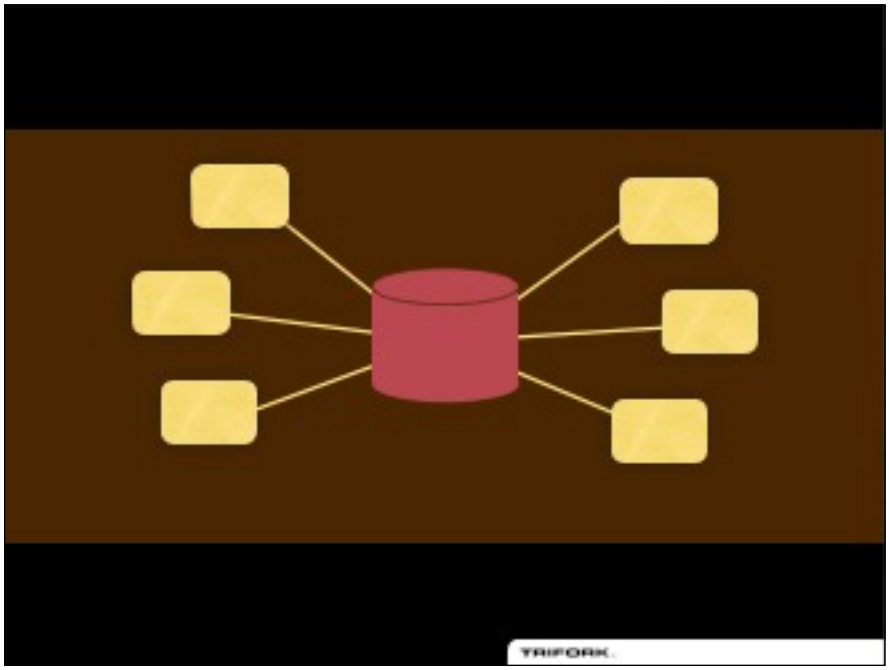
What I learned building Erjang — a JVM-based Erlang VM Or: An object-head in Erlang Land

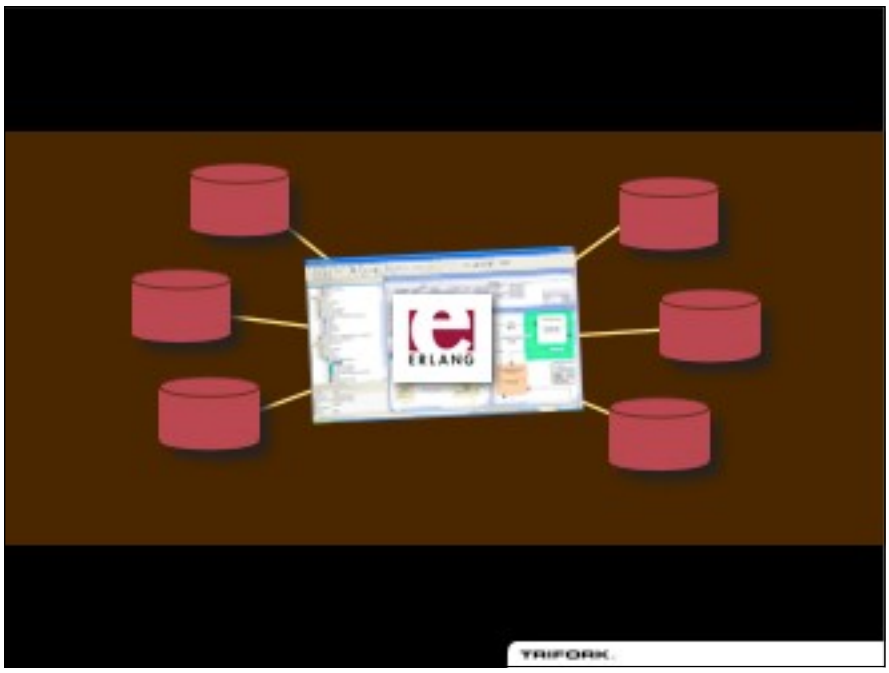
@drkrab

Kresten Krab Thorup
Trifork CTO

software pilots
TRIFORK.

