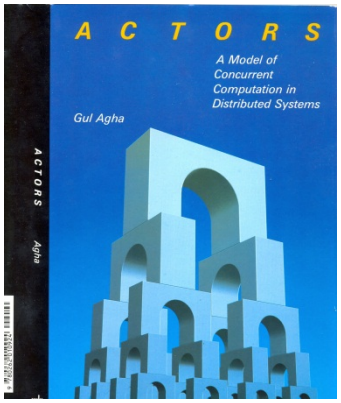
Programming Clouds for Scalability and Security: A Tutorial on Actor Languages



Gul Agha

University of Illinois at Urbana-Champaign
http://osl.cs.uiuc.edu

Acknowledgements

- Work reported here is joint with Carl Hewitt, Chris Houck, WooYoung Kim Pohao Chang, Rajesh Karmani, Mark Astley, Dan Sturman, Stas Negara, Rajendra Panwar, Svend Frolund, Reza Shiftehfar among others.
- Supported in part by AFOSR/AFRL Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement FA8750-11-2-0084 for the Assured Cloud Computing at the University of Illinois at Urbana-Champaign, and by the National Science Foundation under grant NSF CCF 14-38982.

Actor Languages and Frameworks

- Erlang
- E
- Axum
- Stackless Python
- Theron (C++)
- RevActor (Ruby)
- Dart
- …

- Scala Actors
- ActorFoundry
- SALSA
- Kilim
- Jetlang
- Actor's Guild
- Clojure
- Fan
- Jsasb
- … still growing

Example Actor Applications

- Martian Rover (1990s)
- Charm++ molecular dynamics software
- Twitter's message queuing
- LiftWeb Framework (Scala for web applications)
- Image processing in MS Visual Studio 2010
- Vendatta game engine (Erlang)
  - Battelship gallactica has 1M people playing
- Facebook Chat System (Erlang)
- LinkedIn

Examples of Actor-oriented Frameworks

- Distributed programming
  - EJBs, services in SOA
- Dataflow programming
  - Pipeline stages in stream processing (StreamIt)
  - Nodes in visual computing (Labview)
- Reactive, event-driven programming
  - Embedded computing (Ptolemy)
  - Sensor networks (ActorNet)
- Operating Systems
  - Singularity, Unix processes + pipes

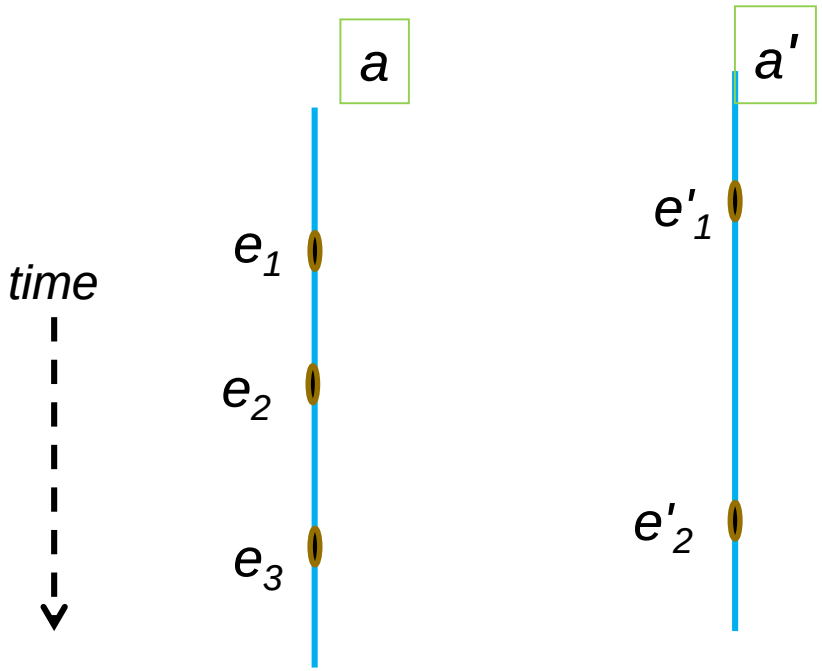
Characteristics of the Actor Model

Computation broken into autonomous, concurrent agents called *actors*:

- Actors do not share state
  - Analogous to animals in natural systems.
- Each actor operates asynchronously
  - The rate at which an actor operates may vary.
  - An actor is like a *virtual processor.*
- An actor may interact with other actors.

[Actors: A Model of Concurrent Computation in Distributed Systems," Gul Agha. MIT Press, 1986.]

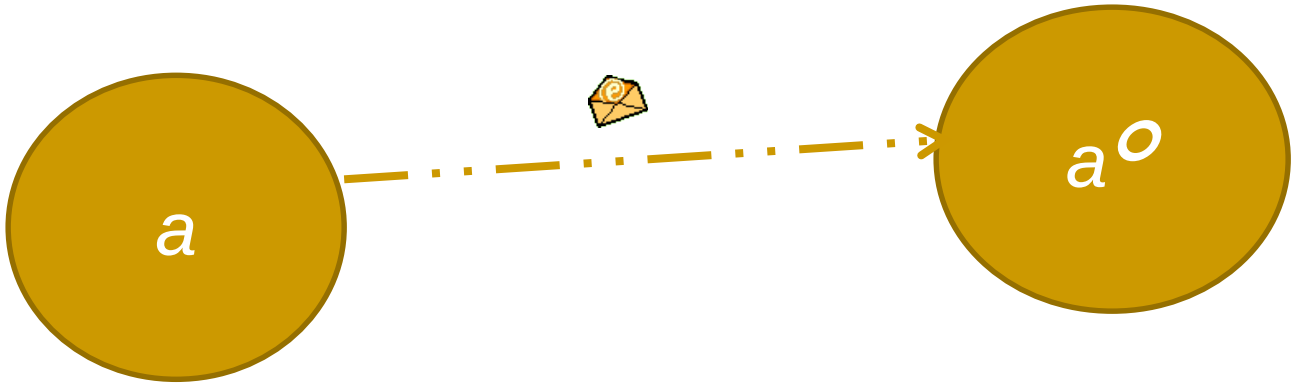
Distribution and Parallelism



- Each actors represents a point in a virtual space.
- Events at an actor are ordered linearly.
- Events may change the state of an actor

Message-Passing

- There is no action at a distance
- An actor *a* an only affect *a°* by sending it a message.
- Messages are asynchronous



Fairness

- Each actor makes progress if it can:
  - *If multiple actors execute on a single processor, each actor is scheduled.*
- Every messages is eventually delivered if it can be:
  - *When an actor is idle and has a pending message, it processes that message.*
  - *Multiple pending messages are processed in an order so none is permanently ignored by the target actor.*

Actor Names

- The *name (mail address)* of each actor is unique.
- The *name (mail address)* of an actor cannot be guessed, ergo:
  - An actor must know the *name (mail address)* of the target actor to send it a message
- Called the *locality property* of actors.
- Locality property provides a built in *capability architecture* for security.

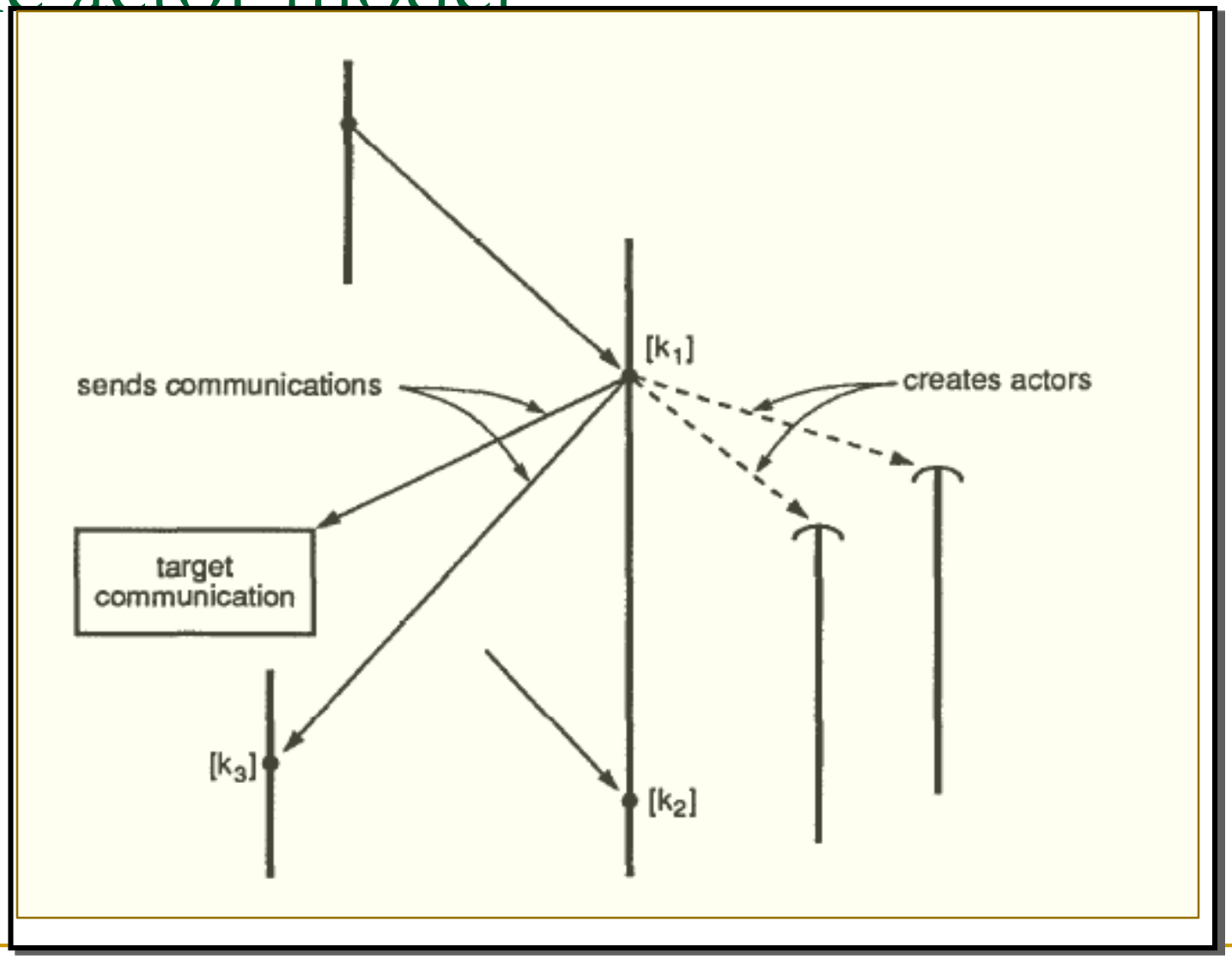
Actor Topology

- If an actor a_1 knows the address of another actor a_2 , a_1 may communicate the name of an a_2 in a message.
- Thus the interconnection topology of actors is dynamic.
- Supports mobility and reconfiguration of actors.

