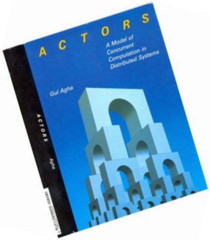
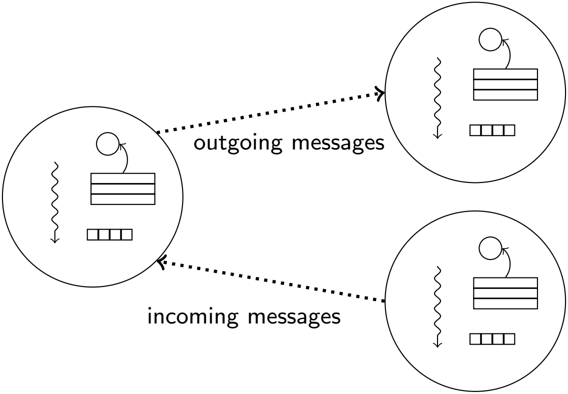
Leveraging Actor Frameworks for the Cloud

Gul Agha
University of Illinois at Urbana-Champaign



Introduction

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)

Actor Behavior

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

Actors: Scalable Concurrency

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 *

*https://www.facebook.com/notes/facebook-engineering/chat-stability-and-scalability/51412338919

Actors: Scalable Concurrency II

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

Core Actor Semantic Properties

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

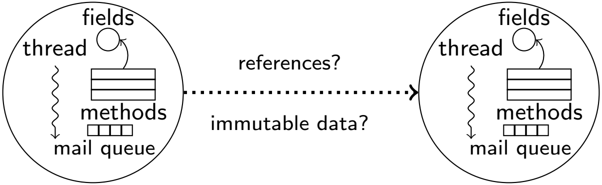
1. **State encapsulation**: no direct access to state of other actors
2. **Safe messaging**: messages have call-by-value semantics
3. **Fair scheduling**: messages are eventually delivered unless recipient is permanently disabled
4. **Location transparency**: sender need not concern itself with actual 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, ...



Properties of Some Actor Implementations[‡]

	SALSA	Akka	Kilim	AF	Jetlang	Erlang
State encapsulation	✓	✗	✗	✓	✓	✓
Safe messaging	✓	✗	✗	✓	✗	✓
Fair scheduling	✓	✓	✗	✓	✗	✓
Location transparency	✓	✓	✗	✓	✓	✓
Mobility	✓	✗	✗	✓	✗	✗

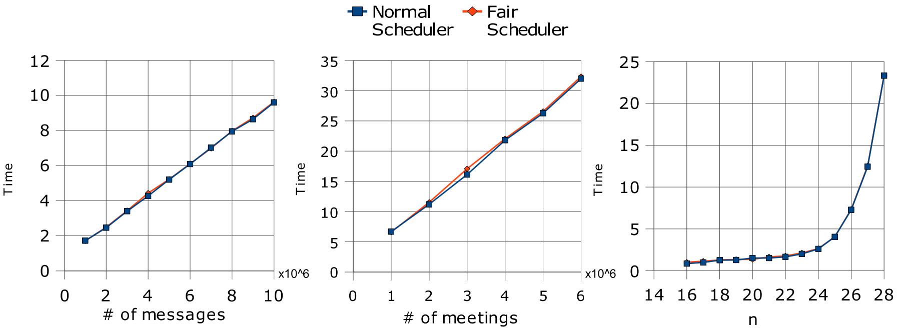
[‡] Karmani et al. Actor Frameworks for the JVM Platform: A Comparative Analysis. PPPJ'09

Properties of Some Actor Implementations*

Implementation	Actor mapping
SALSA	JVM threads
Akka	JVM threads or light-weight tasks
Kilim	continuations
ActorFoundry	continuations
Jetlang	light-weight tasks
Erlang	light-weight tasks

