第 2 章

もしインターネットの 1 秒が 1 年だったら

hakatashi・Mine02C4

2016 年、インターネットが日本中のあらゆる人間に行き渡るようになってから、すでに 10 年単位の時間が経過しています。今日においてインターネットを支えるネットワーク技術が重要であることは言うまでもありませんが、実際にネットワークでどのタイミングで何が起こり、どれくらいの時間が費やされるのかということを身を持って体感している人は、たとえネットワークに精通している人でも少ないのではないでしょうか？ この記事では、1 秒というわずかな時間を 1 年にまで拡大し、ネットワーク上で何が起こっているかを人間スケールでざっくりと解説していきます。

2.1　はじめに

たぶんはじめまして。博多市 (@hakatashi) です。今回は技術書典向けの小企画として、「インターネットの 1 秒がもし 1 年だったら」というだいぶ抽象的で怪しい記事を書こうと思います。

インターネットというのは光の速さを身をもって感じることができるメディアです。先日、Hiraku Nakano さんの「composer の遅さをまじめに考える」というスライド[*1]を拝見したのですが、そこでは PHP のパッケージマネージャーである composer が遅い原因として、**「光の速さが遅すぎる」**という理由が挙げられていました。

ユニークな考え方ですが言われてればたしかに道理で、日本からアメリカのサーバーまでハンドシェイクで何度も往復していると、確かに光といえど 100 ミリ秒単位で時間を浪費しています。サーバーが近ければ通信は早いというのは誰もが知っていることですが、光の速度のせいだと言われるとなにやら圧倒されるものがあります。

この記事では、そんなネットワークの微視的な時間スケールについて徹底的に解剖します。一回の通信を解析し、それぞれ時系列順に細かく分解し、それぞれの操作でどれくらいの時間が費やされ、どんなイベントがいつ発生するのかを逐一追っていきたいと思います。

とはいえ、ネットワーク通信における時間経過はミリ秒単位で数えられます。今回は、そんな微

[*1] http://www.slideshare.net/hinakano/composer-phpstudy

細な時間の移り変わりをなるべくわかりやすくするため、**通信上の 1 秒という時間を 1 年にまで引き伸ばし、約 3000 万倍**の時間スケールでネットワークのイベントを追跡してみます。

対象読者

- インターネットは毎日使っているが、それが動く仕組みについてはあまり理解していない人
- ネットワークについて基礎的な仕組みは理解しているが、実際のネットワーク通信のパケットを見たことがない人
- パケットキャプチャの経験があるが、それを人間スケールで体感して理解したい人
- クライアントちゃんかわいいよクライアントちゃん

共著者について

本記事の筆頭筆者は hakatashi ですが、今回の記事を書くにあたり、同じ SunPro メンバーでありネットワークスペシャリスト取得者の **Mine02C4**[*2]から、企画段階からの多大な協力と大幅な加筆を得ることができました。

ネットワーク知識不足の hakatashi にパケット解析を指南し、本記事では特に TLS 関連の項目の執筆、および全体のレビューを担当しています。共著者として名を連ねていますが、彼の協力なしでは本記事は恐ろしくクオリティの低いものになったであろうことは疑いようがなく、その貢献度に関しては計り知れません。どうぞお見知り置きください。

シチュエーション

今回解剖する対象の通信として、**HTTPS** による通信をメインに解析しました。ふだん我々がブラウザから毎日使っているプロトコルなのは言うまでもなく、SSL/TLS の処理を挟むので、それなりに面白い結果になるのではないでしょうか。

また、HTTPS 通信を行う際に必要となる ARP や DNS 通信についても述べていきます。

さらに、通信を行うサーバーとクライアントの場所ですが、今回は、技術書典の会場でもある東京のインターネット環境から、さくらインターネットが誇る、北海道の**石狩データセンター**までの通信を行います。

Google マップによると、東京から、石狩データセンターがある石狩市新港中央までは 1177.3km。光ファイバーに用いられる石英ガラスの屈折率は 1.50 程度[*3]なので、光は約 20 万 km/s でこの距離を移動します。つまり、東京から石狩までの物理的な片道時間は約 5.9 ミリ秒となります。瞬く間もないほどの時間ですが、時間スケールを 1 年に引き伸ばすと 2 日と 3 時間半もかかります。往復で 4 日と 7 時間。石狩までぶらり旅といった感じですね。

1 秒→ 1 年

1 秒が 1 年になった世界の様子をもう少し見てみましょう。

[*2] HP: http://mine02c4.nagoya/, Twitter: @mine_studio

[*3] 光学ガラス材料 - シグマ光機株式会社 http://www.products-sigmakoki.com/category/opt_d/opt_d01.html

真空中の光は時速 34km で移動します。先ほど出てきた光ファイバー中の光速は時速 23km 程度になるので、自転車か、休憩しながらのドライブ旅程度の速さになります。ちなみに男子マラソンの世界記録は時速 20.6km です。[*4][*5]

3GHz の CPU は、こんな世界でも 1 秒間に 95 回という高橋名人の 6 倍のクロックを刻みます。PC3-12800 のメモリーの転送速度は、もはやダイアルアップ接続よりも遥かにナローバンドですが、最大で 1 秒間に 3200 ビットの情報を読み出すことができます。

制約など

今回パケットを解析する上で、話を簡便化するためにいくつかの制約を加えています。

- DNS のシステムを明確に示すため、ローカルホストに DNS サーバーを置き、そこからリクエストのたびにルート DNS サーバーからアドレスを引いています。
- 複雑かつ時間がかかりすぎるため、ネームサーバーでの DNSSEC の検証処理は無効化しています。
- SSL/TLS の証明書検証処理は無効化しています。

測定環境

測定には、ラップトップマシン上の Ubuntu の仮想環境を使用しました。一回の測定ごとに DNS キャッシュと ARP キャッシュをクリアし、なるべくクリーンな状態で測定を行いました。

また、先程も述べたとおり DNS サーバーはローカルホストに設置し、そこから外部に向けてアドレスを引くようにしました。

サーバーは先ほど述べたさくらの石狩データセンターへのアクセスを確実に測定するため、今回の記事のためにさくら VPS の石狩リージョンのサーバーを調達しました。OS はさくら VPS の初期 OS である CentOS 6.8 で、デフォルトの Apache 2.2.15 をほとんどそのまま使用しました。SSL は自己署名証明書で通信し、証明書の検証は無効化してあります。

