Separating Concerns in Concurrent Systems

Nadeem Jamali

University of Saskatchewan

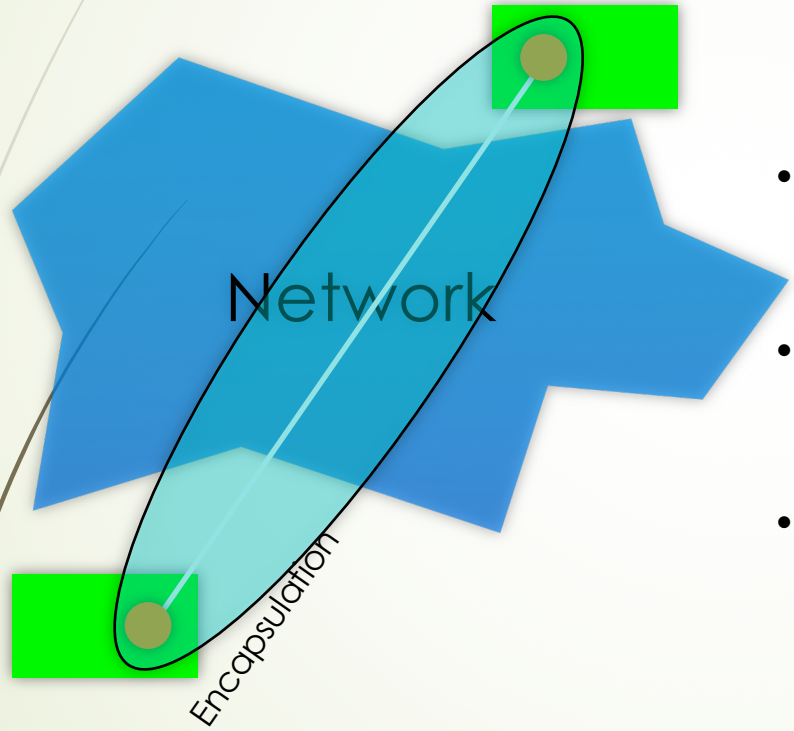
Research Directions

1. Resource Coordination: enabling resource competition between Actor-based applications
   - CyberOrgs
   - Applied to system and application resources
2. Separating communication concerns
   - interActors
   - Applied to protocol separation and crowd-sourced services

1. Resource Coordination

- Enabling resource competition between Actor-based applications
- Approach: Separation of distribution concerns:
  - Encapsulate resources along with distributed applications
    - Local resources (e.g., CPU)
    - Network bandwidth
    - Energy
  - Offer predictable resource environments

Resource Encapsulation



- Decisions about resource ownership
  - Market of resources
- Delivery of resources
  - Fine-grained control
- Primitives for evolving encapsulations
  - Isolate, Assimilate

