Internet Engineering Task Force (IETF)
Request for Comments: 7540
Category: Standards Track
ISSN: 2070-1721

M. Belshe
BitGo
R. Peon
Google, Inc
M. Thomson, Ed.
Mozilla

May 2015

Протокол передачи гипертекста версии 2 (HTTP/2)

Hypertext Transfer Protocol Version 2 (HTTP/2)

Аннотация

Эта спецификация описывает оптимизированное выражение семантики протокола передачи гипертекста (HTTP[1]), называемого HTTP версии 2 (HTTP/2). HTTP/2 обеспечивает более эффективное использование сетевых ресурсов и снижение воздействия задержек за счёт добавления сжатия, а также позволяет организовать множество одновременных обменов через одно соединение. В новой версии также добавлено «выталкивание» представлений с сервера клиенту без запроса.

Данная спецификация дополняет, но не отменяет синтаксис сообщений HTTP/1.1. Существующая семантика HTTP сохранена без изменений.

Статус документа

Документ является проектом стандарт Internet (Internet Standards Track).

Документ является результатом работы IETF[2] и представляет согласованный взгляд сообщества IETF. Документ прошёл открытое обсуждение и был одобрен для публикации IESG[3]. Не все одобренные IESG документы претендуют на статус Internet Standard (см. раздел 2 в RFC 5741).

Информацию о текущем статусе документа, ошибках и способах обратной связи можно найти по ссылке http://www.rfc-editor.org/info/rfc7540.

Оглавление

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

1. Введение

Протокол передачи гипертекста HTTP чрезвычайно успешен. Однако способы использования нижележащего транспорта в HTTP/1.1 (раздел 6 в [RFC7230]) имеют некоторые характеристики, оказывающие отрицательное влияние на производительность современных приложений.

В частности, HTTP/1.0 разрешает в каждый момент времени только один незавершённый запрос на соединение TCP. В HTTP/1.1 добавлен конвейер для запросов (pipelining), но это лишь частично решает проблему одновременных запросов и продолжает вызывать блокировку (head-of-line). Следовательно, клиенты HTTP/1.0 и HTTP/1.1, которым нужно сделать множество запросов, вынуждены организовывать множественные соединения с сервером, чтобы обеспечить их параллельную обработку и снизить задержку.

Кроме того, поля заголовков HTTP зачастую многословны и содержат повторы, что ведёт к бессмысленному росту сетевого трафика, приводящему к быстрому заполнению начального окна насыщения TCP [TCP]. Это может приводить к избыточным задержкам при отправке множества запросов через новое соединение TCP.

HTTP/2 решает эти проблемы, определяя оптимизированное отображение семантики HTTP на нижележащие уровни. В частности, разрешается чередование сообщений с запросами и откликами в одном соединении и применяется более эффективное кодирование полей заголовка HTTP. Кроме того, обеспечивается приоритизация запросов, что позволяет в первую очередь отправить более важные запросы с соответствующим ростом производительности.

Новый протокол более «дружелюбен» к сети за счёт использования меньшего числа соединений TCP по сравнению с HTTP/1.x. Это снижает конкуренцию с другими потоками и долгосрочными соединениями, что, в свою очередь, повышает эффективность использования пропускной способности сети.

Наконец, HTTP/2 обеспечивает возможность более эффективной обработки сообщений за счёт применения двоичного кадрирования сообщений.

2. Обзор протокола HTTP/2

HTTP/2 обеспечивает оптимизированный транспорт для семантики HTTP. HTTP/2 поддерживает все основные свойства HTTP/1.1, но его целью является повышение эффективности несколькими способами.

Базовым блоком протокола HTTP/2 является кадр (frame, см. параграф 4.1). каждый тип кадров служит для своих целей. Например, кадры HEADERS и DATA служат базой для запросов и откликов HTTP (параграф 8.1), а другие кадры типа SETTINGS, WINDOW_UPDATE и PUSH_PROMISE служат для поддержки иных функций HTTP/2.

Мультиплексирование запросов обеспечивается связыванием каждого обмена HTTP «запрос-отклик» со своим потоком (stream, см. раздел 5). Потоки в значительной мере не зависимы один от другого, поэтому блокировка или приостановка того или иного запроса или отклика не препятствует работе других потоков.

Управление потоком и приоритизация обеспечивают возможность эффективного использования мультиплексируемых потоков. Управление потоком (параграф 5.2) помогает обеспечить передачу только данных, которые могут быть использованы получателем. Приоритизация (параграф 5.3) обеспечивает возможность направить ограниченные ресурсы сначала на более важные потоки.

HTTP/2 добавляет новый режим взаимодействия, в котором сервер может «выталкивать» (push) отклики для клиента (параграф 8.2). Выталкивание позволяет серверу по своему усмотрению передать данные, которые сервер считает требующимися для клиента, что требует некого компромисса между загрузкой сети и снижением задержки. Сервер делает это, синтезируя запрос, который он передаёт, как кадр PUSH_PROMISE. После этого сервер имеет возможность отправить отклик на синтезированный запрос в отдельном потоке.

Поскольку используемые в соединении поля заголовков HTTP могут включать большой объем избыточных данных, содержащие эти поля кадры сжимаются (параграф 4.3). В общем случае это очень полезно для запросов, поскольку позволяет сжать многочисленные запросы в один пакет.

2.1. Организация документа

Спецификация HTTP/2 разделена на 4 части:

- начало работы HTTP/2 (раздел 3) - инициирование соединений HTTP/2;
- уровни кадров (раздел 4) и потоков (раздел 5), описывающие структуру кадров HTTP/2 и формирование из них мультиплексируемых потоков;
- определения кадров (раздел 6) и ошибок (раздел 7) подробно описывают типы кадров и ошибок, применяемые в HTTP/2;
- отображения HTTP (раздел 8) и дополнительные требования (раздел 9) описывают выражение семантики HTTP с использованием кадров и потоков.

Хотя часть концепций кадров и потоков отделена от HTTP, данная спецификация не определяет полностью базовый уровень кадров. Уровни кадров и потоков адаптированы к потребностям протокола HTTP и выталкивания данных с серверов.

2.2. Соглашения и термины

Ключевые слова **необходимо** (MUST), **недопустимо** (MUST NOT), **требуется** (REQUIRED), **нужно** (SHALL), **не нужно** (SHALL NOT), **следует** (SHOULD), **не следует** (SHOULD NOT), **рекомендуется** (RECOMMENDED), **возможно** (MAY), **необязательно** (OPTIONAL) в данном документе должны интерпретироваться в соответствии с RFC 2119 [RFC2119].

Все числовые значения используют сетевой порядок байтов и предполагаются беззнаковыми, пока явно не указано иное. Литералы приводятся в десятичном или шестнадцатеричном формате. Шестнадцатеричные значения указываются префиксом 0x.

Ниже приведены определения основных терминов, используемых в документе.

client - клиент
Конечная точка, выступающая инициатором соединения HTTP/2. Клиенты передают запросы HTTP и получают отклики HTTP.

connection - соединение
Соединение транспортного уровня между двумя конечными точками.

connection error - ошибка соединения
Ошибка, воздействующая на соединение HTTP/2 в целом.

endpoint - конечная точка
Клиент или сервер в данном соединении.

frame - кадр
Наименьший коммуникационный блок в соединении HTTP/2, состоящий из заголовка и последовательности октетов переменного размера в зависимости от типа кадра.

peer - партнёр
Конечная точка. При обсуждении конкретной конечной точки термин «партнёр» используется для конечной точки на другой стороне соединения.

receiver - получатель
Конечная точка, получающая кадры.

sender - отправитель
Конечная точка, передающая кадры.

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

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

stream error - ошибка потока
Ошибка отдельного потока HTTP/2.

Термины gateway (шлюз), intermediary (посредник), proxy (прокси) и tunnel (туннель) определены в параграфе 2.3 [RFC7230]. Посредники в разные моменты могут выступать как клиент или как сервер.

Термин payload body (тело данных) определён в параграфе 3.3 [RFC7230].

3. Начало работы HTTP/2

Соединение HTTP/2 представляет собой протокол прикладного уровня, работающий через соединение TCP ([TCP]). Клиент является инициатором соединения TCP.

HTTP/2 использует те же схемы http и https для URI, которые применяются в HTTP/1.1. По умолчанию в HTTP/2 используются те же номера портов - 80 для http URI и 443 для https URI. В результате от реализации, обрабатывающей запрос для URI целевого ресурса типа http://example.org/foo или https://example.com/bar, требуется сначала определить, поддерживает ли восходящий сервер (партнёр, с которым клиент желает организовать непосредственное соединение) HTTP/2.

Способы определения поддержки HTTP/2 различаются для http и https URI. Обнаружение поддержки для http URI описано в параграфе 3.2, для https URI - в параграфе 3.3.

3.1. Идентификация версии HTTP/2

Определённый в этом документе протокол имеет два идентификатора.

- Строка h2 указывает протокол, где HTTP/2 использует защиту транспортного уровня (TLS[1]) [TLS12]. Этот идентификатор применяется в поле согласования TLS прикладного уровня (ALPN[2]) [TLS-ALPN] и в любом другом месте, где указывается HTTP/2 «поверх» TLS.

  Строка h2 включается в идентификатор протокола ALPN в форме последовательности из 2 октетов 0x68, 0x32.

- Строка h2c указывает протокол, где HTTP/2 работает на основе открытого TCP. Этот идентификатор применяется в поле заголовка HTTP/1.1 Upgrade и в любых других местах, где указывается HTTP/2 на базе TCP.

  Строка h2c зарезервирована из пространства ALPN, на протокол не использует TLS.

Согласование h2 или h2c предполагает использование транспорта, защиты, кадрирования и семантики сообщений, описанных в этом документе.

3.2. Начало работы HTTP/2 для http URI

Клиент, делающий запрос для http URI, не зная о поддержке HTTP/2 на следующем интервале, использует механизм HTTP Upgrade (параграф 6.7 d [RFC7230]). Это выполняется путём отправки запроса HTTP/1.1, включающего поле заголовка Upgrade с маркером h2c. Такой запрос HTTP/1.1 **должен** включать в точности одно поле заголовка HTTP2-Settings (параграф 3.2.1) .

Например,

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

[1]Transport Layer Security.
[2]Application-layer protocol negotiation.

Запросы, содержащие информационное тело, **должны** быть переданы целиком до того, как клиент сможет передавать кадры HTTP/2. Это означает, что большой запрос может заблокировать соединение до тех пор, пока он не будет передан полностью.

Если важна одновременная передача начального и последующих запросов, можно использовать запрос OPTIONS для перехода на HTTP/2 за счёт задержки в один период кругового обхода.

Не поддерживающий HTTP/2 сервер может передать в ответ на запрос отклик без поля Upgrade в заголовке.

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html
...
```

Сервер **должен** игнорировать маркер h2 в поле заголовка Upgrade. Присутствие маркера h2 предполагает работу HTTP/2 через TLS и для этого случая согласование происходит иначе (см. параграф 3.3).

Сервер, поддерживающий HTTP/2, воспринимает Upgrade с откликом 101 (переключение протокола). После пустой строки, завершающей отклик 101, сервер может начать передачу кадров HTTP/2. Эти кадры **должны** включать отклик на запрос, вызвавший переключение протокола (Upgrade).

Например,

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
```

Первым кадром HTTP/2 от сервера **должен** быть серверный префикс соединения (параграф 3.5), состоящий из кадра SETTINGS (параграф 6.5). При получении отклика 101 клиент **должен** передать префикс соединения (параграф 3.5), включающий кадр SETTINGS.

Запросу HTTP/1.1, переданному до Upgrade, присваивается идентификатор потока 1 (параграф 5.1.1) с принятыми по умолчанию значениями (параграф 5.3.5). Поток 1 является неявно «полузакрытым» соединением от клиента к серверу (параграф 5.1), поскольку запрос завершается, как HTTP/1.1. После начала работы соединения HTTP/2 поток 1 используется для отклика.

3.2.1. Поле заголовка HTTP2-Settings

Запросы для перехода от HTTP/1.1 к HTTP/2 (обновления) **должны** включать в точности одно поле заголовка HTTP2-Settings. Это поле зависит от контекста и включает параметры, управляющие соединением HTTP/2 в предположении, что сервер примет запрос на переход.

```
HTTP2-Settings    = token68
```

Серверу **недопустимо** переводить соединение на HTTP/2, если данное поле отсутствует в заголовке или задано в нескольких экземплярах. Серверам **недопустимо** передавать это поле в заголовке.

Содержимое поля заголовка HTTP2-Settings - это информационные элементы (payload) кадра SETTINGS (параграф 6.5), представленные в форме строки base64url (т. е., URL и безопасное для имён файлов представление Base64, описанное в разделе 5 [RFC4648] без трейлерных символов =). Создание ABNF [RFC5234] для token68 определено в параграфе 2.1 [RFC7235].

Поскольку обновление предназначено лишь для следующего незамедлительно соединения, передающий поле HTTP2-Settings клиент **должен** также передать HTTP2-Settings в качестве опции соединения в поле заголовка Connection для предотвращения пересылки (см. параграф 6.1 в [RFC7230]).

Сервер декодирует и интерпретирует эти значения, как и остальные значения в кадре SETTINGS. Явное подтверждение этих установок не требуется (параграф 6.5.3), поскольку отклик 101 служит неявным подтверждением. Предоставление этих значений в запросе Upgrade даёт пользователю возможность обеспечить параметры до получения от сервера каких-либо кадров.

3.3. Начало работы HTTP/2 для https URI

Клиент, делающий запрос для https URI, использует TLS [TLS12] с расширением ALPN[1] [TLS-ALPN].

В HTTP/2 на основе TLS используется идентификатор протокола h2. Идентификатор h2c **недопустимо** передавать от клиентов или выбирать на сервере - этот идентификатор описывает протокол, не использующий TLS.

После завершения согласования TLS клиент и сервер **должны** передать префикс соединения (параграф 3.5).

3.4. Начало работы с заранее известным HTTP/2

Клиент может получить информацию о поддержке конкретным сервером HTTP/2 иными способами. Например, в [ALT-SVC] описан механизм анонсирования такой поддержки.

Клиент **должен** отправить префикс соединения (параграф 3.5) и сразу после этого **может** передавать такому серверу кадры HTTP/2. Сервер может идентифицировать такие соединения по полученным префиксам. Это оказывает влияние лишь на организацию соединений HTTP/2 через открытые (не шифрованные) соединения TCP, а реализации, поддерживающие HTTP/2 через TLS **должны** использовать согласование протокола в TLS [TLS-ALPN].

Аналогично, сервер **должен** передать префикс соединения (параграф 3.5).

Без дополнительной информации известная заранее поддержка сервером HTTP/2 вовсе не означает, что он будет поддерживать такие соединения и впредь. Например, конфигурация сервера может измениться, на разных экземплярах кластеризованного сервера конфигурации могут отличаться, могут также измениться условия в сети.

[1]Application-layer protocol negotiation - согласование протокола на прикладном уровне.

3.5. Префикс соединения HTTP/2

В HTTP/2 от каждой из оконечных точек требуется отправить префикс соединения в качестве финального подтверждения использования протокола для организации начальных установок соединения HTTP/2. Клиент и сервер передают разные префиксы соединения.

Клиентский префикс начинается с последовательности из 24 октетов с шестнадцатеричным представлением

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

Таким образом, префикс соединения начинается со строки «PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n[1]». За этой последовательностью **должен** быть передан кадр SETTINGS (параграф 6.5), который **может** быть пустым. Клиент передаёт свой префикс соединения непосредственно по получении отклика 101 (Switching Protocols), указывающего успешную смену протокола, или в качестве первых октетов данных приложения в соединении TLS. Если соединение HTTP/2 начинается до получения информации о поддержке сервером этого протокола, клиентский префикс соединения передаётся сразу после организации соединения.

> Примечание. Клиентский префикс соединения выбран так, что большая часть серверов HTTP/1.1 и HTTP/1.0, а также посредников не будет пытаться обработать последующие кадры. Однако это не решает вопросов, поднятых в [TALKING].

Серверный префикс соединения состоит из (возможно пустого) кадра SETTINGS (параграф 6.5), который **должен** быть первым кадром от сервера в соединении HTTP/2.

Кадры SETTINGS, полученные от партнёра, как часть префикса соединения, **должны** подтверждаться (см. параграф 6.5.3) после отправки префикса соединения.

Для предотвращения ненужных задержек клиентам разрешается передача дополнительных кадров серверу сразу же после отправки клиентского префикса соединения, не дожидаясь получения префикса от сервера. Однако важно подчеркнуть, что кадр SETTINGS с префиксом соединения сервера может включать параметры, которые обязательно изменят способ взаимодействия клиента с сервером. Предполагается, что после получения кадра SETTINGS клиент будет соблюдать установленные параметры. В некоторых конфигурациях сервер может передать кадр SETTINGS до того, как клиент передаст дополнительные кадры, что позволяет избежать упомянутой сложности.

Клиенты и серверы должны трактовать неприемлемый префикс соединения, как ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR. В этом случае кадр GOAWAY (параграф 6.8) **может** быть опущен, поскольку неприемлемый префикс показывает, что партнёр не использует HTTP/2.

4. Кадры HTTP

После того, как соединение HTTP/2 организовано, его оконечные точки начинают обмен кадрами.

4.1. Формат кадра

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

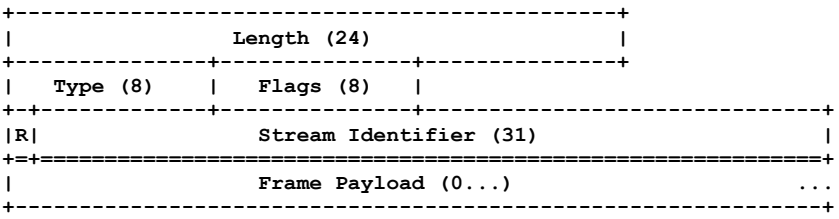

Рисунок 1. Схема кадра.

Поля фиксированного заголовка определены ниже.

Length

Размер информационных полей кадра (frame payload), выраженный 24-битовым целым числом без знака. Кадры размером более 2^{14} (16384) **недопустимо** передавать, если получатель не установил большее значение для параметра SETTINGS_MAX_FRAME_SIZE.

9 октетов заголовка кадра не учитываются в поле размера.

Type

8-битовый идентификатор типа кадра, определяющего его формат и значение. Реализации **должны** игнорировать и отбрасывать любые кадры неизвестного типа.

Flags

8-битовое поле, зарезервированное для логических флагов, определяемых типом кадра.

Семантика флагов указывается типом кадра. Не определённые семантикой соответствующего типа флаги **должны** игнорироваться, а при передаче для них **должно** устанавливаться значение 0 (0x0).

R

1-битовое резервное поле. Назначение этого бита не определено, он **должен** оставаться сброшенным (0x0) при передаче и **должен** игнорироваться на приёмной стороне.

Stream Identifier

Идентификатор потока (см. параграф 5.1.1), указанный 31-битовым целым числом без знака. Значение 0x0 зарезервировано для кадров, связанных с соединением в целом, а не с отдельным потоком.

Структура и содержимое информационных элементов кадра полностью определяется его типом.

[1] В оригинале строка ошибочно содержит закрывающую круглую скобку. См. https://www.rfc-editor.org/errata/eid5031. *Прим. перев.*

4.2. Размер кадра

Размер информационных полей в кадре ограничен анонсированным получателем значением SETTINGS_MAX_FRAME_SIZE, которое может находиться в диапазоне от 2^{14} (16 384) до 2^{24}-1 (16 777 215) октетов, включительно.