クライアントは、Windows 上の仮想マシンの Ubuntu 14.04.4 を用いました。クライアントアプリケーションは cURL を用いています。ネームサーバーはローカルホストに BIND 9.9.5 を立て、キャッシュを無効化した上でルートサーバーからアドレスを引くよう設定してあります。

通信内容は、100KB(102400 バイト) のバイナリファイルを GET リクエストで取得するもので、HTTP パラメータはほぼ cURL のデフォルトを使用しています。

パケットキャプチャはクライアント側とサーバー側で同時に行い、tcpdump で同じ通信を 2 度キャプチャして解析しています。時刻に関しては毎度 NTP で時刻補正しましたが精度が得られなかったのでサーバー側とクライアント側の測定値を見て適当に補正を行いました。

今回記事中で使用した数字は、以上の条件のもとに行った 10 回の測定結果の平均値をもとにしています。

[*4] 2014 年ベルリンマラソンのデニス・キメット選手の記録、2 時間 2 分 57 秒より。

[*5] ただし、この速度で走行するとこの世界では体重が 26% 増えるのでオススメしない。

サーバーちゃんとクライアントちゃん

今回、せっかく時間軸を人間スケールで語るので、通信処理の登場人物も人間という体で解説していきます。

この話の主人公は「クライアントちゃん」と「サーバーちゃん」です。クライアントちゃんは東京で悠々とした生活を送っている普通の女の子であり、趣味は暗号と文通です。一方サーバーちゃんは石狩のデータセンターで、ひしめき合うたくさんのサーバーとともに、Web サーバーとして多忙な毎日を送っています。

こんな対称的な 2 人が、今回は年明けと同時に通信を開始します。さらに、通信は手紙のやり取り、IP アドレスは住所、ルーターは郵便ポスト……というように、適宜シチュエーションに合わせた読み替えを行っていきます。あくまで例えですので、あまり正確ではないかもしれませんが、今回の目的はあくまで通信処理の全貌をふわっと理解することなので、あまり厳密に考えず、空っぽの頭で読んでください。

それでは、クライアントちゃんとサーバーちゃんの通信の始まりです。

2.2 ルーターのアドレス解決

1 月 1 日 午前 0 時 0 分 ARP リクエスト送信

あけましておめでとうございます。NHK の「ゆく年くる年」を見ながらインターネットの年が明けます。

この瞬間、東京でまったりと紅白歌合戦を見ていたクライアントちゃんは、石狩にいるサーバーちゃんに聞きたいことがあるのを思い出しました。長い長い通信の始まりです。

さっそくクライアントちゃんはサーバーちゃんに手紙を書くことにしました。

クライアントちゃんは何もわかりません。郵便ポスト (ルーター) の場所も、サーバーちゃんの住所 (IP アドレス) も、いつも使っていた住所録 (DNS サーバー) の場所も覚えていません。キャッシュを削除したクライアントちゃんは記憶を失ってしまったのです。

覚えているのは、ただ石狩にいるはずのサーバーちゃんのことだけ……。

困ったクライアントちゃんは、まずは郵便ポスト (ルーター) を探すことにしました。家の中や向こう三軒両隣のことならともかく、遠く石狩に住むサーバーちゃんに手紙を届けるには、何よりポストがなければ話が始まりません。

クライアントちゃんはもうネットワークに接続しているので、ルーターの IP アドレスはすでに分かっています。しかし実際にルーターと通信するには、ルーターの MAC アドレスが必要です。これを引くために、**ARP**（Address Resolution Protocol）と呼ばれるプロトコルを用います。

ARP リクエストは IP アドレスに対応する MAC アドレスを検索するリクエストです。MAC アドレスの**ブロードキャストアドレス** (FF:FF:FF:FF:FF:FF) に向けて発信することで、同一ネットワーク内のすべての機器にこのリクエストを送信することができます。

クライアントちゃんは街中に響く声でポストの場所を尋ねました。

1 月 1 日 午前 3 時 26 分 ARP レスポンス受信

いったい今何時だと思っているんでしょうか。クライアントちゃんの声は街中に響き渡り、近隣住民からたいへん大目玉を食らいましたが、努力の甲斐あって、3 時間半後に親切な人がポストの場所を教えてくれました。

ARP は、ブロードキャストでネットワーク全体に配信されたイーサネットフレームに対して、尋ねられている IP アドレスが自分のものであれば MAC アドレスを応答するというシンプルな仕組みで動作します。

今回は事前に ARP キャッシュを削除してから通信を行ったので、通信前に必ず ARP の問い合わせが走るようにして計測したのですが、そもそも ARP は頻繁にキャッシュを消去します。[*6] クライアントちゃんはとても忘れっぽいのです。

2.3 DNS 解決

1 月 1 日 午前 5 時 7 分 DNS クエリ送信 (1 回目)

親切なおじさんのおかげで、クライアントちゃんは郵便ポストの場所を知ることができました。またすぐ忘れてしまうのですが、この世界では約 900 年後のことなので、特に気にすることはありません。

となると次にクライアントちゃんが知るべきはサーバーちゃんの住所 (=IP アドレス) です。郵便ポストを見つけても相手の住所がわからないと手紙は送れません。サーバーちゃんの住所を知るために、クライアントちゃんはルートサーバーに住所を問い合わせることにしました。

電話番号に電話番号問い合わせサービスがあるように、ネットワークの世界にも、ドメイン名から相手のアドレスを問い合わせるための **DNS**（Domain Name Service）があります。

電話番号の場合は 104 番という番号を知っていれば他の番号を問い合わせることができます。インターネットにおける“104”は、世界に 13 個存在する**ルートサーバー**のアドレスです。今回は、13 個の中で唯一日本の団体が管理している、M ルートサーバーに問い合わせるようにルートサーバーの設定を変更しています。[*7]

サーバーちゃんの住所を一秒でも早く正確に知りたいクライアントちゃんは、世界中に数ある DNS サーバーの中でも最も権威あるルートサーバーに一筆したためることにしました。あらゆるキャッシュを削除したクライアントちゃんも、一番大事なルートサーバーの住所は覚えています。あれこれ悩んで手紙を書き、ようやく完成したのはそれから 1 時間半後のことでした。早朝の冷たい冬風が骨身に染みます。クライアントちゃんはコートを厚めに羽織ってポストに手紙を投函しに行きました。

