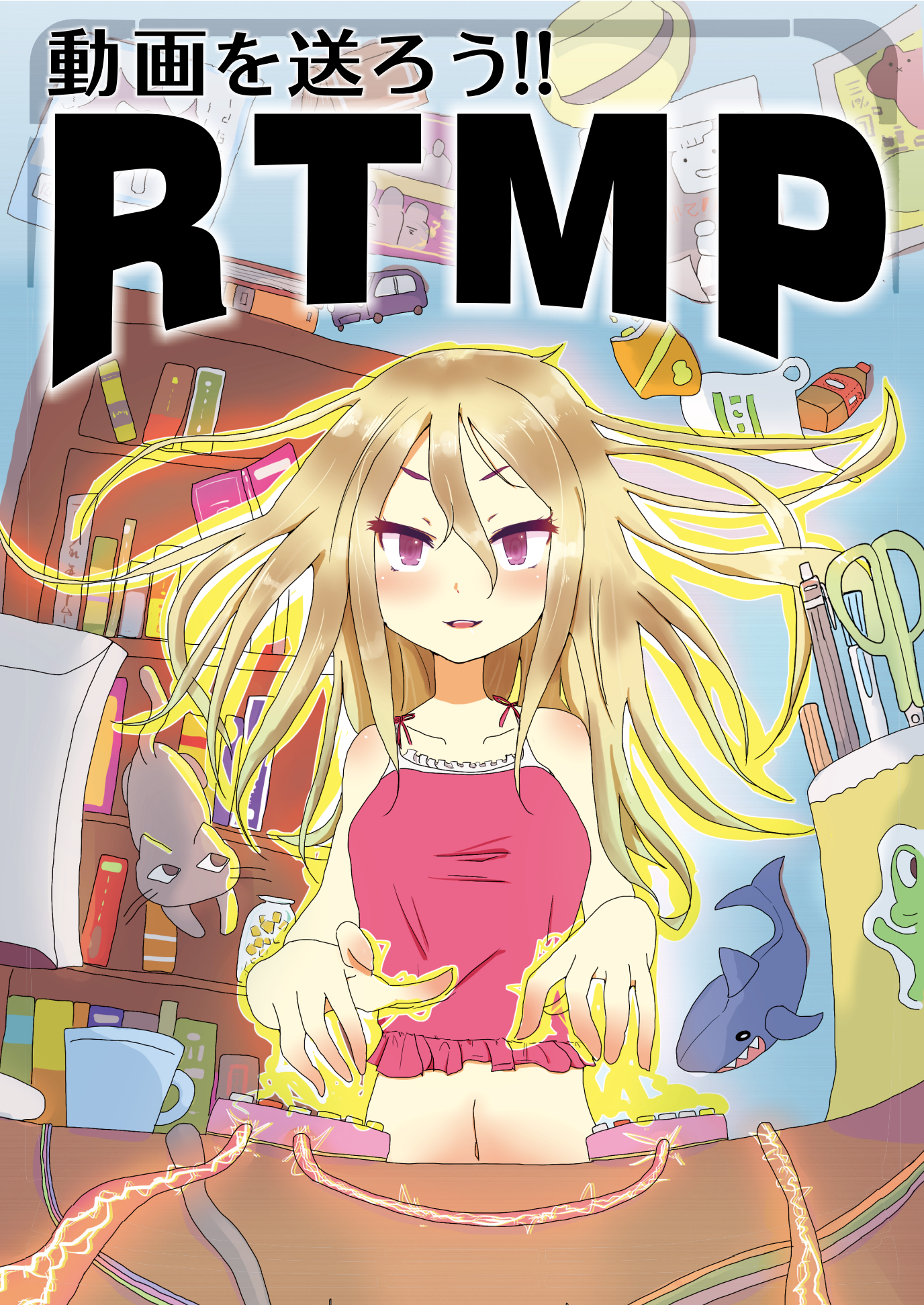

動画を送ろう!!
RTMP

動画を送ろう!!

RTMP

2019 あれくま

目次

1. はじめに 6
2. RTMP とは 7
   2.1. RTMP で出来ること・出来ないこと 7
   2.2. 実装について 9
3. RTMP 概観 10
   3.1. TCP 10
   3.2. RTMP Chunk Stream Protocol 10
   3.3. RTMP Message 11
4. RTMP Chunk Stream 14
   4.1. バイナリフォーマットについて 14
   4.2. TCP での接続 14
   4.3. ハンドシェイク 14
   4.4. チャンクの送受信 18
   4.5. RTMP Control Message 25
   4.6. RTMP Chunk Stream のまとめ 28
5. RTMP Message 31
   5.1. RTMP Message Format 31
   5.2. RTMP Message の種類 31
   5.3. User Control Message (タイプ ID:4) 32
   5.4. Data Message (タイプ ID:15 または 18) 37
   5.5. Audio Message (タイプ ID:8) 39
   5.6. Video Message (タイプ ID:9) 41
   5.7. Shared Object Message (タイプ ID:16 または 19) 43
   5.8. Aggregate Message (タイプ ID:22) 43
   5.9. Command Message (タイプ ID:17 または 20) 44
   5.10. NetConnection コマンドと NetStream コマンド 45
   5.11. connect コマンド (NetConnection) 46
   5.12. createStream コマンド (NetConnection) 50
   5.13. deleteStream コマンド (NetConnection) 51
   5.14. close コマンド (NetConnection) 52
   5.15. その他のコマンド (NetConnection) 52
   5.16. onStatus コマンド (NetStream) 54
   5.17. play コマンド (NetStream) 55
   5.18. play2 コマンド (NetStream) 56
   5.19. pause コマンド (NetStream) 58
   5.20. seek コマンド (NetStream) 59

5.21. receiveAudio コマンド (NetStream) 59
5.22. receiveVideo コマンド (NetStream) 60
5.23. publish コマンド (NetStream) 60
6. AMF0/3 62
6.1. AMF0 62
6.2. AMF3 66
6.3. RTMP Message での型の扱いについて 72
7. RTMP で通信しよう 74
7.1. 共通処理 74
7.2. 切断処理 76
7.3. クライアントからサーバーへ動画配信したい 76
7.4. 動画の視聴をしたい (サーバーからクライアントへの動画の配信をしたい)80
8. FLV ファイル 82
8.1. FLV ファイルの構造 82
8.2. FLV ファイルまとめ 84
9. 参考文献 85
9.1. ドキュメント類 85
9.2. ツール類 87

1. はじめに

この本はいまさらながら RTMP を解説する本です。

一般ユーザーによる動画の生配信が一般的になり、動画配信サービスも沢山ある今日このごろ。

動画配信の視聴と言えばちょっと前までは Flash がよく使われておりましたが、今となってはもう過去の物となりはて、HTML5(というか Web ブラウザだけ) で視聴ができる便利な世の中になりました。

現在動画を視聴しようとすると HTTP ベースのストリーミングプロトコル (HTTP Live Streaming や MPEG-DASH) がよく使われるということはご存知の方も多いでしょう。

逆に視聴でなく自分で配信したい！と思ったときにどんなプロトコルで送ってるのかな？と不思議に思った人も少なくないはずです。いや実は少ないと思っていますが、少なくないことにしておきます。 不思議に思わないまでも、HTTP で視聴できるんだから配信も逆にサーバーに HTTP で送ってるだけでしょー？と思っている人も居るんじゃないでしょうか。

だがしかし！実は配信元からサーバーへの動画送信には HTTP は使われておらず、Flash で使われていたプロトコルの RTMP がデファクトスタンダードになっています。 Flash を使わずとも Adobe Media Server(旧 Flash Media Server) とやりとりするためのプロトコルなので、一応 RTMP は Flash とは独立しており仕様も公開されています。 そんな事情もあり色々な配信用のソフトウェアやハードウェアでも実装され、逆に既存の配信用環境に合わせるようにサーバー側も Adobe Media Server でなくとも RTMP を受信できるよう対応し、というのが続いてデファクトスタンダードになってしまったようです。

というわけで、この本はそんな RTMP について解説する本です。

RTMP 自体はもう改定されそうもないですし、新しいコーデックへの対応もされないため、今後はまた別なプロトコルに置き換わっていくだろうと予想されます。しかしこれを書いている 2019 年初頭の時点ではまだこれといって有力な代替候補が定まっているとは言い難い状況ですので、そろそろ RTMP を解説できる最後の機会かなーと思ってこんな本を書きました。

元々 Flash のプロトコルなのですが、いまとなっては動画配信にしか使われないと思いますし、書いてる自分自身がそんなに Flash に詳しくないこともありまして、動画関係を中心に解説していきます。

2. RTMP とは

RTMP とは！ Real-Time Messaging Protocol の略で、Adobe Flash(旧 Macromedia Flash) と Adobe Media Server(旧 Flash Media Server) の通信プロトコルです。

Flash で使われる ActionScript[1] から簡単に接続でき、動画や音声のやりとりができるため、Flash ベースのインターネットアプリケーション、ビデオ会議や動画配信を作るのに使われました。

メディアサーバーなんぞは他にも沢山ありましたが、RTMP の特徴としては ActionScript のオブジェクトをやりとりできるプロトコルという点があるでしょう。 動画や音声だけでなく ActionScript のデータがいっしょに載せられることで、動画と同期してスクリプトでの動作を行ったりすることもできました。 Flash Media Server はサーバー側アプリケーションも ActionScript で開発するので、クライアントとサーバーでシームレスにオブジェクトをやりとりできると非常に便利です。 動画や音声を扱わない、Flash ベースのオンラインゲームにも使われました。

今となっては Flash と Adobe Media Server でアプリケーションを作ることはもうほとんど無いでしょうし、それで作るにしても RTMFP という後継に近いプロトコル[2]があるので、RTMP が元の用途で使われることは少ないんじゃないかと思います。

しかしながら、動画をやりとりするのに Flash が便利だった時代はそれなりに長く、クライアントからサーバーに動画を送るのに RTMP を採用したソフトウェアやハードウェアがいくつも作られました。それらを使えるようにするため、また他にちょうど良いプロトコルがあったわけでもないという理由もあるためか、もう Flash も Adobe Media Server を使ってなくともクライアントからサーバーに動画 (と音声) を送るプロトコルとして RTMP が採用され、デファクトスタンダードの地位を確立しました。

RTMP は仕様が公開されており[3]、TCP で 1 本つなげばずっとやりとりできる比較的シンプルなプロトコルといったあたりも数多く実装された理由の一つでしょう。

2.1. RTMP で出来ること・出来ないこと

RTMP の機能は次のようなものがあります。

- TCP で接続
- シリアライズされた ActionScript オブジェクトの送受信
  - AMF0
  - AMF3
- 音声の送受信
  - ADPCM

1 Flash で使われるスクリプト言語。だいたい JavaScript みたいなもの。

2 UDP ベースで RTMP と名前は似てるけど中身はだいぶ別物

3 https://www.adobe.com/devnet/rtmp.html ただしいつまであるかはわからない。

    - MP3
    - NellyMoser
    - AAC
    - Speex
- 動画の送受信
    - Sorenson (Flash Video)
    - Flash Video Screen V1・V2
    - On2 (VP6)
    - H.264
- 上記各種データ (オブジェクト・音声・動画) の 1 ストリームへの混載
    - オブジェクトデータも含めてタイムスタンプ付き
- 一つの接続に複数ストリームの重畳
- サーバー・クライアントの双方向通信

オブジェクトも含めてタイムスタンプが付いているので、オブジェクトと動画・音声を同期することができるのは嬉しいですね。

複数ストリームの重畳ができるので、1 接続で複数の動画やオブジェクトをそれぞれ非同期で流したりすることもできます。

もちろんいずれにせよアプリケーションが対応しないと意味ないですけど……。

逆に出来ないことは以下のようなものがあります。

- 暗号化
- P2P 通信
- 新しいコーデックへの対応

TLS などを使わない普通の TCP 通信なので暗号化はされていません。抜けば抜けます。 SSL で暗号化する RTMPS や、もう少し軽い暗号化である RTMPE というプロトコルがあるようですが、詳細なドキュメントがないためわかりません。 まあ少なくとも現在のクライアントからサーバーへの動画配信では使われてはいないようです。

TCP であるため P2P 通信はできません。いや TCP だからというわけではないですが NAT 越えなどを考えると TCP は不利ですし。UDP ベースの RTMFP だと P2P 通信もできるようになっています。

音声や動画のコーデックは独自の値が割り当てられていますので、仕様の改定がない限りは増えませんし、増やすこともできません。最新のコーデックが AAC・Speex・H.264 なので、もっと新しい H.265 や AV1、Opus なんかを使いたくても対応できません。 この点が RTMP は実用的限界でもありますので、万が一バージョンアップでもされない限りは今後は使われなくなることでしょう。

2.2. 実装について

前にも述べた通り、仕様は公開されていますし、それほど複雑なものでもないため仕様をよく読めば誰でも実装することができます。

……と言いたいところだが。

仕様書が公開されているのはいいんですが、あちこち曖昧だったり省略されていたり古かったりして、他のデータの仕様書やドキュメントと突き合わせないとわからないところがいくつもあります。 公式のドキュメントを見て実装しようとすると何度も首をかしげることになるでしょう！ さすが旧 Macromedia だぜ！！

この本ではなんとかそれなりに実装できる程度にはまとめたつもりなので、参考になれば幸いです。

3. RTMP 概観

詳細に入るまえにざっくりと RTMP 全体を見渡してみましょう。

RTMP はいくつかのプロトコル階層によって成り立っています。

1. TCP
2. RTMP Chunk Stream Protocol
3. RTMP Message

建前的にはそれぞれの階層は独立したプロトコルになっています。 実際には密接に関連している部分があるのですが、ある程度独立になってる建前で仕様書が書かれているので逆に読みづらくなっているのが困ったところです。

3.1. TCP

まずベースは TCP です。普通の TCP なので特に説明するところもないです。

TLS/SSL は使わず、素の TCP を使います。 TLS/SSL を使う RTMPS というのもあるらしいですが、詳細はよくわからない (仕様書にはない) ので割愛します。 まあたぶん普通に TLS/SSL の上に乗せただけのものですけど。

ポート番号は 1935 です。もちろんサーバーとクライアント双方が指定すれば他のポートでも使えますが、どのアプリも特にポート番号を指定しなければ標準の 1935 を使うでしょう。

3.2. RTMP Chunk Stream Protocol

ここから RTMP 独自のプロトコルになります。

TCP の上に RTMP Chunk Stream Protocol というバイナリフォーマットで接続を確立します。TCP 接続が確立されると、次にこのレイヤーでハンドシェイクが行われ、さらに上位の通信を行います。

Chunk Stream Protocol という名の通り、このプロトコルではチャンク (= バイナリの塊) 単位でやりとりを行います。

一つのチャンクには次のような情報が含まれます。

- チャンクフォーマット
- チャンクストリーム ID
- タイムスタンプ
- メッセージ長
- メッセージタイプ ID
- メッセージストリーム ID

- ペイロード (メッセージボディ)

Chunk Stream Protocol にはチャンクストリーム ID というのが含まれますが、この ID が異なるチャンクは仮想的に異なる接続として扱われます。 つまり一つの TCP 接続上に ID が異なるチャンクストリームを多重化することで、仮想的に複数の接続を確立することができます。

それと別にメッセージストリーム ID というのも持っており、これはより上位の RTMP Message のストリーム ID で、チャンクストリーム ID とは別物です。混同しないように注意しましょう。

3.2.1. Protocol Control Message

Chunk Stream Protocol でやりとりされるメッセージは基本的にはより上位の RTMP Message になるのですが、一部 Chunk Stream レベルで設定や通知するメッセージがあり、これらは Protocol Control Message と呼ばれます。

Protocol Control Message は接続全体のメッセージとなります。多重化はされずに常にチャンクストリーム ID が 2 番が使われます。

3.3. RTMP Message

RTMP Message はよりアプリケーションに近いメッセージです。

RTMP Chunk Stream Protocol まではアプリケーションに関係なく、RTMP プロトコルであれば必ず決まった動作をするものですが、RTMP Message は主にアプリケーションの責任で送受信するものです。 RTMP Message が正しく送受信できるところまで作れれば、あとはアプリケーションを作るだけと言えるでしょう。やったね！

RTMP Message には以下のような情報で構成されます。

- メッセージ長
- メッセージタイプ ID
- メッセージストリーム ID
- ペイロード (メッセージボディ)

おっとどこかで見たような……。Chunk Stream のチャンクに含まれる情報ですね。

RTMP ではこれらは Chunk Stream のチャンクに載ってきますので、チャンクのやりとりさえできていれば、RTMP Message で独自に規定されるのはメッセージタイプ毎の内容くらいです。

3.3.1. User Control Message

RTMP Message でもアプリケーションが基本的に処理するのは RTMP Command Message なのですが、他に接続制御みたいなのを行う User Control Message というのが存在します。 Chunk Stream Protocol における Protocol Control Message が、RTMP Message レベルでは User Control Message として用意されています。

内容は Protocol Control Message ほど重要なものでもなく、ヒント程度の情報を通知するものです。

3.3.2. RTMP Command Messages

RTMP Command Messages はアプリケーションがなんか処理する必要があるメッセージです。

この中でもいくつかメッセージの種類があります。

- **Command Message**
  サーバー (またはクライアント) への関数呼び出しをリクエストします。
- **Data Message**
  なんらかのデータを送ります。が、通常は動画や音声のメタデータ (タイトル情報とか) を送信するのに使われます。
- **Video Message**
  動画データを送ります。
- **Audio Message**
  音声データを送ります。
- **Shared Object Message**
  ActionScript の共有オブジェクトやそれらに対する変更について送ります。
- **Aggregate Message**
  複数のメッセージをまとめるメッセージです。何に使うんだろう……。

アプリケーションで使うのは基本的に Command Message で、動画の再生開始だとか配信開始だとかをこれで通知しますし、受け取った側はその内容に応じた処理をします。 あとは再生や配信が開始されると Data Message や Video・Audio Message がだらだらと流れます。

Shared Object Message や Aggregate Message は (少なくとも動画配信では) 使いません。

3.3.3. Action Message Format

Command Message では ActionScript の関数 (メソッド) 呼び出しの形でリクエストが送られますが、引数や戻り値も ActionScript のオブジェクトとして送受信されます。

そのため ActionScript のオブジェクトをエンコード (シリアライズ) する必要があるのですが、そのフォーマットが Action Message Format(AMF) です。 AMF は 0 と 3 という 2 つのバージョンがありますが、そのいずれもが RTMP で使えるようになっています。 ちょっと面倒ですがどちらも扱えるように実装しておかないといけません。

ちなみに AMF3 は ActionScript3 でのフォーマットで、AMF0 はそれ以前の ActionScript でのフォーマットのようです。AMF0 の方が当然簡単です。

3.3.4. 動画・音声のフォーマット

Video Message や Audio Message には動画や音声が格納されますが、その内容は仕様書には書かれていません。

Flash で使う動画ファイルフォーマットの仕様書にある、FLV ファイルの仕様を見るとそれらしきものが見つかります[4]。基本的にはコーデック毎の生データが入っていますが、いくらか付加情報があるのでこれを見ないと再生はできないでしょう……。

4　https://www.adobe.com/devnet/f4v.html の FLV and F4V File Format Specification。例によっていつまであるかわからない。

4. RTMP Chunk Stream

ここからは、ついに RTMP の詳細に入っていきます。

なんとなく実装できそうなくらいには解説するつもりですが、コードとかは無いので本気で書く場合には頑張ってください。

4.1. バイナリフォーマットについて

詳しく見ていく前にちょっとバイナリフォーマットについて書いておきましょう。

まず 1 バイトは 8 ビットとします。1 オクテットと書く方が正確でしょうけど面倒なので 1 バイトと書きます。 1 バイトが 8bit じゃない環境は今や見たことないくらい稀ですしね。

複数バイトにまたがる数値がある場合は、主にネットワークバイトオーダー (= ビッグエンディアン) が使われます[5]。主に、ということでたまーにリトルエンディアンなところがありますが、そこはその都度注記します。特に記述が無い限りはビッグエンディアンです。

ビット単位で見ていくこともありますが、その場合は 0 ビット目を MSB(最上位ビット) とします。 1 バイトのデータの 0 ビット目と言えば一番上のビットですし、7 ビット目と言えば一番下のビットになります。

実際に実装しない限りは流しちゃっていいですが、実装する時には気を付けて見てもらえばいいかと思います。

4.2. TCP での接続

TCP については普通なので特に解説しません。TCP でつないでください。ポート番号は 1935 です。

IPv6 でも IPv4 でもプロトコル上は関係ないので、どちらでも安心して使えます。

RTMPS のプロトコルを使う場合は TCP でつないだ上で SSL/TLS を使うと良いようです。ポート番号とかはよくわからないですね……。

4.3. ハンドシェイク

TCP でつながったら、まずは RTMP Chunk Stream のハンドシェイクを行います。 ハンドシェイクとは直訳してしまうと握手ですが、つまりお互いにデータをやりとりしはじめる前の初期通信です。

ハンドシェイクのやり方は仕様書に書いてあるのですが、実はこれは古いハンドシェイクで

5 ネットワークバイトオーダーとかビッグエンディアンとかいまいちわからんという人は調べてみてください。そんなに難しくはないんですが長くなるのでここでは解説しません。

Flash のいつからか (10 くらい？) 新しいハンドシェイクが行われているようです。が、仕様書に書いてないので詳細はわかりません。

とりあえず古い方のハンドシェイクでも問題なくつながるので、まずは仕様書通りにやるのが良いでしょう。

4.3.1. 仕様書に書いてあるハンドシェイク

ハンドシェイクではクライアントとサーバーが互いに 3 つのパケットを送り合います。

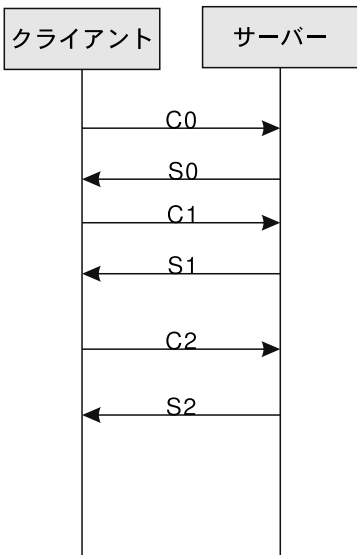


図 1. ハンドシェイクの手順

それぞれ C0、C1、C2 と S0、S1、S2 という名前が付いていて、C0 ～ C2 はクライアントから送るパケット、S0 ～ S2 はサーバーから送るパケットです。C0 ～ 2 の C はクライアント (Client) の頭文字、S0 ～ 2 の S はサーバー (Server) の頭文字ですね。

まずクライアントが C0 パケットを送信します。中身は 1 バイトのバージョン番号が入ります。

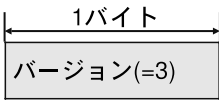


図 2. C0/S0 パケットの構造

バージョン番号は 3 固定です。以前は 0 ～ 2 もあったようですが仕様もないのでわかりませんが、どっちにしろ使われてないので気にしなくて良いでしょう。4 以降は現在無いようです。