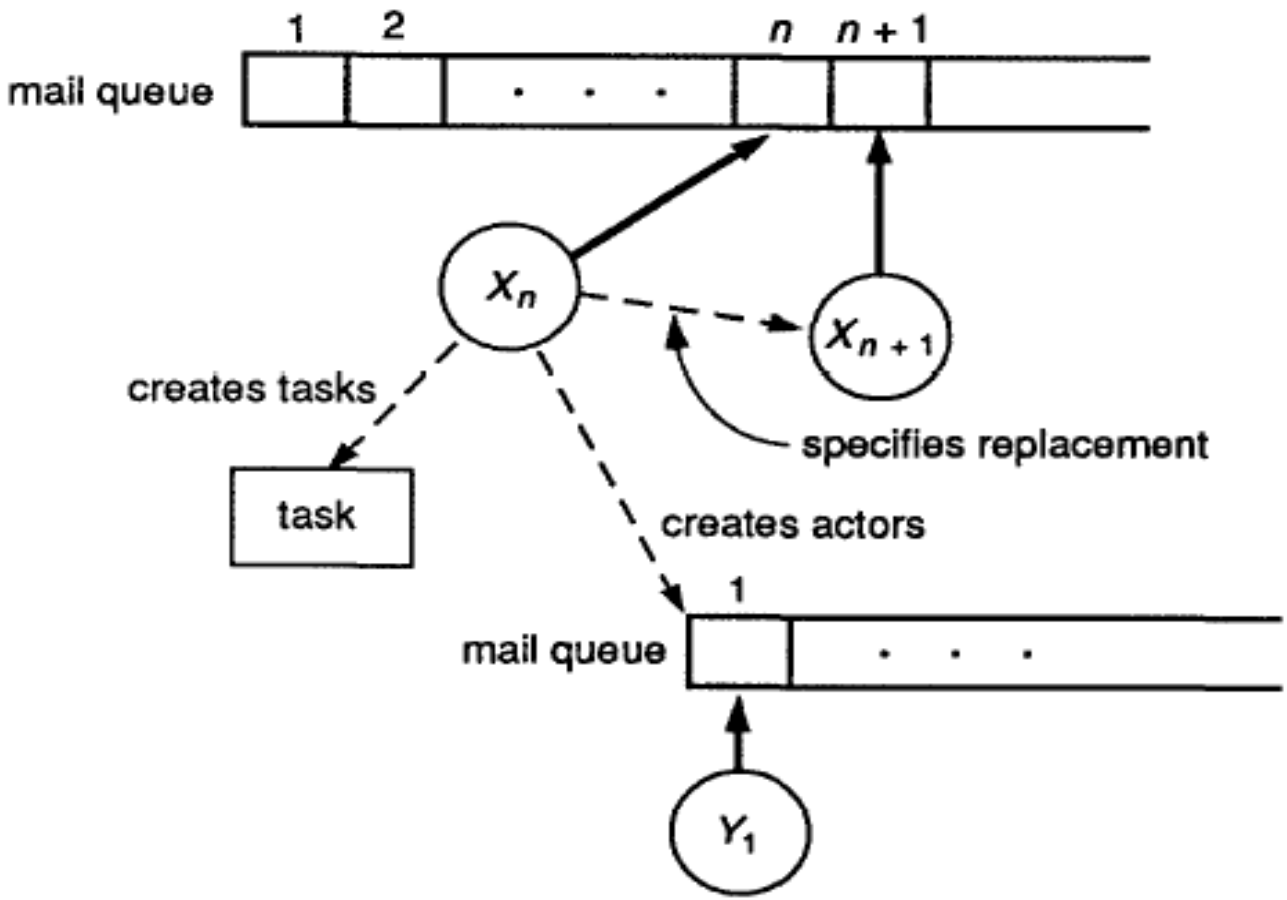
Actor Creation

- Unlike some *process models*, new actors may be created:
  - Increases the available concurrency in a computation.
  - Facilitates dynamic parallelism for load balancing.
  - Enables mechanisms for fault-tolerance.

The actor model



The actor model with buffer queues



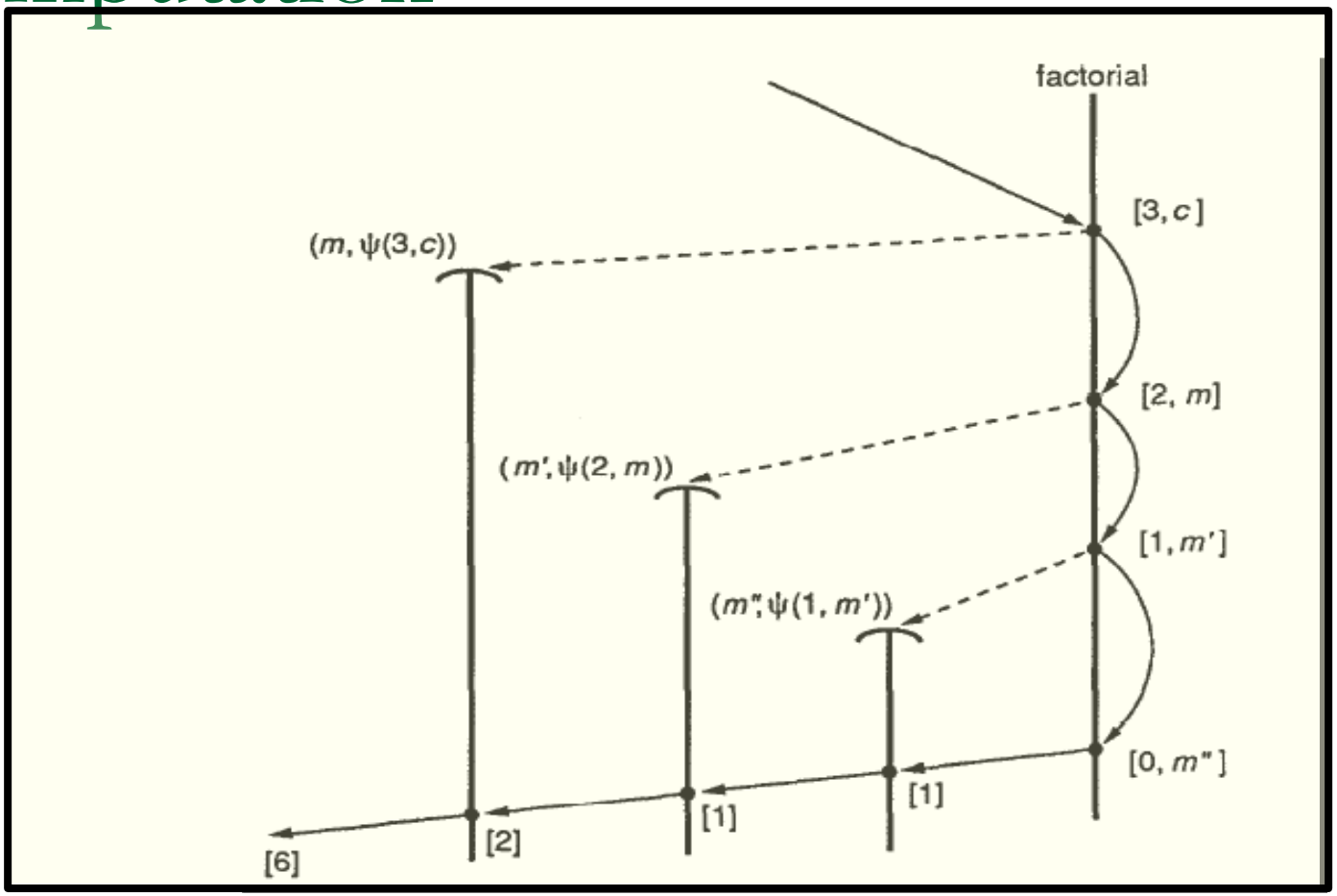
Example: An actor implementation of sequential factorial

- In response to a communication with a non-zero integer *n*, the actor with the 'factorial' behavior:
- Creates an actor whose behavior will be to multiply *n* with an integer it receives and send the reply to the mail address to which the factorial of *n* was to be sent.
- Sends itself the "request" to evaluate the factorial of $n - 1$ and send the value to the customer it created.

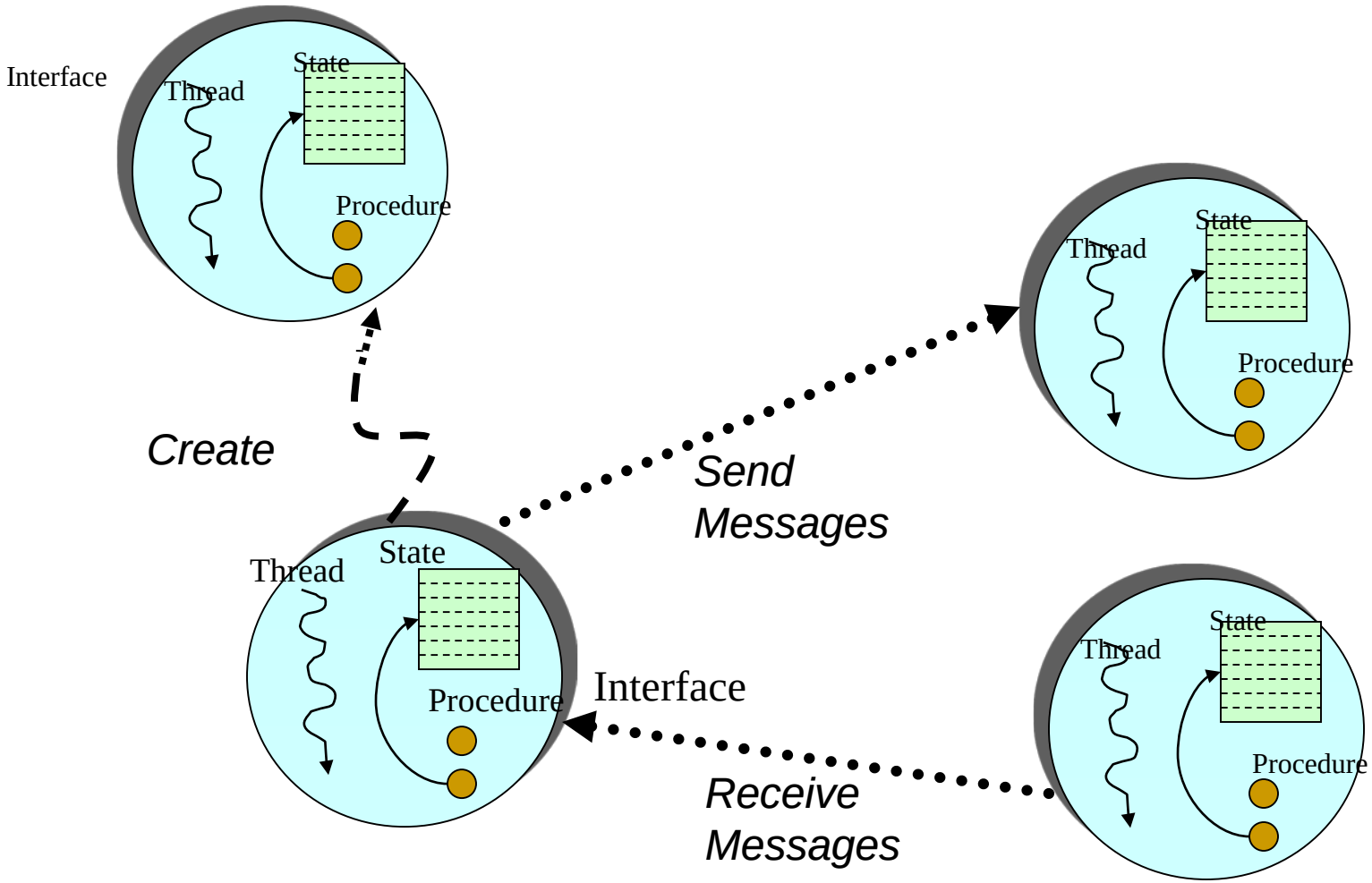
Example: An actor implementation of sequential factorial

- create a customer which waits for the appropriate communication, in this case from the factorial actor itself.
- The factorial actor is free to concurrently process the next communication.
- We assume that a communication to a factorial includes a mail address to which the value of the factorial is to be sent.