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

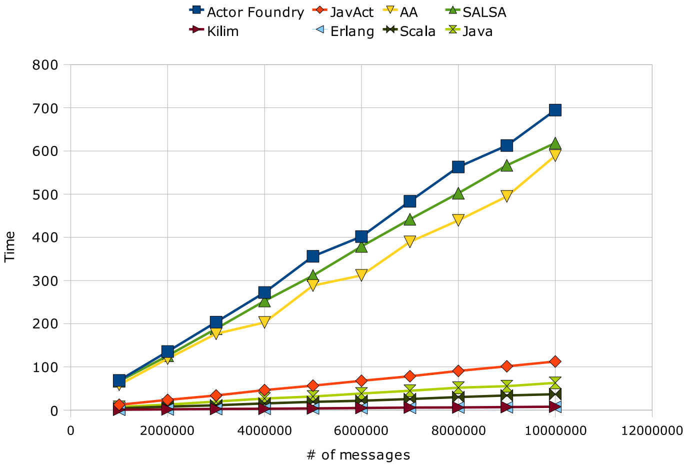
Fairness and Performance*



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

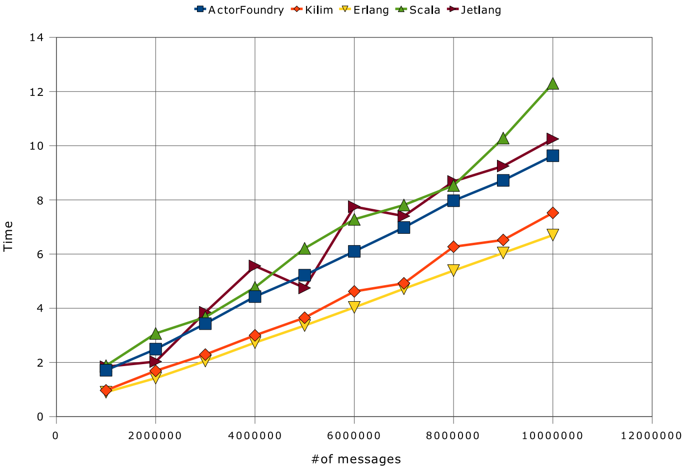
Copying for Safe Messaging in a Single Node *



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

Improving Local Messaging Performance*

- 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

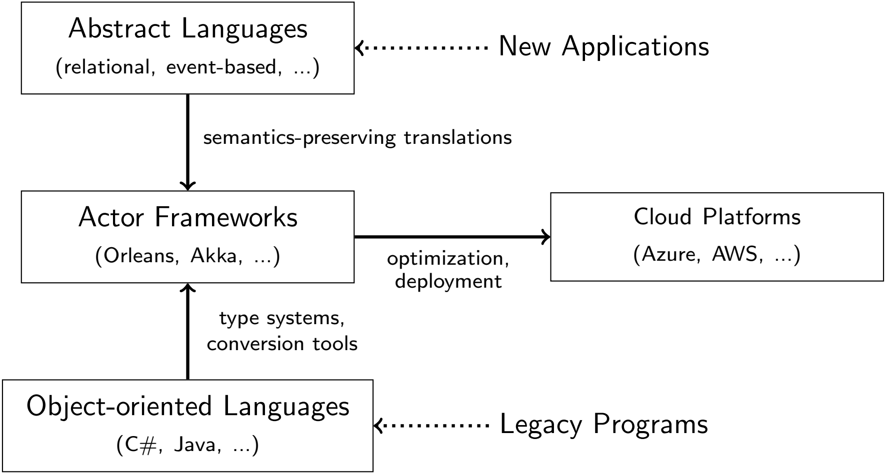
Improving Messaging Performance†

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

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



Example: Cloud-based Web Programming with Sunny

- 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

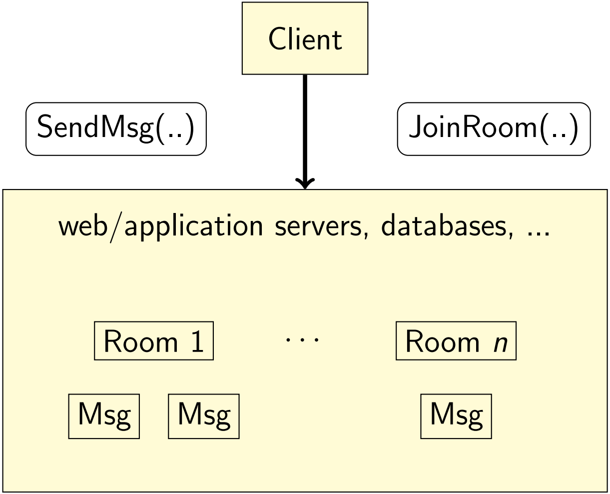
```
record Room {
  name: String,
  members: set User,
  msgs: set Msg
}
```

```
record Msg {
  text: String,
  time: Timestamp,
  sender: User
}
```