サーバーは C0 パケットを受け取って中身が 3 だったら S0 パケットを返します。中身は同じく 1 バイトのバージョン番号で 3 を入れます。

サーバーが S0 パケットを送ったら、次はクライアントから C1 パケット、サーバーから S1 パケットを送り合います。 どちらが先でも大丈夫そうに見えますが、とある事情により C1 が先になるよ

うにしましょう。 クライアントが C1 を送って、サーバーが C1 を受け取ってから S1 を返すべきです。



図 3. C1/S1 パケットの構造

C1/S1 パケットの中身もどちらも同じです。まず 4 バイトのタイムスタンプ、4 バイトの `0`、そして 1528 バイトのランダム値が続きます。

タイムスタンプは、通信開始時点からの経過時間をミリ秒単位で送るのですが、今通信開始するところなので当然は `0` が入ります。他の接続と同期したい時にそっちの通信開始時点からの経過時間を送るという場合があるようですが、まあ使わないでしょうし `0` を入れとけばいいでしょう。

4 バイトの `0` はそのまま 4 バイト分 `0` を入れておきます。

1528 バイトのランダム値はなんでもいいので 1528 バイト分なんか適当な値を入れて送ります。固定値でもかまいません。

C1/S1 パケットを受けとったら、返事として C2/S2 パケットを返します。



図 4. C2/S2 パケットの構造

C2/S2 パケットの中身もどちらも同じです。まず 4 バイトのタイムスタンプ返信、次に 4 バイトの受信時タイムスタンプ、そして 1528 バイトのランダム値返信が続きます。

タイムスタンプ返信と 1528 バイトのランダム値返信は、さっき受け取った C1/S1 に入っていたタイムスタンプ値とランダム値をそのままコピーして入れます。

受信時タイムスタンプには、前の C1/S1 を受け取った時点のタイムスタンプを設定します。 たとえばクライアントが C1 パケットにタイムスタンプ 0 を入れて送信後、3 ミリ秒経って S1 パケットを受け取ったとしましょう。その時 C2 パケットの受信時タイムスタンプには `0+3` で `3` を設定して送ります。 まあ実際のところだいたいで大丈夫です。

で、あとは C2/S2 として返ってきたタイムスタンプとかランダム値が、C1/S1 で送ったのと一致していればハンドシェイク完了、本番の通信に入ることができます。

実際のところ大して何か処理することもなく、固定データを送って、受け取った C1/S1 パケットをそのまま送り返してやるだけでも通ります。それでいいのかっていう気もするけど、楽ちんだね。

まあ一応通信にかかる時間 (RTT) を計測したいという意図はありそうなので、できれば C2/S2 のタイムスタンプくらいはちゃんと設定した方が良いんじゃないかと思います。

4.3.2. 真のハンドシェイク

仕様書に書いてある簡単ハンドシェイクは上記の通りですが、仕様書に書いていない真のハンドシェイクが実は存在します。真のというかたぶん新しいだけなんですけど。

ただ簡単ハンドシェイクでもなんら問題なく動作します。一部を除いて。

H.264(AAC もかも？) の再生をする時だけは簡単ハンドシェイクだと動きません。他のコーデックを使っている場合は特に問題なく動くんですが、H.264 の動画を再生しようとすると、Adobe Media Server に play コマンドを送っても何のエラーも返さないくせに動画データだけ全く送ってこなくなります。 Flash Player でも、新しいハンドシェイクに対応していないサーバーでは Flash Player が送ってきた play コマンドに対して H.264 の動画データを返しても無視されます。 何のエラーも出ないのでさっぱりわかりませんが、実は全く関係なさそうなハンドシェイクの時点で判別されるようです。

一方で再生でなく録画 / 配信する時には関係なさそうです。簡単ハンドシェイクで済ましても、サーバーに送った H.264 のデータは受け付けてくれるように見えましたし、各種エンコーダーも H.264 のデータを送ってくれます。

真のハンドシェイクの方法についてもできればここで書いておきたいところですが、署名なんかの処理があって結構複雑ですし、署名の鍵データも必要になって紙面をとられてしまいます。というか載せていいのかそれ。 Flash Player(や AIR) で RTMP 経由で動画再生したい時くらいにしか使わないですし、今需要があるか怪しいので今回は見送ります。

ちょこっとだけ書いておきますと、新しい形式のハンドシェイクでは、C1/S1 でタイムスタンプの後の常に `0` だったところに `1` が入ってきます。クライアントが送ってきた C1 に `1` が入ってきたら、サーバーもそれに対応した新しい S1 を返さないといけません。 これが C1 を受け取ってから S1 を返すべき事情です。 無視して簡単ハンドシェイクの S1 を返した場合は簡単ハンドシェイクのまま進みます。

詳しく知りたい場合はオープンソースの RTMP 実装でいくつかみつかります。というか仕様はないのでそれしかないです。でもこれ最初に書いた人はどうやって解析したんだろう……。

4.3.3. ハンドシェイクのエラー処理

ハンドシェイクに失敗した場合はどうしたら良いでしょう。たとえば期待したバージョン番号が違うとか。

失敗した場合は特に規定がないので、そのまま接続を切るしかなさそうです。ぶち切りましょう。通信できないんだもん仕方ないよね。

4.4. チャンクの送受信

ハンドシェイクが完了したらチャンクの送受信ができるようになります。

チャンクは次の図のような構成になっています。

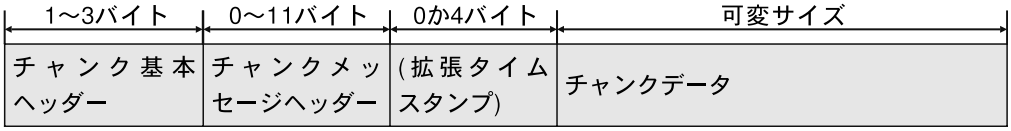


図 5. チャンクの構造

4.4.1. チャンクの基本ヘッダー

1 つのチャンクはまず 1 〜 3 バイトのチャンク基本ヘッダー (Chunk basic header) から始まります。

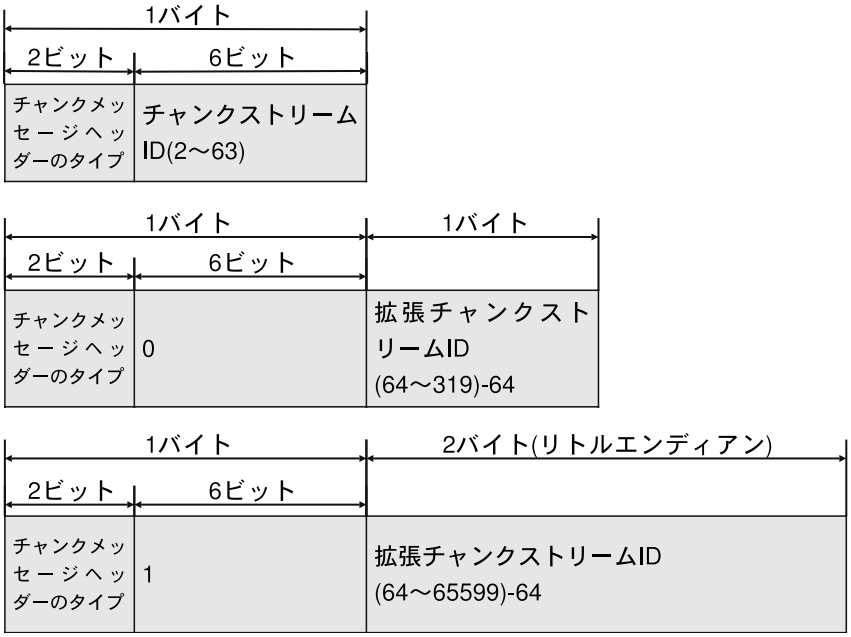


図 6. チャンク基本ヘッダーの構造

最初の 1 バイトのうち、上位 2 ビット (ビット 0 〜 1) には後に続くチャンクメッセージヘッダー (Chunk message header) のフォーマットタイプが入っています。 下位 6 ビット (ビット 2 〜 7) にはチャンクストリーム ID が入っています。

チャンクストリーム ID は 6 ビット分あるので 0 〜 63 まで表すことができますが、**0** 番と **1** 番はより大きな ID を使いたい時のための拡張用として予約されています。

チャンクストリーム ID 部分に **0** が入っていたら 1 バイト、**1** が入っていたら 2 バイトの拡張チャンクストリーム ID が続きます。 拡張チャンクストリーム ID には **0** 〜 **255** か **0** 〜 **65535** の値が入っていますが、これに **64** を足したものが実際のチャンクストリーム ID となります (0 〜 63 だったら

1 バイトのヘッダーで足りるので)。つまり 64 ～ 319 番または 64 ～ 65599 番までのチャンクストリーム ID はこの拡張チャンクストリーム ID で表されます。

2 バイトの拡張チャンクストリーム ID を使う場合、拡張部分はリトルエンディアンになっていることに気をつけてください。

このチャンクストリーム ID というのは自由に決められるのですが (0 ～ 2 を除く)、63 より大きなチャンクストリーム ID を使うとチャンク毎のデータが大きくなってしまいます。 基本的には 3 ～ 63 の範囲のチャンクストリーム ID を使い、拡張はよほどのことがない限り使わないと思います。

4.4.2. チャンクメッセージヘッダー

チャンク基本ヘッダー (と拡張チャンクストリーム ID) の後には、チャンクメッセージヘッダーが続きます。 このヘッダーのフォーマットはチャンク基本ヘッダーの上位 2 ビットの値 (チャンクメッセージヘッダーのタイプ) によって変わります。

チャンクメッセージヘッダーには、タイムスタンプ・メッセージ長・メッセージタイプ ID・メッセージストリーム ID が含まれるのですが、メッセージ毎にこれを送っていると送りたいメッセージボディのデータに対してヘッダ部分が大きくて通信効率が悪くなってしまいます。そのため、前のチャンクと同じ値は省略してヘッダサイズを小さくできるフォーマットが用意されています。

4.4.3. タイプ 0

チャンク基本ヘッダーの上位 2 ビットの値が 0 の場合はタイプ 0 と呼ばれるチャンクメッセージヘッダーが入ります。

タイプ 0 のチャンクメッセージヘッダは全部入りの省略無しパターンです。これが基本形です。



図 7. チャンクメッセージヘッダー タイプ 0 の構造

タイムスタンプには、チャンクの絶対タイムスタンプ (通信開始時からの経過時間をミリ秒単位) を指定します。 3 バイトありますが、もし足りなくなったら 0xFFFFFF(16777215) を入れておき、後に続く 4 バイトの拡張タイムスタンプを使用します。

メッセージ長には後に続くメッセージデータのバイト数が入ります。3 バイトあるので最大は 16777215 です。それより長いメッセージは作れません。

メッセージタイプ ID には、1 バイトのメッセージタイプが入ります。メッセージタイプ ID によってメッセージの中身が動画か音声かはたまた各種コマンドかといった判別をします。どんな種類があるかはまたあとで見ていきましょう。

メッセージストリーム ID には 4 バイトでそのままメッセージストリーム ID が入ります。ここ

だけなぜかリトルエンディアンで入っていますので注意してください。 4 バイトありますが有効なのは 24bit 分で、(リトルエンディアンなので) 最後の 8bit は常に 0 が入ります。 ちなみに、メッセージストリーム ID とチャンクストリーム ID は別物です。ストリーム ID て前にあったような？と混乱しますが気をつけてください。

4.4.4. タイプ 1

基本ヘッダの上位 2 ビットの値が 1 の場合はタイプ 1 と呼ばれるチャンクメッセージヘッダが入ります。

タイプ 1 のチャンクメッセージヘッダは、メッセージストリーム ID が省略されタイムスタンプが差分になったパターンです。

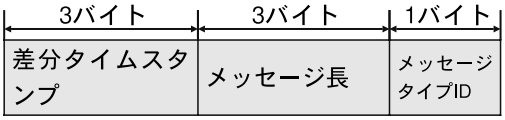


図 8. チャンクメッセージヘッダー タイプ 1 の構造

タイプ 0 で絶対タイムスタンプが入っていたところは、差分タイムスタンプが入っています。差分タイムスタンプには、「前のチャンクからの」経過時間がミリ秒単位で入ります。 通信開始時からの経過時間ではないところが重要です。

長い時間通信していると通信開始時からの経過時間は 3 バイトで収まらなくなり、拡張タイムスタンプを使いまくることになってしまいますが、前のチャンクからの経過時間であればよほどのことがない限り 3 バイトで収まるはずです。これで比較的少ないバイト数で時間経過を表現することができます。

メッセージ長とメッセージタイプ ID の内容はタイプ 0 と同じです。

メッセージストリーム ID は省略されており、最後のタイプ 0 チャンクで指定されたものをそのまま使用します。

4.4.5. タイプ 2

基本ヘッダの上位 2 ビットの値が 2 の場合はタイプ 2 と呼ばれるチャンクメッセージヘッダが入ります。

タイプ 2 のチャンクメッセージヘッダは、差分タイムスタンプだけが残ったパターンです。

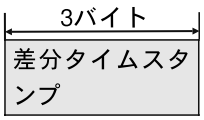


図 9. チャンクメッセージヘッダー タイプ 2 の構造

差分タイムスタンプの意味はタイプ 1 のものと同じです。

メッセージ長とメッセージタイプ ID も省略されており、最後のタイプ 0 またはタイプ 1 のチャンクで指定されたものをそのまま使用します。

メッセージストリーム ID も省略されており、最後のタイプ 0 チャンクで指定されたものをそのまま使用します。

特に音声データなどは固定長のデータを連続して送ることが多くなりますので、メッセージタイプ ID(動画だとか音声だとか) は連続していれば変わりませんし、固定長であればメッセージ長も変わりません。ヘッダ内で変わるのは時刻だけ、という場合にタイプ 2 のチャンクメッセージヘッダが使われます。

4.4.6. タイプ 3

基本ヘッダの上位 2 ビットの値が 3 の場合はタイプ 3 と呼ばれるチャンクになります。

タイプ 3 のチャンクメッセージヘッダはありません。全部省略されちゃいました。

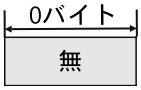


図 10. チャンクメッセージヘッダー タイプ 3 の構造

メッセージ長とメッセージタイプ ID、メッセージストリーム ID は最後のタイプ 0(または 1) チャンクで指定されたものをそのまま使用します。

差分タイムスタンプもないですが、最後のチャンクの差分タイムスタンプ (または絶対タイムスタンプ) の値を、このチャンクの差分タイムスタンプとして採用します。 わかりやすく言うと、この前のチャンクがタイプ 1 かタイプ 2 だった場合は、それの差分タイムスタンプの値を、このチャンクの差分タイムスタンプとして使います。 前のチャンクがタイプ 0 だった場合は通信開始時からの絶対タイムスタンプが入っていますが、そうだったとしても、その値を無理矢理このチャンクの差分タイムスタンプとして使います。

4.4.7. ヘッダの省略とチャンクストリーム ID

チャンクメッセージヘッダは省略して前のチャンクのものを採用することがありますが、どのチャンクの物も採用するわけではありません。

別なチャンクストリーム ID のチャンクは別な接続として扱うべきで、省略された値は同じチャンクストリーム ID の前の値を採用してください。

たとえば、以下のチャンクが A → B → C → D の順に流れたとします。

通信順番	チャンク ストリーム ID	チャンクタイプ	タイムスタンプ	メッセージ ストリーム ID	メッセージ長
チャンク A	2	タイプ 0	絶対 :0	0	42
チャンク B	3	タイプ 0	絶対 :100	1234	42

通信順番	チャンク ストリーム ID	チャンクタイプ	タイムスタンプ	メッセージ ストリーム ID	メッセージ長
チャンク C	2	タイプ 0	絶対 :100	0	42
チャンク D	3	タイプ 1	差分 :100	-	52

表 1. チャンクメッセージヘッダ省略の例

この時に、最後のチャンク D はチャンクストリーム ID が同じ 3 であるチャンク B からの差分であるものとします。 一つ前はチャンク C ですが、チャンクストリーム ID が異なるので差分の元としては採用しません。

上のチャンクで省略されたのを合成した結果は以下のチャンク A' ～ D' のようになります。

通信順番	チャンク ストリーム ID	タイムスタンプ	メッセージ ストリーム ID	メッセージ長
チャンク A'	2	絶対 :0	0	42
チャンク B'	3	絶対 :100	1234	42
チャンク C'	2	絶対 :100	0	42
チャンク D'	3	絶対 :200	1234	52

表 2. チャンクメッセージヘッダ省略の合成例

これにより一つの接続に複数のストリームを同時に載せてもチャンクメッセージヘッダが大きくならずに済みます。

逆に実装する側としてはチャンクストリーム ID 毎に、前のチャンクを保持してないといけません。

あと、前のチャンクがないチャンクストリーム ID でいきなりタイプ 1 ～ 3 の省略されたチャンクが来た場合に、省略された部分をどうすべきかなのですが、これも仕様書にははっきり書かれていません。 そんなことすんなよてのが正直なところですが出来てしまうので、もし来た時には省略された部分の前の値は 0 が入っていたものとするのが良いかと思われます。知らんけど……。

4.4.8. チャンクストリーム ID の割り当て

以前にチャンクストリーム ID は 0 ～ 2 を除いて自由に使えると説明しましたが、自由にと言われてもざっくりしすぎてて困りますね。

どうしたもんかというと、まあ本当に自由に決めていいので、3 番固定とかでもかまいません。マジで。

ただ複数のメッセージストリームを重畳する場合は、3 番固定ですとチャンクメッセージヘッダに頻繁にメッセージストリーム ID を載せないといけないので省略が効きづらくて効率が落ちてしまいます。 ですので、メッセージストリーム ID 一つにたいし、チャンクストリーム ID 一つを割り当てるのが良さそうです。

新しいメッセージストリーム ID を割り当てたら、チャンクストリーム ID3 番を、また新しいメッ

セージストリーム ID を割り当てたら、チャンクストリーム ID4 番を、というように使っていくのが良いと思います。

普通に割り当てられるメッセージストリーム ID は 1 番からなので、チャンクストリーム ID もメッセージストリーム ID+2 で自動的に割り当てるのも可能です。 しかしメッセージストリーム ID は 1 ～ 16,777,215 番まで自由に使えるのに対し、チャンクストリーム ID は 3 ～ 63 までなら 1 バイト、64 ～ 309 なら 2 バイトと、値が小さい方が通信効率が良くなります。 メッセージストリーム ID の割り当てはサーバー側で行うため[6]、メッセージストリーム ID に何が入ってくるかわからない場合にはチャンクストリーム ID には独立した値を割り当てた方が安心です。もしあなたがサーバーを書いている場合は、メッセージストリーム ID をなるべく 1 ～ 61 の間で割り当ててしまう手もありでしょう。

4.4.9. 拡張タイムスタンプ

タイプ 0 ～ 2 の絶対タイムスタンプや差分タイムスタンプに 0xFFFFFF(16777215) が入っていた場合は、チャンクメッセージヘッダの後に 4 バイトの拡張タイムスタンプが続きます。

拡張タイムスタンプがある場合は、ここの値を絶対タイムスタンプや差分タイムスタンプのかわりに採用します。

さすがに差分タイムスタンプが 4 バイトで足りないことはないと思いますが、通信開始時からの経過時間を入れる絶対タイムスタンプでは 4 バイトで足りなくなることがあります。 だいたい 50 日弱で足りなくなるようですが、その場合は単純にループして 0 に戻ります。ループしてもちゃんと動くようにしておきましょう。

4.4.10. チャンクデータ

拡張タイムスタンプ (あれば) の後にはチャンクデータが続きます。

チャンクデータの長さはチャンクメッセージヘッダに書かれているメッセージ長とチャンクの最大長によって決まります。 チャンクの最大長はあとで説明する Set Chunk Size メッセージで設定されますが、既定値は 128 バイトです。 メッセージ長とチャンク最大長のどちらか短い方の分だけチャンクデータが続きます。

チャンクデータの中身はメッセージタイプ ID によって決まります。

4.4.11. メッセージの分割

チャンクデータはチャンク最大長分までしか入りませんが、メッセージ長はそれより大きい場合があります。 その場合は複数のチャンクに分割してメッセージを送信します。

6　クライアントがあとで説明する createStream コマンドを発行することで、サーバーが新しいメッセージストリーム ID を割り当てて返します。

ただ、メッセージの分割については仕様にあまりはっきり書いてません。 分割する場合はタイプ 3 で続きのチャンクデータを送るのが普通、という感じには書いてありますが、もしそれ以外のタイプで送られた場合にどうすべきかというのがはっきりしません。

普通に考えると、タイプ 3 相当の場合には続きであると判定するのが良さそうです。

- 前のチャンクとチャンクストリーム ID が同じ
- 前のチャンクでまだメッセージ長分のデータを受け取っていない (残りがある)
- 前のチャンクとメッセージストリーム ID が同じ
- 前のチャンクとタイムスタンプが同じ
- 前のチャンクとメッセージ長が同じ

以上の条件が成立する場合には続きのチャンクデータと扱いましょう。

たとえばチャンク最大長が 128 バイトの場合にメッセージ長が 131 バイトあった時には以下のように分割されます。

通信順番	チャンク ストリーム ID	タイムスタンプ	メッセージ ストリーム ID	メッセージ長	チャンクデータ
チャンク A	3	差分 :7144	1234	131	128 バイト
チャンク B	3	差分 :7144	1234	131	3 バイト

表 3. メッセージの複数チャンクへの分割の例

メッセージが完成していないうちに、途中に別なチャンクストリーム ID のチャンクが入った場合は完成していないメッセージはそのまま保持しておくべきですが、同じチャンクストリーム ID でメッセージストリーム ID が別なチャンクが挟まった時はどうすべきでしょうね。 仕様には何も書いてないのでわかりませんけど、破棄してしまって良さそうな気もします。そういうことすんなよってのはもっともですけど。

4.4.12. Acknowledgement

RTMP Chunk Stream ではなぜか輻輳制御のような機能が付いており、一定バイト数を受け取ったら Acknowledgement メッセージを返さなければなりません。Acknowledgement メッセージの中身については後で説明します。

一定バイト数、というのは後で説明する Window Acknowledgement Size メッセージで通信相手から指定されます。 通信開始時、または前回 Acknowledgement メッセージを送ってから、指定のバイト数を受信した時点で Acknowledgement メッセージを送る必要があります。 そうしないとそれ以上のデータが送られてきません。

逆に送信する時も、Window Acknowledgement Size メッセージで指定したバイト数を送信したら Acknowledgement メッセージが受信されるまでそれ以上送信してはいけません。

Window Acknowledgement Size で指定されるバイト数は送受信でそれぞれ異なる値になることがあるので、別に管理してください。

Acknowledgement を送信するまでの受信バイト数は、チャンクストリームのボディのバイト数等ではなく、TCP で送受信するバイト数です。チャンクのハンドシェイクやヘッダも含みますし、チャンクストリーム ID に関係なく全チャンクストリーム共通で管理してください。

