2009-02-20 1st Key-Value Store Workshop

たけまる

Kai = (Dynamo + memcache API) / Erlang

Takeru INOUE

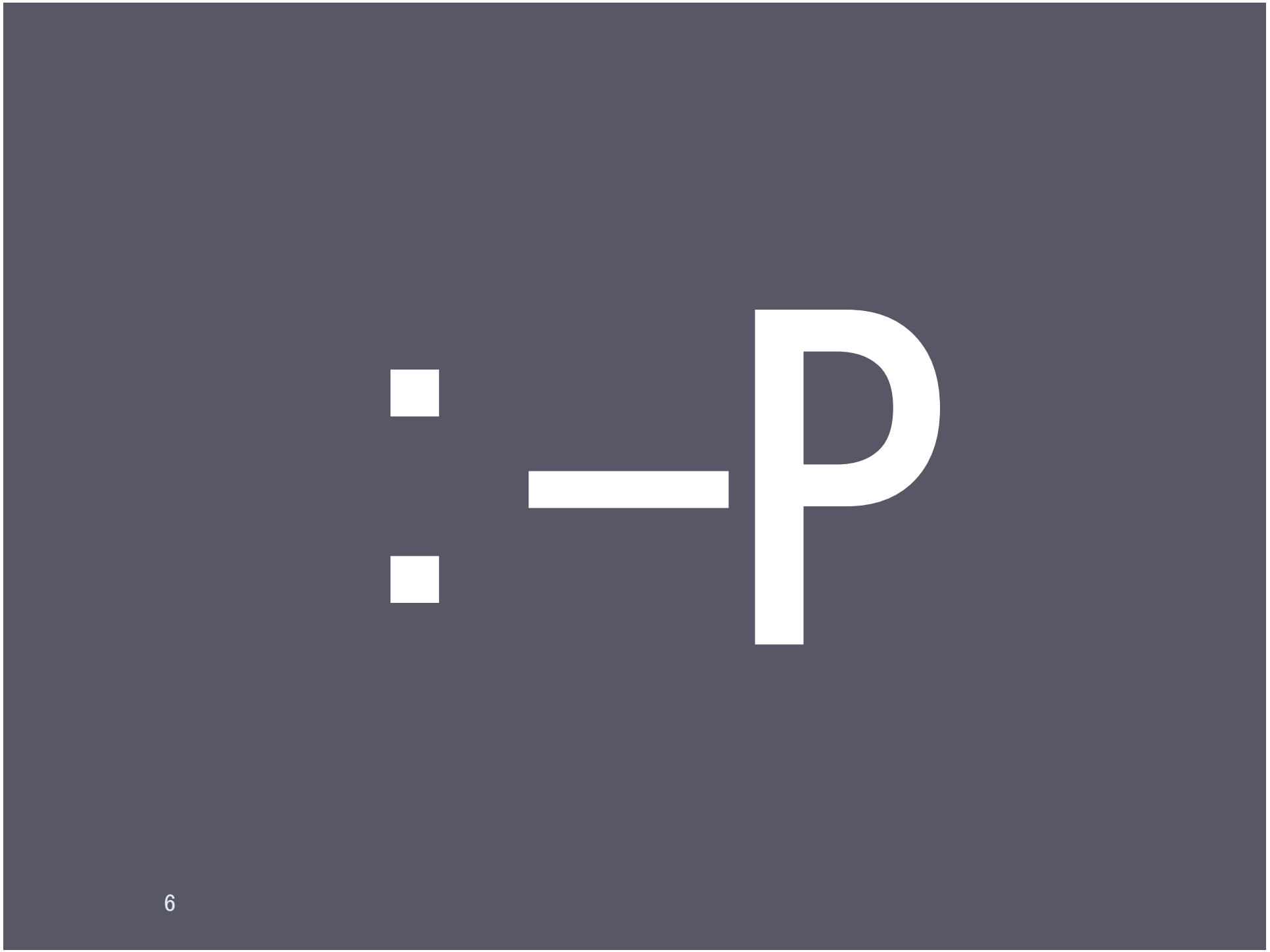
Kaiとは?

Dynamo + memcache API

Erlang

以上


質問？


: -P

自己紹介

- 井上 武（たける）
  - http://teahut.sakura.ne.jp/
- 主な仕事
  - 分散システムの研究開発
    - マルチキャストとその応用システム
    - Web アーキテクチャとデータストア
  - オープンソース
    - AtomPub (Perl)
    - Kai (Erlang)
      - Erlang 分散システム勉強会を開催（次回は5~6月頃?）

Kaiとは?

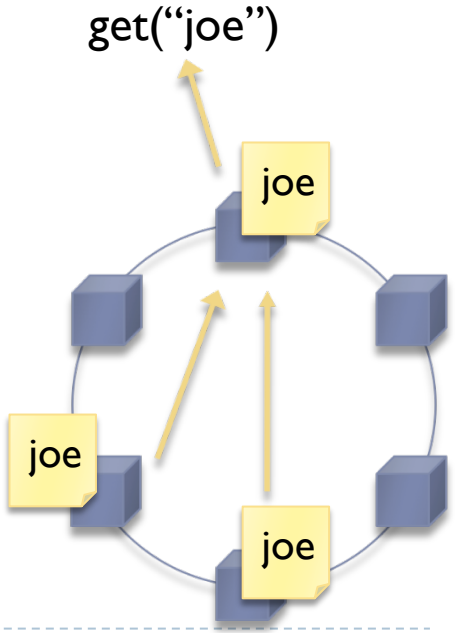
Dynamo + memcache API

Erlang

Dynamo の特徴（1/2）

- Key, valueデータストア
  - 分散ハッシュテーブルっぽい
- 高い分散透過性
  - Peer-to-peer アーキテクチャ
  - クラスタ管理コストの抑制
- 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

