

# Cloudの技術的特徴について

## --- ScalabilityとAvailability ---

---

早稲田大学

丸山不二夫

# Agenda

---

- システムのScalability
  - システムのAvailability
  - CAP Theorem
  - ScalabilityとAvailabilityの両立がConsistencyに与える影響
  - BASE Transaction
-

# はじめに

---

- クラウド技術の最大の特徴は、安価なサーバを沢山並べて処理能力を拡大するというScale-outの戦略である。
  - このことは、多数のマシンからなるScale-outのシステム構成では、システムを構成するマシンのエラーが、確率的には避けられないことを意味している。
  - これは、システムのAvailabilityにとっては、重大な問題である。
-

# はじめに

---

- 講演では、分散システムでは、Scalabilityと Availabilityが矛盾することから出発して、現在のクラウドシステムが、どのように、Scalabilityと Availabilityを両立させようとしているかを見ていく。
-

# はじめに

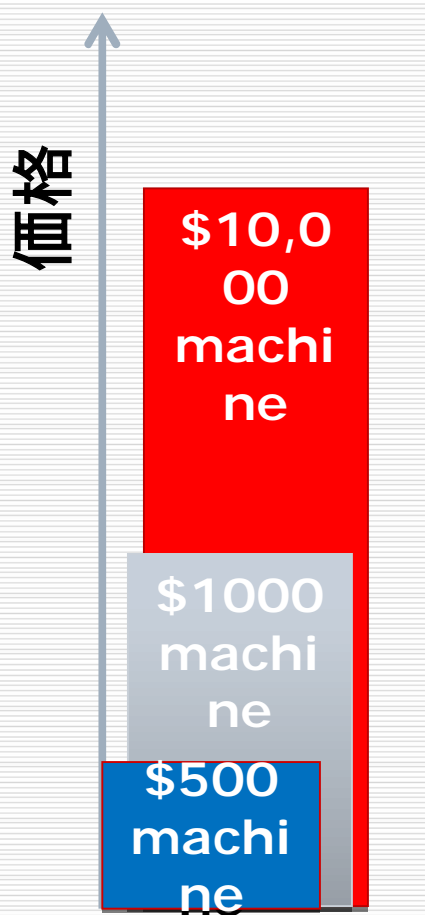
---

- クラウドのAvailabilityは、基本的には、マシンやデータのReplicaを複数抱える実装によって担われている。
  - ここでも、典型的には、Replicaとの同期の問題が、新しい問題を引き起こす。
  - 講演では、Eventually Consistencyという概念の導入や、TransactionにおけるACIDモデルの見直しと、新しいBASEモデルの提案を紹介する。
-

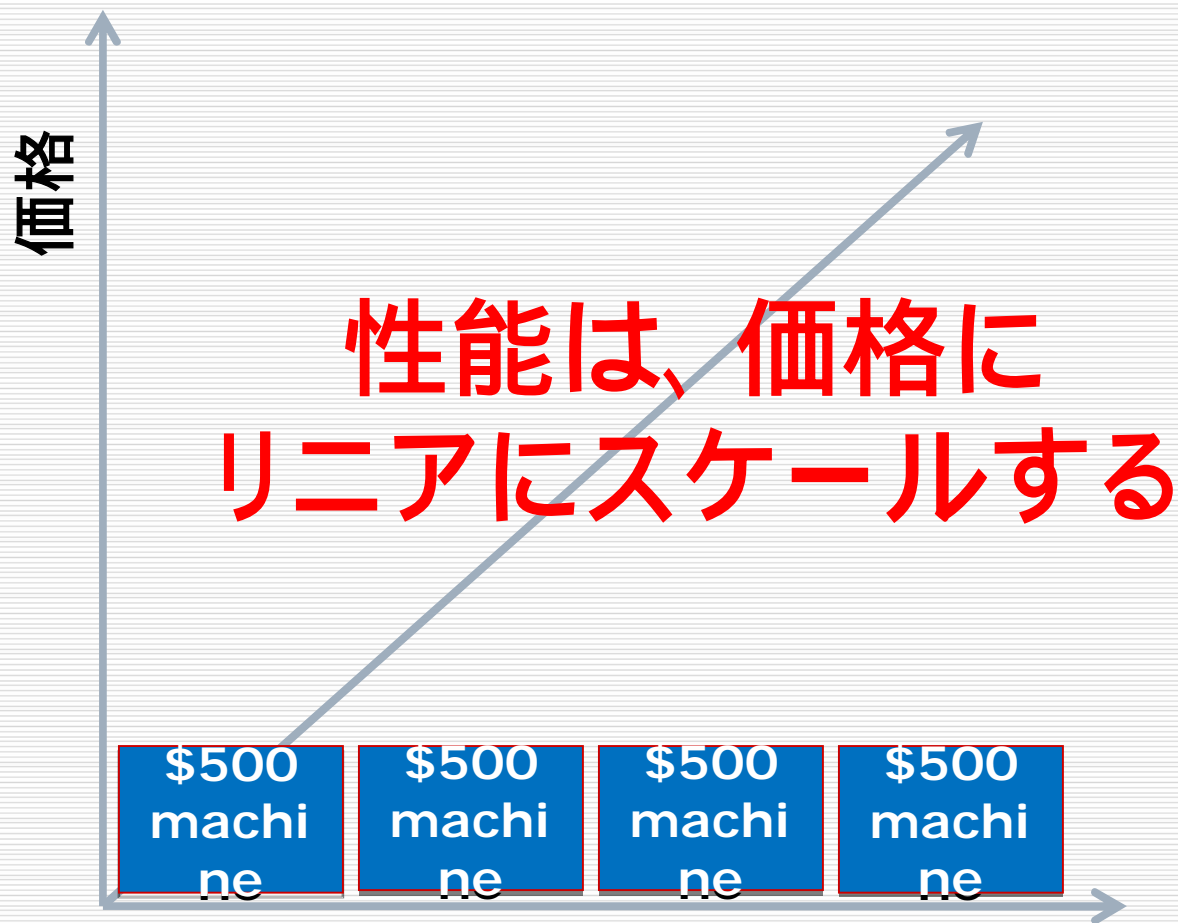
Scale-upからScale-outへ

---

# Scale-up と Scale-out



Scale Up



Scale Out

# Machines

# Scale-up:

---



# Scale-out

---



# Googleは、なぜ、安いIPCを使うのか？

## High-end SMP server

- HP Integrity Superdome
  - 64 1.5GHz Itanium2s
  - 128GB DDR DRAM
- Performance per CPU
  - 13K tpmC
- Price/CPU: ~\$42,000

## PC-class DP server

- HP ProLiant DL-360
  - 2 3.0GHz Pentium4 Xeons
  - 4GB DDR DRAM
- Performance per CPU
  - 26K tpmC
- Price/CPU: ~\$2,500

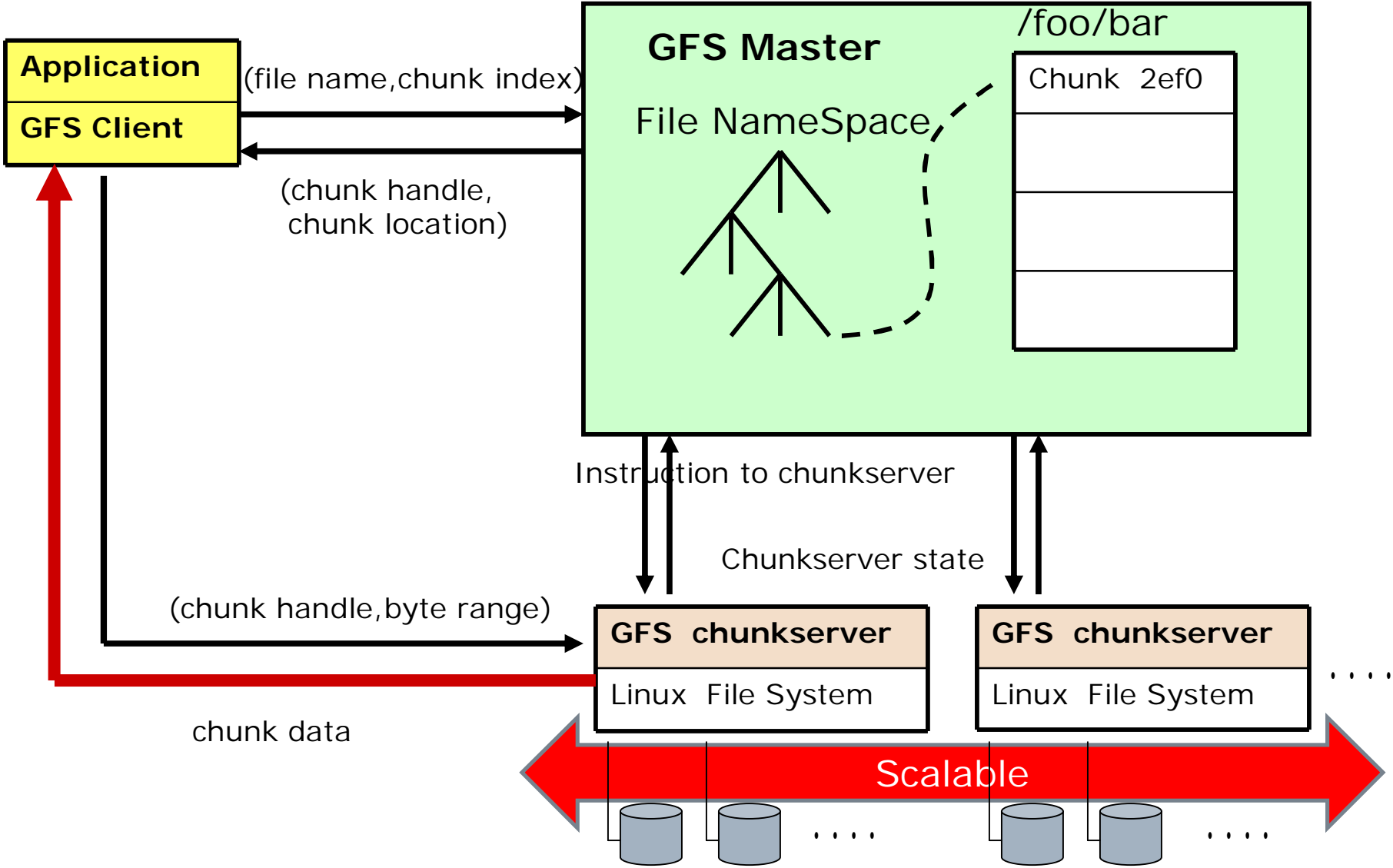
33倍ほど、PCクラスのサーバのほうが  
コストパフォーマンスがいい

# システムのScalability

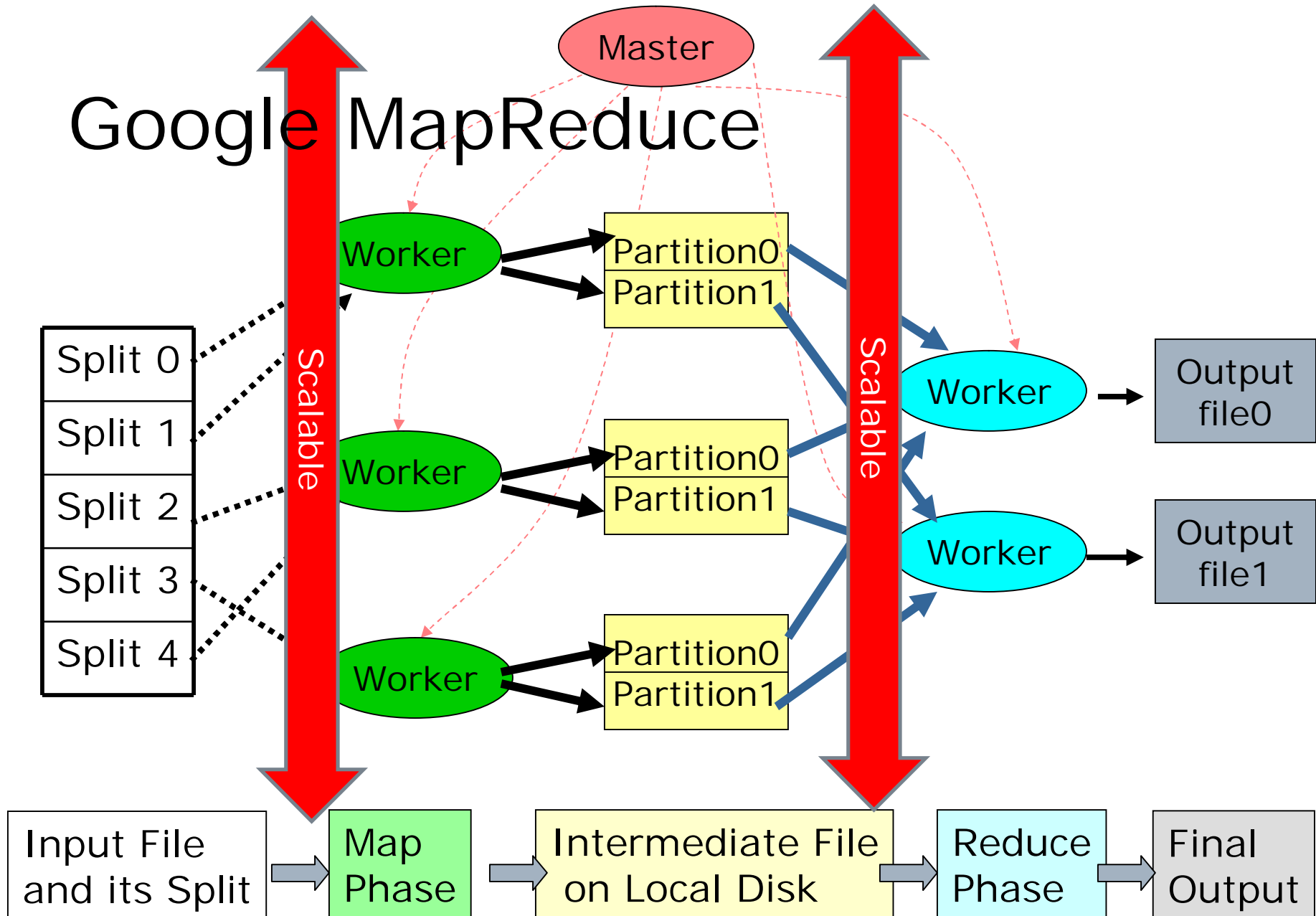
---

Scale-outについていえば、システムの「安さ」「コストパフォーマンスの高さ」だけでは十分ではない。クラウド・システムが、**Scalability**という、従来のシステムには欠けていた、新しい質を獲得していることが、より本質的で重要である

