

PRACTICAL CONCURRENT - & - PARALLEL PROGRAMMING

MESSAGE PASSING CONCURRENCY II / II



Claus Brabrand
(((brabrand@itu.dk)))

Associate Professor, Ph.D.
(((Dept. of Computer Science)))
 **IT University of Copenhagen**

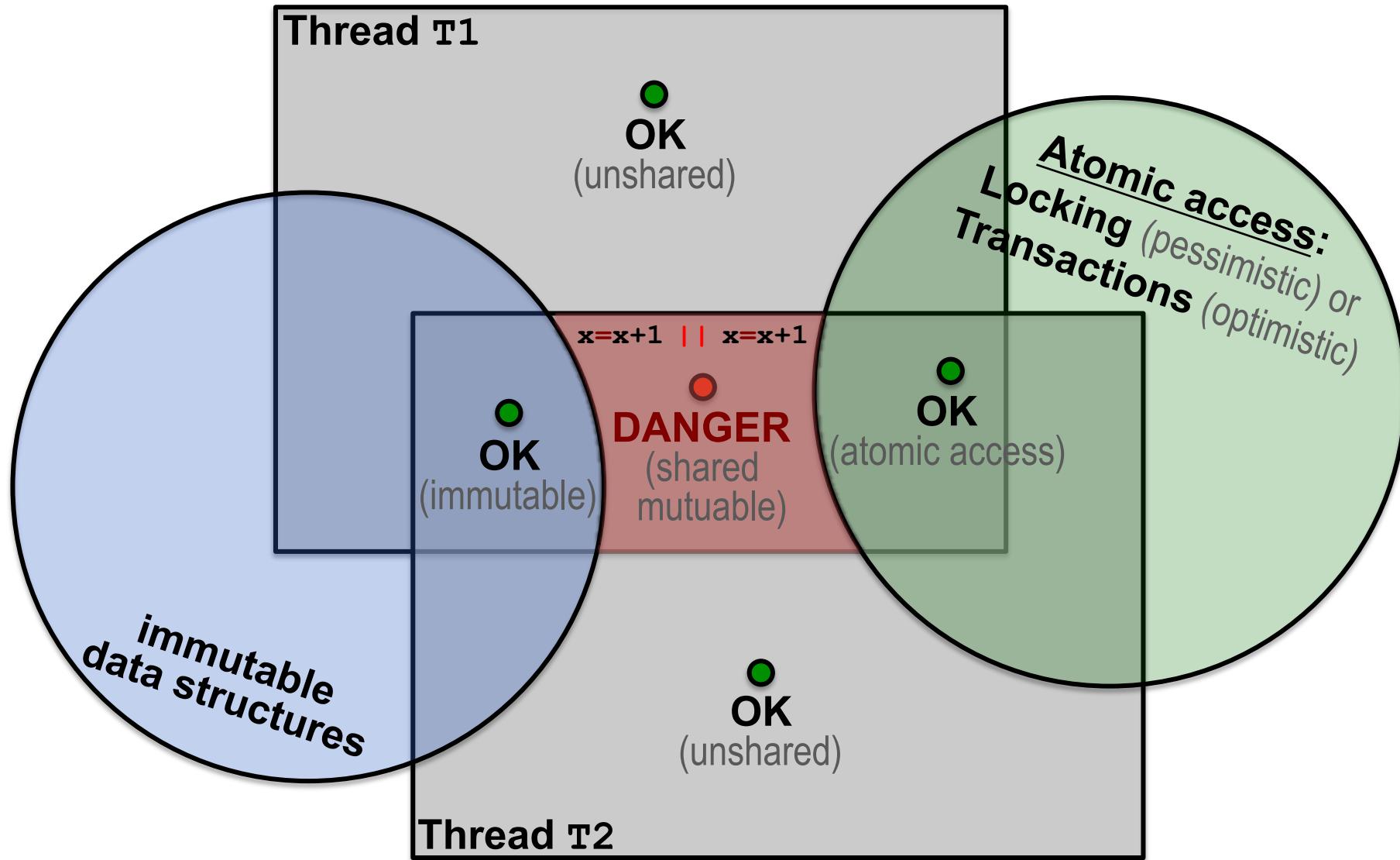
Recap

Introduction to Message Passing

PROBLEM: **Sharing && Mutability!**

SOLUTIONS:

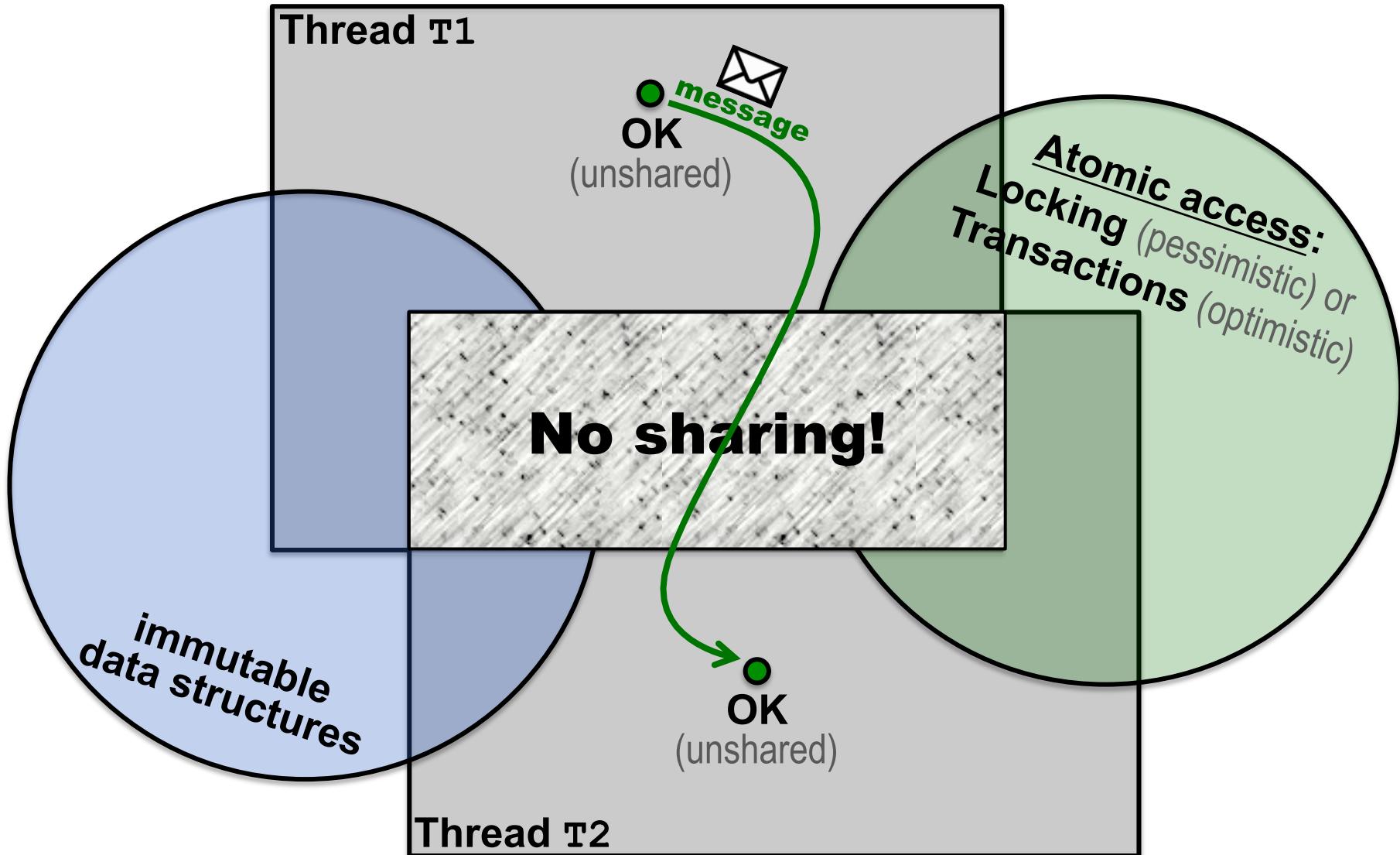
- 1) atomic access!
locking or transactions
NB: avoid deadlock!
- 2) avoid mutability!
- 3) avoid sharing...



PROBLEM: **Sharing && Mutability!**

SOLUTIONS:

- 1) atomic access!**
locking or transactions
NB: avoid deadlock!
- 2) avoid mutability!**
- 3) avoid sharing...**



Philosophy & Expectations!

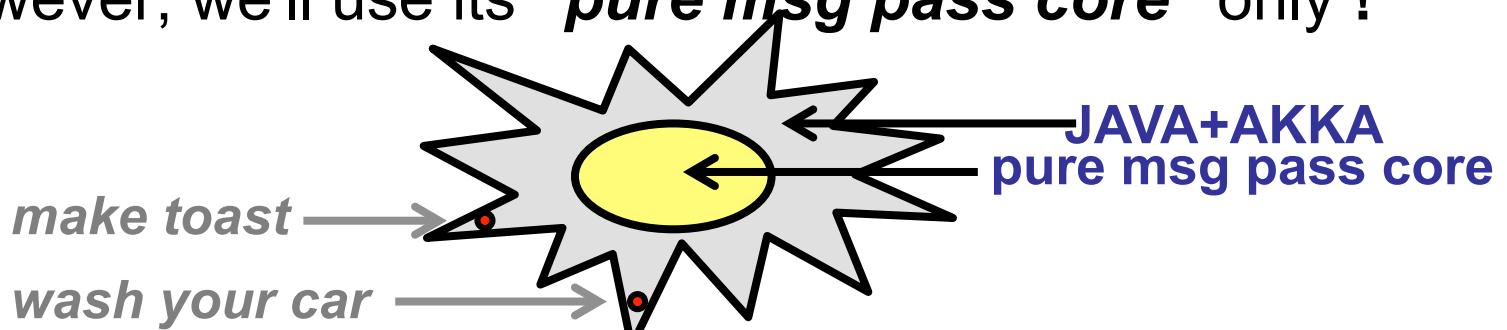
■ ERLANG:

- We'll use as message passing *specification language*
- You have to-be-able-to *read* simple ERLANG programs
 - (i.e., not *write*, nor *modify*)

■ JAVA+AKKA:

- We'll use as msg passing *implementation language*
- You have 2-b-a-2 *read/write/modify* JAVA+AKKA p's
- However, we'll use its "*pure msg pass core*" only !

NB: we're not going to use all of its fantazilions of functions!



Actor: Send / Receive / Spawn

■ Send:

- `Pid ! M` // Message M is sent to process Pid
- `Pid ! {some, {complex, structured, [m,s,g]}, 42}`

■ Receive:

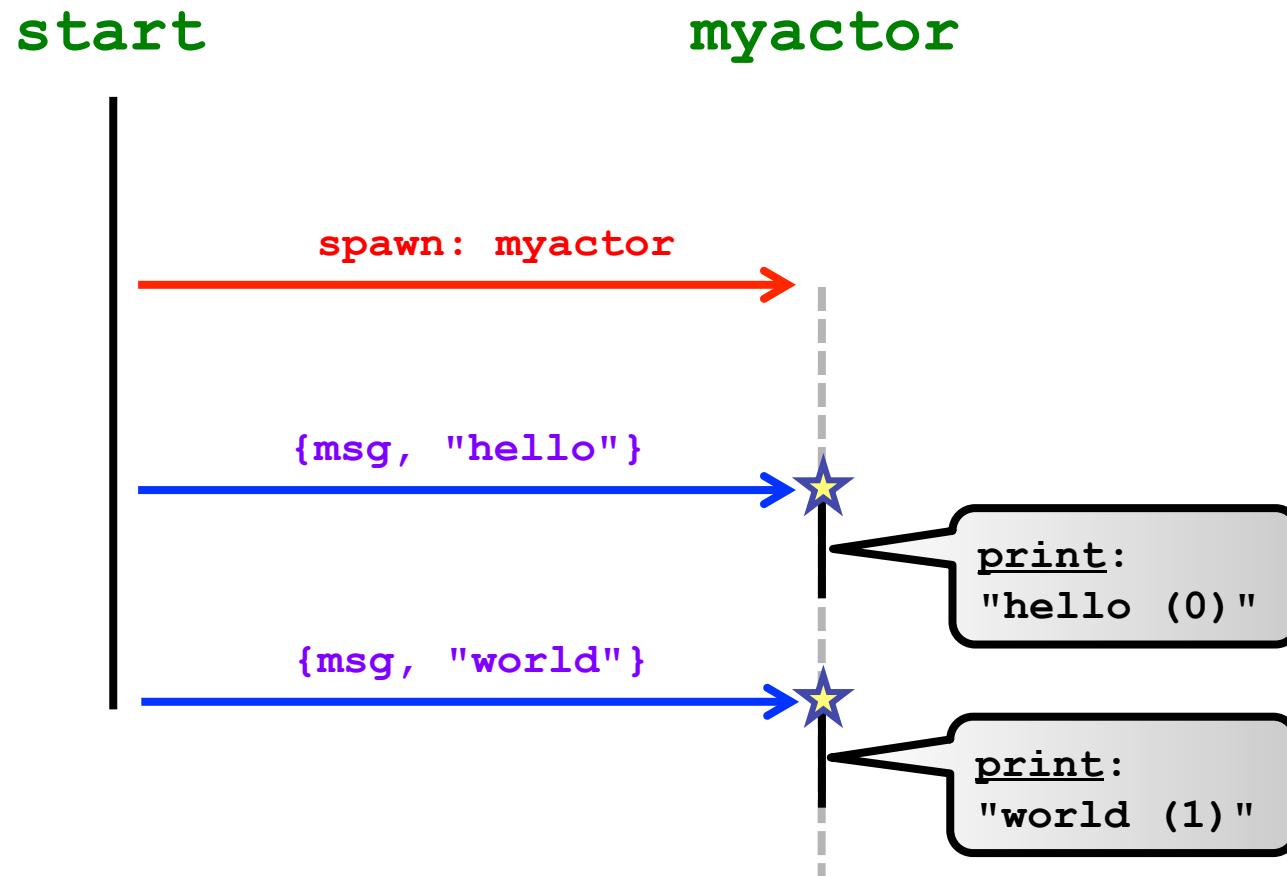
- ```
receive
 pattern1 -> ...
 ;
 pattern2 -> ...
end
```
- ```
receive
    {init,N} when N>0 -> ...
    ;
    {init,N} -> ...
end
```

■ Spawn:

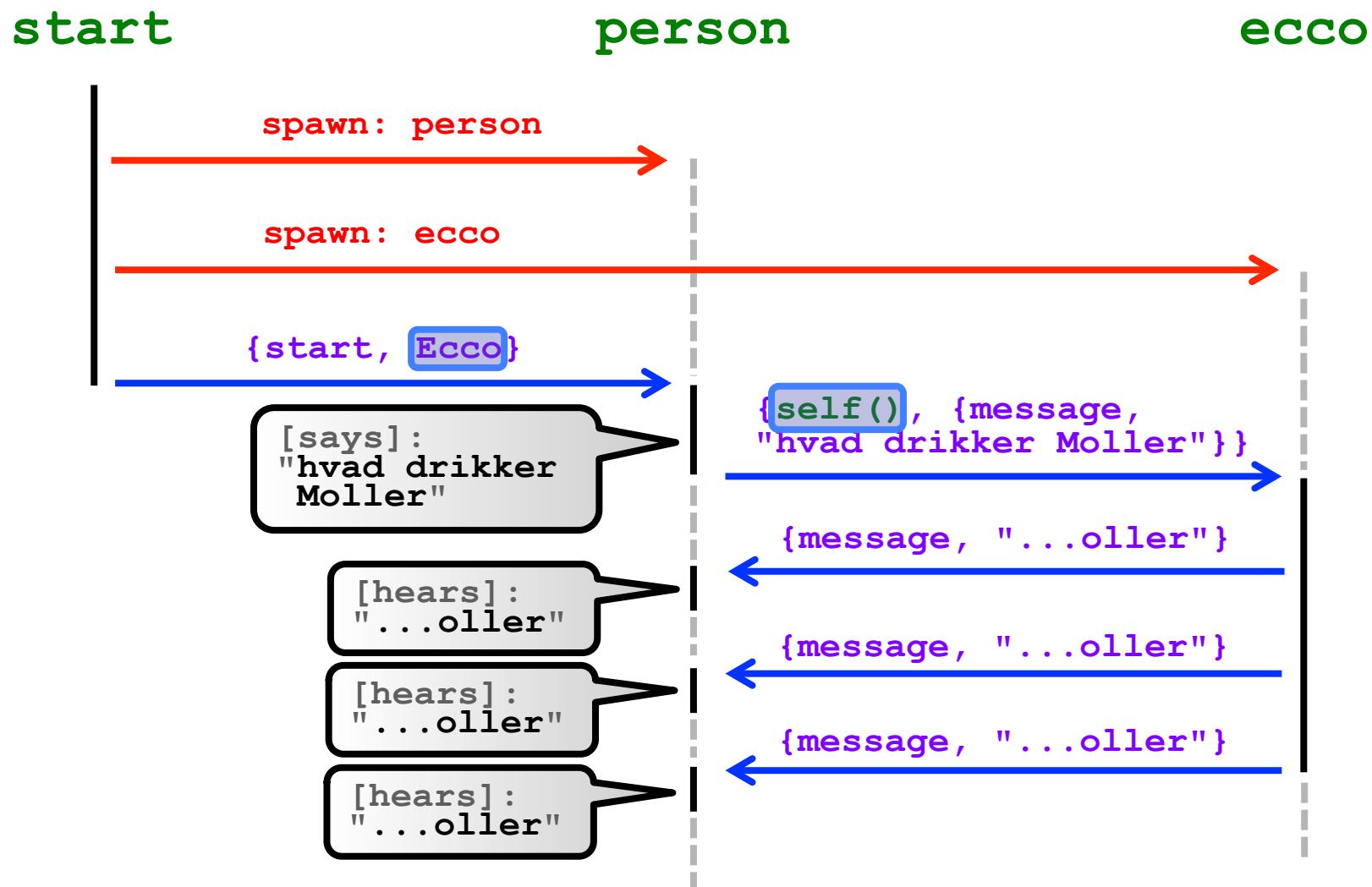
- `MyActorId = spawn(mymodule,myactor,[a,r,g,s])`

1) HelloWorld

LEGEND:
send, receive, msgs
actors, spawn, rest.



