# Kai – An Open Source Implementation of Amazon's Dynamo

takemaru

# Outline

- Amazon's Dynamo
  - Motivation
  - Features
  - Algorithms
- Kai
  - Build and Run
  - Internals
  - Roadmap

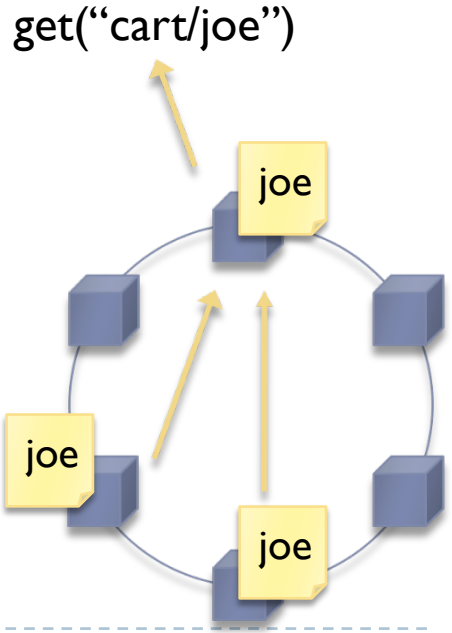
# Dynamo: Motivation

- 最大の電子商取引サイト
  - 75K query/sec (推定)
    - 500 req/sec * 150 query/req
  - O(10M) ユーザ，多くの商品


- RDBMSはダメなのか?
  - スケールアウトや負荷分散が簡単にいかない
  - おおくのサービスは主キーアクセスのみ
- Amazonのためのデータベース
  - 主キーアクセスは *Dynamo*
  - 複雑なクエリは *SimpleDB*
  - 大きめのファイルは *S3*
- Dynamoが使われているのは
  - ショッピングカート，顧客の趣向，セッション管理，売り上げランキング，商品カタログ

# Dynamo: Features

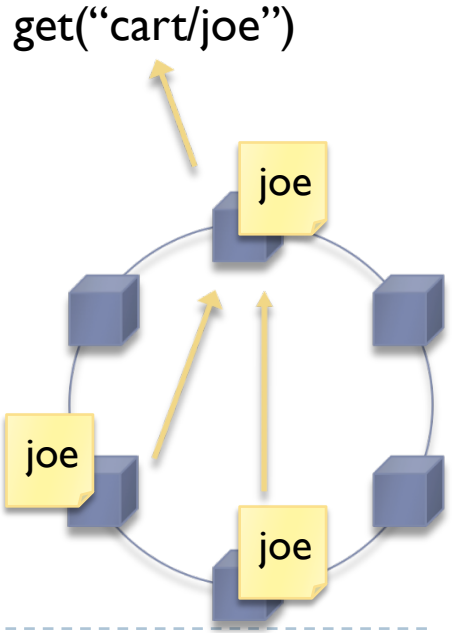
- Key, valueデータストア
  - 分散ハッシュテーブル
- 高いスケーラビリティ
  - マスタ不在，P2P
  - 巨大なクラスタ，おそらくO(1K)
- 耐障害性
  - たとえデータセンター障害であっても
  - 障害時にも待ち時間要求を満たす



# Dynamo: Features, cont'd

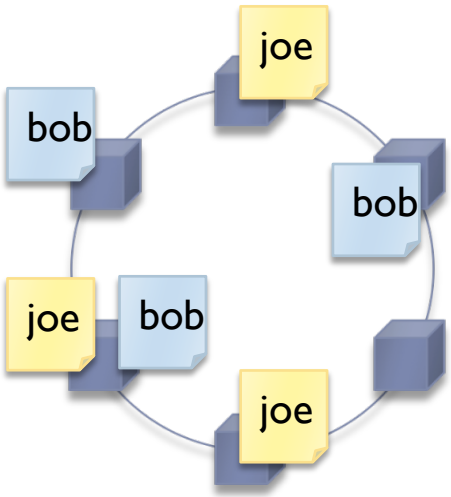
- Service Level Agreements
  - 99.9%のクエリに対して300ms以内
  - 平均では，読み取りに15ms, 書き込みに30ms
- 高い可用性
  - ロックなし，いつでも書き込める
- 結果整合性 (Eventually Consistent)
  - 複製はゆるく同期
  - 不整合はクライアントが解決

可用性と一貫性のトレードオフ
- RDBMSは一貫性を優先
- Dynamoは可用性を重視



# Dynamo: Overview

- Dynamoクラスタ (instance)
  - 等価なノードにより構成
  - キーごとにNの複製を保持



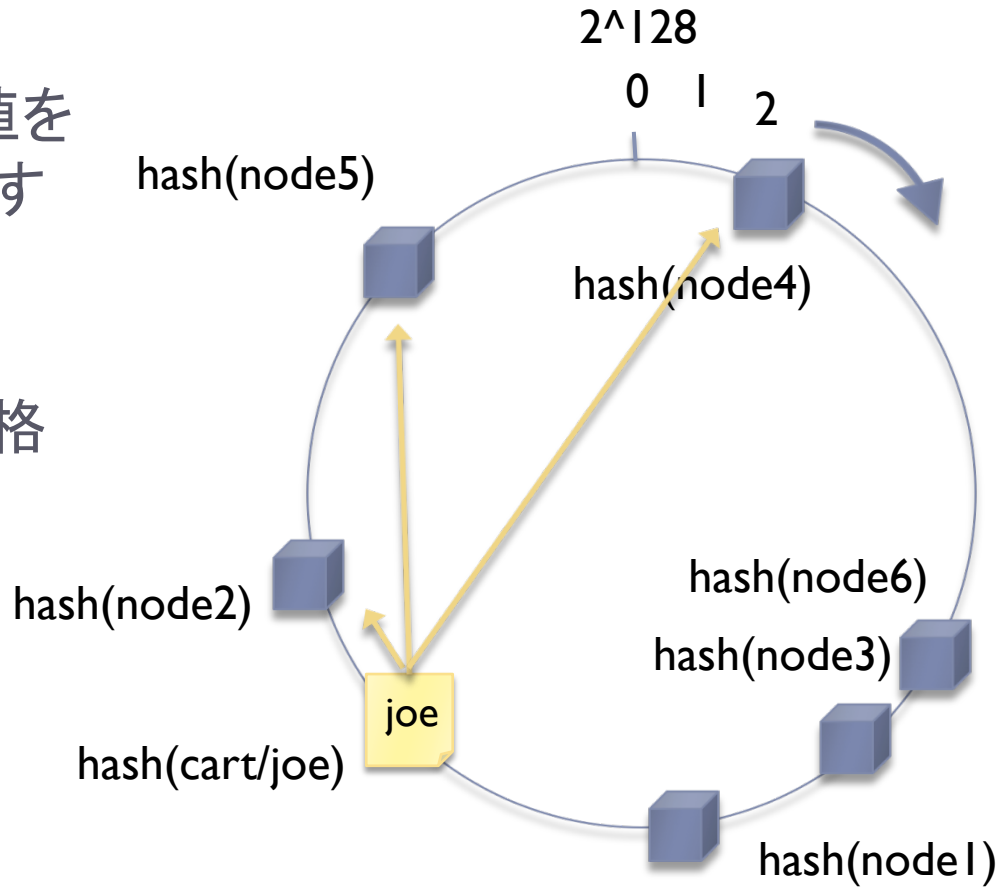
*Dynamo of N = 3*

- Dynamo APIs
  - get(key)
  - put(key, value, context)
    - memcacheの*cas*のようで，contextは一種のバージョン
  - *delete* について論文には記述がない（たぶん定義されてる）

