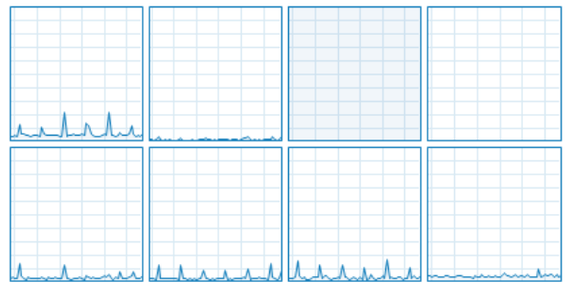
Actor Model

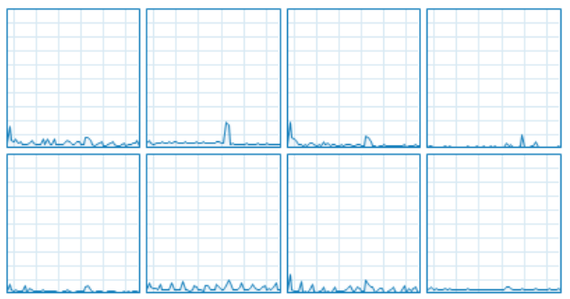
Concurrency that Feels Like Home

Dmitry Ledentsov
https://github.com/d-led

Motivation

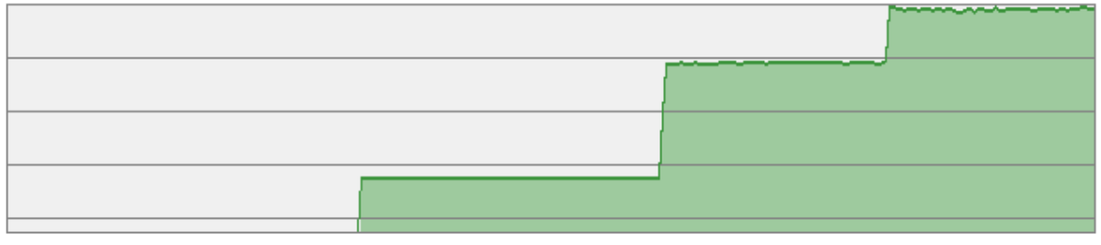
- Multicore hardware is mainstream
- Distributed computing is mainstream
- Parallel programming is hard in sequential languages
  - Race conditions, deadlocks, livelocks, …
  - False sharing, thread contention, thread hogging, …
  - Forgotten synchronization, instruction reordering, …
  - Error handling, debugging, cancellation, scaling out, …?
  - …





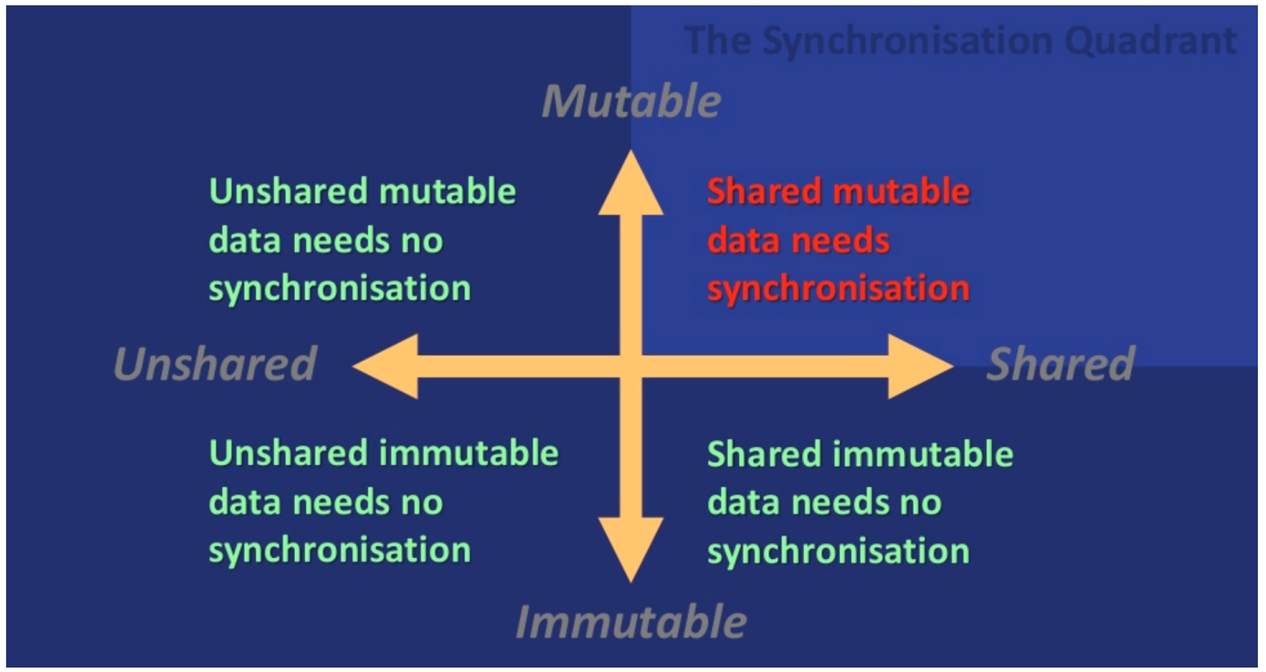
Actor Model to stay sane:

- Simplify concurrency & parallelism
- Unify concurrency & distribution
- Build safety in
- Frameworks not required (at first)



Kevlin Henney:
Thinking Outside
the Synchronisation Quadrant

"All the pain you are feeling is in the top right hand corner.
What are you doing there? Get out of it!"



"… in the other cases that synchronization is doesn't exist or commoditized and hidden from you"
https://youtu.be/yI25p91fILY

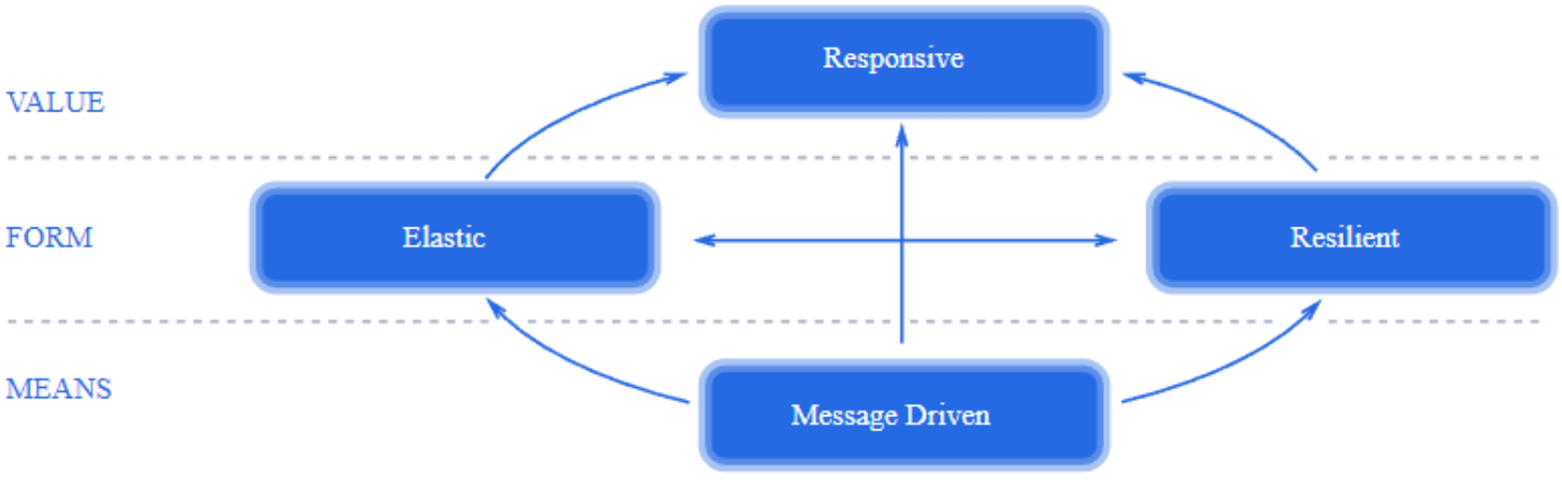
Source: https://www.slideshare.net/Kevlin/thinking-outside-the-synchronisation-quadrant
Used with permission