2) Ecco



AGENDA

■ 3) Broadcast:

- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one ⇒ one-to-many)

■ AKKA: A proper introduction

- Motivations and benefits of Actors & Message Passing
- Recommendations

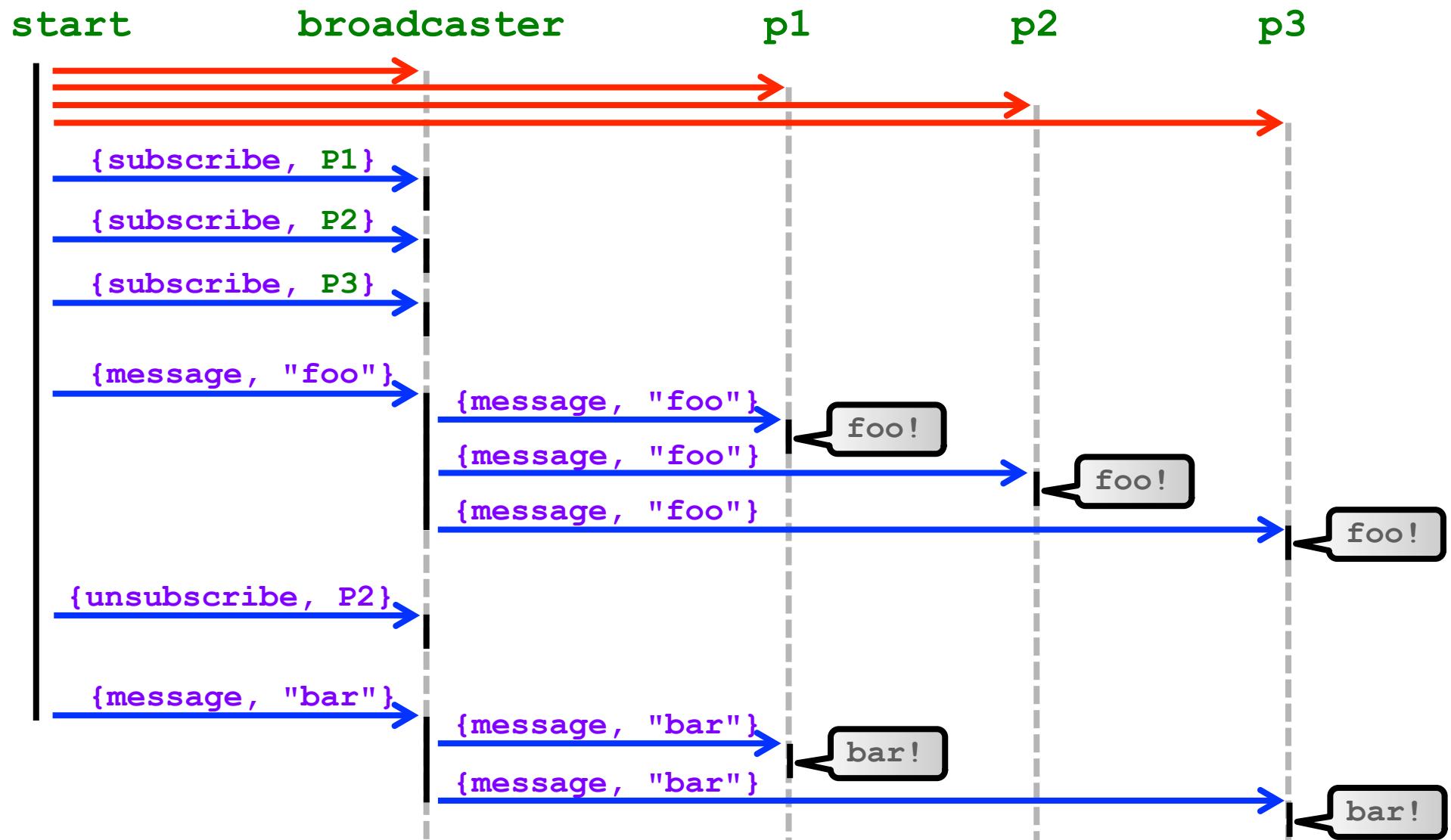
■ 4) Primer:

- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

■ ★ Scatter-Gatherer:

- Prototypical AKKA Service (dynamic load balancing)
- Extensions...

3) Broadcast



3) Broadcast.erl

```

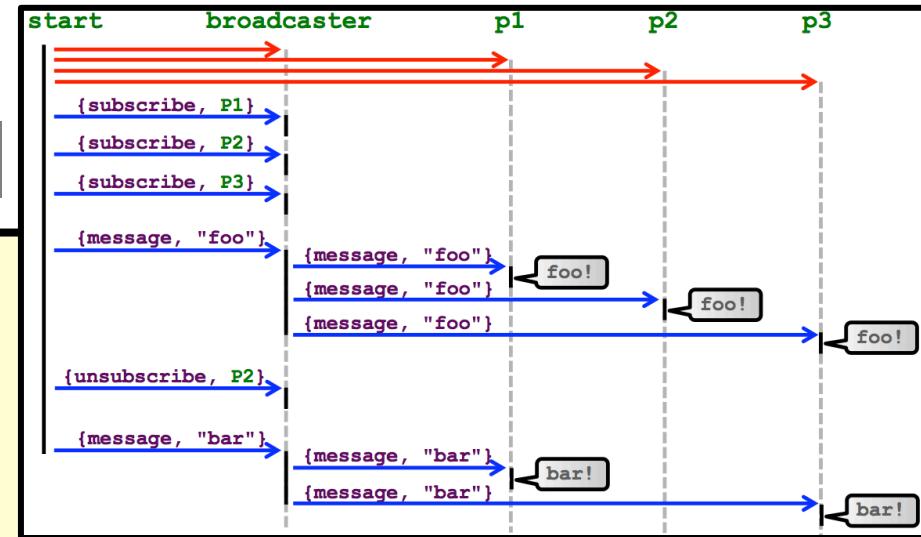
-module(helloworld).
-export([start/0, person/0, broadcaster/1]).

person() ->
    receive
        {message,M} ->
            io:fwrite(M ++ "\n"),
            person()
    end.

% ∀P∈L : send M to P
broadcast([],_) -> true;
broadcast([Pid|L],M) ->
    Pid ! {message,M},
    broadcast(L,M).

broadcaster(L) ->
    receive
        {subscribe,Pid} ->
            broadcaster([Pid|L]);
        {unsubscribe,Pid} ->
            broadcaster(lists:delete(Pid,L)) ; % L \ Pid
        {message,M} ->
            broadcast(L,M),
            broadcaster(L)
    end.

```



```

start() ->
    Broadcaster = spawn(helloworld,broadcaster,[[ ]]),
    P1 = spawn(helloworld, person, [ ] ),
    P2 = spawn(helloworld, person, [ ] ),
    P3 = spawn(helloworld, person, [ ] ),
    Broadcaster ! { subscribe, P1 },
    Broadcaster ! { subscribe, P2 },
    Broadcaster ! { subscribe, P3 },
    Broadcaster ! { message, "Purses half price!" },
    Broadcaster ! { unsubscribe, P2 },
    Broadcaster ! { message, "Shoes half price!!" }.

```

purses half price!
purses half price!
purses half price!
shoes half price!!
shoes half price!!

3) Broadcast.java

```

import java.util.*;
import java.io.*;
import akka.actor.*;

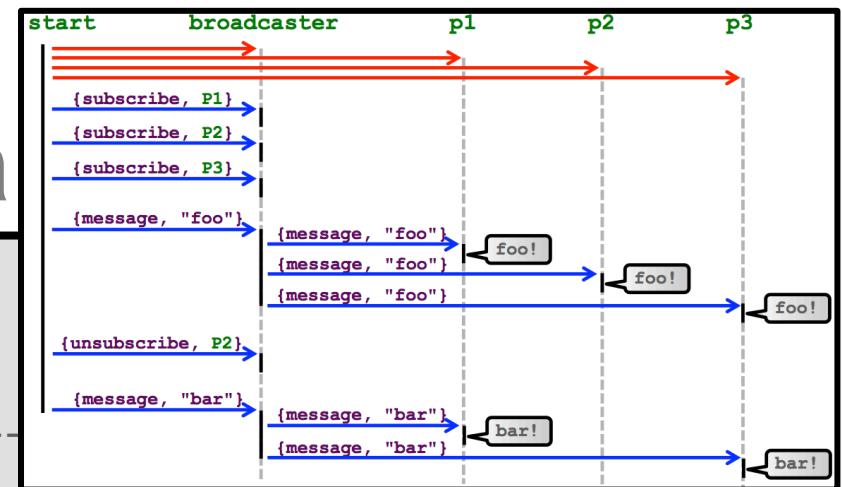
// -- MESSAGES --

class SubscribeMessage implements Serializable {
    public final ActorRef subscriber;
    public SubscribeMessage(ActorRef subscriber) {
        this.subscriber = subscriber;
    }
}

class UnsubscribeMessage implements Serializable {
    public final ActorRef unsubscribe;
    public UnsubscribeMessage(ActorRef unsubscribe) {
        this.unsubscribe = unsubscribe;
    }
}

class Message implements Serializable {
    public final String s;
    public Message(String s) {
        this.s = s;
    }
}

```

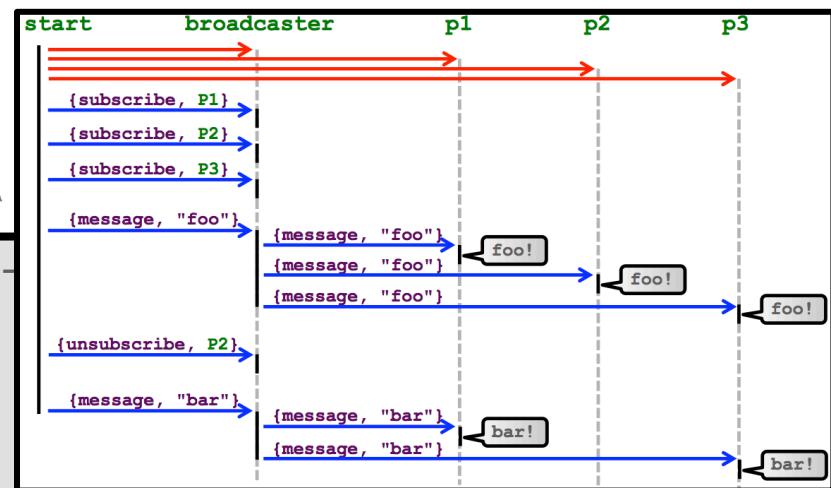


purses half price!
purses half price!
purses half price!
shoes half price!!
shoes half price!!

3) Broadcast.java

```
// -- MAIN -->

public class Broadcast {
    public static void main(String[] args) {
        final ActorSystem system =
            ActorSystem.create("BroadcastSystem");
        final ActorRef broadcaster =
            system.actorOf(Props.create(BroadcastActor.class), "broadcaster");
        final ActorRef p1 = system.actorOf(Props.create(PersonActor.class), "p1");
        final ActorRef p2 = system.actorOf(Props.create(PersonActor.class), "p2");
        final ActorRef p3 = system.actorOf(Props.create(PersonActor.class), "p3");
        broadcaster.tell(new SubscribeMessage(p1), ActorRef.noSender());
        broadcaster.tell(new SubscribeMessage(p2), ActorRef.noSender());
        broadcaster.tell(new SubscribeMessage(p3), ActorRef.noSender());
        broadcaster.tell(new Message("purses half price!"), ActorRef.noSender());
        broadcaster.tell(new UnsubscribeMessage(p2), ActorRef.noSender());
        broadcaster.tell(new Message("shoes half price!!!"), ActorRef.noSender());
        try {
            System.out.println("Press return to exit");
            System.in.read();
        } catch(IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}
```



```

start() ->
Broadcaster = spawn(helloworld,broadcaster,[]),
P1 = spawn(helloworld,person,[]),
P2 = spawn(helloworld,person,[]),
P3 = spawn(helloworld,person,[]),
Broadcaster ! {subscribe,P1},
Broadcaster ! {subscribe,P2},
Broadcaster ! {subscribe,P3},
Broadcaster ! {message,"Purses half price!"},
Broadcaster ! {unsubscribe,P2},
Broadcaster ! {message,"Shoes half price!!!"}.
  
```

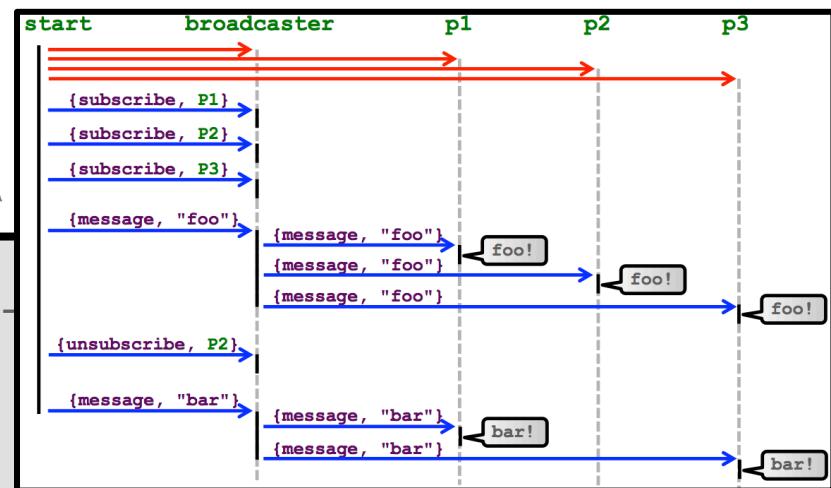
3) Broadcast.java

```
// -- ACTORS --

class PersonActor extends UntypedActor {
    public void onReceive(Object o) throws Exc' {
        if (o instanceof Message) {
            System.out.println(((Message) o).s);
        }
    }
}

class BroadcastActor extends UntypedActor {
    private List<ActorRef> list =
        new ArrayList<ActorRef>();

    public void onReceive(Object o) throws Exception {
        if (o instanceof SubscribeMessage) {
            list.add(((SubscribeMessage) o).subscriber);
        } else if (o instanceof UnsubscribeMessage) {
            list.remove(((UnsubscribeMessage) o).unsubscriber);
        } else if (o instanceof Message) {
            for (ActorRef person : list) {
                person.tell(o, getSelf());
            }
        }
    }
}
```



```

person() ->
receive
{message,M} ->
io:fwrite(M ++ "\n"),
person()
end.

```

```

broadcaster(L) ->
receive
{subscribe,Pid} ->
broadcaster([Pid|L]);
{unsubscribe,Pid} ->
broadcaster( L \ Pid );
{message,M} ->
% ∀P∈L : send M to P,
broadcaster(L)
end.

```

3) Broadcast.java

■ Compile:

```
javac -cp scala.jar:akka-actor.jar Broadcast.java
```

■ Run:

```
java -cp scala.jar:akka-actor.jar:akka-config.jar:. Broadcast
```

■ Output:

```
purses half price!
purses half price!
purses half price!
shoes half price!!
shoes half price!!
```

AGENDA

- **3) Broadcast:**

- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one ⇒ one-to-many)

- **AKKA: A proper introduction**

- Motivations and benefits of Actors & Message Passing
- Recommendations

- **4) Primer:**

- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

- **★ Scatter-Gatherer:**

- Prototypical AKKA Service (dynamic load balancing)
- Extensions

AKKA

- **Mountain in Sweden:**

- Northern Sweden
- (Close to Norway)



- **Nordic Goddesses: "Àhkkas"**

- From Nordic/Arctic/Saami Mythology
- The Àhkkas: daughters of Mother Sun
- Ancient creator goddesses of the past



- **Software runtime middleware:**

- For Java Virtual Machine (made in Scala)





Telefonica

csc

htc
quietly brilliant

BLIZZARD®
ENTERTAINMENT

UBS

SIEMENS

amazon.com®

hp

KLOUT

W3C®

HSBC

CISCO

HUAWEI

banksimple

svt

Autodesk

CREDIT SUISSE

IGN

Atos

O₂

The vmware logo features a stylized blue square composed of smaller squares.

vmware®

dialog®
Smart Stream Platform

SEVEN®
Networks

xerox

UNIBET

DRW TRADING GROUP

OOYALA®

novus

navirec

T8 Webware

CARTOMAPIC

The moshi monsters logo features the words "moshi" and "monsters" in a green, bubbly font with small cartoon characters.

BBC

JUNIPER.
NETWORKS

ngmoco:)

Maritime Poker

Answers.com®
The world's leading Q&A site

azavea

zeebox
The best thing to happen to TV since TV

Why the Sudden Popularity?!?