Все реализации **должны** обеспечивать возможность приёма и минимальной обработки кадров размером до 2^{14} октетов плюс 9-октетный заголовок кадра (параграф 4.1). При описании размера кадров фиксированный заголовок кадра не учитывается.

> Примечание. Для некоторых типов кадров, например, PING (параграф 6.7), имеются дополнительные ограничения на размер.

Конечная точка **должна** передать код ошибки FRAME_SIZE_ERROR, если размер полученного кадра превосходит SETTINGS_MAX_FRAME_SIZE, любой из пределов, заданных типом кадра, или слишком мал, чтобы вместить обязательные для кадра данные. Ошибка размера для кадра, который меняет состояние всего соединения, **должна** трактоваться, как ошибка соединения (параграф 5.4.1) - это включает все кадры, содержащие блок заголовка (параграф 4.3) (HEADERS, PUSH_PROMISE и CONTINUATION), SETTINGS, а также все кадры с идентификатором потока 0. Конечные точки не обязаны использовать все доступное пространство кадра и применение кадров с размером меньше разрешённого максимума позволяет ускорить реакцию на них. Большие кадры могут вызывать задержки передачи чувствительных к задержкам кадров (типа RST_STREAM, WINDOW_UPDATE или PRIORITY), блокировка которых может негативно влиять на производительность.

4.3. Сжатие и декомпрессия заголовка

Как и в HTTP/1, поле заголовка HTTP/представляет собой имя и одно или несколько связанных с ним значений. Поля заголовков используются в сообщениях с запросами и откликами HTTP, а также в операциях выталкивания (push) на серверах (см. параграф 8.2).

Список заголовка может включать множество полей или не включать ни одного. При передаче через соединение список преобразуется (serialize) в блок заголовка с использованием сжатия заголовков HTTP [COMPRESSION]. Последовательный блок заголовка затем может быть разделен на множество последовательностей октетов, называемых фрагментами блока заголовка, которые передаются в информационных блоках (payload) кадров HEADERS (параграф 6.2), PUSH_PROMISE (параграф 6.6) или CONTINUATION (параграф 6.10).

Для поля заголовка Cookie [COOKIE] используется особая трактовка с помощью отображения HTTP (см. параграф 8.1.2.5).

Принимающая сторона восстанавливает блок заголовка из фрагментов и выполняет декомпрессию для восстановления списка заголовка.

Полный блок заголовка может использоваться в одном из двух вариантов:

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

Сжатие заголовка выполняется с учётом состояния. Для соединения в целом используется один контекст сжатия и один контекст декомпрессии. Ошибки при декодировании блока заголовка **должны** трактоваться, как ошибки соединения (параграф 5.4.1) типа COMPRESSION_ERROR.

Каждый блок заголовка обрабатывается, как дискретный элемент. Блоки заголовков **должны** передаваться в виде непрерывной последовательности кадров без чередования с кадрами других типов или из других потоков. Последний кадр в последовательности HEADERS или CONTINUATION имеет установленный флаг END_HEADERS. Последний кадр в последовательности PUSH_PROMISE или CONTINUATION имеет установленный флаг END_HEADERS. Это обеспечивает логическую эквивалентность блока заголовка одному кадру.

Фрагменты блока заголовка могут передаваться только в кадрах HEADERS, PUSH_PROMISE или CONTINUATION, поскольку эти кадры переносят данные, которые могут изменять контекст сжатия на приёмной стороне. Конечной точке, принимающей кадры HEADERS, PUSH_PROMISE или CONTINUATION требуется собрать блоки заголовков и выполнить декомпрессию даже если кадры будут отброшены. Получатель **должен** разорвать соединение с возвратом ошибки (параграф 5.4.1) типа COMPRESSION_ERROR, если он не смог выполнить декомпрессию блока заголовка.

5. Потоки и мультиплексирование

«Поток» представляет собой независимую, двунаправленную последовательность кадров, передаваемых между клиентом и сервером через соединение HTTP/2. Потоки имеют несколько важных характеристик:

- в одном соединении HTTP/2 может быть открыто множество одновременных потоков и любая из конечных точек может чередовать кадры разных потоков;
- потоки могут устанавливаться и применяться в одностороннем порядке или совместно использоваться клиентом и сервером;
- любая из конечных точек может закрыть поток;
- порядок передачи кадров в потоке имеет значение и получатель обрабатывает кадры в порядке их приёма; в частности, кадры HEADERS и DATA значимы в семантическом смысле;
- потоки обозначаются целочисленными идентификаторами, которые выделяются создавшей поток конечной точкой.

5.1. Состояния потока

Жизненный цикл потока показан на рисунке 2.

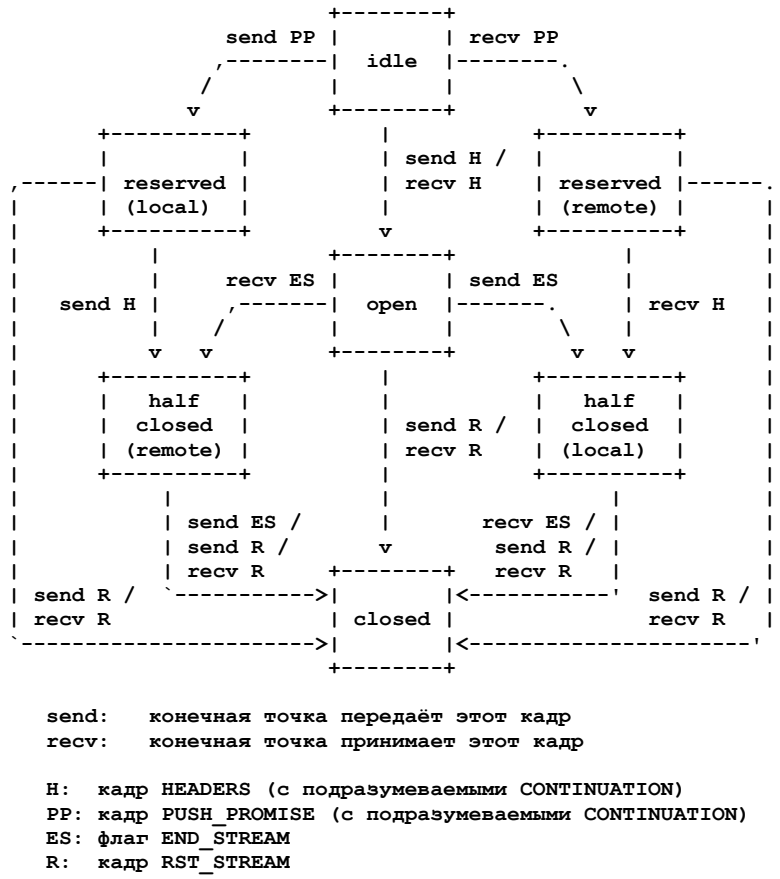

Рисунок 2. Состояния потока

На рисунке показаны переходы состояний потока, а также кадры и флаги, воздействующие только на эти переходы. В этом смысле кадры CONTINUATION не оказывают влияния на смену состояний - эффективно они являются частью кадра HEADERS или PUSH_PROMISE, за которым данный кадр следует.

В плане смены состояний потоков флаг END_STREAM обрабатывается как отдельное от переносящего этот флаг кадра событие - кадр HEADERS с установленным флагом END_STREAM может вызвать две смены состояния.

Каждая из конечных точек имеет своё представление о состоянии потока и эти представления могут различаться, когда часть кадров ещё находится в пути. Конечные точки не координируют создание потоков и каждая из точек создаёт потоки по своему усмотрению. Негативные последствия рассогласования состояний ограничиваются возможностью получения кадров в течение некоторого времени после «закрытия» потока с помощью RST_STREAM.

Состояния потоков описаны ниже.

idle - бездействие

Все потоки начинаются в состоянии idle.

Ниже перечислены переходы, которые приемлемы из этого состояния.

- Отправка или получение кадра HEADERS вызывает переход потока в состояние open. Идентификатор потока выбирается в соответствии с параграфом 5.1.1. Этот же кадр HEADERS может вызвать незамедлительный переход потока в состояние half-closed.
- Отправка кадра PUSH_PROMISE в другой поток резервирует бездействующий поток для использования в будущем. Состояние потока меняется на reserved (local).
- Получение кадра PUSH_PROMISE из другого потока резервирует бездействующий поток для использования в будущем. Состояние потока меняется на reserved (remote).
- Отметим, что кадр PUSH_PROMISE не передаётся через бездействующий поток, но указывает вновь резервируемый поток в поле Promised Stream ID.

Приём кадра, отличного от HEADERS или PRIORITY, через поток в состоянии **должна** трактоваться, как ошибка соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

reserved (local) - локально зарезервирован

Поток в состоянии reserved (local) - этот поток, который был зарезервирован отправкой кадра PUSH_PROMISE. Кадр PUSH_PROMISE резервирует бездействующий (idle) поток, путём связывания с открытым потоком, который был инициирован удаленным партнёром (см. параграф 8.2).

Из этого состояния возможны только указанные ниже переходы.

- Конечная точка передаёт кадр HEADERS который переводит поток в состояние half-closed (remote).
- Любая из конечных точек может передать кадр RST_STREAM, переводящий поток в состояние closed с отменой имеющегося резервирования.

В этом состоянии конечной точке **недопустимо** передавать какие-либо кадры, за исключением HEADERS, RST_STREAM или PRIORITY.

В этом состоянии **может** быть принят кадр PRIORITY или WINDOW_UPDATE. Получение кадра любого типа, кроме RST_STREAM, PRIORITY или WINDOW_UPDATE, в этом состоянии **должно** трактоваться, как ошибка соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

reserved (remote) - зарезервирован удалённой стороной

Поток в состоянии reserved (remote) - этот поток, который был зарезервирован удаленным партнёром.

Из этого состояния возможны только указанные ниже переходы.

- Получение кадра HEADERS вызывает переход потока в состояние half-closed (local).
- Любая из оконечных точек может передать кадр RST_STREAM, который переводит поток в состояние closed, отменяя резервирование.

Оконечная точка в этом состоянии **может** передать кадр PRIORITY для повторной приоритизации резервного потока. В этом состоянии для конечных точек **недопустимо** передавать какие-либо кадры, за исключением RST_STREAM, WINDOW_UPDATE или PRIORITY.

Получение кадра любого типа, кроме HEADERS, RST_STREAM или PRIORITY, в этом состоянии **должно** трактоваться, как ошибка соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

open - открыт

Поток в открытом состоянии может использоваться обоими партнёрами для передачи кадров любого типа. В этом состоянии передающие партнёры наблюдают за анонсируемыми другой стороной ограничениями, относящимися к управлению потоками данных на уровне потока в целом (параграф 5.2).

Из этого состояния любая из конечных точек может передать кадр с установленным флагом END_STREAM, который вызывает переход потока в одно из полузакрытых (half-closed) состояний. Конечная точка, передавшая флаг END_STREAM вызывает переход потока в состояние half-closed (local), а принявшая флаг END_STREAM - в состояние half-closed (remote).

Любая из конечных точек может передать из этого состояния кадр RST_STREAM, вызывающий незамедлительный переход потока в состояние closed.

half-closed (local) - полузакрыт локальной стороной

Поток в состоянии half-closed (local) не может быть использован для передачи каких-либо кадров за исключением WINDOW_UPDATE, PRIORITY и RST_STREAM.

Поток переходит из состояния half-closed (local) в закрытое (closed) при получении кадра с установленным флагом END_STREAM или при отправке любым из партнёров кадра RST_STREAM.

Конечная точка может получать через находящийся в этом состоянии поток кадры любого типа. Требуется обеспечение «кредита» управления потоком данных с помощью кадров WINDOW_UPDATE для продолжения приёма кадров с управлением потоком данных. В этом состоянии получатель может игнорировать кадры WINDOW_UPDATE, которые могут прийти за короткое время после отправки кадра с установленным флагом END_STREAM.

Кадры PRIORITY, полученные в этом состоянии, служат для повторной приоритизации потоков, зависящих от указанного потока.

half-closed (remote) - полузакрыт удалённой стороной

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

Если конечная точка получает какие-либо кадры, отличные от WINDOW_UPDATE, PRIORITY и RST_STREAM, для потока в этом состоянии, она **должна** возвращать ошибку потока (параграф 5.4.2) типа STREAM_CLOSED.

Поток в состоянии half-closed (remote) может использоваться конечной точкой для передачи кадров любого типа и конечная точка будет продолжать наблюдение ограничений, связанных у управлением потоком данных на уровне потока в целом (параграф 5.2).

Поток может перейти из состояния half-closed (remote) в закрытое состояние путём передачи кадра с флагом END_STREAM или передачи любой из сторон кадра RST_STREAM.

closed - закрыт

Состояние closed является завершающим для потока.

Конечной точке **недопустимо** передавать в закрытый поток какие-либо кадры, за исключением PRIORITY. Конечная точка, получившая отличный от PRIORITY после приёма RST_STREAM, **должна** трактовать его, как ошибку потока (параграф 5.4.2) типа STREAM_CLOSED. Аналогично, конечная точка, получившая любой кадр после приёма кадра с флагом END_STREAM, **должна** трактовать это, как ошибку соединения (параграф 5.4.1) типа STREAM_CLOSED, если кадр не разрешён в соответствии с приведённым ниже описанием.

Кадры WINDOW_UPDATE или RST_STREAM могут приниматься в этом состоянии в течение короткого периода после отправки кадра DATA или HEADERS с флагом END_STREAM. Пока удалённый партнёр не получит и не обработает RST_STREAM или кадр с флагом END_STREAM, передача этих типов кадров возможна. Конечные точки **должны** игнорировать кадры WINDOW_UPDATE и RST_STREAM, принятые в этом состоянии, хотя они **могут** выбрать трактовку кадров, принятых по истечении достаточно продолжительного времени после отправки END_STREAM, как ошибок соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадры PRIORITY могут передаваться через закрытые потоки для приоритизации потоков, которые зависят от закрытого. Конечным точкам **следует** обрабатывать кадры PRIORITY, хотя они могут игнорировать их, если поток был удалён из дерева зависимостей (см. параграф 5.3.4).

Если это состояние достигнуто в результате передачи кадра RST_STREAM, может оказаться, что получивший такой кадр партнёр уже передал или поместил в очередь на передачу кадры для закрываемого потока. Конечная точка **должна** игнорировать кадры, которые она получает через закрываемые потоки после отправки кадра RST_STREAM. Конечная точка **может** ограничить период, в течение которого она будет такие кадры, а потом начнёт трактовать их, как ошибки.

Кадры с управлением потоком данных (т. е., DATA), принятые после отправки RST_STREAM, учитываются в окне управления потоком данных соединения. Эти кадры могут игнорироваться, но, поскольку они были переданы до получения их отправителем кадра RST_STREAM, отправитель будет учитывать их в окне управления потоком данных.

Конечная точка может получить кадр PUSH_PROMISE после отправки ею RST_STREAM. Кадр PUSH_PROMISE вызывает переход потока в состояние reserved, даже если связанный с ним поток сброшен. Следовательно, для закрытия нежелательного потока требуется RST_STREAM.

Если в этом документе не содержится более конкретных указаний, реализациям **следует** трактовать получение кадра, не разрешённого явно в описании состояния, как ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Отметим, что кадры PRIORITY могут передаваться и приниматься в любом состоянии потока. Кадры неизвестных типов игнорируются.

Пример смены состояния для обмена запрос-отклик HTTP приведены в параграфе 8.1, примеры смены состояний для выталкивания на сервере (server push) - в параграфах 8.2.1 и 8.2.2.

5.1.1. Идентификаторы потоков

Для идентификации потоков используются целые числа без знака размером 31 бит. Потоки, инициируемые клиентами, **должны** иметь нечётные идентификаторы, а потоки, инициируемые серверами, - чётные. Идентификатор с нулевым значением (0x0) служит для сообщений управления соединением и не может применяться для организации нового потока.

На запросы HTTP/1.1, обновлённые до HTTP/2 (см. параграф 3.2), отвечают с идентификатором потока 1 (0x1). По завершении процесса обновления поток 0x1 переходит в состояние half-closed (local) для клиента. Следовательно, идентификатор 0x1 не может быть выбран для нового потока клиентом, который обновляется с HTTP/1.1.

Идентификатор вновь созданного потока **должен** быть численно больше идентификаторов всех потоков, которые инициирующая точка открыла или зарезервировала. Это относится к потокам, открываемым с помощью кадров HEADERS и резервируемым с помощью PUSH_PROMISE. Конечная точка, получившая неожиданный идентификатор потока, **должна** ответить на него сообщением об ошибке соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Первое использование идентификатора нового потока неявно закрывает все бездействующие (состояние idle) потоки, которые были инициированы этим партнёром и имеют меньшие значения идентификаторов. Например, если клиент передаёт кадр HEADERS в поток 7 без передачи кадров в поток 5, то поток 5 переводится в состояние closed при отправке или получении первого кадра для потока 7.

Идентификаторы потока не могут использоваться повторно. Продолжительное использование соединения может приводить к исчерпанию у конечной точки доступных для организации нового потока идентификаторов. Клиент, который не имеет возможности создать новый идентификатор потока, может организовать для такого потока новое соединение, а сервер в такой ситуации может передать кадр GOAWAY, заставляющий клиента организовать новое соединение для новых потоков.

5.1.2. Одновременные потоки

Партнёр может ограничить число одновременно активных потоков, используя параметр SETTINGS_MAX_CONCURRENT_STREAMS (см. параграф 6.5.2) в кадре SETTINGS. Максимальное число одновременных потоков задаётся для каждой конечной точки и применяется только к партнёру, получившему эти настройки. Т. е., клиент определяет максимальное число одновременных потоков, которые может инициировать сервер, а серверы задают максимальное число одновременных потоков, которые может инициировать клиент.

Потоки в состоянии open или любом из полузакрытых (half-closed) состояний учитываются в числе разрешённых одновременных потоков. Потоки в любом из этих трёх состояний учитываются при контроле предела, заданного параметром SETTINGS_MAX_CONCURRENT_STREAMS. Потоки в любом из резервных (reserved) состояний не учитываются в числе одновременных.

Конечной точке **недопустимо** превышать заданный её партнёром предел числа потоков. Конечная точка, получившая кадр HEADERS, который вызывает превышение анонсированного предела, **должна** трактовать это как ошибку потока (параграф 5.4.2) типа PROTOCOL_ERROR или REFUSED_STREAM. Выбор типа определяется желанием конечной точки разрешать автоматические повторы (см. параграф 8.1.4).

Конечная точка, желающая снизить SETTINGS_MAX_CONCURRENT_STREAMS до значения меньше текущего числа открытых потоков может закрыть потоки, выходящие за пределы нового значения, или сохранять их до завершения.

5.2. Управление потоком

Использование потоков для мультиплексирования порождает конфликты с использованием соединений TCP, приводящие к блокировке потоков. Схема управления потоком данных обеспечивает отсутствие деструктивного воздействия одного потока на другой в рамках одного соединения. Управление потоком данных применяется как для отдельных потоков, так и для соединения в целом.

HTTP/2 обеспечивает управление потоком данных на основе кадров WINDOW_UPDATE (параграф 6.9).

5.2.1. Принципы управления потоком данных

Управление потоком данных в потоках HTTP/2 имеет целью обеспечить возможность применения множества разных алгоритмов управления потоком данных без изменения протокола. Характеристики управления потоком данных в HTTP/2 перечислены ниже.

1. Управление потоком данных относится к конкретному соединению. Оба типа управления потоком реализуются между конечными точками одного интервала пересылки (hop), а не сквозного пути.

