Developing Amazon's Dynamo in POE and Erlang – Prologue of "From POE to Erlang"

takemaru

Dynamoとは?

- Amazon CTO の Werner Vogels らが開発

All Things Distributed

Werner Vogels' weblog on building scalable and robust distributed systems.

Amazon's Dynamo

By Werner Vogels on October 2, 2007 3:10 PM | Permalink | Comments (11) | TrackBacks (11)

In two weeks we'll present a paper on the Dynamo technology at SOSP, the prestigious biannual Operating Systems conference. Dynamo is internal technology developed at Amazon to address the need for an incrementally scalable, highly-available key-value storage system. The technology is designed to give its users the ability to trade-off cost, consistency, durability and performance, while maintaining high-availability.

Let me emphasize the internal technology part before it gets misunderstood: Dynamo is not **directly** exposed externally as a web service; however, Dynamo and similar Amazon technologies **are** used to power parts of our Amazon Web Services, such as S3.

We submitted the technology for publication in SOSP because many of the techniques used in Dynamo originate in the operating systems and distributed systems research of the past years; DHTs, consistent hashing, versioning, vector clocks, quorum, anti-entropy based recovery, etc. As far as I know Dynamo is the first production system to use the synthesis of all these techniques, and there are quite a few lessons learned from doing so. The paper is mainly about these lessons.

We are extremely fortunate that the paper was selected for publication in SOSP; only a very few true production systems have made it into the conference and as such it is a recognition of the quality of the work that went into building a real incrementally scalable storage system in which the most important properties can be

Contact Info

Werner Vogels
CTO - Amazon.com

werner@allthingsdistributed.com

Dynamoとは?

- Key-valueストア（ハッシュテーブル）
  - コレのすごいやつ！

```
my %hash = (
    key1 => “value1”,
    key2 => “value2”,
);
```

  - memcached のデータがなくならないやつ，のが正確かも

Dynamoとは?

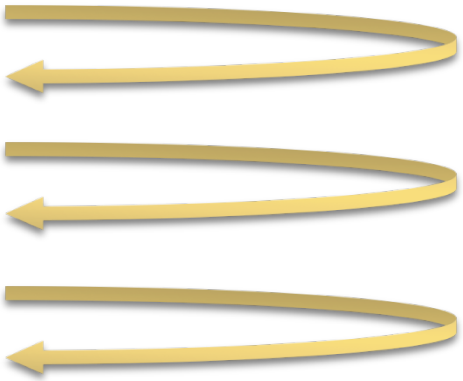
- Key-valueストア（ハッシュテーブル）
  - 簡単にスケールアウトできる（数百台とか）

Dynamoとは?

- Key-valueストア（ハッシュテーブル）
  - 障害に強い（マシン障害はもちろんラック障害にも耐える）

Dynamoとは?

- Key-valueストア（ハッシュテーブル）
  - レスポンスタイム（Latency）が安定している



< 300ms

，マシンが故障しても
ネットワーク障害が起きても

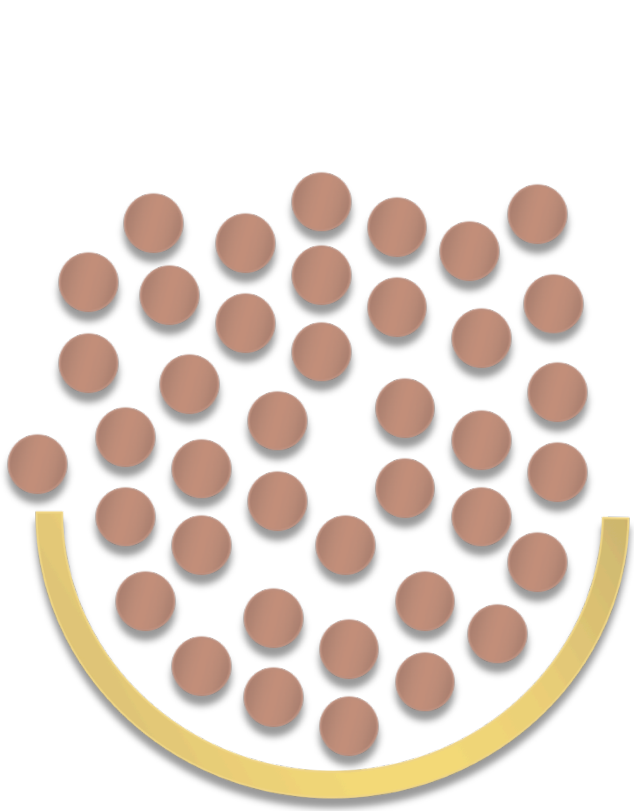
Dynamoとは?

- Key-valueストア（ハッシュテーブル）
  - いつでも読み書きできる（Lockによるストールがない）

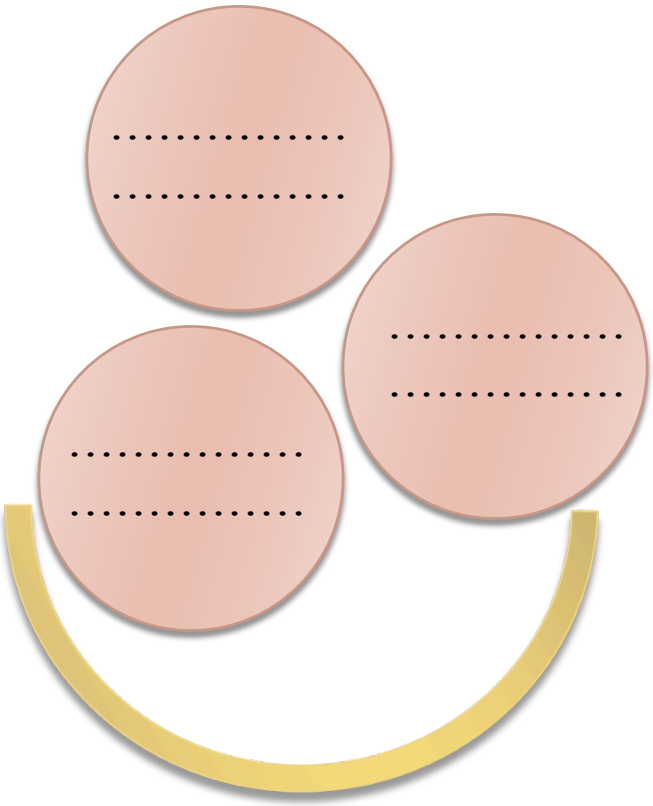
というようなことがない

Dynamoとは?

- Key-valueストア（ハッシュテーブル）
  - 小さなデータをたくさん格納するのに向いている



Dynamo



Bigtable/GFS

Kai – Yet Anothoer Amazon's Dynamo

- さて，Dynamo のオープンソース実装について
  - まず POE で

    POE: Perl Object Environment

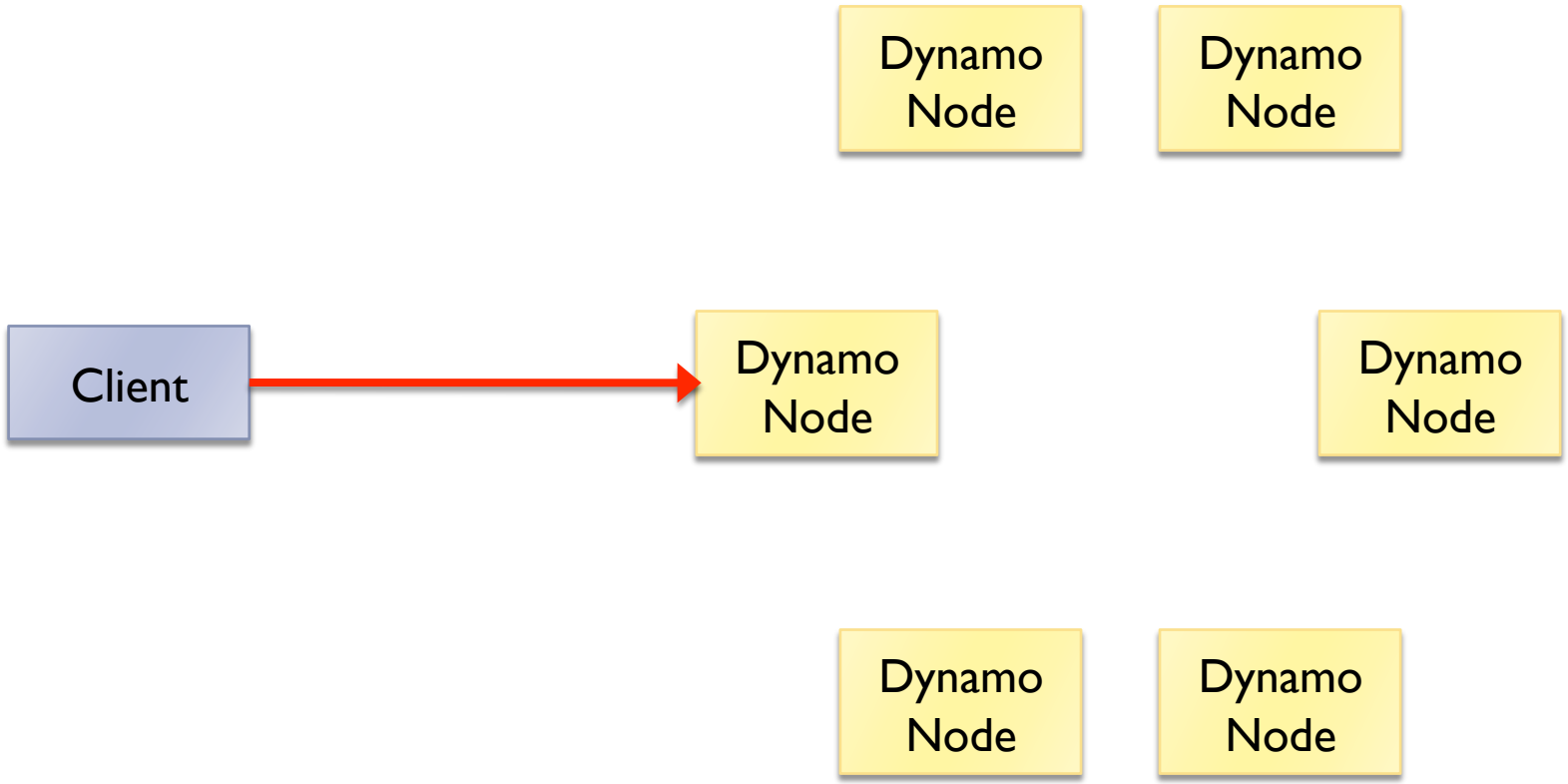
  - 次に Erlang で

memcache プロトコルから使えるよ

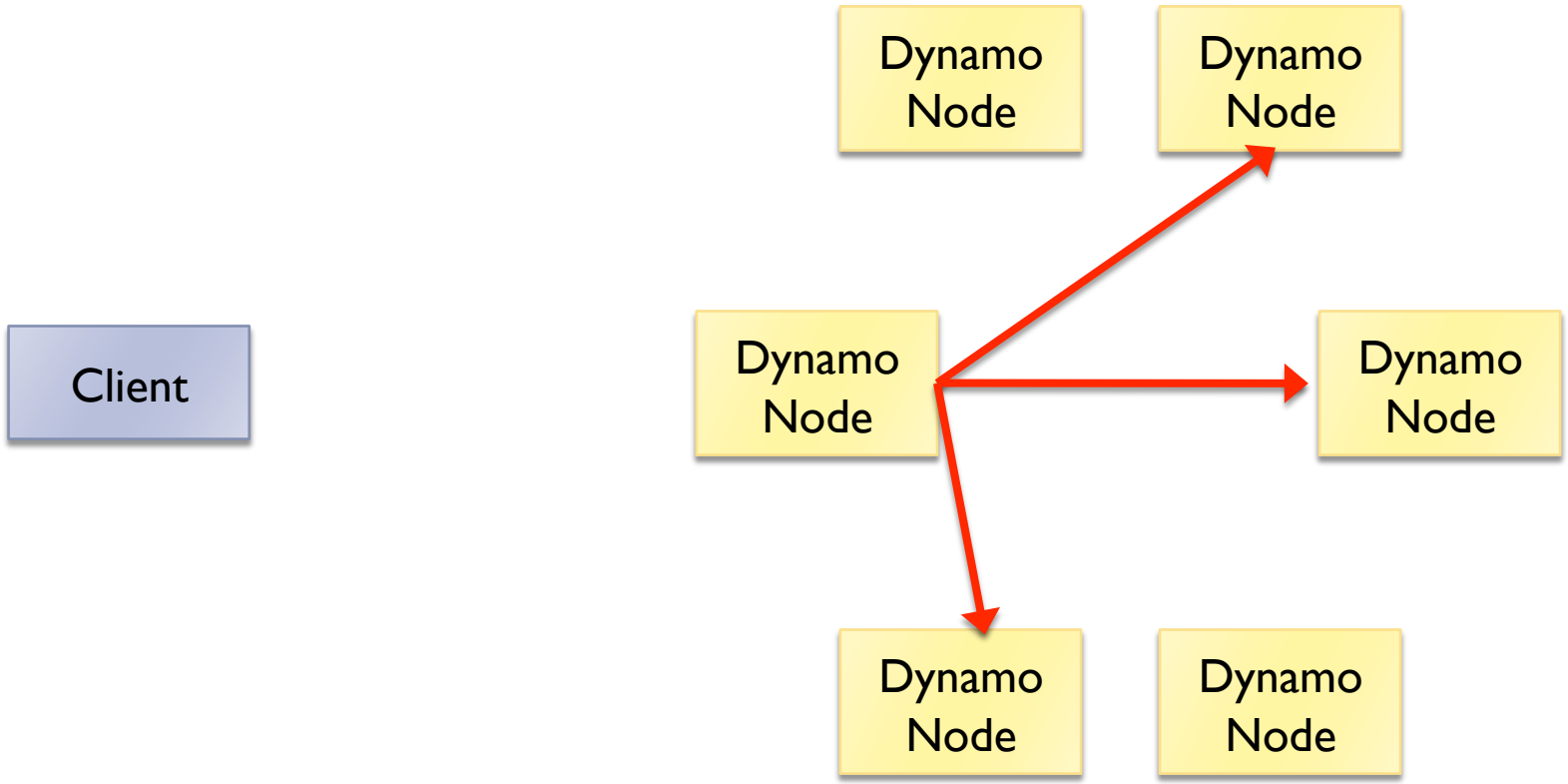
動作例

- クライアントがリクエスト



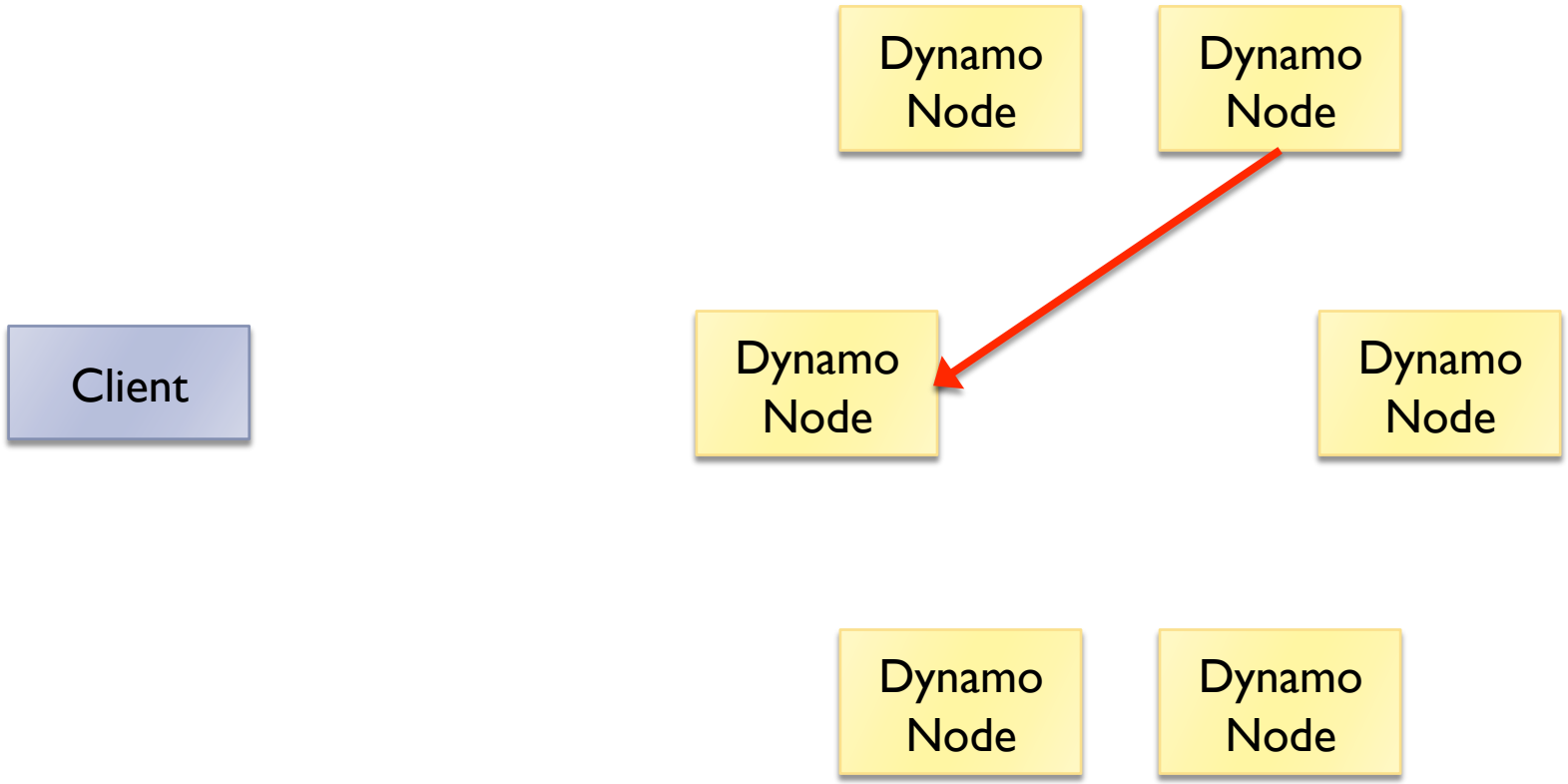
動作例

- レプリカを持っている3ノードに転送
  - Consistent Hashing で選択



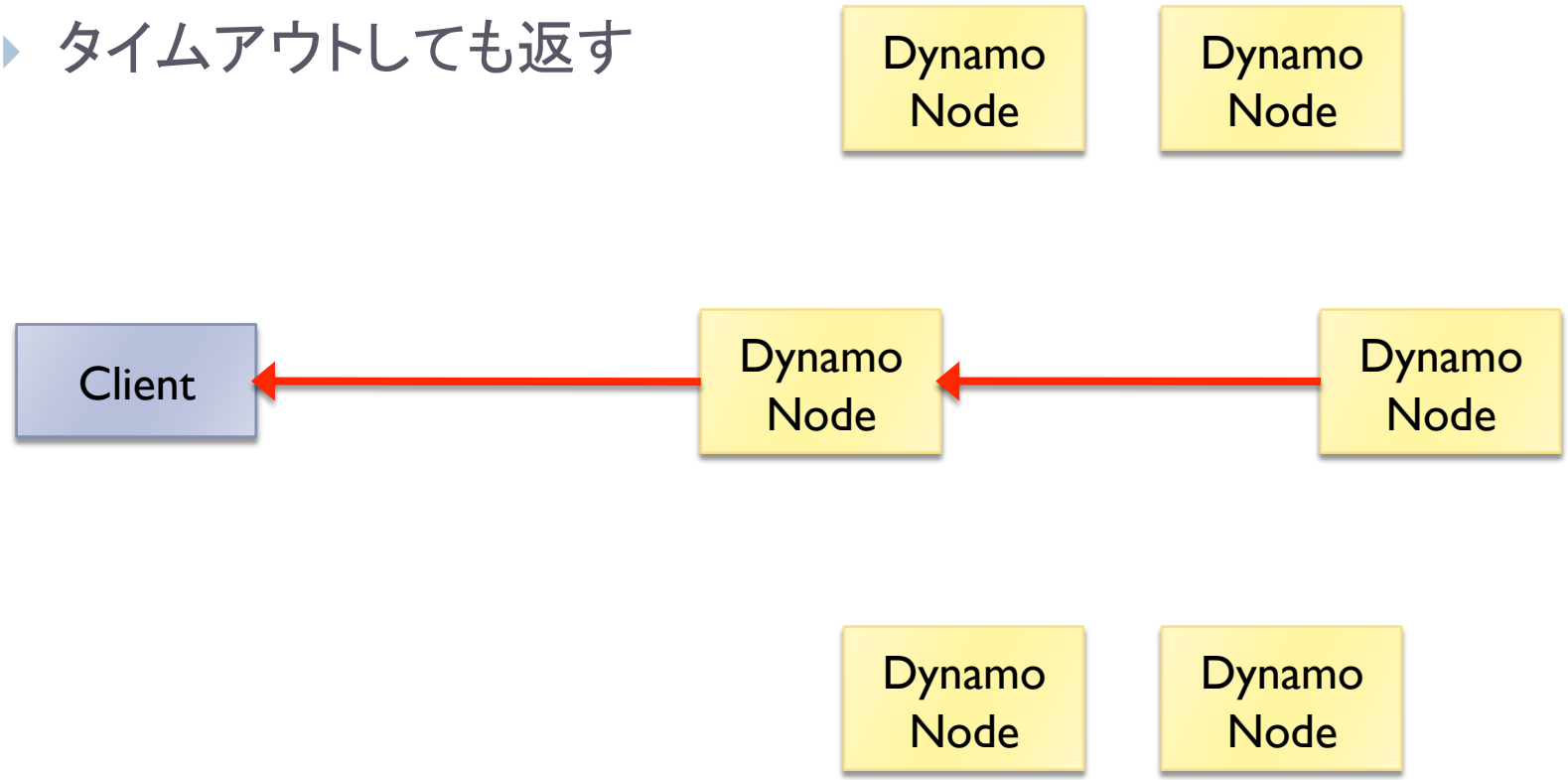
動作例

- 1つめからレスポンス



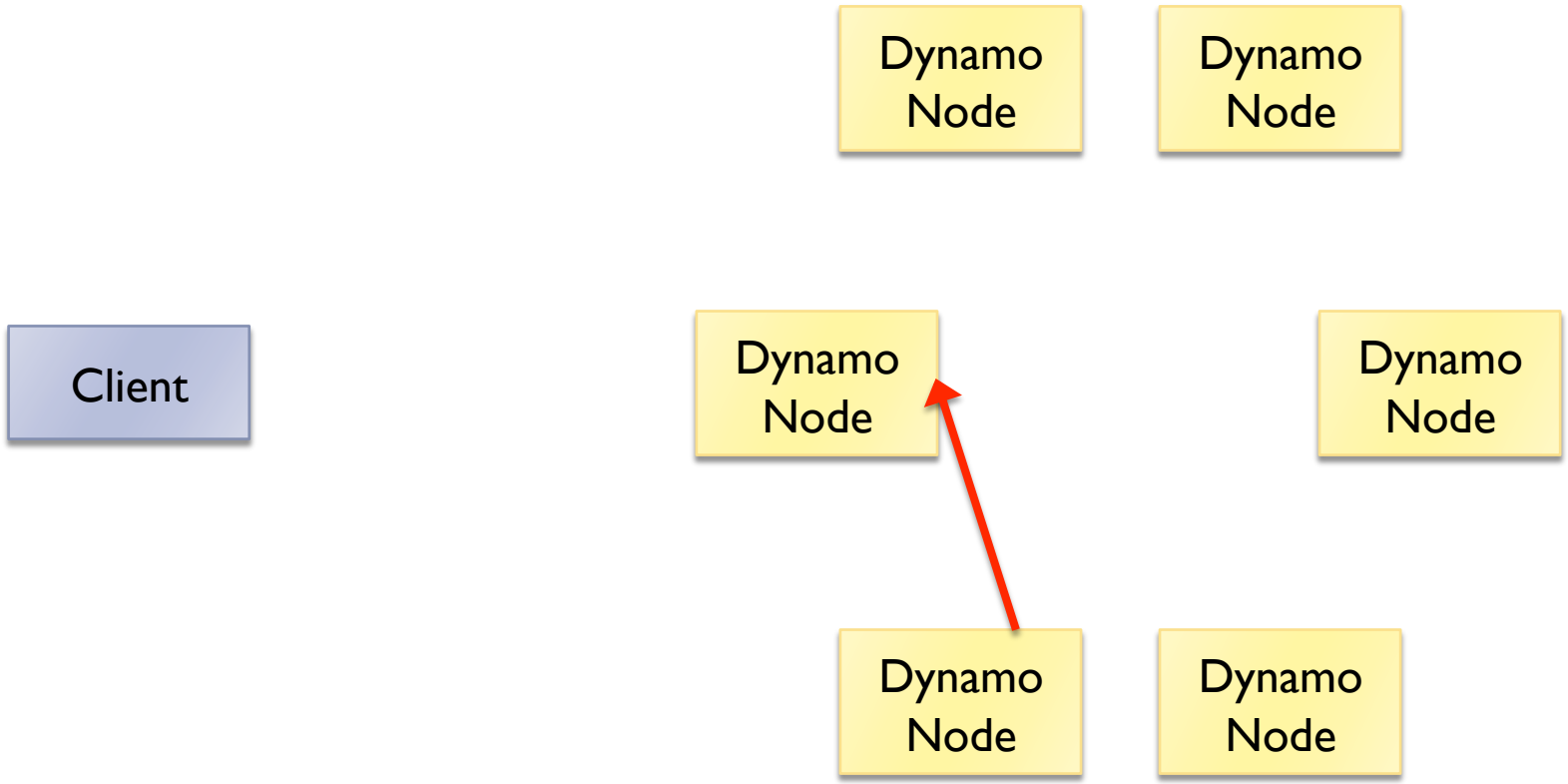
動作例

- 2つめのレスポンスでクライアントに返す
  - バージョンチェックなどを実行
  - タイムアウトしても返す



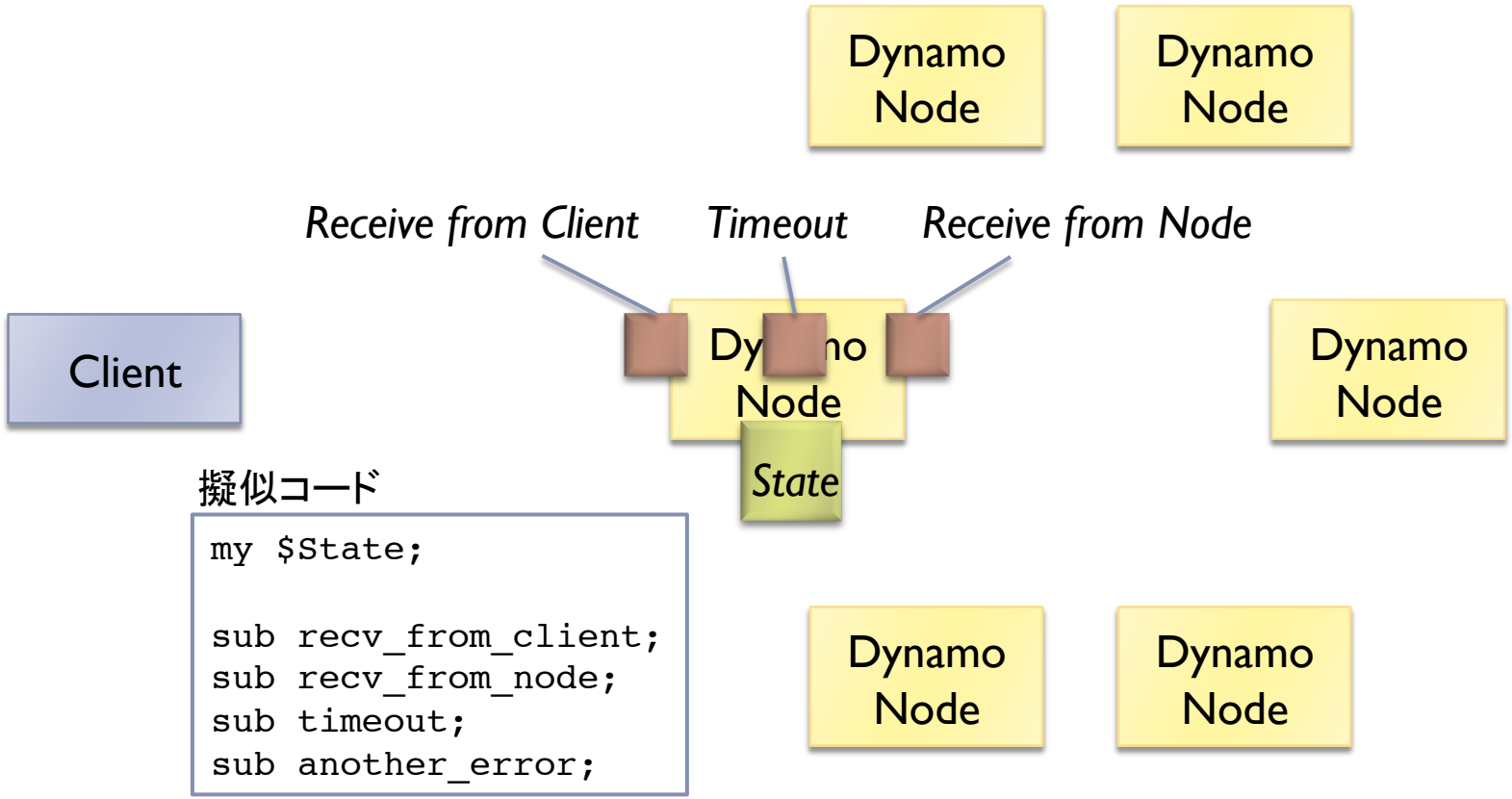
動作例

- 遅れてきたレスポンスを処理
  - バージョン修復などを実行



POE (Event Driven) 実装

- イベント単位に関数を実装



擬似コード

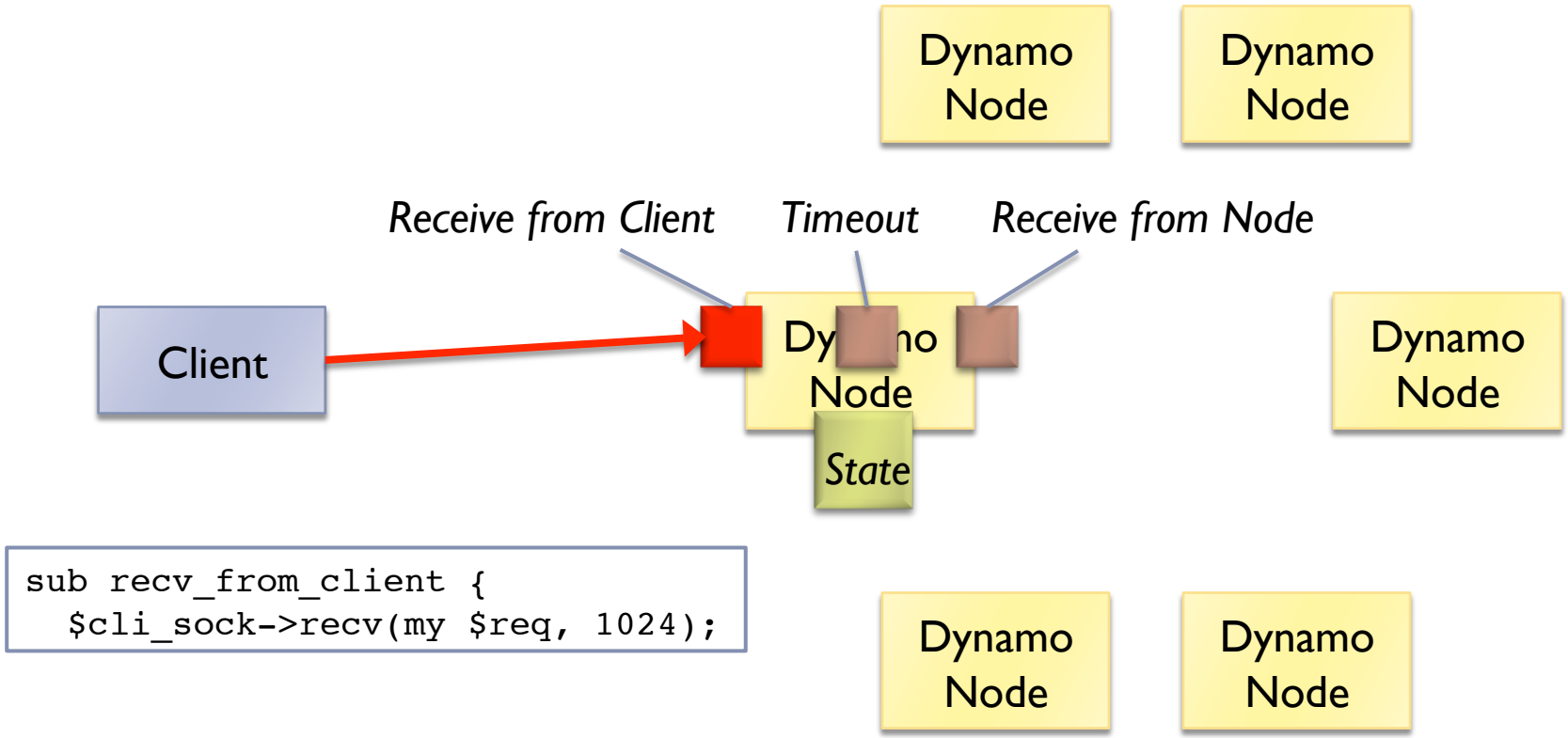
```
my $State;

sub recv_from_client;
sub recv_from_node;
sub timeout;
sub another_error;
```

実際にはちょっと違う実装をしたのですが，
まぁ普通ならこうするだろうってを紹介します

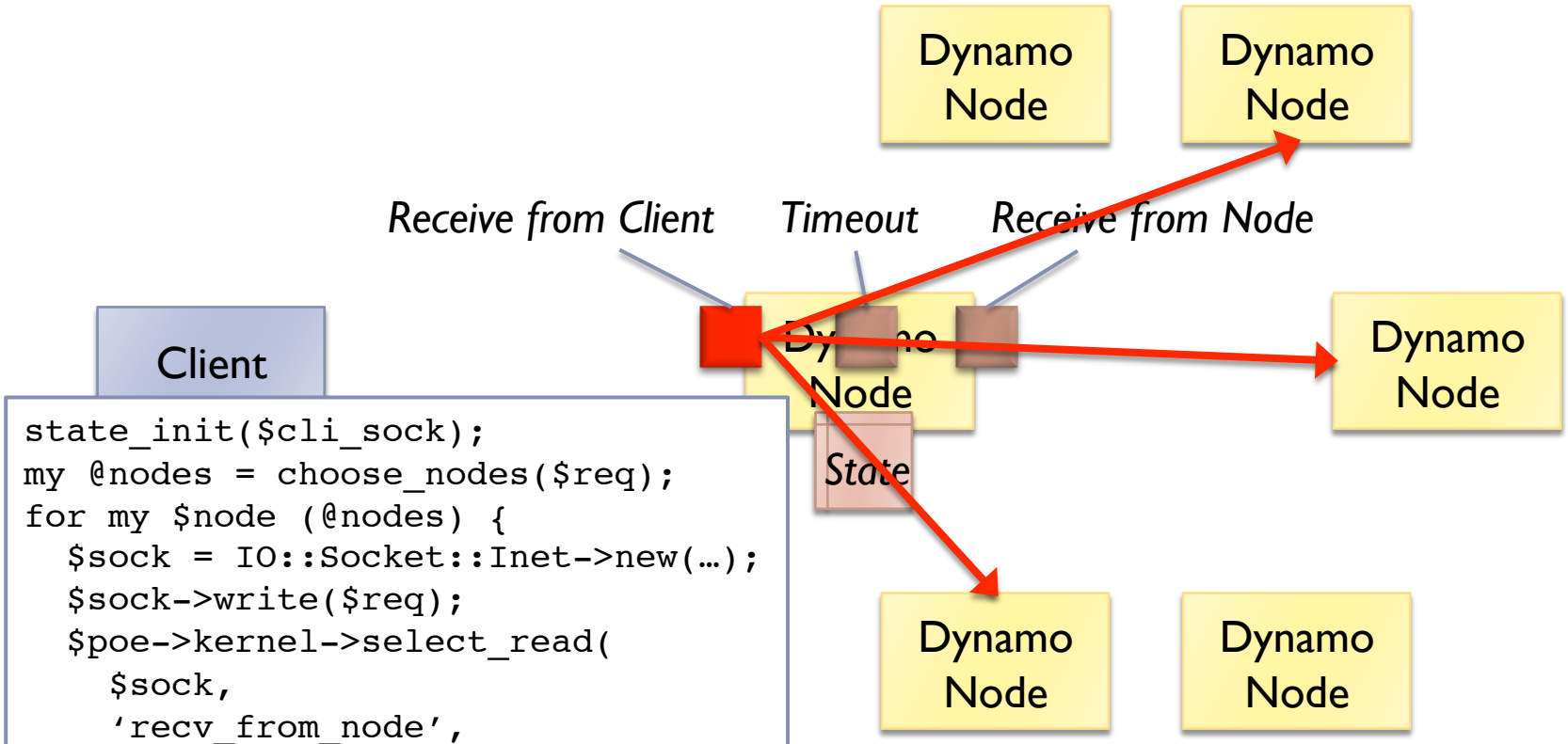
POE (Event Driven) 実装

- クライアントがリクエスト



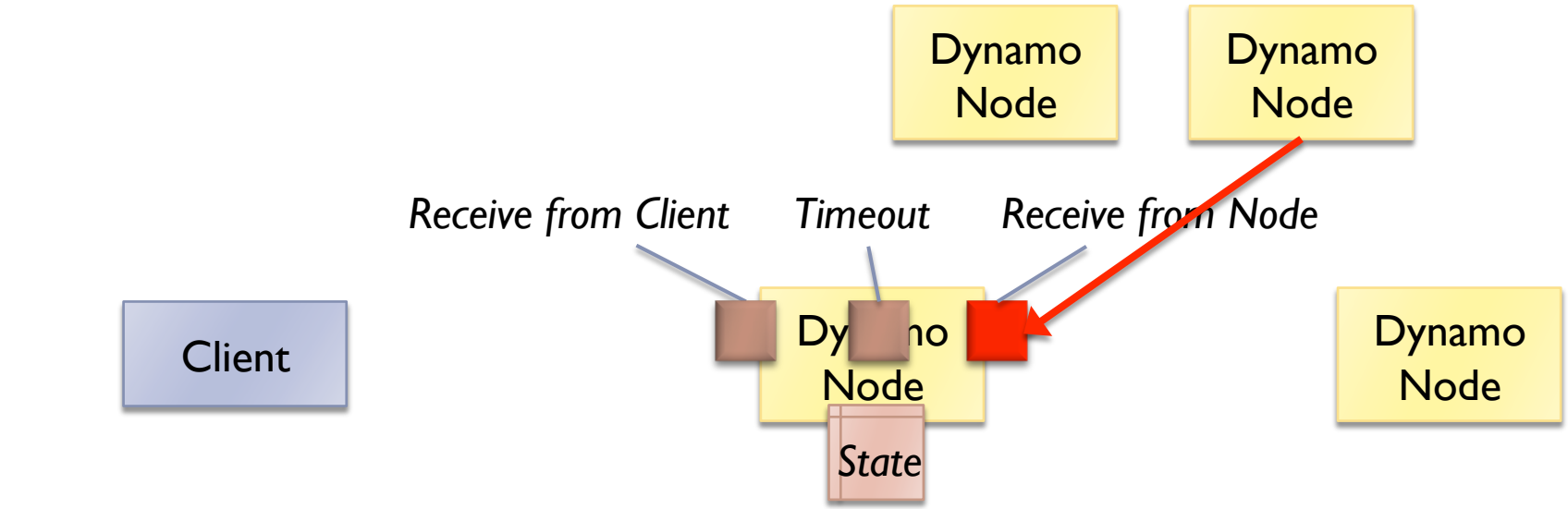
POE (Event Driven) 実装

- レプリカを持っている3ノードに転送



POE (Event Driven) 実装

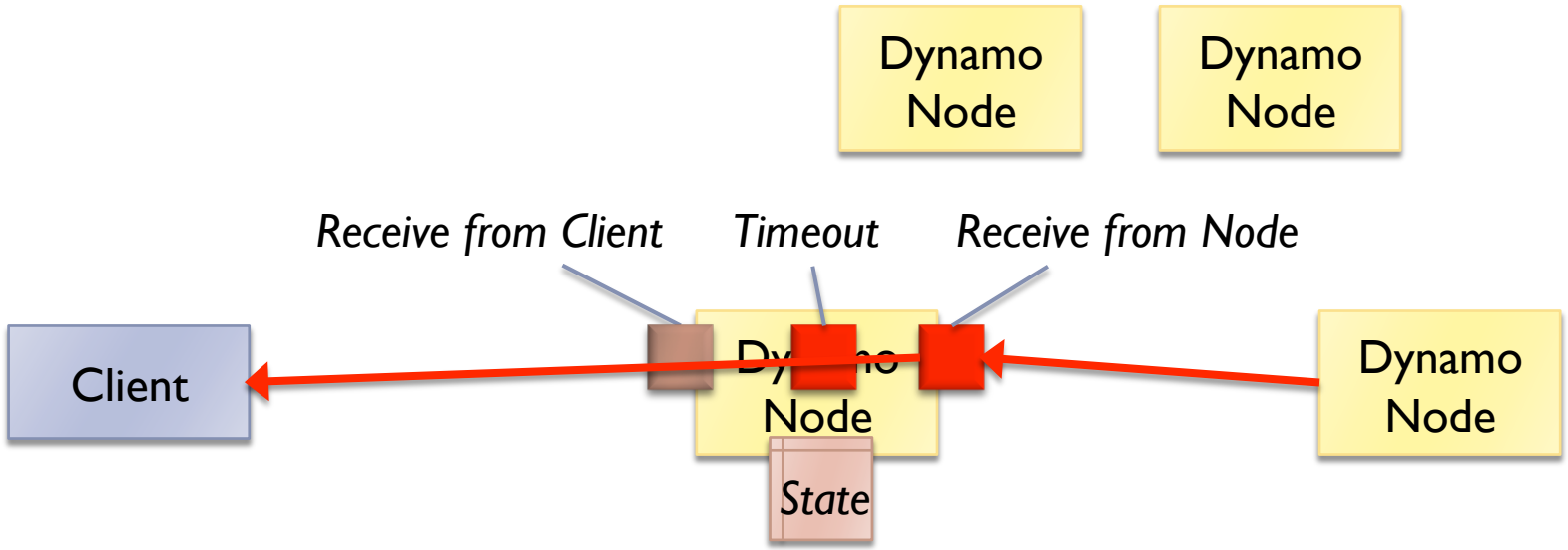
- 1つめからレスポンス




```
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) {
```

POE (Event Driven) 実装

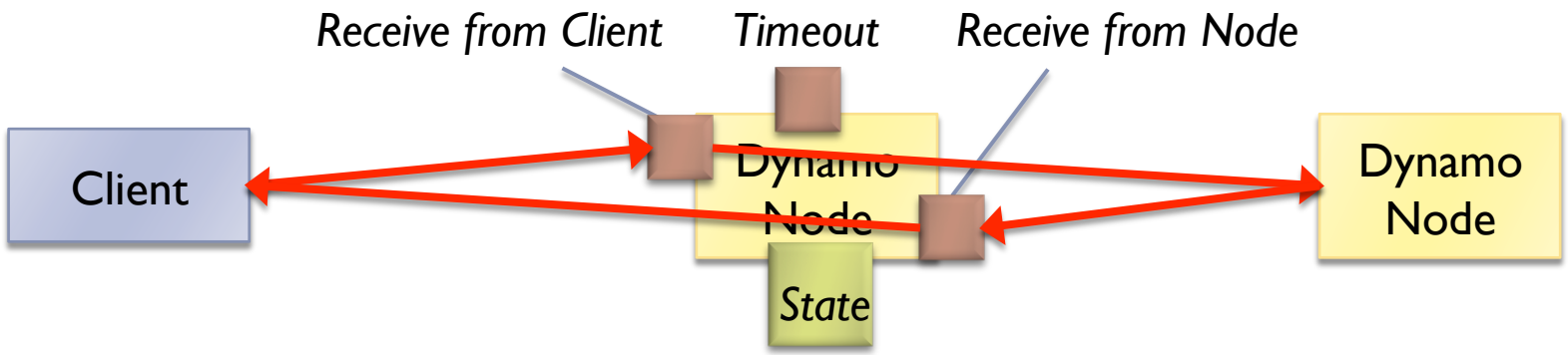
- 2つめのレスポンスでクライアントに返す
  - タイムアウトしても返す

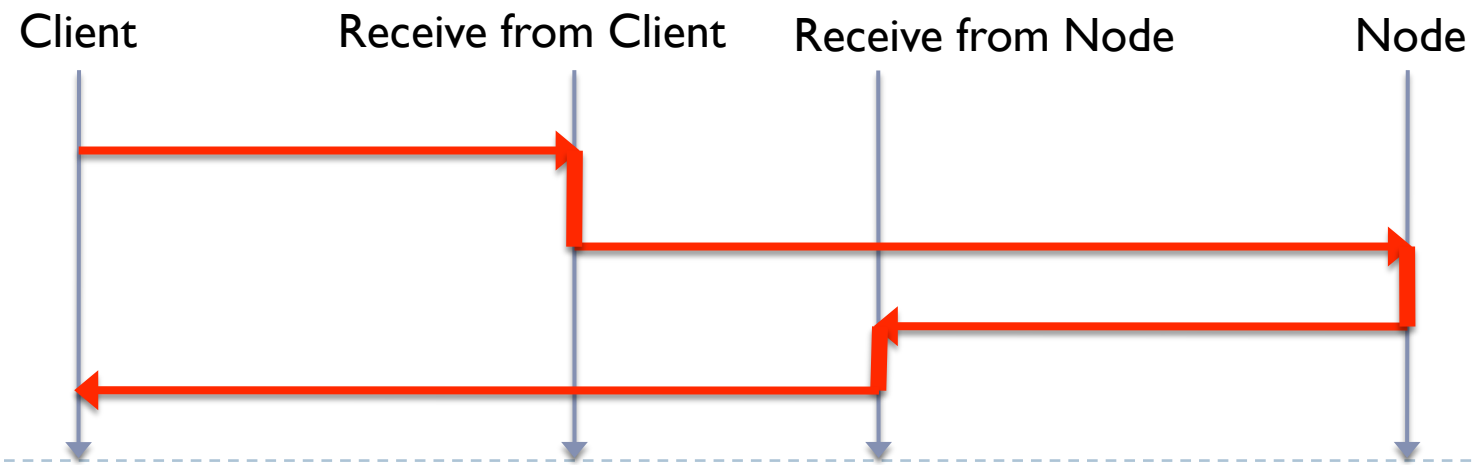



```
sub recv_from_node {
  my $cli_sock = $poe->args->[2];
  $node_sock->read(my $res, 1024);
  stat_add_res($cli_sock, $res);
  my $count = stat_count_res($cli_sock);
  if ($count == 2) {
    my $res = stat_uniq_res($cli_sock);
    $cli_sock->write($res);
  }
```

POE (Event Driven) 実装

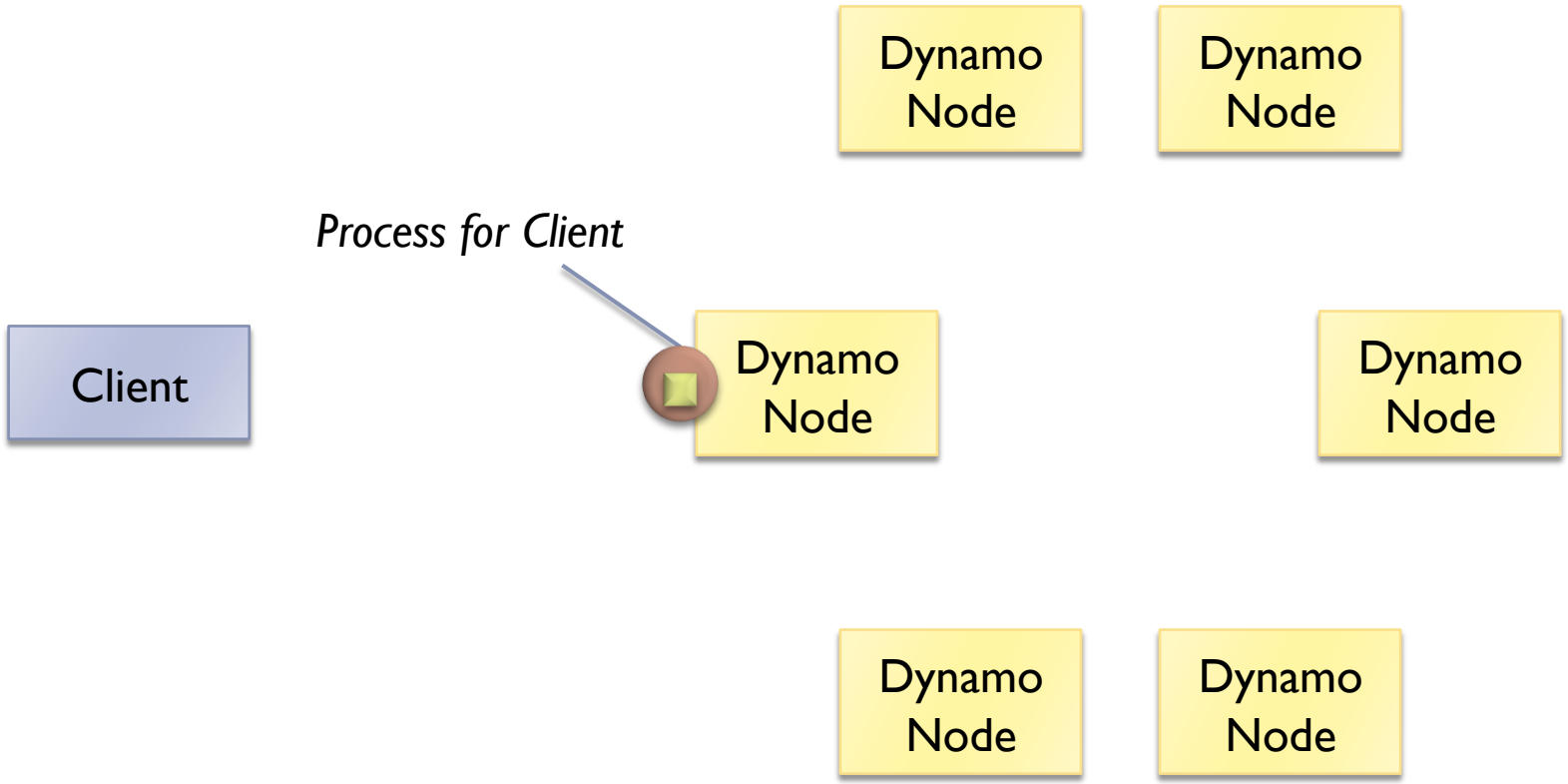
- イベント(時間の進み)単位の実装





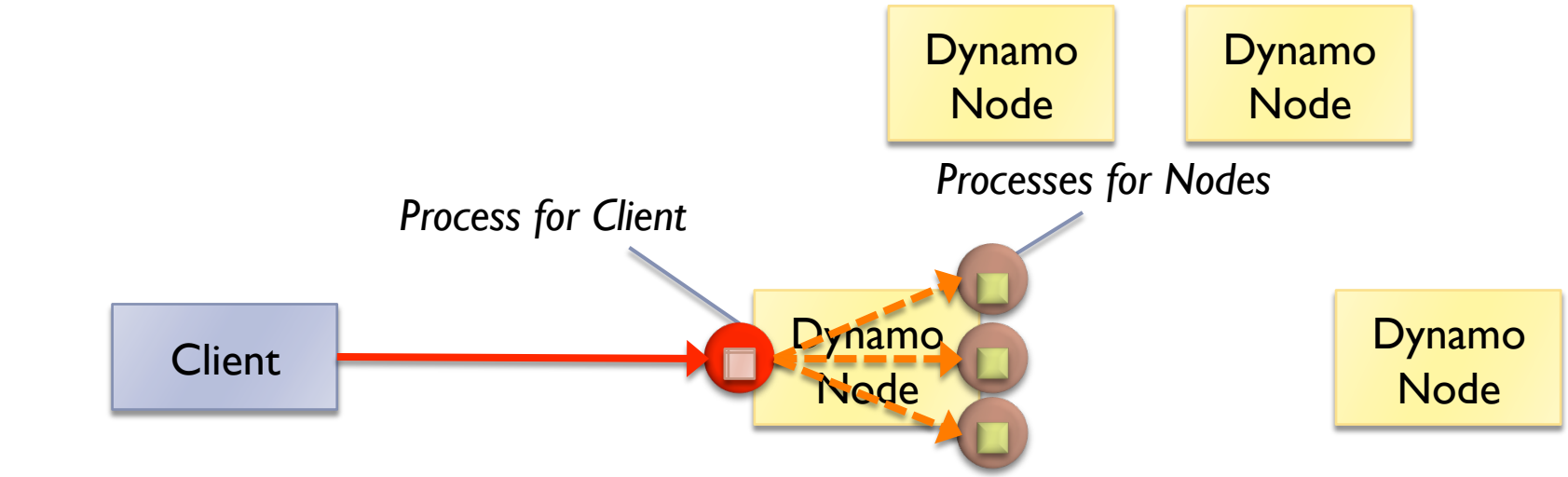
Erlang の実装

- プロセス単位の実装



Erlang の実装

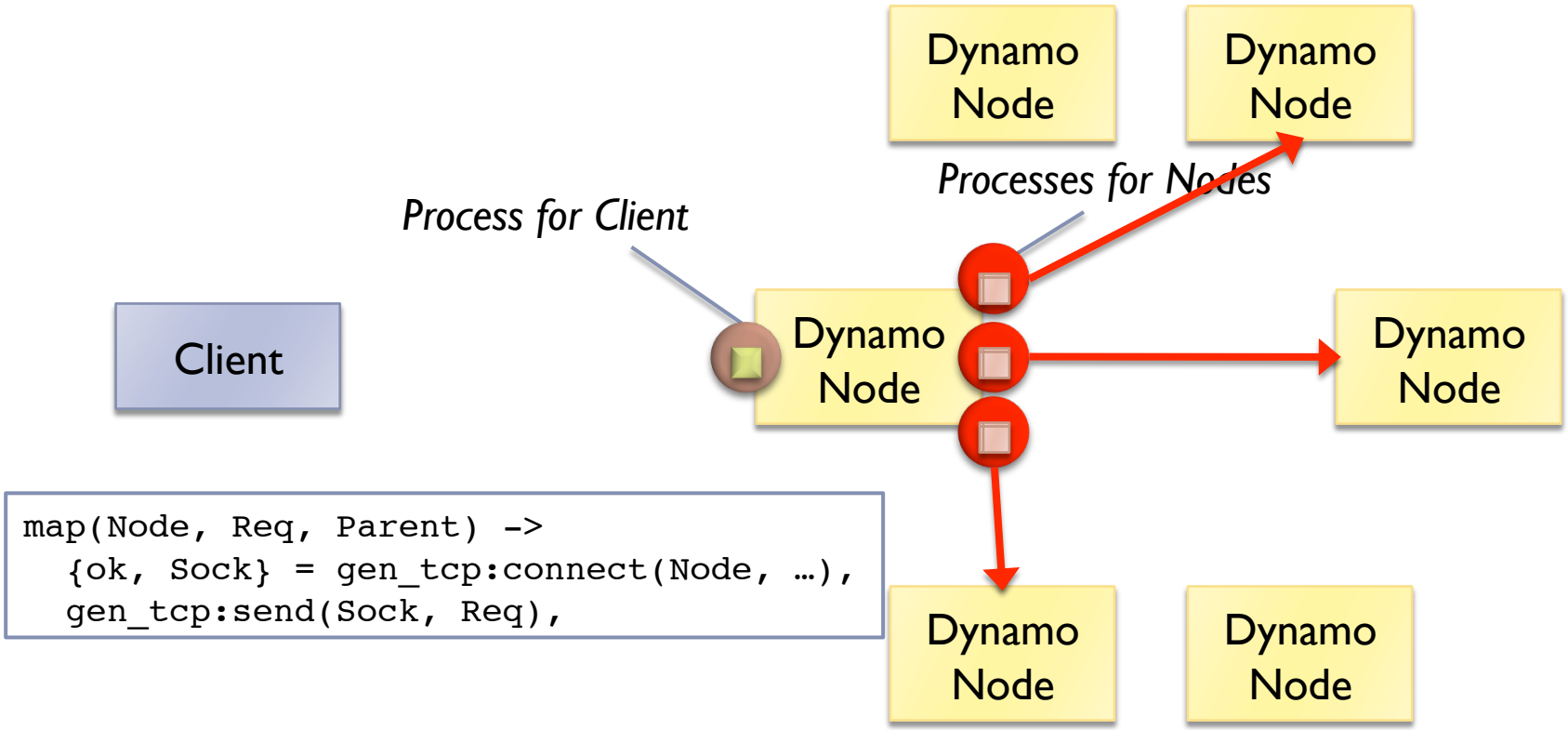
- クライアントがリクエスト




```
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, []),
```

Erlang の実装

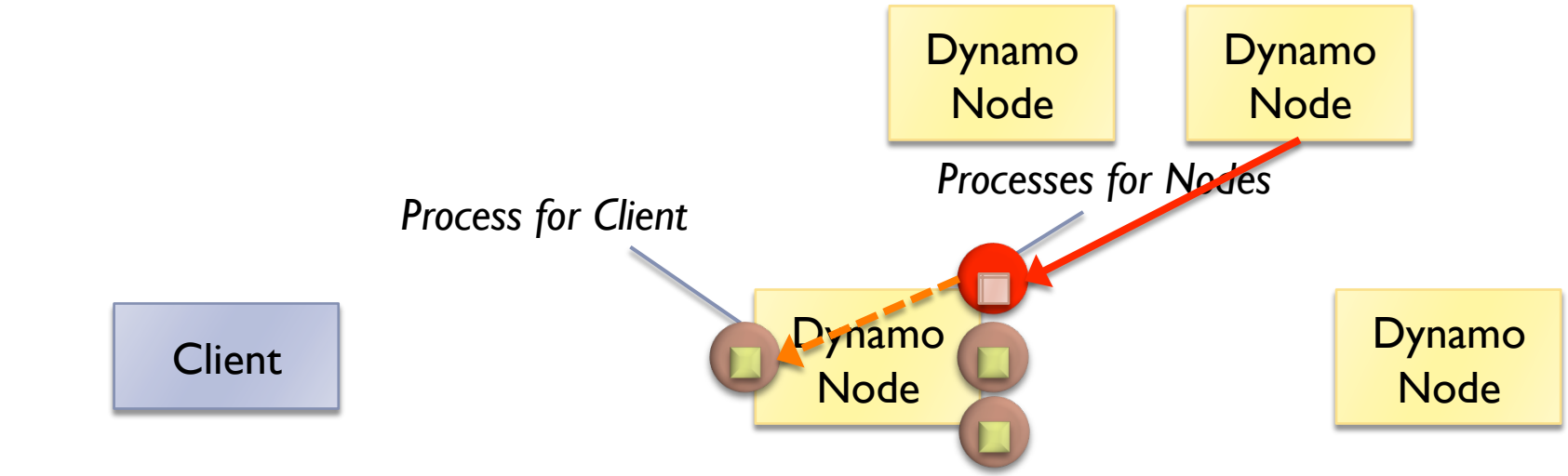
- レプリカを持っている3ノードに転送




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

Erlang の実装

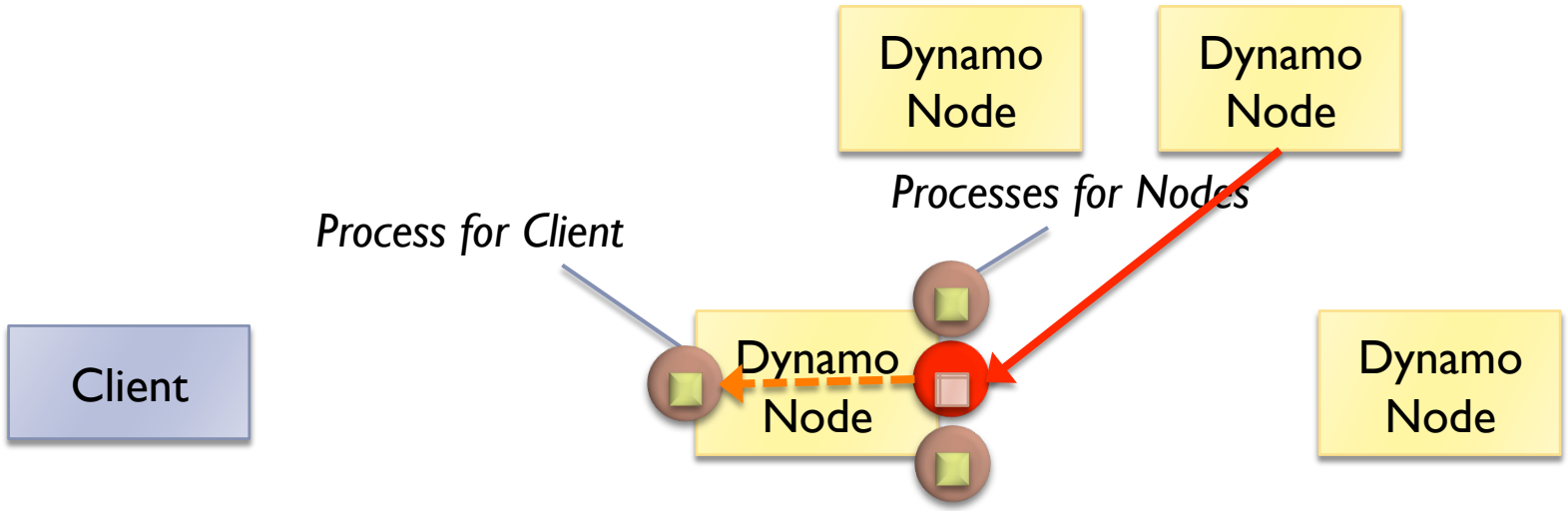
- 1つめからレスポンス




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

Erlang の実装

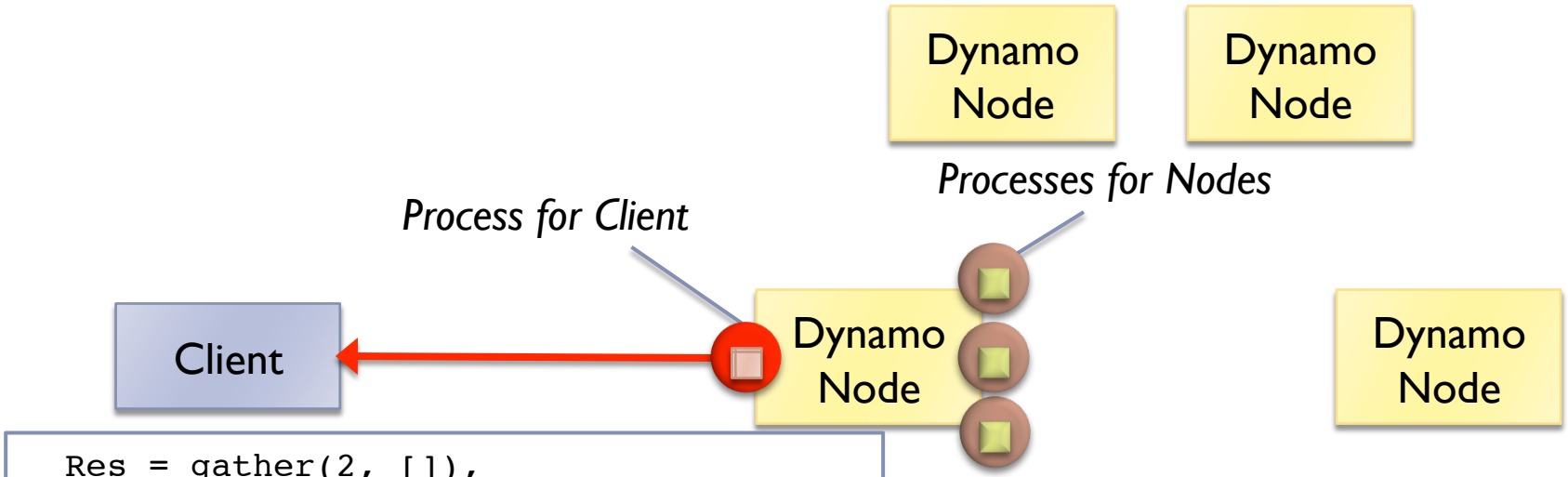
- 2つめのレスポンスでクライアントに返す
  - タイムアウトしても返す




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

Erlang の実装

- 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

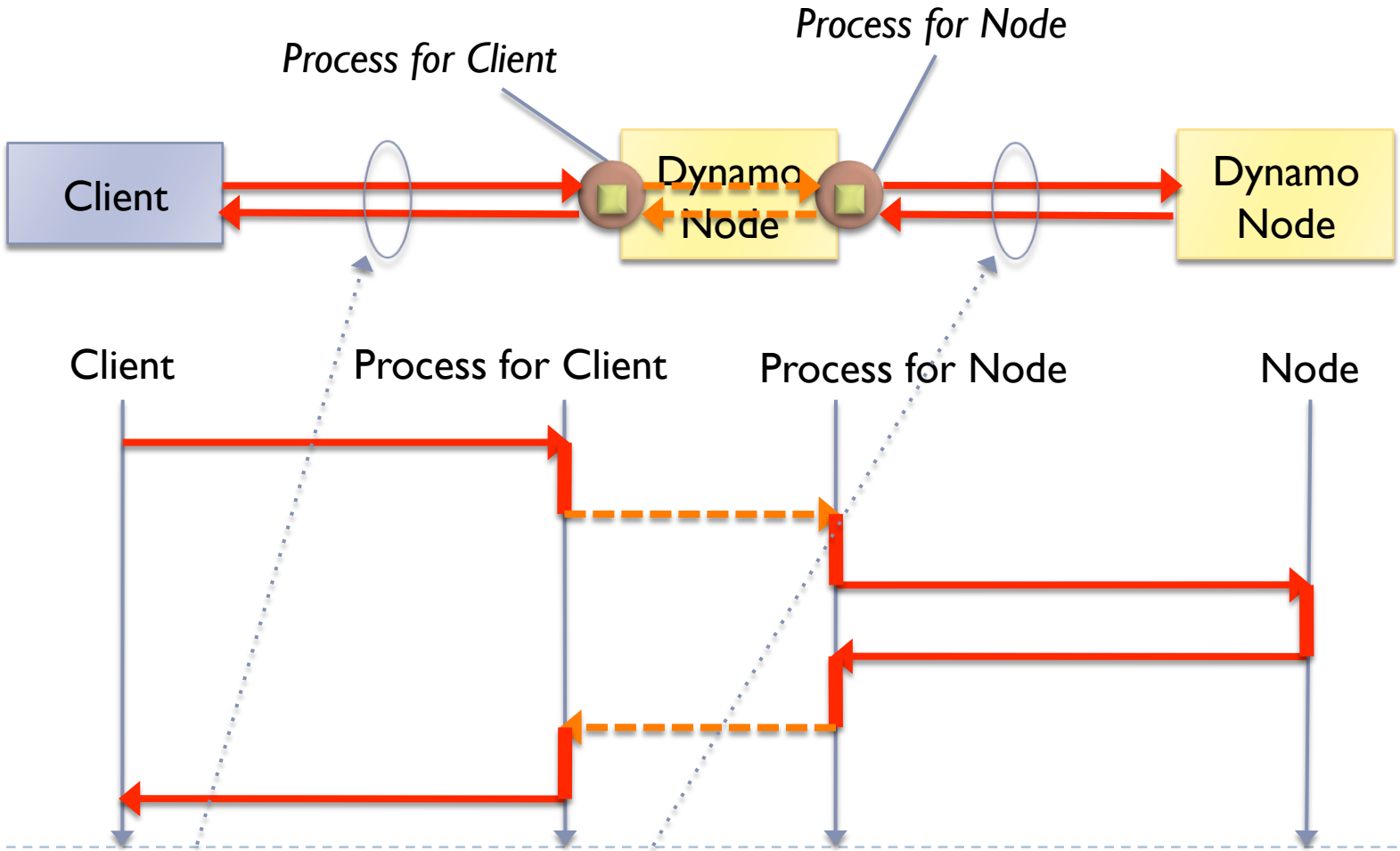



```
  Res = gather(2, []),
  gen_tcp:send(Sock, Res).

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

Erlang の実装

- プロセス単位の実装



- *クライアントとのやり取りと，ノード間のやり取りを，それぞれのプロセスにまとめられる*

まとめ

- Erlang いいよ
  - 振る舞い（動作モデル）に従った素直なコードが書ける
  - マルチスレッドのような排他制御問題がない
    - 共有メモリではなくメッセージ交換なので
  - プロセス生成やメッセージのコストが低い
    - 数千万プロセスを起動できるとか
  - といっても，分散システム以外には向かないかも
    - ファイルや文字列といった基本的な操作が弱い