- Recently, processor speed "hit the wall":
 - Speed of light: $c = 3 \cdot 10^8$ m/s (meters per sec)
 - Processor speed: $s = 3 \cdot 10^9$ x/s (instructions per sec)
 -  $c/s = (3 \cdot 10^8 \text{m/s} / 3 \cdot 10^9 \text{x/s}) = 0.1 \text{ m/x}$ (meters per instruction)
 - i.e., "***light travels 10cm per instruction***"
- Before (more speed):
 -  **Moore's Law:** #Transistors-per-CPU **doubles** every 1.8 years
 -  **Dennard Scaling:** Performance-per-Watt **doubled** every 1.6 years
(until ~2006)
- Now (more speed):
 - We have to ***increase parallelism*** !
 - Recent developments: "***Work Stealing Queue***"

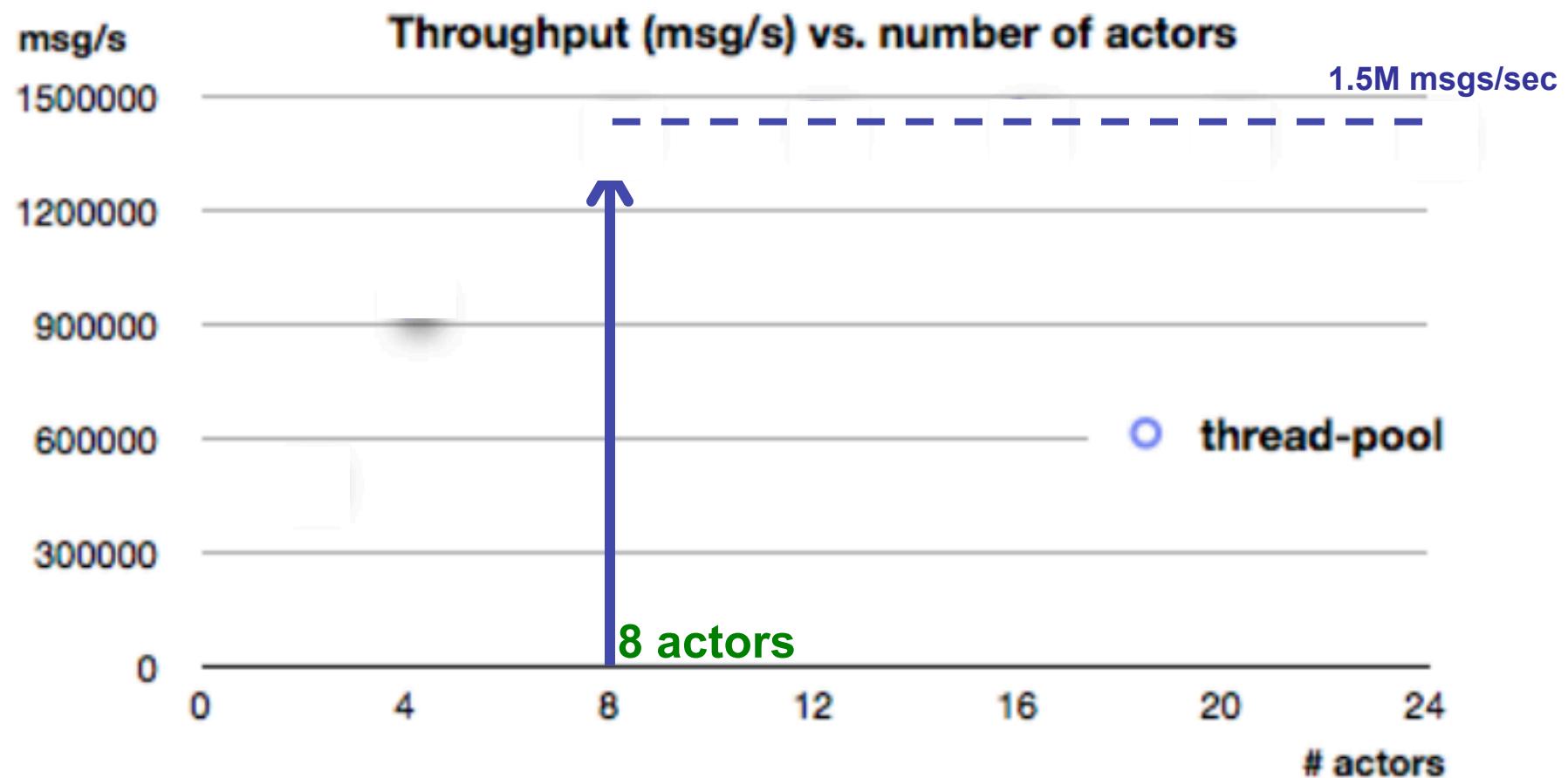
used for:
managing
the actors
effectively

Chase and Lev: "***Dynamic Circular Work-Stealing Queue***", SPAA 2005

Michael, Vechev, Saraswat: "***Idempotent Work Stealing***", PPoPP 2009

Scalability? :-()

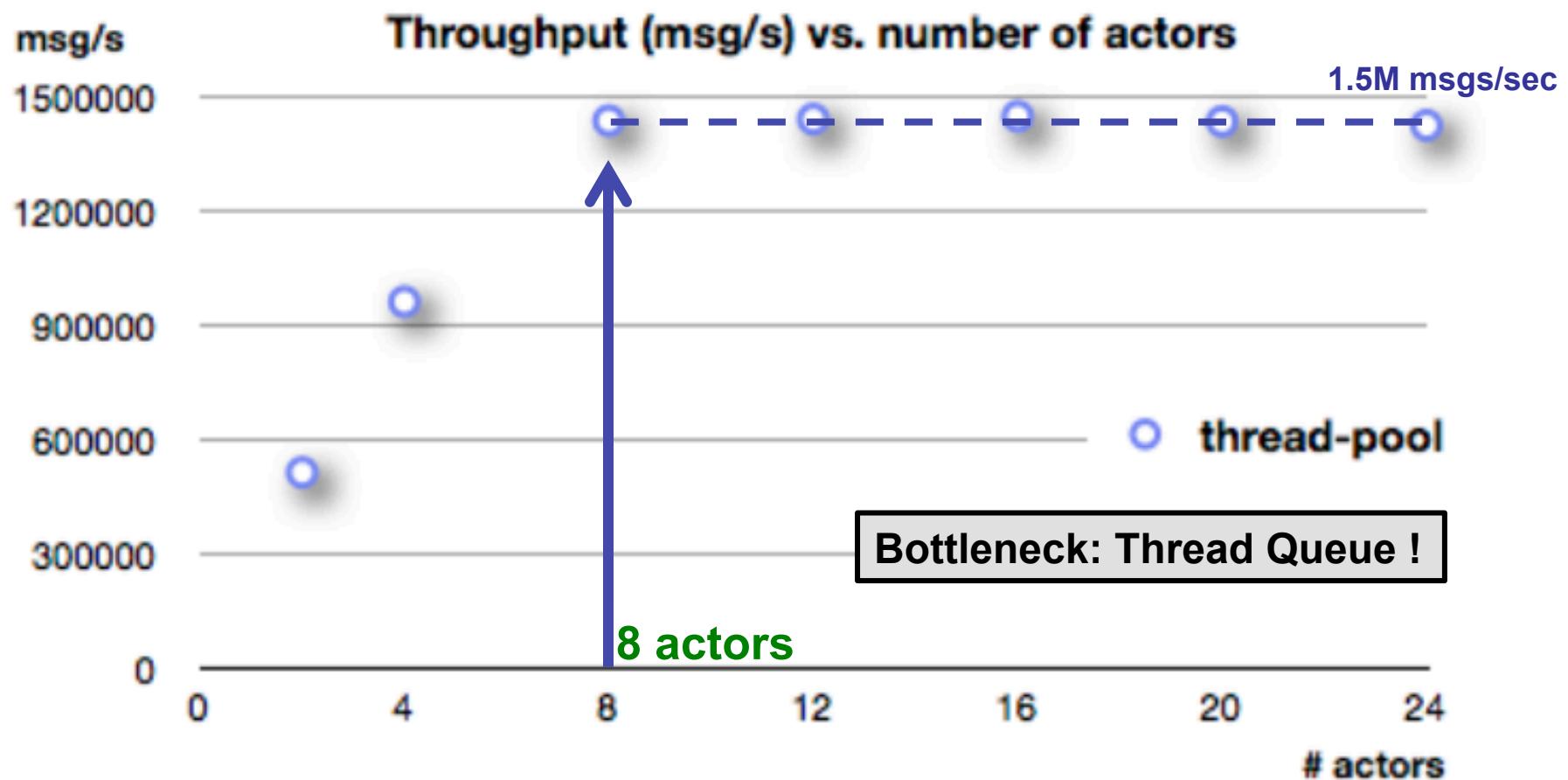
- Using a conventional "Thread Pool Executor":



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

Scalability? :-()

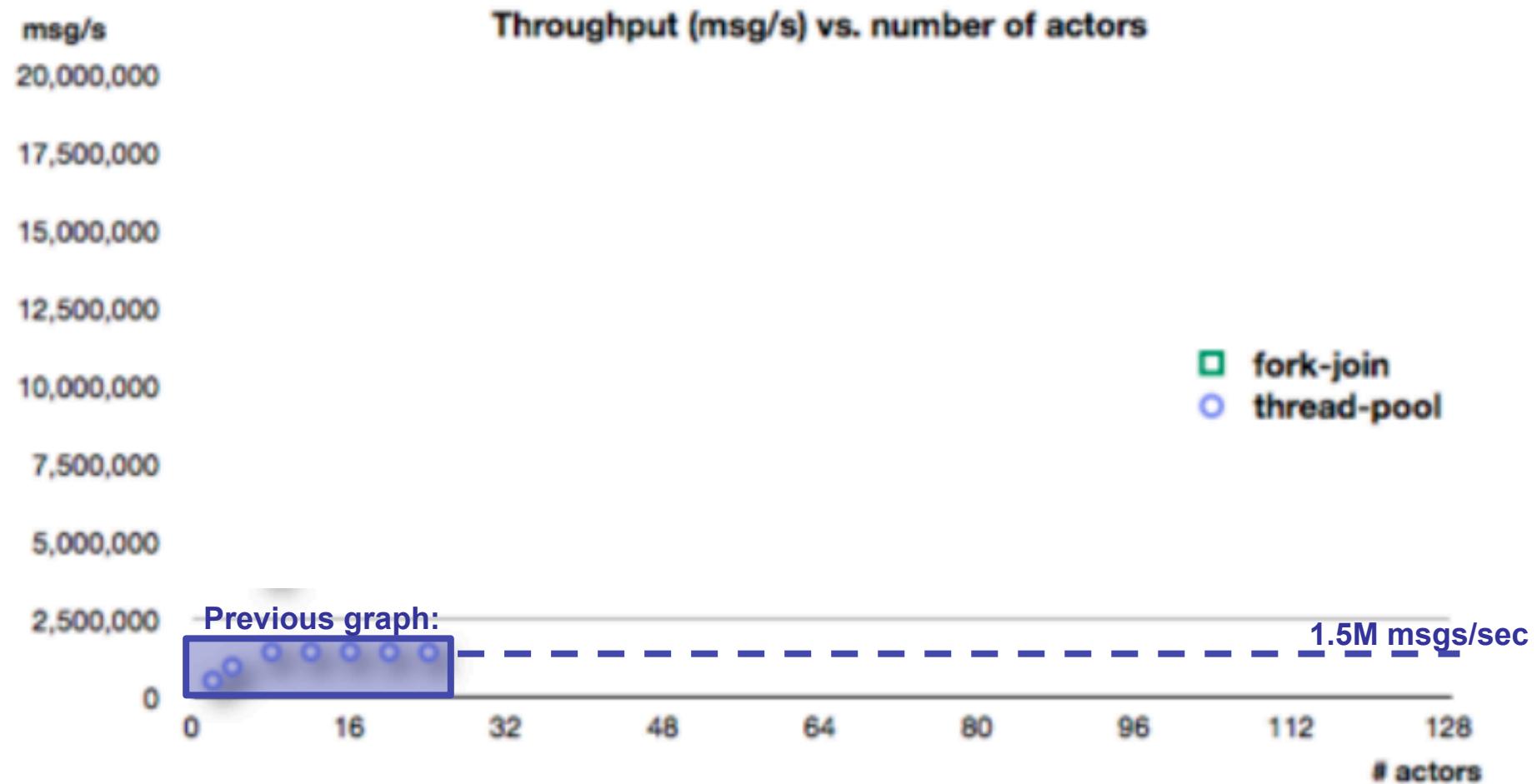
- Using a conventional "Thread Pool Executor":



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

Scalability!

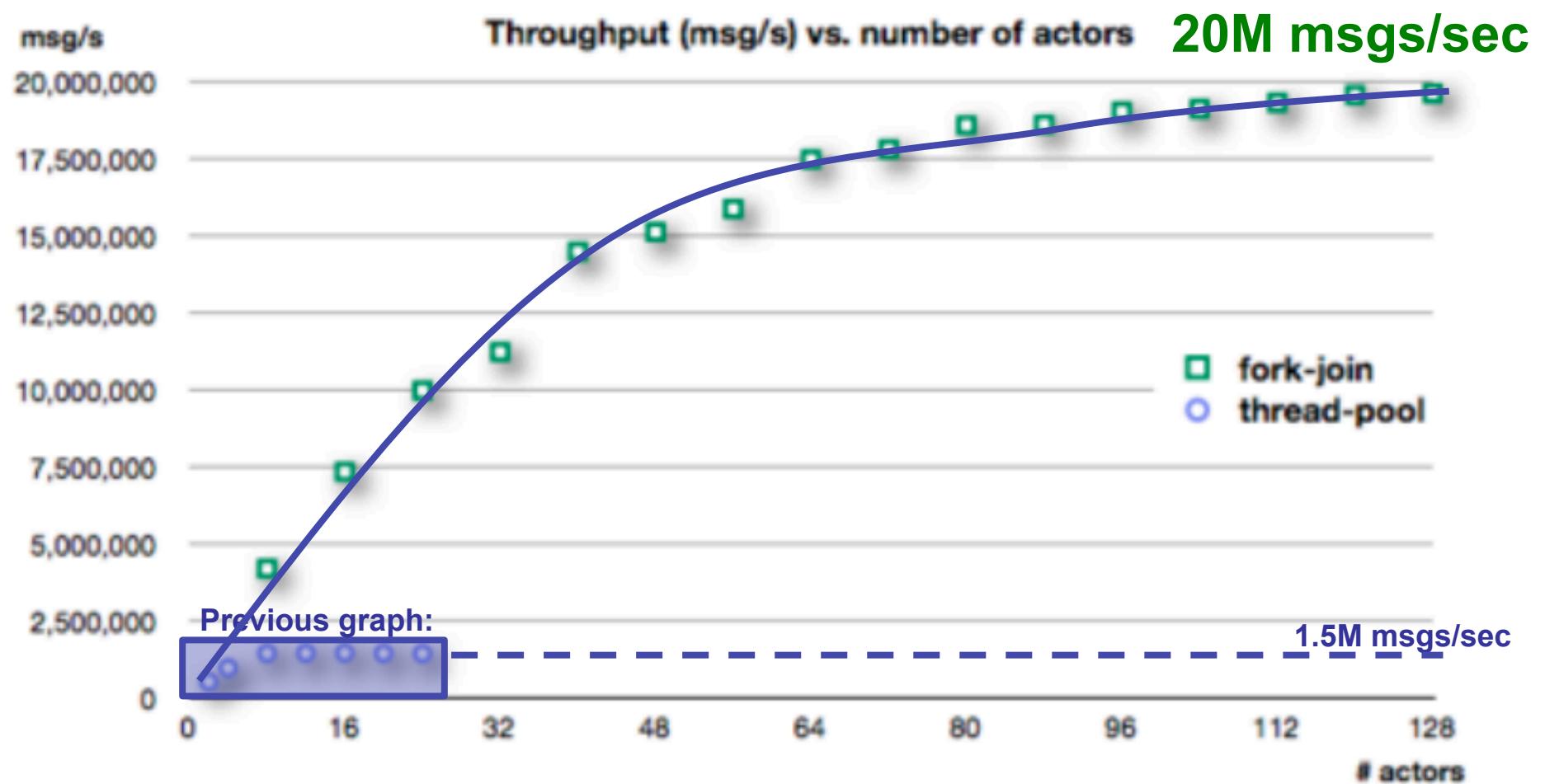
- Java 8's New Fork Join Pool (Work Stealing):



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

Scalability!

