software pilots
TRIFORK.

Actors – Towards Object Oriented Modeling of Concurrent Systems

Kresten Krab Thorup, Trifork

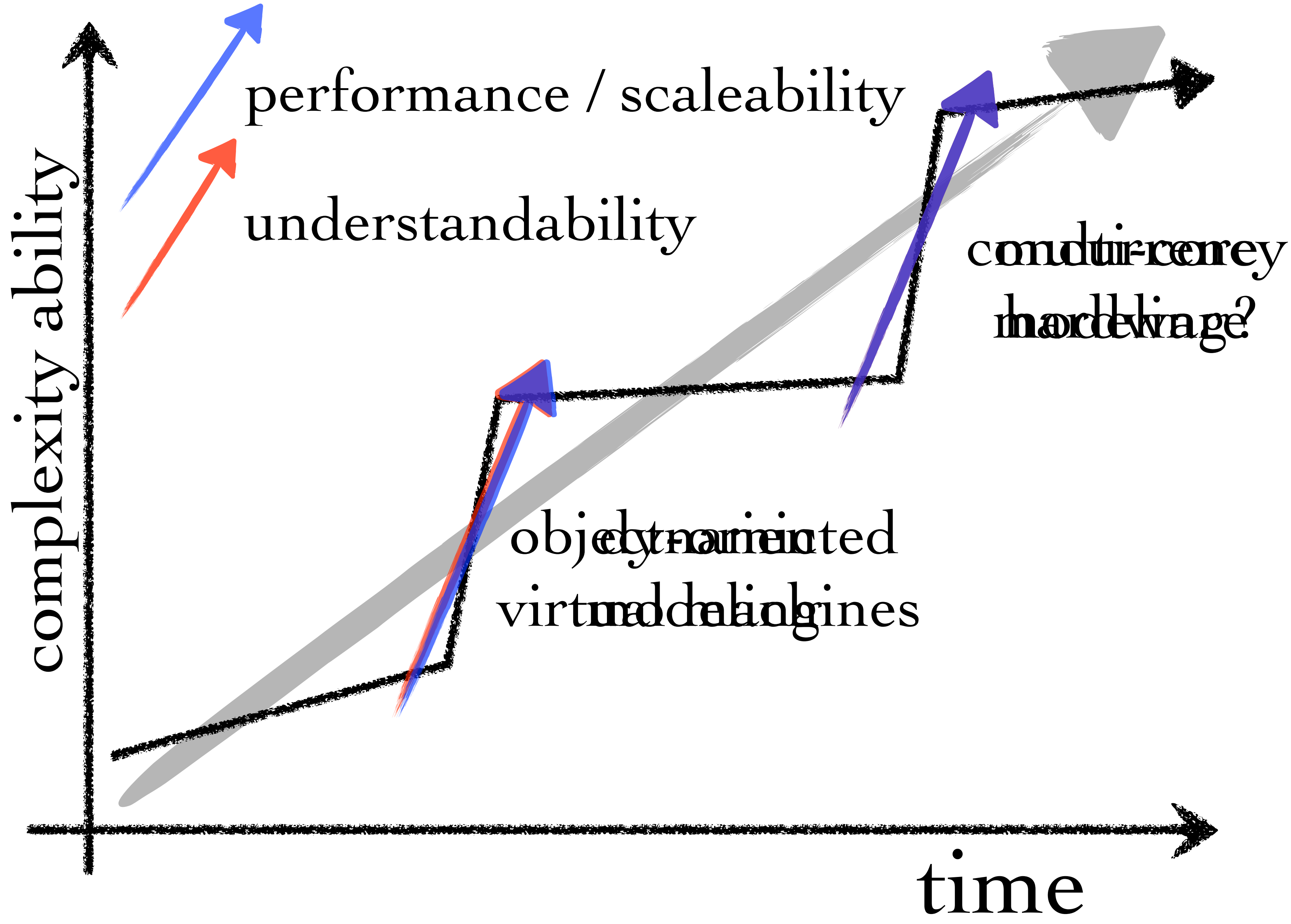
I’m no expert

I’m on a mission to figure out how to “think concurrently”.

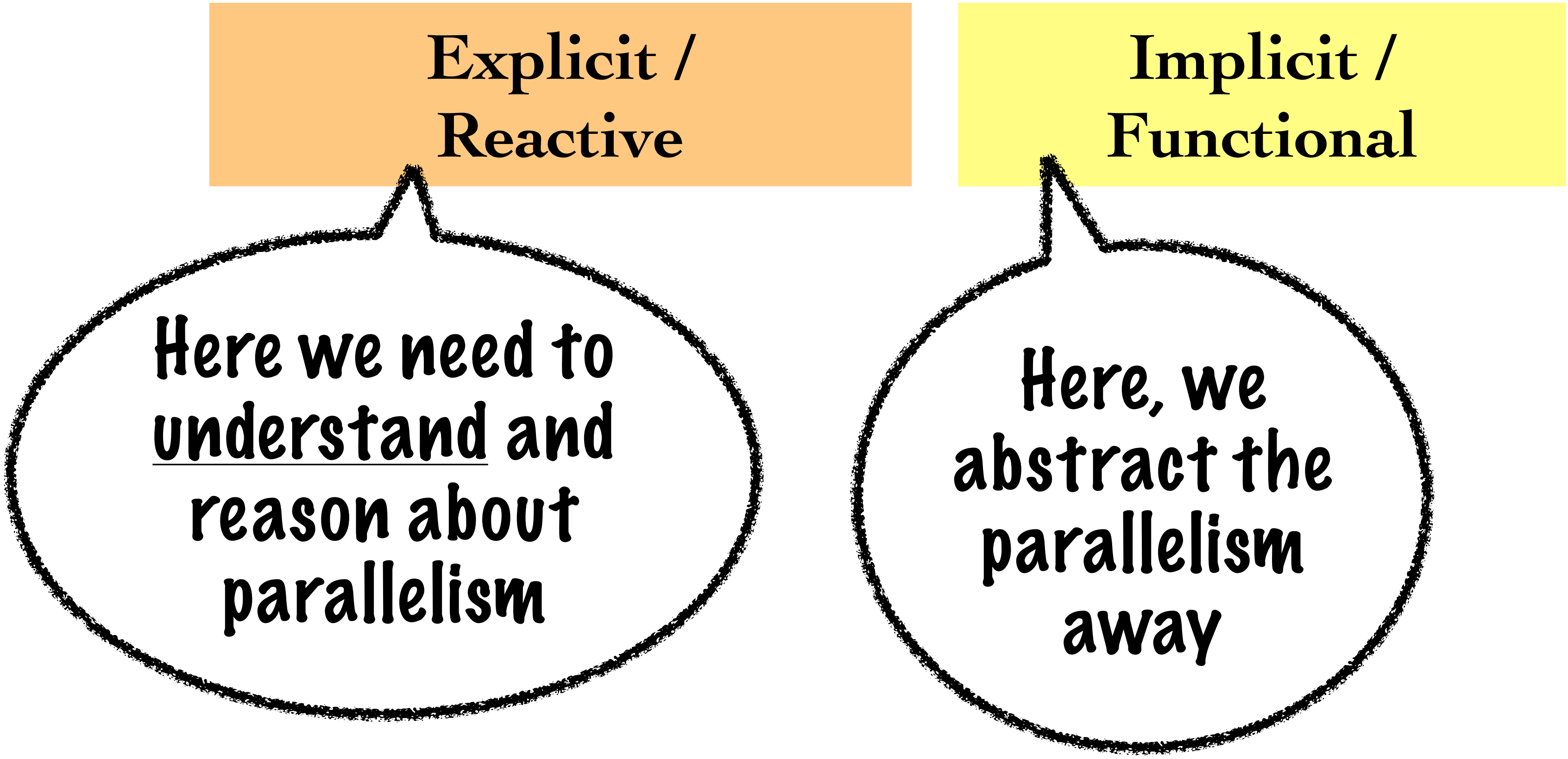
What factors increase our Capacity for Complexity?

A. Our system's ability to perform and scale as problem size grows.

B. Our ability to understand and reason about systems as they grow big.


performance / scaleability
understandability
complexity ability
object-oriented
dynamic
virtual machines
modelling
concurrency
multi-core
hardware?
modelling
time

Concurrency Landscape



Explicit / Reactive

Implicit / Functional

Here we need to understand and reason about parallelism

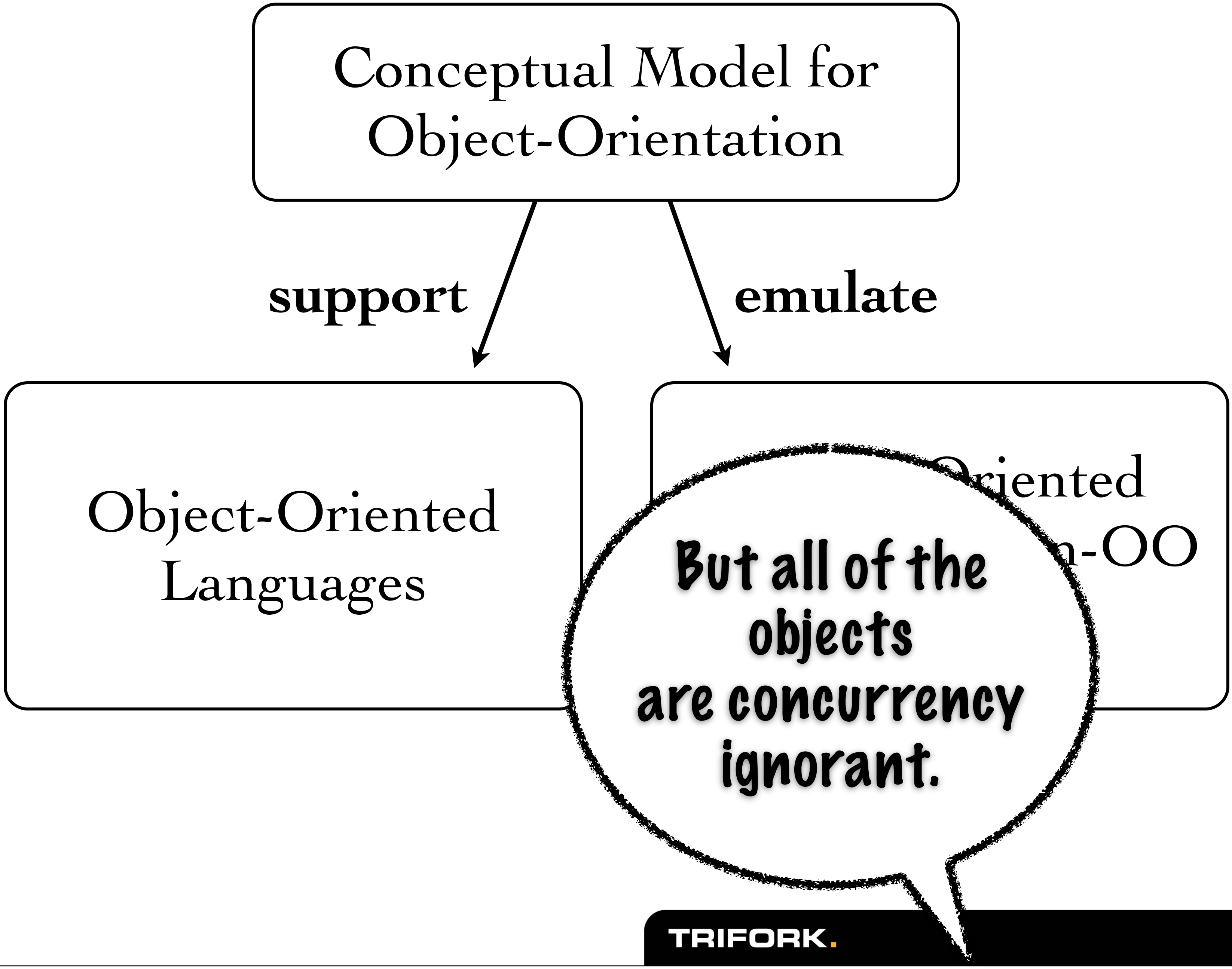
Here, we abstract the parallelism away

Concurrency Landscape

	Explicit / Reactive	**Implicit / Functional**
Distributed	Telephone Systems Trading Systems **Erlang / CORBA** **Message Middleware**	Search Engine Indexing Model Simulations, Weather Forecasts
Local	GUI-applications Control Systems **Threads**	**Google/Hadoop** **Map-Reduce** **Data-Parallelism**

“Thinking Tools” of Object-Oriented Modeling

objects with identity,
classes with specialization,
virtual methods,
... and patterns.



Conceptual Model for Object-Oriented

support

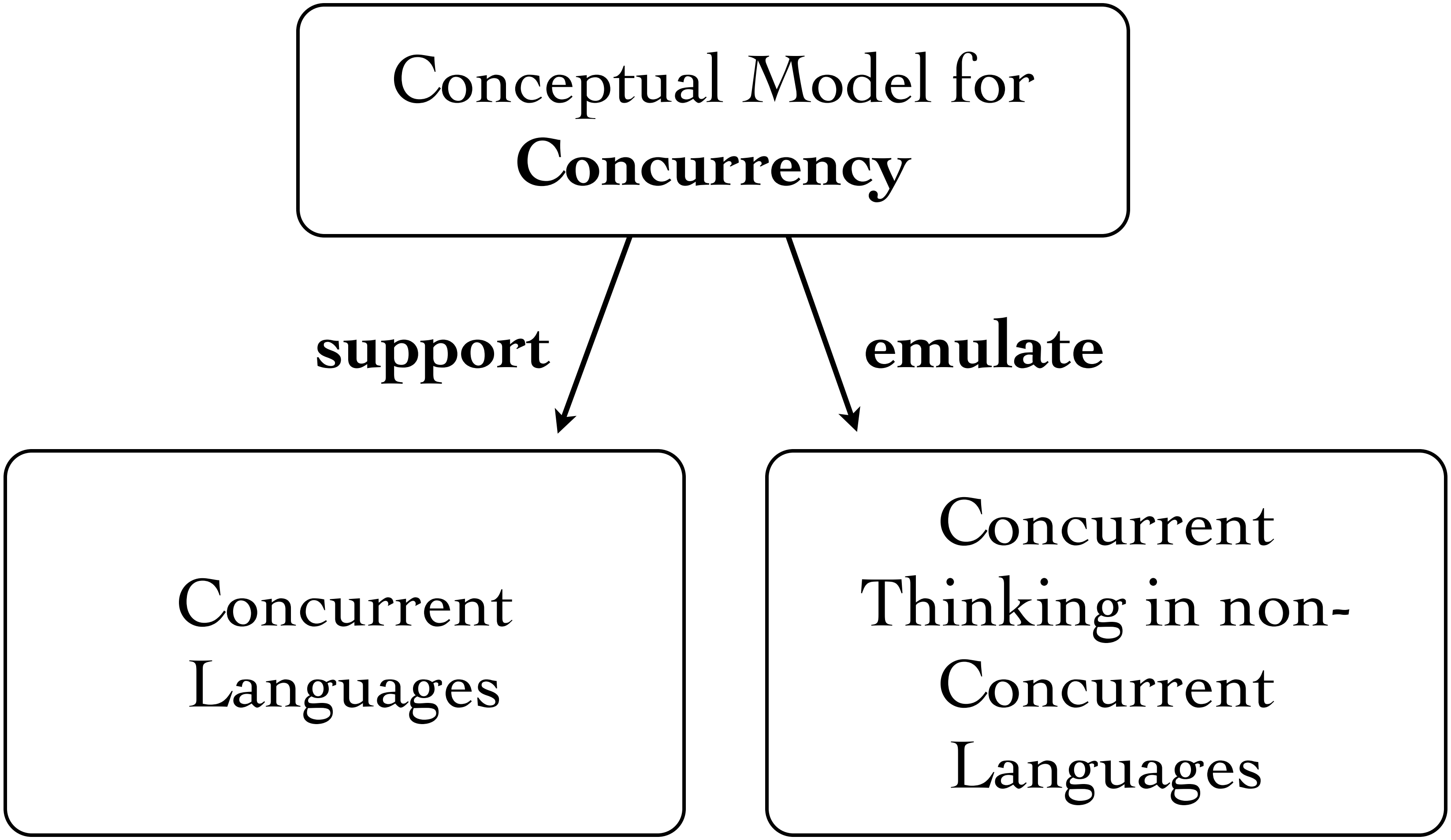
emulate

Object-Oriented Languages

Oriented

n-OO

But all of the objects are concurrency ignorant.



Conceptual Model for **Concurrency**

support

emulate

Concurrent Languages

Concurrent Thinking in non-Concurrent Languages

Where is the Conceptual Model for Concurrent (Object-Oriented) Programming?

Concurrency Mechanisms

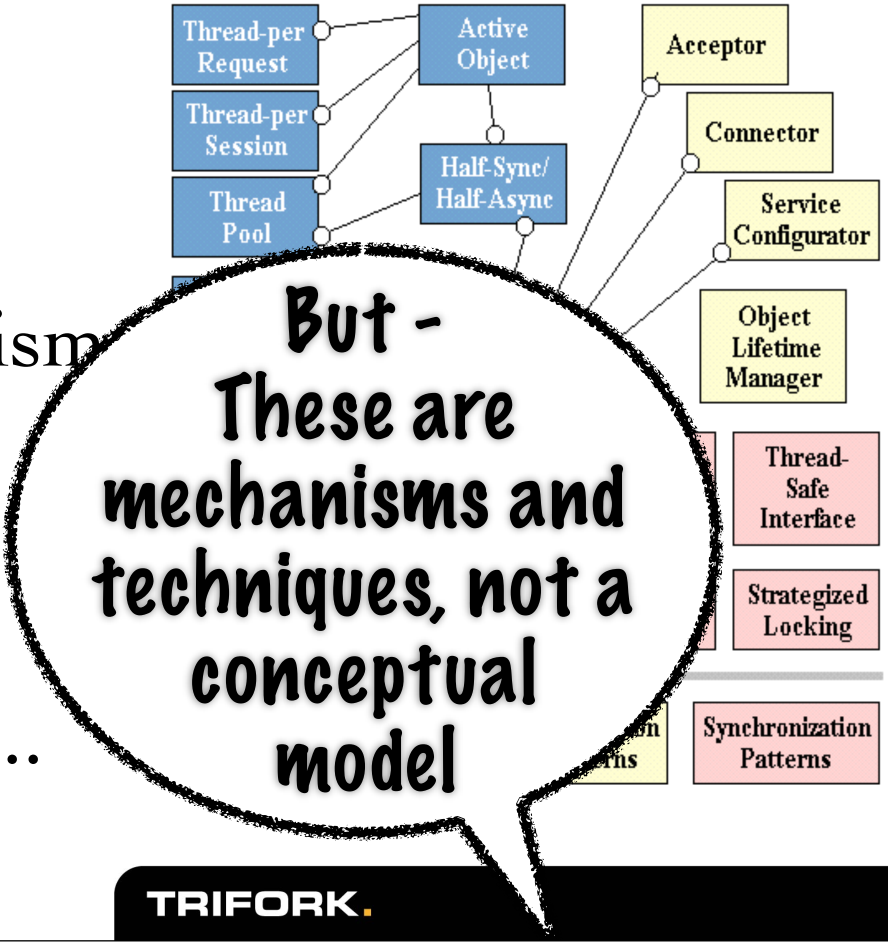
Runtime

Threads, Processes, Semaphores, Locks, Monitors, Condition Variables, Data-Parallelism

Formalisms

CSP, π-calculus, concurrent linear logic, …

Patterns



But - These are mechanisms and techniques, not a conceptual model

Actors

have the potential to provide
an OO conceptual model
for concurrency

Some Actor Systems

- C.E. Hewitt's actor model [Hewitt, 1977]
- SAL (Simple Actor Language) [Agha, 1986]
- ABCL/1 [Yonezawa, 1986]
- Concurrent Smalltalk [Tokoro, 1986]
- Actra Smalltalk [Thomas, et.al., 1989]
- Erlang [Armstrong, 1988]
- Clojure [Hickey, 2008], Kilim, ...

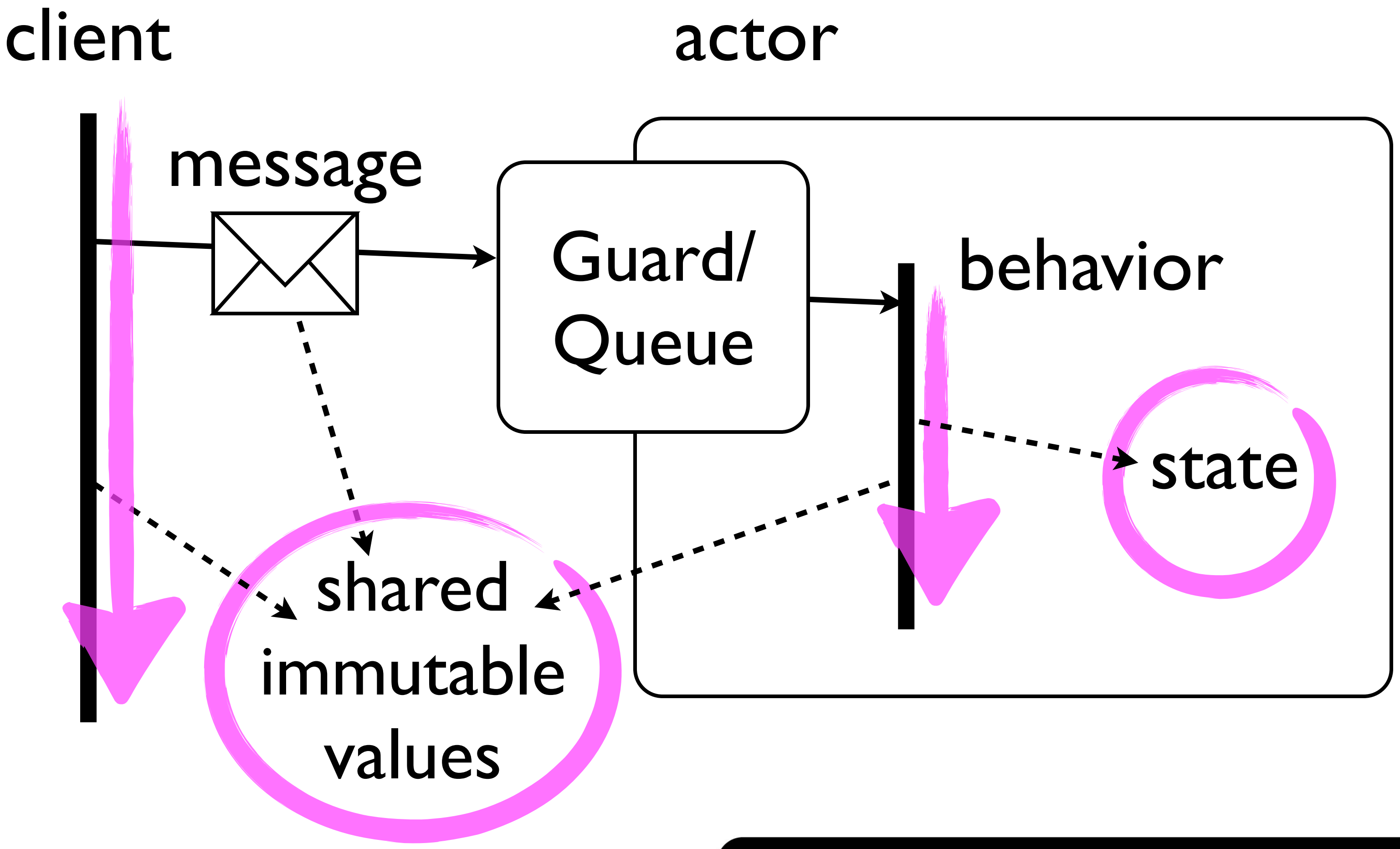
Some More Systems

- **Scala** has a nice framework for programming with actors.
- **Kilim, Jetlang, Actors Guild,** and **Actor Foundry, ...** are frameworks for actor programming in Java.
- **Axum** is an actor language based on C#.

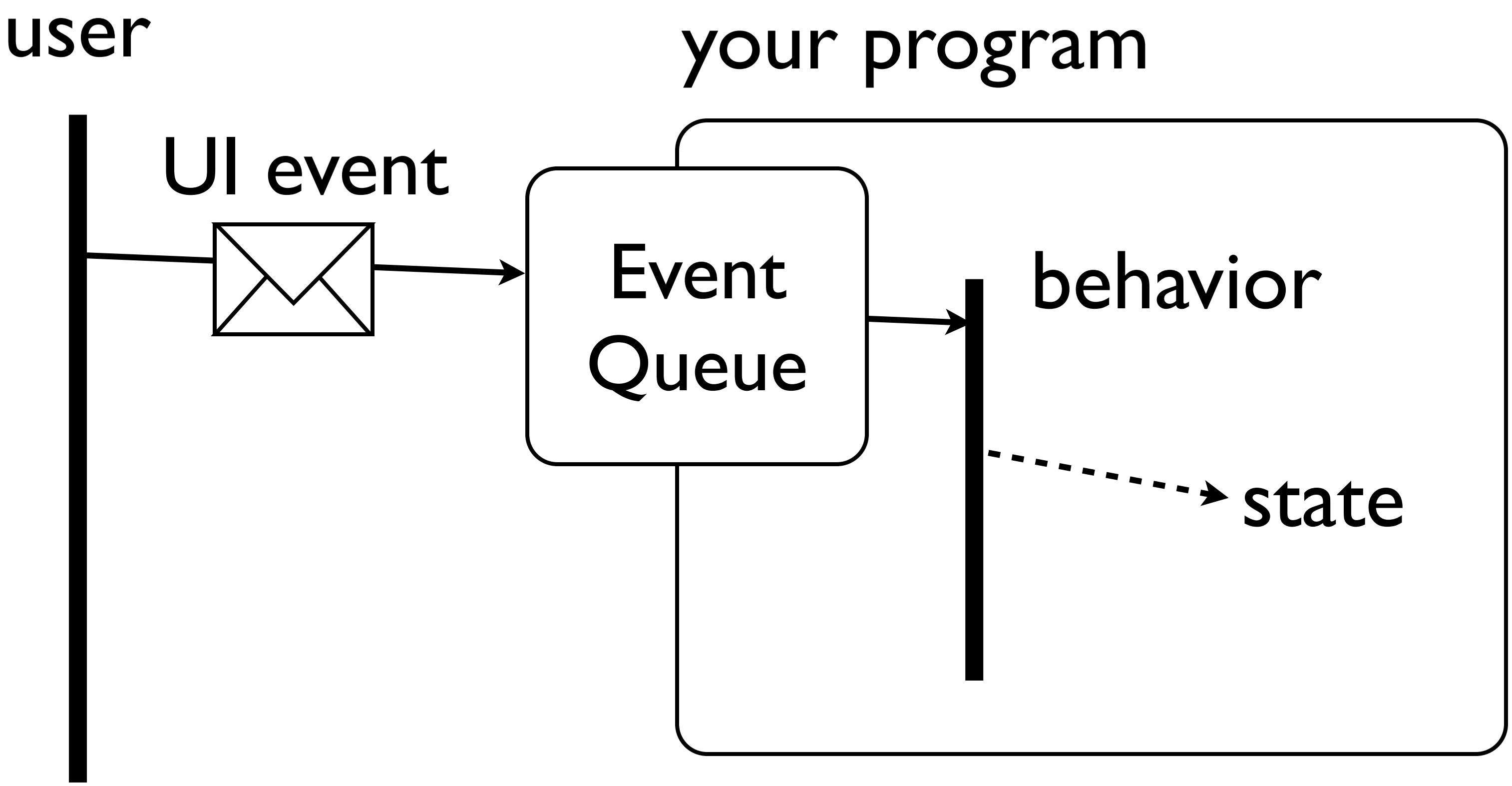
An actor model...

- Is a conceptual model for time/state management
- Is a conceptual model for computations and their concurrent execution
- Mechanisms for abstraction and composition

Actor Essentials...



You know this...



Gul Agha's Actor Model

If there is time, I can go through this, as it solves many of the "issues" discussed in previous slides.

- In this model, an actor is...
  - A **mail queue** (with identity), and
  - A **behavior**, describing the state and what to do when a message arrives.
- In many ways, Erlang is similar to this model.

[Gul Agha, 1986]

An actor's behavior can

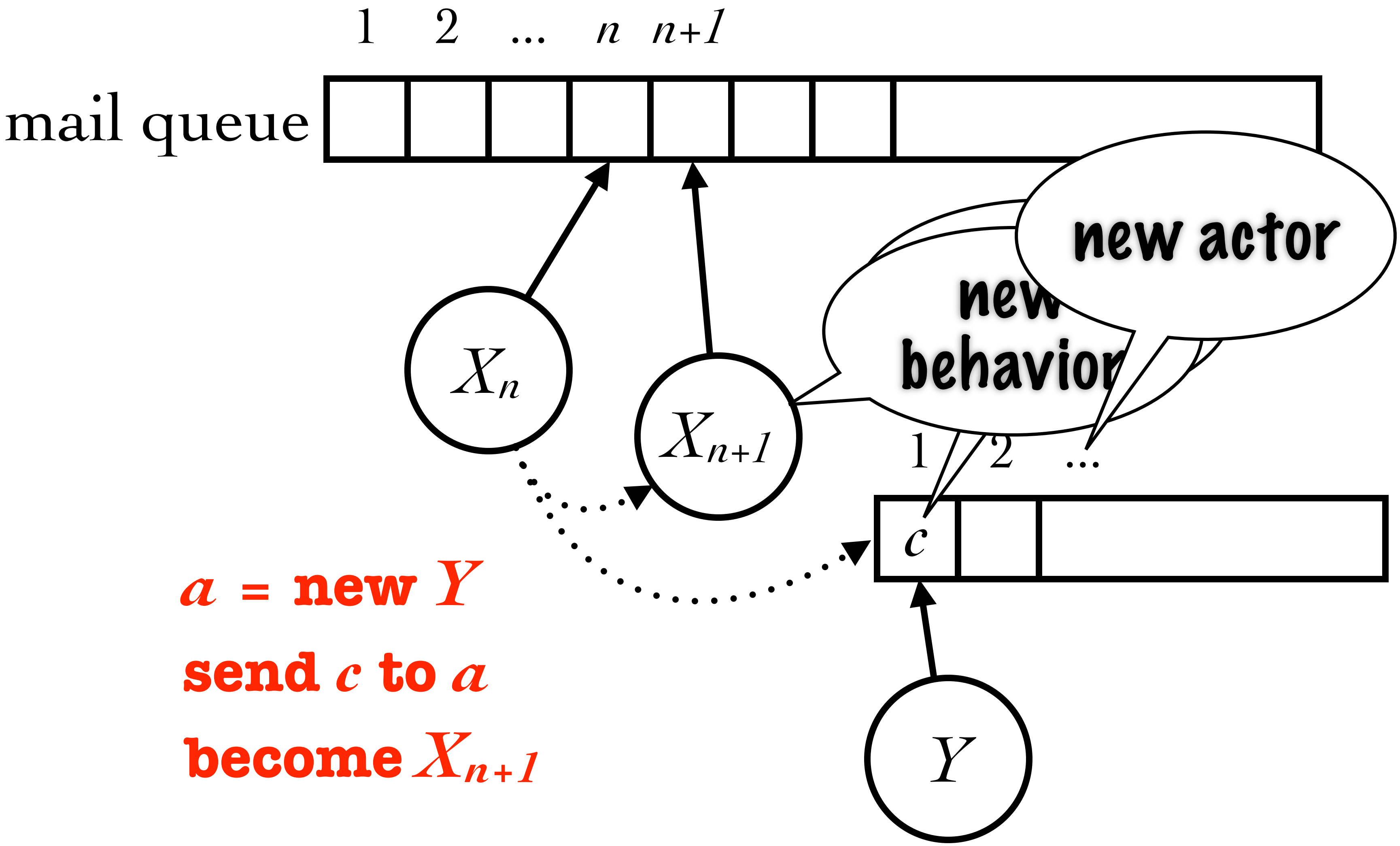
- Perform computation, if-then-else, ...
- Create new actors,
- Send messages to other actors
- Specify that the next message should be processed with a different behavior.

Message processing

- Messages are processed asynchronously: “send” starts a new processing task.
- In Agha’s actor model, a message task can execute when either

  1. The previous behavior completes, or
  2. A replacement behavior is given.

  which ever comes first.



a = **new** Y

send c **to** a

become X_{n+1}

Two things that introduce concurrency

- Message send, lets the receiving actor start processing concurrently.
- Become, lets the actor process the next message concurrently.

A simple cell

```
$\mathbf{behavior}\ \mathrm{cell}(\mathit{value})[\mathit{msg}] \equiv$
  if $\mathit{msg} = \langle \mathrm{FETCH}, \mathit{client} \rangle$ then
      $\mathbf{send}\ \mathit{value}\ \mathbf{to}\ \mathit{client}$
  if $\mathit{msg} = \langle \mathrm{STORE}, \mathit{value}_2 \rangle$ then
      $\mathbf{become}\ \mathrm{cell}(\mathit{value}_2)$

$x = \mathbf{new}\ \mathrm{cell}(0)$
$\mathbf{send}\ \langle \mathrm{STORE}, 1 \rangle\ \mathbf{to}\ x$
```

How we are Modeling Behavior

- Event Loops
- State Tables / State Machines
- Actor Languages
  - E, Actra, Erlang

Actor Languages

- Structure your program as <u>many</u> concurrent event loops.
- Messages between actors (events) are asynchroneous.
- This seems to introduce a lot of complexity; we cannot apply our linear thinking.

Actor Languages

- You need to think of your program as a team collaboration
- Apply organization theory to program behavior
    - Secretaries, Workers, Managers, Gate keepeers, Cleaners,
    - Hierarchical / Agile, Kanban, ...
    - Supply chain, warehousing,

With N+1 on a Team you need to...

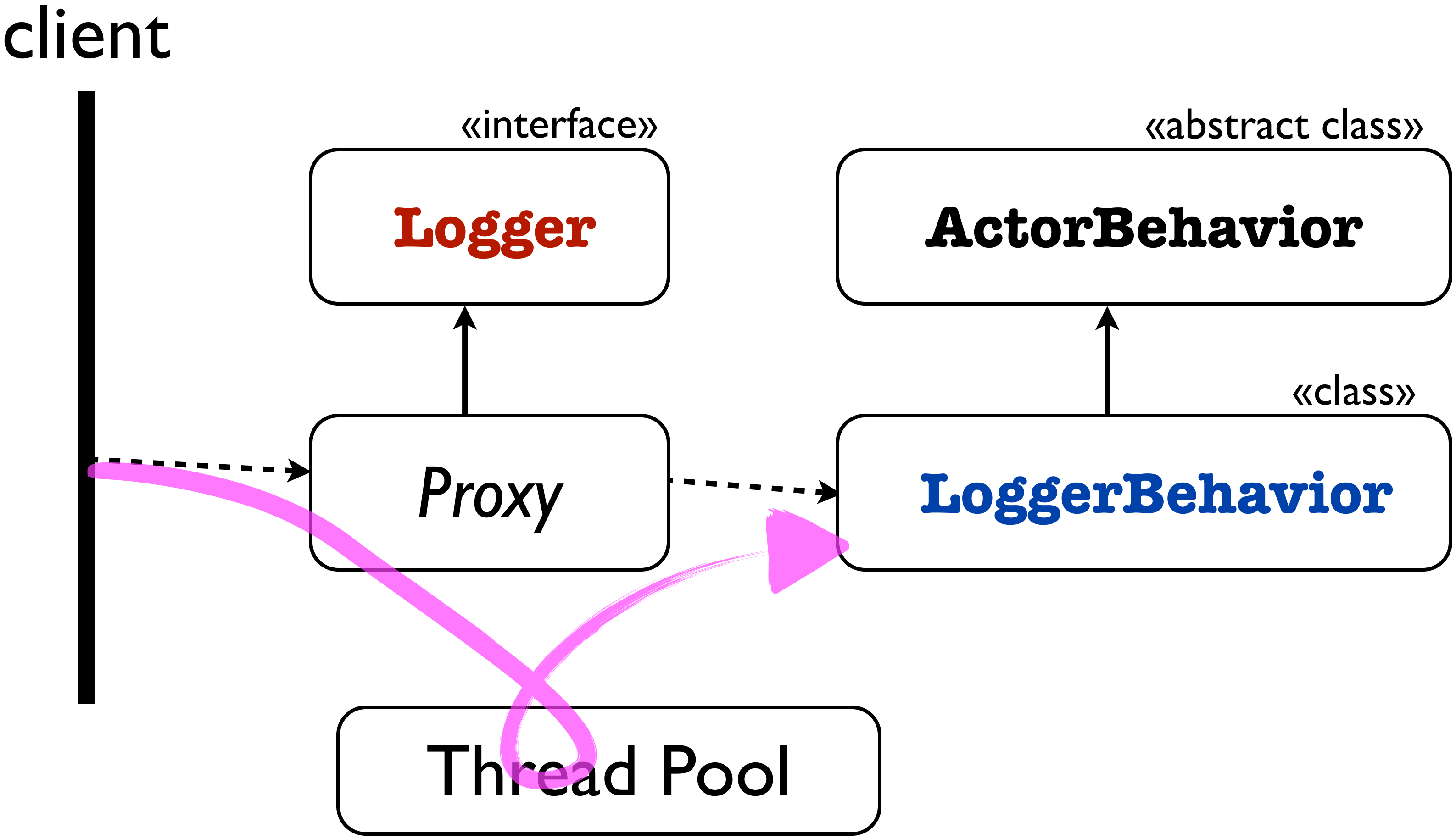
- Manage ordering of events (protocol)
- Manage shared resources (facilities)
- Throttle/Scale work load (workload)
- Hide implementation details

Understanding Actors

Learning by Doing: Don't Dissect the Frog, Build It.

- To really understand actors, I wrote a simple actor framework for Java.
- Each "actor" has an interface, and a behavior that implements that interface.
- The framework creates a proxy that implement the interface and dispatches via a thread pool...

Java Actor Framework



Java Actor Framework

```
// the actor’s interface
interface Logger {
    void log(String val);
}

// ... and it’s behavior
class LoggerBehavior extends ActorBehavior<Logger> {
   void log(String val) { System.out.println(value); }
}

// ... then use it like this...
Logger logger = new LoggerBehavior().actor();
logger.log(“Something happened”);
```

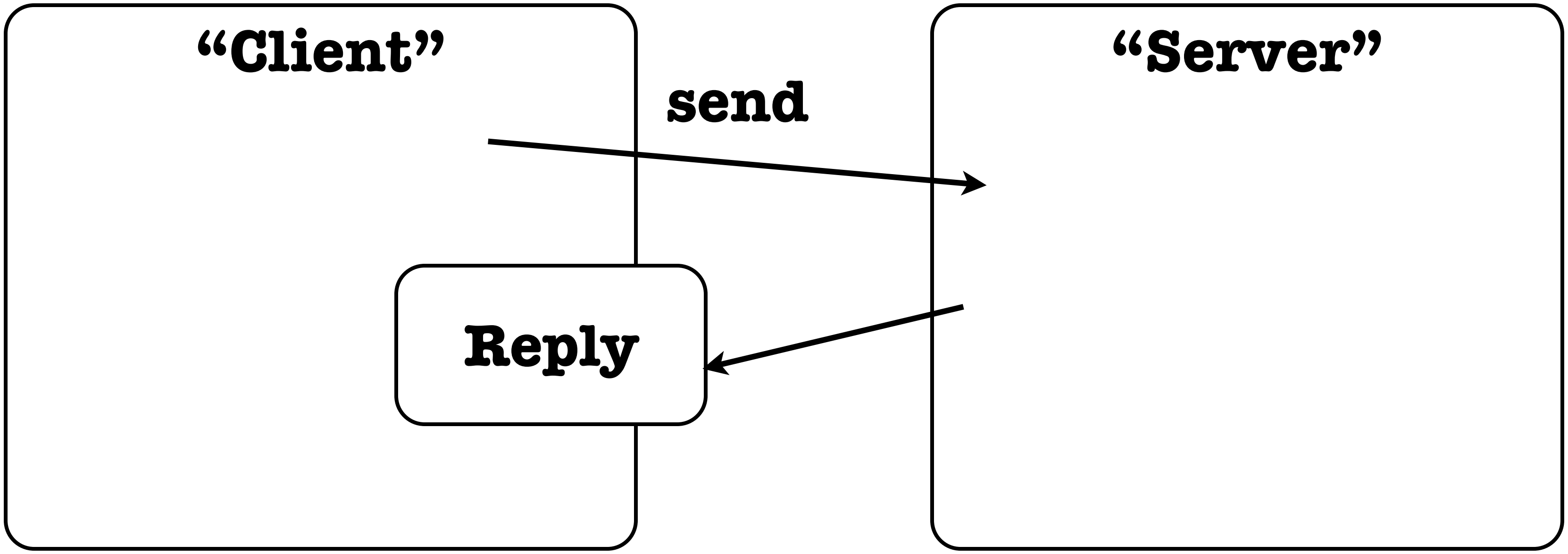
Issues with this approach

Sharing. If an actor receives a reference to a shared object then multiple actors/threads may mutate that object concurrently.

Threads. If an actor blocks during it's operation, it is holding a precious resource, namely a thread.

Concurrency. If the actor's methods returns a value, then the client will block, or what?

Async Reply (a.k.a. Future)



Asynchronous Reply

```
// the actor's interface
interface Logger {
   Reply<String> getStatus();
}
```

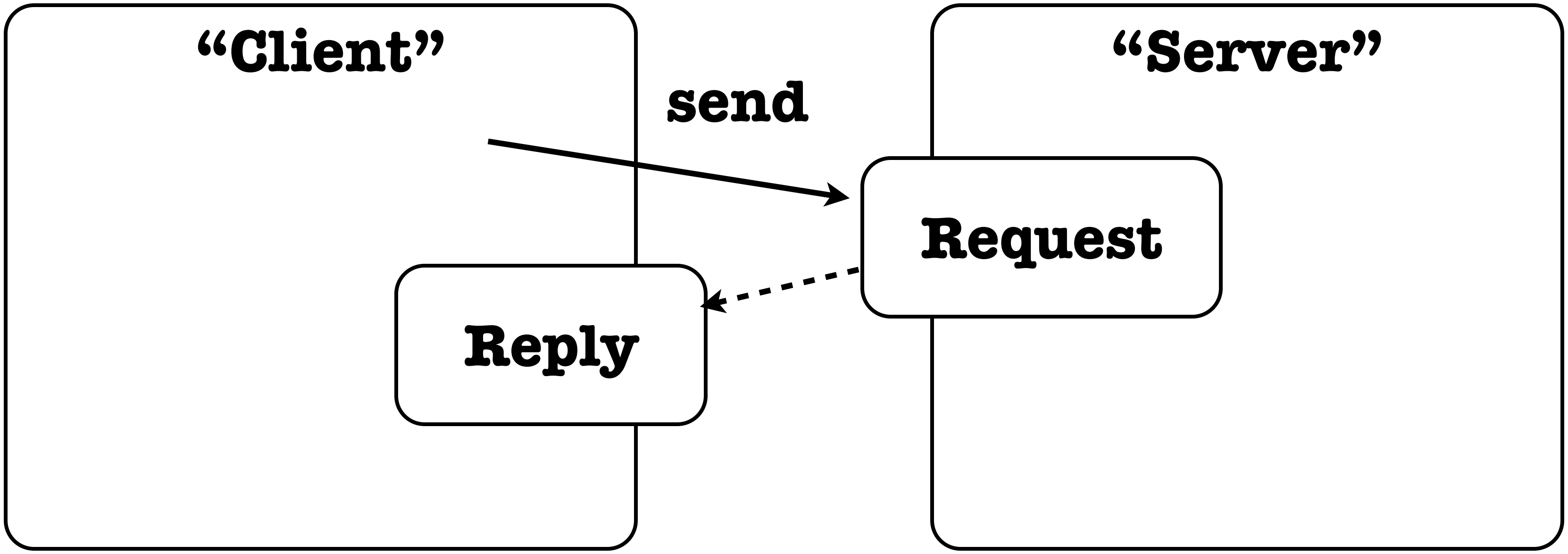
These two correspond

```
class LoggerBehavior extends ActorBehavior<Logger> {
  String getStatus() { return ⟨Compute Status⟩; }
}
```

Asynchronous Reply

```
// ... then use it like this...
Logger logger = new LoggerBehavior().actor();

// get a “future” for the status response
Reply<String> reply = logger.getStatus();

// try to get the response
String status = reply.get();
```

Async Request



Async Request/Reply

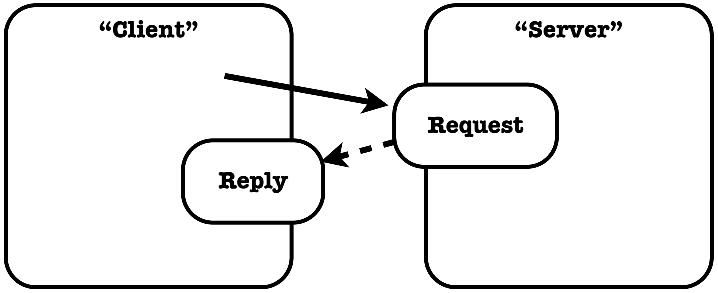
```
interface Reply<T>{
   T get();
}

interface Request<T> {
  void answer(T value);
}
```

Async Request/Reply

These two correspond

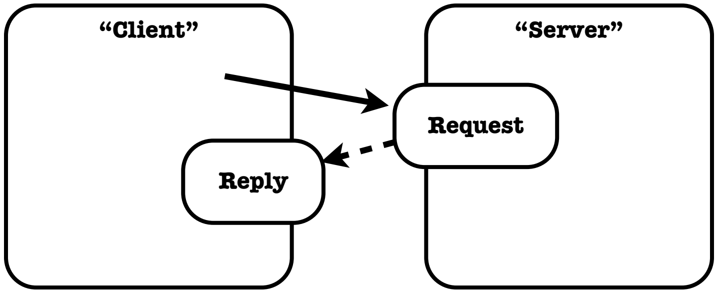
```
// the actor's interface
interface Logger {
   Reply<String> getStatus();
}

class LoggerBehavior extends ActorBehavior<Logger> {
  void getStatus(Request<String> req){
    req.answer ( ⟨Compute Status⟩ );
    ... continue computation ...
  }
}
```

Async Request/ Reply Pattern



- A generalized model for request/reply interactions, that enables deferring the decision of
  - when (and how long) to wait for a reply
  - when to answer a request
- "Feels" like the interactions we have with agents in the real world.

Async Request/ Reply Pattern



Original actor languages provide "only" one-way asynchronous message send

- a good building block, but ...
- asynch request/reply provides a way to bridge the gap to our classic request/ reply thinking.

Async Request/Reply

```
interface Reply<T> extends Future<T>{
    T get() throws Exception;
    void forwardTo(Request<T> sink);
}

interface Request<T> {
   void answer(T value);
   void deny(Exception e)
}

interface Filter<IN,OUT> extends
      Request<IN>, Reply<OUT> {
}
```

Variations

- Actor languages/frameworks provide different variations of the async request/reply
  - Original Actor Model
  - E Programming Language
  - Erlang
  - Actra (OTI's concurrent smalltalk)

Promises in E

```
// ... then use it like this...
Logger logger = new LoggerBehavior().actor();

// get a “future” for the status response
Reply<String> reply = async logger.getStatus();

// install “callback” for the async reply
reply.when( fun(String s) { ... use s ... } );

// ... will run in “this thread” to avoid races/sharing.
```

	Async Send computation can continue after message send	Async Reply computation can continue after message reply	Message Queue messages are queued or synchroneous
E	YES	NO	YES
Actra	NO	YES	NO
Erlang	YES	YES	YES

Sharing & Threads

An actor language should also provide isolation for actors, so that multiple actors don’t mutate each others / shared state.

Threads are evil - actor languages provide light-weight processes. Your thinking changes dramatically when threads are very cheap.

Kilim Framework

Sharing: The **Kilim** framework rewrites and validates Java byte code to check this. Object references become **null** in the sender's context.

Threads: Kilim rewrites the actor behavior to CPS (continuation passing style), permitting actors to "suspend" without holding a thread.

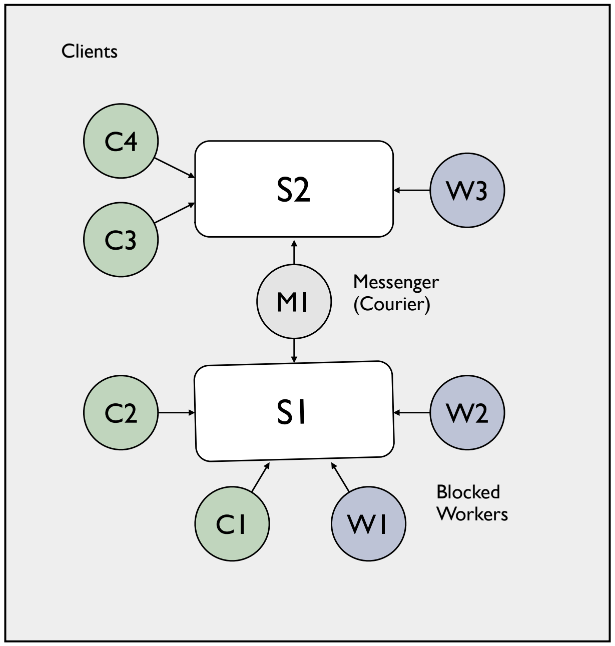
Scala Actor Framework

Sharing: Scala makes it easy to write immutable classes/values, but there is no mechanism to guarantee avoiding sharing.

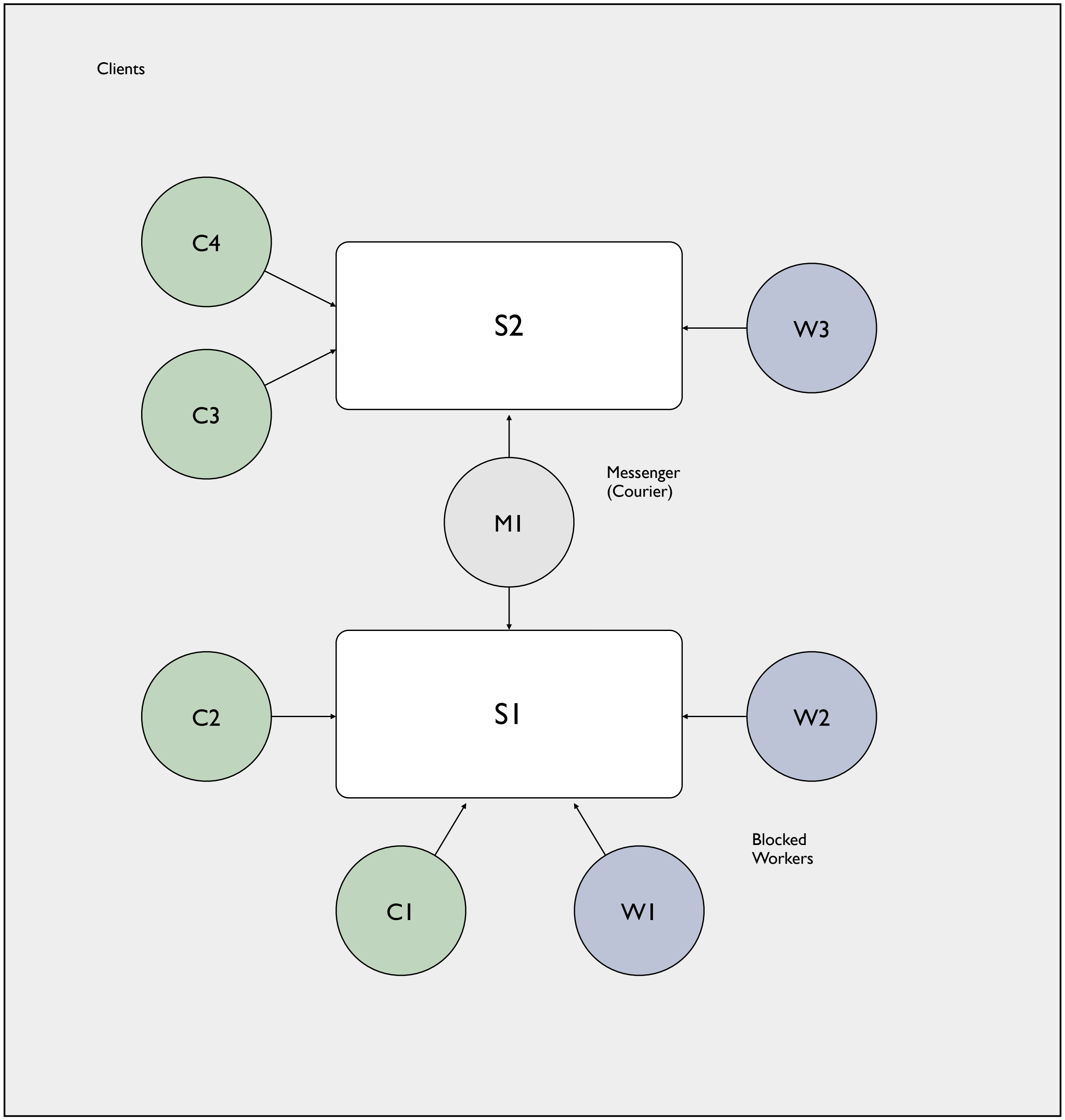
Threads: Scala provides for a model in which you avoid having threads for idle actors, but blocking operations have same issues as “my” framework.

Anthropomorphic Style

- Computations are organized in personified roles
- Managers, Administrators, Workers, Couriers, and Notifiers…
- Each of these have well known pre-defined semantics which can be subclasses for specific applications
- **Servers(Managers) must be responsive, so** delegate most of the work
  - Spend most of their life in a "receive any" loop waiting for work
- Most computation done by Workers



W. Morven Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", Software Practice and Experience, Vol. 11, Pp. 435-466, 1981.


Clients
C4
C3
S2
W3
Messenger
(Courier)
M1
C2
S1
W2
Blocked
Workers
C1
W1

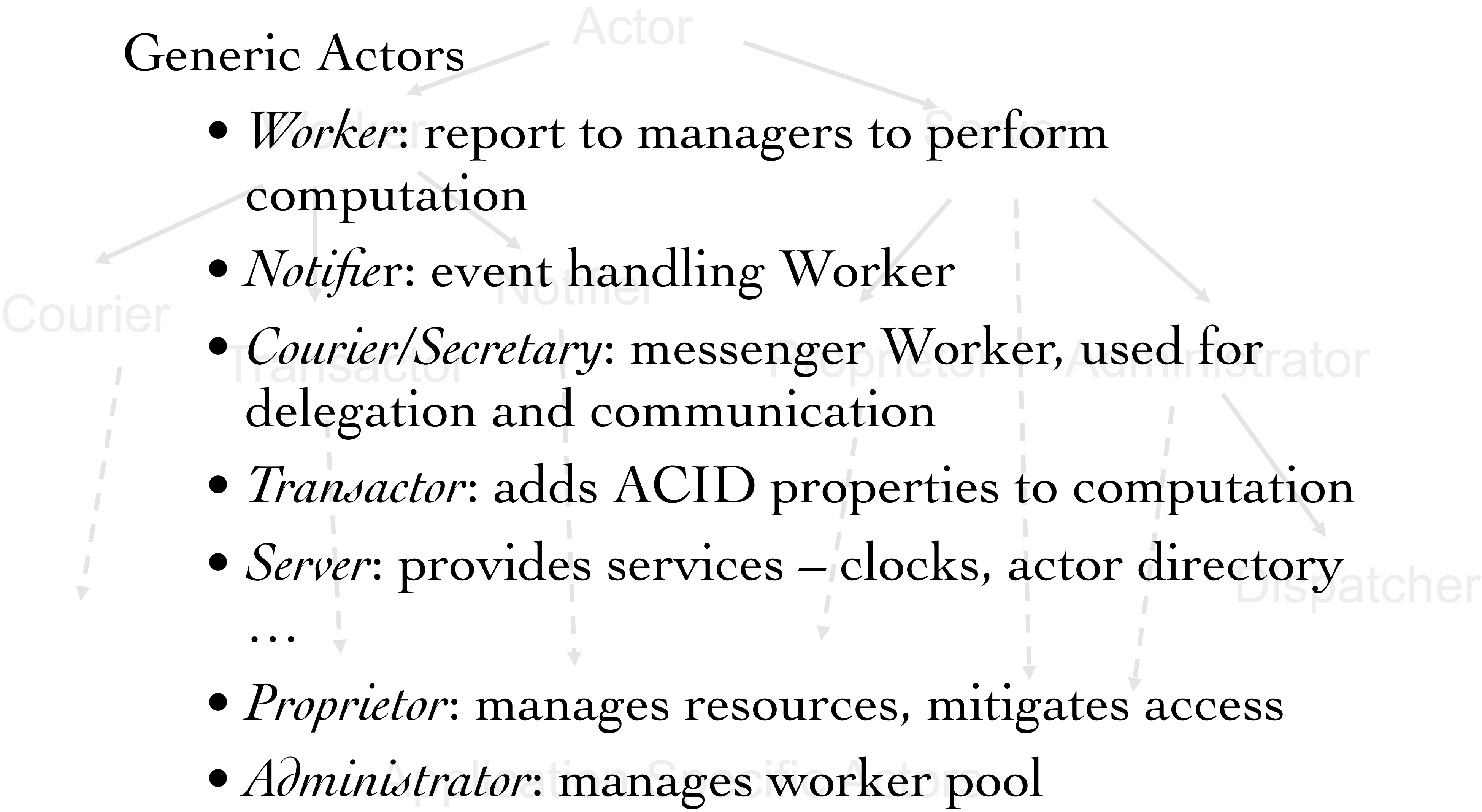
Worker

```
class Worker extends ActorBehavior {

  Worker (Manager mgr) { this.manager = mgr; }

  run() {
    while(true) {
        Work work = manager.getWork();   // blocks!
        perform ( work )
    }
  }

}
```

Manager

```
class Manager extends ActorBehavior {

  Queue<Request<Work>> workers;

  getWork (Request<Work> req) {   // from worker
        workers.enqueue(req);
  }

  handle(Question q, Request<Answer> req){
     workers.dequeue().answer ( new Work(q, req) );
  }

}
```

Actor Taxonomy



Generic Actors

- *Worker*: report to managers to perform computation
- *Notifie*r: event handling Worker
- *Courier/Secretary*: messenger Worker, used for delegation and communication
- *Transactor*: adds ACID properties to computation
- *Server*: provides services – clocks, actor directory …
- *Proprietor*: manages resources, mitigates access
- *Administrator*: manages worker pool
- *Dispatcher*: provides asynchronous

Protocol

- When you interact with an actor, it becomes apparent that you need some way to control (and talk about) the ordering of interactions.
- Java “interfaces” describe what you “may say”, but says nothing about what makes sense to say when.
- You want some kind of state machine abstraction to manage this

Protocol Enforcement

- Erlang - receive uses pattern matching, so only certain messages are accepted. Message mismatch is an error in the receiving actor!
- ABCL/x - receive can look ahead in the message queue to match certain criteria.
- Some OO-style languages have “guards” that control which messages are applicable in the current state.

Erlang Cell

```
fun cell(nil) ->
  receive
     {put, Value} -> cell(Value);
  end;

fun cell(Value) ->
  receive
     {take, Sender} ->
            Sender ! Value,
            cell(nil);
  end.
```

Coordination

- Actors don't easily provide for coordination or transaction-like behavior. ... all those asynchronous messages are rather slippy!
- In many cases, you have to write the coordination code explicitly, ... tricky!
- Many research projects have worked on this, e.g. [Frølund96, Callsen94, Varela01].

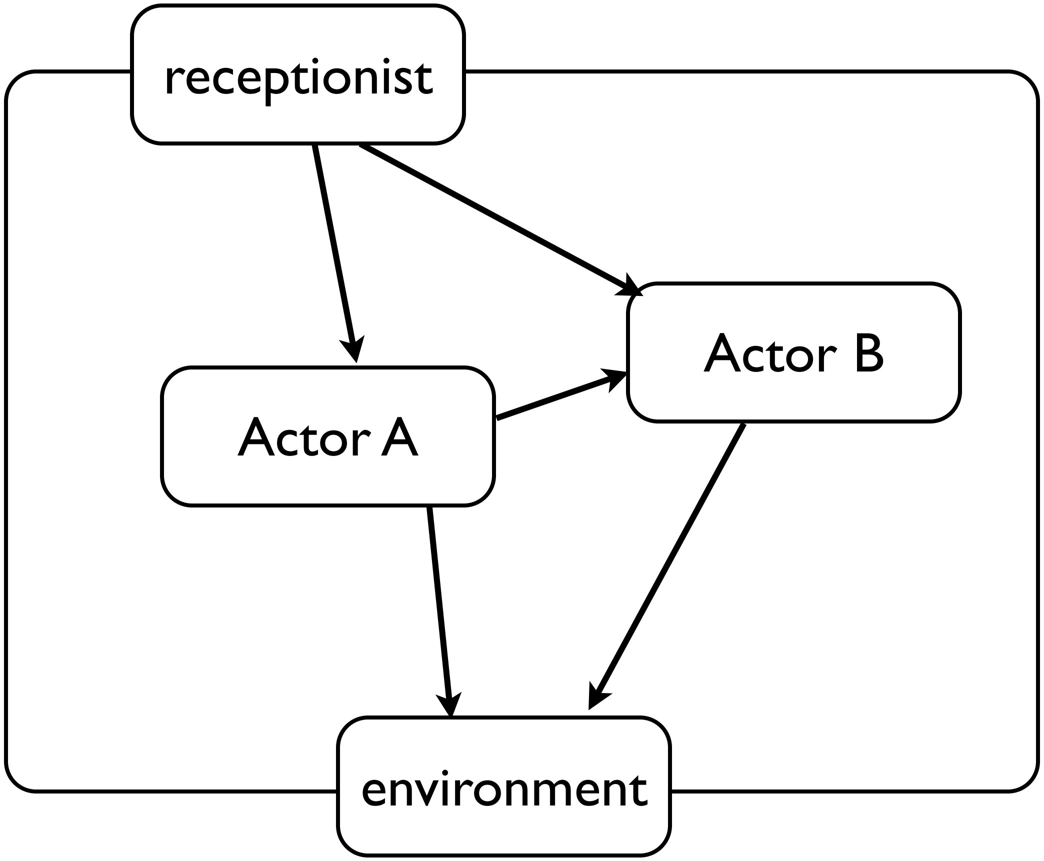
Transactions & Actors

- Clojure has transactional cells built-in, otherwise known as “refs”.
- In Erlang and Actra, you would program these using a framework
  - Actra - inherit “Transactor”
  - Erlang - Use “tx_server”

- **An Actor Model** needs to address
  - Resources
  - Sharing
  - Asynchronous Messaging
- But also (patterns for) ...
  - Composition,
  - Abstraction, and
  - Coordination.

Encapsulation & Comp

I'm taking the Erlang full-day tutorial on Sunday, perhaps I'll be enlightened here.



receptionist

Actor A

Actor B

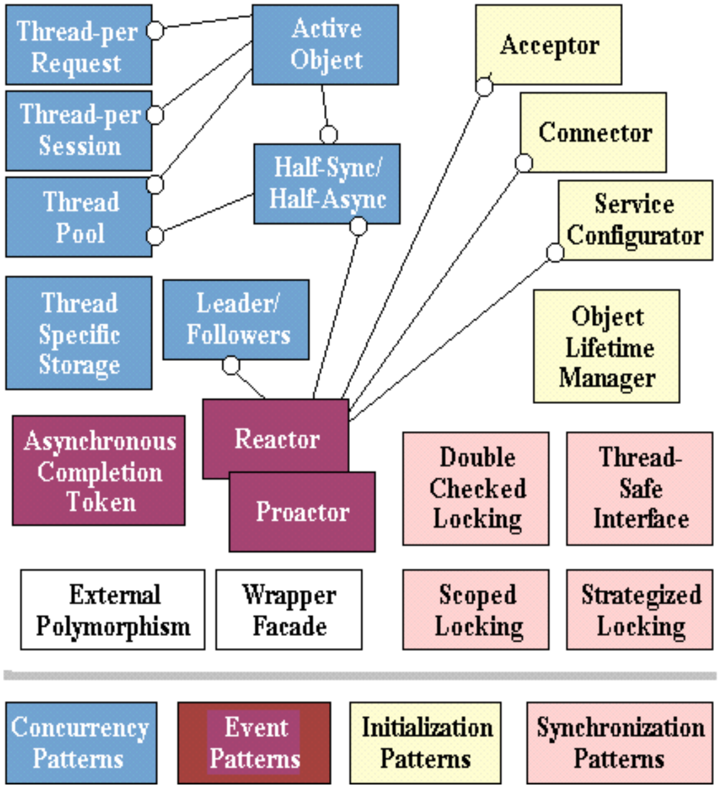
environment

Abstraction

- Some actor languages have reflection (ABCL/R* family), or higher-order actors (Erlang), i.e., actors that produce or consume actor behaviors. In Erlang, an actor behavior is simply a function.
- These mechanisms are very powerful for creating control structures, and meta-programming for actors.

Actor Patterns

- Active Object, Pipes-and-Filters
- All of Gregor's Integration Patterns [Messaging]
- Anthropomorphic Patterns



Thanks!