インフラ屋さん、今昔物語

2019年 はてな サマーインターン

自己紹介 id:nabeop

- サービス•システム開発本部 システムプラットフォーム部
  ~~Web オペレーションエンジニア~~ SRE
  渡辺 道和
- 2018年3月 入社
- L3 からサーバサイドをフラフラしているインフラ屋さん

目的

- 一般的な経緯を交え、どのようにインフラが変遷していったのか知ってもらう
- その中でインフラ屋さんを始めとするエンジニアにどのような変化があったのか知ってもらう
- 課題も無いので、気軽に聞いてください
  - 質問もお気軽に

インフラ is 何

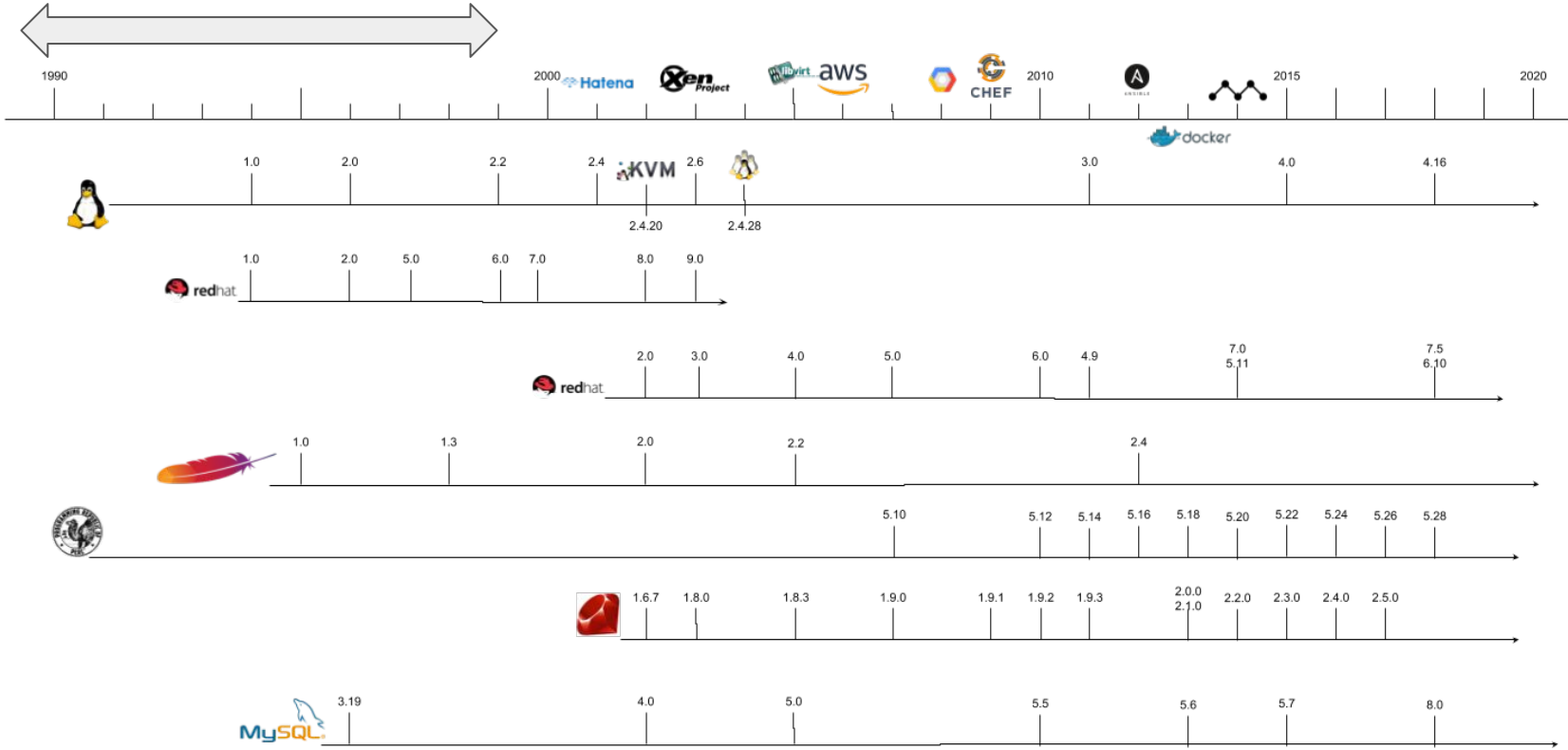
- アプリケーションエンジニア
  - 自身が開発に携わっているアプリケーションやミドルウェアから　OS のユーザランドくらいまでが見えていれば事足りることが多い
- オペレーションエンジニア
  - OS からネットワークレイヤ (IP層、場合によっては物理層も) までみる必要がある
  - ハードウェア側の知識もある程度必要

- 最近はこの辺りの前提が崩れて、アプリケーションエンジニアとオペレーションエンジニアが見ている領域がオーバーラップしてきている

お品書き

1. インフラ構成の変遷
    a. 黎明期
    b. オンプレ全盛期
    c. クラウドシフト期
    d. サーバレスアーキテクチャの夜明け
2. イマドキなインフラとは?
3. まとめ

