Leveraging Actor Frameworks for the Cloud

Gul Agha
University of Illinois at Urbana-Champaign
The actor model is a natural fit for programming cloud-based systems.
Actor Model of Computation

- Actors are autonomous agents which respond to messages
- Actors operate asynchronously, potentially in parallel with each other
- Each actor has a unique name (address) which cannot be guessed
- Actor names may be communicated
- Actors interact by sending messages, which are by default asynchronous (and may be delivered out-of-order)
Upon receipt of a message, an actor may:

- create a new actor with a unique name (address)
- use message contents to perform some computation and change state
- send a message to another actor
Constructing Actor Languages and Frameworks

Add to a sequential language:

- actor creation (local or remote): `create(node, class, params)`
- message sending: `send(actor, method, params)`
- ready (to process the next message)

Other typical constructs:

- request-reply messages
- local synchronization constraints (e.g., message pattern matching)
A Proliferation of Actor Implementations and Applications

- **Erlang** (Ericsson): web services, telecom, Cloud Computing
- **E-on-Lisp, E-on-Java**: P2P systems
- **SALSA, SALSA Lite** (UIUC/RPI): multicore, Cloud Computing
- **Charm++** (UIUC): scientific computing
- **Ptolemy** (UCB): real-time systems
- **ActorNet** (UIUC): sensor networks
- **ActorFoundry** (UIUC): multicore, Cloud Computing
- **Akka/Scala** (EPFL/Typesafe): multicore, web services, banking, ...
- **Kilim** (Cambridge): multicore and network programming
- **Orleans** (Microsoft): multicore programming, Cloud Computing
- **DART** (Google): Cloud Computing
- **Retlang/Jetlang**: multicore programming, Cloud Computing
Large-scale concurrent systems such as Twitter, LinkedIn, Facebook Chat are written in actor languages and frameworks.

Facebook

“[T]he actor model has worked really well for us, and we wouldn’t have been able to pull that off in C++ or Java. Several of us are big fans of Python and I personally like Haskell for a lot of tasks, but the bottom line is that, while those languages are great general purpose languages, none of them were designed with the actor model at heart.” –Facebook Engineering

Large-scale concurrent systems such as Twitter, LinkedIn, Facebook Chat are written in actor languages and frameworks.

**Twitter**

“When people read about Scala, it’s almost always in the context of concurrency. Concurrency can be solved by a good programmer in many languages, but it’s a tough problem to solve. Scala has an Actor library that is commonly used to solve concurrency problems, and it makes that problem a lot easier to solve.” – Alex Payne, “How and Why Twitter Uses Scala”

†http://blog.redfin.com/devblog/2010/05/how_and_why_twitter_uses_scala.html
1. **State encapsulation**: no direct access to state of other actors
Core Actor Semantic Properties

1. **State encapsulation**: no direct access to state of other actors

2. **Safe messaging**: messages have call-by-value semantics
Core Actor Semantic Properties

1. **State encapsulation**: no direct access to state of other actors

2. **Safe messaging**: messages have call-by-value semantics

3. **Fair scheduling**: messages are eventually delivered unless recipient is permanently disabled
Core Actor Semantic Properties

1. **State encapsulation**: no direct access to state of other actors

2. **Safe messaging**: messages have call-by-value semantics

3. **Fair scheduling**: messages are eventually delivered unless recipient is permanently disabled

4. **Location transparency**: sender need not concern itself with actual location of message recipient
Core Actor Semantic Properties

1. **State encapsulation**: no direct access to state of other actors

2. **Safe messaging**: messages have call-by-value semantics

3. **Fair scheduling**: messages are eventually delivered unless recipient is permanently disabled

4. **Location transparency**: sender need not concern itself with actual location of message recipient

5. **Mobility**: actors can move across network nodes
Actor Semantics vs. Actor Implementations

- Semantics *does not* prescribe mapping actors to objects or threads
- Many frameworks do not enforce encapsulation and lack mobility
- Some frameworks lack fairness and location transparency
- Programmers must *adapt* to each framework’s design choices
- Workarounds: type systems, middleware, testing, ...

```
thread
methods
mail queue
```

```
thread
methods
mail queue
```

References?