Event diagram for a factorial computation



The Actor Model



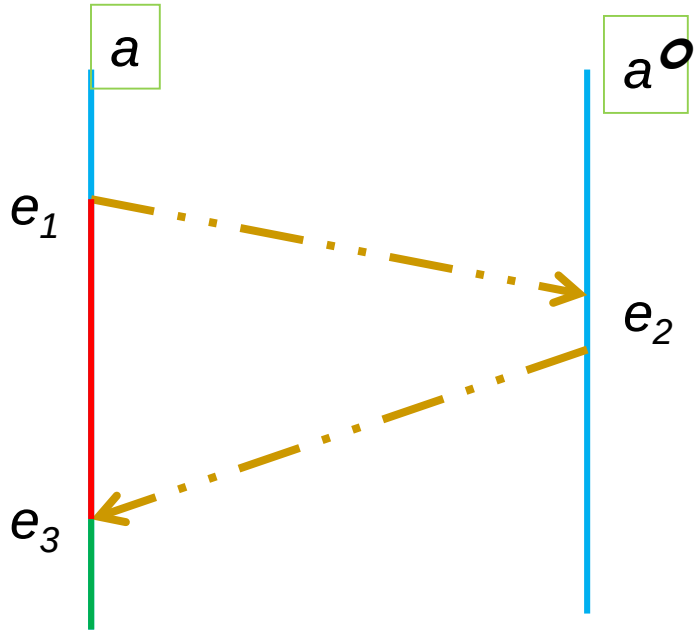
Defining an actor language

Start with a sequential object-based language or framework, add concurrency to objects, operators for:

- actor creation
  - create(node, class, params)
  - *Locally or at remote nodes*
- message sending
  - send(actor, method, params)
- State change
  - Become idle (ready to process next message)

Message Patterns

- More complex message patterns may be defined in terms of asynchronous messages:



rpc like messaging

Actor Event Diagrams

Standard Actor Semantics

- Encapsulation (no shared, concurrent access to state)
- Fairness
- Location Transparency
- Mobility

Actor Encapsulation: State Isolation

- Recall: *no shared state* between actors
- 'Access' another actor's state *only* by sending it a message and requesting it:
  - Messages have send-by-value semantics
  - Implementation may be relaxed on shared memory platforms, if "safe"

Why State Encapsulation?

- Reasoning about safety properties becomes "easier"
- JVM memory safety is not sufficient for Actor semantics.
- Languages such as Erlang and some actor frameworks (SALSA, ActorFoundry) enforce encapsulation.
- Frameworks such as Scala and Kilim impose conventions instead of enforce isolation
  - Easier to introduce inadvertant state sharing

Fairness

- *Permanently Disabled Actor:* An actor executing an infinite loop, blocked on a external call..
- *Enabled Actor:*
  - not permanently disabled, and
  - has a pending message
- *Actor Scheduling fairness:* An enabled actor is eventually scheduled
- *Fair Message Delivery*: a message is eventually delivered if the recipient is not permanently disabled.

Why Fairness?

- Required for proving liveness:
  - An actor in an infinite loop (or stuck) could starve enabled actors.
  - Cannot guarantee a result is eventually produced.
- Fairness specially critical in libraries for existing languages
  - Interaction with existing code-base, plug-ins, 3rd party components

Why Location Transparent Naming?

- Enables *automatic* load-balancing and fault-tolerance mechanisms
  - Run-time can exploit resources available on cluster, grid or scalable multicores (distributed memory)
- Uniform model for multicore and distributed programming

Actors Reduce Concurrency Bugs

For example:

- Macro-step semantics
  - No bugs caused by interleaving between steps of different actors
  - Granularity of abstraction is increased
- No deadlocks caused by synchronous messaging

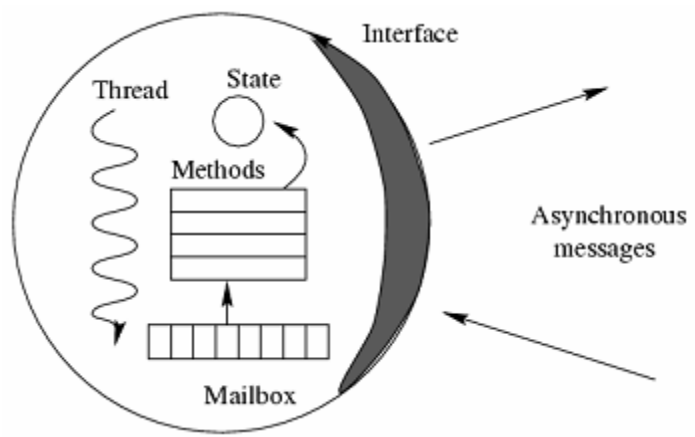
Actor programs can have bugs…

- ‘Old fashioned’ bugs in sequential code
- Inconsistent state updates between actors
- Livelocks due to *rpc*-like communication

Actor anatomy

Actors = ~~encapsulated state + behavior +~~
independent control + mailbox

Object



ActorFoundry

- Off-the-web library for Actor programming
- Major goals: Usability and Extensibility
- Other goals: Reasonable performance
- Supports standard Actor semantics

ActorFoundry – Programming Model

- Actors are like objects
  - Implicit 'receive'
  - Run-time provides fetch-decode loop
- One-to-one correspondence between message and Java method
- Primitives for asynchronous (`send`) as well as synchronous, RPC-like (`call`) messages
- Wraps standard IO objects as actors (`stdout`, `stdin`, `stderr` actors)

Implicit Control (ActorFoundry)

```
public class HelloWorld extends Actor {

  @message public void boot() throws RemoteCodeException {
    ActorName a2 = create(HelloWorld.class);
    send(a2, "relayGreeting", "Hello ");
    send(a2, "relayAudience", "Sukkur!");
  }

  @message public void relayGreeting(String item) {
    send(stdout, "print",  item);
  }

  @message public void relayAudience(String item) {
    send(stdout, "println",  item);
  }
}
```

Base class for Actors

Location-independent address

Messages as methods

Annotation for message declaration

Asynchronous message

Explicit control & mailbox (Kilim)

```
public class SimpleTask extends Task {
    static Mailbox<String> mb = new Mailbox<String>();

    public static void main(String[] args) throws Exception {
        Task t = new SimpleTask().start();

        mb.putnb("Hello ");
        mb.putnb("World\n");
        mb.putnb("done");
    }

    @pausable
    public void execute() {
        while (true) {
            String s = mb.get();
            if (s.equals("done")) break;
            System.out.print(s);
        }
    }
}
```

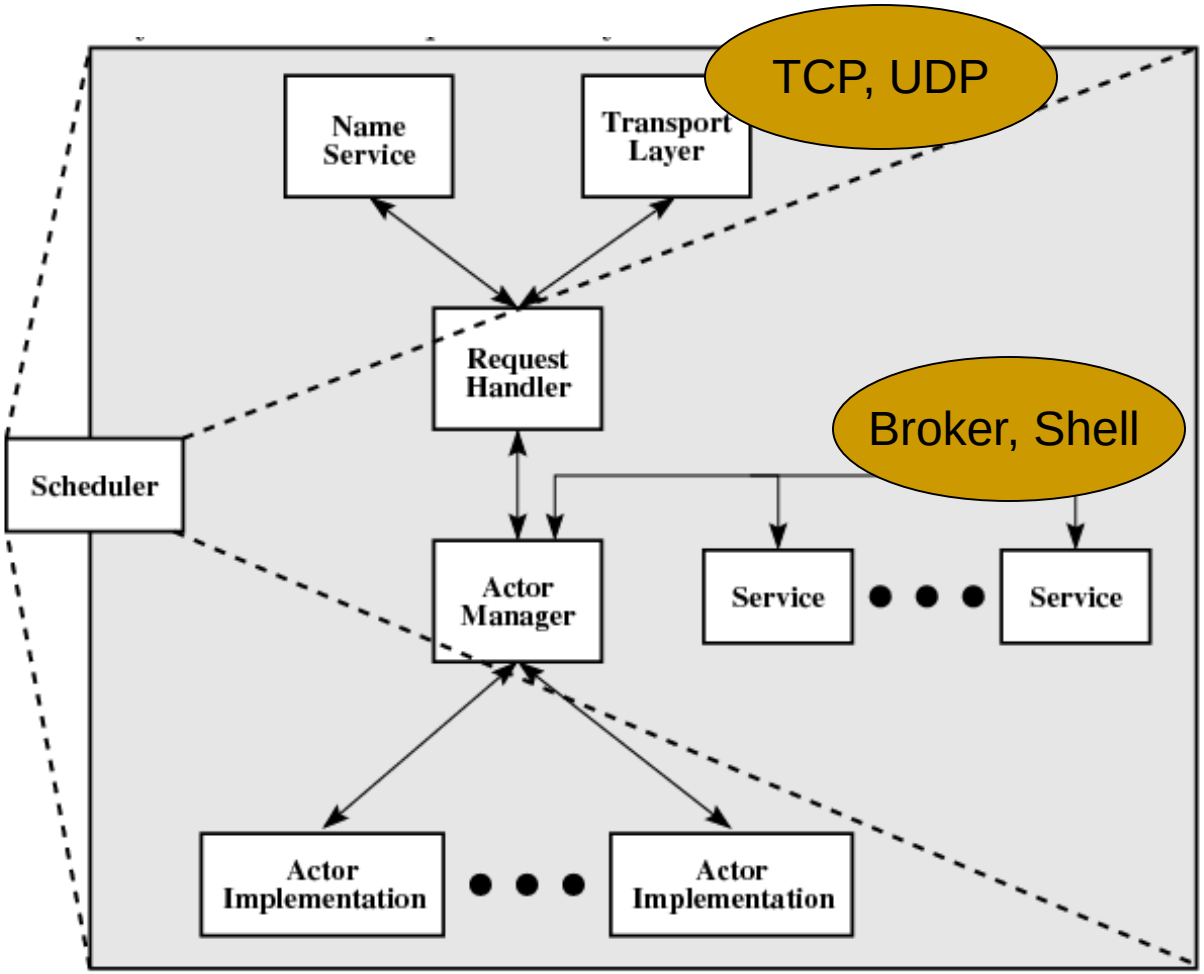
Explicit control (Scala)

```
object helloworld {
    class HelloActor() extends Actor {

        def act() {
            loop { react {
                case ("hello") => {
                    print("Hello World!")
                }
                case ("other") => {
                    System.exit(0)
                }
                }}}
    }

    def main(args : Array[String]) : Unit = {
            var hello = new HelloActor()
            hello.start
            hello ! "hello"
    }
}
```