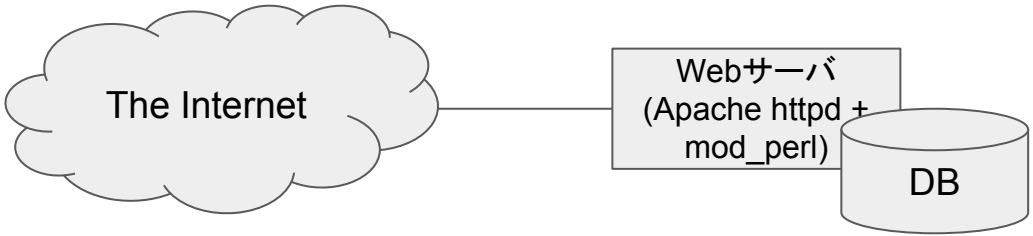
黎明期


1990
2000
Hatena
Xen Project
aws
CHEF
2010
2015
2020
docker
1.0
2.0
2.2
2.4
KVM
2.6
3.0
4.0
4.16
2.4.20
2.4.28
redhat
1.0
2.0
5.0
6.0
7.0
8.0
9.0
redhat
2.0
3.0
4.0
5.0
6.0
4.9
7.0
5.11
7.5
6.10
1.0
1.3
2.0
2.2
2.4
5.10
5.12
5.14
5.16
5.18
5.20
5.22
5.24
5.26
5.28
1.6.7
1.8.0
1.8.3
1.9.0
1.9.1
1.9.2
1.9.3
2.0.0
2.1.0
2.2.0
2.3.0
2.4.0
2.5.0
MySQL
3.19
4.0
5.0
5.5
5.6
5.7
8.0

- 素朴な構成
  - web/db 同居とかあたりまえ
  - イケている運用ツールもまだない時代
- 全て自組織で運用する必要があった
  - ネットワーク、ラック、サーバ、全て
  - イケている運用ツールもない時代
    - 自分でツールを開発
  - サーバ、ネットワーク機器を愛でるように扱う時代
    - 各自の想いを込めて、運用する
    - 『ペット』のように愛でる

素朴な構成

- web サーバ + アプリケーション + ストレージ (DB) の構成

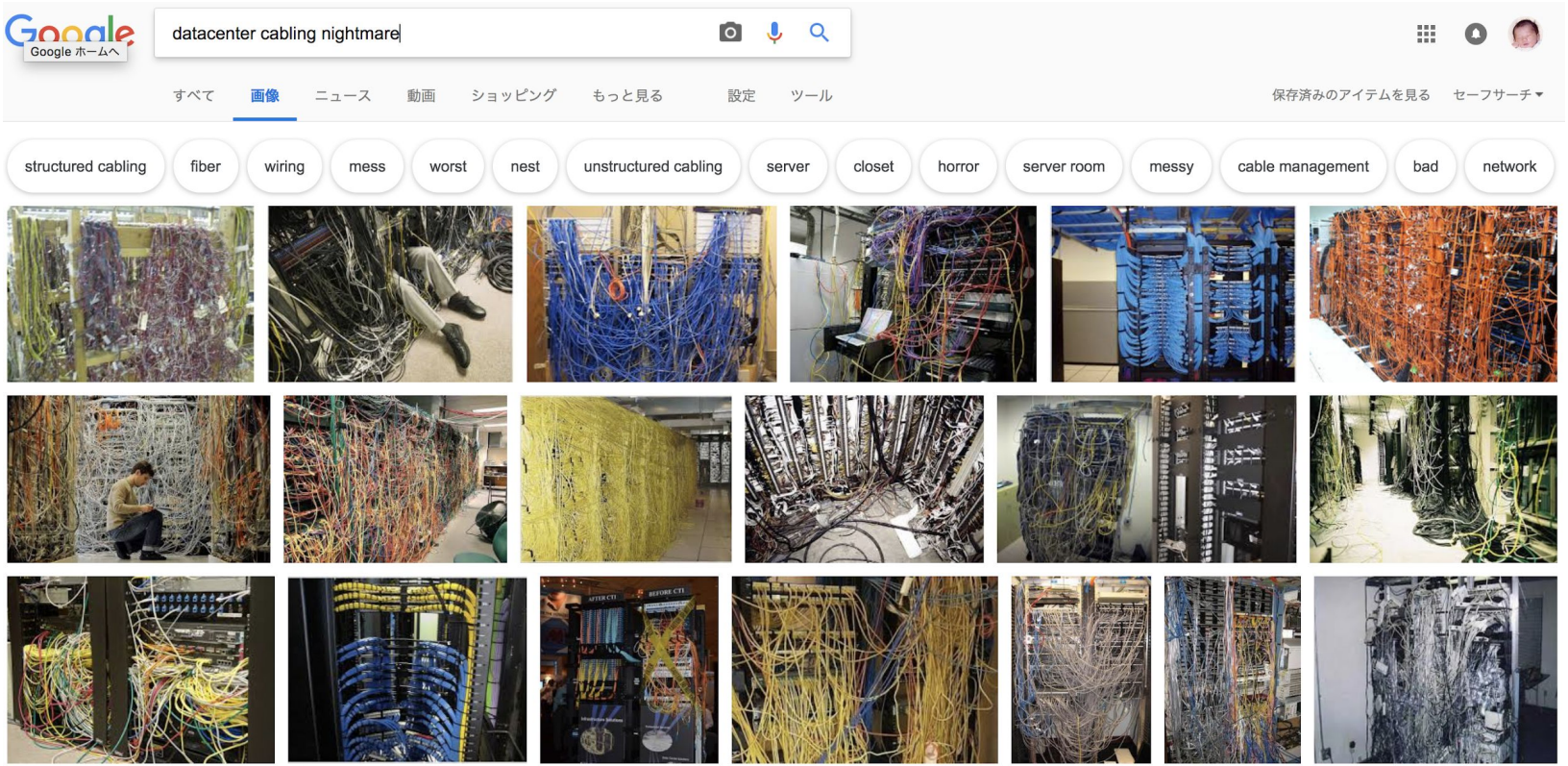


- 実際にここまで素朴な構成で商用サービスを運用はしない
- 構成の規模や構成するプロダクトは変わっても基本は変わっていない
- 現在は負荷分散装置(LB)やプロキシなどが構成要素に加わっているくらい