それ TCP レベルでやってるからいらなくない？という気もするんですけど、何故かやらないといけないんですよねえ……。

4.5. RTMP Control Message

ここまででチャンクの送受信ができるようになりました！上手くいっていればな。

あとは受信できたメッセージの中身を処理します。

メッセージの中身はメッセージタイプ ID から判別されます。

その中でも 1、2、3、5、6 の 5 つは RTMP Control Message といって、接続全体の制御や設定に使うメッセージです。 RTMP Control Message は必ずチャンクストリーム ID は 2 番、メッセージストリーム ID は 0 番を指定します。

4.5.1. Set Chunk Size メッセージ (タイプ ID:1)

メッセージタイプ ID の 1 番は Set Chunk Size メッセージです。 自分が送信する最大のチャンク長を相手に通知します。

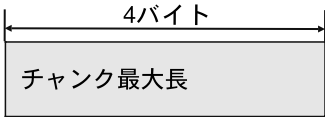


図 11. Set Chunk Size メッセージの内容

メッセージの中身は 4 バイトのチャンク最大長です。 最上位 1 ビットは必ず 0 になるので、有効なのは 31 ビットで 0x7FFFFFFF まで指定できますが、メッセージ長の最大が 0xFFFFFF なのでそれ以上の値は事実上意味がありません。 また下限は 1 で、0 は指定できません。ただし 128 以上が推奨されているのでそんなに小さくしない方が良いでしょう。

チャンク最大長はクライアントとサーバーそれぞれで管理します。たとえばクライアントが Set Chunk Size メッセージで 0xFFFFFF を指定し、サーバーが Set Chunk Size メッセージで 1024 を指定した場合、クライアントは最大で 16,777,215 バイトのボディを持ったチャンクを送信できますが、サーバーは最大で 1,024 バイトのボディを持つチャンクしか送信しません。

チャンク最大長は既定で 128 バイトです。 Set Chunk Size メッセージを送信・受信しない限り、チャンク最大長は 128 バイトとしてください。

こちらからチャンク最大長を指定するならいくらにすべきかですが、一つの TCP 接続で複数のチャンクストリームを載せる場合は一つのストリームで占有しないくらいのサイズにすべきでしょう。 ただ一つの TCP 接続に複数のチャンクストリームを載せることはあんまりしないと思います。

その場合は 0xFFFFFF を指定しちゃいましょう。分割しないで済むならその方が良いです。

4.5.2. Abort メッセージ (タイプ ID:2)

メッセージタイプ ID の 2 番は Abort メッセージです。 途中まで送ったメッセージの破棄を通知します。

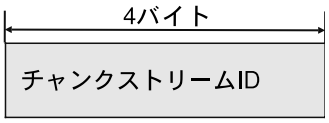


図 12. Abort メッセージの内容

メッセージの中身は 4 バイトのチャンクストリーム ID です。 32bit 指定できますが、チャンクストリーム ID 自体がそこまで無いので、3 ～ 65,599 の範囲だけが有効です。

指定したチャンクストリーム ID にメッセージが分割で送られているときに、残りの部分を送らずにメッセージ全体をキャンセルする場合に使います。 これを受け取ったら、指定のチャンクストリーム ID で分割の残り待ちしているメッセージを破棄してください。使う理由がほとんどないので、たぶん使われないんじゃないかと思います。長いメッセージを送っている途中で接続を切りたい場合に、残りはないことを示すのに使うそうです。

4.5.3. Acknowledgement メッセージ (タイプ ID:3)

メッセージタイプ ID の 3 番は Acknowledgement メッセージです。 一定バイト数受信したことを通知します。

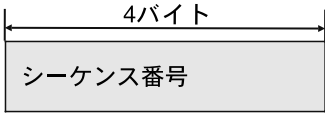


図 13. Acknowledgement メッセージの内容

メッセージの中身は 4 バイトのシーケンス番号です。 シーケンス番号とは単純に接続開始から今まで受信した総バイト数です。 総バイト数が 4GB 越えてたらどうするか書いてませんが、ループして 0 に戻ればいいんじゃないかと思います。

前に Acknowledgement の項で書きましたが、こちらから指定した Acknowledgement Window Size バイトだけ受信したら Acknowledgement を返信しない限り、相手からそれ以上の送信が行われません。 逆に相手に指定された Acknowledgement Window Size バイトだけ送信したら、次に Acknowledgement を受信できるまで、それ以上の送信を行ってはいけません。

しかし、ちょうど双方が Acknowledgement 待ちになってしまうとデッドロックしてしまいますね。仕様書には何も書いてないのでどうすべきなのかはわかりません。 現実的には Acknowledgement

受信できるまで送信を止めるというのは無視してもいいんじゃないかと思います。 もちろん Acknowledgement の返信はしっかりと実装すべきですが。

4.5.4. Window Acknowledgement Size メッセージ (タイプ ID:5)

メッセージタイプ ID の 5 番は Window Acknowledgement Size メッセージです。 Acknowledgement メッセージを待たずに送信できるバイト数を通知します。



図 14. Window Acknowledgement Size メッセージの内容

メッセージの中身は 4 バイトの Acknowledgement Window Size(単位はバイト) です。 32bit あるので 0x00000000 ～ 0xFFFFFFFF まで指定できそうに見えますが、実装によってはここの値を符号付きで扱ってしまうものがあるようなので、0x7FFFFFFF までにしておいた方が無難です。あとごく小さい値も指定できますが、ほとんどメッセージが送れなくなるのでやめましょう。小さくても 4096 以上にはしておいた方が無難でしょう。

Acknowledgement Window Size の既定値は困ったことに指定されていません。全く Window Acknowledgement Size を受けとらなかった時にどうすべきかは書いていません。 指定されない限りは Acknowledgement を送る必要はないのかもしれませんが、よくわからないので connect コマンドの直後くらいには Window Acknowledgement Size メッセージで指定しておくのが無難でしょう。

こちらから指定する Acknowledgement Window Size はいくらにするべきかですが、これを積極的に使う理由が見当たらないので、0x7FFFFFFF とかとても大きい値を指定してしまっていいんじゃないかと思います。

4.5.5. Set Peer Bandwidth メッセージ (タイプ ID:6)

メッセージタイプ ID の 6 番は Set Peer Bandwidth メッセージです。 相手に Acknowledgement Window Size をこのくらいにしてほしいなあというのを通知します。

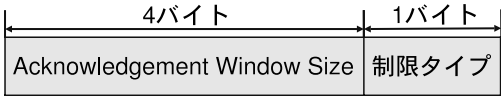


図 15. Set Peer Bandwidth メッセージの内容

メッセージの中身は 4 バイトの Acknowledgement Window Size(単位はバイト) と、1 バイトの制限タイプです。

中身が似ているので Window Acknowledgement Size メッセージとの違いがわかりづらいのですが、Window Acknowledgement Size メッセージがこの値を使いますと相手に通知するものである一方で、Set Peer Bandwidth はこの値で Window Acknowledgement Size メッセージを送ってくれませんか？と相手にお願いするものです。

このメッセージを受け取った場合は、制限タイプ (0 ～ 2) によって次のような動作をすべきです。

制限タイプの値が 0 の場合は、制限タイプは「ハード」です。通知された Acknowledgement Window Size の値を使って Window Acknowledgement Size メッセージを送るべきです。

制限タイプの値が 1 の場合は、制限タイプは「ソフト」です。通知された Acknowledgement Window Size が、前に Window Acknowledgement Size メッセージを送った時のより大きければ、新しい値であらためて Window Acknowledgement Size メッセージを送ります。通知された値が以前の値と同じかより小さければ特に何もしません。

制限タイプの値が 2 の場合は、制限タイプは「ダイナミック」です。前回の Set Peer Bandwidth メッセージの制限タイプが「ハード」だった場合、このメッセージも「ハード」として扱い、新しい値であらためて Window Acknowledgement Size メッセージを送ります。前回の Set Peer Bandwidth メッセージの制限タイプがそれ以外だったら何もしません。

ただ、Set Peer Bandwidth メッセージですが、必ずこの動作をしないといけないわけではなく、しなくてもかまいません。SHOULD(すべきである) としか書かれていないので、Window Acknowledgement Size メッセージを送らなくても動作はします。

まあ動作自体はそんなに難しくないので実装してあげるといいんじゃないでしょうか。 相手にでかい Acknowledgement Window Size を指定してもらいたい時には自分から積極的に送るのも良いでしょう。

4.6. RTMP Chunk Stream のまとめ

RTMP Chunk Stream については以上です。 いかがだったでしょうか。 なんだかめんどくさいですね。

RTMP Chunk Stream は仕様に書いてないところがあったり、チャンクストリーム ID とメッセージストリーム ID が別だったり、通信の設定をするのにメッセージの送受信ができないといけない循環構造になってたり、微妙な輻輳制御が入っていたり、読んでる分にはわかるような気がするけど実装しようとすると混乱しそうな点が詰まっています。

ここで RTMP Chunk Stream での通信の流れをまとめておきます。

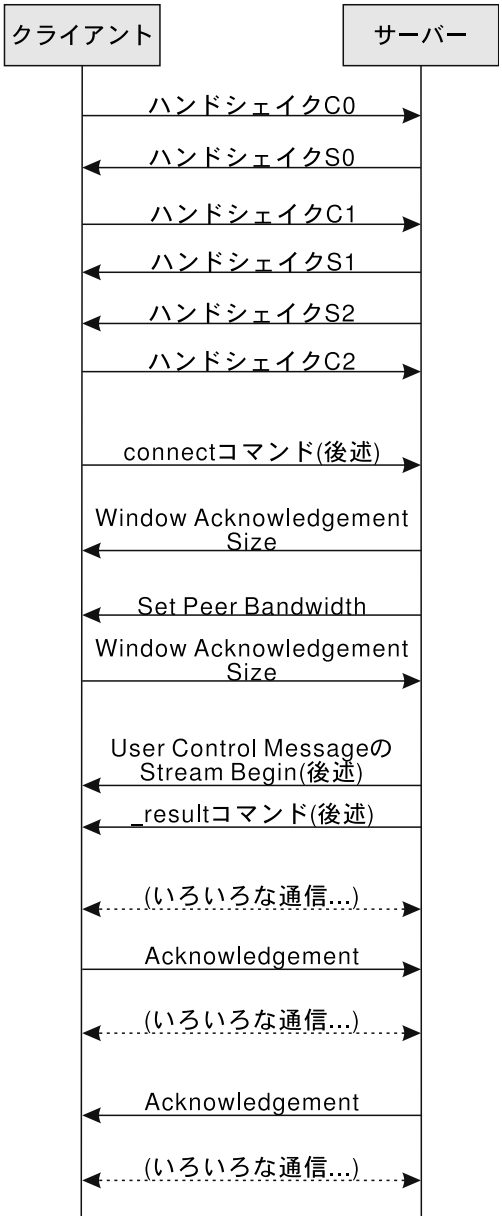


図 16. RTMP Chunk Stream 通信の流れ

まずハンドシェイクから始まり、クライアントとサーバーが C0 ～ C2、S0 ～ S2 のパケットを送り合います。 これが終わって初めてメッセージの送受信ができるようになります。

次に RTMP Message レベルの接続メッセージとして、connect コマンドが発行されます。connect コマンドというのはこれから説明する RTMP Message の物で、ここから RTMP Message での通信をするよーという意味です。今のところはあんま気にしなくて良いです。

接続できたら Set Chunk Size や Window Acknowledgement Size で通信設定をします。 しなくても動きますが、標準のチャンクサイズが 128 バイトと小さかったり、Acknowledgement Window Size がいくつなのかわからなかったりするので、最初に設定しておくのが良いでしょう。

また、Set Peer Bandwidth を送って相手にも Acknowledgement Window Size を送ってもらうよう通知すると良いかもしれません。

User Control Message の Stream Begin や _result コマンドというのもあとで説明する RTMP Message の物で、connect コマンドの返信です。

そのあと、しばらく通信すると設定した Acknowledgement Window Size 分だけ受信すると思いますので、その度に Acknowledgement を返してください。 相手から通知された Acknowledgement Window Size 分だけ送信したら、Acknowledgement を受信するまで送信をやめてください。 ……ということになっていますが、現実的には Acknowledgement を受信するまで送信をやめるというのはやらなくて良さそうです。 Acknowledgement 送信ができなくてデッドロックする可能性がありますので。

このフローのようにすれば、なんとか RTMP Chunk Stream での通信はできると思います。RTMP Chunk Stream はなかなか面倒かつ微妙にふんわりした仕様で不安になりますが、これで通信が上手くいきさえすれば、さらに上の RTMP Message はまだわかりやすいので頑張りましょう。

5. RTMP Message

RTMP Message は RTMP Chunk Stream のチャンクや RTMP Control Message よりアプリケーションに近いメッセージです。

RTMP Control Message なんかはアプリケーションによらず RTMP Chunk Stream で通信する時にはいつも同じ処理をするものでしたが、RTMP Message はアプリケーションによって処理される想定になっており、Flash Media Server や Flash Player では (直接ではないにしろ)ActionScript から操作できるようになっています。

RTMP Message はアプリケーションに近いメッセージということもあって、いくらか ActionScript のオブジェクトについての知識が必要になっていきますが、それも都度説明していこうと思います。

5.1. RTMP Message Format

RTMP Message Format は RTMP Message のエンコードフォーマットです。 仕様書には記載されていますが、RTMP Chunk Stream で RTMP Message をやりとりする場合にはほぼ使いません。

というのも、RTMP Message Format に載るデータはメッセージタイプ ID・メッセージデータ長・タイムスタンプ・メッセージストリーム ID とメッセージデータであり、RTMP Chunk Stream のレベルで既に載っているので通常は不要なのです。

使うのはあとで説明する Aggregate Message というメッセージ内にメッセージを入れる場合だけです。 なので RTMP Message Format の構造については Aggregate Message のところで説明しましょう。

5.2. RTMP Message の種類

RTMP Message もいくつか種類の種類に分けられます。

まず大きくは User Control Message と RTMP Command Messages に分かれます。

User Control Message は RTMP Mesage レベルの接続制御用メッセージです。 これは RTMP Message ではありますが、ほぼ機械的に処理されるメッセージです。

それ以外の RTMP Command Messages がアプリケーションで処理すべきメッセージとなっています。

メッセージの種類はメッセージタイプ ID によって識別され、対応は以下の表になっています。

メッセージ タイプ ID	メッセージ種類名	簡単な内容
4	User Control Message	接続制御用のメッセージです。メッセージデータに持つイベントタイプによってさらに細かくわかれます。
17 または 20	Command Message	通信相手側の関数 (メソッド) を実行する RPC 用のメッセージです。メッセージデータには関数名やパラメーターを持ちます。17 は AMF3 用、20 は AMF0 用です。
15 または 18	Data Message	なんかしらの ActionScript のデータを送るメッセージです。典型的には動画や音声のメタデータ (ビットレート等) を送るのに使われます。15 は AMF3 用、18 は AMF0 用です。
16 または 19	Shared Object Message	ActionScript のオブジェクト共有をするためのメッセージです。Data Message と違い一度共有したオブジェクトの変更点だけを送ったりできます。16 は AMF3 用、19 は AMF0 用です。
8	Audio Message	音声データを送るメッセージです。
9	Video Message	動画データを送るメッセージです。
22	Aggregate Message	複数のメッセージをグループ化して一つにまとめたメッセージです。

表 4. メッセージタイプ ID とその中身

RTMP Message のメッセージタイプ ID は RTMP Control Message のメッセージタイプ ID と被らないようになっているので無事に判別可能です。 一つの項目にレベルの違うプロトコルのデータが混ざってるのはやべえ気がしますけどそれが RTMP なので諦めましょう。

AMF0 用や AMF3 用と書いてあるのは、メッセージのデータ (主にオブジェクトデータ) がどんなフォーマットでエンコードされているかの違いです。 データのエンコードに AMF0 と AMF3 というフォーマットがあるのですが、どちらも使えるためメッセージタイプ ID によって判別するようになっています。 AMF の中身についてはまたあとで説明します。

5.3. User Control Message (タイプ ID:4)

メッセージタイプ ID の 4 番は User Control Message です。 User Control Message は RTMP Message レベルでの通信制御メッセージです。

User Control Message は接続単位で処理されるメッセージなので多重化されません。常にチャンクストリーム ID が 2、メッセージストリーム ID が 0 で送られます。

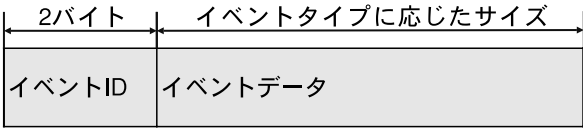


図 17. User Control Message の内容

メッセージの中身は 2 バイトのイベントタイプと、イベントタイプに応じたイベントデータです。

イベントタイプは以下のようになっています。

イベントタイプ	イベント名	送信方向 (S= サーバー、C= クライアント)	内容
0	Stream Begin	S → C	新しいメッセージストリームの開始を通知します。
1	Stream EOF	S → C	指定したメッセージストリームの終了を通知します。
2	StreamDry	S → C	メッセージストリームでの通信が一定時間行われていないことを通知します。
3	SetBufferLength	C → S	クライアントが保持できる動画や音声のバッファ長をミリ秒単位で通知します。
4	StreamIsRecorded	S → C	指定したストリームが録画 (録音) されているものであることを通知します。
6	PingRequest	S → C	クライアントに対して接続性確認のため PingResponse を返すように通知します。
7	PingResponse	C → S	サーバーの送ってきた PingRequest への返信を通知します。

表 5. User Control Message のイベントタイプ

イベントタイプの 5 番はなぜか欠番です。

イベント名にスペースが入ってるものと入ってないものがあるのはなんなんでしょうね。仕様書に合わせたのでこうなってますけど大した意味はなさそうな気がします……。

5.3.1. Stream Begin イベント (イベントタイプ :0)

Stream Begin イベントはサーバーからクライアントに新しいメッセージストリーム開始を通知します。

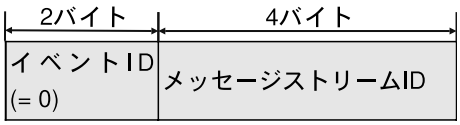


図 18. Stream Begin イベントの内容

イベントデータには 4 バイトのメッセージストリーム ID が入ってきます。 このメッセージストリーム ID はビッグエンディアンで入ります。

このメッセージストリーム ID はサーバーからクライアントにメッセージを送るのに、これから

この番号を使いますよ、という通知です。

クライアントがサーバーに送るのに使うメッセージストリーム ID と、サーバーがクライアントに送るのに使うメッセージストリーム ID は同じなのか別なのかが気になるところですが、仕様にはなんとも書いていません。ただ、別にすると問題が出る箇所があるので同じ物として扱った方が良さそうです。

このイベントがどういう時に通知されるかというと、サーバーからクライアントにメッセージを送りたくなる前です。 基本的にはクライアントからサーバーに接続した時や、クライアントが新しいメッセージ ID でサーバーにメッセージを送った時に、その返事を返すのに通知されます。

クライアントは今までこのイベントで通知された以外のメッセージストリーム ID でメッセージを受け取った場合は、無視してかまいません。

5.3.2. Stream EOF イベント (イベントタイプ :1)

Stream EOF イベントはサーバーからクライアントにメッセージストリームの終了を通知します。

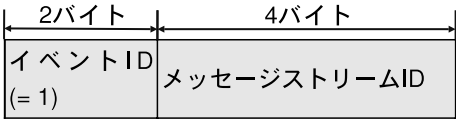


図 19. Stream EOF イベントの内容

イベントデータには 4 バイトのメッセージストリーム ID が入ってきます。 このメッセージストリーム ID はビッグエンディアンで入ります。

指定した ID のメッセージストリームは終了、もうこれ以上メッセージを送りません、というのを通知します。 このイベントを受け取ったクライアントは、それ以降にこのメッセージストリーム ID のメッセージを受け取っても無視してください。

5.3.3. StreamDry イベント (イベントタイプ :2)

StreamDry イベントはサーバーからクライアントに、メッセージストリームでの通信がしばらくないことを通知します。

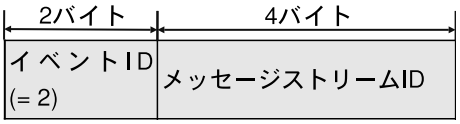


図 20. StreamDry イベントの内容

イベントデータには 4 バイトのメッセージストリーム ID が入ってきます。 このメッセージストリーム ID はビッグエンディアンで入ります。

指定した ID のメッセージストリームでサーバーからのメッセージ送信が一定時間ないことを通知します。

どういう時に使うのか書いていないのではっきりしませんが、おそらく動画再生時などにサーバー側にもデータが無い事を通知するのに使うのだと思います。 たとえば、ライブ配信を視聴していて配信側の問題でデータが途切れてしまった場合に、サーバーからクライアント間の接続は問題ないもののサーバー側にデータがないので StreamDry イベントでそのことを通知するもの、じゃないかなあ……。

5.3.4. SetBufferLength イベント (イベントタイプ :3)

SetBufferLength イベントは、クライアントが保持できる動画や音声のバッファ長をサーバーに通知します。

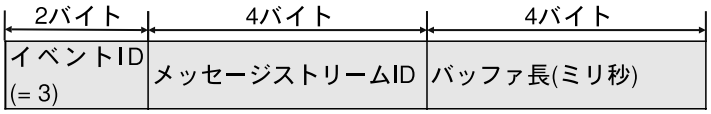


図 21. SetBufferLength イベントの内容

イベントデータには 4 バイトのメッセージストリーム ID と 4 バイトのバッファ長 (ミリ秒単位) が入ってきます。 このメッセージストリーム ID(とバッファ長) はビッグエンディアンで入ります。

このイベントはクライアントからサーバーに通知されるものです。 動画や音声再生の時に、クライアント側で保持できるバッファをサーバーに教えます。

ここで設定したバッファ長を埋められるようにサーバーが動画や音声を送れない場合は StreamDry イベントを通知するものじゃないかと思われます。

5.3.5. StreamIsRecorded イベント (イベントタイプ :4)

StreamIsRecorded イベントは、クライアントに動画や音声が録画 (録音) されているものであること (ライブじゃないこと) を通知します。

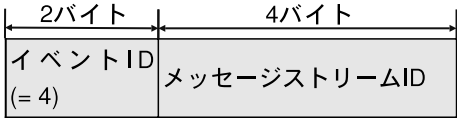


図 22. StreamIsRecorded イベントの内容

イベントデータには 4 バイトのメッセージストリーム ID が入ってきます。 このメッセージストリーム ID はビッグエンディアンで入ります。

指定した ID のメッセージストリームで送られる動画や音声がライブ配信でない時に通知されます。 追い掛け再生みたいな時はどうするのかは……まあアプリによって決めればいいでしょう。

