分散システム基礎とクラウドでの活用

第5回：クラウドにおける設計

国立情報学研究所
石川　冬樹
f-ishikawa@nii.ac.jp

今回の内容

- スケーラビリティや伸縮性を嗜好したクラウドサービスにおける設計思想を学ぶ
  - 盛んに行われているCAP定理や一貫性に関する議論を概観する
  - GoogleやAmazonにおける既存サービスの設計思想を概観する

目次

- クラウド概論（ごく簡単に）
- アーキテクチャ・プログラミングの例
- CAP定理
- フロントエンド向けサービスの例

クラウド：定義（例）

- 「計算資源へのアクセス」のためのモデル
  - On-demand self-service：提供者との人手を介したやりとりなく，自動で取得可能
  - Broad network access：様々なプラットフォームから標準的な方法で利用可能
  - Resource pooling：詳細が隠蔽された形でプールされ，複数の利用者に提供

（次頁に続く）

[The NIST Cloud Definition (Final Ver. Sep 2011)]

クラウド：定義（例）

- 「計算資源へのアクセス」のためのモデル

  （前頁からの続き）

  - Rapid elasticity：迅速に，伸縮可能に提供されており，無限に見えるものから必要な分だけ取得
  - Measured Service：抽象的な指標での測定に基づき，利用が可視化された形で制御，最適化

[The NIST Cloud Definition (Final Ver. Sep 2011)]

クラウド：サービスモデル

- SaaS (Software-as-a-Service)
- PaaS (Platform-as-a-Service)
- IaaS (Infrastructure-as-a-Service)

※ PaaS/IaaSの区別は本講義では気にしない

- それらはいずれにしても計算資源を提供する側
- それらを利用するアプリケーション（やSaaS）に提供される保証の内容や，その実現アプローチを，本講義では議論している

クラウド：語られる例（１）

スケーラビリティ（scalability）

伸縮性（Elasticity）

- Animoto：動画作成・閲覧サービスをFacebookユーザが利用可能に（2008）
  - Amazon EC2を利用・3日後には25万ユーザ（元々2万5千）に対応するために4000サーバ（元々50）
- ホワイトハウス：アンケートWebサイト（2009）
  - Google App Engineを利用・2日間で10万の質問と3600万の返答（最大秒あたり600クエリ）

クラウド：語られる例（２）

事前または保有のコストをかけず柔軟な利用（Quick Start, No Up-front Investment, Pay-per-Use）

- New York Times：100年分の新聞をPDFに変換，テキスト抽出（2007）
  - Amazon EC2・24時間100台の仮想マシンを利用（1000ドル強）
- 日本の公的機関：突然一時的に必要となるシステムの立ち上げ（定額給付金管理）（2009）
  - Force.com

クラウド: スケールメリット

- 多く買うほど安い

	1,000 servers	50,000 servers
Network (per 1M/sec)	$95	$13
Storage (per 1G)	$2.2	$0.4
Management (per 1 supervisor)	140 servers	1000 servers

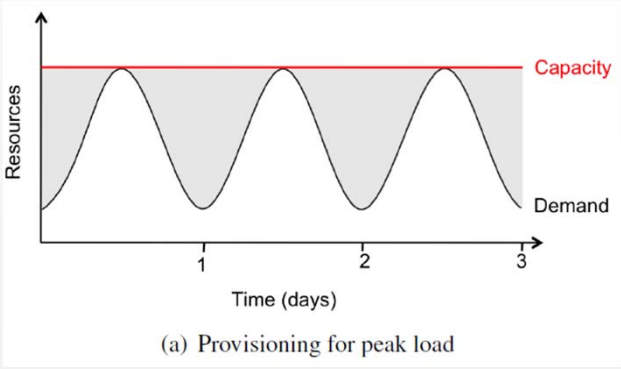
[Above the clouds: A berkeley view of cloud computing, 2009]