ネットワーク、ラック、サーバ、全てを管理する

- OS より下のレイヤを管理/運用する必要がある
- ラックの管理
  - ラックにきている電源容量を超えないように機器を設置
  - ラックの耐荷重を超えないように機器を設置
- スイッチやルータなどのネットワーク機器の管理
  - 結線状況
  - ルータの経路情報やラックにきているネットワークとのつなぎこみ
  - ネットワークデザインも必要
- 機器の保守管理
  - 機器のベンダー保守期限の管理

ケーブリングの乱れは心の乱れ



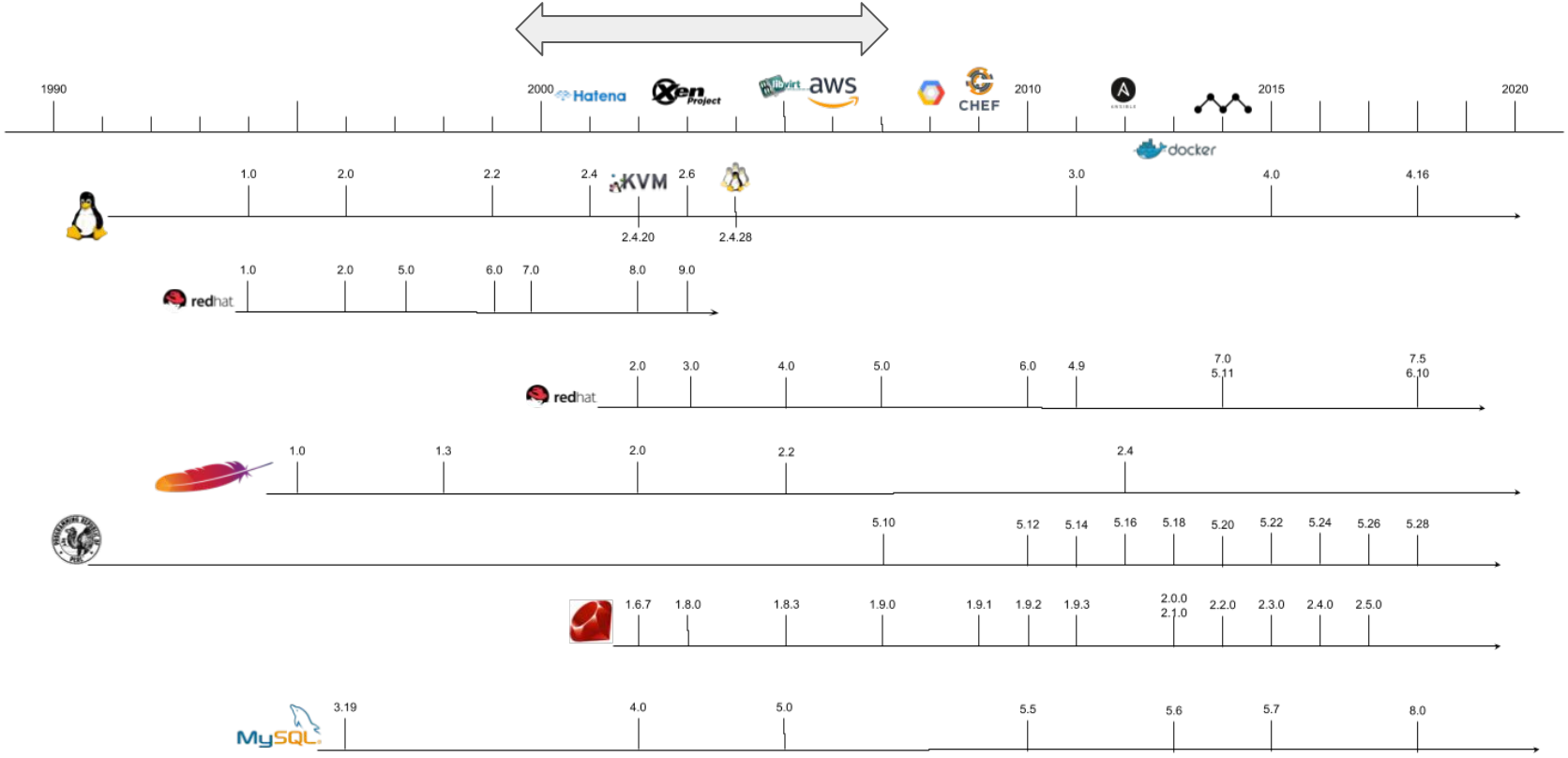
なぜ、想いを込められていたか?

- 台数が少ないので、各自の思い思いの名前をつけることができた
  - はてなの場合は『峠』の名前
- 冗長化していない
- 落としても上がってくる保証がない (わりに、時々機嫌がわるくなる)
- デプロイ方式が未熟だったため、新しく作れる保証がない
  - chef / ansible が無い時代
  - tar 玉を持って行って展開、shellscript でバーンとやる (で、バーンと死ぬ)

各自の想いを込めた運用の結果

先日、uptime 3849days なホストを落としたときの様子

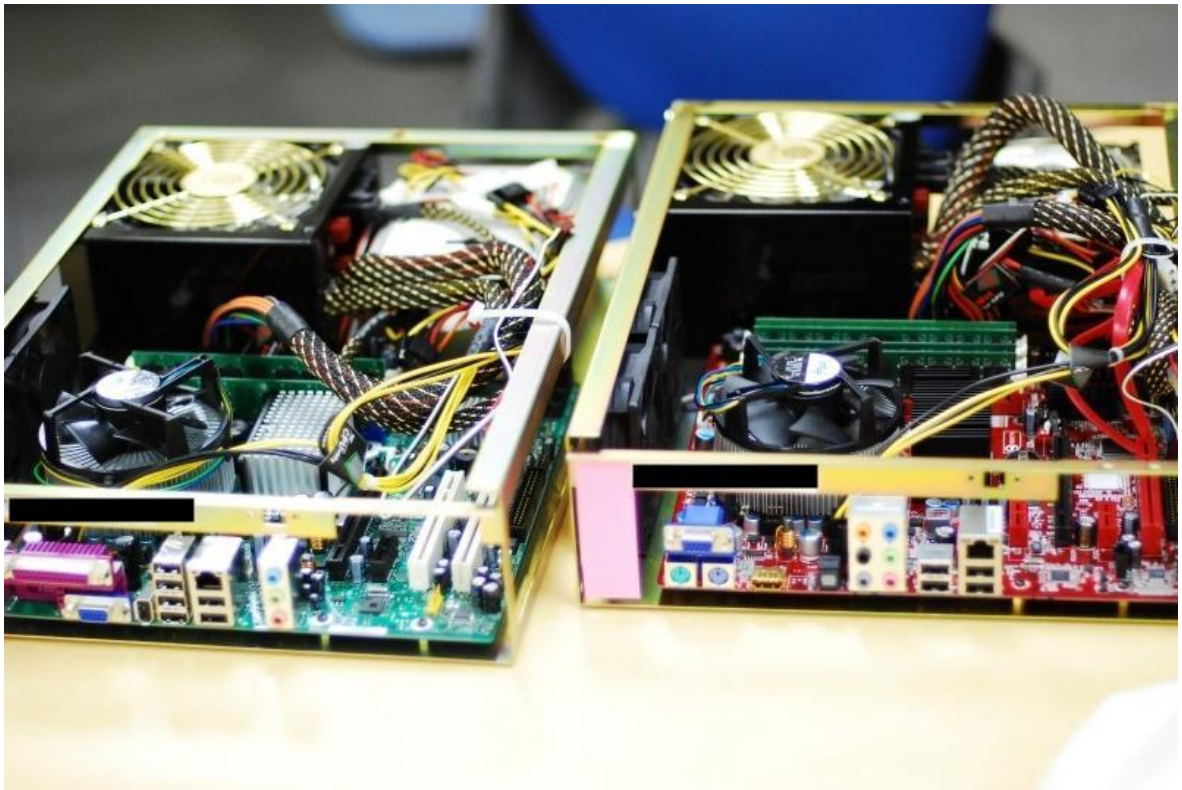
オンプレ全盛期


1990
2000
Hatena
Xen Project
aws
CHEF
2010
2015
2020
docker
1.0
2.0
2.2
2.4
KVM
2.6
3.0
4.0
4.16
2.4.20
2.4.28
redhat
1.0
2.0
5.0
6.0
7.0
8.0
9.0
redhat
2.0
3.0
4.0
5.0
6.0
4.9
7.0
5.11
7.5
6.10
1.0
1.3
2.0
2.2
2.4
5.10
5.12
5.14
5.16
5.18
5.20
5.22
5.24
5.26
5.28
1.6.7
1.8.0
1.8.3
1.9.0
1.9.1
1.9.2
1.9.3
2.0.0
2.1.0
2.2.0
2.3.0
2.4.0
2.5.0
MySQL
3.19
4.0
5.0
5.5
5.6
5.7
8.0

- オンプレミス (on-premiss) : 自前でサーバ、ネットワークを運用
  - 黎明期との違いは規模
    - インターネット上で toB 向けサービスなどが普通になってきた
    - インターネット自体の習熟が進んだ
  - ハードウェアまで含めたの選定、運用、保守を回す必要があった
    - サーバを発注して、利用可能になるまで下手をすると数ヶ月単位かかる世界
- vSphere / Xen / KVM などのサーバ仮想化技術によるサーバ集約
- 規模が大きくなったので、サーバ管理の重要性が増した
  - ホスト情報 -> 各自で作り込み
  - ホスト監視 -> nagios (2002年) / zabbix (2001年)
  - 効率的なデプロイ方法の模索 -> のちに puppet / chef / ansible などが生まれる素地ができてきた

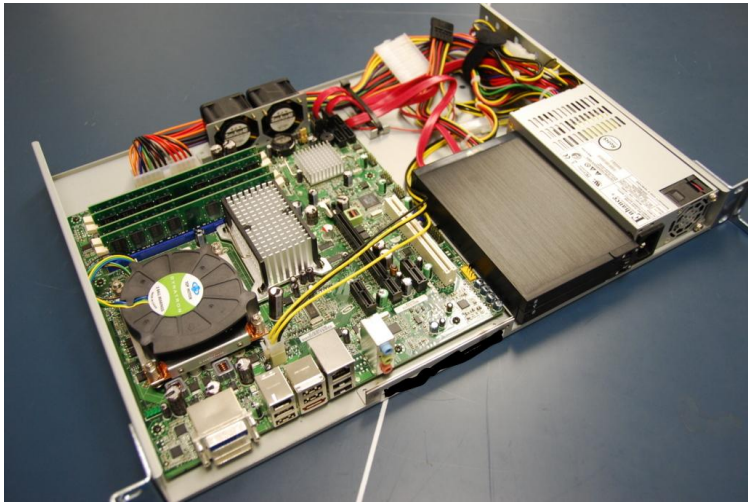
この頃から はてな のシステムの黎明期が始まる

自作サーバ (CMX-50)



- http://jkondo.hatenablog.com/entries/2003/09/28 に登場

自作サーバ (marqs-60)



- http://marqs.hatenablog.com/entry/20090622/1245670553 開発者によるエントリ
- CMX-50 の後継

最近の自作サーバ界隈

- Google や Facebook、Microsoft などの超大規模なサービサーで自作サーバ(というよりも自作 DC)の熱が高まっている
- https://www.opencompute.org/
  - Projects をみると、サーバ筐体だけではなく、DC 全体の仕様を作って、モノも作るということが見て取れる
  - 各 Project のリーダーを見ても、サービサーだけではなくハードウェアベンダーも入っている

ラックマウントされている普通のサーバ

- 写真は 1U サーバ
  - U は筐体の高さの単位
  - 一般的なラックでは 42U まである
- 他にも 2U / 4U などのサーバがある
  - 主にローカルストレージをどれだけ詰めるか

サーバの高性能化と仮想化技術の発達

- サーバスペックとアプリケーションが求める要求のアンバランスが生じるようになった
  - CPU の高速化と many core 化
  - メモリ の大容量化
  - ローカルストレージの大容量化
- 1つの筐体に複数のサーバ(VM)を起動させる仮想化技術が発達してきた
  - 集約率を上げることで、ハードウェアコストを下げる戦略が可能になった
  - ハードウェアトラブル時にマイグレーションによるダウンタイムの短縮
- 効率よくVMを配置するための技術やVMのスケールアップ技術が重要になった

ネットワークの仮想化

- サーバだけではなくネットワークの仮想化も進んでいる
  - SDN (Software Defined Network) の文脈
- 異なるネットワークに所属している VM 同士を同一のネットワークに所属しているように見せる
- 仮想化基盤が肥大化すると単一のネットワークに収容することが困難になる
  - VM 側としては同一のネットワークにいて欲しい
- サーバ仮想化とは時期がずれて、ここ数年で普及が進んだイメージ

構成の高度化と複雑化

- 負荷分散技術の発達
  - L3 ロードバランサ
  - L4/L7 ロードバランサ
  - DNS ラウンドロビン
  - アプリケーション側に負荷分散を組み込む
- ベストプラクティスの普及
  - proxy + アプリケーション + ストレージ
- 永続ストレージやキャッシュストレージ
  - MySQL / PostgreSQL などの RDB
  - memcached / Redis などのキャッシュ
  - 実用的なストレージクラスタ

負荷分散技術

- DNS ラウンドロビンを負荷分散技術に入れるかは微妙なところ
- L3 ?? L4 ?? L7 ??
  - 負荷分散対象がネットワークレイヤ (OSI参照モデル) のどの部分を対象としているか
    - L3 : IP レイヤ
    - L4 : TCP/UDP レイヤ
    - L7 : アプリケーションレイヤ (HTTPとか)
  - ロードバランサの選定にはどのレイヤが対象になっているかが重要になる
    - L3 を対象にしている LVS では HTTP リクエストに応じて負荷分散できない
    - 言い換えるとネットワーク通信のどのレイヤを終端しているかを意識する必要がある
- 代表的なロードバランサの種類
  - LVS
  - HAProxy

なぜ負荷分散技術が必要なのか

- 主に2つの目的がある
  - スケールアウトによる処理限界能力の向上
  - 冗長化による可用性の確保
- 負荷分散方式
  - ラウンドロビン
  - 平等性
  - 重み付け
- ヘルスチェック
  - 本当にトラフィックを向けて良いのか？
- トラフィックがコントロールできるのでメンテナンス性が向上する
- 最近のHTTP(S) 向け L7 ロードバランサではリバースプロキシと区別がつかなくなってきている傾向もある

ベストプラクティスの普及

- 3層構造アーキテクチャ
  - プロキシ + アプリケーション + 共有ストレージ
- 以前はプロキシとアプリケーションが同居していた
  - 処理のボトルネックの分離とスケールアウト時の戦略に幅を持たせる
- 接続方式の違い
  - 新規のネットワーク接続を作る時はネットワークと OSの両方の観点でコストフル
  - ネットワーク側：TCP ハンドシェイク
  - OS 側：ソケット作成時のメモリコピーなど

プロキシ

- HTTP リクエストをアプリケーションに中継する
- 頻繁に更新されるアプリケーションのデプロイを容易にする
- nginx や Apache httpd などが代表的
- クライアントからのリクエストとアプリケーションを分離
  - HTTP の keepalive 接続を利用することで、アプリケーション側への接続を効率よくする
  - 場合によっては悪意あるリクエストをプロキシ側で止めて、アプリケーションを守る　WAF 的なこともする
- ロードバランサ的な役割を担う場合もある
  - リクエスト内容によって割り当てるアプリケーションを変更する

アプリケーション

- はてな 的には backend と呼ばれている
- Perl, Go, Ruby など web サービスの機能を提供する
- 3層構造の中ではアプリケーションエンジニアの主戦場とも言える
    - 後段のデータベース層も重要だが、自家製のプログラムが動いているところ、という意味

ストレージ

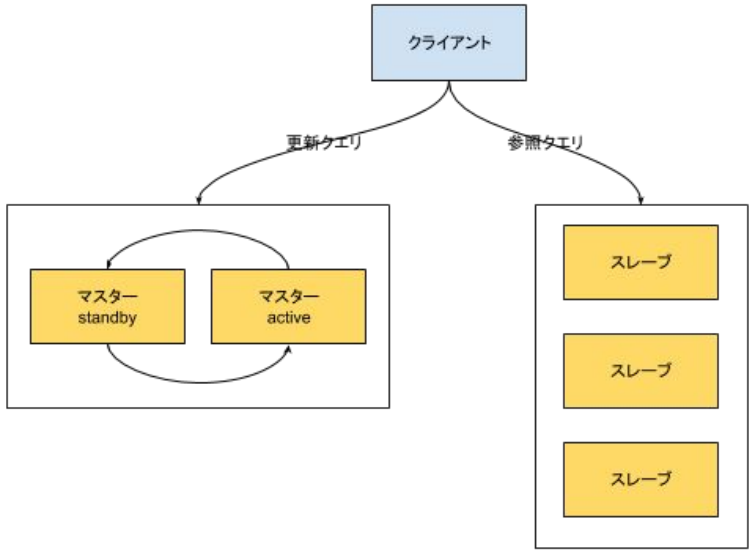
- 基本的にプロキシ/アプリケーションではデータの永続化はされていない
  - データの永続性や状態の有無は運用では大きな意味を持ってくる
- データを格納するための箱
  - ネットワークストレージ (NFS, SMB, GlusterFS, Ceph ….)
  - RDBMS : MySQL, PostgreSQL ...
  - KVS/NoSQL : memcached, Redis, mongodb ...
- 複数のアプリケーションインスタンスから1つのデータを参照する必要がある

ネットワークストレージ

- はてなではあまり使われていないので割愛
- ネットワークとストレージの両方の知識が必要とされるので、難易度が高い

RDBMSのクラスタ化

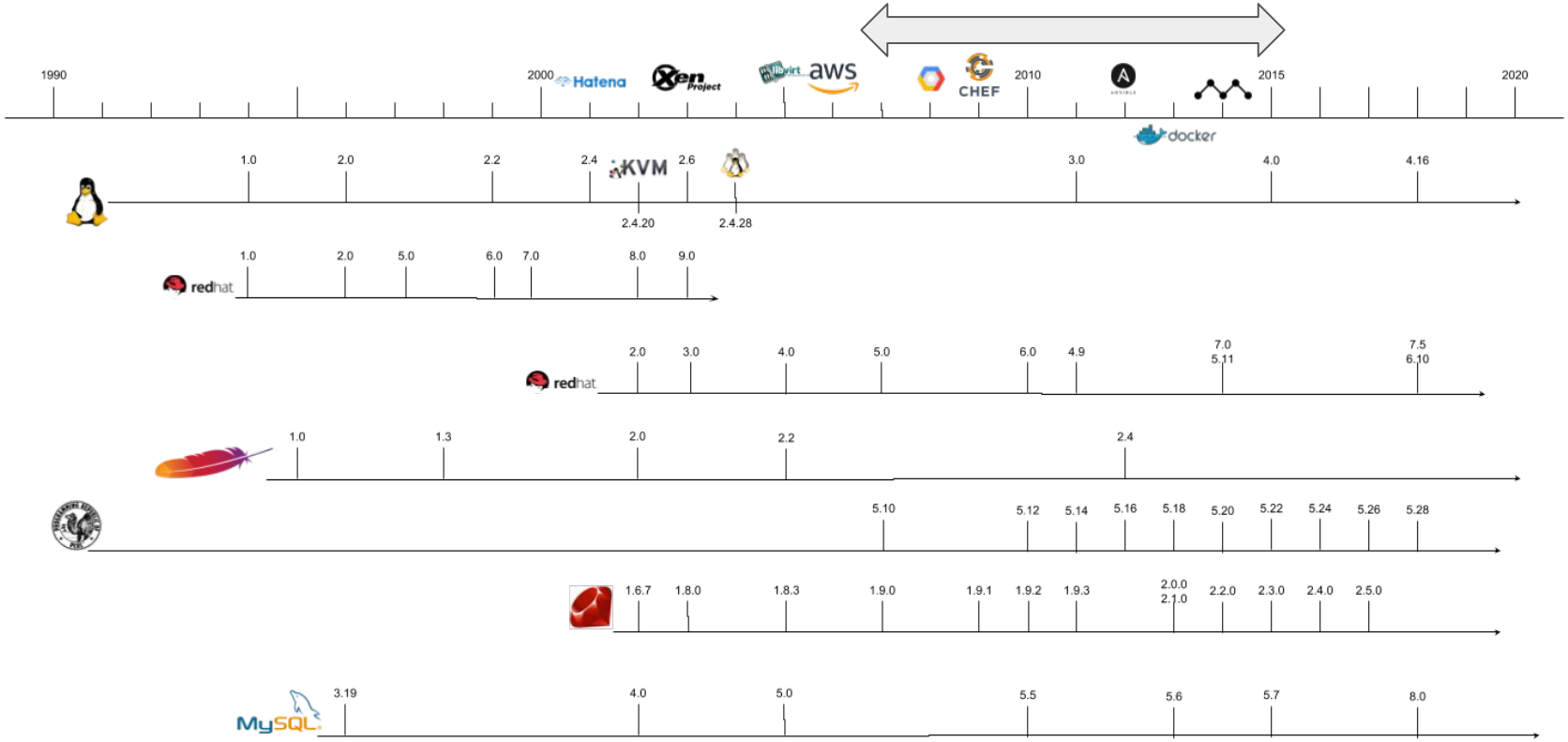
- データ is 大事
- パフォーマンスを稼ぐために役割を分けることもある



キャッシュストレージとしての KVS/NoSQL

- RDBMS よりも低レイテンシなストレージが欲しい場合
  - memcached, Redis, mongodb など
- データの永続性はこだわらない
  - セッション情報など
- 高速な memcached などをキャッシュとして使用するケースがある
  - まず、アプリケーションはキャッシュを見に行く
  - キャッシュになければストレージを見に行く
  - ストレージから取得したデータを利用し、キャッシュに取得結果も書き込む

クラウドシフト期


1990
2000
Hatena
Xen Project
aws
CHEF
2010
2015
2020
docker
1.0
2.0
2.2
2.4
KVM
2.6
3.0
4.0
4.16
2.4.20
2.4.28
redhat
1.0
2.0
5.0
6.0
7.0
8.0
9.0
redhat
2.0
3.0
4.0
5.0
6.0
4.9
7.0
5.11
7.5
6.10
1.0
1.3
2.0
2.2
2.4
5.10
5.12
5.14
5.16
5.18
5.20
5.22
5.24
5.26
5.28
1.6.7
1.8.0
1.8.3
1.9.0
1.9.1
1.9.2
1.9.3
2.0.0
2.1.0
2.2.0
2.3.0
2.4.0
2.5.0
MySQL
3.19
4.0
5.0
5.5
5.6
5.7
8.0

『ペット』から『家畜』へ

- サーバ仮想化技術の発展により AWS を始めとした『クラウドサービス』が出てきた
- 設備の調達コスト、運用コストに疲弊していたオンプレ組が徐々にクラウドサービスを使い始める
- このあたりからインフラへの向き合い方が変わってきた

クラウドサービスの特徴

- XaaS 類
  - IaaS (Infrastructure as a Service) : 基盤部分
  - PaaS (Platform as a Service) : IaaS 上で稼働している MySQL、Redis などのミドルウェア
  - SaaS (Software as a Service)
- インフラリソースを時間単位などで購入
- フルマネージドサービスで利用者や開発者が欲しい部分だけに注力できる
- 複数のサービスを API によってつなぎ合わせることでサービスを構築する
- インフラをコードによって表現できる

AWS サービスとオンプレ時代の比較

- サーバインスタンス → AWS EC2 など
- ストレージ → AWS S3 など
- ロードバランサ → AWS ALB/NLB など
- RDBMS → AWS Aurora, RDS for MySQL, RDS for PostgreSQL
- キャッシュ → AWS ElasticCache
- 上記以外にも AWS で提供されているサービスは多岐にわたり、それぞれを組み合わせて活用する

API 操作によるインフラ自動化

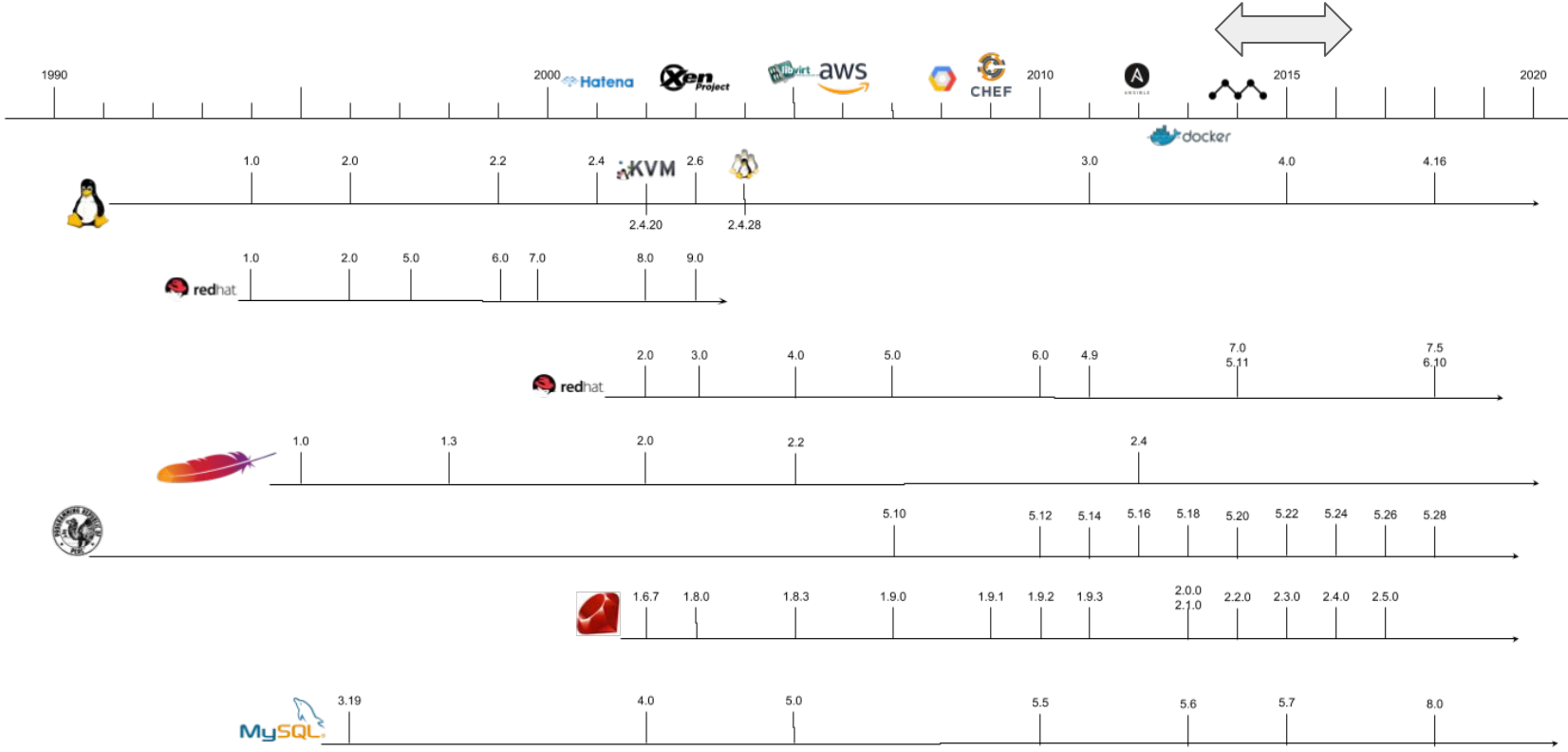
- インフラの各種操作を自動化できるためのパーツが揃っている
  - サーバインスタンスの作成、起動、終了が API 経由で可能
  - サーバインスタンスの構成情報などが API 経由で取得可能
  - 各種 webhook の呼び出しが可能

インフラ自体をコード化する

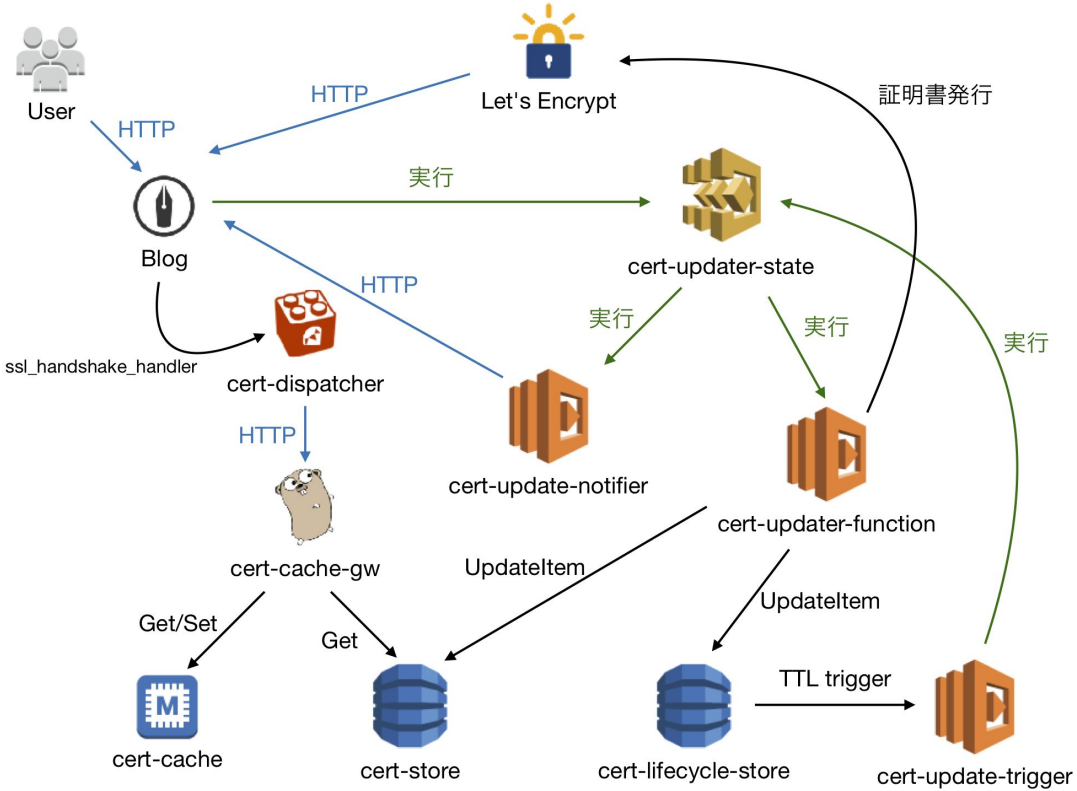
- AWS CloudFromation や Terraform など、インフラの状態を定義して、管理する
- Netfix 社が公開している Spinnaker など、API を利用することでインフラの状態変化を CD パイプラインに組み込む動きもある
    - API を叩くことでクラウドベンダー間の差異を吸収して 1つのインフラとして扱える

- インフラエンジニアとアプリケーションエンジニアの境界が曖昧になってきた
- むしろ、今までインフラエンジニアの領域だった部分がフルマネージドサービスで提供可能だったり、アプリケーションエンジニアが安心して触れるようになった

サーバレスアーキテクチャの夜明け


1990
2000
Hatena
Xen Project
aws
CHEF
2010
2015
2020
docker
1.0
2.0
2.2
2.4
KVM
2.6
3.0
4.0
4.16
2.4.20
2.4.28
redhat
1.0
2.0
5.0
6.0
7.0
8.0
9.0
redhat
2.0
3.0
4.0
5.0
6.0
4.9
7.0
5.11
7.5
6.10
1.0
1.3
2.0
2.2
2.4
5.10
5.12
5.14
5.16
5.18
5.20
5.22
5.24
5.26
5.28
1.6.7
1.8.0
1.8.3
1.9.0
1.9.1
1.9.2
1.9.3
2.0.0
2.1.0
2.2.0
2.3.0
2.4.0
2.5.0
MySQL
3.19
4.0
5.0
5.5
5.6
5.7
8.0

- Docker に代表されるコンテナ技術の成功と普及
  - 今までも lxc / OpenVZ などコンテナ技術はあった
  - サーバ仮想化技術を 1歩進めて namespace 単位での分離を実施
- フルマネージドサービスのさらなる充実
  - Step Functions や AWS Lambda など FaaS サービスが登場し、アプリケーションの処理をフルマネージドサービスに載せることが可能になった
  - Fargate / ECS / EKS などコンテナ基盤のフルマネージドサービス
- 今までのインフラ層を意識しなくてもサービス構築が可能になってきた



出典

[Hatena Engineer Seminar #10
AWS ではてなブログの常時HTTPS配信をバーンとやる話]

イマドキなインフラとは?

- 基本的にフルマネージドサービスを使う
  - とくに運用が難しいステートフルなサービスはフルマネージドサービスを積極的に使う
- サーバにログインする必要を極力減らす
  - ステートレスな機能は何か問題があれば、落として作り直すくらいのノリ
  - API を使い倒すことでアプリケーションエンジニアが安心してインフラを触れる環境を構築する
- 監視について考え直す
  - もはや、1つ1つのサーバを丁寧に監視する必要性が薄れてきている
  - 同じ機能を提供するアプリケーションを一括りにして監視して、パフォーマスが落ちてきたら監視を鳴らすくらい
- デプロイ周りの自動化と高度化の促進

まとめ

- 時代の趨勢や技術の進歩に合わせて環境も目まぐるしく変化する
  - 大きな流れの中で個々の技術は洗練され、進化し続けていく
  - 今使っている技術がどのような経緯で世に出てきたのかを考えることは、技術の理解に大きく役立つはず
- 如何に進化し続けるインフラを利用してサービスを『素早く』『進化させ続ける』のか
  - 巨人の肩にただ乗っているだけではなく、巨人の使い方や新しい巨人の出現に敏感になって欲しい
- インフラエンジニア is dead?
  - そんなことはない
  - クラウドサービスの中身はオンプレ
  - クラウドサービスからの出戻り傾向もある

インターネットの歴史

- インターネット白書アーカイブ
- JPNIC によるインターネット歴史年表