# Google File System



# Google MapReduce



# Google BigTable

Bigtable Client

Bigtable Client  
Library

Open

Bigtable セル

Bigtable Master

メタデータのオペレーションを実行  
ロードバランシング

Scalable

Bigtable  
Tablet Server

サーバデータ

Bigtable  
Tablet Server

サーバデータ

Bigtable  
Tablet Server

サーバデータ

Cluster  
Sheduling  
System

フェイルオーバのハンドリング  
モニタリング

Google  
File  
System

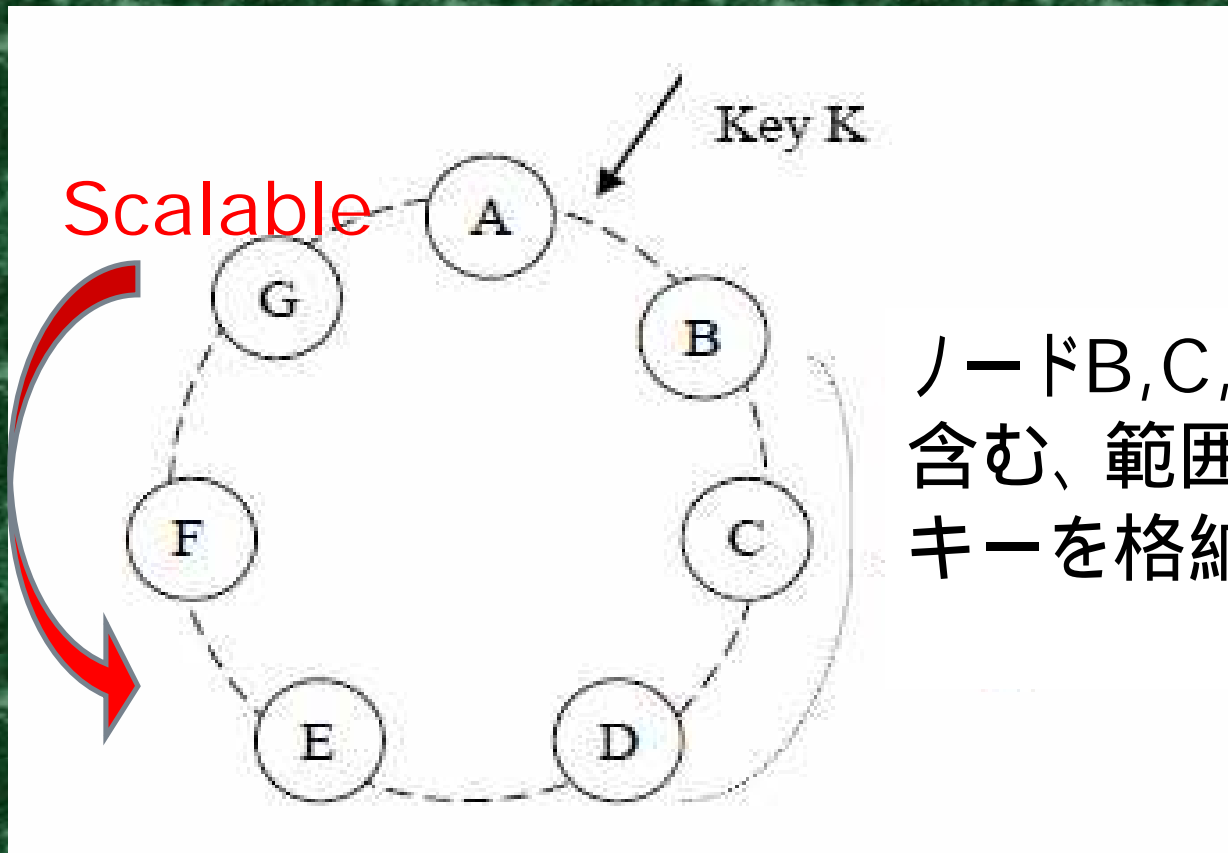
タブレットデータの  
保持 ログ

Chubby  
Lock  
Service

メタデータ保持、  
マスター選定のハンドリング

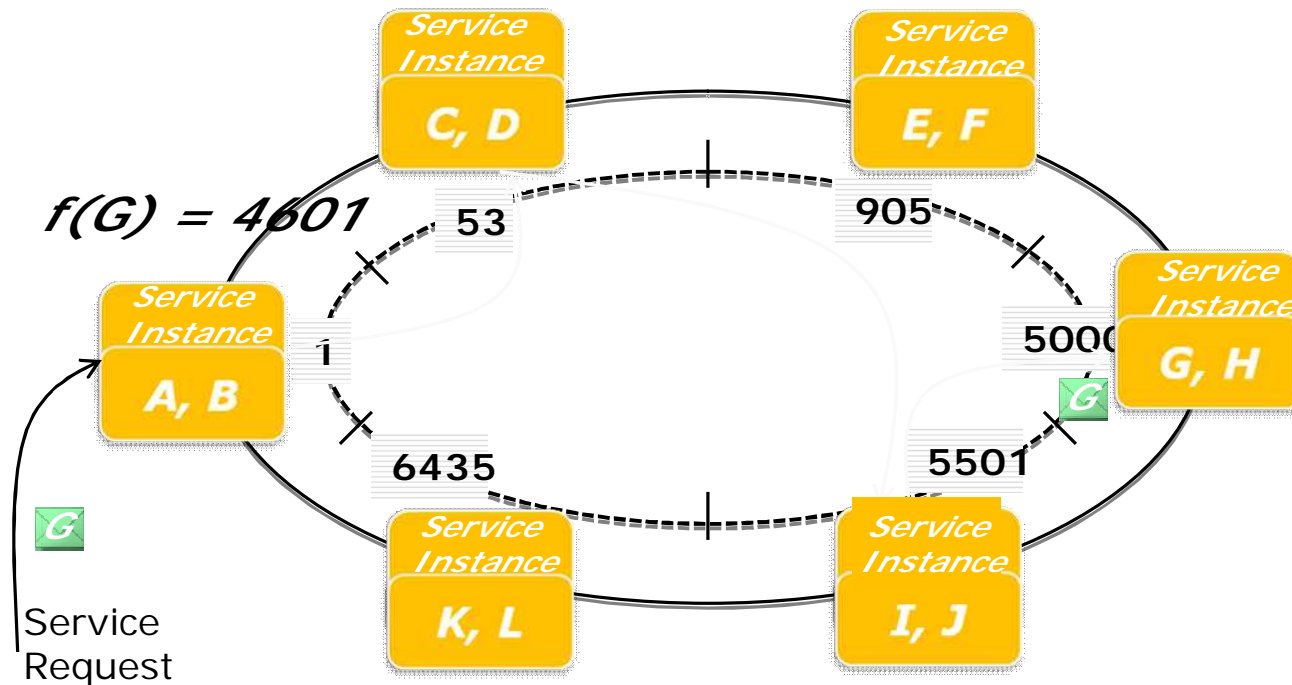
# Amazon Dynamo

## Consistent Hashing



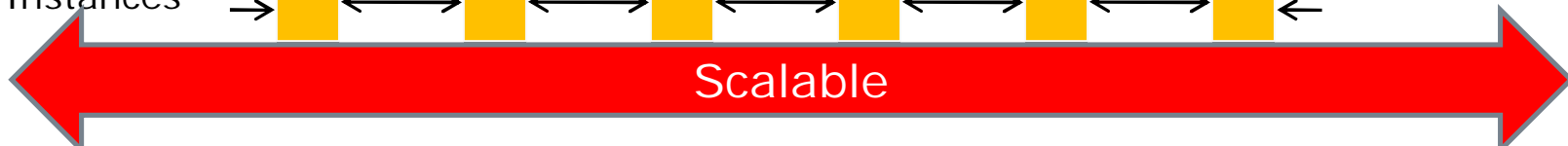
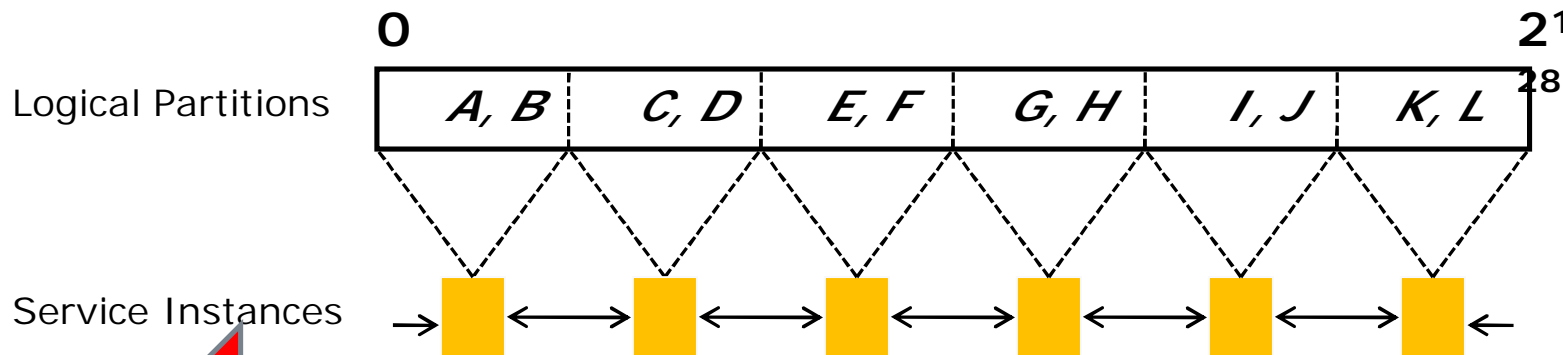
ノードB,C,Dが、Kを含む、範囲(A,B]のキーを格納する。

# Microsoft Azure SDS

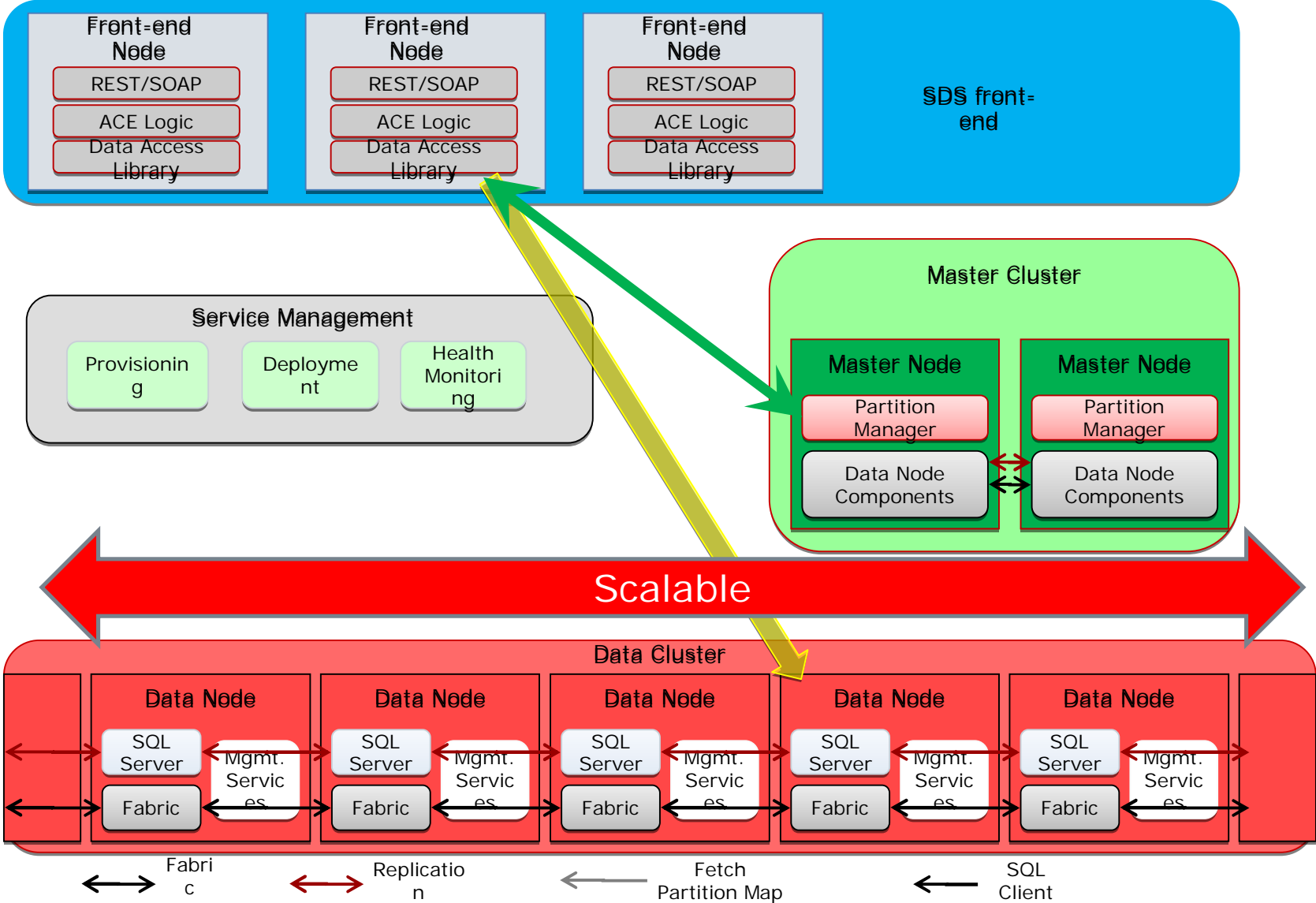


サービスは、エンティティからPartitionへのマッピングを持っている

Data Overlay の操作は、ノード上でローカルに行われる



# Microsoft Azure SDS



# システムのScalabilityという観点

---

# クラウドのタイプを考える

---

- クラウドが提供するサービスにも、Scalableなシステムが提供するサービスと、必ずしもScalableではないシステムが提供するサービスの二つのタイプがあることになる。
  - On Premiseのクラウドでも、この二つのタイプの区別は可能である。
-

# 仮想化とScalability

---

- あるシステムの能力は、その物理的な構成によって決定される。
- Scalabilityとは、いわば、このシステムの物理的な構成を動的に拡大・縮小する能力のことである。
- Virtualizationの技術は、そのシステムの物理的な構成の範囲で、その能力を柔軟に引き出すのには有効ではあるが、Scalabilityを保障するものではない。
- もちろん、Scalabilityの技術は、Virtualizationを必要とする。

# システムのScalabilityと ユーザ・サービスのScalability

---

- サービスの提供者としてのクラウド・システムの持つScalabilityと、サービスを受け取るユーザにとってのサービスのScalabilityとは、異なる概念である。両者を混同してはならない。
  - ユーザにとってScalableなサービスは、それ自身では、サービス提供者のクラウド・システムがシステムとしてのScalabilityを持つことを意味しない。
-

# システムのAvailability

---

# システム障害—Scale-outの新しい問題

---

- 3-year MTBFだとしても、1000台のうち一台は、毎日だめになるという計算になる。
  - 最小のGoogleのアプリケーションでも、2000台のマシンを必要とする。
  
  - こうした障害をソフトでどう対応するか？
  - データの多重化と冗長化は、この規模ではどうしても必要となる。
-

## Don't Confuse

本当に安いマシンで  
大丈夫か？



Commoditization of  
Computers

With



Commoditization of  
Computing

# システム障害についての Google流の考え方の一例

---

- だから、なぜ、高価な信頼性の高いハードのことで思い悩むのか？
- 信頼性の高いハードは、ソフトウェア技術者を怠け者にする
- 障害に強いソフトウェアが、安いハードを役に立つものに変えるのだ

Ben Jai, Google Platforms Architect  
「Googleはなにをしているのか？」より

# システム障害についての Cloudの考え方のポイント

---

- 沢山のマシンから構成されるシステムでは、障害は、確率的には必ず起きるものである。
- 障害が起きるのは、当然のことであるという前提にたってシステムを構成すること。

Cloud = Scalability + Availability

---

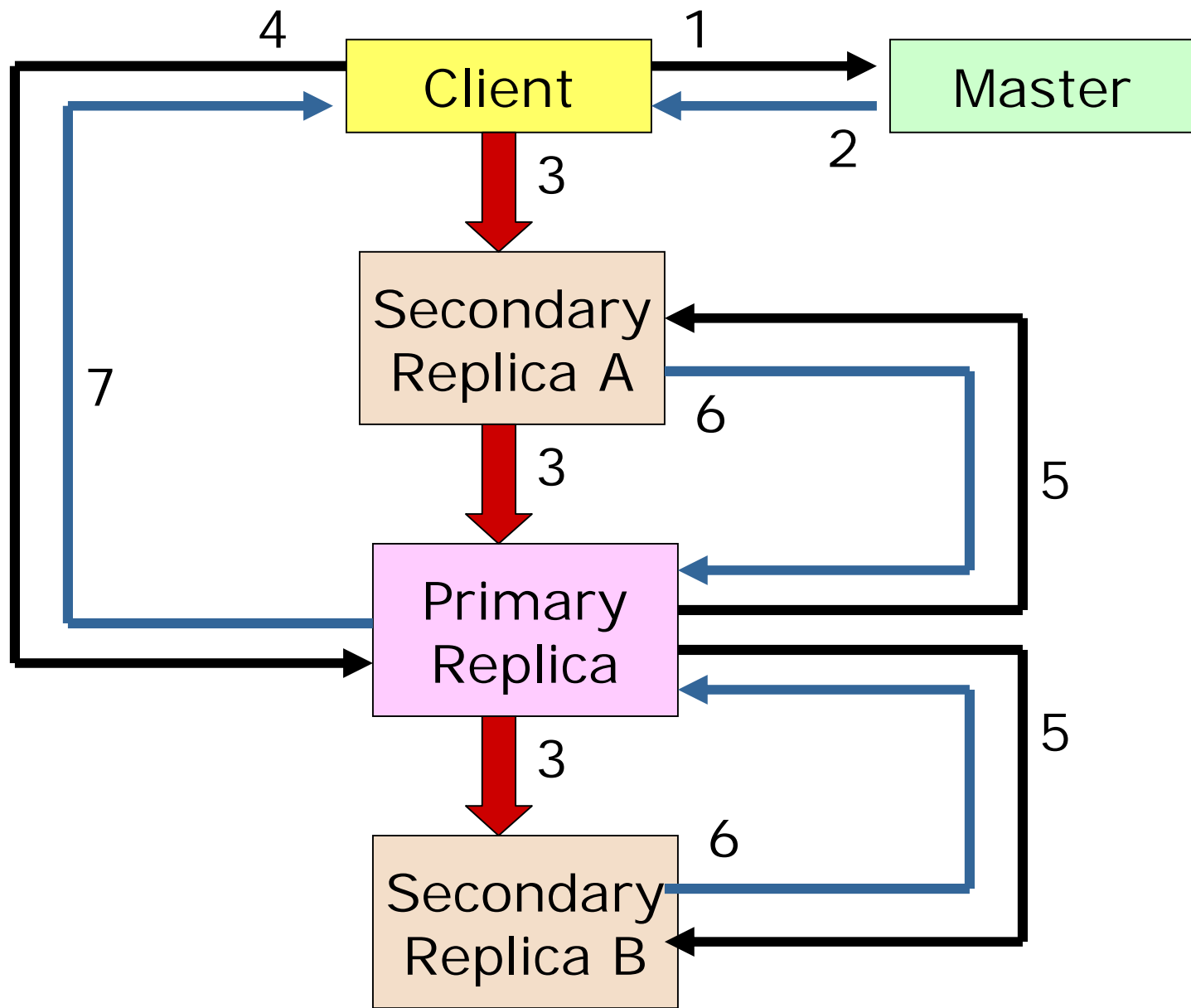
# Availabilityと多重化・Fail-Over

---

# Google File Systemの Availability

---

Google File SystemのAvailabilityは、基本的には、データを蓄える役割を果たすChunk Serverを多重化(通常は三重化)することによって支えられている。



