

# REST と ROA

---

2008年2月12日

NTTコミュニケーションズ

先端技術セミナー ウェブテクノロジーシリーズ

**RICOH**

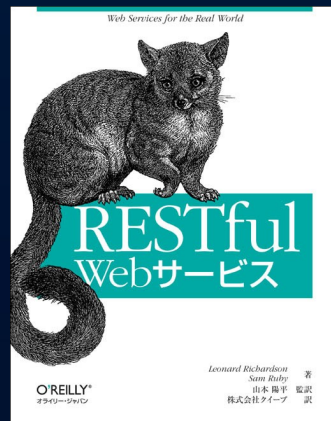
株式会社リコー ソフトウェア研究開発本部

山本陽平

<http://blogs.ricollab.jp/webtech/>

# 自己紹介

- 山本陽平
  - (株) リコー
  - ソフトウェア研究開発本部
  - ソリューション研究所
  - インテグレーションエンジニアリング研究センター
- XML guy
- 朝倉さんの1年後輩(NAIST)
- REST 関係でいろいろ執筆



---

RESTって何だろう

# REST って何だろう

---

- Representational State Transfer の略
- Roy Fielding の博士論文(の第5章)
- ソフトウェア工学の研究成果
- ネットワークシステムの「アーキテクチャスタイル」

---

# アーキテクチャスタイル

# アーキテクチャスタイル

---

- Architectural Style
- != Architecture
- デザインパターンと
- デザイン(設計)の関係

# デザインパターン

---

- クラス設計の方針(pattern/style)
- クラスを役割で X と Y と Z に分ける
- 「分け方」= デザインパターン
- 分けた結果のクラス = デザイン(設計)

