7章
HTTP/2、HTTP/3のシンタックス：プロトコルの再定義

本章では、HTTP/2やHTTP/3、その前後に策定された新しい世代のさまざまなプロトコルについて紹介します。

それ以外に紹介するWebSocketやWebRTCも、HTTP/1.1では実現できなかった、高速な双方向通信であったり、P2Pの接続、動画の特性に合わせた通信などを実現するために生み出されました。ウェブの可能性をさらに広げるための進化と言えます。

7.1 HTTP/2、HTTP/3で変わらないこと

HTTP/1.1までの、メールやニュースグループを参考に作られたテキストで表現されたプロトコルに比べると、HTTP/2、HTTP/3は大きな飛躍を遂げました。ウェブサイトで使われる画像やCSS、JavaScriptなどのファイルが数もサイズも増えていく中で、より高速に転送できる仕組みがもとめられたからです。単純な機能の漸進的な修正ではなく、すべての要素が一度分解されて、大量のアセットが必要な時代の要件に合致するようにゼロから組み上げられました。

通信のオーバーヘッドを小さくするためにバイナリプロトコルとなり、HTTPの特性に合わせたヘッダー圧縮などが組み込まれました。また、その下のTCPのレイヤーで行っていることをエミュレーションしたり、高度な制御機構を取り込みました。

一方で、今まで紹介してきた、メソッド、ヘッダー、ステータス、ボディという「HTTPの提供する4つの基本要素」は変わらないため、HTTP/2やHTTP/3のプロトコルを直接読み書きするクライアントコードやサーバーから見ると大きな変更ですが、ブラウザのフロントエンドや、ウェブアプリケーションから見ると、HTTP/1.1と差はありません。

たとえば`Cache-Control`を使ったキャッシュの仕組みは、同じ時期に一緒に更新されたHTTP/1.1の集大成となるRFC（**RFC7230**～**RFC7235**）をそのまま参照しています。そのため、ここまで本書で説明してきた内容で無駄になるものはほぼありません。これらのRFCは、それまでのHTTP1.1以降に追加されたものを集約しただけでなく、HTTP/2を見越して、セマンティクス部分を共有するために作ら

れました。また、HTTP/3も同様です。

なお、HTTP/2が出てきたからといって、HTTP/1.1が不要になるわけではなく、HTTP/3が出てきたからといって、HTTP/2が不要になるわけではありません。TCPしか使えない環境で最速を求める場合にはHTTP/2になるでしょう。

複数のマイクロサービスが連携する場合の通信は依然としてHTTP/1.1です。HTTP/3は暗号化が必須ですし、HTTP/2も多くのサーバーでTLSがないとHTTP/2が有効化しないようになっています。大規模なウェブシステムを作るとして、ロードバランサーなどのエッジまではTLSの暗号化を使った通信になりますが、その中のロードバランサーの内側はTLSを使わない通信を使います。そうなるとHTTP/1.1となります。

HTTP/2もHTTP/3も、インターネット越しの通信を高速化するために策定されているので、サービス内通信もHTTP/2などが使いたいと思われるかもしれませんが、たくさんのリソースを並列ダウンロードしないようなウェブAPIのような使い方で、なおかつKeepAliveをきちんとすれば、HTTP/1.1でもローカルエリア内での速度ペナルティは大きくありません。

7.2 HTTP/2

HTTP/1.1は長いあいだ最新のバージョンであり続けました。HTTP/1.1がRFCになったのは1999年です。その後**HTTP/2**が規格化される2015年まで実に16年もの歳月が流れています。新しい規格が次々と現れるコンピューター関連の業界にあって、異例とも言える期間です。HTTP/2は久々の大きなアップデートで、パフォーマンス向上にフォーカスしています。プロトコルをどのようなバイト列として表現して通信するかが大きく変わりました。

HTTP/2の**RFC7540**と**RFC7541**にも、バイナリ通信フォーマットの説明しかなく、たとえば`Cache-Control`を使ったキャッシュの仕組みは、同じ時期に一緒に更新されたHTTP/1.1のRFC（**RFC7230**～**RFC7235**）をそのまま参照しています。

7.2.1 SPDY

HTTP/2の歴史を語る上ではずせないのが**SPDY（スピーディー）**です。SPDYはGoogleが開発したHTTP代替のプロトコルで、これがほぼそのままHTTP/2となりました。

SPDYはGoogleのサービス内部で使われ、Google ChromeとFirefox、Internet Explorer、Safariにも実装されました。SPDYは、2010年にChromeへと実装された後もバージョンアップを重ね、その後2014年にHTTP/2へバトンを渡し、その役目を終えました。最初にSPDYが実装されたChromeも2016年5月にリリースされたChrome 51でこれを無効化し、HTTP/2に一本化されました。

GoogleがSPDYを開発したのは、これまでのHTTPが改善してきた転送速度を一段と向上させるためです。ウェブサイトの構成によって効果が大きく変わるため、導入の効果としては30%から3倍以上

まで、いろいろな数値があげられましたが[†1]、並列アクセスでのブロッキングが減るため小さなファイルをたくさん転送するほど高速化します。

Googleの強みは、大規模なトラフィックのウェブサービスとシェアの高いブラウザの両方を押さえている点にあります。これにより自社サービスと自社ブラウザ間でのみ有効化されるプロトコルであっても、大規模な実証実験が可能です。

「ウェブサイトのJavaScriptやCSS、画像などはなるべく少ないファイル数にまとめることで高速化させる」というのがHTTP/1.1時代に高速なウェブサイトを作るためのテクニックとして一般的でした。SPDYとHTTP/2以降、ファイルをまとめる効果は小さくなることが見込まれています。

圧縮されているとはいえヘッダーのサイズはゼロではありませんし、ファイルサイズが小さくなるほどパケットの隙間がムダになることも増えるので[†2]、結合した方が通信量は減ります。一方でファイルが細かく分割されている方が、変更があったときにキャッシュが生き残る確率は高くなります。どちらも一長一短で、差はあまりないかもしれません。

7.2.2 HTTP/2の改善点

HTTP/2の大きな変更点は次の通りです。HTTP/2の目的は、通信の高速化に尽きます。

- ストリーム（1.1のパイプラインに近いもの）を使ってバイナリデータを多重に送受信する仕組みに変更
- ストリーム内での優先順位設定や、サーバーサイドからデータ通信を行うサーバーサイドプッシュを実装
- ヘッダーが圧縮されるようになった

これ以外には、TLSの策定以降に向上した計算機速度の影響により短期間で解読できるようになってしまった暗号スイートを非推奨化するといった内容も含まれています。HTTPとTLSの両方が相互に関係しあってバージョンアップされています。

HTTP/1.0からHTTP/1.1にかけて、TCPソケットレベルで見ると、次のような改善が行われてきました。

†1 http://d.hatena.ne.jp/jovi0608/20120523/1337742400

†2 TCPの最大セグメントサイズはおよそ1500バイト。ジャンボフレームを使用している場合はさらに大きく、およそ9000バイト。

機能	効果
キャッシュ（max-age）	通信そのものをキャンセル
キャッシュ（ETag、Date）	変更がなければボディ送信をキャンセル
Keep-Alive	アクセスごとに接続にかかる時間（1.5TTL）を削減
圧縮	レスポンスのボディサイズの削減
チャンク	レスポンス送信開始を早める
パイプライニング	通信の多重化

1回の通信では、接続の確立リクエスト送信、受信と何往復もパケットが飛び交います。また、データサイズを通信速度で割った時間だけ通信時間がかかります。通信待ちがあれば、そのぶん通信完了までの時間は長くなります。

これまで行われてきた高速化は、この通信のあらゆる個所での高速化に寄与してきました。HTTP/2では、これまで手が付けられてこなかったヘッダー部の圧縮や、規格化されたもののいまひとつ活用されていなかったパイプライニングの代替実装が追加されました。

4章のパイプライニングの紹介で、前方互換性の問題から限定的にしか使われていないことを紹介しました。

HTTP/2はこれまでとは完全に異なったプロトコルであるため、逆に後方互換性の問題が起きにくくなっています。HTTPのテキストプロトコル内部でのバージョン切り替えではなく、TLS内に作られたプロトコル選択機能（4章で紹介したALPN）を使い、まるごと通信方式を切り替える方式になっています。HTTP/1.1とはまったく別のプロトコルとして扱われるため、パイプライニングのような問題が起きることはありません。

7.2.3 ストリームによる通信の高速化

HTTP/2の一番大きな変化は、テキストベースのプロトコルからバイナリベースへのプロトコルに変化したという点です。各データは**フレーム**と呼ばれる単位で送受信が行われます。HTTP/1.1まではひとつのリクエストがTCPのソケットを専有してしまうため、ひとつのオリジンサーバーに対して2～6本のTCP接続を行って並列化していました。

HTTP/2では1本のTCP接続の内部に、**ストリーム**という仮想のTCPソケットを作って通信を行います。ストリームはフレームに付随するフラグで簡単に作ったり閉じたりできるルールになっており、通常のTCPソケットのようなハンドシェイクは必要ありません。そのため、IDの数値[†3]とTCPの通信容量が許す限り、簡単に数万接続でも並列化できます。

