A Language and Architecture for Distributed Computing over the Internet

Carlos A. Varela

Worldwide Computing Lab

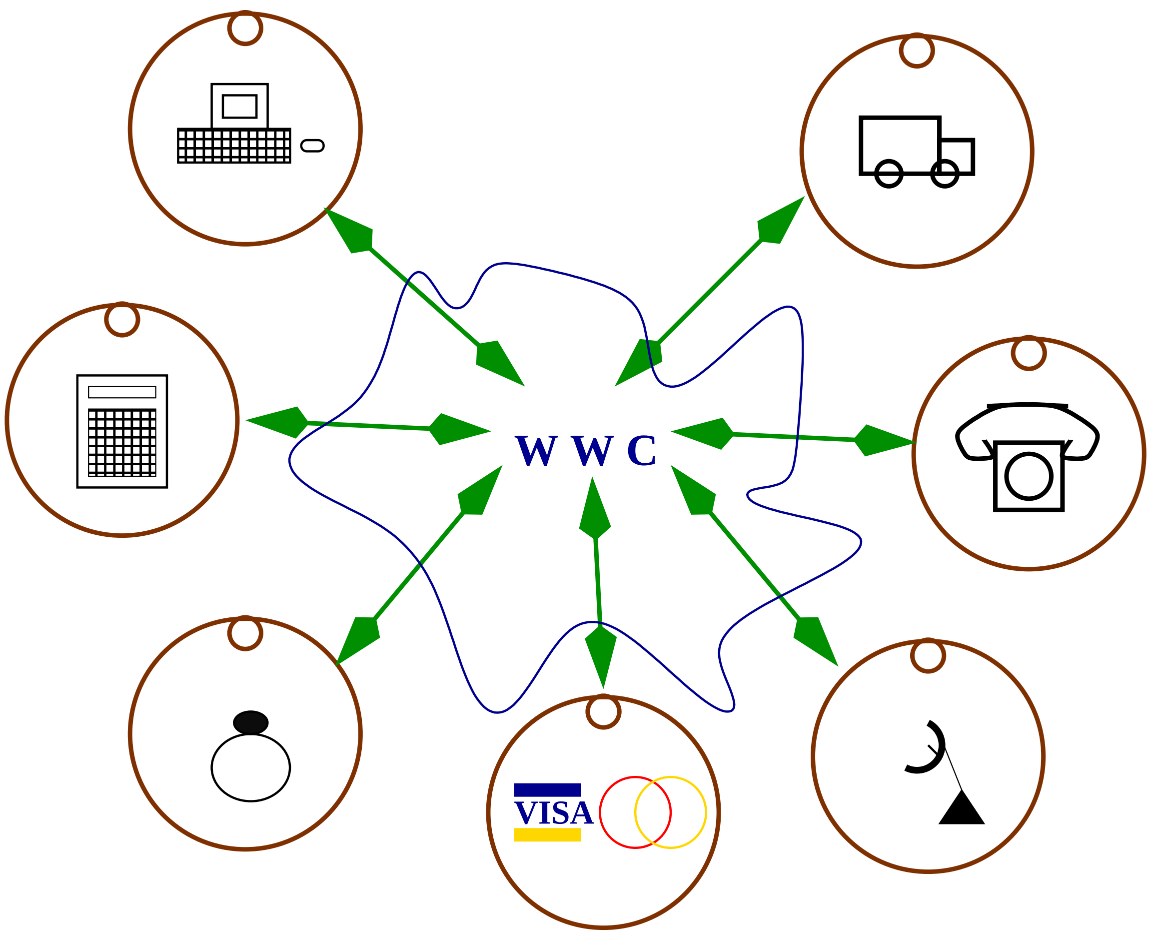
Department of Computer Science

Rensselaer Polytechnic Institute

http://www.cs.rpi.edu/wwc/

July 2002

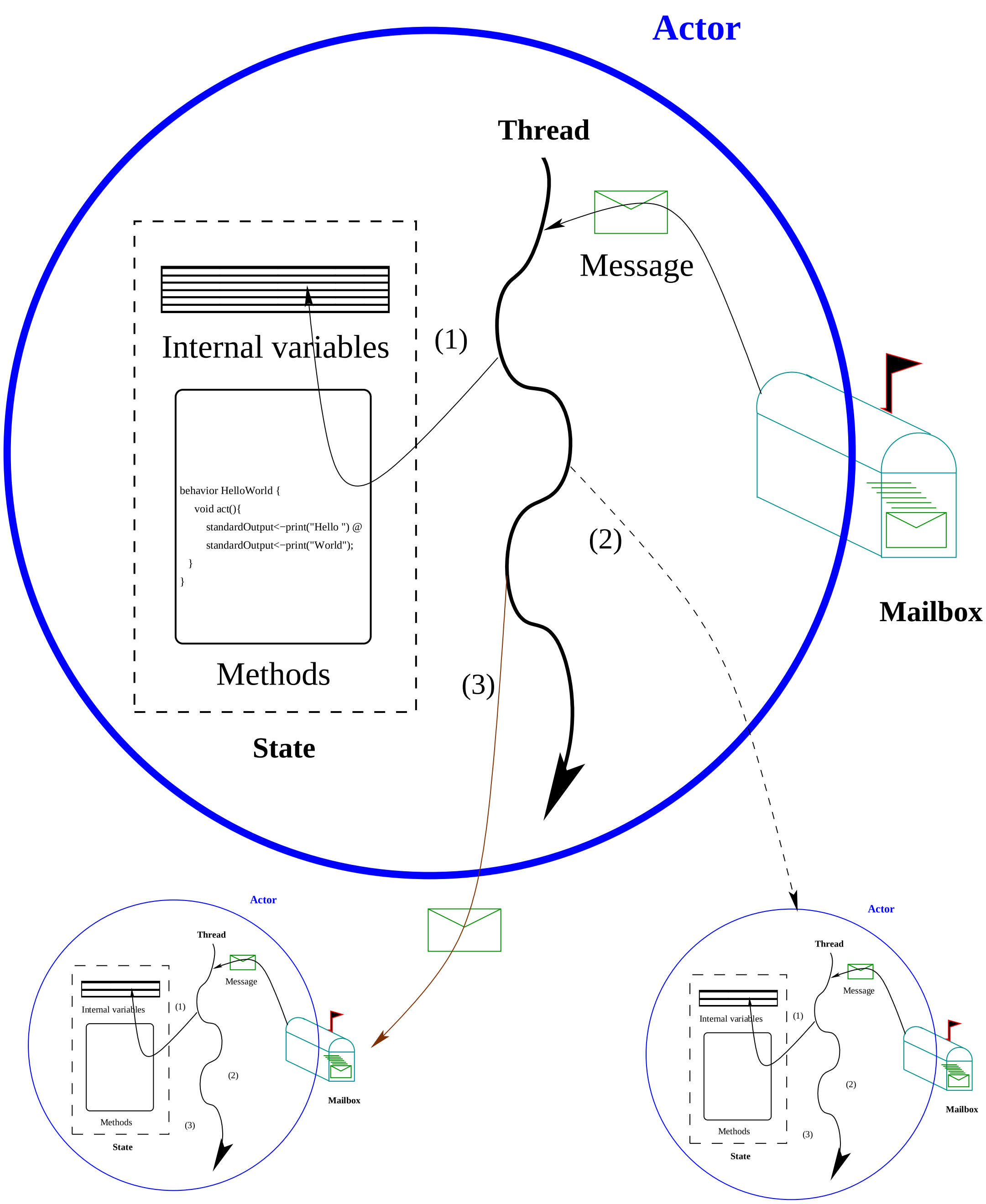
Worldwide Computing



Limitations of Java for Worldwide Computing

- passive objects
- shared memory
- synchronous communication
- non-universal naming
- only primitive synchronization mechanisms

Actor (Agent) Model



Actor (Agent) Model (continued)