CyberOrgs

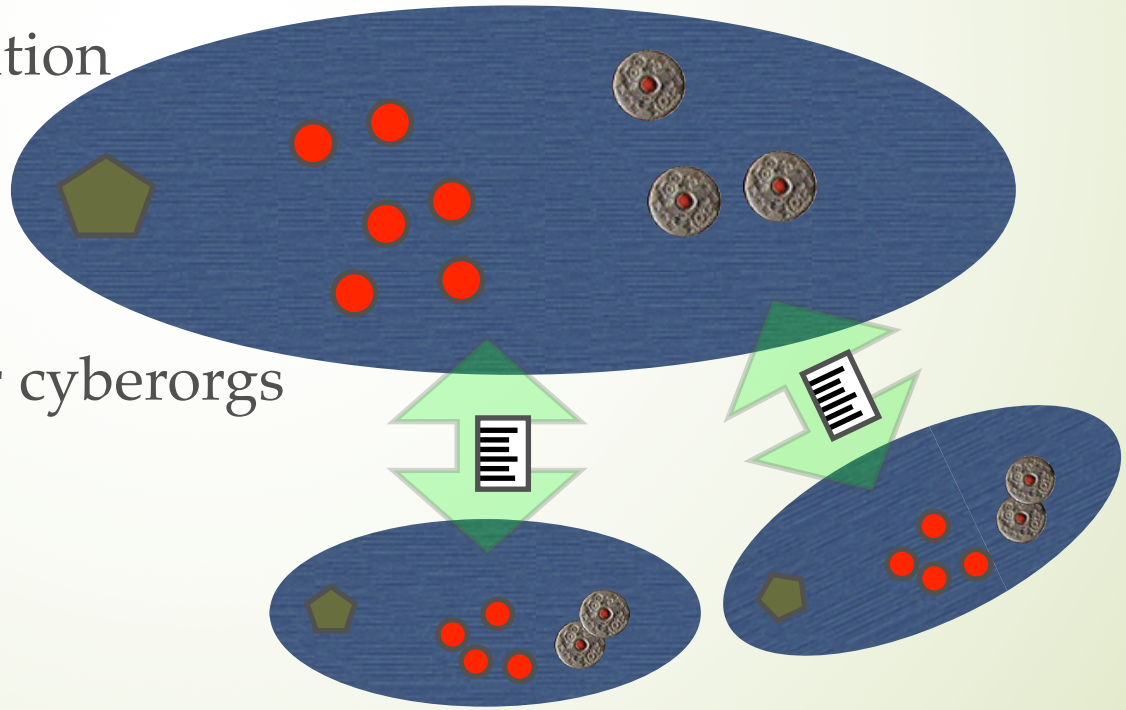
[Nadeem Jamali, Xinghui Zhao,“ Hierarchical Resource Usage Coordination for Large-Scale Multi-agent Systems,” in Massively Multi-Agent Systems, Ed. T. Ishida, L. Gasser, H. Nakashima, LNCS 3446, pp 40-54, 2004]

[Nadeem Jamali, Xinghui Zhao,” A Scalable Approach to Multi-Agent Resource Acquisition and Control.” AAMAS 2005, pp 868-875]

A cyberorg encapsulates:

- A distributed computation
- Purchasing power
- Commitments to other cyberorgs
- Facilitator

Cyberorgs pay each other for resource ticks, as determined by contracts



System Resources
(with Xinghui Zhao, Chen Liu)

- Processor time
  - To support deadline-constrained computations; parallel computations with non-uniform tasks

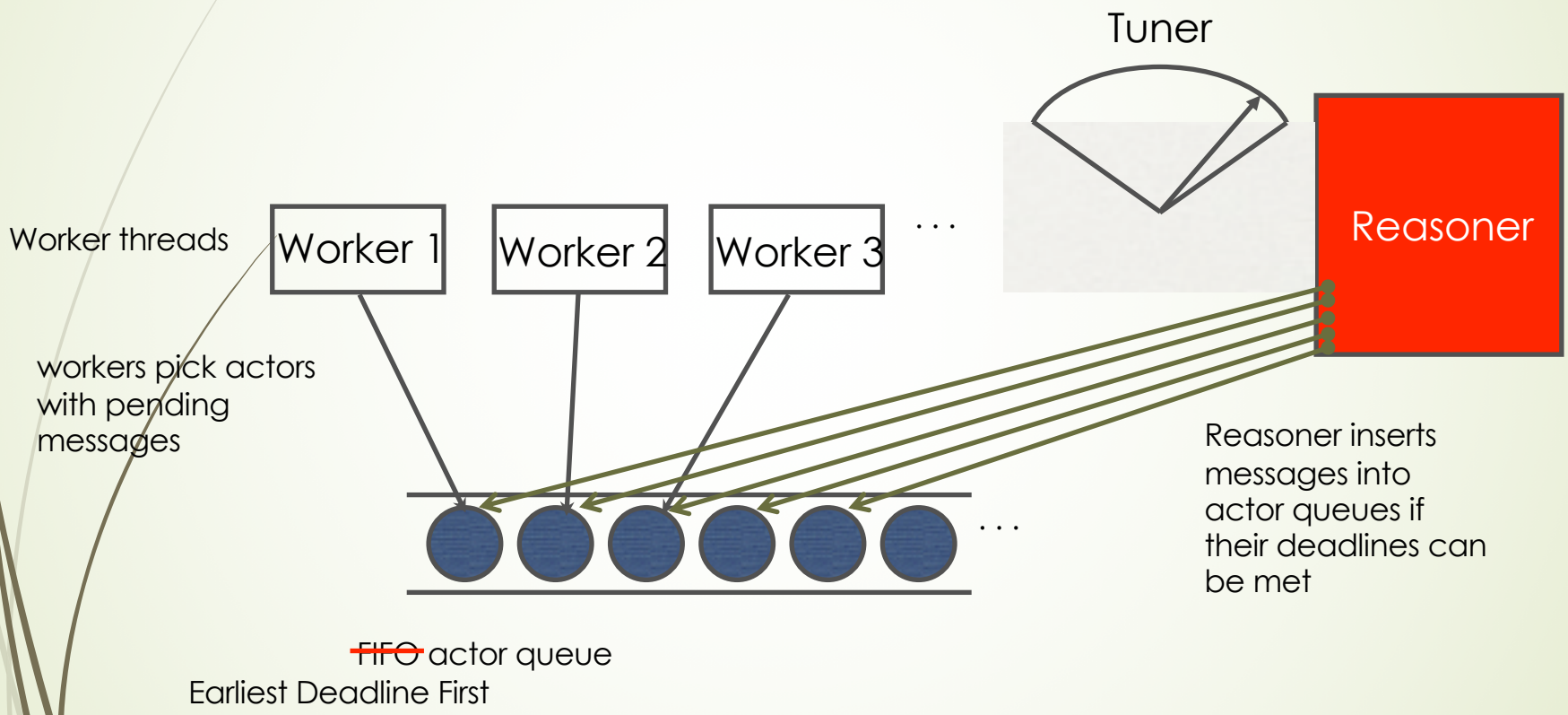
    [Xinghui Zhao, Nadeem Jamali, "Supporting Deadline Constrained Distributed Computations on Grids," Grid Computing 2011, pp 165-172]