†3　31ビットありますが、1ビット目は送受信方向の決定で使われるので30ビット≒約10億回分並列できます。

それぞれ、接続状態のステートマシンが定義されており、似たようなモデルになっていることがわかります。

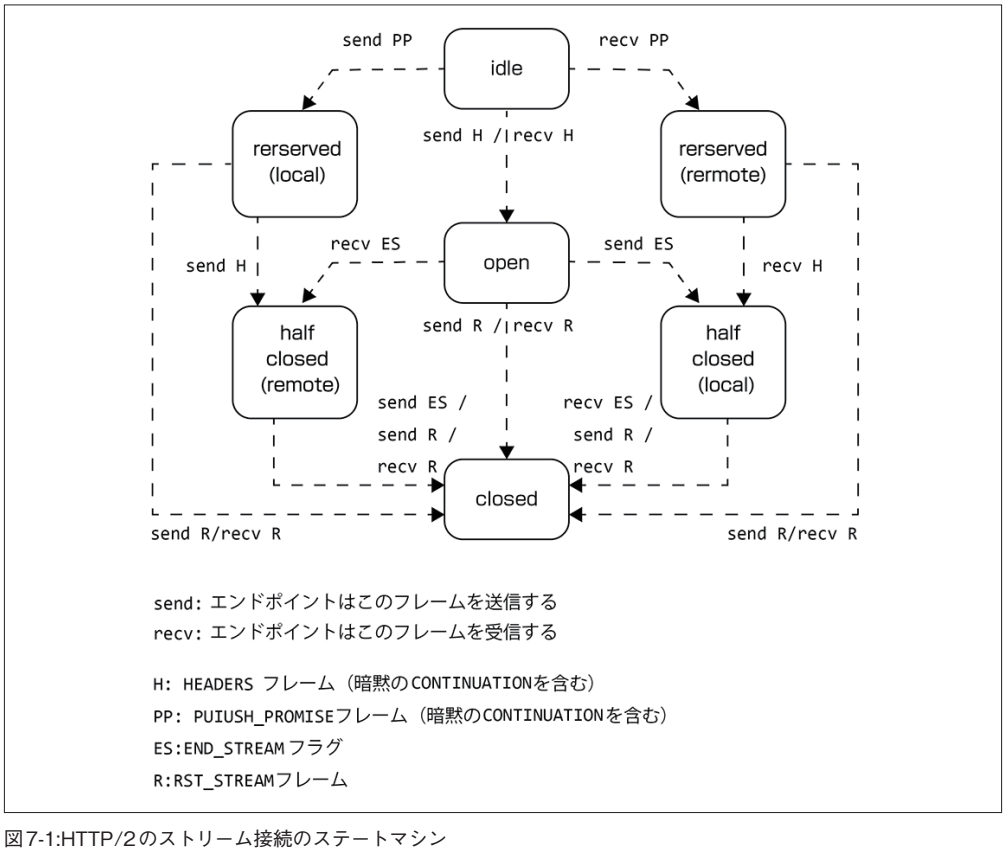


図7-1:HTTP/2のストリーム接続のステートマシン

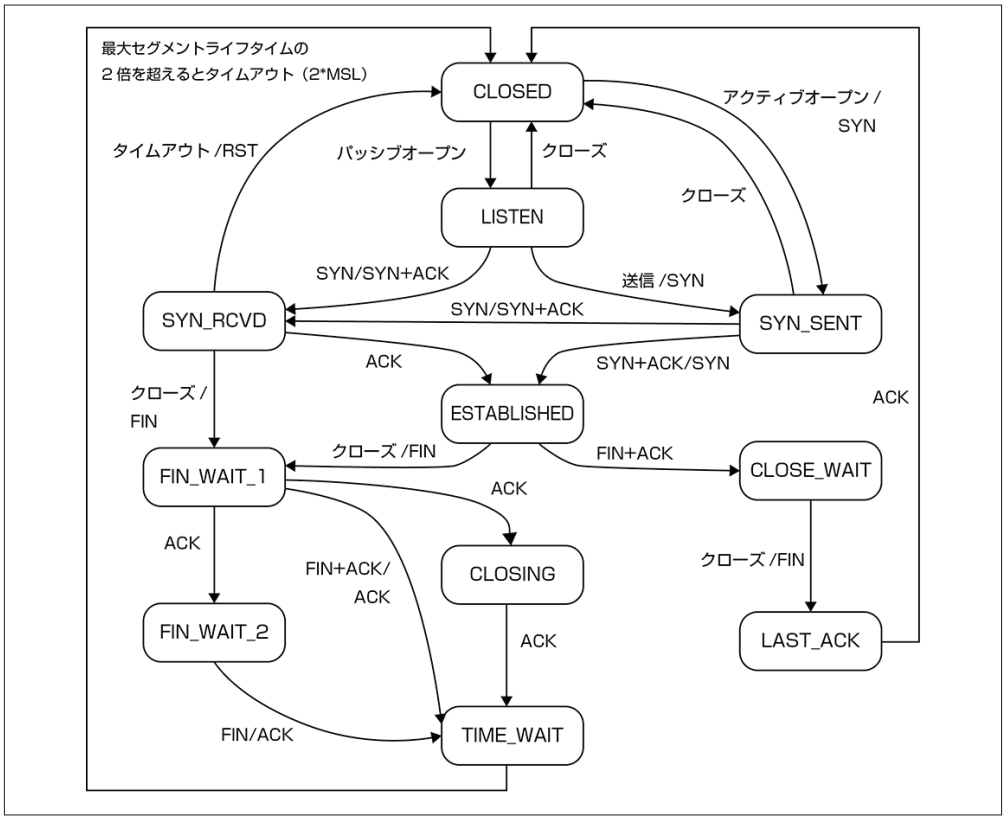


図7-2:TCPのステートマシン

TCPの方はクローズ状態からLISTENして、クライアントから接続リクエストがあって初めてESTABLISH（通信可能）状態になります。HTTP/2のストリームは最初からLISTENとほぼ同じIDLE状態で、ヘッダーを受け取ると即座に通信可能なOPENになり、通信ができるまでのステップが少なくなっています。

各フレームには9バイトの共通ヘッダーがあります。

要素	サイズ	意味
Length	24	ペイロードのサイズ（共通ヘッダーは除く）
Type	8	フレームの種類
Flags	8	
R	1	予約領域（常に0）
Stream Identifier	31	ストリームの識別子。同じ数値なら同じストリームの関連フレーム
Frame Payload	Lengthで指定された長さ	フレームの実データ

この中で注目すべきポイントはStream Identifierです。HTTP/2では擬似的なソケットとしてストリームが作られました。TCPソケット上のデータを見ても、ストリームという実体が存在するわけではありません。同じStream Identifierを持つ一連のフレームが受信時にグループ化されて、「同じストリームから出てきたデータ」として扱われます。

Stream Identifierの「0」は予約されており、使うとエラーになります。奇数はクライアントからサーバー、偶数はサーバーからクライアントへの通信に使われます。

表7-1: フレームの種類

種類	データ	オプショナルデータ	説明
HEADERS	ヘッダー	依存するストリーム＆優先度＆排他フラグ	圧縮されたヘッダー。優先度は最初のヘッダーのみで使用可
DATA	データ		ボディの送信で使う
PRIORITY	依存するストリーム、優先度、排他フラグ		
RST_STREAM	エラーコード		エラー情報を返し、ストリームを即座に終了
SETTINGS	識別子（16ビット）、設定値（32ビット）の組が複数個		
PUSH_PROMISE	ストリームID	リクエストヘッダーフィールド	サーバーからのサーバープッシュ開始の予約
PING	8バイト分のデータ		応答速度計測用フレーム。PINGを受け取ったら、ACKフラグを設定して返す
GOAWAY	最終ストリームID、エラーコード	追加のデバッグ情報	コネクションを終了させる
WINDOW_UPDATE	ウインドウサイズ	追加で受信可能なデータサイズ	
CONTINUATION		HEADERS/PUSH_PROMISEの続きのデータ	

SETTINGSではヘッダーテーブルサイズ、プッシュの許可、最大の並列ストリーム数、初期ウインドウサイズ、最大フレームサイズ、最大のヘッダーリストのサイズ、拡張CONNECTを利用するかどうか（WebSocket通信用）を変更できます。HTTP/2では後方互換性のない機能拡張を認めていますが、そのときはこのSETTINGSを使ってオプトインする仕様になっています。

オプションとなっているのは、共通ヘッダーのFlagsに「このデータがあります」という指定のビットが立つと、決まったサイズの情報が決まった順序でフレームに追加されます。ビットが立たなければそのデータは枠ごと省略されるため、不要なデータでデータ長が伸びるのを防いでいます。

例えば、HEADERSフレームは、必須の項目はヘッダーそのもののデータだけなので、図7–3が最小のフレーム構造となります。

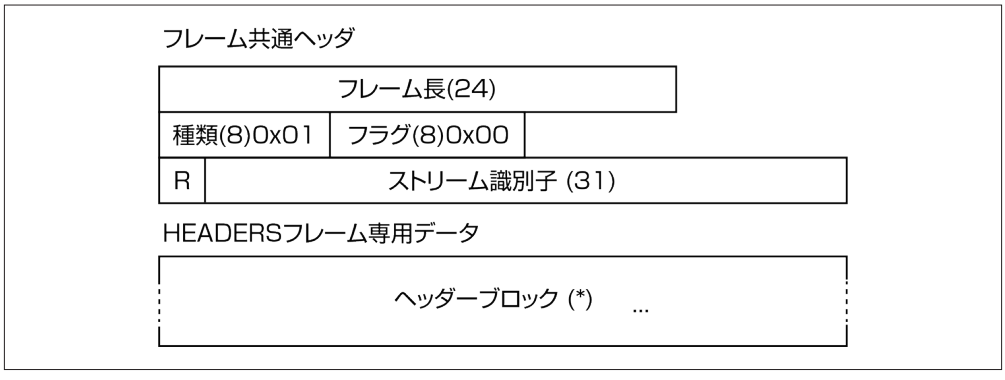


図7-3:最も小さなHEADERSフレーム

種類の次の8ビットのところに次のビットが付与されると、ヘッダーブロックの前に指定されたデータがあるものとしてパーサーがバイナリデータを解釈します。

PADDEDフラグ (0x08)

パッド長の8ビットと、そのパッド長で指定されたパディングが追加される

PRIORITYフラグ (0x20)

ダミーの1ビットと、ストリーム依存関係を表す31ビット、重要度の8ビットの合計40ビットのデータが追加される

図7-4は、すべての設定可能なフラグが設定された時のフレームフォーマットです。フラグは個別にON/OFFできるため、例えば、PRIORITYフラグだけが追加されると、最小の構造に40ビット分のデータが追加されたフレームフォーマットになります。

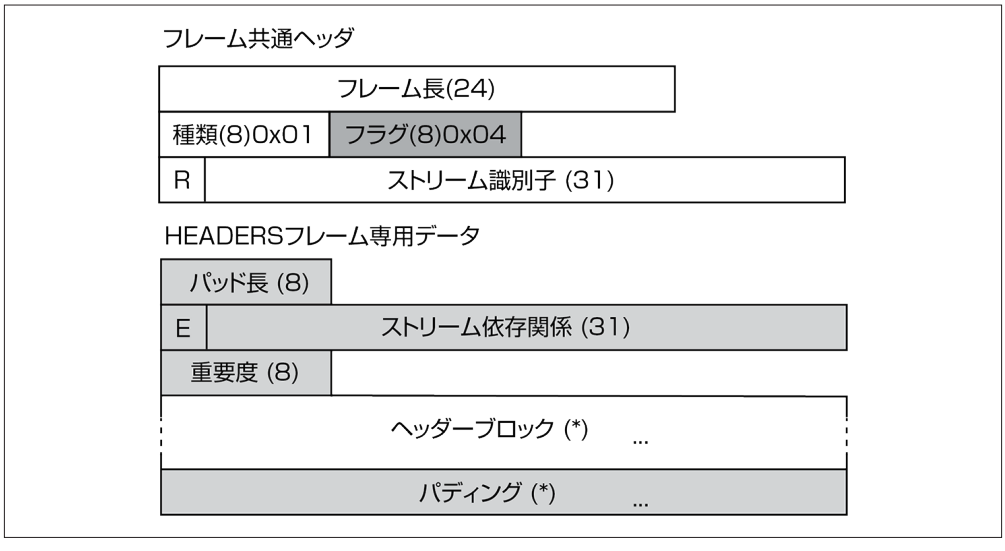


図7-4: フラグが設定されたHEADERSフレーム

HEADERSフレームや、CONTINUATIONフレームはヘッダー終了フラグを、HEADERSフレームとDATAフレームはストリーム終了（END_STREAM）フラグ（0x01）を持つことがあります。そのフラグが立っていると、そこでヘッダーやストリームが終了し、その後にはデータが続かないことが明示されます。それ以上ヘッダーが来ないことを示す（END_HEADERS）フラグ（0x04）もあります。これらのフラグは制御の変更用で、フレームフォーマットには影響はありません。ただし、CONTINUATIONフレームはストリーム終了フラグを持つHEADERSフレームの後に続く可能性があります。CONTINUATIONフレームはストリーム終了フラグが持てないため、ヘッダーが大量にあってボディがない時は、HEADERSフレームが終了フラグを持つことで、ボディがないことを示します。

「4.7 チャンク」で紹介したチャンク方式ですが、ヘッダーでchunkedエンコーディングの開始を指示して、本文をサイズとデータを改行区切りで送信する方法はHTTP/2にはありません。フレームの機構に完全に組み込まれており、複数のフレームに分割して送信することで、同等のことができます。

なお、フレームタイプや、SETTINGSの項目はIANAのHypertext Transfer Protocol version 2 (HTTP/2) Parameters[†4]にまとまっています。HTTP/2の仕様策定から本書の改訂版の執筆までの間に追加されたのはHTTP/2上でWebSocketを実行するためのSETTINGSの値1つだけですが、これからもプロトコルに対する小さい拡張はこちらに追加されていくでしょう。

†4 https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml

7.2.4 HTTP/2のアプリケーション層

HTTP/1.0は単にデータを運ぶ箱でした。その後Keep-Alive、パイプライニングなど、下のレイヤーの通信処理に影響を与える機能も含み始めましたが、本書で何度も取り上げている、メソッドとパス、ヘッダー、ボディ、ステータスという4つの基本要素が存在するのは変わりません。しかし、メソッドとパス、ステータス、あとはステータス行に含まれていたプロトコルバージョンは、すべて疑似ヘッダーフィールド化されて、ヘッダーの中に組み込まれました。

いくつか減った情報もあります。まず、ステータス行の理由文はなくなりました。上位のアプリケーションからみた機能性には変更がありませんが、実装上は「ヘッダー」「ボディ」しかありません。また、以前は`HTTP/1.1 200 OK`のように、レスポンスの中にサーバーのレスポンスがどのバージョンで作られたものかを表す情報もありましたが、それも無くなりました。しかし、ヘッダーの意味といったセマンティクスはHTTP/1.1でもHTTP/2でも、次に紹介するHTTP/3でも変わりません。