amazon.com

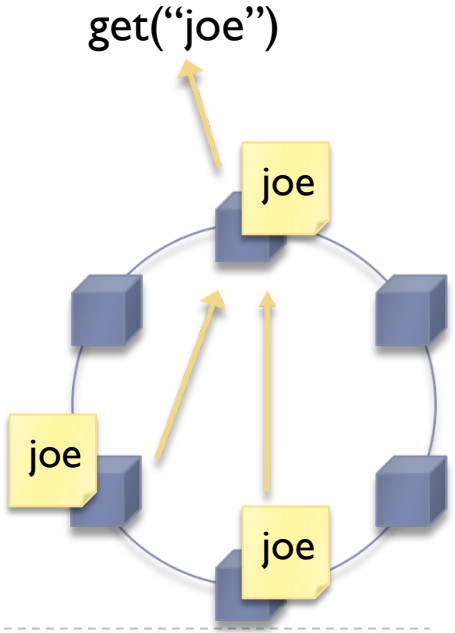


Dynamo の特徴（2/2）

- 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
  - 不整合はあとで解消

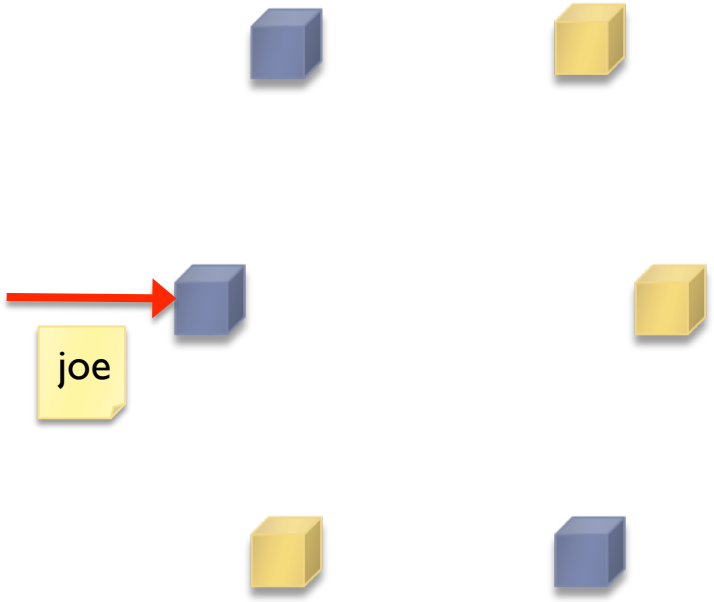
amazon.com

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



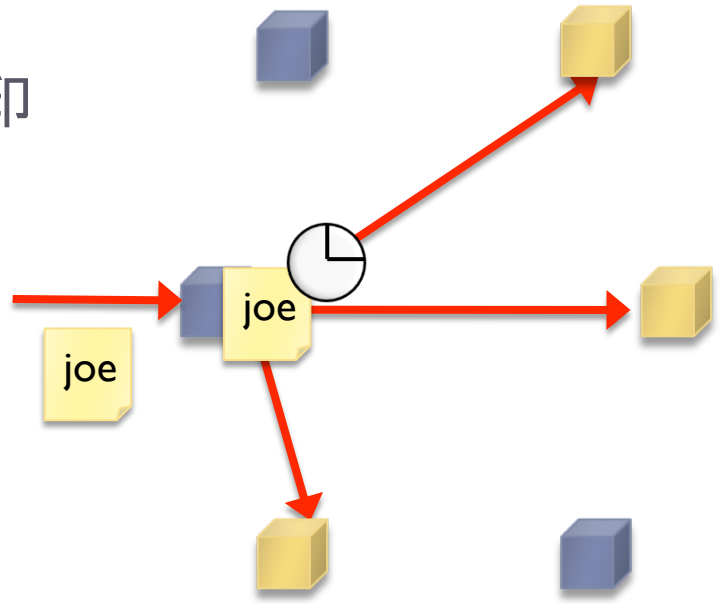
Eventually Consistent – 書き込みの振る舞い

- 3レプリカを並行して書き込み
  - Consistent Hashing で選択



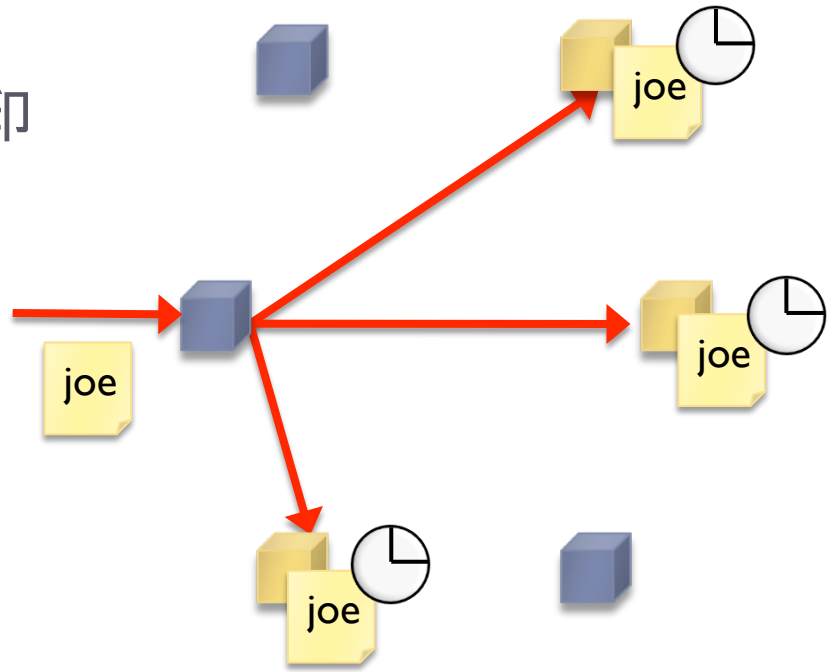
Eventually Consistent – 書き込みの振る舞い

- 3レプリカを並行して書き込み
  - Consistent Hashing で選択
- 分散ロック・コミットなし
  - ベクトルタイムスタンプを刻印
  - 不整合は読み取り時に解消
- 高い可用性!



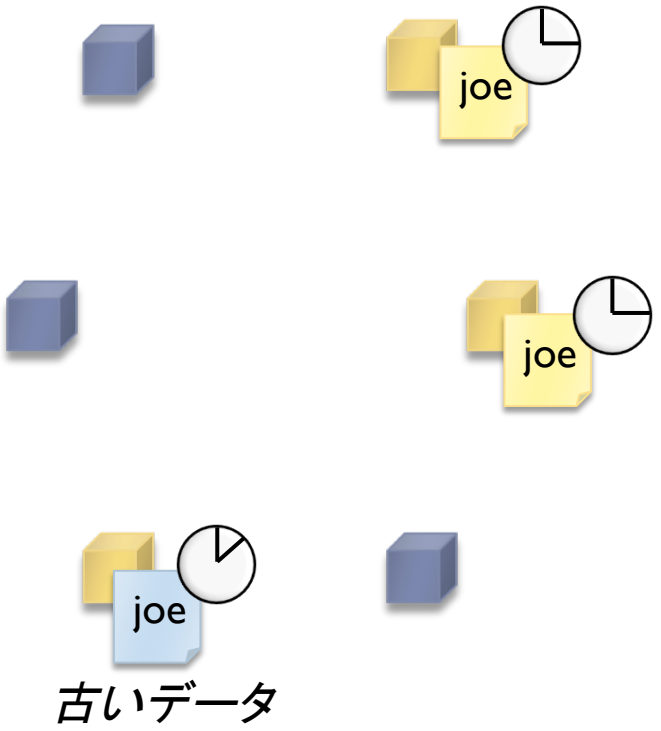
Eventually Consistent – 書き込みの振る舞い

- 3レプリカを並行して書き込み
  - Consistent Hashing で選択
- 分散ロック・コミットなし
  - ベクトルタイムスタンプを刻印
  - 不整合は読み取り時に解消
- 高い可用性!



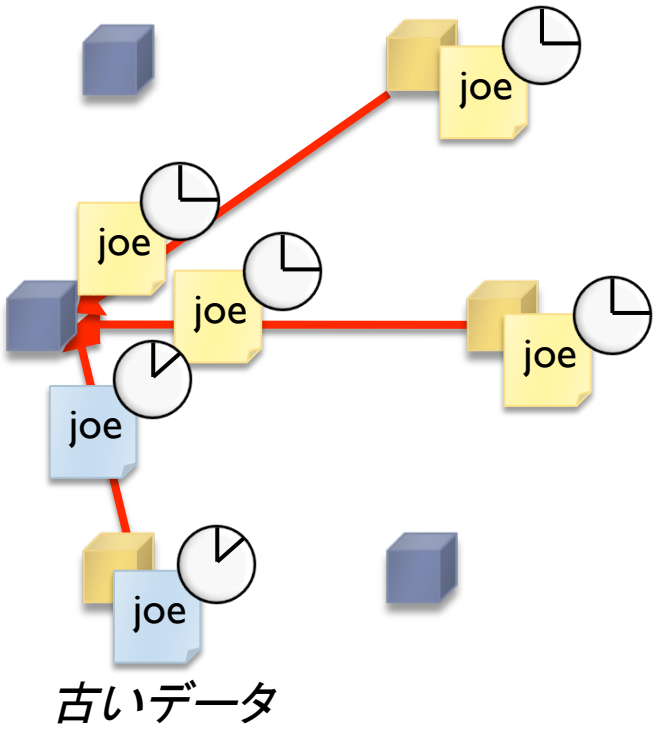
Eventually Consistent – 読み取りの振る舞い

- 3レプリカから並行して読み取り
  - Consistent Hashing で選択



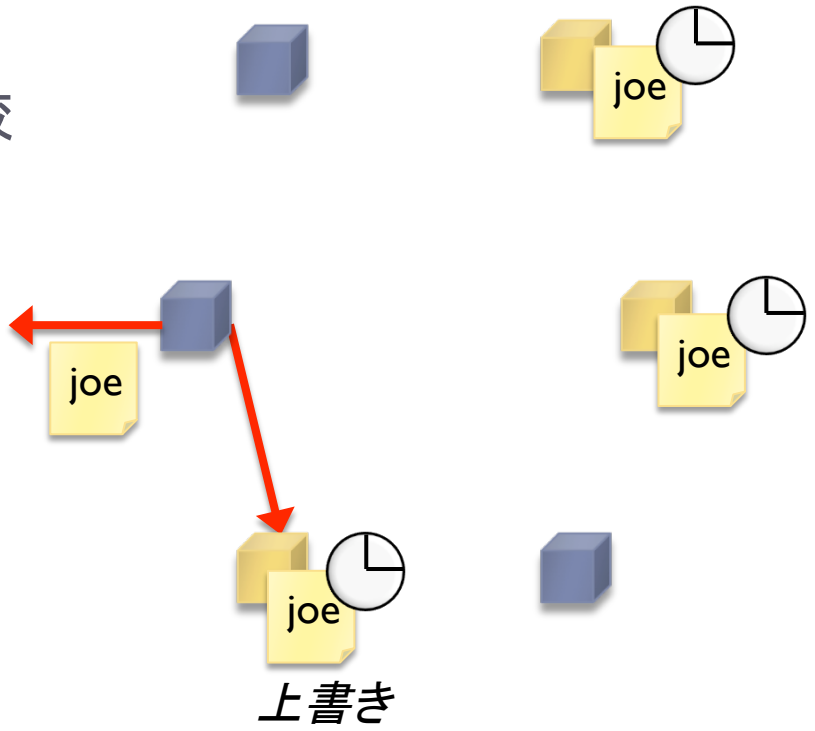
Eventually Consistent – 読み取りの振る舞い

- 3レプリカから並行して読み取り
  - Consistent Hashing で選択
- バージョン不整合を解消
  - ベクトルタイムスタンプを比較



Eventually Consistent – 読み取りの振る舞い

- 3レプリカから並行して読み取り
  - Consistent Hashing で選択
- バージョン不整合を解消
  - ベクトルタイムスタンプを比較
  - 古いデータを上書き
- 結果的に整合性がとれた
  - Eventually Consistent!