ActorFoundry - Implementation

- Maps each actor onto a Java thread
- Actor = Java Object + Java Thread + Mailbox + ActorName
  - ActorName provides encapsulation as well as location transparency
- Message contents are deep copied
- Fairness: Reliable delivery (but unordered messages) and fair scheduling
- Support for distribution and actor migration

ActorFoundry – Runtime Library



Comparison of Semantics

	Scala Actors	Kilim	JavAct	Jetlang	SALSA	AA	AF
State Encapsulation	No	No	Yes	Yes	Yes	Yes	Yes
Safe Messaging	No	No	No	No	Yes	Yes	Yes
Fair Scheduling	Yes	No	No	No	Yes	Yes	Yes
Location Transparency	No	No	Yes	Yes	Yes	Yes	Yes
Mobility	No	No	Yes	No	Yes	Yes	Yes

ActorFoundry Benefits

- Extensibility
  - Modular and hence extensible
- Usability
  - Actors as objects + small set of library calls
  - Leverage Java libraries and expertise
- Performance
  - Let's check…

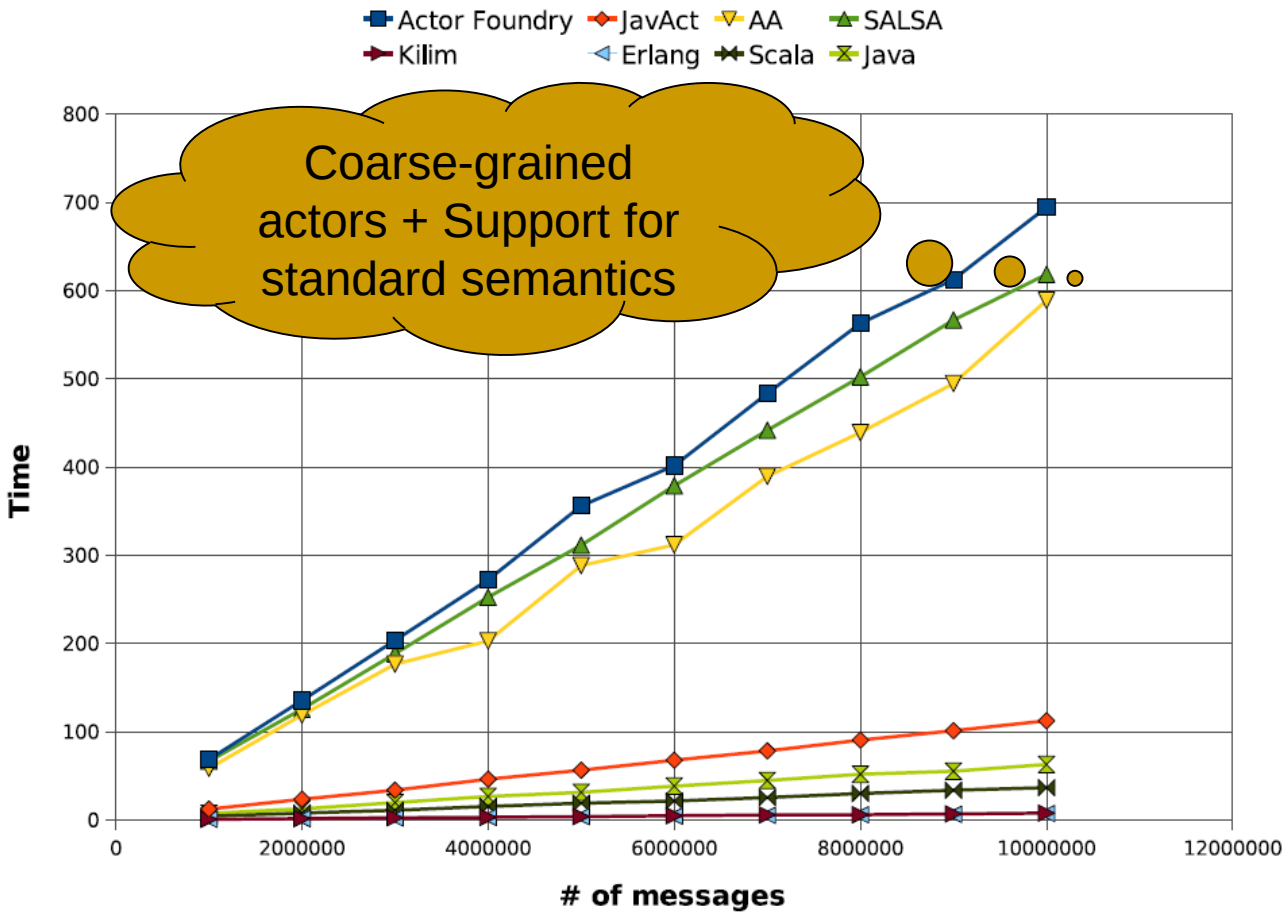
Great Language Shootout

- Ported ActorFoundry to Java6
- The Computer Language Benchmarks Game [2]
- Implemented a concurrent benchmark in ActorFoundry: Thread-ring
- **Thread-ring:** Pass a token around 503 concurrent entities 10^7 times

Platform	**Time (s)**
Java threads	134.98
Haskell threads	6.70
Erlang light-weight processes	7.49
Scala actors	56.5
ActorFoundry actors	13 minutes

[2] http://shootout.alioth.debian.org/

and now in a Chart



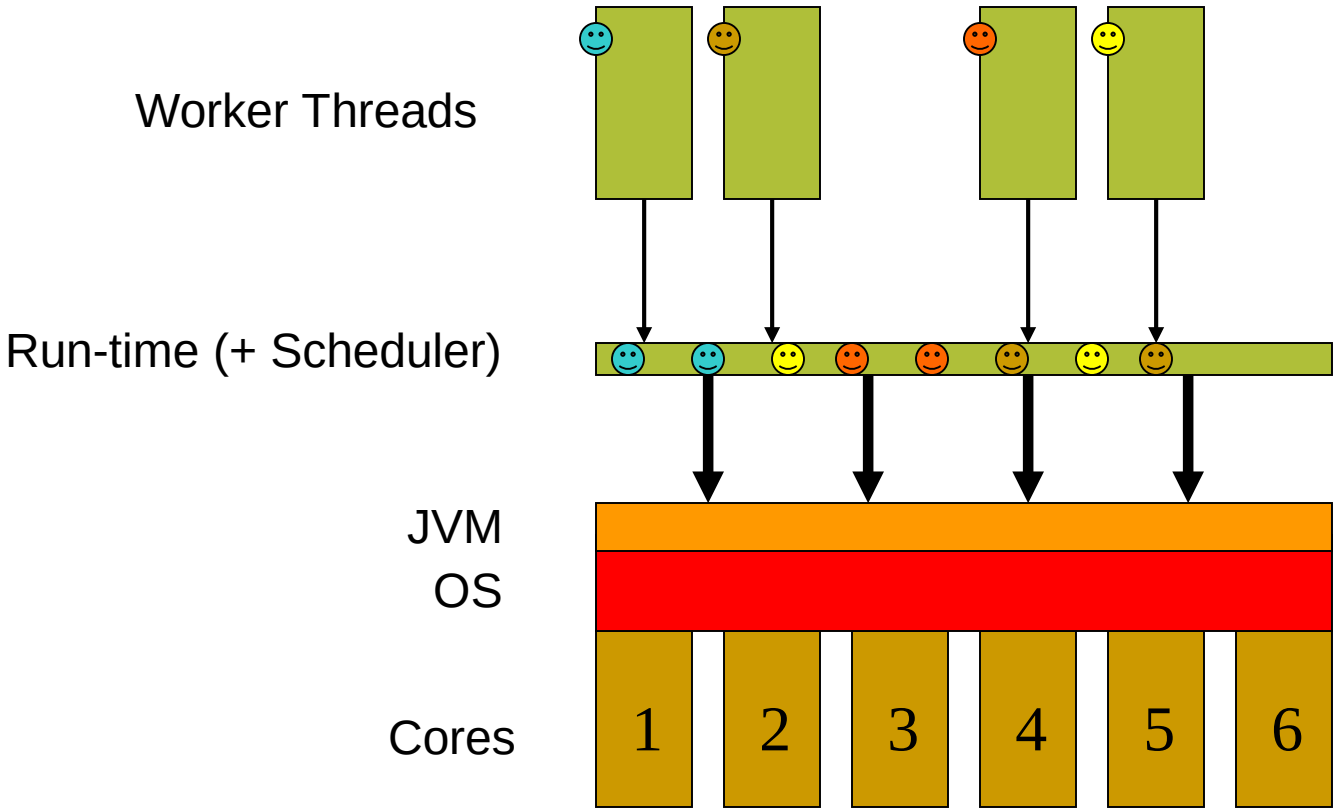
Intel Core2 Duo, 2.4GHz, 4GB RAM, Java Heapsize: 256M

[Karmani and Shali and Agha, *PPPJ* 2009.]

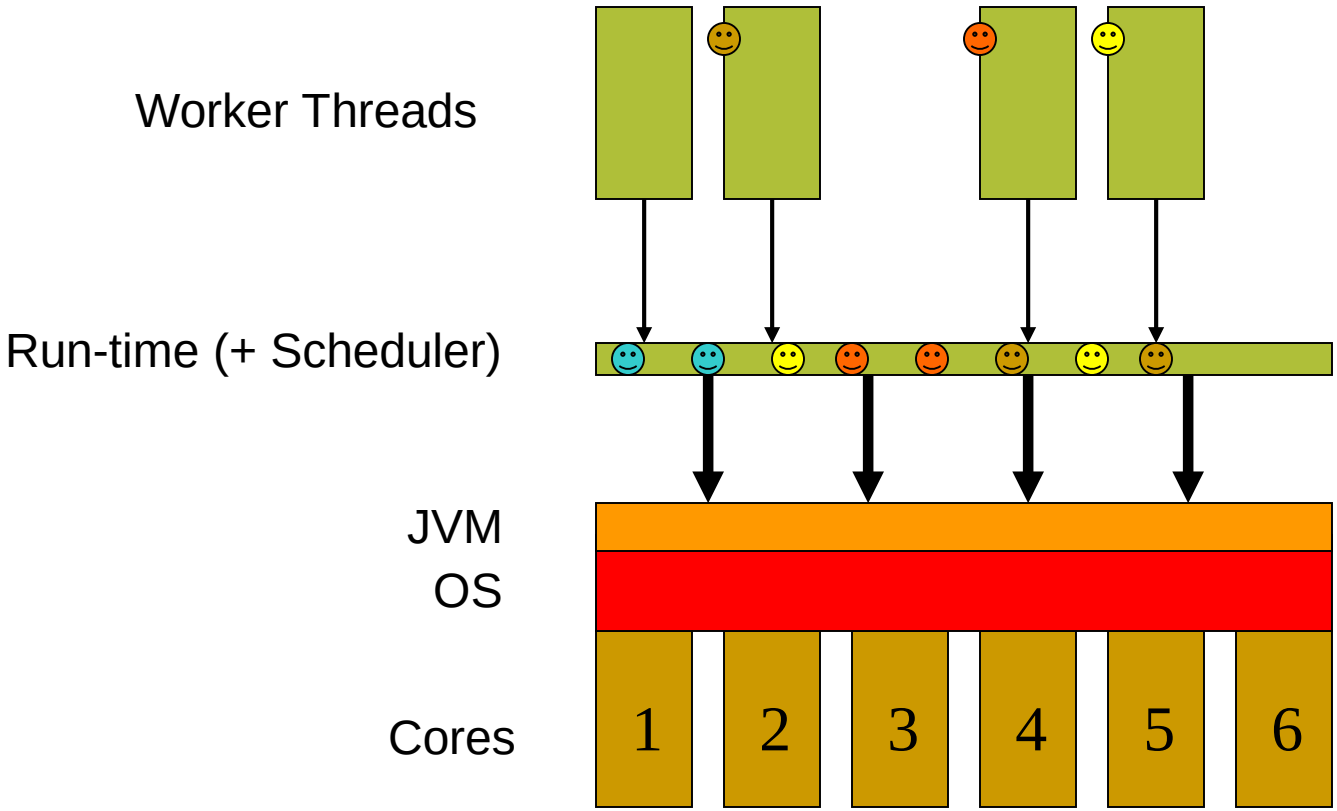
The Quest for Continuations

- Kilim actor runtime library provides a "Weaver" (Java bytecode post-processor) for CPS transform
- With a custom continuations based scheduler
  - M:N architecture