Actor mobility is simpler than object mobility:

- distributed memory
- universal naming

Coordination of actors in worldwide open systems remains difficult.

- non-blocking, asynchronous communication
- inherent concurrency – threads are encapsulated in objects

Worldwide Computing Architecture

The World-Wide Computer consists of concurrent, distributed, and mobile Universal Actors.

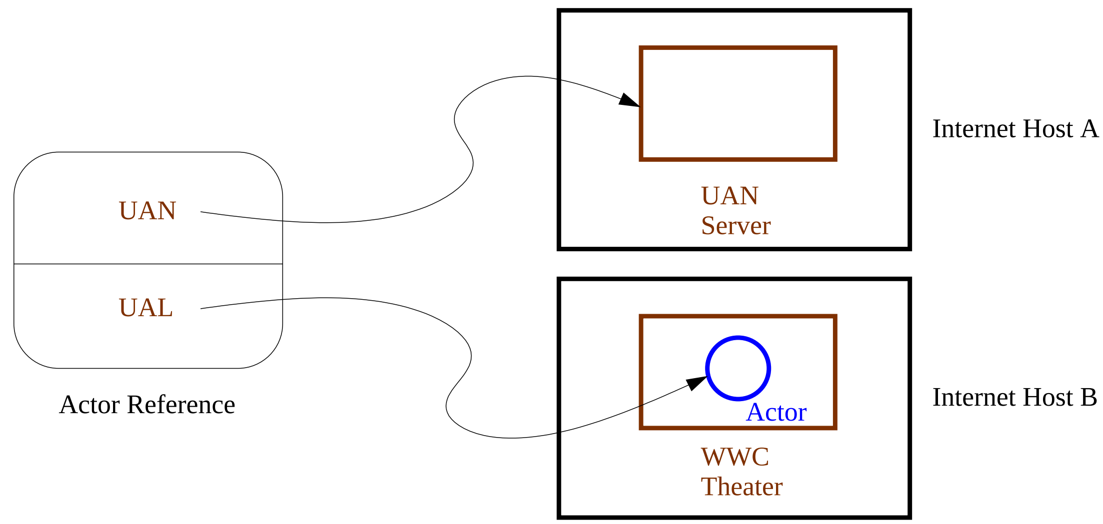
- Universal naming strategy
- Run-time support: Theaters
- Remote communication protocol
- Migration support
- Preliminary performance results