- クライアントへの要求
  - memcacheと異なりすべてのノードを知る必要はない
    - どのノードにリクエストを投げてもよい

# Dynamo: Partitioning

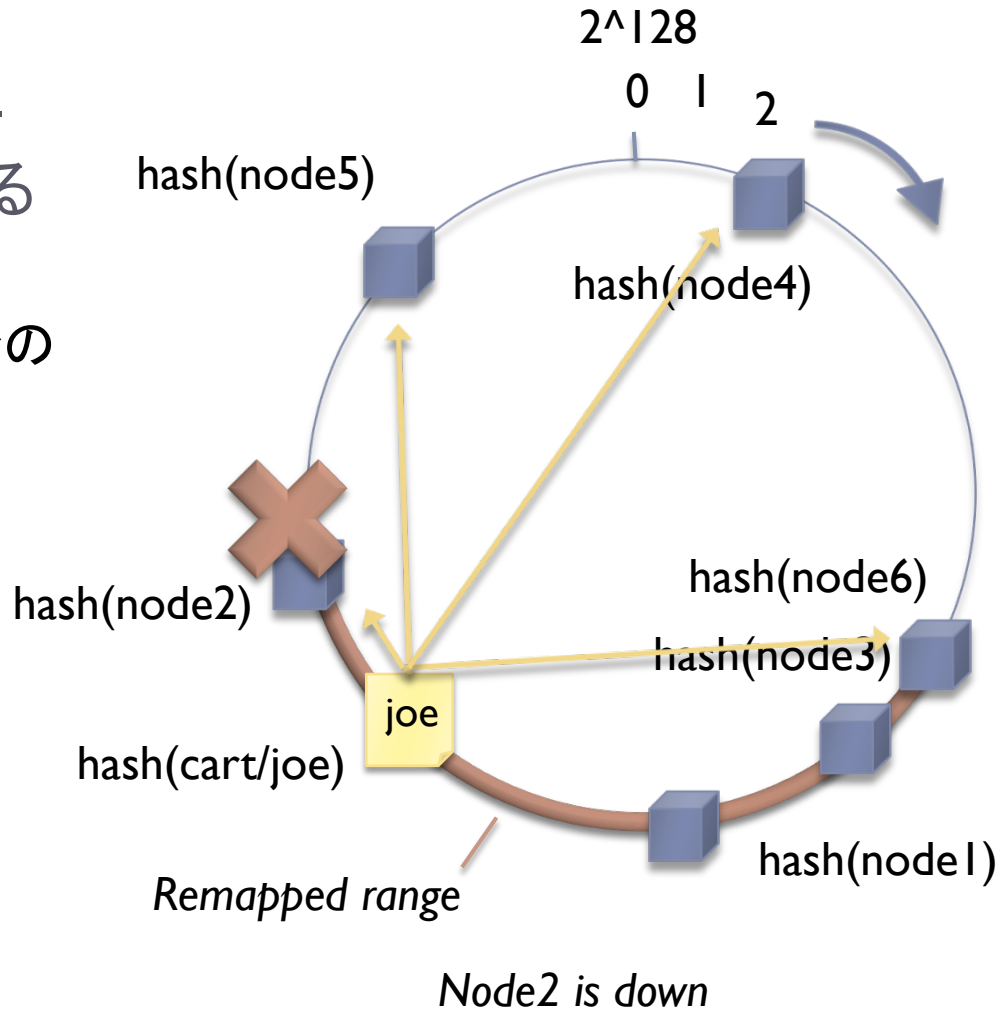
- Consistent Hashing
  - ノードとキーのハッシュ値を計算し，リング上の対応する位置に配置
    - MD5 (128bits)
  - キーは後続のNノードに格納