The Reactive Manifesto

https://www.reactivemanifesto.org/

*… we want systems that are **Responsive**, **Resilient**, **Elastic** and **Message Driven**.*
*We call these **Reactive Systems**.*

- Systems built as Reactive Systems are more **flexible**, **loosely-coupled** and **scalable**.
- This makes them easier to develop and amenable to change.
- They are significantly more **tolerant of failure** and when failure does occur they meet it with elegance rather than disaster.
- Reactive Systems are highly responsive, giving users **effective interactive feedback**.

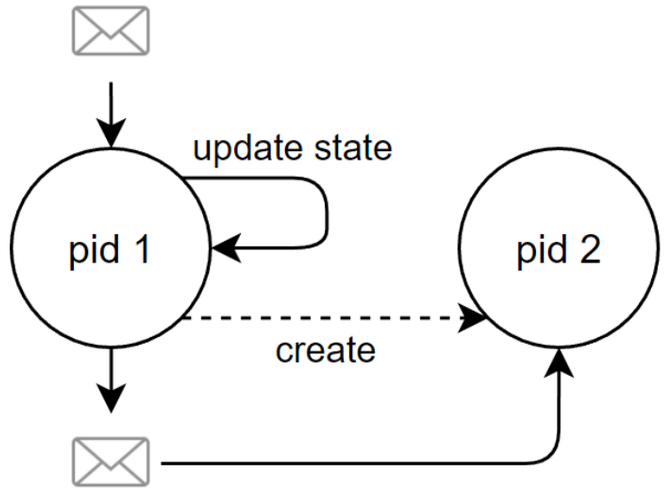


Actor Model: What is it?

"A Universal Modular Actor Formalism for Artificial Intelligence“
by Carl Hewitt, Peter Bishop, Richard Steiger (1973)

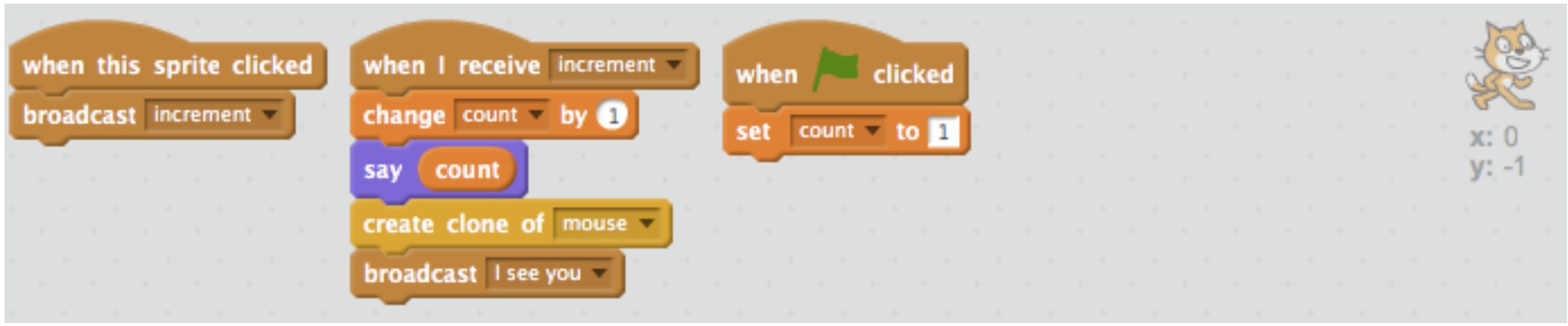
- The **Actor Model** is a mathematical model of concurrent computation that treats “actors" as the universal primitives of concurrent computation

- An Actor:
  - is a computational entity (with identity)
  - In response to a message can:
    - create a finite number of new Actors
    - send a finite number of messages to other Actors
    - choose its next behavior*
  - Messages are processed one-by-one
  - “One actor is no actor. They come in systems.”

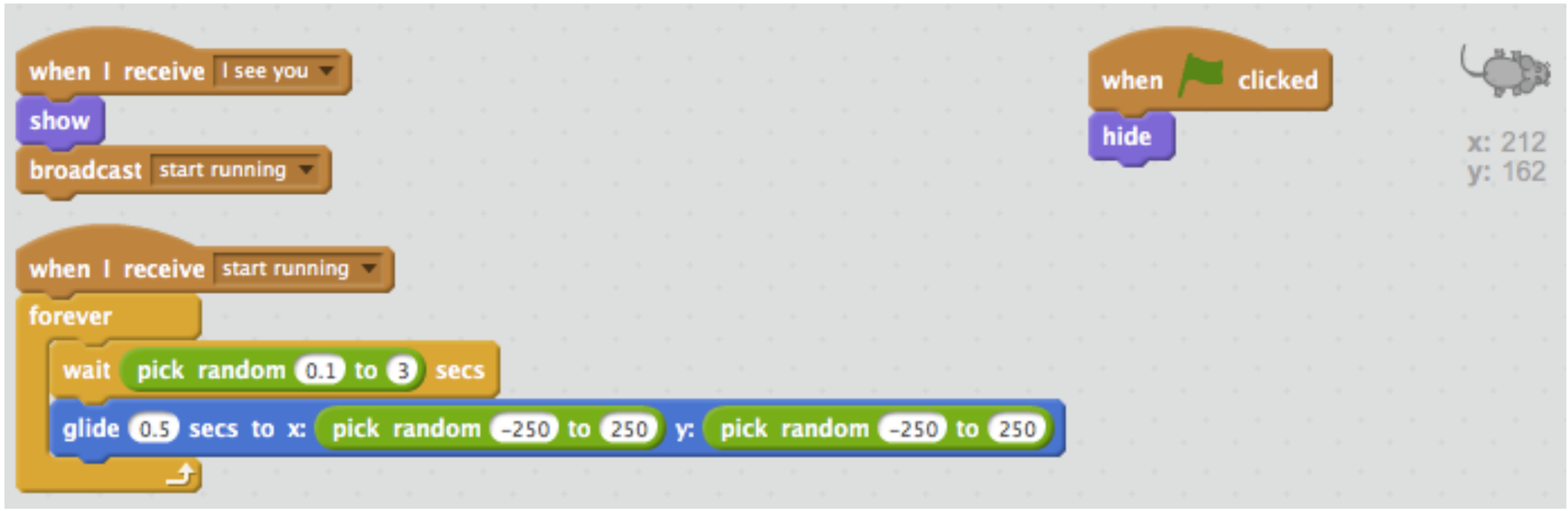


https://arxiv.org/vc/arxiv/papers/1008/1008.1459v8.pdf

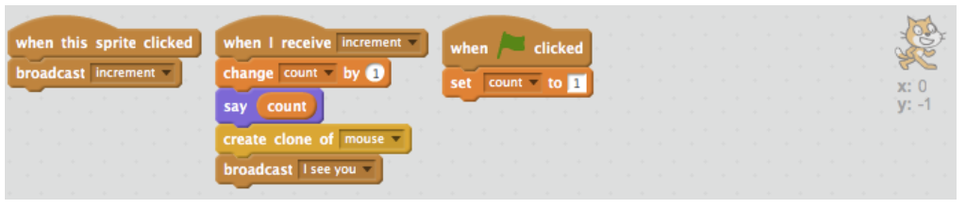
Kids can do it



http://scratch.mit.edu/



Kids can do it



"Being able to program "massively parallel" and easily is important. Be critical of languages that can't do this easily."

- Alan Kay

http://www.vpri.org/pdf/rn2010001_programm.pdf

https://scratch.mit.edu/projects/247963945/#editor

Typical Properties

- Actors don’t share data
- An Actor processes incoming messages synchronously & sequentially
- Messages are immutable
- Actors are scheduled by the run-time
- Actors are lightweight
- Lock-free programming of application code
- Hierarchies of Actors for scaling, isolation and fault-tolerance

https://arxiv.org/vc/arxiv/papers/1008/1008.1459v8.pdf

Object-Oriented Programming

I'm sorry that I long ago coined the term "objects" for this topic
because it gets many people to focus on the lesser idea.
The big idea is “messaging”

The Actor model retained more of what I thought were the good features of the object idea

- Alan Kay on Messaging & Dr. Dobb’s Interview & Email

Object-Oriented Programming

Abstraction

- Send / Receive only the relevant data

Encapsulation

- Communicate via immutable messages
- State is invisible outside the actor

Inheritance

- The receiver actor/object is free to use other actors to inherit behavior

Polymorphism

- Message processing is not tied to concrete implementation
- Structure of acceptable messages defines the object's type

Not tied to any OO-like language!

What programming language is this?

```
thing.jump();
```

… Java, Groovy, Kotlin, Go, C#, C++, Python, Lua*, Ruby, Pony*, …

Is this Actor Model code?

```
thing.jump();
```

when is it going to jump?

is the caller going to wait for it to end?

does it return a value?

when can I use the value?

When?

```
thing.jump();
```

immediately ⟶ class

later ⟶ actor

a set of open source tools
that simplify distributed, concurrent, reactive,
event-driven, and microservices architectures,

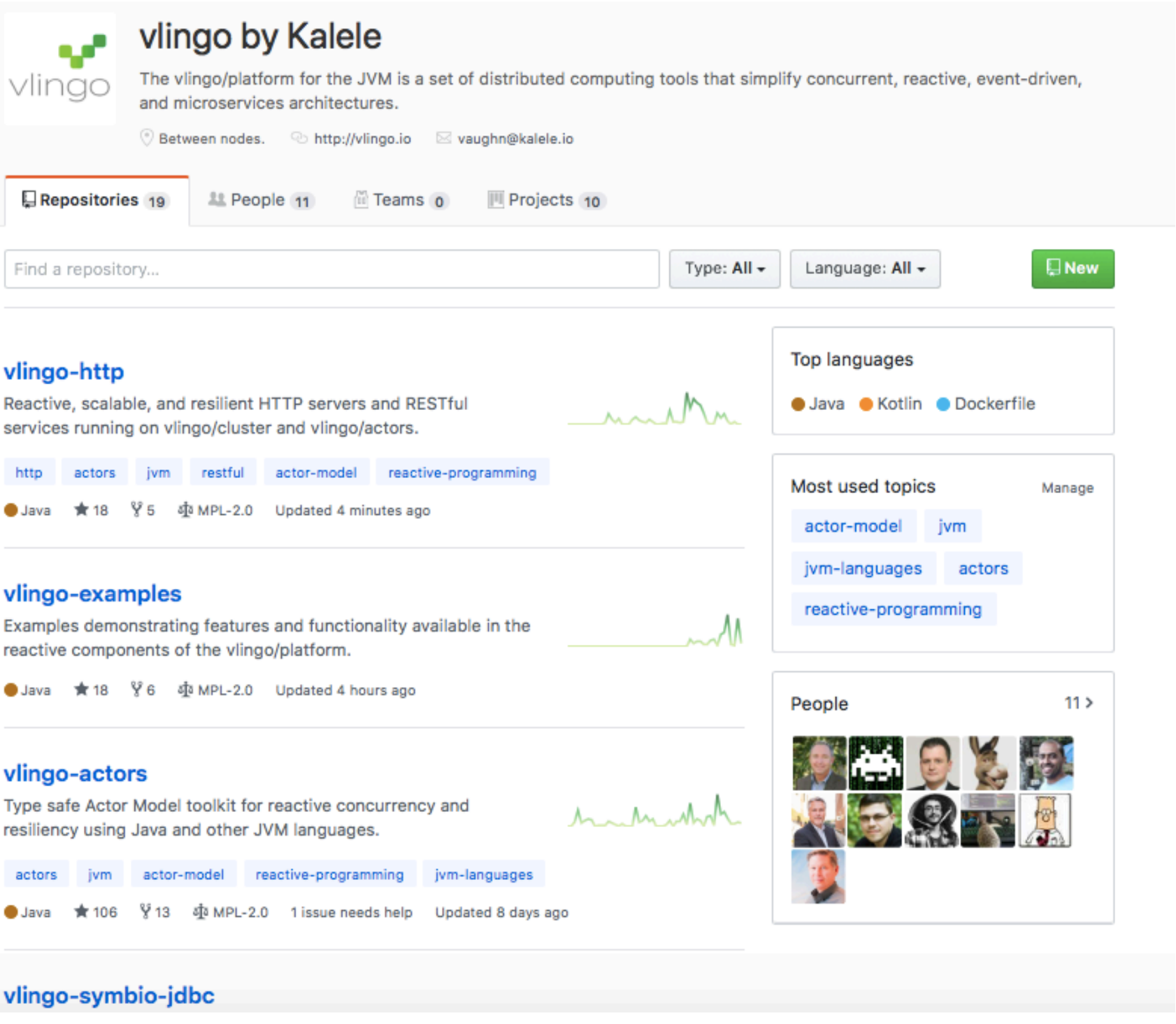
and which facilitate the contemporary ways
that Domain-Driven Design fluent models are expressed.

https://github.com/vlingo/

https://vlingo.io

https://kalele.io

vlingo community



vlingo-actors

```
import io.vlingo.common.Completes;

public interface Thingy {
    void jump();
    Completes<String> howDoYouFeel();
}
```

https://github.com/d-led/hello_vlingo

vlingo-actors

```
public class Thing extends Actor implements Thingy, Scheduled {
    @Override
    public void jump() {
        System.out.println("preparing to jump  ... ");
        this.scheduler().scheduleOnce(…);
    }

    @Override
    public Completes<String> howDoYouFeel() {
        return completes().with("everything's fine");
    }

    @Override
    public void intervalSignal(……) {
        System.out.println("jumping now  ... ");
    }
}
```

vlingo-actors

```
final World world = World.start("playground");

final Thingy thing = world
    .actorFor(
        Definition.has(Thing.class, Definition.NoParameters),
        Thingy.class
    );

thing.jump();

System.out.println("we continue…");

thing.howDoYouFeel().andThenConsume(reply → {
    System.out.println("it replied: "+reply);
});

System.out.println("the thing will answer shortly");
```

vlingo-actors

```
thing.jump();
System.out.println("we continue…");

Completes<String> answer = thing.howDoYouFeel();
answer.andThenConsume(reply → {
    System.out.println("it replied: "+reply); return reply;
});
System.out.println("the thing will answer shortly");
```

```
vlingo/actors: Generating proxy for main:
                                        github.dled.demo.Thingy
we continue …
preparing to jump  …
jumping now  …
the thing will answer shortly
it replied: everything's fine
```

Isn’t messaging slow?

https://travis-ci.org/d-led/hello_vlingo/builds/450299134

```
greeter.startBenchmark();

for (long i = 0; i < N; i++)
    greeter.tick();

greeter.reportProgress();
```

100,000,000 times ConsoleGreeter increment: 7.614s (13,133,000 times/s)
100,000,000 times simple increment: 0.771s (129,701,000 times/s)

Interfacing with the world: Reactive Extensions / Streams

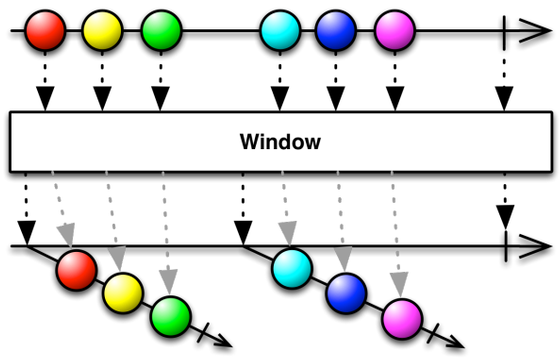
- Focusing on data flow and structural/temporal transformation
- Reactive Streams add back-pressure protocols
- Explicit scheduler abstraction
- Simplifies concurrency testing via a deterministic test scheduler
- Mindset switch from imperative to functional transformation

http://www.introtorx.com

http://reactivex.io

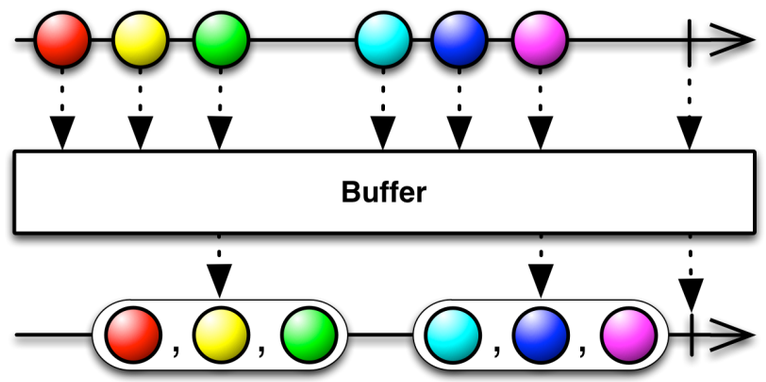
http://www.reactive-streams.org

https://www.github.com/ReactiveX/RxJava



vlingo + Kotlin + RxKotlin

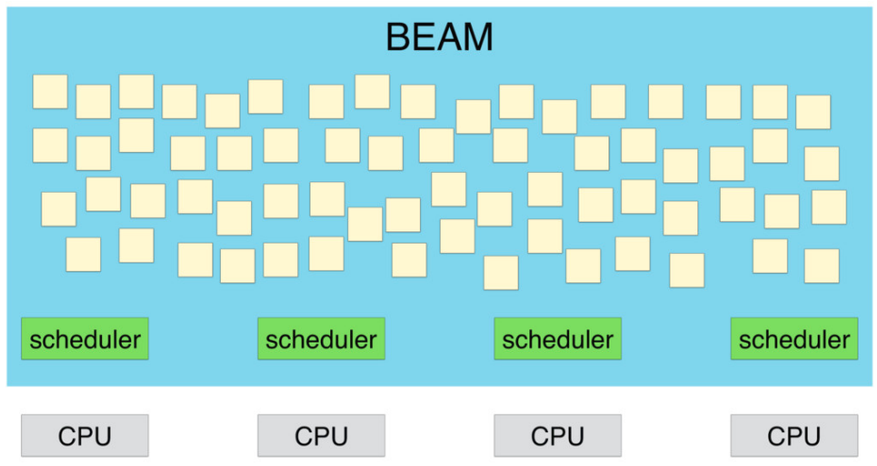
```
class RxBatcher : Actor, Batcher {
…

    constructor(processor: Processor) : super() {
        this.processor = processor

        val window = subject
                .buffer(1, TimeUnit.SECONDS, 10)
                .filter { l → l.isNotEmpty() }

        window
                .subscribe({l →
                    processor.process(l)
                },{e→println("interrupted: ${e.localizedMessage}")})
                .addTo(subscriptions)
    }

    override fun batch(item: Item) {
        subject.onNext(item)
    }
…
}
```

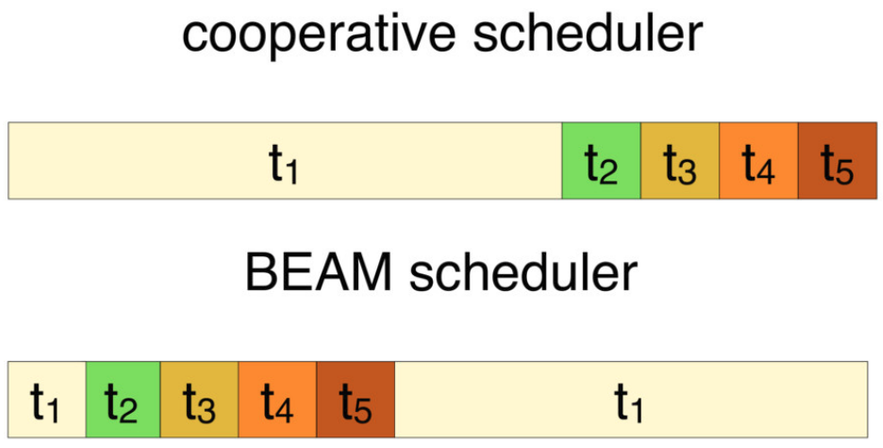


https://github.com/d-led/vlingo_experiments/tree/master/batching_with_rx

vlingo + why Erlang/Elixir?

- Distribute CPU time fairly between requests
- Robust run-time & platform
- Keeps good quality of service for all requests
  → frequent context switches*
- Garbarge collection per process (actor)





- Tight loops are not possible in Erlang
- Each actor gets a budget of function calls
- Native code can block → out of process / dirty schedulers
- Isolation: crash of one request will not crash the VM/other processes
- Glue other processes: native, java, etc. together

* Source: Solid Ground by Saša Jurić https://youtu.be/pO4_Wlq8JeI, Slides
Used with permission

vlingo + Elixir

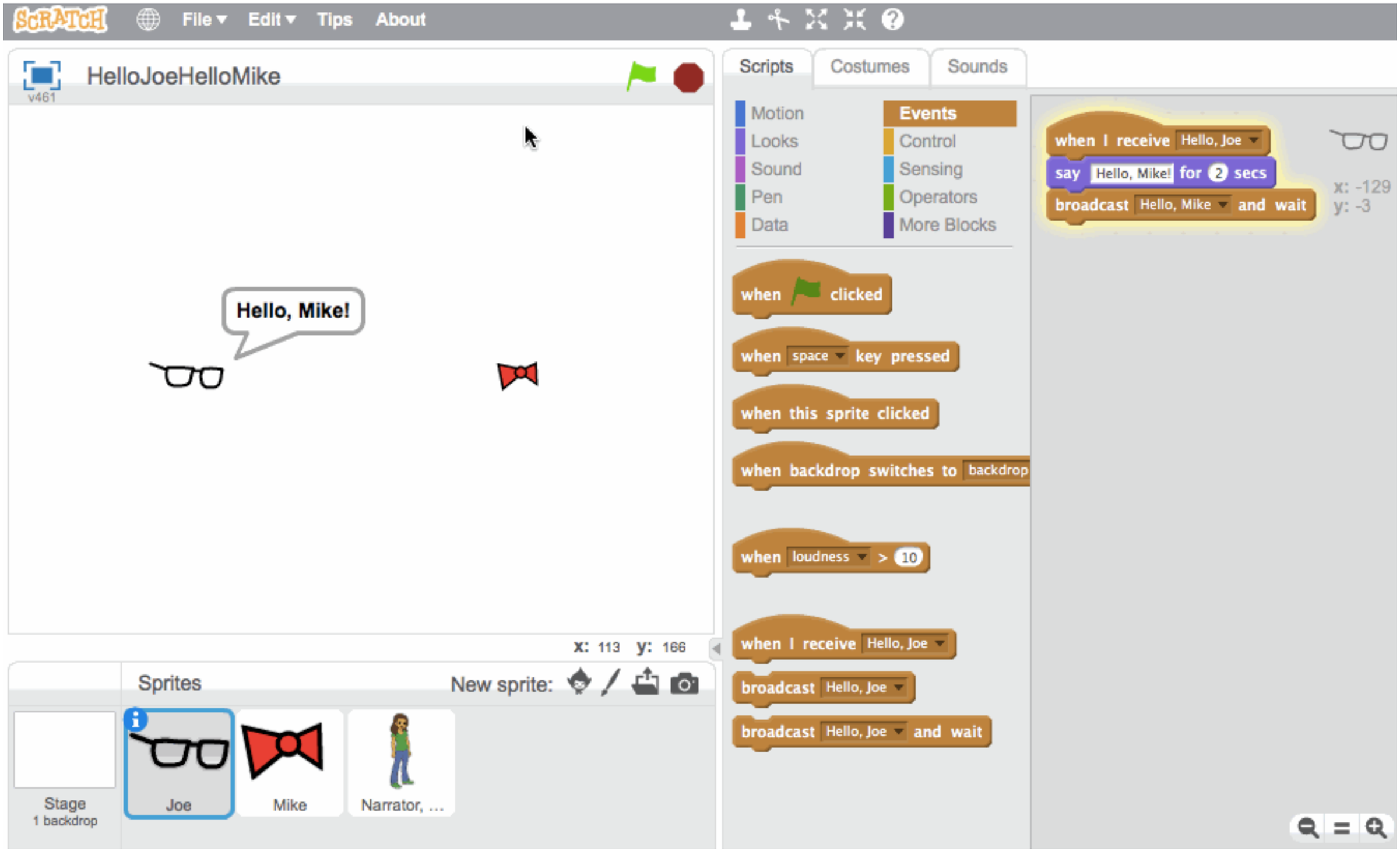
```
# https://www.youtube.com/watch?v=xrIjfIjssLE
defmodule Mike do
…
  defp try_talk_to(…) do
    send(
         {:joe,:joe@localhost},
         {self(),"Hello, Joe!”}
    )
    receive do
      m →
          IO.puts "Mike: received: #{m}"
    end
  end
end
Mike.call_joe()
```

```
public class Joe
       extends Actor
       implements Listener {
  final Mailbox mailbox;
…
  @Override
  public Completes<Stoppable> waitForCall() {
    ErlangTerm payload = mailbox.receive()…
…
     System.out.format("Joe: received ‘%s’…\n”…);
      mailbox.send(from, string("Hello, Mike!"));
  }
}
```

```
Mike: beep
Mike: beep
Mike: received: Hello, Mike!
Joe: received 'Hello, Joe!' from #PID<3.96.0>
```

https://github.com/d-led/vlingo_experiments/tree/master/encon_vlingo https://github.com/appulse-projects/encon-java

You know it already



https://scratch.mit.edu/projects/247963945/

vlingo-platform

https://github.com/vlingo

vlingo-examples
Examples demonstrating features and functionality available in the reactive components of the vlingo/platform.

vlingo-http
Reactive, scalable, and resilient HTTP servers and RESTful services running on vlingo/cluster and vlingo/actors.

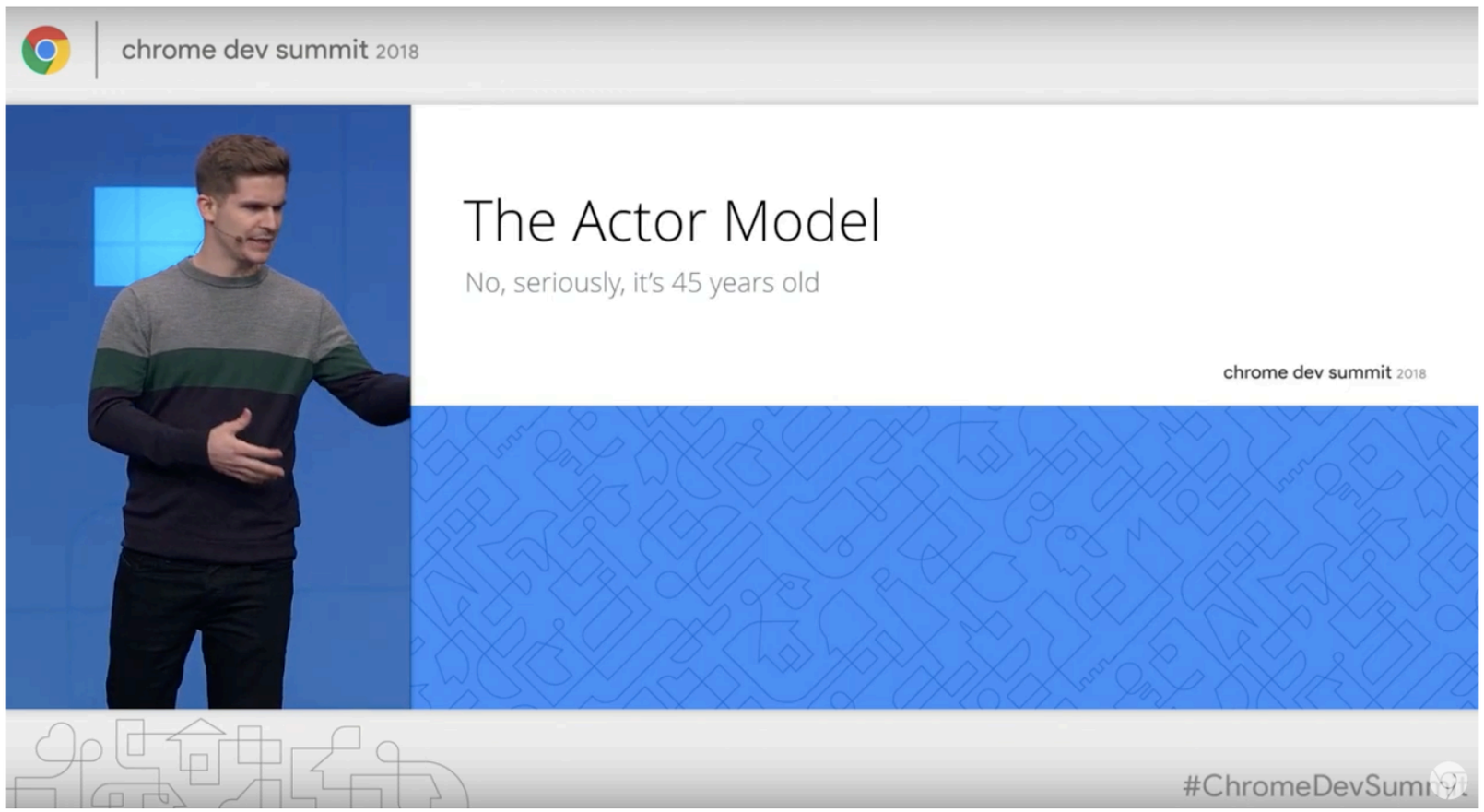
vlingo-symbio
The reactive, scalable, and resilient CQRS storage and projection tool for services and applications built on the vlingo/platform.

vlingo-lattice
Tooling for reactive Domain-Driven Design projects that are highly concurrent. Includes compute grid, actor caching, spaces, cross-node cluster messaging, CQRS, and Event Sourcing support.

and more under heavy development …

It’s coming
to the browser too

Architecting Web Apps - Lights, Camera, Action!
(Chrome Dev Summit 2018)

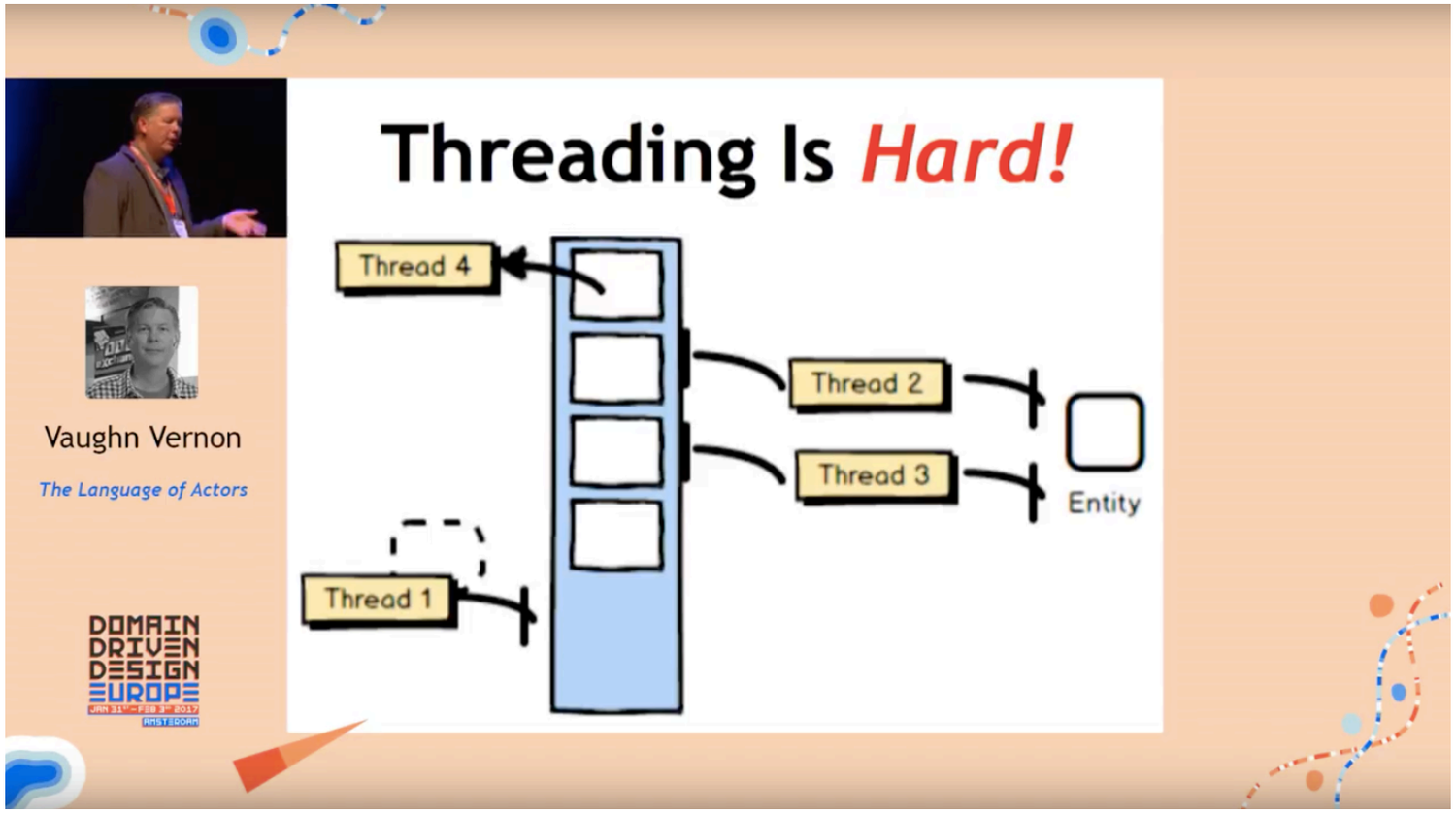


https://youtu.be/Vg60lf92EkM

Motivation in Depth

Wednesday, March 6, 2019: Vaughn Vernon
at the Microservices Meetup Munich

The Language of Actors - Vaughn Vernon
https://youtu.be/KtRLIzG5c54



GOTO 2017 • DDD Today - Modeling Uncertainty • Vaughn Vernon
https://youtu.be/8Y-XPIXOWoA

More Examples

https://github.com/vlingo/vlingo-examples

https://github.com/kmruiz/vlingo-bank-example

https://github.com/daferpi/dfp-kotlin-vlingo-actors

https://github.com/d-led/vlingo_experiments/tree/master/vlingo_0mq_python

Thank you!

Questions?

Actor Model Ecosystem

ERLANG

akka

vlingo

elixir

PONY

C++ Actor Framework

Orleans

riak

proto actor

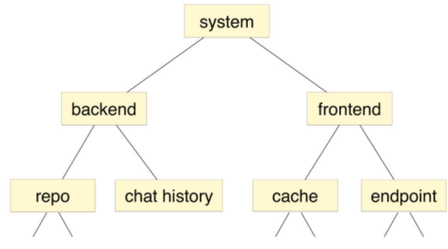
RabbitMQ

CouchDB

Phoenix Framework

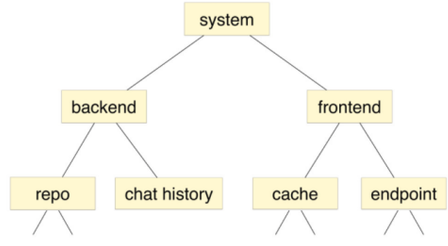
ejabberd

How to Design Actor Systems



- Two separate concerns
  - how to use the computer resources efficiently
  - How to obtain correct results
- Model each truly concurrent activity as one actor
  - 1 subsystem /1 connection / 1 connection-less request / 1 worker
- Subdivide the problem hierarchically using supervisor trees
- Strive to a unidirectional flow
- Separate side effects (send/receive/etc.) from well-behaved functions (referentially transparent)
- Sequential application code can be verified formally, without the concurrency
- Reuse patterns (behaviors) for concurrency, fault tolerance, state machines, etc…
- Model transactions as messages
  - message processing either succeeds or not
  - state is not changed/corrupted on failed processing
- Use actors as concurrency orchestrator to external sequential code
- Practice

How to Design Actor Systems



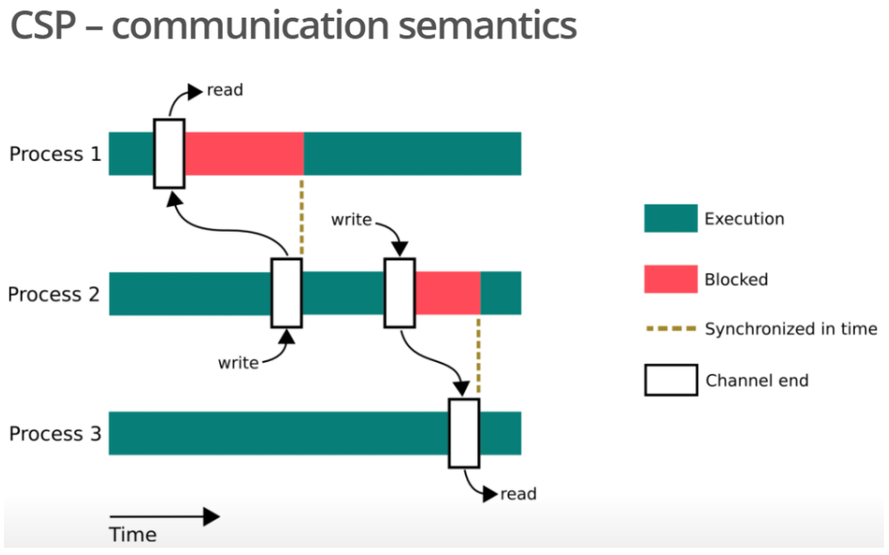
- First approximation: Active Object
  - Threads are expensive: limited number of objects/actors
- Use message passing between threads (lightweight or not)
- e.g. use ZeroMQ (jeromq) to communicate between threads, processes or machines
  - Unified interface again: socket-like send/receive
  - No shared data → no synchronization primitives needed
  - Use *attached* threads for concurrency structure
    - Connect the new threads to the parent threads
  - Use *detached* threads for independent background tasks
  - Switching from in-process to inter-process to tcp is a configuration
- Problems that remain to be solved: granularity, scheduling, thread hogging / DoS, error handling

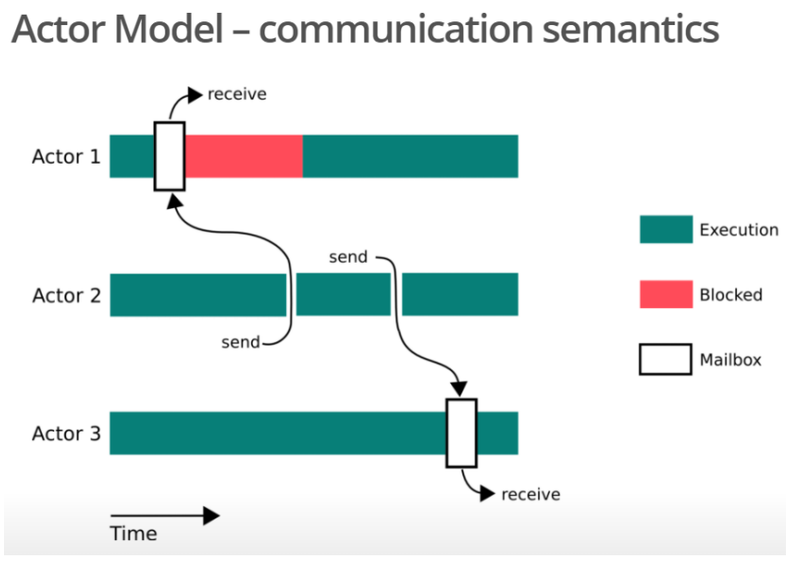
http://zeromq.org/blog:multithreading-magic

http://zguide.zeromq.org/page:all#Multithreading-with-ZeroMQ

Communicating Sequential Processes

- Communicating Sequential Processes (CSP, Hoare 1978)
  - Channels have identity, but not the processes
  - Current implementations
    - Go: shared mutable state is still possible, programmers resort to explicit synchronization
    - Clojure
  - Degrades into blocking send
    - Partially mitigated by buffering





Source: Arild Nilsen https://arild.github.io/csp-presentation. Used with permission

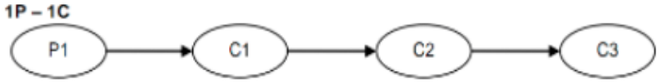
LMAX Architecture

https://lmax-exchange.github.io/disruptor/

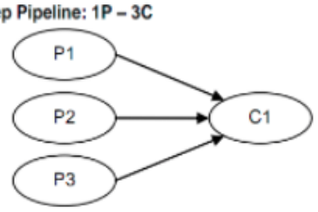
https://github.com/LMAX-Exchange/disruptor/wiki/Performance-Results

https://www.slideshare.net/trishagee/introduction-to-the-disruptor

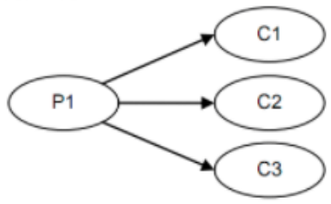
Unicast: 1P – 1C



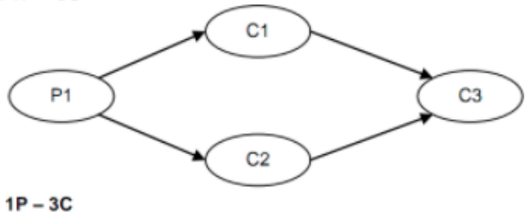
Three Step Pipeline: 1P – 3C



Sequencer: 3P – 1C



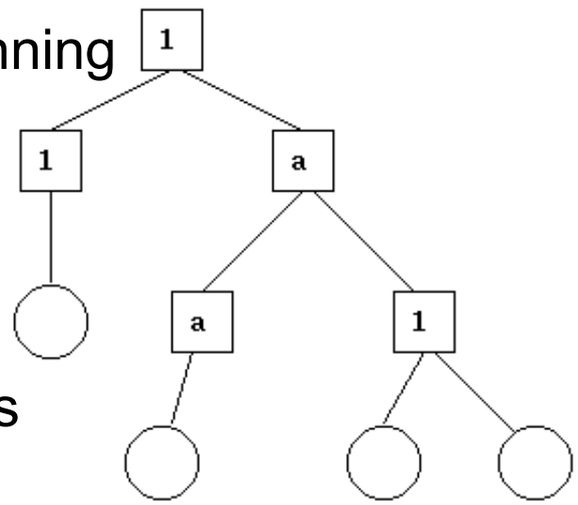
Multicast: 1P – 3C



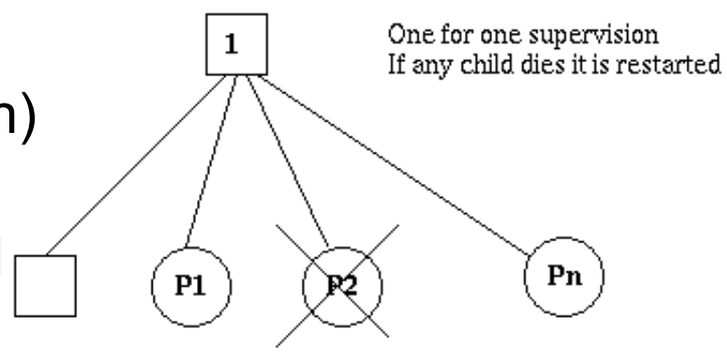
Diamond: 1P – 3C

Actor Model, the Erlang Way

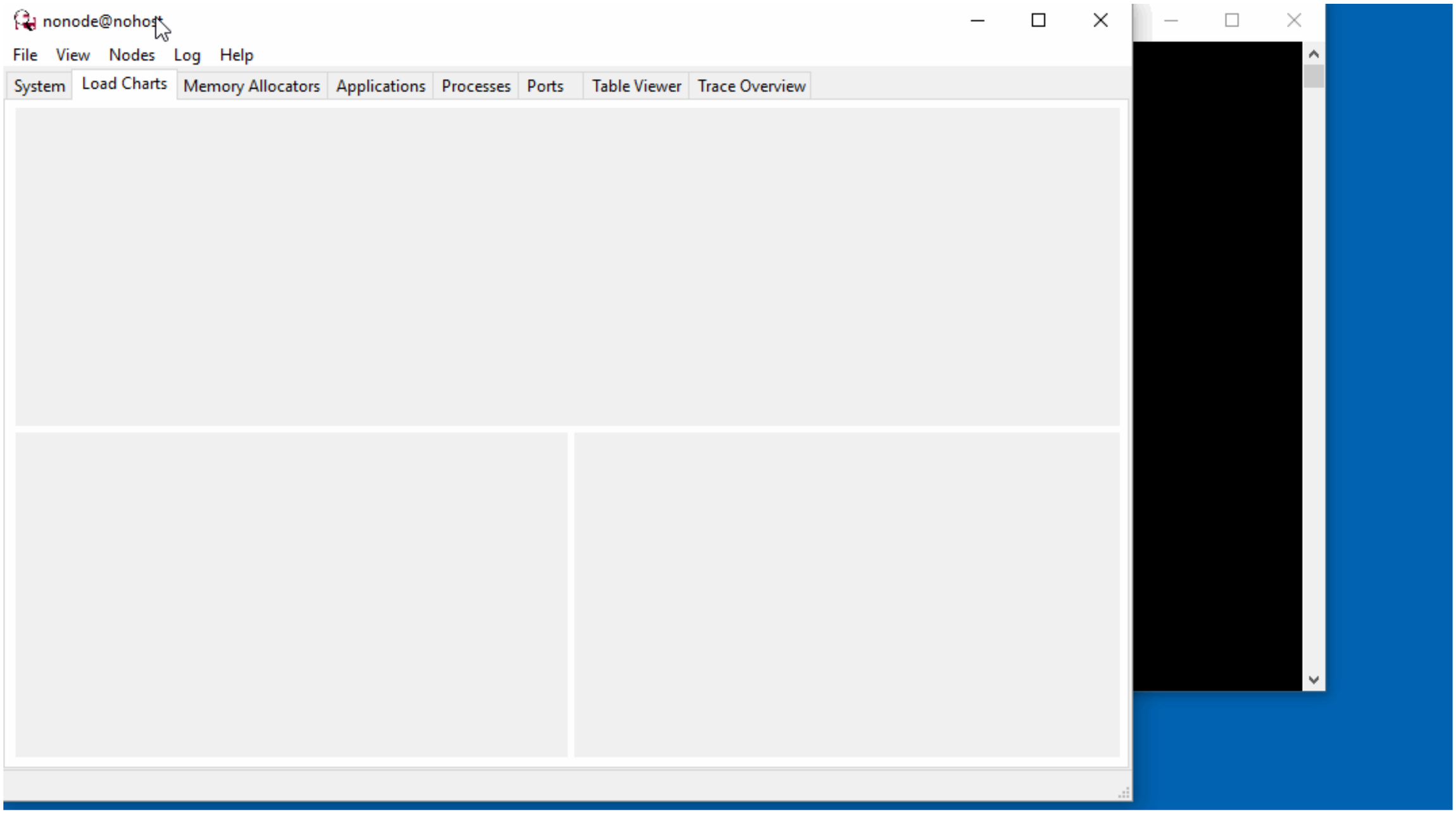
- Concurrency-Oriented Programming Language from the beginning
- Fail fast / Let it crash (safely and by design, not by accident!)
- Data is immutable, no shared data
- Pattern matching (even of binary data)
- Functional programming
- Hot code reload for zero downtime maintenance and upgrades
- Distribution built in



- Debuggable without code change
- Rich batteries included (OTP): tracing, debugging, state machines, DBs, etc.
- Battle-hardened reusable patterns (e.g. process supervision)
- Bytecode compiled
- Efficient cross-platform VM implementation(s). Main: BEAM
- No stop-the-world garbage collection (concurrent/per actor)
- Message ordering preserved between two actors (causal)
- …



Preemptive Scheduling Example



https://gist.github.com/d-led/da1bd5ffbad868d029569db3bbf67534