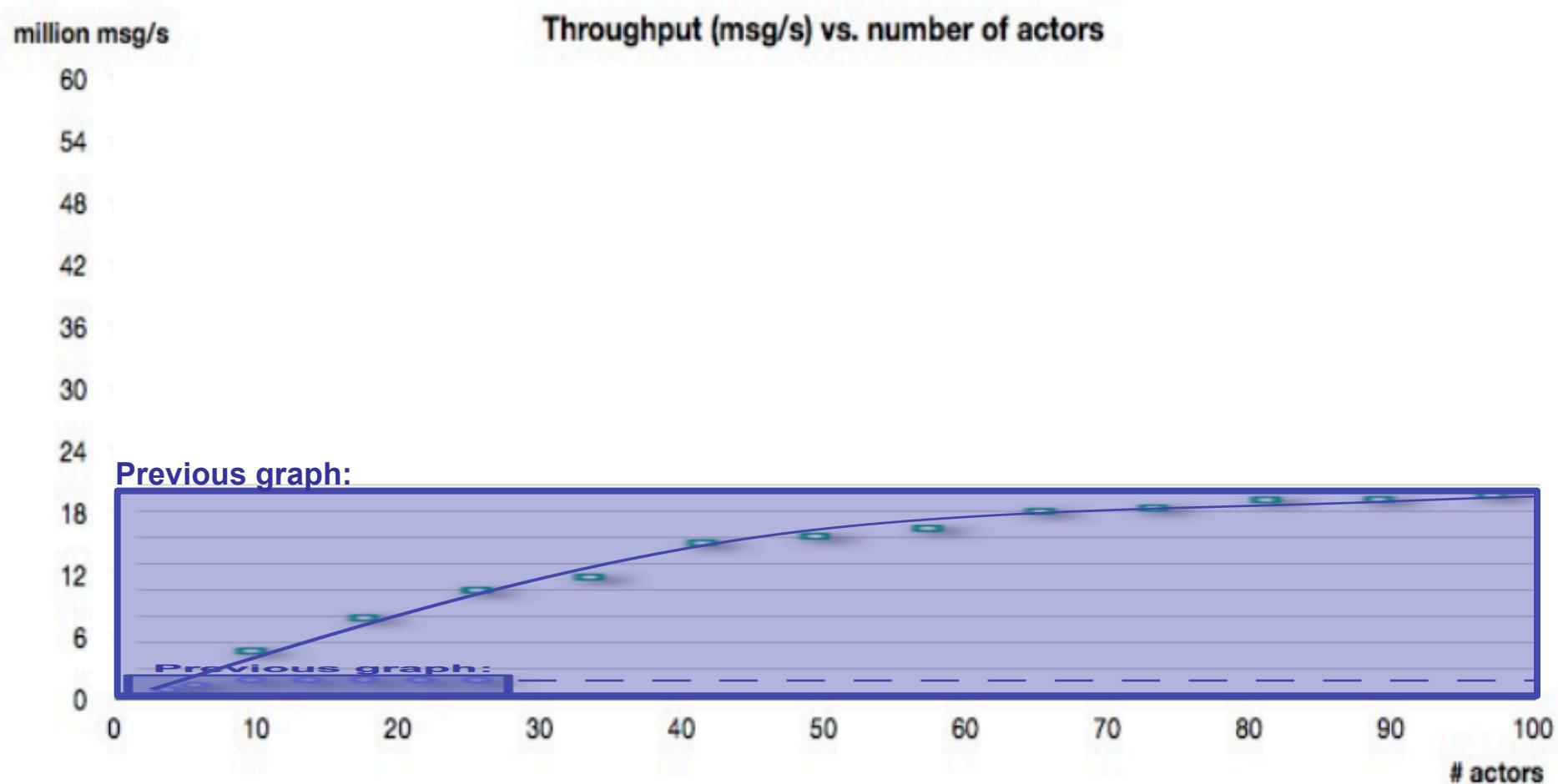
- Java 8's New Fork Join Pool (Work Stealing):



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

Scalability! :-)

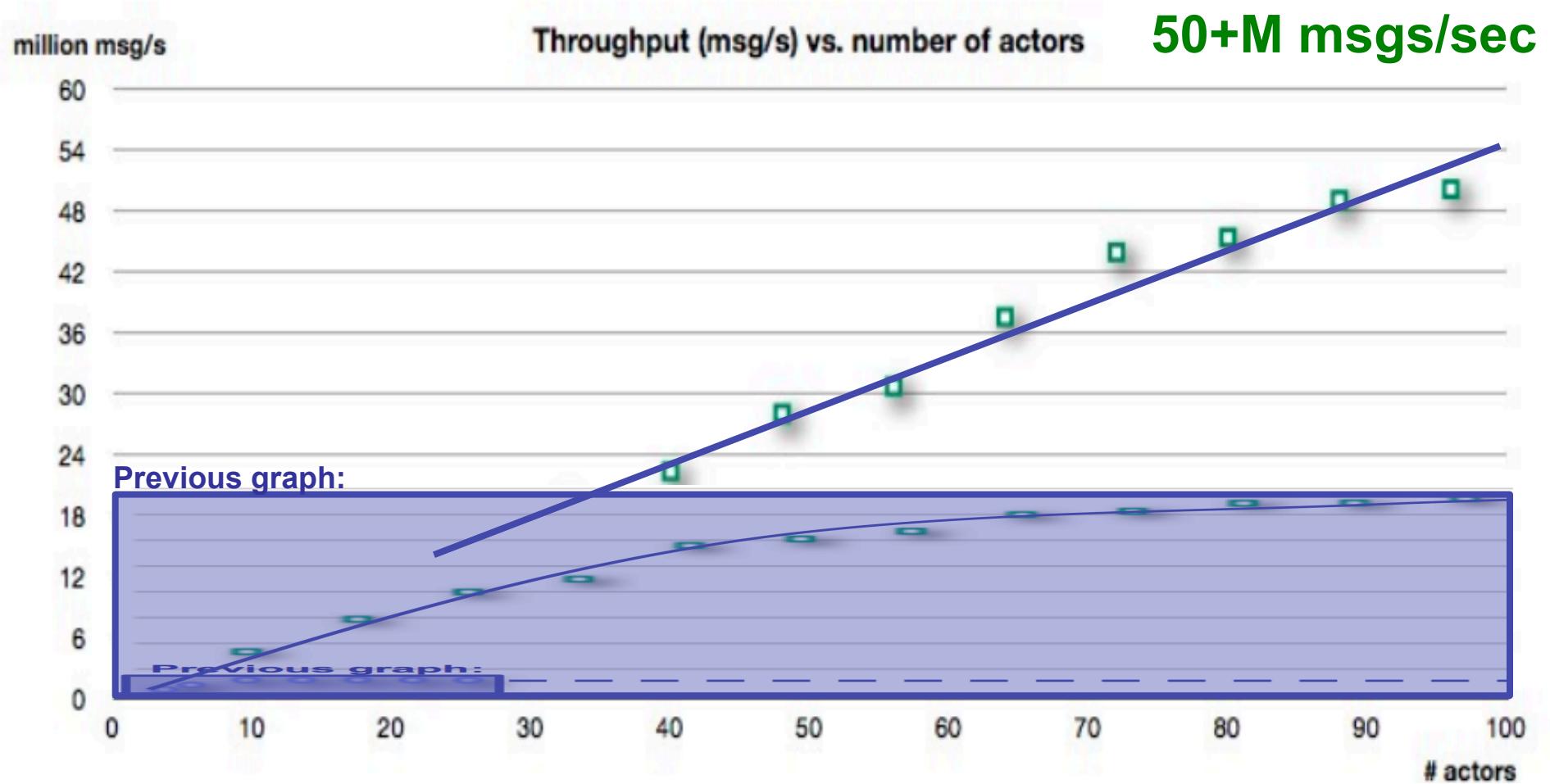
- ...and after optimizing for throughput:



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

Scalability! :-)

- ...and after optimizing for throughput:



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

Actors

*Definition of an Actor
The People Metaphor*

An Actor is....:

- A fundamental *unit of computation* that embodies:
 - 1) Processing (usually simple)
 - 2) State (local)
 - 3) Communication
- In particular: no **shared & mutable state!**
- When an actor receives a message, it can....:
 - 1) perform computation
 - 2) change state
 - 3) send messages to actors it knows
 - +) spawn new actors!





Actors & Msg Passing: Benefits

- **Correct highly concurrent systems:**

- Higher level abstraction (via message passing)
 - coordination (declarative/what) ≠ heap mgmt (imperative/how)
- No low-level locking (no **shared && mutable** state)

- **Truly scalable systems:**

- Actors are extremely lightweight entities
- Actors are ***location transparent***
 - => Distributable-by-design
- Transparently map MP programs onto given hardware:
 - "Scale up" (more processors), "Scale out" (more machines)

- **Self-healing, fault-tolerant systems:**

- Adaptive load balancing and actor migration
- "Let it crash" model (deal w/ failure, great success in telecom industry)
- Manage system overload (graceful service degradation)





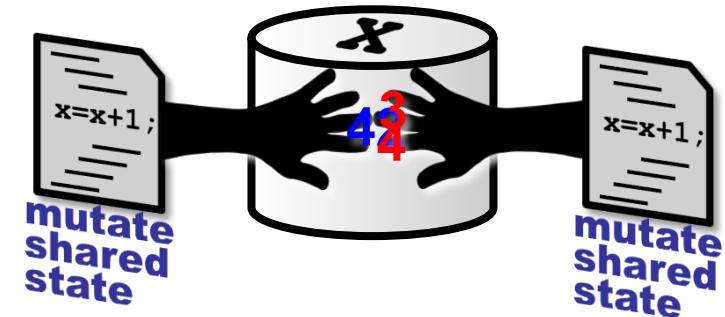
Actors & Msg Passing: Drawbacks

- **New & different paradigm:**
 - Learning curve: programmers unfamiliar with paradigm
- **Overhead of (high level) message passing:**
 - Overhead of sending messages:
 - Less efficient than **shared && mutable state**
 - (Analogy: explicit memory mgmt vs garbage collection)
- **The Correlation problem:**
 - A --what-is-your-favorite-fruit?--> B
 - A --what-is-your-favorite-color?--> B
 - B --orange--> A // now: correlates with which request?!?
- **Hard to identify global state properties:**
 - i.e., computing: **f (global-state)**:
 - e.g., termination condition of a distributed algorithm

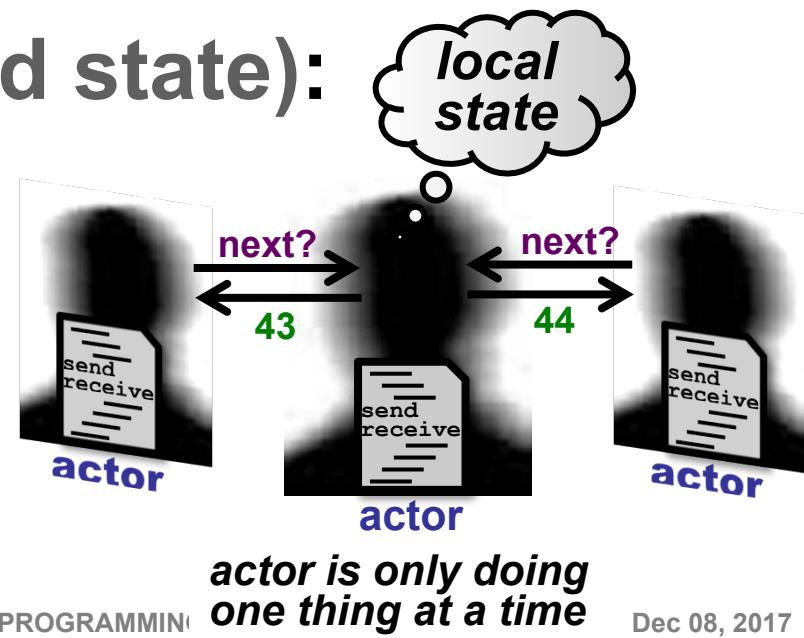


The People Metaphor

- Shared Mutable state:
 - Concurrency problems!
 - Shared && mutable !
 - Hard to (later) distribute!



- Actors (with encapsulated state):
 - Can't "look inside head" of a living person (actor) !
 - Instead: ask questions ?
 - ...and get responses !
 - ...one at a time !



The People Metaphor (cont'd)

- Programming Metaphor:

- "The People Analogy"



- Think of it as "**coordinating lots of people**":

- Each can do simple tasks
 - Consider workflow (of orders/messages in your system)

The People Metaphor (cont'd)

- Programming Metaphor:

- "The People Analogy"



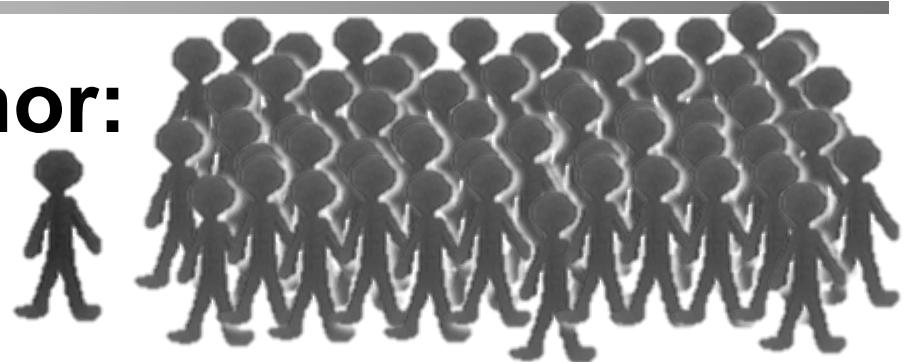
- Think of it as "**coordinating lots of people**":

- Each can do simple tasks
 - Consider workflow (of orders/messages in your system)
 - Need more work ⇒ hire more people (spawn more actors)

The People Metaphor (cont'd)

- Programming Metaphor:

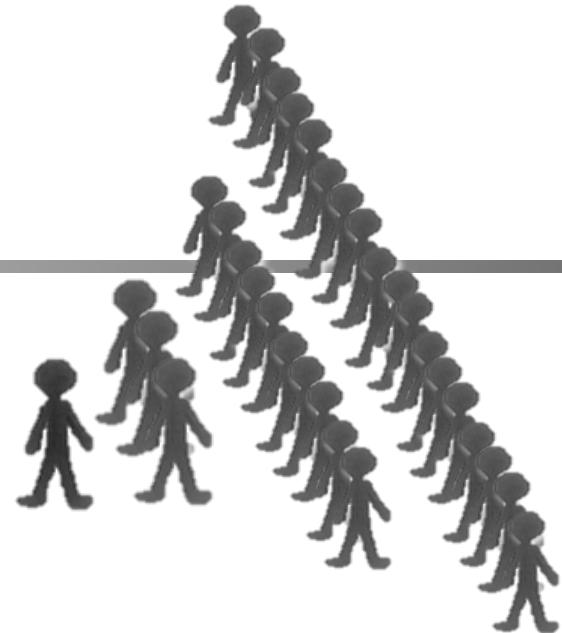
- "The People Analogy"



- Think of it as "**coordinating lots of people**":

- Each can do simple tasks
 - Consider workflow (of orders/messages in your system)
 - Need more work ⇒ hire more people (spawn more actors)

The People Metaphor



- Programming Metaphor:
 - "The People Analogy"
- Think of it as "**coordinating lots of people**":
 - Each can do simple tasks
 - Consider workflow (of orders/messages in your system)
 - Need more work ⇒ hire more people (spawn more actors)
 - Hierarchic organization: managers supervise workers
 - Fault tolerance (expect failures and deal with them)

Recommendations

- **1) Actors should be like nice co-workers:**
 - do their job effectively w/o bothering others needlessly
 - should not roll thumbs (idle or blocking operations)
- **2) Actors should not send mutable objects:**
 - O/w we're back to "shared && mutable" ⇒ problems !
- **3) Actors should send data, not programs:**
 - O/w we're back to "shared && mutable" ⇒ problems !
 - (Note: ERLANG does not have higher-order functions)
- **4) Create few top-level actors:**
 - If these crash, your whole system will crash
 - If their workers die, they just hire (spawn) new ones

Recommendations

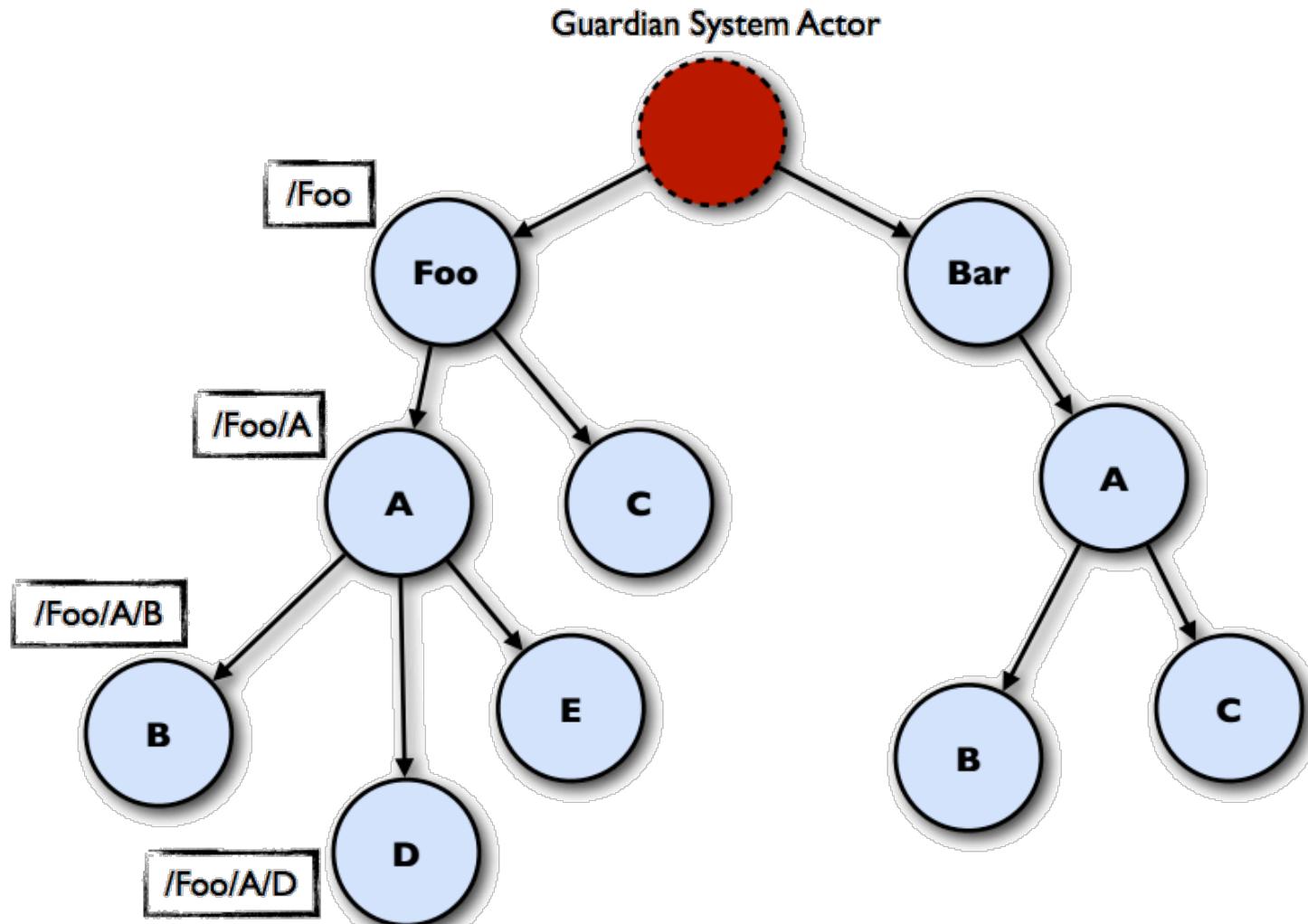
- **5) Managers should supervise Workers:**
 - organization as a hierarchy
 - pass on and schedule tasks for workers
 - "hire" (spawn) more workers by need
 - deal with failure (of your workers)
- **6) Actors should spawn workers for "dangerous operations" (Qatar 2022 ?):**
 - avoid crashing with important data
 - spawn workers for "dangerous operations"
 - deal with failure (of your workers)