おまけ: 現在, Amazonのクラウドへのトラフィックは,「本」業のものより多くなっている

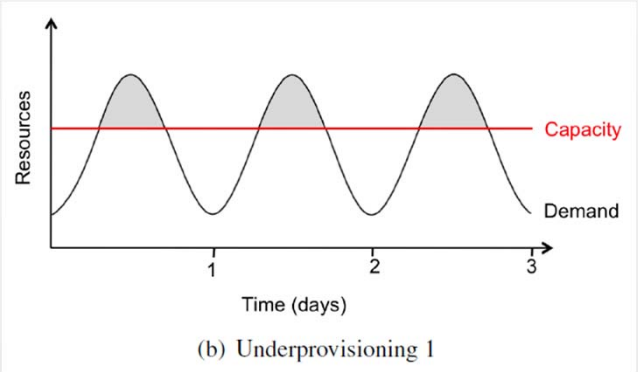
- 「本」業の基盤のコスト減少にもますます効く

クラウド：伸縮性の考え方

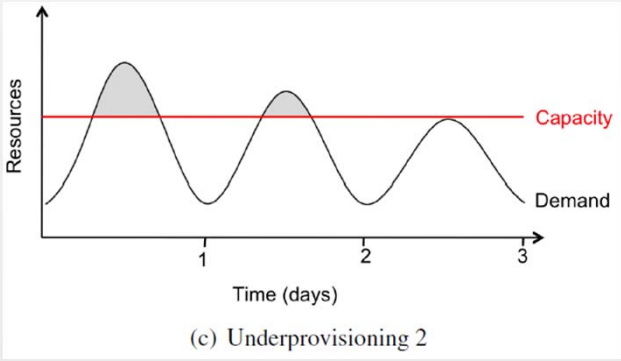
- 固定数のサーバにおける
  - 無駄な余剰資源
    または
  - 機会損失



(a) Provisioning for peak load



(b) Underprovisioning 1



(c) Underprovisioning 2

[Above the clouds: A berkeley view of cloud computing, 2009]

クラウド：固有の設計思想

- 今までのシステムを，単にそのままクラウド上で走らせることも考えられる
  - 今まで手元のサーバ上で動かしていたものを，Amazon EC2などのIaaS上で（仮想マシンとして）配備，動作させてもよい
  - Webフレームワーク，RDBとトランザクションなど
- 一方，今までのシステムとは異なる特長，そのための設計思想が注目されることも多々ある
- ➡ 大規模分散（性能，スケーラブル，耐故障）

クラウド：固有の設計思想

- Webスケールのデータ量・アクセス量に対しては，どんな「すごい」コンピュータでも1台では足りない
- 安い（普通の）ハードを大量に使う
- 大量にあると，
  - 常に障害がどこかで発生している
  - 障害が発生した部分を（ある程度の期間）切り捨てても全体としては機能を維持できる
- 大量のデータやアクセスに対応できる設計？

＋

一部の障害による影響を受けにくい設計？

メモ：Scale UpとScale Out

- 大量の処理（リクエストなど）をさばくには
  - Scale Up：強力なサーバを用いる
    - 従来のオンプレミスサーバ
  - Scale Out：管理ソフトウェアツールとともに，大量のサーバを用いる
    - Webスケールのデータ（例：サーチエンジン）
    - 落ちているサーバが常にあるという仮定に基づいた運用
    - データや機能について，並行実行・障害復帰しやすい特定の形式を用いる
      （例：Key-Valueデータストアや Map Reduce）