# アーキテクチャスタイル

---

- クライアントサーバ
- MVC
- パイプ&フィルタ
  
- 設計者はこれらのスタイルを使って  
アーキテクチャを設計する

# REST は複合アーキテクチャスタイル

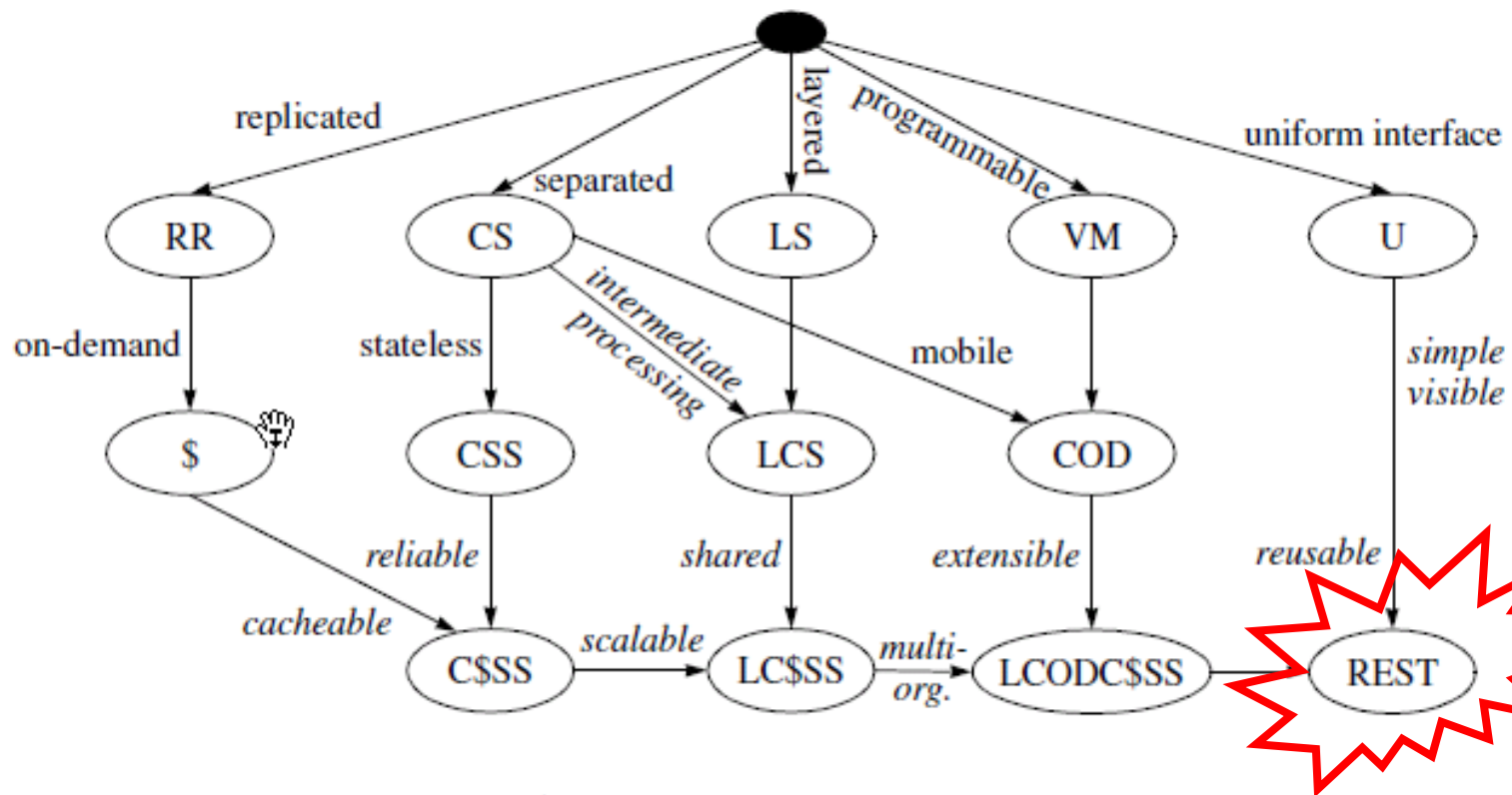


Figure 5-9. REST Derivation by Style Constraints

特に WWW のアーキテクチャスタイル (WWW は REST のインスタンス) クラサバ (CS) から派生した複合型スタイル

# Client-Server (CS)

---

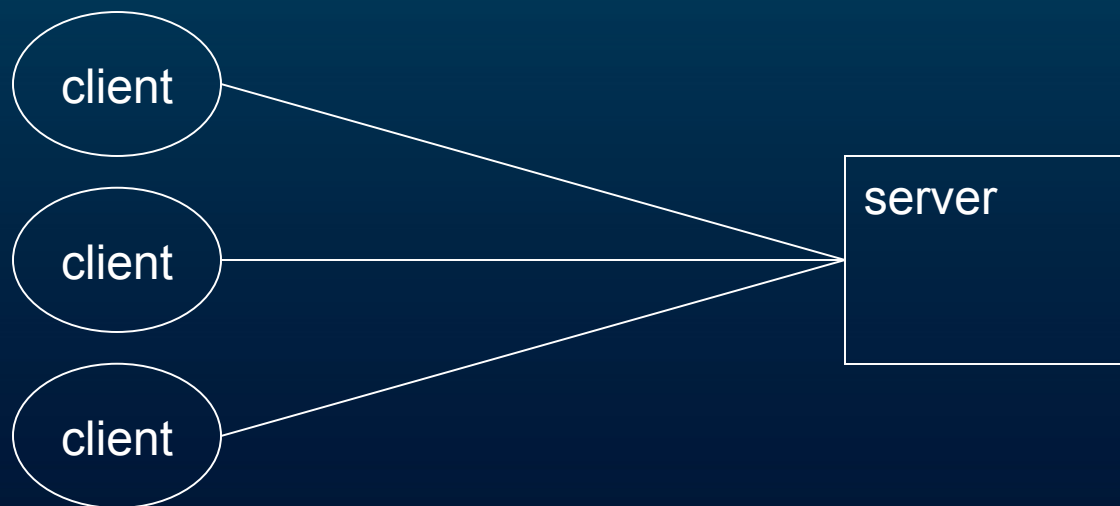
- ネットワークシステムで一番有名なもの
- 二つのコンポーネント
  - サーバ: クライアントからのリクエストを待ちながらサービスを提供
  - クライアント: サービスにリクエストを投げ、レスポンスを受け取る
- 制約: ユーザインターフェースとデータストレージの分離



- 特徴
  - マルチプラットフォーム
  - スケーラビリティ、サーバコンポーネントの単純化

# Client-**Stateless**-Server (CSS)

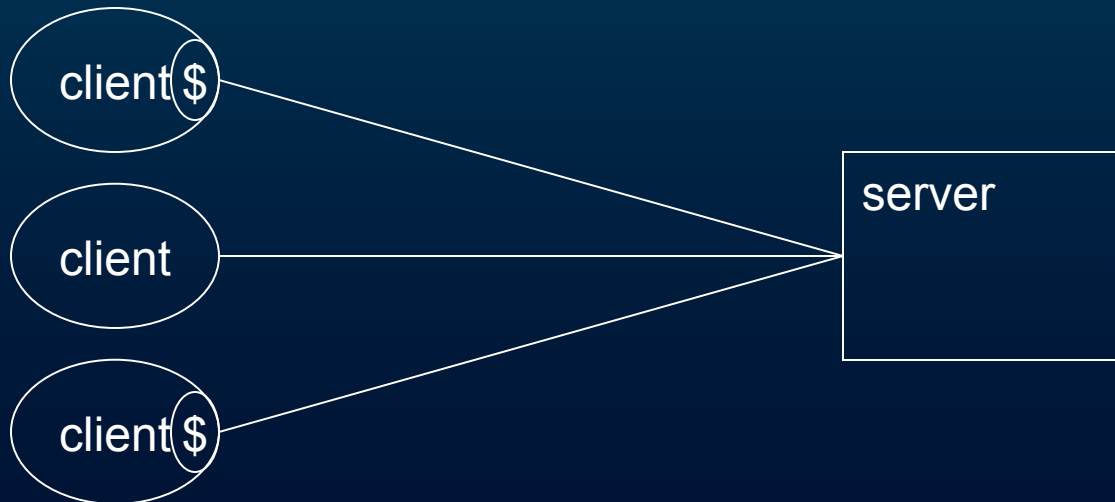
- ステートレス: サーバにセッション状態なし
- 制約: リクエストには処理に必要な全ての情報を含む



- 特徴
  - 可視性: 一つのリクエストだけモニタリングすればよい
  - 信頼性: 部分的な失敗から回復 (最初からやり直さなくていい)
  - スケーラビリティ: リソースをすぐに開放できる
  - ×認証情報などを繰り返し送ることによるパフォーマンスの減少