Scalability & Fault-Tolerance

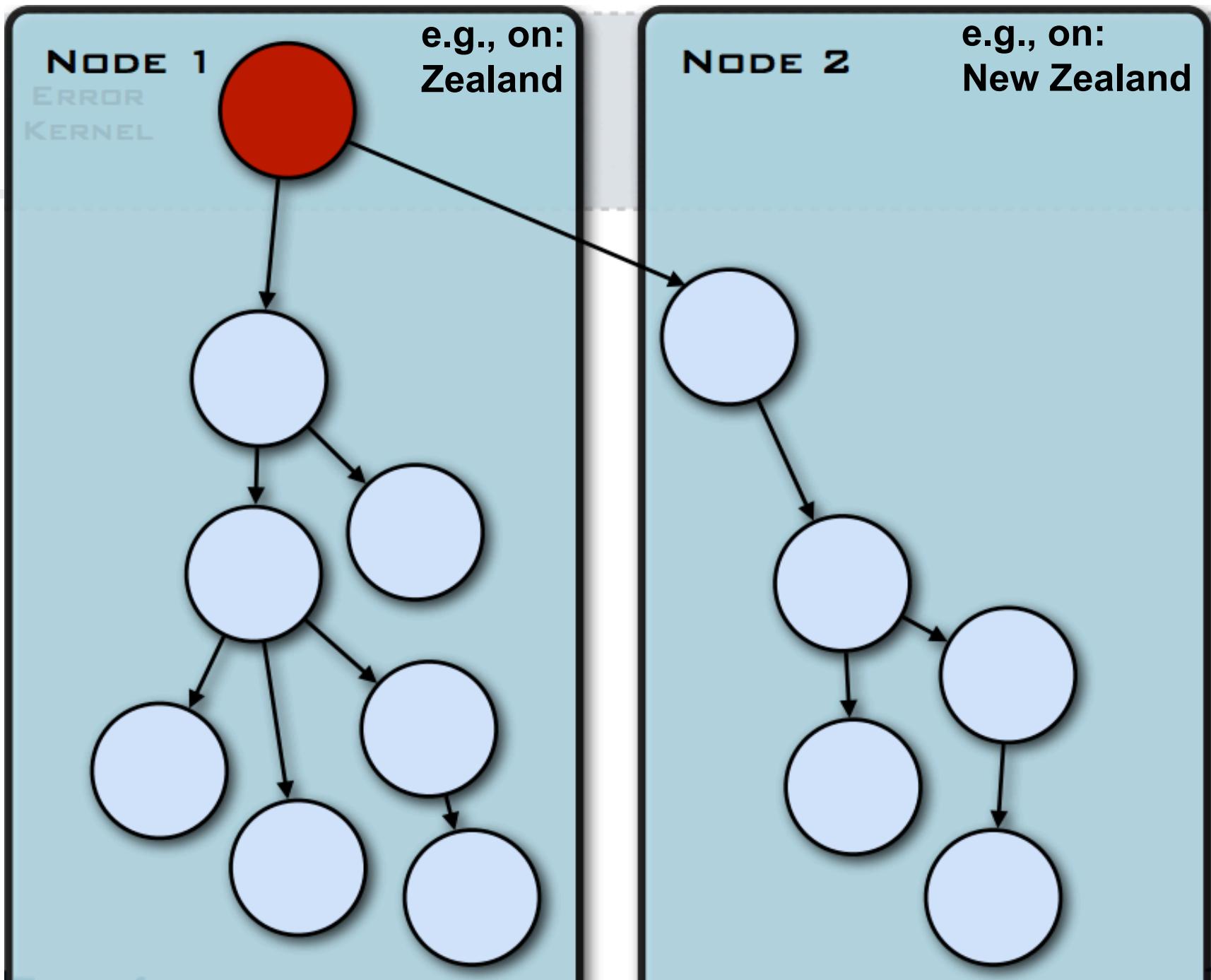
Scalability for free ("scaling up, up, and out")

Fault Tolerance (restart, resume, escalate)

A Hierarchy of Actors

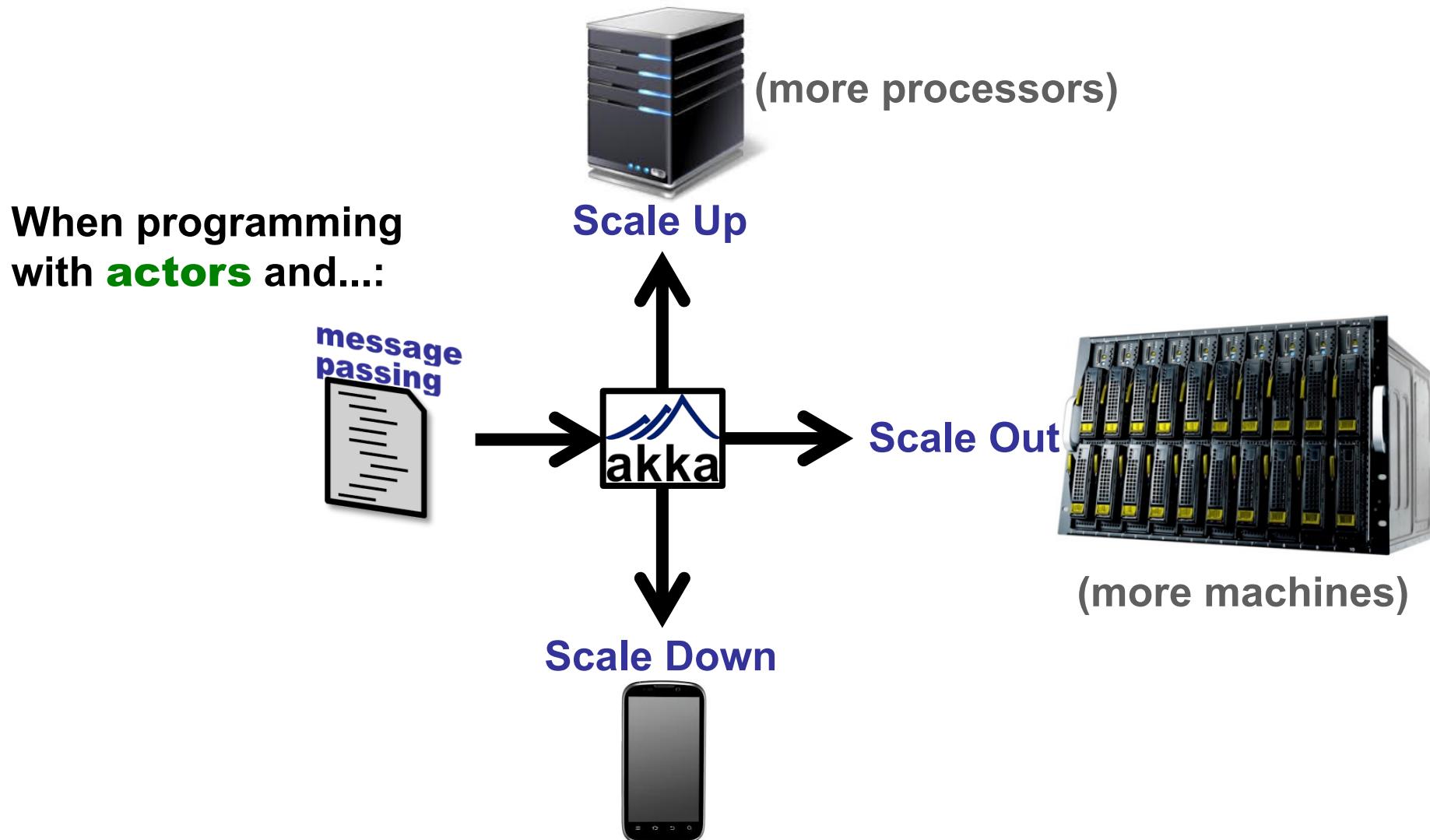


[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

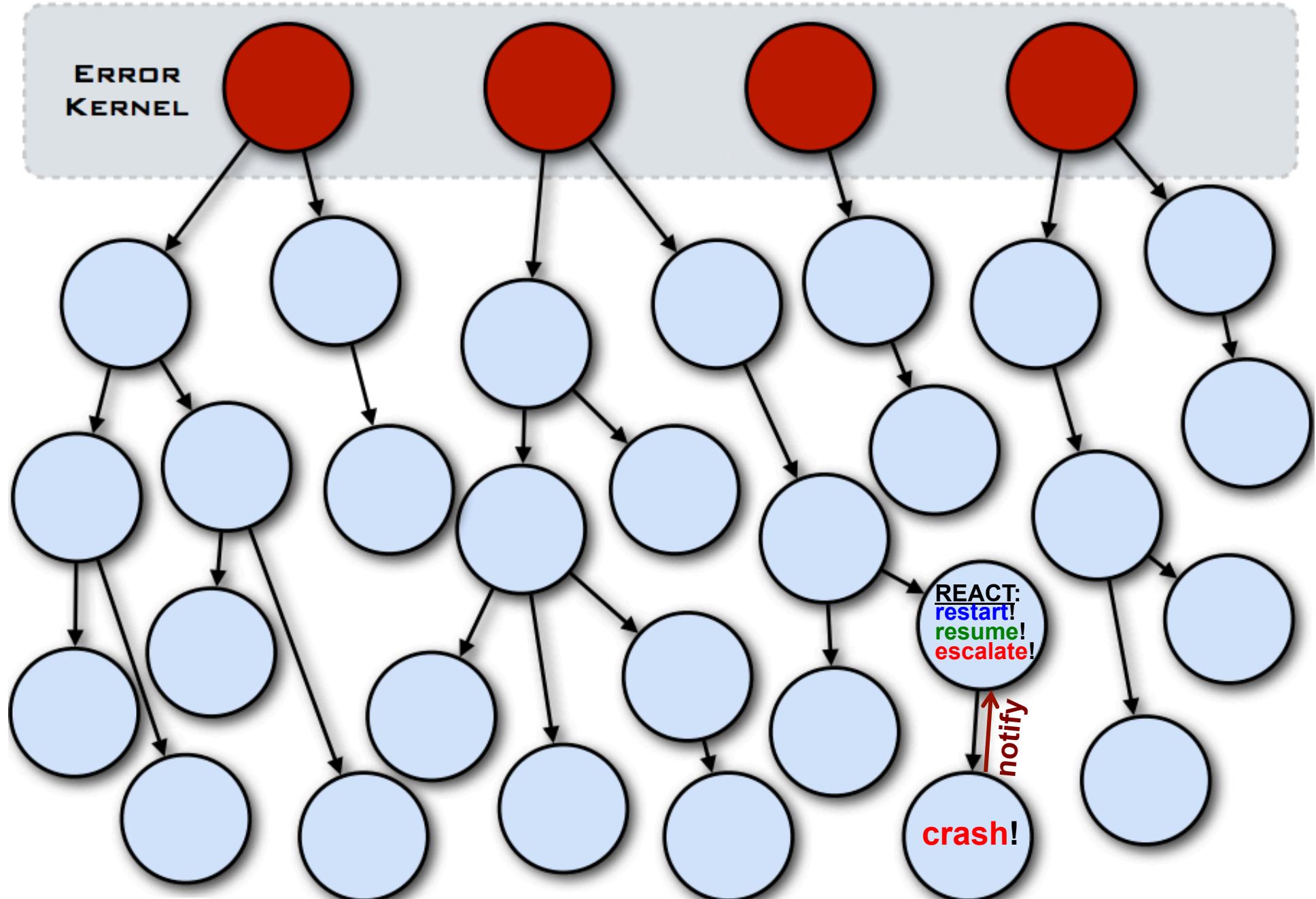
Scale Down, Scale Up, Scale Out



Fault Tolerance



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]



[From "UP UP AND OUT: SCALING SOFTWARE WITH AKKA", Jonas Boner, GOTO Conference, Aarhus, Denmark]

AGENDA

- **3) Broadcast:**

- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one ⇒ one-to-many)

- **AKKA: A proper introduction**

- Motivations and benefits of Actors & Message Passing
- Recommendations

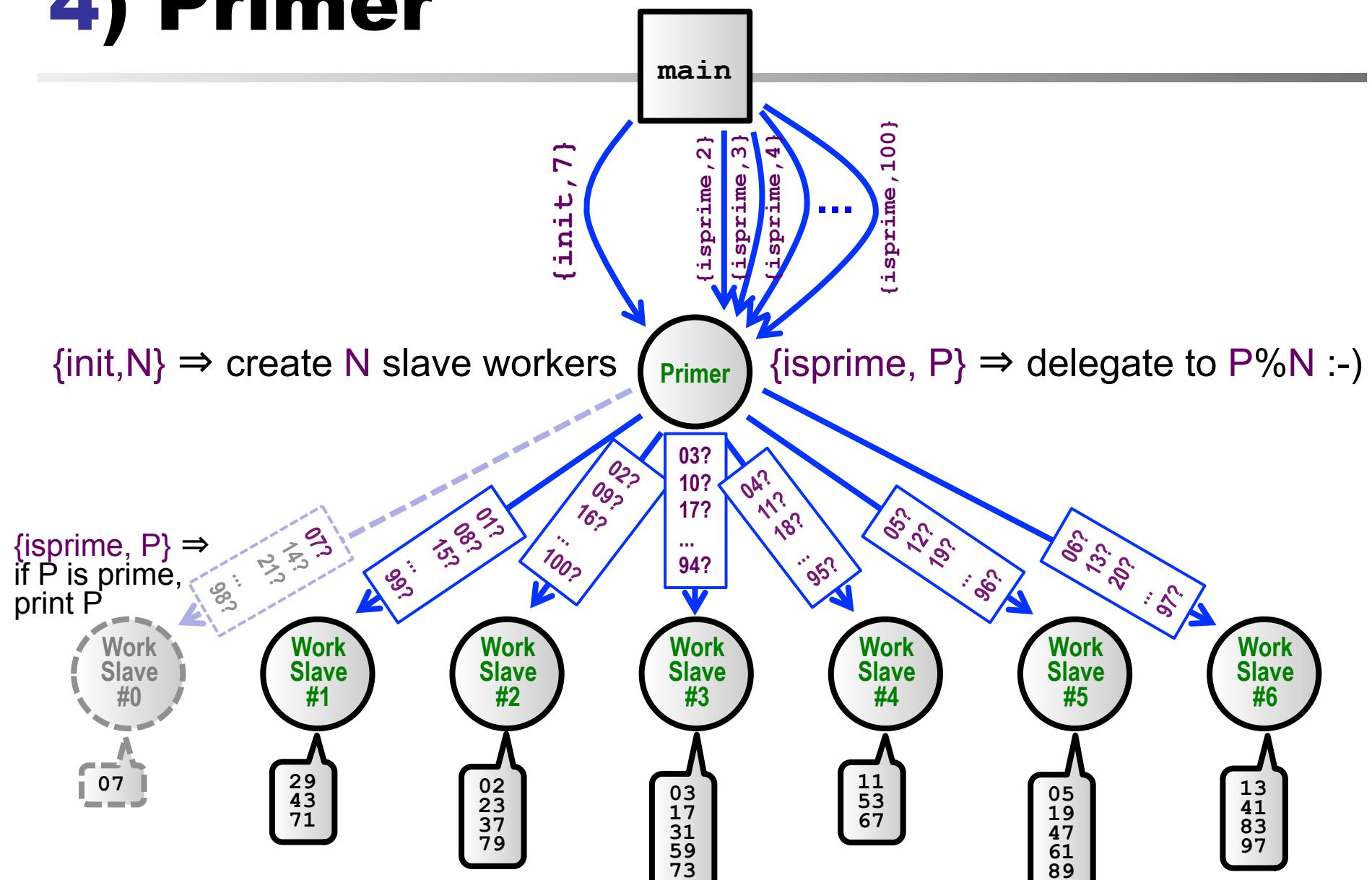
- **4) Primer:**

- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

- **★ Scatter-Gatherer:**

- Prototypical AKKA Service (dynamic load balancing)
- Extensions

4) Primer



4) Primer.erl

```

-module(helloworld).
-export([start/0, slave/1, primer/1]).

is_prime_loop(N, K) ->
    K2 = K * K, R = N rem K,
    case (K2 =< N) and (R /= 0) of
        true -> is_prime_loop(N, K+1);
        false -> K
    end.

is_prime(N) ->
    K = is_prime_loop(N, 2),
    (N >= 2) and (K*K > N).

n2s(N) ->
    lists:flatten(io_lib:format("~p", [N])).

slave(Id) ->
    receive
        {isprime, N} ->
            case is_prime(N) of
                true -> io:fwrite("#" ++
n2s(Id) ++ ":" ++ n2s(N) ++ "\n");
                false -> []
            end,
            slave(Id)
    end.