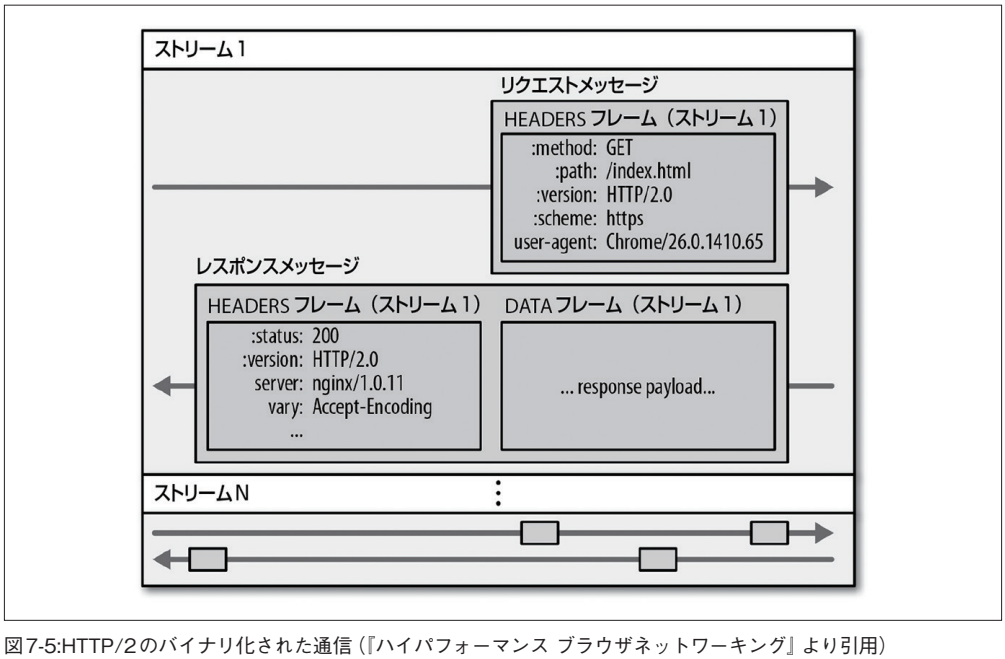


図7-5:HTTP/2のバイナリ化された通信（『ハイパフォーマンス ブラウザネットワーキング』より引用）

HTTP/1.1はテキストプロトコルでした。ヘッダーの終端を探すには空行を見つけるまで1バイトずつ読みこんで発見する必要があります。エラー処理などもあるため、サーバーとしてはパースまで含めて逐次処理せざるを得ず、高度な並列化は難しいと思われます。HTTP/2はバイナリ化されており、レスポンスの先頭にフレームサイズが入っています。TCPソケットのレイヤーでは、データをフレーム単位へと簡単に切り分けられるため、受信側のTCPソケットのバッファをすばやく空にでき、通信相手

に対しても次のデータを高速にリクエストできるようになります。

ボディについてはContent-Lengthでサイズが一意に決まることが多いですし、チャンク形式の場合もサイズが記述されていることから、ひとつのリクエストに対する読み込みの処理コストはそれほど変わらないでしょう。ただし複数のリクエストがある場合には話が変わります。HTTP/1.1ではチャンク形式といっても、1リクエストの最中に他のリクエストを処理することができませんでした。6つのTCPセッションで処理していたとしても、重いレスポンスが6本あると他の通信をいっさい行えなくなります。HTTP/2ではチャンクはフレームに分割されますし、フレーム同士は独立していますので、途中に他のフレームが挟まっても問題ありません。

7.2.5 フローコントロール

HTTP/2はインターネットの4階層モデルのうちのアプリケーション層に当たるものですが、トランスポート層に近いものを内部に持っているのが特徴です。フローコントロールとしてTCPソケットが行っているのとほぼ同じ機能を実装しています。もちろん、TCPがパケットの順序の制御や再送処理を行ってくれるため、実装としてはシンプルです。TCPソケットとHTTP/2のストリームの関係は、OSスレッドとグリーンスレッド[†5]の関係に似ていると言えます。

フローコントロールは、ストリームを効率よく流すために用いられる通信量の制御処理です。通信速度差がありすぎる機器の組み合わせで通信するときに、速い方の機器が遅い方に大量のパケットを送りつけて処理できなくなるような事態を防ぐのが目的です。これを実現する具体的な手段は、通信先のウインドウサイズの管理を使うものです。ウインドウサイズは受け取ることのできる空きバッファサイズで、デフォルトの初期ウインドウサイズは64キロバイトです。送信側は相手のバッファサイズの限界量のデータを送信します。受信側で送信されてきたパケットを処理してバッファに余裕が出てくると、WINDOW_UPDATEフレームを使って、新たに余裕ができたバッファサイズを送信側に返します。送信側がこの通知を受け取ると、空いた量を埋めるぶんだけ続きのデータを送ります。

自動車で交差点や踏切を渡るときには、その先の道路に自分の車が入れるスペースが開くまで、交差点の手前や遮断器の後ろで待たなければなりません。スペースがないのに交差点や踏切に入ってしまうと、他の通信を妨げたり、近づいて来る電車が避けられなくなって重大な事故が起きる可能性があります。フローコントロールが行っていることはそれと同じようなことです。

SETTINGSフレームを使うと 初期ウインドウサイズ、最大の並列ストリーム数、最大フレームサイズ、最大ヘッダーリストサイズといった速度に関するパラメータをチューニングできます。

7.2.6 サーバープッシュ

HTTP/1.1とHTTP/2でセマンティクスは変わらないと本章の冒頭で紹介しましたが、**サーバープッ**

†5 カーネルではなくユーザー領域で作られる疑似スレッド。Go言語のgoroutineもこれに該当します。

シュと呼ばれる機能だけは新たに追加されました。サーバープッシュによって、クライアントから要求される前に優先度の高いコンテンツを送信できるようになりました。ただしWebSocketのような双方向通信を実現する仕組みとは異なり、あくまでもCSSやJavaScript、画像など、ウェブページを構成するファイルのダウンロードに用いられます。クライアントがリクエストを送信するまで、データがサーバー側からプッシュされていることは検知できません。プッシュされたコンテンツは事前にキャッシュに入ります。コンテンツがキャッシュに入った後で、クライアントがそのファイルをリクエストすると、遅延なくダウンロードできたように見えるというわけです。チャットやGoogle Spreadsheetで共同編集をしているときに、他のユーザーが入力したコメントをサーバー側から配信するような用途には使えません。

7.2.7 HTTP/2とプリロードによるリソース取得の最適化

HTTP/2が優先度、サーバープッシュなどのダウンロード最適化機能を備えています。1回のJSON通信で終わってしまうウェブAPIではこれらの機能は役に立ちませんが、スクリプトやCSS、画像、APIアクセスなどの関連ファイルがあるときに効果的にダウンロードを行えるようなしくみがいろいろあります。

- サーバープッシュ
- 優先度

まだ紹介していないものもありますので、これらと一緒にどのようにこれらの機能が関連しているのか紹介していきます。

- リソースヒント
- プリロードヒント
- 事前の通信準備

一番効果が大きいのはサーバープッシュです。これはリクエストが発生する前に送信を開始するため、もっとも効果が高くなります。クライアント側からサーバーに対してリクエストを行う通信が1つ省略されるからです。

サーバープッシュ以外の通常のケースは、HTMLファイルをロードして、その中で使われているファイルをピックアップして、リクエストを開始します。

TLSやHTTP/2の説明では、プロトコルによってTTLが短くなって高速になったという話を紹介しましたが、実際の通信にあたっては、このTLSの前にもいろいろな準備が必要です。HTMLを取得

してくるサーバーと、他のリソース（ウェブフォント[†6]やXMLHTTPRequestやFetch APIによる外部APIへのアクセス）のサーバーが違う場合、HTTP/1.1やHTTP/2ならTCPレイヤーのハンドシェイクなどのコストはかかりますし、TLSのハンドシェイクも事前に鍵が交換できなければ0-RTTになりません。これらを先行して行うように、<link>タグやLinkヘッダーを使って指示できます。

これらのタグと、あとで説明するprerenderは、**リソースヒント**という仕様にまとまっています[†7]。

```
<!-- DNSの名前解決、TCPハンドシェイク、TLSハンドシェイクを先に行う -->
<link rel="preconnect" href="//cdn.example.com" crossorigin>


Link: <https://fonts.gstatic.com>; rel=preconnect; crossorigin
```

ファイルが必要なことを事前に知らせる方法が**プリロード**[†8]です。HTTP/1.1ではchunkedエンコーディングを使ってHTMLのヘッダー部分を先に送信することで通信の最適化が行える余地があることを紹介しましたが、これをLinkヘッダーか、<link>タグを使って実現します。

```
Link: <https://example.com/other/styles.css>; rel=preload; as=style
```

この仕組みがあると、HTMLファイルの解釈前のHTML本体ダウンロード開始前に関連リソースのダウンロードを開始できます。なお、nginx 1.13.9ではhttp2_push_preloadディレクティブをonに設定[†9]することで、またH2O[†10]にもhttp2-push-preloadディレクティブをONに設定すること[†11]で、Linkタグをサーバープッシュにアップグレードする機能があります。これはプリロードのドラフトでも、HTTPサーバーはサーバープッシュを行ってもよい（MAY）と書かれている振る舞いになります。nopushを付けることで、このサーバープッシュへのアップグレードを抑制することもできます。

なお、プリロードもサーバープッシュも、それ自身ではファイルの取得にしかならず、その後実際に使用する宣言があって初めて処理（JavaScriptのファイルなら実行）されます。次のHTMLファイルでは、スタイルシートとJavaScriptのファイルをプリロードしています。

リスト7-1:プリロードと実際に使う宣言の違い

```
<head>
  <meta charset="utf-8">
  <title>JS and CSS preload example</title>
  <!-- プリロード -->
```

†6 ウェブページに表示するためのフォントで、HTTPを利用してダウンロードします。ダウンロードまではデフォルトフォントで表示され、ロード後に指定フォントで表示されるため、なるべく速くダウンロードすることがよりユーザー体験には大切です。しかし日本語や中国語だとファイルサイズが大きく、ダウンロードに時間がかかるという問題もあります。

†7 https://www.w3.org/TR/resource-hints/

†8 https://www.w3.org/TR/preload/

†9 https://www.nginx.com/blog/nginx-1-13-9-http2-server-push/

†10 https://github.com/h2o/h2o

†11 https://h2o.examp1e.net/configure/http2_directives.html#http2-push-preload

```
    <link rel="preload" href="style.css" as="style">
    <link rel="preload" href="main.js" as="script">

    <!-- CSSを実際に使う -->
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>bouncing balls</h1>
    <canvas></canvas>
    <!-- JavaScriptを実際に使う -->
    <script src="main.js"></script>
  </body>
```

プリロードが効果的に働くのは、CSSの中で定義されているウェブフォントなどでしょう。CSSを読み込んで、CSSをパースして初めて必要なことがわかってサーバーにリクエストを送ってフォントファイルをロードします。リードタイムが伸びるのでフォントをプリロードしておくことには価値があります。

その他にはサーバープッシュとは異なり、別のドメインのリソースにも使えるという特徴があります。最初のページでは不要だが、遷移先のページで必要なリソースを先に宣言しておくことも可能なようですが、Google Chromeでは3秒以内に実際にロードをしない場合に警告を出します。

遷移先のページがわかっているなら、先にその次のページを取得してレンダリングしておく、プリレンダリングというさらにアグレッシブな方法もあります。

```
<link rel="prerender" href="//example.com/next-page.html">
```

もちろん、これらの方法は万能ではありません。遷移先のページが決まっていることはあまりないでしょう。ブラウザ表示をブロッキングしてしまうのも、HTMLがすごく小さく、JavaScriptでページのほとんどを組み上げてしまうようなシングルページアプリケーションではそもそも問題にならなかったりします。シングルページアプリケーションのページ遷移ではHTMLファイルを取得することがなく、JavaScriptだけでページを組み替えてしまうため、プリレンダリングはききません。

Akamai社とGoogle社の社員が現在検討中の仕様として、Priority Hints[†12][†13]というものがあります。importance属性を<link>、<img>、<script>などのタグにつけることでHTTP/2の優先度を変更できます。

```
<img src="img/carousel-1.jpg" importance="high">
```