一刻も早く返事が帰ってくることを願って……。

[*6] Ubuntu の場合、キャッシュの有効時間は 15 分程度。

[*7] もっとも、ルートサーバーはクラスタ構成になっており国単位で分散しているため、必ずしも問い合わせたマシンが日本に存在するかどうかは保証されないのですが……。

1 月 14 日 午前 3 時 57 分 DNS レスポンス受信 (1 回目)

最初に手紙を送ってから 2 週間が経過しました。冬休みも終わって正月気分もようやく抜けてくる頃です。もうこの時点で直接会いに行ったほうが手っ取り早いんじゃないかという気がしますが、石狩は遠いのです。そんな気軽に会いに行けないのです。たぶん。

ともあれ、ルートサーバーからようやく返事が帰ってきました。郵便局が怠慢なのかルートサーバーがお役所仕事をしてるのか知りませんが、とにかくこれでサーバーちゃんの住所を知ることができた……というわけではありません。

ルートサーバーは世界中のすべてのマシンの IP アドレスを記録しているわけではありません。ドメイン名を問い合わせられた DNS サーバーは、自らが委任する DNS ゾーンのネームサーバーの情報を返します。今回問い合わせたドメイン名はさくらの VPS サーバーにデフォルトで割り当てられたドメイン[*8]なので、ルートサーバーは jp サブドメイン[*9]のネームサーバー[*10]を返してきました。つまり jp ドメインのことは jp ドメインの担当者に聞けということです。

1 月 14 日 午前 9 時 52 分 DNS クエリ送信 (2 回目)

jp ドメインの担当者というのは要は **JPRS**（日本レジストリサービス）のことです。サーバーちゃんの住所を聞き出す手がかりを得たクライアントちゃんは、夜が明けるのを待ってから今度は JPRS の DNS サーバーに対して手紙を書きました。

実を言うと、jp ドメインのネームサーバーを管理しているのは JPRS その人ですが、実際の運用は、JPRS を含めた 5 つの独立した組織によって行われています。これはルートサーバーが世界に 13 個あるのと同じく、ネームサーバーがダウンした際のリスクを可能な限り最小限に抑えるためです。

2 回目の DNS レイヤーの内容は、トランザクション ID を除いて前回ルートサーバーに対して送ったのと全く同じ内容です。DNS はなるべく高速に動作するように簡明に設計されているため、このようにパケットの内容を容易に再利用できるようになっています。決してクライアントちゃんがサボっているわけではありません。

1 月 27 日 午後 2 時 30 分 DNS レスポンス受信 (2 回目)

ふたたび待つこと 2 週間。すでに 1 月が終わりに近づいてきました。

クライアントちゃんの元に JPRS からの返信が届きました。サーバーちゃんの住所はまだわかりません。今度は sakura.ne.jp の権威サーバーである、さくらインターネットのネームサーバー[*11]の情報が返ってきました。

ここに来てようやくさくらインターネットの影に辿り着きました。サーバーちゃんの住所ゲットまであと一息です。

[*8] .vs.sakura.ne.jp で終わるドメイン名

[*9] .jp は TLD(Top Level Domain) なのでサブドメインと呼んでいいか微妙ですが……

[*10] a.dns.jp, b.dns.jp, c.dns.jp, d.dns.jp, e.dns.jp, f.dns.jp, g.dns.jp の 7 つ

[*11] ns1.dns.ne.jp, ns2.dns.ne.jp

1 月 27 日 午後 6 時 56 分 DNS クエリ送信 (3 回目)

同じやり取りも 3 回繰り返すと飽きてしまいますね。けれどクライアントちゃんはサーバーちゃんに手紙を届けるために、めげずに DNS のパケットを送り続けます。今度は先ほど入手したさくらインターネットのネームサーバーの住所に対して、みたびサーバーちゃんの住所を尋ねる手紙を出します。

ところで、先ほどからクライアントちゃんは気軽にあちこちに DNS パケットを送ったり受け取ったりしていますが、これは DNS が **UDP プロトコル**の上で動作しているからです[*12]。UDP は、後ほど解説する TCP と異なり、相手のマシンとの接続を確立する必要がないため、気軽に任意のアドレスにいきなりデータを送りつけることができます。

逆に言うと、TCP のような再送要求やエラー訂正などの機能を持たないため、伝送経路のどこかでパケットが損失した場合、クライアントちゃんは返事を待ちぼうけということになってしまいます。

もっとも、それもタイムアウトするまでですが。

2 月 10 日 午前 9 時 35 分 DNS レスポンス受信 (3 回目)

2 月に突入しました。幸い 3 度目のクエリも待って待って待ちぼうけという事にはならず、ポストに投函してから 2 週間経って再び返事が帰ってきました。そこには待ち望んでいたサーバーちゃんの住所がくっきりと書かれています。

これにて名前解決完了。ようやくサーバーちゃんに対してパケットを送りつけることができるようになります。もうあちこちのネームサーバーとパケットをやりとりする必要はありません。

年が明けてから 41 日間、実時間にして 110 ミリ秒が経過しました。この調子で年内にサーバーちゃんとの通信を終えることができるのでしょうか。クライアントちゃんの通信はまだまだこれからです。

2.4 TCP 接続ハンドシェイク

2 月 19 日 午後 3 時 27 分 TCP ハンドシェイク SYN パケット送信

各地のネームサーバーの協力を得て、無事サーバーちゃんの住所を入手したクライアントちゃんですが、お淑やかなクライアントちゃんはいきなり本題の手紙を送りつけるような真似はしません。まずはサーバーちゃんにご挨拶をします。

TCP 上の通信では、データ伝送を行う前にコネクションの確立という処理を行う必要があります。これは、相手のサーバーが通信可能な状態であることを保証したり、以降のデータが正しい順序で到着することを保証するための**シーケンス番号**を互いに交換したりするためです。

シーケンス番号とは、現在のパケットが送信しているデータが、全体のデータのうちのどの部分に該当するのかを示す値であり、ハンドシェイクで最初にランダムな値にセットされ、以降データを送信するごとに増加していく値です。TCP は双方向通信なので、このシーケンス番号はサー

[*12] DNS 自体は TCP 上でも動作することが義務付けられています。

バーとクライアントで別々の値を保持しています。

クライアントちゃんはランダムに生成したシーケンス番号を端に添えて、サーバーちゃんに文通してよいかを問う内容の手紙を作りました。色よい返事が返ってくることを期待して、再びポストに投函しました。手紙はいよいよ石狩に向かいます。