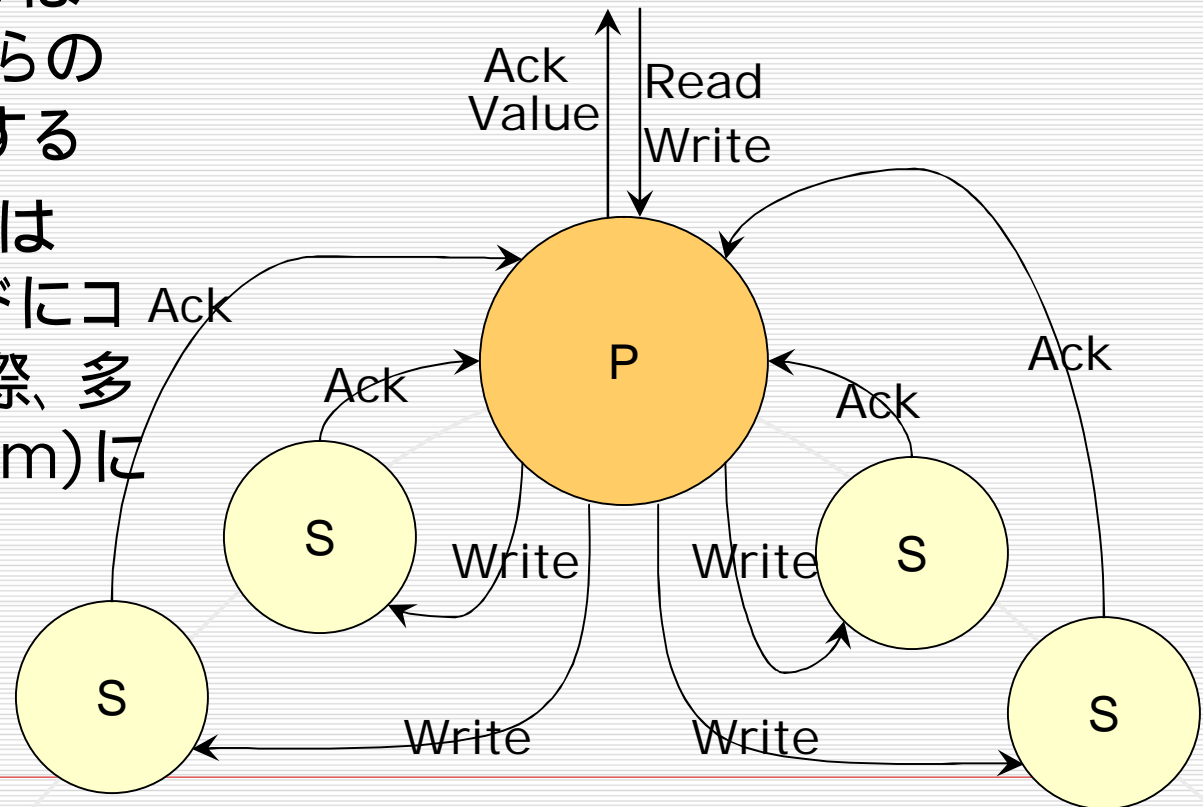
# Windows AzureのAvailability

---

Windows Azureでは、File SystemのレベルではなくData StorageのレベルでReplicaが導入されている。また、Fail-overについて、いくつかのシナリオが用意されているので、それを見ておこう。

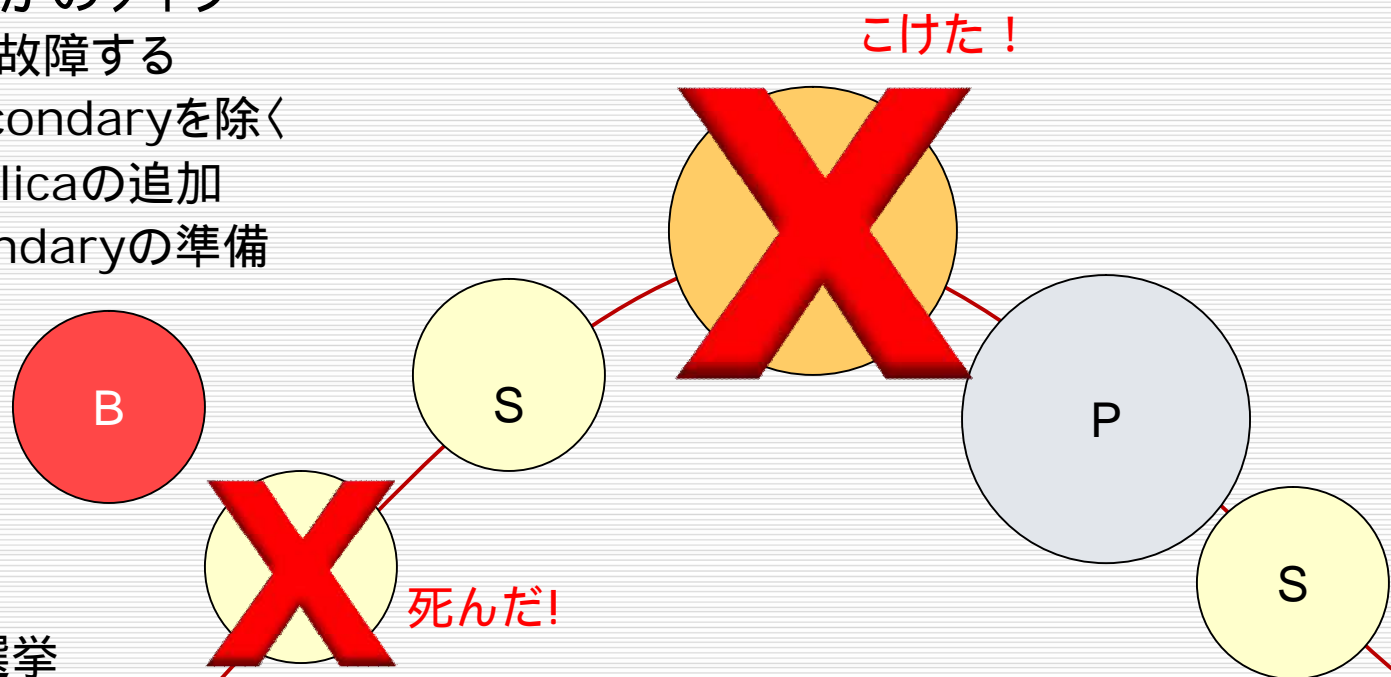
# MS Azureのデータノードの多重化

- データの読み込みは Primaryノードからの読み込みで完了する
- データの書き出しは Secondaryノードにコピーされる。この際、多数決原理(quorum)に従う。



# MS Azureのデータノードの再構成

- 再構成のいくつかのタイプ
  - Primary が故障する
  - 故障したSecondaryを除く
  - 修復したreplicaの追加
  - 新しいSecondaryの準備



- 前提
  - 故障の検出
  - リーダーの選挙

これらの故障が重複して起きても安全のように設計する

# Glassfish (アプリケーション・サーバ) でのFail-Overのシナリオ

---

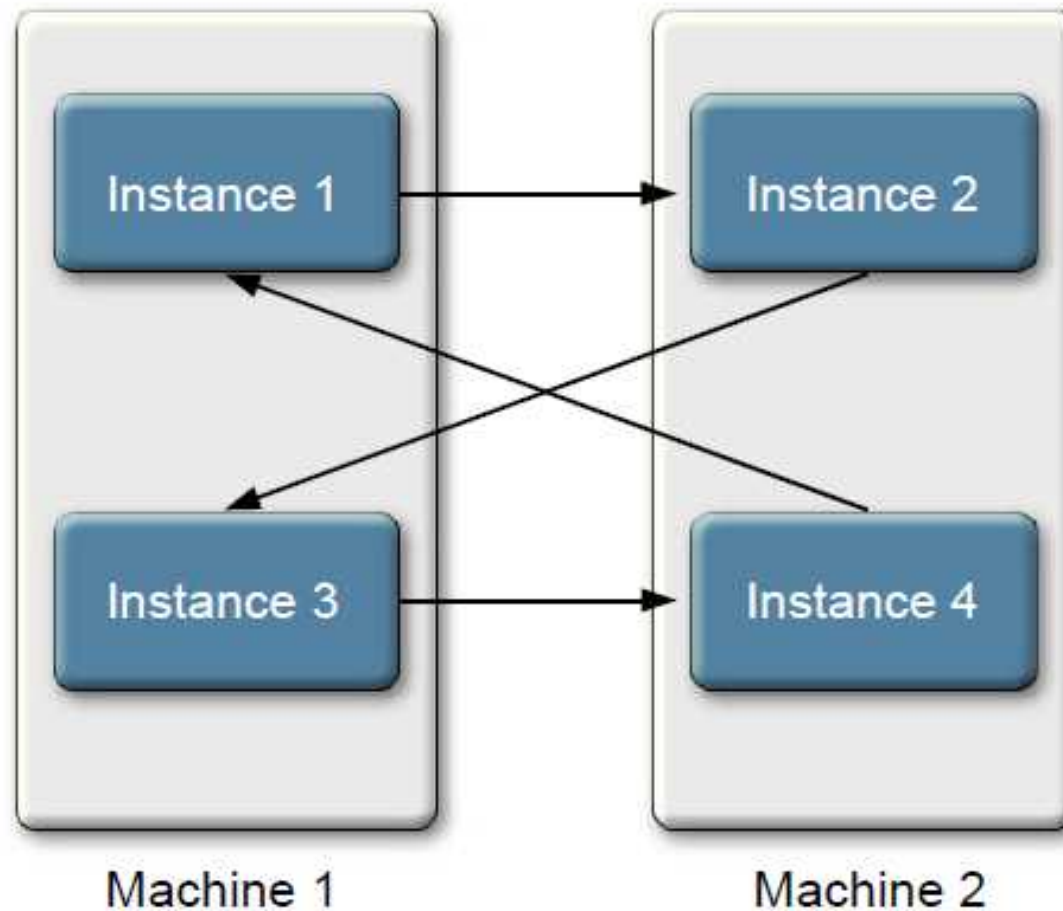
クラウド・システムが行おうとしていることは、現在、エンタープライズの基幹系のシステムが行おうとしていることと、ある意味、大きな違いはないのである。

こうしたシナリオが基幹系のエンタープライズ・システムで受け入れられるのなら、クラウド・システムのシナリオが、Consistencyへの脅威と受け止められることはないはずである。