5.3.6. PingRequest イベント (イベントタイプ :6)

PingRequest イベントは、クライアントに対して接続性確認のため PingResponse を返すように通知します。

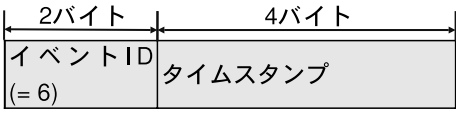


図 23. PingRequest イベントの内容

イベントデータには 4 バイトのタイムスタンプが入ってきます。 タイムスタンプの値は送信時の絶対タイムスタンプ値 (通信開始時からの経過ミリ秒数) です。

通信ができてるか確認するイベントです。 クライアントはこのイベントを受け取ったら次の PingResponse イベントを返しましょう。

どんな時にこのイベントが送られるのかは謎です。

5.3.7. PingResponse イベント (イベントタイプ :7)

PingResponse イベントは、サーバーの送ってきた PingRequest への返信を通知します。

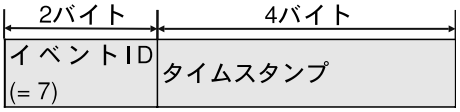


図 24. PingResponse イベントの内容

イベントデータには 4 バイトのタイムスタンプが入ってきます。 タイムスタンプの値は PingRequest 受信時の絶対タイムスタンプ値 (通信開始時からの経過ミリ秒数) です。

クライアントが PingRequest を受け取ったらこのイベントで返信しましょう。 サーバーが無事イベントを受け取れたら、クライアントと通信ができていることが確認されます。

5.3.8. User Control Message まとめ

User Control Message のイベントは以上です。

受け取った時の動作が具体的に決まっているのは Stream Begin、Stream EOF、PingRequest くらいです。 その他はヒントのようなものなので、あんまり処理しなくても良いでしょう。PingRequest は実際使うことないような気もしますし、特別処理しなくても動くんで、ほぼ全部処理する必要ないくらいではありますが……。

サーバーを実装する場合は少なくとも Stream Begin はちゃんと送るようにしましょう。他は最

悪どうでもいいんじゃないかな。

5.4. Data Message (タイプ ID:15 または 18)

Data Message はなんかしらの ActionScript のデータを送るメッセージです。

アプリケーション固有のデータを送るのに使いますが、典型的には動画や音声のメタデータ (ビットレート等) を送るのに使われます。

このメッセージはメッセージストリーム毎に送られますので、メッセージストリーム ID は 0 以外が使われます。



図 25. Data Message の内容

Data Message にはデータ名と値の列が入ってきます。 このデータ名と値はそれぞれ ActionScript のデータ型で格納されます。 ActionScript のデータ型のフォーマットは AMF0 と AMF3 というのがあり、どっちのフォーマットで入っているかによってメッセージタイプ ID が異なります。 メッセージタイプ ID が 15 の場合は AMF3、18 の場合は AMF0 で入っています。 違うのはフォーマットだけなので値の意味は同じです。 AMF0 と AMF3 についてはまたあとで詳しく説明します。

データ名と値にどんな値が入るかはアプリケーションによって決まりますので基本的には規定されていません。最低でもデータ名と値が 1 つは入りますが、値が 2 つ以上続く場合もあります。

動画や音声のメタデータについてはデータ名と値に関して規定がありますので紹介しましょう。

メタデータを送る時には、名前に文字列で onMetaData、値にオブジェクトまたは配列型でメタデータのプロパティ名と値の組を設定します。 onMetaData の値に入るプロパティは以下のようになっています。

プロパティ名	値の型	値の内容
duration	数値	動画・音声の長さを秒単位で格納します。
filesize	数値	ファイルの長さをバイト単位で格納します。FLV でのみ使います。
creationdate	文字列	作成日時を文字列で格納します。
canSeekToEnd	真偽値	最後の動画フレームがキーフレームかどうかを格納します。
audiocodecid	数値	音声のコーデック ID を格納します。Audio Message に入ってるコーデック ID と同じ値が入ります。
audiodatarate	数値	音声のビットレートを kbps 単位で格納します。
audiodelay	数値	音声コーデックにより音声の遅延を秒単位で格納します。

プロパティ名	値の型	値の内容
audiosamplerate	数値	音声のサンプリング周波数を格納します。
audiosamplesize	数値	音声のサンプリング解像度を (たぶんビット単位で) 格納します。
stereo	真偽値	音声がステレオかどうかを格納します。
videocodecid	数値	動画のコーデック ID を返します。Video Message に入ってるコーデック ID と同じ値が入ります。
videodatarate	数値	動画のビットレートを kbps 単位で格納します。
width	数値	動画の幅をピクセル単位で格納します。
height	数値	動画の高さをピクセル単位で格納します。
framerate	数値	動画の秒間フレーム数を格納します。

表 6. onMetaData のプロパティ

これらのプロパティはかならずしも全部入ってくるとは限りませんし、入れなくてもかまいません。特に上の方の duration・filesize・creationdate・canSeekToEnd なんかはライブだと入れようがないですしね。

また、onMetaData に入っているのは主に表示用の参考情報であり、再生に必要な情報は全て Audio Message や Video Message 側に入っています。 onMetaData は必須ではないので、無くても再生はできるようになっています。

他に XML 形式のメタデータである XMP を格納する、onXMPData というのもあります。onXMPData の値に入るプロパティは以下のようになっています。

プロパティ名	値の型	値の内容
liveXML	文字列	XMP メタデータを XML で格納します。

表 7. onXMPData のプロパティ

XMP メタデータについてはここでは解説しません。 気になる方は XMP の仕様[7]を確認してみてください。 ただこれが RTMP で実際使われているかはわかりません……。

また、配信時にはクライアントからサーバーに @setDataFrame と @clearDataFrame というデータが送られてきます。 これらはデータをその場で処理するのではなく、配信データとして記録することを指示しています。

@setDataFrame の場合は値が 2 つ続き、一つ目は記録するデータ名の文字列 (onMetaData とか)、二つ目は記録するデータの値 (onMetaData に設定するオブジェクトまたは配列など) が入ります。

@clearDataFrame の場合は値が 1 つ続き、データ名の文字列 (onMetaData とか) が入ります。

@setDataFrame が来た場合には、現在配信されているまたはこれ以降に配信される動画や音声のファイルに、メタデータとして名前と値の列を記録するように覚えておきます。 @clearDataFrame が来た場合には、前の @setDataFrame で覚えていたメタデータをクリアして、それ以降に配信される動画や音声のファイルにもメタデータとして記録しないようにします。

7 https://www.adobe.com/devnet/xmp.html

なんだかちょっと難しいですね。クライアントからサーバーへの配信時には、`@setDataFrame` で一段ラップされた `onMetaData` が入ってくるという認識で大丈夫です。

5.5. Audio Message (タイプ ID:8)

Audio Message は音声データを送るメッセージです。

このメッセージもメッセージストリーム毎に送られますので、メッセージストリーム ID は `0` 以外が使われます。

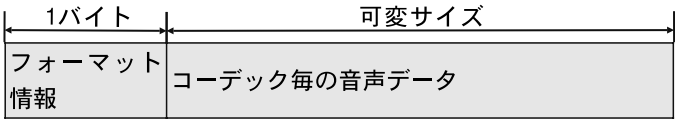


図 26. Audio Message の内容

Audio Message は先頭に 1 バイトのフォーマット情報が付いて、残りは基本的に各コーデックでエンコードされたデータです。 ただしコーデックが AAC の場合のみフォーマット情報の後にもう 1 バイトの AAC パケット種別が付いてきてから、AAC のデータが続きます。

フォーマット情報にはさらにビット単位で情報が詰め込まれています。

フォーマット情報の 0 〜 3 ビット目 (上位 4 ビット) がコーデック ID です。コーデック ID は以下の表のようなものがあります。 この表以外にも番号は定義されていますが、実際は使えないようなので省略しています。 ただここに挙げてある中でも、今の動画配信だと MP3 か AAC しか使えないと思います……。

コーデック ID	内容
1	ADPCM
2	MP3
3	リニア PCM(リトルエンディアン)
4	Nellymoser 16kHz モノラル
5	Nellymoser 8kHz モノラル
6	Nellymoser
10	AAC
11	Speex 16kHz モノラル

表 8. Audio Message のコーデック ID

フォーマット情報の 4 〜 5 ビット目 (2 ビット) はサンプリング周波数です。 2 ビットだとそのままは入らないので以下の表から実際のサンプリング周波数を計算します。

数値	サンプリング周波数
0	5.5kHz
1	11kHz

数値	サンプリング周波数
2	22kHz
3	44kHz

表 9. Audio Message のサンプリング周波数

フォーマット情報の 6 ビット目 (1 ビット) は量子化ビット数が 8bit か 16bit かを表します。

数値	量子化ビット数
0	8bit
1	16bit

表 10. Audio Message の量子化ビット数

量子化ビット数はリニア PCM にのみ適用されます。他のコーデックでは基本的に 16bit か、コーデック固有のデータに含まれるヘッダで決定されます。

フォーマット情報の 7 ビット目 (下位 1 ビット) はチャンネル数を表します。

数値	チャンネル数
0	モノラル
1	ステレオ

表 11. Audio Message のチャンネル数

コーデックによってはサンプリング周波数、量子化ビット数、チャンネル数は無視されます。サンプリング周波数とチャンネル数が固定されているコーデックはもちろん、MP3 と AAC もコーデック固有のデータとして入っているヘッダ側であらためて指定されるため、フォーマット情報の値は無視されます。

コーデックが AAC(コーデック ID が **10**) の場合のみ、フォーマット情報の後には AAC パケット種別が 1 バイトの続きます。

AAC パケット種別は **0** の場合はデータとして AudioSpecificConfig が入ります。**1** の場合は AAC の生データが入ります。

残りのデータはコーデック毎の生データが入ります。

MP3 の場合は MP3 ファイルの MPEG オーディオフレームデータが入ります。

AAC の場合は AAC パケット種別によって変わりますが、AAC パケット種別が **0** の場合は AudioSpecificConfig が入ります。**1** の場合は AAC のフレームデータが入ります。どちらも中身は MPEG-4 の AAC 仕様を参照してください。[8]

Speex は 16bit、16kHz、モノラルで固定で、生データが入ります。

Nellymoser は正確には Asao というコーデックのようですが、詳細は非公開なようなので不明です。

8 AAC のフォーマットは ISO/IEC 14496-3 を参照のこと。仕様書のお値段が高い。

Audio Message は AAC 以外のコーデックではどのパケットを取り出しても再生できるようになっています。 AAC だけは最初に AudioSpecificConfig が必要になります。 AudioSpecificConfig さえあればその後はどのパケットを取り出しても再生可能です。

5.6. Video Message (タイプ ID:9)

Video Message は動画データを送るメッセージです。

このメッセージもメッセージストリーム毎に送られますので、メッセージストリーム ID は 0 以外が使われます。

一つの Video Message は 1 フレーム分のデータを格納し、1 フレーム分のデータを複数の Video Message に分割することは許されていません。 ただし、全部キーフレームというわけではなく、イントラフレーム (中間フレーム) でもかまいません。イントラフレーム自体が分割されてはならないという意味です。

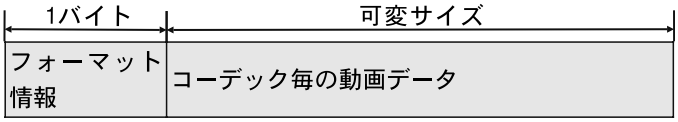


図 27. Video Message の内容

Video Message も先頭に 1 バイトのフォーマット情報が付いて、残りは基本的に各コーデックでエンコードされたデータです。 ただしコーデックが AVC(H.264) の場合のみフォーマット情報の後にもう 1 バイトの AVC パケット種別と 3 バイトの Composition Time が付いてきてから、AVC のデータが続きます。

フォーマット情報にはさらにビット単位で情報が詰め込まれています。

フォーマット情報の 0 〜 3 ビット目 (上位 4 ビット) はフレーム種別です。フレーム種別は以下の表のようになっています。

フレーム種別	内容
1	キーフレーム (AVC の場合はシーク可能フレーム)
2	イントラフレーム (AVC の場合は非シーク可能フレーム)
3	破棄可能なイントラフレーム (H.263 のみ)
5	ビデオ情報 / コマンドフレーム

表 12. Video Message のフレーム種別

フォーマット情報の 4 〜 7 ビット目 (下位 4 ビット) はコーデック ID です。コーデック ID には以下の表のようなものがあります。

コーデック ID	内容
2	Sorenson H.263 (Flash Video)

コーデック ID	内容
3	Screen Video
4	On2 VP6
5	On2 VP6 (アルファチャンネル付き)
6	Screen Video V2
7	AVC (H.264)

表 13. Video Message のコーデック ID

Screen Video は Flash 独自のコーデックで、デスクトップの画面取り込みに特化したものです。 Sorenson H.263 は標準の H.263 をいじったもので、Flash Video と呼ばれることもあるコーデックです。 On2 VP6 は (今は亡き)On2 Technologies という会社が作ったコーデックで、アルファチャンネル付きで透過ビデオが作れるのは面白いですね。 いずれにせよ、今の動画配信だと AVC(H.264) しか使えないと思いますが……。

コーデックが AVC(コーデック ID が 7) の場合のみ、フォーマット情報の後には 1 バイトの AVC パケット種別と 3 バイトの Composition Time が続きます。

AVC パケット種別は 0 の場合はデータとして AVCDecoderConfigurationRecord、1 の場合は AVC の NAL unit の列、2 の場合は AVC の End of Sequence を表します。

Composition Time には AVC パケット種別が 1 の時のみ Composition Time Offset がミリ秒単位で入ります。それ以外は常に 0 です。

残りのデータはコーデック毎のデータが入ります。

コーデック毎のデータはほぼ Flash 専用になるのでここでは AVC のみ紹介します。他のコーデックについての詳細は SWF の仕様書[9]にありますので知りたい場合はそちらを参照してください。

コーデックが AVC の場合は、AVC パケット種別によって変わります。

AVC パケット種別が 0 の場合は AVCDecoderConfigurationRecord が、1 の場合は NALUSample が入ります。これらは mp4 ファイルに AVC のデータを入れる時のフォーマットなので詳しくはそちらを参照してください[10]。2 の End of Sequence は何が入ってるかよくわかりません……。

NALUSample は NAL Unit の列ですが、この先頭にはエンコーダーによって Access Unit Delimiter[11] が入っていたりいなかったりするようです。 一つの Video Message は完成したピクチャ (フレーム) を一つだけ含むことになっているので Access Unit Delimiter は必要ないのですが、入れてくるエンコーダーが存在する以上はあってもなくても正しく処理できるようにしておくべきでしょう。

9 https://www.adobe.com/devnet/swf.html

10 より正確には ISO base media file format ですが、つまり mp4 ファイルなので細かいことは気にするな。mp4 に AVC なんかを入れるフォーマットは ISO/IEC 14496-15 を参照のこと。仕様書のお値段が高い。

11 AVC(H.264) でフレームの区切りを表すためのデータ

5.7. Shared Object Message (タイプ ID:16 または 19)

Shared Object Message は ActionScript のオブジェクトを複数のクライアントやサーバー間で共有するためのメッセージです。

このメッセージもメッセージストリーム毎に送られますので、メッセージストリーム ID は 0 以外が使われます。たぶん。

Data Message でもオブジェクトの送信はできますし、Command Message でメソッド呼び出しはできるのですが、Shared Object Message は変更点の通知などを行うことができます。

が、ここでは解説しません。 動画配信では関係ないのと、ActionScript にそんなに詳しくないので具体的な使い方がよくわからないためです……。

仕様書には一応データ構造は書いてありますので気になる方は見てみてください。 例によって詳細は書いてないですけど。

5.8. Aggregate Message (タイプ ID:22)

Aggregate Message は複数の RTMP メッセージをグループ化して一つにまとめたメッセージです。

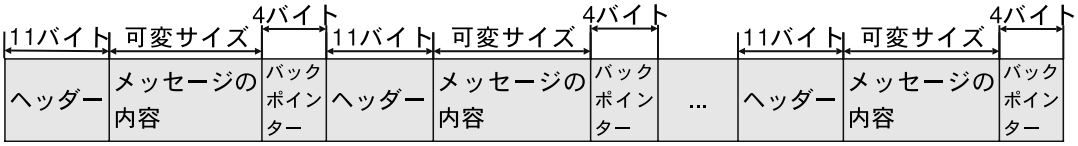


図 28. Aggregate Message の内容

メッセージは RTMP Message Format で並べ、メッセージ間にはバックポインターというものが挟まります。

RTMP Message Format は次のような形になっています。Aggregate Message はこれにバックポインターを付けたものを並べただけです。

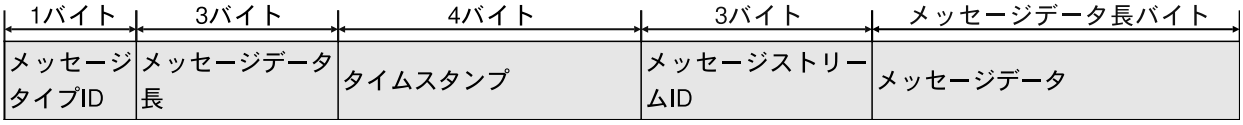


図 29. RTMP Message Format の構造

バックポインターは 4 バイトで、一つ前のメッセージのサイズ (ヘッダを含む) をバイト単位で格納します。 しかしバックポインターの使い道は無いです。FLV ファイルとなぜかフォーマットを合わせるために入っているようです。

Aggregate Message に含まれる各メッセージのタイムスタンプは、この Aggregate Message 自身のタイムスタンプからの差分で格納します。 つまり、Aggregate Message 自身のタイムスタンプは 2345 だったときに、含まれる最初のメッセージのタイムスタンプが 0 だったら、最初のメッセージのタイムスタンプは 2345(0+2345) として処理されます。 含まれる二番目のメッセージがタイムスタンプ 10 だったら、2355(10+2355) として処理してください。

Aggregate Message を使う意味ですが、仕様書によるとメッセージサイズを大きくできるのが利点とのことです。 最大チャンク長を大きく設定すると一つのチャンクで大きなメッセージを送信できるので、Aggregate Message でまとめるとチャンクヘッダの分が節約される、という言い分です。

が、しかし。Aggregate Message は含まれるメッセージ一つずつに RTMP Message Format のヘッダがフル (11 バイト) で載る上に、4 バイトの (本来不要な) バックポインターまで付いてきます。RTMP Message Format のヘッダに載る情報は RTMP Chunk Stream なら省略して小さくできるのに、Aggregate Message だとフルで載せないといけないので逆に通信効率が落ちてますね。

使わねえわこれ。

5.9. Command Message (タイプ ID:17 または 20)

Command Message は通信相手側の関数を実行する RPC 用のメッセージです。メッセージデータには関数名やパラメーターを持ちます。17 は AMF3 用、20 は AMF0 用です。

Command Message が一番アプリケーションレベルに近いメッセージと言いますか、ActionScript からは主にこれを操作することになります。

このメッセージはコマンドによってメッセージストリーム ID が 0 だったりそれ以外が使われたりします。

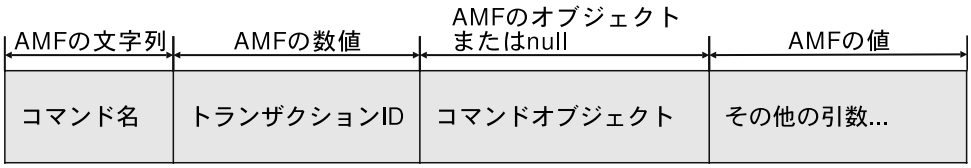


図 30. Command Message の内容

中身は全て ActionScript の値で、AMF0 または AMF3 でエンコードされて入っています。

先頭のコマンド ID、トランザクション ID、コマンドオブジェクトはどのコマンドでも共通で入っています。 それ以降にはコマンドによっていくつか値が続いたり続かなかったりします。

コマンド名は文字列で、コマンド名 (つまり関数名) を表します。当然必須です。

トランザクション ID は数値で、戻り値を受け取る時の識別子を設定します。戻り値が不要な場

合は 0 が設定されます。 トランザクション ID に 0 以外が設定されている場合は、受け取った側はコマンド実行した戻り値を _result コマンド (または _error コマンド) に入れて返しますが、その際に同じトランザクション ID を設定することでどのコマンド実行の戻り値かを識別します。 何番を割り当てるかはコマンドを発行する側が適当に決めます。

コマンドオブジェクトはオブジェクト型 (か配列型) または null が入り、なんかしらコマンドの情報を設定します。これはコマンドによって内容が変わります。使わない場合は null が設定されます。

コマンドによってはコマンドオブジェクトの後にもいくつか値が続きます。受信時はメッセージの長さ分まで値を読み出してください。

Command Message でやりとりできるコマンドについては次から見ていきましょう。

5.10. NetConnection コマンドと NetStream コマンド

Command Message でやりとりするコマンドには NetConnection コマンドと NetStream コマンドに分かれます。 中身は違いがないんで、仕様を読んでいても分かれてる意味を読み取るのが難しいところです。

まず NetConnection と NetStream というのは ActionScript のクラス名です。 ActionScript で RTMP(や RTMFP) 通信をしようとすると、この二つのクラスを使うことになります。

NetConnection オブジェクトは接続自体を表し、こいつの connect() メソッドを呼ぶとサーバーへの接続を確立します。 NetConnection オブジェクトはデータの送受信は行わず、接続確立と切断など接続全体にかかわるコマンドの発行しかできません。

動画の再生や配信要求など、アプリケーションレベルのデータや要求の送受信は NetStream オブジェクトを使います。

NetStream クラスはコンストラクタに NetConnection オブジェクトを受け取り、NetConnection の接続上にメッセージのやりとりができるストリームを構築します。 一つの NetConnection に対して複数の NetStream オブジェクトを構築することも可能です。 つまり一つの NetConnection に対して複数の NetStream を多重化することができます。

察しの良い方はもうわかったかもしれませんが、NetConnection オブジェクトは RTMP Chunk Stream の接続に対応します。 NetStream オブジェクトはそれぞれがメッセージストリーム ID 一つに対応します。

このため NetConnection コマンドは常にメッセージストリーム ID が 0 番で送受信されます。 ただ、メッセージストリーム ID は 0 番ですが、チャンクストリーム ID は 2 番以外を使ってください。チャンクストリーム ID2 番は RTMP Control Message と User Control Message でのみ使います。

NetStream コマンドはメッセージストリーム ID が 0 番以外で送受信されます。チャンクストリー

ム ID ももちろん 2 番以外を使ってください。

流れとしては、まず NetConnection コマンドでアプリケーションレベルの接続を確立して、そのあとの NetStream コマンドでアプリケーション毎の処理を行うようになります。

5.11. connect コマンド (NetConnection)

connect コマンドは一番最初に発行するコマンドです。クライアントからサーバーに送られます。NetConnection のコマンドなのでメッセージストリーム ID は 0 番で送ります。 チャンクストリーム ID は 2 番以外を使ってください。