2. Управление потоком данных основано на кадрах WINDOW_UPDATE. Получатели анонсируют число октетов, которые они готовы принимать в потоке и соединении в целом. Эта схема является кредитной (credit-based).

3. Управление потоком данных сопряжено с общим управлением, обеспечиваемым получателем. Получатель **может** выбрать любой размер окна для каждого потока и соединения в целом. Отправитель **должен** соблюдать приделы управления потоком данных, заданные получателем. Клиенты, серверы и промежуточные узлы независимо анонсируют своё окно управления потоком данных как получатели и принимают ограничения партнёра, выступая в качестве отправителей.

4. Начальное значение размера окна управления потоком данных составляет 65 535 октетов как для нового потока, так и для соединения в целом.

5. Тип кадра определяет применение к нему управления потоком данных. Из описанных в документе типов кадров управление потоком применяется только к кадрам DATA, а остальные типы кадров не учитываются в

окне управления потоком данных. Это предотвращает блокировку важных кадров управления системой контроля потока данных.

6. Управление потоком данных не может быть отключено.

7. HTTP/2 определяет лишь формат и семантику кадров WINDOW_UPDATE (параграф 6.9). Этот документ не задаёт условий, при которых получатель передаёт такие кадры или отправляемые в них значения, и не указывает, как отправитель выбирает пакеты для передачи. Реализации могут выбирать любые алгоритмы, соответствующие их потребностям.

Реализации также отвечают за управление передачей запросов и откликов на основе их приоритета, а также предотвращение блокировки head-of-line для запросов и управление созданием новых потоков. Алгоритмы, выбранные для решения этих задач, могут взаимодействовать с алгоритмом управления потоком данных.

5.2.2. Допустимое использование управления потоком данных

Управление потоком данных определено для защиты конечных точек, работающих в условиях ограниченных ресурсов. Например, посреднику (proxy) требуется разделять память между множеством соединений, которые могут включать медленные нисходящие соединения в комбинации с быстрыми восходящими. Управление потоком данных предназначено для случаев, когда получатель не способен обработать данные в одном потоке, но хочет продолжать обработку данных в другом потоке того же соединения.

Системы, где такая возможность не требуется, могут анонсировать окно управления потоком данных максимального размера (2^{31}-1) и поддерживать это окно, предавая кадр WINDOW_UPDATE при получении любых данных. Это эффективно отключает управление потоком данных для этого получателя. А отправитель всегда использует окно управления потоком данные, анонсированное получателем.

Системы с ограниченными ресурсами (например, памятью) могут реализовать управление потоком данных для ограничения объёма памяти, которым может воспользоваться партнёр. Однако следует отметить, что это может приводить к неоптимальному использованию доступных ресурсов сети, если управление потоком данных включено без информации о произведении «пропускная способность - задержка» (см. [RFC7323]).

Даже при наличии полной информации о произведении «пропускная способность - задержка» реализация управления потоком данных может сталкиваться с трудностями. При использовании управления потоком данных получатель **должен** своевременно считывать содержимое приёмных буферов TCP. Невыполнение этого требования может порождать ситуации, когда критически важные кадры типа WINDOW_UPDATE не будут прочитаны и не подействуют.

5.3. Приоритет потока

Клиент может задать приоритет для нового потока, включив соответствующую информацию в кадр HEADERS (параграф 6.2), открывающий поток. В любой другой момент для смены приоритета потока может использоваться кадр PRIORITY (параграф 6.3).

Целью приоритизации является обеспечение конечным точкам возможности указать партнёру свои предпочтения для выделения им ресурсов при управлении одновременными потоками. Наиболее важна возможность использования приоритета для выбора потоков при передаче кадров в случаях наличия ограничений.

Потоки можно приоритизировать путём их маркировки в зависимости от завершения других потоков (параграф 5.3.1). Для каждой зависимости выделяется относительный «вес» - число, которое используется для определения относительной доли доступных ресурсов, выделяемых потокам, которые зависят от одного и того же потока.

Явно заданные приоритеты для потоков служит входными данными для процесса приоритизации. Это не гарантирует какого-либо определённого порядка обработки или передачи для данного потока по сравнению с другими потоками. Конечная точка не может вынудить партнёра обрабатывать одновременные потоки в конкретном порядке на основе приоритета. Следовательно, указание приоритета служит лишь рекомендацией.

Данные о приоритизации могут быть опущены и в этом случае будут применяться принятые по умолчанию настройки приоритетов (параграф 5.3.5).

5.3.1. Зависимости потока

Для каждого потока может быть задана явная зависимость от другого потока. Включение зависимости выражает предпочтение выделять указанному потоку ресурсы раньше, нежели зависимому от него потоку.

Поток, который не зависит от какого-либо другого потока, имеет значение зависимости 0x0. Иными словами, несуществующий поток 0 образует корень дерева (зависимостей).

Поток, зависящий от другого потока, является зависимым. Поток, от которого зависит другой поток, называется родительским. Зависимость потока, не включённого в текущее дерево (такой поток находится в состоянии простоя - idle), приводит к тому, что данный поток получает используемый по умолчанию приоритет (параграф 5.3.5).

При определении зависимости от другого потока поток добавляется в качестве новой зависимости родительского потока. Зависимые потоки с общим родителем не упорядочиваются один относительно другого. Например, если потоки B и C зависят от потока A и создаётся поток D, зависящий от потока A, это приведёт к зависимости от A потоков B, C и D в любом порядке.

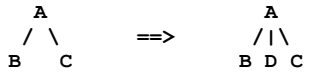

Рисунок 3. Пример создания зависимости «по умолчанию».

Флаг эксклюзивности позволяет добавить новый уровень зависимости. Этот флаг делает поток зависящим лишь от его родителя, делая остальные потоки (зависимости) зависящими от данного исключительного потока. В предыдущем примере создание потока D с исключительной зависимостью от потока A делает поток D родительским для B и C.

Рисунок 4. Пример создания исключительной зависимости.

Внутри дерева зависимостей зависимому потоку **следует** выделять ресурсы только в тех случаях, когда все потоки, от которых он зависит (цепочка родителей вплоть до 0x0), закрыты или в них ничего не происходит.

Поток не может зависеть от самого себя. Конечная точка **должна** трактовать это как ошибку потока (параграф 5.4.2) типа PROTOCOL_ERROR.

5.3.2. «Взвешивание» зависимостей

Все зависимости потоков выражаются «весом» - целыми числами от 1 до 256 (включительно).

Потокам, имеющим общего родителя **следует** распределять ресурсы пропорционально весу. Таким образом, если поток B зависит от потока A и имеет вес 4, поток C тоже зависит от A, но имеет вес 12 и в потоке A ничего не происходит, в идеальном случае B получит треть ресурсов, выделенных для потока C.

5.3.3. Повторная приоритизация

Приоритет для потоков изменяется с помощью кадров PRIORITY. Установка зависимости вынуждает поток стать зависимым от указанного родительского потока.

Зависимые потоки перемещаются вместе с их родителем, если приоритет того изменяется. Установка зависимости с флагом исключительности для приоритизируемого заново потока вынуждает все зависимости нового родительского потока стать зависимыми от повторно приоритизированного потока.

Если поток становится зависимым от одной из своих недавних зависимостей, эта зависимость (поток) перемещается так, чтобы стать зависимой от прежнего родителя потока для которого меняется приоритизация. Вес перемещённого зависимого потока сохраняется.

В качестве примера рассмотрим дерево зависимостей, где B и C зависят от A, D и E зависят от C, а F - от D. Если поток A становится зависимым от D, то D занимает место A. Все остальные отношения зависимости сохраняются за исключением потока F, который становится зависимым от A, если реприоритизация является исключительной.

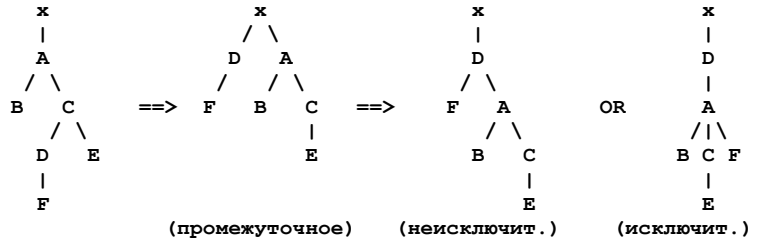

Рисунок 5. Пример изменения порядка зависимостей.

5.3.4. Управление состоянием приоритизации

При удалении потока из дерева зависимостей его зависимости могут перемещаться, чтобы стать зависимыми от родителя закрытого потока. Веса новых зависимостей пересчитываются с распределением веса зависимости закрытого потока пропорционально весам его зависимостей.

Потоки, которые удаляются из дерева зависимостей, вызывают некоторую потерю приоритизационных данных. Ресурсы совместно используются потоками с общим родителем и это означает, что при закрытии или блокировании потока из этой группы, выделенные для него ресурсы распределяются между его непосредственными соседями. Однако при удалении общей зависимости из дерева зависимостей эти потоки используют ресурсы вместе с потоками следующего вышележащего уровня.

В качестве примера предположим, что потоки A и B имеют общего родителя, а потоки C и D зависят от A. До удаления потока A при невозможности продолжения A и D поток C получит все ресурсы, выделенные для A. Если поток A удалён из дерева, вес этого потока будет поделён между C и D. Если продолжение D становится невозможным, поток C получит только часть ресурсов. При одинаковых начальных весах это составит 1/3, а не половину доступных ресурсов.

Возможны ситуации, когда поток закрывается, а приоритизационная информация, которая создаёт зависимости от этого потока ещё находится «в пути». Если поток, указанный в зависимости, не имеет связанной с ним информации о приоритете, зависимый поток получает установленный по умолчанию приоритет (параграф 5.3.5). Это может создавать неоптимальную приоритизацию, поскольку потоку может быть присвоен приоритет, отличный от нужного.

Для предотвращения этой проблемы конечной точке **следует** удерживать состояние приоритизации потока на некоторое время после закрытия потоков. Чем дольше сохраняется состояние, тем меньше шансов выделения потоку некорректного или принятого по умолчанию приоритета.

Аналогично, бездействующему потоку может быть присвоен приоритет или такой поток может стать родителем других потоков. Это позволяет организовать группирующий узел в дереве зависимостей, что позволяет более гибко выражать приоритеты. Бездействующие потоки начинаются с принятым по умолчанию приоритетом (параграф 5.3.5).

Удержание информации о приоритете для потоков, которые не учитываются в пределе, устанавливаемом SETTINGS_MAX_CONCURRENT_STREAMS, могут сильно загружать конечную точку. Следовательно, объем сохраняемой информации о приоритетах **может** ограничиваться.

Объем дополнительной информации о состоянии, которую поддерживает конечная точка для приоритизации, может зависеть от нагрузки. При высокой загрузке информация о состоянии может отбрасываться в целях экономии ресурсов. В экстремальных случаях конечная точка может отбрасывать даже данные приоритизации для активных или резервных

потоков. Если задан предел, конечной точке **следует** поддерживать состояние по меньшей мере для числа потоков, разрешённого значением SETTINGS_MAX_CONCURRENT_STREAMS. Реализациям **следует** также пытаться удерживать состояния для потоков, которые активно используются в дереве приоритетов.

Если конечная точка сохраняет достаточно данных о состоянии и получает кадр PRIORITY, меняющий приоритет закрытого потока, ей **следует** изменить зависимости потоков, зависящих от него.

5.3.5. Приоритеты по умолчанию
Все потокам изначально задаётся неисключительная зависимость от потока 0x0. Выталкиваемые потоки (параграф 8.2) изначально зависят от связанных с ними потоков. В обоих случаях потокам присваивается принятый по умолчанию вес 16.

5.4. Обработка ошибок
Кадрирование HTTP/2 допускает два класса ошибок:

- ошибки, не позволяющие использовать соединение совсем, называются ошибками соединения;
- ошибки в отдельных потоках называются ошибками потока.

Список кодов ошибок представлен в разделе 7.

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, полученного без ошибок от партнёра. Кадр GOAWAY включает код ошибки, указывающий причину разрыва соединения. После отправки кадра GOAWAY конечная точка **должна** закрыть соединение TCP.

Возможны ситуации, когда нет гарантии получения кадра GOAWAY принимающей стороной (в параграфе 6.6 [RFC7230] описано, как незамедлительный разрыв может вызывать потерю данных). В случае ошибки соединения GOAWAY обеспечивает лишь попытку сообщить партнёру о причине разрыва соединения.

Конечная точка может завершить соединение в любой момент. В частности, конечная точка **может** трактовать ошибку потока, как ошибку соединения. Конечным точкам **следует** передавать кадр GOAWAY при завершении соединения, указывая в нем обстоятельства завершения.

5.4.2. Обработка ошибок потока
Ошибки потока относятся к конкретному потоку и не затрагивают работу других потоков.

Конечная точка, обнаружившая ошибку потока, передаёт кадр RST_STREAM (параграф 6.4) с идентификатором потока, в котором возникла ошибка. Кадр RST_STREAM включает код, обозначающий тип ошибки.

RST_STREAM служит последним кадром, который конечная точка передаёт в поток. Передавший RST_STREAM партнёр **должен** быть готов к приёму любых кадров, которые уже были отправлены или помещены в очередь удаленным партнёром. Эти кадры можно игнорировать за исключением тех, которые меняют состояние соединения (например, кадры, управляющие сжатием заголовков (параграф 4.3) или управлением потоками данных).

Обычно конечным точкам **не следует** отправлять более одного кадра RST_STREAM для любого потока. Однако конечная точка **может** передавать дополнительные кадры RST_STREAM, если она продолжает получать кадры через закрытый поток по истечении периода кругового обхода. Такое поведение разрешено для обеспечения совместимости с ошибочными реализациями.

Для предотвращения петель конечным точкам **недопустимо** передавать кадры RST_STREAM в ответ на получение RST_STREAM.

5.4.3. Разрыв соединения
Если соединение TCP закрывается или сбрасывается, а потоки остаются в открытом (open) или полузакрытом (half-closed) состоянии, затронутые этим потоки невозможно восстановить автоматически (см. параграф 8.1.4).

5.5. Расширения HTTP/2
HTTP/2 допускает расширения протокола. В рамках описанных в этом параграфе ограничений протокольные расширения могут применяться для предоставления дополнительных услуг или изменения некоторых аспектов протокола. Расширения действуют только в рамках одного соединения HTTP/2.

Это применимо к элементам протокола, определённых в данном документе, и не оказывает влияния на существующие опции для расширения HTTP типа определения новых методов, кодов состояний или полей заголовков.

В расширениях разрешается использовать новые кадры (параграф 4.1), новые параметры (параграф 6.5.2) или новые коды ошибок (раздел 7). Для поддержки этих расширений организованы новые реестры: тип кадров (параграф 11.2), параметры (параграф 11.3), коды ошибок(параграф 11.4).

Реализации **должны** игнорировать неизвестные или неподдерживаемые значения во всех расширяемых элементах протокола. Кадры неизвестных или неподдерживаемых типов реализации **должны** отбрасывать. Это означает, что любую из точек расширения можно применять без предварительного согласования. Однако кадры расширения не позволяется помещать в середину блока заголовка (параграф 4.3), это **должно** трактоваться, как ошибка соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Расширения, которые могут менять семантику существующих компонент протокола, **должны** согласовываться до начала использования. Например, расширение, которое меняет схему кадра HEADERS нельзя применять, пока партнёр не сообщил о возможности их восприятия. В таких случаях может потребоваться также согласования начала использования изменённой схемы. Отметим, что предположение об использовании управления потоком для каких-либо кадров, отличных от DATA, является таким изменением семантики и может применяться лишь после согласования.

Этот документ не задаёт конкретного метода согласования использования расширений, но отмечает, что для этого могут служить настройки (параграф 6.5.2). Если оба партнёра устанавливают значение, которое показывает готовность использовать расширение, это расширение может применяться. При использовании настроек для согласования расширения начальное значение **должно** определяться с запретом расширения по умолчанию.

6. Определения кадров

Данная спецификация определяет множество типов кадров, каждый из которых идентифицируется уникальным 8-битовым кодом типа. Каждый тип кадров служит своим целям в процессе организации и поддержки соединения в целом и его отдельных потоков.

Передача определённых типов кадров может менять состояние соединения. Если конечная точка не способна поддерживать синхронизированное представление состояния соединения, связь через такое соединение становится невозможной. Следовательно, для конечных точек важно иметь общее представление о том как может измениться состояние соединения в результате использования данного кадра.

6.1. DATA

Кадры DATA (type=0x0) передают произвольные последовательности октетов переменного размера, связанные с потоком. Например, один или множество кадров DATA может быть использован для передачи данных запроса или отклика HTTP.

Кадры DATA **могут** также включать заполнение, которое может использоваться для сокрытия размеров сообщений. Заполнение относится к функциям защиты (см. параграф 10.7).

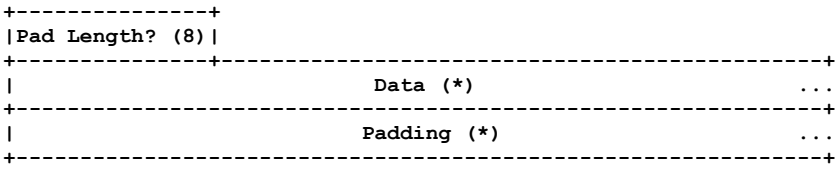

Рисунок 6. Информация кадра DATA

Поля кадров DATA перечислены ниже.

Pad Length
: 8-битовое поле, содержащее размер заполнения кадра в октетах. Это поле является условным (указано символом ? на рисунке) и присутствует только при установленном флаге PADDED.

Data
: Данные приложения, размер которых определяется вычитанием размера заполнения и других полей из общего размера кадра.

Padding
: Октеты заполнения не содержат осмысленных данных приложения. При передаче октеты заполнения **должны** заполняться нулями. Получатель не обязан обрабатывать заполнение, но **может** трактовать отличное от нулей заполнение, как ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Ниже перечислены флаги, определённые для кадров DATA.

END_STREAM (0x1)
: Установленный бит 0 показывает, что этот кадр является последним, который данная конечная точка передала для указанного потока. Установка этого флага вызывает переход потока в полузакрытое (half-closed) или закрытое (closed) состояние (параграф 5.1).

PADDED (0x8)
: Установленный бит 3 указывает на присутствие поля Pad Length и октетов заполнения, указанных им.

Кадр DATA **должны** быть связаны с потоком. В ответ на кадр DATA с идентификатором потока 0x0 получатель **должен** передать сигнал об ошибке соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Для кадров DATA применяется управление потоком данных и они могут передавать только в потоках с состоянием open или half-closed (remote). При управлении потоком данных учитывается вся информация кадра DATA, включая поля Pad Length и Padding при их наличии. Если кадр DATA принят в потоке, состояние которого является open или half-closed (local), получатель **должен** ответить индикацией ошибки потока (параграф 5.4.2) типа STREAM_CLOSED.

Общее число октетов заполнения определяется значением поля Pad Length. Если размер заполнения совпадает с размером данных в кадре или превышает его, получатель **должен** трактовать это как ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Примечание. Размер кадра может быть увеличен на один октет путём включения поля Pad Length = 0.

6.2. HEADERS

Кадры HEADERS (type=0x1) служат для создания потоков (параграф 5.1) и для передачи фрагментов блока заголовков. Кадры HEADERS могут передаваться в потоки с состоянием idle, reserved (local), open или half-closed (remote).

Элементы данных кадров HEADERS приведены ниже.