*Hash ring (N=3)*

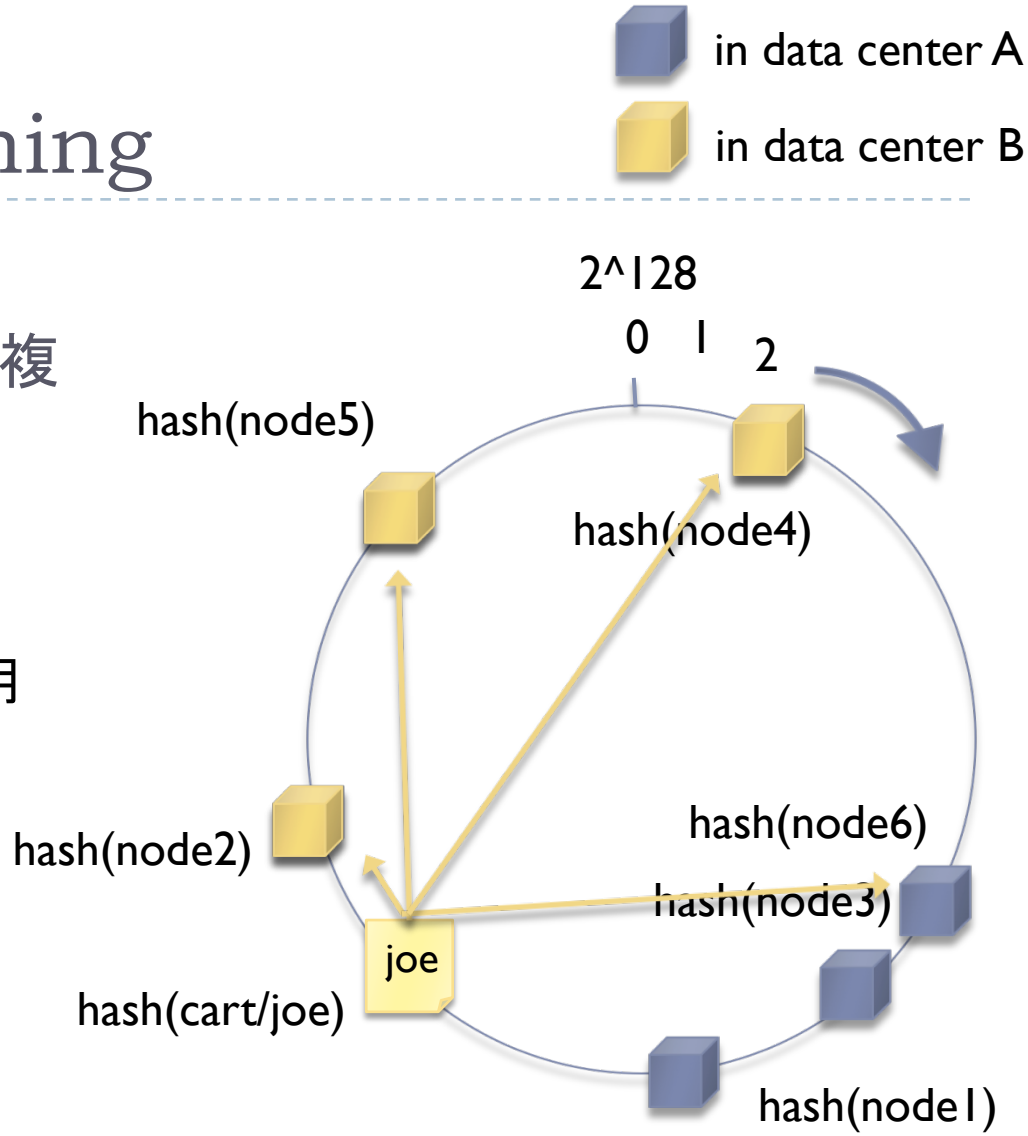
# Dynamo: Partitioning, cont'd

- 利点
  - メンバーシップ（構成ノード）変更時に再配置されるキーが少ない
    - 障害ノードからN台前までの間にあるキー



# Dynamo: Partitioning

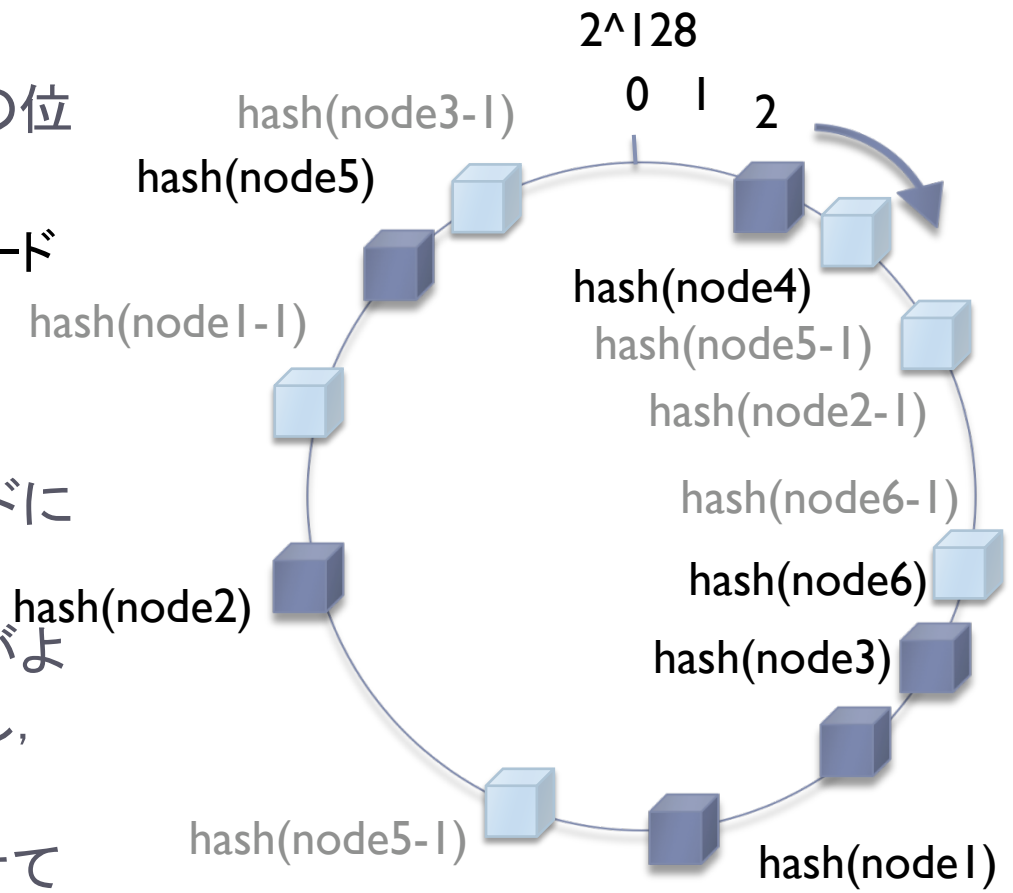
- 複製の物理配置
  - 複数のデータセンターに複製
  - 配置方法は不明
    - ネットマスクかなぁ
    - 複製分布への影響は不明
- 利点
  - データセンター障害でもデータにアクセスできる




*joe is replicated in multiple data centers*

# Dynamo: Partitioning, cont'd

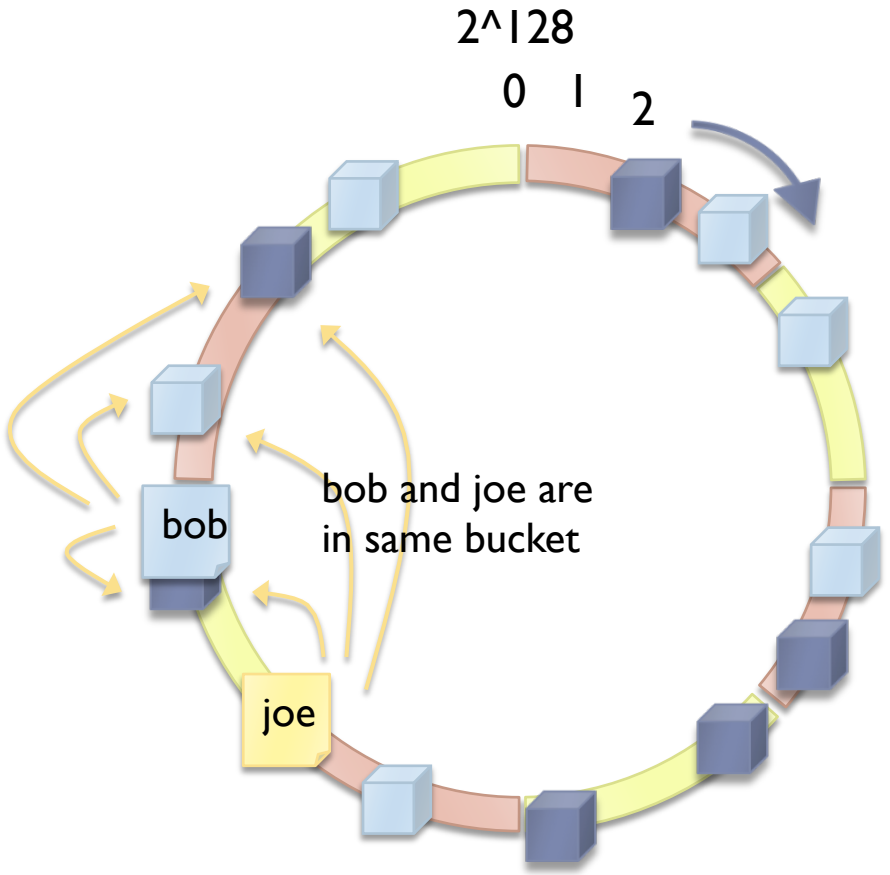
- 仮想ノード
  - ひとつの物理ノードを複数の位置に配置
    - O(100) 仮想ノード/物理ノード
- 利点
  - 統計効果により，物理ノードに対する分布がより一様に
  - 再配置の対象となるキーがより多くの物理ノードに分散し，負荷分散に
  - 物理ノードの性能に合わせて仮想ノード数を調整可能




*Two virtual nodes per each physical node*

# Dynamo: Partitioning, cont'd

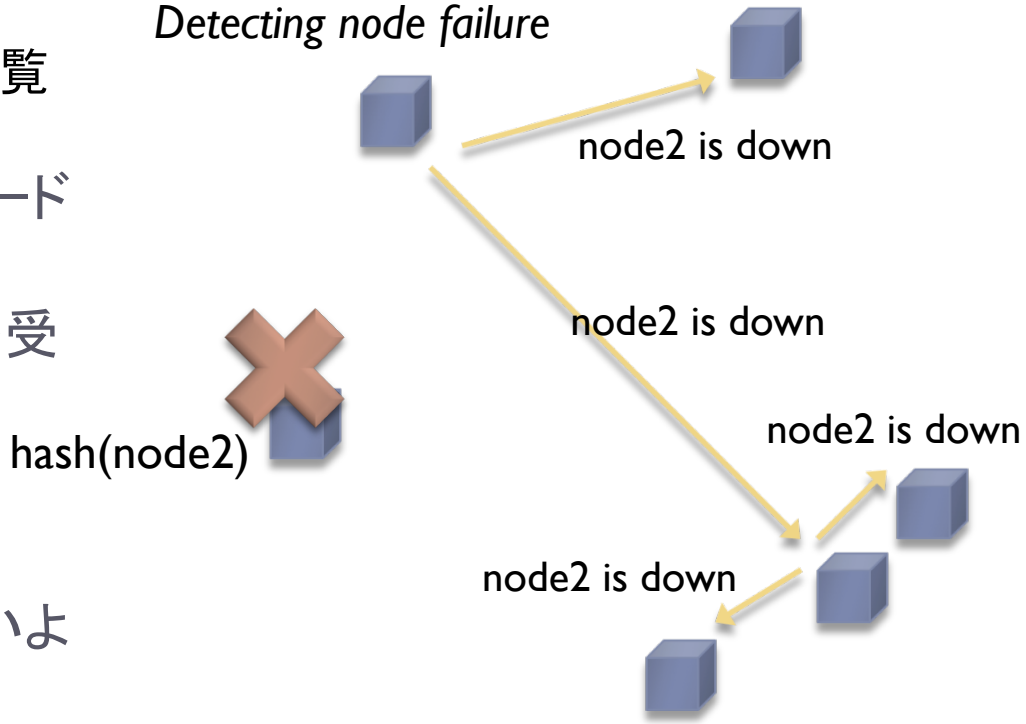
- バケット
  - リングをバケットに等分割
    - バケットはすべての仮想ノード数より多くすること
  - 同じバケットにあるキーは，同じノードに格納される
- 利点
  - キーの同期をバケット単位で簡単に
    - 詳しくはMerkleツリーのスライドで