```

Slave

OUTPUT	#1:	43	#4:	67
-----	#2:	23	#5:	47
	#0:	7	#3:	17
	#1:	29	#4:	53
	#2:	2	#5:	19
	#3:	3	#6:	41
	#4:	11	#1:	71
	#5:	5	#2:	37
	#6:	13	#3:	31
			#5:	89

Primer

```

create_slaves(Max,Max) -> [];
create_slaves(Id,Max) ->
    Slave = spawn(helloworld, slave, [Id]),
    [Slave|create_slaves(Id+1,Max)].

primer(Slaves) ->
    receive
        {init, N} when N>0 ->
            primer(create_slaves(0,N));
        {init, N} ->
            throw({nonpositive,N});
        {isprime, _} when Slaves == [] ->
            throw({uninitialized});
        {isprime, N} when N=<0 ->
            throw({nonpositive,N});
        {isprime, N} ->
            SlaveId = N rem length(Slaves),
            lists:nth(SlaveId+1, Slaves)
                ! {isprime,N},
            primer(Slaves)
    end.

spam(_, Max, Max) -> true;
spam(Primer, N, Max) ->
    Primer ! {isprime, N},
    spam(Primer, N+1, Max).

start() ->
    Primer =
        spawn(helloworld, primer, [[]]),
    Primer ! {init,7},
    spam(Primer, 2, 100).

```

4) Primer.java

```
import java.util.*;
import java.io.*;
import akka.actor.*;

// -- MESSAGES ----

class InitMessage implements Serializable {
    public final int n;
    public InitMessage(int n) {
        this.n = n;
    }
}

class IsPrimeMessage implements Serializable {
    public final int n;
    public IsPrimeMessage(int n) {
        this.n = n;
    }
}
```

{init, N}

{isprime, N}



4) Primer.java

```
// -- SLAVE ACTOR ----->

class SlaveActor extends UntypedActor {
    private boolean isPrime(int n) {
        int k = 2;
        while (k * k <= n && n % k != 0) k++;
        return n >= 2 && k * k > n;
    }

    public void onReceive(Object o) throws Exception {
        if (o instanceof IsPrimeMessage) {
            int p = ((IsPrimeMessage) o).n;
            if (isPrime(p)) System.out.println("#" + p % Primer.P + ":" + p); // HACK
        }
    }
}
```

```
is_prime_loop(N,K) ->
    K2 = K * K, R = N rem K,
    case (K2 =< N) and (R /= 0) of
        true -> is_prime_loop(N, K+1);
        false -> K
    end.

is_prime(N) ->
    K = is_prime_loop(N,2),
    (N >= 2) and (K*K > N).
```

```
slave(Id) ->
    receive
        {isprime, N} ->
            case is_prime(N) of
                true -> io:fwrite("#" ++ n2s(Id) ++ ":" ++ n2s(N) ++ "\n");
                false -> []
            end,
            slave(Id)
    end.
```

4) Primer.java

```

class PrimeActor extends UntypedActor {
    List<ActorRef> slaves;

    private List<ActorRef> createSlaves(int n) {
        List<ActorRef> slaves =
            new ArrayList<ActorRef>();
        for (int i=0; i<n; i++) {
            ActorRef slave =
                getContext().actorOf(Props.create(
                    SlaveActor.class), "p" + i);
            slaves.add(slave);
        }
        return slaves;
    }

    public void onReceive(Object o) throws Exception
        if (o instanceof InitMessage) {
            int n = ((InitMessage) o).n;
            if (n<=0) throw new RuntimeException("!!! non-positive number!");
            slaves = createSlaves(n);
            System.out.println("initialized (" + n + " slaves ready to work)!");
        } else if (o instanceof IsPrimeMessage) {
            if (slaves==null) throw new RuntimeException("!!! uninitialized!");
            int n = ((IsPrimeMessage) o).n;
            if (n<=0) throw new RuntimeException("!!! non-positive number!");
            int slave_id = n % slaves.size();
            slaves.get(slave_id).tell(o, getSelf());
        }
    }
}

```

```

create_slaves(Max,Max) -> [];
create_slaves(Id,Max) ->
    Slave = spawn(helloworld,slave,[Id]),
    [Slave|create_slaves(Id+1,Max)].

```

```

primer(Slaves) ->
receive
    {init, N} when N>0 ->
        primer(create_slaves(0,N)) ;
    {init, N} ->
        throw({nonpositive,N}) ;
    {isprime, _} when Slaves == [] ->
        throw({uninitialized}) ;
    {isprime, N} when N<=0 ->
        throw({nonpositive,N}) ;
    {isprime, N} ->
        SlaveId = N rem length(Slaves),
        lists:nth(SlaveId+1, Slaves)
        ! {isprime,N},
        primer(Slaves)
end.

```

4) Primer.java

```

public class Primer {
    public static int P; // HACK
    public static int MAX = 100;

    private static void spam(ActorRef primer, int min, int max) {
        for (int i=min; i<max; i++) {
            primer.tell(new IsPrimeMessage(i), ActorRef.noSender());
        }
    }

    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("PrimerSystem");
        final ActorRef primer =
            system.actorOf(Props.create(PrimeActor.class), "primer");
        primer.tell(new InitMessage(7), ActorRef.noSender());
        try {
            System.out.println("Press return to initiate...");
            System.in.read();
            spam(primer, 2, 100);
            System.out.println("Press return to terminate...");
            System.in.read();
        } catch(IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}

```

```

spam(_, Max, Max) -> true;
spam(Primer, N, Max) ->
    Primer ! {isprime, N},
    spam(Primer, N+1, Max).

```

```

start() ->
    Primer = spawn(helloworld, primer, []),
    Primer ! {init,7},
    spam(Primer, 2, 100).

```

4) Primer.java

■ Compile:

```
javac -cp scala.jar:akka-actor.jar Primer.java
```

■ Run:

```
java -cp scala.jar:akka-actor.jar:akka-config.jar:. Primer
```

■ Output:

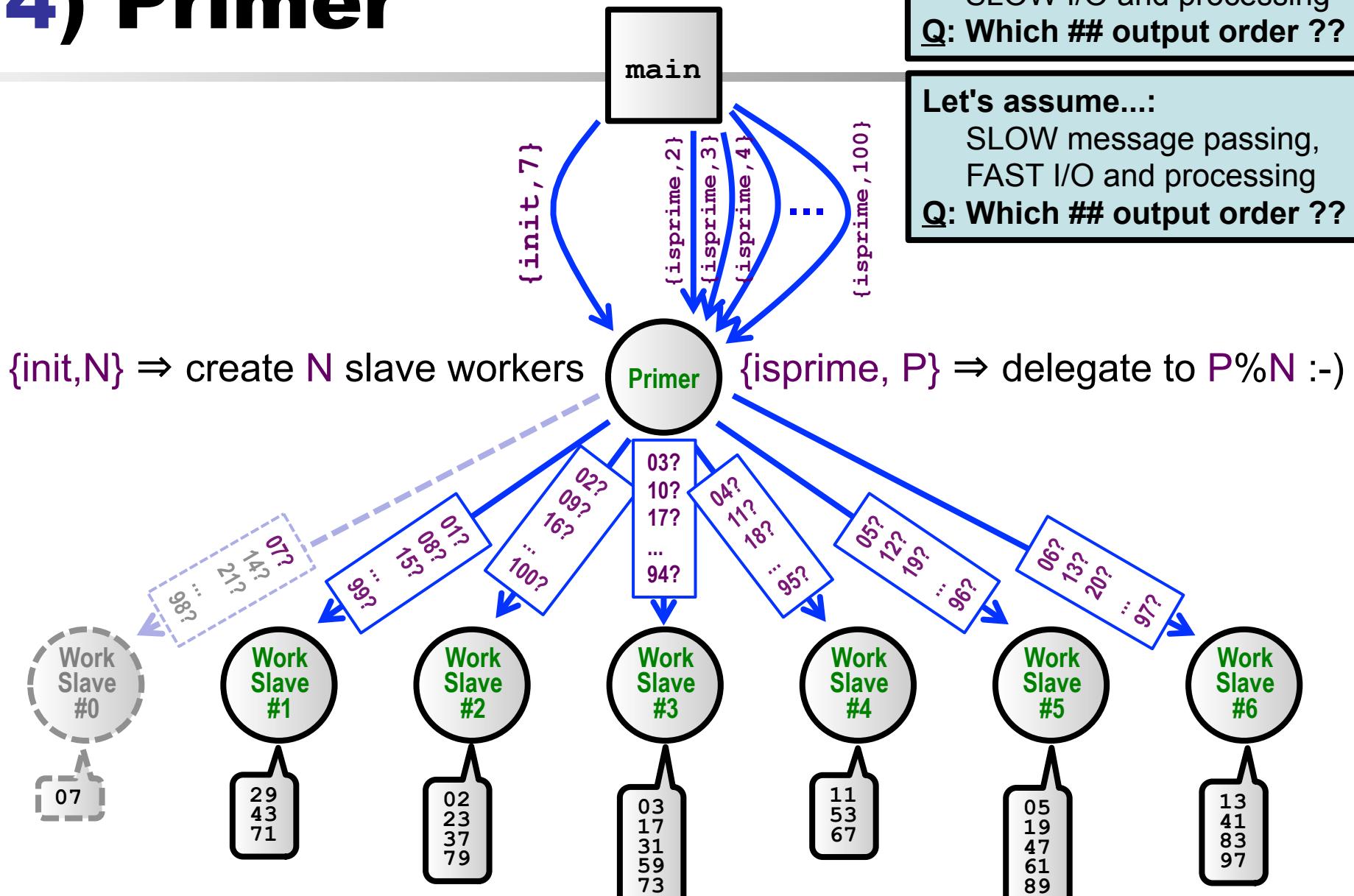
```
press return to initiate...

initialized (7 slaves ready to work)!

#2: 2
#3: 3
Press return to terminate...
#0: 7
#5: 5
#4: 11
#6: 13
#3: 17
#5: 19
#2: 23
#1: 29
```

```
#3: 31
#2: 37
#6: 41
#1: 43
#5: 47
#4: 53
#3: 59
#5: 61
#4: 67
#1: 71
#3: 73
#2: 79
#6: 83
#5: 89
#6: 97
```

4) Primer



ERLANG

-VS-

JAVA+AKKA

■ ERLANG:

SLOW input/output*

=predicted=effect=>

You would get numbers in
slave-worker order:

- 07, 29, 02, 03, 11, 05, ...

[observed in ERLANG]

#0: 07
#1: 29
#2: 02
#3: 03
#4: 11
#5: 05
#6: 13
#1: 43
#2: 23
#3: 17
#4: 53
#5: 19
#6: 41
...: ...

#2: 02
#3: 03
#5: 05
#0: 07
#4: 11
#6: 13
#3: 17
#5: 19
#2: 23
#1: 29
#3: 31
#2: 37
#6: 41
...: ...

.vs.

■ JAVA+AKKA:

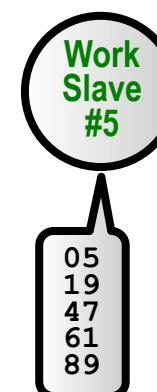
FAST input/output*

=predicted=effect=>

You would get numbers in
numerical order:

- 02, 03, 05, 07, 11, 13, ...

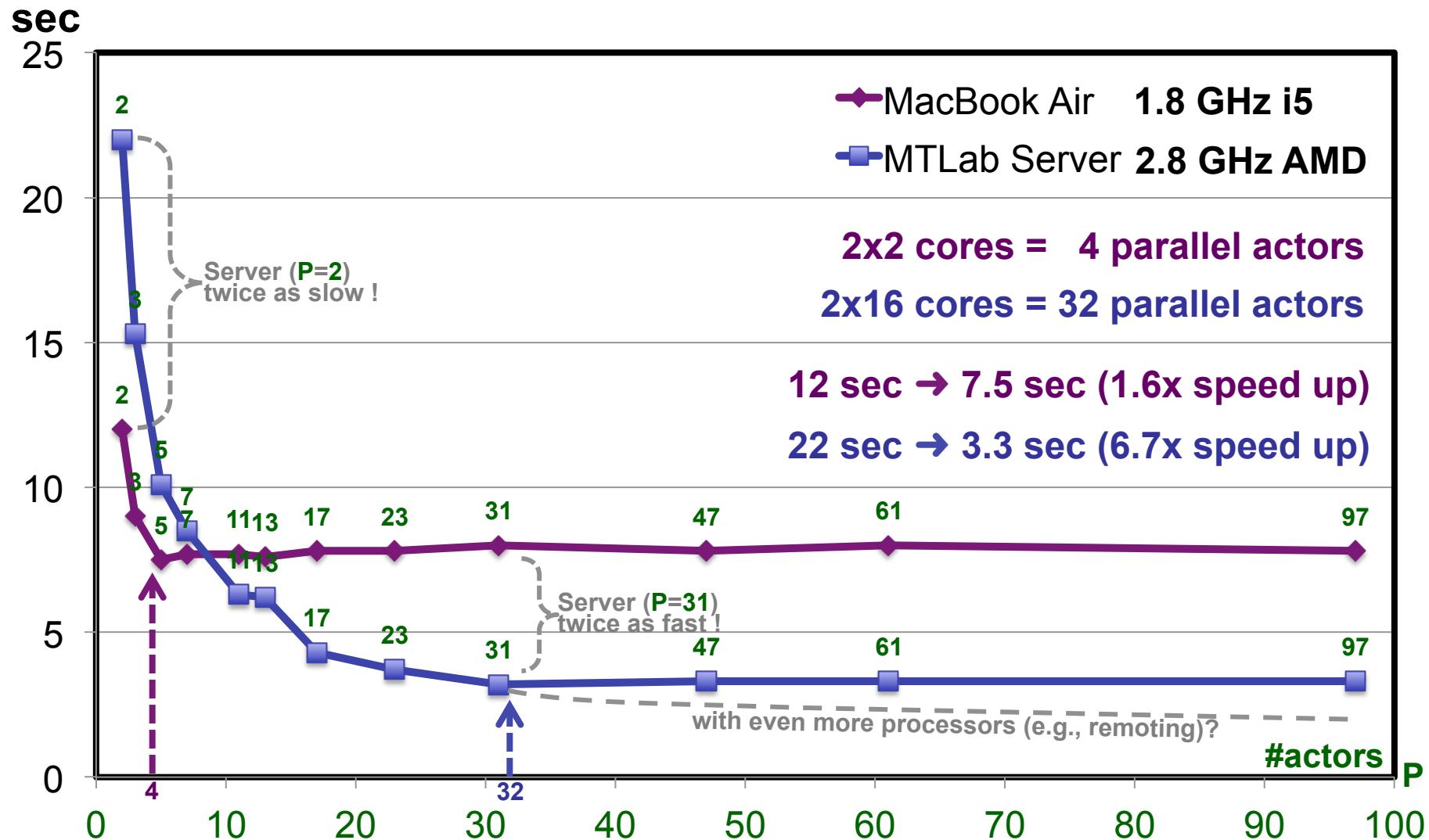
[observed in JAVA+AKKA]



*) relative to computation and passing of messages

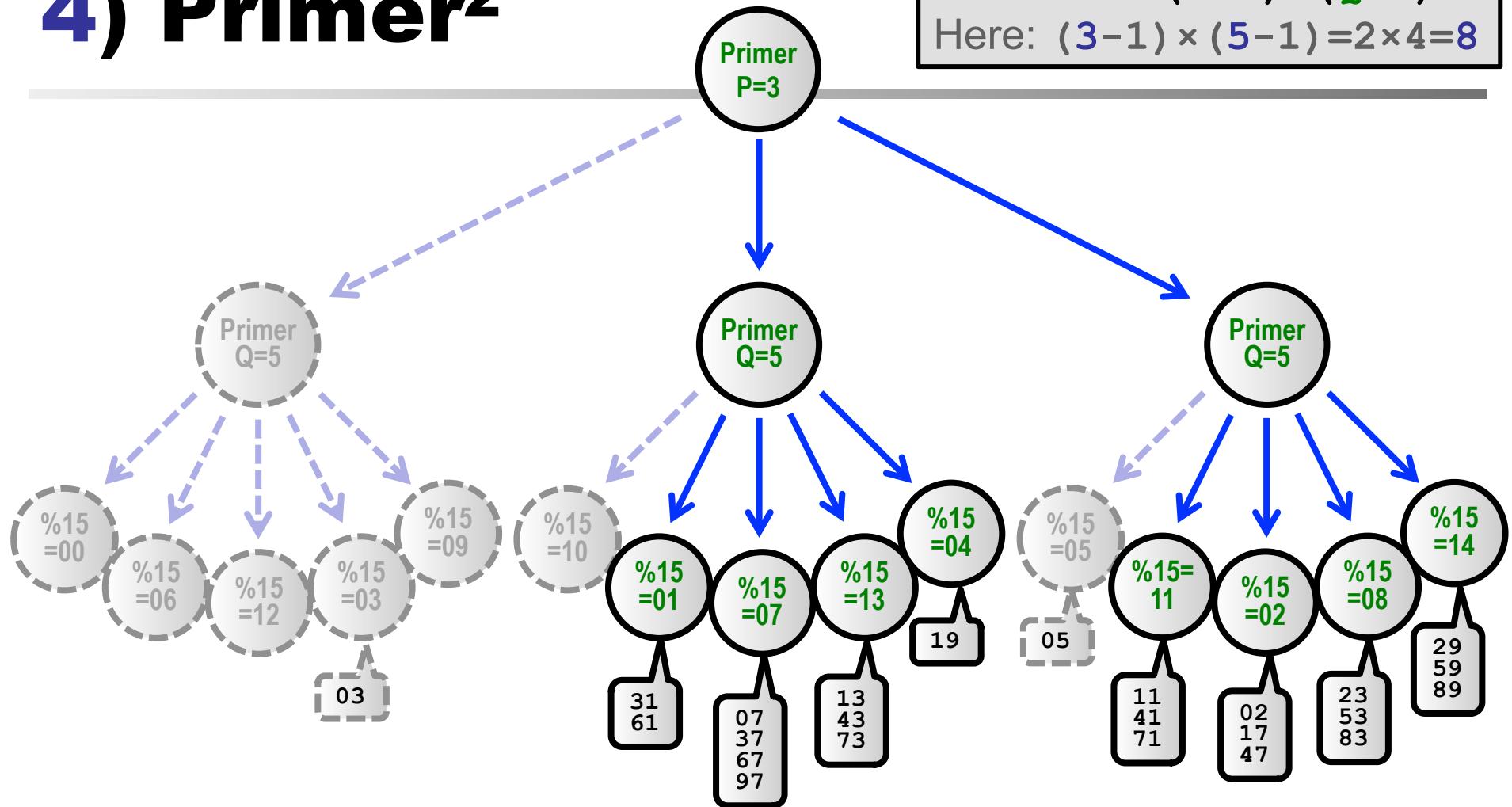
Note: added silly time-consuming computation to every `isPrime()` method call !