ところで、サーバーちゃんの住所が判明してから最初にサーバーちゃんにコンタクトをとるまで 9 日もかかっています。きっとバのつくイベントで忙しかったのでしょう。妬ましい。[*13]

2 月 27 日 午後 6 時 30 分 TCP ハンドシェイク SYN パケット受信

北海道——新千歳空港から車で 50 分の石狩の大地に、目的のデータセンターは存在します。冬の北海道の空気は冷たく厳しく、この時期の気温は昼間でも 0 度を上回ることはありません。この冷涼な外気がサーバールームから効率的に排熱するのです。

すぐ脇を通る道は国道 337 号線です。地元ではかつて天売島に住んでいた鳥の名前からとって「オロロンライン」と呼ばれるこの道は、眼前に手稲山を望むゆったりとした広い道路です。小樽からこの道を進んで左手側、空港のターミナルを髣髴とさせる白い横長な建物が石狩データセンターです。

この場所でサーバーちゃんは他のサーバーと肩を並べて[*14]、静かに 443 番ポートを開けて待ち続けています。今回キャプチャしたパケットの中で最も数が多かったのは ARP のパケットでした。512 台[*15]近い数のサーバーが ARP で常に囁きあい、互いの居場所を確認しあっている状態といえるかもしれません。

そんな退屈な生活の中で、サーバーちゃんはクライアントちゃんからの手紙を受け取りました。クライアントちゃんが手紙を投函してから 8 日目のことです。伝送には 22 ミリ秒かかりました。冒頭では東京と石狩の物理的な片道時間は 5.9 ミリ秒と述べましたが、当然これは理想的な通信のことであり、実際には無線通信やルーティングなどにおけるオーバーヘッドによってそれ以上の時間がかかります。

受け取った手紙は一般的な TCP ハンドシェイクでした。手紙にはシーケンス番号が添えられています。クライアントちゃんからの久しぶりの手紙に喜んだサーバーちゃんは、喜んで通信を受け入れました。

2 月 27 日 午後 8 時 11 分 TCP ハンドシェイク SYN+ACK パケット送信

サーバーは TCP による通信を受け入れた証として、クライアントからの SYN パケットに対する応答を返します。サーバーはクライアントから受け取ったシーケンス番号を認識し、これに 1 を加えた値を返答パケットに記して送ります。シーケンス番号は送信するデータの先頭バイトを表すので、本来はデータを送信しない段階では加算しないのですが、ハンドシェイクにおいてはパケットを正しく受け取った印として特別に 1 を加算します。

同時に、サーバー側でもシーケンス番号を生成して返信用のパケットに記します。これでサーバーとクライアントの間で一対のランダムなシーケンス番号が初期化されます。

[*13] 実際の理由はおそらくローカルホストの DNS サーバーからアプリケーション (curl) に DNS 情報を受け渡す際にオーバーヘッドが発生するためです。

[*14] もちろん VPS なので物理的なサーバーマシンとして存在しているわけではありませんが。

[*15] 今回使用したサーバーはサブネットマスク 23 ビットという中途半端な値のネットワークに繋がっていました。

サーバーちゃんはクライアントちゃんに向けて通信可能な旨を記した手紙を書きました。クライアントちゃんからの手紙と同じく、隅っこにシーケンス番号を記しておきます。このパケットは SYN と ACK のフラグが立てられているため、**SYN+ACK パケット**などと呼ばれます。

3 月 9 日 午前 3 時 43 分 TCP ハンドシェイク SYN+ACK パケット受信

今度は石狩から東京へと手紙が運ばれます。配達にはふたたび 10 日近い時間を要しました。

クライアントちゃんにとってはサーバーちゃんからの初めての手紙です。だいぶ非常識な時間に配達された手紙ですが、クライアントちゃんは飛び起きて、サーバーちゃんからの手紙をじっくり読み、さっそく返事に取りかかりました。

クライアントは、サーバーからのハンドシェイクを受け取ると、先ほど説明したシーケンス番号の他に、Window Size Value や Maximum Segment Size などの TCP 通信に必要な値を確認し、記録しておきます。

3 月 9 日 午前 4 時 5 分 TCP ハンドシェイク ACK パケット送信

サーバーちゃんからの手紙で、サーバーちゃんが手紙を出せる状態であることを確認したクライアントちゃんは、その手紙に問題がないことを伝えるため、ハンドシェイクを完了させる手紙を送ります。

クライアントもサーバーと同じく、シーケンス番号を認識した証として、サーバーから送られたシーケンス番号に 1 を加えて返答します。このパケットにより両者の間でそれぞれのシーケンス番号が共有され、お互いにデータを送り合う事ができるようになります。

3 月 16 日 午後 3 時 21 分 TCP ハンドシェイク ACK パケット受信

ふたたび手紙は東京から石狩へ。サーバーちゃんはクライアントちゃんからちゃんと返事が届いたことにほっと安心しました。これで、いつでも通信を受け入れることができます。

このように、TCP のハンドシェイクでは通信路を 3 回通る必要があるため、**3 ウェイ・ハンドシェイク**とも呼ばれます。

2.5 TLS ハンドシェイク

TLS ハンドシェイクが終わって、クライアントちゃんとサーバーちゃんは無事通信が開始できるようになりました。クライアントちゃんはさっそく本題の質問を投げかけようと思ったのですが、ここでふと思ったことがあります。

クライアントちゃんとサーバーちゃんは、ハガキを使って互いにやり取りをしています。ハガキには特に何も細工をしていないので、書いている内容は周りの人には丸見えです。クライアントちゃんがポストに投函しに行くまではいいとしても、そこから先、ルーターから先のことに関しては何も保証できません。実はポストの中に盗撮カメラが仕掛けてあるかもしれませんし、郵便局員がハガキの内容をチラ見するかもしれませんし、サーバーちゃんの上司に内容を検閲されているかもしれません。

これは困ります。花も恥じらう乙女であるところのクライアントちゃんは、サーバーちゃん以

外の誰にも手紙の内容を知られたくありません。クライアントちゃんはサーバーちゃんとのやり取りを暗号化するため、**TLS**（Transport Layer Security）を使用することにしました。

TLS は、TCP のようなコネクションの上で暗号技術を使って通信の機密性や完全性を確保するための仕組みです。公開鍵暗号を用いて安全に交換した鍵を使って共通鍵暗号を行い、暗号化されたコネクションをアプリケーションに提供します。