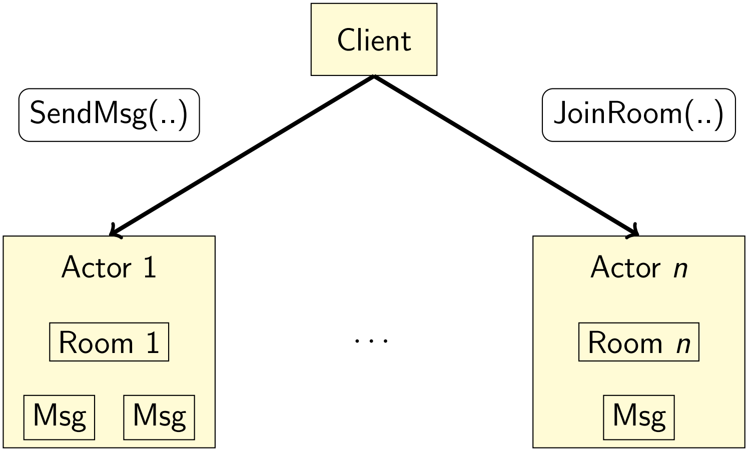
```
event JoinRoom(r: Room, u: User)
 on (not u in r.members) {
  r.members += u
}
```

```
event SendMsg(r: Room, m: Msg)
  on (m.sender in r.members) {
  r.msgs += m
}
```

Chat Application After Deployment?



Chat Application Using Abstract Actors



Twitter-like Application in the Sunny Language

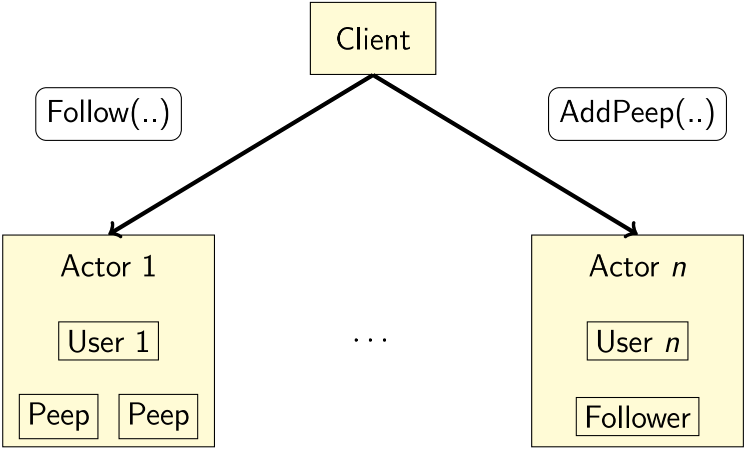
```
record Peep {
  text: String,
  time: Timestamp
}
```

```
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())
}
```

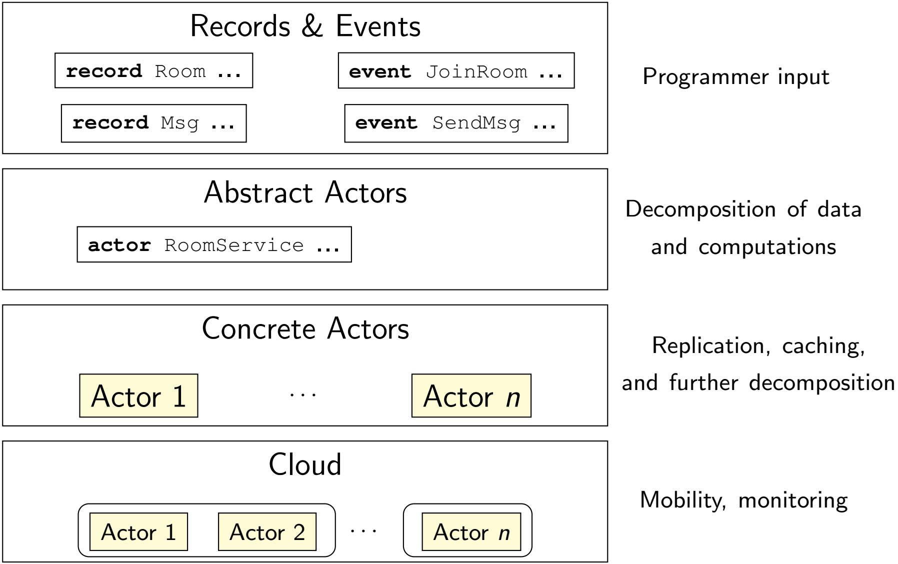
Twitter-like Application using Abstract Actors



