

# ネットワークゲームにおける TCPとUDPの使い分け

(株)モノビット  
取締役CTO 中嶋謙互  
2017年10月



ネットワークゲームのプログラマが  
必ず疑問に思うこと



いま作っているゲームで、UDPとTCP  
のどちらを使うのが良いのだろうか



# ぐぐると

- UDPはとにかく遅延が小さい
- だから対戦格闘みたいな速い反応速度が必要なゲームはUDP
- MMOはTCP (でもUDPのもある)
- できるだけUDPのほうがいいだろ
- TCPはデバッグしやすいよ
- UDPは面倒なことが多いからTCP
- でも、あのゲームもこのゲームもUDPを使ってる
- しかし、あのゲームもこのゲームもTCPを使ってる
- 同じジャンルでもいろいろ違う
- 両方使ってるゲームもあるよ

**結論ができない・・・！**



# 簡単に決まる場合もある

- ・ ゲーム機でのアドホック対戦ならUDP
- ・ ブラウザゲームならWebSocket(つまりTCP)

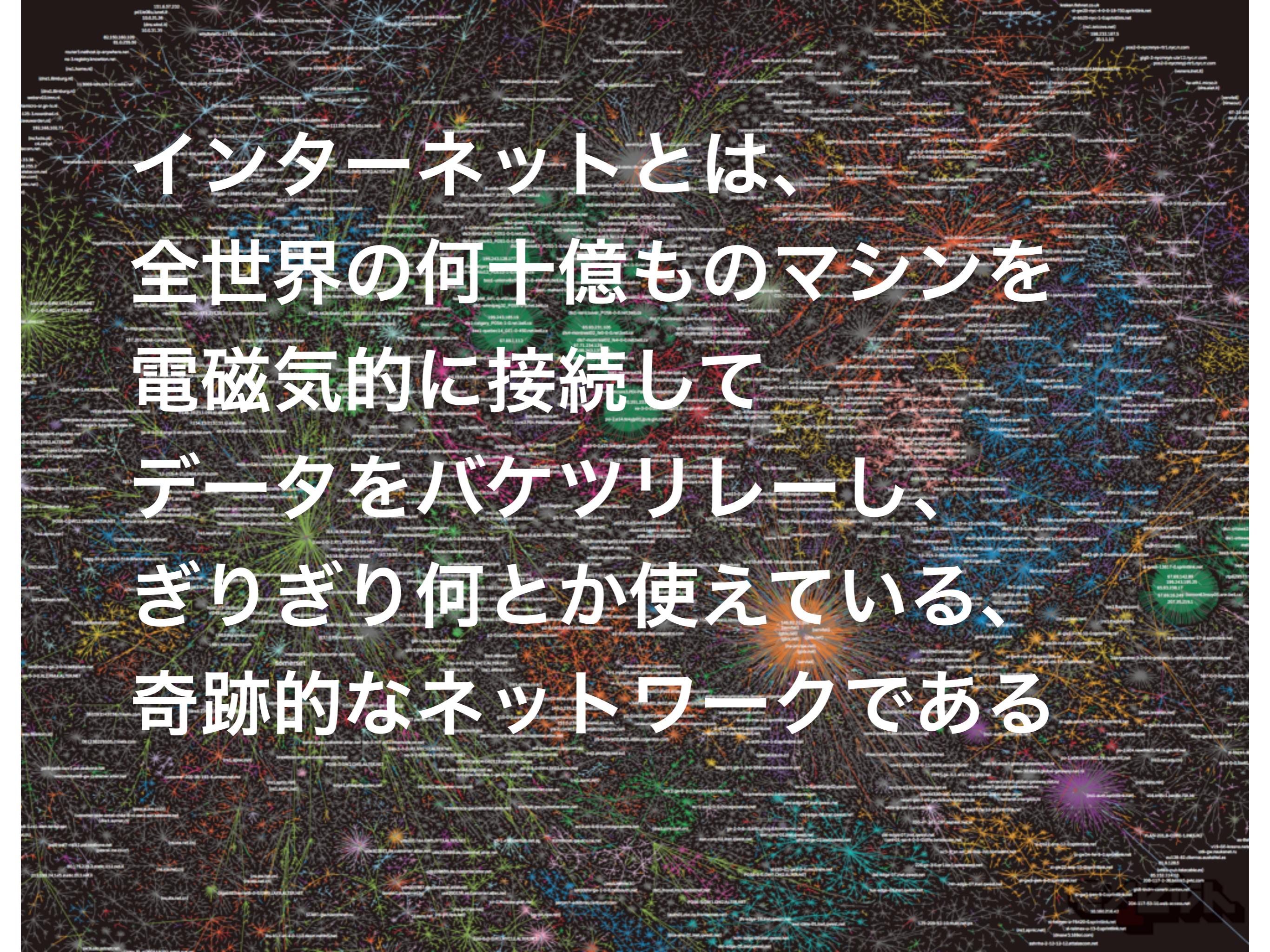
(ブラウザでのWebRTCは置いておく)



# 考える手順

- 1.インターネットの動きを知る
- 2.インターネットの基礎であるIPを知る
- 3.TCPを詳しく知る
- 4.TCPとUDPの違いを知る
- 5.ゲーム内容からプロトコルを選択する





インターネットとは、全世界の何十億ものマシンを電磁気的に接続してデータをバケツリレーし、ぎりぎり何とか使っている、奇跡的なネットワークである

# インターネットはなぜ可能か

- ・ インターネットに参加するすべてのマシンが、「インターネットプロトコル階層」に従った通信方式を採用しているから

# プロトコルの階層構造

OSI 参照モデル階層	TCP/IP 階層	実際のプロトコル階層									
アプリケーション層	アプリケーション層				DNS		TELNET	SMTP			
プレゼンテーション層		DHCP	SNMP	SSDP	プロトコル	FTP		POP3	MIME	HTTP	NNTP
セッション層											
トランスポート層	トランスポート層	UDP				TCP					
ネットワーク層	インターネット層	IP (IPv4, IPv6)		ICMP							
データリンク層	ネットワークインターフェイス層	PPP			ARP RARP	メディアアクセス制御(MAC):トーケンリングなど					
物理層	ハードウェア層	10Base, 100BaseEthernet, ISDN, ATM, フレームリレー、トーケンリング、アナログ電話等									

40年かけて人類が構築した共通財産

# 各層のおおざっぱな機能

OSI 参照モデル階層	TCP/IP 階層	実際のプロトコル階層									
アプリケーション層	アプリケーション層	DNS	TELNET	SMTP	HTTP	NNTP					
プレゼンテーション層		DHCP	SNMP	SSDP	FTP		POP3	MIME			
セッション層											
トランスポート層	トランスポート層	UDP			TCP						
ネットワーク層	インターネット層	IP (IPv4, IPv6)	ICMP								
データリンク層	ネットワークインターフェイス層	PPP	ARP RARP	メディアアクセス制御(MAC):トークンリングなど							
物理層	ハードウェア層	10Base, 100BaseEthernet, ISDN, ATM、フレームリレー、トークンリング、アナログ電話等									

ゲームのキャラデータの最新状態を送りあう

欲しいデータが世界のどこにあるか探せるようにする。  
第三者にデータを盗まれないようにする

大きなデータを送る。ポート番号を使ってプロセス単位で届ける

ルータを何段階も経由して遠くのマシンに届ける

電気ノイズによる誤りを訂正する、接続台数を増やす

マシンからマシンに電磁気の信号を届ける



# IPアドレスとルータ

55.44.33.22

## ルータ

ルー

11.22.33.44

ルータ

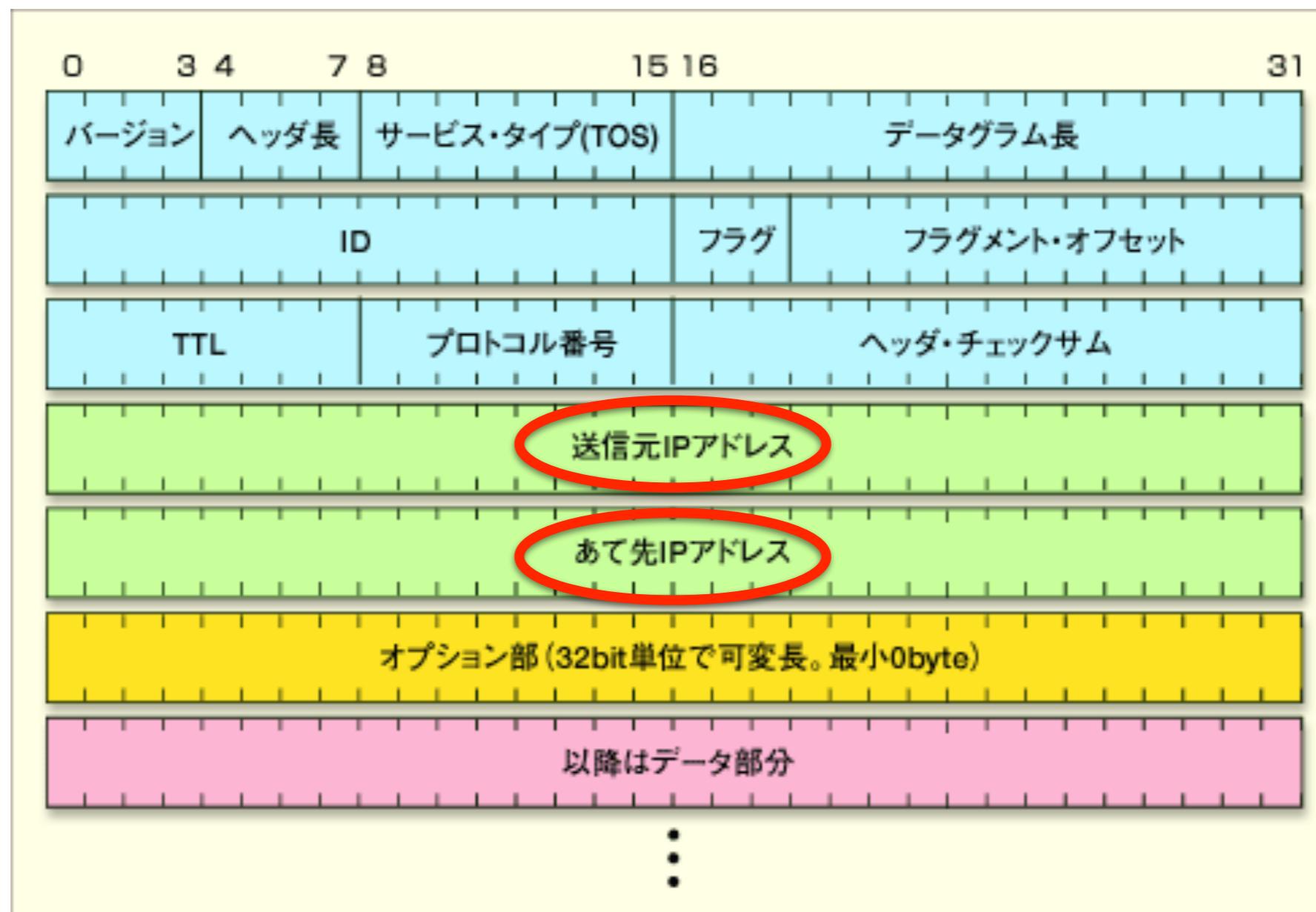
ルータ

ルータ

ルータ

IPアドレスをたよりにルータからルータへ転送されていく

# IPのパケット



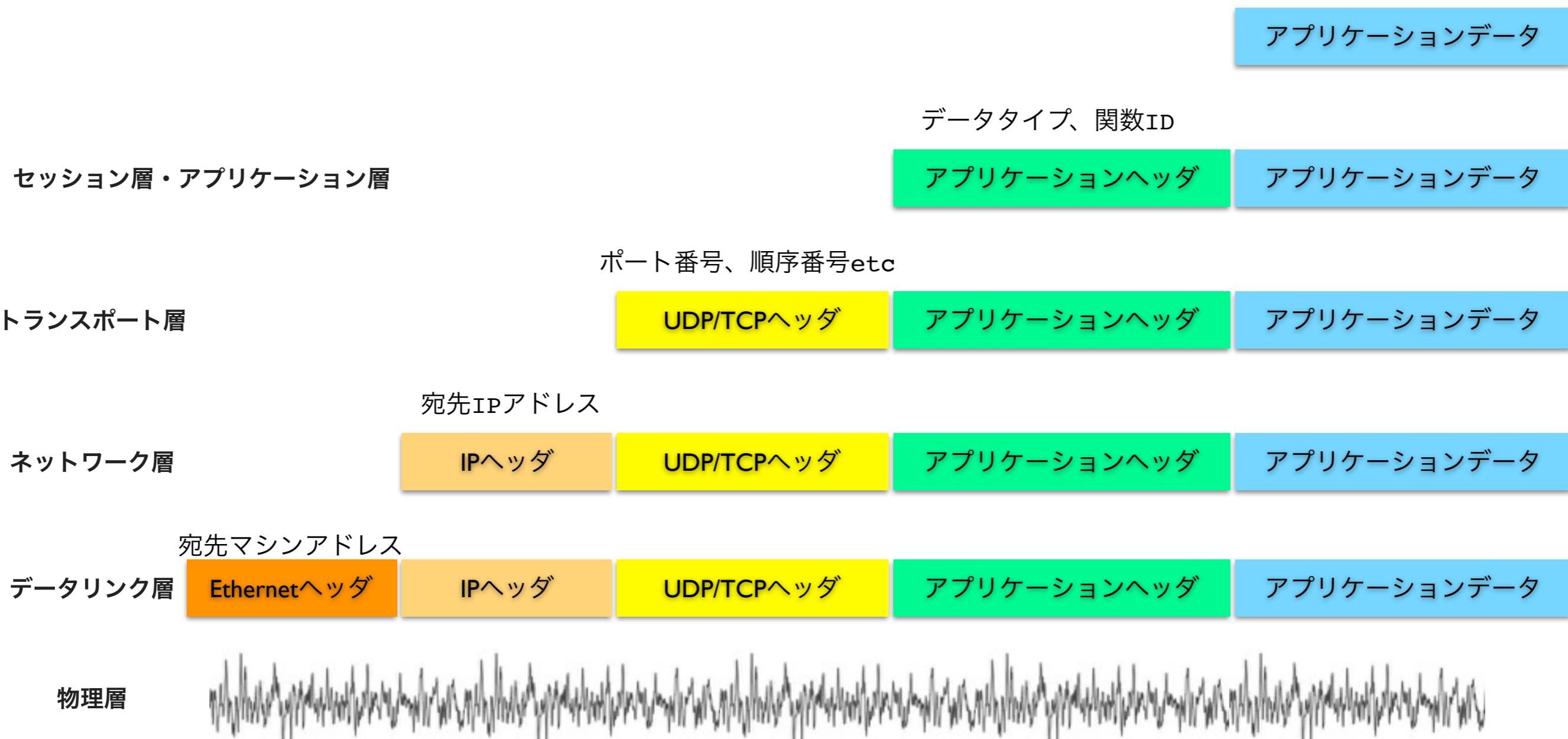
ほとんど全てのネットワークでは1500バイト以内しか送れない

[http://www.atmarkit.co.jp/ait/articles/0304/04/news001\\_2.html](http://www.atmarkit.co.jp/ait/articles/0304/04/news001_2.html)