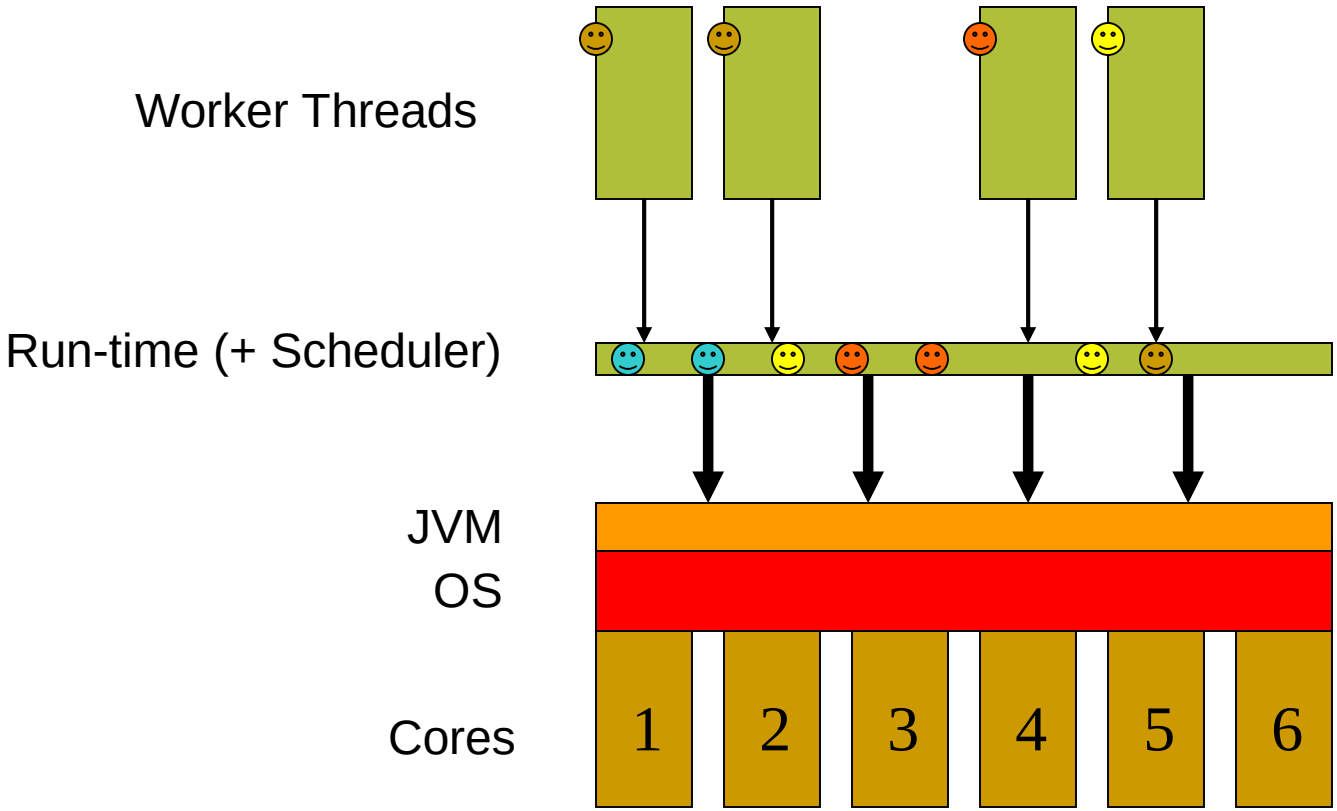
M:N Runtime Architecture



M:N Runtime Architecture



M:N Runtime Architecture

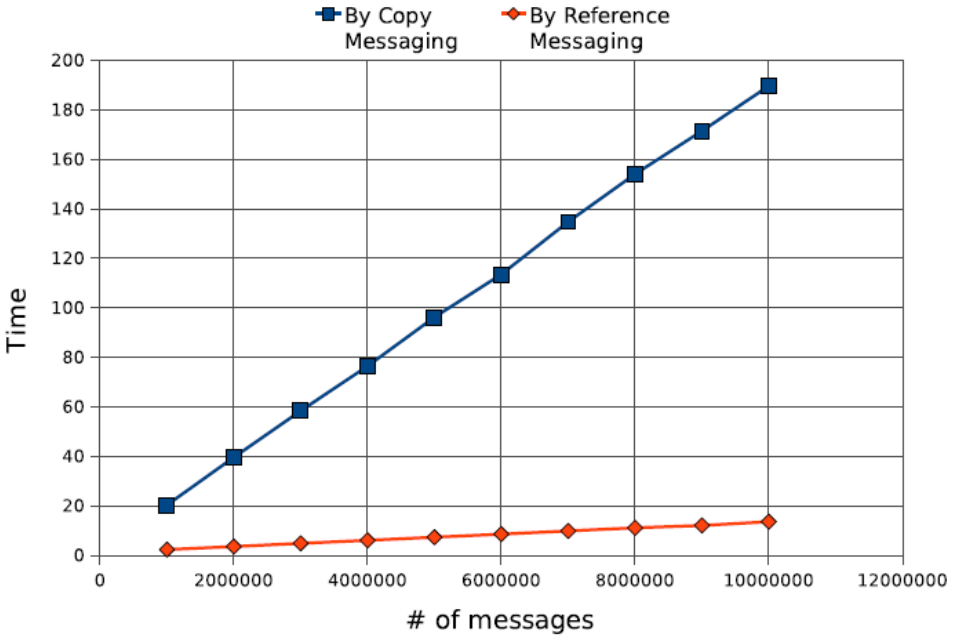


Fairness – ActorFoundry Weaving

- Scheduler thread periodically monitors "progress" of worker threads
  - Progress means executing an actor from schedule queue
- If no progress has been made by any worker thread => possible starvation
  - Launch a new worker thread
  - Conservative but not incorrect
- Integrate with ActorFoundry improves *threadring* performance: ~ 7 minutes

The cost of isolation

- Major bottleneck: deep copying of message contents
  - serializing/de-serializing object



[Karmani and Shali and Agha, *PPPJ* 2009.]

To copy or not to copy?

1. Exclude immutable types
2. Introduced new run-time functions:
   - sendByRef (actor, method, params)
   - callByRef (actor, method, params)

- Threadring performance: ~ 30s
- Scala, Kilim provide zero-copy messages
  - Onus on programmers to copy contents, if needed
  - Breaks encapsulation, infeasible in general

Avoiding Message Copying

```
@message
public void getMatchedWords(String q, ActorName client) {
        List<String> retVal = new ArrayList<String>();
        q = q.trim();
        if (q != null && !q.equals("")) {
                for (String is : wordList) {
                        if (is.startsWith(q)) {
                                retVal.add(is);
                        }
                }
        }
        send(client, "matchedWords", retVal.toArray());
}
```

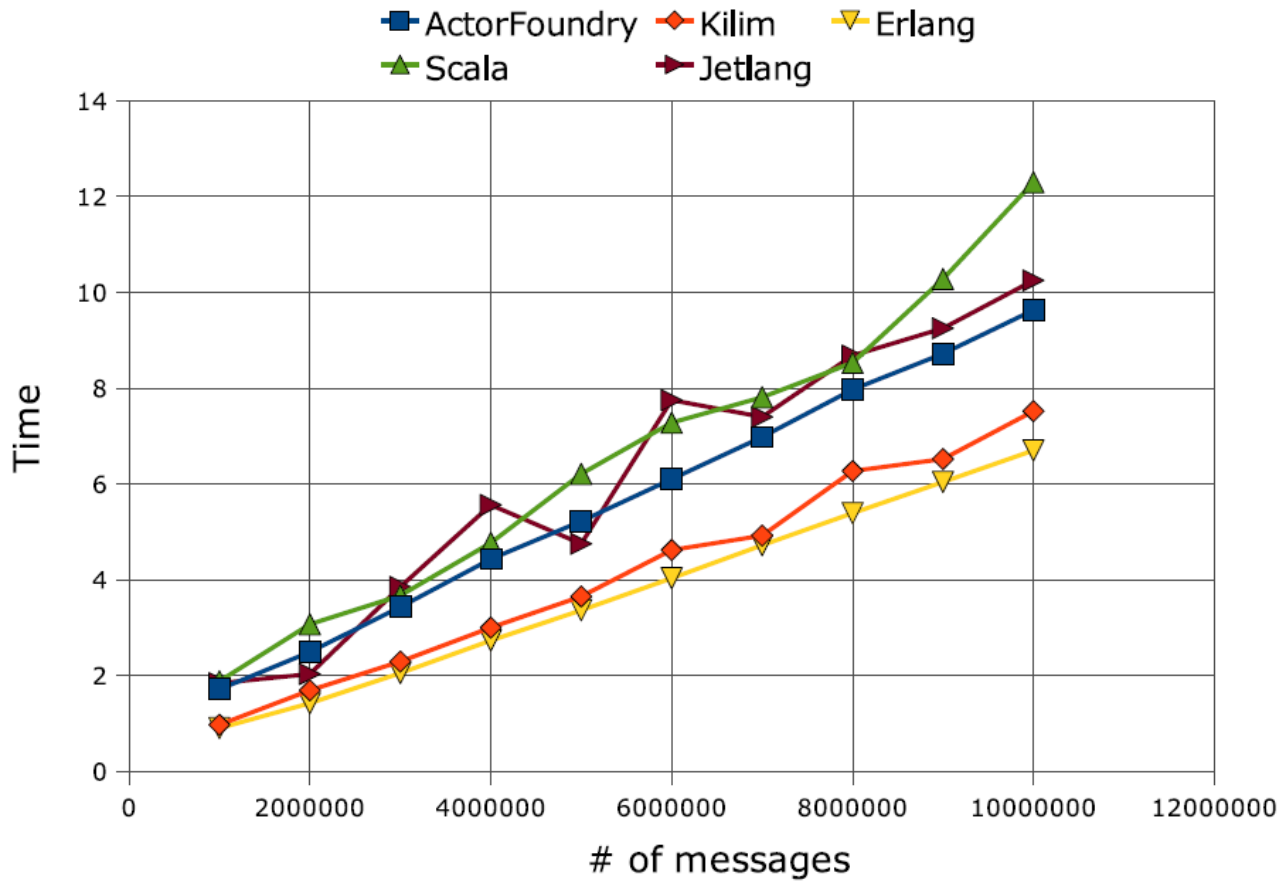
sendbyref

[Negara and Karmani and Agha, in submission, 2010.]