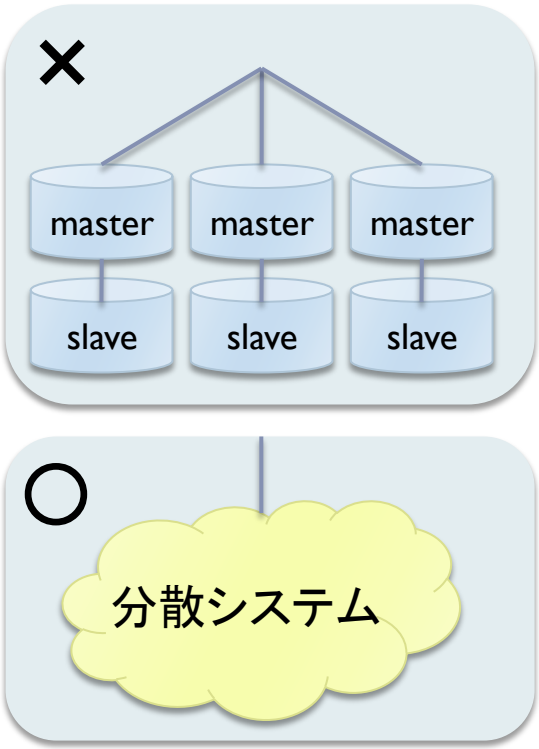


分散透過性とは?

- 分散透過性とは?
  - 分散していることの隠蔽度
    - Peer-to-peer アーキテクチャは透過性が高い(ことが多い)
  - 透過性が高いと管理コストは低下
    - 自律的なクラスタシステム
    - クライアント(アプリケーションサーバ)の簡易化
  - 性能とのトレードオフに注意
    - 自律制御のためのオーバヘッドあり
    - *日本のWebサイトは透過性より性能を優先する傾向あり?*

分散透過性の種類

- 場所
  - × クライアントがデータとノードの対応関係を保持
    - パーティショニング情報を管理
  - ○ システムが個々のデータの位置を隠蔽
    - どのノードにアクセスしてもよい
- 移動
  - × ノード追加・離脱時にクライアントがデータを移動
    - 責任クライアントの選択と信頼性に課題
  - ○ システムが適切に移動
- 障害
  - × クライアントが障害を検知して回避
  - × 性能の急激な低下（Master-Slave だと半減）
  - ○ システムが障害装置を除去，軽微な性能低下
- 管理コストの低下
  - Peer-to-peer アーキテクチャ



Dynamo まとめ

- 分散 key-value store
  - 高い分散透過性
    - Peer-to-peer アーキテクチャ
    - クラスタ管理コストの抑制
  - 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）

Dynamo + memcache API

Erlang

memcache API

Perl

```
use Cache::Memcached;

# サーバを指定
my $cache = Cache::Memcached->new({
  servers => ['127.0.0.1:11211'],
});

# set: 'key' に 'value' を格納
$cache->set('key', 'value');
```

Ruby

```
require 'memcache'

# サーバを指定
cache = MemCache.new '127.0.0.1:11211'

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

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

PHP

```
<?php
# サーバを指定
$cache = new Memcache;
$cache->connect('127.0.0.1', 112

# set: 'key' に 'value' を格納
$cache->set('key', 'value');

# get: 'key' に対応するデータを取得
$cache->get('key');
?>
```

Java

```
# サーバを指定
SockIOPool pool = SockIOPool.getInstance();
pool.setServers(new String[]{"127.0.0.1:11211"});
pool.initialize();
MemCachedClien tcache = new MemCachedClient();

# set: 'key' に 'value' を格納
cache.set("key", "value");

# get: 'key' に対応するデータを取得
System.out.println((String) cache.get("key"));
```

memcache API

- クライアント実装がたくさん存在
  - 開発の手間を省略
- Not perfect, but enough
  - じつは Dynamo の機能を完全に実現するには貧弱
    - Dynamo は独自 API
  - ほとんどの場合は問題なし

Dynamo + memcache API

Erlang

Erlang の使いどころ

- Distributed key-value store に適
  - 11 のうち 5つが Erlang 製（右表）
    - *Anti-RDBMS: A list of distributed key-value stores* より（Last.fm の中の人）
  - 分散透過性の高いシステムに多い
    - Kai も紹介されてます ☺
    - *どうして日本では使われないのだろう?*

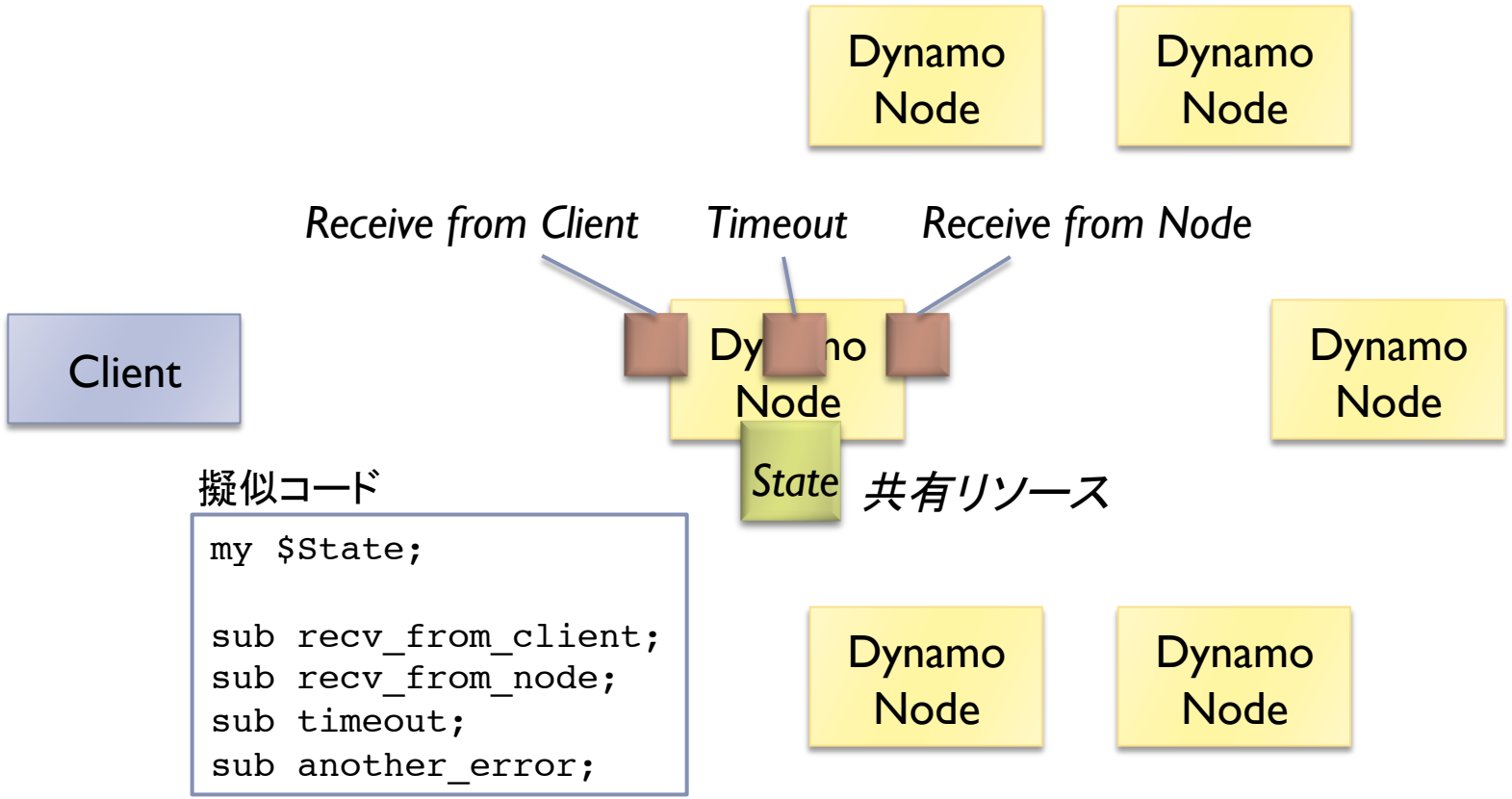
Name	Language	Fault-tolerance	Persistence	Client Protocol	Dat
Project Voldemort	Java	partitioned, replicated, read-repair	Pluggable: BerkleyDB, Mysql	Java API	Stru blol
Ringo	Erlang	partitioned, replicated, immutable	Custom on-disk (append only log)	HTTP	blol
Scalaris	Erlang	partitioned, replicated, paxos	In-memory only	Erlang, Java, HTTP	blol
Kai	Erlang	partitioned, replicated?	On-disk Dets file	Memcached	blol
Dynomite	Erlang	partitioned, replicated	Pluggable: couch, dets	Custom ascii, Thrift	blol
MemcacheDB	C	replication	BerkleyDB	Memcached	blol
ThruDB	C++	Replication	Pluggable: BerkleyDB, Custom, Mysql, S3	Thrift	Doc orie
CouchDB	Erlang	Replication, partitioning?	Custom on-disk	HTTP, json	Doc orie (jso
Cassandra	Java	Replication, partitioning	Custom on-disk	Thrift	Big me Dyn
HBase	Java	Replication, partitioning	Custom on-disk	Custom API, Thrift, Rest	Big
Hypertable	C++	Replication, partitioning	Custom on-disk	Thrift, other	Big

分散システム開発におけるErlangの特徴

- マルチコア・プログラミングの簡易化
  - アクターモデル
    - イベント駆動と比較
    - 例：3レプリカから並行して読み取り，2つを得たらレスポンス
  - 非破壊メモリ（変数への再代入不可）
- 分散システム開発の簡易化
  - 組み込みの直列化関数
  - 組み込みのノード管理システム
- そこそこ速い
  - Java未満，LL以上
  - 弱点もある（正規表現など）

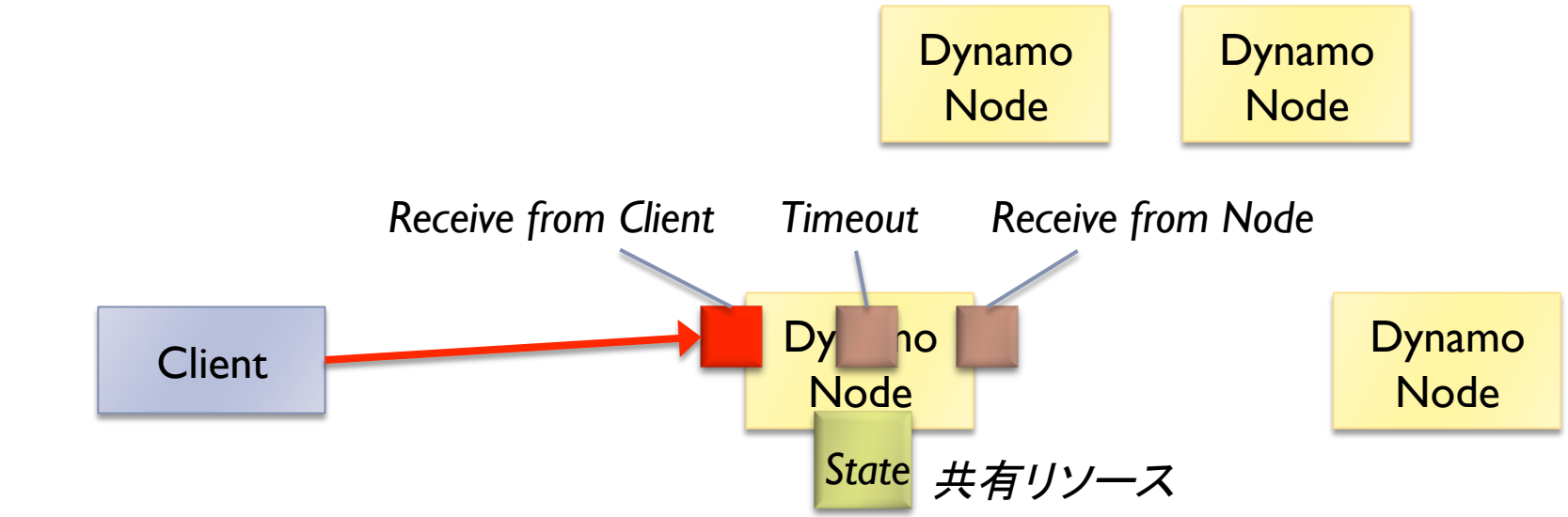
イベント駆動（Perl POE）

- イベント単位に関数を実装



イベント駆動（Perl POE）

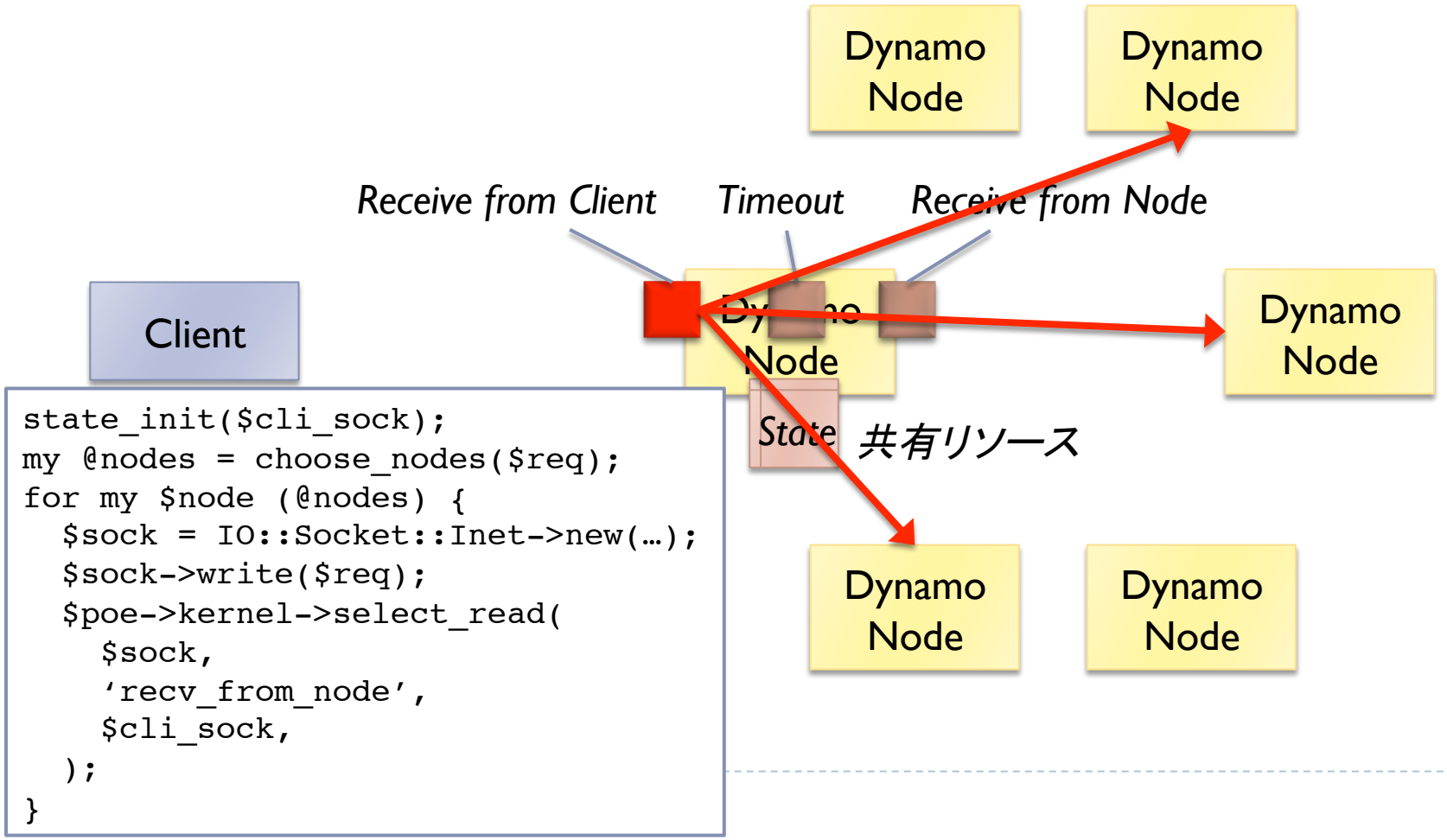
- クライアントがリクエスト




```
sub recv_from_client {
  $cli_sock->recv(my $req, 1024);
```

イベント駆動（Perl POE）

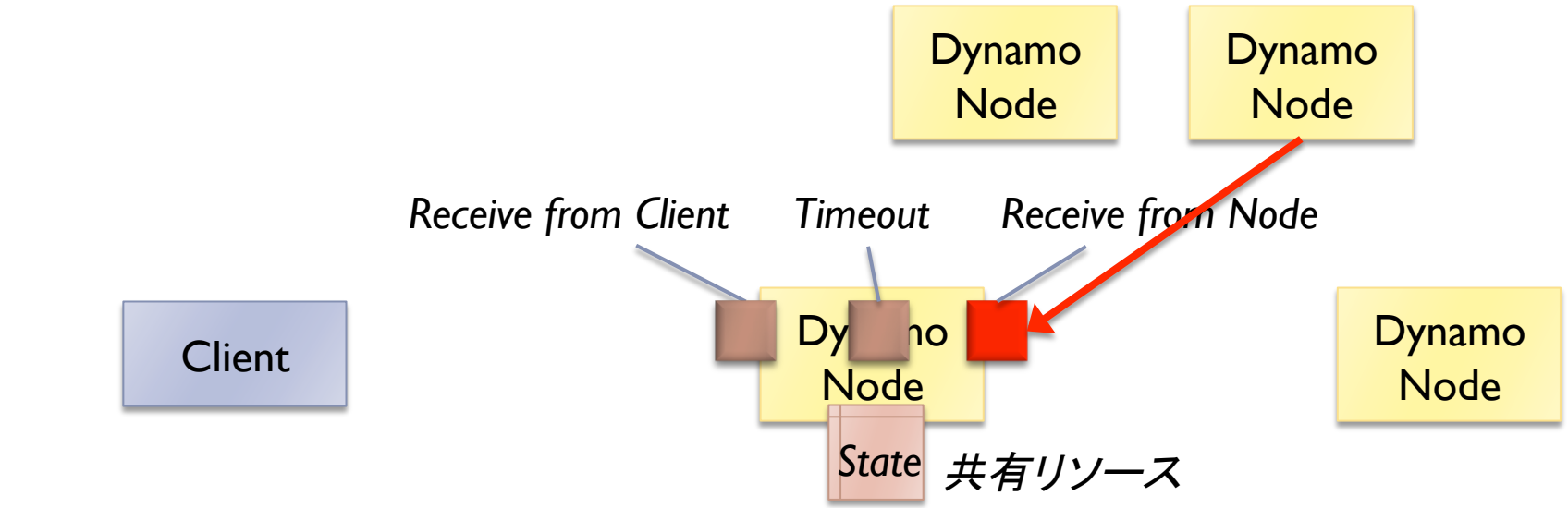
- レプリカを持っている3ノードに転送




```
state_init($cli_sock);
my @nodes = choose_nodes($req);
for my $node (@nodes) {
  $sock = IO::Socket::Inet->new(…);
  $sock->write($req);
  $poe->kernel->select_read(
    $sock,
    'recv_from_node',
    $cli_sock,
  );
}
```

イベント駆動（Perl POE）

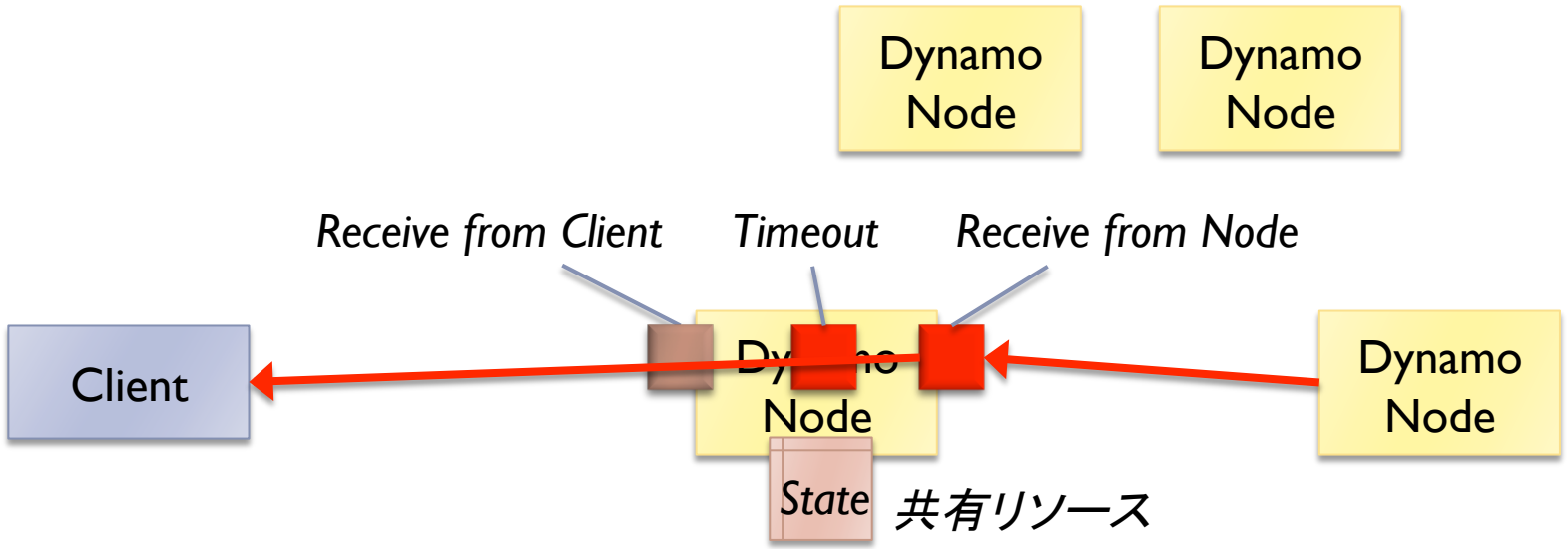
- 1つめからレスポンス




```
sub recv_from_node {
  my $cli_sock = $poe->args->[2];
  $node_sock->read(my $res, 1024);
  stat_add_res($cli_sock, $res);
  my $count = stat_count_res($cli_sock);
  if ($count == 2) {
```

イベント駆動（Perl POE）

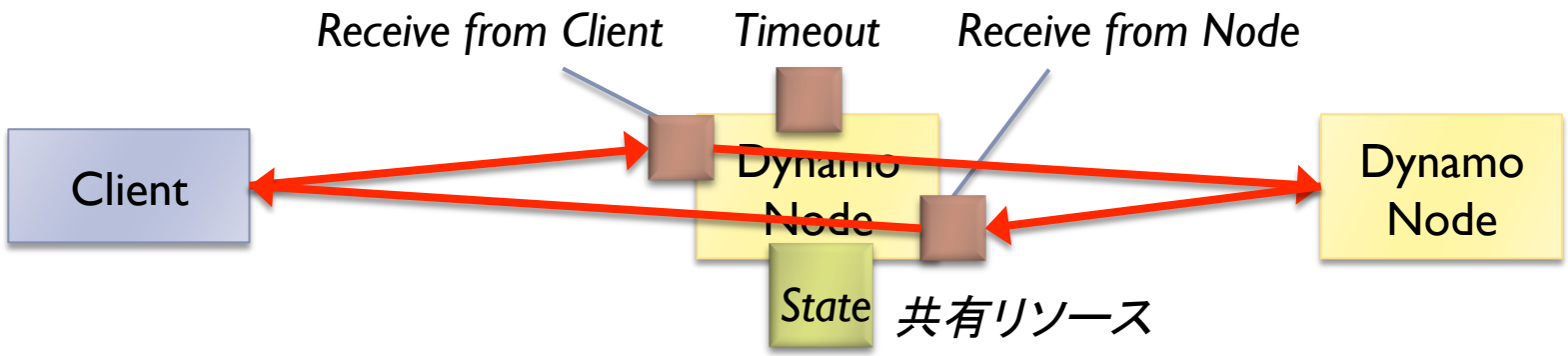
- 2つめのレスポンスでクライアントに返す
  - タイムアウトしても返す

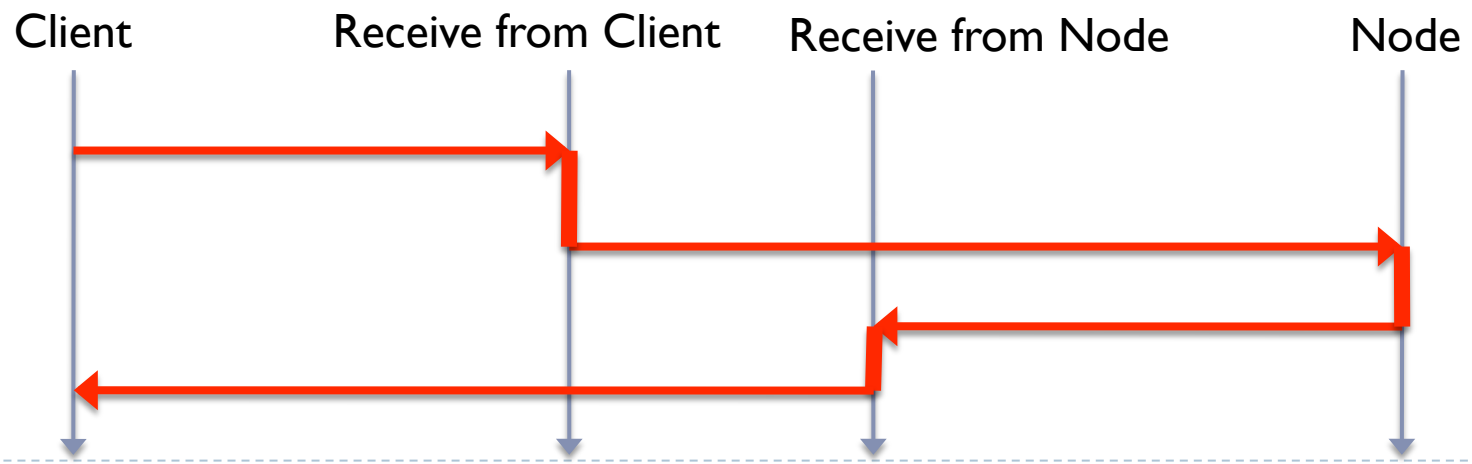



```
sub recv_from_node {
  my $cli_sock = $poe->args->[2];
  $node_sock->read(my $res, 1024);
  stat_add_res($cli_sock, $res);
  my $count = stat_count_res($cli_sock);
  if ($count == 2) {
    my $res = stat_uniq_res($cli_sock);
    $cli_sock->write($res);
  }
```

イベント駆動（Perl POE）

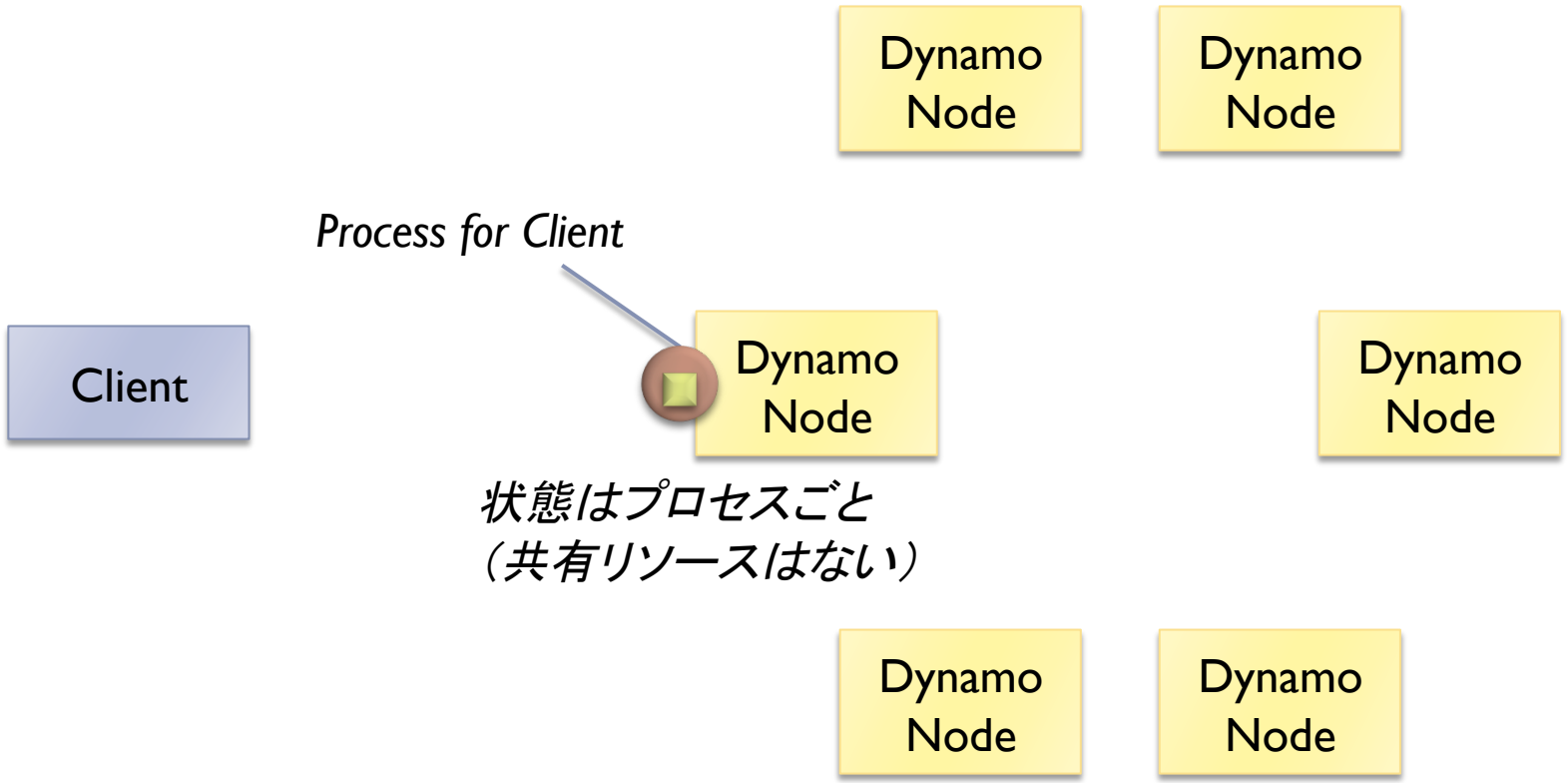
- イベント(時間の進み)単位の実装





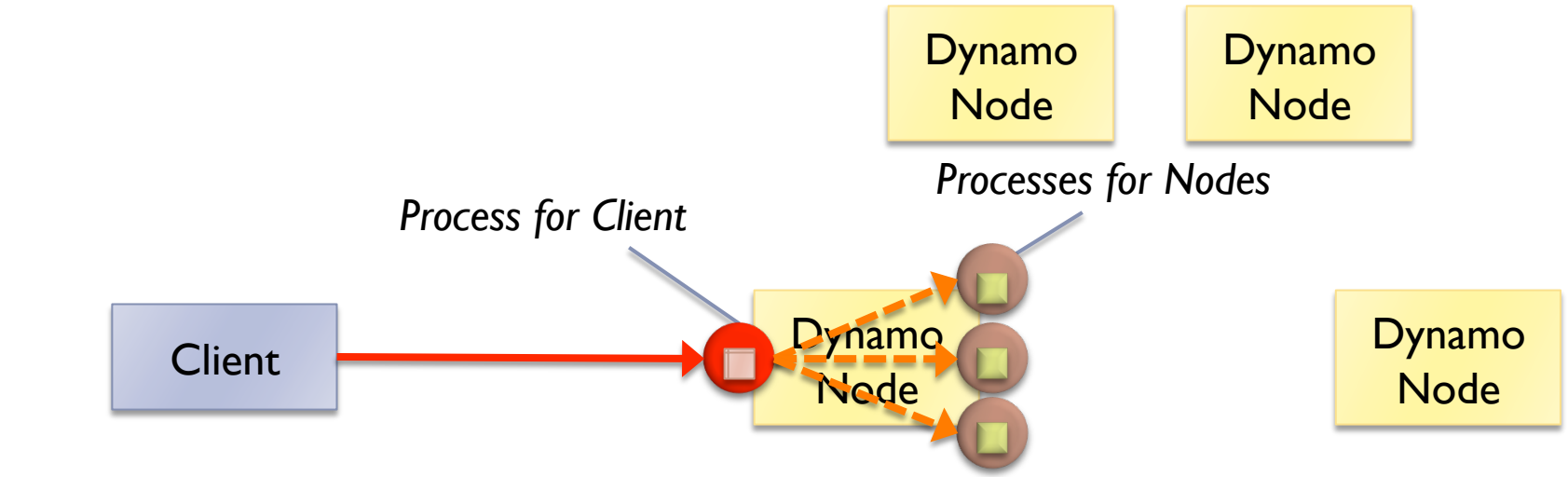
アクターモデル（Erlang）

- プロセス単位の実装



アクターモデル（Erlang）

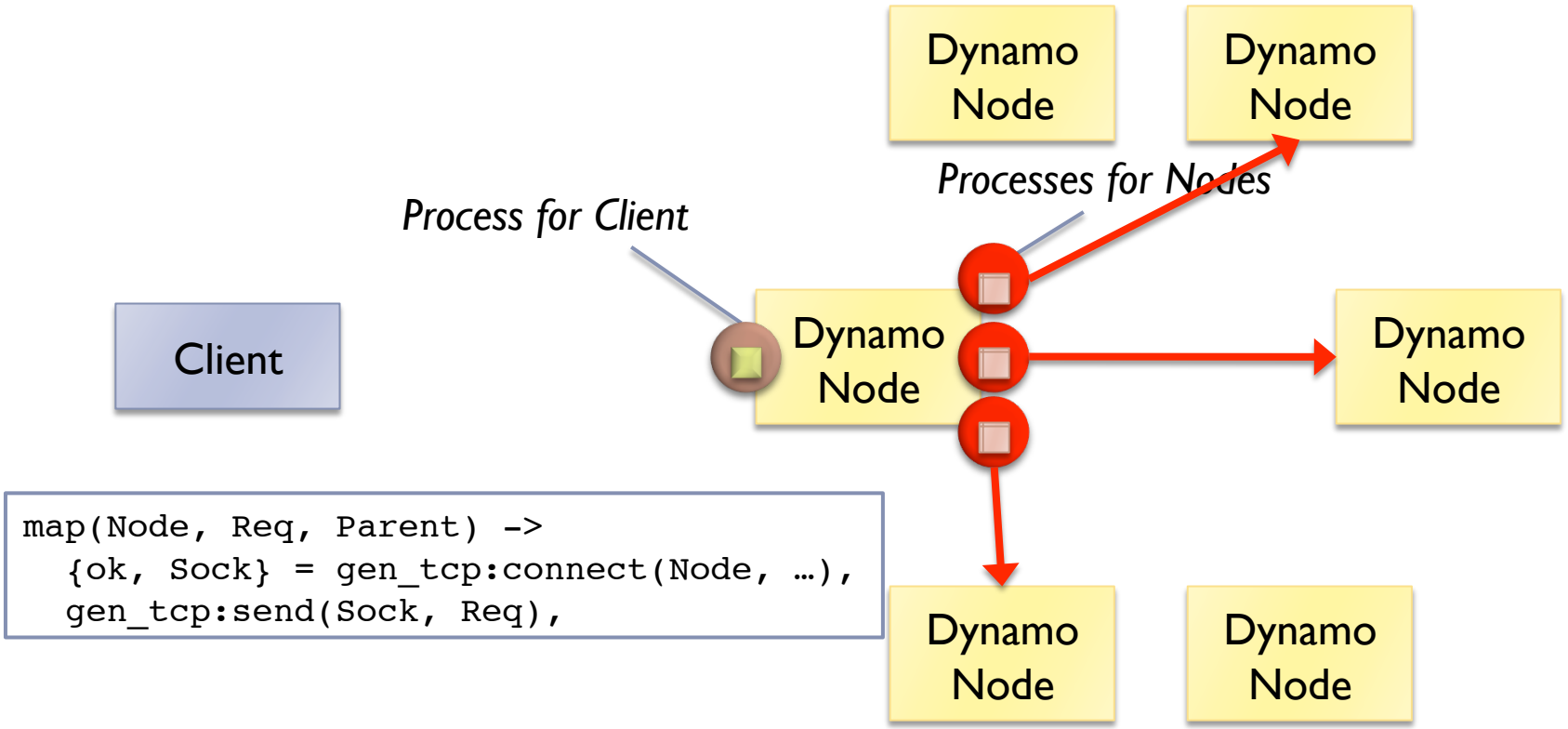
- クライアントがリクエスト




```
recv(Sock) ->
  {ok, Req} = gen_tcp:recv(Sock, 0),
  lists:foreach(
    fun(Node) -> spawn(Mod, map, [Node, Req, self()]) end,
    choose_nodes(Req)
  ),
  Res = gather(2, []),
```

アクターモデル（Erlang）

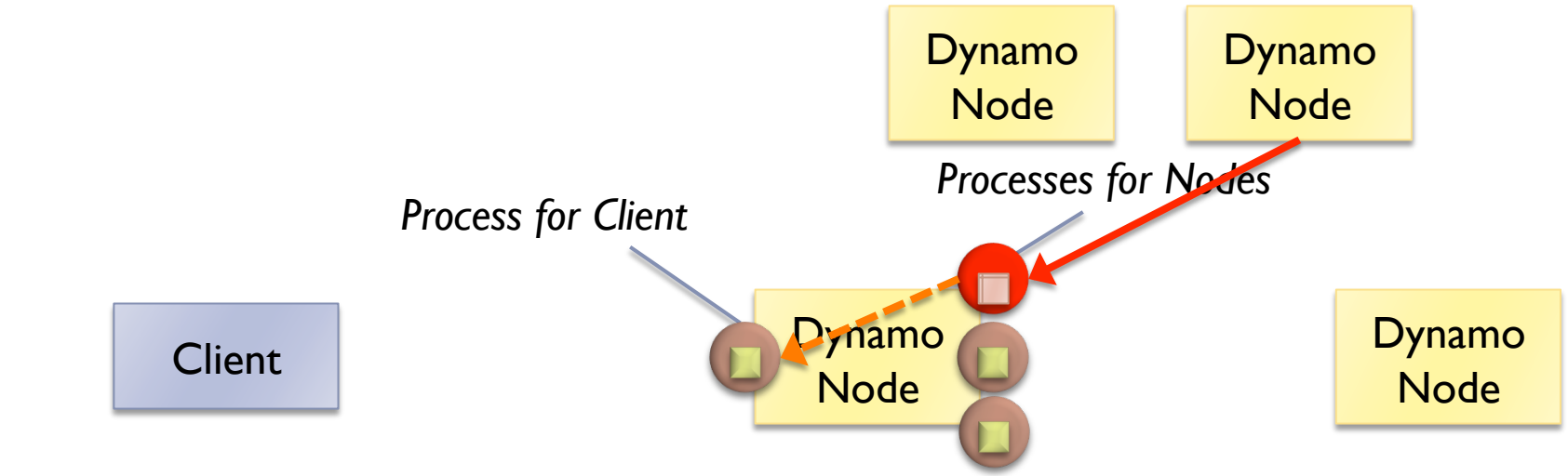
- レプリカを持っている3ノードに転送




```
map(Node, Req, Parent) ->
  {ok, Sock} = gen_tcp:connect(Node, …),
  gen_tcp:send(Sock, Req),
```

アクターモデル（Erlang）

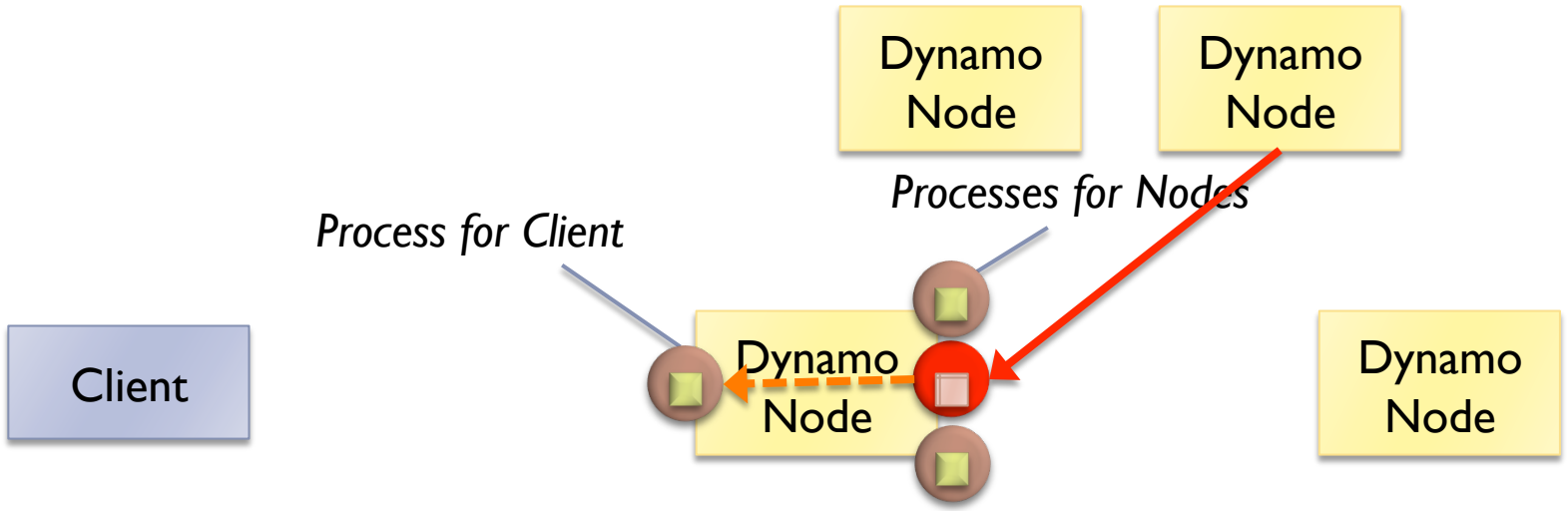
- 1つめからレスポンス




```
map(Node, Req, Parent) ->
  {ok, Sock} = gen_tcp:connect(Node, …),
  gen_tcp:send(Sock, Req),
  receive
    {tcp, Sock, Res} ->
      send(Parent, Res);
```

アクターモデル（Erlang）

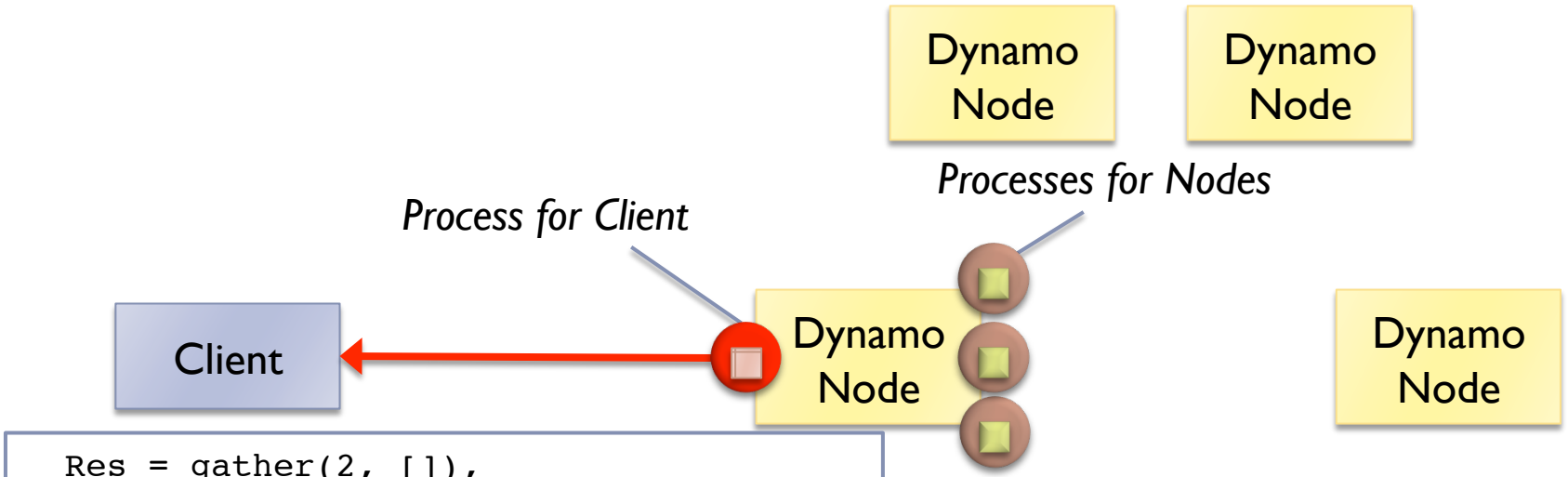
- 2つめのレスポンスでクライアントに返す
  - タイムアウトしても返す




```
map(Node, Req, Parent) ->
  {ok, Sock} = gen_tcp:connect(Node, …),
  gen_tcp:send(Sock, Req),
  receive
    {tcp, Sock, Res} ->
      send(Parent, Res);
```

アクターモデル（Erlang）

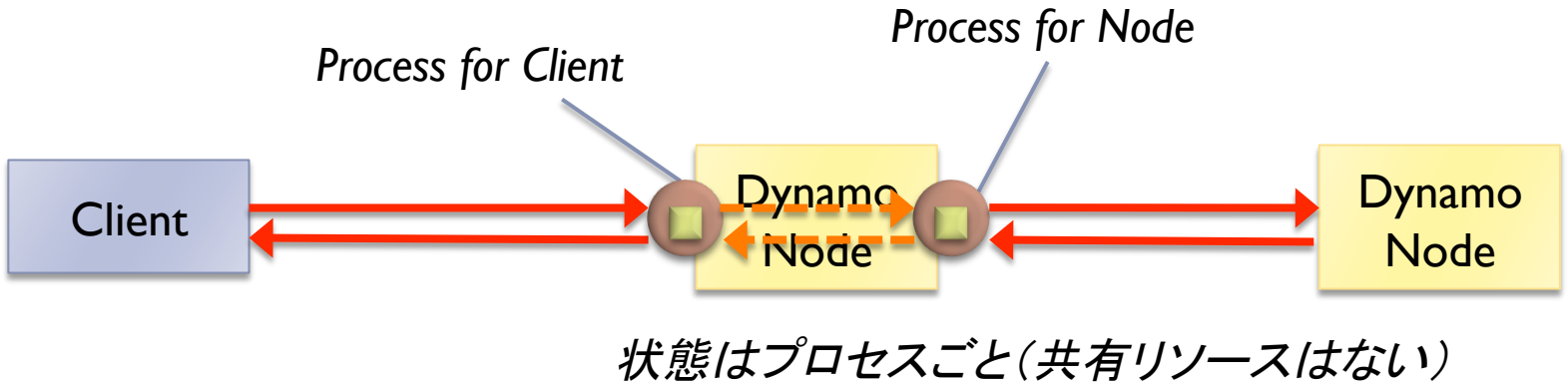
- 2つめのレスポンスでクライアントに返す
  - タイムアウトしても返す



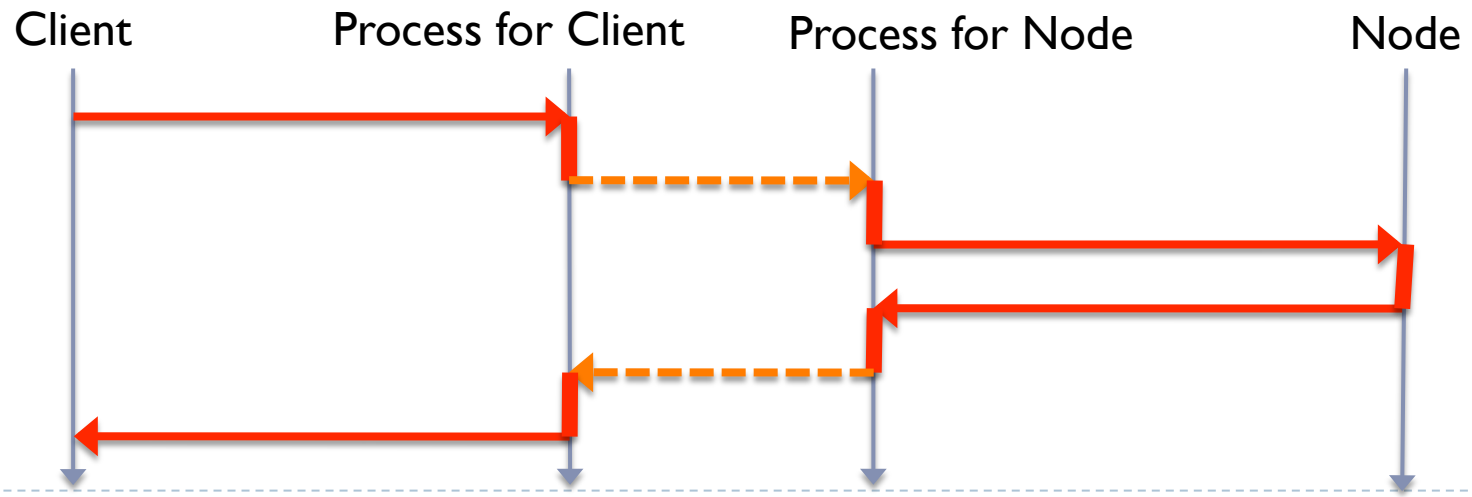

```
  Res = gather(2, []),
  gen_tcp:send(Sock, Res).

gather(0, Acc) ->
  Acc;
gather(N, Acc) ->
  receive
    Res -> gather(N-1, uniq([Res|Acc])),
  after 100 -> % timeout
    Acc
  end.
```

アクターモデル（Erlang）

- プロセス単位の実装



状態はプロセスごと（共有リソースはない）



Erlang まとめ

- マルチコア・プログラミングの簡易化
  - アクターモデル
    - 共有メモリがなく安全
    - プロセスが軽い
    - Copy on write 的にメッセージパッシングを効率化
  - 非破壊メモリ（変数への再代入不可）
- 分散システム開発の簡易化
  - 組み込みの直列化関数
  - 組み込みのノード管理システム
- そこそこ速い
  - Java未満，LL以上
  - 弱点もある（正規表現など）

宣伝

- 情報処理学会誌に Erlang 解説記事を寄稿
  - 2009年3月号

解説

プログラミング言語
Erlangの動向

井上　武　NTT未来ねっと研究所

はじめに

筆者の職場で，さまざまなプログラミング言語による
フィボナッチ数列の計算速度を比較したことがあった．
指定された長さの数列が得られるまでの時間を測定した
ところ，驚いたことに，C言語を抑えて最速の座を勝ち

速くなる．
筆者がErlangを使
アにある．以前は，
ト駆動プログラムを
アプローチでは1つ
い，マルチスレッドを
後述するように共有

Kaiとは?

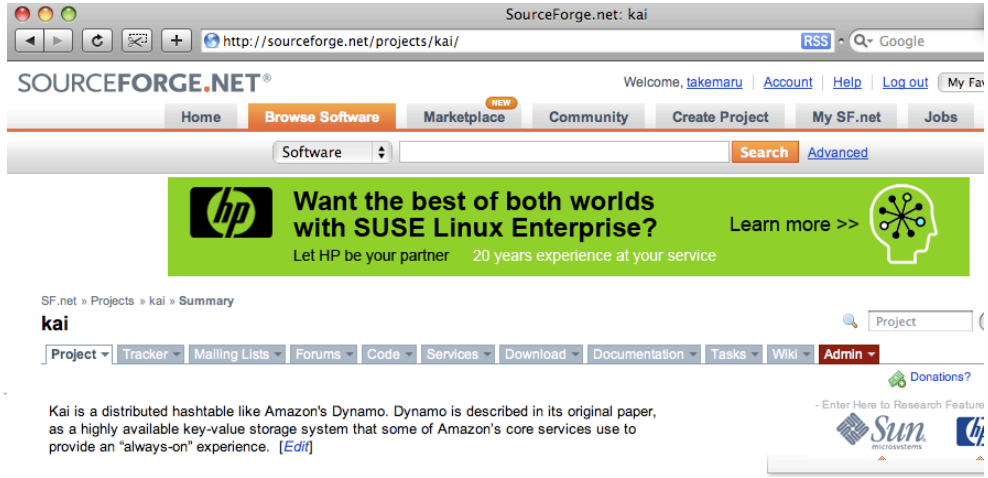
Dynamo + memcache API

Erlang

Kai の概要

- Dynamoのオープンソース実装
  - 開発者の本籍地から命名
    - 検索しにくい…
      - kie とかにしておけばよかった orz
      - erlang, dynamo, sourceforge などと組み合わせて検索
  - memcache API
  - Erlang

  - sourceforge.net で開発中
    - http://kai.sf.net/
    - 開発者 3.5名
    - 2,260行
    - 基本的な振る舞いは完了



Kai の性能

- 条件
  - Xeon 2.13 GHz x4
  - 1 KB/データ
  - メモリストレージ（ディスクも選択可）

- 1ノード（like a memcached）
  - 10,193 qps（readのみ, 複製なし）
- 5ノード
  - 14,212 qps（read:write = 80:20, 複製あり）

    - *少し前に測った値なので，いまはもう少し低いかも*

Kaiとは?

Dynamo + memcache API

Erlang

以上


質問？