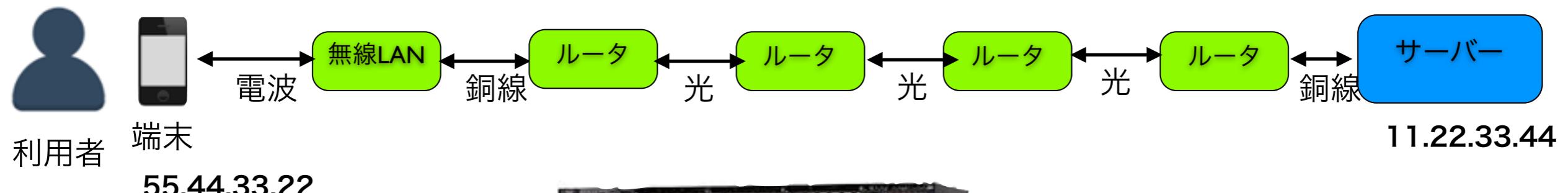


# パケットヘッダの積み重なり

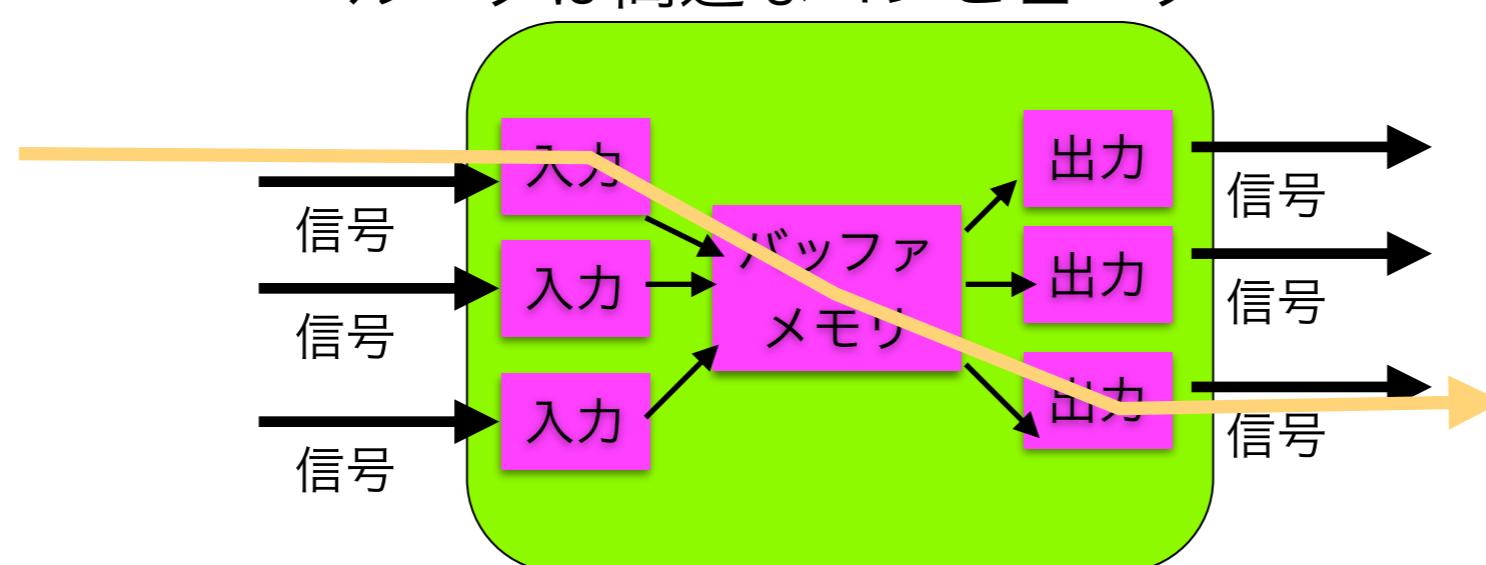
{ "x": 10, "y": 20 }



# IP：ルーターを使ってデータを転送

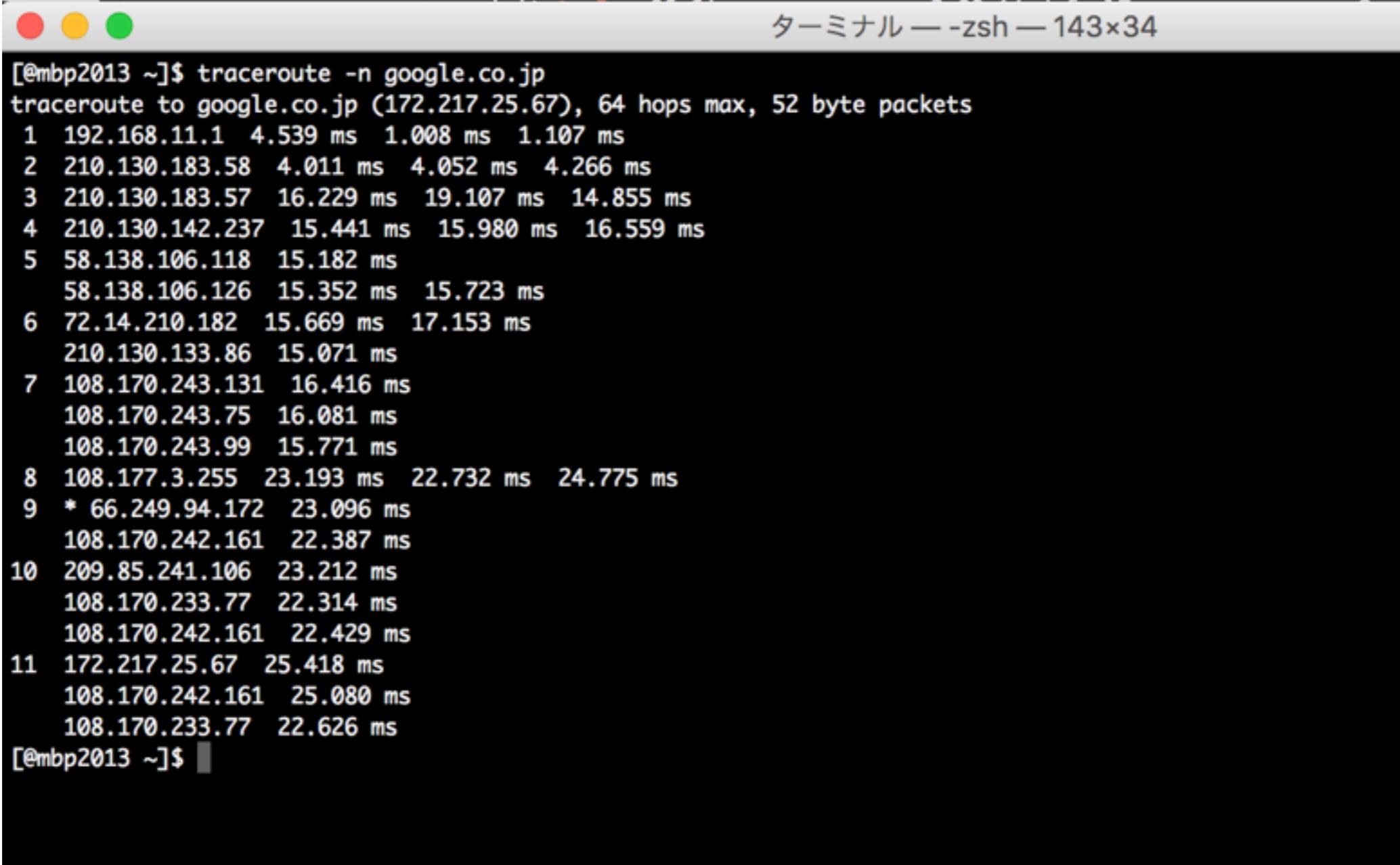


ルータは高速なコンピューター



インターネットの巨大な住所録をメモリに載せ、  
11.22.33.44がどの光ケーブルの先にあるかを判定

# tracerouteコマンド



ターミナル - zsh - 143x34

```
[@mbp2013 ~]$ traceroute -n google.co.jp
traceroute to google.co.jp (172.217.25.67), 64 hops max, 52 byte packets
 1  192.168.11.1  4.539 ms  1.008 ms  1.107 ms
 2  210.130.183.58  4.011 ms  4.052 ms  4.266 ms
 3  210.130.183.57  16.229 ms  19.107 ms  14.855 ms
 4  210.130.142.237  15.441 ms  15.980 ms  16.559 ms
 5  58.138.106.118  15.182 ms
      58.138.106.126  15.352 ms  15.723 ms
 6  72.14.210.182  15.669 ms  17.153 ms
      210.130.133.86  15.071 ms
 7  108.170.243.131  16.416 ms
      108.170.243.75  16.081 ms
      108.170.243.99  15.771 ms
 8  108.177.3.255  23.193 ms  22.732 ms  24.775 ms
 9  * 66.249.94.172  23.096 ms
      108.170.242.161  22.387 ms
10  209.85.241.106  23.212 ms
      108.170.233.77  22.314 ms
      108.170.242.161  22.429 ms
11  172.217.25.67  25.418 ms
      108.170.242.161  25.080 ms
      108.170.233.77  22.626 ms
[@mbp2013 ~]$
```

手元の端末から目的マシンへの途中のルーターを調べる



# インターネットを使う上での問題



# 問題：光は速いけど遅い

- ・ 真空中：1秒間に30万km
- ・ 光ファイバの中：1秒間に20万km
- ・ 1ミリ秒(ms)では200km
- ・ 東京 <-> 大阪 400km = 片道2ms 東京<->サンフランシスコ 片道40ms



# 問題：短い経路ではなく安い経路が使われる

- 富山から東京にパケットを投げたら、金沢>大阪>東京 と伝達される
- NTT西日本と東日本の取引がそうなっている



tracerouteの結果

```
[embp2013 mrs_bench]$ traceroute 130.69.251.23
traceroute to 130.69.251.23 (130.69.251.23), 64 hops max, 52 byte packets
 1  192.168.11.1 (192.168.11.1)  1.202 ms  1.005 ms  0.921 ms
 2  toyama03-z01.flets.2iij.net (210.130.183.58)  3.949 ms  4.040 ms  3.984 ms
 3  osk004lip30.2iij.net (210.130.183.57)  15.007 ms  15.182 ms  15.288 ms
 4  osk004bb00.2iij.net (210.130.142.237)  18.534 ms  15.289 ms  15.471 ms
 5  osk004ix50.2iij.net (58.138.106.118)  15.922 ms  15.583 ms
  osk004ix51.2iij.net (58.138.106.126)  15.653 ms
 6  210.173.178.25 (210.173.178.25)  15.043 ms  21.361 ms  19.622 ms
 7  tokyo1-rm-et-4-1-0-152.s5.sinet.ad.jp (150.99.71.104)  31.463 ms
    tokyo1-rm-et-5-1-0-1152.s5.sinet.ad.jp (150.99.90.28)  23.288 ms
    tokyo1-rm-et-4-1-0-152.s5.sinet.ad.jp (150.99.71.104)  22.476 ms
 8  utnet-2.gw.sinet.ad.jp (150.99.190.98)  22.963 ms  23.848 ms  28.087 ms
 9  ra56-vlan5.nc.u-tokyo.ac.jp (133.11.127.94)  25.059 ms  23.065 ms  22.938 ms
10  ra35-vlan12.nc.u-tokyo.ac.jp (133.11.127.109)  39.699 ms  80.270 ms  27.060 ms
```