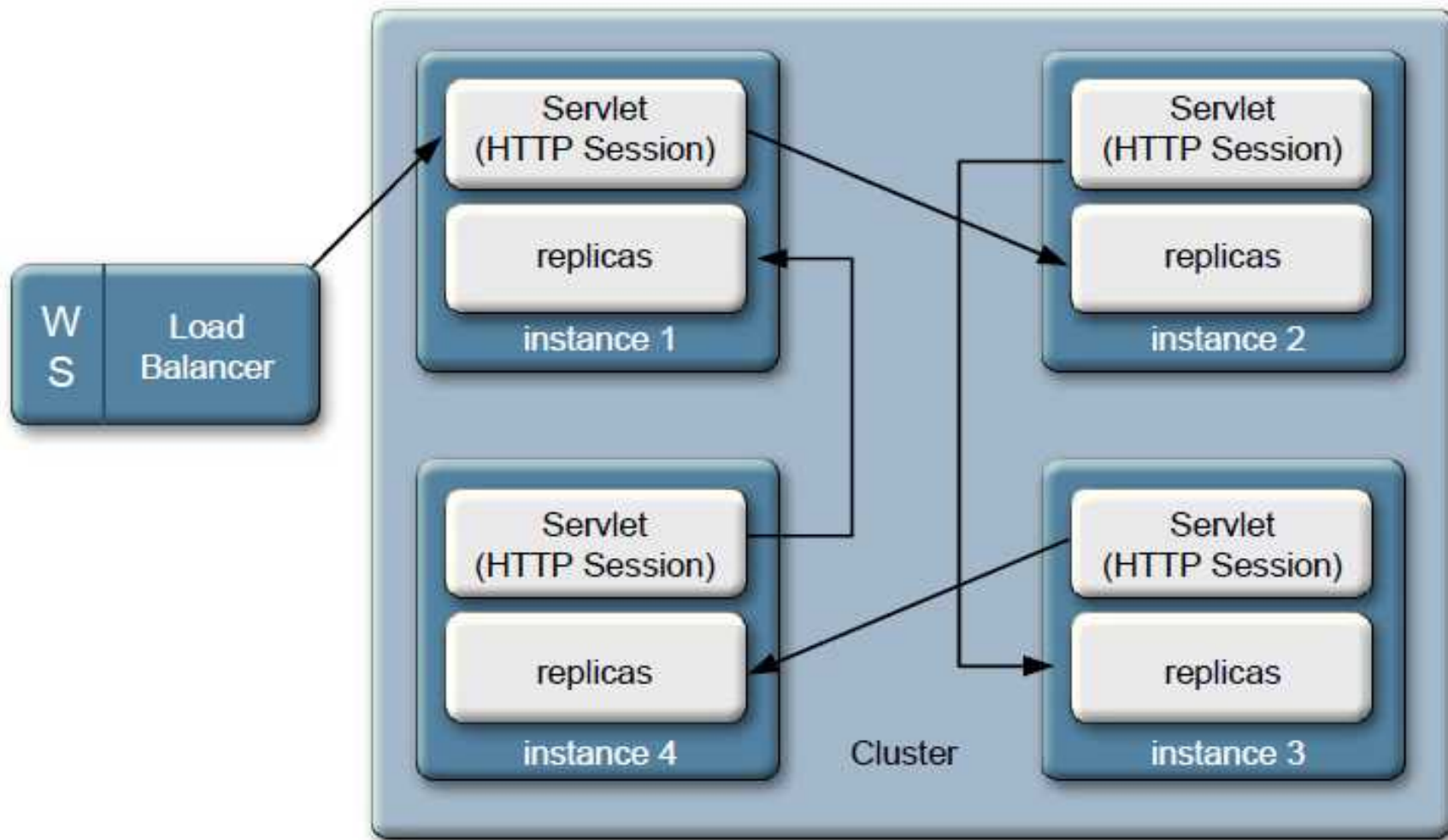
# 典型的なClusterのトポロジー

矢印は、Cacheのコピーの関係を表す

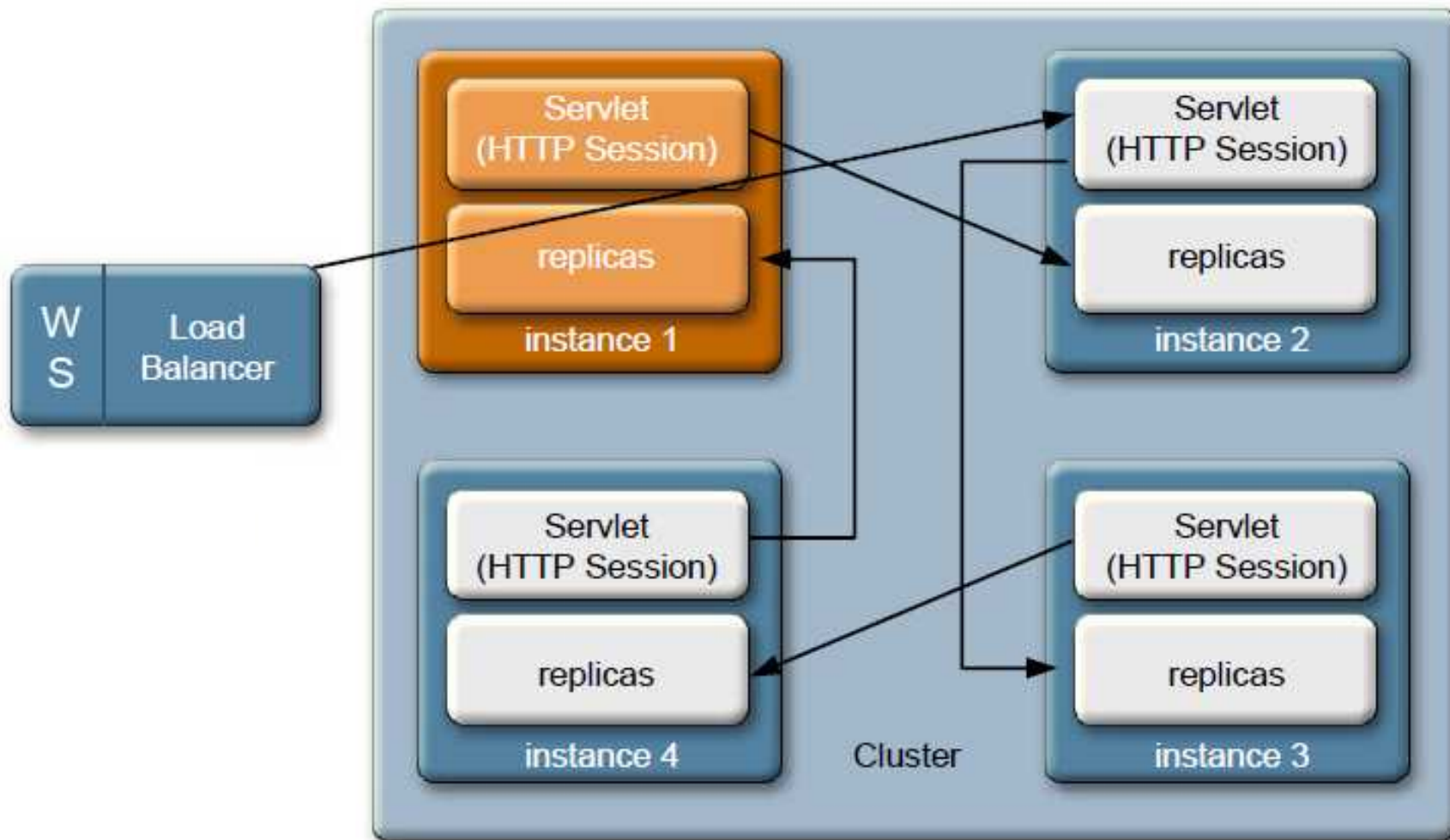
二台のマシン上の、  
4ノードのclusterの  
可用性を最大にする



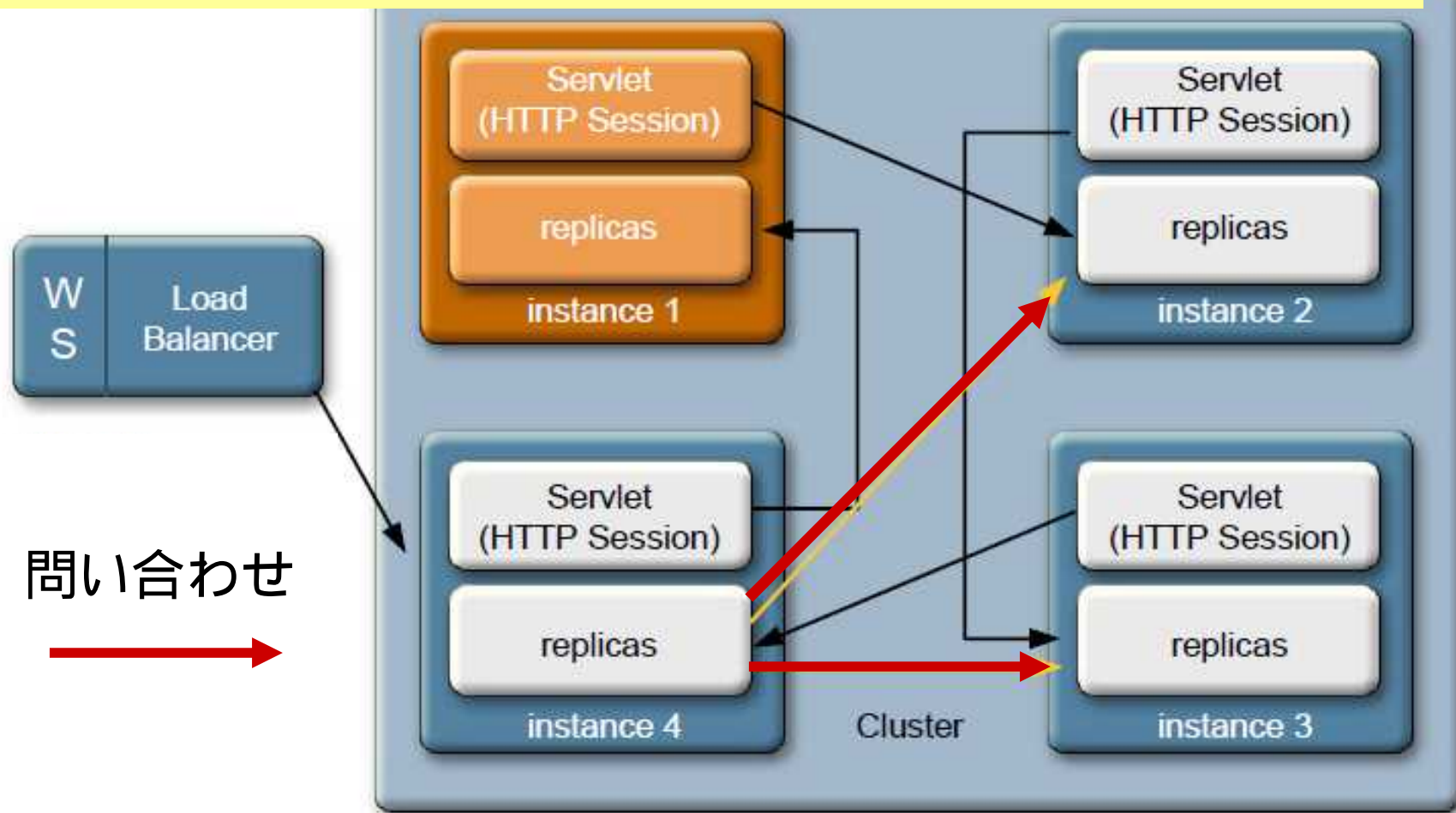
それぞれが、自身のCacheのほかにも、他ノードのCacheのコピー-replicaを持っている



ノード1に障害が起きたとしよう。制御がノード2に移るなら、ノード2には、ノード1のCacheのコピーがあるから、それを利用できる



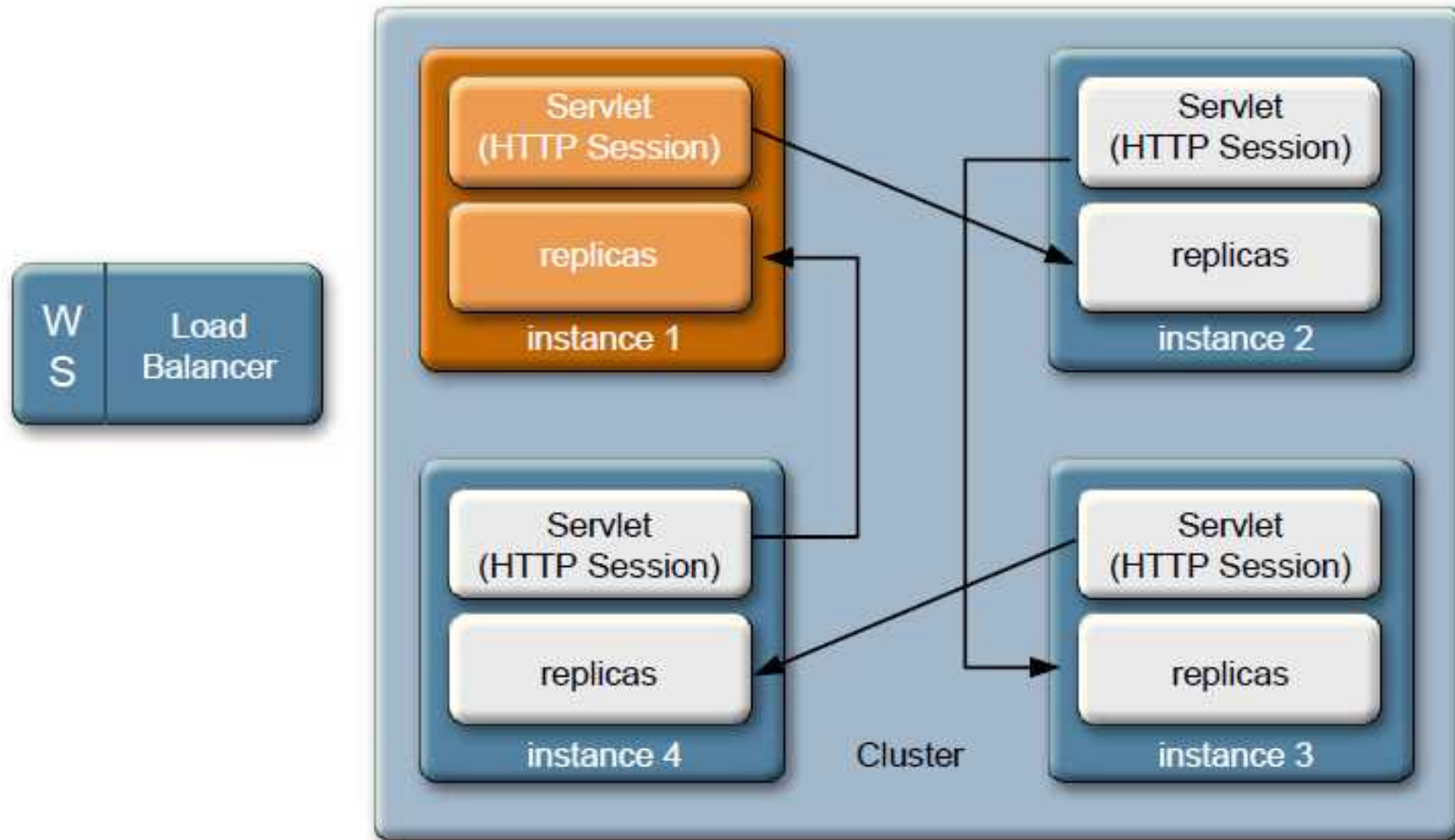
今度は、制御がノード3に移る場合を考えよう。  
ノード3には、ノード1のCacheのコピーはないので、  
他のノードにノード1のコピーがないか問い合わせる





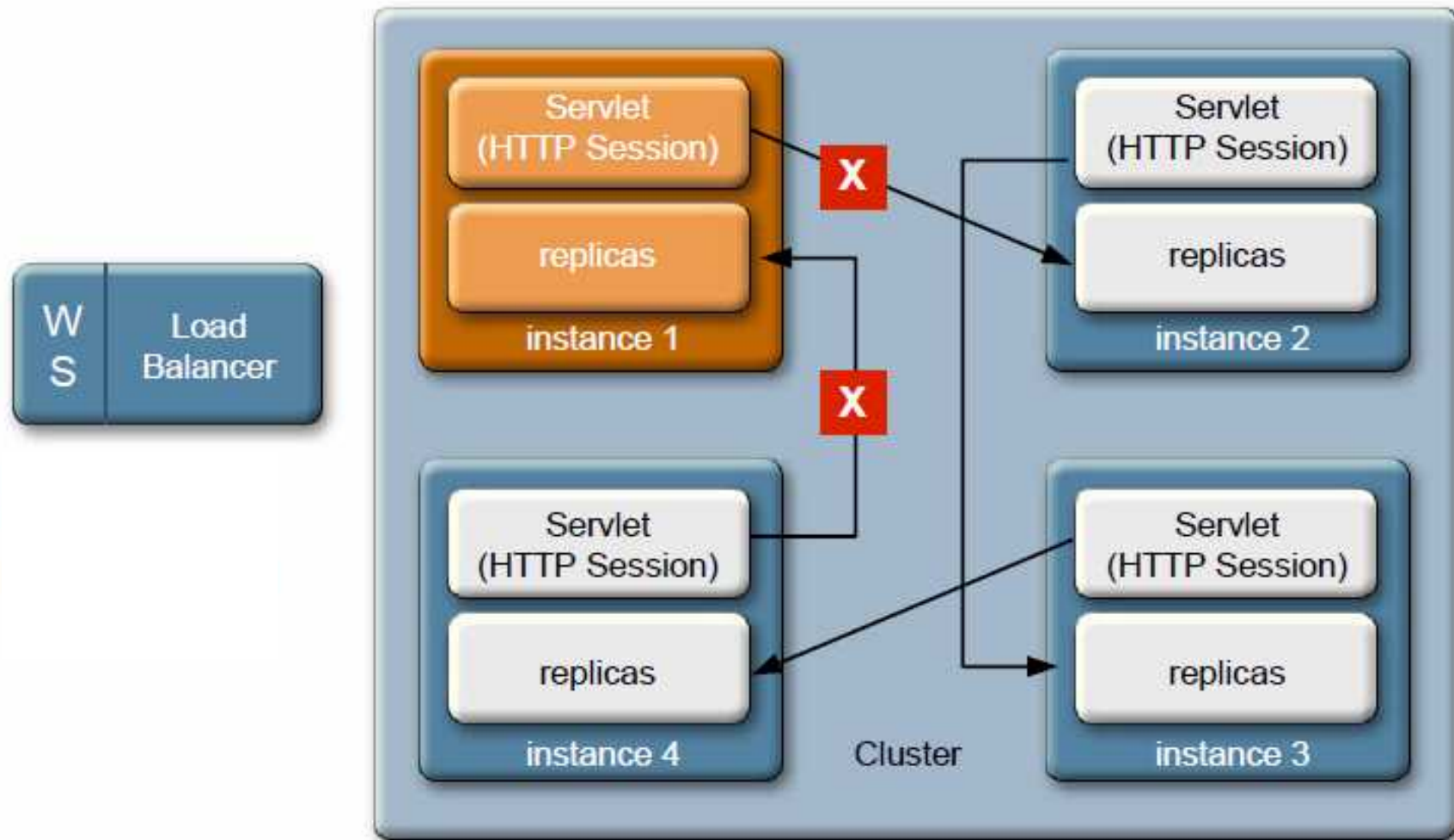
# Clusterの形を、動的に変化させる (1)