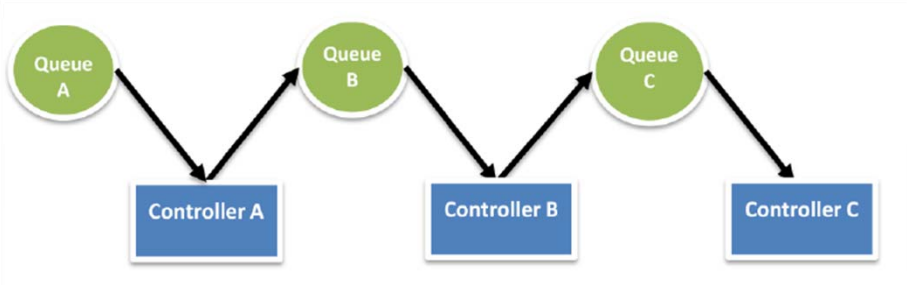
目次

- クラウド概論（ごく簡単に）
- アーキテクチャ・プログラミングの例
- CAP定理
- フロントエンド向けサービスの例

「クラウドらしい(？)」アーキテクチャ

Amazonのドキュメントを見てみると・・・

- 複製による並行処理・耐故障化を容易にするコンポーネント分割
  - 各コンポーネントの失敗や遅れが互いに影響しないように疎結合を
  - ➡ 特にバッチ処理の場合など，水平(同機能の)スケールのためには非同期アーキテクチャに

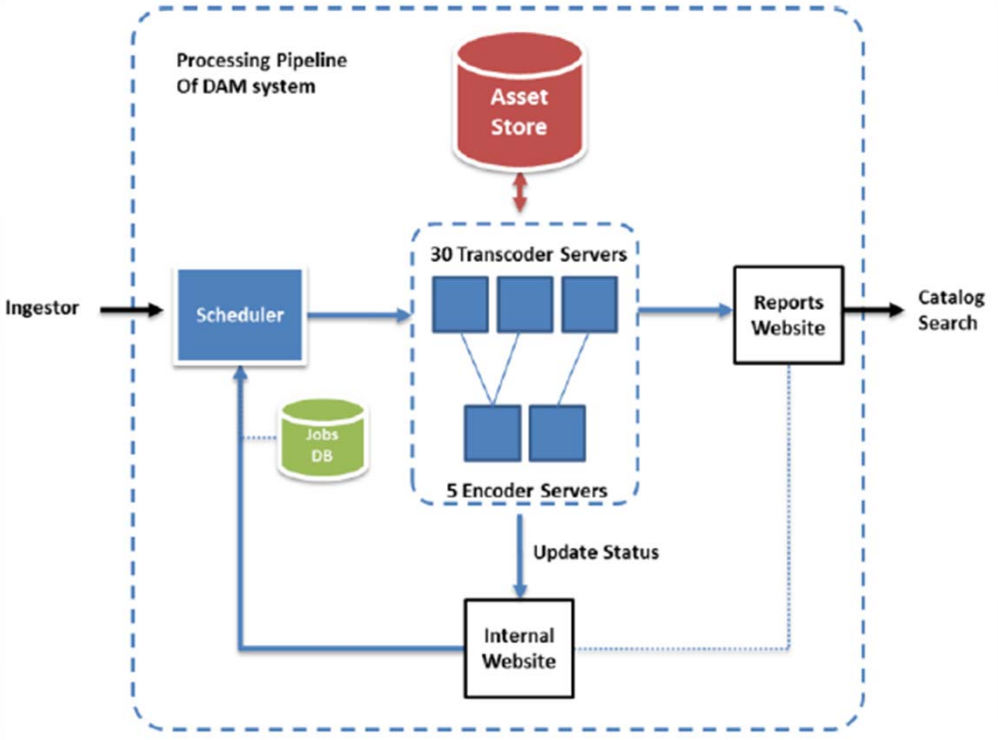


[J. Varis, Architecting for The Cloud: Best Practices, 2010]

「クラウドらしい(？)」アーキテクチャ

Amazonのドキュメントを見てみると・・・

- バッチシステムの移行例(before)