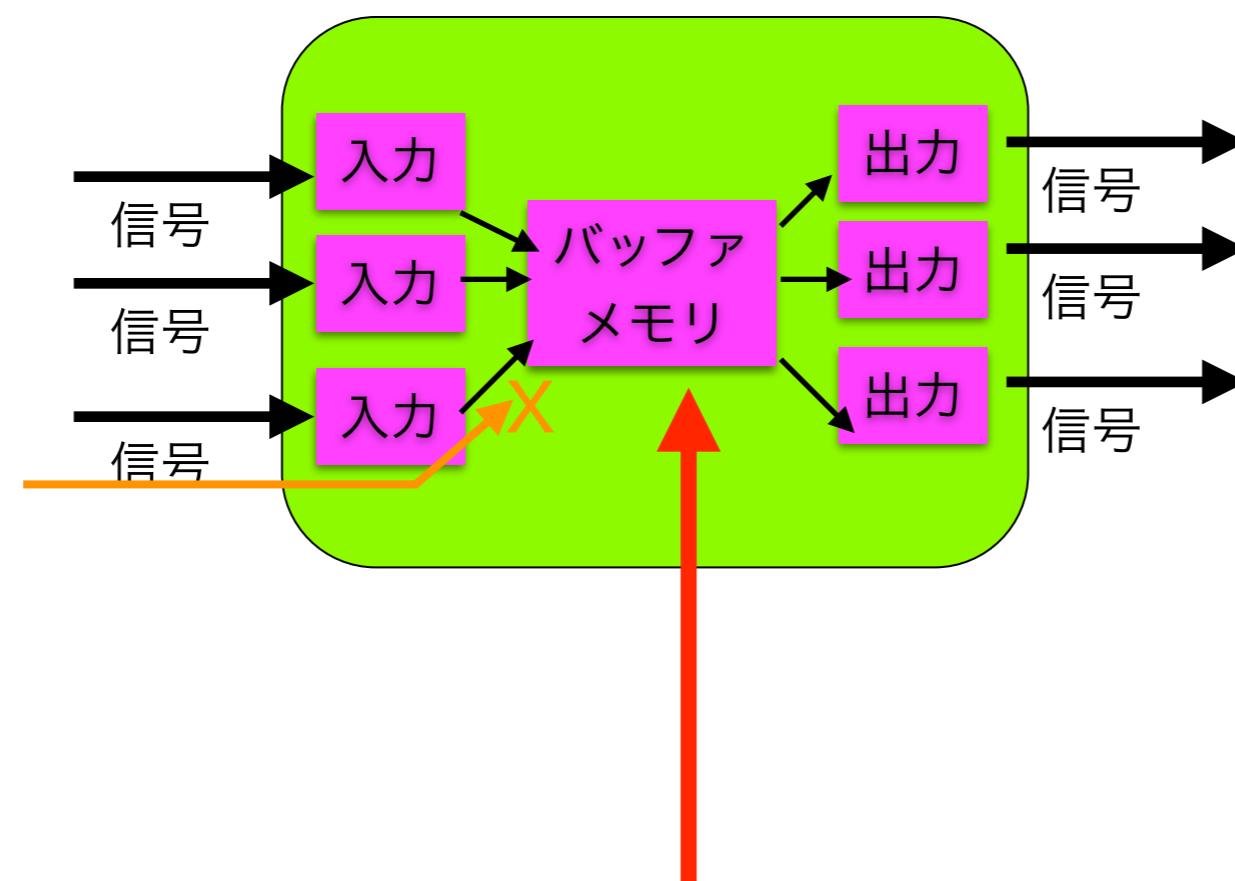
# 問題：ルータは遅い

- 富山から大阪は300kmしかないので1.5msで済むはずが、10ms以上かかっている



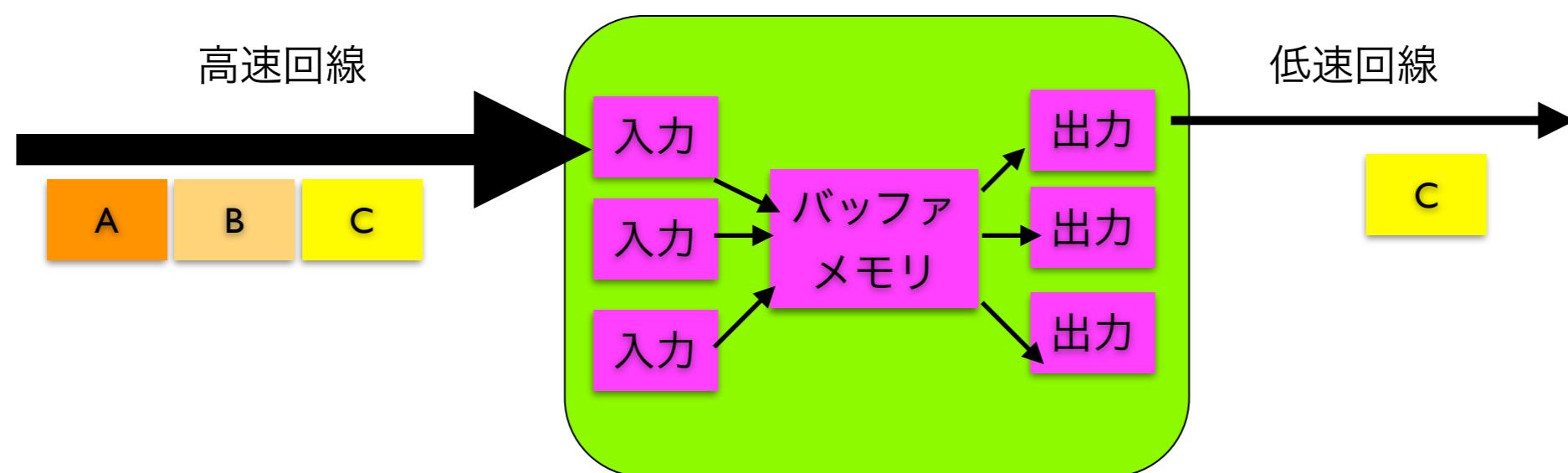
- ルータがやっていること
  - ケーブルの信号を読み取ってバッファに積む
  - パケットを解析して送り先IPアドレスを調べる
  - IPアドレスの住所録(数十万～数百万件)を検索してどのケーブルに出力するか決定
  - 送信バッファに出力
  - ケーブルに送信する
  - パケットの断片化や暗号化、アタック防止やファイアウォール、NATの変換、統計データ取得など膨大な量のタスクを同時進行させている

# 問題: ルーターはパケットを捨てる



1台のルータを数千人、数万人で共有する。  
メモリもCPUも、だいたいいつも足りないので、  
入力信号を捨てるしかない

# 問題: ルータの出口が細いと捨てる



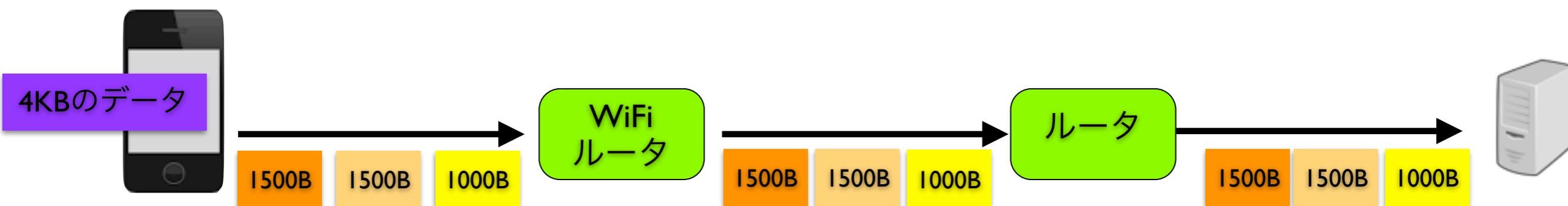
入力が速すぎたので  
AとBは捨てる

# 問題: 小さい単位でしか送れない

- ・ 物理層の制限 (MTU)
  - ・ できるだけ多くの人が1本のケーブルを共有するために、時間を短く切りきざむ必要がある。
  - ・ WiFiやEthernetでは1500Bytes ~ 9KBytes (数マイクロ秒以下)
  - ・ モバイルだと1000Bytes以下のこともある
  - ・ それ以外でもだいたい1KB~5KB以下
- ・ それ以上のパケットは「断片化」して送る



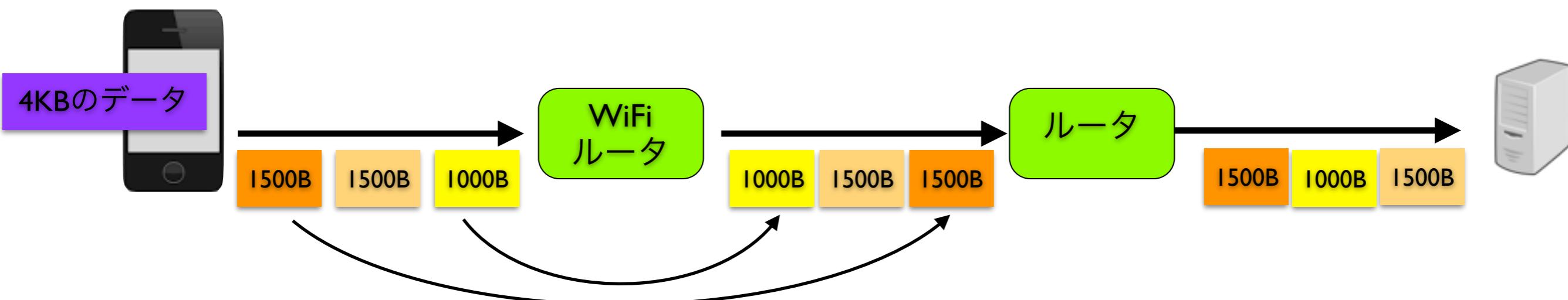
# 4KBを送るとき(成功)



3個に分かれた。

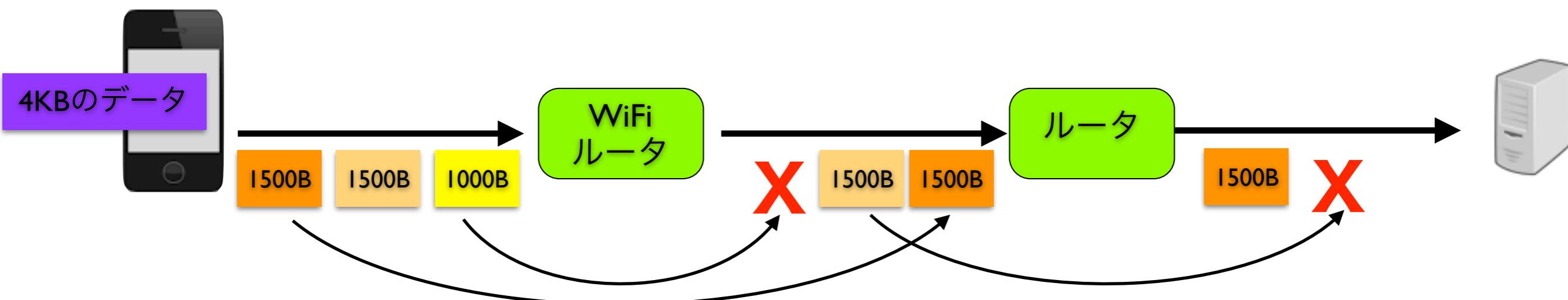
パケットを分割する動作を「IP断片化」と呼ぶ

# 4KBを送るとき(成功:パケット順)



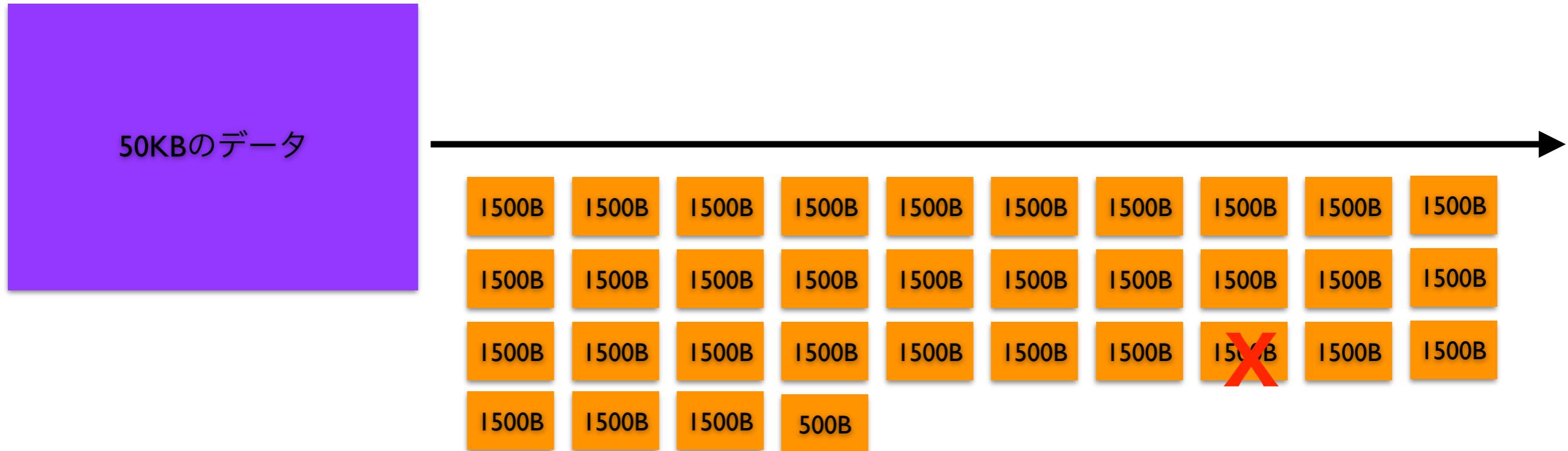
順番が入れ替わったら  
受信したサーバー(エンドポイント)が  
並べ替えて元に戻す

# 4KBを送るとき(失敗:パケットロス)



IPのパケットが2個、消えた。  
元に戻すことができない!

