времени обработки от значения секрета является лучшей формой защиты от атак по времени.

11. Взаимодействие с IANA

В этой редакции HTTP/2 поле заголовка HTTP2-Settings и маркер обновления h2c из [RFC7540] признаны устаревшими.

Раздел 11 [RFC7540] регистрирует идентификаторы ALPN h2 и h2c вместе с методом HTTP PRI. В RFC 7540 также организован реестр для типов кадров, параметров и кодов ошибок. Эти регистрации и реестры применяются к HTTP/2, но не переопределяются этим документом.

Агентство IANA обновило ссылки на RFC 7540 указанием на этот документ в реестрах TLS Application-Layer Protocol Negotiation (ALPN) Protocol IDs, HTTP/2 Frame Type, HTTP/2 Settings, HTTP/2 Error Code, HTTP Method Registry. Регистрация метода PRI обновлена со ссылкой на параграф 3.4, остальные номера параграфов не изменились.

Агентство IANA изменило правила для тех частей HTTP/2 Frame Type и HTTP/2 Settings, которые были зарезервированы в RFC 7540 для экспериментов (Experimental Use), и сейчас для них применяется та же процедура, что м для остальной части каждого реестра.

11.1. Регистрация поля заголовка HTTP2-Settings

В этом параграфе признано устаревшим поле заголовка HTTP2-Settings, зарегистрированное параграфом 11.5 [RFC7540] в реестре Hypertext Transfer Protocol (HTTP) Field Name Registry. Это свойство может быть удалено (см. параграф 3.1). регистрация обновлена для включения деталей, требуемых параграфом 18.4 в [HTTP]:

```
Field Name:  HTTP2-Settings
Status:  obsoleted
Reference:  параграф 3.2.1 в [RFC7540]
Comments:  Устарело, см. параграф 11.1 в этом документе.
```

11.2. Маркер обновления h2c

В этом параграфе признан устаревшим маркер обновления h2c, включенный параграфом 11.8 [RFC7540] в реестр Hypertext Transfer Protocol (HTTP) Upgrade Token Registry. Это свойство может быть удалено (см. параграф 3.1). Обновление регистрации приведено ниже.

```
Value:  h2c
Description:  (OBSOLETE) Hypertext Transfer Protocol version 2 (HTTP/2)
Expected Version Tokens:  None
Reference:  Section 3.1 of this document
```

12. Литература

12.1. Нормативные документы

[CACHING] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/info/rfc9111>.

[COMPRESSION] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

[COOKIE] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[HTTP] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[QUIC] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier", RFC 8422, DOI 10.17487/RFC8422, August 2018, <https://www.rfc-editor.org/info/rfc8422>.

[RFC8470] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <https://www.rfc-editor.org/info/rfc8470>.

[TCP] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[TLS-ALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[TLS-ECDHE] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", RFC 5289, DOI 10.17487/RFC5289, August 2008, <https://www.rfc-editor.org/info/rfc5289>.

[TLS-EXT] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[TLS12] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[TLSBCP] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <https://www.rfc-editor.org/info/rfc7525>.

12.2. Дополнительная литература

[ALT-SVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc7838>.

[BREACH] Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the CRIME Attack", 12 July 2013, <https://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>.

[DNS-TERMS] Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <https://www.rfc-editor.org/info/rfc8499>.

[HTTP-PRIORITY] Oku, K. and L. Pardue, "Extensible Prioritization Scheme for HTTP", RFC 9218, DOI 10.17487/RFC9218, June 2022, <https://www.rfc-editor.org/info/rfc9218>.

[HTTP/1.1] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", STD 99, RFC 9112, DOI 10.17487/RFC9112, June 2022, <https://www.rfc-editor.org/info/rfc9112>.

[NFLX-2019-002] Netflix, "HTTP/2 Denial of Service Advisory", 13 August 2019, <https://github.com/Netflix/security-bulletins/blob/master/advisories/third-party/2019-002.md>.

[PRIVACY] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/info/rfc6973>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <https://www.rfc-editor.org/info/rfc1122>.

[RFC3749] Hollenbeck, S., "Transport Layer Security Protocol Compression Methods", RFC 3749, DOI 10.17487/RFC3749, May 2004, <https://www.rfc-editor.org/info/rfc3749>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <https://www.rfc-editor.org/info/rfc6125>.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <https://www.rfc-editor.org/info/rfc6585>.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <https://www.rfc-editor.org/info/rfc7323>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC8441] McManus, P., "Bootstrapping WebSockets with HTTP/2", RFC 8441, DOI 10.17487/RFC8441, September 2018, <https://www.rfc-editor.org/info/rfc8441>.