この記事は暗号理論の説明はいたしません。ちょこちょこ出てくる用語の説明はほとんどしていません。なので、やり取りする情報のさらなる意味を知りたい方は、ぜひ他の専門書を参照してください。

3 月 9 日 午後 7 時 27 分 ClientHello 送信

サーバーちゃんとのやり取りに暗号を使うことを決めたクライアントちゃんですが、具体的にどんな暗号を使えばいいのかはまだわかりません。クライアントちゃんはこう見えて暗号の達人なので、何種類もの暗号を駆使することができるのですが、相手と文通する以上、クライアントちゃんとサーバーちゃんが共通に理解できる暗号で会話しないといけません。

TLS そのものは暗号を行う方式ではなく、別に定められた暗号方式を使って通信を行うための仕組みです。そのため、TLS ハンドシェイクではこの暗号方式を決めることがコアになります。

暗号というものは時間が経つにつれて、その弱点が明らかになったり、コンピューターの性能向上とともに解読に必要な時間が短くなったりして弱くなります。そのため、情報セキュリティにおいて暗号を利用する際には、使用する暗号方式を新しい物に移行できることが重要になります。TLS は将来開発しうる暗号方式を受け入れる事ができる、一種のフレームワークとして設計されています。

今回はクライアントちゃんとサーバーちゃんとで暗号通信を行いますが、当然ながらお互いに使える暗号方式は限られており、事前にそれらはわかりません。TLS ではこの暗号方式を後述する **CipherSuite** と呼ばれる 16bit の識別子で取り扱います。ClientHello ではクライアントが使うことができる好みの CipherSuite のリストをサーバーに送ります。

他にもバージョン番号や、セッション再開 (resumption) や master secret 生成に使用する乱数、使用できる署名及びアルゴリズムのリストなどが送信されます。

クライアントちゃんは、サーバーちゃんとの通信に暗号を使いたいという話と、自分がマスターしている暗号のリストを書いた手紙をサーバーちゃんに送りました。手紙の書き出しは、もちろん「ハロー」です。

クライアント側のハンドシェイクは ACK パケットを送信した 3 月 9 日の段階ですでに完了しているので、これは 3 月 16 日にサーバーちゃんがクライアントちゃんからの ACK パケットを受け取る前の話になります。

CipherSuite

CipherSuite とは鍵交換アルゴリズム (Key Exchange)、暗号アルゴリズム (Cipher)、メッセージ認証符号 (MAC) の組み合わせであり、16bit の識別子が割り振られています。例えば TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 は鍵交換に ECDHE_RSA、暗号に AES_256_GCM、メッセージ認証符号に SHA384 を使用することを示しています。

master secret

ある意味ではハンドシェイクの成果物とも言える、通信当事者以外には秘密でありながら、通信当事者同士で共有された値です。長さは 48 バイトであり、同じく秘密に共有された premaster secret と、Hello で交換した乱数から擬似乱数関数 (PRF) を用いて生成されます。擬似乱数関数は TLS ではハッシュ関数の様な使い方をする関数であり、入力から出力が一意に定まるが、出力から入力を予測することは不可能な関数です。

3 月 19 日 午前 10 時 30 分 ClientHello 受信

それから 10 日後、世間ではそろそろ冬休みが始まる頃に、クライアントちゃんからの通信がサーバーちゃんのもとに届きました。

クライアントちゃんからの 3 日ぶりの手紙を喜び勇んで開封すると、中にはサーバーちゃんとのやり取りに暗号を使いたいという内容が書かれています。

実を言うと、わざわざ 443 番ポートに向けて手紙を送ってきた時点で予想はしていました。サーバーちゃんも暗号にはそれなりに精通している自信があるので、手紙に書いてある暗号リストの中から、自分が使える暗号でもっとも難しい暗号を選びます。今回、サーバーちゃんは先ほど例に挙げた TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 を選択しました。

ClientHello を受信したサーバーは、その中の CipherSuite のリストや署名、およびハッシュアルゴリズムのリストから、実際に使用する方式を選定します。この選定方法はアプリケーションによって異なり、管理者が自由に設定を行えます。

3 月 20 日 午後 7 時 20 分 ServerHello, Certificate, ServerKeyExchange 送信

サーバーちゃんは自分が選択した暗号を書いて、翌日の晩にはクライアントちゃんに返事を書きました。一日待つ必要があったのは、その他に証明書を用意したり署名をしたりといった、暗号通信に必要ないろいろな手続きが必要なためです。

TLS におけるサーバーからのこの返答は複数のメッセージで構成されることがあります。

ServerHello では実際に使用する CipherSuite とセッション再開や master secret 生成に使用する乱数などの情報を送ります。

Certificate ではサーバー証明書、及びその証明書を検証するために必要な証明書をクライアントちゃんに送ります。この証明書は先に決定した CipherSuite で使えるものでなければなりません。運用上、実際には証明書の方を認証局から発行してもらい、簡単にいじれるものではないため、証明書に適合した CipherSuite を選ぶことになります。この Certificate は任意ですが、これを送らないということは、通信先であるサーバーが、本当に想定されるサーバーなのかの手がかりを与えない、匿名な通信となります。暗号通信において、通信相手が攻撃者などではなく、意図した通信相手であることを保証することは重要です。

ServerKeyExchange では公開鍵や署名をクライアントちゃんに送ります。具体的なデータ内容などは CipherSuite で指定した鍵交換アルゴリズムによって異なります。この ServerKeyExchange は任意であり、単純な RSA のように Certificate に公開鍵や署名を含む場合にはこのメッ

セージは送られません。しかしながら、この公開鍵が署名と一体であるということは、すべての通信でこれを使いまわしているということであり、安全性に疑問が残ります。将来的に公開鍵とセットである秘密鍵が破られた際に、後から秘匿性が失われる恐れがあります。そのため、鍵を使い捨てにして**前方秘匿性**（forward secrecy）を確保することができる、DHE 等のアルゴリズムが推奨されています。この使い捨ての鍵を証明書とは別に送信する際に ServerKeyExchange は使われます。