# 50KBを送るとき



どれか1個がロスしたら全体が復元できなくなる!

IPパケットは最大64KBまで。  
断片の数が多いほど全ロスの確率が指数的に高まる

# pingコマンドでパケットロスの度合いを確認

```
選択コマンドプロンプト
最小 = 25ms、最大 = 38ms、平均 = 29ms

C:\Users\k.nakajima>ping 210.140.72.117 -n 20

210.140.72.117 に ping を送信しています 32 バイトのデータ:
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =26ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =26ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =26ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =26ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53
要求がタイムアウトしました。
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53
要求がタイムアウトしました。
210.140.72.117 からの応答: バイト数 =32 時間 =29ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =27ms TTL=53
要求がタイムアウトしました。
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =27ms TTL=53
要求がタイムアウトしました。
210.140.72.117 からの応答: バイト数 =32 時間 =27ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53
210.140.72.117 からの応答: バイト数 =32 時間 =25ms TTL=53

210.140.72.117 の ping 統計:
パケット数: 送信 = 20、受信 = 16、損失 = 4 (20% の損失)、
ラウンドトリップの概算時間 (ミリ秒):
最小 = 25ms、最大 = 29ms、平均 = 25ms

C:\Users\k.nakajima>
```



# iperf3コマンドでもロス率を見る

- Linux,Windows,Mac,Androidなど各OSで使えるベンチマークツール
- TCPもUDPも測れる
- pingコマンドよりも、ゲームの送信パターンにより近くすることができる

```
[@mbp2013 ~]$ iperf3 -c 210.140.72.117 -u -b 100k
Connecting to host 210.140.72.117, port 5201
[ 5] local 192.168.11.7 port 58265 connected to 210.140.72.117 port 5201
[ ID] Interval          Transfer     Bitrate      Total Datagrams
[ 5]  0.00-1.00   sec  12.3 KBytes   101 Kbits/sec  9
[ 5]  1.00-2.00   sec  12.3 KBytes   101 Kbits/sec  9
[ 5]  2.00-3.00   sec  12.3 KBytes   101 Kbits/sec  9
[ 5]  3.00-4.00   sec  12.3 KBytes   101 Kbits/sec  9
[ 5]  4.00-5.00   sec  12.3 KBytes   101 Kbits/sec  9
[ 5]  5.00-6.00   sec  12.3 KBytes   101 Kbits/sec  9
[ 5]  6.00-7.00   sec  12.3 KBytes   101 Kbits/sec  9
[ 5]  7.00-8.00   sec  12.3 KBytes   101 Kbits/sec  9
[ 5]  8.00-9.00   sec  12.3 KBytes   101 Kbits/sec  9
[ 5]  9.00-10.00  sec  12.3 KBytes   101 Kbits/sec  9
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval          Transfer    Bitrate      Jitter   Lost/Total Datagrams
[ 5]  0.00-10.00  sec  123 KBytes   101 Kbits/sec  0.000 ms  0/90 (0%)  sender
[ 5]  0.00-10.00  sec  122 KBytes   99.8 Kbits/sec 3.758 ms  1/90 (1.1%) receiver
```



# 参考データ

- ・ 無線LAN
  - ・ 電波最強で0.01~0.1%程度、電波弱いと5%~100%ロス
- ・ モバイルキャリア
  - ・ 電波最強でも0.1%以上ロスあり　電波弱いと10%以上ロス。
- ・ 有線LAN
  - ・ 0.00000001%~ケーブルクオリティ低いと0.01%ロス
  - ・ ルータが混んでたらいくらでもロス
- ・ データセンター内部
  - ・ スループット制限を超えた分は全部ロス



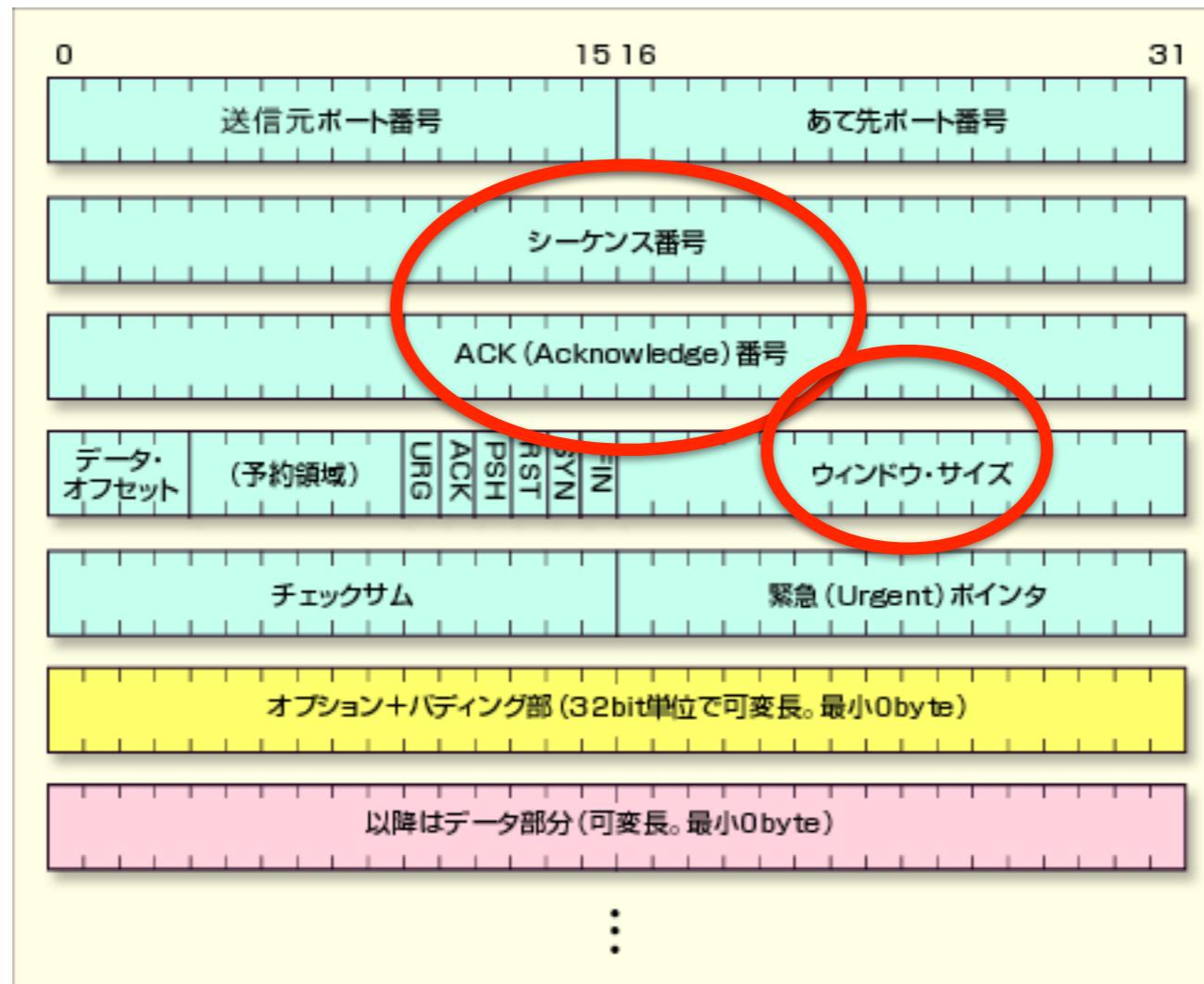
パケロスが多いことはわかった。  
でも、LINEやブラウザはを使っている。  
それはTCPのおかげ



TCPがわかれば  
UDPの使いどころがわかる



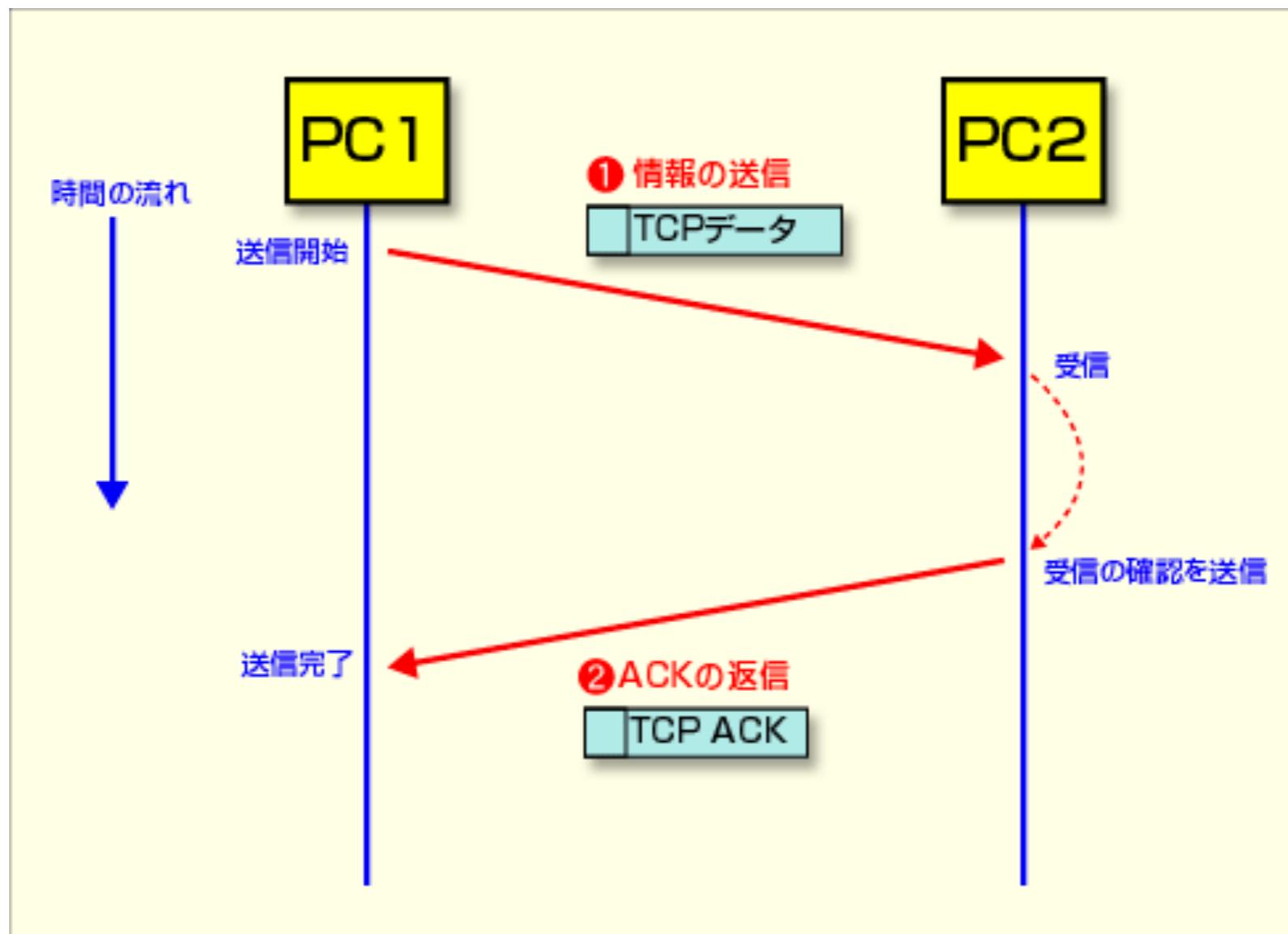
# TCPパケット



[http://www.atmarkit.co.jp/ait/articles/0401/29/news080\\_2.html](http://www.atmarkit.co.jp/ait/articles/0401/29/news080_2.html)

IPに対して「シーケンス(順序)番号」と  
「ACK(受け取り)番号」「ウインドウサイズ」を追加し、  
パケットの再送、正しい順序、送り過ぎの防止を実装

# TCP：パケットの再送

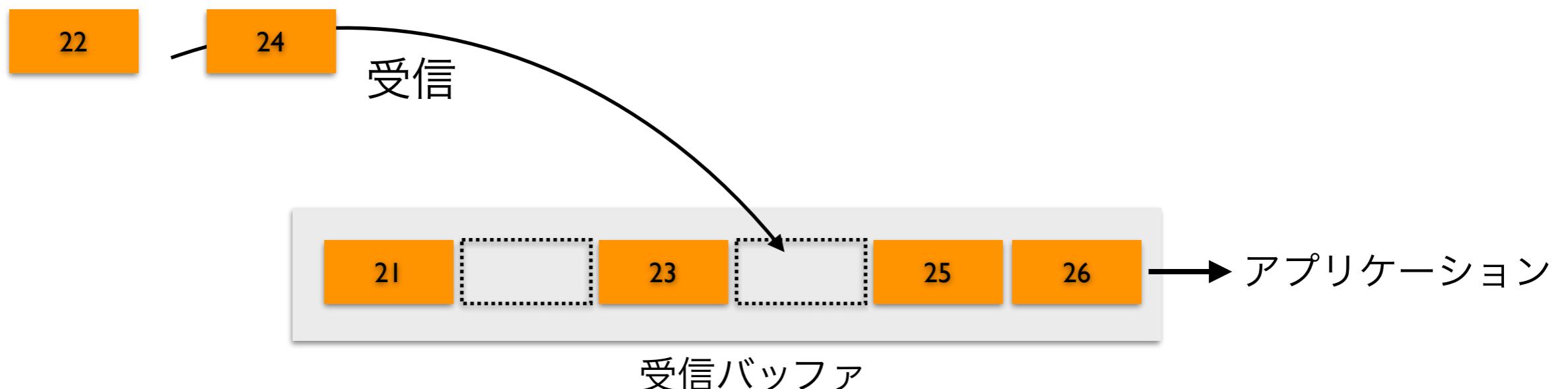


[http://www.atmarkit.co.jp/ait/articles/0312/25/news001\\_2.html](http://www.atmarkit.co.jp/ait/articles/0312/25/news001_2.html)