図 31. connect コマンドの内容

コマンド名は当然ながら connect です。 connect コマンドは戻り値を受け取りますので、トランザクション ID には常に 1 が入ります。 connect コマンドの情報はコマンドオブジェクトにほぼ入ってきます。

ユーザー指定の引数にはアプリケーションで指定した追加の引数が入ってきます。 ActionScript では NetConnection オブジェクトの connect メソッドに好きなだけ追加の引数が渡せるようになっているのですが、その値がユーザー指定の引数に入ってきます。 アプリケーション固有の物なので内容は決まっていません。

RTMP のハンドシェイクが終わったら、クライアントはなによりもまずこのコマンドを発行してください。 サーバーは connect コマンドが来ないまま他のコマンドが来ちゃったら、無視するか切るかするといいんじゃないでしょうか。

重要なのはコマンドオブジェクトの内容です。ここにはオブジェクト型 (または配列型) でプロパティと値の組が入ってきます。ちなみにプロパティ名は常に文字列です。

プロパティ名	型	内容	値の例
app	文字列	接続対象のサーバーアプリケーションを指定し ます。	liveapp
flashver	文字列	クライアントの Flash Player のバージョンを表 します。	FMSc/1.0
swfUrl	文字列	接続元の Flash(swf) ファイルの URL を表します。	http://example.com/hoge.swf

プロパティ名	型	内容	値の例
pageUrl	文字列	接続元の HTML ファイルの URL を表します。	http://example.com/hoge.html
tcUrl	文字列	接続対象の URL を指定します。	rtmp://example.com/liveapp/livestream
fpad	真偽値	プロキシを使っているかどうかを表します。	false
audioCodecs	数値	クライアントがサポートしている音声コーデック を表します。	0x0404
videoCodecs	数値	クライアントがサポートしている動画コーデック を表します。	0x0080
videoFunction	数値	クライアントがサポートしている動画操作を 表します。	1
objectEncoding	数値	クライアントがサポートしているオブジェクト エンコード方式 (AMF0 か AMF3) を表します。 AMF0 のみの場合は 0、AMF3 をサポートする場合は 3 が入ります。	3

表 14. connect コマンドのコマンドオブジェクトのプロパティ

場合によってはこのプロパティが全部あるとは限りませんし、ここに無いものが入ってくるかもしれません。たとえばクライアントが動画エンコーダーだったら swfUrl や pageUrl は無いでしょうし。

audioCodecs には以下の値 (フラグ) を論理和 (OR) で組み合わせた値が入ってきます。 フラグ名は仕様書に書いてあるので一応ここにも書いていますが、データとしては特に使いません。

値	フラグ名	対応コーデック
0x0001	SUPPORT_SND_NONE	リニア PCM
0x0002	SUPPORT_SND_ADPCM	ADPCM
0x0004	SUPPORT_SND_MP3	MP3
0x0020	SUPPORT_SND_NELLY8	Nellymoser 8kHz モノラル
0x0040	SUPPORT_SND_NELLY	Nellymoser 5 ～ 44kHz
0x0200	SUPPORT_SND_NELLY16	Nellymoser 16kHz モノラル
0x0400	SUPPORT_SND_AAC	AAC
0x0800	SUPPORT_SND_SPEEX	Speex
0x0FFF	SUPPORT_SND_ALL	全部

表 15. connect コマンドの audioCodecs

videoCodecs にも以下の値 (フラグ) を論理和 (OR) で組み合わせた値が入ってきます。 こちら

のフラグ名もデータとしては特に使いません。

値	フラグ名	対応コーデック
0x0004	SUPPORT_VID_SORENSON	Sorenson H.263
0x0008	SUPPORT_VID_HOMEBREW	Screen Video
0x0010	SUPPORT_VID_VP6	On2 VP6
0x0020	SUPPORT_VID_VP6ALPHA	On2 VP6 (アルファチャンネル付き)
0x0040	SUPPORT_VID_HOMEBREWV	Screen Video V2
0x0080	SUPPORT_VID_H264	AVC(H.264)
0x0FFF	SUPPORT_VID_ALL	全部

表 16. connect コマンドの videoCodecs

videoFunction にも以下の値 (フラグ) を論理和 (OR) で組み合わせた値が入ってきます。 詳しい意味はよくわかりません……。

値	フラグ名	内容
0x0001	SUPPORT_VID_CLIENT_SEEK	クライアントがフレーム精度のシークができることを表します。

表 17. connect コマンドの videoFunction

プロパティで重要なのは app と tcUrl でしょう。 一つのサーバーでは複数のアプリケーションをホスティングできるので app でどのアプリケーションに接続したいか判別します。

tcUrl ではサーバー名、アプリケーション名に加えて追加のパスも指定されています。 この URL はプロトコル名 :// ホスト名 : ポート番号 / アプリケーション名 / インスタンス名の形で入ってきます。 プロトコル名は RTMP でつないでいるので当然 rtmp、ホスト名はサーバーのホスト名、ポート番号は標準の 1935 の場合は省略されます。アプリケーション名は app に入っているのと同じで、インスタンス名はアプリケーションの複数のインスタンスがある場合にどのインスタンスに接続するかを指定するものです。省略した場合はデフォルトのインスタンスに接続します。

例 の rtmp://example.com/liveapp/livestream の う ち、example.com は サ ー バ ー 名、liveapp はアプリケーション名で、livestream がインスタンス名です。

インスタンスについては特に規定はないので、アプリケーションで適当に処理します。 複数のインスタンスに対応するか、対応する場合はどんな名前にするか、名前をどうやってクライアントに伝えるかはアプリケーション次第です。

場合によっては tcUrl には ?user=name&pass=1234 のようにパラメーターが付く場合があります。これらのパラメーターの処理はアプリケーションによります。

サーバーは connect コマンドを受け取ったら結果を返す必要があります。

結果は _result または _error コマンドをクライアントに送るのですが、その前に User Control Message で Stream Begin イベントを通知しておきましょう。 というのも、connect コマン

ドを受け取った時点ではサーバーからクライアントへのメッセージ送信は初めてのはずなので、サーバーからクライアントへの通信にメッセージストリーム ID0 番を使いますよーと通知しておきます。これはメッセージストリーム ID0 番でも必要なようです。無くても動く気はしますけど……。

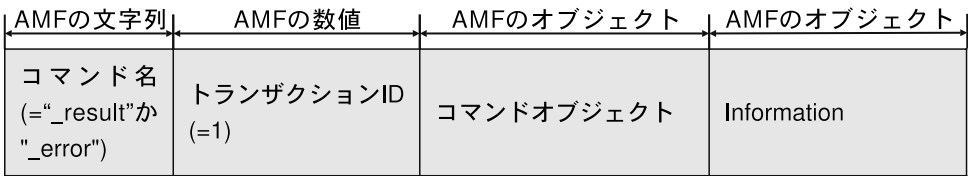


図 32. connect コマンドの結果の内容

コマンド名には _result または _error を入れます。使い分けについて詳しく書いてないのですが、まあ普通に接続できれば _result、アプリケーション名やインスタンス名が間違っていたりその他対応できない理由により続けられそうにない場合は _error を返せばいいのではないでしょうか。

トランザクション ID は connect コマンドに 1 が入ってきたはずなので、そのまま 1 にして返します。

コマンドオブジェクトにはサーバー側の接続情報が入ります。 またそのあとに Information として結果の追加情報が入ります。

結果のコマンドオブジェクトに以下のような情報が入ります。

プロパティ名	型	内容	値の例
fmsVer	文字列	サーバーのバージョン (?)	FMS/3,5,5,2004
capabilities	数値	不明	31
mode	数値	不明	1

表 18. connect コマンド結果のコマンドオブジェクトのプロパティ

全部不明です。仕様書には何も書いてないので……。 サーバーに接続するとこんな値が返ってきてるっぽいというだけで、中身は不明です。 クライアントを作る場合には無視するしかないでしょう。 サーバーを作る場合には同じような値を返すしかないんじゃないかな……。

Information には以下のような情報が入ります。

プロパティ名	型	内容	値の例
level	文字列	成功したかどうかを格納します。 成功の場合は status、失敗の場合は error を入れます。	status
code	文字列	エラーコード (もしくは成功した状態コード) を格納します。	NetConnection.Connect.Success
description	文字列	エラーコードに対応した人間が読める メッセージを格納します。	Connection succeeded
data	オブジェクト	サーバーの情報を格納します。	{ "version", "3,5,5,2004" }

プロパティ名	型	内容	値の例
clientId	数値	クライアント毎に割り当てた ユニークな ID を格納します。	12356
objectEncoding	数値	主に使用するオブジェクトエンコード方式 (AMF0 か AMF3) を格納します。 AMF0 は 0、AMF3 は 3 を入れます。	3

表 19. connect コマンド結果の Information のプロパティ

こちらも仕様書には内容はほぼ書いてないんですが、まあまあ分かります。 data、clientId、objectEncoding についてはなんとなく入ってますが本当に必要かどうかは謎です。

level・code・description がエラーや成功状態についての詳細を格納します。 description は人が読むメッセージなのでなんでもいいです。 code もアプリケーションによってしまうのでなんでも良いと言えばいいんですが、ある程度決まったコードがあるのでそれを使うのが良いでしょう。 level はコードによって status か error が決まっているのでそれを使います。

connect コマンドの返信に入る code や level は以下のようなものがあります。

code	level	意味
NetConnection.Connect.AppShutdown	error	アプリケーションが終了しています。
NetConnection.Connect.Failed	error	接続試行が失敗しました。
NetConnection.Connect.InvalidApp	error	アプリケーション名が無効です。
NetConnection.Connect.Rejected	error	アプリケーションへのアクセス許可がありません。
NetConnection.Connect.Success	status	正常に接続できました。

表 20. connect コマンド結果の code と level

これらの値を受け取った時にクライアントがどう反応すべきかは特に規定されていません。

接続失敗した時にはサーバー側は _error コマンドを返すべきですが、そのあとどうしたらいいかも特に規定されていません。実装しようとするとどうしたらいいのかと悩むところですが、どうしようもないのでまあ切るしかないんじゃないでしょうか……。

5.12. createStream コマンド (NetConnection)

connect コマンドで接続を確立したら、createStream コマンドでサーバーにメッセージストリーム ID を割り当ててもらって、NetStream を使えるようにします。

createStream コマンドはクライアントからサーバーに送られます。 NetConnection のコマンドなのでメッセージストリーム ID は 0 番で送ります。 チャンクストリーム ID は 2 番以外を使ってください。



図 33. createStream コマンドの内容

コマンド名は `createStream` です。 createStream コマンドは戻り値を受け取りますので、トランザクション ID には `0` 以外を入れます。connect コマンドと違って `1` である必要はありません。コマンドオブジェクトには `null` が入ります。

サーバー側はこのコマンドを受け取ったら、新しいメッセージストリーム ID を割り当てて返します。 新しいメッセージストリーム ID は他と被ってなければ (ついでに `0` 番でもなければ) 何番でもかまいません。

サーバーからの返信は以下のようになります。

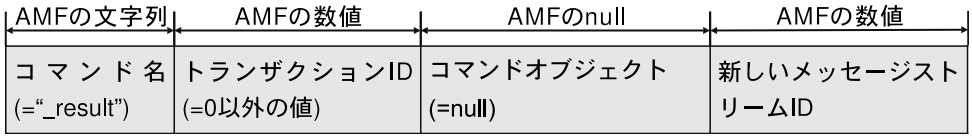


図 34. createStream コマンドの結果の内容

コマンド名は `_result` です。 トランザクション ID には createStream コマンドに入ってきたものがそのまま入ります。 コマンドオブジェクトには `null` が入ります。 最後に新しいメッセージストリーム ID を数値で入れます。

クライアントは受け取った新しいメッセージストリーム ID を使って、以降の NetStream コマンドを呼び出すことができます。

5.13. deleteStream コマンド (NetConnection)

使わなくなったメッセージストリーム ID は deleteStream ストリームコマンドで削除を要求します。

deleteStream コマンドはクライアントからサーバーに送られます。 NetConnection のコマンドなのでメッセージストリーム ID は `0` 番で送ります。 チャンクストリーム ID は `2` 番以外を使ってください。

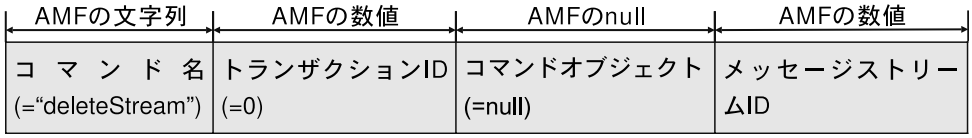


図 35. deleteStream コマンドの内容

コマンド名は `deleteStream` です。 戻り値は不要なので、トランザクション ID には `0` を入れま

す。 コマンドオブジェクトには null が入ります。 最後に削除するメッセージストリーム ID を数値で入れます。

仕様書には NetStream のコマンドと書いてあるのですが、実際は NetConnection のコマンド (メッセージストリーム ID が 0 番) として送られるようです。そりゃそうよな。

5.14. close コマンド (NetConnection)

接続を閉じるのを要求します。

close コマンドはサーバーとクライアントどちらからも送ることができるんじゃないかと思います。 思います、というのは仕様には名前だけ書いてあって中身について書いてないからです……。クライアントから送られることがあるかは不明ですけど。

NetConnection のコマンドなのでメッセージストリーム ID は 0 番で送ります。チャンクストリーム ID は 2 番以外を使ってください。



図 36. close コマンドの内容

コマンド名は close です。 戻り値は不要なので、トランザクション ID には 0 を入れます。 コマンドオブジェクトには null が入ります。その後に何か値が続くかは不明ですがたぶん続きません。

これを受け取った側は接続を切ります。

5.15. その他のコマンド (NetConnection)

その他のコマンドとして、アプリケーション定義のコマンド呼び出しを行うことができます。

先に connect コマンドを成功させていないと呼び出しできません。 NetConnection のコマンドなのでメッセージストリーム ID は 0 番で送ります。 チャンクストリーム ID は 2 番以外を使ってください。

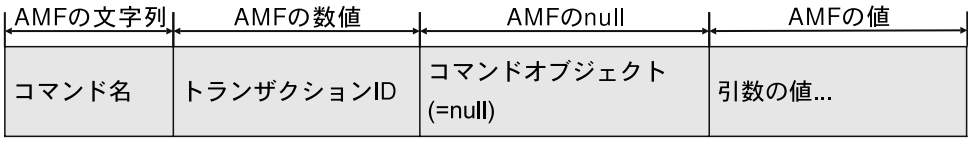


図 37. アプリケーション定義コマンドの内容

コマンド名にはアプリケーションで指定したコマンド名が入ります。アプリケーション定義なのでなんでもいいんですが、既存のコマンドと被るのは明らかにまずいでしょう。 トランザクショ

ン ID には戻り値が不要なら 0、必要なら 0 以外の適当な値が入ります。 コマンドオブジェクトはアプリケーション定義でなんかオブジェクトが入ります。無ければ null が入ります。 たぶん常に null が入ってそうな気がします。

引数の値にはアプリケーションで指定した追加の引数が入ってきます。 アプリケーション固有の物なので内容は決まっていません。

戻り値を返す場合は以下のような形で返します。内容は決まっていません。

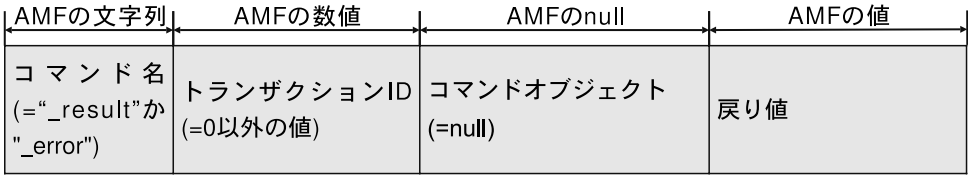


図 38. アプリケーション定義コマンドの結果の内容

コマンド名には _result または _error を入れると良いと思います。たぶん。仕様にはなんとも書いてませんが。

トランザクション ID はリクエストに入ってきたものをそのまま入れます。

コマンドオブジェクトはわかりませんが、たぶん null で大丈夫かと思います。

_result で返す場合は戻り値にはアプリケーション固有の値を入れればよさそうです。

_error で返す場合には次のような値の入ったオブジェクトを入れます。 必要であれば他の値を入れることも可能です。

プロパティ名	型	内容	値の例
level	文字列	成功したかどうかを格納します。 成功の場合は status、失敗の場合は error を入れます。	error
code	文字列	エラーコード (もしくは成功した状態コード) を格納します。	NetConnection.Call.Failed
description	文字列	エラーコードに対応した 人間が読めるメッセージを格納します。	Call failed

表 21. その他のコマンド結果の code と level

関数呼び出しの失敗時のコードは NetConnection.Call.Failed くらいしか規定されてません。

その他のコマンドとして、仕様には書かれていないものの巷で実際送られてくるものがいくつかあるので紹介します。 詳しくはどこにも書いてないので、具体的にどう処理したらいいものかわからないのですが、特に処理しなくても動きます……。

5.15.1. releaseStream コマンド

(NetConnection)

配信時にクライアントから送られてきたりします。

コマンドオブジェクトは null で、引数にはファイル名が入った文字列が入ります。

このファイル名にこれから配信したいから使ってたら解放してください、という感じの通知なんじゃないかと思われます。

5.15.2. FCPublish コマンド (NetConnection)

配信時にクライアントから送られてきたりします。

コマンドオブジェクトは null で、引数にはファイル名が入った文字列が入ります。

このファイル名にこれから配信するつもりです、という感じの通知のように見えます。

5.15.3. FCUnpublish コマンド (NetConnection)

配信終了時にクライアントから送られてきたりします。

コマンドオブジェクトは null で、引数にはファイル名が入った文字列が入ります。

このファイル名への配信を終わります、という感じの通知のように見えます。

5.16. onStatus コマンド (NetStream)

ここから NetStream のコマンドになります。

NetStream のコマンドなのでメッセージストリーム ID は createStream で割り当てた番号を使います。

onStatus コマンドはサーバーからクライアントに送信されるコマンドです。 NetStream のコマンドは多くが戻り値を使用しません。 かわりにサーバーは onStatus コマンドで結果や状態をクライアントに通知するようになっています。

中身はだいたい _result とか _error と同じです。

AMFの文字列	AMFの数値	AMFのnull	AMFのオブジェクト
コマンド名 (=“onStatus”)	トランザクションID (=0)	コマンドオブジェクト (=null)	Info

図 39. onStatus コマンドの内容

コマンド名には onStatus が、トランザクション ID は 0、コマンドオブジェクトには null が入

ります。

Info には以下のような値の入ったオブジェクトを入れます。 必要であれば他の値を入れることも可能です。

プロパティ名	型	内容	値の例
level	文字列	成功したかどうかを格納します。 成功の場合は status、失敗の場合は error を入れます。	error
code	文字列	エラーコード (もしくは成功した状態コード) を格納します。	NetStream.Play.FileNotFound
description	文字列	エラーコードに対応した 人間が読めるメッセージを格納します。	File Not Found

表 22. onStatus コマンドの Info のプロパティ

onStatus コマンドは特定のコマンドの戻り値ではなくサーバーの状態を通知するだけのコマンドです。 前に要求したコマンドのどれかが失敗したぽい……というくらいにしか通知されないので処理するのが面倒ですが、コードを見るとコマンド名に近い物が入っているので、ここから頑張って判別するしかないです。

5.17. play コマンド (NetStream)

play コマンドはサーバーに動画や音声の再生 (送信) をリクエストします。

NetStream のコマンドなのでメッセージストリーム ID は createStream で割り当ててもらった番号を使います。

AMFの文字列	AMFの数値	AMFのnull	AMFの文字列	(AMFの数値)	(AMFの数値)	(AMFの真偽値か数値)
コマンド名 (="play")	トランザクション ID (=0)	コマンドオブジェクト (=null)	ストリーム名	再生開始時刻	再生時間	プレイリストのリセット

図 40. play コマンドの内容

コマンド名には play が、トランザクション ID は 0、コマンドオブジェクトには null が入ります。

ストリーム名には再生したいファイル名を文字列で指定します。 ファイル名の指定にはなぜかフォーマットによる制限があり、flv ファイルを再生したい場合は拡張子を含まないファイル名のみ (sample など)、mp3 ファイルを再生したい場合は拡張子を含まないファイル名の頭に mp3: を付けたもの (mp3:sample など)、mp4 のファイルを再生したい場合は拡張子まで含んだファイル名の頭に mp4: を付けたもの (mp4:sample.m4v など) を指定することとなっています。

ストリーム名には文字列でなく false が入ってくる場合もあります。この場合は再生停止です。残りのパラメーターは無視されます。

再生開始時刻にはどこから再生開始するかを数値 (秒単位) で指定します。この値は省略でき、

その場合は -2 が指定されたものとします。 ライブ配信を再生する場合には -2 または -1 を指定します。-2 を指定した場合はライブ配信が行なわれている場合はそれを、行われていない場合は録画があれば録画を先頭から再生開始します。-1 を指定した場合はライブ配信が行われている場合のみ再生開始します。 0 以上の数値を指定すると、録画されているもののみ指定の秒数から再生開始します。

再生時間には再生する秒数を数値で指定します。この値は省略でき、その場合は -1 が指定されたものとします。 -1 を指定した場合はライブ配信または録画の終了まで再生し続けます。0 を指定した場合は、1 フレームだけ再生します。それより大きい値を指定した場合はその秒数分 (それより早く終わったら最後まで) だけ再生します。

プレイリストのリセットにはすぐ再生するか、プレイリストに入れるかを真偽値または数値で指定します。 この値は省略でき、その場合は true または 1 が指定されたものとします。 false や 0 を指定した場合は、既に再生中のものがあればそれが終わってから今回指定したファイルが再生されるように、内部的なプレイリストに追加します。 true や 1 を指定した場合は、既に再生中のものや内部的なプレイリストは破棄し、すぐに指定したファイルが再生されるようします。 2 と 3 はいまいち意味がよくわからないのですが、2 はプレイリストを維持したまま、3 はプレイリストを破棄し、どちらも全てのストリームメッセージを一括で返すようです。ストリームメッセージを一括で返す、の意味がよくわからない……。

play コマンドは戻り値を要求しません。そのかわりサーバーは onStatus コマンドで実行結果を通知します。

再生が上手くできる場合には、code が NetStream.Play.Reset の onStatus コマンドと、code が NetStream.Play.Start の onStatus コマンドの 2 つを送ります。level はどちらも status です。Reset が 0 や false だった場合は NetStream.Play.Start の onStatus コマンドだけ送ります。

再生が上手くできない場合には、code が NetStream.Play.StreamNotFound か NetStream.Play.Failed の onStatus コマンドを送ります。level は error です。

再生停止を実行した場合には、code が NetStream.Play.Stop の onStatus コマンドを送ります。level は status です。

再生開始できた場合は、この後にサーバーから Video Message や Audio Message が垂れ流されます。 クライアントはそれを受け取って再生しましょう。