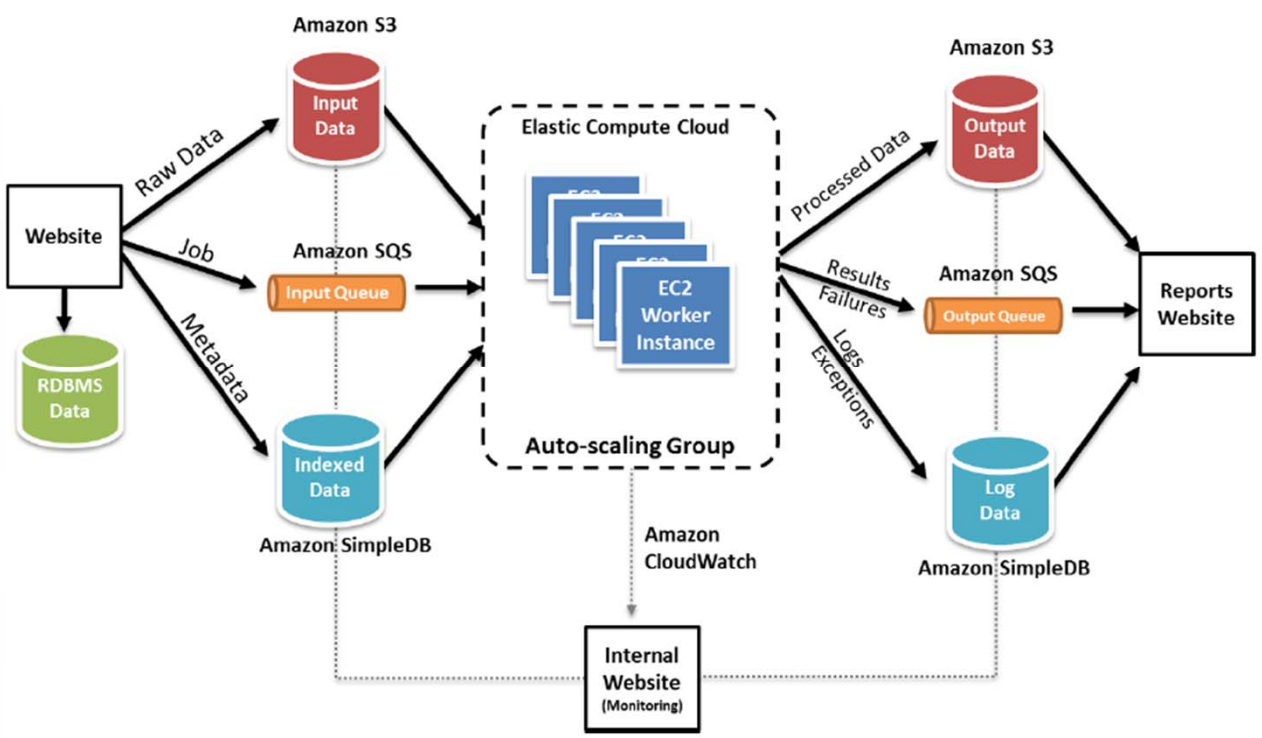


[J. Varis, Migrating your Existing Applications to the AWS Cloud, Oct 2010]

「クラウドらしい(？)」アーキテクチャ

Amazonのドキュメントを見てみると・・・

- バッチシステムの移行例(after)



[J. Varis, Migrating your Existing Applications to the AWS Cloud, Oct 2010]

補足：Amazonのクラウドサービス（の一部）

- 前スライドに出ているもの
  - EC2 (Elastic Compute Cloud)：仮想マシンを走らせる計算資源サービス（いわゆるIaaS）
  - S3：ファイルをget/putするストレージ
  - Simple DB：非リレーショナル型のDB（Key-Valueデータストア）
  - SQS（Simple Queue Service）：分散，複製されたキューを提供