Naming in Worldwide Computing (Requirements)

The main goals of naming in worldwide computing are to provide:

- platform independence – names should appear coherent on all nodes independently of underlying architecture
- scalability of name space management
- transparent actor migration
- openness by allowing unanticipated actor reference creation and protocols that provide access through names
- both human and computer readability

Proposed Solution: Universal Actor Names and Locators



- Uniform Resource Identifiers (URI) syntax [Berners-Lee]
- UAN/UAL support transparent actor migration.

Sample UAN:

```
uan://wwc.osl.cs.uiuc.edu:3030/Agha/Calendar
```

Sample Universal Actor Locators for this WWC actor:

```
rmsp://agha.cs.uiuc.edu:4040/Agents/Calendar
rmsp://howard.cs.uiuc.edu:4040/AghaCalendar
rmsp://agha.pda.com:1234/Calendar
```

Universal Actor Naming Protocol

The UANP defines how to interact with the WWC Naming Service. Similarly to HTTP, UANP is text-based, and includes methods for the following actions:

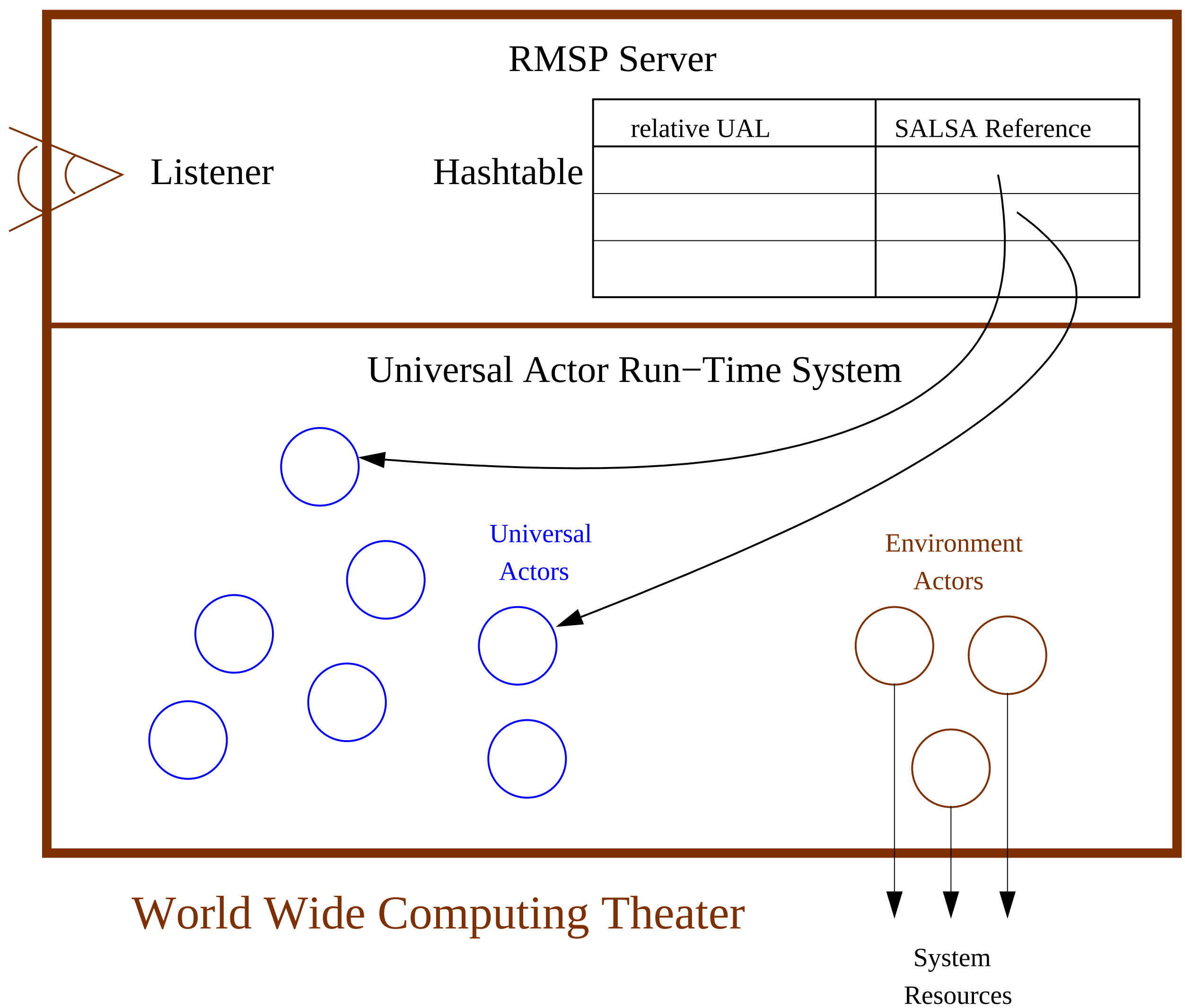
Method	Parameters	Action
PUT	relative UAN, UAL	Creates a new entry in the database
GET	relative UAN	Returns the UAL entry in the database
DELETE	relative UAN	Deletes the entry in the database
UPDATE	relative UAN, UAL	Updates the UAL entry in the database