最後に **ServerHelloDone** を送ることで、返事を終わらせます。

3 月 31 日 午後 10 時 26 分 ServerHello, Certificate, ServerKeyExchange 受信

もうまもなく新年度が始まろうとするときに、サーバーちゃんから証明書などが届きます。

サーバーちゃんから届いた情報を元に証明書の検証を行い、通信相手が間違いなくサーバーちゃん自身であることを確認します。そして、ここで初めてサーバーの公開鍵が手に入ります。公開鍵暗号において、暗号化に必要なのは相手の公開鍵なので、裏を返せばここまでの通信内容はすべて暗号化されていない平文の状態で行われていたということです。この暗号化されていないハンドシェイク部分は度々脅威にさらされてきました。最近で言えば OpenSSL の CVE-2015-0204(FREAK) という脆弱性は、このハンドシェイク時に中間者攻撃により脆弱な暗号化方式に切り替えさせられ、通信の機密性を脅かすというものでした。

その後、サーバーに送るための公開鍵を用意します。公開鍵を生成するにあたり、暗号論的な安全性を持った乱数を用いて秘密鍵を生成し、指定された CipherSuite に従って公開鍵を生成し、次のメッセージに載せます。そして、自身はその自分の秘密鍵と受け取った公開鍵から premaster secret を求めます。(Diffie-Hellman 等) これにより master secret をクライアント側で用意できるようになります。master secret を用意したら premaster secret はすぐに抹消します。もしこの 2 人のお便りを盗み見してる人に、premaster secret が漏れてしまえば、そこから master secret が計算できるので、あっという間に秘密が破られてしまうからです。

手紙に含まれてる情報をもとに、クライアントちゃんは master secret を生成しました。長い長いやり取りのあとに手に入れた、初めての 2 人だけの秘密です。クライアントちゃんにとってそれは単なる 48 バイトのデータの羅列ではなく、いつでも蕎麦にサーバーちゃんの面影を感じることができる、大切な大切なデータです。

4 月 1 日 午前 9 時 39 分 ClientKeyExchange, ChangeCipherSpec, Finished 送信

翌朝、クライアントちゃんはサーバーちゃんとの秘密のやり取りに初めて取りかかります。エイプリルフールですが、クライアントちゃんはサーバーちゃんに嘘なんてついたりしません。第一、この手紙が相手に届く頃にはエイプリルフールなんて忘れられています。きっと。

ClientKeyExchange では、自分が使用する公開鍵などを送ります。RSA の場合には premaster secret を暗号化して送りますが、通常良く使われる Diffie-Hellman やその派生では、その公開鍵を交換することで premaster secret が計算可能です。

ChangeCipherSpec は次からハンドシェイクの結果に基づき、暗号化したデータを送るというメッセージであり、いよいよ次から暗号化したデータに突入します。

Finished は暗号化された最初のメッセージです。と言っても、データ本体を暗号化する CipherSuite で指定された、例えば AES のような暗号化ではなく、きちんとハンドシェイクが成功したかを検証するための verify_data を送ります。この verify_data は master secret とこれまでのハンドシェイクメッセージから擬似乱数関数を通すことで生成され、通信が改竄されていなければサーバーはこれを検証することができます。

master secret を手に入れたクライアントちゃんは、サーバーちゃんへの手紙を試しに暗号で書いてみました。ちゃんとサーバーちゃんは書いてあることを理解してくれるでしょうか？

4 月 9 日 午前 8 時 10 分 ClientKeyExchange, ChangeCipherSpec, Finished 受信

クライアントちゃんの公開鍵とともに、初めての秘密のお便りが届きました。もう、秘密のコネクションの確立は目前です！

サーバーは受け取った公開鍵と、自分が送った公開鍵に対応する秘密鍵から premaster secret を求めます。この値は鍵交換アルゴリズムの性質からクライアントと同じものになっているはずです。もしそうでなければお便りは輸送路のどこかで改竄されています。そして、そこから master secret を計算し、クライアントと同様に premaster secret は抹消します。

その後、暗号化された Finished の中身を TLS で定められた手法でサーバーも試しに生成してみて、送られてきたものと一致していることを確認します。もし一致してなければその通信は危険な状態に晒されていることになります。fatal レベルのアラートを送って通信を切断します。

サーバーちゃんはクライアントちゃんからの暗号文を自慢の暗号力で読み解き、正しく復元できることを確認、誰にも知られることなく master secret が 2 人の間で共有されました。これで鍵交換成功です！

4 月 9 日 午後 3 時 10 分 NewSessionTicket, ChangeCipherSpec, Finished 送信

さて、サーバーちゃんもきちんと検証用の情報を送り、ハンドシェイクに成功したことをクライアントちゃんに伝えます。ここからはサーバーちゃんも手に入れた master secret で手紙を暗号化し、2 人だけにしかわからない秘密のやりとりが始まります。

NewSessionTicket では、セッション再開に必要な、サーバーが持っているべき情報を、クライアントに暗号化した状態で持ってもらうための情報です。実はハンドシェイクの冒頭の Hello で、お互いに TLS Session Resumption without Server-Side State を利用するかどうかの情報が交換できます。もしお互いにこれを利用できるのであれば、セッション再開情報をサーバーが持たずに、このチケットという形でクライアントに持ってもらい、再開時にサーバーがそれを受け取り復号化して、セッションを再開できます。が、今回は 1 回しか通信を行わないので、セッション再開の話はなしにしましょう。

ChangeCipherSpec と **Finished** はクライアントと同じような流れです。ちょっと擬似乱数関数に入れる情報が違うだけです。

4 月 19 日 午前 5 時 49 分 NewSessionTicket, ChangeCipherSpec, Finished 受信

TLS ハンドシェイク開始から 42 日が経過しました。早朝サーバーちゃんから届いたハンドシェイク最後の手紙によって、クライアントちゃんもハンドシェイクの成功を確認できる時が来ました。

セッション再開用のチケットを引き出しにしまいつつ、送られてきたデータを検証します。これで正常な通信の確立が確認できたらいよいよ暗号通信の始まりです。

TLS によってロスした 42 日という時間は、実時間に直すと 110 ミリ秒です。TLS ハンドシェイクは必ずサーバーとクライアントの間を 2 往復する必要があるため、この時間はクライアントとサーバーの距離によって大きく左右されます。HTTPS を導入する場合は、keep-alive によるコネクションの再利用や、CDN による通信距離の改善などを検討したいものです。

2.6 HTTP リクエストとレスポンス

4 月 19 日 午前 9 時 26 分 HTTP リクエスト送信

1 ヶ月半に渡るやりとりによって、サーバーちゃんとクライアントちゃんとの間で暗号通信を行う準備が整いました。もはや 2 人の世界を妨げるものは何もありません。これで少々神経過敏なクライアントちゃんも安心してサーバーちゃんに質問を投げかけることができます。