[RFC8740] Benjamin, D., "Using TLS 1.3 with HTTP/2", RFC 8740, DOI 10.17487/RFC8740, February 2020, <https://www.rfc-editor.org/info/rfc8740>.

[TALKING] Huang, L., Chen, E., Barth, A., Rescorla, E., and C. Jackson, "Talking to Yourself for Fun and Profit", 2011, <https://www.adambarth.com/papers/2011/huang-chen-barth-rescorla-jackson.pdf>.

Приложение A. Запрещённые шифры TLS 1.2

Реализация HTTP/2 **может** считать согласование любого из приведённых ниже шифров для TLS 1.2 ошибкой соединения (параграф 5.4.1) типа INADEQUATE_SECURITY.

```
TLS_NULL_WITH_NULL_NULL
TLS_RSA_WITH_NULL_MD5
TLS_RSA_WITH_NULL_SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5
TLS_RSA_WITH_RC4_128_MD5
TLS_RSA_WITH_RC4_128_SHA
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
TLS_RSA_WITH_IDEA_CBC_SHA
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_RSA_WITH_DES_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_DSS_WITH_DES_CBC_SHA
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_RSA_WITH_DES_CBC_SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_RSA_WITH_DES_CBC_SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5
TLS_DH_anon_WITH_RC4_128_MD5
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_anon_WITH_DES_CBC_SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA
TLS_KRB5_WITH_DES_CBC_SHA
TLS_KRB5_WITH_3DES_EDE_CBC_SHA
TLS_KRB5_WITH_RC4_128_SHA
TLS_KRB5_WITH_IDEA_CBC_SHA
TLS_KRB5_WITH_DES_CBC_MD5
TLS_KRB5_WITH_3DES_EDE_CBC_MD5
TLS_KRB5_WITH_RC4_128_MD5
TLS_KRB5_WITH_IDEA_CBC_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA
TLS_KRB5_EXPORT_WITH_RC4_40_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_RC4_40_MD5
TLS_PSK_WITH_NULL_SHA
TLS_DHE_PSK_WITH_NULL_SHA
TLS_RSA_PSK_WITH_NULL_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA
TLS_DH_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DH_anon_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_DH_DSS_WITH_AES_256_CBC_SHA
TLS_DH_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_NULL_SHA256
TLS_RSA_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256
TLS_DH_DSS_WITH_AES_128_CBC_SHA256
TLS_DH_RSA_WITH_AES_128_CBC_SHA256
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
TLS_DH_DSS_WITH_AES_256_CBC_SHA256
TLS_DH_RSA_WITH_AES_256_CBC_SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
TLS_DH_anon_WITH_AES_128_CBC_SHA256
TLS_DH_anon_WITH_AES_256_CBC_SHA256
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA
TLS_PSK_WITH_RC4_128_SHA
TLS_PSK_WITH_3DES_EDE_CBC_SHA
TLS_PSK_WITH_AES_128_CBC_SHA
TLS_PSK_WITH_AES_256_CBC_SHA
TLS_DHE_PSK_WITH_RC4_128_SHA
TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA
TLS_DHE_PSK_WITH_AES_128_CBC_SHA
TLS_DHE_PSK_WITH_AES_256_CBC_SHA
TLS_RSA_PSK_WITH_RC4_128_SHA
TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA
TLS_RSA_PSK_WITH_AES_128_CBC_SHA
TLS_RSA_PSK_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_SEED_CBC_SHA
TLS_DH_DSS_WITH_SEED_CBC_SHA
```

```
TLS_DH_RSA_WITH_SEED_CBC_SHA
TLS_DHE_DSS_WITH_SEED_CBC_SHA
TLS_DHE_RSA_WITH_SEED_CBC_SHA
TLS_DH_anon_WITH_SEED_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_DH_RSA_WITH_AES_128_GCM_SHA256
TLS_DH_RSA_WITH_AES_256_GCM_SHA384
TLS_DH_DSS_WITH_AES_128_GCM_SHA256
TLS_DH_DSS_WITH_AES_256_GCM_SHA384
TLS_DH_anon_WITH_AES_128_GCM_SHA256
TLS_DH_anon_WITH_AES_256_GCM_SHA384
TLS_PSK_WITH_AES_128_GCM_SHA256
TLS_PSK_WITH_AES_256_GCM_SHA384
TLS_RSA_PSK_WITH_AES_128_GCM_SHA256
TLS_RSA_PSK_WITH_AES_256_GCM_SHA384
TLS_PSK_WITH_AES_128_CBC_SHA256
TLS_PSK_WITH_AES_256_CBC_SHA384
TLS_PSK_WITH_NULL_SHA256
TLS_PSK_WITH_NULL_SHA384
TLS_DHE_PSK_WITH_AES_128_CBC_SHA256
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
TLS_DHE_PSK_WITH_NULL_SHA256
TLS_DHE_PSK_WITH_NULL_SHA384
TLS_RSA_PSK_WITH_AES_128_CBC_SHA256
TLS_RSA_PSK_WITH_AES_256_CBC_SHA384
TLS_RSA_PSK_WITH_NULL_SHA256
TLS_RSA_PSK_WITH_NULL_SHA384
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA256
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA256
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA256
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA256
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA256
TLS_EMPTY_RENEGOTIATION_INFO_SCSV