The Desert of Java



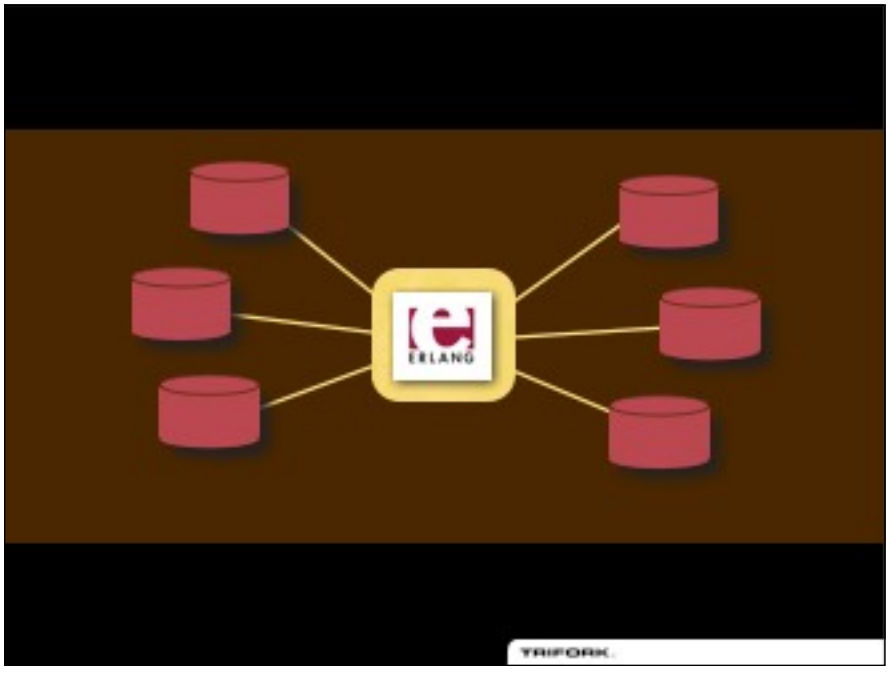


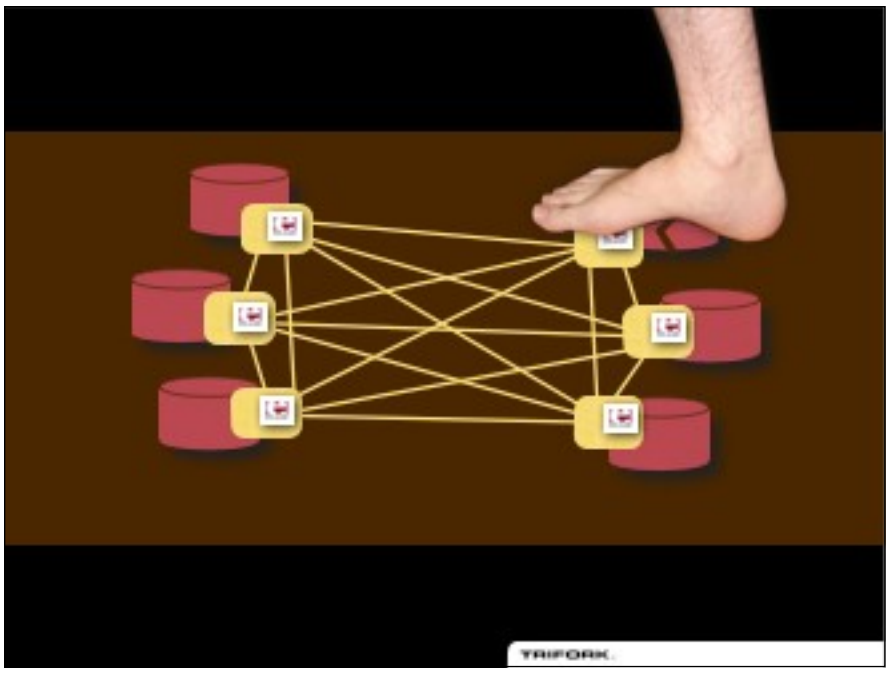
Erlang's sweet spot

Express coordination logic

⇒ Async events

⇒ Cheap processes + blocking msgs



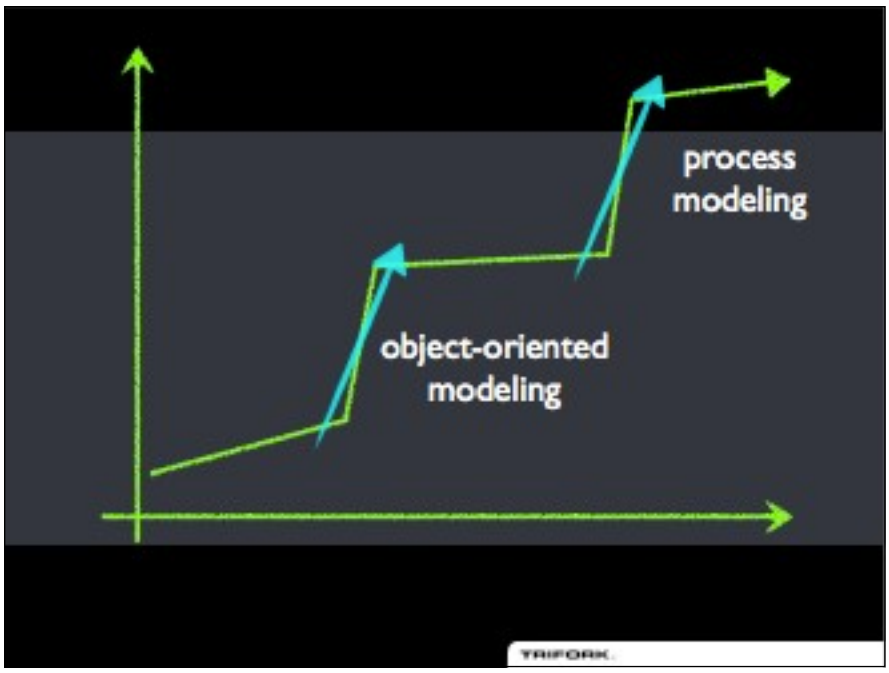


Erlang's raison d'être

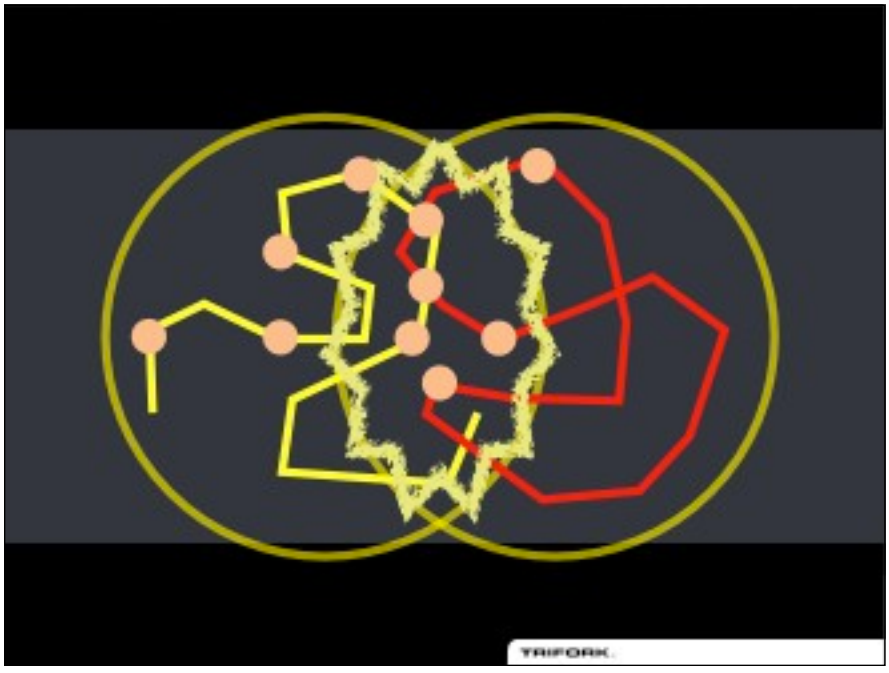
Build reliable systems
in the presence of errors