## 障害発生時の状態

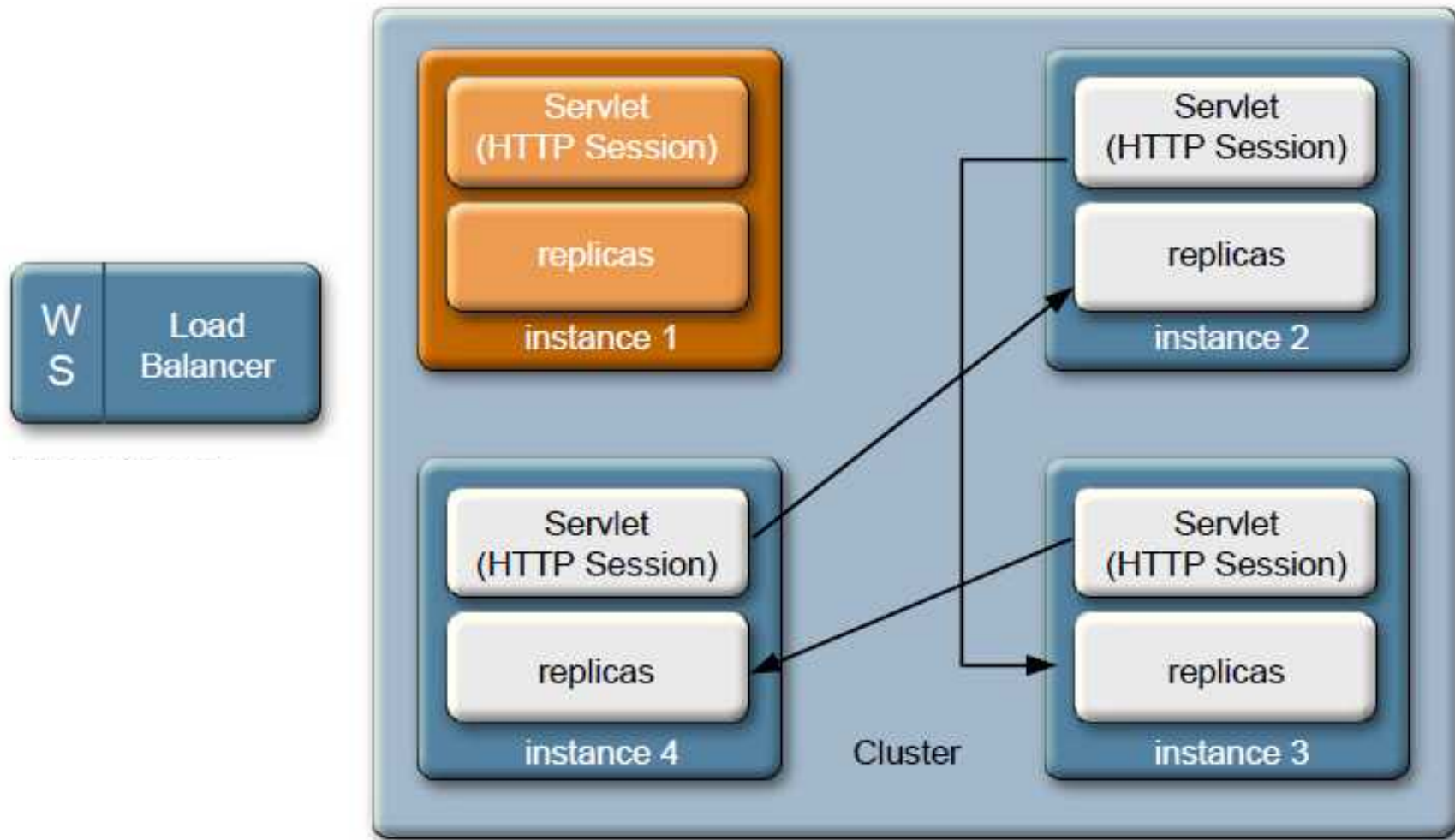


# Clusterの形を、動的に変化させる (2)

## 障害ノードの切り離し



# Clusterの形を、動的に変化させる (3) 参照するレプリカの切り替え



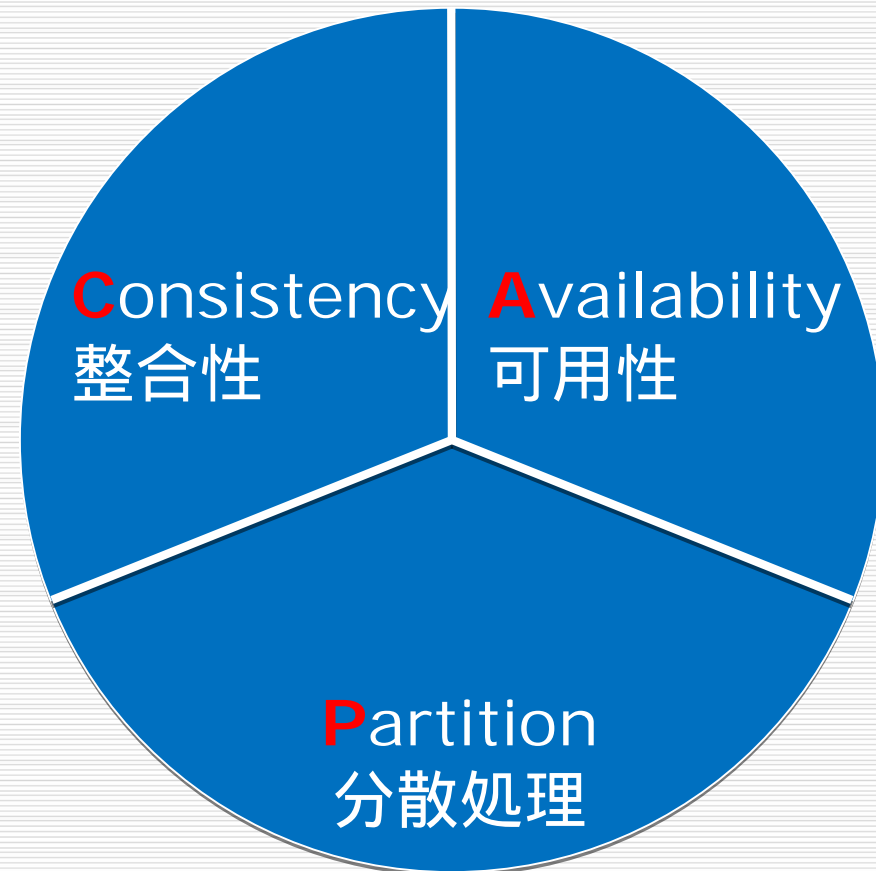
# CAP Theorem

---

# CAP定理とは？

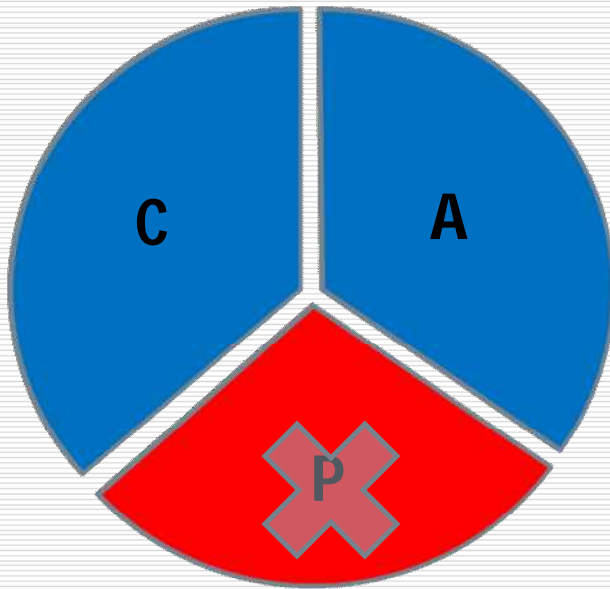
---

システムのC,A,Pのうち、同時には、二つまでしか満たすことは出来ないという主張



## CAP定理

整合性と可用性をとると、分散処理は出来ない。

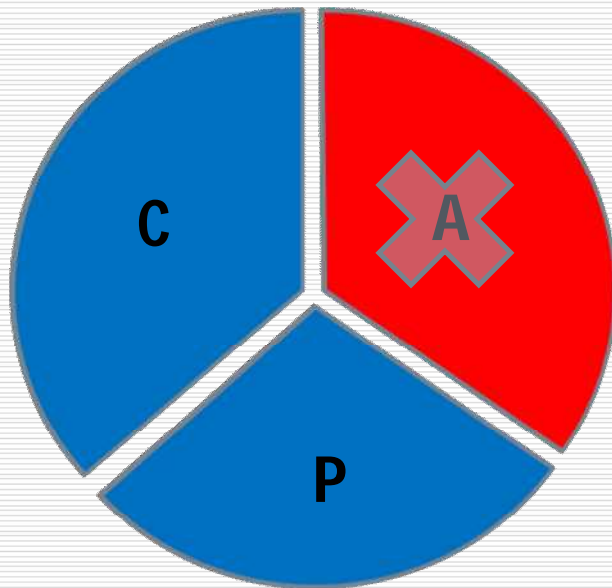


- Consistency + Availability
- 単一サーバ

## CAP定理

整合性と分散処理をとると、可用性は失われる。

---

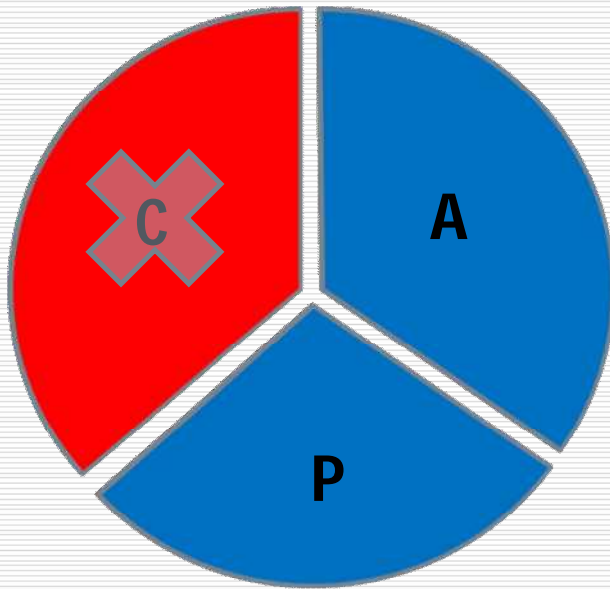


- Consistency + Partition
  - 分散 database / 分散ロック
-

## CAP定理

分散処理と可用性をとると、整合性が失われる。

---

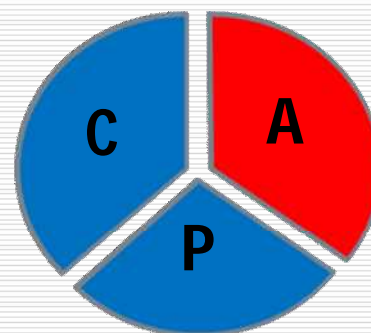
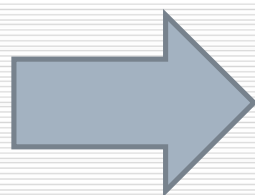
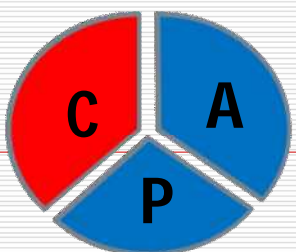
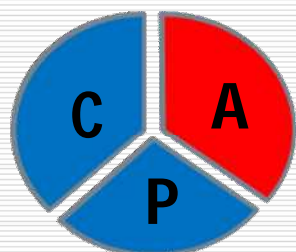
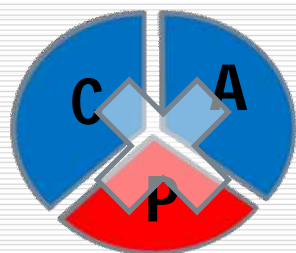


- Availability + Partition
  - 分散キャッシュ / DNS
-

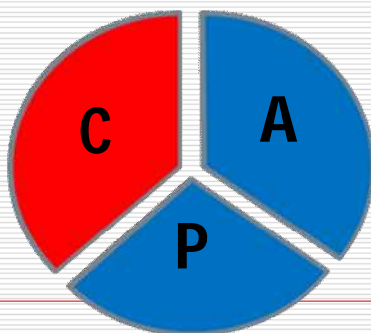
# Cloudでの二つの可能な選択

---

- Cloudでは、分散処理は必須である。



整合性を  
取るか?



可用性を  
取るか?

# 可能な二つの選択を考える

---

この選択はシステムにとっての二者択一と考えるべきではない。同じ、クラウド・システムで稼働するアプリケーション、あるいは、そこでハンドルされるデータに応じて、Availabilityを取る方がいいかConsistencyを取る方がいいか分かれると考えた方がいい。

あるe-サイトのショッピング・カート为例に、この問題を具体的に考えてみよう。

## e-サイトで、ショッピングをしている間

---

- ショッピングをしている間は、反応がすぐ返って動くのがユーザには使いやすい。
  - カートに一つの商品を入れたとあって、注文が確定したわけではない。カートに商品を入れるたびに、在庫に問い合わせ、データベースに注文受付中のフラグを書き込む必要はない。それは、システムの負荷を高め、ユーザの使い勝手を悪くするだけである。
  - ここでは、Availabilityが重要である。
-

## ユーザの注文が確定してから

---

- 注文は、在庫データベースとつき合わされ正確に処理されなければならない。
  - もしも、在庫データベースがすぐに応答しなかったり、他の問い合わせによってロックされているなら、応答が返るまで待てばいい。
  - ここでは、Availabilityより、Consistencyが重要なのである。
-

# 出荷の処理が終われば

---

- この取引に関するデータは基本的には変更されることはない。
  - Read-Onlyで、たいていは、Primary-Keyで参照されるだけである。
  - こうした処理パターンは、クラウド向きで、高速な処理が可能となる。
  - こうした、クラウドに適した処理としては、地図情報やブログのエントリーや商品カタログの参照などがあげられる。
-

# クラウドは、 基幹業務に向かないという誤解

---

- クラウドにも得意・不得意があるという常識的な認識は、典型的には、「コンシューマ向きには、クラウドのサービスは向いている」  
「しかし、エンタープライズの基幹業務には、クラウドは向かない」という認識とオーバーラップしてはいないだろうか？
- こうした認識は、クラウドで先行したGoogleがもっぱらコンシューマ向けのサービスを展開していて、他方では、エンタープライズ向けのクラウド・サービスが始まっていない現状を反映しているだけである。

# 基本的なソリューション

---

- 基本的なソリューションは、先のe-サイトの例でも見たように、扱うデータとその処理のタイプに応じて、クラウド上のアプリケーションの側が、Availability優先か、Consistency優先かで、クラウド・システムのトランザクションのタイプを選択していくことに帰着する。
  - クラウドが、コンシューマ向けと、エンタープライズ向けの二つのタイプに分かれると考えるのは、必ずしも、十分な認識ではない。
-

# ScalabilityとAvailabilityの両立が Consistencyに与える影響

---

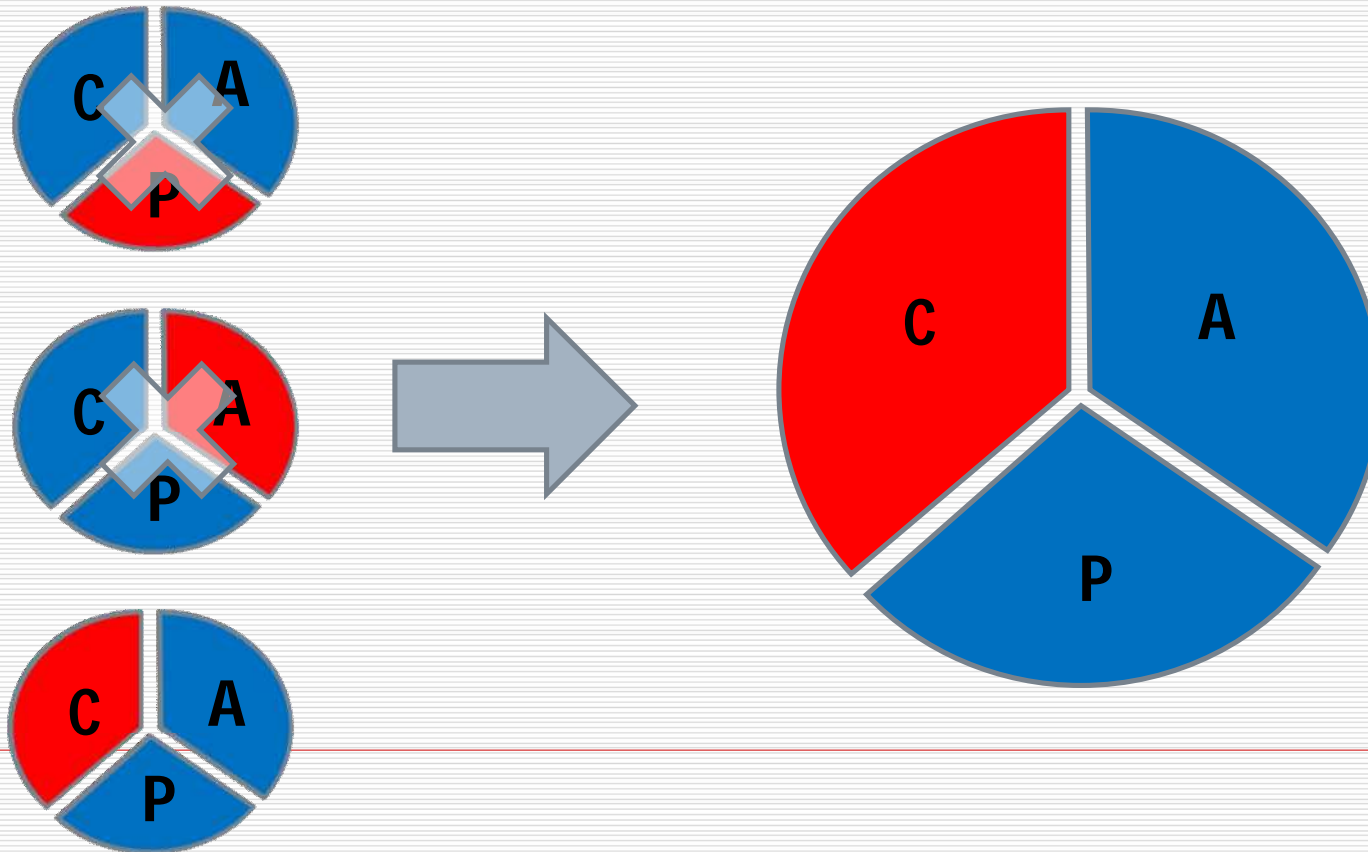
CAP定理の、ScalabilityとAvailabilityを確保すると、Consistencyが損なわれるという主張をもう少し詳しく検討してみよう。

Cloudのエンタープライズ利用の中心問題

# CAP定理のもう一つの帰結

---

- Cloudでは、分散処理と可用性は必須である。



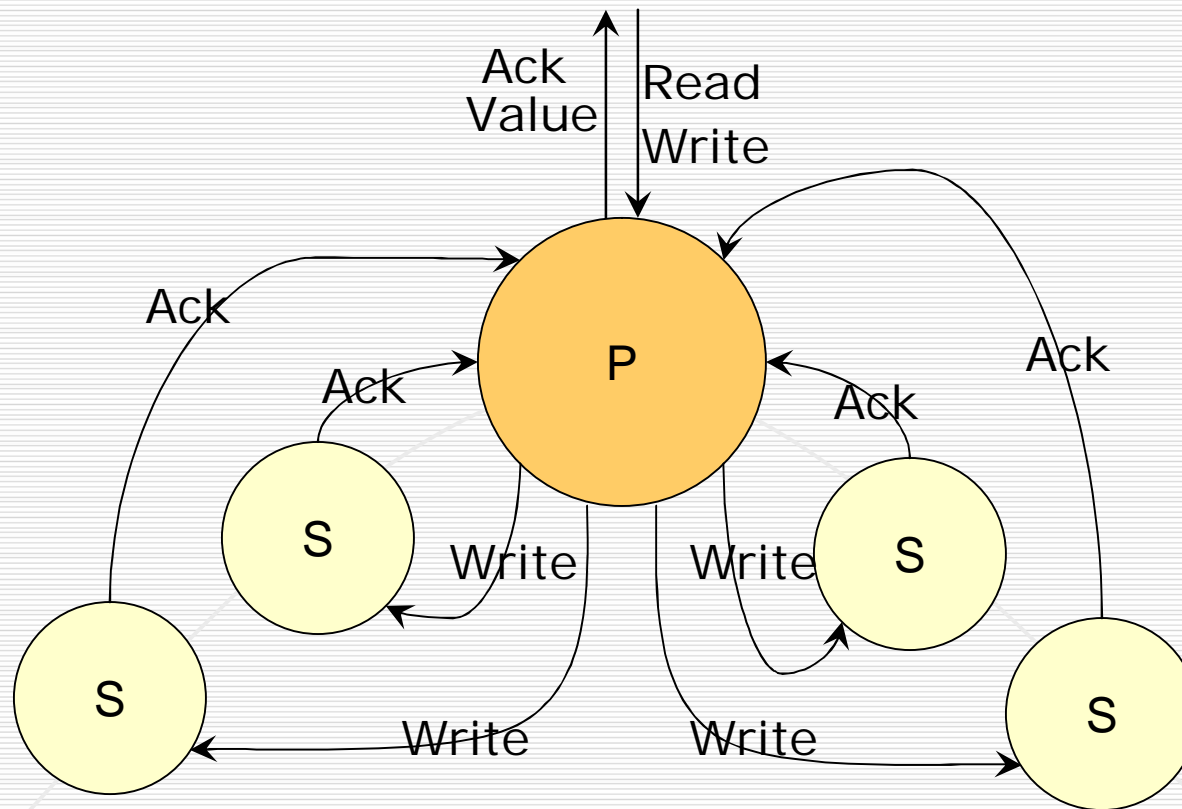
# Consistencyに問題が起きるケース

---

- 現在、主流のクラウド・システムの構成をみると、ScalableでAvailableなクラウド・システムがConsistencyにインパクトを与える状況というのは、基本的には、Availabilityを支える複数のレプリカ・ノードに同一のデータを送ろうとする時に生ずることが分かる。
- あるノードでは書き込みに成功しても、他のノードでなんらかの事情で書き込みに失敗すれば、システム内に異なるデータの状態が併存し、Consistentでない状態が生まれる。