TLS_ECDH_ECDSA_WITH_NULL_SHA
TLS_ECDH_ECDSA_WITH_RC4_128_SHA
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_NULL_SHA
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDH_RSA_WITH_NULL_SHA
TLS_ECDH_RSA_WITH_RC4_128_SHA
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_RSA_WITH_NULL_SHA
TLS_ECDHE_RSA_WITH_RC4_128_SHA
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_ECDH_anon_WITH_NULL_SHA
TLS_ECDH_anon_WITH_RC4_128_SHA
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA
TLS_ECDH_anon_WITH_AES_128_CBC_SHA
TLS_ECDH_anon_WITH_AES_256_CBC_SHA
TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA
TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA
TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA
TLS_SRP_SHA_WITH_AES_128_CBC_SHA
TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA
TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA
TLS_SRP_SHA_WITH_AES_256_CBC_SHA
TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA
TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_PSK_WITH_RC4_128_SHA
TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384
TLS_ECDHE_PSK_WITH_NULL_SHA
TLS_ECDHE_PSK_WITH_NULL_SHA256
TLS_ECDHE_PSK_WITH_NULL_SHA384
TLS_RSA_WITH_ARIA_128_CBC_SHA256
TLS_RSA_WITH_ARIA_256_CBC_SHA384
TLS_DH_DSS_WITH_ARIA_128_CBC_SHA256
TLS_DH_DSS_WITH_ARIA_256_CBC_SHA384
TLS_DH_RSA_WITH_ARIA_128_CBC_SHA256
TLS_DH_RSA_WITH_ARIA_256_CBC_SHA384
TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256
TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384
TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256
TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384
TLS_DH_anon_WITH_ARIA_128_CBC_SHA256
TLS_DH_anon_WITH_ARIA_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_ARIA_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_ARIA_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_ARIA_256_CBC_SHA384
TLS_ECDH_RSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDH_RSA_WITH_ARIA_256_CBC_SHA384
TLS_RSA_WITH_ARIA_128_GCM_SHA256
TLS_RSA_WITH_ARIA_256_GCM_SHA384
TLS_DH_RSA_WITH_ARIA_128_GCM_SHA256
TLS_DH_RSA_WITH_ARIA_256_GCM_SHA384
TLS_DH_DSS_WITH_ARIA_128_GCM_SHA256
TLS_DH_DSS_WITH_ARIA_256_GCM_SHA384
TLS_DH_anon_WITH_ARIA_128_GCM_SHA256
TLS_DH_anon_WITH_ARIA_256_GCM_SHA384
TLS_ECDH_ECDSA_WITH_ARIA_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_ARIA_256_GCM_SHA384
TLS_ECDH_RSA_WITH_ARIA_128_GCM_SHA256
TLS_ECDH_RSA_WITH_ARIA_256_GCM_SHA384
TLS_PSK_WITH_ARIA_128_CBC_SHA256
TLS_PSK_WITH_ARIA_256_CBC_SHA384
TLS_DHE_PSK_WITH_ARIA_128_CBC_SHA256
TLS_DHE_PSK_WITH_ARIA_256_CBC_SHA384
TLS_RSA_PSK_WITH_ARIA_128_CBC_SHA256
TLS_RSA_PSK_WITH_ARIA_256_CBC_SHA384
TLS_PSK_WITH_ARIA_128_GCM_SHA256
TLS_PSK_WITH_ARIA_256_GCM_SHA384
TLS_RSA_PSK_WITH_ARIA_128_GCM_SHA256
TLS_RSA_PSK_WITH_ARIA_256_GCM_SHA384
TLS_ECDHE_PSK_WITH_ARIA_128_CBC_SHA256
TLS_ECDHE_PSK_WITH_ARIA_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDH_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDH_RSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_DH_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_DH_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_DH_DSS_WITH_CAMELLIA_128_GCM_SHA256
TLS_DH_DSS_WITH_CAMELLIA_256_GCM_SHA384
TLS_DH_anon_WITH_CAMELLIA_128_GCM_SHA256
TLS_DH_anon_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDH_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDH_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_PSK_WITH_CAMELLIA_128_GCM_SHA256
TLS_PSK_WITH_CAMELLIA_256_GCM_SHA384
TLS_RSA_PSK_WITH_CAMELLIA_128_GCM_SHA256
TLS_RSA_PSK_WITH_CAMELLIA_256_GCM_SHA384
TLS_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_DHE_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_DHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_RSA_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_RSA_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDHE_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
```

```
TLS_RSA_WITH_AES_128_CCM          TLS_PSK_WITH_AES_128_CCM
TLS_RSA_WITH_AES_256_CCM          TLS_PSK_WITH_AES_256_CCM
TLS_RSA_WITH_AES_128_CCM_8        TLS_PSK_WITH_AES_128_CCM_8
TLS_RSA_WITH_AES_256_CCM_8        TLS_PSK_WITH_AES_256_CCM_8
```