*Divided into 8 buckets*

# Dynamo: Membership

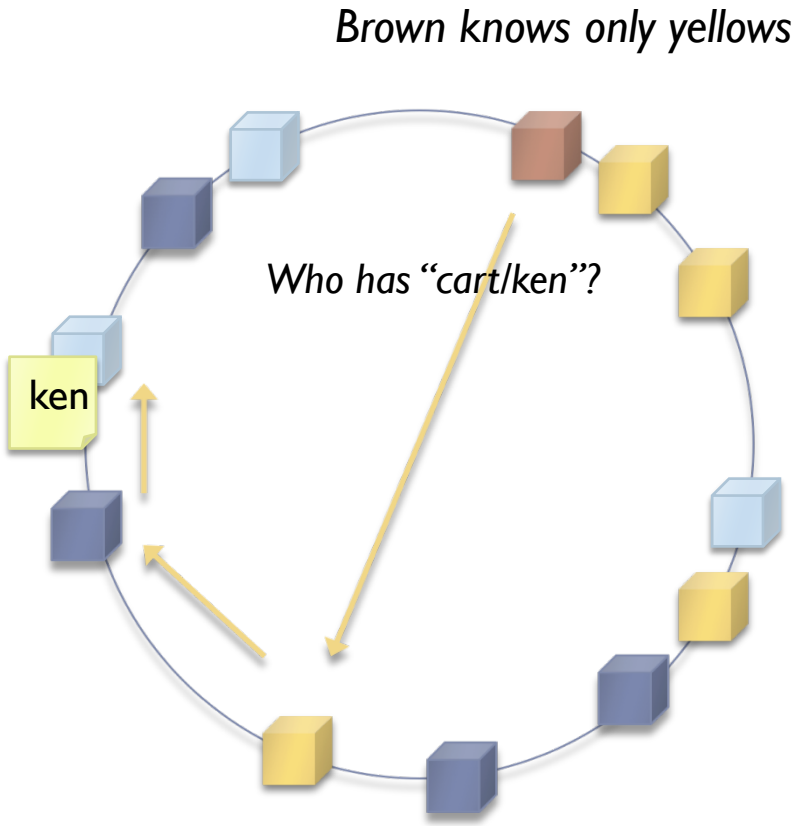
- ゴシップ・プロトコル
  - メンバーシップ（構成ノード）を噂のように拡散
    - メンバーシップとはノード一覧や変更履歴など
  - 毎秒，ランダムに選んだノードとメンバーシップを交換
  - より新しいメンバーシップを受け取ったら更新
- 簡単
  - 頑強，誰も噂を止められないように
  - 指数的に素早く拡散




*Membership change is spread by the form of gossip*

# Dynamo: Membership, cont'd

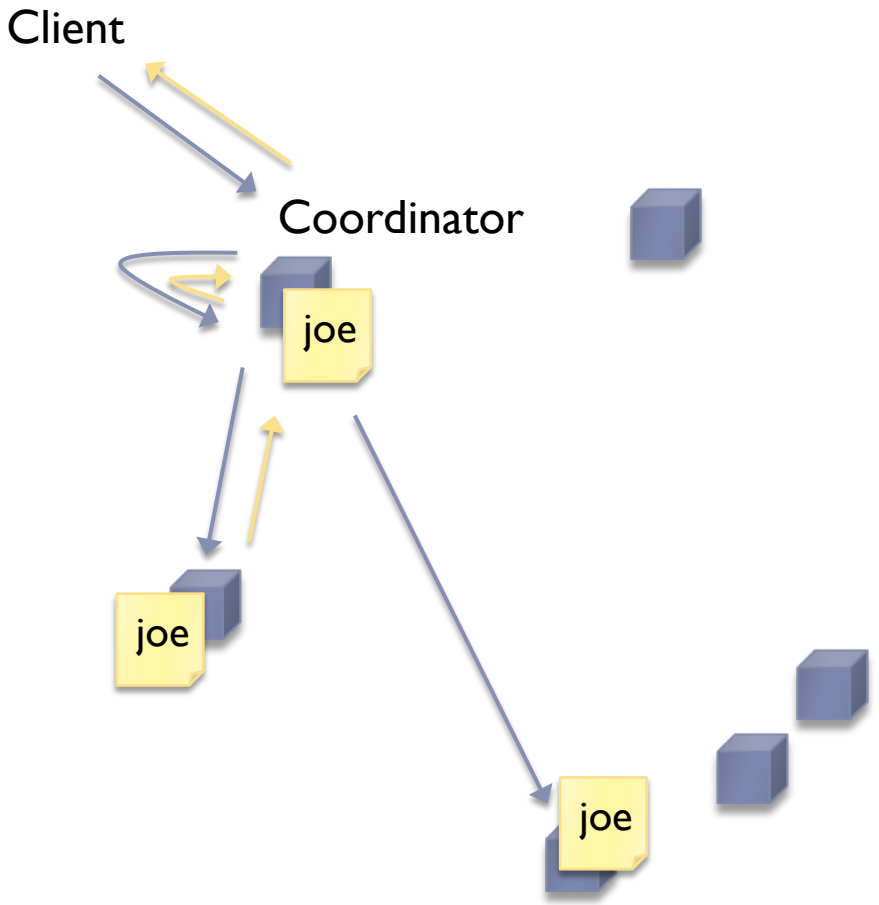
- Chord
  - 大規模クラスタ向け
    - 仮想ノード数 > O(10K)
  - 各ノードはO(log *v*)のノードのみを知ればよい
    - *v* は仮想ノード数
    - ハッシュリングの近傍ノードほどよく知るイメージ
  - いくつかのノードをホップして目的ノードに到達
    - ホップ数 < O(log *v*)
  - 詳細は論文を参照




*Routing in Chord*

# Dynamo: *get/put* Operations

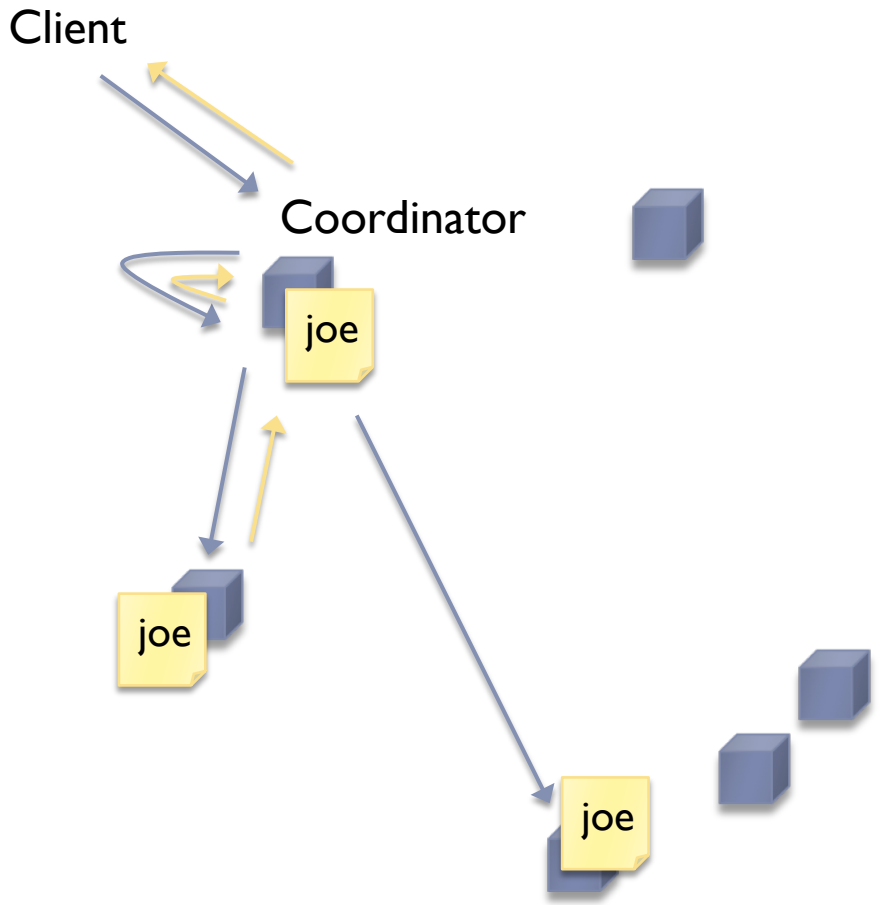
- クライアントの動作
  1. リクエストを任意のノードに送信
  - リクエストはコーディネータへと転送
    - コーディネータとは，キーと担当するノードのどれか
- コーディネータの動作
  1. Consistent hashingによりNのノードを選択
  2. 選択ノードにリクエストを転送
  3. *R* あるいは*W*台のノードからの返答を待機（タイムアウトあり）
  4. getであれば複製のバージョン整合性を確認
  5. レスポンスをクライアントに送信