再生してたファイルが終わったら、サーバーは code が NetStream.Play.Complete の onStatus コマンドを送ります。level は status です。

5.18. play2 コマンド (NetStream)

play2 コマンドはサーバーに動画や音声の再生および切り替えをリクエストします。

基本的には play コマンドと同じく動画や音声の再生ができますが、現在再生中の動画や音声から上手いこと切り替えるためのパラメーターが増えています。

NetStream のコマンドなのでメッセージストリーム ID は createStream で割り当ててもらった番号を使います。

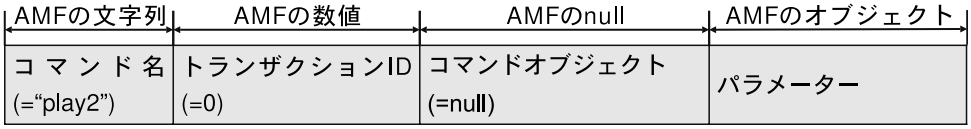


図 41. play2 コマンドの内容

コマンド名には play が、トランザクション ID は 0、コマンドオブジェクトには null が入ります。

パラメーターには以下のような値の入ったオブジェクトを入れます。

プロパティ名	型	内容
streamName	文字列	新しく再生したいファイル名を指定します。play コマンドの Stream Name と同じです。
oldStreamName	文字列	切り替え元のファイル名を指定します。切り替え前のファイルが無い場合は null を指定します。
start	数値	新しく再生したいファイルの再生開始秒数を指定します。play コマンドの Start と同じです。
len	数値	新しく再生したいファイルの再生する秒数を指定します。play コマンドの Duration と同じです。
transition	文字列	新しく再生したいファイルへの切り替え方法を指定します。
offset	数値	transition が switch だった時に何秒後に切り替えるかを指定します。-1 を指定すると可能な限りすぐに切り替えます。

表 23. play2 コマンドのパラメーターのプロパティ

transition 以外はだいたい play コマンドと同じですね。 transition には次のような値が指定できます。

transition の値	意味
append	新しいファイルをプレイリストの最後に追加して再生開始します。play コマンドの Reset を false にしたのと同じです。
appendAndWait	新しいファイルをプレイリストに追加しますが再生は開始しません。resume でプレイリストを再生開始することができます。
reset	プレイリストを破棄して新しいファイルを再生開始します。play コマンドの Reset を true にしたのと同じです。
resume	前回の接続で止まったところから再生を再開します。
stop	再生を停止してプレイリストを破棄します。play コマンドのストリーム名を false にしたのと同じです。
swap	プレイリスト内の oldStreamName を streamName で置き換えます。oldStreamName が既に再生中の時は失敗します。

transition の値	意味
switch	再生中の oldStreamName から streamName に再生を切り替えます。oldStreamName がプレイリストに入っている場合は streamName で置き換えます。

表 24. play2 コマンドの transition

resume は接続エラーか何かで止まっちゃった時に続きから再生するのに使うようです。実現するにはサーバーがだいぶ頑張らないといけないですね。

switch は同じ動画の別ビットレートに切り替えるのに使います。全く別の動画への切り替えは想定していないようなので失敗する可能性があります。

サーバー側の返信は play コマンドと同じ onStatus コマンドを送って通知します。

再生の切り替えが行なわれた場合は、code が NetStream.Play.Transition、level が status の onStatus コマンドが送られるようです。

5.19. pause コマンド (NetStream)

pause コマンドはサーバーに再生中の動画や音声の一時停止と再開をリクエストします。

NetStream のコマンドなのでメッセージストリーム ID は createStream で割り当ててもらった番号を使います。



図 42. pause コマンドの内容

コマンド名には pause が、トランザクション ID は 0、コマンドオブジェクトには null が入ります。

一時停止 / 再開フラグには一時停止するのか再開するのかが真偽値で入ります。true の場合は一時停止、false の場合は再開です。

停止位置にはクライアントがどこで一時停止したのかをミリ秒単位の数値で指定します。 再開する時にはサーバーはこの時間以降のデータのみ送信します。

pause コマンドもサーバーは onStatus コマンドで実行結果を通知します。

一時停止できた時は code が NetStream.Pause.Notify、再開できた時は code が NetStream.Unpause.Notify の onStatus コマンドを送ります。level はどちらも status です。

仕様によると操作が失敗した時は _error コマンドを返すことになっていますが、トランザクション ID が 0 なんで _error コマンドは返るか怪しいですね。実際どういう動作をするかはわかりません……。

5.20. seek コマンド (NetStream)

seek コマンドはサーバーに再生中の位置の変更をリクエストします。

NetStream のコマンドなのでメッセージストリーム ID は `createStream` で割り当ててもらった番号を使います。

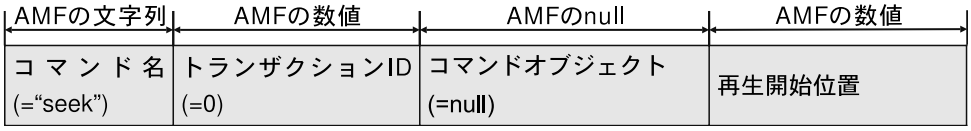


図 43. seek コマンドの内容

コマンド名には `seek` が、トランザクション ID は `0`、コマンドオブジェクトには `null` が入ります。

再生開始位置には再生開始する位置をミリ秒単位の数値で指定します。 プレイリストに複数のメディアが入っている場合には、プレイリストの先頭から時間を指定するようです。

位置の変更ができた場合は `code` が `NetStream.Seek.Notify`、`level` が `status` の onStatus コマンドを送ります。この場合、`seekPoint` というプロパティに変更した位置を数値で格納します。

失敗した場合は `code` が `NetStream.Seek.Failed` か `NetStream.Seek.InvalidTime`、`level` が `error` の onStatus コマンドを送ります。 `NetStream.Seek.InvalidTime` の場合は `details` というプロパティに、変更できる最後の位置を数値で格納します。

仕様によると操作が失敗した時は `_error` コマンドを返すことになっていますが、トランザクション ID が `0` で `_error` コマンドは返せないので、失敗した時も onStatus コマンドで通知するのが正しいんじゃないかと思います。

5.21. receiveAudio コマンド (NetStream)

receiveAudio コマンドはクライアントが音声を受信するかどうかをサーバーに通知します。

NetStream のコマンドなのでメッセージストリーム ID は `createStream` で割り当ててもらった番号を使います。

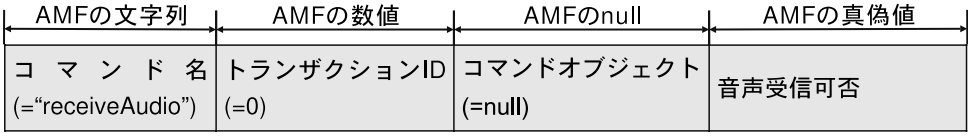


図 44. receiveAudio コマンドの内容

コマンド名には `receiveAudio` が、トランザクション ID は `0`、コマンドオブジェクトには `null` が入ります。

音声受信可否には音声を受信するかどうかを真偽値で指定します。true の場合は受信する、false の場合は受信しません。

受信しないを通知した場合はサーバーはそれ以降音声データを送りません。受信するを再設定すれば音声の送信が再開されます。

受信するを設定した場合はサーバーは code が NetStream.Seek.Notify と NetStream.Play.Start の onStatus コマンド 2 つを送ります。level はどちらも status です。 受信しないを設定した場合はサーバーからの返答は何もありません。

5.22. receiveVideo コマンド (NetStream)

receiveVideo コマンドはクライアントが動画を受信するかどうかをサーバーに通知します。

NetStream のコマンドなのでメッセージストリーム ID は createStream で割り当ててもらった番号を使います。

AMFの文字列	AMFの数値	AMFのnull	AMFの真偽値
コマンド名 (=“receiveVideo”)	トランザクションID (=0)	コマンドオブジェクト (=null)	動画受信可否

図 45. receiveVideo コマンドの内容

コマンド名には receiveVideo が、トランザクション ID は 0、コマンドオブジェクトには null が入ります。

動画受信可否には動画を受信するかどうかを真偽値で指定します。true の場合は受信する、false の場合は受信しません。

受信しないを通知した場合はサーバーはそれ以降動画データを送りません。受信するを再設定すれば動画の送信が再開されます。

受信するを設定した場合はサーバーは code が NetStream.Seek.Notify と NetStream.Play.Start の onStatus コマンド 2 つを送ります。level はどちらも status です。 受信しないを設定した場合はサーバーからの返答は何もありません。

5.23. publish コマンド (NetStream)

publish コマンドはサーバーにクライアントから動画 (や音声) の配信を開始することを通知します。

NetStream のコマンドなのでメッセージストリーム ID は createStream で割り当ててもらった番号を使います。



図 46. publish コマンドの内容

コマンド名には publish が、トランザクション ID は 0、コマンドオブジェクトには null が入ります。

ストリーム名には配信するファイル名 (ストリーム名) を文字列で指定します。 指定できる名前は play コマンドと同じです。 各種動画配信アプリではストリーム名はストリームキーという名前の項目で設定できることが多いです。 空文字列や null の場合もあります。

配信タイプには配信種別を文字列で指定します。次のどれかが指定できます。 null だった場合は live と同じです。

配信タイプの値	内容
live	ライブ配信です。ファイルには保存しません。既に同じ名前のファイルがあった場合は削除されます。
record	サーバーにファイルとして保存します。既に同じ名前のファイルがあった場合は上書きします。
append	サーバーにファイルとして保存します。既に同じ名前のファイルがあった場合は追記します。

表 25. publish コマンドの配信タイプ

publish コマンドもサーバーは onStatus コマンドで実行結果を通知します。

サーバーが受け入れる場合には code が NetStream.Publish.Start、level が status の onStatus コマンドを返します。

失敗した場合には code が NetStream.Publish.BadName、level が error の onStatus コマンドを返します。

再生開始できた (code が NetStream.Publish.Start の onStatus を受け取った) あとはクライアントから Video Message や Audio Message が垂れ流されます。 サーバーはそれを受け取って保存するなりライブ配信用にバッファリングするなりしましょう。

6. AMF0/3

度々出てきたものの、まだ解説していなかった AMF0 と AMF3 について説明しましょう。

AMF は Action Message Format の略で、どちらも ActionScript の値をエンコード (シリアライズ) するフォーマットです。 ActionScript を知らない人向けに簡単に説明しておくと、ActionScript は JavaScript とだいたい同じです。つまり AMF は JavaScript の値をバイナリにシリアライズするフォーマットと思って OK です。

RTMP Chunk Stream では整数値ぐらいしか使いませんでしたけど、AMF では文字列、数値 (浮動小数点数および整数)、真偽値、null、undefined、日時、キーと値を組にした配列やオブジェクトなんかをエンコードできます。

AMF0 は ActionScript2 まで用のフォーマット、AMF3 は ActionScript3 用のフォーマットで、AMF3 はエンコードできる型が増えてたりしますが、RTMP で使う分には大して違いはありません。できることは大した違いがないのに中身は別物です。実装が面倒なだけですけど、仕方ないですね……。

ここだけの話、Adobe の製品はともかく他の実装は AMF0 を主に使いますので、AMF3 は実装しなくても動くかもしれません。

この先は AMF0 と AMF3 両方について詳細を説明していきます。 ただ、RTMP では使わない部分については細かく説明しません。 必要であれば仕様書を確認してください。

6.1. AMF0

ここでは AMF0 の詳細について説明していきます。

まず AMF0 内での数値のエンディアンについてですが、全部ネットワークバイトオーダー (ビッグエンディアン) になっています。RTMP と同じです。良かった。

AMF0 での文字列は全部 UTF-8 で、先頭に 2 バイトまたは 4 バイトの整数値で文字列の長さ (バイト単位) が付きます。 つまり 2 バイト (または 4 バイト) の文字列長とそれに続く実際の UTF-8 文字列で構成されます。 文字列の長さは基本的に 2 バイトの物が使われます。

AMF0 のデータはまず 1 バイトのマーカーから始まります。 マーカーは値のデータ型を表わします。

マーカー	型
0	数値
1	真偽値
2	文字列
3	オブジェクト

マーカー	型
5	null
6	undefined
7	参照
8	連想配列
9	オブジェクト終端
10	普通の配列
11	日時
12	長い文字列
13	Unsupported
15	XML
16	型付きオブジェクト
17	AMF3 への切り替え

表 26. AMF0 のマーカー

マーカーの後は型ごとのデータが続きます。

これだけです。簡単ですね。AMF0 の値が複数続く時にはまたマーカーから始まるだけです。

型ごとのデータ内容も説明していきましょう。

6.1.1. 数値型 (マーカー :0)

数値型は浮動小数点数が入ります。

ActionScript2 では JavaScript と同じく数値は全部浮動小数点数なので、AMF0 でも整数型はありません。

データとしては 8 バイトで浮動小数点数 (IEEE-754 の倍精度浮動小数点数) が入っています。

6.1.2. 真偽値型 (マーカー :1)

真偽値型は `true` か `false` を表す値が入ります。

データとしては 1 バイトの整数値で、`0` なら `false`、それ以外は `true` として扱います。

6.1.3. 文字列型 (マーカー :2、12)

文字列型は UTF-8 の文字列が入ります。

マーカーが 2 番の文字列では、データとして 2 バイトの長さ付き文字列が入ります。つまり最初に文字列の長さ (バイト単位) を表す 2 バイト整数が入り、その後に実際の文字列データが入ります。 文字列の長さが 2 バイト分しかないので 655,535 バイトまでの文字列だけが格納できます。

マーカーが 12 番の長い文字列では、データとして 4 バイトの長さ付き文字列が入ります。最初

に文字列の長さ (バイト単位) を表す 4 バイト整数が入り、その後に実際の文字列データが入ります。こちらは 655,535 バイトより長い文字列が格納できます。

空文字列の場合は文字列の長さとして 0` が入り、残りの文字列データがないものとなります。

6.1.4. オブジェクト型 (マーカー :3)

オブジェクト型はプロパティ名と値の組を複数格納できるデータです。 プロパティ名は文字列のみですが、値は AMF0 で表現できる好きな値を保持できます。

データはまず 2 バイトの長さ付き文字列が入り、その後に AMF0 でエンコードされた値 (AMF0 のマーカーから始まる) が入ります。 長さ付き文字列の方がプロパティ名で、AMF0 でエンコードされた値の方がプロパティの値になります。そのあとには、また長さ付き文字列と AMF0 でエンコードされた値が続き……というようにプロパティ名と値の組が続きます。 最後はプロパティ名に空文字列 (長さ 0) と値にオブジェクト終端 (マーカー 9 番) が来るので、ここで終わりです。

マーカー 9 番のオブジェクト終端は、データは 0 バイトです。 つまりマーカーの 1 バイト分だけでオブジェクトの終端を表す値です。 ここだけでしか使われません。

オブジェクト型のプロパティの値にオブジェクトが入るような、入れ子構造にもなることに注意しましょう。

6.1.5. null 型 (マーカー :5)

null 型は null そのものを表します。よく考えるとなんだか難しそうな言葉になってしまっていますが、真面目に考えないようにしましょう。

データはありません。つまりマーカーの 1 バイト分だけで、null を表します。

6.1.6. undefined 型 (マーカー :6)

undefined 型は undefined そのものを表します。これまた難しそうな言葉になってしまっていますが、そういうものとして受け入れましょう。

データはありません。つまりマーカーの 1 バイト分だけで、undefined を表します。

6.1.7. 参照型 (マーカー :7)

参照型は、以前に出現したオブジェクトや配列への参照を表します。

同じオブジェクトや配列をもう一度参照するのに使うのですが、RTMP ではほぼ使わないでしょうから詳しくは省略します。

データは参照インデックスとして 2 バイトの整数値が入ります。

6.1.8. 連想配列型 (マーカー :8)

連想配列は、ほぼオブジェクト型と同じで、プロパティ名と値の組を複数格納できるデータです。連想配列は ECMAScript の配列とも呼ばれますが、ECMAScript(=JavaScript) の配列ってほぼオブジェクトと同じ扱いなので、AMF0 に入っていても同じになります。

データは、まずエントリー数 (プロパティ名と値の組数) が 4 バイトの整数値で入ります。 そのあとに 2 バイトの長さ付き文字列と AMF0 でエンコードされた値 (AMF0 のマーカーから始まる) が交互に、それぞれエントリー数分だけ続きます。

オブジェクト型は空文字列のプロパティ名とオブジェクト終端の値が来たら終わりでしたが、連想配列型は最初に組数が付いているのが違いです。 逆に言うと違いはそれだけです。

6.1.9. 普通の配列型 (マーカー :10)

普通の配列型は、値のリストです。普通は配列と言うとこれだよなあというデータです。

データは、まずエントリー数 (値の数) が 4 バイトの整数値で入ります。 そのあとに AMF0 でエンコードされた値 (AMF0 のマーカーから始まる) がエントリー数分だけ続きます。

6.1.10. 日時型 (マーカー :11)

日時型はそのまま日時を表します。 内部的には 1970 年 1 月 1 日 00:00:00(UTC) からの経過時間 (ミリ秒単位) の数値を保持します。

データは、日時を表す 8 バイトで浮動小数点数 (IEEE-754 の倍精度浮動小数点数) とタイムゾーンを表す 2 バイト整数が入ります。 ただしタイムゾーンは常に 0 が入るので UTC しか表現できません。

6.1.11. Unsupported 型 (マーカー :13)

Unsupported 型はエラーマーカーです。 なんかエンコードできないデータをエンコードしようとした時にこのマーカーを入れます。

通常は使いませんし、これに出会った時はエラーとしてください。

データはありません。

6.1.12. XML 型 (マーカー :15)

XML のドキュメントを表す型です。が、AMF0 ではただの長い文字列として格納されます。

データは長い文字列 (マーカー :12) と同じで、4 バイトの長さ付き文字列が入ります。

6.1.13. 型付きオブジェクト型 (マーカー :16)

型付きオブジェクトはクラスのインスタンスを表します。RTMP ではクラスとか気にしないので、クラス名が付いてくるだけで扱いは普通のオブジェクトと同じです。

データは、まずクラス名が 2 バイトの長さ付き文字列で入ります。その後にオブジェクト型 (マーカー 3 番) と同じように、プロパティ名と値の組が続きます。 プロパティ名に空文字列 (長さ 0) と値にオブジェクト終端 (マーカー 9 番) が来たら終わりです。

6.1.14. AMF3 への切り替え (マーカー :17)

これ以降の値は AMF3 でエンコードされていることを表すマーカーです。RTMP では使わないでしょう。

データはありません。

6.2. AMF3

次に AMF3 についても説明します。

AMF3 内での数値のエンディアンも全部ネットワークバイトオーダー (ビッグエンディアン) になっています。

6.2.1. AMF3 の整数値

AMF3 では U29 という整数値の可変長エンコードが使われます。

これが大変めんどくさいのですが、最大 29 ビットの整数値を 1 ～ 4 バイトで表すフォーマットです。

各最上位 1 ビットを次のバイトもあるかどうかのフラグに使い、最上位 1 ビットが 0 だったらそこまでで終了、1 だったら次のバイトも有りとなります。 ただし 4 バイト目 (あれば) は最後なので最上位 1 ビットも数値データに含めます。

説明では難しいので図を見た方が早いですね。

エンコードする値	1 バイト目	2 バイト目	3 バイト目	4 バイト目
0x00000000 ～ 0x0000007F	0xxxxxxx	なし	なし	なし
0x00000080 ～ 0x00003FFF	1xxxxxxx	0xxxxxxx	なし	なし
0x00004000 ～ 0x001FFFFF	1xxxxxxx	1xxxxxxx	0xxxxxxx	なし
0x00200000 ～ 0x3FFFFFFF	1xxxxxxx	1xxxxxxx	1xxxxxxx	xxxxxxxx

表 27. U29 の値範囲とバイナリ表現

二進数の方の x は数値データとして採用するビットを表します。

これで小さな値はなるべく少ないバイト数で表しつつ、大きな値まで対応することができます。

6.2.2. AMF3 の文字列

AMF3 の文字列表現も解説しておきます。

AMF3 の文字列も AMF0 と同じように先頭に長さを表す整数値が付いた UTF-8 文字列を扱います。

ただし AMF0 とは違って長さを表す整数値が U29 型です。 さらに U29 で表された整数値の下位 1 ビットが、参照かどうかのフラグに使われます。 下位 1 ビットが `0` だったら残りの上位 (最大)28 ビット分を参照インデックスとして扱います。 下位 1 ビットが `1` だったら残りの上位 (最大)28 ビット分を文字列の長さ (バイト単位) として扱います。

数値が参照インデックスだった場合 (下位 1 ビットが `0` だった場合) は後には文字列データは続きません。一連の AMF3 データ中で出てきた (参照でない) 文字列の、参照インデックス番目 (0 番目が最初) のものを値として採用してください。 つまり AMF3 のデータをデコードする時に (参照でない) 文字列が出てきたら、あとから参照されるものとして記憶しておく必要があります。

数値が参照インデックスでなかった場合 (下位 1 ビットが `1` だった場合) は、残りの上位 28 ビット分だけそのあとに文字列データが続きます。あと参照されるものとして記憶しておいてください。

たとえば、“hello” という文字列は `0x0B` “hello” となります。`0x0B` は “hello” の長さである `5` を左に 1 ビットシフトして最下位ビットに 1 を論理和 (OR) したものです (`(5 << 1) | 1`)。

同様に空文字列は `0x01` になります。

6.2.3. AMF3 の値

整数値と文字列のエンコードが面倒な以外は AMF3 も AMF0 もあまり変わりません。

AMF3 の値も型を表す 1 バイトのマーカーから始まります。

マーカー	型
0	undefined
1	null
2	false
3	true
4	整数値
5	浮動小数点数値
6	文字列
7	XML ドキュメント
8	日時
9	配列
10	オブジェクト
11	XML

マーカー	型
12	バイト配列

表 28. AMF3 のマーカー

マーカーの後は型ごとのデータが続きます。 AMF3 の値が複数続く時にはまたマーカーから始まるだけです。

マーカーの番号や型こそ違いますが、形は AMF0 と同じですね。

6.2.4. undefined 型 (マーカー :0)

undefined 型は undefined そのものを表します。

データはありません。マーカーの 1 バイト分だけで、undefined を表します。

6.2.5. null 型 (マーカー :1)

null 型は null そのものを表します。

データはありません。マーカーの 1 バイト分だけで、null を表します。

6.2.6. false 型、true 型 (マーカー :2、3)

false 型、true 型は真偽値のそれぞれ false と true を表します。

データはありません。マーカーの 1 バイト分だけで false または true を表します。

AMF0 と違い、false・true それぞれにマーカーの番号が割り振られているおかげでマーカーの 1 バイト分だけで真偽値を表すことができます。

6.2.7. 整数値型 (マーカー :4)

符号無し整数値を表します。

データは U29 での整数値が入ります。

負の整数値や U29 の最大値である 0x3FFFFFFF を越える整数値は入りません。 それらは浮動小数点数としてエンコードされます。

6.2.8. 浮動小数点数値型 (マーカー :5)