# Azureの多重化で考える

---



# Azureの多重化で考える

---

- 多数決原理でいったん制御を戻しても、制御マシンは、失敗したノードに再書き込みを試みるだろう。
  - 再書き込みが成功すれば、それでいいし、どうしても書き込みができないなら、このノードをレプリカプールから削除する。
  - いずれにしても、一定時間後には、レプリカプール内のデータのConsistencyは回復される。
-

# Azureの多重化で考える

---

- Consistentではない状態のデータが、外部に出ていく可能性はないだろうか？
  - これも大丈夫である。データの書き込みには、複数のレプリカへの書き込みが必須なのに対して、データの読み出しには、Primaryノード一個からの読み出ししか行われない。
  - 矛盾したデータが、外に出ていく心配はない。
-

# Eventually Consistency

---

# Eventually Consistencyとは？

---

- システム内に、一時的にConsistentでない状態が生まれても、ある期間の後には、Consistentな状態になるような性質を、Eventually Consistencyという。
  - タイムスパンは違うのだが、前にあげたグローバルなシステムとしてのDNSシステムは、Eventually Consistentなシステムである。
-

# Consistency概念の緩和の意味

---

- Consistencyの概念を、Eventually Consistencyにまで緩めれば、CAP定理の主張に対して、次のような命題を対置することができる。
  - 「ScalableでAvailableで、かつ、Eventually Consistentなシステムは可能である。」
  - そして、この命題の実現こそが、現在のエンタープライズ向けのクラウド・システムが目標としているところなのである。
-

# BASEトランザクション

---

Consistencyの見直しだけでなく、従来のデータベースのトランザクションの基本であった、ACID特性 (Atomic, Consistent, Isolation, Durable) を相対化して見直そうという動きが出ていることを紹介しよう。

# OSとデータベースの融合

---

- 従来は、データベースとOSとは、明確にことなる存在であった。データの格納場所として、従来のOSが責任を持つのは、そのファイルシステムまでである。
  - では、OSとファイルシステムとの結びつきは、必然的なものであろうか？ 多分、その結びつきは必然的というより、歴史的なものである。
  - 筆者は、クラウドでは、OSとデータベースの機能が融合を始めているのだと考えている。
-

# BASE概念

---

- Basically Available
- Soft-State
- Eventually Consistent

- ここでは、ACIDが、個別のデータベースのトランザクションの特性であるのに対して、BASEは、データベース機能を含んだシステム全体の特性であることに注意しよう。
  - BASEの概念は、実は、個別のテーブルがACIDを満たすように処理されることを排除してはいない。
-

# Soft-State

---

- あるノードの状態は、その内部に埋め込まれた情報によって決まるのではなく、外部から、送られた情報によって決まるという状態の考え方。
  - あるノードの状態が、いったん、失われても、定期的に状態情報を取得すれば、状態は復元される。
  - ネットワークのPartial Failureの問題への対応として有効。
-

# ACID Transactionの例

---

個人台帳	取引台帳
名前	取引ID
売上総額	売り手名
購入総額	買い手名
	金額

Begin Transaction

```
insert 取引台帳(10001, '丸山', '植田', 100)
```

```
update 個人台帳 set 売上総額 = 売上総額 + 100 where 名前 = '丸山'
```

```
update 個人台帳 set 購入総額 = 購入総額 + 100 where 名前 = '植田'
```

End Transaction

---

# 思考実験



個人台帳

名前

売上総額

購入総額

二つのテーブルが、非常に遠く離れていたと考えてみよう。

取引台帳

取引ID

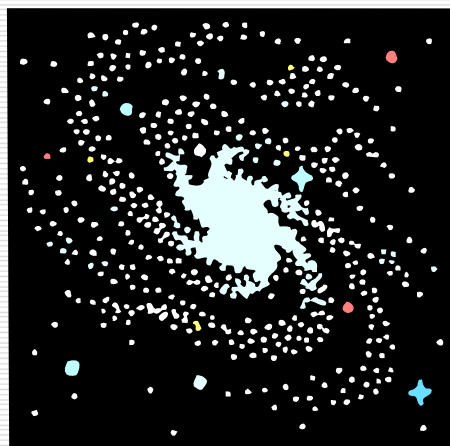
売り手名

買い手名

金額



# 思考実験



個人台帳

名前

売上総額

購入総額

あるいは、もっと。

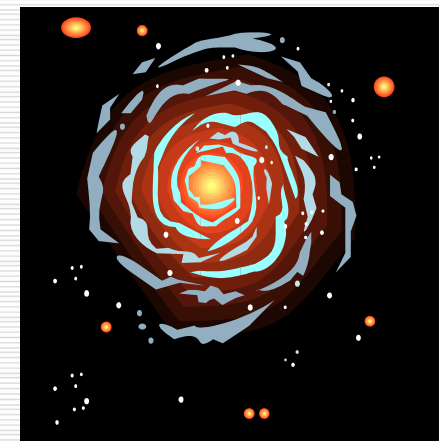
取引台帳

取引ID

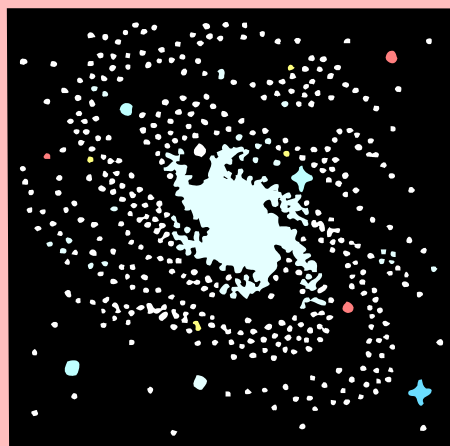
売り手名

買い手名

金額



# ACIDの想定だと難しいこと (可能かもしれないが)



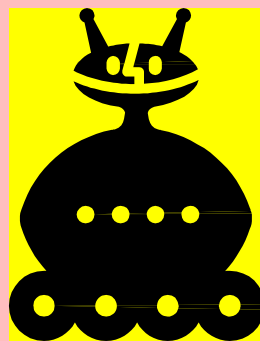
個人台帳

名前

売上総額

購入総額

## ACID



全体を見通す  
Transaction  
Managerが  
必要

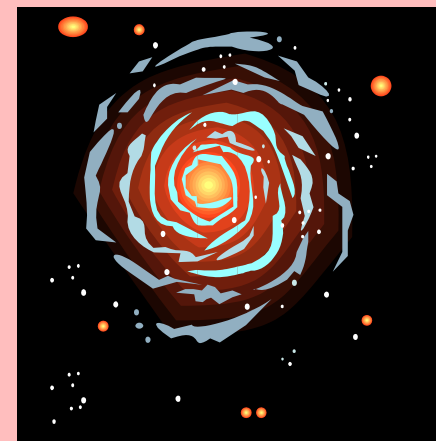
取引台帳

取引ID

売り手名

買い手名

金額



## もっと、簡単な方法がある

---

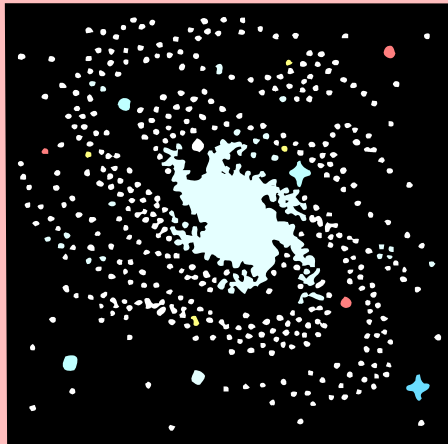
- 取引台帳の取引の挿入をローカルに、ACIDで行う。
- その後、その情報を送り出す。
- 個人台帳に情報が届くまでの間、Consistentではないと言える。



# Soft-Stateと Eventually Consistent

---

## ACID



個人台帳

名前

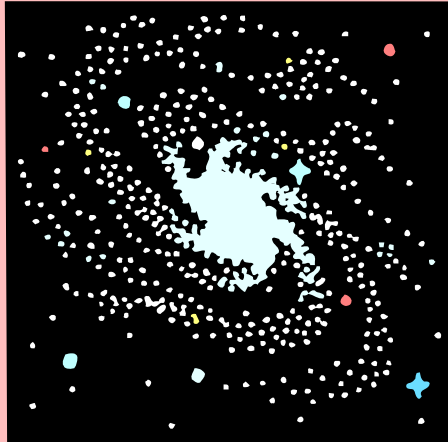
売上総額

購入総額

- 情報が個人台帳に届いたら、ACIDで、テーブルの書き換えをする。この状態変化は、**Soft-State**と考えられる。
- 情報が正確に到達すれば **Eventually Consistent**といえる。

# 確実な情報伝達路とSoft-Stateと Eventually Consistent

## ACID



個人台帳

名前

売上総額

購入総額

全体を見通す  
Transaction  
Managerは  
要らない

## ACID

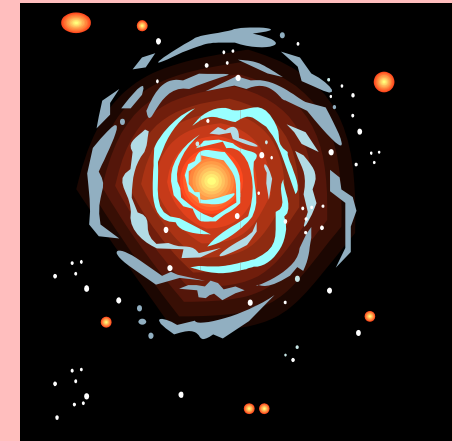
取引台帳

取引ID

売り手名

買い手名

金額



# 確実な情報伝達路とSoft-Stateと Eventually Consistent

---

## ACID

取引台帳

取引ID

売り手名

買い手名

金額

全体を見通す  
Transaction  
Managerは  
要らない

## ACID

個人台帳

名前

売上総額

購入総額

## BASE

Transaction

---

# 情報システムの原理としてのSoft-StateとEventually Consistent

---

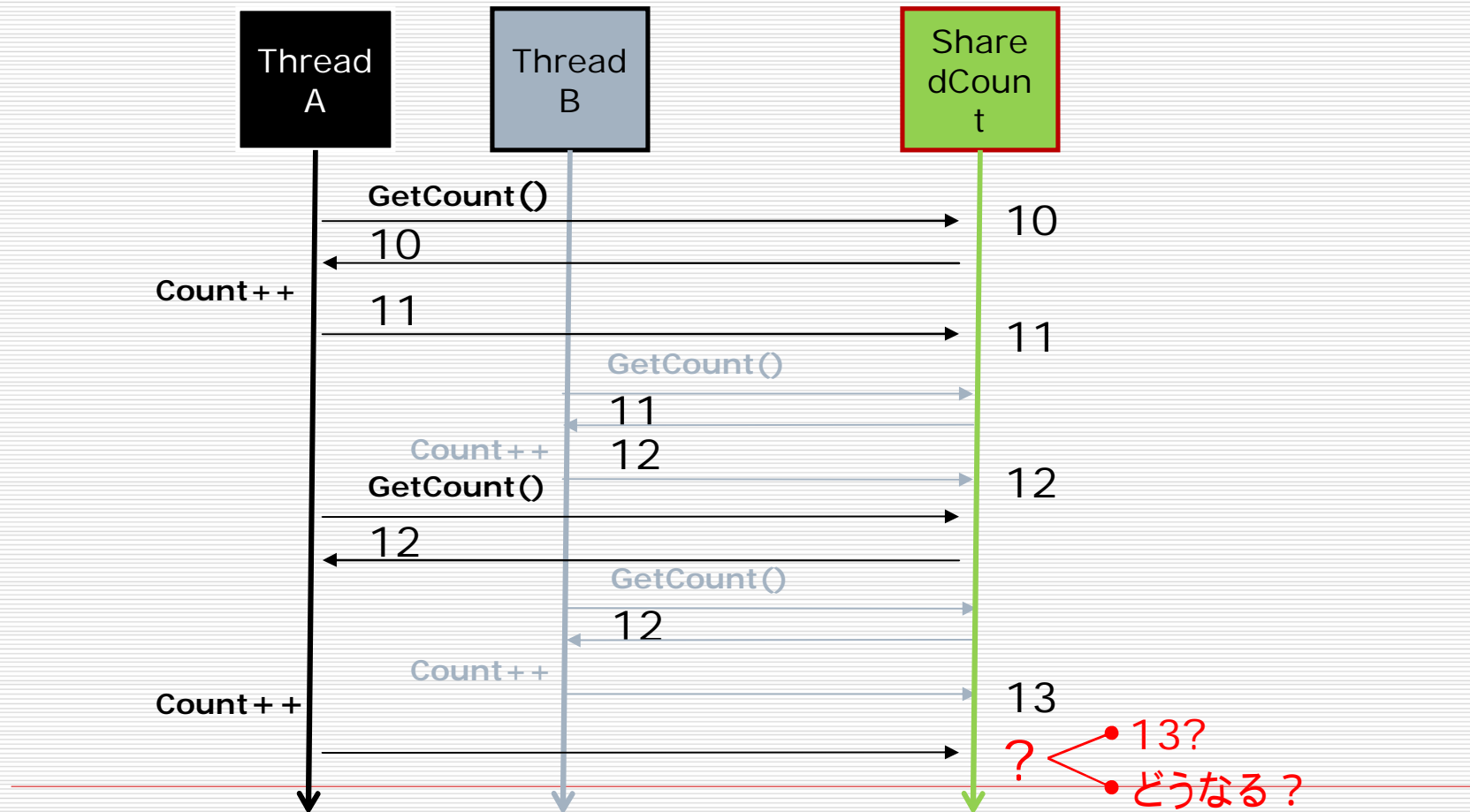
- もう少し、一般的に、あるノードの状態変化が他のノードの状態変化を引き起こすとしよう。
- この時、「二つのノードのConsistency」という概念は、原理的には、Soft-Stateに基づく、Eventually Consistencyでしかありえないことに注意しよう。
- Eventually Consistencyは、便宜的にConsistency概念を緩めたものではなく、むしろ、Consistency概念の基礎原理なのである

# Basically Availability

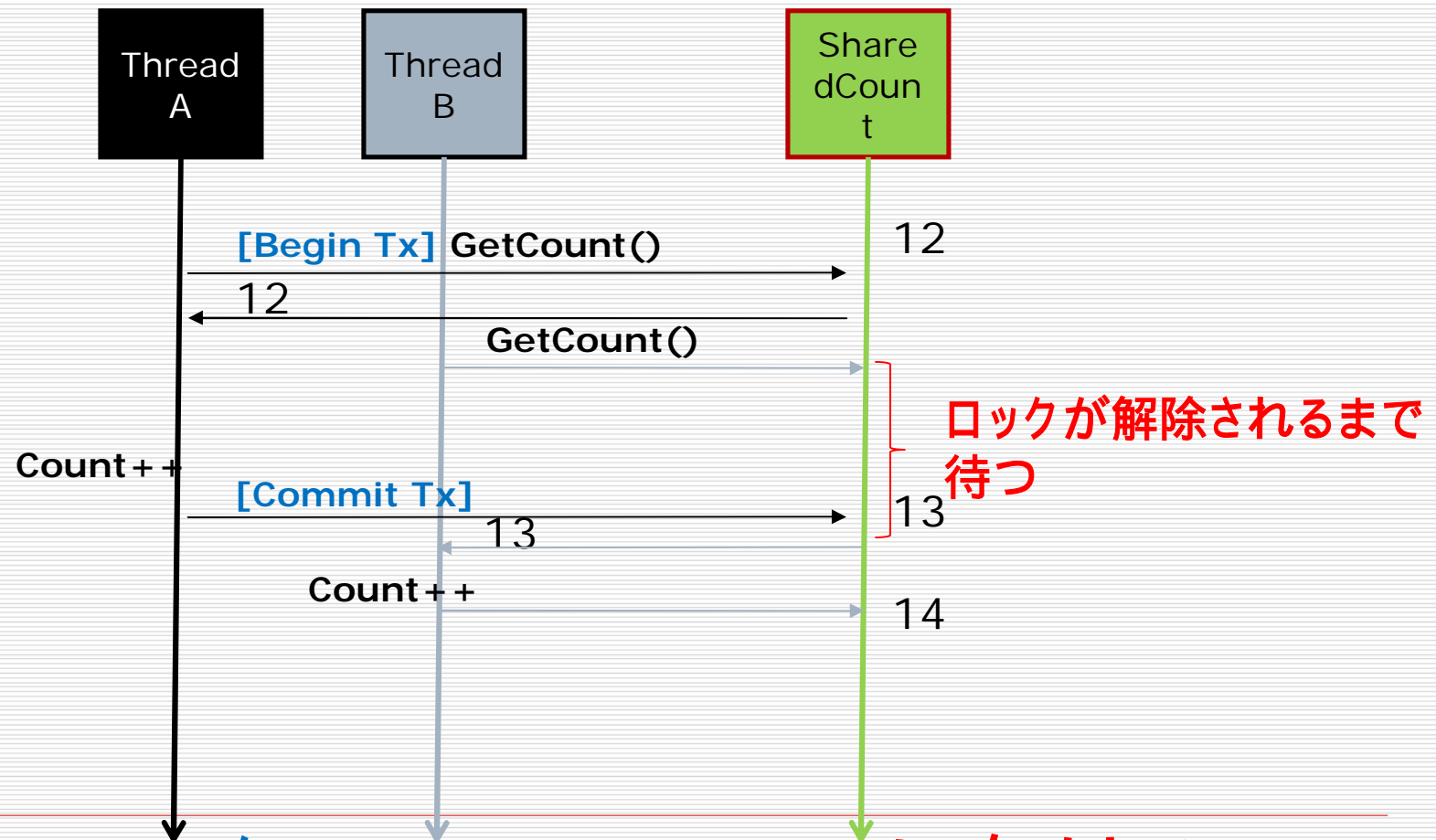
---

ここでは、CloudのBasically Availabilityをささえる、Optimisticな Concurrent Controlの手法をいくつか紹介する。