Low -vs- High Parallelization !



4) Primer²

#Workers: $P \times Q$
#Effective: $(P-1) \times (Q-1)$
Here: $(3-1) \times (5-1) = 2 \times 4 = 8$



Note: Two primers with $P=3 \times Q=5$ is equivalent to one primer with $P=15$.
(Note': This is why you should always use a prime # with a hash function.)

AGENDA

- **3) Broadcast:**

- From ERLANG to JAVA+AKKA
- Communication protocols (one-to-one ⇒ one-to-many)

- **AKKA: A proper introduction**

- Motivations and benefits of Actors & Message Passing
- Recommendations

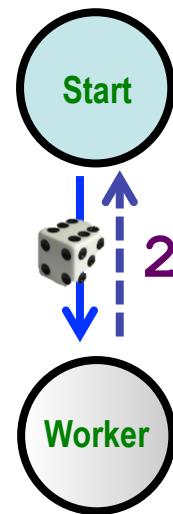
- **4) Primer:**

- Hierarchic organization: managers supervise workers
- Performance: MacBook Air -vs- MTLab Server

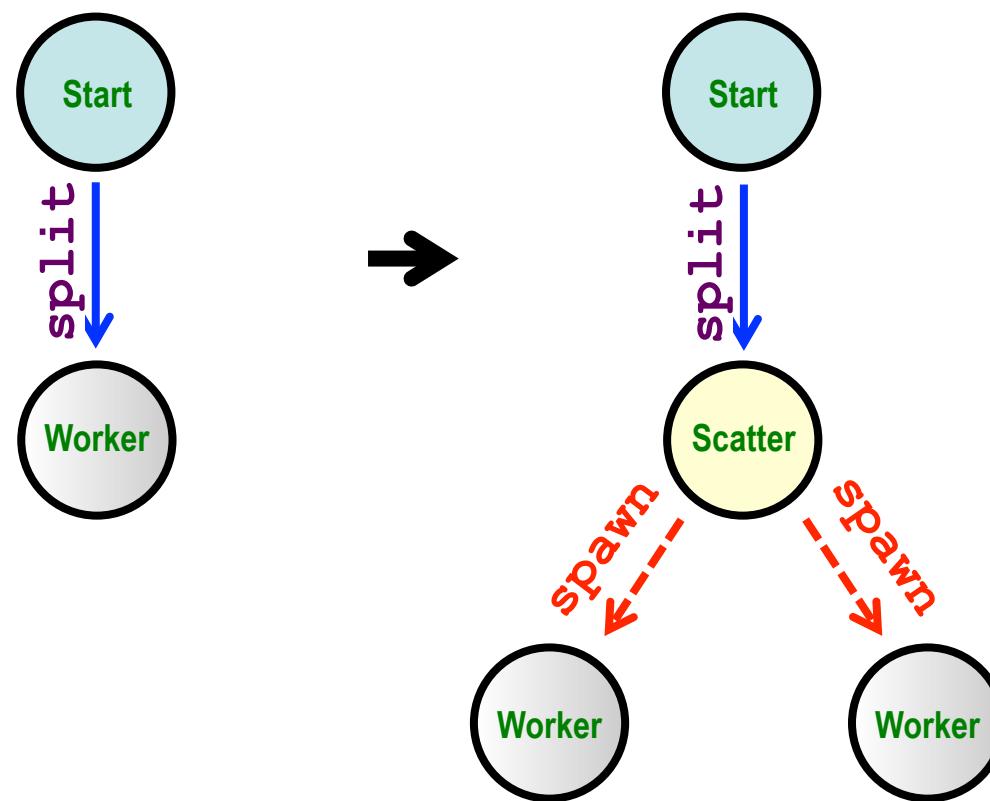
- **★ Scatter-Gatherer:**

- Prototypical AKKA Service (dynamic load balancing)
- Extensions

6) Scatter-Gather

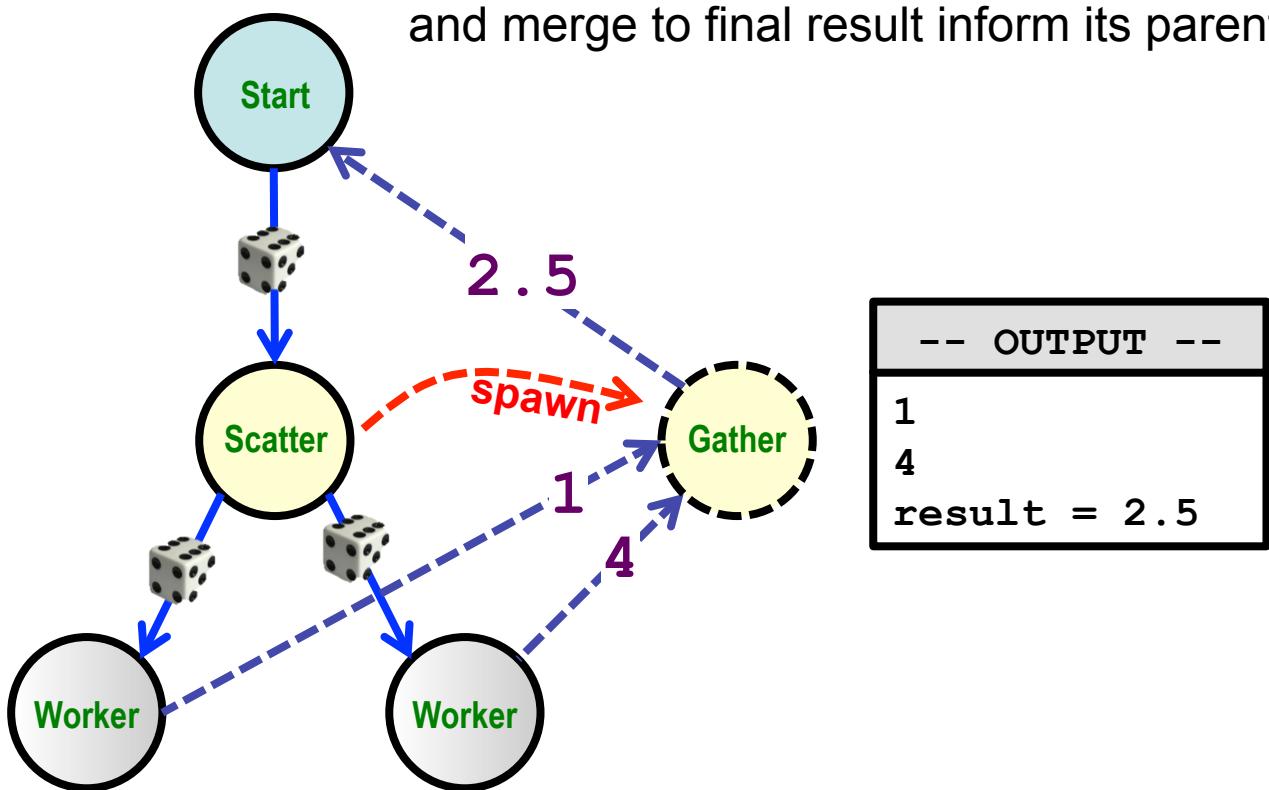


6) Scatter-Gather



6) Scatter-Gather

Gatherer:
collect incoming responses
and merge to final result inform its parent



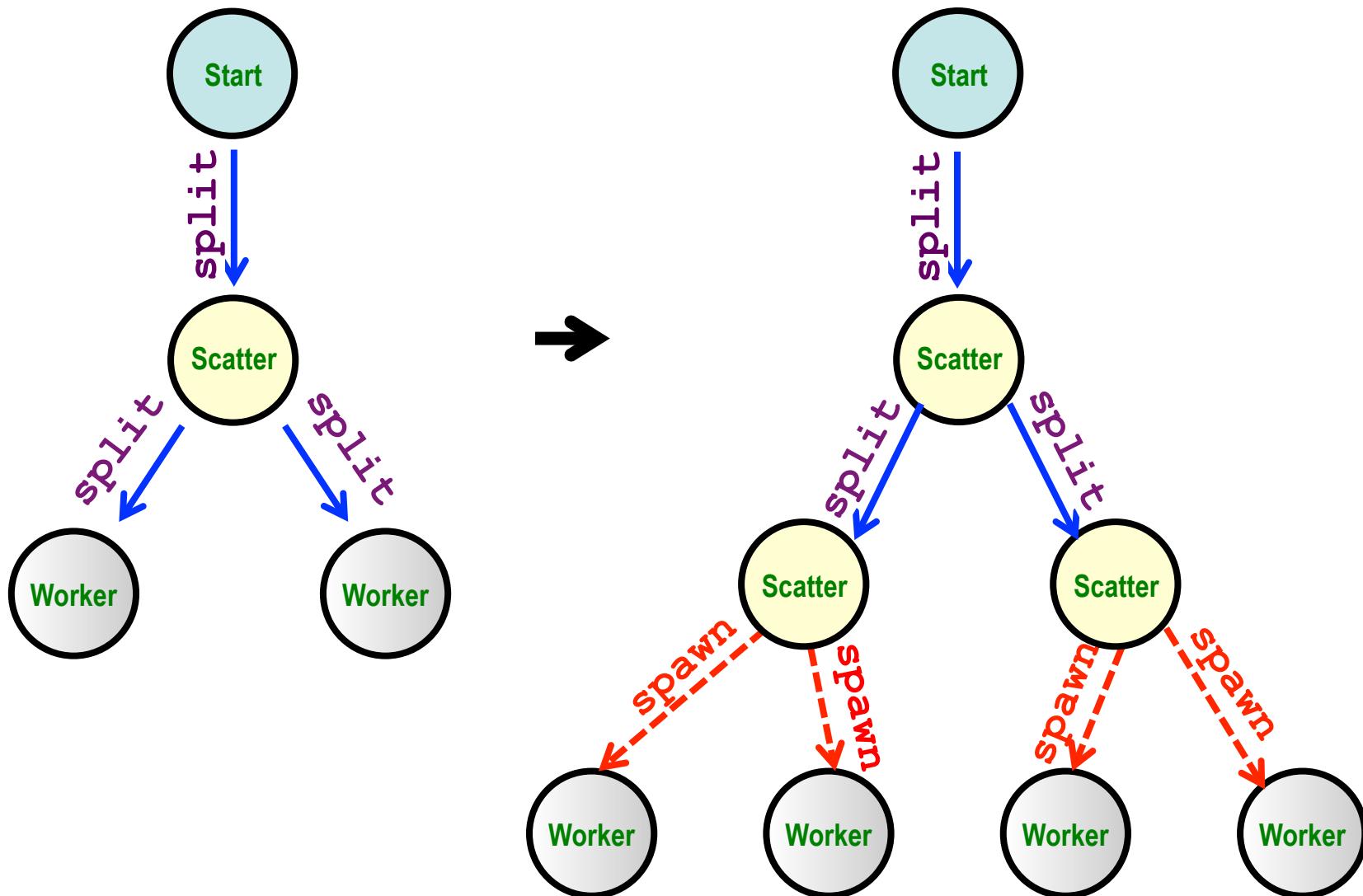
-- OUTPUT --
1
4
result = 2.5

Scatter:

I don't want the result,
send it to my gatherer:

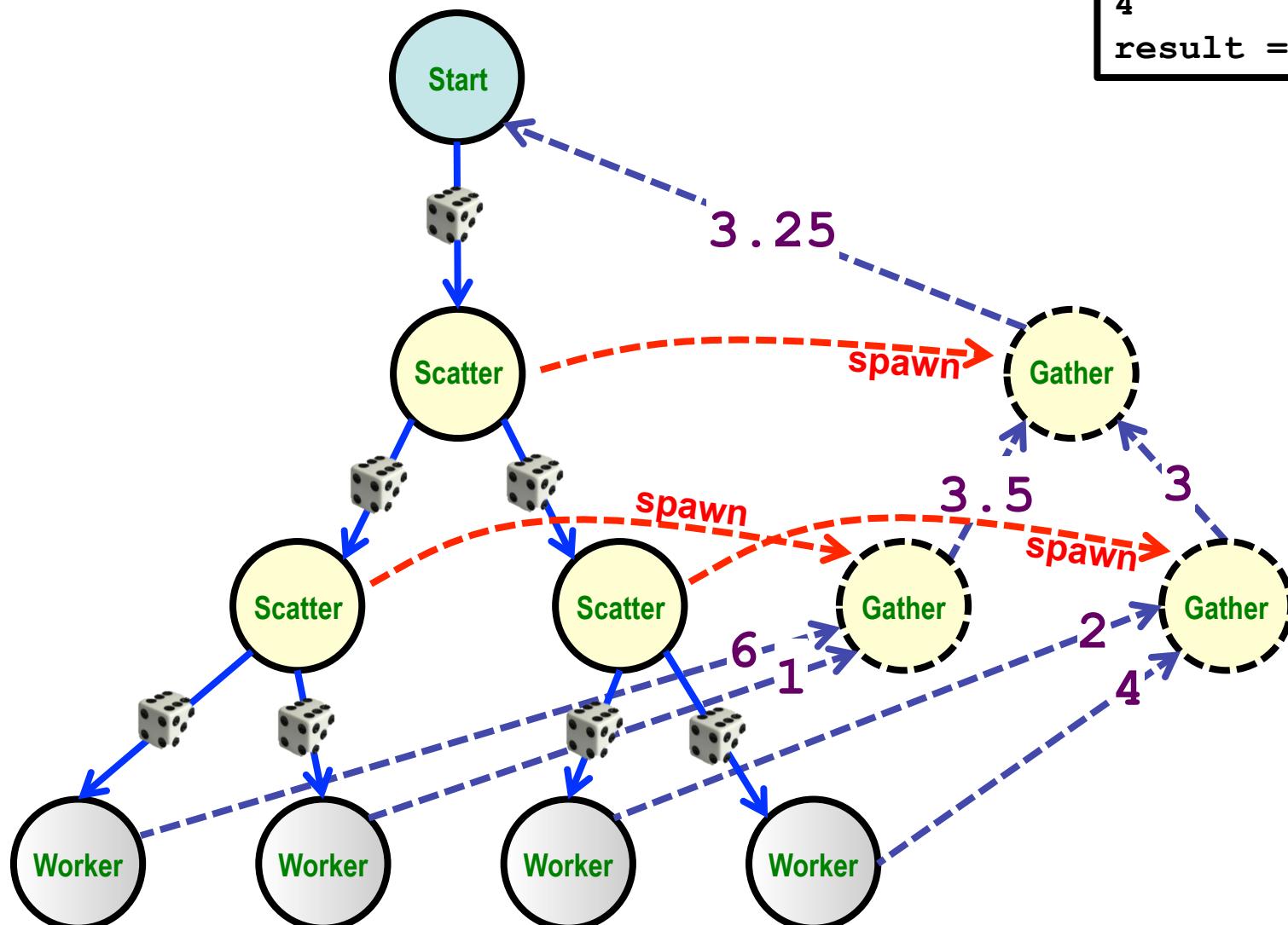
- I'm too busy processing new incoming requests
- besides, it's hard to correlate requests-responses (aka, "the correlation problem")