浮動小数点数値型は浮動小数点数を表します。 負の整数値や U29 の最大値である 0x3FFFFFFF を越える整数値も浮動小数点数値型として入れられます。

データとしては 8 バイトで浮動小数点数 (IEEE-754 の倍精度浮動小数点数) が入っています。

6.2.9. 文字列型 (マーカー :6)

文字列型は文字列を表します。

データとしては、AMF3 での文字列形式が入ります。

6.2.10. XML ドキュメント型、XML 型 (マーカー :7、11)

XML ドキュメント型と XML 型は XML のデータを表します。 何が違うかというと ActionScript でのクラスの違いのようですが、AMF3 上ではどちらもただの文字列として格納されます。

データとしては、AMF3 での文字列形式が入ります。

6.2.11. 日時型 (マーカー :8)

日時型はそのまま日時を表します。 内部的には 1970 年 1 月 1 日 00:00:00(UTC) からの経過時間 (ミリ秒単位) の数値を保持します。

データは、まず U29 で参照インデックスが入ります。参照インデックスの下位 1 ビットが `0` であれば、上位 28 ビット分を日時の参照インデックスとして扱い、これ以降のデータはありません。参照インデックスの下位 1 ビットが `1` であれば、上位 28 ビットは無視し、このあとに 8 バイトで日時を表す浮動小数点数 (IEEE-754 の倍精度浮動小数点数) が入ります。

6.2.12. 配列型 (マーカー :9)

配列型は ActionScript3 の配列を表します。

AMF0 では連想配列型と普通の配列型がありましたが、AMF3 では両方が混ざった配列を表現できます。その分複雑です。つらい。

データは、まず U29 で参照インデックスまたはエントリー数が入ります。参照インデックスまたはエントリー数の下位 1 ビットが `0` であれば、上位 28 ビット分を配列の参照インデックスとして扱い、これ以降のデータはありません。

参照インデックスまたはエントリー数の下位 1 ビットが `1` だった時は、上位 28 ビット分を配列の (普通の配列部分の) エントリー数として扱います。 普通の配列部分のエントリー数であるところが注意点です。連想配列部分のエントリー数は不定です。

配列が参照でなかった場合は、エントリー数の後には連想配列部分のエントリーが入ります。

連想配列部分はキーとなる文字列と AMF3 でエンコードされた値の組が任意の数だけ続き、空文字列で終わります。AMF0 のオブジェクト型と似ていますが、最後は空文字列だけで終わることに気をつけてください。オブジェクト終端の値はありません。キーに空文字列が来たらその瞬間に

終わります。 連想配列部分が空の場合は、空文字列が 1 つだけが入ります。

連想配列部分の後には普通の配列部分が続きます。普通の配列部分はエントリー数分だけ AMF3 でエンコードされた値が入ります。

6.2.13. オブジェクト型 (マーカー :10)

オブジェクト型は ActionScript3 のオブジェクトを表します。 クラスが前提になった形式になっているため複雑です。つらい。

データは、まず U29 で参照インデックス等の数値が入ります。参照インデックス等の数値の下位 1 ビットが `0` であれば、上位 28 ビット分を配列の参照インデックスとして扱い、これ以降のデータはありません。

参照インデックス等の数値の下位 2 ビットが `01` であれば、上位 27 ビット分をクラス情報の参照インデックスとして扱います。 以降にはプロパティの値だけが入ります。

参照インデックス等の数値の下位 3 ビットが `111` であれば、クラスは外部クラスであることを示します。上位 26 ビット分は無視されます。後には文字列で外部クラス名と、外部クラス固有のバイナリデータが続きます。外部クラス固有のバイナリデータは長さも内容不定で、外部クラスについてサーバー・クライアント間で合意が無いと使えません。つまりどういうことかというと、RTMP ではほぼ使えないので気にすんなってことです。

参照インデックス等の数値の下位 4 ビットが `0011` であれば、クラスは通常クラスであることを表し、`1011` であれば動的プロパティを持つクラスであることを表します。残りの上位 25 ビット分はクラスの固定プロパティ数を表します。いずれもこの後にはクラス情報とプロパティの値が続きます。

クラス情報は参照インデックス等の数値の後に続きます (下位 4 ビットが `0011` か `1011` の時だけ)。まずクラス名が文字列で入り、クラスのプロパティ名が参照インデックス等の数値に入っていた固定プロパティ数分だけ文字列で続きます。

クラス情報の後は固定プロパティの値が続きます。 クラス情報の固定プロパティ数分だけ AMF3 でエンコードされた値が入ります。

クラスが動的プロパティを持つクラスだった時のみ、最後に動的プロパティの値が入ります。プロパティ名の文字列と AMF3 でエンコードされた値の組が任意の数だけ続き、空文字列で終わります。 AMF0 のオブジェクト型と似ていますが、最後は空文字列だけで終わることに気をつけてください。 オブジェクト終端の値はありません。プロパティ名に空文字列が来たらその瞬間に終わります。

AMF3 のオブジェクト型はフォーマットだけ見るとめっちゃ難しいですね。 まず、外部クラスは使わないので無視しましょう。 そうすると、オブジェクト自体が参照のパターン、クラス情報だけ参照のパターン、全部入っているパターンの 3 つに分けられます。

全部入っているパターンでは、データはクラス情報とプロパティの値が続いています。 いくつ

プロパティの値が入っているか (そして動的プロパティがあるかどうか) はクラス情報に定義されています。

クラス情報だけ参照のパターンでは、データはプロパティの値だけが入ります。 いくつプロパティの値が入っているか (そして動的プロパティがあるかどうか) はクラス情報に定義されていますが、クラス情報はここには入ってきません。以前に出てきたクラス情報をインデックスで参照してください。

オブジェクト自体が参照のパターンでは、参照インデックスの後にデータは続きません。クラス情報もプロパティの値も前に出てきたオブジェクトを参照してください。

RTMP では特にクラスが指定されているところがないので AMF3 のオブジェクト型が使われるのかはかなり疑問ですが、もし入ってきたとしてもクラスは無視されて、単にプロパティ名と値の組として扱われるでしょう。こんなに面倒なのに。

6.2.14. バイト配列型 (マーカー :12)

バイト配列型はバイナリデータを格納します。

データとしては、AMF3 での文字列形式と同じように最初に長さまたは参照インデックスを表す U29 の整数値が入り、下位 1 ビットが **1** だったら残り上位の 28 ビット分をバイト長としたバイナリデータが続きます。

下位 1 ビットが **0** だったら残り上位の 28 ビット分を参照インデックスとして、一連の AMF3 データ中で出てきた (参照でない) バイト配列型データの、参照インデックス番目 (0 番目が最初) のものを値として採用してください。

6.2.15. AMF3 の参照について

AMF3 では文字列・日時・配列・オブジェクト・クラス情報・バイト配列・XML ドキュメント・XML が参照インデックスで参照できるようになっています。

この参照インデックスってどんな値が入るかですが、最初に出現した参照でない値を 0 番、次の出現した参照でない値が 1 番……という形でテーブルを保持して、そのテーブルへのインデックス (0 番とか 1 番とか) が入ります。

参照テーブルは文字列・クラス情報・その他の 3 つがあって、それぞれで別個に 0 番からインデックスを振っていきます。 その他の参照テーブルには日時・配列・オブジェクト・クラス情報・バイト配列・XML ドキュメント・XML の全部がごっちゃに入ります。 インデックスが間違って型が違う値を参照してたらどうするんでしょうね。知らん。

ただ、この参照は RTMP ではほぼ使いません。

参照テーブルは最初に出てきた値を 0 番としないといけないのですが、メッセージ毎に AMF0 と AMF3 を切り替えることができることからも、一メッセージ毎に参照テーブルをリセットするの

が妥当なところでしょう。[12]

ただその場合、参照を使うところがないのです。メッセージ一つに同じ値が複数回出現するかというと、文字列でもしかしたらあるかも？程度です。なので RTMP で使う場合はせいぜい文字列の参照について管理して、他は諦めるという手もあるでしょう。 自分から AMF3 にエンコードする場合も、RTMP ではなるべく参照は使わない方が良いと思われます。

6.3. RTMP Message での型の扱いについて

AMF0 と AMF3 について説明しましたが、RTMP Message では型はあまり細かく扱われません。

RTMP Message で使われる型は文字列・数値・真偽値・オブジェクトくらいです。

数値は浮動小数点数も整数も同じに扱われます。

オブジェクトはプロパティ名と値の組として扱えればなんでもかまいません。 普通のオブジェクト、連想配列、型付きのオブジェクトのどれでも同じものとして扱ってください。 自分からエンコードする場合もどれでもいいです。まあ連想配列か普通のオブジェクトのどっちかが良いと思いますが。

想定する型と違う型の値が来た時にどう処理すべきかですが、ActionScript の型変換規則に従うのが良さそうです。この規則に収まらない場合 (オブジェクトを期待するところに数値が来たとか) はエラー値として扱うしかないでしょう。

ActionScript の型変換規則の概要を挙げておきます。

変換元の値	**数値への変換結果**
真偽値	`true` の場合は `1`、それ以外の場合は `0`
日時	1970 年 1 月 1 日 00:00:00(UTC）からの経過時間 (ミリ秒単位)
`null`	`0`
オブジェクト	`0`
文字列	数値に変換できる場合は数値、それ以外の場合は `NaN`
`undefined`	`NaN`

表 29. ActionScript の数値型への変換規則

変換元の値	**真偽値への変換結果**
文字列	空文字列の場合は `false`、それ以外の場合は `true`
`null`	`false`
数値	`NaN` または `0` の場合は `false`、それ以外の場合は `true`

12 この " 最初 " の定義ははっきり規定はされてないのですが、もし接続の最初からだとした場合は、通信中に出現する値を全部保持しつづける参照テーブルを持つはめになり現実的ではありません。

変換元の値	真偽値への変換結果
オブジェクト	`true`

表 30. ActionScript の真偽値型への変換規則

変換元の値	文字列への変換結果
配列	すべての要素を文字列化したもの
真偽値	“true” または “false”
日時	日時の文字列表現
数値	数値の文字列表現
`null`	“null”
オブジェクト	“[object Object]”

表 31. ActionScript の文字列型への変換規則

7. RTMP で通信しよう

プロトコルの中身についての解説は一通り終わりました。

でもいざ通信してみようとすると、いつ何をどう送ったらいいものやらわかりませんね。

ここからは目的毎にどう通信していけばいいのか見ていきましょう。

7.1. 共通処理

まずは、何をするにも必ず通る道を見ていきましょう。

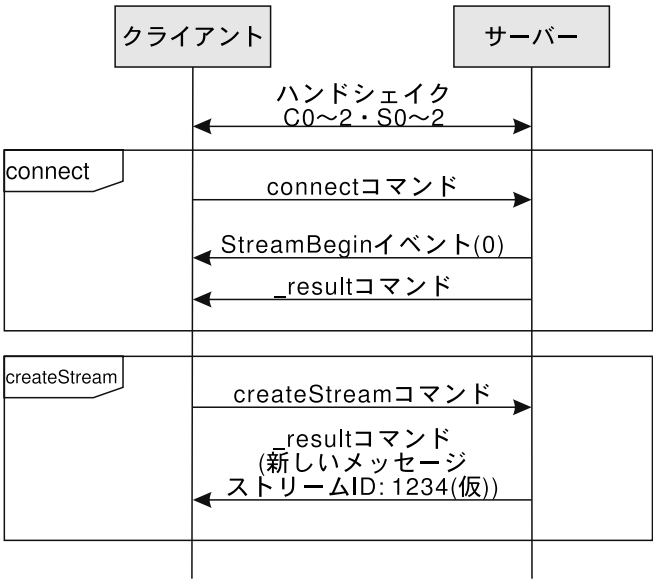


図 47. 接続からの共通処理

まず RTMP のハンドシェイクから始まります。ハンドシェイクについては前に説明したので詳細は省きます。

ハンドシェイクができたら、なにはさておきクライアントは connect コマンドを発行します。connect コマンドは NetConnection のコマンドなのでメッセージストリーム ID は常に `0` 番です。

connect コマンドが来たらサーバーは `_result` コマンド (失敗してたら `_error` コマンド) を返します。 と行きたいところですが、サーバーが新しいメッセージストリーム ID を使う場合はそれが `0` 番だとしてもクライアントに通知する必要があります。 そのため、`_result` コマンドを返す前に User Control Message の StreamBegin イベントでメッセージストリーム ID`0` 番を使うことを通知します。

StreamBegin イベントでメッセージストリーム ID`0` 番を使うことを通知したら、メッセージストリーム ID`0` 番で `_result` コマンドを返し、接続できたことを通知します。

クライアントはこの後 NetStream のコマンドを使いたくなるのですが、NetStream のコマンドを使うには 0 番以外のメッセージストリーム ID が必要になるので、createStream コマンドで新しいメッセージストリーム ID の発行を要求します。 createStream コマンドも NetConnection のコマンドなのでメッセージストリーム ID は常に 0 番です。

サーバーは新しくメッセージストリーム ID を適当な番号で割り当て _result コマンドで返します。 この _result コマンド自身のメッセージストリーム ID は 0 番で、中に次から使えるメッセージストリーム ID(1 番以降の値、ここでは 1234 番としましょう) が入っています。

これが一番簡単な流れです。最低限これだけあればたぶん動くんじゃないかな……。

実際にはもうちょっとだけ複雑になります。というのも接続確立した時点でいくらか通信設定をしたくなるので、RTMP Control Message が飛び交います。

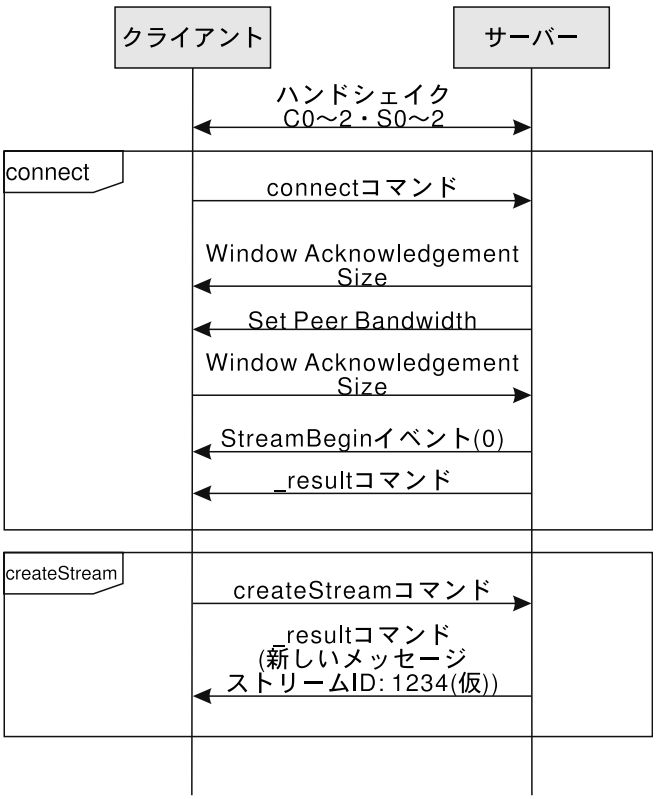


図 48. 接続からの共通処理詳細

クライアントが connect コマンド発行したところで、Window Acknowledgement Size や Set Peer Bandwidth などのメッセージが飛び交うようになりました。 必ずしもこの順番やこのタイミングで行う必要はありませんが、Window Acknowledgement Size は早い段階でやっておいた方が良さそうなので、ここで指定するのが良いでしょう。 Window Acknowledgement Size や Set Peer Bandwidth はクライアントから先に送ってもかまいません。

ついでにこのタイミングで Set Chunk Size を送ってチャンクの最大長を大きくしてもいいですね。もちろん大きなデータを送るタイミングでやるのでもかまいません。

7.2. 切断処理

切断する時は大して手間はかかりません。

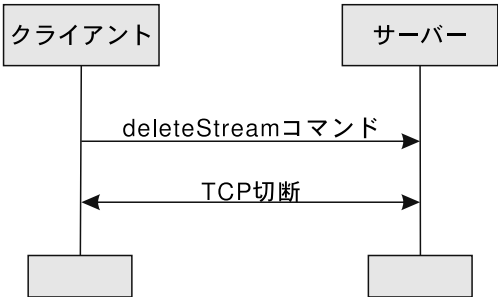


図 49. 切断処理

createStream コマンドで割り当ててもらったメッセージストリーム ID を、deleteStream コマンドで解放要求しましょう。 deleteStream コマンドは NetConnection コマンドなのでこれ自身のメッセージストリーム ID は `0` 番です。中に解放したいメッセージストリーム ID(ここでは仮に 1234 番) を指定します。

あとは接続を切りましょう。ばちーん。

7.3. クライアントからサーバーへ動画配信したい

2019 年にもなって RTMP を使う理由はクライアントからサーバーへ動画配信したいという目的くらいしかないでしょうし、まずこれから見ていきましょう。

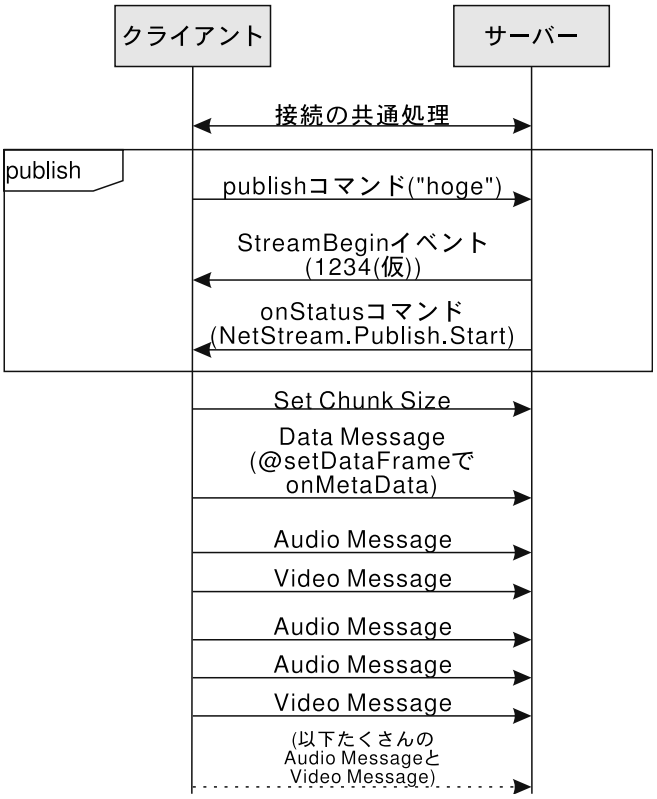


図 50. publish 処理

まず共通処理まで済ませておいて NetStream を使えるようにします。 割り当てたメッセージストリーム ID は仮に 1234 だとしておきます。実際には 1234 に限ったものではなく、1 以降でサーバーが適当に割り当てた番号になります。

クライアントが publish コマンドで配信開始を要求します。 この publish コマンドのメッセージストリーム ID は 1234 番です。 配信先のファイル名は hoge だとしておきましょう。

サーバーは配信開始の準備が出来たら onStatus コマンドでクライアントに通知します。 ここでは onStatus コマンドはメッセージストリーム ID1234 番で送り返したいのですが、サーバーが新しいメッセージストリーム ID を使う場合には StreamBegin イベントでクライアントに通知する必要があるのでした。 先に User Control Message の StreamBegin イベントでサーバーも 1234 番を使うことを通知し、それから onStatus コマンドを返します。 onStatus コマンドには code が NetStream.Publish.Start、level が status のオブジェクトを入れときます。

クライアントは code が NetStream.Publish.Start な onStatus コマンドを受け取ったら、あとは動画のデータを送りまくります。

その前に Set Chunk Size を送っておくのをおすすめします。チャンクの最大長設定はここでやらなくともっと以前に十分なサイズを設定していればそれで良いですし、最悪初期値の 128 バイトのままでも動作はします。しかしながら動画や音声のデータは大きくなりがちで、あまりチャンクの分割が発生すると通信効率が落ちますので、動画や音声データを送る前には適当にチャンクの

最大長を大きくしておくのをおすすめします。

動画や音声を送り始めたいところですが、まずメタデータを最初に入れておく必要があります。Audio Message や Video Message より先に Data Message で `onMetaData`(やその他のメタデータ) を送りましょう。 その際に、クライアントからサーバーに配信する時には `@setDataFrame` という名前で Data Message を送る必要があります (中身は `onMetaData` とその値)。単純に `onMetaData` という名前の Data Message ではだめなので注意してください。

あとは終わるまで Audio Message や Video Message を送り続けて、気が済んだら切断処理をして終わりです。

受け取った Audio Message や Video Message やメタデータをサーバー側がどうするかは、サーバーの自由です。 普通はなんかしらのフォーマットで書き出すなり変換して保持して、他のクライアントに配信するとか保存することになると思いますが、どんなフォーマットで書き出すかやどうやって変換するかはサーバー (およびサーバーアプリケーション) の実装によるので RTMP では規定されません。

これが最低限の流れですが、実際の動画配信ソフトではいくつか NetConnection コマンドでの追加の通知が挟まります。

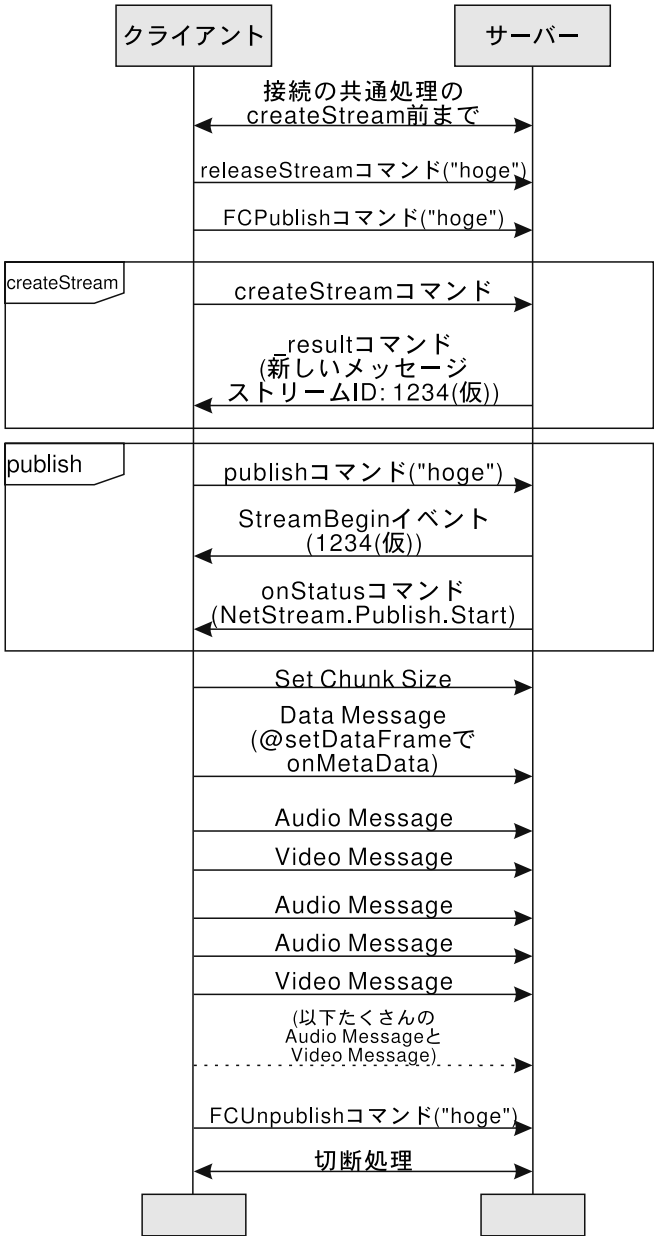


図 51. publish 処理詳細

createStream コマンドの前に releaseStream コマンドと FCPublish コマンドが挟まりました。また切断処理の前に FCUnpublish コマンドも追加されました。 これらは NetConnection のコマンドなので常にメッセージストリーム ID0 番で送られます。 内容はいずれも配信ファイル名 (今回の例では hoge) を引数に一つ持つだけのコマンドです。具体的な処理は特に書いてないのですが、サーバー側での追加の処理をするための通知コマンドなので、送っておいた方が良いでしょう。サーバーがこれらを受け取ったら、何かやりたかったらやればいいし、不要なら何もする必要はありません。