# Client-**Cache**-Stateless-Server (C\$SS)

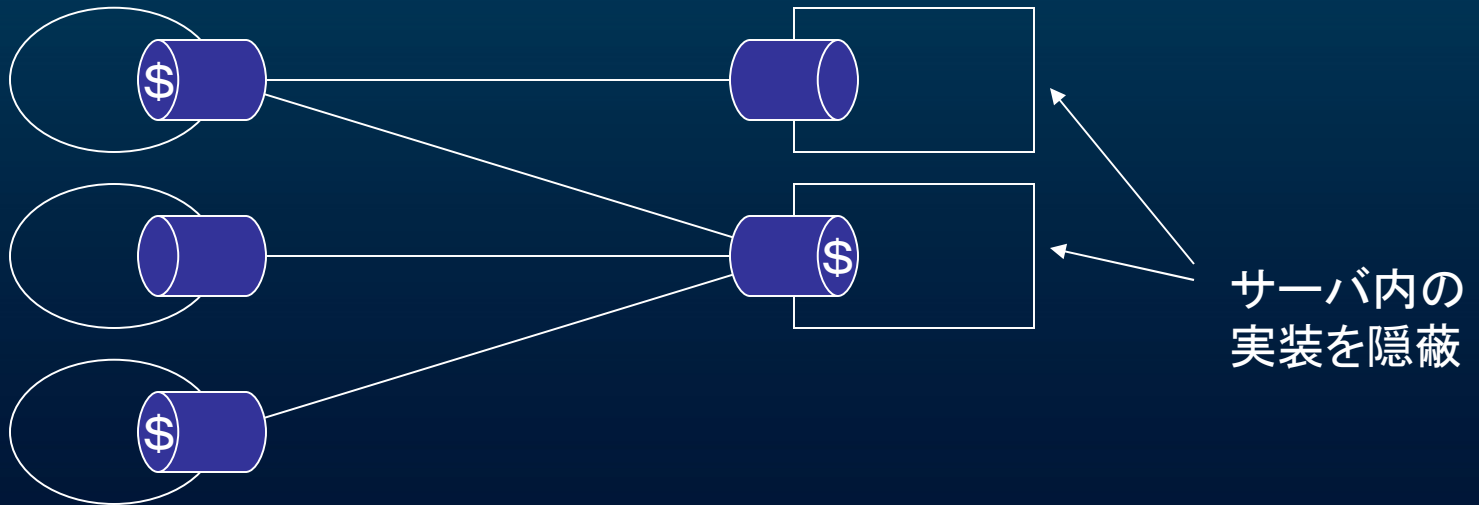
- キャッシュ: 同じリクエストは再利用する
- 制約: レスポンスがキャッシュ可能かどうかラベル付け (Cache-Control, Expires, ...)。キャッシュ可能なら、クライアント側で保持



- 特徴
  - サーバ、クライアントの対話を減らせる→パフォーマンス向上
  - ×信頼性の低下 (情報が更新されないケース)

# Uniform-Client-Cache-Stateless-Server (UC\$SS)

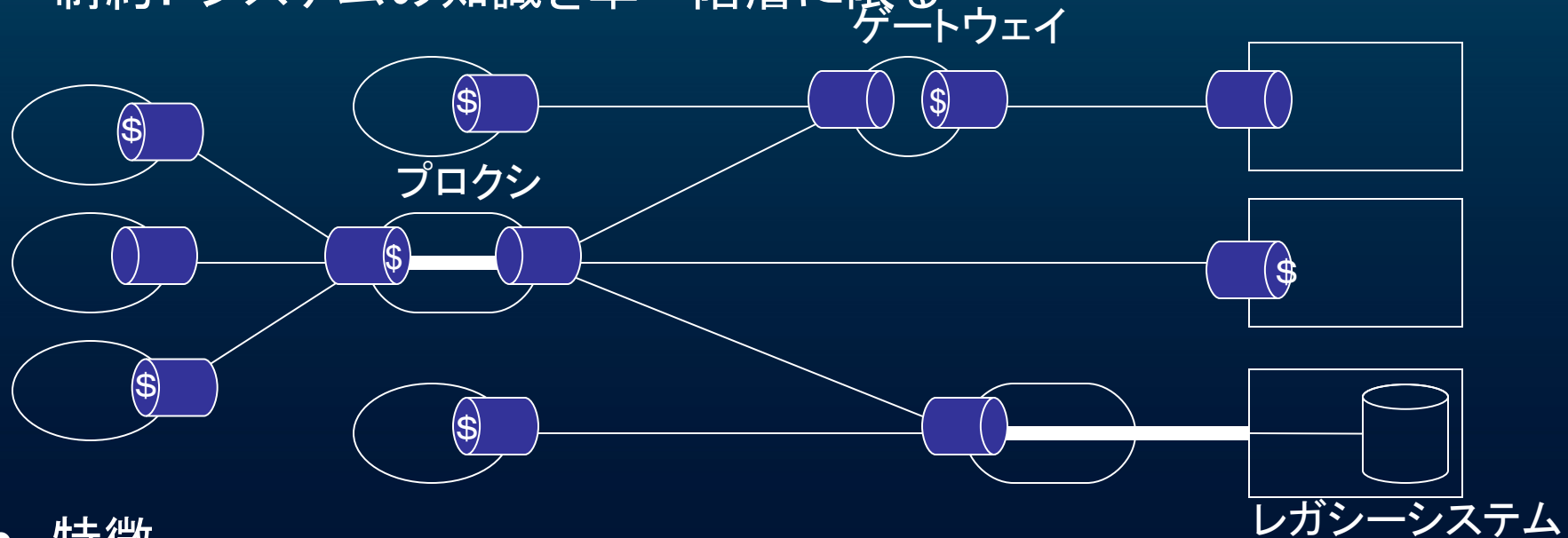
- 統一インターフェース: REST を最も特徴付けるスタイル
- 制約: コンポーネント間のインターフェースを固定



- 特徴
  - アーキテクチャがシンプルに
  - 可視性の向上
  - クライアント/サーバ実装の独立性向上
  - ×情報粒度によってはトレードオフが発生

# Uniform-Layered-Client-Cache-Stateless-Server (LUC\$SS)

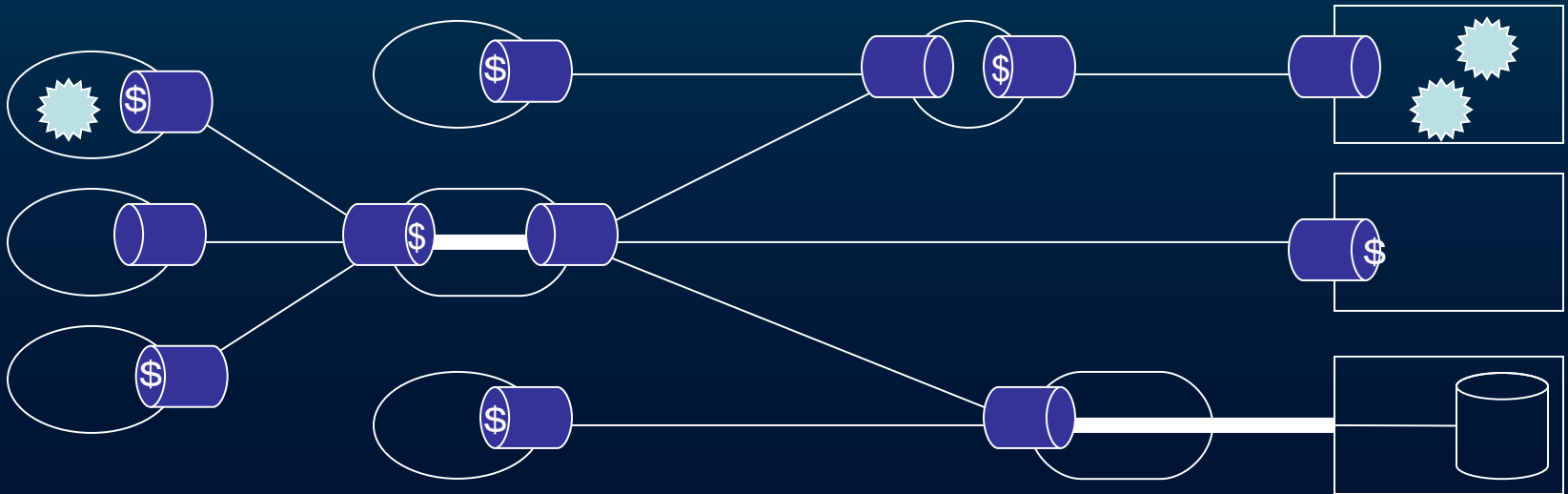
- 階層化システム: システムを複数階層に分割
- 制約: システムの知識を単一階層に限る



- 特徴
  - コンポーネントの単純化 (他レイヤは知らなくてすむ)
  - 中間子の配置 (プロキシやロードバランサなど)
  - レガシーシステムの封じ込め
  - ×ネットワークオーバーヘッドによる待ち時間の増加

# REST = Uniform-Layered-Code-on-Demand-Client-Cache-Stateless-Server (ULC\$DC\$SS)

- コードオンデマンド: クライアント側でコードをダウンロードして実行 (Flash, JavaScript)
- 実は REST では COD はオプション



- 特徴
  - ○クライアントを拡張できる
  - ×可視性の低下

---

というわけで

---

これで REST は完璧

---

というわけにはいかない

---

# RESTはあくまでも アーキテクチャスタイル

---

実際のアーキテクチャが必要

---

# Resource Oriented Architecture (リソース指向アーキテクチャ)

# リソース指向アーキテクチャ(ROA)

---

- 「RESTful Web サービス」で提唱されたアーキテクチャ
- Web 技術 (URI/HTTP/XML) を使った RESTful なアーキテクチャ
  
- URI の設計
- XML の使い方
- HTTP メソッドの使い分け

---

ところで

---

リソースって何でしょう

# リソース

---

- REST における超重要な概念のひとつ
- ROA 的には Web 上に存在する情報/データ
- リソースの例
  - 東京の天気予報
  - RFC 5023
  - Apache 2.2.8
  - “REST” に関する検索結果

# リソースの識別子 URI

---

- すべてのリソースは識別子 (identifier) を持つ
  - 一つのリソースに複数の識別子がつくこともある
- URI
  - 東京の天気予報
    - <http://weather.yahoo.co.jp/weather/jp/13/4410.html>
  - RFC 5023
    - <http://tools.ietf.org/rfc/rfc5023.txt>
  - Apache 2.2.8
    - <http://ftp.riken.jp/net/apache/httpd/httpd-2.2.8.tar.bz2>
  - “REST” に関する検索結果
    - <http://www.google.co.jp/search?q=REST>

# URI はなぜ素晴らしいのか

---

- URI はメールに貼り付けられる
- URI はブックマークできる
- URI はリンクできる
- URI は別アプリに渡すことができる

# URI はなぜ素晴らしいのか

---

- Apache の最新バージョンの示し方
  - ftp.riken.jp に anonymous FTP でログインして、/net/apache/httpd に cd し、binary として httpd-2.2.8.tar.bz2 を取得する
- URI なら1行
- `http://ftp.riken.jp/net/apache/httpd/httpd-2.2.8.tar.bz2`

# URI はなぜ素晴らしいのか

---

- “REST” の検索結果をメールしたい
  - [www.google.co.jp](http://www.google.co.jp) をブラウザで開き、検索ボックスに REST と入れて、「検索」ボタンを押してください
- URI なら1行
- <http://www.google.co.jp/search?q=REST>

---

これらは全て

---

URI がアドレス可能だから!

# ROA の4つの特徴

---

- アドレス可能性
  - Addressability
- ステートレス性
  - Statelessness
- 接続性
  - Connectedness
- 統一インターフェース
  - Uniform interface

# アドレス可能性

---

- URI はアドレス可能
- データセットのさまざまな側面を個別に URI として公開可能
- 全てのリソースはアドレス可能で URI を持つ
- Ajax アプリケーションはアドレス可能ではないことが多い
  - 例: gmail の URI は常に mail.google.com
  - 各メールの permalink は隠蔽されている

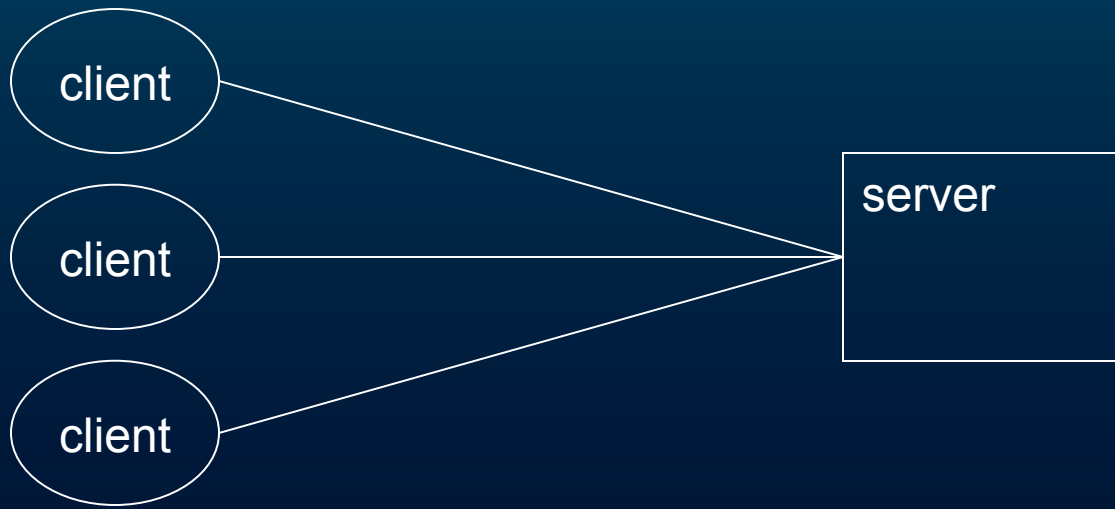
# ROA の4つの特徴

---

- アドレス可能性
  - Addressability
- ステートレス性
  - Statelessness
- 接続性
  - Connectedness
- 統一インターフェース
  - Uniform interface

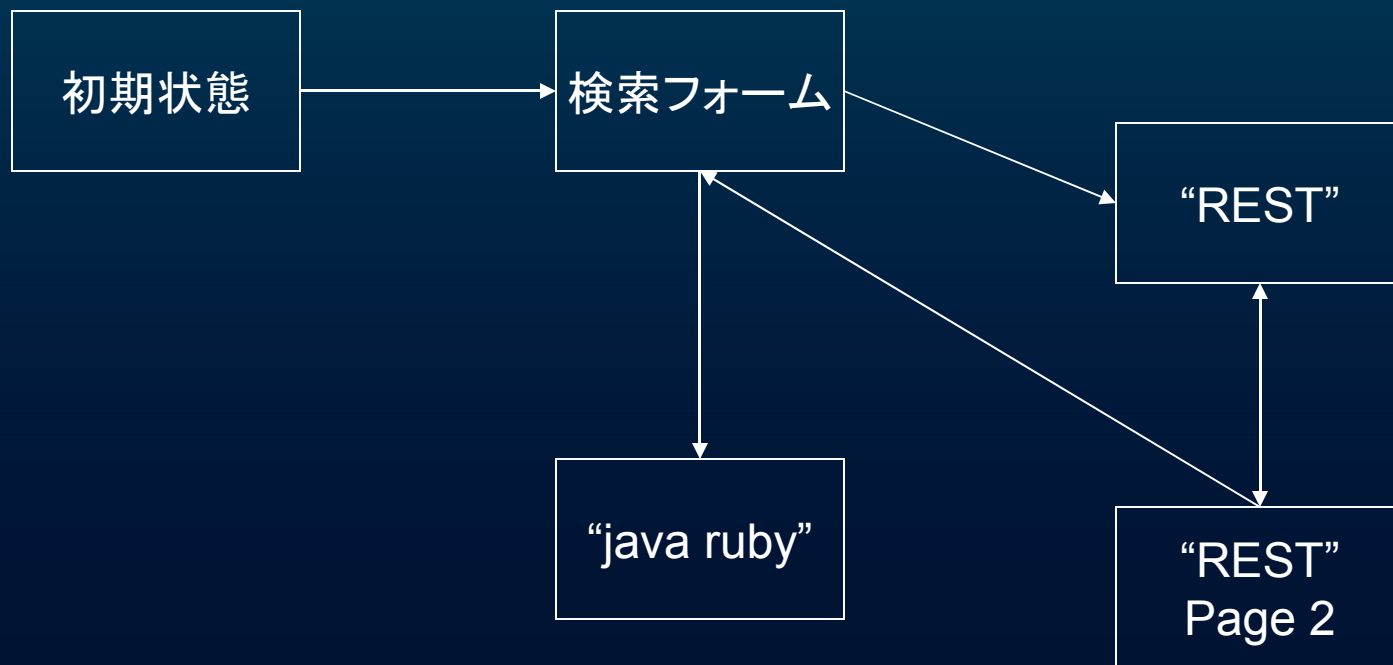
# ステートレス性

- ステートレス: サーバにセッション状態なし
- 制約: リクエストには処理に必要な全ての情報を含む



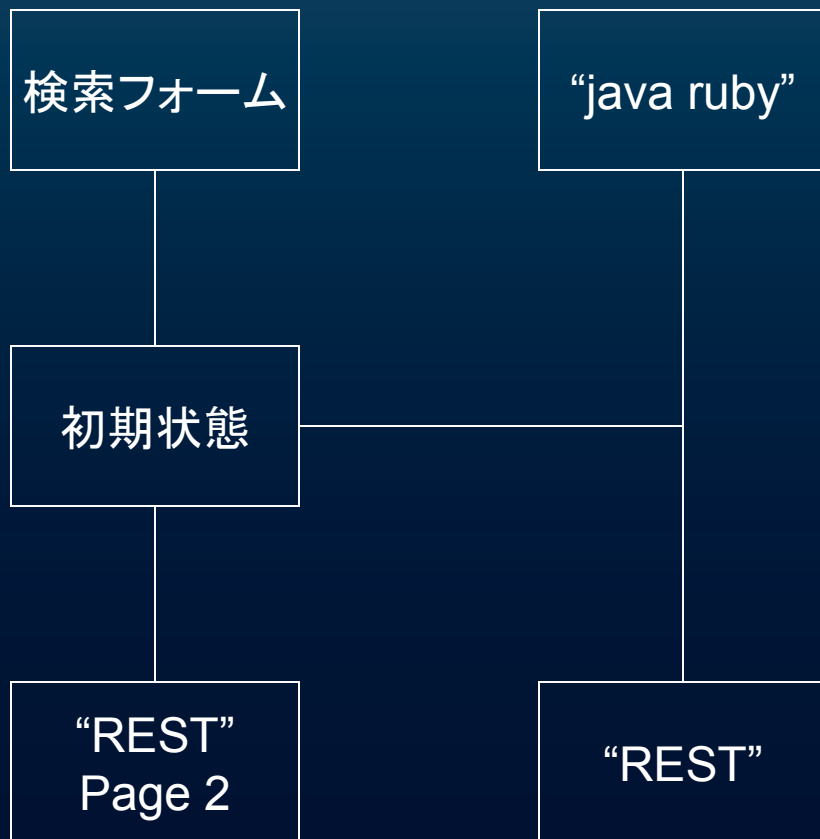
- 特徴
  - 可視性: 一つのリクエストだけモニタリングすればよい
  - 信頼性: 部分的な失敗から回復 (最初からやり直さなくていい)
  - スケーラビリティ: リソースをすぐに開放できる
  - ×認証情報などを繰り返し送ることによるパフォーマンスの減少

# ステートフルな検索エンジン



# ステートレスな検索エンジン

---



# ステートレス重要

---

- ステートレス = 状態なし
  - メッセージ間のコミュニケーション状態がない
- 前のリクエストで××だったから、次のリクエストには〇〇で答える、という実装にしないですむ
  - 計算機リソースを開放できるので、スケーラビリティが向上
  - 疎結合への第一歩
- ステートレスであるためには、リクエストメッセージは自己記述的である必要がある
  - Host ヘッダ、キャッシュ情報、認証情報、など
- クッキーでのセッション管理は NG

# ステートレスの特徴

---

- Pros

- いつでも、全ての状態に移動可能
- 失敗したら、やり直せる
- サーバ側にクライアントのアプリケーション状態を保存しない

- Cons

- 1つのメッセージに全ての情報が必要なのでメッセージが複雑に
- 認証処理などを毎回行うため、パフォーマンスが劣化

---

ところで

---

状態って何の状態でしょうか

---

# 二つの状態

---

アプリケーション状態

リソース状態

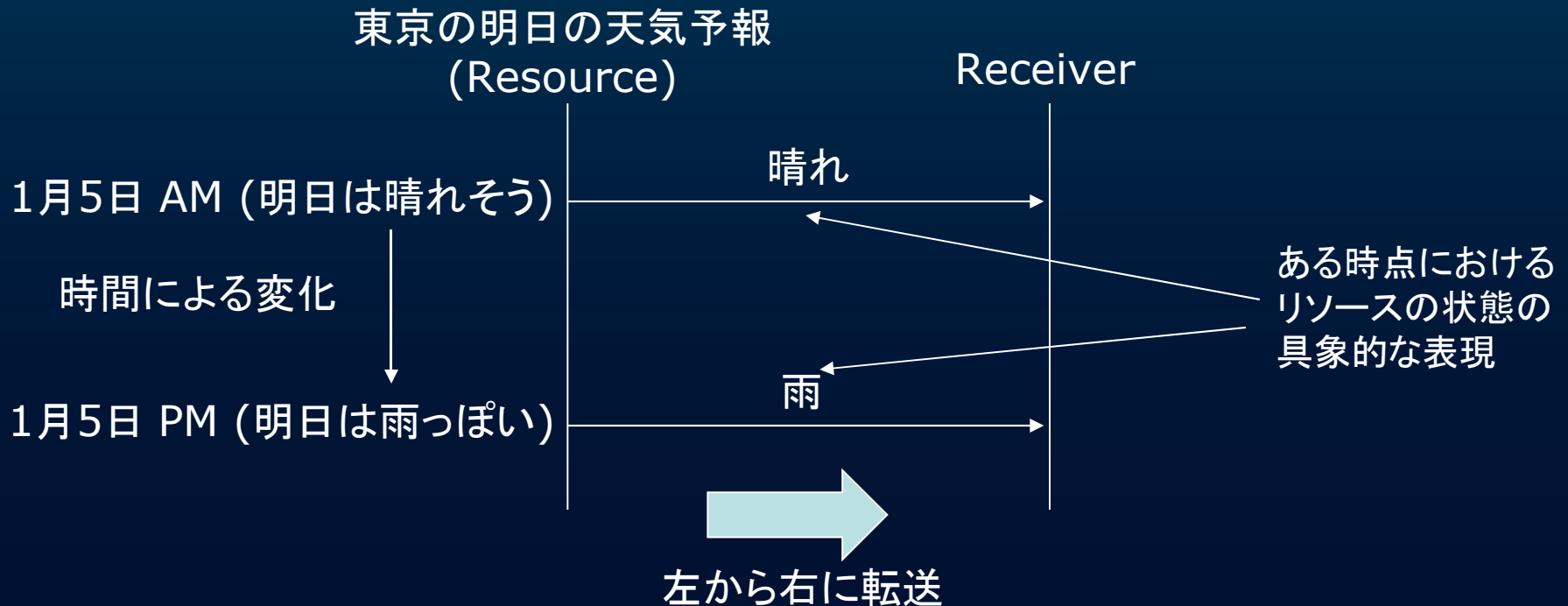
# アプリケーション状態

---

- クライアントは状態 (state) を持つ
  - 今、どの Web ページを表示しているか
  - どのチェックボックスを選択し
  - どのユーザで認証しているか
- これがアプリケーション状態
  
- サーバがアプリケーション状態を記憶しない
- = ステートレス

# リソース状態

- リソースは状態 (state) を持つ
  - 時間や条件とともに内容が変化する可能性がある
  - 「リソースの意味」は時間や条件によらず不変である



- REST はリソースの representational state を transfer する

# ROA の4つの特徴

---

- アドレス可能性
  - Addressability
- ステートレス性
  - Statelessness
- 接続性
  - Connectedness
- 統一インターフェース
  - Uniform interface

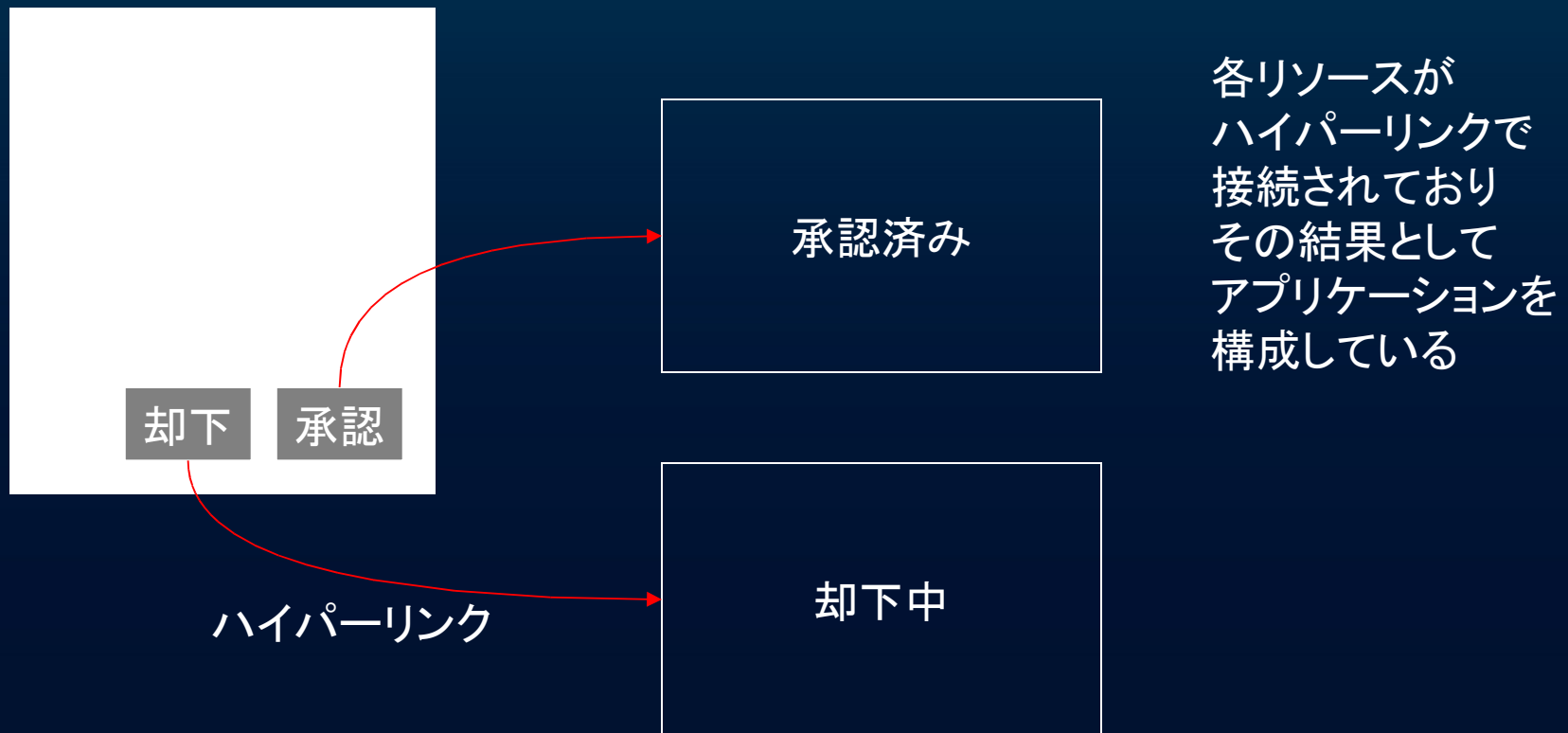
# リンクをたどる -- ハイパーメディア

---

- A.html のリンクをクリックして B.html へ移動
  - A というリソースの表現から B というリソースの表現へ移動
  - クライアントのアプリケーションの状態が A から B になる
- リソース同士は単純に URI と HTTP (リンク) で結びついている
- 実はこれはすごいこと
  - 中央リンク管理サーバやローカルリンク管理が必要ない
- URI さえあれば HTTP でリソースを操作できるので、アプリケーション連携がとても簡単
  - いわゆる疎結合の実現
  - 拡張性もあるよ！