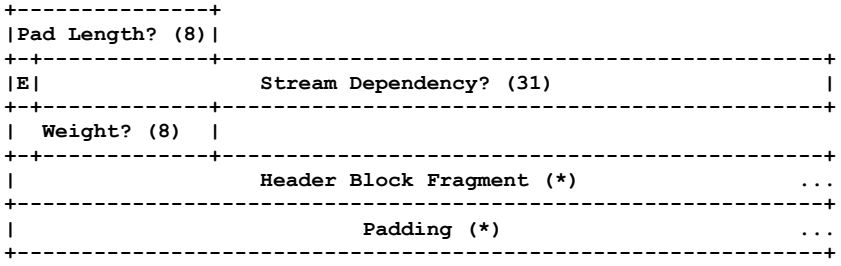

Рисунок 7. Информация кадра HEADERS

Pad Length
8-битовое поле, содержащее размер заполнения кадра в октетах. Это поле присутствует только при установленном флаге PADDED.

E
Однобитовый флаг, указывающий на исключительную зависимость потока (см. параграф 5.3). Это поле присутствует лишь при установленном флаге PRIORITY.

Stream Dependency
31-битовый идентификатор потока, от которого зависит данный поток (см. параграф 5.3). Это поле присутствует лишь при установленном флаге PRIORITY.

Weight
8-битовое целое число без знака, представляющее вес приоритета для потока (см. параграф 5.3). Добавление 1 к значению поля даёт вес от 1 до 256. Это поле присутствует лишь при установленном флаге PRIORITY.

Header Block Fragment
Фрагмент блока заголовка (параграф 4.3).

Padding
Октеты заполнения.

Ниже перечислены флаги, определённые для кадров HEADERS.

END_STREAM (0x1)
Установленный бит 0 показывает, что блок заголовка (параграф 4.3) является последним, который данная конечная точка передала для указанного потока.
Кадры HEADERS передают флаги END_STREAM, указывающие конец потока. Однако за кадром HEADERS с установленным флагом END_STREAM могут следовать кадры CONTINUATION того же потока. Логически кадры CONTINUATION являются частью кадра HEADERS.

END_HEADERS (0x4)
Установленный бит 2 показывает, что кадр содержит целый блок заголовка (параграф 4.3) и за ним не следует кадров CONTINUATION.
За кадром HEADERS со сброшенным флагом END_HEADERS **должен** следовать кадр CONTINUATION для того же потока. Получатель **должен** трактовать приём другого типа кадра или кадра в другом потоке как ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

PADDED (0x8)
Установленный бит 3 указывает на присутствие поля Pad Length и октетов заполнения, указанных им.

PRIORITY (0x20)
Установленный бит 5 говорит о присутствии полей Exclusive Flag (E), Stream Dependency и Weight (см. параграф 5.3).

Данные кадра HEADERS включают блок заголовка (параграф 4.3). Если этот блок не помещается в кадр HEADERS, он продолжается в кадре CONTINUATION (параграф 6.10).

HEADERS **должны** привязываться к потокам. В ответ на кадр HEADERS с идентификатором потока 0x0 получатель **должен** передать сигнал об ошибке соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадр HEADERS меняет состояние соединения в соответствии с параграфом 4.3.

Кадры HEADERS могут включать заполнение. Поля заполнения идентичны используемым в кадрах DATA (параграф 6.1). Заполнение, размер которого превышает размер оставшийся после размещения фрагмента блока заголовка, **должен** трактоваться как PROTOCOL_ERROR.

Информация о приоритете в кадрах HEADERS логически эквивалентна отдельному кадру PRIORITY, но её включение в HEADERS позволяет предотвратить ошибочную приоритизацию при создании нового потока. Поля приоритизации в кадрах HEADERS, следующих за первым в потоке, меняют приоритизацию потока (параграф 5.3.3).

6.3. PRIORITY

Кадр PRIORITY (type=0x2) задаёт предложенный отправителем приоритет для потока (параграф 5.3). Эти кадры могут передаваться при любом состоянии потоков, включая бездействующие (idle) и закрытые (closed).

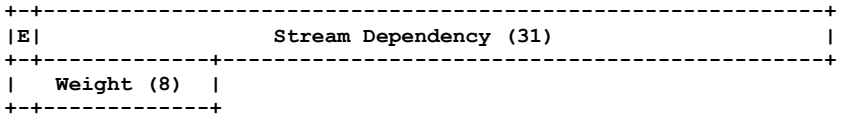

Рисунок 8. Информация кадра PRIORITY

Информационные поля кадра PRIORITY показаны на рисунке.

E
Однобитовый флаг, указывающий на исключительную зависимость потока (см. параграф 5.3).

Stream Dependency
31-битовый идентификатор потока, от которого зависит данный поток (см. параграф 5.3).

Weight

8-битовое целое число без знака, представляющее вес приоритета для потока (см. параграф 5.3). Добавление 1 к значению поля даёт вес от 1 до 256.

Для кадров PRIORITY не определено каких-либо флагов.

Кадр PRIORITY всегда идентифицирует поток. В ответ на кадр PRIORITY с идентификатором потока 0x0 получатель **должен** передать сигнал об ошибке соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадр PRIORITY может быть передан в любом состоянии потока, однако он не может передаваться между последовательными кадрами, составляющими один блок заголовка (параграф 4.3). отметим, что этот кадр может приходить после завершения обработки или передачи кадра и в результате не повлияет на указанный поток. Для потоков в состоянии half-closed (remote) или closed этот кадр может влиять лишь на обработку указанного потока и зависимых от него потоков, не оказывая влияния на передачу самого кадра в этом потоке.

Кадр PRIORITY может быть передан в потоки, находящиеся в состоянии idle или closed. Это позволяет изменить приоритизацию группы зависимых потоков, поменяв приоритет неиспользуемого или закрытого родительского потока.

Кадры PRIORITY с размером, отличным от 5 октетов, **должны** трактоваться как ошибка потока (параграф 5.4.2) типа FRAME_SIZE_ERROR.

6.4. RST_STREAM

Кадр RST_STREAM (type=0x3) позволяет незамедлительно прервать поток. RST_STREAM передаётся для запроса разрыва потока или индикации возникшей ошибки.

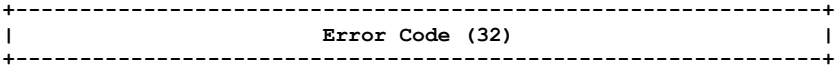

Рисунок 9. Информация кадра RST_STREAM

Кадр RST_STREAM содержит одно 32-битовое целое число без знака, указывающее код ошибки (параграф 7), который показывает причину прерывания потока.

Для кадров RST_STREAM не определено каких-либо флагов.

Кадр RST_STREAM полностью прерывает указанный поток и вызывает его переход в состояние closed. После приёма кадра RST_STREAM в потоке его получателю **недопустимо** передавать в этот поток дополнительные кадры за исключением PRIORITY. Однако после отправки RST_STREAM передавшая этот кадр конечная точка **должна** быть готова к получению и обработке дополнительных кадров, которые могли быть переданы в этот поток до прибытия кадра RST_STREAM.

Кадр RST_STREAM **должен** быть связан с потоком. В ответ на кадр RST_STREAM с идентификатором потока 0x0 получатель **должен** передать сигнал об ошибке соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадры RST_STREAM **недопустимо** передавать в поток, находящийся в состоянии idle. В ответ на кадр RST_STREAM в бездействующем потоке получатель **должен** передать сигнал об ошибке соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадры RST_STREAM с размером, отличным от 4 октетов, **должны** трактоваться как ошибка потока (параграф 5.4.2) типа FRAME_SIZE_ERROR.

6.5. SETTINGS

Кадры SETTINGS (type=0x4) служат для передачи конфигурационных параметров, воздействующих на коммуникации конечной точки, типа предпочтений и ограничений в поведении партнёра. Кадры SETTINGS служат также для подтверждения приёма таких параметров. Отдельные параметры из кадров SETTINGS могут называться также «установками» (setting).

Параметры в SETTINGS не согласуются, они описывают характеристики передающего кадр партнёра, которые используются принимающим партнёром. Каждый из партнёров может анонсировать разные значения для одного и того же параметра. Например, клиент может задать большое канальное окно управления потоком данных, тогда как сервер может выбрать меньшее значение в целях экономии ресурсов.

Кадры SETTINGS **должны** передаваться обеими конечными точками при организации соединения и **могут** также передаваться каждой из конечных точек в любой момент в течение срока существования соединения. Реализации **должны** поддерживать параметры, определённые этой спецификацией.

Каждый параметр из кадра SETTINGS устанавливается взамен имеющегося значения этого параметра. Параметры обрабатываются в порядке их следования в кадре и получателю кадра SETTINGS не требуется поддерживать какое-либо состояние в дополнение к текущим значениям своих параметров. Следовательно, значение параметра SETTINGS является последним значением, которое увидел получатель.

Параметры SETTINGS подтверждаются принявшим кадр партнёром. Для этого в кадрах SETTINGS используется описанный ниже флаг.

ACK (0x1)

Установленный бит 0 показывает, что кадр подтверждает приём и применение кадра SETTINGS от партнёра. При установленном флаге поля данных (payload) кадра SETTINGS **должны** быть пустыми. Приём кадра SETTINGS с флагом ACK и отличным от 0 полем размера должен трактоваться как ошибка соединения (параграф 5.4.1) типа FRAME_SIZE_ERROR. Дополнительная информация приведена в параграфе 6.5.3. Синхронизация настроек.

Кадры SETTINGS всегда относятся к соединению, а не к отдельным потокам. Поле идентификатора потока в кадрах SETTINGS **должно** иметь нулевое значение (0x0). При получении кадра SETTINGS с отличным от 0 идентификатором потока конечная точка **должна** ответить сигналом об ошибке соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадры SETTINGS влияют на состояние соединения. Некорректно сформированные или неполные кадры SETTINGS должны трактоваться как ошибка соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Кадры SETTINGS с размером, не кратным 6 октетам, должны считаться ошибкой соединения (параграф 5.4.1) типа FRAME_SIZE_ERROR.

6.5.1. Формат SETTINGS

Информационные поля кадра SETTINGS могут (но не обязаны) включать параметры, каждый из которых включает 16-битовый идентификатор и 32-битовое значение.

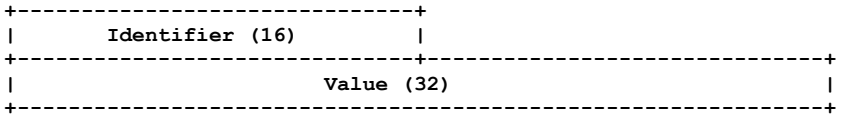

Рисунок 10. Информация кадра SETTINGS.

6.5.2. Параметры SETTINGS

Определённые настоящим документом параметры перечислены ниже.

SETTINGS_HEADER_TABLE_SIZE (0x1)

Позволяет отправителю сообщить удалённой стороне максимальный размер (в октетах) таблицы сжатия заголовков, применяемой для декодирования блоков заголовка. Кодировщик может выбрать размер, не превышающий это значение, используя сигнализацию для конкретного формата сжатия внутри блока заголовка. (см. [COMPRESSION]). Начальное значение составляет 4096 октетов.

SETTINGS_ENABLE_PUSH (0x2)

Эта установка может служить для запрета выталкивания данных сервером (параграф 8.2). Конечной точке **недопустимо** передавать кадр PUSH_PROMISE, если она получила этот параметр со значением 0. Конечная точка, установившая для этого параметра значение 0 и имеющая подтверждение этого, **должна** трактовать получение кадра PUSH_PROMISE, как ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

В качестве начального значения параметра используется 1, что показывает возможность использования на сервере выталкивания данных. Все значения этого параметра, кроме 0 и 1, **должны** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

SETTINGS_MAX_CONCURRENT_STREAMS (0x3)

Показывает максимальное число потоков, которые будет разрешать отправитель. Это ограничение является направленным и указывает число потоков, которые отправитель позволяет создавать получателю. Изначально для этого значения не задано ограничений. Рекомендуется устанавливать для этого параметра значение не менее 100 чтобы не возникало неоправданного ограничения параллельной работы.

Нулевое значение SETTINGS_MAX_CONCURRENT_STREAMS **не следует** считать особым, оно просто предотвращает создание новых потоков. Однако такая же ситуация может возникать при любом значении в случае достижения заданного предела. Серверам **следует** устанавливать нулевое значение лишь на короткий период - если сервер не хочет принимать запросы, лучше закрыть соединение.

SETTINGS_INITIAL_WINDOW_SIZE (0x4)

Показывает размер начального окна отправителя (в октетах) для управления потоком данных на уровне потока. Начальное значение составляет 2^{16}-1 (65 535) октетов.

Этот параметр влияет на размер окна для всех потоков (см. параграф 6.9.2).

Значения, превышающие максимальный размер окна управления потоком данных 2^{31}-1, **должны** считаться ошибкой соединения (параграф 5.4.1) типа FLOW_CONTROL_ERROR.

SETTINGS_MAX_FRAME_SIZE (0x5)

Показывает максимальный размер данных (payload) в кадре, которые отправитель готов получать (в октетах).

Начальное значение составляет 2^{14} (16 384) октетов. Анонсируемые конечной точкой значения **должны** лежать в диапазоне от этого начального значения до максимального разрешённого размера кадра (2^{24}-1 или 16 777 215 октетов), включительно. Выходящие за предел этого диапазона значения **должны** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

SETTINGS_MAX_HEADER_LIST_SIZE (0x6)

Это предложение информирует партнёра о максимальном размере списка заголовков, который отправитель готов получать (в октетах). Значение учитывает размер несжатых полей заголовков, включая размер имени и значения в октетах с добавлением «отягощения» (overhead) в 32 октета на каждое поле заголовка.

Для любого конкретного запроса **может** быть установлен предел ниже заявленного. Начальное значение для этого параметра не устанавливается.

Конечная точка, получившая кадр SETTINGS с неизвестными или неподдерживаемыми идентификаторами, **должна** игнорировать такой кадр.

6.5.3. Синхронизация настроек

Большинство значений в кадрах SETTINGS начинают работать после получения кадра партнёром и применения им этого значения и отправителю следует понимать, когда это было сделано. Для обеспечения такой синхронизации получатель кадра SETTINGS со сброшенным флагом ACK **должен** применить обновлённые параметры сразу же после получения кадра.

Значения из кадра SETTINGS **должны** обрабатываться в порядке их следования без включения обработки других кадров между обработкой двух значений из одного кадра. Не поддерживаемые параметры **должны** игнорироваться. После обработки всех значений получатель **должен** незамедлительно отправить кадр SETTINGS с установленным флагом ACK. При получении кадра SETTINGS с флагом ACK отправитель изменённых параметров может считать, что они применены партнёром.

Если отправитель кадра SETTINGS не получает подтверждения в течение разумного интервала, он **может** передать сигнал об ошибке соединения (параграф 5.4.1) типа SETTINGS_TIMEOUT.

6.6. PUSH_PROMISE

Кадры PUSH_PROMISE (type=0x5) служат для упреждающего уведомления конечной точки-партнера об инициировании потоков. Кадр PUSH_PROMISE включает 31-битовый беззнаковый индикатор потока, который конечная

точка планирует открыть, вместе с набором заголовков, обеспечивающих для потока дополнительный контекст. Использование кадров PUSH_PROMISE подробно описано в параграфе 8.2.

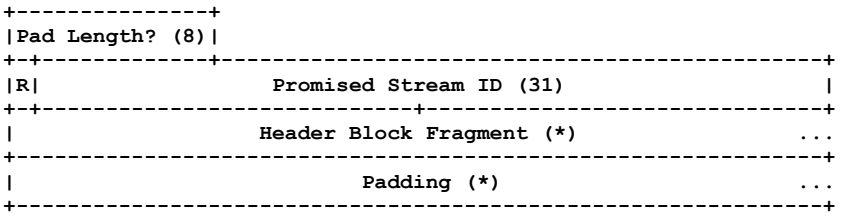

Рисунок 11. Информация кадра PUSH_PROMISE.

Информационные поля кадра PUSH_PROMISE описаны ниже.

Pad Length
8-битовое поле, содержащее размер заполнения кадра в октетах. Это поле присутствует только при установленном флаге PADDED.

R
Один резервный бит.

Promised Stream ID
31-битовое целое число без знака, идентифицирующее поток, который резервируется с помощью PUSH_PROMISE. Идентификатор предлагаемого потока **должен** быть приемлемым выбором для следующего потока, передаваемого отправителем (см. параграф 5.1.1).

Header Block Fragment
Фрагмент блока заголовка (параграф 4.3), содержащий поля заголовка запроса.

Padding
Октеты заполнения.

Ниже описаны флаги, определённые для кадров PUSH_PROMISE.

END_HEADERS (0x4)
Установленный бит 2 показывает, что этот кадр целиком содержит блок заголовка (параграф 4.3) и за ним не следует кадров CONTINUATION.
Кадр PUSH_PROMISE без флага END_HEADERS **должен** сопровождаться последующим кадром CONTINUATION для того же потока. Получатель **должен** считать получение вслед за таким кадром END_HEADERS кадра другого типа или кадра для другого потока ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

PADDED (0x8)
Установленный бит 3 говорит о наличии поля Pad Length и соответствующего размера поля заполнения.

Кадры PUSH_PROMISE **должны** передаваться только в инициированный партнёром поток, находящийся в состоянии open или half-closed (remote). Идентификатор потока в кадре PUSH_PROMISE указывает поток, с которым кадр связан. Получение такого кадра с идентификатором потока 0x0 **должен** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Предложенные потоки не требуется применять в том порядке, в котором они были обещаны. Кадры PUSH_PROMISE лишь резервируют идентификаторы потока для последующего использования.

Кадры PUSH_PROMISE **недопустимо** передавать, если для партнёра установлено значение SETTINGS_ENABLE_PUSH = 0. Конечная точка с такой установкой, если она была подтверждена, **должна** считать получение кадра PUSH_PROMISE ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Получатели кадров PUSH_PROMISE могут отвергнуть предложенные потоки, возвращая RST_STREAM с указанием идентификатора потока отправителю кадра PUSH_PROMISE.

Кадры PUSH_PROMISE меняют состояние соединения двумя путями. Во-первых, включение блока заголовка (параграф 4.3) может поменять состояние, поддерживаемое в части сжатия заголовков. Во-вторых, PUSH_PROMISE также резервирует поток для последующего использования, переводя предложенный поток в состояние reserved. Отправителю **недопустимо** передавать кадр PUSH_PROMISE в поток, который не находится в состоянии open или half-closed (remote), отправитель **должен** обеспечить пригодность предложенного потока для выбора в качестве нового (параграф 5.1.1) (т. е., предложенный поток **должен** находиться в состоянии idle).

Поскольку кадр PUSH_PROMISE резервирует поток, игнорирование PUSH_PROMISE делает состояние потока неопределённым. Получатель кадра PUSH_PROMISE в потоке с состоянием отличным от open и 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 полученные кадры PUSH_PROMISE, как будто они приняты до того, как был отправлен и обработан кадр RST_STREAM.

Получатель **должен** считать кадр PUSH_PROMISE, предлагающий недопустимый идентификатор потока (параграф 5.1.1), ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Отметим, что недопустимыми считаются идентификаторы потоков, которые не находятся в состоянии idle.

Кадры PUSH_PROMISE могут включать заполнение. Поля и флаги заполнения идентичны описанным для кадров DATA (параграф 6.1).

6.7. PING

Кадры PING (type=0x6) обеспечивают механизм измерения минимального времени кругового обхода от отправителя, а также проверки работоспособности соединения. Кадры PING может передавать любая из конечных точек.

В дополнение к заголовку кадр PING **должен** содержать 8 октетов не обрабатываемых (opaque) данных в информационном поле. Отправитель может выбирать содержимое этого поля произвольным способом.