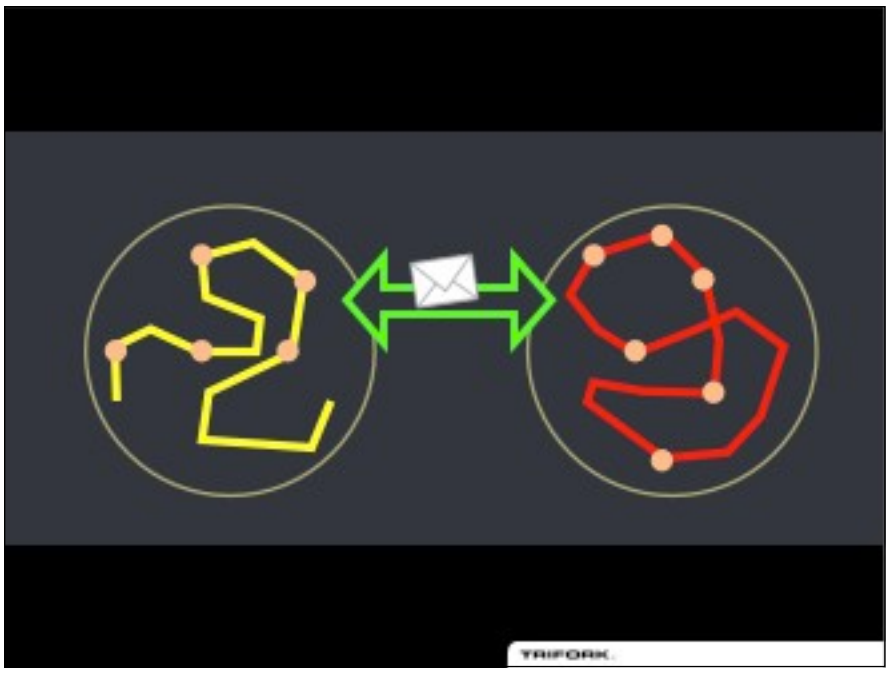
⇒ Isolation + Concurrency

process modeling

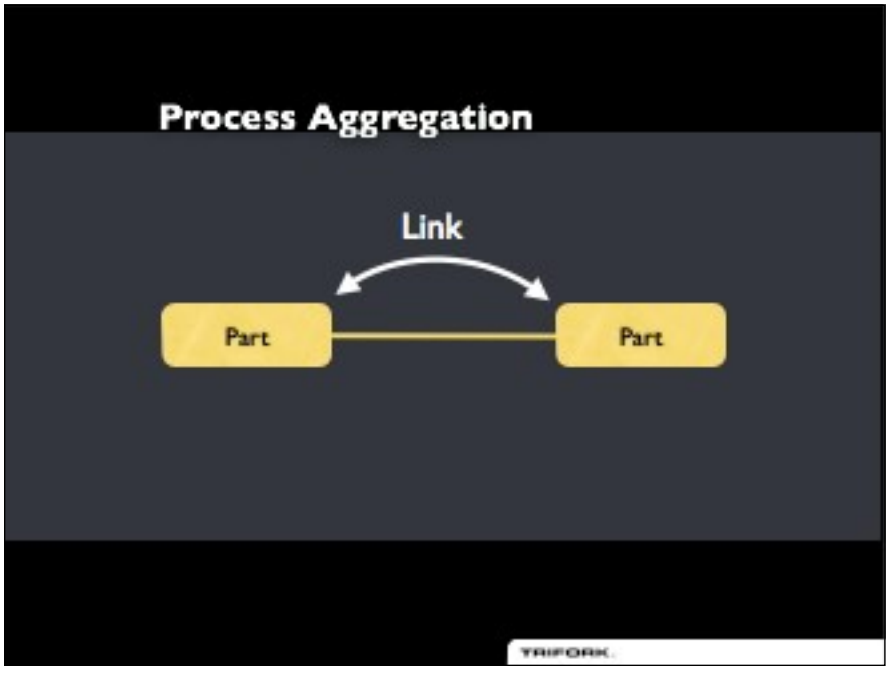

process
modeling
object-oriented
modeling


1UP

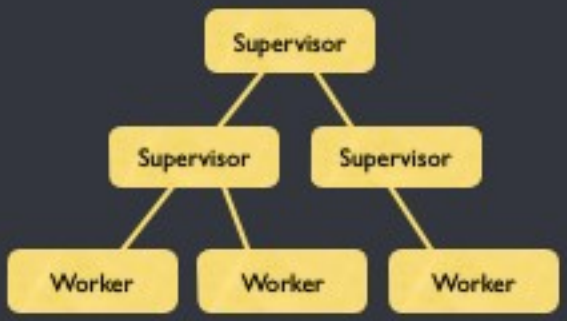



TRIFORK


TRIFORK

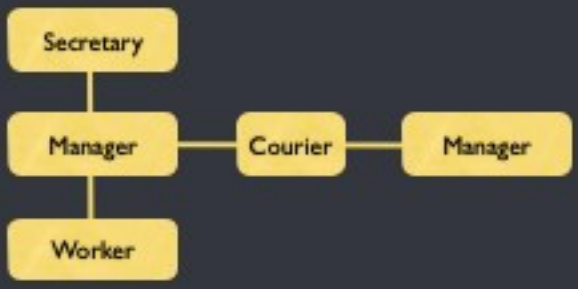

Process Aggregation
Link
Part
Part
TRIFORK

Hierachical Organizations



TRIFORK

Anthropomorphic Programming



Morven Gentleman, SP&E 1981; Thomas & Barry, JOT 2004.

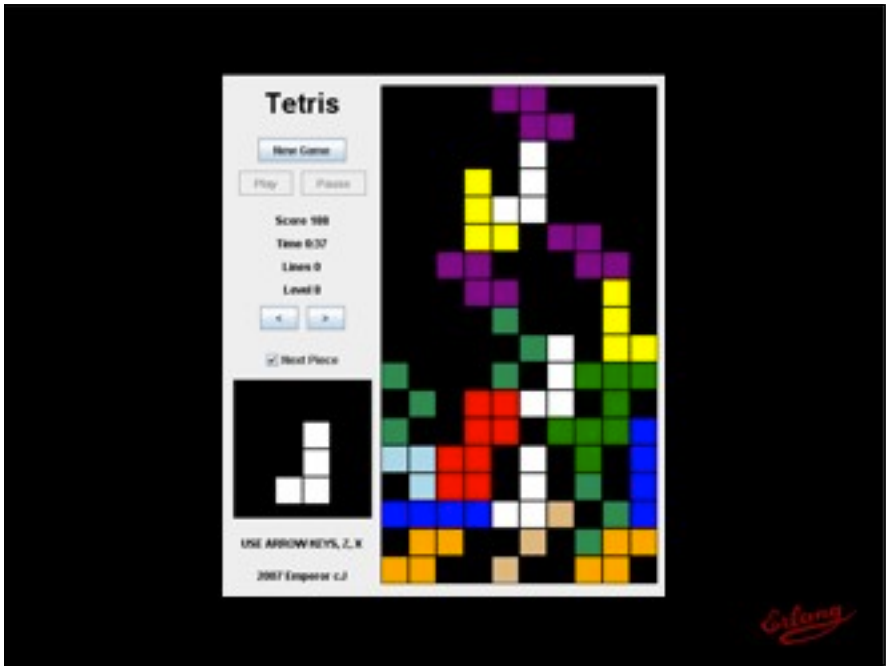
TRIFORK

Sanity check

- When assessing a concurrency pattern in software, try to imagine what it would correspond to in real-life, enacted by humans

[Next 5 slides credit to Ulf Wiger]


SCORE
6629
LEVEL
3
LINES
33


Tetris
New Game
Play
Pause
Score 988
Time 0:37
Lines 0
Level 0
Next Piece
USE ARROW KEYS, Z, X
2007 Emperor CJ

Event Handling Strategies

- Twist and place the next piece - before it lands
- In cheat mode, you get to peek at the next one
- Otherwise, hope for the best

- Search for a specific piece
- Put away pieces that don't fit
- Keep at it until fitting piece found

Erlang

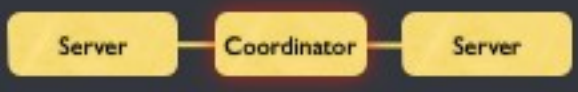
Event Handling in Software

- FIFO, run-to-completion event handling
- Not allowed to block
- Fine, as long as the pieces fit...