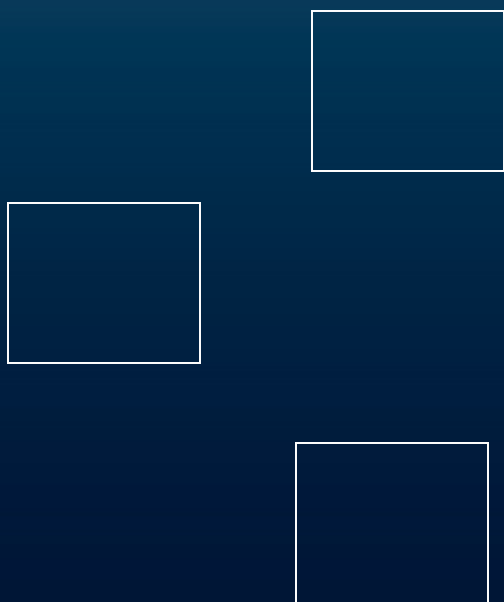
# Hyper media as the engine of application state

- アプリケーション状態エンジンとしてのハイパーメディア
- 例: Web 上のワークフローシステム

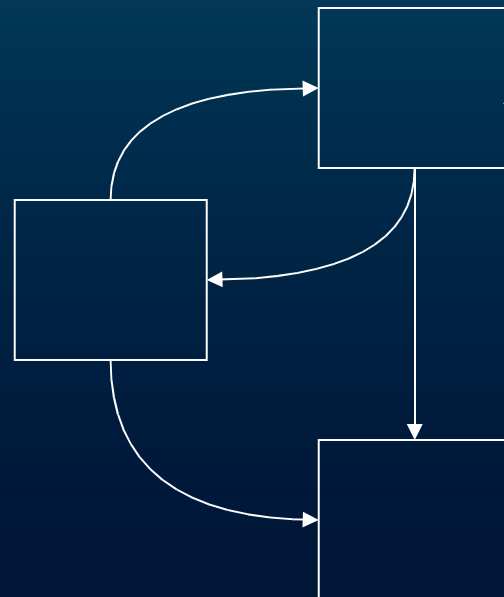


# 各リソースは接続されているべき

---



接続されていない



接続されている

---

リソースが接続されていると

---

クライアントが作りやすい

---

接続性重要

# ROA の4つの特徴

---

- アドレス可能性
  - Addressability
- ステートレス性
  - Statelessness
- 接続性
  - Connectedness
- 統一インターフェース
  - Uniform interface

# 統一インターフェース

---

- REST を構成する超重要なスタイルの一つ
  - HTML を取得するのが GET\_HTML で、PDF を取得するのが GET\_PDF だったら、今の Web はどうなっていたでしょう?
- 統一されるのはコンポーネント間のインターフェース
  - ブラウザ | プロキシ | リバースプロキシ | サーバ
- リソースを識別する統一的な識別子の存在
  - URI
- 全てのリソースに適用できる洗練されたオペレーション (CRUD)
  - GET                    リソースの取得 (Retrieve)
  - PUT                    リソースの更新 (Update)
  - DELETE                リソースの削除 (Delete)
  - POST                   リソースの作成 (Create ...だけではない)
  - HEAD                   リソースのメタデータの取得
  - OPTIONS                リソースがサポートするメソッドを調べる

# メソッドの安全性

---

- GET/HEAD/OPTIONS は安全
- このメソッドを適用しても、リソース状態に副作用がない
  - 例:  $\times 1$  ( $3 \times 1 \times 1 = 3$ )
- URI をメールでもらったら、安心して GET できる

# メソッドのべき等性

---

- GET/HEAD/OPTIONS/PUT/DELETE  
はべき等
- 何回このメソッドを適用しても、結果としてのリソース状態は変化しない
  - 例:  $\times 0$  ( $3 \times 0 = 0$ ,  $3 \times 0 \times 0 \times 0 = 0$ )

# 統一インターフェースの重要性

---

- 全てのコンポーネントで同じインターフェースを利用する
  - 全てのコンポーネントが GET は「読み取り」を意味することを知っている
- 統一インターフェースによって、分散システムを作るときの最大の障壁がなくなった

# ROA の4つの特徴

---

- アドレス可能性
  - Addressability
- ステートレス性
  - Statelessness
- 接続性
  - Connectedness
- 統一インターフェース
  - Uniform interface

---

# 設計指針

# 良いURI とは

---

- URI は名詞で構成する
  - 動詞(get/put 等)は入れない
- 記述的な URI を使う
  - わかりやすい名前(SEO効果)
- アドレス可能性

# XML の使い方

---

- ハイパーメディアフォーマットとしての XML を意識する
- リンクを入れよう
  - XHTML/Atom を使えば、自動的にリンク機能が入る
- オレオレフォーマットは作らず、なるべく再利用 + 拡張を心がける
- 接続性

# HTTP の利用方法

---

- HTTP メソッドは正しく使う
  - GET で削除しない
- ステータスコードも正しく使う
- 全てのリクエストに必要な情報を含める
  
- ステートレス性
- 統一インターフェース

---

最後に

# まとめ

---

- REST は WWW のアーキテクチャスタイル
  - REST を知らずして Web 2.0 を語ることなかれ
- ROA 4つの特徴
  - アドレス可能性
  - ステートレス性
  - 接続性
  - 統一インターフェース
- REST を構成する各制約の意味をきちんと把握しよう
  - スペックだけみても駄目
- アーキテクチャスタイル重要
  - ソフトウェア技術者の基本的スキルとなる
- <http://yohei-y.blogspot.com/>
- <http://blogs.ricollab.jp/webtech/>

---

おわり