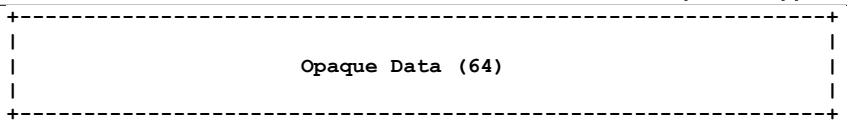

Рисунок 12. Информация кадра PING

Получатель кадра PING без флага ACK **должен** передать в ответ кадр PING с установленным флагом ACK и исходным информационным полем (payload). Откликам PING **следует** отдавать приоритет по отношению к другим кадрам.

Для кадров PING определён описанный ниже флаг.

ACK (0x1)

Установленный бит 0 показывает, что данный кадр PING является откликом на запрос PING. Конечные точки **должны** устанавливать этот флаг в откликах PING. Конечным точкам **недопустимо** отвечать на кадры PING с установленным флагом.

Кадры PING не связываются с каким-либо потоком. Получение кадра PING с отличным от 0x0 идентификатором потока **должно** считаться ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Получение кадра PING с отличным от 8 полем размера **должно** считаться ошибкой соединения (параграф 5.4.1) типа FRAME_SIZE_ERROR.

6.8. GOAWAY

Кадры GOAWAY (type=0x7) служат для инициирования разрыва соединения или сигнализации о серьёзной ошибке. GOAWAY позволяет конечной точке аккуратно прекратить восприятие новых потоков, продолжая обработку ранее созданы до их завершения. Это важно для административных операций типа обслуживания серверов.

Между конечной точкой, начинающей новые потоки, и удалённой точкой, передающей кадр GOAWAY, возникает «соперничество». Для его преодоления в кадр GOAWAY включается идентификатор последнего инициированного партнёром потока, который передающая кадр конечная точка может обработать в рамках данного соединения. Например, если сервер передаёт кадр GOAWAY, в нем указывается инициированный клиентом поток с максимальным номером.

Передав кадр, отправитель будет игнорировать кадры, отправленные другой стороной через потоки, идентификатор которых превышает значение, включённое в GOAWAY. Получателю GOAWAY **недопустимо** создавать в соединении дополнительные потоки, но он может организовать для них новое соединение.

Если получатель кадра GOAWAY уже передал данные в поток, идентификатор которого превышает указанное в кадре GOAWAY значение, этот поток не будет обработан. Получатель кадра GOAWAY может считать, что этот поток просто не создавался и предпринять попытку ещё раз создать его позднее в новом соединении.

Конечным точкам **следует** всякий раз передавать кадр GOAWAY перед завершением соединения, чтобы удалённая сторона могла понимать какие из потоков будут обработаны. Например, если клиент HTTP передаёт запрос POST в тот же момент, когда сервер закрывает соединение, клиент просто не может знать начал ли сервер обработку этого запроса, если сервер не передаст кадр GOAWAY, показывающий какие потоки будут обработаны.

При некорректном поведении партнёра конечная точка может закрыть соединение без отправки кадра GOAWAY.

Кадр GOAWAY не обязательно присутствует закрытию соединения непосредственно. Получателю кадра GOAWAY, который больше не намерен использовать соединение, **следует** передать кадр GOAWAY перед разрывом соединения.

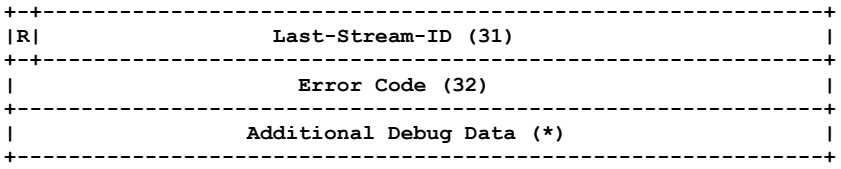

Рисунок 13. Информация кадра GOAWAY.

Для кадров GOAWAY не определено каких-либо флагов.

Кадры GOAWAY связаны с соединением, а не с конкретным потоком. Конечная точка **должна** считать кадры GOAWAY с отличным от 0x0 идентификатором потока ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

Идентификатор последнего потока в кадре GOAWAY указывает наибольший номер потока, для которого отправитель GOAWAY может предпринимать те или иные действия. Все потоки, вплоть до указанного значения идентификатора (включая его) будут так или иначе обработаны. Для последнего потока может быть указано значение 0, если потоки больше не будут обрабатываться совсем.

Примечание. В этом контексте «обработка» означает, что некие данные из потока будут переданы некому вышележащему программному уровню, который в результате может выполнить те или иные действия.

Если соединение разрывается без передачи кадра GOAWAY, идентификатор последнего потока эффективно служит максимально возможным идентификатором.

В потоках с меньшими или равным последнему значениями идентификаторов, которые не были полностью закрыты до разрыва соединения, попытки запросов, транзакций и иных протокольных действий невозможны, за исключением идемпотентных[1] запросов типа HTTP GET, PUT или DELETE. Любые протокольные действия с потоками, идентификаторы которых превышают последний, можно безопасно повторить в новом соединении.

Действия в потоках, номера которых не превышают значение идентификатора последнего потока, могут быть выполнены полностью. Отправитель кадра GOAWAY может аккуратно закрыть соединение путём передачи кадра GOAWAY и поддержки для соединения состояния open пока не будут завершены все предшествующие потоки.

[1]Дающих такой же результат при повторном запросе. *Прим. перев.*

При изменении ситуации конечная точка **может** передать множество кадров GOAWAY. Например, конечная точка, передавшая кадр GOAWAY с кодом NO_ERROR, в процессе аккуратного разрыва соединения может столкнуться с обстоятельствами, требующими немедленно разорвать соединение. Идентификатор последнего потока из полученного сообщения GOAWAY показывает поток, до которого может продолжаться обработка. Конечным точкам **недопустимо** увеличивать значение идентификатора последнего потока, поскольку партнёры могут уже повторить необработанные запросы через другое соединение.

Клиент, не способный повторять запросы, будет терять все запросы, которые были «на лету» в момент разрыва соединения сервером. Это верно и для промежуточных устройств (посредников), которые могут не обслуживать клиентов, использующих HTTP/2. Серверу, который пытается аккуратно завершить соединение, **следует** передать начальный кадр GOAWAY с идентификатором последнего потока 2^{31}-1 и кодом NO_ERROR. Это информирует клиента о неизбежности разрыва соединения и запрете отправки новых запросов. Выждав время для находящихся в процессе передачи запросов на создание потока (хотя бы один интервал кругового обхода), сервер может передать другой кадр GOAWAY с обновлённым идентификатором последнего потока. Это позволяет разорвать соединение без потери запросов.

После передачи кадра GOAWAY отправитель может отбрасывать кадры от потоков, инициированных его получателем, если они содержат идентификаторы сегментов с номерами выше последнего. Однако кадры, меняющие состояние соединения, не могут быть полностью проигнорированы. Например, кадры HEADERS, PUSH_PROMISE и CONTINUATION **должны** пройти минимальную обработку для поддержки согласованности состояния сжатия заголовков (см. параграф 4.3). Кадры DATA **должны** учитываться в окне управления потоком данных соединения. Отказ от обработки указанных кадров может нарушать синхронизацию состояний управления потоком данных и сжатия заголовков.

Кадр GOAWAY содержит также 32-битовый код ошибки (параграф 7), указывающий причину разрыва соединения.

Конечные точки **могут** добавлять «непрозрачные» (opaque) данные в информационное поле любого кадра GOAWAY. Дополнительные отладочные данные предназначены лишь для диагностики и семантика их не важна. Отладочная информация может содержать конфиденциальные данные, относящиеся к безопасности или приватности. Запись отладочных данных в системный журнал или иное хранилище **должна** сопровождаться адекватными мерами защиты от несанкционированного доступа.

6.9. WINDOW_UPDATE

Кадры WINDOW_UPDATE (type=0x8) служат для реализации управления потоком данных (см. параграф 5.2).

Управление потоком данных осуществляется на двух уровнях - отдельные потоки и соединение в целом.

Оба варианта управления потоком являются поэтапными (hop by hop), т. е., реализуются между парой конечных точек. Промежуточные устройства не пересылают кадры WINDOW_UPDATE между зависимыми соединениями. Однако дросселирование (торможение) передачи данных любым получателем может опосредованно вызывать распространение информации управления потоком данных в направлении исходного отправителя.

Управление потоком данных применяется лишь для кадров, обозначенных в качестве управляемых. Этот документ определяет единственный тип управляемых кадров - DATA. Кадры, не включаемые в управление потоком данных, **должны** восприниматься и обрабатываться, пока получатель способен выделять ресурсы для их обслуживания. Получатель **может** указать на ошибку потока (параграф 5.4.2) или соединения (параграф 5.4.1) типа FLOW_CONTROL_ERROR, если он не способен воспринять кадр.

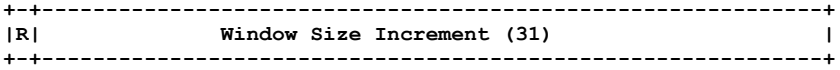

Рисунок 14. Информация кадра WINDOW_UPDATE.

Информационное поле (payload) кадра WINDOW_UPDATE представляет собой один резервный бит и 31-битовое целое число без знака, указывающее число октетов, которые отправитель может передать в дополнение к имеющемуся окну управления потоком данных. Допустимые значения добавок к окну управления потоком данных лежат в диапазоне от 1 до 2^{31}-1 (2 147 483 647) октетов.

Для кадров WINDOW_UPDATE не определено каких-либо флагов.

Кадр WINDOW_UPDATE может относиться к конкретному потоку или соединению в целом. В первом случае в кадре указывается идентификатор соответствующего потока, во втором идентификатор имеет значение 0.

Получатель **должен** считать кадр WINDOW_UPDATE с нулевым инкрементом окна управления потоком данных ошибкой потока (параграф 5.4.2) типа PROTOCOL_ERROR или ошибкой соединения (параграф 5.4.1), если кадр относится к соединению в целом.

Кадр WINDOW_UPDATE может быть передан отправителем кадра с флагом END_STREAM. В таких случаях получатель может принять кадр WINDOW_UPDATE через поток в состоянии half-closed (remote) или closed. **Недопустимо** считать этот случай ошибкой (см. параграф 5.1).

Получатель кадра, на который воздействует управление потоком данных, всегда **должен** учитывать такой кадр в окне управления потоком данных соединения, если данный кадр не связан с ошибкой соединения (параграф 5.4.1). Это требуется делать даже для ошибочных кадров. Отправитель учитывает кадр в окне управления потоком данных и если получатель не будет делать то же, окна управления потоком данных у получателя и отправителя могут стать разными.

Кадры WINDOW_UPDATE с размером, отличающимся от 4 октетов, **должны** считаться ошибкой соединения (параграф 5.4.1) типа FRAME_SIZE_ERROR.

6.9.1. Окно управления потоком данных

Управление потоком данных в HTTP/2 реализуется с использованием окна, сохраняемого каждым отправителем в каждом потоке. Окно управления потоком данных задаётся целочисленным значением, показывающим число данных,

которые разрешено передать отправителю. Таким образом, размер окна определяется буферной ёмкостью получателя.

Применимы два окна управления потоком данных - одно для потока, другое для соединения в целом. Отправителю **недопустимо** передавать кадры, к которым применяется управление потоком данных, если они выходят за пределы пространства, доступного в любом из анонсированных получателем окон управления потоком данных. Кадры нулевого размера с установленным флагом END_STREAM (т. е., пустые кадры DATA) **могут** передаваться даже при отсутствии пространства в окнах управления потоком данных.

При расчётах для управления потоком данных 9 октетов заголовка кадра не учитываются.

После передачи кадра, к которому применяется управление потоком данных, отправитель уменьшает размер доступного пространства в обоих окнах управления потоком на размер переданного кадра.

Получатель кадра передаёт кадр WINDOW_UPDATE после того, как он воспримет данные и освободит пространство в окне управления потоком данных. Для окон управления потоком данных соединения в целом и отдельного потока передаются раздельные кадры WINDOW_UPDATE.

Отправитель, получив кадр WINDOW_UPDATE, увеличивает соответствующее окно на указанный в кадре размер. Отправителю **недопустимо** увеличивать размер окна управления потоком данных сверх 2^{31}-1 октетов. Если отправитель получает кадр WINDOW_UPDATE со значением, приводящим к нарушению этого предела, он **должен** разорвать поток или соединение, к которому это относится. Для потоков в таких случаях отправитель передаёт RST_STREAM с кодом ошибки FLOW_CONTROL_ERROR, для соединений - кадр GOAWAY с кодом ошибки FLOW_CONTROL_ERROR.

Кадры, подлежащие управлению потоком от отправителя и кадры WINDOW_UPDATE от получателя не имеют какой-либо синхронизации. Это позволяет получателю активно обновлять размер окна, сохраняемый отправителем, для предотвращения «замораживания» потока.

6.9.2. Начальный размер окна управления потоком данных
При первоначальной организации соединения HTTP/2 новые потоки создаются с начальным окном управления потоком данных в в 65535 октетов. Размер окна управления потоком данных для всего соединения также составляет 65535 октетов. Обе конечных точки могут установить начальный размер окна для новых потоков, включая значение SETTINGS_INITIAL_WINDOW_SIZE в кадр SETTINGS, формирующий часть префикса соединения. Размер окна управления потоком данных для соединения в целом можно изменить лишь с помощью кадров WINDOW_UPDATE.

До получения кадра SETTINGS, устанавливающего значение SETTINGS_INITIAL_WINDOW_SIZE, конечная точка может использовать при передаче кадров, для которых используется управление потоком, лишь принятого по умолчанию размера окна. Аналогично на уровне соединения в целом используется принятый по умолчанию начальный размер окна до получения кадра WINDOW_UPDATE.

Кроме изменения размера окна управления потоком данных для ещё не созданных потоков кадр SETTINGS может менять размер начального окна для потоков с уже активным окном управления потоком данных (т. е., потоков в состоянии open или half-closed (remote)). При изменении SETTINGS_INITIAL_WINDOW_SIZE получатель **должен** исправить размер окна управления потоком данных для всех потоков, которые он поддерживает, на разницу между новым и прежним значением.

Изменение SETTINGS_INITIAL_WINDOW_SIZE может сделать размер доступного в окне управления потоком данных пространства отрицательным. Отправитель **должен** отслеживать возникновение окон отрицательного размера и **недопустимо** передавать новые кадры, для которых используется управление потоком данных, пока не будет получен кадр WINDOW_UPDATE, увеличивающий размер окна управления потоком данных до положительного значения.

Например, если клиент передал 60 Кбайт данных сразу после организации соединения, а сервер установил для начального окна размер 16 Кбайт, клиент получит для доступного в окне управления потоком пространства отрицательное значение -44 АН при получении кадра SETTINGS. Клиент сохраняет отрицательный размер окна управления потоком данных, пока кадры WINDOW_UPDATE не восстановят положительное значение и лишь после этого клиент может возобновить передачу.

Кадры SETTINGS не могут менять окно управления потоком данных соединения.

Конечная точка **должна** считать значение SETTINGS_INITIAL_WINDOW_SIZE, увеличивающее размер окна управления потоком данных сверх максимально разрешённого значения, ошибкой соединения (параграф 5.4.1) типа FLOW_CONTROL_ERROR.

6.9.3. Снижение размера окна управления для потока
Получатель, желающий уменьшить окно управления потоком данных по сравнению с текущим значением, может отправить новый кадр SETTINGS. Однако этот получатель **должен** быть готов к получению большего объёма данных,, которые отправитель мог уже передать до обработки кадра SETTINGS.

После отправки кадра SETTINGS, уменьшающего окно управления потоком данных, получатель **может** продолжать обработку потоков, выходящих за пределы окна. Это разрешение не позволяет получателю незамедлительно снизить объем доступного для окон управления потоком данных пространства памяти. Обработка этих потоков может быть «заморожена», поскольку отправителю для продолжения передачи будут требоваться кадры WINDOW_UPDATE. Получатель **может** вместо этого передать RST_STREAM с кодом ошибки FLOW_CONTROL_ERROR для соответствующих потоков.

6.10. CONTINUATION
Кадры CONTINUATION (type=0x9) служат для продолжения последовательности фрагментов блока заголовка (параграф 4.3). Может передаваться любое число кадров CONTINUATION, пока такому кадру предшествует кадр HEADERS, PUSH_PROMISE или CONTINUATION без флага END_HEADERS.

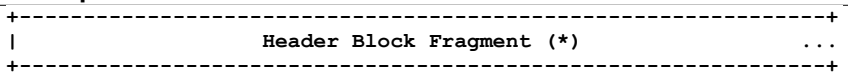

Рисунок 15. Информация кадра CONTINUATION

Информационное поле кадра CONTINUATION содержит фрагмент блока заголовка (параграф 4.3).

Для кадров CONTINUATION определён один флаг, описанный ниже.

END_HEADERS (0x4)

Установленный бит 2 показывает, что данный кадр завершает блок заголовка (параграф 4.3).

Если флаг END_HEADERS не установлен, за этим кадром **должен** следовать другой кадр CONTINUATION. Получатель **должен** в таких случаях считать получение любого другого типа кадра или кадра в другом потоке ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

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

Кадры CONTINUATION **должны** быть привязаны к потокам. Приём кадра CONTINUATION с полем идентификатора потока 0x0 получатель **должен** считать ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

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

7. Коды ошибок

Коды ошибок указываются 32-битовыми полями в кадрах RST_STREAM и GOAWAY для передачи информации о причине ошибки потока или соединения.

Для кодов ошибок потоков и соединений используется общее пространство значений. Некоторые коды применимы лишь к потоку или соединению в целом и не имеют смысла в ином контексте.

Определённые в данной спецификации коды ошибок приведены ниже.

NO_ERROR (0x0)

Соответствующее условие не является результатом ошибки. Например, кадр GOAWAY может включать этот код для индикации аккуратного разрыва соединения.

PROTOCOL_ERROR (0x1)

Конечная точка столкнулась с неуказанной ошибкой протокола. Этот код применяется в тех случаях, когда для ошибки нет более конкретного кода.

INTERNAL_ERROR (0x2)

Конечная точка столкнулась с неожиданной внутренней ошибкой.

FLOW_CONTROL_ERROR (0x3)

Конечная точка обнаружила нарушение её партнёром протокола управления потоком данных.

SETTINGS_TIMEOUT (0x4)

Конечная точка передала кадр SETTINGS, но не получила своевременного отклика (см. параграф 6.5.3. Синхронизация настроек).

STREAM_CLOSED (0x5)

Конечная точка получила кадр после того, как поток был наполовину закрыт (half-closed).

FRAME_SIZE_ERROR (0x6)

Конечная точка получила кадр с неприемлемым размером.

REFUSED_STREAM (0x7)

Конечная точка отвергла поток до выполнения какой-либо прикладной обработки (см. параграф 8.1.4).

CANCEL (0x8)

Используется конечной точкой для индикации того, что поток больше не требуется.

COMPRESSION_ERROR (0x9)

Конечная точка не способна поддерживать контекст сжатия заголовков для соединения.

CONNECT_ERROR (0xa)

Соединение, организованное по запросу CONNECT (параграф 8.3) было сброшено или аварийно завершено.

ENHANCE_YOUR_CALM (0xb)

Конечная точка обнаружила у партнёра поведение, которое может создать избыточную нагрузку.

INADEQUATE_SECURITY (0xc)

Свойства нижележащего транспорта не соответствуют требованиям безопасности (см. параграф 9.2).