*get/put operations for N,R,W = 3,2,2*

# Dynamo: *get*/*put* Operations, Cont'd

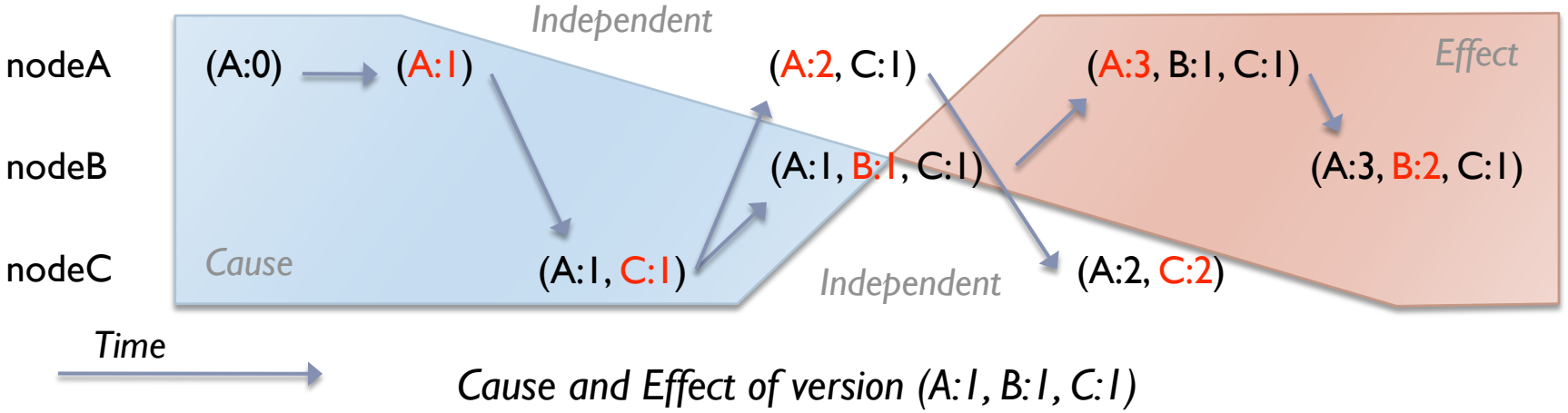
- 定足数アルゴリズム (Quorum)
  - パラメータ
    - *N*: 複製数
    - *R*: 最小の読み取り成功数
    - *W*: 最小の書き込み成功数
  - 条件
    - *R*+*W* > *N*
      - 読み取った複製の少なくともひとつは，書き込みに成功したものとなる
    - *R* < *N*
      - 待ち時間を向上する
    - 通常は *N*,*R*,*W* = 3,2,2




*get/put operations for N,R,W = 3,2,2*

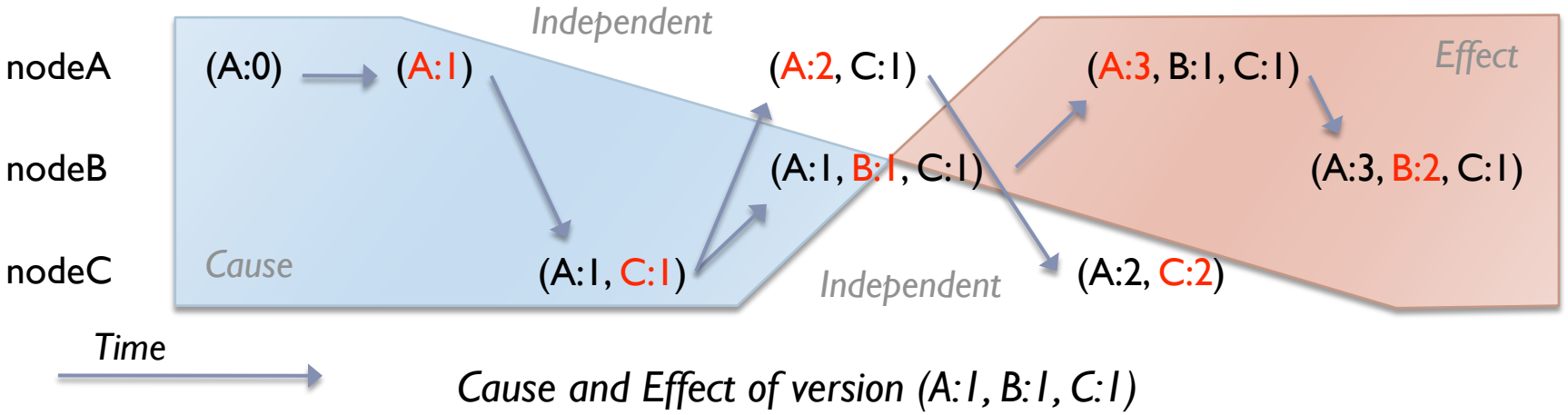
# Dynamo: Versioning

- Vector Clocks
  - 分散システムにおけるバージョンの順序づけ
    - マスター不在，時刻同期なし
    - 並行ブランチの検出可能
  - 各ノードに対応したクロック（カウンタ）のリスト
    - 各クロックはコーディネータが更新
    - (A:2, C:1) は，Aのクロックが2, Bのが1のときに更新されたという意味



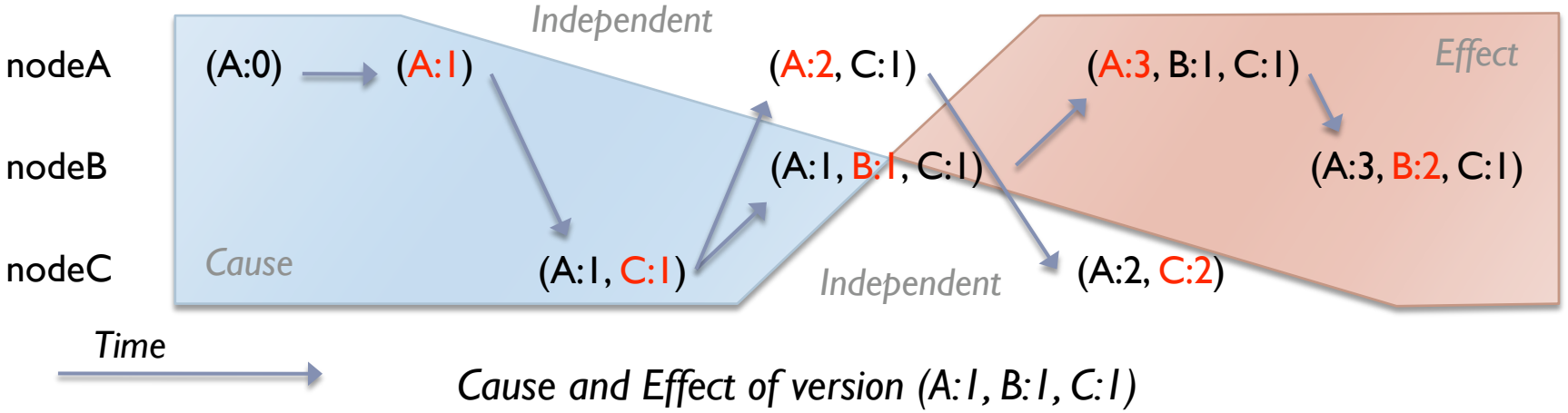
# Dynamo: Versioning, cont'd

- Vector Clocks
  - コーディネータがキーを更新するときの動作
    - 自分のクロックをインクリメント
    - Vector Clocksにある自分のクロックを更新