Immutable data?
Properties of Some Actor Implementations

<table>
<thead>
<tr>
<th></th>
<th>SALSA</th>
<th>Akka</th>
<th>Kilim</th>
<th>AF</th>
<th>Jetlang</th>
<th>Erlang</th>
</tr>
</thead>
<tbody>
<tr>
<td>State encapsulation</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Safe messaging</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Fair scheduling</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Location transparency</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mobility</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

‡ Karmani et al. Actor Frameworks for the JVM Platform: A Comparative Analysis. PPPJ’09
Properties of Some Actor Implementations*

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Actor mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALSA</td>
<td>JVM threads</td>
</tr>
<tr>
<td>Akka</td>
<td>JVM threads or light-weight tasks</td>
</tr>
<tr>
<td>Kilim</td>
<td>continuations</td>
</tr>
<tr>
<td>ActorFoundry</td>
<td>continuations</td>
</tr>
<tr>
<td>Jetlang</td>
<td>light-weight tasks</td>
</tr>
<tr>
<td>Erlang</td>
<td>light-weight tasks</td>
</tr>
</tbody>
</table>

*Karmani et al. Actor Frameworks for the JVM Platform: A Comparative Analysis. PPPJ’09*
Overhead of Fairness for (a) Threadring (b) Chameneos-redux (c) Naïve Fibonacci

*Karmani et al. Actor Frameworks for the JVM Platform: A Comparative Analysis. PPPJ’09*
Actor Semantics

Copying for Safe Messaging in a Single Node

Threadring performance without optimizations. $10^7$ message sends in a token ring of 503 concurrent entitles.

*Karmani et al., Actor Frameworks for the JVM Platform: A Comparative Analysis, PPPJ’09*
Local Message Send by Reference*

Threadring performance with optimizations

*Karmani et al. Actor Frameworks for the JVM Platform: A Comparative Analysis. PPPJ’09
Using deep copying to achieve safe messaging is expensive
Many messages have an ownership transfer semantics
Passing references in such cases is safe for shared memory
Conservative static analysis can reveal if message contents is compatible with ownership transfer

*Negara et al. Inferring Ownership Transfer for Efficient Message Passing. PPOPP’11
Improving Messaging Performance

<table>
<thead>
<tr>
<th>Program</th>
<th>Parameters</th>
<th>Improvement</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>threading</td>
<td>504 actors, 1 mil passes</td>
<td>92.7%</td>
<td>13.76</td>
</tr>
<tr>
<td>concurrent</td>
<td>601 actors</td>
<td>91.5%</td>
<td>11.73</td>
</tr>
<tr>
<td>copymessages</td>
<td>31810 actors, 10000 elements</td>
<td>52.0%</td>
<td>2.08</td>
</tr>
<tr>
<td>sor</td>
<td>6402 actors, 80 x 80 matrix</td>
<td>19.9%</td>
<td>1.25</td>
</tr>
<tr>
<td>chameneos</td>
<td>14 actors, 100000 rendezvous</td>
<td>35.6%</td>
<td>1.55</td>
</tr>
<tr>
<td>leader</td>
<td>30001 actors</td>
<td>41.7%</td>
<td>1.72</td>
</tr>
<tr>
<td>philosophers</td>
<td>60001 actors, 30000 philosophers</td>
<td>85.5%</td>
<td>6.92</td>
</tr>
<tr>
<td>pi</td>
<td>3002 actors, 30000 intervals</td>
<td>7.6%</td>
<td>1.08</td>
</tr>
<tr>
<td>quicksortCopy</td>
<td>200002 actors, 100000 elements</td>
<td>81.6%</td>
<td>5.44</td>
</tr>
<tr>
<td>quicksortCopy2</td>
<td>200002 actors, 100000 elements</td>
<td>70.2%</td>
<td>3.35</td>
</tr>
</tbody>
</table>

Performance improvements achieved by static inference of ownership transfer

†Negara, Karmani, and Agha, Inferring Ownership Transfer for Efficient Message Passing. PPOPP’11
Leveraging Actor Frameworks for the Cloud

Abstract Languages
(relational, event-based, ...)

semantics-preserving translations

Actor Frameworks
(Orleans, Akka, ...)

type systems, conversion tools

Cloud Platforms
(Azure, AWS, ...)

optimization, deployment

Legacy Programs

New Applications

Object-oriented Languages
(C#, Java, ...)

Gul Agha
Developing web applications using the Sunny language requires only:
- defining a data model (records), and
- client-server interactions (events).