デフォルトでは、CSSはhighest、JavaScriptはhighの優先度でダウンロードされます。プリフェッチの場合はlowになります。Priority Hintsを使うとこれらの優先度を上書きできます。

†12 https://developers.google.com/web/fundamentals/performance/resource-prioritization#preload
†13 https://wicg.github.io/priority-hints/

JavaScriptとCSSのダウンロードだけではなく、実行の優先度もユーザー体験に影響があります。async/deferを付けると、そのタイミングではブロックせずにバックグラウンドダウンロードを行います。ダウンロード完了したらその場で表示をブロッキングして実行するのがaysncで、最後に<script>タグを書くのと同じタイミングで実行されます。ただし、ダウンロードの優先度はlowに設定されます。

```
<script src="script1.js" async></script>
<script src="script2.js" defer></script>
```

サーバープッシュと優先度をきちんと設定しきるためには、アプリケーション特性までかなり踏み込んだ設定が必要なことがわかると思います。HTTPのプロトコルの関連機能の紹介ですが、HTMLのタグとも関連していました。何かの優先度を上げるということは、他の優先度を下げることになります。実際のウェブサイトの特性によって効果が変わってきますし、間違って適用すると改悪になりえるため、慎重な運用が必要です。

7.2.8 HPACKによるヘッダーの圧縮

ヘッダーはHPACKという方式で圧縮されます。世の中の圧縮アルゴリズムの多くは、データ圧縮時に、辞書と、辞書のキーの配列という2つのデータを作ります。同じ長い文が多ければ多いほど、辞書の項目は少なくなり、同じキーが何度も使われるようになって圧縮率が上がります。HPACKは普段使われるファイル圧縮のアルゴリズムと違い、事前に定義済みの辞書（静的テーブル）を持っています。HTTPヘッダーでは決まった名前や結果がよく出現するため、それを外部辞書に入れておくと圧縮後のサイズが小さくなります。HTTP/2では静的テーブルという名前で、事前に頻出するヘッダー名とヘッダー値の組をテーブルに持っています[†14]。

それに加えて、同じコネクション中に登場したHTTPヘッダーはインデックス化されて動的テーブルに格納されます、再度登場したときにはそれのインデックス値だけで表現できるため、小さなサイズで送信できます。

HTTP/2では、今までヘッダーで扱っていなかったいくつかの項目が擬似ヘッダーとしてヘッダーに格納されるようになりました。

`:authority`

リクエスト先のホストとポート

`:method`

メソッド

†14 https://tools.ietf.org/html/rfc7541#appendix-A

`:scheme`

スキーム（HTTP/HTTPS）

`:status`

ステータスコード

なお、`:status = 200`といった、ヘッダーとその値の組み合わせでよく使われるものは、静的テーブルにあらかじめ登録されています。この例であれば、インデックス値8が割り振られています。

『ハイパフォーマンス ブラウザネットワーキング』では「HPACKはサイズを減らすためにヘッダーの差分だけを送信する」と説明されていますが、これは正式仕様になる前にあったReference Setという機能です。機能が複雑な割に削減効果が小さかったため、辞書を使った方式に一本化されました[†15]。

7.3 HTTP/3

SPDYがHTTP/2の基盤となりましたが、HTTP/3でもQUICが最初に作られ、それがRFC化される中でHTTP/3となっていきました。本節ではQUICからスタートしてHTTP/3への流れを紹介していきます。

7.3.1 QUIC

HTTP/2はあくまでもHTTPと同じレイヤーである、TCPソケットの上に実装されましたが、Googleはさらなる高速化のためにUDPソケット上に**QUIC（クイック）**というプロトコルを用意しました。TCPは接続の初期に何度が通信を往復する必要があります。また、エラー訂正や順序の整列を行いますが、そのために受信通知を返す必要があるなど、高機能なぶんパフォーマンスは多少落ちます。

TCPと対になっている軽量プロトコルがUDPです。UDPはTCPから再送処理、輻輳制御などの高機能な機能を取り払い、初回接続時のネゴシエーションを軽くしたプロトコルです。QUICはTCPが行っているような、パケットがロストした時の再送処理、通信経路が輻輳（混雑してパフォーマンスが劣化している状態）した時の制御など、多くのことを自前で実装しています。また、TLSの機能を最初から融合しており、シーケンス番号など、TCPのレイヤーで持っているデータも隠蔽します。

通常のHTTPS通信では、TCPのハンドシェイクを行った後に別途TLSのハンドシェイクを行う必要があり、何往復もパケットを交換する必要がありました。QUICは両方を統合しており、より少ない通信で接続できます。初回でも1往復の通信でネゴシエーションしたり、再接続ではネゴシエーション

†15 http://qiita.com/iwanaga/items/0247e6873496067591ec

なしに0RTTで再送できるような仕組みになっています。また、スマートフォンの3G/4Gの回線からWifiに切り替わった時の再接続もスムーズになっています。

QUICが優れている点はこの再接続や初回の通信コストだけではありません。これら以外はTCPと変わらないように思えますが、HTTP/2が行っていた処理とTCPで行っていた処理を整理することで、2つのレイヤーで重複していたタスクを簡略化しています。例えばTCPとHTTP/2の両方に、ほぼ同様の機能としてあったフローコントロールが一元化されています。パケット順序の整列化では、アプリケーション層を知らないTCPは全パケットを愚直に整列化しますので、このレイヤーで通信エラーと再送が発生すると、すべての通信が止まってしまいますが、HTTP/2ではストリーム単位で必要なだけ整列させます。

7.3.2 HTTP/3への道

Googleは2013年にQUICの発表を行い、2015年にRFC化のための最初の提案を行いました[†16]。その後、この最初の実装を元に議論の中で仕様が変更されており、Google版とIETF版では少しずつ違いが生じています。そこでGoogle版のQUICを**gQUIC**、IETF版を**iQUIC**と区別して呼ぶこともあります。その中で、このQUICのIETF版として議論されているHTTP over QUICを**HTTP/3**という名前とすることが決定されました。現在、2020年7月の標準化に向けて、作業が急ピッチで進められています。

QUICも、SPDYと同じようにすでにChromeに組み込まれています。ユーザーがGoogleのサービスを利用するときには、すでに半分のリクエストがQUICで行なわれているとGoogleは発表しています。この規格は標準化を目指しRFC化も進行中です[†17]。

7.3.3 HTTP/3のレイヤー

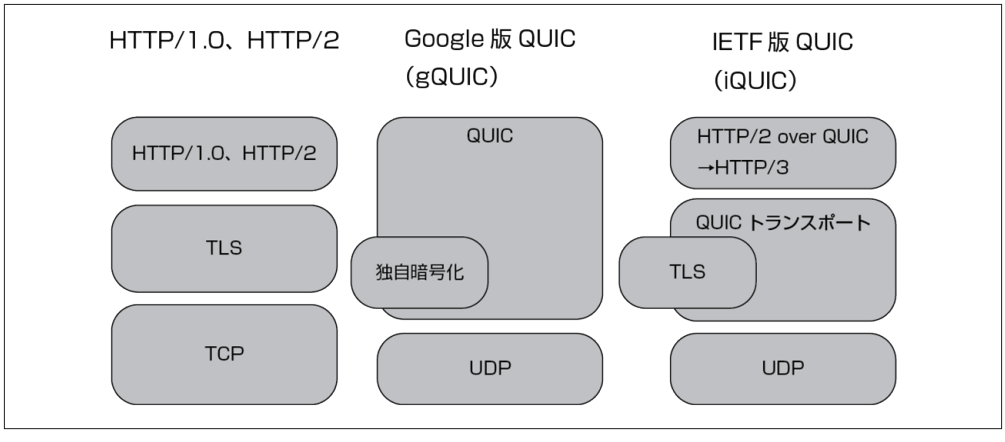
Googleが実装していたQUICは、HTTP/2までのプロトコルと比較すると、HTTPのレイヤー、暗号化のレイヤー、さらに、その下のTCPで行なっている再送処理などのレイヤーをまとめた大きなプロトコルでした。その後、IETFの標準化版のQUIC（iQUIC）は、議論のなかでいろいろな変更を受け入れ、Google版のQUIC（gQUIC）からさまざまな点で改善されています。暗号化を独自のものからTLS 1.3のハンドシェイクや暗号スイートをそのまま活用する方向でマイグレーションするとともに、おおきく2つのレイヤーに分割されました。

- QUICトランスポート: ストリーム制御などのTCPの上位互換のレイヤー（並列対応版）と、TLS 1.3
- HTTP over QUIC: HTTP/2の機能から、QUICトランスポートと重複する機能を落としてシンプ

†16 https://tools.ietf.org/html/draft-tsvwg-quic-protocol-00
†17 https://datatracker.ietf.org/wg/quic/charter/

ルにして、HPACKをQPACKに置き換えたもの

このうち、上位レイヤーのHTTP over QUICと呼ばれていたプロトコルが**HTTP/3**となることが内定しています。



QUICトランスポートプロトコル

QUICトランスポートプロトコルは、UDPを元にして、再送処理、輻輳制御、暗号化などを実装したものになります。大雑把にいってしまうと、TCPとTLSのレイヤーの置き換えです。

TCPの場合は、1つのコネクション内のパケットがすべて整列されて転送されることを保証するプロトコルでした。QUICは、セッションは張るものの、その中に複数の通信の疑似セッション（ストリーム）を張り、ストリームの内部では順序保証はするものの、ストリーム間での保証はしません。それにより、1つのストリームが失敗したときの影響が、他のストリームに及ばなくなります。

QUICは、より21世紀のネットワーク事情に適した仕様です。具体的には4Gや5G通信をしていたところから、Wifiがつながって通信をオフロードして高速に伝送する、というネットワーク経路が動的に変化しうるようなモバイル通信でも、再接続にともなうラグが発生しないようになっています。

それまでは、5-tupleと呼ばれる5つの情報を使ってコネクションの識別が行われてきました。この場合、ネットワーク経路が変わってしまうと、別のコネクションと識別されてしまいます。

一方、QUICの接続時にはコネクションIDが発行され、これを元にストリームの通信を行います。このコネクションIDは通信経路が変わっても維持されるため、どのような経路の通信が混在しても、同じコネクションに属するパケットであると判断でき、接続を維持できます。これにより下位の変更が上位に及ばないようになっています。

表7-2: コネクションを識別する方法

HTTP/2まで	送信元アドレス、宛先アドレス、送信元ポート番号、宛先ポート番号、プロトコル
QUICトランスポート	コネクションID

QUICではストリームごとにフロー制御を行ったり、ウインドウサイズを変更したりできます。このあたりは実装こそ違いますが、HTTP/2の概念にかなり近いといえるでしょう。

HTTP/2と異なるのは、TLSを使わないプロトコルが定義されていない点です。もっとも、HTTP/2もTLSでないと有効にならないライブラリが多く、非TLSで利用する方法はありませんでした。それがそのまま仕様化されました。なお、TLSについてはハンドシェイクのマスターシークレット生成部分だけはそのままの利用になっていますが、ハンドシェイク用の情報交換もQUICトランスポートのフレームを使って行うため、HTTP/2以前と完全には同じではありません。利用可能な暗号スイートはTLS 1.3準拠ですが、一部の弱いアルゴリズムが無効化されています。

HTTPのレイヤーとトランスポート層が分離されたので、HTTP以外の他のアプリケーションへの応用も期待されるところですが、現在のところはHTTPにフォーカスして標準化作業が行われています。それでも、まえがきで触れたように、Googleはすでに、duoというアプリケーションでVP9/SVC over WebRTC over QUICという組み合わせをしていましたし、特にWebRTC方面での動きが活発になっています[†18][†19]。

7.3.4 HTTP Alternative Servicesによるアップグレード

HTTP/2までは、TCP上のTLSのコネクションを張るところまでは同じで、ALPNを使ってプロトコルのネゴシエーションが行えました。HTTP/3の場合はそれができないため、別の方法が必要です。RFC7838を応用してHTTP/3での再接続ができることをクライアントに伝える方法があります。このRFCは、同一のサービスが別のオリジンでも動作していることを示します。その際に、プロトコルなどを指定することができるため、これを使ってHTTP/3が動作しているサービス側に誘導できるというわけです。

```
Alt-Svc: h3=":50781"
```