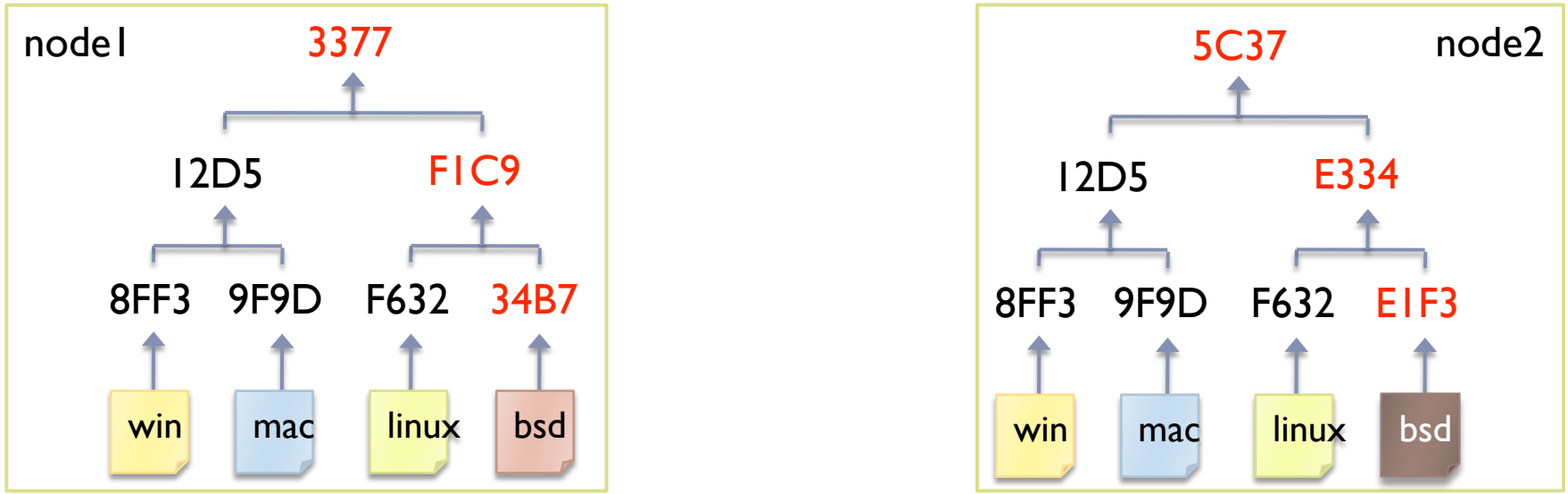
# Dynamo: Versioning, cont'd

- Vector Clocks
  - Vector Clocksを比較して順序づける方法
    - すべてのクロックが同じか大きければ
      - 大きい方が新しい
        - 例: (A:1, B:1, C:1) < (A:3, B:1, C:1)
    - そのような関係になければ
      - 並行ブランチなので順序づけられない
        - 例: (A:1, B:1, C:1) ? (A:2, C:1)



# Dynamo: Synchronization

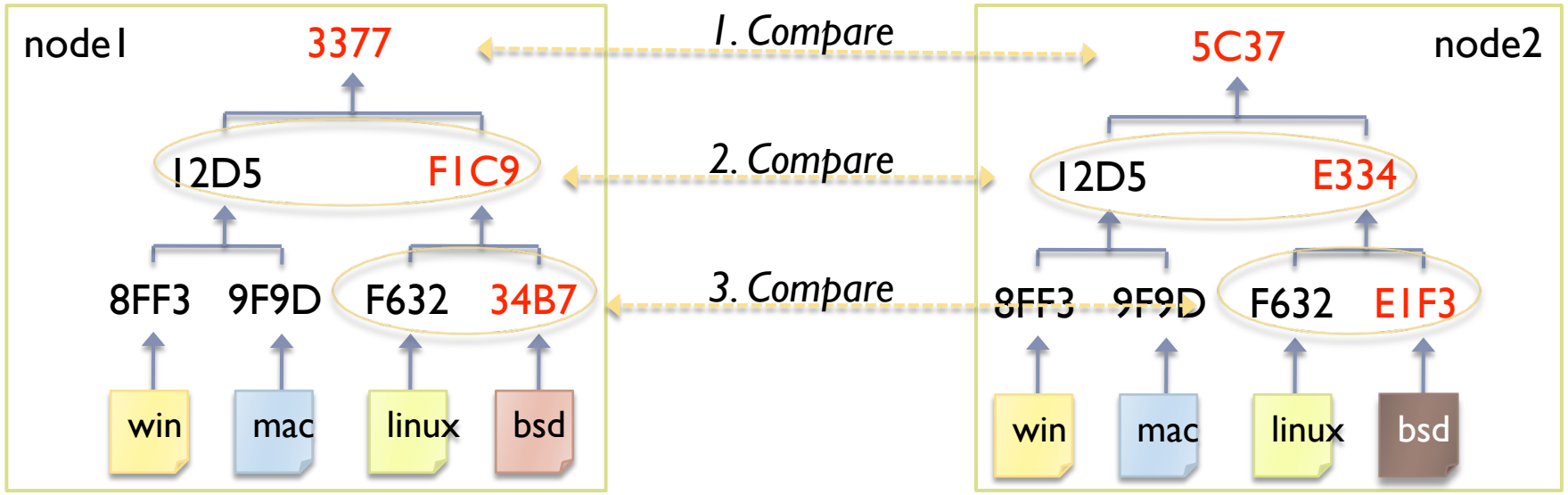
- Merkleツリーによる複製同期
  - 階層的なチェックサム (Merkleツリー) を計算しておく
    - ツリーはバケットごとに構築
  - 定期的あるいはメンバーシップが変更されたときに同期を実行




*Comparison of hierarchical checksums in Merkle trees*

# Dynamo: Synchronization, cont'd

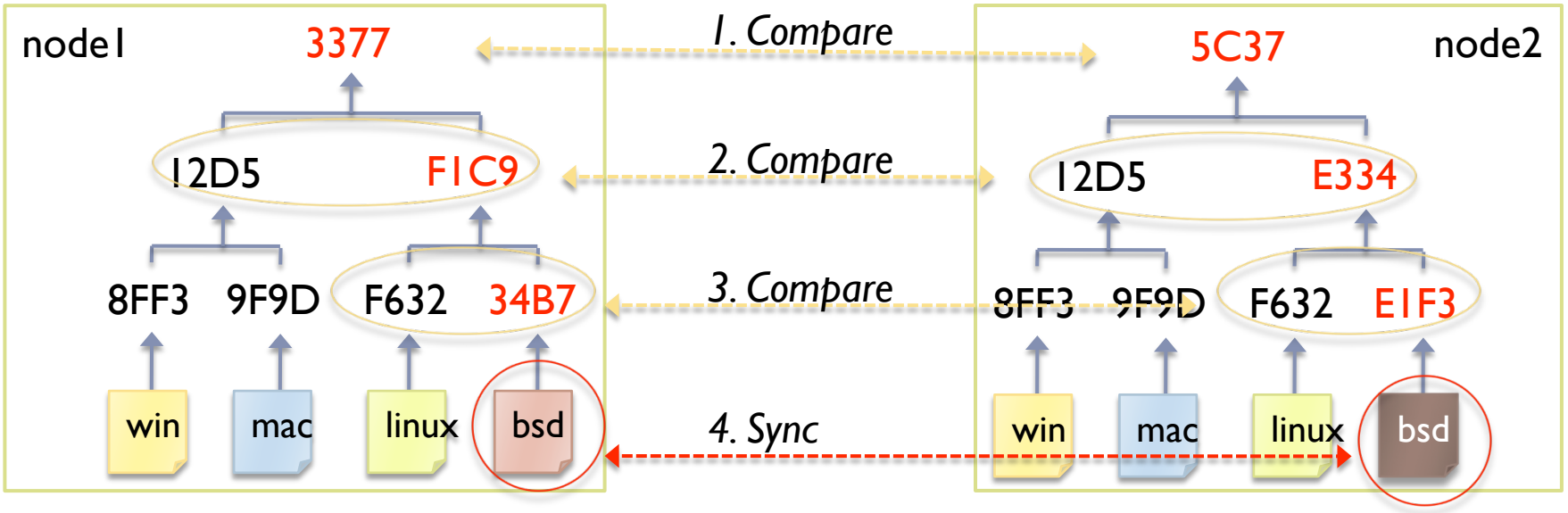
- Merkleツリーによる複製同期
  - 他のノードとMerkleツリーを比較
    - ツリーの根から葉に向けて，チェックサムが一致するところまで比較