Static Analysis

```
@message
public void getMatchedWords(String q, ActorName client) {
        List<String> retVal = new ArrayList<String>();
        q = q.trim();
        if (q != null && !q.equals("")) {
                for (String is : wordList) {
                        if (is.startsWith(q)) {
                                retVal.add(is);
                        }
                }
        }
        sendByRef(client, "matchedWords", retVal.toArray());
}
```

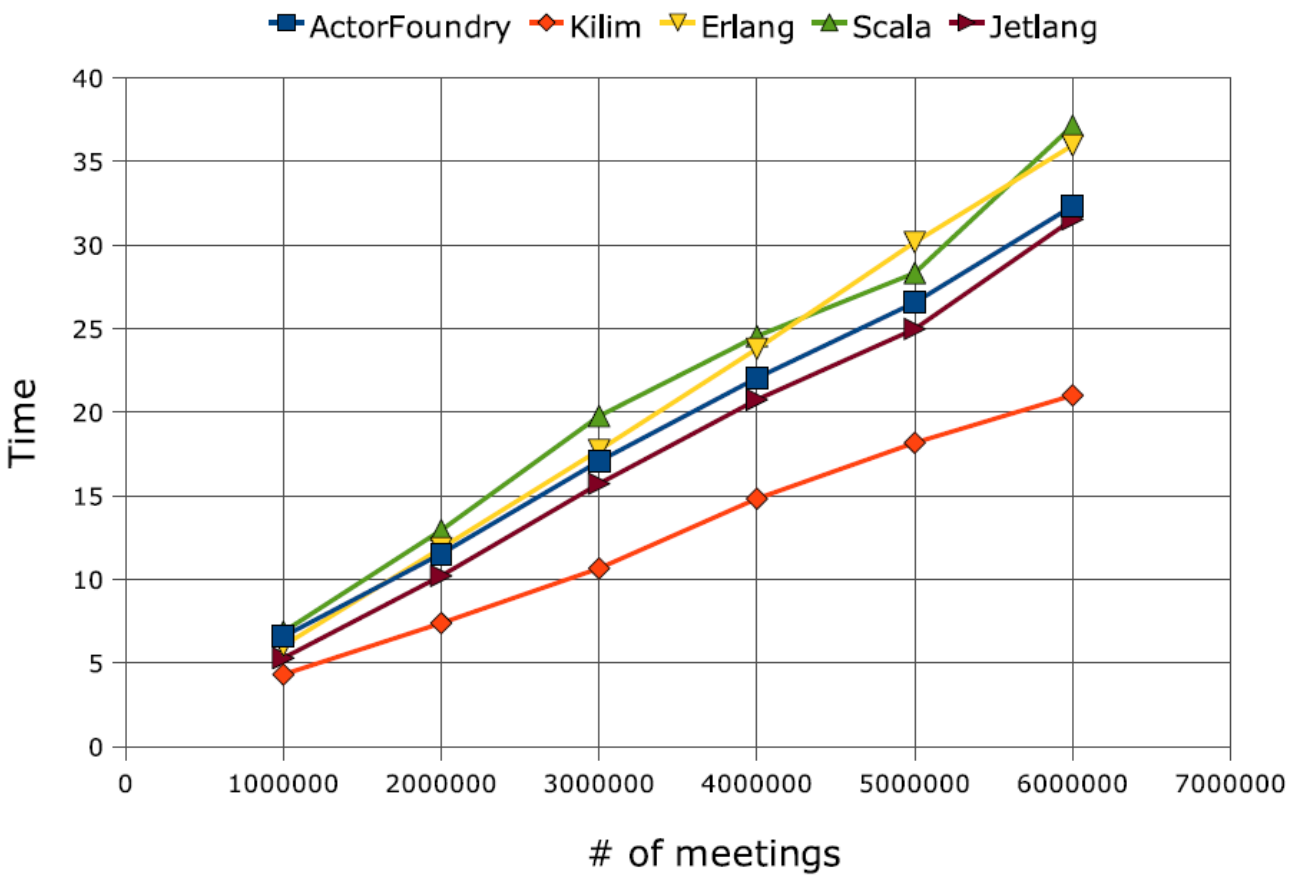
Performance – crude comparison



Threadring benchmark on Intel Core2 Duo, 2.4GHz, 4GB RAM, Heapsize: 256M

[Karmani and Shali and Agha, *PPPJ* 2009.]

Performance – crude comparison



Chameneos-redux benchmark on Intel Core2 Duo, 2.4GHz, 4GB RAM, Heapsize: 256M

[Karmani and Shali and Agha, *PPPJ* 2009.]

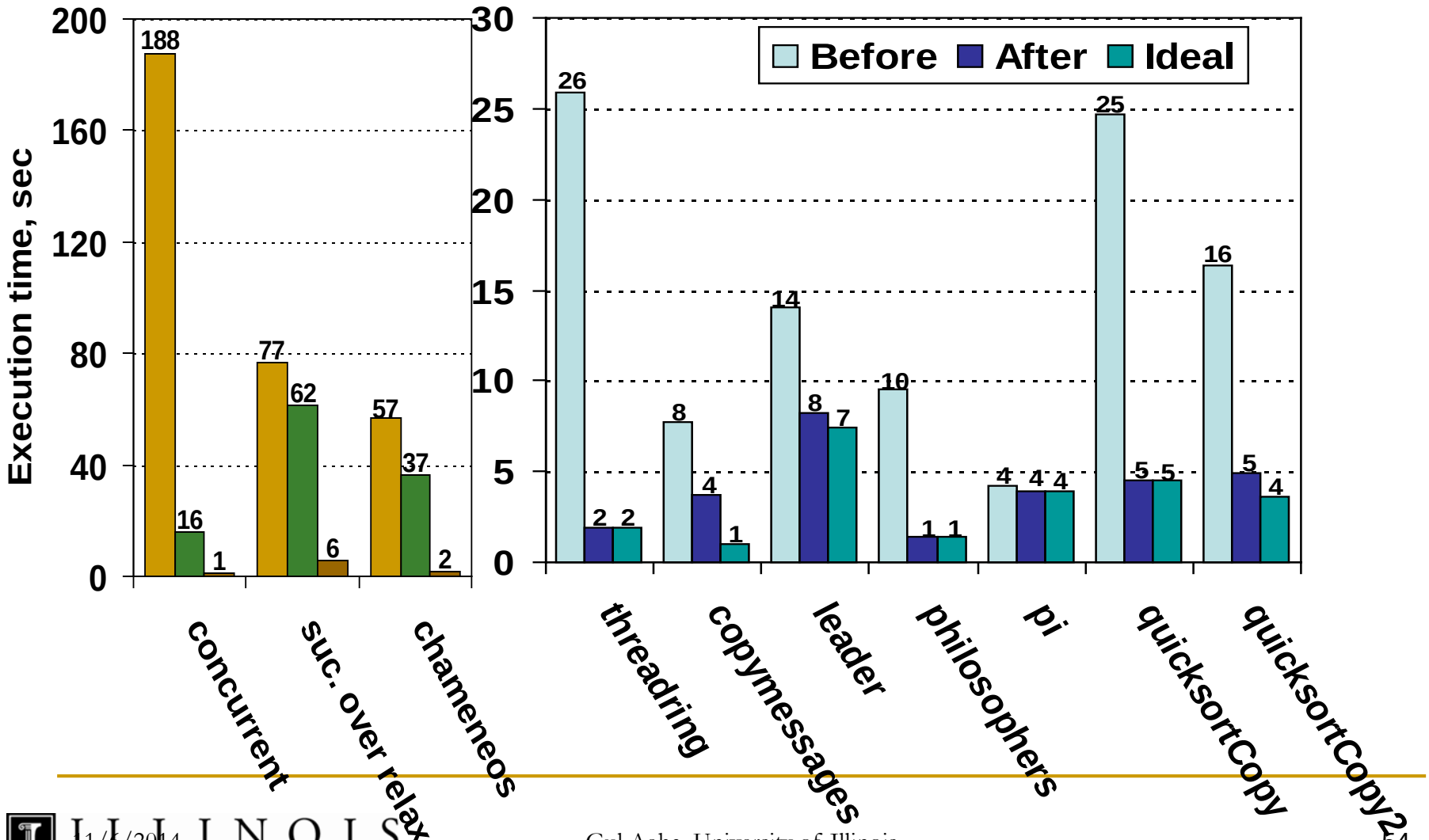
Can we infer ownership transfer?

- *Context sensitive* call graph:
  - Receiver instance context: separate nodes for the same method invoked on different objects (receivers)
  - Actors' messages serve as entry points
  - Bigger but sparser than *context insensitive* call graph
  - More precise but not much slower points-to analysis
- *Field sensitive*, *flow insensitive* interprocedural Andersen's points-to analysis
- Custom interprocedural live variable analysis
- Incorporated in a tool called SOTER

Sources of imprecision

- Imprecise, *flow insensitive* points-to analysis
- Assumption that fields are always live (insignificant impact)
- *Open world* assumption: not checking for safe read/read sharing
- General static analysis limitations
  - e.g. not detecting false sharing
- Not considering semantics
  - e.g. immutable objects

SOTER ActorFoundry usefulness



Observations

- Ownership transfer is very common
- Although conservative, *SOTER* can infer most safe *pass by reference* sites:
  - *ActorFoundry* programs: 71%
  - *Scala* actor programs: 84%
- *SOTER* regains much of the optimal performance without requiring any additional effort from a developer
- *SOTER* handles actor programs in reasonable time:
  - *ActorFoundry* programs: max analysis time = 24 seconds
  - *Scala* actor programs: max analysis time ≈ 78 seconds

Promise for scalable performance?