現在はHTTP/3を表すウェルノウンポートが決まっていない（GoogleのQUICはUDP/443ポート）ので、とりあえずHTTP/3で接続しに行くというのは今はないですが、一度接続したサービスを覚えておいて再接続時にHTTP/3で最初からつなぎに行く、HSTSのようにHTTP/3接続が可能なデータベースをブラウザベンダーが作るなど、何かしらの改善がされるでしょう。

†18 「WebRTC DataChannelに忍び寄るQUICの影」https://medium.com/shiguredo/webrtc-datachannel-%E3%81%AB%E5%BF%8D%E3%81%B3%E5%AF%84%E3%82%8B-quic-%E3%81%AE%E5%BD%B1-89cf81b861a3

†19 https://w3c.github.io/webrtc-quic/

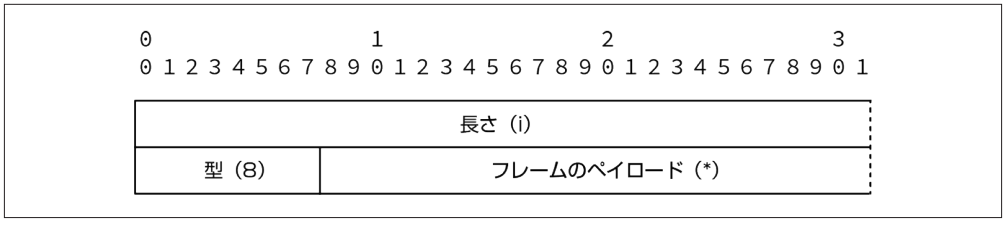
HTTP/3

HTTP/2と比較すると、QUICトランスポート側と、HTTP/3とに分かれたため、HTTP/3側はシンプルになっています。

全体的な違いとしては

- ストリームIDのビット数が31から62に増えてオーバーフローの危険性がかなり減った
- 各フレームはストリームIDとフラグを持たなくなった
- フラグによって増減するフィールドがなくなった
- ヘッダー圧縮のアルゴリズムが、HPACKから、QPACKという別の方式に変わった

フレームの構造を見るとかなりシンプルで共通ヘッダー長がかなり短くなっています。



長さは種類フィールドの1バイトを除いたペイロード長を表しますが、先頭2ビットを使ってデータ長を区別し、1から4バイトまでの可変長の表現になっています。例えば、00であれば、残りの6ビットを使って最大63バイトまでのサイズが表現できます。最大では先頭11のときに4バイト、有効ビット数62ビットとなり、4エクサバイトまでの長さが表現できます。

各フレームでも、ストリームの制御に関するフレームがなくなったり、全体的にシンプルになっています。

表7-3: フレームの種類

種類	データ	h2	h3	違い
HEADERS	ヘッダー	○	○	優先度がPRIORITYフレームに移動
DATA	データ	○	○	パディングがなくなる
PRIORITY	依存するストリーム、優先度、排他フラグ	○	○	コントロールストリーム上でのみ流れる
RST_STREAM	エラーコード	○		
SETTINGS	識別子（16ビット）、設定値（32ビット）の組が複数個	○	○	コネクションの最初でだけ呼ばれる
PUSH_PROMISE	ストリームID	○	○	サーバーからのサーバープッシュ開始の予約
PING	8バイト分のデータ	○		存在しない

種類	データ	h2	h3	違い
GOAWAY	最終ストリームID	○	○	サーバー→クライアントでのみ送信される。エラーコードはない
WINDOW_UPDATE	ウインドウサイズ	○		
CONTINUATION	HEADERS/PUSH_PROMISEの続きのデータ	○		
MAX_PUSH_ID	クライアントが受取可能なPush IDの最大値		○	クライアント→サーバーでのみ送信可能
CANCEL_PUSH	サーバープッシュの受け取り拒否		○	
DUPLICATE_PUSH	サーバー→クライアント 複数のクライアントのリクエスト関連していることを示す		○	

QUICトランスポートによって、ストリーム内部でのパケットの順序は保証されますが、ストリーム間でパケットの順序を保証されなくなりました。そのため、送信順序が常に同じというネットワークの要件に依存していたHPACKが使えなくなり、別の方式になっています。QPACKはHPACKから大きく変わるものではなく、静的テーブルと動的テーブルに入れたデータを使いインデックスで参照するということは変わりません。HPACKは動的テーブルにヘッダーのキーを登録し、実際のリクエストではそのインデックスを設定することで通信量を削減しています。しかし、ヘッダーへのエントリーの登録と参照が必ず決まった順序で行われることが前提とされていました。QPACKは圧縮率を少しだけ犠牲にして、登録時に「どこのインデックスに登録するか」という情報も追加で送信するようになっています。それにより、挿入順序によってテーブル内のレコードの順序が入れ替わることを防いでいます。ただし、利用側がまだ未登録のテーブルのインデックスを参照しようとした場合には、そこで待ちが発生するため、ヘッドオブラインブロッキングが完全になくなるわけではなく、一部残っています。

本書ではバイナリプロトコルの詳細は紹介しません。現時点で日本で一番詳しいのは、まえがきで紹介した西田佳史氏の記事「パケットの設計から見るQUIC」です。興味を持った方はこちらを参照してください。

7.4　新たなJavaScript用の通信API

ウェブサービスが社会のインフラになり、XMLHttpRequestを活用したAjaxのような技術を使った、動的でインタラクティブなサイトが数多く使われています。その中で、つぎはぎのように拡張された仕様をウェブのエコシステム全体にきちんと適合させようと、XMLHttpRequestを再設計したFetch APIが追加されました。またクライアントからの単純なHTTPリクエストだけでは効率良く実装するのが難しいユースケースも増えています。その問題を解消するServer-Sent Events、WebSocket、WebRTC（Web Real-Time Communication）が追加されました。

Fetch API

ワンショットのクライアント起点のHTTPのリクエストが再設計されたもの

Server-Sent Events

サーバーからクライアント側への通信を規格化したもの

WebSocket

効率の良い双方向通信（エラー訂正あり）を実現したもの

WebRTC

P2P通信（サーバーを介した通信も可）での動画・音声・データ通信を実現するUDPベースの通信

7.4.1 Fetch API

Fetch API[†20]はXMLHttpRequestと同様の、サーバーアクセスを行う関数です。JavaScriptから用いられ、以下のような特徴を持ちます。

- XMLHttpRequestよりも、オリジンサーバー外へのアクセスなど、CORSの取扱いが制御しやすくなっている
- JavaScriptのモダンな非同期処理の記述法であるPromiseに準拠している
- キャッシュを制御できる
- リダイレクトを制御できる
- リファラーのポリシーを設定できる
- Service Worker内から利用できる

Fetch APIは低レベルなAPIと紹介されることもありますし、W3Cの仕様にも低レベルと書かれていますが、決してソケットレベルで扱えるという意味ではありません。キャッシュなどを制御しようと思えば可能であるという意味であって、送受信時に制限されるヘッダー、同一生成元ポリシーの厳格な適応など、セキュリティ制限があって、HTTPリクエストに限定されたサンドボックスであること変わりはありません。ブラウザからsshで外部サーバーにつないだり、gitプロトコルを送信したり、ウェブサーバーを開発したりといった用途には使えません。

詳細な使い方はJavaScriptの章で紹介します。

†20 https://developer.mozilla.org/ja/docs/Web/API/Fetch_API

7.4.2 Server-Sent Events

Server-Sent EventsはHTML5の機能のひとつです[†21]。技術的にはHTTP/1.1のチャンク形式による通信機能を基にしています。チャンク形式は、巨大なファイルコンテンツを小分けにして送信するための通信方式でした。このチャンク形式の「少しずつ送信」という特徴を応用して、サーバーから任意のタイミングでクライアントにイベントを通知できる機能を実現しています。2014年にはGREEチャットのバックエンドに採用されています[†22]。

HTTPは基本的にクライアントからリクエストをサーバーに送り、サーバーがそれに対して応答を返すというクライアント／サーバーモデルです。通信の開始はクライアントが決めますし、クライアントのリクエストが1回行われれば、それに対応するレスポンスが1回発生するというのが基本構成です。

サーバーから情報を返す方法には、4章で紹介したCometがあります。クライアントから定期的にリクエストを送ることでサーバー側のイベントを検出する（ポーリング）、あるいはリクエストを受け取った状態で返事を保留する（ロングポーリング）方法が良く使われていました。未だにこの方法は他の方法が使えなくなった時のフォールバック先として使われています。

Server-Sent EventsはCometのロングポーリングとChunkedレスポンスを組み合わせて、1度のリクエストに対して、サーバーから複数のイベント送信を実現します。実績のあるチャンク形式を使っているため、プロキシ対応も含め後方互換性に問題はありません。

Server-Sent EventsはChunked形式を使っていますが、HTTPの上に別のテキストプロトコルを載せています。これはイベントストリームと呼ばれ、MIMEタイプは`text/event-stream`です[†23]。

```
id: 10
event: ping
data: {"time": 2016-12-26T15:52:01+0000}

id: 11
data: Message from PySpa
data: #eng channel
```

テキストの文字コードはUTF-8です。データはタグの後に書かれます。表7-4のように、4種類のタグがあります。空白行でデータが区切られます。dataタグが連続で送信されているところは、改行が含まれたひとつのdataとして処理されます。上記のサンプルには2つのデータが含まれます。内容はテキストであればなんでもよく、BASE64エンコードすればバイナリも扱えますが、多くの場合はJSONが使われます。

†21 https://developer.mozilla.org/ja/docs/Server-sent_events/Using_server-sent_events
†22 http://labs.gree.jp/blog/2014/08/11070/
†23 https://www.w3.org/TR/2009/WD-eventsource-20091029/#parsing-an-event-stream

表7-4: イベントストリームのタグ

タグの種類	説明
id	イベントを識別するID。再送処理で利用される
event	イベント名を設定
data	イベントと共に送られるデータ
retry	再接続の待ち時間のパラメータ（ミリ秒）

JavaScript側では、Server-Sent EventsにEventSourceクラスを使ってアクセスします。詳しくはJavaScriptの章で紹介します。

7.4.3 WebSocket

WebSocketはサーバー／クライアント間で、オーバーヘッドの小さい双方向通信を実現します。通信が確立すると、サーバー／クライアント間で一対一の通信を行います。フレーム単位で送受信しますが、相手が決まっているために送信先の情報などは持ちません。HTTPの4つの基本要素のうち、ボディのみを送っているようなものです。フレームはデータサイズなどを持っているだけで、オーバーヘッドも2バイトから14バイトしかありません。通信がスタートしたら双方から自由にデータを送受信できます。RFC6455で定義されており、ブラウザのAPIはW3Cで定められています[†24]。

用途としては、チャットなどのリアルタイムな双方向のアプリケーションであったり、通知に使われます。Mozillaは後述するウェブプッシュのバックエンドや、IIJエンジニアリング社が提供する地震速報のAPIで利用されています[†25]。実際のコード例はJavaScriptの章で紹介します。

WebSocketのプロトコルの詳細は『ハイパフォーマンスブラウザネットワーキング』を参照してください。

WebSocketはステートフル

WebSocketが他のHTTPベースのプロトコルと異なるのは「ステートフルな通信である」という点です。HTTPは通信速度向上のためにKeep-Aliveなどの複雑な機構も備えるようになってきていますが、基本的にリクエスト単位で接続が切れてもセマンティクス上は問題がありません。ロードバランサーを使ってサーバーを複数台に分散しておき、リクエストのたびに別のサーバーが応答していても問題ありません。Server-Sent Eventsも、送信済みIDを一元管理して保証できれば、リクエストをさばくサーバーが入れ替わっても問題がないように設計されています。

†24 https://www.w3.org/TR/websockets/
†25 https://doc01.pf.iij-engineering.co.jp/pub/sdkdoc/