HTTP_1_1_REQUIRED (0xd)

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

Недопустимо запускать какую-либо специальную обработку не известных или не поддерживаемых кодов ошибок. Их **можно** трактовать, как INTERNAL_ERROR.

8. Обмен сообщениями HTTP

Протокол HTTP/2 разрабатывался с учётом обеспечения возможной совместимости с текущими приложениями HTTP. Это означает, что с точки зрения приложений большая часть функциональности сохраняется неизменной. Для обеспечения этого сохранена семантика всех запросов и откликов, хотя синтаксис передачи этой семантики изменён.

Таким образом, спецификации и требования к семантике и содержимому HTTP/1.1 (Semantics and Content) [RFC7231], условным запросам (Conditional Requests) [RFC7232], запросам части документа (Range Requests) [RFC7233], кэшированию (Caching) [RFC7234] и проверке подлинности (Authentication) [RFC7235] остаются применимыми в HTTP/2. Отдельные части синтаксиса сообщений и маршрутизации HTTP/1.1 (Message Syntax and Routing) [RFC7230], типа схем URI для HTTP и HTTPS также применимы в HTTP/2, но выражение их семантики для этого протокола отличается и описано ниже.

8.1. Обмен запросами и откликами HTTP

Клиент передаёт запрос HTTP на организацию нового потока, указывая не использованный ранее идентификатор потока (параграф 5.1.1). Сервер передаёт отклик HTTP с запрошенным идентификатором потока.

Сообщение HTTP (запрос или отклик) включает:

1. отклики (и только они) могут включать один или множество кадров HEADERS (за каждым может следовать один или множество кадров CONTINUATION), содержащих заголовки информационных (1xx) откликов HTTP (см. параграф 3.2 в [RFC7230] и параграф 6.2 в [RFC7231]);
2. один кадр HEADERS (за которым может следовать один или множество кадров CONTINUATION), содержащий заголовки сообщения (см. параграф 3.2 в [RFC7230]);
3. может содержать кадры DATA с информационными полями (payload) сообщения (см. параграф 3.3 в [RFC7230]);
4. дополнительно может содержать один кадр HEADERS, за которым могут следовать кадры CONTINUATION, с трейлерной частью (trailer-part), если она имеется (см. параграф 4.1.2 в [RFC7230]).

Последний кадр в цепочке имеет флаг END_STREAM, показывающий, что за этим кадром HEADERS могут следовать кадры CONTINUATION с оставшимися частями блока заголовков.

Другие кадры (из любого потока) **недопустимо** помещать между кадром HEADERS и следующими за ним кадрами CONTINUATION.

HTTP/2 использует кадры DATA для передачи информационного содержимого сообщений. В HTTP/2 **недопустимо** использовать описанное в параграфе 4.1 [RFC7230] chunked-кодирование передачи.

Трейлерный (задние) поля заголовка передаются в блоке заголовка, который также завершает поток. Текой блок заголовка представляет собой последовательность, начинающуюся с кадра HEADERS, за которым может следовать один или несколько кадров CONTINUATION, при этом кадр HEADERS включает флаг END_STREAM. Блоки заголовка после первого, которые не завершают поток, не являются частью запроса или отклика HTTP.

Кадр HEADERS (и связанные с ним кадры CONTINUATION) может появляться только в начале или в конце потока. Конечная точка, получившая кадр HEADERS без флага END_STREAM после приёма финального (не информационного) кода состояния, **должна** считать соответствующий запрос или отклик сформированным некорректно (параграф 8.1.2.6).

Обмен HTTP запрос-отклик полностью занимает один поток. Запрос начинается с кадра HEADERS, который переводит поток в состояние open. Завершается запрос кадром с флагом END_STREAM, который переводит поток в состояние half-closed (local) для клиента и half-closed (remote) для сервера. Отклик начинается с кадра HEADERS и завершается кадром с флагом END_STREAM, который переводит поток в состояние closed .

Отклик HTTP завершается после того, как сервер передаст (или клиент получит) кадр с флагом END_STREAM (включая все кадры CONTINUATION, требуемые для завершения блока заголовка). Сервер может передать полный отклик до того, как клиент полностью передаст запрос, если отклик не зависит от какой-либо части запроса, которая ещё не передана и не получена. Когда это справедливо, сервер **может** запросить у клиента прерывание запроса без ошибки путём отправки RST_STREAM с кодом NO_ERROR после передачи всего отклика (т. е., кадра с флагом END_STREAM). Клиенту **недопустимо** отбрасывать отклики в результате получения RST_STREAM, хотя клиент всегда может по своему разумению отбрасывать отклики в связи с другими причинами.

8.1.1. Обновление HTTP/2

HTTP/2 прекращает поддержку информационного кода состояния 101 (Switching Protocols), описанного в параграфе 6.2.2 [RFC7231].

Семантика кода 101 не применима к протоколам с мультиплексированием. Другие протоколы способны применять те же механизмы, которые в протоколе HTTP/2 служат для согласования его использования (см. параграф 3).

8.1.2. Поля заголовка HTTP

Информация в полях заголовка HTTP представляется в форме последовательности пар «ключ - значение» (key-value). Список зарегистрированных полей заголовков HTTP можно найти в реестре Message Header Field, доступном по ссылке https://www.iana.org/assignments/message-headers.

Как и в HTTP/1.x, имена полей заголовка представляются строками символов ASCII, которые сравниваются без учёта регистра символов. Однако при кодировании в HTTP/2 имена полей заголовка **должны** приводиться к нижнему регистру. Запросы и отклики, содержащие заглавные буквы в именах полей заголовков, **должны** считаться некорректно сформированными (параграф 8.1.2.6).

8.1.2.1. Поля псевдозаголовка

Хотя в HTTP/1.x используется стартовая строка сообщения (см. параграф 3.1 в [RFC7230]) для передачи целевого идентификатора URI, метода запроса и кода состояния в отклике, в для этого HTTP/2 служат специальные поля псевдозаголовка, начинающиеся с символа двоеточия («:», ASCII 0x3a).

Поля псевдозаголовка не относятся к заголовку HTTP. Конечным точкам **недопустимо** генерировать поля псевдозаголовка за исключением определённых в этом документе.

Поля псевдозаголовка действительны лишь в том контексте, где они определены. Поля псевдозаголовка, определённые для запросов, **недопустимо** включать в отклики и наоборот. Поля псевдозаголовка **недопустимо** включать в трейлер. Конечные точки **должны** считать запросы и отклики с неопределёнными или недействительными полями псевдозаголовков ошибкой форматирования (параграф 8.1.2.6).

Все поля псевдозаголовка **должны** размещаться в блоке заголовка до обычных полей заголовка. Любые запросы и отклики с полями псевдозаголовка после полей обычного заголовка **должны** считаться ошибкой форматирования (параграф 8.1.2.6).

8.1.2.2. Поля заголовка, относящиеся к соединению

HTTP/2 не использует поле заголовка Connection для индикации связанных с соединением полей заголовка, в этом протоколе относящиеся к соединению метаданные передаются иными способами. Конечный точкам **недопустимо** создавать сообщение HTTP/2 с относящимися к соединению полями заголовка. При получении сообщения с такими полями оно **должно** считаться некорректно сформированным (параграф 8.1.2.6).

Единственным исключением является поле TE, которое **может** присутствовать в заголовках запросов HTTP/2, но в это поле **недопустимо** включать какие-либо значения кроме trailers.

Это означает, что промежуточные узлы, преобразующие сообщения HTTP/1.x в HTTP/2 должны удалять все поля заголовка, указанные в поле Connection вместе с этим полем. Таким промежуточным узлам **следует** также удалять все относящиеся к соединению поля заголовка типа Keep-Alive, Proxy-Connection, Transfer-Encoding и Upgrade, даже если они не указаны в поле Connection.

> Примечание. HTTP/2 осознанно не поддерживает обновление до других протоколов. Методы согласования, описанные в разделе 3, считаются достаточными для согласования использования дополнительных протоколов.

8.1.2.3. Поля псевдозаголовка запроса

Поля псевдозаголовков, определённые для запросов HTTP/2 перечислены ниже.

- Поле :method включает метод HTTP (раздел 4 [RFC7231]).
- Поле :scheme включает относящуюся к схеме часть целевого идентификатора URI (параграф 3.1 [RFC3986]).

  Значения поля :scheme не ограничиваются схемами http и https для идентификаторов URI. Прокси-серверы и шлюзы могут транслировать запросы для схем, не относящихся к HTTP, позволяя использовать HTTP для взаимодействия с другими (не HTTP) службами.
- Поле :authority включает часть authority идентификатора URI (параграф 3.2 [RFC3986]). **Недопустимо** включение в это поле отменённой субкомпоненты userinfo для URI со схемой http или https.

  Для обеспечения гарантии точно воспроизведения строки запроса HTTP/1.1 это поле **должно** опускаться при трансляции запросов HTTP/1.1, в которых цель запроса указана в форме origin или asterisk (см. параграф 5.3 в [RFC7230]). Клиентам, генерирующим запросы HTTP/2 непосредственно, **следует** использовать поле псевдозаголовка :authority вместо поля заголовка Host. Промежуточные узлы, преобразующие запрос HTTP/2 в HTTP/1.1, **должны** создавать поле заголовка Host, если его нет в запросе, путём копирования содержимого поля псевдозаголовка :authority.
- Поле :path включает части path и query целевого идентификатора URI (path-absolute и необязательная компонента из символа ? И следующей за ним части query, как описано в параграфах 3.3 и 3.4 [RFC3986]). Запрос в форме звёздочки (asterisk) включает символ * для поля псевдозаголовка :path.

  Недопустимо оставлять это поле пустым для идентификаторов URI со схемой http или https. Идентификаторы URI со схемами http и https, не содержащие компоненты path, **должны** включать значение /. Исключением из этого правила является запрос OPTIONS для http или https URI, который не включает компоненты path. Эти запросы **должны** включать поле псевдозаголовка :path со значением * (см. параграф 5.3.4 [RFC7230]).

Все запросы HTTP/2, за исключением CONNECT (параграф 8.3), **должны** включать в точности по одному приемлемому значению для полей псевдозаголовка :method, :scheme и :path. Запросы HTTP с пропущенными обязательными полями псевдозаголовка считаются некорректно сформированными (параграф 8.1.2.6).

HTTP/2 не определяет способа передачи идентификатора версии, который включается в строку запроса HTTP/1.1.

8.1.2.4. Поля псевдозаголовка отклика

Для откликов HTTP/2 определено одно поле псевдозаголовка :status, которое включает код состояния HTTP (см. раздел 6 в [RFC7231]). Это поле **должно** включаться во все отклики, которые в противном будут считаться некорректно сформированными (параграф 8.1.2.6).

HTTP/2 не определяет способа передачи идентификатора версии, который включается в строку состояния HTTP/1.1.

8.1.2.5. Сжатие поля Cookie в заголовке

В полях заголовка Cookie [COOKIE] используется точка с запятой (;) для разделения cookie-пар (или «крошек» - crumbs). Для этого поля не выполняется правило создания списков в HTTP (см. параграф 3.2.2 в [RFC7230]), что препятствует разбиению cookie-пар на разные пары «имя-значение» (name-value). Это может существенно снижать эффективность сжатия при обновлении отдельных cookie-пар.

Для повышения эффективности сжатия поле заголовка Cookie **можно** разделить на отдельные поля заголовка, каждое их которых включает одну или множество cookie-пар. Если после декомпрессии имеется множество полей Cookie, они **должны** быть собраны (конкатенация) в одну строку октетов с двухоктетными разделителями 0x3B, 0x20 (точка с запятой и пробел - строка ASCII «; ») до их передачи в не относящийся к HTTP/2 контекст типа соединения HTTP/1.1 или серверного приложения базового HTTP.

Следовательно приведённые ниже два списка полей Cookie семантически эквивалентны.

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

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

8.1.2.6. Запросы и отклики с некорректным форматом

Некорректно сформированными считаются запросы и отклики, содержащие допустимую последовательность кадров HTTP/2, но неприемлемые по причине наличия избыточных кадров, запрещённых полей в заголовке, отсутствия в заголовке обязательных полей или указания имён полей заголовка с использованием заглавных букв (верхний регистр).

Запрос или отклик с информационными полями (payload) может включать в заголовок поле размера содержимого. Если значение этого поля не соответствует сумме размеров информационных полей в кадрах DATA с содержимым, такой запрос или отклик считается некорректно сформированным. Отклик, для которого не определены информационные поля, как описано в параграфе 3.3.2 [RFC7230], может иметь отличное от 0 поле размера содержимого даже при отсутствии содержимого в кадрах DATA.

Промежуточным узлам, обрабатывающим запросы или отклики HTTP (т. е., всем промежуточным узлам, не выступающим в качестве туннеля) **недопустимо** пересылать запросы или отклики, сформированные некорректно. Обнаруженные запросы или отклики этого вида **должны** считаться ошибкой потока (параграф 5.4.2) типа PROTOCOL_ERROR.

Для запросов некорректной формы сервер **может** передать отклик HTTP до закрытия или сброса потока. Клиентам **недопустимо** воспринимать отклики некорректной формы. Отметим, что эти требования предназначены для защиты от некоторых распространённых атак на HTTP и по этой причине достаточно строги, поскольку их ослабление может привести к раскрытию уязвимостей в реализациях.

8.1.3. Примеры

В этом параграфе приведены запросы и отклики HTTP/1.1 и эквивалентные запросы и отклики HTTP/2.

Запрос HTTP GET включает поля заголовка запроса без информационного элемента в теле и, следовательно, передаётся в одном кадре HEADERS, за которым могут следовать кадры CONTINUATION, содержащие упорядоченный блок полей заголовка запроса. Приведённый ниже кадр HEADERS имеет флаги END_HEADERS и END_STREAM, а кадры CONTINUATION не передаются.

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

Подобно этому, отклик, включающий только поля заголовка, передаётся в кадре HEADERS (за которым также могут следовать кадры CONTINUATION), содержащем упорядоченный блок 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 ...
```

Запрос HTTP POST, включающий заголовок запроса и информацию (payload), передаётся в кадре HEADERS, за которым могут следовать кадры CONTINUATION, содержащие поля заголовка запроса, а также следует один или множество кадров DATA. В последнем кадре CONTINUATION (или кадре HEADERS) устанавливается флаг END_HEADERS, а в последнем кадре DATA - 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
                                      :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, входящие в любое конкретное заголовка, могут быть разделены между фрагментами блока заголовка. Распределение полей заголовка поле по кадрам в примере приведено лишь для иллюстрации.

Отклик, включающий поля заголовка и данные (payload) передаётся в форме кадра HEADERS, за которым могут следовать кадры CONTINUATION, а затем один или множество кадров DATA, в последнем из которых установлен 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}
```

Информационные отклики с кодом состояния 1xx, отличным от 101, передаются в форме кадра HEADERS, за которым могут следовать кадры CONTINUATION.

Трейлерные поля заголовка передаются в виде блока заголовка после блока заголовка запроса или отклика и всех кадров DATA. Кадры HEADERS с трейлерным блоком заголовка имеют флаг END_STREAM.

Приведённый ниже пример включает код состояния 100 (Continue), который передаётся в откликах на запросы с маркером 100-continue в поле заголовка Expect, и трейлерные поля заголовка.

```
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-length = 123
123                                    content-type = image/jpeg
{двоичные данные}                      trailer = Foo
0
Foo: bar                           DATA
                                     - END_STREAM
                                   {двоичные данные}

                                   HEADERS
                                     + END_STREAM
                                     + END_HEADERS
                                       foo = bar
```

8.1.4. Механизмы обеспечения надёжности для запросов HTTP/2

В HTTP/1.1 клиент HTTP не способен повторить неидемпотентный запрос при возникновении ошибки, поскольку у него нет способа определения причины ошибки. До возникновения ошибки на сервере могла быть выполнена некоторая обработка и повторный запроса моет привести к нежелательным эффектам.

HTTP/2 обеспечивает два механизма предоставления клиенту гарантии того, что запрос не был обработан:

- кадр GOAWAY показывает наибольший номер потока, который мог быть обработан, поэтому для запроса с большим номером гарантируется безопасность повтора;
- в RST_STREAM может быть включён код ошибки REFUSED_STREAM для индикации того, что поток был закрыт до начала его обработки, поэтому любой запрос, переданный в сброшенный поток, можно безопасно повторить.

Необработанные запросы не связаны с отказами и клиенты **могут** автоматически повторять их даже при использовании неидемпотентных методов.

Серверу **недопустимо** указывать, что поток не обработан, пока он не может этот факт гарантировать. Если кадры потока переданы прикладному уровню для любого потока, для этого потока **недопустимо** использовать REFUSED_STREAM, а кадр GOAWAY **должен** включать идентификатор потока, который не меньше идентификатора данного потока.

В дополнение к этим механизмам кадры PING обеспечивают клиентам простой способ проверки соединений. Бездействующее соединение может быть разорвано, поскольку некоторые промежуточные устройства (например, трансляторы адресов и балансировщики нагрузки) просто отбрасывают без уведомления привязки соединений. Кадры PING позволяют клиенту безопасно проверить активность соединения без отправки запроса.

8.2. Выталкивание на сервере

HTTP/2 позволяет серверам отправлять с упреждением (или «выталкивать» - push) отклики (вместе с соответствующими «ожидаемыми» - promised - запросами), связанные с предыдущим запросом этого клиента. Это может быть полезно в тех случаях, когда сервер знает о наличии у него откликов для полного ответа на исходный запрос.

Клиент может запросить отключение выталкивания на сервере, однако это нужно независимо согласовывать для каждого этапа пересылки (hop). Для индикации запрета выталкивания служит установка SETTINGS_ENABLE_PUSH = 0.

Ожидаемые запросы **должны** быть кэшируемыми (см. параграф 4.2.3 в [RFC7231]), **должны** быть безопасными (см. параграф 4.2.1 в [RFC7231]) и в них **недопустимо** включать тело запроса. Клиент, получающий ожидаемый запрос, который не является кэшируемым является заведомо небезопасным или указывает наличие тела запроса, **должен** сбросить предложенный (promised) поток с ошибкой потока (параграф 5.4.2) типа PROTOCOL_ERROR. Отметим, что это может приводить к сбросу предложенного потока, если клиент не считает недавно определённый метод безопасным.

Вытолкнутые сервером отклики, которые являются кэшируемыми (см. раздел 3 в [RFC7234]), могут сохраняться клиентами, реализующими HTTP-кэш. Вытолкнутые отклики считаются в должной мере проверенными на сервере-источнике (например, если присутствует директива no-cache, как описано в параграфе 5.2.2 [RFC7234]), пока поток, указанный идентификатором предложенного потока, остаётся открытым.

Некэшируемые вытолкнутые отклики **недопустимо** сохранять в каком-либо кэше HTTP. Их **можно** сделать доступными приложению отдельным способом.

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

Промежуточный узел может принять вытолкнутый сервером отклик и не переслать его клиенту. Иными словами, использование вытолкнутых данных зависит от посредников. Точно также, промежуточный узел может дополнительно вытолкнуть данные клиенту без участия сервера.

Клиент не может использовать выталкивания. Поэтому сервер, получивший кадр PUSH_PROMISE, **должен** считать это ошибкой соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Клиенты **должны** отвергать любые попытки изменить установку для SETTINGS_ENABLE_PUSH отличных от нуля значений, трактуя сообщение как ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR.

8.2.1. Запросы на выталкивание

Выталкивание на сервере семантически эквивалентно ответу сервера на запрос, однако в этом случае сам запрос также передаёт сервер в форме кадра PUSH_PROMISE.