Application Scalability

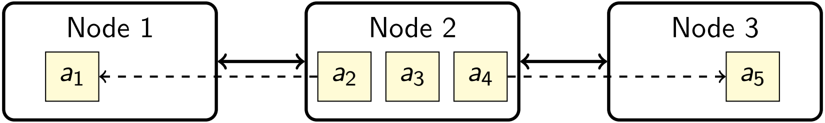
- Data model decomposition allows for scalable data storage
- Events represented as client/server message exchanges at runtime
- Concurrency/communication abstracted from application programmer
- Distributing event processing among services, represented as mobile actors, allows scaling event throughput horizontally by adding more cloud servers
- Mapping to services and compilation to actors enables trading availability for consistency

Application Stack



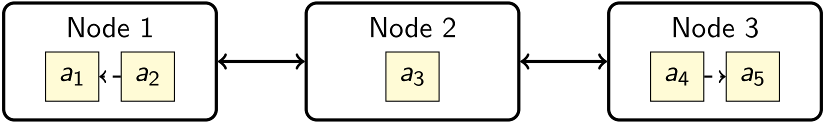
Scaling at Runtime

- 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



Scaling at Runtime

- 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



Legacy Object-oriented Programs and Actors

- 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

Concurrency Semantics via Data-centric Synchronization

- *Data-centric synchronization*‡ 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

‡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

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

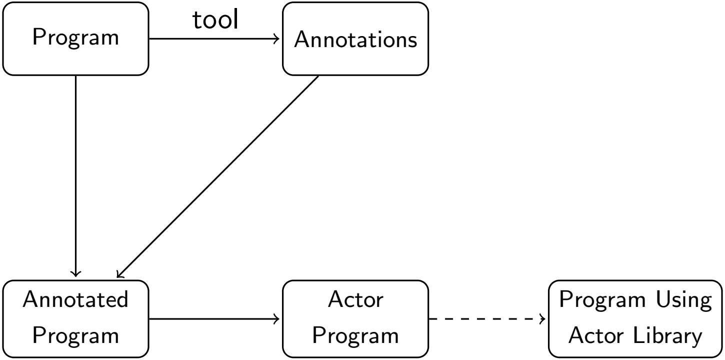
Algorithm Idea

- 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



Example Java Legacy Program

```
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;
  }
  /* ... */
}
```

Example Java Legacy Program

```
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();
 }
}
```

Converted Program with Java 8 Type Annotations

```
@AtomicSets({"L"})
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"})
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;
  }
  /* ... */
}
```

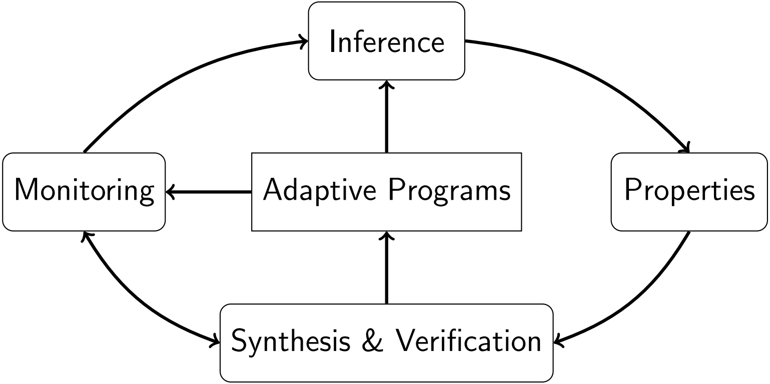
Converted Program with Java 8 Type Annotations

```java
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



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 I

Peter Dinges, Karl Palmskog, and Gul Agha.
Automated inference of atomic sets for safe concurrent execution.
http://www.sti.uniurb.it/events/sfm15mp/slides/agha.pdf.

Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek.
A data-centric approach to synchronization.
ACM TOPLAS, 34(1):4, 2012.

Rajesh K. Karmani, Amin Shali, and Gul Agha.
Actor frameworks for the jvm platform: a comparative analysis.
In *PPPJ*, pages 11–20, 2009.

Stas Negara, Rajesh K. Karmani, and Gul A. Agha.
Inferring ownership transfer for efficient message passing.
In *PPOPP*, pages 81–90, 2011.

Mandana Vaziri, Frank Tip, and Julian Dolby.
Associating synchronization constraints with data in an object-oriented language.
In *POPL '06*, pages 334–345, 2006.