Process Design and Polymorphism: Lessons Learnt from Development of Kai

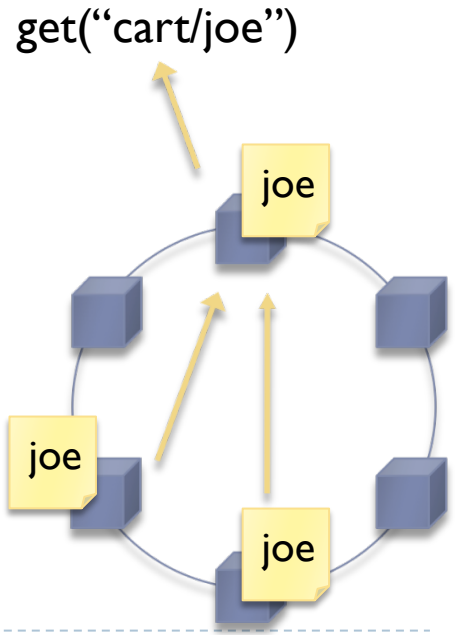
takemaru

Outline

- Reviewing Dynamo and Kai
  - Kai: an Open Source Implementation of Amazon's Dynamo
  - Features and Mechanism
- Process Design for Better Performance
  - Based on Calling Sequence and Process State
- Polymorphism in Actor Model
  - Two approaches to implement polymorphism

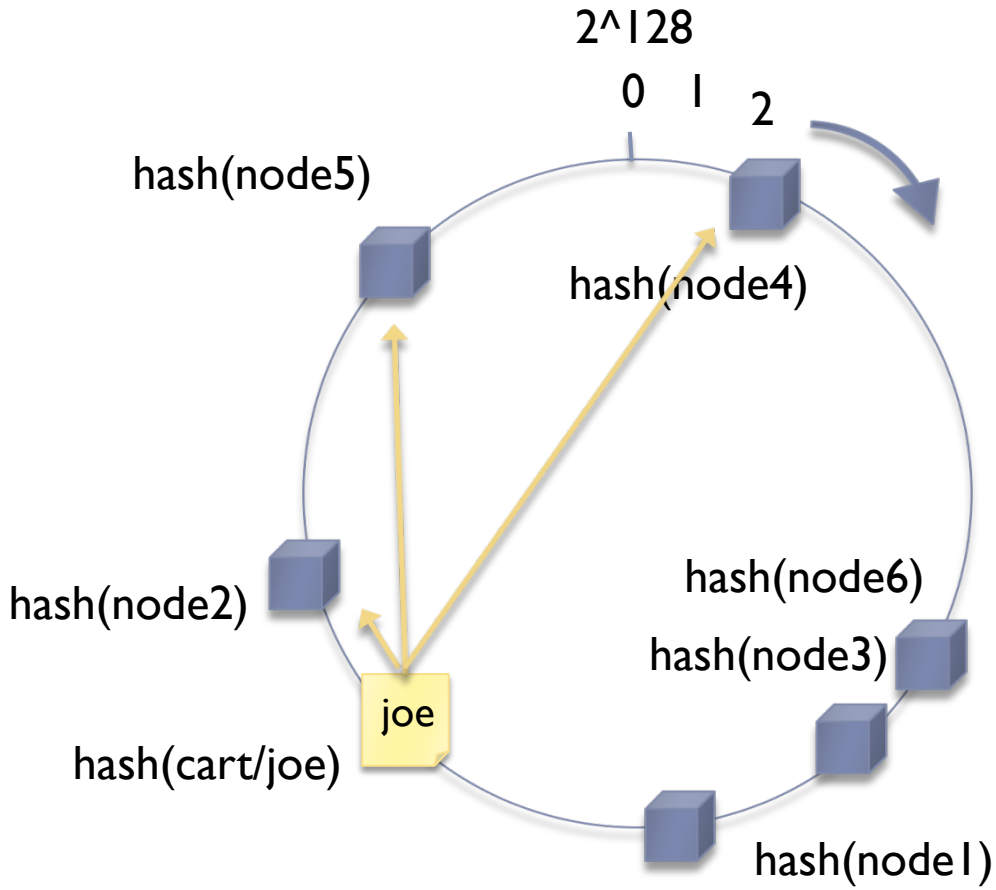
Dynamo: Features

- Key, value store
  - Distributed hash table
- High scalability
  - No master, peer-to-peer
  - Large scale cluster, maybe O(1K)
- Fault tolerant
  - Even if an entire data center fails
  - Meets latency requirements in the case



Dynamo: Partitioning

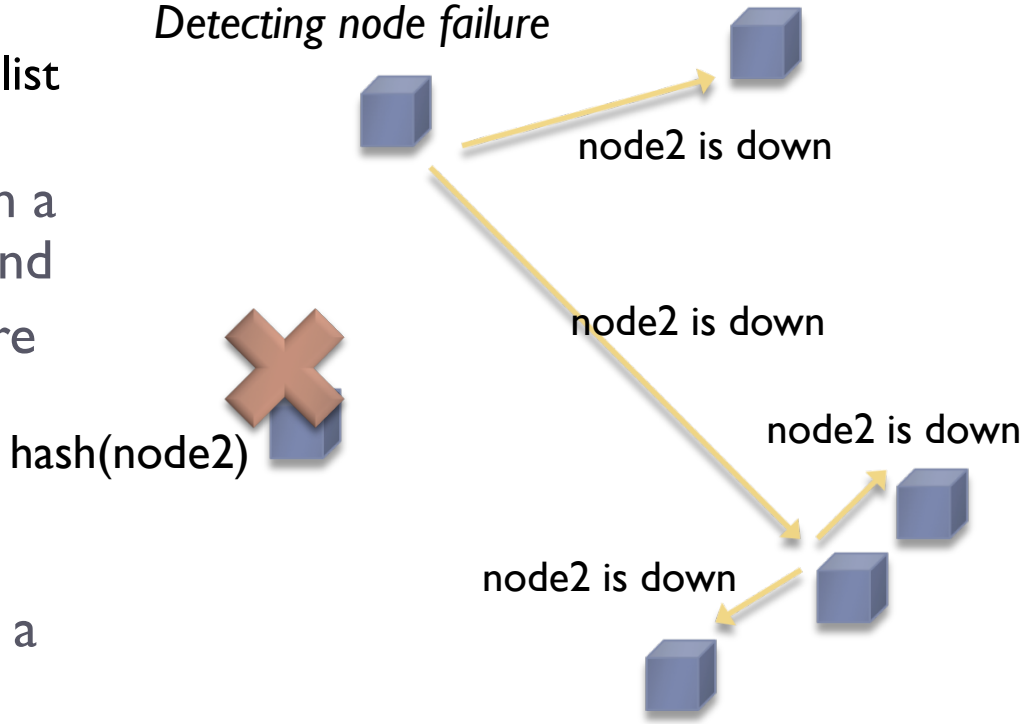
- Consistent Hashing
  - Nodes and keys are positioned at their hash values
    - MD5 (128bits)
  - Keys are stored in the following *N* nodes



Hash ring (N=3)

Dynamo: Membership

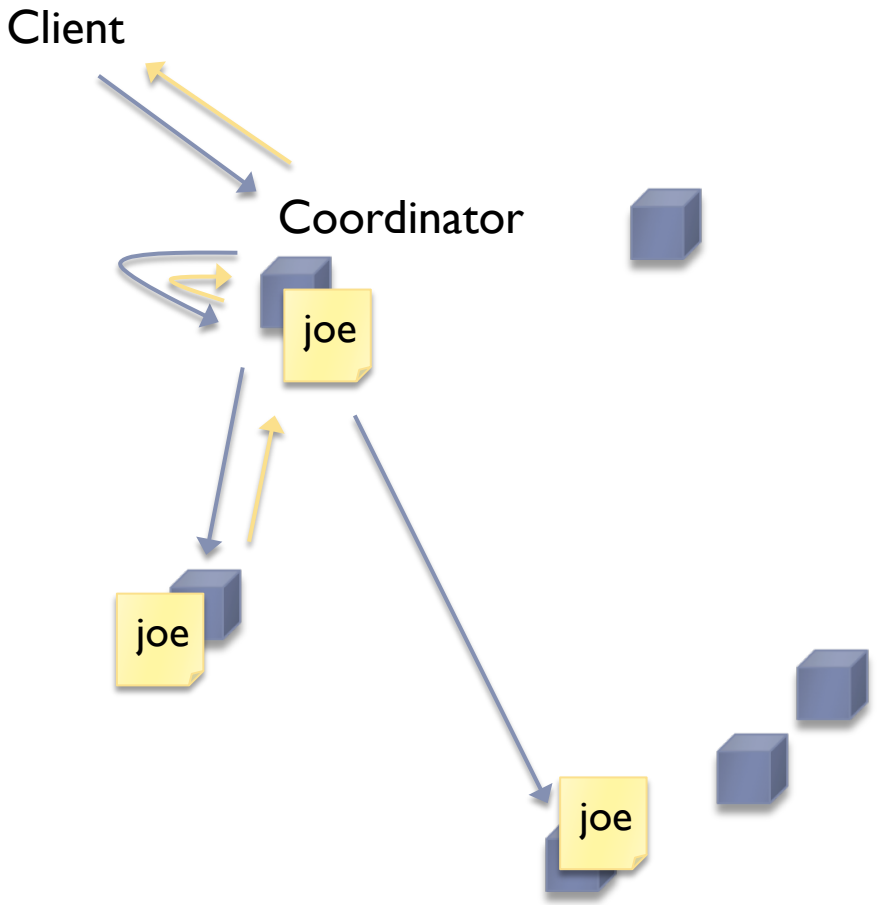
- Gossip-based protocol
  - Spreads membership like a rumor
    - Membership contains node list and change history
  - Exchanges membership with a node at random every second
  - Updates membership if more recent one received
- Advantages
  - Robust; no one can prevent a rumor from spreading
  - Exponentially rapid spread



Membership change is spread by the form of gossip

Dynamo: *get*/*put* Operations

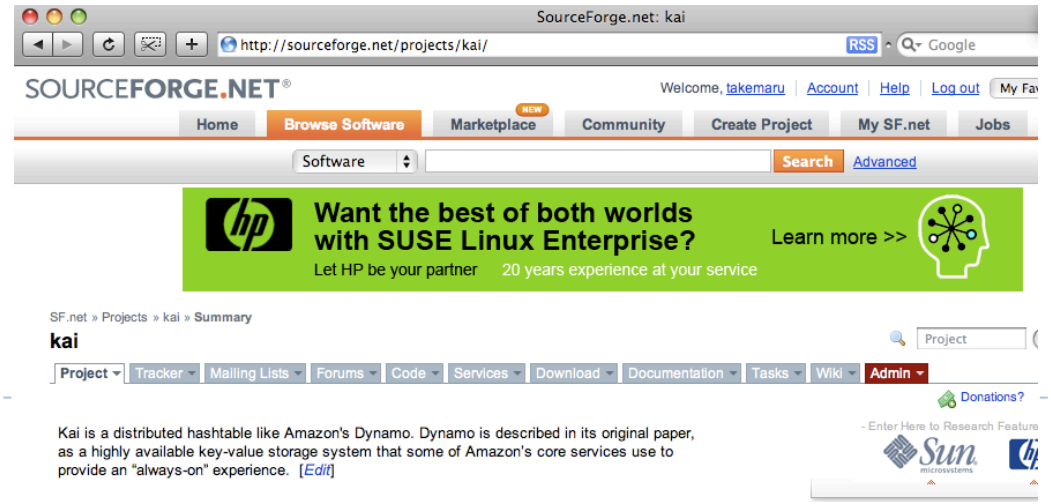
- Client
  1. Sends a request any of Dynamo node
  - The request is forwarded to coordinator
    - Coordinator: one of nodes associated with the key
- Coordinator
  1. Chooses *N* nodes by using consistent hashing
  2. Forwards a request to *N* nodes
  3. Waits responses from *R* or *W* nodes, or timeouts
  4. Checks replica versions if *get*
  5. Sends a response to client



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

Kai: Overview

- Kai
  - Open source implementation of Amazon's Dynamo
    - Named after my origin
    - *OpenDynamo* had been taken by a project not related to Amazon's Dynamo ☹
  - Written in Erlang
  - memcache API
  - Found at http://sourceforge.net/projects/kai/

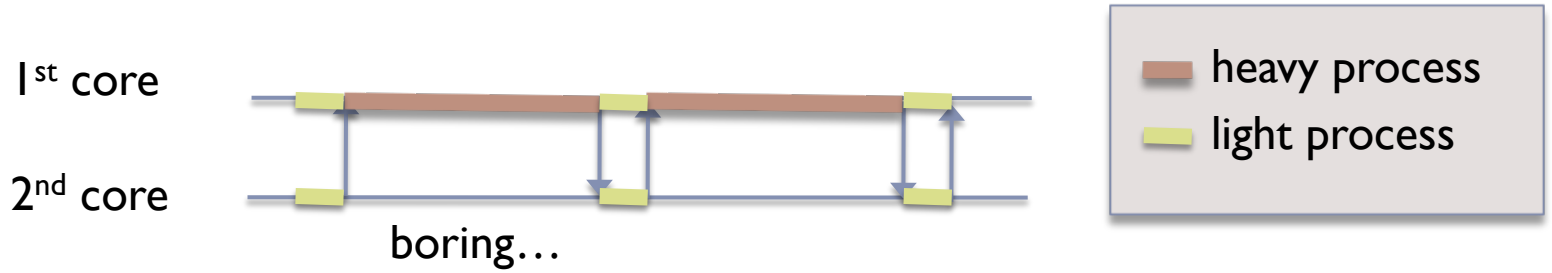


Process Design for Better Performance

Two approaches to improve software performance

- To process it with less CPU resource
  - Solved by introducing better algorithms
  - e.g. Binary search is used instead of linear search

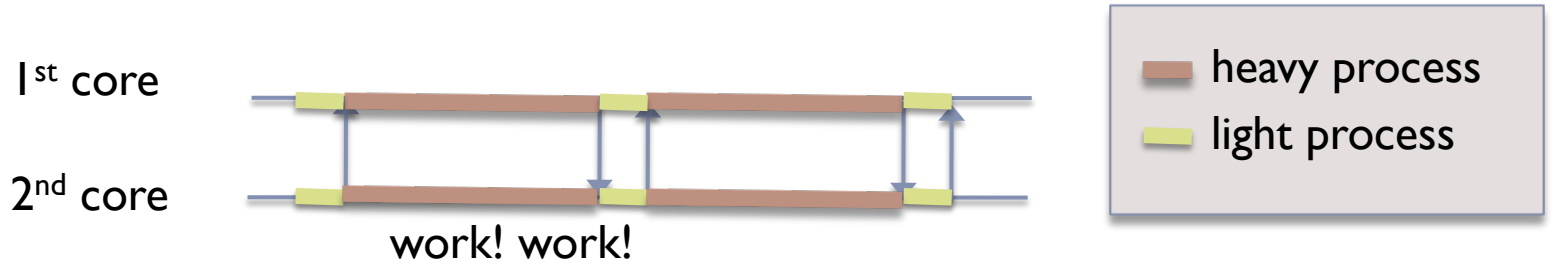
- To use all CPU resources
  - Issues identified in multi-core environment
  - Solved by rearranging process-core mapping
  - e.g. Heavy process and light process run on multi-core



Two approaches to improve software performance

- To process it with less CPU resource
  - Solved by introducing better algorithms
  - e.g. Binary search is used instead of linear search

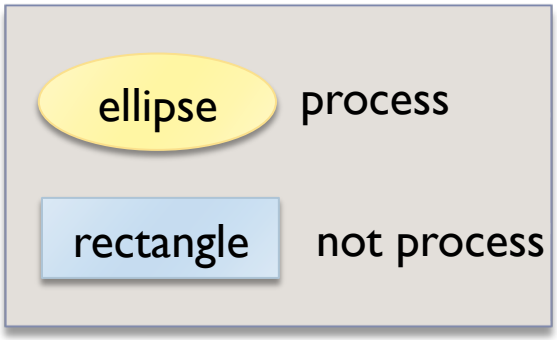
- To use all CPU resources
  - Issues identified in multi-core environment
  - Solved by rearranging process-core mapping
  - e.g. Heavy process and light process run on multi-core



Latter case will be discussed

Diagram Convention

- Procedural programming
  - Procedures are called by a process

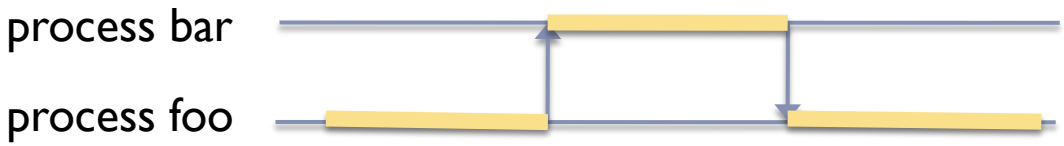


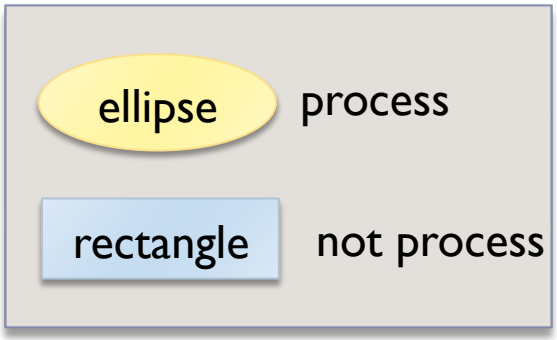

```
-module(foo).
-behaviour(gen_server).

ok(State) ->
  {reply, bar:ok(), State}.
```

```
-module(bar).

ok() ->
  ok.
```

Diagram Convention, cont'd

- Actor model
  - 2 processes interact with each other





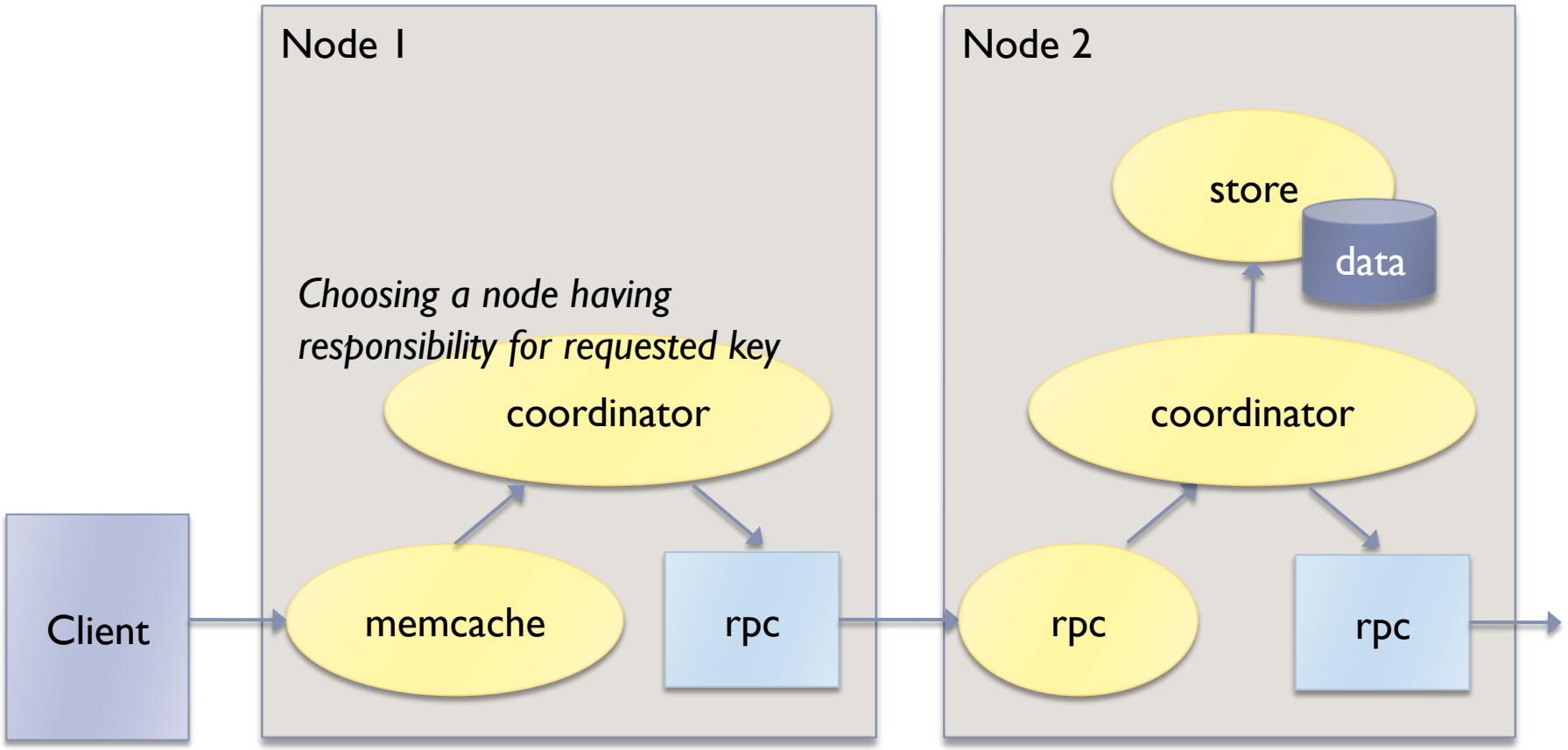

```
-module(foo).
-behaviour(gen_server).

ok(State) ->
  {reply, bar:ok(), State}.
```

```
-module(bar).
-behaviour(gen_server).

ok(State) ->
  {reply, ok, State}.
ok() ->
  gen_server:call(?MODULE, ok).
```

Design Rules in Erlang

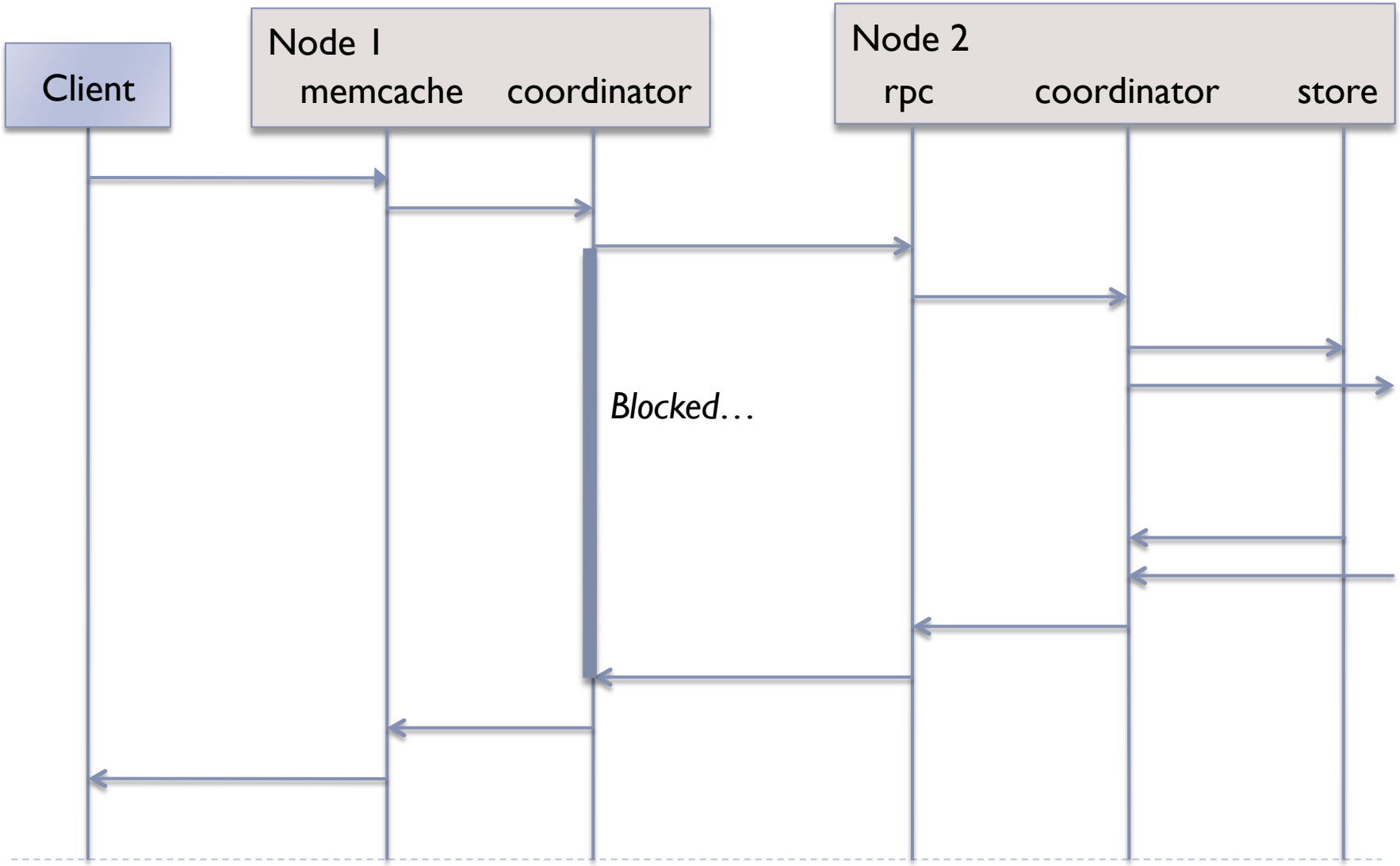
- Some of rules on process design:
  - *"Assign exactly one parallel process to each true concurrent activity in the system"*
  - *"Each process should only have one role"*
    - from Program Development Using Erlang - Programming Rules and Conventions

Processes in getting/putting data

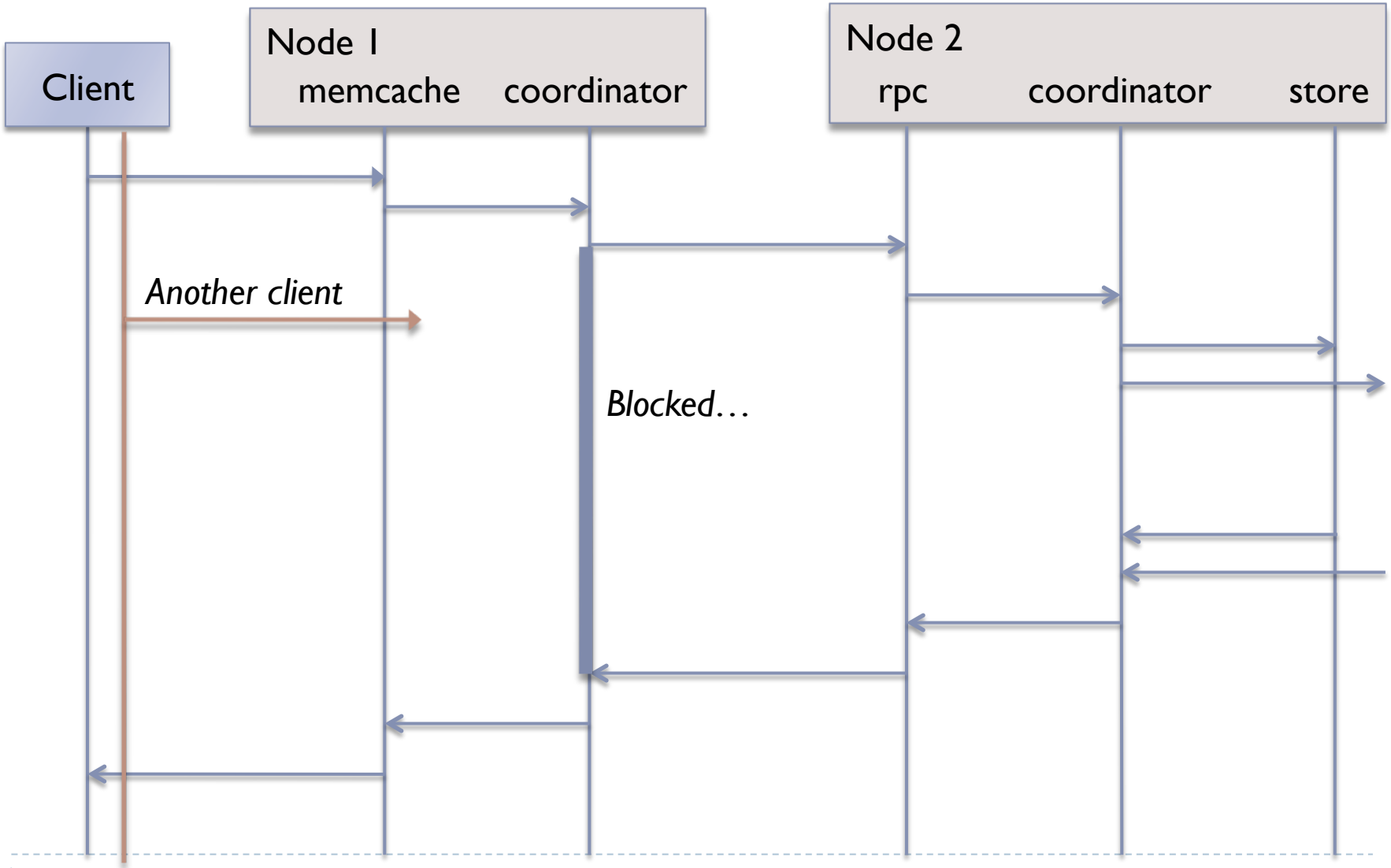


For clarity, details of architecture are omitted.

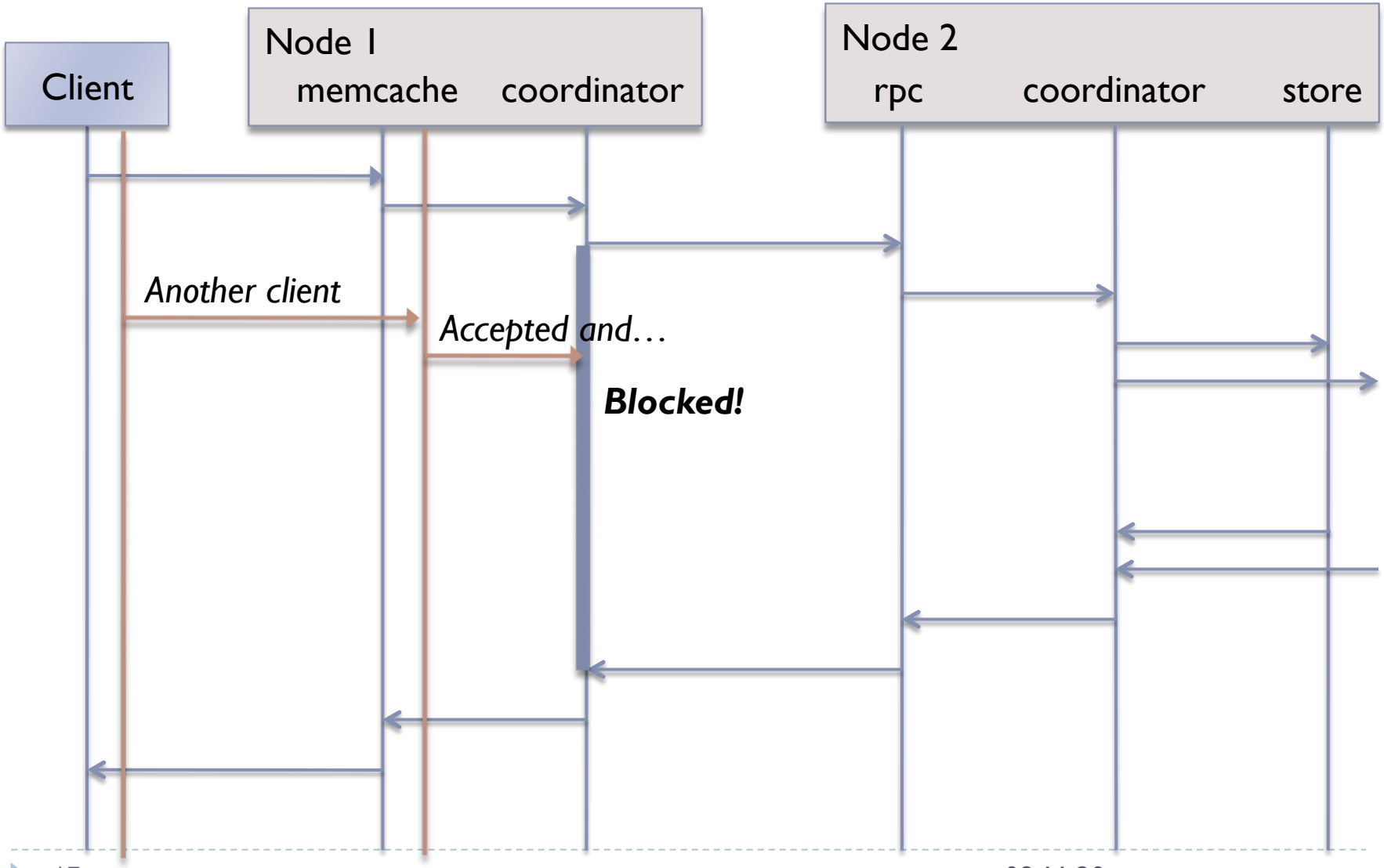
Sequence in getting/putting data



Sequence in getting/putting data, cont'd



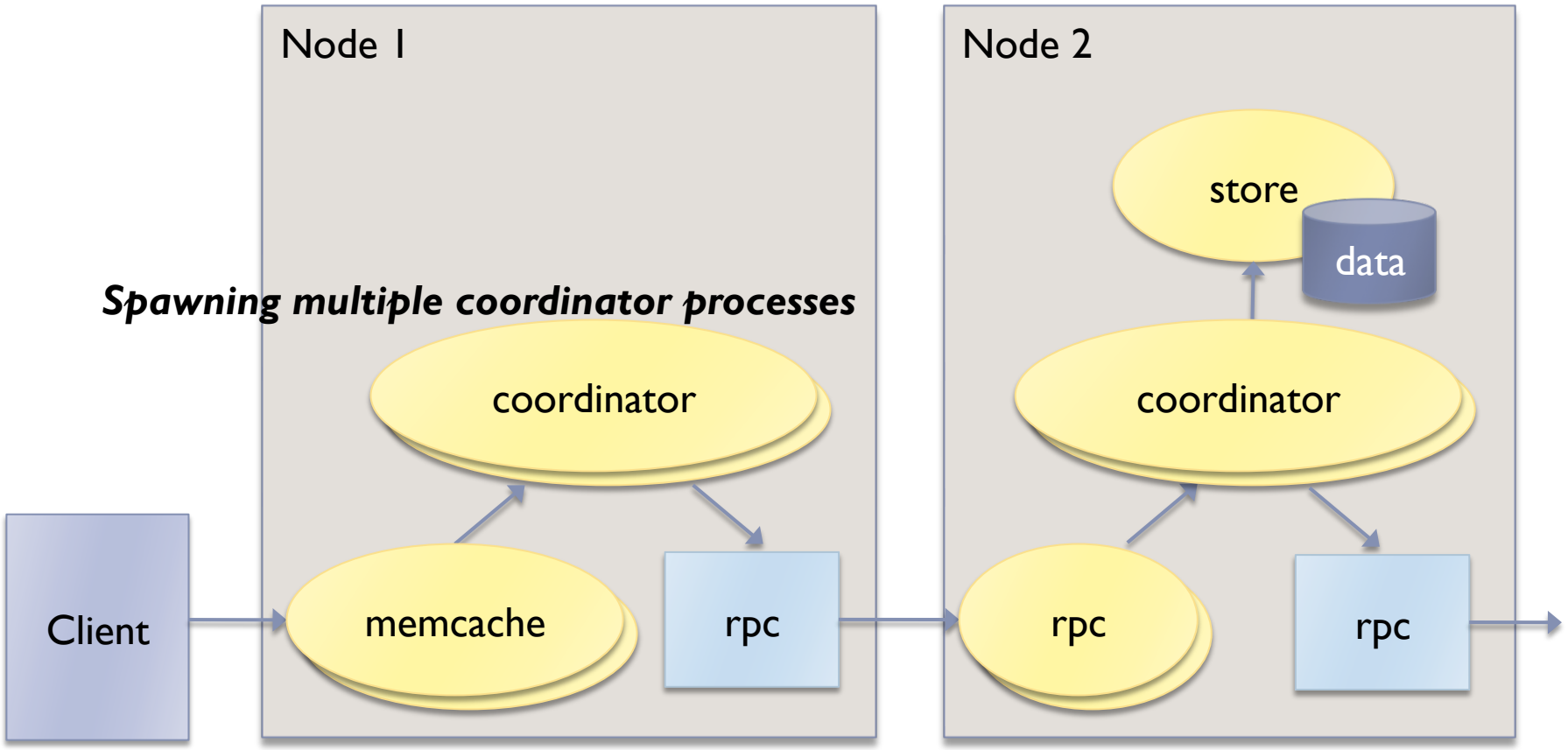
Sequence in getting/putting data, cont'd



Design Rules in Erlang, again

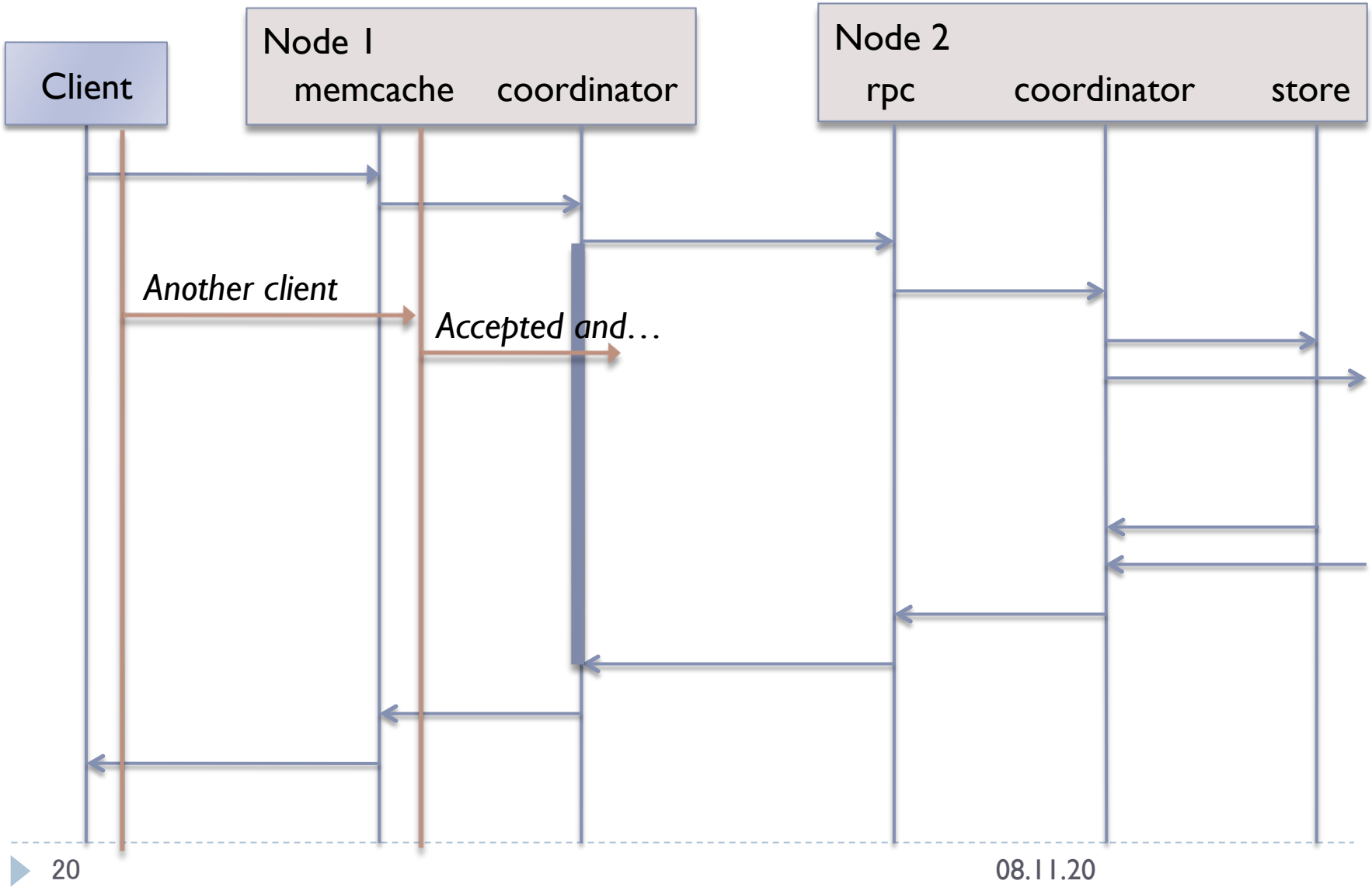
- Another rule on process design:
  - *"Use many processes"*
    - Many processes are almost uniformly assigned to each processor by statistical effect
    - from Chap.20 Programming Erlang

Processes in getting/putting data, again

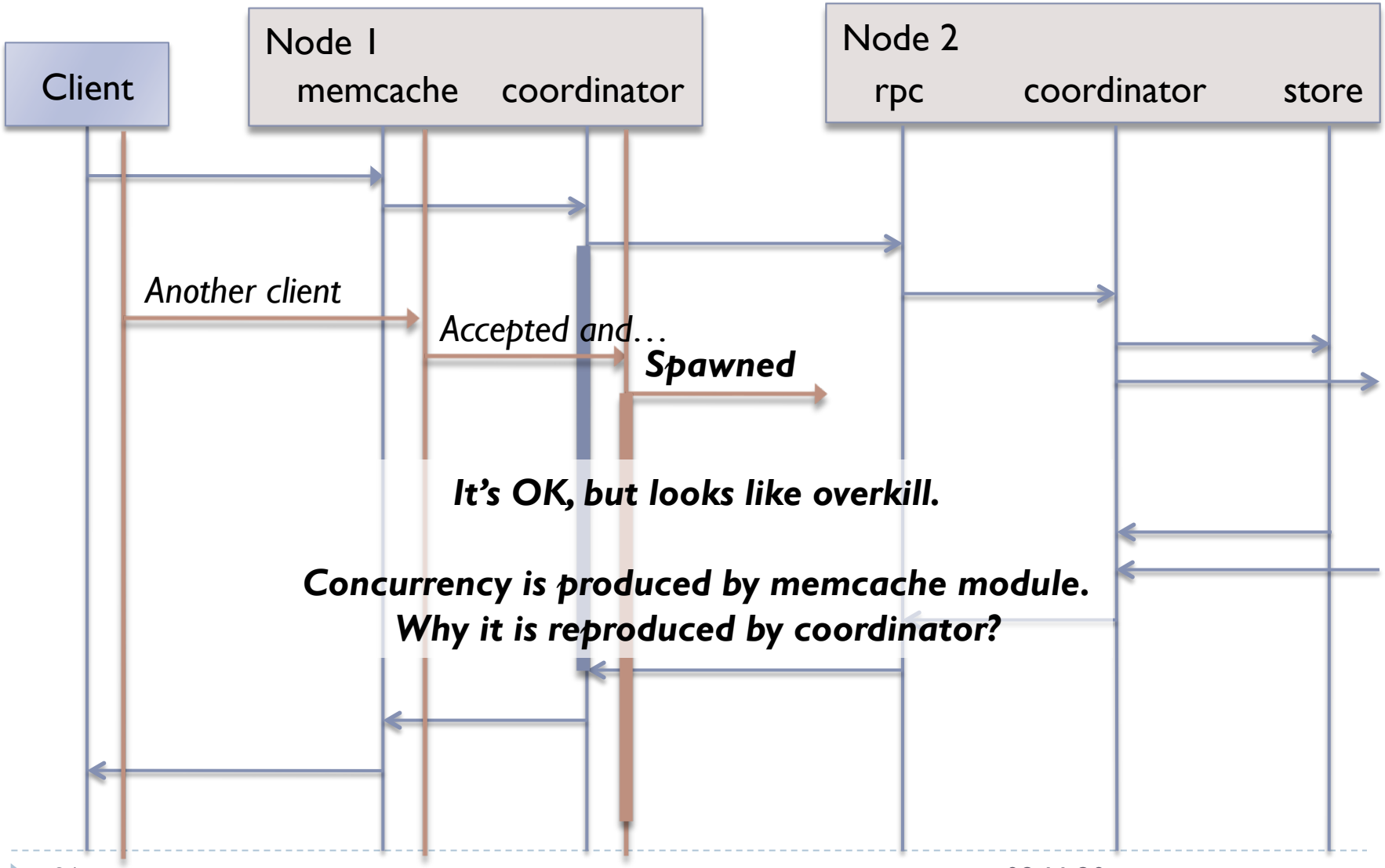


For clarity, details of architecture are omitted.

Sequence in getting/putting data, again



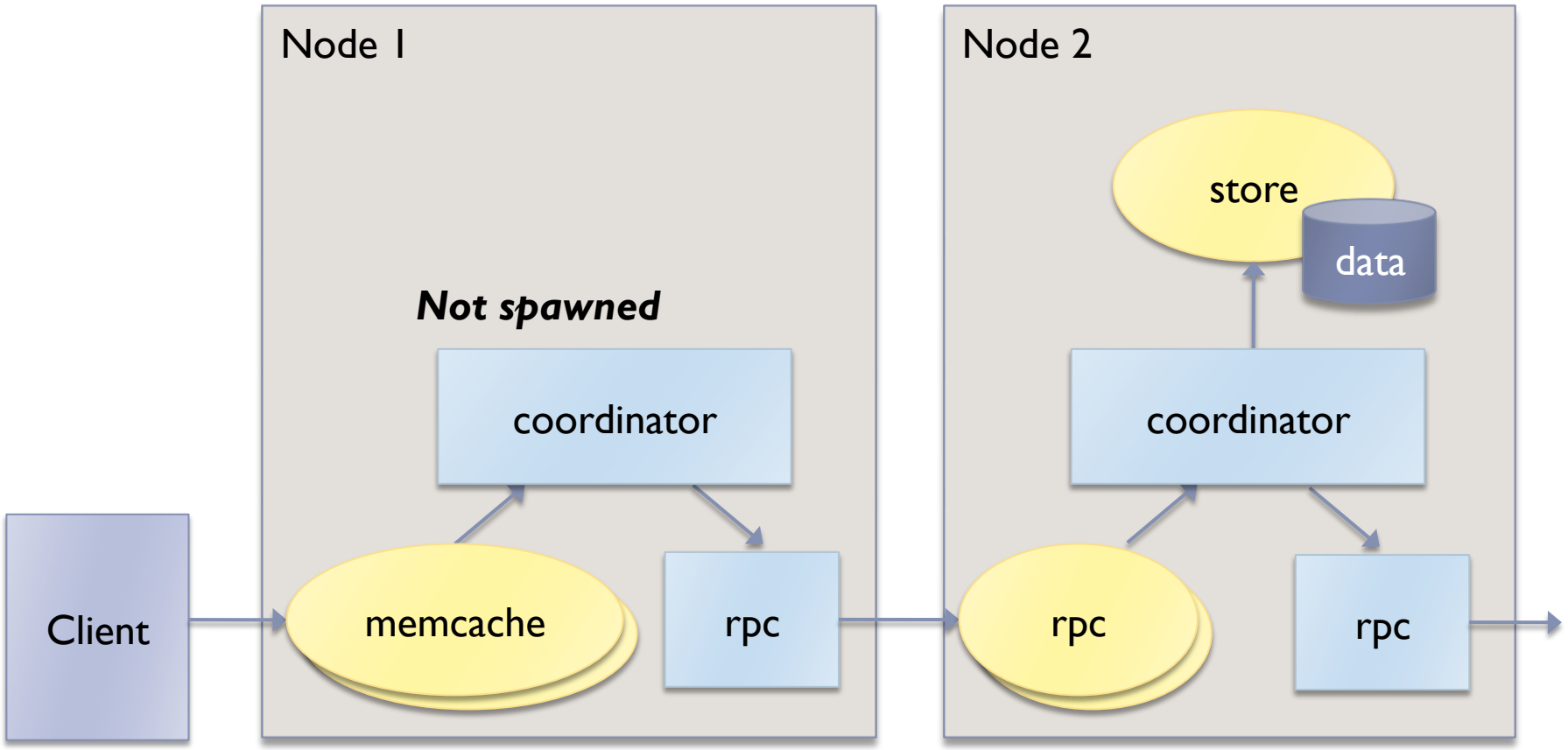
Sequence in getting/putting data, again



Design Rules in Erlang, in final

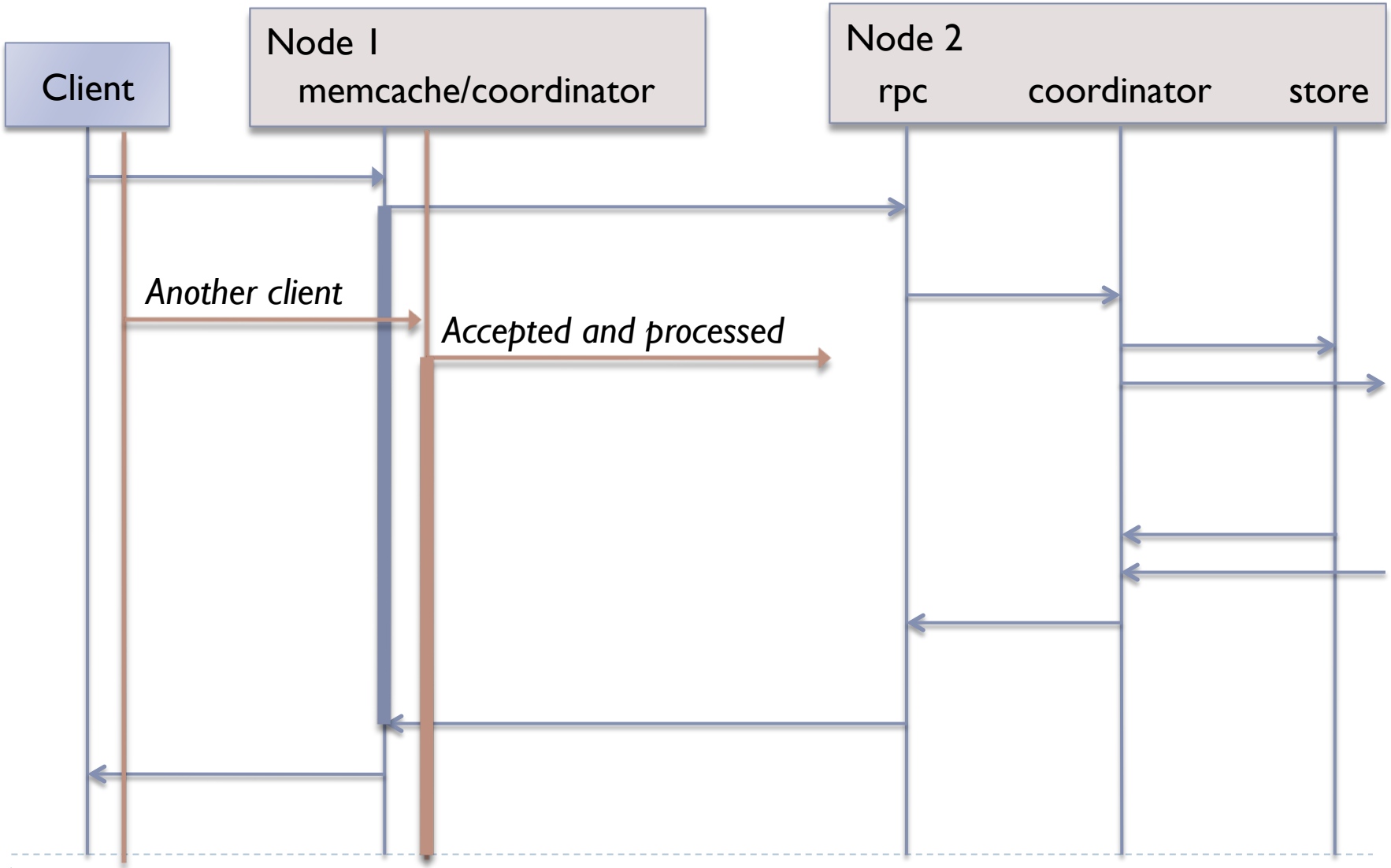
- Another rule on process design:
  - *"Don't spawn stateless processes"*
    - Called as procedures from concurrent processes
    - Introduced by me ☺

Processes in getting/putting data, in final



For clarity, details of architecture are omitted.

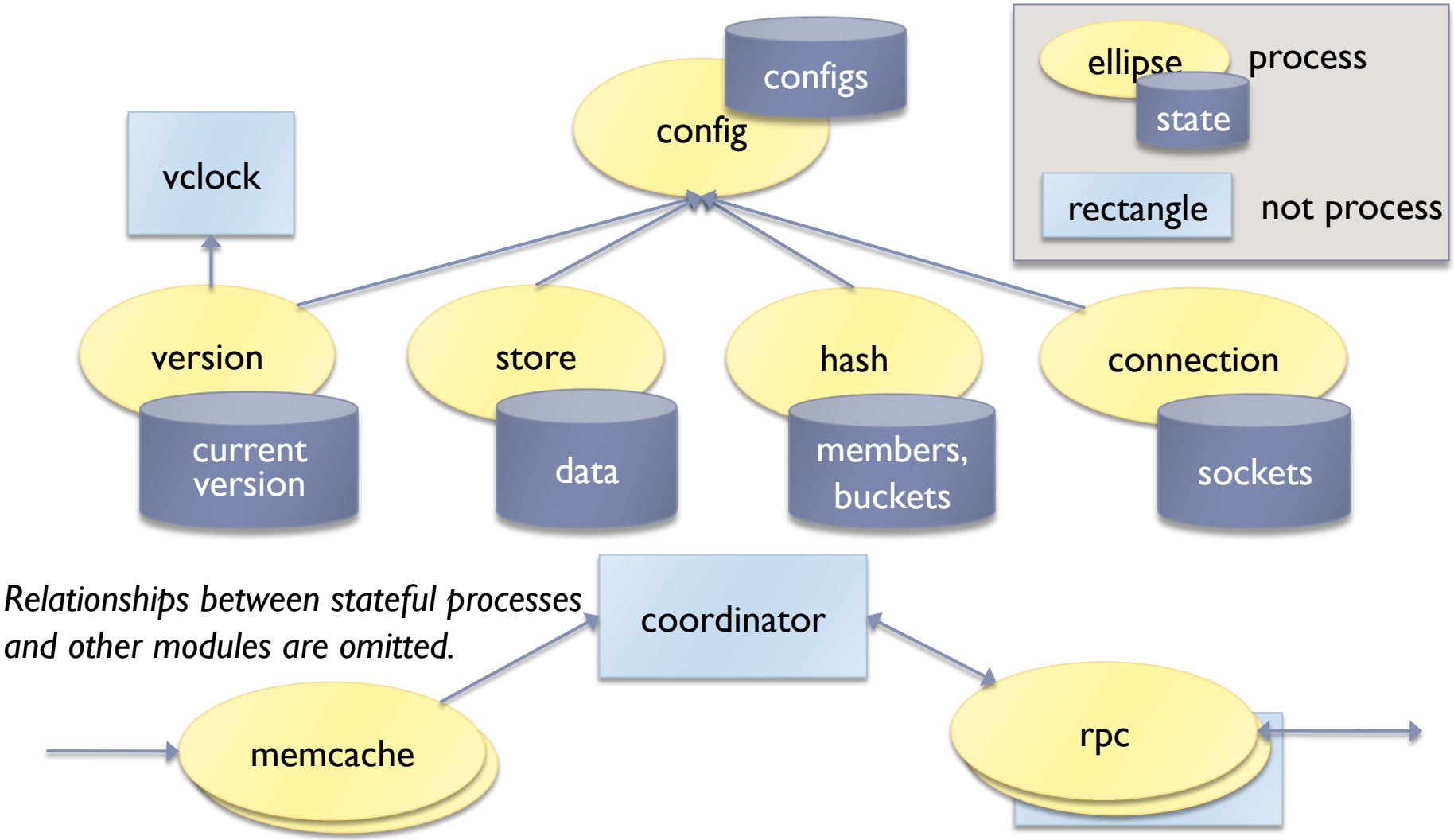
Sequence in getting/putting data, in final



Lessons Learnt

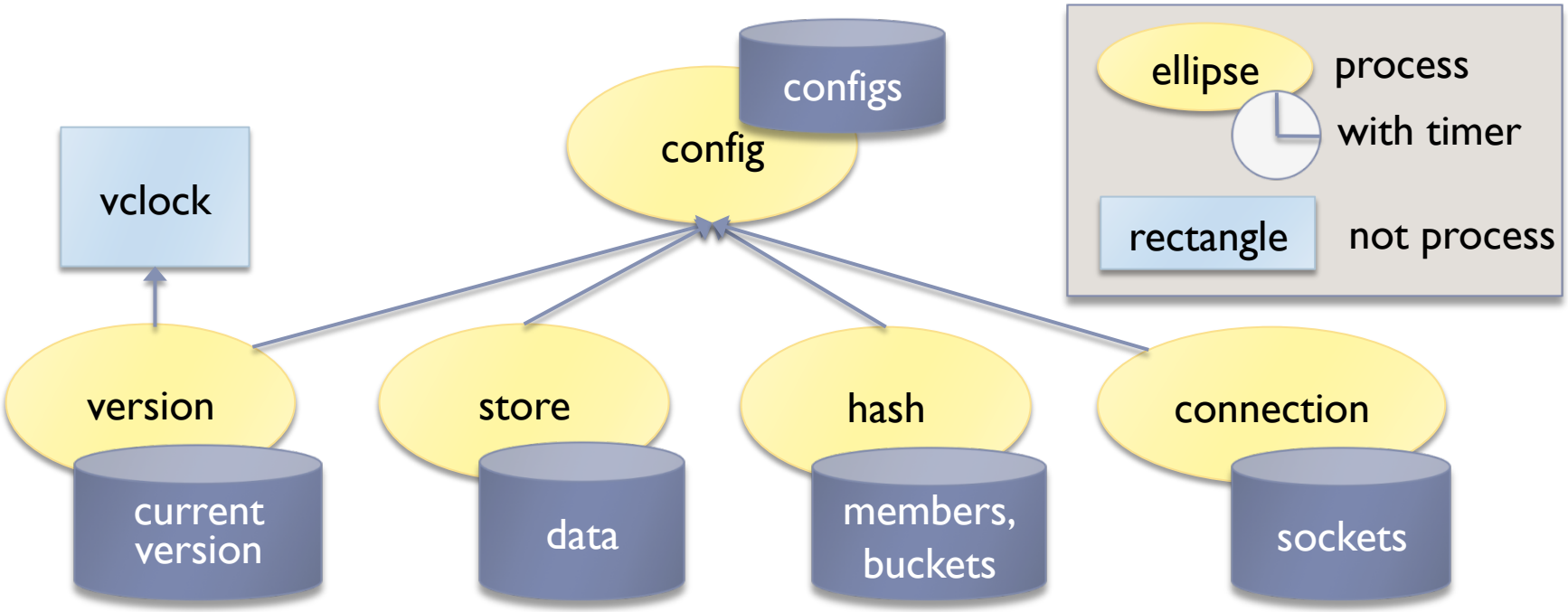
- Process design based on calling sequence and process state
  - Externally called module
    - Must be spawned to produce concurrency
    - Runs as multiple processes if needed
    - e.g. TCP listening process, timer process
  - Stateless module
    - No need to be spawned
    - e.g. coordinator of Kai
  - Stateful module
    - Should be spawned for state consistency
    - Runs as multiple processes if possible
      - If a single process, it must be a terminal one (never call other processes synchrnously) to avoid blocking
    - e.g. database process, socket pool

Process Relationship in Kai

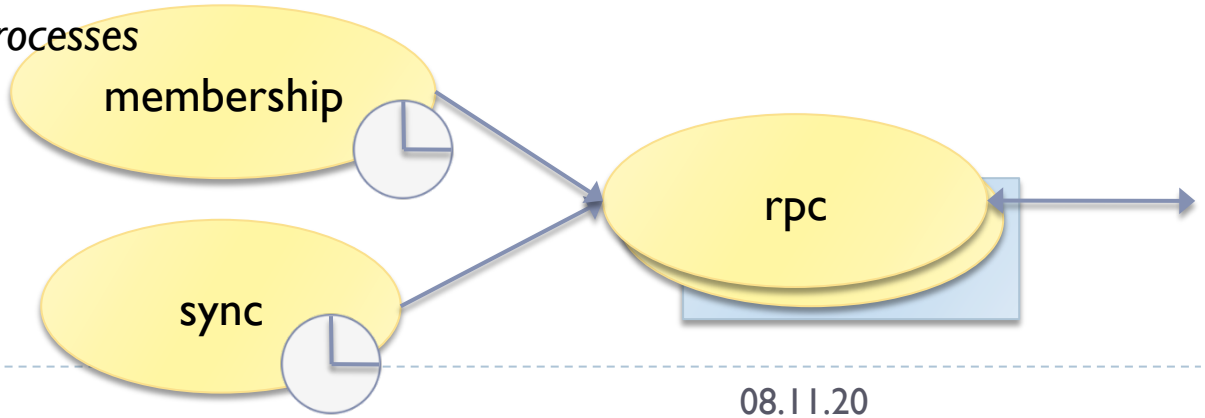


Relationships between stateful processes and other modules are omitted.

Process Relationship in Kai, cont'd



Relationships between stateful processes and other modules are omitted.



Process Relationship in Kai, cont'd

- "Lessons learnt" are almost satisfied
  - Externally called modules are spawned
    - As multiple processes if needed
    - e.g. kai_rpc, kai_memcache, kai_sync, kai_membership
  - Stateless modules are not spawned
    - e.g. kai_coordinator
  - Stateful modules are spawned
    - e.g. kai_config, kai_version, kai_store, kai_hash, kai_connection
    - However, some of them are NOT terminal ones
      - e.g. ***connection*** module calling config process, is potential bottle neck

- "Lessons learnt" can point out *potential bottle necks*
  - Yes, ***connection*** module is just a thing!

Advanced Issues

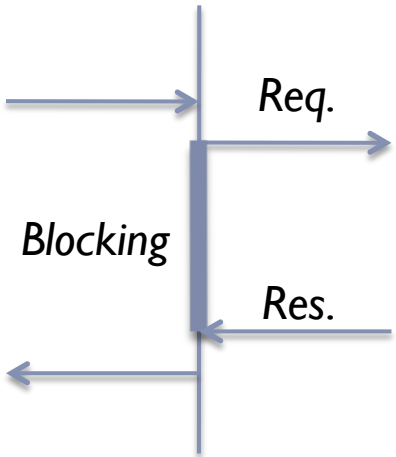
- More concurrency may be introduced if needed
  - Design rules from "Lessons Learnt" can be applied locally
  - e.g. coordinator produces *N* concurrency for asynchronous calls

- Referred to Web application servers in MVC model

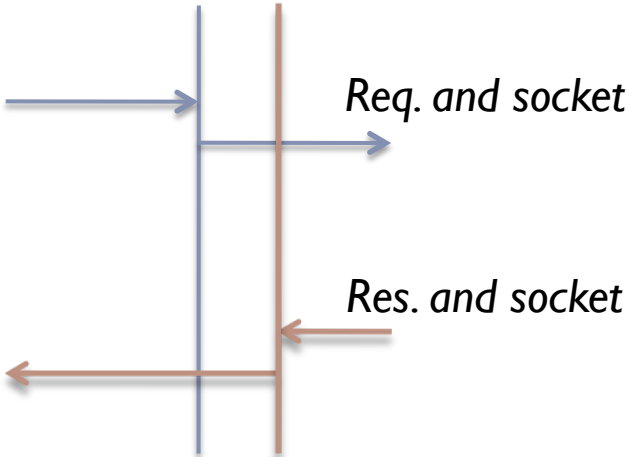
	Web application servers	**Process Design from "Lessons Learnt"**
Concurrency is produced by	Web servers, e.g. Apache	Externally called modules
Application is controlled by	Controller of MVC	Stateless modules
State is managed by	Model of MVC	Stateful modules

Advanced Issues, cont'd

- Is *blocking* never occurred in pure functional programming?
  - In Kai, a process receiving requests waits for data to be replied
  - In pure functional programming, another process handles data to be replied?
    - Not straightforward for me…



Kai



Pure functional programming model?

Polymorphism in Actor Model

What's Polymorphism?

- "a programming language feature that allows values of different data types to be handled using a uniform interface"
  - from Wikipedia

Polymorphism in Java

- Interface

```
interface Animal {
  void bark();
}
```

Including no implementation

- Implementation class

```
class Dog implements Animal {
  public void bark() {
    System.out.println("Bow wow");
  }
}
```

Polymorphism in Java, cont'd

- Abstract class

```
abstract class Animal {
  public void bark() {
    System.out.println(this.yap());
  }
  abstract String yap();
}
```

Including some implementation

- Concrete class

```
class Dog extends Animal {
  String yap() {
    return "Bow wow";
  }
}
```

Polymorphism Inspired by Interface

- Interface module
  - Initializes implementation module with a name
  - Calls the process with the name

- Implementation module
  - Spawns a process and registers it as the given name
  - Implements actual logics, which are called from interface

Polymorphism Inspired by Interface, cont'd

```
-module(animal).

start_link(Mod) ->
  Mod:start_link(?MODULE).

bark() ->
  gen_server:call(?MODULE, bark).
```

```
-module(dog).
-behaviour(gen_server).

start_link(ServerName) ->
  gen_server:start_link({local, ServerName}, ?MODULE, [], []).

handle_call(bark, _From, State) ->
  bark(State).

bark(State) ->
  io:format("Bow wow~n"),
  {reply, ok, State}.
```

Some required callbacks are omitted.

Polymorphism Inspired by Interface, cont'd

- How to use

```
animal:start_link(dog),
animal:bark(). % Bow wow
```

Polymorphism Inspired by Interface, cont'd

- Example in Kai
  - Provides two types of local storage with a single interface
  - Interface module
    - kai_store
  - Implementation modules
    - kai_store_ets, uses ets, memory storage
    - kai_store_dets, uses dets, disk storage
  - See actual codes in detail

Polymorphism Inspired by Abstract Class

- Abstract module
  - Defines abstract functions by using behavior mechanism
  - Spawns a process and stores a name of concrete module
  - Implements base logics
  - Calls the process

- Concrete module
  - Implements callbacks, which are called from the abstract module

Polymorphism Inspired by Abstract Class, cont'd

```
-module(animal).
-behaviour(gen_sever).

behaviour_info(callbacks) -> [{yap, 0}]; % abstract void yap();
behaviour_info(_Other)    -> undefined.

start_link(Mod) ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [Mod], []).

init(_Args = [Mod]) ->
    {ok, _State = {Mod}}.

bark(_State = {Mod}) ->
    io:format("~s~n", [Mod:yap()]),
    {reply, ok, Mod}.

handle_call(bark, _From, State) ->
    bark(State).

bark() ->
    gen_server:call(?MODULE, bark).
```

Some required callbacks are omitted.

Polymorphism Inspired by Abstract Class, cont'd

```
-module(dog).
-behaviour(animal).

yap() ->
    "Bow wow".
```

- How to use
  - Same as an example of interface

```
animal:start_link(dog),
animal:bark(). % Bow wow
```

Polymorphism Inspired by Abstract Class, cont'd

- Example in Kai
  - Provides two types of TCP listening processes with a single interface
  - Abstract module (behavior)
    - kai_tcp_server
  - Concrete modules
    - kai_rpc, listens RPC calls from other Kai nodes
    - kai_memcache, listens requests from memcache clients
  - See actual codes in detail

Lessons Learnt

- Two approaches to implement polymorphism
  - Inspired by interface
    - Simple
    - Not efficient
      - Actual logics have to be implemented in each child
  - Inspired by abstract class
    - Erlang-way
    - Efficient
      - Abstract class can be shared by children

Summary

Outline

- Reviewing Dynamo and Kai
  - Kai: an Open Source Implementation of Amazon's Dynamo
  - Features and Mechanism
- Process Design for Better Performance
  - Process Design Based on Calling Sequence and Process State
- Polymorphism in Actor Model
  - Two approaches to implement polymorphism

Kai: Roadmap

1. Initial implementation (May, 2008)
   - 1,000 L

2. Current status
   - 2,200 L
   - Following tickets done

Module	Task
kai_coordinator	Requests from clients will be routed to coordinators
kai_version	Vector clocks
kai_store	Persistent storage
kai_rpc, kai_memcache	Process pool
kai_connection	Connection pool