*Comparison of hierarchical checksums in Merkle trees*

# Dynamo: Synchronization, cont'd

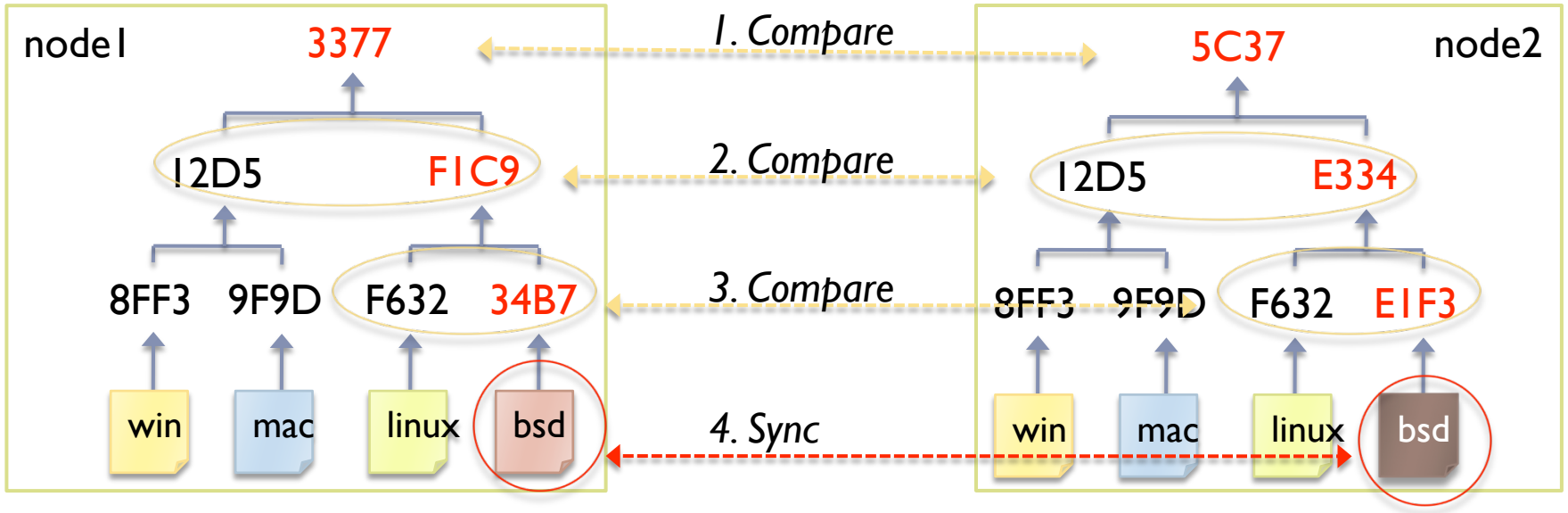
- Merkleツリーによる複製同期
  - 古い（チェックサムの一致しない）複製を同期
    - 優先度低めのバックグラウンド・タスクとして
    - バージョンを順序づけられなければ同期しない




*Comparison of hierarchical checksums in Merkle trees*

# Dynamo: Synchronization, cont'd

- 利点
  - ほとんどの複製で同期がとれていれば，少ない比較で済む
    - 根のチェックサムが一致すれば，それ以上の比較は不要



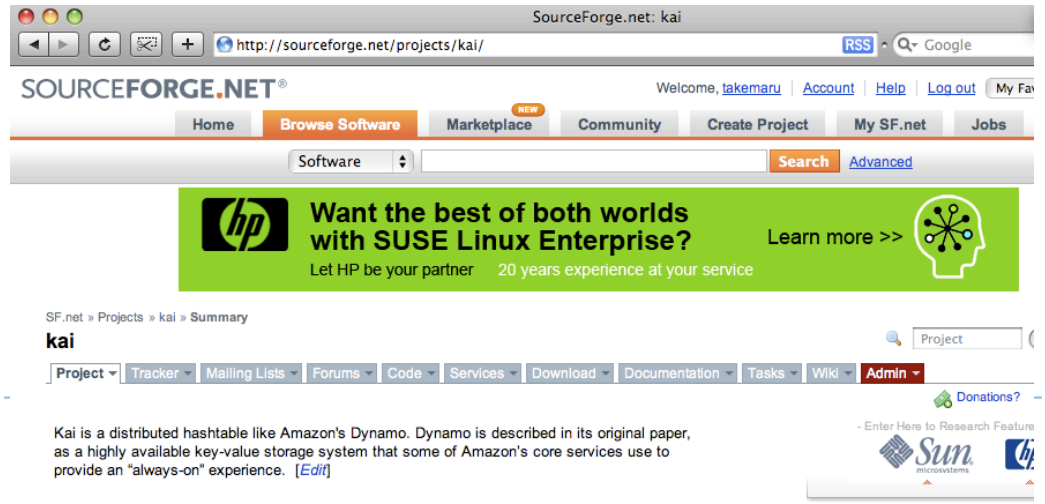

*Comparison of hierarchical checksums in Merkle trees*

# Dynamo: Implementation

- 実装
  - Java
  - クローズ・ソース ☹
- APIs
  - 独自HTTP
- ストレージ
  - BDBあるいはMySQL
- セキュリティ
  - なし

# Kai: Overview

- Kai
  - Dynamoのオープンソース実装
    - 開発者の本籍地から命名
    - *OpenDynamo*という名前は，Amazon Dynamoとは関係ないプロジェクトにとられてた ☹
  - Erlang
  - memcache API
  - http://sourceforge.net/projects/kai/



# Kai: Building Kai

- 必要パッケージ
  - Erlang OTP (>= R12B)
  - make
- Build

```
% svn co http://kai.svn.sourceforge.net/svnroot/kai/trunk kai
% cd kai/
% make
% make test
```

  - *make test* の前に，*Makefile* のRUN_TESTを環境に合わせて編集（MacOSXに合わせてある）
    - *./configure* で自動設定できるといいなぁ

# Kai: Configuration

- *kai.config*
  - 設定項目はすべてオプション（デフォルト値があるので）

| Parameter | Description | Default value |
|---|---|---|
| logfile | ログファイル名 | 標準出力 |
| hostname | ホスト名，複数のネットワーク・インタフェースを持つときは指定した方がよい | 自動検出 |
| port | 内部APIのポート番号 | 11011 |
| memcache_port | memcache APIのポート番号 | 11211 |
| n, r, w | 定足数アルゴリズムのN, R, W | 3, 2, 2 |
| number_of_buckets | バケット数，他のノードと一致していること | 1024 |
| number_of_virtual_nodes | 仮想ノード数 | 128 |

# Kai: Running Kai

- スタンドアローンで実行

```
% erl -pa src -config kai -kai n 1 -kai r 1 -kai w 1

1> application:load(kai).
2> application:start(kai).
```

  - 引数

| Arguments | Description |
|---|---|
| -pa src | ディレクトリ *src* をErlangライブラリパスに追加 |
| -config kai | 設定ファイル *kai.config* |
| -kai n1 -kai r1 -kai w1 | 設定の上書き N, R,W = 1, 1, 1 |

- memcacheクライアントで127.0.0.1:11211にアクセス

# Kai: Running Kai, cont'd

- クラスタとして実行
  - 1台のマシン上にポート番号を変えて3ノードを起動

**Terminal 1**

```
% erl -pa src -config kai -kai port 11011 -kai memcache_port 11211

1> application:load(kai).
2> application:start(kai).
```

**Terminal 2**

```
% erl -pa src -config kai -kai port 11012 -kai memcache_port 11212

1> application:load(kai).
2> application:start(kai).
```

**Terminal 3**

```
% erl -pa src -config kai -kai port 11013 -kai memcache_port 11213

1> application:load(kai).
2> application:start(kai).
```