HTTP は、原則としてクライアントがサーバーに対してリクエストを投げて、サーバーがそれに対する応答を返すという、先程までの TCP ハンドシェイクや TLS と比べてシンプルな仕組みで動作します。ネットワークの中でもかなり高レイヤーなプロトコルなので、リクエストの内容やヘッダーはプレーンテキストで表現され、人間が見てそのまま理解しやすいようになっています。

今回はクライアントアプリケーションとして cURL を用いたので、HTTP バージョン 1.1 のシンプルな GET リクエストでパケットを送信しました。

クライアントちゃんは勇気を振り絞って、年が明けてからずっと聞きたかったことを質問しました。その内容はヘッダも含めて 4 行、文字数に直すと 99 バイトですが、TLS で暗号化することによって 130 バイト、イーサネットフレーム全体では 184 バイトと、実際に転送される情報としては倍近くになりました。ハガキは裏面しか使えませんからね。

4 月 28 日 午前 9 時 27 分 HTTP リクエスト受信

クライアントちゃんから暗号化された秘密の手紙が届きました。サーバーちゃん側から見ても、最初に SYN パケットを受け取ってから 2 ヶ月という時間が経過しています。その間、Web サーバーとしての諸々の雑務をこなしながら、クライアントちゃんからの手紙を待ち続けていました。

HTTP リクエストを受け取ったサーバーは、その内容を元にリクエストの解釈を行い、文脈に応じて何らかの適切な応答を返します。今回の測定ではデフォルトの Apache へのリクエストだったので、GET リクエストはシンプルにファイルの内容を取得するという処理となります。

手紙の内容を見て、さっそくサーバーちゃんは部屋の中 (=ファイルシステム) から目的のデータを探し始めました。サーバーちゃんはファイルサーバーなので、探しものはお手のものです。

4 月 28 日 午後 9 時 45 分 HTTP レスポンスの最初のパケット 送信

お手のものと言いながら、なんだかんだ言って半日仕事になってしまいましたが、ともあれサーバーちゃんはクライアントちゃんが尋ねてきた質問に答えるためのデータを見つけ出しました。

サーバーはクライアントからのリクエストを元に処理を行い、HTTP レスポンスを返します。これは**ステータス行**（status line）と **HTTP ヘッダー**と**メッセージ本体**で構成されます。

ステータス行には、レスポンスの状態を表す **HTTP ステータスコード**などが記されています。今回はリクエストされたリソースが正常に見つかったので、OK を意味するステータスコード 200 を返します。

HTTP ヘッダーでは、その他のレスポンスの付加的なメタ情報などを表現します。今回のレスポンスには、処理日時 (Date)、サーバー情報 (Server)、キャッシュ関連の情報 (Last-Modified, ETag)、本文のメタ情報 (Content-Length, Content-Type) などが含まれています。

そして、通信の主目的であるメッセージ本体がその後に続きます。今回転送するデータは 100KB と、とても TCP セグメントや IP パケットやイーサネットフレームに収まる大きさではないので、必然的に複数のセグメントに分割されて転送されます。このセグメントの大きさは TCP コネクション時の SYN パケットに指定された **MSS**（Maximum Segment Size）オプションによって決定されます。

今回の通信経路はおそらくイーサネットによって断片化することなく最後まで伝達できる経路であり、**経路 MTU**（Path Maximum Transmission Unit）はイーサネットフレームの最大サイズである 1500 バイト、実際に転送されるデータセグメントはここから IP ヘッダ 20 バイトと TCP ヘッダ 20 バイトを除いた 1460 バイトとなるため、一般的に MSS は 1460 バイトに設定するのが標準的です。PPPoE などではこれより若干小さな値になるので、断片化を防ぐためには個別に設定が必要です。

今回サーバーが送信したデータは 100KB(102400 バイト) のメッセージ本文に 275 バイトのヘッダーを付与した HTTP レスポンスなので、これを TCP セグメントで送出するには (102400 + 273) / 1460 = 70.3... より最低 71 個のパケットが必要となります。測定結果を見ると、送出したパケットはきっちり 71 個の TCP セグメントに分けられていました。

そういうわけで、サーバーちゃんはクライアントちゃんの質問に答えるための 71 枚の手紙を用意しました。これを一気にポストに投函すると重量オーバーで郵便局員に怒られてしまうので、まずは 1 枚目から、順番にポストに入れていきます。

まあ、往復する回数が増えるぶんかえって郵便局員の負担は増えるんですが、それはそれ、これはこれです。

5 月 11 日 午後 2 時 4 分 HTTP レスポンスの最初のパケット 受信

ゴールデンウィークが終わり、7 月中旬の海の日まで祝日のない地獄の 2 ヶ月間が始まるころ、ようやくサーバーちゃんからメッセージ本文を含む最初の手紙が届きます。

ゴールデンウィークではしゃぎすぎてグダっていたクライアントちゃんも、3 週間ぶりのサーバーちゃんからの手紙で生き返りました。今回の通信で用いる暗号は、鍵長 256 ビット、ブロック長 128 ビットの AES256 なので、1 つのパケットに平均 11 個強のブロックが含まれているこ

とになります。

実際には暗号利用モード[*16]によってオーバーヘッドが生じるため、必ずしも送られてきたパケットをすぐに利用できるとは限らないのですが、少なくともこの最初のパケットの段階で、クライアントちゃんがメッセージ本文の最初の部分を読み始めることができることは間違いないでしょう。

クライアントちゃんは、待ちに待った手紙を解読しながら喜んで読み始めました。通信開始してから 130 日も経っていますが、クライアントちゃんが年明けと同時に感じたサーバーちゃんへの思いは少しも色褪せていません。物理的には 1300km 離れているサーバーちゃんの存在も、今ではとっても近くに感じます。

2.7 TLS 終了アラートと TCP コネクション切断

9 月 13 日 午前 2 時 51 分 HTTP レスポンスの最後のパケット 送信

お別れが近づいてきました。夏の間ずっと HTTP レスポンスの手紙を送り続けてきたサーバーちゃんですが、伝えるべき話もだんだんと減っていき、残暑険しい 9 月の中旬に、ついに 71 通目の最後の手紙をポストに投函しました。

今回の通信ではデータを送りきってしまえば TCP コネクションはもう必要ないため、名残惜しいですが、サーバーちゃんはデータを送る最後の手紙の TCP ヘッダに FIN フラグを立てて、クライアントちゃんにメッセージの終わりを告げます。