※ 全体像は http://aws.amazon.com/jp/products/

Amazon Simple DB

- Amazon Simple DB
  - 非リレーショナル型（Key-Value型）データストア
    - CustomerID=123
      GName: Bob, LName: Smith, Addr: 100 Main St.
    - CutomerID=456
      Gname: James, LName: Johnson, Tel: 1112223333
  - 複製分散され，全体を見渡すような演算は効率よく行えない（合計やいわゆるJoin）

http://aws.amazon.com/jp/simpledb/

Amazon Simple DB

- Amazon Simple DB（続）
  - 一貫性：デフォルトはEventual Consistency
    - 最新の値を読み込まないかもしれないが，一定時間（1秒程度）内に読み込むようになる
    - すべての事前の書き込みを反映するような読み込みを（少し時間を長くかけて）行うことも可能
  - トランザクションは一部のみ
    - 値を判定するとともに書き換える・消去する

http://aws.amazon.com/jp/simpledb/

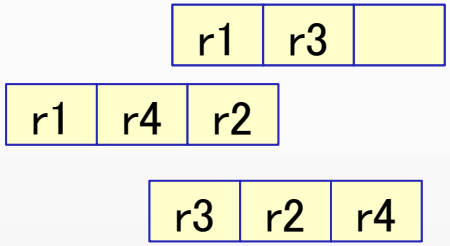
（前回より）イベンチュアル一貫性

- イベンチュアル一貫性（eventual consistency）
  - 「更新はすべてのコピーに伝搬する」（更新がなければすべての複製は同一に収束していく）
  - 特別な性質を持つデータストアが対象
    - 書き込み同士の衝突はない
    - 読み込みが必ずしも最新でなくてもかまわない（例：Webサイト，DNS）
  - ***追記：性能やスケーラビリティの追求のためにあえて採用することがクラウド関連では多い***

NoSQL

- **NoSQL**: Relational DBのモデルに従わないデータストアのモデル
  - **Key-Value Store（KVS）**が代表的
  - 大規模並行処理・耐故障のための大量複製
  - インターフェースは基本Get/Putであり，SQLは用いない（それっぽい言語を用意することもある）
  - 本質的に下記が高コストとなるため扱わない
    - JOIN演算
    - 合計値など全体にまたがる演算
    - RDBにあるような豊富なトランザクション

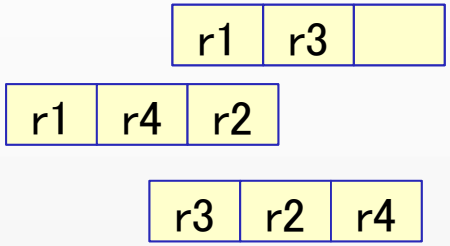
Amazon SQS



- Amazon SQS
  - メッセージの送受信のための分散キューを提供
  - キューは複製され，送受信はそのどれかに対して行われる
    - 全てのキューが同じデータを持つ状態に「収束」していく（Eventual Consistency）
    - キューを読み出したとき，その前に送られたメッセージが含まれるとは限らない

http://aws.amazon.com/jp/sqs/

Amazon SQS



- Amazon SQS（続）
  - 受信されたメッセージは，一時的に見えなくなる（受信されなくなる）が，一定時間のタイムアウトの後に再びキューに含まれる
    - メッセージを受信しタスクを行うプロセスは，終了後に明示的にメッセージ削除を行う
    - もしもそのプロセスがクラッシュしたり，繋がらなくなったりした場合も，確実に実行するようにする（重複して実行される可能性はある）

http://aws.amazon.com/jp/sqs/

デモ：簡単なサンプルプログラム

- SQSに対し，
  - タスクを表すメッセージ（実際はただの数字）を順に送信
  - 複数スレッドを立ち上げ，各スレッドはメッセージを最大10個受信し，1個ずつ，処理（一定時間をかけるだけ）し削除する
    - 受信するが，クラッシュして削除をしないような挙動も入れてみる
    - まとめて10個受信した後に，1個ずつ処理，削除をしていくので，後の方に処理されるメッセージほどタイムアウトする可能性がある