Примечание. Список шифров составлен на основе набора зарегистрированных шифров TLS на момент разработки [RFC7540]. Список включает шифры, не предоставляющие обмен эфемерными ключами, а также шифры, основанные пустом (null), потоковом и блочном шифровании TLS (см. параграф 6.2.3 в [TLS12]). Могут быть заданы дополнительные шифры с такими свойствами, которые не будут запрещены явно.

Дополнительные сведения приведены в параграфе 9.2.2.

Приложение B. Отличия от RFC 7540

- Использование TLS 1.3 было задано на основе [RFC8740], отменённого данным документом.
- Схема приоритетов из RFC 7540 устарела. Сохранены определения формата кадра PRIORITY и полей приоритета в кадрах HEADERS, а также правила отправки и получения кадров PRIORITY, но семантика полей описана только в RFC 7540. Схема сигнализации приоритета из RFC 7540 оказалась неуспешной и рекомендуется более простая сигнализация [HTTP-PRIORITY].
- Механизм HTTP/1.1 Upgrade устарел и исключён из этого документа. Он не был широко развернут, поскольку пользователи HTTP/2 с открытым текстом предпочитали реализацию предварительных сведений.
- Сужена проверка имён и значений полей, с точным указанием, когда она обязательна для посредников, и изменены сообщения об ошибках в запросах с рекомендацией передавать коды статуса группы 400.
- Диапазоны кодов установок и типов кадров, зарезервированные для экспериментов (Experimental Use) сделаны доступными для общего пользования.
- Специфичные для соединения поля заголовка, которые были запрещены, указаны более точно и полно.
- Значения Host и :authority больше не могут различаться (запрет).
- В параграфе 4.3.1 уточнены правила отправки инструкций по обновления размера динамической таблицы после смены настроек.

Внесены также редакционные изменения. В частности, поменялась терминология и структура документа в результате изменения семантического ядра HTTP [HTTP]. Документы теперь включают некоторые концепции, заданные в RFC 7540, такие как код статуса 421 и слияние (coalescing) соединений.

Благодарности

Заслуги внесения нетривиального вклада в этот документ принадлежат большому числу людей, годами участвовавших в работе группы HTTP. В [RFC7540] приведён более полный список людей, заслуживающих признания за их вклад.

Участники работы

Mike Belshe и Roberto Peon создали текст, послуживший основой для этого документа.

Адреса авторов

Martin Thomson (editor)
Mozilla
Australia
Email: mt@lowentropy.net

Cory Benfield (editor)
Apple Inc.
Email: cbenfield@apple.com

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru