software pluss
TRIFORK.

Actors — Towards Object Oriented Modeling of Concurrent Systems

Kresten Krab Thorup, Trifork
Dave Thomas, Bedarra

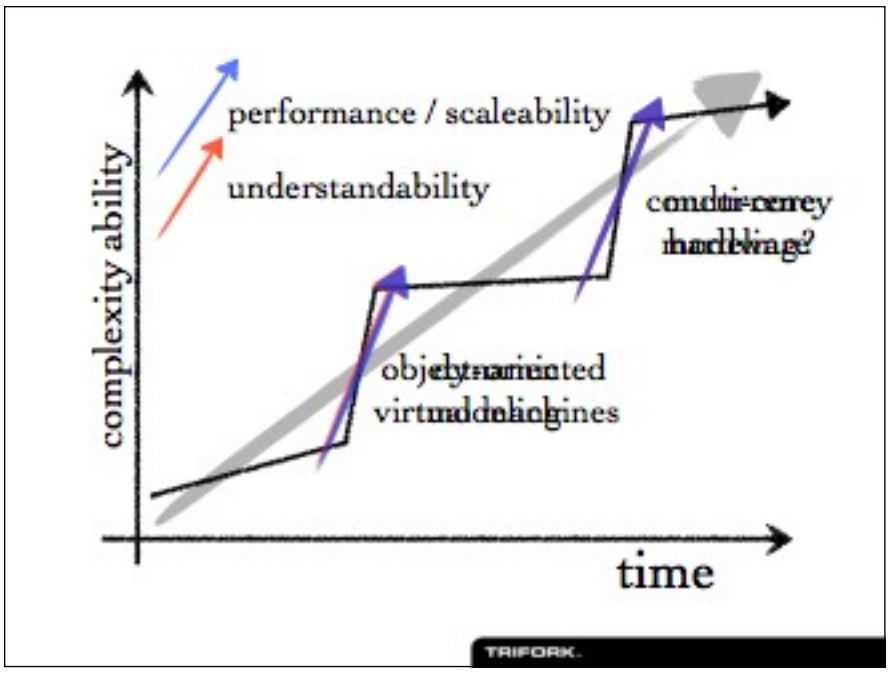
A Talk in Two Acts

ACT I, in which Kresten speaks on
Modeling Concurrency with Actors

ACT II, in which Dave explains
The Actra Programming Model

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.



Concurrency Landscape

Explicit / Reactive	Implicit / Functional
Here we need to understand and reason about parallelism	Here, we abstract the parallelism away

TRIFORK.

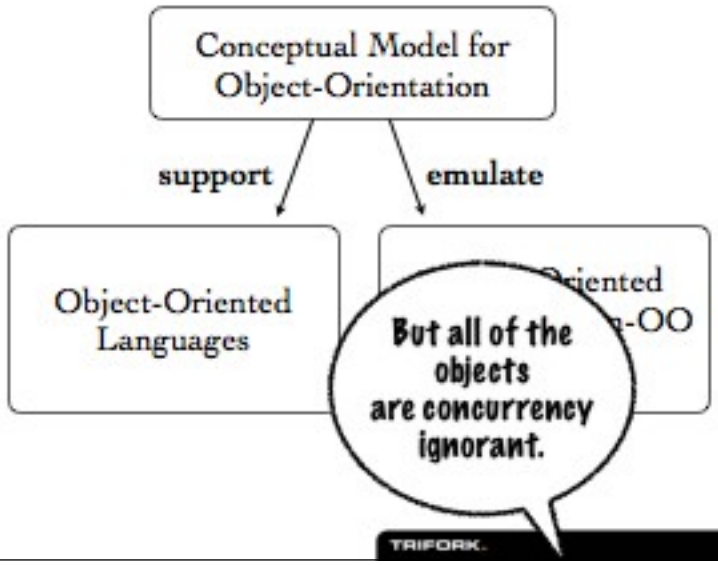
Concurrency Landscape

	Explicit / Reactive	Implicit /
Distributed	Telephone Systems Trading Systems Erlang / CORBA Message Middleware	This area has big potential to change programmers every-day life for the better!
Local	GUI-applications Control Systems Threads	Google/Hadoop Map-Reduce Data-Parallelism

TRIFORK.

"Thinking Tools" of Object-Oriented Modeling

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



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

Concurrency Mechanisms

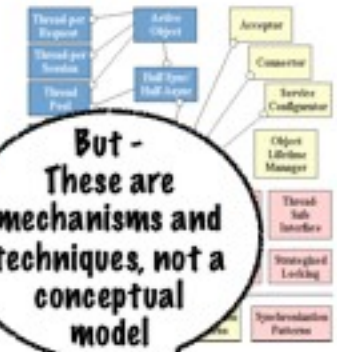
Runtime

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

TRIFORK.

Actors

have the potential to provide an OO conceptual model for concurrency

...and can help generating more concurrent work

TRIFORK.

Some Actor Systems

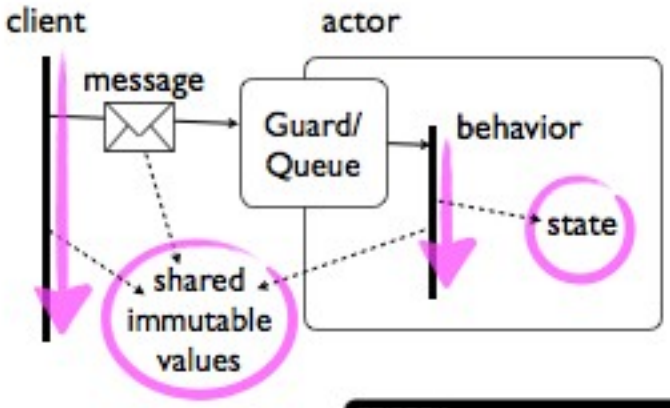
- C.E. Hewitt's actor model [Hewitt, 1977]
- SAL (Simple Actor Language) [Agha, 1986]
- ABCL/1 [Yonezawa, 1986]
- Concurrent Smalltalk [Tokoro, 1986]
- Actra Smalltalk [Thomas, et.al., 1989]
- Erlang [Armstrong, 1988]
- Clojure [Hickey, 2008], Kilim, ...

TRIFORK.

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

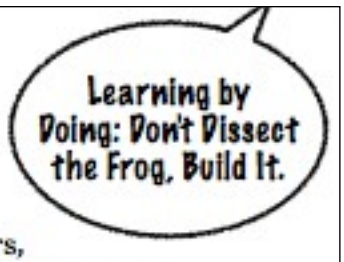
Actor Essentials...



Some More Systems

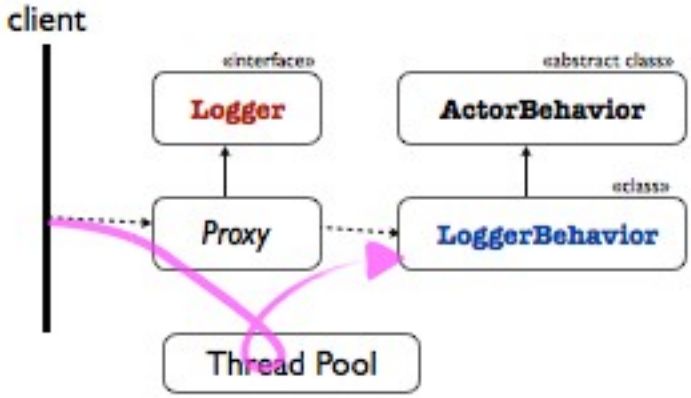
- **Scala** has a nice framework for programming with actors.
- **Kilim, Jetlang, Actors Guild,** and **Actor Foundry, ...** are frameworks for actor programming in Java.
- **Axum** is an actor language based on C#.

Understanding Actors



- 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?

Asynchronous Replies

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

These two correspond

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


Asynchronous Replies

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

// get a "future" for the status response
ReplySource<String> future = logger.getStatus();

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

Asynchronous Replies

```
interface ReplySource<T> extends Future<T>{
  T get();
  void sendTo(ReplySink<T> sink);
}

interface ReplySink<T> {
  void put(T value);
}

interface ReplyPipe<IN,OUT> extends
    ReplySink<IN>, ReplySource<OUT> {
}
```


Asynchronous Replies

These two correspond

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

class LoggerBehavior extends ActorBehavior<Logger> {
  void getStatus(ReplySink<String> sink){
    sink.put (⟨Compute Status⟩);
    ... continue computation ...
  }
}
```


Recursive Factorial

```
factorial (from, to) {
  switch(to-from) {
    case 0: return to;
    case 1: return to*from;
    default:
      mid = (from+to)/2;
      return factorial(from, mid)
            * factorial(mid+1, to);
  }
}

factorial (1, 20);
```

Recursive Factorial

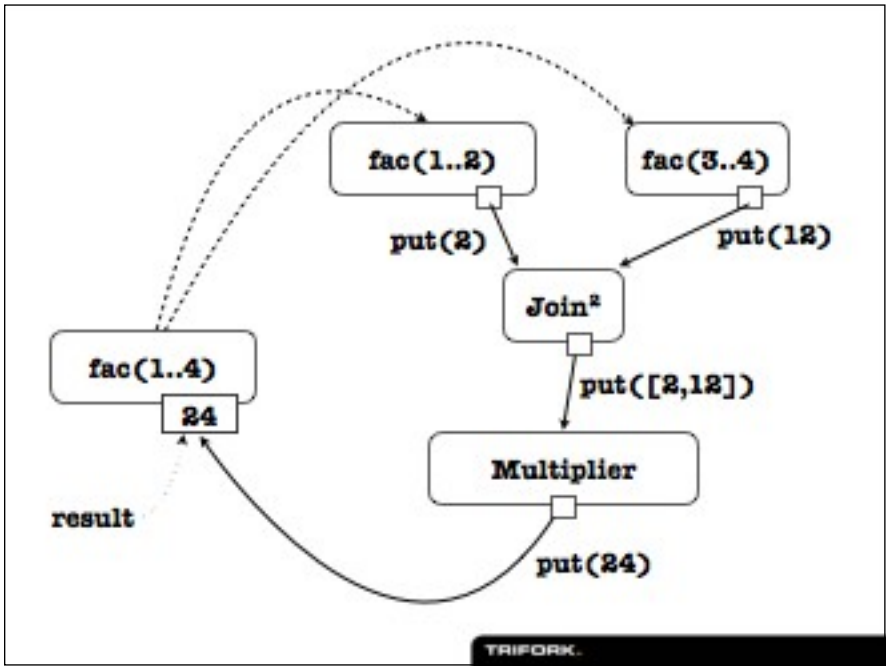
```
interface Factorial {
   ReplySource<Long> compute(long from, long to);
}

// compute factorial(4)

Factorial fac = new FactorialBehavior().actor();

ReplySource<Long> value = fac.compute(1, 4);

System.out.println( value.get() );
```


```
interface Reducer<T> extends
   ReplyPipe <List<T>, T> {
   ...
}

Reducer<Long> multiply() {
   return new ReducerBehavior() {
      Long reduce(Long[] vals) { return vals[0] * vals[1]; }
   }.actor();
}

      Reducer<Long> mul = multiply();
      join(fst, second).sendTo(mul);
      mul.sendTo(sink)
   }
  }
}
```

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

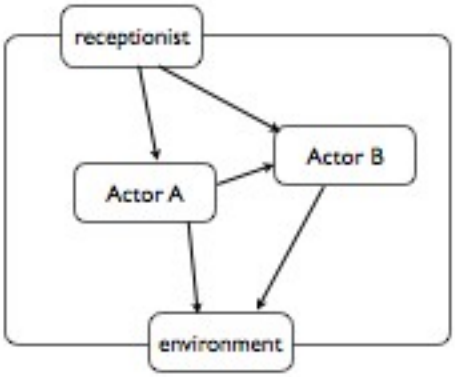
Sharing: Scala makes it easy to write immutable classes/values, but there is no mechanism to guarantee avoiding sharing.

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

- **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.



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.

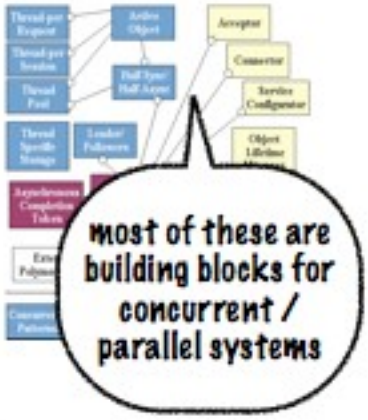
Coordination

- 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
- Many research projects this, e.g. [Frølund96, C

As far as I can see, this is a sore point for actors.

Actor Patterns

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



Act II

Anthropomorphic Programming and the Actra Model

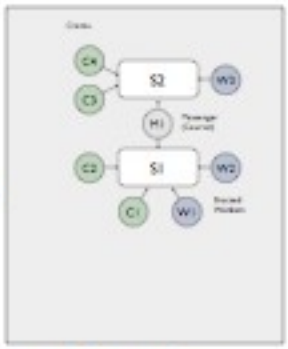
Using Structured Concurrency for Real-Time Embedded Systems, Service Oriented Computing and Agent Base Collaborative Analytics

Message Based OS Kernels – Thoth, QNX, Harmony, V-Kernel…

- Thoth was the archetype (U Waterloo in the late 1970's)
- Many descendants including QNX, V Kernel, Harmony ..
- Unusual messaging primitives force an **anthropomorphic** style
  - Blocking Send
  - Blocking Receive
  - Non Blocking Reply
- Harmony RTOS offered a very robust and stable implementation
  - Portable real-time multi-tasking multi-processing kernel
  - Interrupts unified with messaging model
  - Light weight threads
  - Common interface for local and remote tasks

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.

Actra Actors

First Class Active Objects

Actra adds a new Class called Actor

- Replaces weak Smalltalk co-routine process model
- Light weight tasks which share memory on different processors
- Preserves object message semantics – blocking send, blocking receive
- Method return executes implicit or explicit non-blocking reply

Actra style very natural for Smalltalk programmers

- Actors can be subclasses and collaborate to realize concurrent computation
- Actors an unsuspecting sequential programmer to transform their programs into a structured concurrent program i.e.. Switching active/passive roles didn't affect most of the code

Implementation

MT/MP Smalltalk on VME Bus Shared Memory MP

MT/MP Scavenging GC, Distributed and Persistent GC

MT/MP Services provided by Harmony RTOS

Actor Taxonomy

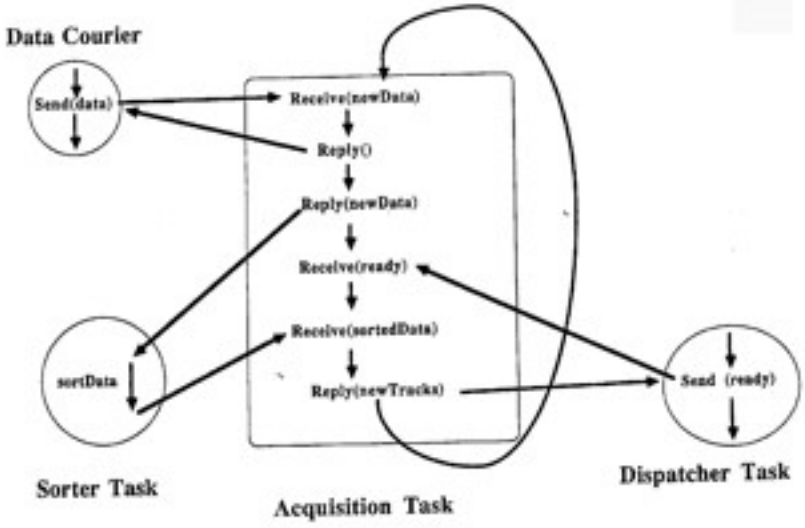
First Class Active Components which encapsulates a set of state and behavior together with a thread of control

Anthropomorphic Programming - Clients, Servers, Agents, Managers, Secretaries, Couriers, Workers, Notifiers …

Generic Actors

- *Worker*: report to managers to perform computation
- *Notifier*: 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 communication typically encapsulating a queue and state machine
- *Business Processes* = Workflow + Rules + Control (e.g., Taylor engines)
- *Agents* = Actor where methods are inference rules
- *Avatar* = Actor where method is script and displayOn: uses OpenGL etc.

AMEP System Architecture



PulseCourier and Acquisition methods

```
initial Task
      super initialTask.
      busInterface := self find: #BusInterface.
      busInterface isNil ifTrue: [
            self error "No BusInterface found by Pulse Courier"].
      busInterface assignBufferFor: self parent.
activity
   "Get data and deliver it to Acquisition"
   | pulses |
    true whileTrue: [
      pulses := busInterface pulsesFor: self parent.
      self parent newData: pulses]
```

```
initialTask
      super initialTask.
      courier := self spawn: PulseCourier.
      dispatcher := self spawn Dispatcher.
      sorter := self spawn: Sorter
trackerDispatcherReady
      trackerDispatcherReady := true.
      newTracks is Empty
            whileTrue: [self receive]. "sender is blocked"
      ^newTracks
newData: trackData
      self buffersFull
            whileTrue: [ self receive]. "sender is blocked"
      dataBuffers add: trackData
```

Actra Summary

Good News

- Works very well on unsuspecting sequential programs
- Good modeling vocabulary/library for structured concurrency
- There are no hidden queues! Cycles are visible by inspection of the message send graph

Bad News

- Sharing state requires every object to have an owner (no longer expensive but used to be)
- Shared Collections beg for immutable collections!
- Write barrier scalability esp. across distributed system
- Multi-core and distributed likely required call by value and == by hash

Thank you

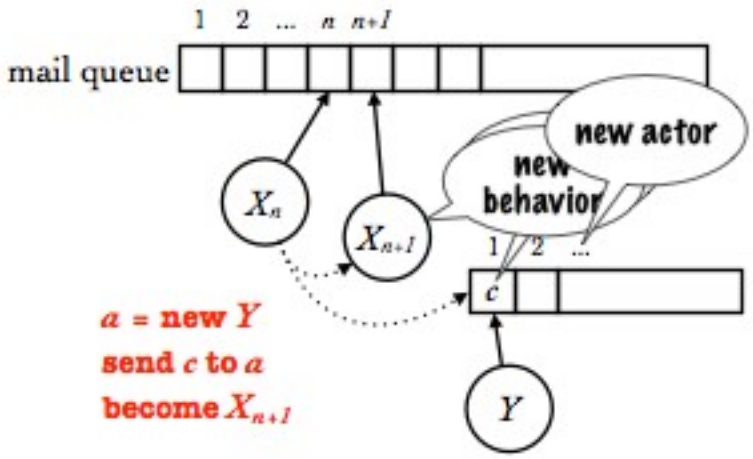
An actor's behavior can

- 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 start when either
  1. The previous behavior completes, or
  2. A replacement behavior is given.

  which ever comes first.



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

```
behavior cell(value)[msg] =
  if msg = ⟨FETCH, client⟩ then
     send value to client
  if msg = ⟨STORE, value2⟩ then
     become cell(value2)

x = new cell(0)
send ⟨STORE, 1⟩ to x
```