デモ：簡単なサンプルプログラム

- 結果
  - 送信された順に処理されるわけでは全くない
  - タイムアウトが短すぎると，同じタスクが別のスレッドにも受信され，かなり重複して実行される
  - クラッシュするスレッドがあっても，受け取っていたタスクは他のスレッドに受信され，処理される
  - タイムアウトが長いと，タスクの残りが少ない際に，未処理タスクがあるにもかかわらず，キューが空に見えることがある
    - クラッシュしたスレッドが受け取ったタスクがまだキューに現れていないため

今までと異なるノウハウ？

- SQSの場合
  - タスクは重複実行されてもよいようにする（これはSQSに限らないが）
  - 例：所要時間が異なるタスクは，別のキューに入れるようにする（と推奨されている）
    - タスクの所要時間を踏まえて，受信されたメッセージが再び見えるようになるタイムアウト時間を定める（と推奨されている）ため

目次

- クラウド概論（ごく簡単に）
- アーキテクチャ・プログラミングの例
- CAP定理
- フロントエンド向けサービスの例

CAP定理

- 複製を用いるアプリケーションは，下記の3つのうち2つのみを実現することができる
  - Consistency（一貫性）
  - Availability（可用性）
  - Partition Tolerance（ネットワーク分断への耐性）

[Eric Brewer, 2000]

※「定理」というより，気持ちは「設計原則」
（かなり状況を絞って，狭義で厳密な定義を行い，「定理」として証明した人たちもいる）

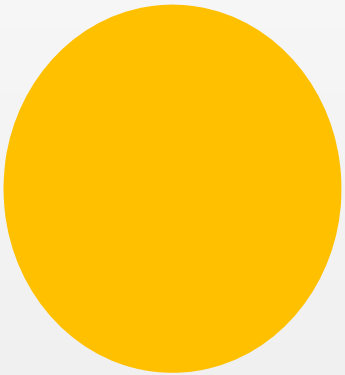
CAP定理

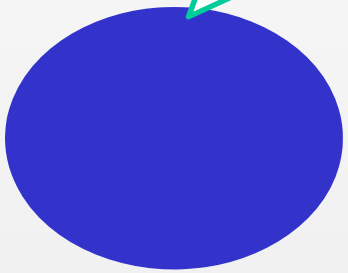
- Consistency
  - 前回扱った，特定の（順序に関する）約束事を守ってのデータ更新やその耐久性（durability）
- Availability
  - 性能と耐故障の両側面を反映（一部の複製のクラッシュなどがあっても，十分に速く反応する）
- Partition tolerance
  - 現在のデータセンターでは，つながらないノード群は落としてしまう方針で，あまり問題にならない
  - データセンター間では十分にあり得る

CAP定理とクラウド

- ありがちなクラウドの層分け

いかに負荷を
前で吸収できるか？





First Tier
リクエスト処理を
行うWebページ

Second Tier
スケーラブルな
Key-Valueデータ

Inner Tiers/Back-end
データベースや
インデックス，
バッチ分析処理など

「とにかくレスポンスを速く，大量に」
（Consistencyを犠牲に？）という議論の対象

CAP定理とクラウド

- フロントエンド側においてCよりAPを優先する例
  - 例：バックエンド側とつながらなくなるなどしても，ユーザにはキャッシュやエラー時用の情報などを用い，すぐに反応する
  - 例：人気新商品の販売開始時に，在庫数が数百まで減るまでは，システム全体で正確な在庫数を共有，合意することはせず，各複製グループごとに販売を記録しながらどんどん売ってしまう

CAP定理への反論

- バックエンド側ではConsistencyが必要
  - 例：購買操作
  - そのための技術（RDBなど）は，十分に確立されている
- Cを実現すると，APが犠牲になる？
  - 「そんなことはない．今のRDBは十分APを実現している．」
  - 「スケーラビリティの程度が違う」
  - 「実現コストが違う」

CAP定理：本講義での理解

- 現実的な設計指針として，

  スケーラビリティ・性能を実現するために，
  整合性を弱めることを受け入れることが
  低コストな解であることが多い

BASE

- ACIDに対するBASE

  [Dan Pritchett (eBay)]