# Kai: Running Kai, cont'd

- クラスタとして実行
  - ノードを接続

```
3> kai_api:check_node({{127,0,0,1}, 11011}, {{127,0,0,1}, 11012}).
4> kai_api:check_node({{127,0,0,1}, 11012}, {{127,0,0,1}, 11013}).
5> kai_api:check_node({{127,0,0,1}, 11013}, {{127,0,0,1}, 11011}).
```

  - memcacheクライアントで127.0.0.1:11211-11213にアクセス
  - クラスタへのノード追加方法

```
% 新しいノードをクラスタに接続
% (kai_api:check_node/2 によるリンクは片方向)
1> kai_api:check_node(NewNode, NodeInCluster).

% バケット同期の完了を待つ…

% クラスタノードのひとつを新しいノードに接続
2> kai_api:check_node(NodeInCluster, NewNode).
```

# Kai: Internals

| Function | Module | Comments |
|---|---|---|
| Partitioning | *kai_hash* | • 物理配置は未実装 |
| Membership | *kai_network* | • Chordは未実装<br>• *kai_membership* に改名する予定 |
| Coordinator | *kai_memcache* | • *kai_coordinator* に分離して実装し直す予定 |
| Versioning | | • 未実装 |
| Synchronization | *kai_sync* | • Merkle tree は未実装 |
| Storage | *kai_store* | • *ets* |
| Internal API | *kai_api* | |
| API | *kai_memcache* | • get, set, delete のみ実装 |
| Logging | *kai_log* | |
| Configuration | *kai_config* | |
| Supervisor | *kai_sup* | |

# Kai: *kai_hash*

- ベースモジュール
  - *gen_server*
- 現状
  - Consistent hashing
    - ハッシュ空間は 32bit
    - 仮想ノード
    - バケット
- これから
  - 物理配置
  - 読み取りにおける並列アクセスを許可
    - ハッシュ再計算における待ち時間を回避
    - *gen_server:call* を使わない

**Synopsis**

```
kai_hash:start_link(),

# メンバーシップ変更に伴うハッシュ再計算
{replaced_buckets, ListOfReplacedBuckets} =
    kai_hash:update_nodes(ListOfNodesToAdd, ListOfNodesToRemove),

# Key に対応するノードを発見
{nodes, ListOfNodes} = kai_hash:find_nodes(Key).
```

# Kai: *kai_network*, will be renamed to *kai_membership*

- ベースモジュール
  - *gen_fsm*
- 現状
  - EPMDのような組み込み分散環境はスケールしなそうなので利用せず
  - ゴシップ・プロトコル
- これから
  - Chord あるいは Kademlia
    - Kademlia は BitTorrent で使われてる

**Synopsis**

```
kai_network:start_link(),

# Node の生死を確認し，必要ならハッシュを再計算
kai_network:check_node(Node),

# バックグラウンドで，毎秒，ランダムに選んだノードの生死を確認
```

# Kai: *kai_coordinator*, now implemented in *kai_memcache*

- ベースモジュール
  - *gen_server* (予定)
- 現状
  - *kai_memcache* に実装されている
  - 定足数アルゴリズム
- これから
  - *kai_memcache* から分離
  - クライアントからのリクエストをコーディネータに転送
    - 現在は，最初にリクエストを受信したノードにコーディネータをさせている

**Synopsis (予定)**

```
kai_coordinator:start_link(),

% N のノードに get を送信し，受け取ったデータを kai_memcache に返信
Data = kai_coordinator:get(Key),

% N のノードに put を送信，ここで Data は data 型レコードの変数
kai_coordinator:put(Data).
```

# Kai: *kai_version*

- ベースモジュール
  - *gen_server* (予定)
- 現状
  - 未実装
- これから
  - Vector Clocks

**Synopsis (予定)**

```
kai_version:start_link(),

% VectorClocks の LocalNode に対応するクロックを更新
kai_version:update(VectorClocks, LocalNode),

% 順序づけを実行
{order, Order} = kai_version:order(VectorClocks1, VectorClocks2).
```

# Kai: *kai_sync*

- ベースモジュール
  - *gen_fsm*
- 現状
  - データがないときのみ同期
    - バージョン比較なし
- これから
  - バルク転送
    - メンバーシップが変更されたときはバケットを丸ごと送信
  - 並行ダウンロード
    - 複数ノードと並行して同期を実行
  - Merkle tree

**Synopsis**

```
kai_sync:start_link(),

# 他のノードのバケットにあるキー一覧を取得し，
# 存在しないものをダウンロード
kai_sync:update_bucket(Bucket),

# バックグラウンドで，毎秒，ランダムに選んだバケットを同期
```

# Kai: *kai_store*

- ベースモジュール
  - *gen_server*
- 現状
  - Erlang組み込みのメモリストレージである *ets* を利用
- これから
  - 容量制限のない永続的なストレージ
    - *dets*, *mnesia*, MySQLなど
    - ただし，*dets* と *mnesia* のテーブルは4GBまで
  - 遅延削除

**Synopsis**

```
kai_store:start_link(),

# Retrieves Data associated with Key
Data = kai_store:get(Key),

% Stores Data, which is a variable of data record
kai_store:put(Data).
```

# Kai: *kai_api*

- ベースモジュール
  - *gen_tcp*
- 現状
  - 内部API
    - *kai_hash*, や *kai_store*, *kai_network* へのRPC
- これから
  - プロセス・プール
    - API呼び出しを受信するプロセス数を制限
  - 接続プール
    - TCP接続の再利用

**Synopsis**

```
kai_api:start_link(),

# Node からノード一覧を取得
{node_list, ListOfNodes} = kai_api:node_list(Node),

# Node から Key に対応するデータを取得
Data = kai_api:get(Node, Key).
```

# Kai: *kai_memcache*

- ベースモジュール
  - *gen_tcp*
- 現状
  - memcache API の *get*, *set*, *delete* をサポート
    - Kai はキャッシュではないので，*set* の *exptime* はゼロでなければならない
    - 複数のバージョンが存在するときは *get* は複数の値を返す
- これから
  - *cas*, *stats*
  - プロセス・プール
    - API呼び出しを受信するプロセス数を制限

**Synopsis in Ruby**

```
require 'memcache'

cache = MemCache.new '127.0.0.1:11211'

# 'key' に 'value' を格納
cache['key'] = 'value'

# 'key' に対応するデータを取得
p cache['key']
```

# Kai: Testing

- 実行方法
  - *make test*
- 実装
  - *common_test*
    - 組み込みのテストフレームワーク
  - テスト用サーバ
    - 現在はスクラッチから書いている
    - *test_server* とやらが使えるのか?

# Kai: Miscellaneous

- Node ID

```
# 内部 API に用いるソケットアドレスでノードを識別する
{Addr, Port} = {{192,168,1,1}, 11011}.
```

- データ構造

```
# このデータ構造はメタデータだけでなくデータ本体を含む
-record(data,     {key, bucket, last_modified, checksum, flags, value}).

# これは，キーやバケット番号，MD5チェックサムなどのメタデータのみを含む
-record(metadata, {key, bucket, last_modified, checksum}).
```

- 戻り値のルール

```
# Key に対応するデータがないときは ‘undefined’ を返す
undefined = kai_store:get(Key).

# エラーが発生したら，’error’ というタグに理由 Reason を付けて返す
{error, Reason} = function(Args).
```

# Kai: Roadmap

1. 基本部分の実装
   - 現状

2. ほとんどDynamo

| Module | Task |
|---|---|
| kai_hash | 読み取りにおける並列アクセスを許可 |
| kai_coordinator | クライアントからのリクエストをコーディネータに転送 |
| kai_version | Vector clocks |
| kai_sync | バルク転送，並行ダウンロード |
| kai_store | 容量制限のない永続的なストレージ |
| kai_api | プロセス・プール |
| kai_memcache | プロセス・プール，cas |

# Kai: Roadmap, cont'd

3. Dynamo

| Module | Task |
|---|---|
| kai_hash | 物理配置 |
| kai_membership | Chord あるいは Kademlia |
| kai_sync | Merkel tree |
| kai_store | 遅延削除 |
| kai_api | 接続プール |
| kai_memcache | stats |

- 開発環境など
  - *configure, test_server*

# Conclusion

- 開発者募集中です

http://sourceforge.net/projects/kai/