動画配信はこれで全部です。 細かい実装はわりと面倒なのですが、流れとしてはそれほど複雑ではないです。

7.4. 動画の視聴をしたい (サーバーからクライアントへの動画の配信をしたい)

2019 年にもなって、サーバーからクライアントへの動画の配信を RTMP でしたいということはないでしょう。 他の方法を使ったほうがいいよ？

とはいえ、せっかく play コマンドの説明も書いたので解説はしておきましょう。

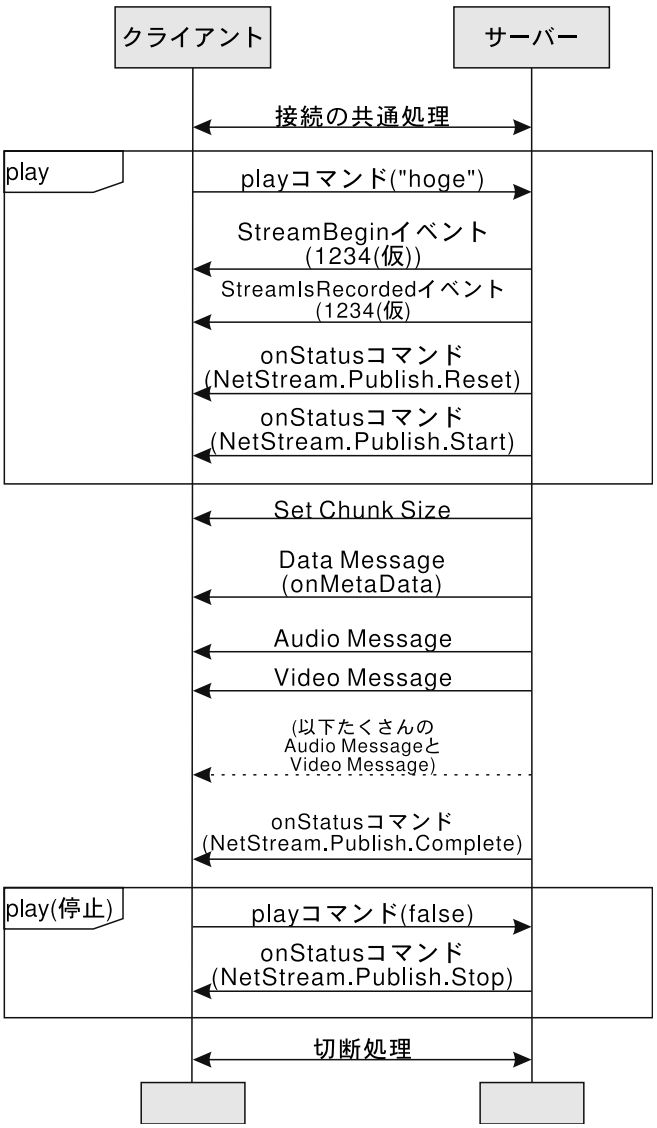


図 52. play 処理

まず共通処理まで済ませて NetStream を使えるようにしたら、クライアントが play コマンドで再生開始を要求します。 この play コマンドのメッセージストリーム ID は 1234 番です。 配信先

のファイル名はここでも hoge だとしておきましょう。

サーバーは再生開始の準備が出来たら onStatus コマンドでクライアントに通知します。

その前に再生するファイルが録画されていたものであれば、StreamIsRecorded イベントで 1234 番のストリームが録画であることをクライアントに通知しておきます。ライブ配信の場合にはこの通知は送りません。

onStatus コマンドはメッセージストリーム ID1234 番で送り返したいのですが、先に User Control Message の StreamBegin イベントでサーバーも 1234 番を使うことを通知します。

play コマンドでプレイリストのリセットが要求されていた場合は code が NetStream.Play.Reset、level が status の onStatus コマンドを先に送ります。

そして再生開始できたことを通知するため code が NetStream.Play.Start、level が status の onStatus コマンドを送ります。

その後はサーバーから動画や音声データを送ります。

と、行きたいところですが、その前にサーバーからも Set Chunk Size を送っておきましょう。もちろんもっと前にもっと以前に十分なサイズを設定していればここでは必要ないですし、なんならもっと後でもいいんですが、まあこの辺りが妥当じゃないでしょうか。

動画や音声データを送り始める時には最初にメタデータを送っておきます。 Data Message で onMetaData(やその他のメタデータ) を送りましょう。 サーバーからクライアントに再生用にメタデータを送る時には、@setDataFrame を使わずに直接 onMetaData という名前の Data Message を送ってください。 @setDataFrame を使うのはクライアントからサーバーへの配信時だけです。

あとは終わるまで Audio Message や Video Message を送り続けます。

ファイル (やライブ配信) が終わったら、サーバーから code が NetStream.Play.Complete、level が status の onStatus コマンドが送られます。

もしくは、クライアントがファイル名に false を指定した play コマンドを送ると停止になり、サーバーから code が NetStream.Play.Stop、level が status の onStatus コマンドが送られます。

いずれにせよ、こうなると再度再生開始しない限りは Audio Message や Video Message は送られてきません。 切断処理をして終了しましょう。

play2 コマンドの方は基本的に同じなので特に解説しません。 切り替え系の機能が増えていますが、詳細な動作に関してはサーバーの実装に強く依存しそうな感じです。

8. FLV ファイル

RTMP で動画がやりとりできるくらいの通信方法を解説しましたが、実際に実装してみると、本当にちゃんと送受信できてるかテストしようとして詰まることがあります。

というのも、送信した動画データを準備したり、受け取った動画データがちゃんと再生できる形になっているのか確認するのに動画ファイルの読み書きをしたいのですが、動画ファイルのフォーマットを調べたり形式を変換したりは結構大変です。 最終的にはやることになるかもしれませんが、とにかく RTMP で送受信できるか確認したいだけの時にはなんとか手間をかけずに済ませたいところです。

そんなあなたにおすすめのファイルフォーマットが FLV です！

FLV は Flash で使われていた動画フォーマットですが、なんと RTMP Message のフォーマットがほぼそのまま入ります。 なので RTMP で受信した動画データにちょこっとヘッダーを付けて保存するだけで FLV ファイルが出来上がりますし、FLV ファイルを読み込んで RTMP で送信するのも比較的簡単です。

FLV ファイルの再生はオープンソースのメディアプレイヤー (VLC とか) で可能ですし、ffmpeg などを使えば他のフォーマットとの変換もできるので、少なくともテスト用には十分な使い勝手を持っています。

ここでは FLV ファイルの構造を紹介します。

8.1. FLV ファイルの構造

FLV ファイルの構造は以下のようになっています。



図 53. FLV ファイルの構造

最初に FLV ファイルのヘッダが入り、その後はバックポインターと FLV タグが交互に続いていきます。 バックポインターは 4 バイトで、一つ前のメッセージのサイズ (ヘッダを含む) をバイト単位で格納します。 最初のバックポインターだけは 0 が入ります。

この構造は前にちらっと書きましたが、RTMP Message の Aggregate Message とほぼ同じです。FLV タグというのはほぼ RTMP Message そのままなので、Aggregate Message は FLV ファイルヘッダと最初のバックポインターが無いくらいの違いしかありません。 Aggregate Message の読み書きとコードが共用できそうですね。 まあ Aggregate Message は使われませんけど。

ちなみに FLV ファイルに入っている数値データも基本的にはビッグエンディアンです。

8.1.1. FLV ファイルヘッダー

FLV ファイルヘッダーは以下のような構造です。

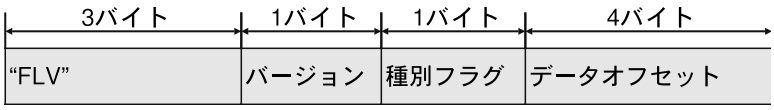


図 54. FLV ファイルヘッダーの構造

最初は "FLV" の文字、そのあとに 1 バイトのバージョン (中身は 0x01) で始まります。 次に 1 バイトの種別フラグが入ります。

種別フラグは次の値を論理和 (OR) で組み合わせたものが入ります。

種別フラグ値	内容
1	動画を含む
4	音声を含む

表 32. FLV ファイルの種別フラグ

最後に 4 バイトのデータオフセットが入ります。データオフセットにはファイル先頭からファイルボディまでのバイト数を格納します。常に 9 が入ります。

めっちゃ簡単。5 バイト目以外は固定値ですし、普通は動画も音声も入っているファイルでしょうし、全部固定値でもいいくらいですね。

8.1.2. FLV タグ

FLV タグは以下のような構造です。

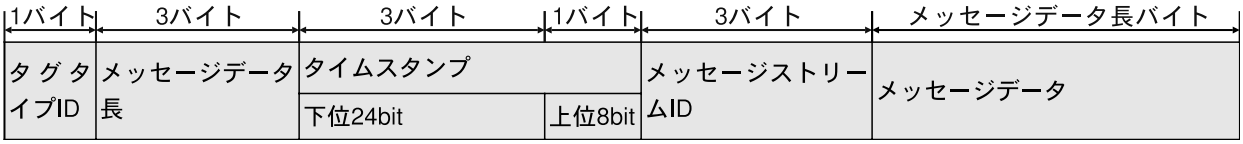


図 55. FLV タグ構造

RTMP Message Format とほぼ同じです。

最初の 1 バイトはタグタイプ ID が入ります。RTMP のメッセージタイプ ID と同じですが、8 と 9 と 18 しか入りません。

タグタイプ ID	タグ名	簡単な内容
8	Audio タグ	音声データが入るタグです。
9	Video タグ	動画データが入るタグです。
18	Script タグ	メタデータ (ビットレート等) が ActionScript のオブジェクトで入ります。エンコードは AMF0 です。

表 33. FLV タグタイプ ID

Script タグは RTMP の Data Message に相当するものですね。エンコードは AMF0 しか使えません。

次に 3 バイトのメッセージデータ長が入ります。FLV タグのヘッダを除くバイト数が入ります。

メッセージデータ長の後には 4 バイトのタイムスタンプが入りますが、格納方法が普通でないので気をつけてください。。 最初の 3 バイトにはビッグエンディアンでタイムスタンプの下位 24 分が入っており、次の 1 バイトはタイムスタンプの上位 8bit 分として扱われます。 タイムスタンプは FLV ファイル先頭からの経過秒数をミリ秒単位で格納します。

タイムスタンプの後にメッセージストリーム ID が 3 バイト続きます。FLV ファイルの場合はメッセージストリーム ID は常に 0 が入ります。

残りはタグタイプ ID により Audio タグか Video タグか Script タグのデータが続きます。

それぞれのデータ内容は RTMP Message の Audio Message、Video Message、Data Message と全く同じなので省略します。 Script タグでは名前に `@setDataFrame` は使わず、`onMetaData` などをそのまま指定する点だけ気をつけてください。 また AMF0 しか使えないので、メッセージタイプ ID が **15** 番の AMF3 の Data Message を受け取った時は AMF0 に変換して **18** 番にして保存するようにしてください。

8.1.3. バックポインター

FLV ファイルヘッダーおよび FLV タグの後には 4 バイトのバックポインターが続きます。

バックポインターには直前の FLV タグのバイト数 (ヘッダも含む) を入れます。 FLV タグのヘッダが 11 バイトなので、**11+FLV** タグのメッセージデータ長が入っているはずです。 ただし FLV ファイルヘッダー直後のバックポインターだけは常に 0 が入ります。

8.2. FLV ファイルまとめ

FLV ファイルの構造は以上です。簡単ですね。

RTMP で受信したデータは FLV ファイルヘッダーと FLV タグヘッダーとバックポインターを付けるくらいで FLV ファイルになってくれます。 気をつけるところはタイムスタンプの格納方法がちょっと変なくらいです。

古い形式ではありますが、手軽に読み書きできるフォーマットなのでテスト用にはいろいろ使えるでしょう。

9. 参考文献

RTMP は仕様書だけあっても何も全然わからないので参考文献はいっぱいあるよー！

9.1. ドキュメント類

この本でもだいたい説明したとは思うのですが、やっぱり細かいところを調べるのは重要です。

さらに他の解説あれば理解が深まるでしょうから、他のドキュメントも挙げておきます。

9.1.1. RTMP の仕様書

RTMP の仕様書は公開されています。

- **Real-Time Messaging Protocol (RTMP) specification (Version 1.0)**
  https://www.adobe.com/devnet/rtmp.html
  仕様書がひどいようなことを散々言いはしましたが、とはいえだいたいは書いてあるのでまずはこれを見る必要はあるでしょう。

上記のページには RTMP の仕様に関してのライセンスも載っています。

基本的にはストリーミングで配信されている RTMP の動画や音声をぶっこ抜いたり、暗号化の回避をするような物を作ることが禁止されているだけなので、まあ問題になるようなライセンスではないでしょう。 そもそも暗号化の回避ができるような仕様まで書いてないですしね。

9.1.2. Wikipedia の RTMP のページ

Wikipedia の RTMP のページ (英語版) がなにげにわかりやすくて参考になります。

- **Real-Time Messaging Protocol From Wikipedia, the free encyclopedia**
  https://en.wikipedia.org/wiki/Real-Time_Messaging_Protocol
  仕様書にも書いていないような情報が書かれてたりしますが、はたしてその情報はいったいどこから……？

9.1.3. FLV・SWF・AMF0/3 の仕様書

RTMP の Video Message や Audio Message、Data Message なんかの中身は RTMP の仕様書には書いておらず、FLV の仕様を見る必要があります。

- **FLV and F4V File Format Specification**
  https://www.adobe.com/devnet/f4v.html
  F4V は MP4 とだいたい同じのファイルフォーマットで、最近は FLV は非推奨でこちらが使われているようです。 ただ RTMP とは直接関係ないので、読む必要があるのは仕様書の後半にある

FLV のフォーマットだけです。

Video Message や Audio Message のコーデック毎にどんなデータが入っているかは FLV の仕様書にもあまり詳しくは書いていません。 それらは SWF の仕様書を見るともう少し詳しく書いてあります。

- **SWF File Format Specification (version 19)**
  https://www.adobe.com/devnet/swf.html
  SWF ファイルも FLV なんかとちょっと似た形式ですが、ちょっと似てるだけで結構違うのでコーデック部分以外はあんまり参考にしない方が良いでしょう。 まあ RTMP とは関係なく読んでみる分には面白いですが。

AMF の仕様書も上記の SWF ファイル仕様書と同じページに置いてあります。

9.1.4. MPEG-4 の仕様書

Audio Message や Video Message に AAC だの AVC が入っているときの中身の詳細については、MPEG-4 の仕様書を見ないとわかりません。

- **ISO/IEC 14496-3:2009 Information technology – Coding of audio-visual objects – Part 3: Audio**
  https://www.iso.org/standard/53943.html
- **ISO/IEC 14496-15:2017 Information technology – Coding of audio-visual objects – Part 15: Carriage of network abstraction layer (NAL) unit structured video in the ISO base media file format**
  https://www.iso.org/standard/53943.html
  上のが AAC の仕様で、下のが AVC(H.264) を MP4 ファイルフォーマットに入れるための仕様です。

問題は MPEG-4 の仕様書は有料かつめちゃくちゃお値段が高くて、パート毎に 2 万円を越えてきます。上記の 2 パートだけ欲しくても合計 4 万円越えです。 ちょっとだけ調べたいなあというには高すぎてなかなか手が出ないので、図書館などで探して必要な部分だけコピるのがおすすめです。 図書館にもそうそうないんですけど……。

9.1.5. ActionScript のドキュメント

RTMP は ActionScript から使うのを想定した通信形式であるため、ActionScript のドキュメントが役に立ちます。というかこれも見ないとエラーコードとかわかりません。

- **NetConnection (Adobe Flash Platform 用 ActionScript 3.0 リファレンスガイド)**
  https://help.adobe.com/ja_JP/FlashPlatform/reference/actionscript/3/flash/net/NetConnection.html
- **NetStream (Adobe Flash Platform 用 ActionScript 3.0 リファレンスガイド)**
  https://help.adobe.com/ja_JP/FlashPlatform/reference/actionscript/3/flash/net/NetStream.

html
とりあえず NetConnection と NetStream クラスあたりをおさえておけばいいんじゃないかな。

あとサーバーサイド ActionScript というのもあります。これは Adobe Media Server で動かす ActionScript です。こちらのドキュメントも参照しましょう。

- **Server-Side ActionScript Language Reference**
  https://helpx.adobe.com/adobe-media-server/ssaslr/topics/server-side-actionscript-language-reference.html

9.2. ツール類

RTMP を実装して試したいという時に便利なツールや、ソースが参考にもなるオープンソースソフトウェアについても紹介しておきます。

9.2.1. Adobe Media Server

Adobe Media Server は Adobe 純正のメディアサーバーで RTMP で通信するアプリケーションを作ることができます。

有償製品ではありますが、体験版が無料でダウンロードできます。 以前は Adobe Media Server Starter という無料版がダウンロードできたのですが、執筆時点では Starter のダウンロードページに行くと 30 日間体験版のダウンロードページに行ってしまうので無くなったんでしょうか……。

Starter は接続数や接続していられる時間に強い制限が付いていて、実質開発にしか使えないバージョンでしたが、開発に使うにはとても便利でした。

- **Adobe Media Server**
  https://www.adobe.com/jp/products/adobe-media-server-family.html
  一番右下もしくは機能比較ページから Starter 版のダウンロードに行けますが、リンク先は体験版のダウンロードと出てくるので Starter 版はもうないかも？いずれにせよ要登録です。

上のサーバーサイド ActionScript のドキュメントもそうですが、Adobe Media Server の開発者向けドキュメントは無料で読めます。動かせると実際の動作が検証できるので便利なんですが、ドキュメントだけでも参考になるでしょう。

- **Adobe Media Server Developer Center**
  https://www.adobe.com/devnet/adobe-media-server.html

9.2.2. RTMPDump・librtmp

C 言語で書かれたオープンソースの RTMP 実装である librtmp と、それを使った RTMP のダンプツールの RTMPDump です。

RTMPDump はテスト用に使えますし、librtmp は結構あちこちで使われているらしいです。

- **RTMPDump**
  http://rtmpdump.mplayerhq.hu/
  オープンソースなので実装の参考にもなると思います。 RTMPDump は GPLv2 ですが、librtmp はライセンスが LGPLv2.1 なのでプロプライエタリなアプリから使うのも可能ですね。

9.2.3. Open Broadcaster Software

オープンソースの配信用ソフトです。

デスクトップ画面 (やカメラ画像) を取り込んでエンコードし配信する RTMP クライアントです。

RTMP を受信できる多くの配信サイトに対応しており、カスタムサーバーにも接続できるので自前のサーバーでテストする時に便利です。

オープンソースではありますが、RTMP 部分は librtmp を使っているので、ソースを参考にするというよりは純粋にツールとして使えば良いかと思います。

- **Open Broadcaster Software**
  https://obsproject.com/

9.2.4. FFmpeg

言わずと知れた、オープンソースのメディア操作用ツールです。

FLV と他のフォーマットで変換に使えます。

また、FFmpeg 自体も RTMP に対応しており、RTMP から動画を受信してファイルに保存したり、ファイルから読み取った動画を RTMP で配信したりというのも出来ます。 ちょっとコマンドがめんどくさいのでおすすめはしませんが。

RTMP から動画を受信して再生するには ffplay という動画再生ツールの方が便利でしょう。

また、RTMP も librtmp でなく独自で読み書きしていますし、FLV の読み書き対応もありますのでソースも参考になります。

- **FFmpeg**
  https://ffmpeg.org/

9.2.5. PeerCastStation

正直これを紹介するのはどうかとも思ったんですが一応……。

拙作の P2P でストリーミング配信するソフトウェアです。

RTMP で受け取った動画を FLV にして独自プロトコルに入れて P2P で流し、受け取ったクライアントは FLV のまま、もしくは MPEG-2 TS にしたり、また RTMP に戻してプレイヤーで視聴したりということができます。最近は HTTP Live Streaming での視聴もベータ版レベルながら実装され

ました。

ツールとしてはだいぶニッチなものなので使えるところもないと思いますが、C# で書かれた RTMP 実装や FLV(または RTMP) から MPEG-2 TS への変換としてソースは参考になる……かもしれません。

著者紹介

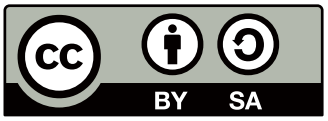
本文 :kumaryu

PeerCastStation とかいう P2P なライブ配信ソフトを作ってる人。
本を書いてるとプログラム書く暇がなくてつらい。

表紙 : ぬか

twitter: @nuka000
ゲームをつくったり絵をかいたりしています。

ライセンス



この作品の本文 (表紙を除く) は、クリエイティブ・コモンズの 表示 - 継承 4.0 国際 ライセンスで提供されています。ライセンスの写しをご覧になるには、 http://creativecommons.org/licenses/by-sa/4.0/ をご覧頂くか、Creative Commons, PO Box 1866, Mountain View, CA 94042, USA までお手紙をお送りください。

ダウンロード版について

この本の電子版が以下の URL(または右下の QR コード) からダウンロードできます。
https://www.kumaryu.net/books/douga-rtmp.html
ID: DOUGARTMP　　パスワード : 9659689040405
いつまでもダウンロードできるとは限らないのでなるべくお早めに御ダウンロードください。

奥付

タイトル :　動画を送ろう !!RTMP
発行日 :　2019 年 4 月 14 日 初版発行
サークル名 :　あれくま
発行者 :　kumaryu
連絡先
Web:　https://www.kumaryu.net/
メール :　kumaryu@kumaryu.net


あれくま
2019