  - Basic Availability
  - Soft-State Replication
  - Eventual Consistency

Elasticity

- Elasticity（伸縮性）
  - アクセスの急激な増加に伴い，複製が50個から1000個，10000個へと変化できる（逆も）
  - First Tierでは，更新を非同期に反映しつつもすぐリクエストに反応ができるようにする

  ※ Availabilityの一部： Rapid Local Responsiveness
- Reconfigurabililty（再設定可能性）
  - それに合わせてルーティングも適宜行う（シンプルなポリシーでないとスケールしない）
  - 複製を急に落とす・増やすことができる

Rapid Local Responsiveness

- キャッシュの活用による反応性の確保
  - 例：ショッピングサイトの商品クエリ
    - 人気ある商品へのクエリ返答は，First Tierにキャッシュ（多少古くともよい）
    - 人気商品やキャンペーン情報などの付随情報はFirst Tierにキャッシュ（多少古くともよい）
  - First Tierでは，Quorumのような仕組みを用いずに反応性を確保することが多くなっている
    - 「収束が十分に速いイベンチュアル一貫性が実現されていれば，クライアントがリロードすればいいじゃない」（例：野球中継の一球速報）

First Tierにおける CAP定理原則の解釈

- Consistency
  - いわゆるACIDは保証しない
- Availability
  - Rapid Local Responsivenessを実現する
- Partition tolerance
  - 負荷が高い，ネットワーク分断された，障害から復帰途中である，といった他のプロセスを待つために，動きがとれないようなことがないようにする

First Tierにおける BASE

- トランザクションをあえて小さな部品に分割する
  - 可能なら並行に，不可能でもできるだけパイプラインを短く（本当に必要な依存性のみを明示）
- クラッシュの際にはロールバックしない
  - 必要なら補償する（例：ホテルを予約しても，支払いが完了しなければ後日キャンセルされる）
- ロックや同期も可能な限り行わない
  - タスク管理のキューのみ，など
- これらの影響をエンドユーザから隠すように，フロントエンドを設計する

First Tierにおける BASE

- Hard-state： durableなデータ
  - 例： ディスク上のファイルなど
  - 例： 2PCの目標と想定
- Soft-state： durableではないデータ
  - 例： 人気のあるコンテンツのキャッシュ
    （Second-Tierと即時には同期されない）

[Eric Briwer]

Vogels (Amazon CTO)：「少しの不整合やACID違反を『正常動作』と定義することを真剣に検討すべき」

目次

- クラウド概論（ごく簡単に）
- アーキテクチャ・プログラミングの例
- CAP定理
- フロントエンド向けサービスの例

おまけ：DHT

- 分散ハッシュテーブル（DHT: Distributed Hash Table）
  - 特定ノードへの負荷集中なく，データおよびルーティング情報を分散して持ち，効率的に発見
  - （単純化した）例：ノード16個（IDが0～15）
    - 各ノードは自分の「ID＋1, 2, 4, 8 (mod 16)」のIDを持つノードのIPアドレスだけ覚える
    - ノード2からノード15へ行くには，+8，+4，+1と進む（ノード数nに対して O(log(n)) ステップ）
    - データのハッシュを基に，そのデータを置くノードを決める

Memcached

- メモリ上に保持したKey-Valueペアの読み書きのためのAPI(とそのオープンソース実装)
  - 典型的な利用法は,「まずメモリ上から読み出しを試み, もしなければバックエンドから取得し追加」
  - 多数の実装と利用
    - Wikipedia, YouTube, Flickr, Twitter, WordPress.com, mixi
  - 各プロセスのメモリは基本互いに独立
    - 一部の実装では値の更新をプロセス間で共有(メモリアクセスなのでバックエンド経由より速い)