Кадр PUSH_PROMISE включает блок заголовка, содержащий полный набор полей заголовков запроса, которые сервер присваивает этому запросу. Невозможно вытолкнуть отклик на запрос, включающий тело запроса (request body).

Обещанные[1] запросы всегда связываются с явным запросом от клиента. Кадры PUSH_PROMISE, передаваемые сервером, отправляются в поток такого явного запроса. Кадр PUSH_PROMISE включает также обещанный (promised) идентификатор потока, который выбирается из доступных серверу идентификаторов потоков (см. параграф 5.1.1).

Поля заголовка в PUSH_PROMISE и любых последующих кадрах CONTINUATION **должны** быть действительным и полным набором полей заголовка запроса (параграф 8.1.2.3). Сервер **должен** включить в поле псевдозаголовка :method действительный и безопасный метод. Если клиент получает PUSH_PROMISE без действительного и полного набора полей или поле псевдозаголовка :method не является безопасным, он **должен** указать в ответе ошибку потока (параграф 5.4.2) типа PROTOCOL_ERROR.

Серверу **следует** передавать кадры PUSH_PROMISE (параграф 6.6) до отправки каких-либо кадров, ссылающихся на обещанные (promised) отклики. Это позволяет избежать ситуаций ввода клиентом запроса до получения каких-либо кадров PUSH_PROMISE.

Например, если сервер получает запрос на документ, содержащий встроенные ссылки на множество файлов с изображениями, и сервер решил «вытолкнуть» эти дополнительные изображения клиенту, отправка кадров PUSH_PROMISE до передачи кадров DATA, содержащих ссылки на эти изображения, гарантирует, что клиент может узнать о выталкивании ресурсов до того, как обнаружит ссылки на них. Аналогично, если сервер выталкивает отклики, упомянутые в блоке заголовка (например, в полях заголовка Link), отправка PUSH_PROMISE до передачи блока заголовка гарантирует, что клиент не будет запрашивать эти ресурсы.

Передача кадров PUSH_PROMISE клиентом **недопустима**.

Кадры PUSH_PROMISE могут передаваться сервером в отклике на любой инициированный клиентом поток, но поток **должен** находиться в состоянии open или half-closed (remote) по отношению к серверу. Кадры PUSH_PROMISE чередуются с кадрами, которые содержат ответ, но они не могут чередоваться с кадрами HEADERS и CONTINUATION, которые содержат один блок заголовка.

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

8.2.2. Отклики Push

После отправки кадра PUSH_PROMISE сервер может начать доставку вытолкнутого отклика как отклика (параграф 8.1.2.4) в инициированном сервером потоке, который использует обещанный идентификатор потока. Сервер использует этот поток для передачи отклика HTTP, используя последовательность кадров, определённую в параграфе 8.1. Этот поток становится «полузакрытым» для клиента (параграф 5.1) после передачи начального кадра HEADERS.

После того, как клиент получает кадр PUSH_PROMISE и решает воспринять вытолкнутый отклик, ему **не следует** вводить какие-либо запросы, пока не будет закрыт обещанный поток.

Если клиент по какой-либо причине принимает решение о нежелании получать выталкиваемые сервером отклики или сервер слишком долго не начинает отправку обещанного отклика, клиент может передать кадр RST_STREAM с кодом CANCEL или REFUSED_STREAM и указанием идентификатора выталкиваемого потока.

Клиент может использовать параметр SETTINGS_MAX_CONCURRENT_STREAMS для ограничения числа откликов, которые будут одновременно выталкиваться сервером. Анонсирование SETTINGS_MAX_CONCURRENT_STREAMS = 0 запрещает серверу выталкивать отклики, предотвращая создание ненужных потоков. Это не запрещает серверу передавать кадры PUSH_PROMISE; клиенты должны сбрасывать нежелательные потоки.

Клиенты, получающие вытолкнутый отклик, **должны** проверить полномочность сервера (см. параграф 10.1) или разрешение прокси, предоставившему такой отклик, выполнять эту операцию для соответствующего запроса. Например, серверу, который предоставляет сертификат лишь для example.com DNS-ID или Common Name, не разрешено выталкивать отклик 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, и завершается кадром с END_STREAM, который переводит поток в состояние closed.

Примечание. Клиент никогда не передаёт кадров с флагом END_STREAM для «серверного выталкивания».

[1] В оригинале ошибочно сказано «выталкиваемые». См. https://www.rfc-editor.org/errata/eid4720. *Прим. перев.*

8.3. Метод CONNECT

В HTTP/1.x используется псевдометод CONNECT ([RFC7231], параграф 4.3.6) для преобразования соединения HTTP в туннель к удалённому хосту. CONNECT применяется в основном HTTP для организации сессии TLS с исходным сервером при взаимодействии с ресурсами https.

В HTTP/2 метод CONNECT служит для организации туннеля с удаленным хостом через один поток HTTP/2 с похожими целями. Отображение поля заголовка HTTP работает в соответствии с описанием параграфа «8.1.2.3. Поля псевдозаголовка запроса» с некоторыми отличиями, указанными ниже.

- В поле псевдозаголовка :method устанавливается значение CONNECT.
- Поля псевдозаголовка :scheme и :path **должны** быть опущены.
- Поле псевдозаголовка :authority указывает хост и порт для соединения (эквивалент authority-form в request-target запросов CONNECT, см. параграф 5.3 в [RFC7230]).

Запрос CONNECT, не соответствующий этим ограничениям, считается некорректно сформированным (параграф 8.1.2.6).

Прокси, поддерживающие метод CONNECT, организуют соединение TCP [TCP] с сервером, указанным полем псевдозаголовка :authority. После организации соединения прокси передаёт клиенту кадр HEADERS с кодом состояния 2xx, как определено в параграфе 4.3.6 [RFC7231].

После начальных кадров HEADERS, переданных каждым партнёром, все последующие кадры DATA соответствуют данным, отправленным в соединение TCP. Данные из всех кадров DATA, отправленных клиентом, передаются посредником серверу TCP, а данные от сервера TCP собираются посредником в кадры DATA. Типы кадров, отличающиеся от DATA и кадров управления потоком (RST_STREAM, WINDOW_UPDATE и PRIORITY) **недопустимо** передавать в подключенный поток и при получении они **должны** считаться ошибками потока (параграф 5.4.2).

Соединение TCP может быть закрыто любым из партнёров. Флаг END_STREAM в кадре DATA считается эквивалентом бита TCP FIN. Предполагается, что клиент передаёт кадр DATA с флагом END_STREAM после приёма кадра, содержащего флаг END_STREAM. Прокси, получивший кадр DATA с флагом END_STREAM, передаёт присоединённые данные с флагом FIN в последнем сегменте TCP. Прокси, получивший сегмент TCP с флагом FIN, передаёт кадр DATA с флагом END_STREAM. Отметим, что заключительный сегмент TCP или кадр DATA может быть пустым.

Ошибки соединения TCP указываются с помощью RST_STREAM. Прокси считает любую ошибку в соединении TCP, где принят сегмент с установленным битом RST, ошибкой потока (параграф 5.4.2) типа CONNECT_ERROR. Поэтому прокси **должен** передавать сегмент TCP с флагом RST при обнаружении ошибки потока или соединения HTTP/2.

9. Дополнительные требования HTTP

В этом параграфе описаны атрибуты протокола HTTP, повышающие уровень взаимодействия, снижающие раскрытие известных уязвимостей защиты или снижающие возможность отклонений в реализации.

9.1. Управление соединением

Соединения HTTP/2 являются постоянными. Для повышения производительности предполагается, что клиенты не будут закрывать соединения, пока не решат, что сервер больше не нужен (например, уход с конкретной web-страницы) или сервер сам не закроет соединение.

Клиентам **не следует** открывать более одного соединения HTTP/2 для данной пары хоста и номера порта, где хост выводится из URI, выбранной альтернативной службы [ALT-SVC] или настроенного прокси.

Клиент может создавать дополнительные соединения на замену, вместо соединений, у которых заканчивается пространство доступных идентификаторов потоков (параграф 5.1.1), для обновления ключевого материала соединения TLS или взамен соединения, где возникли ошибки (параграф 5.4.1).

Клиент **может** открыть несколько соединений с одним адресом IP и портом TCP, используя разные значения SNI[1] [TLS-EXT] для получения разных сертификатов TLS, но **следует** избегать создания нескольких соединений с одинаковой конфигурацией.

Серверам рекомендуется поддерживать открытые соединения как можно дольше, но разрешается незамедлительно разрывать бездействующие соединения при необходимости. Когда любая из сторон решит закрыть транспортное соединение TCP, этой стороне **следует** сначала передать кадр GOAWAY (параграф 6.8) , чтобы обе точки могли надёжно определить, были ли обработаны отправленные ранее кадры и завершить все оставшиеся задачи.

9.1.1. Повторное использование соединений

Соединения, организованные с исходным сервером напрямую или через туннель, созданный с помощью метода CONNECT (параграф 8.3), **можно** использовать повторно для запросов с множеством разных компонент URI authority. Повторное соединение может применяться, пока исходный сервер остаётся полномочным (параграф 10.1). Для соединений TCP без TLS это зависит от наличия хоста с именем, преобразующимся в такой же адрес IP.

Для ресурсов https повторное использование соединений зависит также от наличия сертификата, который действителен для хоста из URI. Представленный сервером сертификат **должен** проходить все проверки, которые клиент выполняет при организации нового соединения TLS с хостом из URI.

Исходный сервер может представить сертификат с множеством атрибутов или имён subjectAltName с шаблонами, один из которых действует для полномочий (authority) из URI. Например, сертификат с subjectAltName = *.example.com может разрешить использование того же соединения для запроса URI, начинающегося с https://a.example.com/ или https://b.example.com/.

[1]Server Name Indication - указание имени сервера.

В некоторых случаях повторное использование соединения для разных источников может приводить к передаче запроса неподходящему исходному серверу. Например, промежуточное устройство может разорвать соединение TLS, если это устройство применяет расширение TLS SNI [TLS-EXT] для выбора исходного сервера. Это означает, что клиент может отправить конфиденциальную информацию серверу, который не предполагался целью запроса, хотя этот сервер и является полномочным.

Сервер, не желающий разрешать клиентам повторные соединения, может указать, что он не является полномочным для запроса, путём отправки кода 421 (Misdirected Request) в ответе на запрос (см. параграф 9.1.2).

Клиент, настроенный на использование прокси через HTTP/2, направляет запросы этому посреднику через одно соединение, т. е. все запросы, передаваемые через прокси, используют одно соединение с посредником.

9.1.2. Код состояния 421 (ошибочно направленный запрос)

Код 421 (Misdirected Request - ошибочно направленный запрос) указывает, что запрос был направлен на сервер, не способный создать отклик. Этот код может быть передан сервером, не настроенным на передачу откликов для комбинации scheme и authority, включённых в URI запроса[1].

Клиент, получивший код 421 от сервера **может** повторить запрос (в зависимости от того, является ли метод запроса идемпотентным) через другое соединение. Это возможно при повторном использовании соединения (параграф 9.1.1) или при выборе другого сервиса [ALT-SVC].

Этот код состояния **недопустимо** генерировать прокси.

9.2. Использование возможностей TLS

Реализации HTTP/2 **должны** применять TLS версии 1.2 [TLS12] или выше при работе HTTP/2 через TLS. **Следует** выполнять общие рекомендации по использованию TLS [TLSBCP] с дополнительными ограничениями, присущими HTTP/2.

Реализация TLS **должна** поддерживать расширение SNI [TLS-EXT] для TLS. Клиенты HTTP/2 **должны** указывать имя целевого домена при согласовании TLS.

От систем HTTP/2, согласующих TLS 1.3 и выше, требуется лишь поддержка использования расширения SNI, для систем TLS 1.2 имеются дополнительные требования, приведённые в следующих параграфах. Реализациям настоятельно рекомендуется применять по умолчанию соответствующие параметры, но в конечном итоге система сама отвечает за соответствие требованиям.

9.2.1. Возможности TLS 1.2

В этом параграфе описаны ограничения набора функций TLS 1.2, которые могут быть применены в HTTP/2. Из-за ограничений развёртываемых систем отказы при согласовании TLS могут быть не возможны, если не выполняются приведённые здесь ограничения. Конечная точка **может** незамедлительно разорвать соединение HTTP/2, в котором не выполняются эти требования TLS, с возвратом ошибки соединения (параграф 5.4.1) типа INADEQUATE_SECURITY.

В системах HTTP/2 на основе TLS 1.2 сжатие **должно** быть отключено. Компрессия TLS может приводить к раскрытию информации, которая в иных условиях не была бы раскрыта [RFC3749]. Базовая компрессия не нужна, поскольку HTTP/2 обеспечивает функции сжатия, которые лучше осведомлены о контексте и поэтому больше подходят для использования в целях повышения производительности, безопасности или улучшения иных свойств.

Системы HTTP/2 на основе TLS 1.2 **должны** запрещать повторное согласование. Конечная точка **должна** трактовать повторное согласование TLS как ошибку соединения (параграф 5.4.1) типа PROTOCOL_ERROR. Отметим, что запрет повторного согласования может приводить к возникновению долгоживущих, не используемых соединений по причине ограничения числа сообщений, разрешённых для шифрования базовому шифронабору.

Конечная точка **может** применять повторное согласование с целью защиты конфиденциальности для свидетельств клиента, предложенных при согласовании, но повторное согласование **должно** выполняться до передачи префикса соединения. Серверу **следует** запрашивать сертификат клиента, если он видит повторное согласование сразу после организации соединения.

Это эффективно предотвращает использование повторного согласования в ответ на запрос конкретного защищённого ресурса. Будущие спецификации могут предложить решение для поддержки таких ситуаций. В качестве альтернативы сервер может использовать ошибку (параграф 5.4) типа HTTP_1_1_REQUIRED для запроса у клиента использования протокола, поддерживающего повторное согласование.

Реализации **должны** поддерживать обмен эфемерными ключами размером не менее 2048 битов для шифров, использующих эфемерное конечное поле DHE[2] [TLS12], и 224 битов для шифров, использующих эфемерные эллиптические кривые ECDHE[3] [RFC4492]. Клиенты **должны** воспринимать DHE размером до 4096 битов. Конечные точки **могут** считать согласование ключей, размер которых меньше нижнего предела, как ошибку соединения (параграф 5.4.1) типа INADEQUATE_SECURITY.

9.2.2. Шифры TLS 1.2

В системах HTTP/2 на базе TLS 1.2 **не следует** использовать какие-либо шифры из числа включённых в «чёрный список» (Приложение A).

Конечные точки **могут** возвращать ошибку соединения (параграф 5.4.1) типа INADEQUATE_SECURITY при попытке использования шифров из «чёрного списка». Системы, выбравшие использование таких шифров, рискуют столкнуться с сообщениями об ошибках, если они не знают, что партнёры воспримут такие шифры.

[1]Этот абзац предложено удалить в последующей версии документа. См. https://www.rfc-editor.org/errata/eid4666. *Прим. перев.*
[2]Diffie-Hellman.
[3]Ephemeral elliptic curve Diffie-Hellman.

Реализациям **недопустимо** генерировать ошибки в ответ на согласование шифров, не включённых в «чёрный список». Поэтому когда клиенты предлагают шифр, отсутствующий в «чёрном списке», они должны быть готовы использовать этот шифр с HTTP/2.

«Серный список» включает шифры, которые в TLS 1.2 являются обязательными, а это означает, что системы TLS 1.2 могут иметь не пересекающиеся наборы разрешённых шифров. Для предотвращения этой проблемы, вызывающей отказы при согласовании TLS, системы HTTP/2, использующие TLS 1.2, **должны** поддерживать шифр TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 [TLS-ECDHE] с эллиптической кривой P-256 [FIPS186].

Отметим, что клиенты могут анонсировать поддержку шифров из «чёрного списка» для возможности соединений с серверами, которые не поддерживают HTTP/2. Это позволяет серверу выбрать HTTP/1.1 с шифром из «чёрного списка» HTTP/2. Однако это может приводить к согласованию с HTTP/2 с шифром из «чёрного списка», если прикладной протокол и шифр выбираются независимо.

10. Вопросы безопасности

10.1. Полномочия сервера

HTTP/2 опирается на определение полномочий в HTTP/1.1 для решения вопроса о полномочиях сервера предоставлять данный отклик (см. параграф 9.1 в [RFC7230]). Это зависит от локальной трансляции имени в схеме URI «http» и полномочий аутентифицированного сервера для схемы «https» (см. раздел 3 в [RFC2818]).

10.2. Кросспротокольные атаки

В кросспротокольной атаке злоумышленник вынуждает клиента инициировать транзакцию с неким протоколом для сервера, который понимает другой протокол. Атакующий может оказаться способен организовать транзакцию, которая представляется действительной для второго протокола. В комбинации с возможностями web-контекста это может применяться для воздействия на слабо защищённые серверы в частных сетях.

Завершение согласования TLS с идентификатором ALPN для HTTP/2 может считаться достаточной защитой от кросс-протокольных атак. ALPN обеспечивает позитивную индикацию желания сервера обрабатывать HTTP/2, что предотвращает атаки на другие протоколы на базе TLS.

Использование TLS осложняет атакующему контроль над данными, которые могут служит для кросс-протокольных атак при использовании открытых (нешифрованных) протоколов.

Открытая (без шифрования) версия HTTP/2 имеет минимальную защиту от кросс-протокольных атак. Префикс соединения (параграф 3.5) содержит строку, предназначенную для того, чтобы запутать серверы HTTP/1.1, но никакой специальной защиты для других протоколов не предусмотрено. Сервер, готовый игнорировать части запроса HTTP/1.1, содержащие поле заголовка Upgrade в дополнение к клиентскому префиксу соединения, может подвергаться кросспротокольным атакам.

10.3. Атаки с промежуточной инкапсуляцией

Кодирование поля заголовка HTTP/2 позволяет указывать имена, которые не являются разрешёнными именами полей в синтаксисе сообщений Internet, используемом HTTP/1.1. Запросы и отклики с недопустимыми именами полей **должны** считаться некорректно сформированными (параграф 8.1.2.6). Поэтому посредник не может транслировать запрос или отклик HTTP/2, содержащий недопустимое имя поля, в сообщение HTTP/1.1.

Аналогично, HTTP/2 разрешает значения полей, которые не действительны. Хотя большинство значений, которые могут быть закодированы, не меняют синтаксический анализ поля заголовка, символы возврата каретки (CR, ASCII 0xd), перевода строки (LF, ASCII 0xa) и nul-символ (NUL, ASCII 0x0) могут использоваться атакующим при дословной трансляции. Любой запрос или отклик с символами, которые не разрешены в поле заголовка, **должен** считаться некорректно сформированным (параграф 8.1.2.6). Разрешённые символы определены правилом field-content ABNF а параграфе 3.2 [RFC7230].

10.4. Кэшируемость вытолкнутых откликов

Выталкиваемые отклики не имеют явного запроса от клиента, запрос предоставляется сервером в кадре PUSH_PROMISE.

Кэширование вытолкнутых откликов возможно на основе рекомендаций, представленных исходным сервером в поле заголовка Cache-Control. Однако при этом могут возникать проблемы, если на одном серверном хосте имеется несколько арендаторов. Например, сервер может предлагать каждому из множества пользователей небольшую часть своего пространства URI.

Когда множество арендаторов совместно использует пространство на одном сервере, этот сервер **должен** гарантировать, что арендаторы не смогут выталкивать представления ресурсов, для которых они не имеют полномочий. Невыполнение этого требования позволит арендаторам выдавать представления, которые обслуживаться за пределами кэша, переопределяя фактическое представление, которое обеспечил полномочный сервер.