ACK番号を使って到達確認をし、  
確認が戻ってこなかったら同じデータを何度も送る

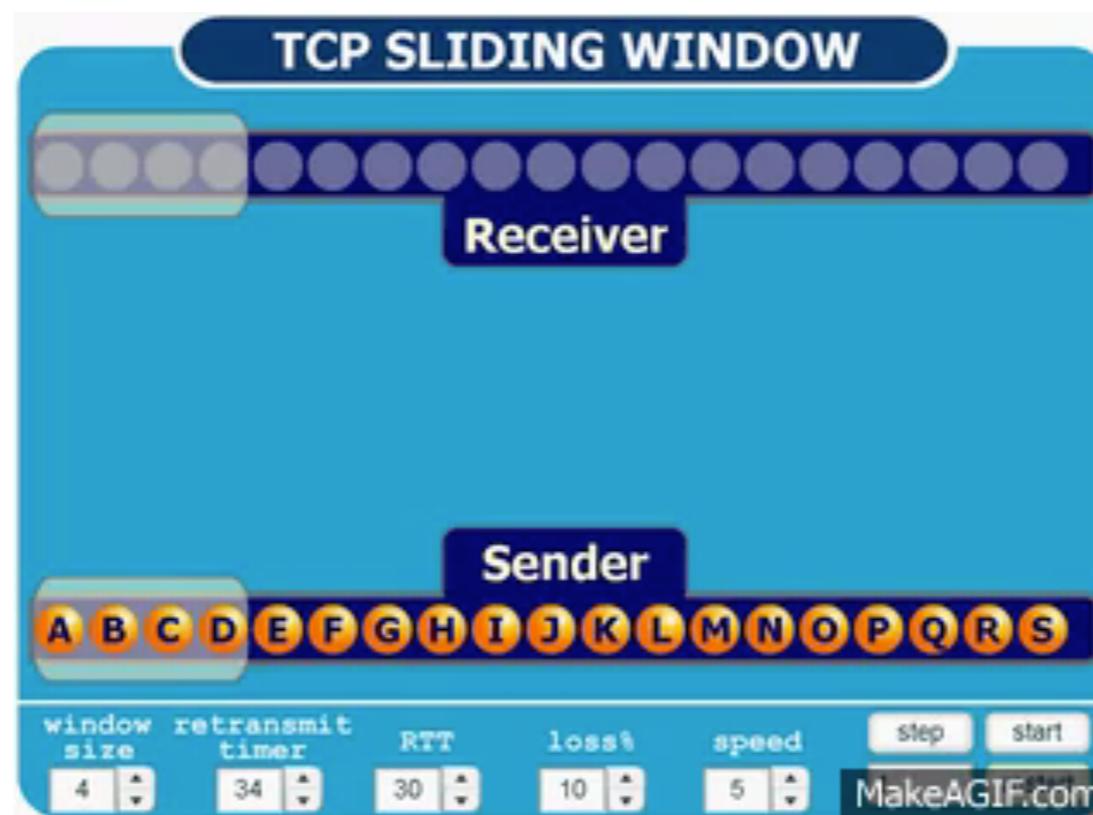
# TCP：正確な受信順序

- ・ パケットに「シーケンス番号」が付いているので、受信側のメモリに貯めておいて順番を入れ替えることができる
- ・ これで何十MBでも何GBでも正しく送れるようになった



# TCP：送りすぎの防止（輻輳制御）

- ・スライディングウインドウ(窓枠ずらし)方式
- ・一度に送信するパケットの量に上限を設定する。これを「窓枠」「ウインドウ」と呼ぶ。
- ・窓の大きさは「ウインドウサイズ」と呼ばれる
- ・速く送るにはウインドウサイズを大きくする必要がある。
- ・この窓枠の大きさを状況によって動的に変更し、性能を調整する

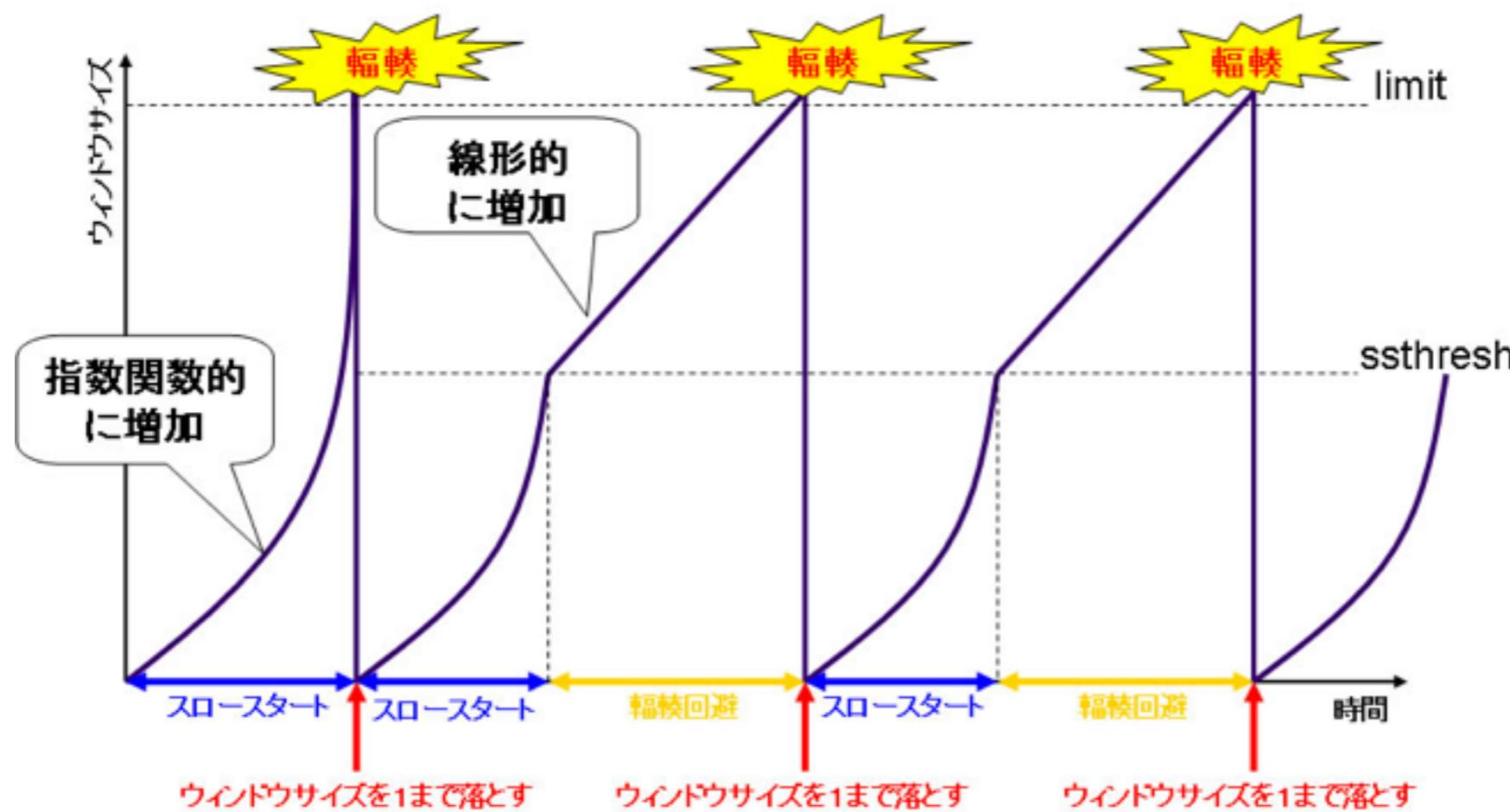


<https://www.youtube.com/watch?v=Ik27yilTOvU>



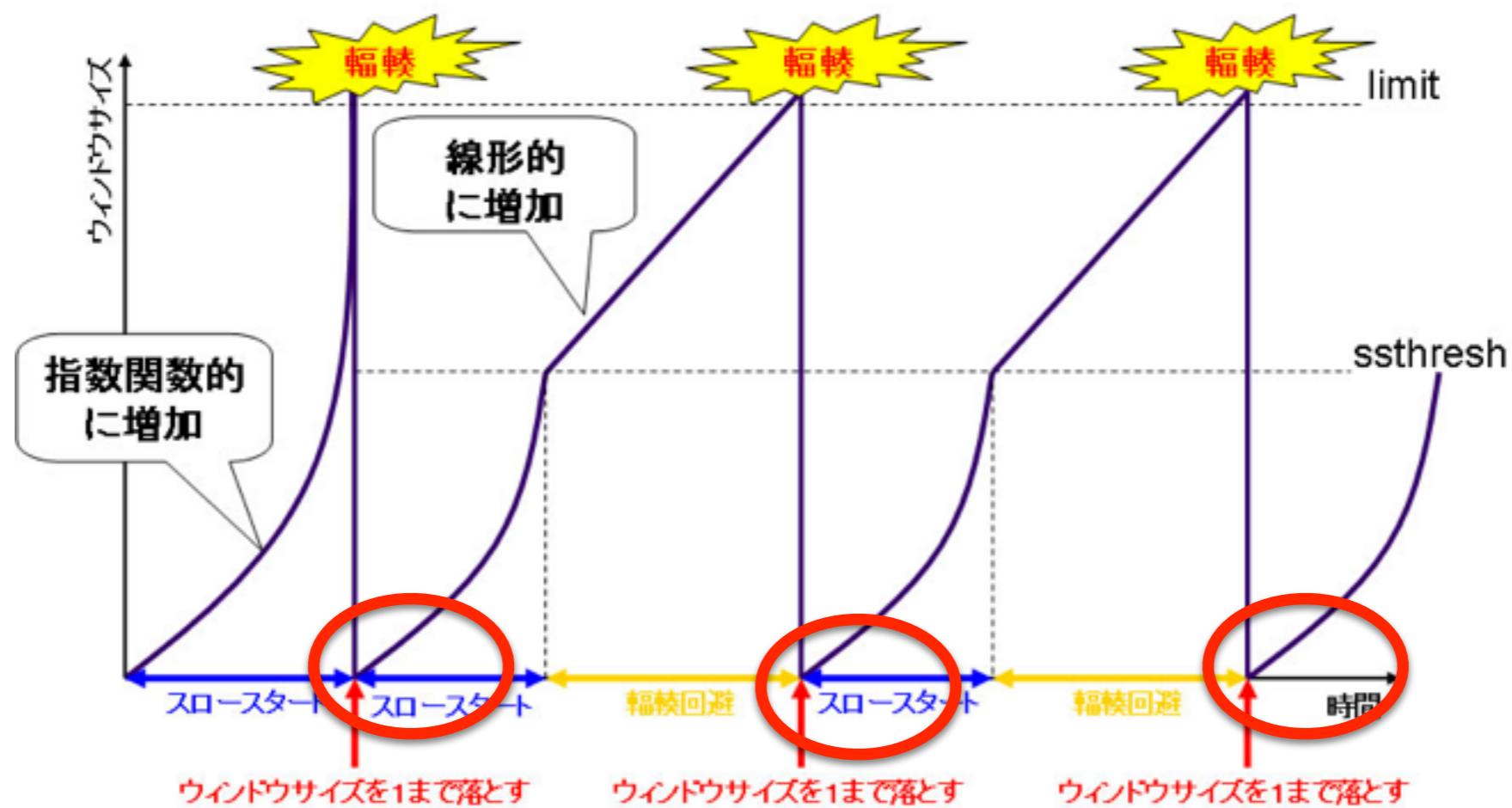
# TCP：スロースタート

- ・ 遅く始めてだんだん加速する。つまり、窓枠をだんだん大きくする。
- ・ パケットがロスしない間は、ウインドウサイズをだんだん大きくして加速していく
- ・ パケロスしたらリセット=窓枠を最小にする



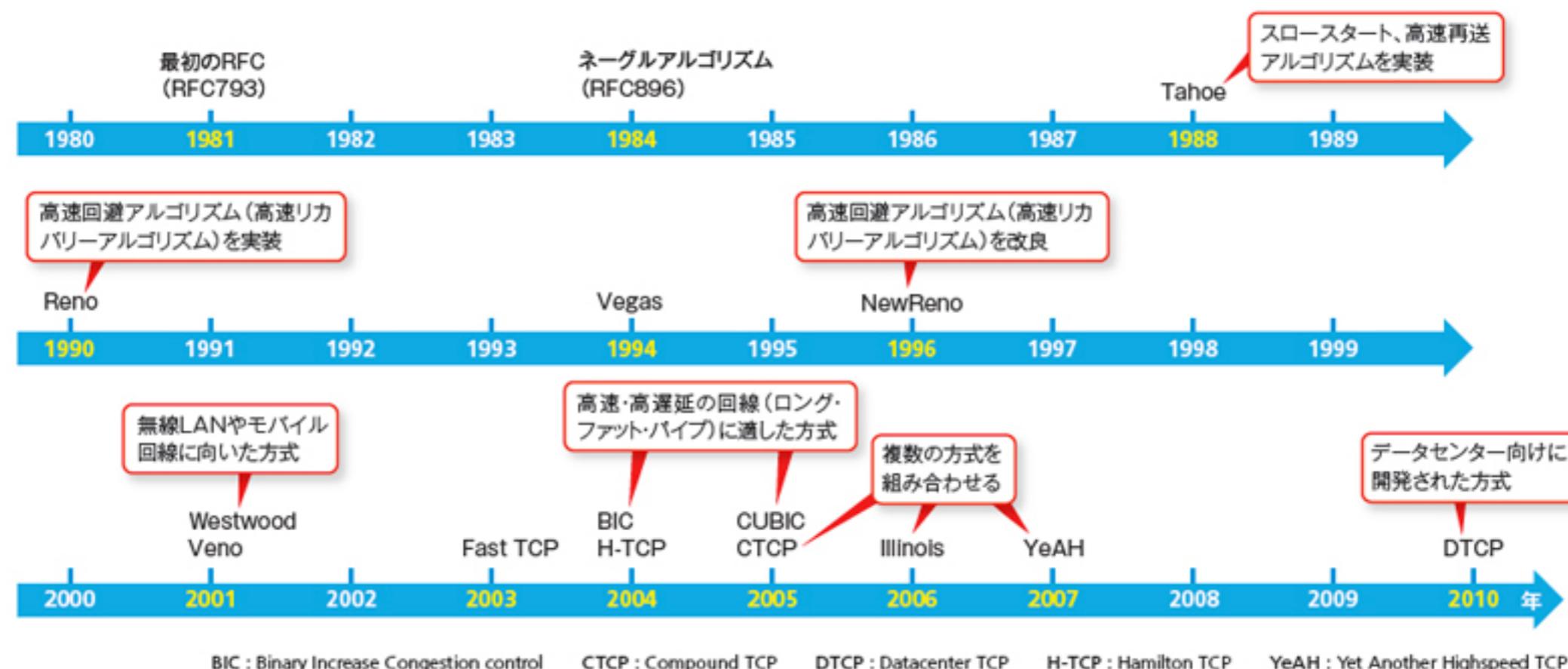
# TCPの大きな問題「ストール」

- TCPはパケットロスを検出するとウィンドウサイズを小さくする。
- ウィンドウサイズが大幅に落ちて極端に遅くなった状態を「ストール」と呼ぶ。ゲームではストールが起きるひどいラグになり、プレイできなくなる。



# TCPの歴史的発展

- 1981年の使用開始から現在に至るまで改善の連続。。。！

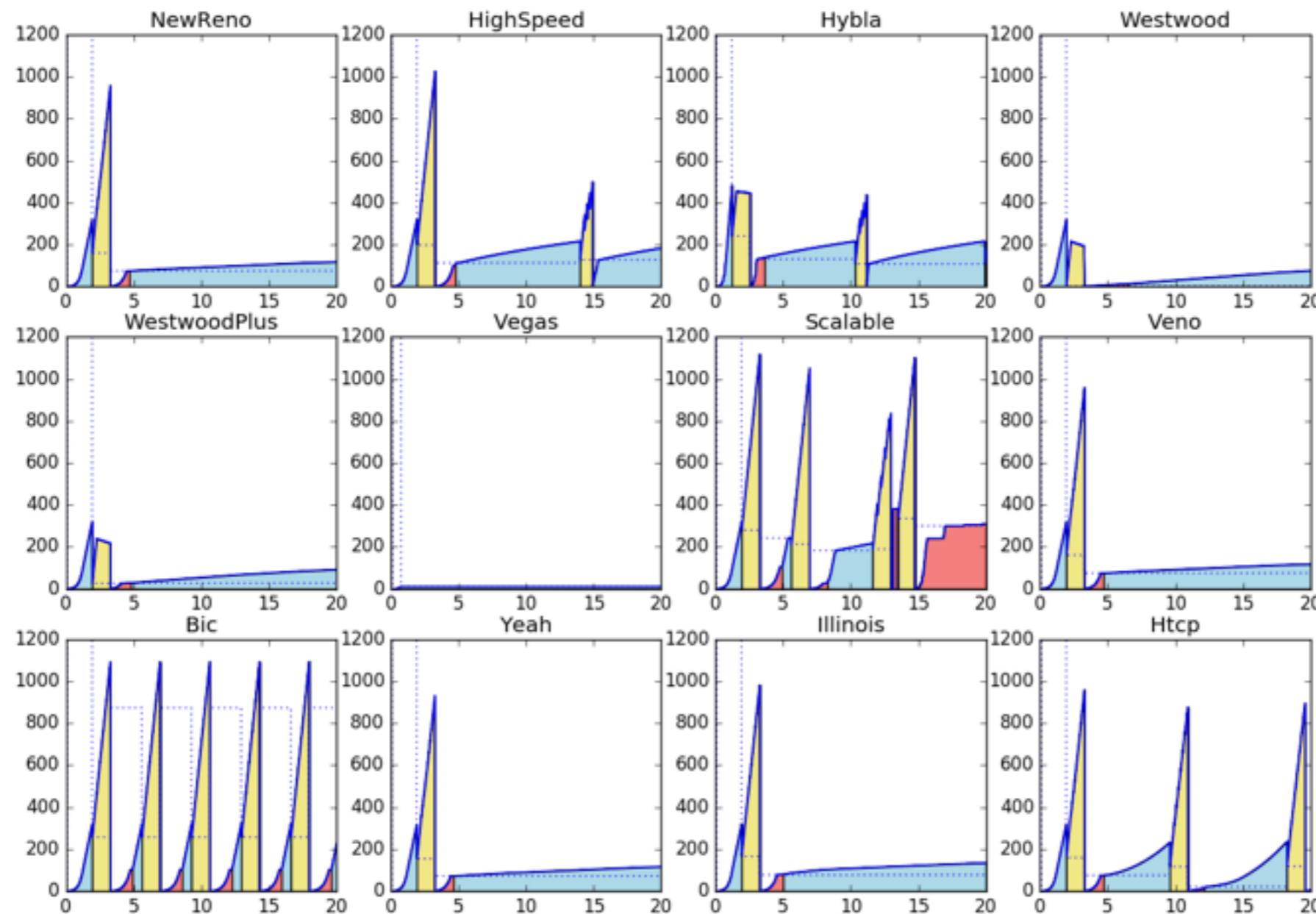


# TCPの時代分け

- ・ 1980年代：TCP登場、telnetでマシンを遠隔操作
- ・ 1990年代：メールやHTTPなどで大ブレイク
- ・ 2000年代：モバイルで長時間のストールを防ぎたい！
- ・ 2010年代：動画や音声を高速送信したい！



# TCPの多種多様なアルゴリズム



<https://qiita.com/haltaro/items/d479538345357f08c595>



# TCPの今後の発展

- ・ より多くの観測情報を統計的に使う
- ・ 2016 Google BBR (Bottleneck Bandwidth and Round-trip propagation time)アルゴリズム登場
- ・ 途中のルータと通信してより正確な状況を把握する
- ・ 宇宙空間やIoTなど極端な状況への対応
- ・ 異なる経路の複数のセッションを同時に活用(MPTCP)
- ・ ...
- ・ スループットの向上とストールの回避をどこまでも追求している
- ・ 現在も早く進化中で、今後もどんどん改善されていく
- ・ GoogleやMSなどの巨大企業による莫大な量の投資



TCPの発展はわかったが  
いま現在はどうなのだろうか



# Centos7でsysctl

- LinuxではTCPの制御アルゴリズムを選択できる。
- sysctlコマンドで選択肢を操作できる

```
[@sakana-1 ~]$ sysctl -a | grep congest
net.ipv4.tcp_allowed_congestion_control = cubic reno
net.ipv4.tcp_available_congestion_control = cubic reno
net.ipv4.tcp_congestion_control = cubic
```

cubic(2007)とreno(1990)..



# mrs\_benchのTCPモードで実験

- ・ サーバ : CentOS7 + cubic
- ・ クライアント : MacOS X + newreno
- ・ 意図的なパケロスなし
- ・ 富山>東京

```
[@mbp2013 mrs_bench]$ ./cl 210.140.72.117 1 10 --size=4000
server:210.140.72.117 conn_num:1 interval:0.010000 send_per_action:1 conn_type:1 nosleep:0 unreliable:0
[1508985523] loop:1 sentN:0 recvN:0(0) conn:0 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:nanms max:0.0ms min:99999000.0ms last::0.0ms
c[1508985524] loop:751 sentN:86 recvN:80(80) conn:1 dis:0 err:0 recvB:0.3(0.3) MiB RTT avg:42.4ms max:42.4ms min:42.4ms last:42.4ms
[1508985525] loop:1493 sentN:179 recvN:173(93) conn:1 dis:0 err:0 recvB:0.7(0.4) MiB RTT avg:39.4ms max:42.4ms min:36.4ms last:36.4ms
[1508985526] loop:2253 sentN:272 recvN:266(93) conn:1 dis:0 err:0 recvB:1.0(0.4) MiB RTT avg:39.0ms max:42.4ms min:36.4ms last:38.2ms
[1508985527] loop:2990 sentN:364 recvN:356(90) conn:1 dis:0 err:0 recvB:1.4(0.3) MiB RTT avg:39.7ms max:42.4ms min:36.4ms last:42.0ms
[1508985528] loop:3730 sentN:456 recvN:447(91) conn:1 dis:0 err:0 recvB:1.7(0.3) MiB RTT avg:39.7ms max:42.4ms min:36.4ms last:39.7ms
[1508985529] loop:4468 sentN:548 recvN:539(92) conn:1 dis:0 err:0 recvB:2.1(0.4) MiB RTT avg:42.2ms max:54.4ms min:36.4ms last:54.4ms
[1508985530] loop:5217 sentN:641 recvN:631(92) conn:1 dis:0 err:0 recvB:2.4(0.4) MiB RTT avg:45.8ms max:67.6ms min:36.4ms last:67.6ms
[1508985531] loop:5958 sentN:733 recvN:721(90) conn:1 dis:0 err:0 recvB:2.8(0.3) MiB RTT avg:44.9ms max:67.6ms min:36.4ms last:38.3ms
[1508985532] loop:6732 sentN:828 recvN:815(94) conn:1 dis:0 err:0 recvB:3.1(0.4) MiB RTT avg:44.3ms max:67.6ms min:36.4ms last:39.9ms
[1508985533] loop:7473 sentN:920 recvN:902(87) conn:1 dis:0 err:0 recvB:3.5(0.3) MiB RTT avg:43.8ms max:67.6ms min:36.4ms last:39.4ms
[1508985534] loop:8217 sentN:1013 recvN:998(96) conn:1 dis:0 err:0 recvB:3.8(0.4) MiB RTT avg:43.8ms max:67.6ms min:36.4ms last:43.1ms
[1508985535] loop:8972 sentN:1106 recvN:1092(94) conn:1 dis:0 err:0 recvB:4.2(0.4) MiB RTT avg:43.8ms max:67.6ms min:36.4ms last:43.1ms
```

mrs\_benchはモノビットエンジン製品”MRS”的  
TCP/UDPベンチマークツール

だいたい50ms以下なので  
快適にゲームできる



# mrs\_benchのTCPモードで実験

- ・ サーバ : CentOS7 + cubic
- ・ クライアント : MacOS X + newreno
- ・ 意図的なパケロス : **5%** (モバイルやWiFiではまあある状態)
- ・ 富山>東京

```
[@mbp2013 mrs_bench]$ ./cl 210.140.72.117 1 10 --size=4000
server:210.140.72.117 conn_num:1 interval:0.010000 send_per_action:1 conn_type:1 nosleep:0 unreliable:0
[1508985563] loop:1 sentN:0 recvN:0(0) conn:0 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:nanms max:0.0ms min:99999000.0ms last:0.0ms
c[1508985564] loop:750 sentN:87 recvN:75(75) conn:1 dis:0 err:0 recvB:0.3(0.3) MiB RTT avg:79.3ms max:79.3ms min:79.3ms last:79.3ms
[1508985565] loop:1516 sentN:180 recvN:131(56) conn:1 dis:0 err:0 recvB:0.5(0.2) MiB RTT avg:163.8ms max:248.2ms min:79.3ms last:248.2ms
[1508985566] loop:2256 sentN:273 recvN:210(79) conn:1 dis:0 err:0 recvB:0.8(0.3) MiB RTT avg:314.0ms max:614.5ms min:79.3ms last:614.5ms
[1508985567] loop:2996 sentN:365 recvN:253(43) conn:1 dis:0 err:0 recvB:1.0(0.2) MiB RTT avg:314.0ms max:614.5ms min:79.3ms last:614.5ms
[1508985568] loop:3737 sentN:457 recvN:310(57) conn:1 dis:0 err:0 recvB:1.2(0.2) MiB RTT avg:611.1ms max:1502.4ms min:79.3ms last:1502.4ms
[1508985569] loop:4476 sentN:549 recvN:361(51) conn:1 dis:0 err:0 recvB:1.4(0.2) MiB RTT avg:611.1ms max:1502.4ms min:79.3ms last:1502.4ms
[1508985570] loop:5214 sentN:642 recvN:419(58) conn:1 dis:0 err:0 recvB:1.6(0.2) MiB RTT avg:958.8ms max:2349.4ms min:79.3ms last:2349.4ms
[1508985571] loop:5979 sentN:736 recvN:476(57) conn:1 dis:0 err:0 recvB:1.8(0.2) MiB RTT avg:958.8ms max:2349.4ms min:79.3ms last:2349.4ms
[1508985572] loop:6721 sentN:828 recvN:537(61) conn:1 dis:0 err:0 recvB:2.1(0.2) MiB RTT avg:1268.0ms max:2814.4ms min:79.3ms last:2814.4ms
[1508985573] loop:7461 sentN:921 recvN:588(51) conn:1 dis:0 err:0 recvB:2.2(0.2) MiB RTT avg:1268.0ms max:2814.4ms min:79.3ms last:2814.4ms
```

これではゲームにならない



# LinuxでのTCPアルゴリズムの調整

- CentOS7で使えるTCPモジュールの確認

```
[root@sakana-2 ~]# grep TCP_ /boot/config-3.10.0-514.26.2.el7.x86_64
CONFIG_INET_TCP_DIAG=m
CONFIG_TCP_CONG_ADVANCED=y
CONFIG_TCP_CONG_BIC=m
CONFIG_TCP_CONG_CUBIC=y
CONFIG_TCP_CONG_WESTWOOD=m
CONFIG_TCP_CONG_HTCP=m
CONFIG_TCP_CONG_HSTCP=m
CONFIG_TCP_CONG_HYBLA=m
CONFIG_TCP_CONG_VEGAS=m
CONFIG_TCP_CONG_SCALABLE=m
CONFIG_TCP_CONG_LP=m
CONFIG_TCP_CONG_VENO=m
CONFIG_TCP_CONG_YEAH=m
CONFIG_TCP_CONG_ILLINOIS=m
CONFIG_TCP_CONG_DCTCP=m
CONFIG_DEFAULT_TCP_CONG="cubic"
CONFIG_TCP_MD5SIG=y
```

- bic,cubic,westwood,htcp,hstcp,hybla,vegas,scalable,lp,veno,yeah,illinois,dctcp が標準インストールされているとわかる
  - westwoodはパケットロスが多く遅延が大きいモバイル環境向け
  - dctcpはデータセンター向けで、高速スイッチ機器と連携できる
  - などなど



# Linuxにwestwood TCPを導入

- rootになり
  - modprobe tcp\_westwood
  - echo "westwood" > /proc/sys/net/ipv4/tcp\_congestion\_control
- 再起動は不要



# mrs\_benchでwestwoodを測定

- ・ サーバ : Centos7 + westwood
- ・ クライアント : MacOS X + newreno
- ・ 意図的なパケロス : 5% (モバイルやWiFiではまあある状態)
- ・ 富山>東京

```
westwoodロス5%
[@mbp2013 mrs_bench]$ ./cl 210.140.72.117 1 10 --size=4000
server:210.140.72.117 conn_num:1 interval:0.010000 send_per_action:1 conn_type:1 nosleep:0 unreliable:0
[1508985619] loop:1 sentN:0 recvN:0(0) conn:0 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:nanms max:0.0ms min:99999000.0ms last:0.0ms
[1508985620] loop:747 sentN:87 recvN:58(58) conn:1 dis:0 err:0 recvB:0.2(0.2) MiB RTT avg:44.5ms max:44.5ms min:44.5ms last:44.5ms
[1508985621] loop:1488 sentN:180 recvN:108(50) conn:1 dis:0 err:0 recvB:0.4(0.2) MiB RTT avg:366.2ms max:687.9ms min:44.5ms last:687.9ms
[1508985622] loop:2232 sentN:273 recvN:196(88) conn:1 dis:0 err:0 recvB:0.7(0.3) MiB RTT avg:366.2ms max:687.9ms min:44.5ms last:687.9ms
[1508985623] loop:2975 sentN:366 recvN:273(77) conn:1 dis:0 err:0 recvB:1.0(0.3) MiB RTT avg:516.4ms max:816.7ms min:44.5ms last:816.7ms
[1508985624] loop:3713 sentN:458 recvN:363(90) conn:1 dis:0 err:0 recvB:1.4(0.3) MiB RTT avg:640.1ms max:1011.1ms min:44.5ms last:1011.1ms
[1508985625] loop:4459 sentN:550 recvN:457(94) conn:1 dis:0 err:0 recvB:1.7(0.4) MiB RTT avg:710.3ms max:1011.1ms min:44.5ms last:991.2ms
[1508985626] loop:5216 sentN:644 recvN:561(104) conn:1 dis:0 err:0 recvB:2.1(0.4) MiB RTT avg:727.9ms max:1011.1ms min:44.5ms last:815.9ms
[1508985627] loop:5964 sentN:737 recvN:655(94) conn:1 dis:0 err:0 recvB:2.5(0.4) MiB RTT avg:739.0ms max:1011.1ms min:44.5ms last:805.4ms
[1508985628] loop:6711 sentN:830 recvN:767(112) conn:1 dis:0 err:0 recvB:2.9(0.4) MiB RTT avg:736.1ms max:1011.1ms min:44.5ms last:716.3ms
[1508985629] loop:7458 sentN:924 recvN:882(115) conn:1 dis:0 err:0 recvB:3.4(0.4) MiB RTT avg:706.5ms max:1011.1ms min:44.5ms last:469.8ms
[1508985630] loop:8208 sentN:1017 recvN:962(80) conn:1 dis:0 err:0 recvB:3.7(0.3) MiB RTT avg:684.1ms max:1011.1ms min:44.5ms last:482.6ms
[1508985631] loop:8952 sentN:1110 recvN:1062(100) conn:1 dis:0 err:0 recvB:4.1(0.4) MiB RTT avg:662.7ms max:1011.1ms min:44.5ms last:448.5ms
[1508985632] loop:9695 sentN:1202 recvN:1170(108) conn:1 dis:0 err:0 recvB:4.5(0.4) MiB RTT avg:637.3ms max:1011.1ms min:44.5ms last:358.2ms
[1508985633] loop:10439 sentN:1295 recvN:1278(108) conn:1 dis:0 err:0 recvB:4.9(0.4) MiB RTT avg:599.4ms max:1011.1ms min:44.5ms last:144.0ms
[1508985634] loop:11188 sentN:1388 recvN:1364(86) conn:1 dis:0 err:0 recvB:5.2(0.3) MiB RTT avg:559.5ms max:1011.1ms min:40.3ms last:40.3ms
[1508985635] loop:11936 sentN:1481 recvN:1452(88) conn:1 dis:0 err:0 recvB:5.6(0.3) MiB RTT avg:527.8ms max:1011.1ms min:40.3ms last:84.9ms
[1508985636] loop:12697 sentN:1575 recvN:1551(99) conn:1 dis:0 err:0 recvB:5.9(0.4) MiB RTT avg:497.3ms max:1011.1ms min:39.7ms last:39.7ms
[1508985637] loop:13451 sentN:1668 recvN:1645(94) conn:1 dis:0 err:0 recvB:6.3(0.4) MiB RTT avg:471.0ms max:1011.1ms min:39.7ms last:49.1ms
[1508985638] loop:14206 sentN:1762 recvN:1726(81) conn:1 dis:0 err:0 recvB:6.6(0.3) MiB RTT avg:449.8ms max:1011.1ms min:39.7ms last:89.7ms
```

最初少し悪化するが、統計データが集まると回復し安定化

これなら問題なくゲームできる



# CentOS以外でもアルゴリズムを確認

- Android
  - 4.4.2 Xperia ZR
  - adb shellにおいて `sysctl -a | grep conges`
    - `net.ipv4.tcp_allowed_congestion_control = cubic reno`
    - `net.ipv4.tcp_available_congestion_control = cubic reno`
    - `net.ipv4.tcp_congestion_control = cubic`
- MacOS XとiOS
  - `sysctl -a | grep sockets`
  - `net.inet.tcp.newreno_sockets: 0`
  - `net.inet.tcp.cubic_sockets: 74`
- Windows
  - デフォルトはCTCP
  - コマンドラインで `netsh int tcp show global` で細かな設定を確認できる



TCPの新しいアルゴリズムを使えば、長時間ストールの問題もかなり解決される。

UDPはだんだん不要になっていくのだろうか？



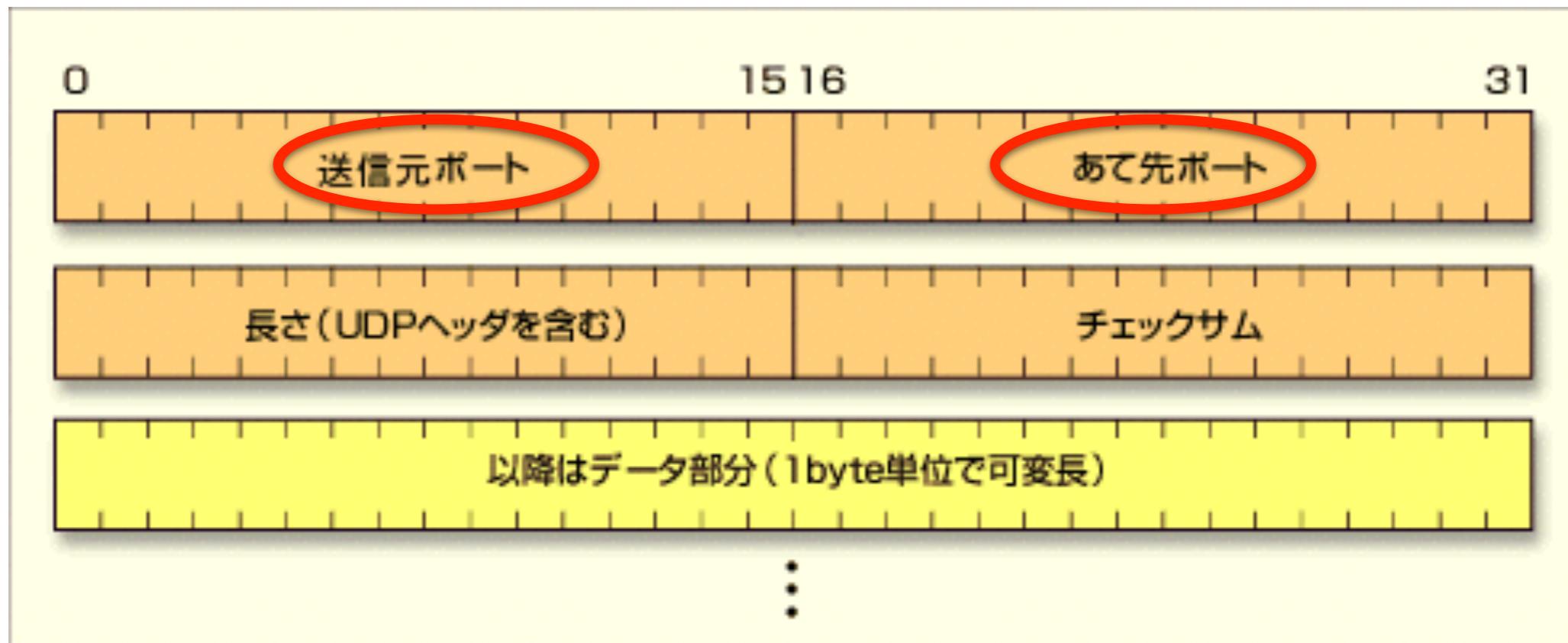
# TCPとUDPの比較

- UDPが即必要になる要件
  - WiFiアドホックネットワークやNATトラバーサルが必要な場合
  - それでもTCPとの併用は選択肢になる
- UDPの明白なデメリット
  - 数%ある「UDPブロックド」タイプのファイアウォールを全く越えられない。
  - 特に企業や学校の「ウェブ検索専用のゲーム禁止ネットワーク」でよく見られる
- 比較検討のポイント
  - 通信遅延
  - サーバのCPU消費
  - RUDPの性能
  - プログラミングコスト(デバッグ効率)

NATトラバーサルについては話が非常に長くなるので、ここでは割愛



# UDPパケットヘッダ



[http://www.atmarkit.co.jp/ait/articles/0310/09/news001\\_3.html](http://www.atmarkit.co.jp/ait/articles/0310/09/news001_3.html)