- Blocking, selective receive
- Wait until the next *desired* piece arrives
- Buffer unknown pieces

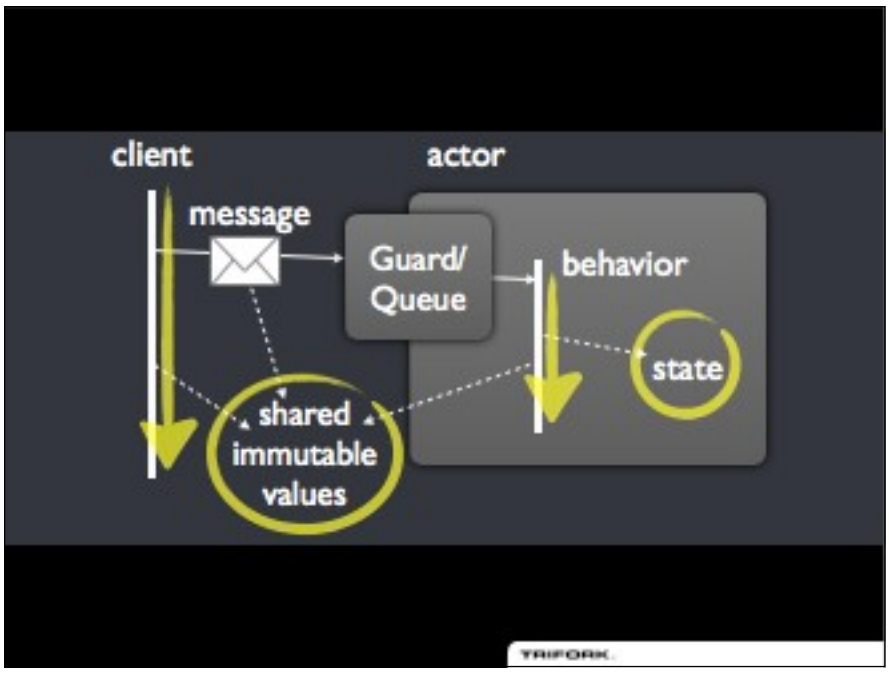
Erlang

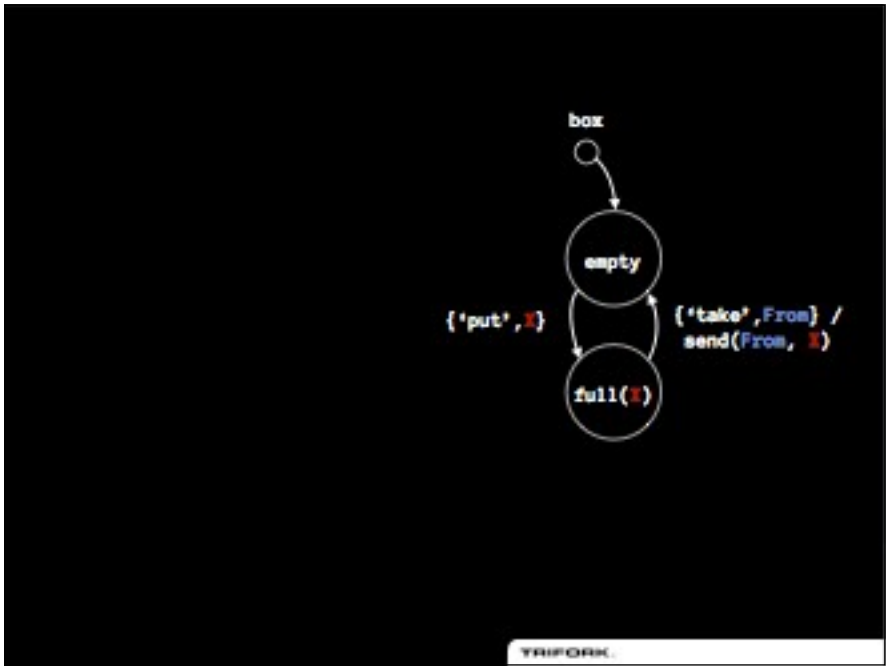
Anthropomorphic Programming



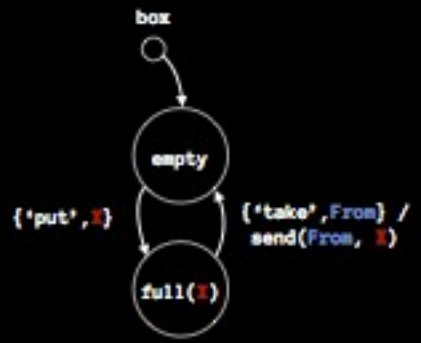
Cheap processes + Blocking Message

TRIFORK

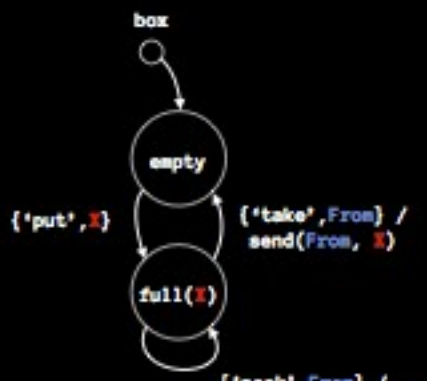





```
box() → empty().

empty() →
  receive
   {'put', X} →
        full(X)
  end.

full(X) →
  receive
   {'take', From} →
        From ! X,
        empty()
  end.
```

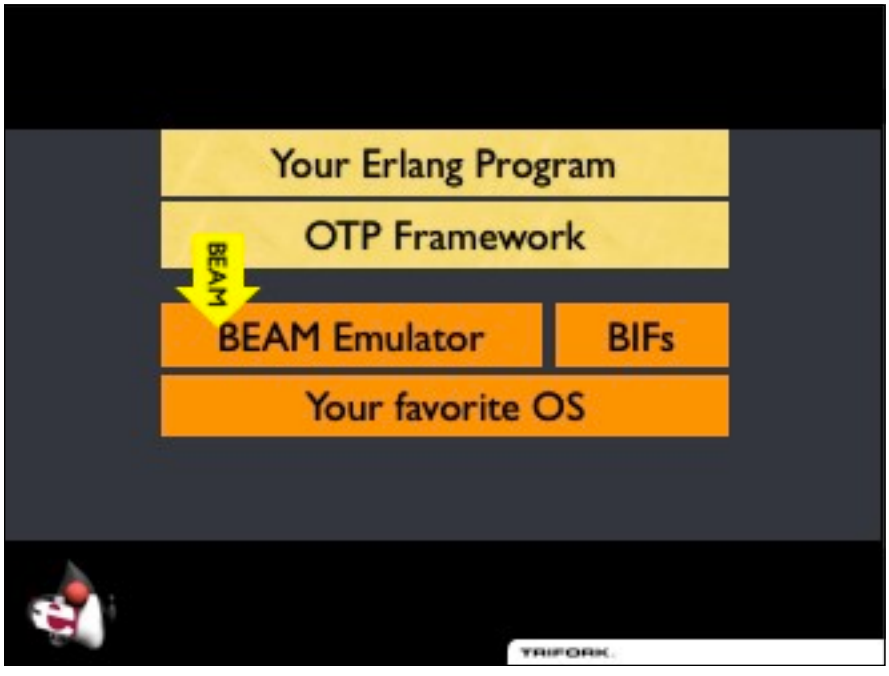


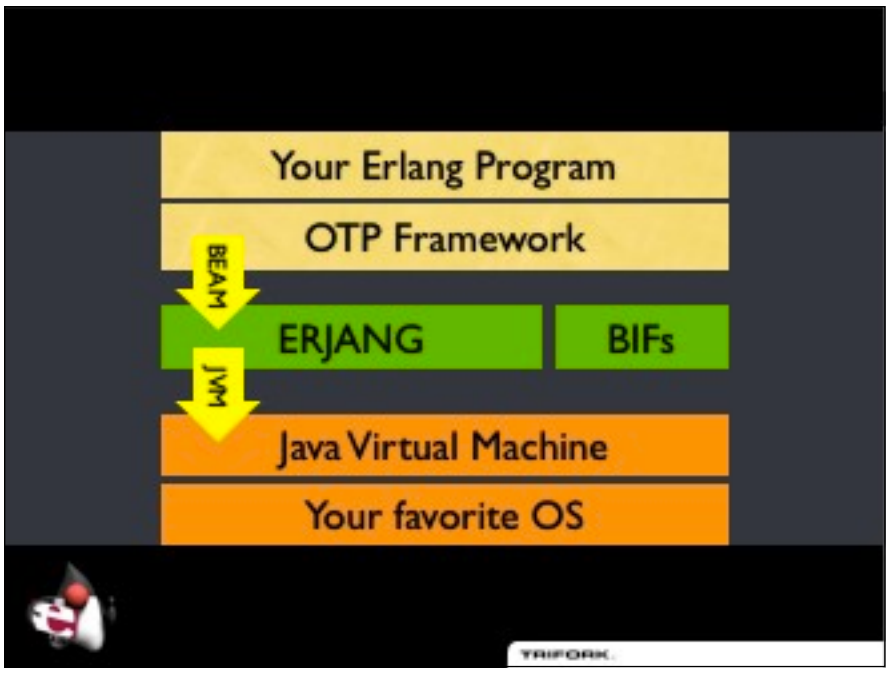
```
box() → empty().

empty() →
  receive
   {'put', X} →
        full(X)
  end.

full(X) →
  receive
   {'take', From} →
        From ! X,
        empty();
   {'peek', From} →
        From ! {'ok', X},
        full(X)
  end.
```



KEY INSIGHTS, Language Perspective

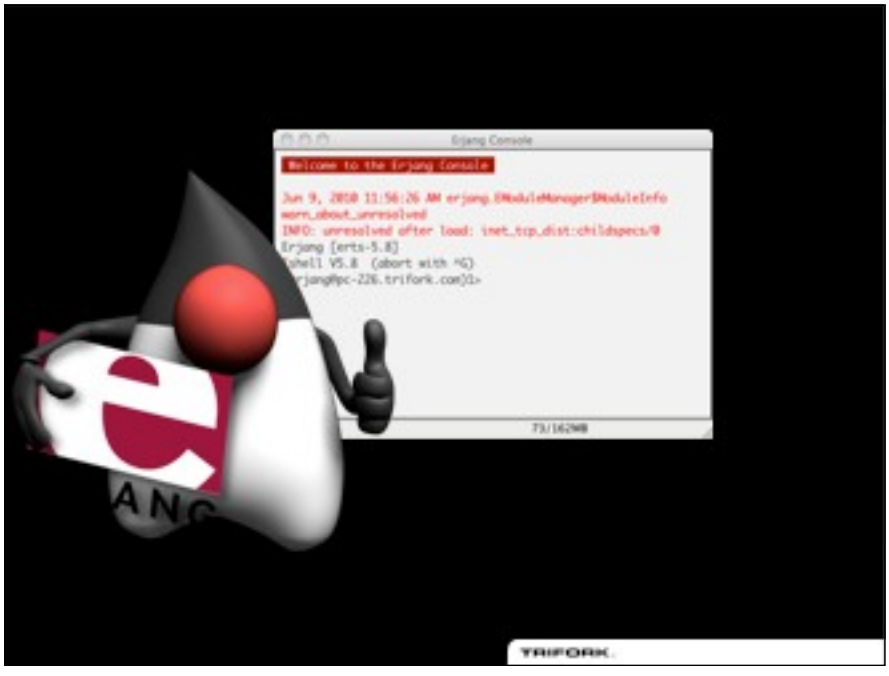
- Proper memory encapsulation is a big improvement; similar to introducing GC
- Cheap processes + Selective (blocking) Message Receive is IMHO a superior alternative to event callbacks





Why Java?

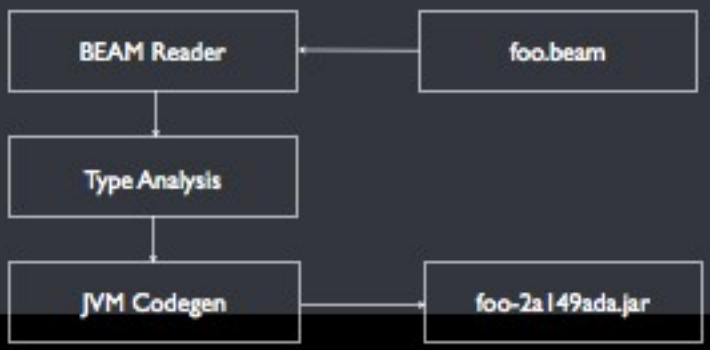
- JVM is everywhere (from mobile to AS/400)
- JVM has many libraries / integrations.
- JVM has 500+ man-years of engineering
- JVM is fast (for Java-ish programs).
- ... and I know JVM pretty well, ...



Erjang - Challenges

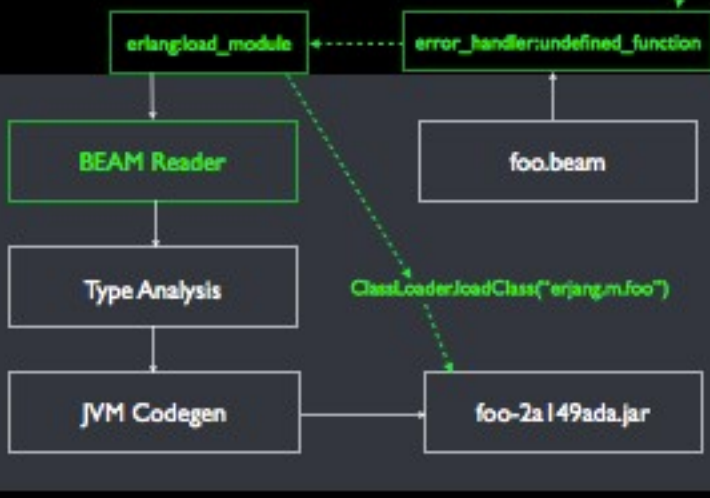
- Ultra Light-Weight Processes
- ~~Real-time Behavior~~
- Tail-recursion
- Arbitrary Precision Numbers
- Pattern Matching
- Erlang Drivers
- JVM is safe, urgh!

Runtime Compiler



TRIFORK

Runtime Compiler



TRIFORK

Language Concepts

Erlang	Erjang
Process + Messaging	Coroutine + Mailbox [Kilim]
Tail Calls	Trampoline Encoding
State Encapsulation	Immutable / Persistent Data

Tail Calls

```
-module(bar).

bat([H | T], T2) ->
  bat(T, foo(H, T2));

bat([], T2) -> T2.

foo(H, T) ->
  lists:reverse(H ++ T).
```

The BEAM Code

```
{function, bat, {nargs,2}}.
{label,264}.
  {test,is_nonempty_list,{else,265},[{x,0}]}.
  {get_list,{x,0},{x,0},{y,0}}.
  {call,2,foo}.
  {move,{x,0},{x,1}}.
  {move,{y,0},{x,0}}.
  {call_last,2,bat,1}.
{label,265}.
  {test,is_nil,{else,263},[{x,0}]}.
  {move,{x,1},{x,0}}.
  return.
{label,263}.
  {func_info,{atom,appmon_bar},{atom,bat},2}.
```

```
public static EObject
    bat___2(EProc eproc, EObject arg1, EObject arg2)
{
   ECons cons; ENil nil;
   tail:
    if((cons = arg1.test_nonempty_list()) != null) {
      // extract list
      EObject hd = cons.head();
      EObject tl = cons.tail();
      // call foo/2
      EObject tmp = foo___2(eproc, hd, arg2);
      // self-tail recursion
      arg1 = tl;
      arg2 = tmp;
      goto tail;
    } else if ((nil = arg1.test_nil()) != null) {
      return arg2;
    }
   throw ERT.Bodyc_info(am_bar, am_bat, 2);
}
```

Erlang ⇒ JVM

```
-module(bar).

bat([H | T], T2) ->
  bat(T, foo(H, T2));

bat([], T2) -> T2.
```

```
foo(H, T) ->
  lists:reverse(H ++ T).
```

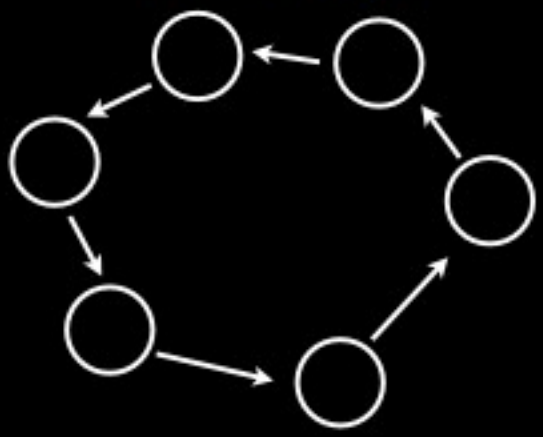

```
foo(H, T) ->
  lists:reverse(H ++ T).
```

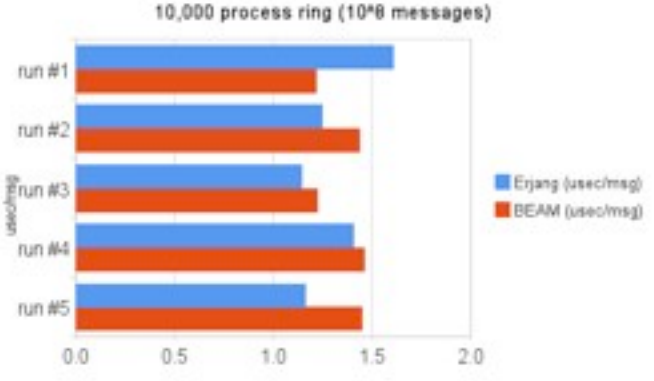
```
public static EObject
    foo__2(EProc p, EObject H, EObject T)
{
    EObject r = foo__2$body(p,H,T);
    while (r == TAIL_MARKER) {
       r = p.tail.go();
    }
    return r;
}
```

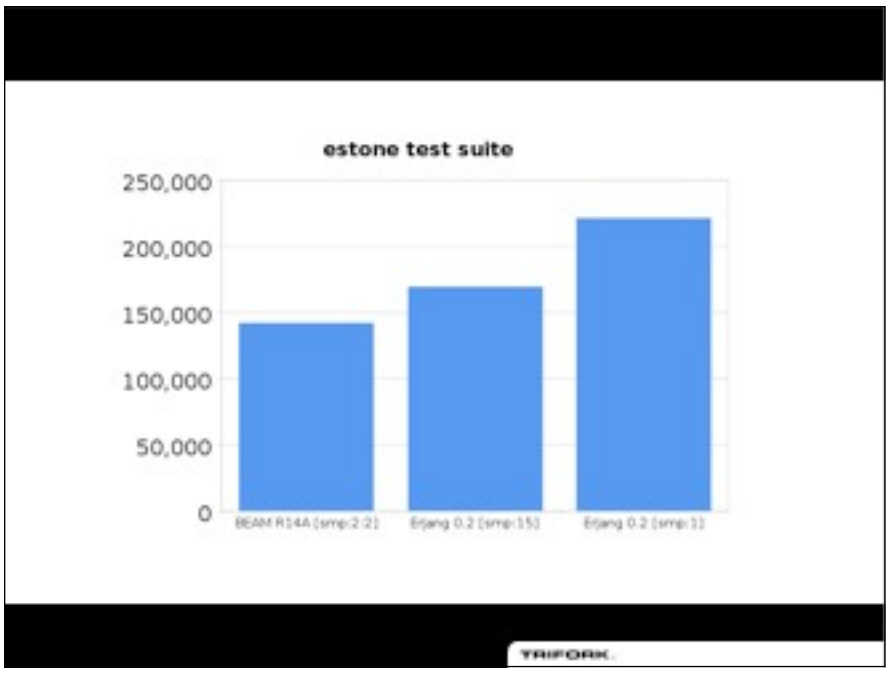
```
foo(H, T) ->
  lists:reverse(H ++ T).
```

```
public static
    EObject foo__2$body(EProc p, EObject H, EObject T)
{
    // Tmp = erlang:'++'(H,T)
    EObject tmp = erlang_append__2.invoke(p,H,T);

    // return lists:reverse(Tmp)
    p.tail = lists__reverse_1;
    p.arg1 = tmp;
    return TAIL_MARKER;
}
```

The ring!







Interfacing to Java

- Erlang's primitive operations "BIFs" are implemented in Java
- `@BIF` annotation makes a static-public method available from Erlang.
- Erlang port concept for "drivers"

Example BIF

```
// foo:bar(...) native Bodyction

package erjang.m.foo;
class foo extends ENative {

  @BIF public static
  EObject bar(EProc proc, EObject arg1, arg2, ...)  {


  }
}
```

Example BIF

```
@BIF public static EObject
spawn_link(EProc proc, EObject mod, EObject fun, EObject args)
     throws Pausable {
        EAtom m = mod.testAtom();
        EAtom f = fun.testAtom();
        ESeq  a = args.testSeq();

        if (m==null||f==null||a==null)
                throw ERT.badarg(mod, fun, args);

        EProc p2 = new EProc(proc.group_leader(), m, f, a);
        p2.link_to(proc);
        ERT.run(p2);

        return p2.self_handle();
}
```

Interfacing to Java

```
demo() ->
   Map = 'java.util.HashMap':new(),
   Map:put('x', "4"),
   Map:put(1, 'foo'),
   print(Map).

print([]) -> ok;
print([{Key,Val}|Tail]) ->
   io:format("key=~p, value=~p~n",
                [Key,Val]),
   print(Tail).
```