- Over-decompose application into fine-grained actors
- As the number of cores increase, spread out the actors
  - Location independent naming facilitates transparent migration
- Of course, speed-up is constrained by parallelism in application
  - Parallelism is bounded by # of actors

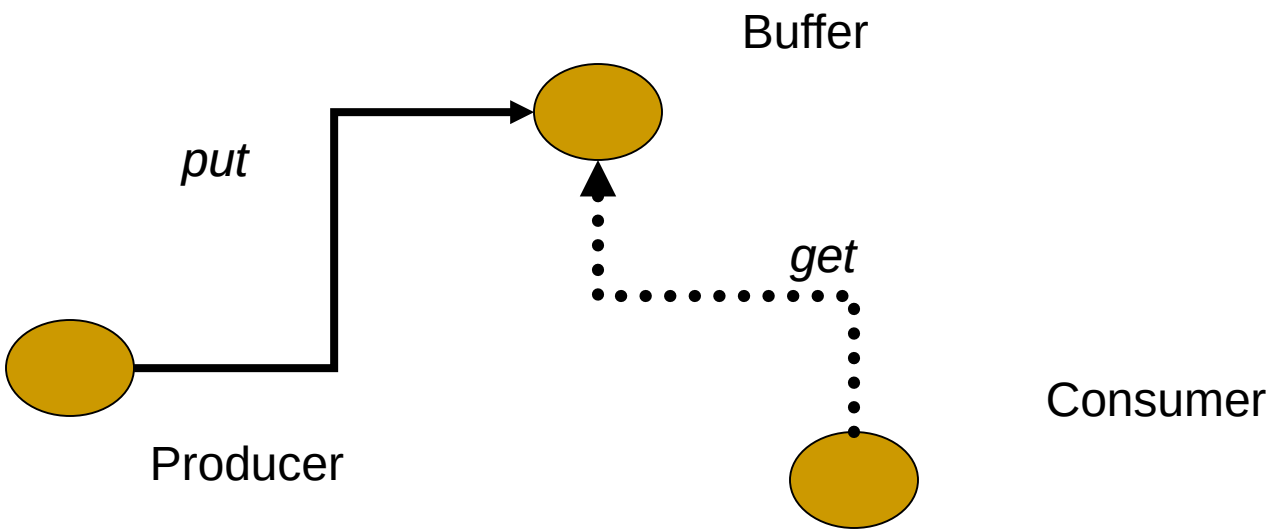
High Level Concurrent Languages

- Primitive actor constructs give a bare bones language for concurrency.
- High level language abstractions are needed to simplify concurrent programming.

Synchronization and Coordination

- Essential for correct functioning of actor systems
- A source of complexity in concurrent programs

Synchronizing in a Concurrent World



- The interface of an actor may be dynamic:
  - Cannot get from an empty buffer
  - Cannot put into a full buffer

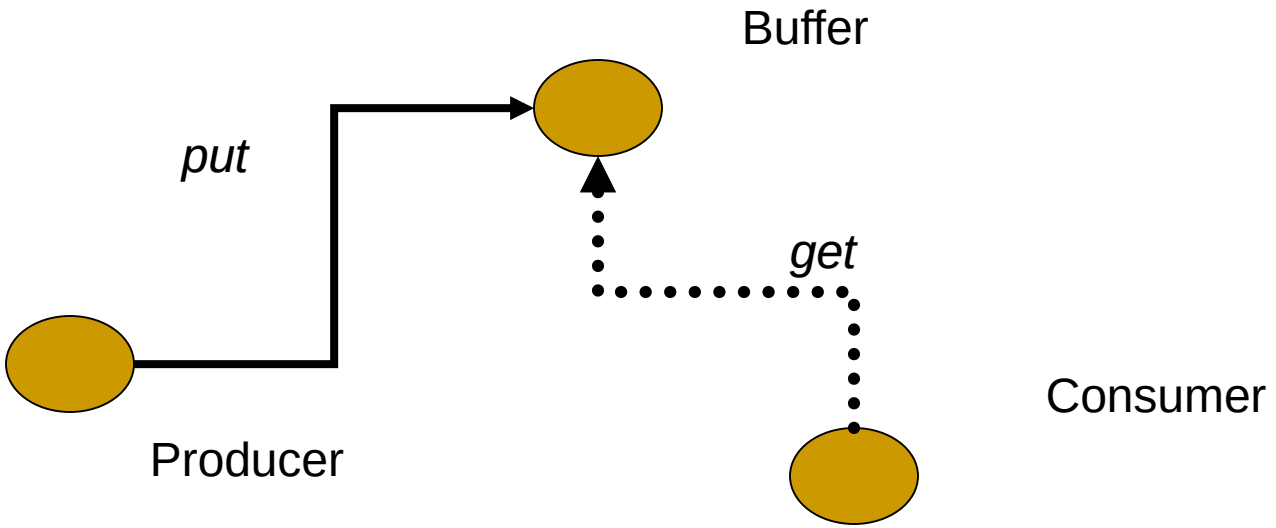
Separation of Concerns

- Abstract Data Types:
  - Enable separation of *interface (what)* from the *representation (how).*
- Actors:
  - *When* actions happen is underspecified (*asynchrony).*
  - Recipient may not be ready to process a message when it arrives – *synchronization constraints.*

Local Synchronization Constraints

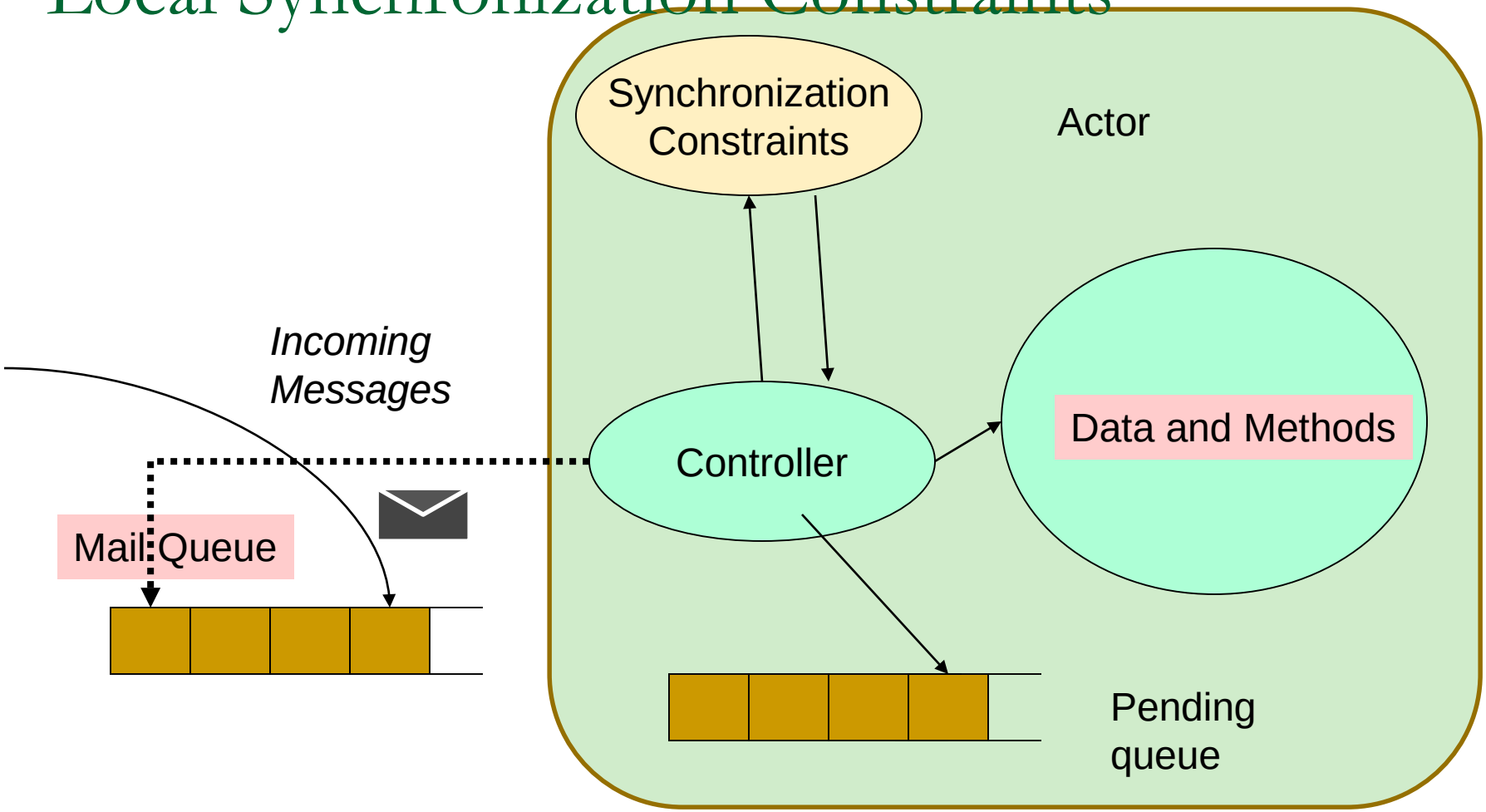
- Constrain the "local" order of processing messages
  - For all sending actors
- Usually specified as disabling conditions on message
  - Function of local state and message contents
- These have *delay* semantics i.e. disabled messages are buffered
- Implementations: Disabling constraints in AF, Pattern matching in Erlang, Scala

Expressing Local Synchronization Constraints



- Per actor logical rules which determine the legality of invocations:
  - disable get when empty? (buffer)

Implementation of Local Synchronization Constraints



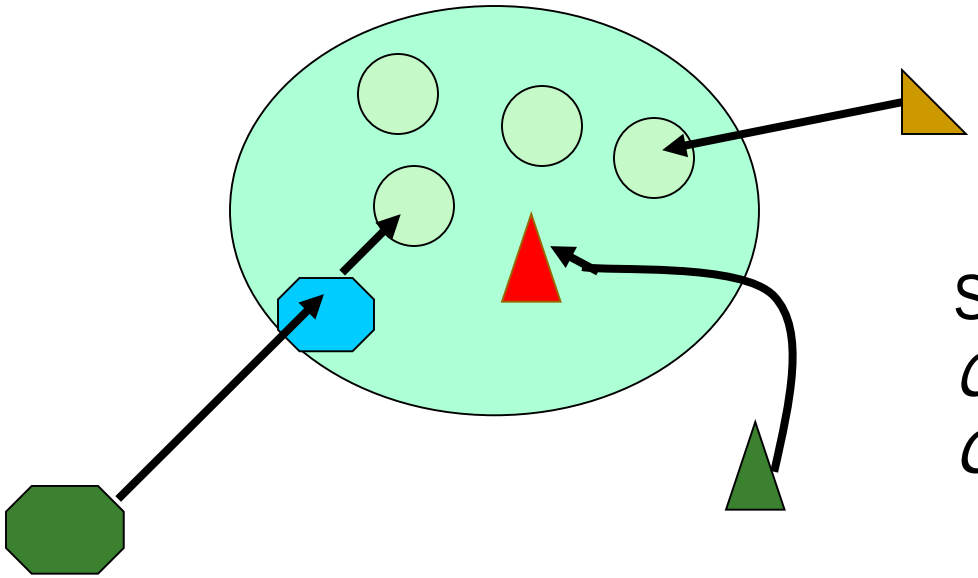
Multiparty Coordination Constraints

- Synchronization between actors
  - precedence of actions
  - “atomic” actions

Example: two robots which need to cooperate

Synchronizers

- A ‘membrane’ which filters and synchronizes messages
- Expresses logical expression of constraints *between* actors



S. Frolund, *Coordinating Distributed Objects,* MIT Press

Synchronizers

- Express interaction among multiple actors
- Can have internal state
- Also allows expressing atomic processing of messages by multiple actors
  - All-or-none semantics
- Declarative style (`disable`, `atomic`)
- Include more actions? `drop`, `reject`

Extensions of the Actor Model

- Real-time systems.
- Sensor networks.
- Control and hybrid systems.
- Cyberphysical systems.