# 並行処理で共有領域に アクセスする際、問題が起きる例



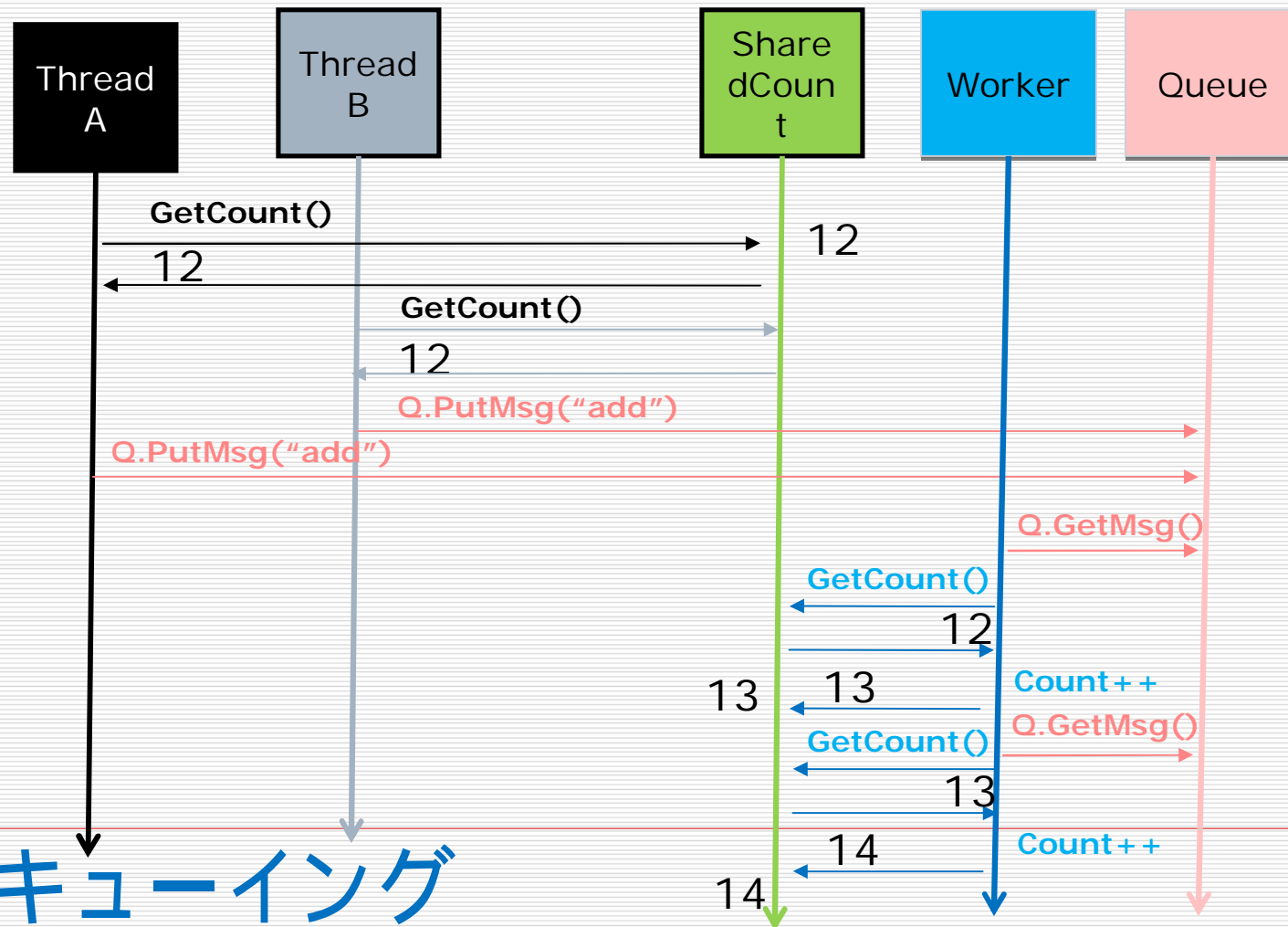
# 並行処理時の競合を防ぐには？



選択1: ロック

Availabilityに欠ける

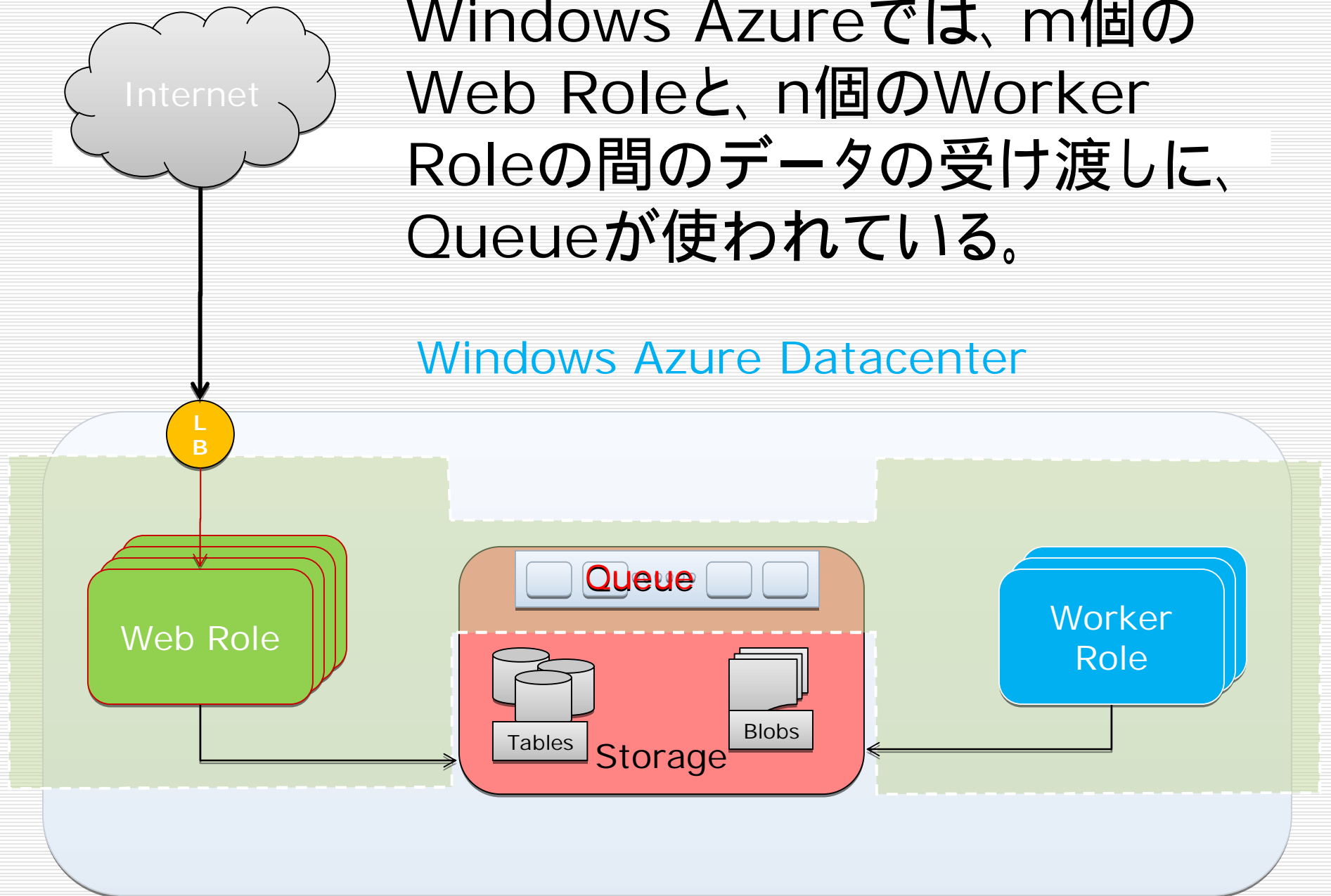
# Basically Availableな処理



選択2: キューイング

Windows Azureでは、 $m$ 個の Web Roleと、 $n$ 個の Worker Roleの間のデータの受け渡しに、Queueが使われている。

Windows Azure Datacenter

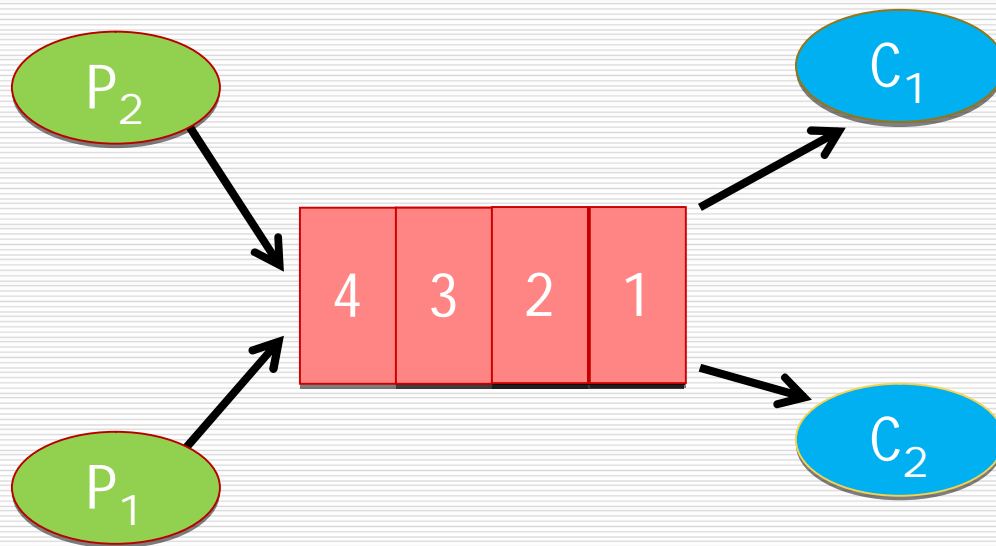


# Queueからのメッセージの取り出しと メッセージのQueueからの消去

---

Producers

Consumers



1. Dequeue(Q, 30 sec)  
命令で msg1が取りだされ  
Queueからはmsg1が  
削除される。

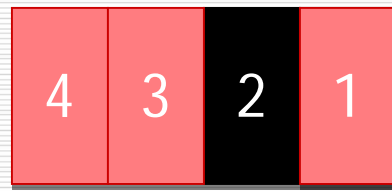
2. 同様に、Dequeue(Q, 30 sec)  
命令で、msg2が取りだされ、  
Queueからは、削除される。
-

# 受け手の側が、 メッセージの利用に失敗した場合

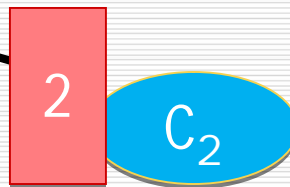
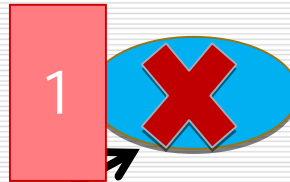
Producers

P<sub>2</sub>

P<sub>1</sub>



Consumers



2. Dequeue(Q, 30 sec) → msg2
3. C2 はmsg2を消費する
4. Delete(Q, msg2)
7. Dequeue(Q, 30 sec) → msg1

1. Dequeue(Q, 30 sec)  
→ msg 1
5. C<sub>1</sub> がこけた!
6. msg1 は、Dequeueの後  
30秒後に、Queueで、復活  
する。

メリット

- 全てのメッセージが、  
少なくとも一回は処  
理されることを保証  
する。

# Basically Availability

## データベースでの楽観的ロック

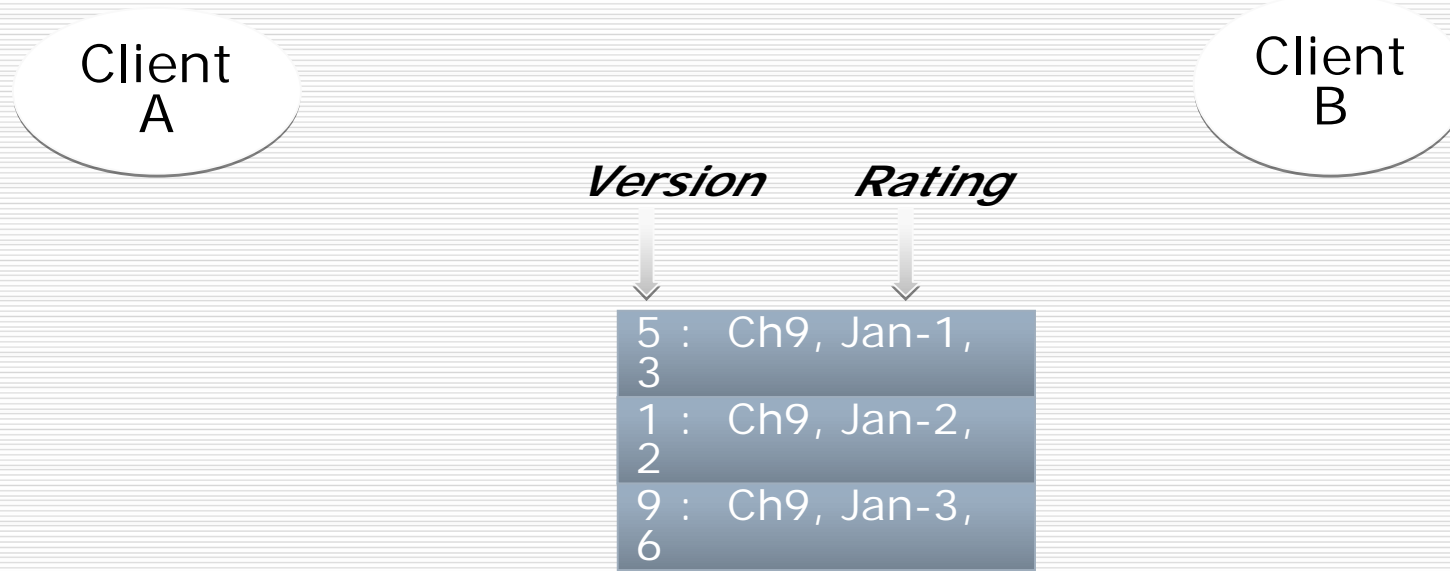
---

データベースで、二つのクライアントが、同時に競合する書き込みをしようとした場合に、どのような処理が行われるのかを見てみよう。

こうした、Optimistic Lockの手法は、現在のエンタープライズ向けのシステムでも、普通に利用されていることに注意しよう。

# Entityの取得

---

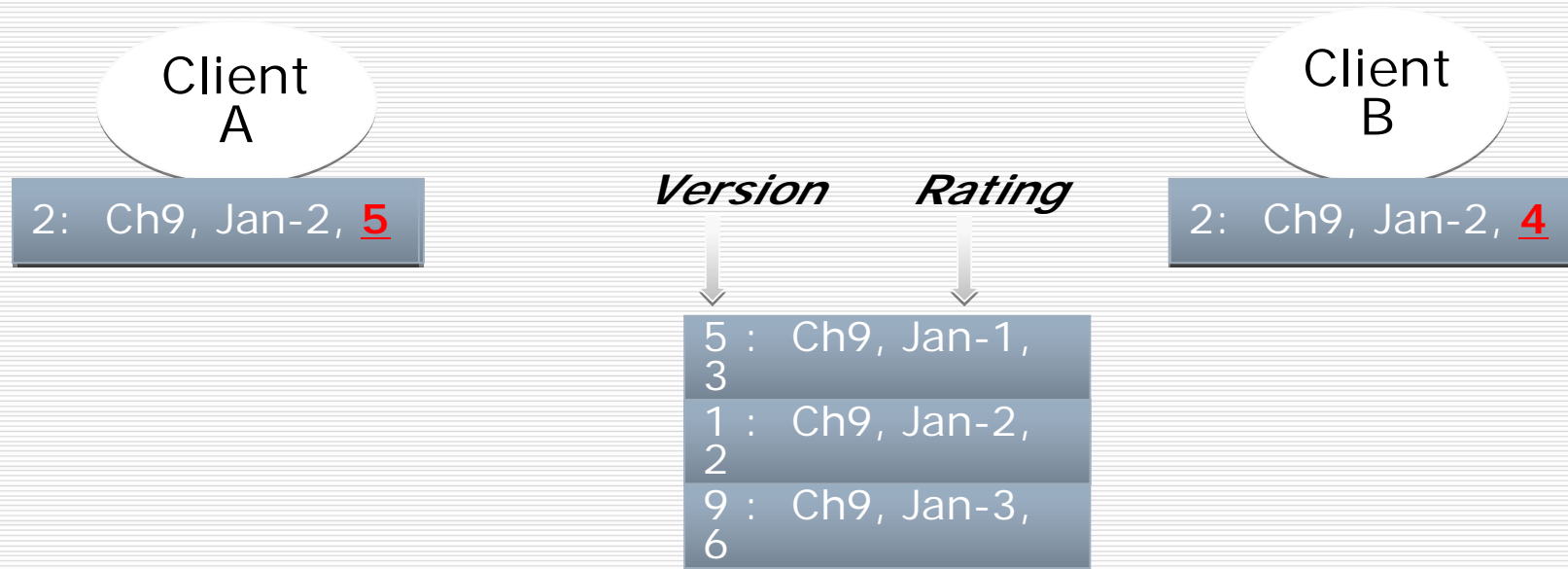


---

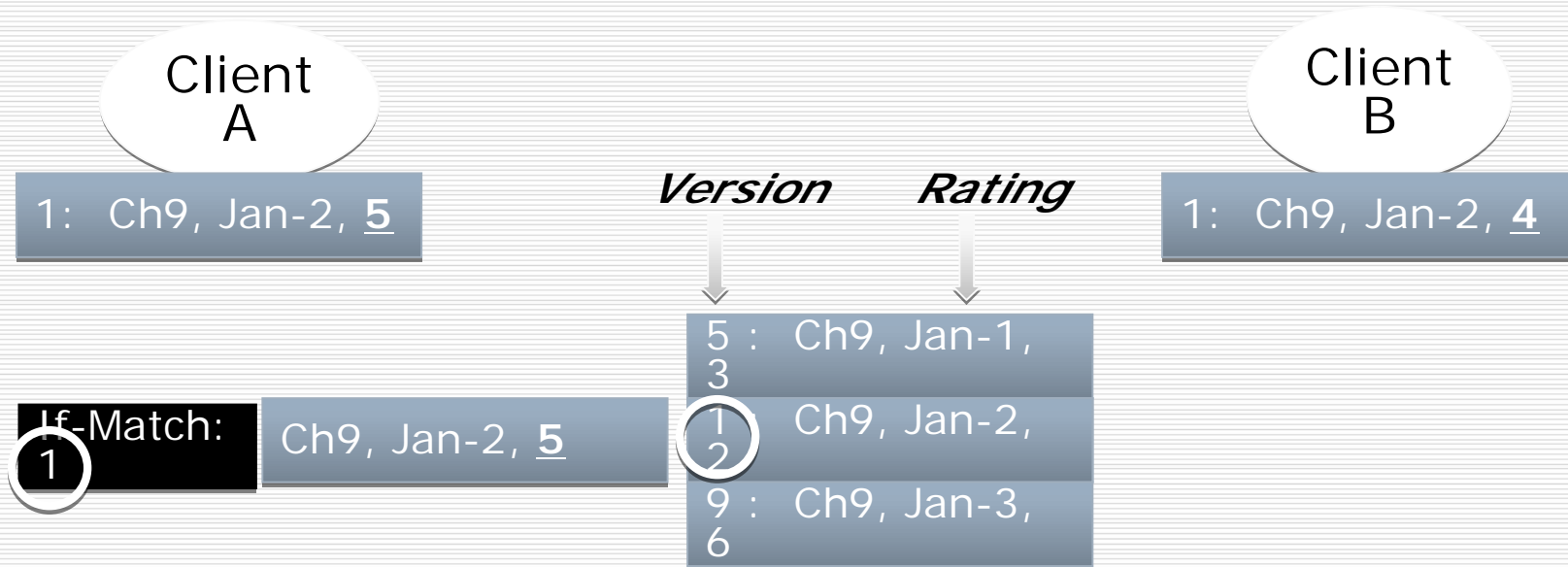
システムの管理するversionを  
Etagとして取得する

# Entityをローカルに更新する

---

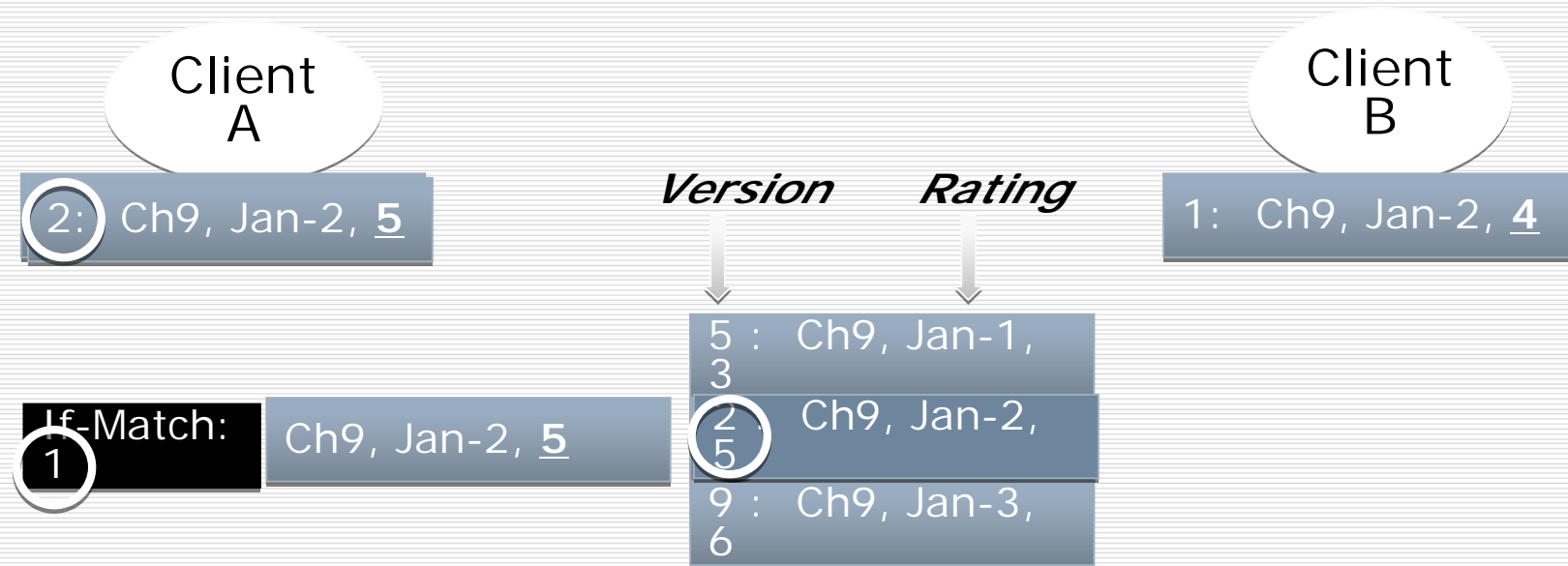


# データを送ってVersionをチェックする



# Versionが合えば、成功である

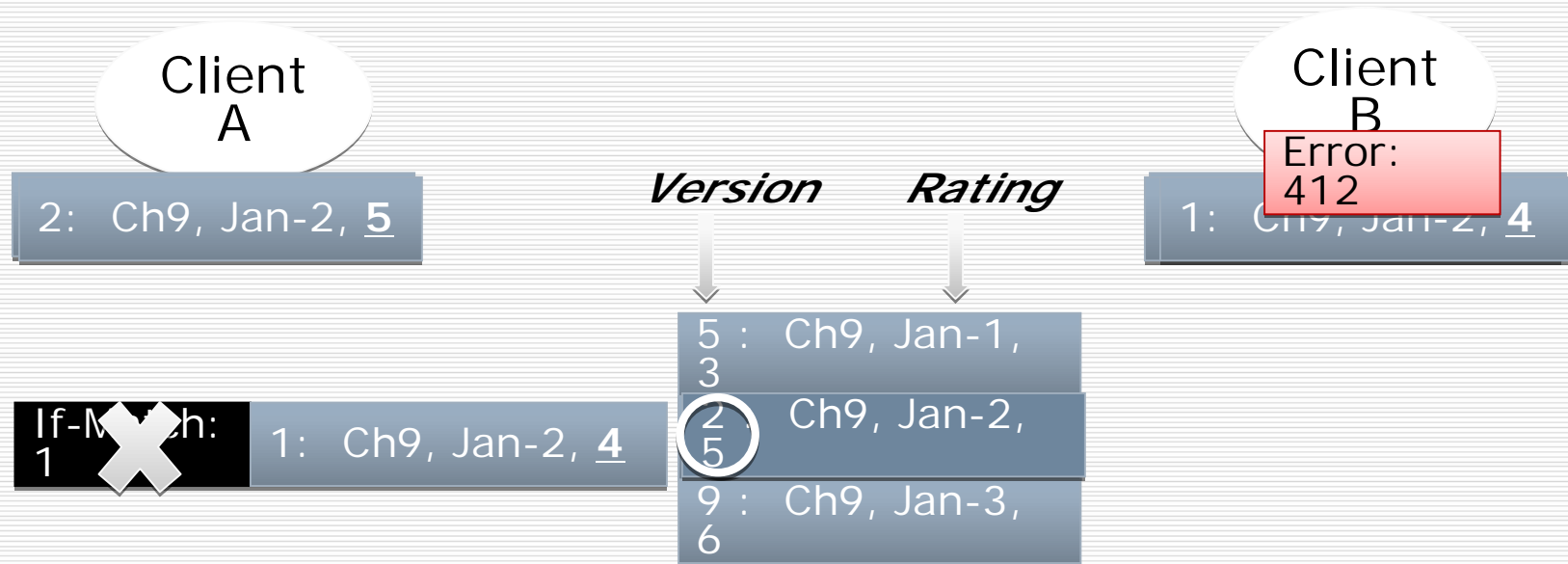
---



システムはVersionとデータを更新し、  
Client-Aを更新する。

---

# Versionが合わなければ、失敗である



システムは、Precondition failed (412) を返す。

# Persistencyの担い手 としてのメモリー

---

これまで、ScalabilityとAvailabilityとConsistency概念の変化を中心に、クラウドの技術の特徴を見てきた。

クラウド技術には、こうした切り口とは別の、もうひとつ大きな特徴がある。それは、クラウド・システムでは、Scale-outで得られた沢山のメモリーをシステムのパフォーマンスの向上に積極的に利用しようという傾向である。

# 1,000台のPCでScale-outしたら？

---

- 今、2Gのメモリーと500Gのハードディスクを備えた、普通のPCを考えてみよう。
  - こうしたPCが1000台集まれば、システム全体のメモリー容量は、2TByteになり、ディスク容量は、0.5Peta Byteに達する。
  - また、この容量は、Scale-outによるScalabilityで、さらに増やすことは可能である。
-

# ファイルとメモリーの区別の相対化

---

- 一年365日休むことなく稼働を続ける、クラウド・システムの高いAvailabilityは、volatileなメモリーとpersistentなファイル・システムという区分を、相対化している。
  - 既に、Coherenceを利用したある基幹系のシステムでは、一年以上、すべてのデータは、メモリー上で処理、格納され、データベースは、ロギングと帳票出力の際にのみ用いられるという事例も生まれている。
-

# メモリー上のデータベースへ

---

- 従来のデータベースは、少し、単純化して言えば、そのベースにあるのは、ファイルシステムに、Indexをつける技術である。
  - データの担い手が、ファイルシステムからメモリーに代わるにあたって、そこでのIndexingの手法が変化するのは、ある意味、当然である。
  - メモリー上の、Key/Value Hashは、メモリー上のIndexingとしては、きわめて自然なものである。
-

# クラウドへのP2P/DHT技術の 導入という新しい流れ

---

ScalabilityとAvailability、メモリーのデータ・ストアとしての利用というクラウド・システムの技術的特徴の集大成として、P2P/DHT技術の利用が、クラウド技術の新しい焦点となりつつある。

# クラウドとP2P/DHTとの接点(1)

---

- 興味深いのは、P2P Overlay技術が目指していた、不随意に発生する物理ノードの欠落・復帰・追加の影響をできるだけ受けないネットワーク網の構築という目標は、想定していたネットワークの規模の違いはあれ、Scale-outで規模の拡大を続けるクラウド・システムのAvailability確保という目標と同じ構造をしているということである。
-

## クラウドとP2P/DHTとの接点(2)

---

- P2P技術とクラウド技術の接点では、もうひとつ重要なことがある。近年のP2P技術の関心は、主に、DHT (Distributed Hash Table) に向けられてきた。
  - DHTは、分散した多数のノード上で、一つの巨大なHash Tableを実現しようという技術である。
  - こうした方向は、分散データベースと分散メモリー・キャッシュの統合を進めようとするクラウド・データベースの発展方向と見事に合致するのである。
-

# まとめ



# まとめ

---

- Scale-outによるScalabilityを持つか否かによって、Cloudシステムのタイプが分かれる。
  - Scale-outによるScalabilityの確保は、CloudシステムがAvailabilityの問題に、真剣に取り組むことを必要とする。
  - Cloudは、ScalabilityとAvailabilityの両立を目指す。それは、従来のConsistency概念の見直しを要求している。
-

# まとめ

---

- こうした中で出てきた、Eventually Consistent Soft State、Basically Availableといった新しい理論的な概念は、重要である。
  - これらの概念は、従来のACID Transactionを否定するのではなく、その性質を深く理解させるものである。
  - Eventually ConsistentとSoft Stateは、情報システムの物理的で原理的な限界を指し示すものである。
-

# まとめ

---

- 実践的には、Basically Availabilityを支える、各種のOptimistic Concurrent Controlの手法の開発が重要である。
  - P2P/DHTの利用は、ScalableでAvailableなCloud技術の基礎になろうとしている。
  - Cloudが、コンシューマ向けで、基幹業務に向かないというのは、誤解である。
-

# 參考資料

---

# “A Note on Distributed Computing”

Jim Waldo et al.

[http://www.sunlabs.com/techrep/1994/sml\\_i\\_tr-94-29.pdf](http://www.sunlabs.com/techrep/1994/sml_i_tr-94-29.pdf)

ローカルなプログラミングとリモートなプログラミングを、はっきりと区別すべきだという立場

ネットワーク上のシステムで相互作用するオブジェクトは、単一のアドレス空間で相互作用するオブジェクトとは、本来的に異なったやり方で取り扱われるべきであると、我々は主張している。

こうした違いが要求されるのは、ネットワーク上のシステムでは、プログラマは遅延の問題を意識せねばならず、異なったメモリアクセスのモデルを持ち、並列性と部分的失敗(*partial failure*)の問題を考慮にいれなければならないからである。

我々は、ローカルとリモートのオブジェクトの違いを覆い隠そうと試みる、沢山のネットワーク・システムを見てきた。そして、これらのシステムは、頑健さと信頼性という基本的な要請を満たすことに失敗していることを示そうと思う。

こうした失敗は、過去においては、構築されたネットワーク・システムの規模の小ささで、隠蔽されていた。しかしながら、近未来に予想される、企業規模のネットワークシステムにおいては、こうした隠蔽は不可能となるであろう。

# Deutsch's 7 Fallacies of Networking

Formulated in  
10 Years Ago

- The Network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- *There is one administrator*
- Transport cost is zero

Network is Homegenous added by Gosling

# 複雑さを考える --- Complexity Quanta and Platform Definition

Summary of [Jim Waldo's](#) Keynote at  
the 10th Jini Community Meeting

[http://www.jini.org/files/meetings/tenth/video/Complexity\\_Quanta\\_and\\_Platform\\_Definition.mov](http://www.jini.org/files/meetings/tenth/video/Complexity_Quanta_and_Platform_Definition.mov)

[http://www.jini.org/files/meetings/tenth/presentations/Waldo\\_keynote.pdf](http://www.jini.org/files/meetings/tenth/presentations/Waldo_keynote.pdf)

# 複雑さにおける基本的な飛躍

---

- **線形実行 (SEQ)** – 人生は善良でシンプルであった
  - **マルチ・スレッド (MT)** – ツールと優秀なプログラマが MT について考えることが必要
  - **マルチ・プロセス (MP)** – カーネルの開発者だけでなく誰もが利用できる。実際には、MT の前に起きた。
  - **マルチ・マシン (MM)** 同一ネットワーク上の – マルチ・プロセスと同じではないのだが、ある人たちは、そう考えている
  - **信頼できないマルチ・マシンたち (MMU)** – 本質的には、Web の世界である
-

## それぞれの段階を通り抜ける際、 我々は、何かを失う

---

### □ マルチ・スレッドへ:

我々は、**順序**を失う(複数のことが同時に起こる)。これは、難しい。なぜなら、我々は、自然には、シーケンシャルに考えるから。

### □ マルチ・プロセスへ:

**単一のコンテキスト**(すなわち、我々が信頼しうる共有コンテキスト)を失う。グローバルな状態が、開発のあらゆるところで利用される。(すべてをスタティックに考えよ)

---

## それぞれの段階を通り抜ける際、 我々は、何かを失う

---

- マルチ・プロセスからマルチ・マシンへ：  
我々は、**状態**を失う。「システム」のグローバルな状態というのは、虚構である。興味深い分散システムには、統合的な状態というものには存在しない。（Lampport: <http://research.microsoft.com/users/lampport/pubs/pubs.html>）  
分散OSのプロジェクトは、グローバルな状態を導入しようとしたが、大々的に失敗した。
- 信頼できないマルチ・マシンたちへ：  
誰を信ずることが出来るか分からない難しい状況の中で、我々は**信頼**を失う。

# しかし、我々は何かを得てきた

---

- Seq to MT : 並列処理
  - MT to MP : プロセスの分離 (安全を与える)
  - MP to MM : 独立した失敗 (何かまずいことが起きても、システムの部分は生きのこる)
  - MM to MMU : スケール (webスケール、インターネットスケール). 誰か他の人のリソースを利用せよ (あるいは、他の誰かが、我々のリソースを利用することを認めよ)
-