WebSocketを使う時のサーバーはメモリ上にデータを持った状態で通信するケースが多いでしょう。テキストチャットのようなユースケースでは、単一のサーバーにすべてのブラウザが接続するだけでなく、MemcachedやRedisを中継して負荷分散させることもできるでしょう。しかし複数人リアルタイム同時プレイのゲームなどの場合、負荷分散にともなう遅延を許容できないこともあります。その場合はチャットであれば「ルーム」のような単位でコネクションをオンメモリで管理します。このような場合、いったん接続が切れた場合の再接続は前と同じサーバーに繋ぐ必要があり、単純なHTTPベースのロードバランサーは使えません。WebSocketの運用時に、クライアントからサーバーを指定して再接続できるようロードバランサを使わずに運用しているという事例も見かけます。あるいはTCPレベルのロードバランサーを駆使する、WebSocket対応のロードバランサーを利用する必要があります。

WebSocketはHTTP/2ができる前に仕様化されたプロトコルですが、HTTP/2との統合方法がRFC8441で定義されました。HTTP/1.1上でWebSocketを使うと、デフォルトで6本程度利用可能なネットワーク接続を1本消耗してしまいます。HTTP/2のストリームに乗ると、HTTP/2の接続にあいのりできて、コネクション数を減らせます。

HTTP/2時代のWebSocket接続

WebSocketsの通信は4章で紹介したプロトコルのアップグレードと似た仕組みを使います。まず通常のHTTPとしてスタートし、その中でプロトコルのアップグレードを行い、WebSocketにスイッチします。

まずはリスト7–2のようにクライアント側からサーバーにリクエストを送ります。`:method`疑似ヘッダーにCONNECT、`:protocol`疑似ヘッダーに`websocket`を付与してアップグレードすることを要請します。

リスト7-2:WebSocketによる通信をリクエスト

```
:method = CONNECT
:protocol = websocket
:scheme = https
:path = /chat
:authority = server.example.com
sec-websocket-protocol = chat, superchat
sec-websocket-extensions = permessage-deflate
sec-websocket-version = 13
origin = http://www.example.com
```

sec-websocket-version

現時点でバージョンは13固定

sec-websocket-protocol

WebSocketは単にソケット通信の機能だけを提供する。その中でどのような形式を使うかは

アプリケーションで決める必要がある。コンテントネゴシエーションのように複数のプロトコルが選択できるように使われる。このヘッダーはオプション

サーバーレスポンスはリスト7-3の通りです。

リスト7-3: サーバーレスポンス

```
:status = 200
sec-websocket-protocol = chat
```

sec-websocket-protocol

クライアントからサブプロトコルのリストを受け取ったときに、その中のひとつを選択して返す。クライアントは送信したプロトコル以外を受け取ったら接続を拒否しなければならない

この後はDATAフレームを使って、双方向通信を行います。

HTTP/1.1時代のWebSocket接続

HTTP/1.1時代のリクエストは、上記の疑似ヘッダーの代わりに次のようなメソッド・ヘッダーを送信していました。Sec-WebSocket-Keyという、ランダムに選ばれた16バイトの値をBASE64エンコードした文字列もありましたが、HTTP/2対応でなくなりました。

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==


HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Sec-WebSocket-AcceptはSec-WebSocket-Keyを決まったルールで変換した文字列です。これにより、クライアントはサーバーとの通信確立を検証できます。ランダムな文字列からSec-WebSocket-KeyとSec-WebSocket-Acceptの文字列を生成するコードはリスト7-4の通りです。Sec-WebSocket-Acceptの値は、特定の文字列（サンプルコード中のsalt変数）を接合した後にSHA1ハッシュを計算し、それをBASE64エンコードした内容になります。

リスト7-4: ランダムな文字列からSec-WebSocket-KeySec-WebSocket-Acceptを生成

```
package main

import (
```

```
    "crypto/sha1"
    "encoding/base64"
    "fmt"
)

func main() {
    clientKeySrc := "the sample nonce"
    key := base64.StdEncoding.EncodeToString([]byte(clientKeySrc))
    fmt.Printf("Sec-WebSocket-Key: %s\n", key)
    // Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

    salt := "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"
    hash := sha1.Sum([]byte(key + salt))
    accept := base64.StdEncoding.EncodeToString(hash[:])
    fmt.Printf("Sec-WebSocket-Accept: %s\n", accept)
    // Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
}
```

Socket.IO

WebSocketは強力なAPIですが、多くの場合、それをさらに使いやすくするSocket.IOというライブラリを経由して使う方法に人気がありました。

Socket.IOのメリットは次の3点です。

- WebSocketが使用できない時は、XMLHttpRequestによるロングポーリングでエミュレーションし、サーバー側からの送信を実現する機能がある
- WebSocketの切断時に再接続を自動で行う
- クライアントだけではなく、サーバー側で使える実装もあり、クライアントが期待する手順でフォールバックのXMLHttpRequest通信をハンドリングしたりできる
- ロビー機能

WebSocketが使われ始めた当初は後方互換性を広く維持できるということで、WebSocketといえばSocket.IOでした。しかし、現在ではWebSocketが使えないブラウザはほとんどありません。企業内のセキュリティ管理のためのウェブプロキシなど、WebSocketが使えなくなる要因はいくつかありますが、後方互換性目的で他のライブラリを使う理由はかなり減ってきています。再接続処理などメリットがゼロではありませんが、今後は直接使われるか、使われるにしてもXMLHttpRequestフォールバックをオフにされることが増えるでしょう。

7.5 WebRTC (Web Real-Time Communication)

WebRTCはこれまで紹介したプロトコルとは大きく異なります。これまで紹介してきたものは、すべてブラウザとサーバーの通信で使われるプロトコルでした。WebRTCはブラウザ／サーバー間の通

信だけではなく、ブラウザとブラウザのP2P通信でも使います。RTCは「リアルタイムコミュニケーション」の略で、テレビ電話などのリアルタイムコミュニケーションを実現するための基盤として設計されています。RFC7478でユースケースが定義されていて、APIの使用方法などはwebrtc.org[†26]にまとめられています。

実現したいアプリケーションも他のプロトコルとは大きく異なるため、使われる機能群も大きく変わります。通信の基盤に使うトランスポート層は、再送処理の面倒を見てくれるTCPではなく、エラー処理や再送処理を行わないUDPをメインで使います。またP2P通信なので、相手のブラウザを探すシグナリング、ルーター内部のプライベートアドレスしか持たないコンピュータで動いているブラウザのため、NATを超えてプライベートアドレスだけで通信できるような技術も使われています。

WebRTCのプロトコルの詳細は『ハイパフォーマンスブラウザネットワーキング』を参照してください。各構成技術はWebRTC用に開発されたものではなく、古くからテレビ電話用に開発されてきた技術や、リアルタイム向けのTLS暗号化を加えたデータグラム通信など、既存の技術を利用したうえで、それぞれを新しくブラッシュアップして組み合わせています。それぞれの技術を細かく紹介するだけで、書籍一冊分には収まらないくらいの分量があります。WebRTCについての概要や最新情報は、時雨堂のVoluntas氏がまとめています[†27]。

7.5.1 WebRTCのユースケース（1）

RFC7478を見ると、どのようなケースに対応したいのかがまとめられています。WebRTCの要素技術はWebSocketほどシンプルなものではなく、組み合わせていろいろなユースケースに対応できるようになっています。機能をボトムアップで紹介するよりも、このユースケースを紹介して技術要素に分解していったほうがスムーズに理解できるでしょう。まずはP2Pを使ったユースケースを紹介します。

シンプルなビデオ通話システム

これから紹介するユースケースの共通の機能要件です。

2人でビデオ会議を行うシステムです。全参加者がブラウザを使って同じサービスプロバイダーにログインをして通話が始まります。各参加者の状態は、サービスプロバイダーがウェブアプリケーション上で公開します。グループで会議しているときに、ユーザーは特定のメンバーを指定して一対一の通話を開始できます。ピア通話の招待を受けたユーザーはそれを承認したり拒否したりできます。また各ユーザーは自分の動画の表示をオン／オフできますし、ビデオのサイズを変更したり、自分や相手のオーディオや映像を一時的にミュートしたり再開できます。

---

†26 https://webrtc.org/

†27 https://gist.github.com/voluntas/67e5a26915751226fdcf 日本におけるWebRTC情報の多くは時雨堂が発信元になっており、また本節のレビューもしていただいています。

サービスプロバイダーの要件についてはこの後の節で2種類紹介します。

ウェブブラウザが直接他のブラウザと繋がるわけではなく、ウェブサービスが起点となります。通信プロトコルの下のレイヤーのトランスポート層には、他のHTTP通信のプロトコルと同様にTCP（ストリーム型ソケット）を使うこともできますが、主にUDP（データグラム型ソケット）を使います。これは再送処理や、通信量制限などを自前で実装しなければならないプロトコルです。ただし、通信経路はUDP用のTLSである、**DTLS**によって暗号化されます。

シンプルといってもブラウザはたくさんのタスクをこなさなければなりません。ブラウザはマイクとビデオカメラを入力デバイスとして利用します。音声と動画をお互いのシステムが利用できるコーデックを使って圧縮します。現在のモバイル環境のデファクトスタンダードは、映像ではMPEG4あるいはBlu-Rayでも使われるH.264です。デスクトップではH.264やVP8などが使われます。音声のコーデックにはOpusが利用されます。これはSkypeが利用していたコーデックで、圧縮・展開の遅延時間が短く、他の形式と較べてファイルサイズが小さくなっても品質が劣化しにくいのが特徴です。これは特許の絡むコーデックですがRFC化されていて（**RFC6716**）、このRFCの範囲で利用するのであれば特許に関するライセンス契約や費用は必要ないことが明示されています。

これ以外にも、ブラウザは送信するビットレートを制御できなければなりません。ネットワーク帯域が広いときはより鮮明な映像で音声もクリアに伝えたいでしょうし、インターネット越しであれば利用できる通信回線に合わせて適切なビットレートを選択する必要があります。映像と音声の同期を取らなければ違和感があるでしょう。

ファイアウォール越しのビデオ通話

アプリケーションの機能要件は前節の「シンプルなビデオ通話システム」と同じですが、非機能要件として次の2つの項目が加わります

- 一部ユーザーがUDPが送信できないファイアウォール越しで使える
- 一部ユーザーがHTTPプロキシ経由の通信しか許可されないファイアウォール越しでも使える

グローバルなネットワークでのビデオ通話

ビデオ会議を行うサービスプロバイダーが、世界規模でインターネットサービスとして提供されています。これを実現するにはICE（Interactive Connectivity Establishment）が必要となります。電話交換機のようなものです。ユーザーIDなど何かしらの情報を元に相手を探し出します。また、NATの内側にいても通話できるように、**STUN（Session Traversal Utilities for NAT）**、**TURN（Traversal Using**

Relay NAT）といった、NATの壁を超えるための仕組みも必要になるかもしれません。IPv4、IPv6などの要件ごとに別のSTUN/TURNが必要になる可能性もあります。

エンタープライズ環境でのビデオ通話

エンタープライズ用途では、社内から社外への通話を管理できる必要があります。そのために、内部ネットワークと外部ネットワークをまたぐTURNサーバーを設置します。ファイヤーウォールは、この監視されているTURNサーバーを使わない通信をブロックします。

スクリーン共有

ユーザーはビデオの代わりに、現在見えているデスクトップ、あるいはデスクトップの一部、特定のアプリケーションのスクリーンショットやスクリーンキャスト（動画）を相手に送信できます。

ファイル交換

複数人とビデオ会議をしながら、特定の相手にファイルを送信できます。

ホッケーゲームビューア

RFCに書かれている例が具体的すぎるのですが、そのまま引用します。アイスホッケーチームのスカウトと監督が新しい選手を発掘しようとしています。ゲームが始まると、スカウトはスマートフォンのリアカメラでホッケーの試合を写し、フロントカメラで自分を写します。通話先の監督のデスクトップPCでは、ブラウザの画面に大きくホッケーの試合と、ピクチャーインピクチャーでスカウトの顔、ウェブカメラの自分の顔が映っています。

複数人でのビデオ会議

中央サーバーなしで複数人同時のビデオ会議を行えます。それぞれのブラウザは他の全参加者のブラウザとセッションが確立されており、お互いに動画と音声のストリームを送受信します。

複数人でのビデオ会議を成立させるためには、音声レベルがまちまちでは困りますので、音量のバランスを調整するミキシング機能が必要です。また、誰が話をしているのかを把握するのが難しくなるため、音量を測定し、今大きな声量で話をしている人の動画を大きく表示する機能や、音声の左右のバランスを調整して音位と映像が近くなる機能が実現されています。