Cloud Computing

- Center of data storage and processing
- Efficient, Elastic, Scalable, Affordable

- Cloud + Mobile:
  1. Remove hardware limitations
  2. Reduce mobile energy consumption
- Mobile-Cloud Computation (MCC)
  - How? Use code-offloading

Cloud Model:

- Public Cloud
  - Affordable/scalable/elastic
  - Problems:
    1. Lack of control and transparency
    2. Legal implications
- Private Clouds
  - User-control
  - Problems:
    - Not efficient/scalable/elastic as public Cloud
- Hybrid Cloud:
  - benefits of public Cloud
  - User-control

Code offloading

- How?
  1. Ask programmers to rewrite the code
  2. Use full process or full VM migration
  3. Combine the two
     - Separation of concerns
     - Fine-Granularity

References: [1]

Our Approach

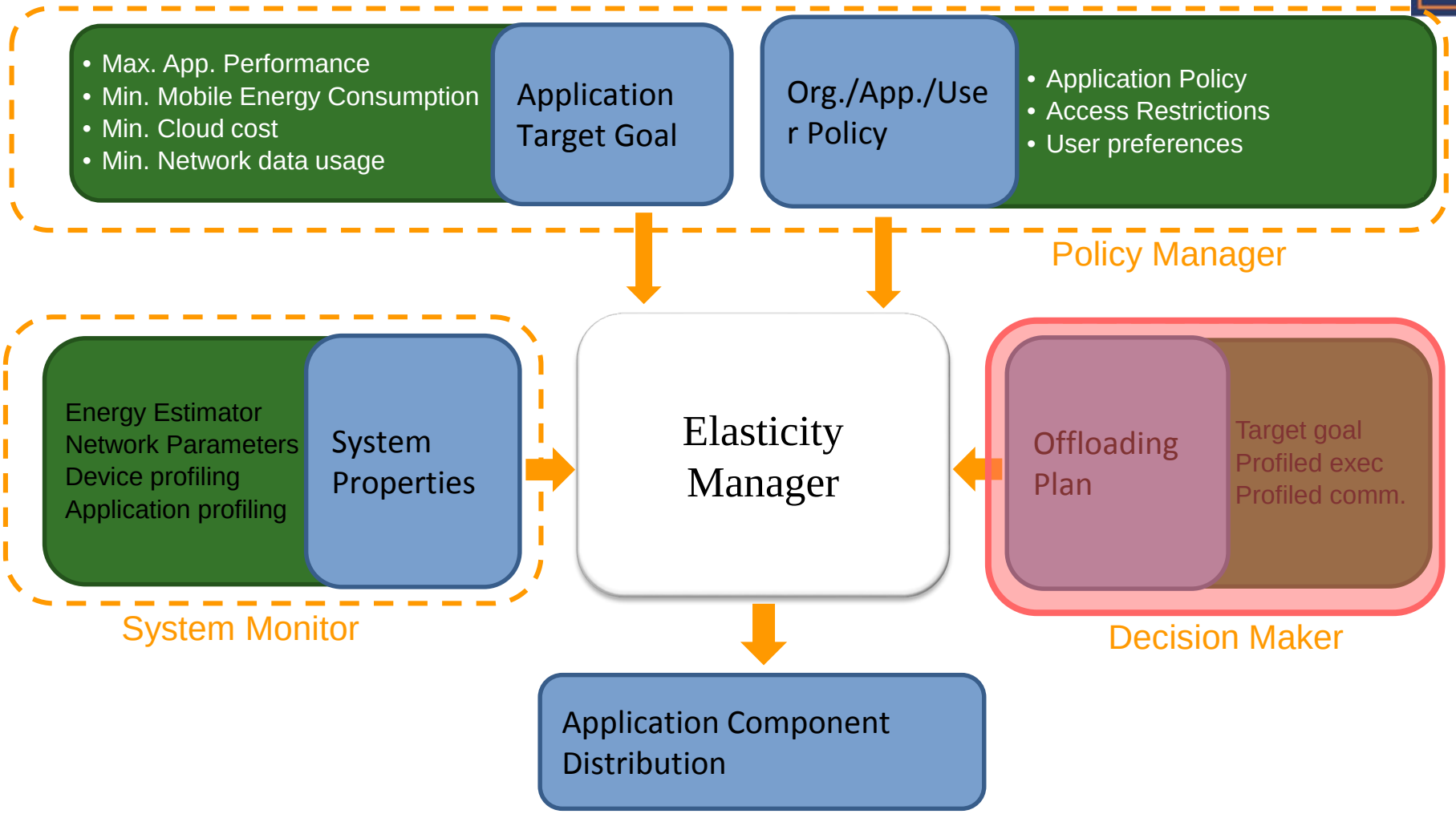
- Bridge the gap between mobile application and cloud computing
- Separate application logic from component distribution
- Allow dynamic distribution of components based on run-time changes
- Support policy-based definition of customized constraints

Related Mobile-Cloud Systems

Year	System Name	Goal	Offloading Decision	Partition Level	Parallel	Policy-based Privacy	Manual Work	No. Cloud spaces
2010	MAUI	Mobile Energy Saving	Dynamic	Method	No	No	Yes	1
2011	CloneCloud	Mobile Energy Saving = Performance Improvement	Static	Method	Pseudo	No	No	1
2012	ThinkAir	Mobile Energy Saving = Performance Improvement	Dynamic	Method	Pseudo	No	Yes	1
2012	Cloud OS (COS)	Load Balancing for Cloud space	Dynamic	Actor	Yes	No	No	Can be Many
2014	IMCM	Mobile Energy Saving, Performance Improvement, Combination for Applications	Dynamic	Actor	Yes	Yes	No	Many

References: [2, 3, 9, 10]

IMCM: Overall System Design



IMCM: Illinois Mobile Cloud application Management

An Adaptive Architecture

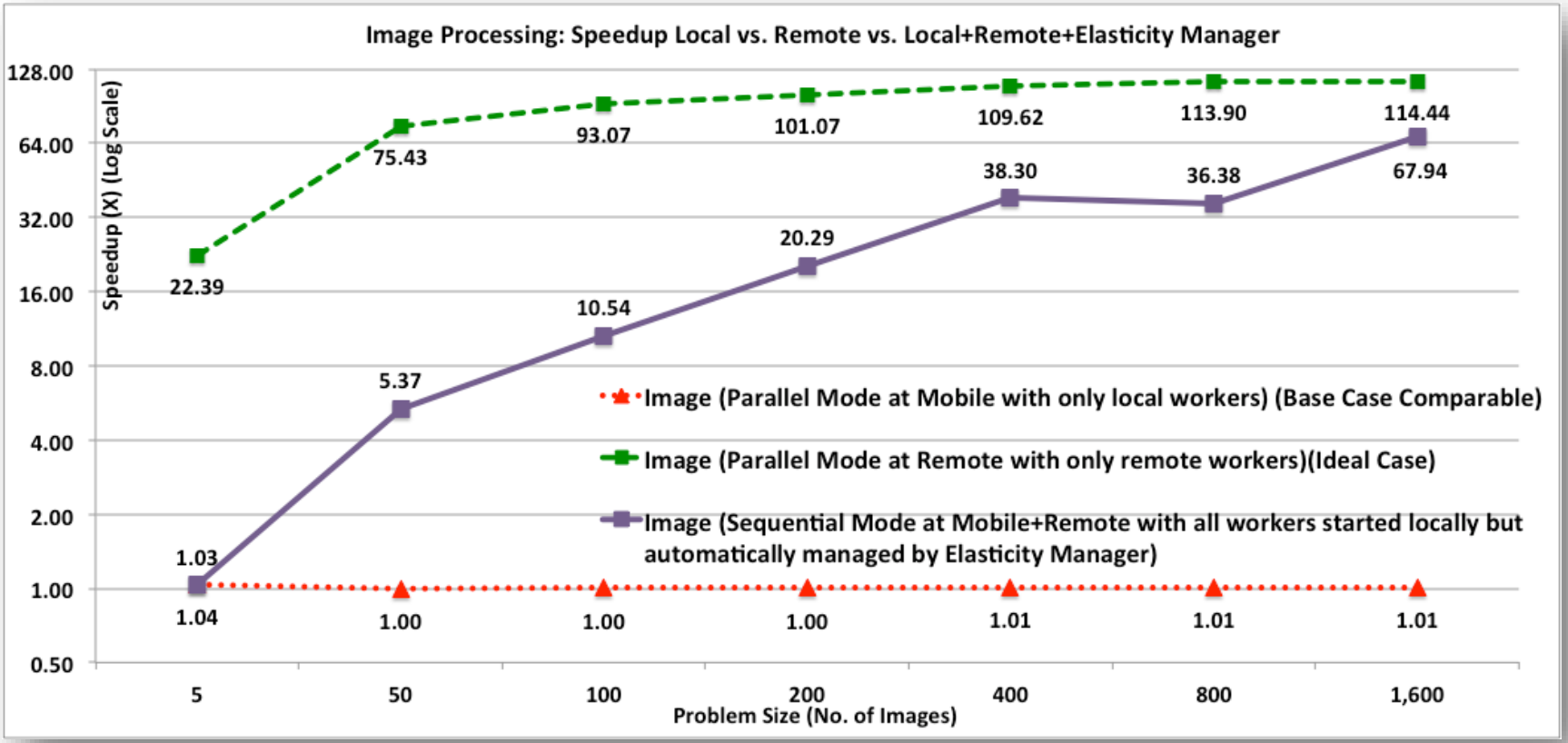


Image proc: Local exec (base case) vs. Remote Exec (Ideal Case) vs. Automatic Management

Flexible Policy-based restrictions

- IMCM Policy rule definition/enforcement system:
  - Authorization policy:
    1. Hard Policy:

       organization-wide policy
    2. Soft Policy:

       User/developer policy (App-instance-specific policy)
  - Every application-instance comes with:
    1. An unchangeable hard policy
    2. A (changeable) soft policy
  - Grammar to define rules
    - Attribute based
    - Static & dynamic binding

References: [ ]

US-Korea-Japan Testbed on Wireless Smart Sensor Networks for SHM

(June 2009 – December 2012)

- Field validation of smart wireless sensor-based SHM system of a cable-stayed bridge in Korea
- Establishment of an international SHM testbed using massive distributed computing
- Participants :
  - Korea : H.J. Jung & C.B. Yun (KAIST), H.K. Kim (SNU), J.J. Lee (Sejong U.) J.W. Seo (Hyundai E&C)
  - US : B.F. Spencer, Jr. & G. Agha (UIUC)
  - Japan : Y. Fujino & T. Nagayama (U. of Tokyo)

2nd Jindo Bridge	
Type	Cable-stayed bridge
Spans	70+344+70 = 484m
Girder	Steel box (12.55m width)
Design velocity	70 km/hr
Designed by	Yooshin cooperation (2000, Korea)
Constructed by	Hyundai construction (2006, Korea)
Owner	Iksan Regional Construction and Management Administration
Special feature	Twin bridge

References

1. Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.
2. Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, 2009. ACM.
3. Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 1997.
4. Rajendra Panwar and Gul Agha. A methodology for programming scalable architectures. In *Journal of Parallel and Distributed Computing,* 1994.
5. Astley, Mark, Daniel C. Sturman, and Gul Agha. "Customizable middleware for modular distributed software." *Communications of the ACM* 44.5 (2001): 99-107.