6) Scatter-Gather



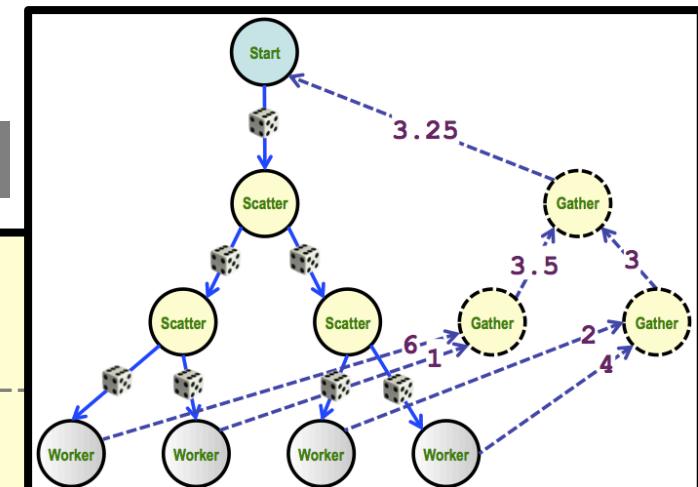
6) Scatter-Gather

```
-- OUTPUT --  
6  
1  
2  
4  
result = 3.25
```



6) ScatterGather.erl

```
-module(helloworld).
-export([start/0,worker/0,scatter/2,gather/1]).  
  
%% -- COMPUTE -----  
  
seed() -> {_, A2, A3} = now(), %% Seed wrt Time & Pid !
    random:seed(erlang:phash(node(), 100000),erlang:phash(A2, A3),A3).  
  
n2s(N) -> lists:flatten(io_lib:format("~p", [N])). %% HACK: num to string conversion!  
  
random(N) -> random:uniform(N).  
  
compute(X) -> random(X).  
  
average(X,Y) -> (X + Y) / 2.
```



```
%% -- START -----  
  
start() ->
    Worker = spawn(helloworld,worker,[]),
    Worker ! split,
    Worker ! split,
    Worker ! {compute,6,self()},
    receive
        {result,R} ->
            io:fwrite("result = " ++ n2s(R) ++ "\n")
    end.
```

-- OUTPUT --

```
6
1
2
4
result = 3.25
```

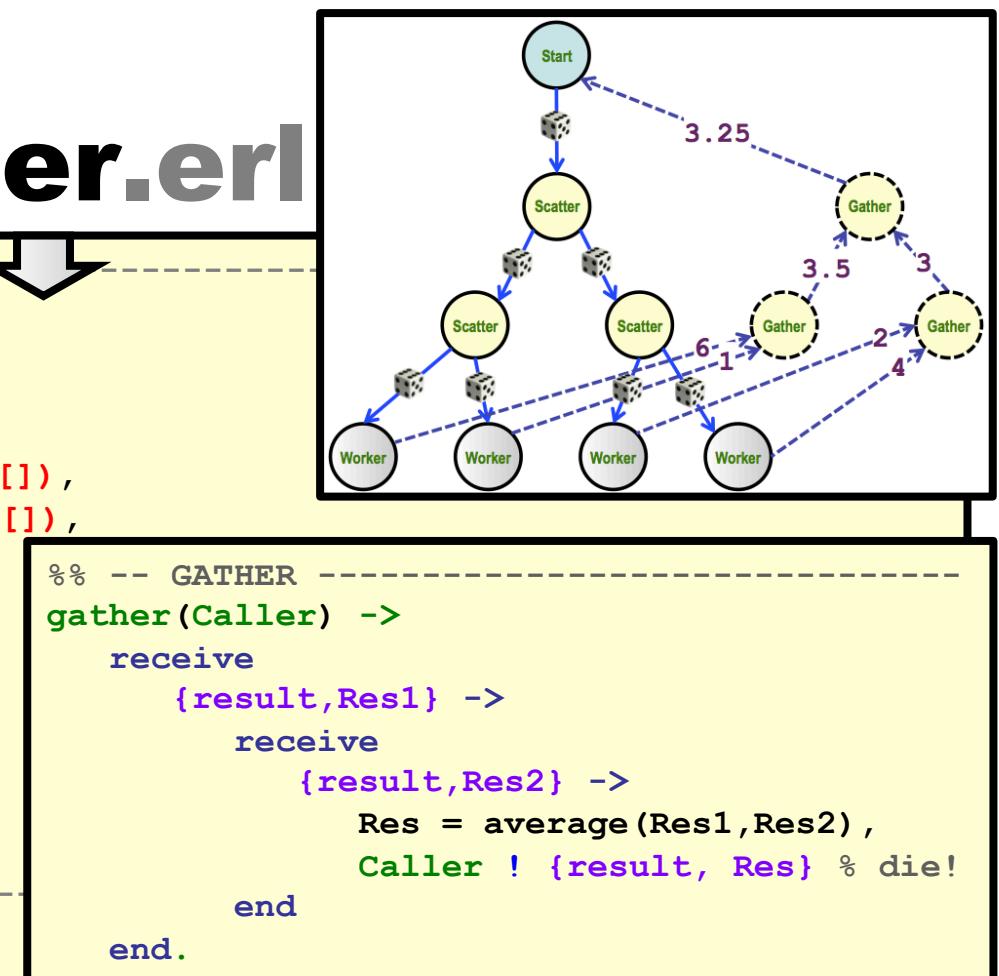
6) ScatterGather.erl

```

%% -- WORKER -----
worker() ->
    seed(),
    receive
        split ->
            Left = spawn(helloworld, worker, []),
            Right = spawn(helloworld, worker, []),
            scatter(Left, Right);
        {compute,X,Caller} ->
            Res = compute(X),
            io:fwrite(n2s(Res) ++ "\n"),
            Caller ! {result,Res},
            worker();
    end.

%% -- SCATTER -----
scatter(Left, Right) ->
    receive
        split ->
            Left ! split,
            Right ! split;
        {compute,X,Caller} ->
            Gather = spawn(helloworld, gather, [Caller]),
            Left ! {compute,X,Gather},
            Right ! {compute,X,Gather}
    end,
    scatter(Left, Right).

```



```

%% -- GATHER -----
gather(Caller) ->
    receive
        {result,Res1} ->
            receive
                {result,Res2} ->
                    Res = average(Res1,Res2),
                    Caller ! {result, Res} % die!
            end
    end.

```

-- OUTPUT --

```

6
1
2
4
result = 3.25

```

Reception (ERLANG vs AKKA)

- In ERLANG:
 - Locally nested receives (depending on local state)
- In JAVA+AKKA:
 - You only have implicit top-level receive onReceive():
- Example ⇒ refactored (ready) for JAVA+AKKA:

```
%% -- GATHER -----  
gather(Pid) ->  
    receive // State #0 ('Res1' not set)  
        {result,Res1} ->  
            receive // State #1 ('Res1' set)  
                {result,Res2} ->  
                    Res = average(Res1,Res2),  
                    Pid ! {result, Res} % die.  
                end  
            end.
```

```
%% -- GATHER' -----  
gather(Pid, Res1) ->  
    receive  
        {result,Res1} when Res1 = undef ->  
            gather(Pid, Res1)  
;  
        {result,Res2} ->  
            Res = average(Res1, Res2),  
            Pid ! {result, Res} % die.  
    end.
```

[See also ERLANG Book, program 5.3]

6) ScatterGather.java

```

import java.util.Random;      import java.io.*;
import akka.actor.*;

// -- MESSAGES --

class StartMessage implements Serializable { public StartMessage() { } }

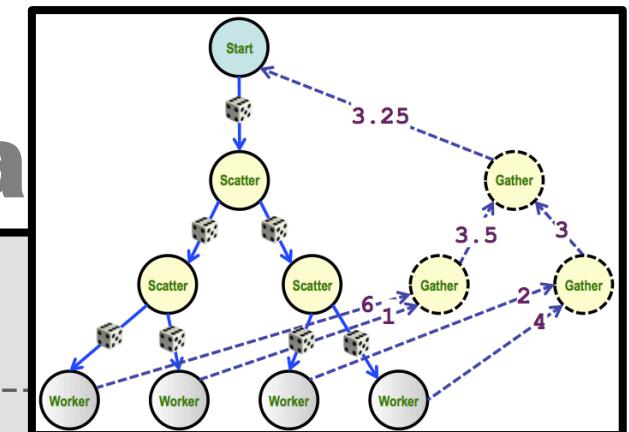
class SplitMessage implements Serializable { public SplitMessage() { } }

class CallerMessage implements Serializable {
    public final ActorRef caller;
    public CallerMessage(ActorRef caller) { this.caller = caller; }
}

class ComputeMessage implements Serializable {
    public final int number;
    public final ActorRef caller;
    public ComputeMessage(int number, ActorRef caller) {
        this.number = number;
        this.caller = caller;
    }
}

class ResultMessage implements Serializable {
    public final double result;
    public ResultMessage(double result) { this.result = result; }
}

```



-- OUTPUT --

```

6
1
2
4
result = 3.25

```

6) ScatterGather.java

```

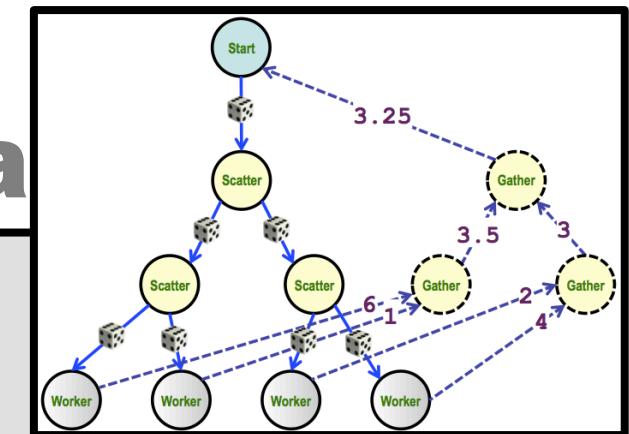
class WorkerScatterActor extends UntypedActor {
    // null => worker , non-null => scatter:
    private ActorRef left, right;
    private final Random rnd = new Random();
    private int random(int n) { return rnd.nextInt(n); }
    private int compute(int n) { return random(n) + 1; }

    public void onReceive(Object o) throws Exception {
        // dispatch according to actor role: 'worker' or 'scatter'
        if (left == null) worker(o);
        else scatter(o);
    }

    private void worker(Object o) throws Exception {
        if (o instanceof SplitMessage) {
            left = getContext().actorOf(Props.create(WorkerScatterActor.class), "left");
            right = getContext().actorOf(Props.create(WorkerScatterActor.class), "right");
        } else if (o instanceof ComputeMessage) {
            ComputeMessage m = (ComputeMessage) o;
            int result = compute(m.number);
            System.out.println(result);
            m.caller.tell(new ResultMessage(result), ActorRef.noSender);
        }
    }

    private void scatter(Object o) throws Exception { [...] }
}

```



-- OUTPUT --

```

6
1
2
4
result = 3.25

```

6) ScatterGather.java

```

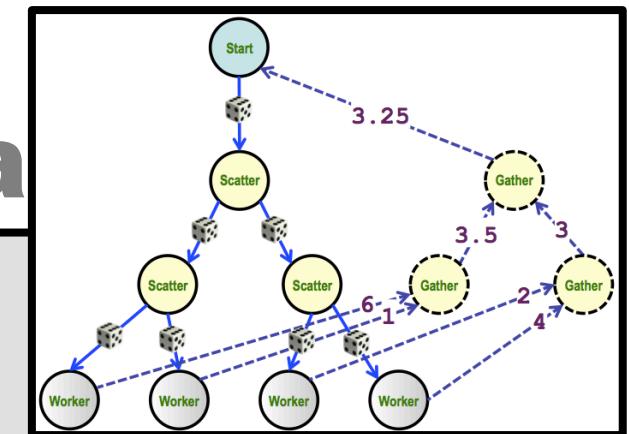
class WorkerScatterActor extends UntypedActor {
    // null => worker , non-null => scatter:
    private ActorRef left, right;
    [...]

    public void onReceive(Object o) throws Exception {
        // dispatch according to actor role: 'worker' or 'scatter'
        if (left == null) worker(o);
        else scatter(o);
    }

    private void worker(Object o) throws Exception { [...] }

    private void scatter(Object o) throws Exception {
        if (o instanceof SplitMessage) {
            left.forward(o, getContext());
            right.forward(o, getContext());
        } else if (o instanceof ComputeMessage) {
            ComputeMessage m = (ComputeMessage) o;
            ActorRef gather = getContext().actorOf(Props.create(Gather.class));
            // send message with caller, instead of arguments to gather
            gather.tell(new CallerMessage(m.caller), ActorRef.noSender());
            left.tell(new ComputeMessage(m.number, gather), ActorRef.noSender());
            right.tell(new ComputeMessage(m.number, gather), ActorRef.noSender());
        }
    }
}

```



-- OUTPUT --

```

6
1
2
4
result = 3.25

```

6) ScatterGather.java

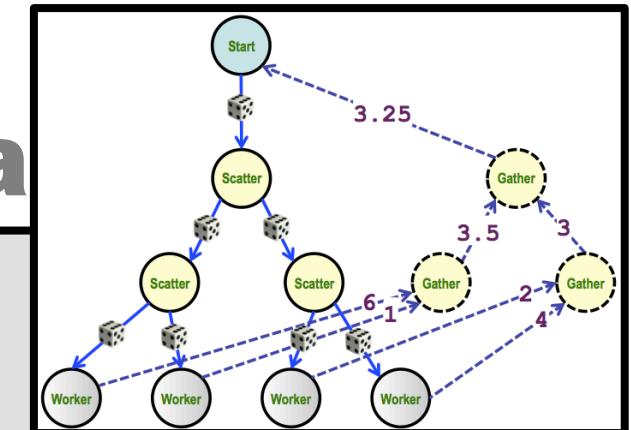
```

class GatherActor extends UntypedActor {
    double res1;
    ActorRef caller;

    private double average(double x, double y) {
        return (x + y) / 2;
    }

    public void onReceive(Object o) throws Exception {
        if (o instanceof CallerMessage) {
            caller = ((CallerMessage) o).caller;
        } else if (o instanceof ResultMessage) {
            if (caller == null) throw new Exception("no caller address!!!");
            if (res1 == 0) {
                res1 = ((ResultMessage) o).result;
            } else {
                double res2 = ((ResultMessage) o).result;
                double res = average(res1, res2);
                caller.tell(new ResultMessage(res), ActorRef.noSender());
                getContext().stop(getSelf()); // die!
            }
        }
    }
}

```



-- OUTPUT --

```

6
1
2
4
result = 3.25

```

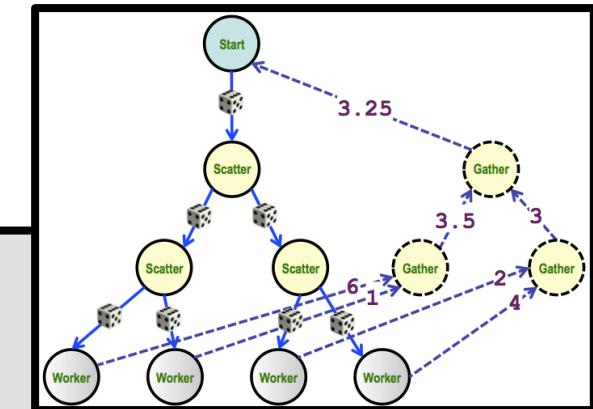
6) ScatterGather.java

```

class StartActor extends UntypedActor {
    public void onReceive(Object o) throws Exception {
        if (o instanceof StartMessage) {
            ActorRef worker =
                getContext().actorOf(Props.create(WorkerScatterActor.class), "worker");
            worker.tell(new SplitMessage(), ActorRef.noSender());
            worker.tell(new SplitMessage(), ActorRef.noSender());
            worker.tell(new ComputeMessage(6, getSelf()), ActorRef.noSender());
        } else if (o instanceof ResultMessage) {
            double result = ((ResultMessage) o).result;
            System.out.println("result = " + result);
        }
    }
}

public class ScatterGather {
    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("HelloWorldSystem");
        final ActorRef starter =
            system.actorOf(Props.create(StartActor.class), "starter");
        starter.tell(new StartMessage(), ActorRef.noSender());
        try { System.out.println("Press return to terminate..."); System.in.read(); }
        catch(IOException e) { e.printStackTrace(); }
        finally { system.shutdown(); }
    }
}

```



-- OUTPUT --	
6	
1	
2	
4	
result = 3.25	

Scatter-Gatherer + ...

- **Adaptive Load balancing:**

- Monitor system to extract up-to-date statistics
- Based on statistics, adjust system capacity (cf. our split) or Quality-of-Service (ak^2a , "graceful degradation")
 - Note: this may be done on **all** nodes in the hierarchy!

- **Memoization/Caching:**

- Often, memoization is used to "cache" already-performed-computations // `Map<Key,Val> cache;`
 - Note: this may be done on **all** nodes in the hierarchy!

- **Fault Tolerance:**

- Supervisors react if workers don't respond or crash
- Then: `resume()`, `subtree.restart()`, `parent.escalate()`

More information...

■ ERLANG:

- [<http://www.erlang.org/download/erlang-book-part1.pdf>]

■ AKKA Video Talks:

- [<https://www.youtube.com/watch?v=GBvtE61Wrto>]
- [<https://www.youtube.com/watch?v=t4KxWDqGfcs>]
 - http://gotocon.com/dl/goto-aar-2012/slides/JonasBonr_UpUpAndOutScalingSoftwareWithAkka.pdf

■ JAVA+AKKA Documentation:

- [<http://doc.akka.io/docs/akka/snapshot/java/untyped-actors.html>]
- [<http://doc.akka.io/docs/akka/2.3.7/AkkaJava.pdf>]

■ JAVA+AKKA API:

- [<http://doc.akka.io/japi/akka/2.3.7/>]

Thx!

Questions?