複数人でのボイスチャット付きのオンラインゲーム

複数人のビデオ会議から映像がなくなりますが、代わりにゲームデータをブラウザ間で送信します。ゲームデータの優先度は音声よりも上です。ビデオ会議とは異なり、音声の音位は映像の位置ではなく、ゲームスクリーン内の他のプレイヤーの操作する戦車の位置で決まります。音声と、ゲーム内の

SEをミックスして再生できます。

7.5.2 WebRTCのユースケース（2）

P2P以外のユースケースです。MCU（Multipoint Control Unit）、SFU（Selective Forwarding Unit）という2つの方式があります。前者はサーバー側で各クライアントの映像を合成して配信するケースで、後者はサーバーが各クライアントの映像を中継するハブ型のネットワークを構成します。前者はサーバー負荷が極めて高いという問題があり、現在は後者のSFUが優勢のようです。日本でも、時雨堂がSoraというSFUのパッケージをリリースしています[†28]。

カスタマーセンター

クライアントはブラウザ対ブラウザではなく、サービスを行っているウェブサーバーとセッションを確立して通話します。

IP電話端末

ウェブブラウザを使って一般の電話番号に電話をかけることのできるウェブサービスが実現できます。実際の電話網との接続はサーバーが行いますが、ユーザーはブラウザを使ってそのサービスを利用します。

よくあるカスタマーサポートでは「料金に対するお問い合わせには1を押してください」という音声自動応答装置が使われます。ウェブサービス上からこの装置へ応答するためのDTMF信号が発信できる必要があります。

中央サーバーを使ったビデオ会議システム

中央サーバー経由で通信を行います。P2Pでは多対多になるとセッション数が一気に増えます。中央サーバー経由になれば各ブラウザの送信先は減るのでセッション数は減らせます。

このシステムではオーディオはサーバーでミックスされ、ひとつの音声ソースとして配信されます。

動画はひとつの高解像度と多数の低解像度の両方がサーバーから配信されてきます。高解像度の映像は発言者の行動によって選択されます。送信側は可能であれば高解像度、低解像度の両方を送信します。あるいは高解像度映像だけをアップロードし、サーバー側で小さいサイズの映像にトランスコードして2つのバージョンを作ります。これはSimulcastと呼ばれます。これはP2P配信でも行えます。両方のストリームを送信し、モバイル端末は低解像度の方を、デスクトップは高解像度の情報を受け取ります。

† 28 https://sora.shiguredo.jp/

7.5.3 RFC以外のユースケース

RFCには書かれていませんが、1方向のストリームも使えますので、1対多のリアルタイム映像配信にも使えます。カンファレンスのライブ配信は次章で紹介する各種ストリーミング手法が現在の主流です。また、事前に作成していたコンテンツの配信も同様です。

ライブ配信におけるWebRTCのメリットはリアルタイム性が高いことです。次章で紹介するHTTPによるストリーミング技術では10数秒から30秒程度の遅延が避けられませんが、WebRTCでは数秒の遅延に抑えることができます。

ライブ配信の用途ではP2PのCDNサービスもいくつか出てきています[†29]。CDNといっても静的ファイル配信ではなく、同じ動画を閲覧している他のユーザーとWebRTCのセッションを接続し、中央サーバーではなくそのユーザーから配信を受けることで、サーバーの負荷を下げます。

ゲームでの活用も活発化しています。ゲームエンジンの大手であるUnreal Engine 4は2018年11月[†30]に、Unity3Dは2019年9月[†31]に、それぞれPixel streaming、Unity Render Streamingと呼ばれる機能を追加しました。これは強力なGPUを持つコンピュータ上でゲームの画面を生成し、それをWebRTCとして配信するものです。再生デバイスの性能が低くても（遅延以外は）高性能GPUと同等のクオリティのゲームが実現できます。Google製のクラウドゲーミング環境のStadiaも裏ではWebRTCを活用しているようです。

7.5.4 RTCPeerConnection

ここからWebRTCの技術要素について説明していきますが、ブラウザから使われるAPIという観点でまとめます。

RTCPeerConnection

通信経路の確保とメディアチャンネルのオープン

mediaDevices.getUserMedia

カメラ、マイクと動画・音声のハンドリング

RTCDataChannel

データチャンネルの通信

WebRTCのベースになっているのはIP電話です。IP電話で使われている技術をひとつにまとめ、

† 29 https://www.mist-t.co.jp/mistcdn
† 30 https://www.unrealengine.com/ja/blog/pixel-streaming-delivering-high-quality-ue4-content-to-any-device-anywhere
† 31 https://blogs.unity3d.com/2019/09/17/stream-high-quality-real-time-graphics-through-your-browser-with-our-new-webrtc-framework/

JavaScriptのAPIを定めたものです。NAT超えなどの基礎をなすプロトコルはどれも古くからあるものです。

SDP（Session Description Protocol: セッション記述プロトコル）

SDPはP2Pのネゴシエーション時に互いのIPアドレスとポート、双方で利用できるオーディオや動画のコーデック情報を共有するためのものです。

WebRTCでは「自分が使えるコーデック情報やIPアドレスなどをSDPというプロトコルのフォーマットで記述して相手に渡し、相手からも互いに利用可能なコーデック情報のレスポンスをもらう」という手続きを定めていますが、「相手に渡す」手段を定めていません。ウェブサービスでHTTPのAPIを経由してもいいですし、WebSocketなどを使っても構いません。そのため、サービスの形態にあわせて柔軟に利用できます。がねこまさし氏の「手動でWebRTCの通信をつなげよう ー WebRTC入門2016」[†32]を見ると、いちばんシンプルなコピー＆ペーストでSDPを交換する方法が載っています。どのような内容がやりとりされているかは、このページのサンプルを動かしてみると分かりやすいでしょう。

ICE（Interactive Connectivity Establishment）

ICEはNATを超えてP2P通信の接続を確立する手法です。STUNとTURNのどちらかのサーバーを利用します。

ルーター内などグローバルのネットワークから隔離された環境で、各コンピューターはローカルアドレスしか知りません。ルーターの外部にアクセスすると、NATが外部のコンピュータと内部のコンピュータのアドレスとポートのマッピングをそれぞれ作成します。これは通信のレスポンスを正しく受け取るための仕組みですが、これを利用して外から中への通信経路を確保します。ローカルコンピューターがこの目印の代わりに、パケットを送りつける先がSTUNサーバーです。STUNサーバーはリクエストがあったグローバルのIPアドレスとポートを、NAT内のコンピューターに返送します。この2つのステップで通信経路の確保と、外部から通信可能なアドレスとポートを取得できます。このアドレスとポートを相手に伝えることで、NAT内からのP2P接続が可能になります。

STUNサーバーで通信ができなかった時には、主にTCPなどにフォールバックしますが、その時にはTURNサーバーが通信の中継を行います。この場合はWebSocketと似た通信形態になります。

Voluntas氏の「WebRTC とはなんだったのか」[†33]によると、STUNはNATを超えるために基本的に必要、TURNはなくてもなんとかなることが多いが、あれば接続の成功率が上がると書かれています。80%はなくてもつながり、TURN-UDPがあれば90%、TURN-TCPとTURN-TLSがあれば100%つ

† 32 https://html5experts.jp/mganeko/19814/
† 33 https://gist.github.com/voluntas/67e5a26915751226fdcf

ながるとあります。

先にSDPの説明をしましたが、SDPに記載するIPアドレスを取得するにはSTUNやTURNが必要です。このICEのプロセスの内部でSDPが使われています。

7.5.5 メディアチャンネルと getUserMedia

相手との通信が確立したら、実際に通信を行います。音声やビデオを取り扱うのがメディアチャンネルです。

navigator.mediaDevices.getUserMedia()

ビデオ会議に使うウェブカメラやオーディオデバイスの、設定と取得を行うブラウザAPIです。カメラの場合は解像度、フレームレート、フロントカメラ／リアカメラなどを指定することもできます。取得されたものはストリームという名前で呼ばれます。

このAPIはWebRTCとセットで使うことを考慮して作られたAPIですが、単独でも使え、HTMLの<video>タグ上にカメラの映像を表示できます。<video>タグの内容はcanvasに貼り付けて画像ファイルとして保存できます。スクリーンキャプチャAPIでは、API名が多少異なります（navigator.mediaDevices.getDisplayMedia）が、ストリームとして取得でき、同じように配信できます。

オーディオはWeb Audio API[†34]を利用することで高度な制御やモニタリングが行えます。ステレオの音位の指定（StereoPannerNode）やミキシング（ChannelMergerNode）、ボリューム調整（GainNode）などがあります。Audio Workerを利用すると音声信号をJavaScriptで分析できるようになるため、ユースケースにあった、喋っている人の検出が可能です。

DTLS

WebRTCのデータ通信には、メディアチャンネルとデータチャンネルの2つがあります。どちらも、DTLS上の通信となっています。DTLSは「D:データグラム」のTLSで、RFC4347で定義されています。データグラムはUDPと同じなので、暗号化したUDPということになります。HTTPのベースになっているTCPは、シーケンス番号等を持ち、パケットの順序が入れ替わったりパケットロスをしたときも整列したり再送をリクエストして正しいデータを受信しようとします。UDPそのものは再送も整列もしないため信頼性に劣る代わりに高速になっています。

テレビ電話や音声通話で再送処理が発生すると、期待するフレームが届くまで、映像や音声が停止することになります。そして、それが到着してから等速で再生されます。人間はそのような遅延に敏感ですし、例えば100mSの遅延が10回発生すると、累積されてお互いの会話が1秒の遅延が発生することになります。これらのユースケースでは再送を依頼するのではなく、通信ミスを握りつぶして遅延

† 34 https://www.w3.org/TR/webaudio/

が伸びないようにした方が使いやすいシステムとなります。動画のストリーム配信では多少停止してもフレームを飛ばさない方が好まれるでしょう。WebRTCがわざわざ他のHTTP通信とは毛色の違う難易度の高い実装を行なっているのは、これらのアプリケーションにおいて大切な「遅くならない」というニーズを実現するためと言えます。

WebRTCメディアチャンネル

ひとつが音声・動画を伝送する**WebRTCメディアチャンネル**です。WebRTCメディアチャンネルはDTLS上で**SRTP**というプロトコルを使っています。TLSのところでも軽く紹介しましたが、TLS上のプロトコルは名前の先頭にSが付くものが多数あります。SRTPもその1つで「S:セキュアな」「RTP:リアルタイムプロトコル」です。音声や動画のストリーミングで情報が入れ替わってしまうのは問題なので、RTPは順序の整列機能だけをUDPに追加しています。再送処理はストリーミングでは負荷が大きいので保証しません。音声や動画では音飛びやコマ落ちとして処理されます。

`RTCPeerConnection`の接続が確立した時点で、メディアチャンネルの通信の準備はできています。

7.5.6 RTCDataChannel

データチャンネルは音声以外のデータをP2Pで送受信するためのものです。ユースケースにはファイルをチャットの相手に送信したり、通信対戦のゲームの操作情報を共有するのに使用していました。

SCTP（Stream Control Transmission Protocol）

データチャンネルはDTLSの上で**SCTP**というプロトコルを載せています。データのユースケースは映像・音声よりも多少幅があります。データファイルのデータが途中で消失してしまったり、一部欠落すると困ることもあれば、それより応答性が損なわれる方が問題になることもあります。

そこでSCTPではパフォーマンスとトレードオフで信頼性を設定できるようになっています。UDP側によせるか、TCP側によせるか、選べるようになっています。

表7-5: 各プロトコルの通信特性（『ハイパフォーマンス ブラウザネットワーキング』より引用）

	TCP	UDP	SCTP
信頼性	到達保証あり	到達保証なし	変更可能
配送順序	順序付けあり	順序付けなし	変更可能
転送方式	バイト指向	メッセージ指向	メッセージ指向
流量制御	あり	なし	あり
輻輳制御	あり	なし	あり