IPに対して「ポート番号」という、  
マシン内部の宛先の概念を追加しているだけ。

IPのパケットロス問題を何も解決しない。  
パケット断片が1個でもロスしたら届かない。

断片化が起きたらほぼ負け=1500バイト以上送ったら負け？

# TCPとUDPの比較：通信遅延

- UDPにはストールがなく、ただ単にパケットが消えるだけ
- TCPのストールについては前述。
- ここではパケットの到達までにかかる時間の長さだけを考える
- サーバ: Centos7 + westwood
- 富山>東京

```
[@mbp2013 ~]$ cd mrs_bench
[@mbp2013 mrs_bench]$ ./cl 210.140.72.117 1 10
server:210.140.72.117 conn_num:1 interval:0.010000 send_per_action:1 conn_type:1 nosleep:0 unreliable:0
[1509046566] loop:1 sentN:0 recvN:0(0) conn:0 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:nanms max:0.0ms min:99999000.0ms last:0.0ms
c[1509046567] loop:744 sentN:87 recvN:78(78) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:96.6ms max:96.6ms min:96.6ms last:96.6ms
[1509046568] loop:1482 sentN:179 recvN:170(92) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:87.3ms max:96.6ms min:78.1ms last:78.1ms
[1509046569] loop:2217 sentN:271 recvN:262(92) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:82.4ms max:96.6ms min:72.5ms last:72.5ms
[1509046570] loop:2953 sentN:363 recvN:351(89) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:76.9ms max:96.6ms min:60.5ms last:60.5ms
[1509046571] loop:3693 sentN:455 recvN:442(91) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:73.8ms max:96.6ms min:60.5ms last:61.3ms
[1509046572] loop:4427 sentN:547 recvN:535(93) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:73.1ms max:96.6ms min:60.5ms last:69.7ms
[1509046573] loop:5164 sentN:639 recvN:626(91) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:71.5ms max:96.6ms min:60.5ms last:61.9ms
^C
[@mbp2013 mrs_bench]$ ./cl 210.140.72.117 1 10 --udp
server:210.140.72.117 conn_num:1 interval:0.010000 send_per_action:1 conn_type:2 nosleep:0 unreliable:0
[1509046577] loop:1 sentN:0 recvN:0(0) conn:0 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:nanms max:0.0ms min:99999000.0ms last:0.0ms
c[1509046578] loop:732 sentN:85 recvN:81(81) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:26.1ms max:26.1ms min:26.1ms last:26.1ms
[1509046579] loop:1461 sentN:176 recvN:171(90) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:33.0ms max:39.9ms min:26.1ms last:39.9ms
[1509046580] loop:2192 sentN:267 recvN:262(91) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:30.7ms max:39.9ms min:26.1ms last:26.1ms
[1509046581] loop:2928 sentN:359 recvN:352(90) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:30.0ms max:39.9ms min:26.1ms last:27.9ms
[1509046582] loop:3658 sentN:450 recvN:443(91) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:33.0ms max:45.1ms min:26.1ms last:45.1ms
[1509046583] loop:4389 sentN:542 recvN:533(90) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:31.8ms max:45.1ms min:25.9ms last:25.9ms
[1509046584] loop:5120 sentN:633 recvN:624(91) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:31.3ms max:45.1ms min:25.9ms last:27.8ms
[1509046585] loop:5850 sentN:724 recvN:713(89) conn:1 dis:0 err:0 recvB:0.0(0.0) MiB RTT avg:33.5ms max:49.6ms min:25.9ms last:49.6ms
^C
```

UDP速い

通信距離が長くなるほど明らかにUDPが速くなるのは、  
途中のルータすべての処理負荷の蓄積が無いため。  
開発中のゲームで、20~30msの遅延は問題になるか？



# TCPとUDPの比較：サーバのCPU消費

- libuv\_bench でTCPとUDPのエコーバーで比較(top)
- TCP: Centos7 + westwood
- IDCFクラウドのデータセンター内通信を使用
- 600Mbpsを送信

```
%Cpu(s): 1.3 us, 9.1 sy, 0.0 ni, 88.9 id, 0.0 wa, 0.0 hi, 0.6 si, 0.0 st
```

TCP 合計10.4%

```
%Cpu(s): 0.9 us, 5.3 sy, 0.0 ni, 93.5 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
```

UDP 合計6.2%

UDPのほうがユーザー時間もシステム時間も小さい  
TCPはOSが接続状態(コネクション)を管理しているため

libuv\_benchは弊社が独自に実装した  
libuvを用いたベンチマークプログラム



# TCPとUDPの比較：RUDPの性能

- RUDP : Reliable(信頼できる) UDP
- UDPにTCPの機能の一部を追加したプロトコルの総称
- モノビットエンジン製品では、再送機能と順序保証を単純なスライディングウインドウ式で実装したRUDPであるENetの独自修正版を使用。
- 富山>東京 パケットロス10%、遅延100ms追加で比較 mrs\_bench

```
avg:313.4ms max:337.6ms min:289.2ms last:289.2ms
avg:327.9ms max:357.0ms min:289.2ms last:357.0ms
avg:327.9ms max:357.0ms min:289.2ms last:357.0ms
T avg:390.5ms max:620.1ms min:289.2ms last:348.9ms
avg:390.5ms max:620.1ms min:289.2ms last:348.9ms
T avg:405.0ms max:620.1ms min:289.2ms last:477.3ms
T avg:392.5ms max:620.1ms min:289.2ms last:317.6ms
avg:432.3ms max:710.9ms min:289.2ms last:710.9ms
avg:454.0ms max:710.9ms min:289.2ms last:627.4ms
avg:442.7ms max:710.9ms min:289.2ms last:340.9ms
RTT avg:441.1ms max:710.9ms min:289.2ms last:426.0ms
TT avg:445.4ms max:710.9ms min:289.2ms last:492.3ms
RTT avg:434.4ms max:710.9ms min:289.2ms last:302.7ms
RTT avg:429.4ms max:710.9ms min:289.2ms last:363.8ms
RTT avg:429.4ms max:710.9ms min:289.2ms last:363.8ms
RTT avg:468.4ms max:1015.3ms min:289.2ms last:1015.3ms
RTT avg:479.5ms max:1015.3ms min:289.2ms last:645.5ms
```

TCP westwood

ACKパケットも連續ロスした場合はどちらも1000ms以上遅れることもある。

ENetは少し平均遅延が長いが健闘している

```
T avg:nanms max:0.0ms min:99999000.0ms last:0.0ms
T avg:1086.6ms max:1086.6ms min:1086.6ms last:1086.6ms
T avg:1086.6ms max:1086.6ms min:1086.6ms last:1086.6ms
MiB RTT avg:1283.7ms max:1928.8ms min:835.7ms last:835.7ms
B RTT avg:1023.7ms max:1928.8ms min:243.5ms last:243.5ms
B RTT avg:866.9ms max:1928.8ms min:239.7ms last:239.7ms
B RTT avg:787.1ms max:1928.8ms min:239.7ms last:388.3ms
B RTT avg:700.0ms max:1928.8ms min:177.5ms last:177.5ms
MiB RTT avg:647.0ms max:1928.8ms min:177.5ms last:275.8ms
B RTT avg:605.4ms max:1928.8ms min:177.5ms last:272.9ms
B RTT avg:605.4ms max:1928.8ms min:177.5ms last:272.9ms
MiB RTT avg:567.9ms max:1928.8ms min:177.5ms last:284.0ms
MiB RTT avg:567.9ms max:1928.8ms min:177.5ms last:284.0ms
MiB RTT avg:542.7ms max:1928.8ms min:177.5ms last:265.7ms
MiB RTT avg:518.9ms max:1928.8ms min:177.5ms last:232.7ms
MiB RTT avg:514.5ms max:1928.8ms min:177.5ms last:457.2ms
```

RUDP ENet



# TCPとUDPの比較：RUDPのサーバCPU

- 100Mbps送受信のMRSサーバで比較(top)

%Cpu(s): 0.9 us, 1.4 sy, 0.0 ni, 97.2 id, 0.0 wa, 0.0 hi, 0.4 si, 0.0 st	TCP 合計2.3%
%Cpu(s): 2.0 us, 6.3 sy, 0.0 ni, 91.1 id, 0.0 wa, 0.0 hi, 0.5 si, 0.0 st	RUDP 合計8.3%
%Cpu(s): 1.4 us, 4.6 sy, 0.0 ni, 93.7 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st	RUDP 合計6% (信頼性なしモード)

これが、30年近くかけて大規模環境で叩かれてきたLinuxのTCPと  
主にゲームで使われてきた10年選手のENetの差・・・！

しかし100Mbpsで1ヶ月通信すると、  
通信費がAWSなら50万円以上だがCPU費用は1000円以下。  
CPUコストは誤差といえる。

モノビットエンジンMRSでは、パケットごとに信頼性のあり・なしを設定可能



# TCPとUDPの比較：RUDPの使い方

- ・ 信頼性(再送、順序)が必要なパケットは再送をさせるが、そうではないパケットは再送させない
- ・ モノビットエンジンではパケットごとにon/offできる
- ・ 信頼性ありパケットの用途
  - ・ ログイン・ログアウト、ショップ、マッチング、アイテム取得など
- ・ 信頼性なしパケットの用途
  - ・ キャラクターの位置更新、エフェクト、ボイスチャット、映像データなど
- ・ 以下はMRSでのコード例

```
mrs_write_record( connection, 0, 0x01, "hello", 5 ); // 再送や順序操作が行われる  
mrs_write_record( connection, MRS_RECORD_OPTION_UDP_UNRELIABLE, 0x01, "hello", 5 ); // 再送をしない  
mrs_write_record( connection, MRS_RECORD_OPTION_UDP_UNSEQUENCED, 0x01, "hello", 5 ); // 順序整列をしない  
mrs_write_record( connection, MRS_RECORD_OPTION_UDP_UNRELIABLE | MRS_RECORD_OPTION_UDP_UNSEQUENCED, 0x01, "hello", 5 ); // 再送も順序整列もしない
```



# TCPとUDPの比較：プログラミングコスト

- ・ UDPとTCPの違いはミドルウェアが自動的に吸収、プログラミングの作業自体は簡単
- ・ 以下はモノビットエンジン製品”MRS”におけるコード例

```
mrs_connect( MRS_CONNECTION_TYPE_UDP, "10.6.0.199", 22222, 5000 );      # UDPクライアント (RUDPに対応)  
mrs_connect( MRS_CONNECTION_TYPE_TCP, "10.6.0.199", 22222, 5000 );      # TCPクライアント  
  
tcp_server = mrs_server_create( MRS_CONNECTION_TYPE_UDP, "0.0.0.0", 22222, 100 );    # UDPサーバー (RUDPに対応)  
tcp_server = mrs_server_create( MRS_CONNECTION_TYPE_TCP, "0.0.0.0", 22222, 100 );    # TCPサーバー  
  
mrs_write_record( connection, 0, 0x01, "hello", 5 );
```

切り替えはライブラリのフラグ一発なので、

TCPで開発を始めてから、隨時UDPにきりかえてテストをしていくことができる。

また、”UDPプロックドNAT”環境でもスイッチャー発で切り替えることができ、問題を解消できる。



# TCPとUDPの比較：プログラミングコスト

- ・ トラブルシューティングはUDPが一手間多く必要
- ・ TCPではOSがコネクションの状態を管理しているため、  
netstatやss, nethogsなどの標準ツールを用いて通信状態の確認がやりやすい
- ・ UDPではコネクションをアプリケーションが管理しているため、tcpdumpや  
wiresharkを用いた確認作業が必要
- ・ モノビットエンジン用のパケット解析スクリプト"mrsdump"を準備中



# TCPとUDPの使い分け

質問

答え

UDPは速いか？

遠いとかなり速い

UDPはサーバが軽いか？

UDPは軽いがRUDPは重い。ただし  
誤差の範囲

UDPはプログラミングが面倒  
か？

実装は変わらないが  
トラブルシュートが面倒

結論: ゲームでは、 UDPとTCPの切り替えが可能な通信ミドルウェアを使えば、  
「UDPブロックドNAT」の問題も解消できるので、 UDPが第一選択肢となる。  
では、開発中のゲームではどうすればよいか？



# ゲームで必要な低遅延と帯域幅

ゲームジャンル	遅延	帯域	プロトコル
FPS(PvP公式戦)	5~15ms	500Kbps~1Mbps	UDP+RUDP
FPS(PvP一般)	10~30ms	200~300Kbps	UDP+RUDP
対戦格闘(1vs1)	5~30ms	50~200Kbps	UDP+RUDP
レースゲーム(PvP)	10~50ms	100~200Kbps	UDP+RUDP
スマブラ系PvP	10~50ms	100~300Kbps	UDP+RUDP
VR空間共有	10~100ms	100K~1Mbps	UDP+RUDP/TCP
FPS(PvE)	30~100ms	200Kbps	UDP+RUDP/TCP
RTS/MOBA	20~100ms	200~500Kbps	UDP+RUDP/TCP
TPS	50~100ms	100~300Kbps	RUDP/TCP
スマホアクション	50~200ms	50~100Kbps	RUDP/TCP
ボイスチャット	50~200ms	50K~300Kbps	RUDP/TCP
超多人数戦争	100~300ms	300Kbps~1Mbps	RUDP/TCP
MMORPG	100~500ms	50~200Kbps	RUDP/TCP
スマホターン制	200~1000	20~100Kbps	TCP

上記は一人あたり必要な帯域幅



# モノビットのミドルウェア製品

- MRS “Monobit Revolution Server”
  - UDP/RUDP/TCP/WebSocketに対応した全プラットフォーム向け通信ミドルウェア
  - <http://www.monobitengine.com/mrs/>
- MUN “Monobit Unity Network”
  - Unity向けの導入が非常に簡単なMRSベースのミドルウェア
  - <http://www.monobitengine.com/mun/>



# モノビットの調査ツール

- ・ 社内で調査やお客さまサポートに使う独自ツール群
  - ・ mrs\_bench : MRSのベンチマークツール
  - ・ enet-benchmark : ENetのベンチマークツール
  - ・ libuv\_bench : libuvのベンチマークツール
  - ・ node\_bench : node.jsのベンチマークツール
- ・ MRSだけではなく下位レイヤー単体の調査を綿密に行って、問題の切り分けを行っています。
- ・ ネットワークゲーム実装で困ったり、何か工夫したいときは遠慮なくお声掛けください!