BigTable

- 一貫した表データアクセスを実現（Google）
  - (K, K`, V)タプルに対する操作（行，列，値）
  - さらに時刻も保持
    - 古くなったものが消えるようにすることができる
    - またはアプリケーションが明示的に削除
  - 元々はWebページの情報保持のために開発

	contents:	anchor:cnnsi.com	anchor:my.look.ca
com.cnn.www	<HTML> …	“CNN”	“CNN.com”

BigTable

- （続）
  - 列のキーは事前登録されたfamilyに分類される
    - 前頁の anchor:XXX, anchor:YYY
  - 同じ行および同じ列のファミリーに属するデータは，高速にアクセスされるよう物理的に「まとめて」置かれる
  - 複数のアプリケーションにより用いられる想定
    - familyの事前登録時に，名前の衝突に気づく
    - 概念的には一つの大きな表だが，値が入っていないところがほとんど（その分の領域は使わない）

BigTable

- （続）
  - トランザクションのサポート
    - 1つの行を原子的に更新
    - あるセルをカウンターとして利用可能
      （同じ値が2度読まれることはない）
  - CAPのすべてを十分に実現する生きた反例？
    - データセンター間分断の場合には，個々が独立して動作し，後でミラーリングにより復旧
      （データセンター内での分断発生は前例なし）

BigTable

- （続）
  - 行や列，列のfamilyを対象とするよう，APIやトランザクションの種類を絞っている
    - RDBのあらゆる操作が可能になっているわけではない
  - それら（だけ）が十分に速く動作するように，複製された小さなサーバ集合に割り当てる
  - （「あきらめ」を含めた）APIの設計と，実装方式の設計との連動により成功している

Dynamo

- AmazonによるKey-Valueストア
  - (バックエンドから見ると)Read-Onlyの，ブラウジングやショッピングカートの実装
    - Cookieによる状態保持など細かいこともあるが
- DHTを要求に合うように拡張
  - ユーザごとに特定の商品リストを取得可能
    (1つのキーが複数の商品に結びつく)
  - ユーザに対してすぐに反応できるようにする
    (あるルート上のノードが反応しない場合にも，代替えルートを利用，ただし一貫性の問題発生)

Dynamo

- （続）
  - 障害への対応
    - 反応しないルートがある場合，別ルートを用いる
    - ただし，「カートへの追加」という情報の反映を複製間で徹底するような復旧プロセスを行う
    - 非常に低い確率では，不整合が起きうる
      「カートに入れた商品が抜けていたから，もう一回入れたら，二個買っていた」

Zookeeper

- Yahoo!による，ファイルシステム風操作を提供するKey-Valueストア
  - ファイル名がKeyとなる
  - ファイルのバージョン番号も保持
  - 全順序アトミックブロードキャストを利用
  - ファイル名の階層構造に基づき，処理を分散

その他

- RDB風の操作を提供しているKey-Valueストア
  - Yahoo PNUTS, Cassandraなど
  - 強い一貫性を保証しないなどトレードオフはある
- Chubby: BigTableの裏側のロックサービス
  - 全順序マルチキャストによるロック管理
  - ノード数に対してスケールしない
    （独立したロック集合を扱うChubbyインスタンスをたくさん作ることはできる）

CAP定理とSecond Tier実装の選択肢

- 「ただ一つの正解」は，（そもそもそんなものがあるかどうかも含めて）明らかになっていない
  - ほぼCAPを実現しているBigTable
  - 一貫性を少し犠牲にしたDynamo

  どちらもうまく動いているし，どちらが真によいという証拠もない

今回のまとめ

- Web企業が自身の要求から産み出したようなクラウド技術においては，これまでと異なる設計思想に基づいている
  - スケーラビリティや伸縮性のため，一貫性を意図的に犠牲にすることがあり得る
  - 複製や障害対応が容易となるように，特定の制約を設けた（特定保証がない状況で正しく動く）プログラミングを前提としている
- 次回：ここまでの内容について，演習を通して理解を深め，議論する