    [Xinghui Zhao, Nadeem Jamali, "Load Balancing Non-Uniform Parallel Computations," AGERE@SPLASH 2013, pp 97-108]
  - Extended to energy-efficient scheduling: run cores as slowly as needed

    [Xinghui Zhao, Nadeem Jamali, "Fine-grained Per-Core Frequency Scheduling for Power-Efficient Multicore Execution," IGCC 2011, pp 1-8]
- Network bandwidth

  [Nadeem Jamali, Chen Liu,"Reifying Control of Multi-Owned Network Resources," HIPS@IPDPS 2007, pp 1-8]

Extend Actors RunTime



[Zhao and Jamali, Grid Computing 2011]

Challenging To Convincingly Evaluate

- Existing applications assume limited notions of concurrency
  - Programmers don't immediately see value of new capabilities (e.g., deadline support)
  - Lack of suitable benchmarks
    - Had to make up our own (Unbalanced Cobwebbed Tree, now part of Savina Actor Benchmark Suite) *[Zhao and Jamali, AGERE 2013]*

Application-Level Resources

- Human attention
  - Treat human attention as owned resource
    - Basis for advertiser-viewer negotiation of displayed advertising

      [Yue Zhang, Nadeem Jamali, "Negotiating Multimedia Advertising with Attention Owners," ACM Multimedia 2010, pp 755-758]
  - Treat message mailbox as owned resource
    - Basis for message sender-recipient negotiation for sending privileges

      [Nadeem Jamali, Hongxing Geng, "A Mailbox Ownership Based Mechanism for Curbing SPAM," Computer Communications 31(15), pp 3586-3593, 2008]
    - Incentivize message-sender help in reduce recipient's time fragmentation

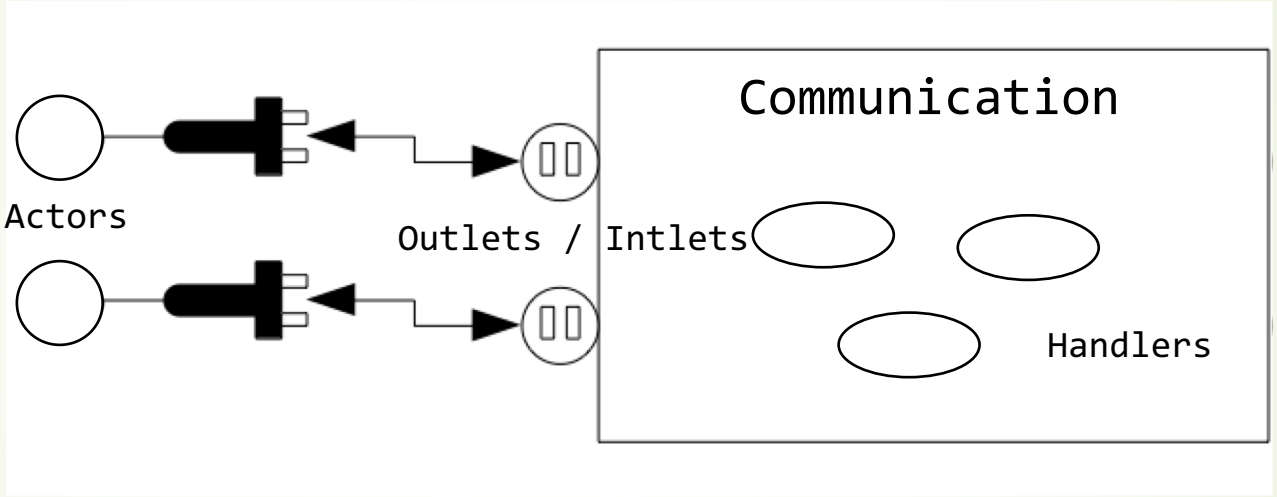
      [Sander Wang, "SchedMail: Sender-Assisted Message Delivery Scheduling to Reduce Time-Fragmentation," MSc Thesis, University of Saskatchewan, 2019]

2. Separation of Communication Concerns (with Hongxing Geng)

- Challenges:
  - Interactions in concurrent systems are increasingly complex
  - Communication concerns are typically mixed with computations
- Approach:
  - Moving control flow to communications (self-driven communication)
  - Dynamically evolving protocols
  - Enabling rendezvous with processes at run-time
  - Supporting creation of libraries of reusable communications
  - Support composability

interActors

[Hongxing Geng, Nadeem Jamali, "interActors: A Model for Separating Communication Concerns of Concurrent Systems," in Programming with Actors, Ed. A Ricci, P. Haller, LNCS 10789, 2018]



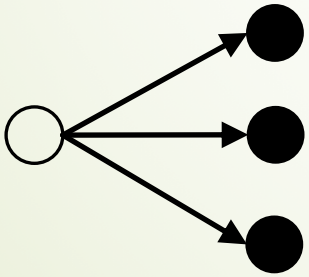
- Actors plug into oilets (outlets / inlets)
- Outlets and handlers have targets and behaviors, such as:
  - Forwarder
  - Counter
  - Applier
  - Filter
  - Selector
  - Aggregator

Communications

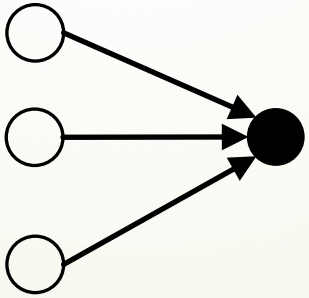
Primitive Communication: Channel

Complex Communications: Defined Compositionally

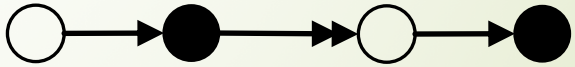
Input Merge



Output Merge

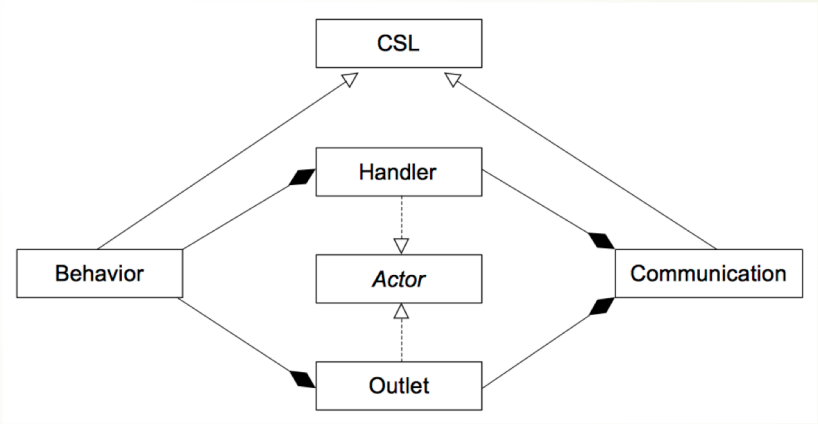


Output-Input Merge



Prototype Implementation

- Implemented using Scala and Akka
- Includes a small number of classes
  - Behavior
  - Outlet
  - Handler
  - Communication
  - CSL



Communication Specification Language (CSL)

- Styled after Scala
- Allows a set of expressions and statements
- Syntactic restrictions:
  - Disallows iterations (but allows external temporal / situational triggers)

Code Examples: Behaviors

```
behavior forwarder(targets): {
   receive(msg) = {
      sendm(targets, msg);
   }
}
```

```
class Forwarder(val ct: ActorContext,
                val targets: List[ActorRef])
                     extends Behavior(ct, targets) {
   def receive(msg: Any, agent: ActorRef) = {
      sendm(targets, msg)
   }
}
```

```
behavior aggregator(targets,
           val cond: List[Any]=> Boolean,
           val aggr: List[Any]=> Any): {
   var msgs = List[Any]();
   receive (msg) = {
      msgs = append(msg, msgs);
      if (cond(msgs)) {
         sendm(targets, aggr(msgs));
      }
   }
}
```

```
class Aggregator(val targets: List [ActorRef],
                 val cond: List[Any] => Boolean,
                 val aggr: List[Any] => Any)
                                  extends Behavior(t) {
   def receive (msg: Any, agent: ActorRef) = {
      msgs = append (msg, msgs)
      if (cond( msgs)) {
         sendm(targets, aggr(msgs))
         // empty the list for the next aggregation
         msgs = List[Any]()
      }
   }
}
```

Code Example: Single-Origin Many-to-Many Communication

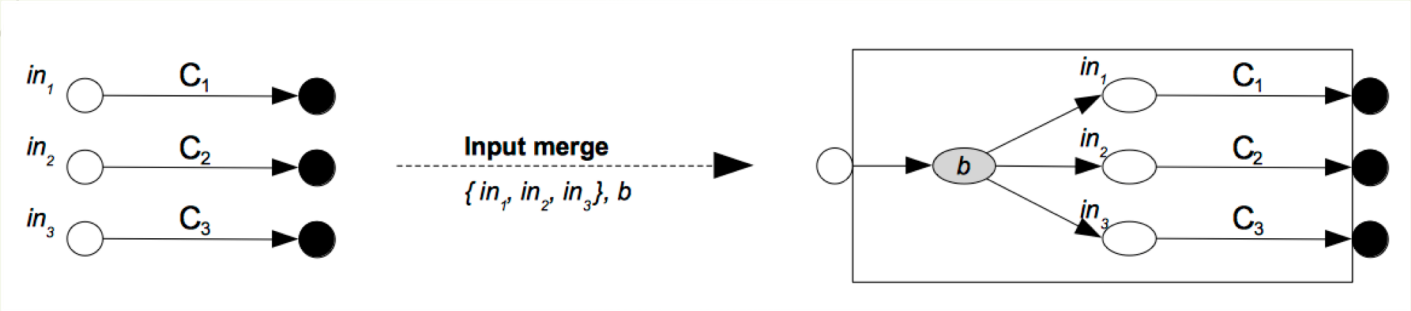
```
communication SOM2M {
    attributes: {
        recipients: List[ActorRef];
        participants: List[ActorRef];
        query: Any;
        cond: List[Any] => Boolean;
        aggr: List[Any] => Any;
    }
    outlet: out(forwarder(recipients));
    handler: aggrhandler(aggregator(out, cond, aggr));
    inlet: in(forwarder(aggrhandler));
    init: {
        tellm(in, participants, query);
    }
}
```

Communications can be composed after having attributes set, but before being launched.

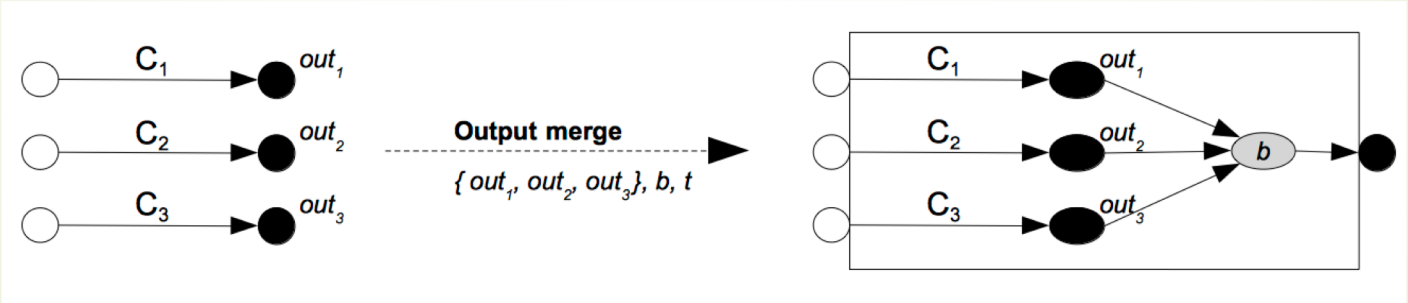
```
val som2m = new SOM2M(context)
som2m.setAttr(Map ("recipients" -> rec,
                   "participants" -> parts,
                   "query” -> query,
                   "cond" -> cond,
                   "aggr" -> aggr))
som2m.launch()
```

Compositions

- Input merge

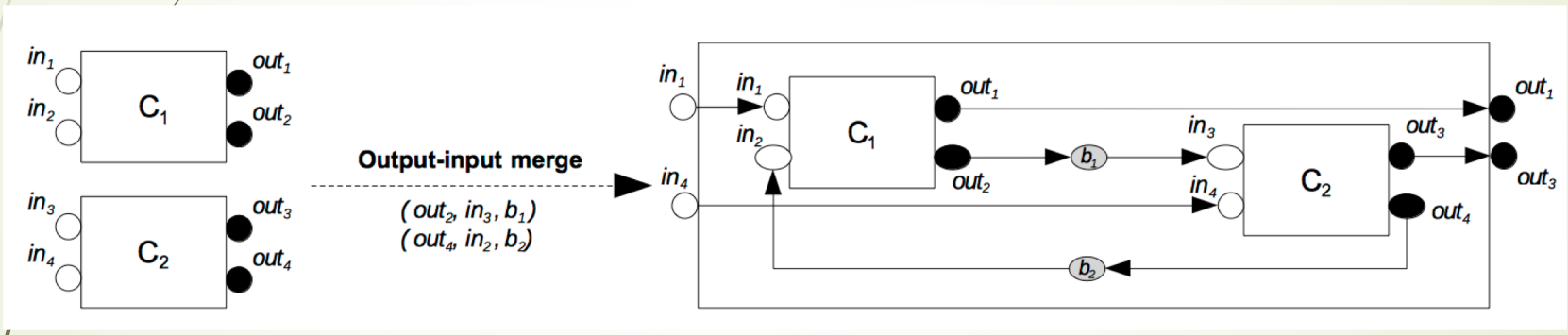


- Output merge



Compositions

- Output-Input merge



interActors Summary

- A way to separate communication concerns
  - Communications:
    - Are self-driven and can dynamically evolve
    - Have outlets into which actors can plug in
    - Can be composed to form more complex communications
  - Implementation in Scala / Akka
  - Communication Specification Language
    - Restrictions to avoid functional concerns in communications

Questions

- Actors allow highly complex systems to be built, with diverse actors and complex communications. However, it appears that this full power is not being utilized. Is this true? If so, why? And what will it take to change that?
- Programming with actors still seems hard (still many bugs). Is this because programmers are trying to do more complex things? [Rather than because "actors are not helping."]
- What would make it easier to implement such systems?