データチャンネルの初期化はリスト7–5のように行います。2つ目の引数が特性を制御するパラメータです。省略時はTCP同等の配送順序保証の高信頼性モードになります。このサンプルのコードは順序を保証せず、再送もしないというUDP互換のモードを設定しています。`maxRetransmitTime`か

maxRetransmitsのどちらかを設定し、再送方式を手動にすると非信頼モードになります。

リスト7-5:データチャンネルの初期化

```
var connection = new RTCPeerConnection();
var dataChannel =
  connection.createDataChannel("data channel", {
     ordered: false,       // 順序保証
     maxRetransmitTime: 0, // 再送しつづける期間
     maxRetransmits: 0     // 再送回数
  });
```

ファイル送信などのユースケースであれば、デフォルトの高信頼性モードが適しています。例えば、画像ファイルやExcelファイルのデータの一部が欠けたり順序が入れ替わってしまったら、ファイルとして用をなさないからです。ゲームのステータスで、UDPの1パケットに収まるサイズであれば、少なくとも順序保証は必要ありません。

7.6 HTTPウェブプッシュ

HTTPウェブプッシュは、ウェブサイトにスマートフォンアプリケーションのような通知機能を提供する仕組みです。通信プロトコルは**RFC8030**で規格化されています。また、JavaScript APIはW3Cで定められています[†35]。本書の執筆時点ではChromeとFireFoxが対応しています。

本章では、「クライアントからリクエストを送り、サーバーからレスポンスを受け取る」というHTTP/1.1までのフローとは異なるプロトコルをいくつも紹介しました。サーバー側からイベントを通知したり、双方向通信、P2Pなどです。ただ、どれも制約としてブラウザがアクティブになっていることが前提としてあります。HTTPウェブプッシュはその機能の特性上、ブラウザがその時点で起動していなかったり、オフラインでも、ユーザーに通知を送れる必要があります。本章で紹介したどのプロトコルよりも特殊なプロトコルです。

ブラウザがないのになぜ通信できるのか、というのがこの機能の不思議なポイントです。秘密はService Workerにあります。

Service Workerはフロントエンドのブラウザの HTML表示機能からすると、ウェブサーバーとフロントエンドの中間にある、プロキシサーバーのような存在です。ただし、Service Workerは常に起動し続けているわけではなく、addEventListener()イベントハンドラを登録しておくと、必要なときだけ起動して処理を行います。現在は表7–6のようなイベントに対応しています。

† 35 https://w3c.github.io/push-api/

表 7-6: Service Worker

呼ばれるタイミング	イベント名
インストール時	activate
フロントエンドからの postMessage()	message
フロントエンドがサーバーにアクセス	fetch
プッシュ通知受信	push

ウェブプッシュは現状プッシュサービスと連携することで実現しています。プッシュサービスは Chrome の場合は Google の通知サービス、Firefox の場合は Mozilla 社の通知サービスを利用します。それぞれのサービスでサービスの ID と、送信用の鍵を取得します。受信には ID が、送信には ID と鍵が必要です。

プッシュには 2 種類のユーザーがいます。片方がプッシュを受けるユーザー、もう片方がプッシュを送信するアプリケーションサーバーです。

ユーザーがプッシュを受けるにはまずはブラウザを起動するときにプッシュサービスを利用する登録をします。プッシュ通知はオプトインです。デフォルトでは有効にはなっておらず、ユーザーに許可を取ります。ユーザーの許可後、プッシュサービスに登録します。有効になると Service Worker で登録した push イベントで通知が受け取れるようになります。

アプリケーションサーバーからプッシュ通知を送るにはプッシュサービスの API を利用します。プッシュサービスは今のところブラウザごとに用意されたサービスです。ブラウザは通知を受信するかどうか、ユーザーに確認を取ります。プッシュ通知はオプトイン機能となっています。ユーザーが承認するときに、プッシュサービスに登録します。鍵もこのときに作られます。この鍵情報はプッシュサービスがブラウザを特定するのに使います。

次にブラウザはプッシュ送信に必要な鍵をアプリケーションサーバーに送付します。これにより、アプリケーションサーバーは特定のブラウザに対して送信ができるようになります。

サーバーが通知を行う時は、このブラウザから送られてきた鍵を使い、Push Service にリクエストを送ります。

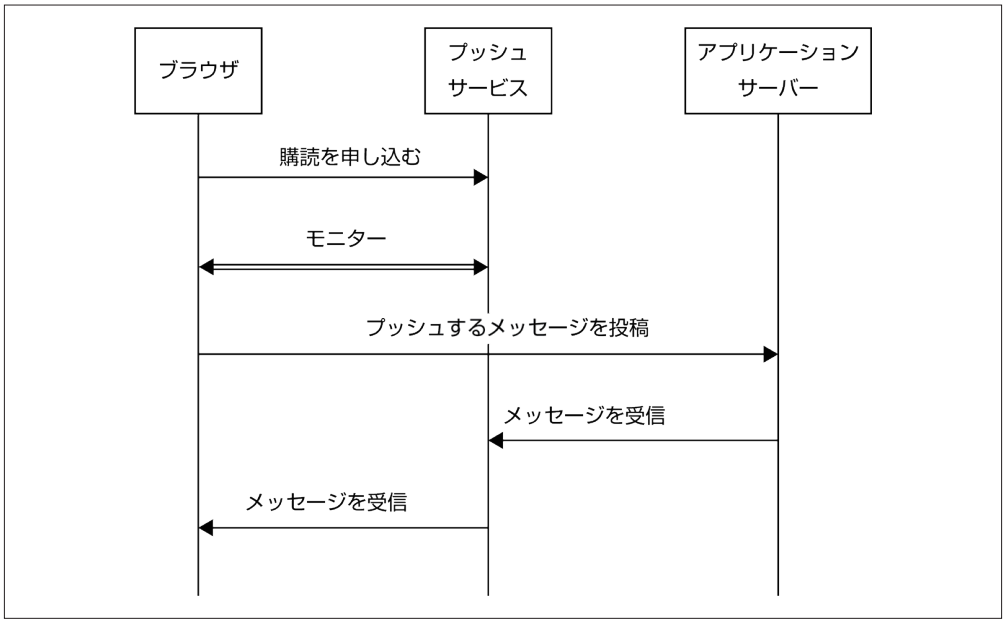


図7-6:HTTP ウェブプッシュの状態遷移

このプッシュサーバーとの通信はHTTPで行う仕様になっています。説明はRFCから引用していますが、実際にこの通りの実装はされていません。

Chromeが利用するプッシュサービスはGoogle Cloud Messaging（GCM）やFirebase Cloud Messaging（FCM）を基盤としています。そのため、そちらでのアプリケーション登録などが必要で、アプリケーションごとに専用の鍵やURLが用意されます。

Firefoxが使っているautopushは、プッシュサービスとの通信にRFCに書かれているHTTP通信ではなく、WebSocketを使っています[†36]。この辺りはブラウザベンダーが各ブラウザ向けに独自にプッシュサービスを用意している状況なので、ブラウザ側のインタフェースは標準に従っている必要はない、という判断がされているのでしょう。本章ですでに説明しましたが、WebSocketはリアルタイム性の高い双方向通信を実現できるため、プッシュ通知の応答速度も上がりユーザーの利益となります。また、autopushはGCN/FCM、Appleのデバイス向けのApple Push Notification Service（APNS）へのブリッジにも対応しています。

本書の説明は特定のサービス向けではなくRFCに準拠しますが、実際のサービスに使う場合はそれぞれ固有の設定や手順に従う必要があります。おおまかな動作原理の紹介としてご理解ください。

† 36 http://autopush.readthedocs.io/en/latest/architecture.html

7.6.1 ブラウザがプッシュサービスに購読を申し込む

最初はブラウザがプッシュサービスに購読を申し込むところから始まります。

ブラウザからプッシュサービスにリクエスト

```
POST /subscribe HTTP/1.1
Host: push.example.net
```

レスポンスは201 Createdで返すことになっています。

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
        rel="urn:ietf:params:push"
Link: </subscription-set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
        rel="urn:ietf:params:push:set"
Location: https://push.example.net/subscription/LBhhw0OohO-Wl4Oi971UG
```

7.6.2 アプリケーションサーバーがプッシュサービスにメッセージを投稿

```
POST /push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
TTL: 15
Content-Type: text/plain;charset=utf8
Content-Length: 36

iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB


HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:55 GMT
Location: https://push.example.net/message/qDIYHNcfAIPP_5ITvURr-d6BGt
```

201のレスポンスは成功を意味しますが、これはサービスが受理したというだけで、クライアントに送付されたかどうかは関係ありません。リクエストにPrefer: respond-asyncヘッダーを付与すると、クライアントの通知が成功してからレスポンスが返るようになります。この時はステータスが202 Acccepted となります。

7.6.3 ブラウザがプッシュメッセージを受信

プッシュメッセージの受信にはHTTP/2が利用されます。通知情報を直接リクエストして取得することはありません。リクエストに対して応答はせず、HTTPのサーバープッシュを使って通知を返します。

```
HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :method        = GET
  :path          = /subscription/LBhhw0OohO-Wl4Oi971UG
```

これに対するサーバーのレスポンスにボディはなく、新しい4番のストリームを使って通信を行う宣言だけが返ってきます。偶数番のストリームはサーバーからクライアントへの通信用として予約されていましたね。

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
  :method        = GET
  :path          = /message/qDIYHNcfAIPP_5ITvURr-d6BGt
  :authority     = push.example.net
```

その後のレスポンスで受信します。サーバープッシュで、アプリケーションサーバーが送信したメッセージが送信されてきます。

```
HEADERS      [stream 4] +END_HEADERS
  :status        = 200
  date           = Thu, 11 Dec 2014 23:56:56 GMT
  last-modified  = Thu, 11 Dec 2014 23:56:55 GMT
  cache-control  = private
  link           = </push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
                   rel="urn:ietf:params:push"
  content-type   = text/plain;charset=utf8
  content-length = 36

DATA         [stream 4] +END_STREAM
  iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB

HEADERS      [stream 7] +END_STREAM +END_HEADERS
  :status        = 200
```

7.6.4 緊急度の設定

Urgencyヘッダーをリクエスト時に付与すると、不必要なメッセージをフィルタリングできます。メッセージ送信、および受信のときに付与します。表7-7に設定可能な値を示します。省略時はnormalになります。

表7-7: プッシュメッセージの緊急度

緊急度	デバイスの状態	用途
very-low	電源がつながっていてWiFi接続あり	広告
low	電源がつながっているか、WiFi接続あり	話題の更新
normal	電源がつながっておらず、WiFi接続もない	チャットやカレンダーのメッセージ
high	バッテリー残量が少ない	電話の着信、もしくは時間厳守の通知

7.7 本章のまとめ

ウェブがインフラとして使われるようになってから、これまでのウェブではできなかったような多様

かつ高度なリクエストに答える機能追加が「HTML5以降」に行われました。この中には、HTML単体では実現できず、Flashなどのブラウザプラグインで行っていた機能も標準化されて取り込まれました。その結果、ウェブの体験はさらにリッチなものとなり、その副作用として通信部分の負荷が高くなりました。また、モバイル端末の利用がデスクトップOSを超え、回線も3G/4Gなどと Wifiが頻繁に切り替わるという使われ方もされるようになりました。HTTP/2やHTTP/3が生み出されたのはこのようなところが背景にあります。

- HTTP/2は増え続けるコンテンツに対処できるよう並列数を増やしたり、今まで圧縮していなかったヘッダーを圧縮してサイズを減らしたり、優先順位が設定できるようになりました。またコンテンツを先行して送るサーバープッシュなど、HTTP/1系では対処できなかった機能が大量に盛り込まれています
- HTTP/3は今のウェブをさらに効率化させるためのものです。HTTP/2とアプリケーションから見た機能的な差はありませんが、接続時間のさらなる短縮や高速化、モバイル端末への最適化が行われています

他の章と異なり、本章のHTTP/3の説明には、まだ策定中の内容も含まれています。それ以外は現在使われているものに限定して説明してきましたが、複数のリソースのコンテンツとヘッダーを一緒にパッケージングして配信するWebPackaging、他のサイトから別のサイトのコンテンツを返すことを実現するSigned Exchangeなどもあれば、HTTP/3のバックエンドのQUICを汎用化して、WebRTCのバックエンドにも使おうとするWebTransportなど、さまざまな規格が生み出されようとしています。これからもウェブの発展により、ワクワクさせられるでしょう。