*Events* can be augmented by *security policies* to prevent unauthorized data access, represented at runtime with low overhead.
Chat Application in the Sunny Language

**Record** Room {
  name: String,
  members: set User,
  msgs: set Msg
}

**Record** Msg {
  text: String,
  time: Timestamp,
  sender: User
}

**Event** JoinRoom(r: Room, u: User) on (not u in r.members) {
  r.members += u
}

**Event** SendMsg(r: Room, m: Msg) on (m.sender in r.members) {
  rmsgs += m
}
Chat Application After Deployment?

Client

SendMsg(..)  JoinRoom(..)

web/application servers, databases, ...

Room 1  ⋮  Room n

Msg  Msg  Msg
Chat Application Using Abstract Actors

Client

- SendMsg(..)
- JoinRoom(..)

Actor 1
- Room 1
  - Msg
  - Msg

Actor n
- Room n
  - Msg
Twitter-like Application in the Sunny Language

```sunny
record Peep {
    text: String,
    time: Timestamp
}

record User {
    handle: String,
    peeps: set Peep,
    followers: set User
}

event Follow(u: User, f: User) on (not f in u.followers) {
    u.followers += f
}

event AddPeep(s: String, u: User) {
    u.peeps += new Peep(s, time())
}
```
Twitter-like Application using Abstract Actors

- **Actor 1**
  - User 1
  - Peep
  - Peep

- **Actor n**
  - User n
  - Follower

- **Client**
  - Follow(..)
  - AddPeep(..)
Data model decomposition allows for scalable data storage

Events represented as client/server message exchanges at runtime

Concurrency/communication abstracted from application programmer

Distributing event processing among services, represented as mobile actors, allows scaling event throughput horizontally by adding more cloud servers

Mapping to services and compilation to actors enables trading availability for consistency
### Application Stack

#### Records & Events
- **record** Room ...
- **event** JoinRoom ...
- **record** Msg ...
- **event** SendMsg ...

#### Abstract Actors
- **actor** RoomService ...

#### Concrete Actors
- Actor 1 ...
- Actor n

#### Cloud
- Actor 1
- Actor 2 ...
- Actor n

**Programmer input**
- Decomposition of data and computations
- Replication, caching, and further decomposition
- Mobility, monitoring
Location independence and mobility enables resource management by spreading out actors over nodes and cores.

Through knowledge of state invariants, an actor can be *fissioned* into several actors, increasing parallelism.

Strategies for actor placement on cloud servers to minimize communication can be inferred by observing communication patterns.
Location independence and mobility enables resource management by spreading out actors over nodes and cores.

Through knowledge of state invariants, an actor can be fissioned into several actors, increasing parallelism.

Strategies for actor placement on cloud servers to minimize communication can be inferred by observing communication patterns.
If an object-oriented program’s concurrency semantics is known, one or more objects can be encapsulated in an actor.

Interaction between objects in different actors must be via call-by-value messages.

Many different object-actor decompositions are possible.

Libraries such as Akka’s Typed Actors for Java can seamlessly mix actors and objects.
Concurrent Semantics via Data-centric Synchronization

- **Data-centric synchronization**\(^\dagger\) has been proposed as an alternative to control-centric locks and monitors
- Class invariants are made explicit as *atomic sets* containing one or more fields
- Fields in an atomic set are implicitly accessed atomically
- *Aliases* and *unit of work* annotations extend atomic sets across class boundaries

\(^\dagger\) Vaziri et al. Associating Synchronization Constraints with Data in an Object-oriented Language. POPL’06
The Need for Inference of Concurrency Semantics

Conversion of legacy programs to use atomic sets requires understanding:

- class invariants
- existing synchronization

Conversion Experience of Dolby et al.

- Takes several hours for rather simple programs
- 2 out of 6 programs lack synchronization of some classes
- 2 out of 6 programs accidentally introduced global locks

Dolby et al. A Data-centric Approach to Synchronization. TOPLAS, 2012
The Need for Inference of Concurrency Semantics

Conversion of legacy programs to use atomic sets requires understanding:
- class invariants
- existing synchronization

Conversion Experience of Dolby et al.:
- Takes several hours for rather simple programs
- 2 out of 6 programs lack synchronization of some classes
- 2 out of 6 programs accidentally introduced global locks

---

*Dolby et al. A Data-centric Approach to Synchronization. TOPLAS, 2012*
Synopsis of a Probabilistic Algorithm for Dynamically Inferring Atomic Sets, Aliases, and Units of Work

Assumptions about Input Programs

- Methods perform meaningful operations (convey intent)
- Fields that a method accesses are likely connected by invariant
Synopsis of a Probabilistic Algorithm for Dynamically Inferring Atomic Sets, Aliases, and Units of Work

Assumptions about Input Programs

- Methods perform meaningful operations (convey *intent*)
- Fields that a method accesses are likely connected by invariant

Algorithm Idea

- Observe which pairs of fields a method accesses atomically and their distance in terms of basic operations
  - This is (Bayesian) *evidence* that fields are connected through an invariant
- Store current beliefs for all field pairs in *affinity matrices*
Actorizing Programs Annotated with Atomic Sets

- Key property: messages to actors are processed *one at a time*
- Fields in one atomic set *should not* span two actors at runtime
- An actor encapsulates one or more objects with atomic sets
Proposed Tool Chain for Actorization

Program -> tool -> Annotations

Annotated Program -> Actor Program

Actor Program -> Program Using Actor Library
public class List {
    private int size;
    private Object[] elements;

    public int size() {
        return size;
    }

    public Object get(int i) {
        if (0 <= i && i < size)
            return elements[i];
        else
            return null;
    }

    /* ... */
}

public class DownloadManager {
    private List urls;

    public synchronized URL getNextURL() {
        if (urls.size() == 0)
            return null;
        URL url = (URL) urls.get(0);
        urls.remove(0);
        return url;
    }

    /* ... */
}
public class DownloadThread extends Thread {
    private DownloadManager manager;
    public void run() {
        URL url;
        while((url = this.manager.getNextURL()) != null) {
            download(url);
        }
        /* ... */
    }
}

public class Download {
    public static void main(String[] args) {
        DownloadManager manager = new DownloadManager();
        for (int i = 0; i < 128; i++) {
            manager.addURL(new URL("http://www.example.com/f" + i));
        }
        DownloadThread t1 = new DownloadThread(manager);
        DownloadThread t2 = new DownloadThread(manager);
        t1.start();
        t2.start();
    }
}
@AtomicSets(\{"L\}\)

```java
public class List {
    private @Atomic("L") int size;
    private @Atomic("L") Object[] elements;

    public int size() {
        return size;
    }

    public Object get(int i) {
        if (0 <= i && i < size)
            return elements[i];
        else
            return null;
    }
}
```

@AtomicSets(\{"M\}\)

```java
public class DownloadManager {
    private @Atomic("M")
        @Aliased("L") List urls;

    public URL getNextURL() {
        if (urls.size() == 0)
            return null;
        URL url = (URL) urls.get(0);
        urls.remove(0);
        return url;
    }
}
```
public class DownloadThread extends Thread {
    private @Actor DownloadManager manager;
    public void run() {
        URL url;
        while ((url = this.manager.getNextURL()) != null) {
            download(url);
        }
    }
    /* ... */
}

public class Download {
    public static void main(String[] args) {
        DownloadManager manager = new @Actor DownloadManager();
        for (int i = 0; i < 128; i++) {
            manager.addURL(new URL("http://www.example.com/f" + i));
        }
        DownloadThread t1 = new @Actor DownloadThread(manager);
        DownloadThread t2 = new @Actor DownloadThread(manager);
        t1.start();
        t2.start();
    }
}
Adaptable Cloud-based Actor Programs

- Atomic sets capture small-scale concurrency semantics
- *Session types* can describe large-scale message passing behavior
- Program monitoring output useful for inference of semantic properties
- Inferred properties can be enforced through program synthesis
A Control Loop for Adaptable Cloud-based Programs

- Inference
- Adaptive Programs
- Monitoring
- Properties
- Synthesis & Verification
Acknowledgements

OSL collaborators
Rajesh Karmani, Peter Dinges, Minas Charalambides, Karl Palmskog, Amin Shali among many others.

Other current collaborators
Darko Marinov, Daniel Jackson

Research partially funded by:
- NSF grant number CCF-1438982
- AFOSR contract FA 9750-11-2-0084

Slides prepared with assistance from Karl Palmskog
References