Вытолкнутые отклики, для которых исходный сервер не является полномочным (параграф 10.1) **недопустимо** использовать или кэшировать.

10.5. Отказ в обслуживании

Соединению HTTP/2 для работы может потребоваться больше ресурсов, нежели соединению HTTP/1.1. Использование сжатия заголовков и управления потоком данных зависит от выделения ресурсов для сохранения большего числа состояний. Настройки этих функций гарантируют строгое ограничение выделяемой памяти.

Число кадров PUSH_PROMISE таким способом не ограничивается. Клиентам, воспринимающим выталкивание от сервера, **следует** ограничить число потоков, которым разрешено находиться в состоянии reserved (remote).

Избыточное число выталкиваемых сервером потоков может считаться ошибкой потока (параграф 5.4.2) типа ENHANCE_YOUR_CALM.

Возможности обработки не могут ограничиваться так же эффективно, как возможности хранения состояний.

Кадры SETTINGS могут использоваться для принуждения партнёра затрачивать на обработку дополнительное время. Это можно сделать путём бессмысленного изменения параметров SETTINGS, установки множества неопределённых параметров или многократного изменения одного и того же параметра в кадре. Кадры WINDOW_UPDATE или PRIORITY могут быть использованы для принуждения к ненужному расходу ресурсов.

С помощью большого числа мелких или пустых кадров можно вынудить партнёра затрачивать значительное время на обработку заголовков. Однако следует отметить, что использование таких кадров может быть легитимным, например, для передачи пустых кадров DATA или CONTINUATION в конце потока.

Сжатие заголовков также позволяет потерю ресурсов на обработку. Возможные варианты злоупотреблений сжатием описаны в разделе 7 [COMPRESSION].

Ограничения в параметрах SETTINGS не могут быть снижены мгновенно и это создаёт для конечных точек риск, связанный с превышением партнёром вновь установленных пределов. В частности, сразу после организации соединения установленные сервером ограничения могут быть не известны клиентам и могут быть превышены без явного нарушения протокола.

Все эти возможности - изменение SETTINGS, мелкие кадры, сжатие заголовков - могут применяться легитимно. Они создают угрозы лишь при ненужном или чрезмерном использовании.

Конечная точка, которая не отслеживает такое поведение, подвергает себя риску атак, нацеленных на отказ в обслуживании. Реализациям **следует** контролировать использование этих функций и устанавливать для него ограничения. Конечная точка **может** считать сомнительные действия ошибкой соединения (параграф 5.4.1) типа ENHANCE_YOUR_CALM.

10.5.1. Ограничения размера блока заголовков

Большой блок заголовков (параграф 4.3) может вынуждать реализацию представлять большое число состояний. Важные для маршрутизации поля заголовка могут оказаться в конце блока, что будет препятствовать организации потока полей заголовков к конечному адресату. Упорядочение и другие причины, такие как обеспечение корректности кэширования, означают, что конечной точке потребуется буферизовать весь блок заголовков. Поскольку размер блока заголовков жёстко не ограничен, некоторые конечные точки будут вынуждены выделять большой блок доступной памяти для полей заголовков.

Конечная точка может использовать SETTINGS_MAX_HEADER_LIST_SIZE для информирования партнёров об ограничениях на размер блока заголовков. Этот параметр носит рекомендательный характер, поэтому конечная точка **может** передавать блоки заголовков, выходящие за этот предел, что ведёт к риску трактовки запроса или отклика, как некорректно сформированного. Этот параметр относится к соединению, поэтому любой запрос или отклик может столкнуться с этапом пересылки (hop), где предел не известен и имеет более жёсткое ограничение. Промежуточный узел может попытаться предотвратить эту проблему, передавая значения, представленные другими партнёрами, но не обязан делать этого.

Сервер, который получает блок заголовков с размером больше, чем он готов обрабатывать, может передать отклик HTTP 431 (Request Header Fields Too Large) [RFC6585]. Клиент может отбрасывать отклики, которые он не способен обработать. Блок заголовков **должен** обрабатываться для поддержки согласованного состояния соединения, пока соединение не закрыто.

10.5.2. Проблемы CONNECT

Метод CONNECT позволяет создать непропорциональную нагрузку на прокси, поскольку создание потоков является «недорогим» в сравнении с созданием и поддержкой соединений TCP. Прокси-сервер может также поддерживать некоторые ресурсы соединения TCP после закрытия потока, который передаёт запрос CONNECT, поскольку исходящее соединение TCP остаётся в состоянии TIME_WAIT. В результате прокси не может ограничивать ресурсы, потребляемые запросами CONNECT, на основании SETTINGS_MAX_CONCURRENT_STREAMS.

10.6. Использование сжатия

Компрессия может позволить атакующему восстановить секретные данные, которые были сжаты в одном контексте с данными, контролируемыми злоумышленником. HTTP/2 позволяет сжимать поля заголовков (параграф 4.3) и рассмотренные ниже проблемы присущи также к использованию сжатого кодирования содержимого HTTP (см. параграф 3.1.2.1 в [RFC7231]).

Имеются очевидные атаки на компрессию, использующие характеристики web (например, [BREACH]). Атакующий инициирует множество запросов, содержащих разный открытый текст, наблюдая размер полученного шифрованного текста, который будет короче при корректном предположении о секрете.

Реализациям, взаимодействующим по защищённому каналу, **недопустимо** совместно сжимать содержимое конфиденциальных и контролируемых атакующим данных, если для каждого источника данных не применяется отдельный словарь. **Недопустимо** применять сжатие, если источник данных не может быть надёжно определён. Базовое сжатие потока, например, обеспечиваемое TLS, **недопустимо** применять с HTTP/2 (см. параграф 9.2).

Дополнительное рассмотрение вопросов, связанных со сжатием полей заголовков, приведено в [COMPRESSION].

10.7. Использование заполнения

Заполнение в HTTP/2 не предназначено для замены заполнений «общего назначения», таких как в TLS [TLS12]. Избыточное заполнение может даже быть контрпродуктивным. Корректность использования заполнения может зависеть от информации о дополняемых данных.

Для смягчения атак, основанных на сжатии, запрет или ограничение компрессии могут оказаться предпочтительней заполнения.

Заполнение может служить для сокрытия точного размера содержимого кадра и применяться для смягчения некоторых типичных для HTTP атак, например, атак, где сжатое содержимое включает контролируемые злоумышленником открытые данные вместе с секретной информацией (например, [BREACH]).

Использование заполнения может приводить к незаметному сразу снижению уровня защиты. В лучшем случае, заполнение лишь осложняет атакующему получение информации о размере, увеличивая число кадров, которые злоумышленник должен наблюдать. Некорректно реализованные схемы заполнения легко преодолеваемы. В частности, случайное заполнение с предсказуемым распределением обеспечивает очень слабую защиту. Аналогично, дополнение полей данных до фиксированного размера предоставляет информацию об этом размере, если атакующий контролирует открытые данные.

Промежуточным узлам **следует** сохранять заполнение в кадрах DATA, но они **могут** отбрасывать заполнение кадров HEADERS и PUSH_PROMISE. Достаточной причиной изменения заполнения кадров на промежуточных узлах является улучшение защиты, обеспечиваемой заполнением.

10.8. Вопросы приватности

Некоторые характеристики HTTP/2 предоставляют наблюдателю возможность сопоставить действия одного клиента или сервера в течение временного интервала. Это включает значения параметров, способ управления окнами контроля потока данных, способы указания приоритета для потоков, время реакции на «раздражители» и обработку любых свойств, управляемых параметрами.

Наблюдая различия в поведении, злоумышленник может создать «отпечатки» для каждого клиента, как указано в параграфе 1.8 [HTML5].

Предпочтение HTTP/2 использовать единственное соединение TCP позволяет сопоставить активность пользователей на сайте. Повторное использование соединений для разных источников позволяет отслеживать эти источники.

Поскольку кадры PING и SETTINGS запрашивают незамедлительный отклик, они могут использоваться конечной точкой при измерении задержки для данного партнёра. В некоторых случаях это может оказывать влияние на приватность.

11. Согласование с IANA

Добавляется строка идентификации протокола HTTP/2 в реестр «Application-Layer Protocol Negotiation (ALPN) Protocol Ids», организованный [TLS-ALPN].

Этот документ организует реестры для типов кадров, параметров и кодов ошибок, представленные ниже.

Этот документ регистрирует поле заголовка HTTP2-Settings для использования в HTTP, а также код состояния 421 (Misdirected Request).

Документ регистрирует метод PRI для использования в HTTP с целью предотвращения конфликтов с префиксами соединений (параграф 3.5).

11.1. Регистрация идентификационных строк HTTP/2

Этот документ определяет две регистрации для идентификации протокола HTTP/2 (см. параграф 3.3) в реестре «Application-Layer Protocol Negotiation (ALPN) Protocol Ids», организованном [TLS-ALPN].

Строка «h2» указывает использование HTTP/2 на базе TLS.

```
Protocol:  HTTP/2 over TLS
Identification Sequence:  0x68 0x32 ("h2")
Specification:  This document
```

Строка «h2c» указывает использование HTTP/2 на базе TCP без шифрования.

```
Protocol:  HTTP/2 over TCP
Identification Sequence:  0x68 0x32 0x63 ("h2c")
Specification:  This document
```

11.2. Реестр типов кадров

Этот документ организует реестр типов кадров HTTP/2. Реестр «HTTP/2 Frame Type» содержит 8-битовые значения, выделяемые по процедуре «IETF Review» или «IESG Approval» [RFC5226] для диапазона значений 0x00 — 0xef. Диапазон 0xf0 - 0xff зарезервирован для экспериментов (Experimental Use).

Для новых записей в реестре требуется предоставить перечисленные ниже сведения.

```
Frame Type:  Имя или метка для типа кадров.
Code:  8-битовый код, выделенный для типа кадра.
Specification:  Ссылка на спецификацию, включающую описание схемы кадра, его семантику и используемые
флаги, которые присутствуют в зависимости от значения типа флага.
```

Типы, регистрируемые данным документом, приведены в таблице.

Тип кадра	*Код*	*Параграф*
DATA	0x0	6.1
HEADERS	0x1	6.2
PRIORITY	0x2	6.3
RST_STREAM	0x3	6.4
SETTINGS	0x4	6.5
PUSH_PROMISE	0x5	6.6
PING	0x6	6.7

GOAWAY	0x7	6.8
WINDOW_UPDATE	0x8	6.9
CONTINUATION	0x9	6.10

11.3. Реестр параметров

Этот документ организует реестр параметров HTTP/2. Реестр «HTTP/2 Settings» содержит 16-битовые значения, выделяемые по процедуре «Expert Review» [RFC5226] из диапазона 0x0000 - 0xefff. Значения 0xf000 - 0xffff зарезервированы для экспериментов (Experimental Use).

При новых регистрациях рекомендуется предоставлять перечисленную ниже информацию.

```
Name:  Символьное имя параметра (необязательно).
Code:  16-битовый код параметра.
Initial Value:  Начальное значение параметра.
Specification:  Необязательная ссылка на спецификацию, описывающую параметр.
```

Параметры, регистрируемые данным документом, приведены в таблице.

Имя	*Код*	*Начальное значение*	*Спецификация*
HEADER_TABLE_SIZE	0x1	4096	Параграф 6.5.2
ENABLE_PUSH	0x2	1	Параграф 6.5.2
MAX_CONCURRENT_STREAMS	0x3	(бесконечность)	Параграф 6.5.2
INITIAL_WINDOW_SIZE	0x4	65535	Параграф 6.5.2
MAX_FRAME_SIZE	0x5	16384	Параграф 6.5.2
MAX_HEADER_LIST_SIZE	0x6	(бесконечность)	Параграф 6.5.2

11.4. Реестр кодов ошибок

Этот документ организует реестр кодов ошибок для HTTP/2. Реестр «HTTP/2 Error Code» содержит 32-битовые значения кодов и пополняется на основе процедуры «Expert Review» [RFC5226].

Регистрация кода ошибки должна включать описание. Рецензентам настоятельно рекомендуется проверять новые регистрации на предмет дублирования уже имеющихся кодов. Использование существующих регистраций следует поощрять, но такое использование не является обязательным.

В новых регистрациях рекомендуется указывать приведённую ниже информацию.

```
Name:  Имя для кода ошибки. Задание имён для ошибок не является обязательным.
Code:  32-битовый код ошибки.
Description:  Краткое описание семантики кода (более подробное, если спецификации нет).
Specification:  Необязательная ссылка на спецификацию, определяющую код ошибки.
```

Коды, регистрируемые данным документом, приведены в таблице.

Имя	*Код*	*Описание*	*Спецификация*
NO_ERROR	0x0	Аккуратное завершение	Раздел 7
PROTOCOL_ERROR	0x1	Обнаружена протокольная ошибка	Раздел 7
INTERNAL_ERROR	0x2	Отказ реализации	Раздел 7
FLOW_CONTROL_ERROR	0x3	Превышение предела, заданного для управления потоком данных	Раздел 7
SETTINGS_TIMEOUT	0x4	Установки не подтверждены	Раздел 7
STREAM_CLOSED	0x5	Получен кадр для закрытого потока	Раздел 7
FRAME_SIZE_ERROR	0x6	Некорректный размер кадра	Раздел 7
REFUSED_STREAM	0x7	Поток не обработан	Раздел 7
CANCEL	0x8	Поток прерван	Раздел 7
COMPRESSION_ERROR	0x9	Состояние компрессии не обновлено	Раздел 7
CONNECT_ERROR	0xa	Ошибка соединения TCP для метода CONNECT	Раздел 7
ENHANCE_YOUR_CALM	0xb	Выход за пределы возможностей обработки	Раздел 7
INADEQUATE_SECURITY	0xc	Согласованные параметры TLS не приемлемы	Раздел 7
HTTP_1_1_REQUIRED	0xd	Требуется использовать HTTP/1.1 для запроса	Раздел 7

11.5. Регистрация поля заголовка HTTP2-Settings

Этот документ регистрирует поле заголовка HTTP2-Settings в реестре Permanent Message Header Field Names [BCP90].

```
Header field name:  HTTP2-Settings
Applicable protocol:  http
Status:  standard
Author/Change controller:  IETF
Specification document(s):  Section 3.2.1 of this document
Related information:  This header field is only used by an HTTP/2
      client for Upgrade-based negotiation.
```

11.6. Регистрация метода PRI

Этот документ регистрирует метод PRI в реестре HTTP Method Registry, параграф 8.1 [RFC7231].

```
Method Name:  PRI
Safe:  Yes
Idempotent:  Yes
Specification document(s):  Section 3.5 of this document
Related information:  This method is never used by an actual client.
      This method will appear to be used when an HTTP/1.1 server or
      intermediary attempts to parse an HTTP/2 connection preface.
```

11.7. Код состояния 421 (Misdirected Request)

Этот документ регистрирует код состояния HTTP 421 (Misdirected Request) в реестре HTTP Status Codes, параграф 8.2 [RFC7231].

```
Status Code:  421
Short Description:  Misdirected Request
Specification:  Section 9.1.2 of this document
```

11.8. Обновление маркера h2c

Этот документ регистрирует маркер обновления h2c в реестре HTTP Upgrade Tokens, параграф 8.6 [RFC7230].

```
Value:  h2c
Description:  Hypertext Transfer Protocol version 2 (HTTP/2)
Expected Version Tokens:  None
Reference:  Section 3.2 of this document
```

12. Литература

12.1. Нормативные документы

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

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

[FIPS186] NIST, "Digital Signature Standard (DSS)", FIPS PUB 186-4, July 2013, <http://dx.doi.org/10.6028/NIST.FIPS.186-4>.

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

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <http://www.rfc-editor.org/info/rfc2818>.

[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, <http://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <http://www.rfc-editor.org/info/rfc4648>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/ RFC5234, January 2008, <http://www.rfc-editor.org/info/rfc5234>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <http://www.rfc-editor.org/info/rfc7231>.

[RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <http://www.rfc-editor.org/info/rfc7232>.

[RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, DOI 10.17487/RFC7233, June 2014, <http://www.rfc-editor.org/info/rfc7233>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <http://www.rfc-editor.org/info/rfc7234>.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <http://www.rfc-editor.org/info/rfc7235>.

[TCP] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://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, <http://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, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc5246>.

12.2. Дополнительная литература

[ALT-SVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", Work in Progress[1], draft-ietf-httpbis-alt-svc-06, February 2015.

[BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004, <http://www.rfc-editor.org/info/bcp90>.

[1]Работа опубликована в RFC 7838. *Прим. перев.*

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

[HTML5] Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Doyle Navara, E., O'Connor, E., and S. Pfeiffer, "HTML5", W3C Recommendation REC-html5-20141028, October 2014, <http://www.w3.org/TR/2014/REC-html5-20141028/>.

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

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <http://www.rfc-editor.org/info/rfc4492>.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <http://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, <http://www.rfc-editor.org/info/rfc7323>.

[TALKING] Huang, L., Chen, E., Barth, A., Rescorla, E., and C. Jackson, "Talking to Yourself for Fun and Profit", 2011, <http://w2spconf.com/2011/papers/websocket.pdf>.

[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, <http://www.rfc-editor.org/info/rfc7525>.

Приложение A. «Серный список» шифров TLS 1.2

Реализация HTTP/2 **может** считать согласование любого из перечисленных ниже шифров с 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_RSA_WITH_AES_256_CCM;
- TLS_RSA_WITH_AES_128_CCM_8;
- TLS_RSA_WITH_AES_256_CCM_8;
- TLS_PSK_WITH_AES_128_CCM;
- TLS_PSK_WITH_AES_256_CCM;
- TLS_PSK_WITH_AES_128_CCM_8;TLS_PSK_WITH_AES_256_CCM_8

Примечание. Этот список создан из множества зарегистрированных шифров TLS на момент подготовки этого документа. Список включает шифры, которые не предлагают обмена эфемерными ключами, а также шифры, основанные на TLS null, потоковом и блочном шифровании (как определено в параграфе 6.2.3 [TLS12]). Могут существовать дополнительные шифры с такими свойствами, они не запрещаются явно.

Благодарности

Этот документ включает существенный вклад перечисленных ниже людей.

- Adam Langley, Wan-Teh Chang, Jim Morrison, Mark Nottingham, Alyssa Wilk, Costin Manolache, William Chan, Vitaliy Lvin, Joe Chan, Adam Barth, Ryan Hamilton, Gavin Peters, Kent Alstad, Kevin Lindsay, Paul Amer, Fan Yang, и Jonathan Leighton (участники SPDY).
- Gabriel Montenegro и Willy Tarreau (механизм Upgrade).
- William Chan, Salvatore Loreto, Osama Mazahir, Gabriel Montenegro, Jitu Padhye, Roberto Peon и Rob Trace (управление потоком данных).
- Mike Bishop (расширяемость).
- Mark Nottingham, Julian Reschke, James Snell, Jeff Pinner, Mike Bishop и Herve Ruellan (существенные редакторские правки).
- Kari Hurtta, Tatsuhiro Tsujikawa, Greg Wilkins, Poul-Henning Kamp и Jonathan Thackray.
- Alexey Melnikov, который был редактором этого документа в 2013 году.

Существенная часть вклада Martin была поддержана компанией Microsoft, пока он там работал.

Японское сообщество HTTP/2 внесло важный вклад, включая множество реализаций, а также технических и редакторских правок.

Адреса авторов

Mike Belshe
BitGo
EMail: mike@belshe.com

Roberto Peon
Google, Inc
EMail: fenix@google.com

Martin Thomson (редактор)
Mozilla
331 E Evelyn Street
Mountain View, CA 94041
United States
EMail: martin.thomson@gmail.com

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru