# コンテナをたくさん詰め込んだシステムとランタイムの変化


- HIROAKI MIZUGUCHI
- INTERNET INITIATIVE JAPAN INC.

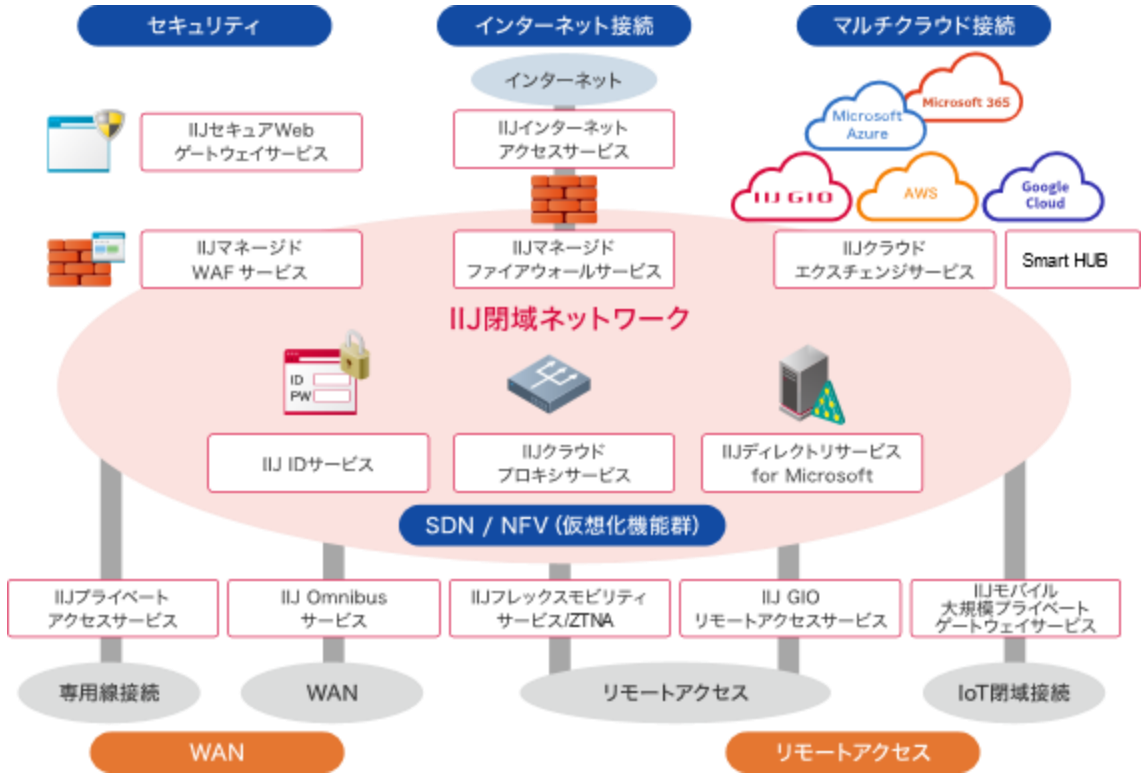

- CONTAINER RUNTIME MEETUP #6
- 2024-12-16

# 自己紹介

- 名前: 水口　弘明
  - X: @m_akihiro, Github: akihiro
- 所属: Internet Initiative Japan Inc.
- 仕事: ネットワーク監視システムの開発運用、サーバOS周りのコンサルティング

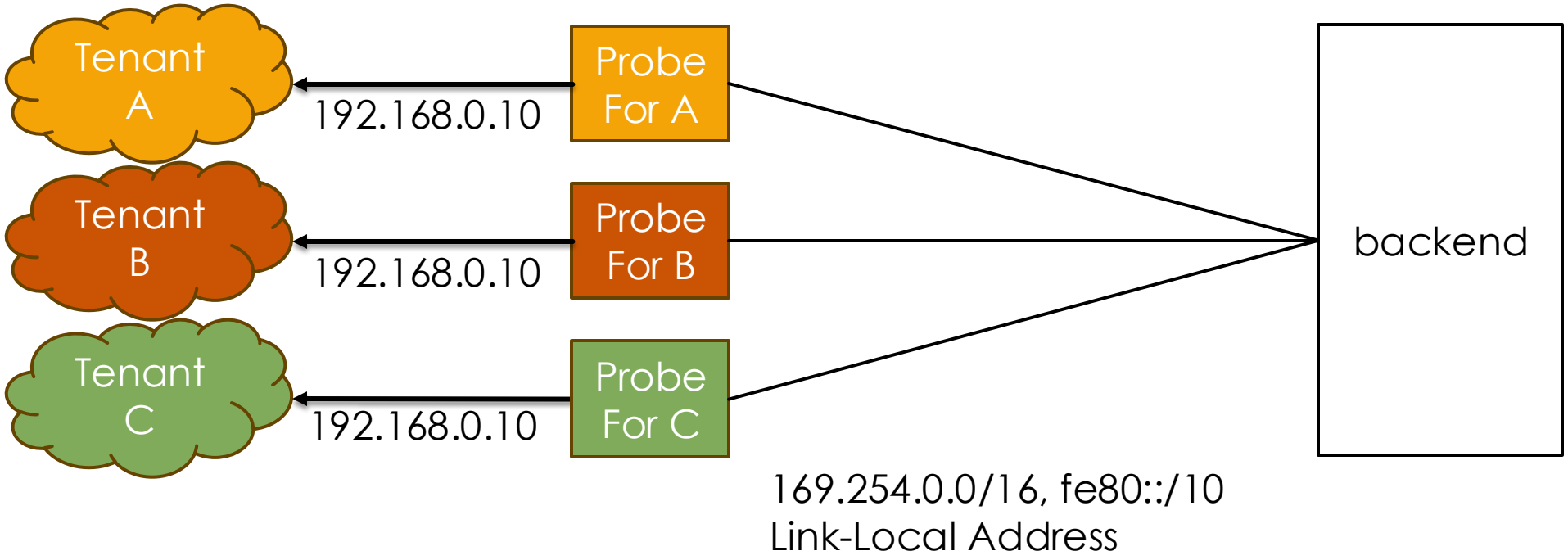
# IIJプライベートバックボーンサービス

- IIJのサービスや他社クラウドを相互接続できる閉域網を提供
- テナントごとに独立した閉域網を多数管理
- 2013年10月サービス開始



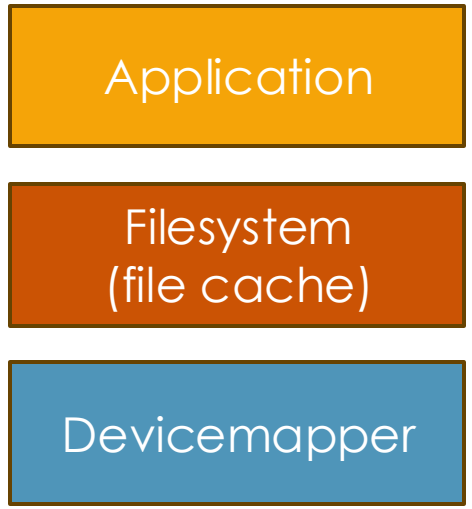
# なぜ多数のコンテナを動かすのか？

- テナント毎にアドレス空間は独立し、テナント間ではアドレスが重複するから



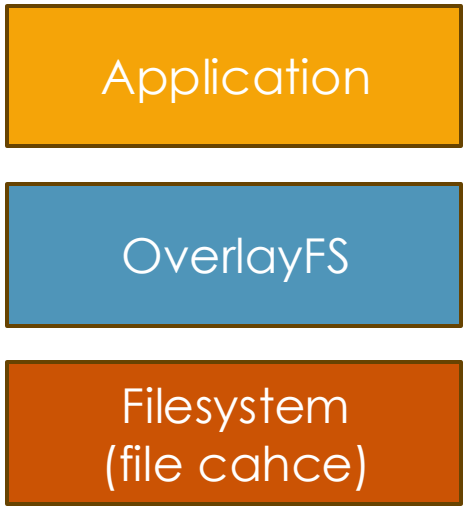
# 2014年 device mapperの導入

- Containerのlayered filesystemとしてdevice mapper実装を導入
- コンテナの収容に対してdevice mapperがボトルネック
  - ブロックデバイスレイヤーで実装
  - コンテナ毎にファイルキャッシュが独立
- 64GBのホストに250テナント/500コンテナを収容
  - NetNSを保持するpause+CNI機能を持つ内製したgolang製のコンテナ
  - 監視サービス用の内製したpython3のコンテナ
  - 50GBほどのメモリ消費で安定していた



# 2018年 overlayfsの導入

- Runtimeのlayered filesystemとしてoverlayfs実装を採用
  - 同一コンテナイメージならファイルキャッシュが共有される
- コンテナ内部プログラムの書き換え
  - NetNS保持コンテナとしてk8sのpauseに置き換え、CNI相当をホスト側へ
  - 監視プログラムをGo言語製に置き換え
- 96GBのホストに500テナント/1000コンテナを収容
  - メモリ消費は15GB程度で安定



# Container Runtimeのfootprint

- 2014年docker、2018年docker+containerdを採用
- コンテナの数に比例してContainer Runtimeのfootprintが問題視
- 500テナント/1000コンテナのOSのThread数
  - GoのRuntimeはCPU数程度のスレッドを作る。不足するともっと沢山作る
  - containerd: 1k threads
  - containerd-shim: total 10k threads
  - pauseコンテナ: 500 threads
  - アプリ本体のコンテナ(tini相当+本体): total 20k threads

# 2024年 daemonlessのpodman

- daemonlessのpodman+quadletを採用
- 管理用の常駐プロセス不要
- containerd-shim相当のconmonはsingle thread動作
- podmanは直接netnsを指定可能 → pauseコンテナ不要
- アプリケーションコンテナの設計見直し
  - Zombie reaperをGo製の内製ツールからtini(single thread動作)に変更
- 1ホスト当たり1000テナント/1000コンテナを収容

# 非k8s環境でのコンテナ管理

- Podman Quadletとsystemdの組み合わせが良い（個人の感想です）
  - QuadletはpodmanのSystemd-generatorとして実装
  - systemd.service likeな設定ファイルでコンテナ管理できる
- Podman quadletとansibleの組み合わせが良い（個人の感想です）
  - Ansibleとpodmanは共にredhatが開発している
  - Podman用のansible roleもメンテナンスされている
  - 沢山のコンテナの設定をjinjaテンプレートで管理できる
- NRIのような高度なリソース管理機能が要らないならおすすめ