この夏の間、サーバーちゃんは平均して 47 時間間隔、つまりおよそ 1 日おきに手紙を送ったことになります。実際にはネットワークの状況によってかなりムラが発生するのですが、今回は実時間換算で 5.4 ミリ秒がサーバー側の平均パケット送出間隔となりました。

この間、データを送出すると同時にクライアント側からの ACK セグメントを適宜受け取り、クライアントがデータをどこまで正常に受信できているかをずっと監視し続けます。もし一定時間内に ACK 応答が返って来ない場合はデータ再送などの処理を行うのですが、今回の計測ではデータ再送は一度も発生しませんでした。郵便局員が有能で助かります。

ちなみに、この最後の FIN パケットを送信した段階で、クライアント側から平均 60KB 程度の ACK 応答を受け取っています。よって残りの 40KB はネットワーク経路上のどこかでバッファリングされていることになります。

10 月 30 日 午前 2 時 29 分 HTTP レスポンスの最後のパケット 受信

秋も終わりに近づいてきました。渋谷が騒がしくなるハロウィンの前日、サーバーちゃんからの手紙が届きます。

サーバー側とクライアント側のスループットが異なるため、最後のパケットはサーバーが送信してからクライアントで受信するまでにだいぶ時間がかかります。経路上でのバッファリングにも限界があるため、TCP のヘッダには、送信側でパケットの勢いを調節するための**ウィンドウサイズ**という値があるのですが、今回はこれがゼロになることはなく、フロー制御による送出停止は発生しませんでした。

[*16] 今回の場合は **GCM**（Galois/Counter Mode）。

クライアントちゃんがサーバーちゃんからの 71 通目のお便りを開封すると、FIN フラグが立っていました。残念ですが、クライアントちゃんはサーバーちゃんにお別れを告げなければなりません。

10 月 30 日 午前 4 時 34 分 TLS 終了アラート 送信

FIN パケットを受け取ったクライアントは、まず TCP コネクションを切断する前に、TLS の切断を示す、closure alert をサーバーに送信します。

この直後に送信される TCP の切断パケットは、当然暗号化されていないので、中間攻撃者が簡単に偽造することができます。なので、攻撃者は任意のタイミングで通信を打ち切り、通信を妨害する **truncation attack** が可能となります。

これによってログアウトなどの操作を堰き止め、セッションを悪用されるなどの被害を軽減するために、TLS ではどのタイミングで暗号通信を終了するのかという情報を暗号化された状態で共有しなくてはなりません。それを行うのが、この closure alert です。

クライアントちゃんはこの手順を正しく踏み、文通を終了する前に暗号通信の最後の手紙を送ります。半年間使い続けた master secret も、他の人にバレたら大変なのでこの段階で破棄します。なんだかちょっと寂しいですね。

10 月 30 日 午前 6 時 4 分 TCP 切断 送信

暗号通信の終了通知に続いて、クライアントは今度は TCP の切断を自分からも伝えます。手順はサーバーと同じで、FIN フラグを立てたパケットをこちらからも送ります。

未練は少しありますが、サーバーちゃんから FIN パケットが届いている以上、これ以上手紙を送ってもサーバーちゃんから返事を受け取ることはできません。クライアントちゃんはサーバーちゃんへ手紙のやり取りを終了するための最後の手紙を送りました。

11 月 14 日 午後 2 時 6 分 TLS 終了アラート 受信

冷ややかな秋風が身に沁みる頃、クライアントちゃんから closure alert を受信したサーバーちゃんは、クライアントちゃんと同じく、各種暗号に使った情報をそっと破棄します。もう、暗号のお便りが届くことはないのです。

11 月 15 日 午前 1 時 42 分 TCP 切断 受信

立て続けにクライアントちゃんから TCP の最後の切断メッセージが届きます。ACK フラグと FIN フラグが立てられた、ヘッダーだけの味気ないパケットですが、半年以上クライアントちゃんと通信してきたサーバーちゃんは、クライアントちゃんがどんな気持ちでこのパケットを送ったのかが手に取るように分かります。

11 月 15 日 午前 1 時 50 分 TCP 切断に対する応答 送信

先だって届いた手紙を受けて、サーバーちゃんはいよいよこの通信の最後の手紙を送ります。またクライアントちゃんと文通したいという万感の思いを込めて、指定された FIN パケットに対する ACK レスポンスを返します。

11 月 30 日 午後 9 時 27 分 TCP 切断に対する応答 受信

さて、ついにサーバーちゃんからの最後の手紙が届きました。クライアントちゃんが送った FIN に対する最後の ACK を受け取り、すべてのやり取りは終焉を迎えます。

1 月 1 日、新しい年が始まると同時に ARP リクエストから始まったこの通信は、次の年を目前に迎えた 334 日目[*17]に、その全過程を終了しました。これは実時間に換算して 914.8 ミリ秒に相当します。

今回使用したインターネット回線がだいぶ貧弱だったためか、実際のシチュエーションで想定される時間よりもだいぶ時間がかかっていますが、それでも、ネットワークの世界では、わずか 1 秒の間にこれだけのことが起こっています。皆さんもたまにはパケットキャプチャを行ってネットワークの深い世界に潜ってみてはいかがでしょうか。

2.8 最後に

「『宇宙の歴史を 1 年で』とか、『世界の人口を 100 人で』みたいな企画はあっても、『コンピューターのスピードを人間スケールで』っていうのは聞いたことがないな」というただの思いつきと、「自分はネットワークのことを全然知らないので、これを機にちゃんと勉強したい」という軽い気持ちで書き始めた記事ですが、実際に作ってみると想定よりも遥かに長い記事になってしまいました。しかしこれでもネットワークの現象のあらましを述べただけであり、具体的なパケットの内容にはほとんど触れてないばかりか、イーサネットや無線 LAN などの物理層の事象や大規模ネットワークのルーティング、さらに HTTP 以外の上位レイヤーの通信については全く触れてないことを考えると、ネットワークの世界というのは実に広大であることを思い知らされます。

僕はふだん Web プログラマーのはしくれとして毎日 HTTP 通信を扱っていますが、今回、その下にある TCP/IP の世界のことを多少でも知ることができたのは大きな学びになりました。やはりプログラマーとして、常に視野を広く持たないといけないということを実感しますね。

皆さまも、よきプログラマーライフを。

[*17] 阪神は関係ない。