An actor's location can be cached for faster future accesses.

World-Wide Computer Theaters

A WWC Theater provides runtime support to Universal Actors. A Theater contains:

- a remote communication module with a hashtable mapping relative UALs to actual SALSA actor references, and
- a runtime system for universal and environment actors.



Remote Communication in Worldwide Computing (Requirements)

The main goals of a remote communication protocol in worldwide computing are to provide:

- asynchronous, non-blocking communication
- an interface to the naming service for target actor location
- data and code mobility

Proposed Solution: Remote Message Sending Protocol

- RMSP defines how an actor sends a message to any other actor in the World-Wide Computer

- RMSP is object-based, and includes support for message serialization, and actor migration

- transparent programming language support for sending messages to local and remote actors

Migration in Worldwide Computing (Requirements)

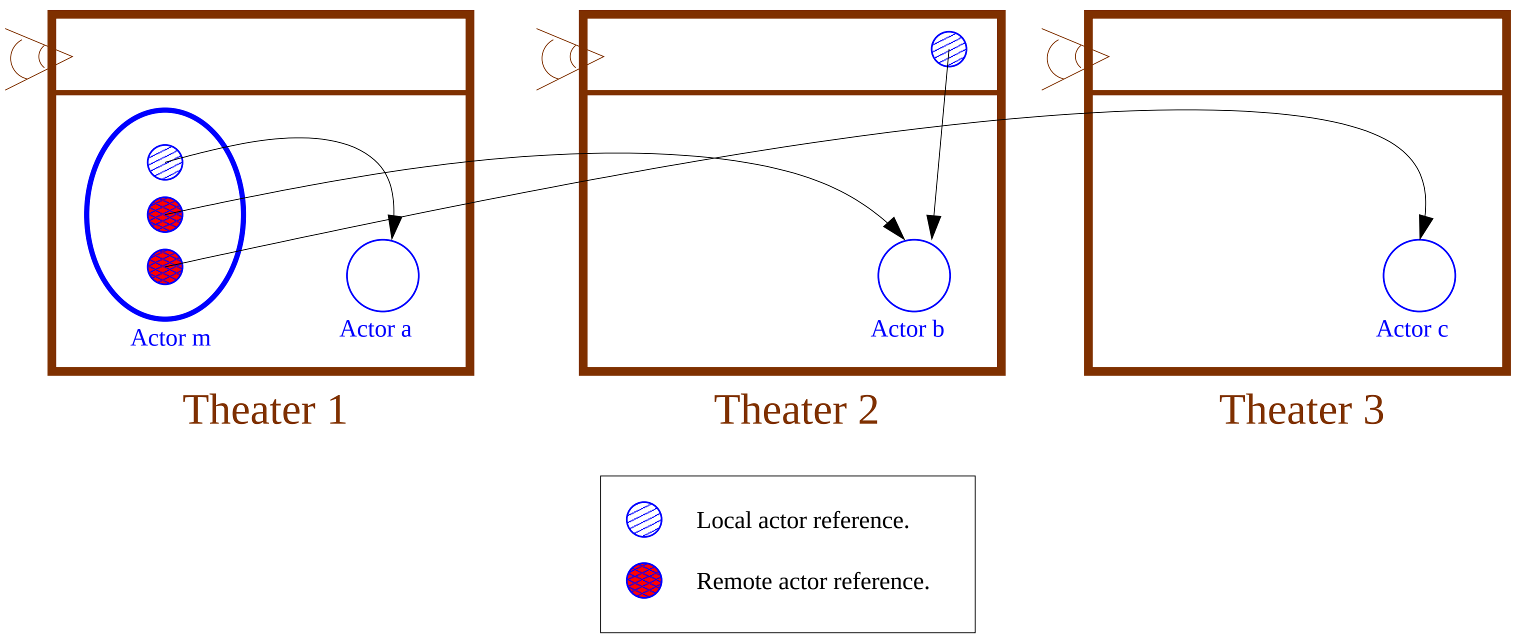
The main goals of migration in worldwide computing are to provide:

- both fine-grained and coarse-grained mobility
- actor reference updating for more efficient local communication
- consistency protocols for shared memory.

Since actors do not have shared memory, actor migration is simpler and more efficient than object/thread migration.

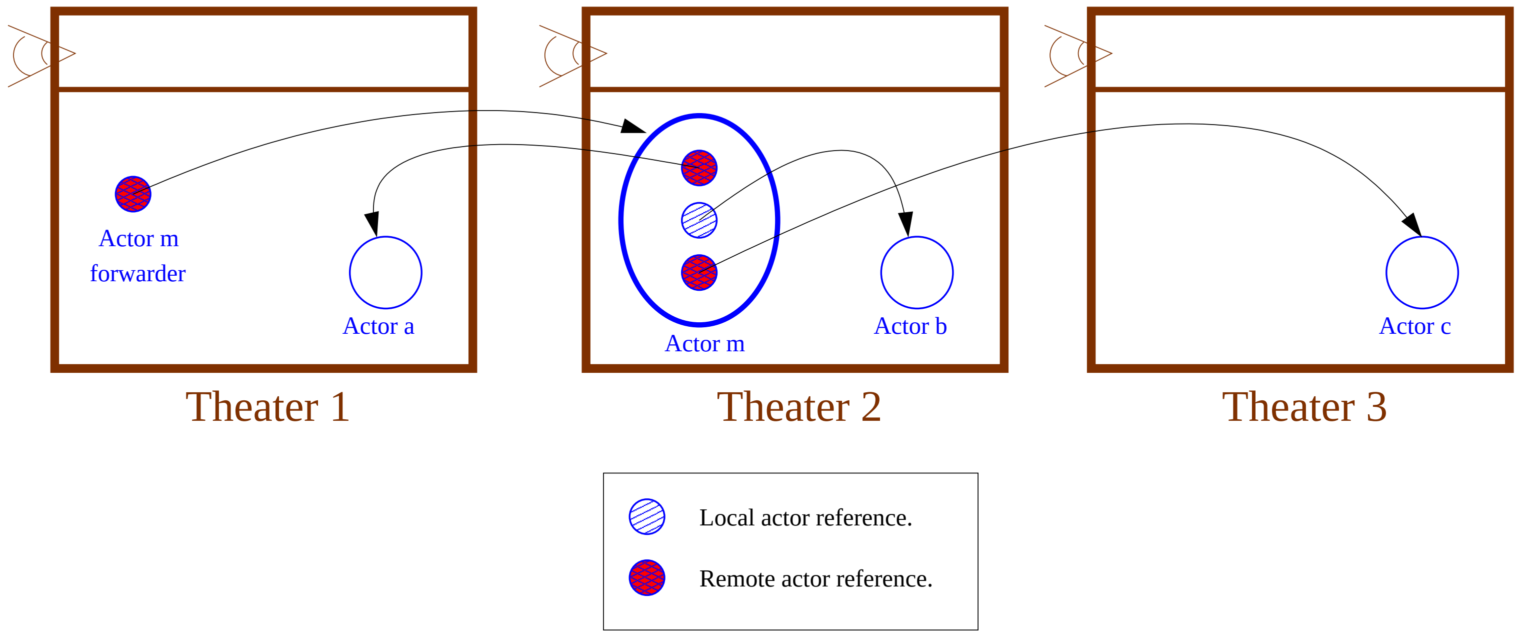
Actor Migration

Before migration of actor m from Theater 1 to Theater 2, its references to actors b and c are remote, while its reference to actor a is local.



Actor Migration (continued)

After migration of actor `m`, its reference to actor `a` becomes remote and its reference to actor `b` becomes local. Its reference to actor `c` remains unchanged.



World-Wide Computer Testbed

Machine Name	Location	OS-JVM	Processor
yangtze.cs.uiuc.edu	Urbana, IL, USA	Solaris 2.5.1-JDK 1.1.6	Ultra 2
vulcain.ecoledoc.lip6.fr	Paris, France	Linux 2.2.5-JDK 1.2pre2	PII, 350MHz
solar.isr.co.jp	Tokyo, Japan	Solaris 2.6-JDK 1.1.6	Sparc 20

Time Ranges (with SALSA 0.3.2)

Local actor creation time	386μs
Local message sending time	148μs
LAN message sending time	30-60ms
WAN message sending time	2-3 secs
LAN actor migration time	150-160ms (minimal actor)
LAN actor migration time	240-250ms (actor with 100Kb of data)
WAN actor migration time	3-7secs (minimal actor)
WAN actor migration time	25-30secs (actor with 100Kb of data)

Simple Actor Language, System and Architecture

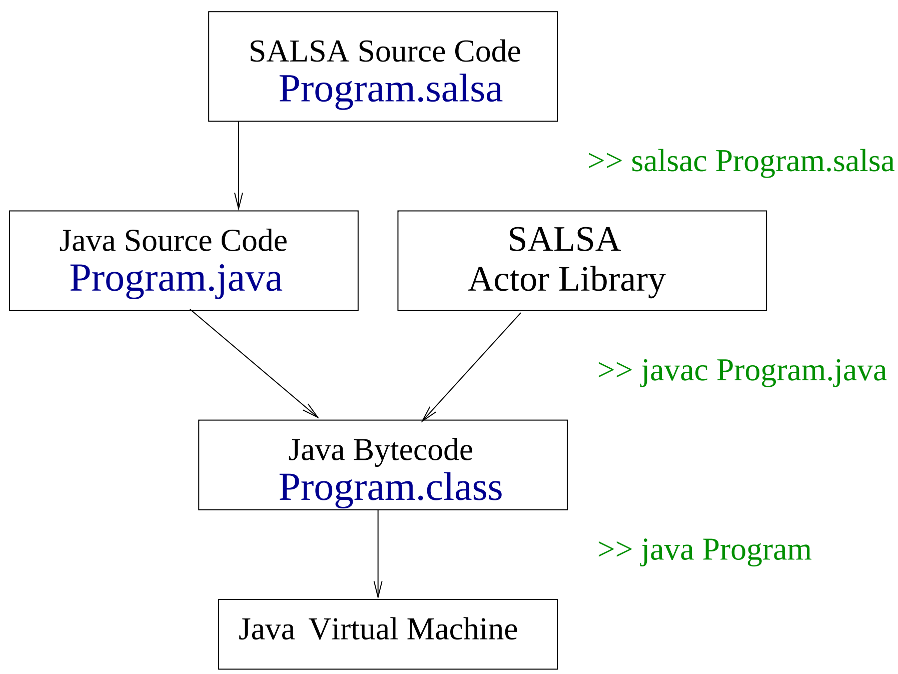
SALSA is a dialect of Java, with support for:

- Concurrent programming with actors.
- Token-passing Continuations to control concurrency by specifying customers for an actor message's return value.
- Join Continuations to provide a synchronization barrier for multiple actor computations into a single continuation.
- Universal Naming to bind and locate actors in the WWC using UANs and UALs.
- RMSP and Migration to send messages to remote actors and to migrate actors across WWC Theaters.

Language Implementation

- SALSA Actor Library
- Join Continuation Code Generation

SALSA Architecture



SALSA Hello World Example

```
module helloworld;

behavior HelloWorld {
    void act(String arguments[]){
        standardOutput <- print("Hello ") @
        standardOutput <- println("World!");
    }
}
```

Actor Model Support

- All SALSA behaviors inherit from the `UniversalActor` class. To create a new actor instance:

  ```
  HelloWorld helloWorld = new HelloWorld();
  ```

- To send messages to acquaintance actors:

  ```
  acquaintance <- message(arg1, arg2,...);
  ```

  For example:

  ```
  standardOutput <- print("Hello ");
  hello();
  ```

- Sending a message returns immediately (it is asynchronous) with a `void` return value.

- Only an actor itself can change its internal state through assignments to its instance variables. No shared memory or static variables are allowed in SALSA.

Token-Passing Continuations

- SALSA messages are potential Java method invocations. Message passing is asynchronous.
- Continuations allow to specify a customer for a message's return value (called `token`):

  ```
  acquaintance<-m1(args) @
  customer<-m2(arg0, ... ,token, ..., argn);
  ```

  For example:

  ```
  fractal<-computePixel() @
  screen<-draw(token);
  ```

- The return type of `computePixel()` needs to match the formal argument type of `draw(argument)`. SALSA allows method overloading and will choose the most specific method according to the token run-time type.
- `a<-m` with no arguments, is syntactic sugar for `a<-m(token)`. For example:

  ```
  fractal<-computePixel() @ screen<-draw;
  ```

- It is possible to ”chain” continuations. For example:

  ```
  a1<-m1() @ a2<-m2 @ a3<-m3(token,10);
  ```

Join Continuations

- A join statement allows to provide a synchronization barrier for multiple actor computations into a single continuation:

```
join(a1<-m1(args), a2<-m2(args), ... ) @ customer <- n;

join(actorArray<-m()) @ customer <- n;
```

For example:

```
join(author1<-writeChapter(1), author2<-writeChapter(2)) @
editor<-review @ publisher<-print;
```

will only send the message `review(token)` to the `editor` when both authors have finished writing their chapters. The token passed as an argument is an array containing the return values of the messages inside the `join` statement.

Universal Actor Naming in SALSA

```
behavior Agent {

    void printItinerary(){...}

    void act(String[] args){
        Agent a = new Agent();
        try {
            a<-bind("uan://yangtze.cs.uiuc.edu:3030/agent",
                    "rmsp://yangtze.cs.uiuc.edu:4040/agent");
        } catch (Exception e){
            standardOutput<-println(e);
        }
    }
}
```

RMSP and Actor Migration in SALSA

Getting a remote actor reference by name and sending a message:

```
Agent a = new Agent();
a<-getReferenceByName("uan://yangtze.cs.uiuc.edu/agent") @
a<-printItinerary();
```

Getting the reference by location:

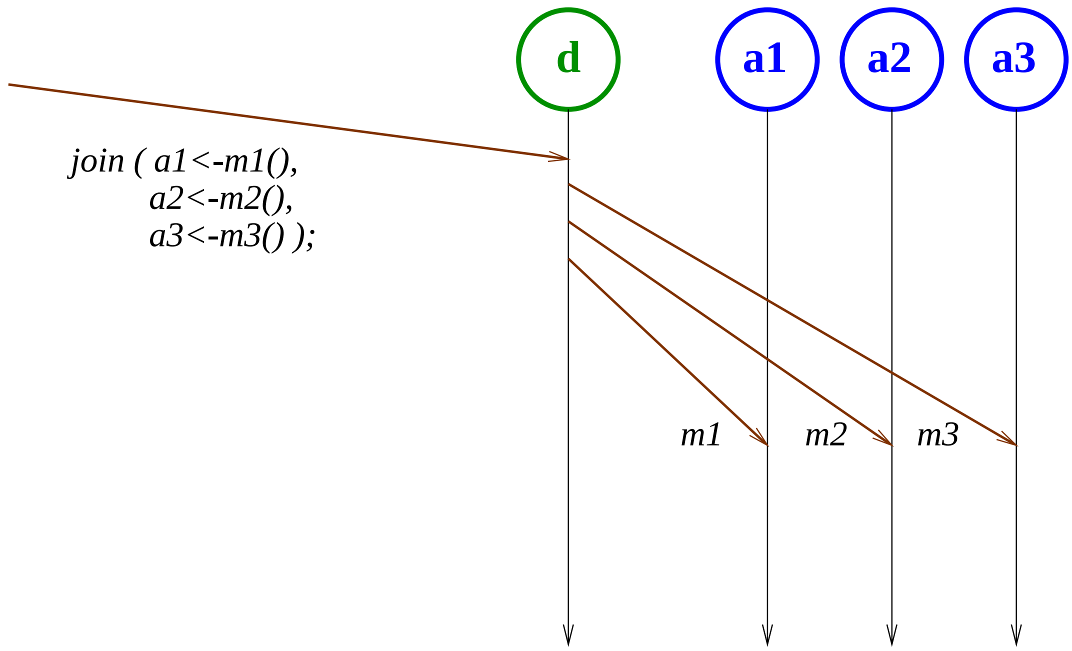
```
Agent a = new Agent();
a<-getReferenceByLocation("rmsp://yangtze.cs.uiuc.edu/agent") @
a<-printItinerary();
```

Migrating an agent to a remote WWC Theater:

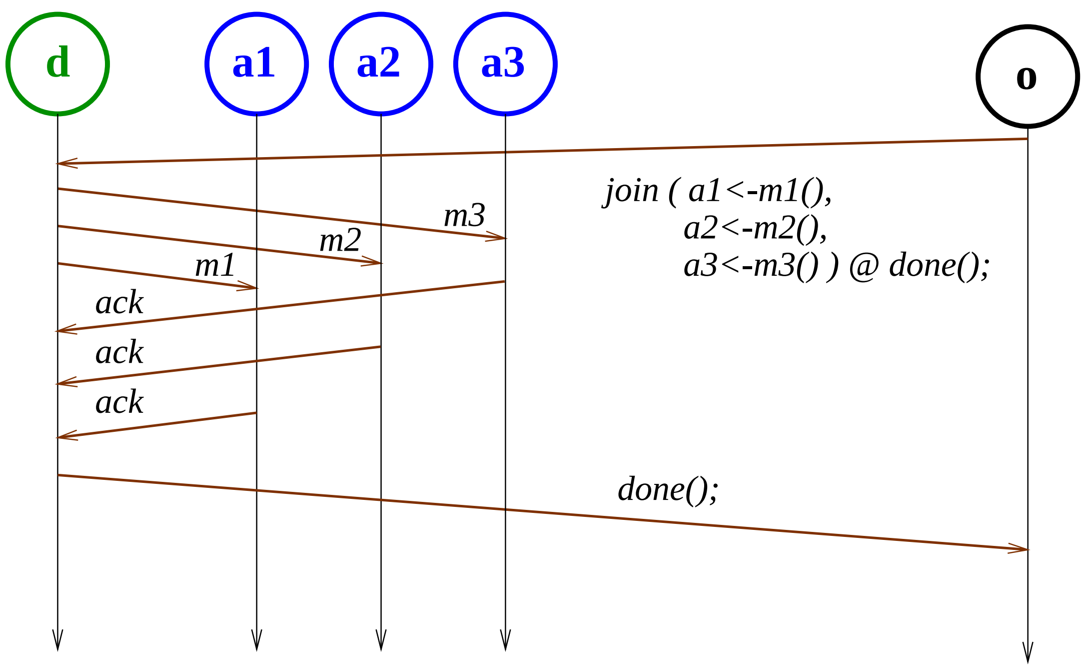
```
Agent a = new Agent();
a<-getReferenceByName("uan://yangtze.cs.uiuc.edu/agent") @
a<-migrate("rmsp://vulcain.ecoledoc.lip6.fr/agent");
```

Example 1: Multicast Protocols

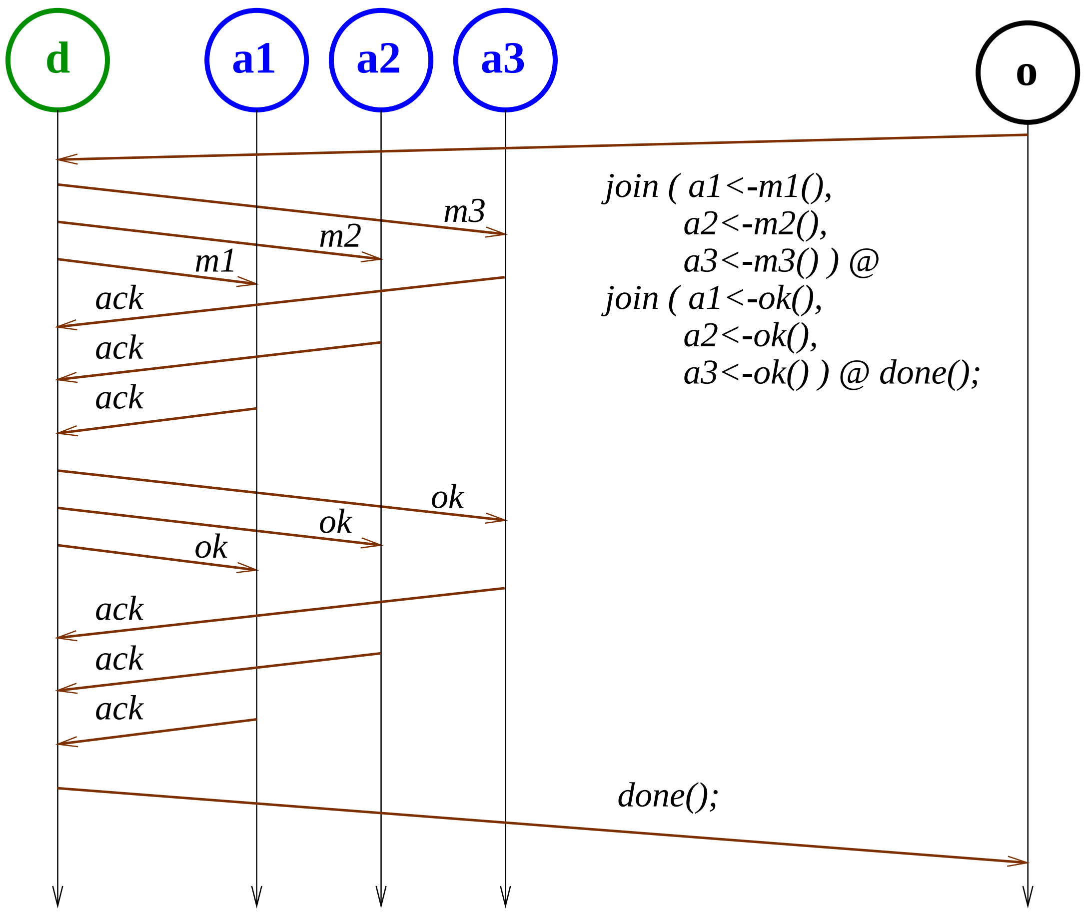
Multicast



Acknowledged multicast



Group knowledge multicast (Fagin, Halpern, Moses and Vardi, 1995)

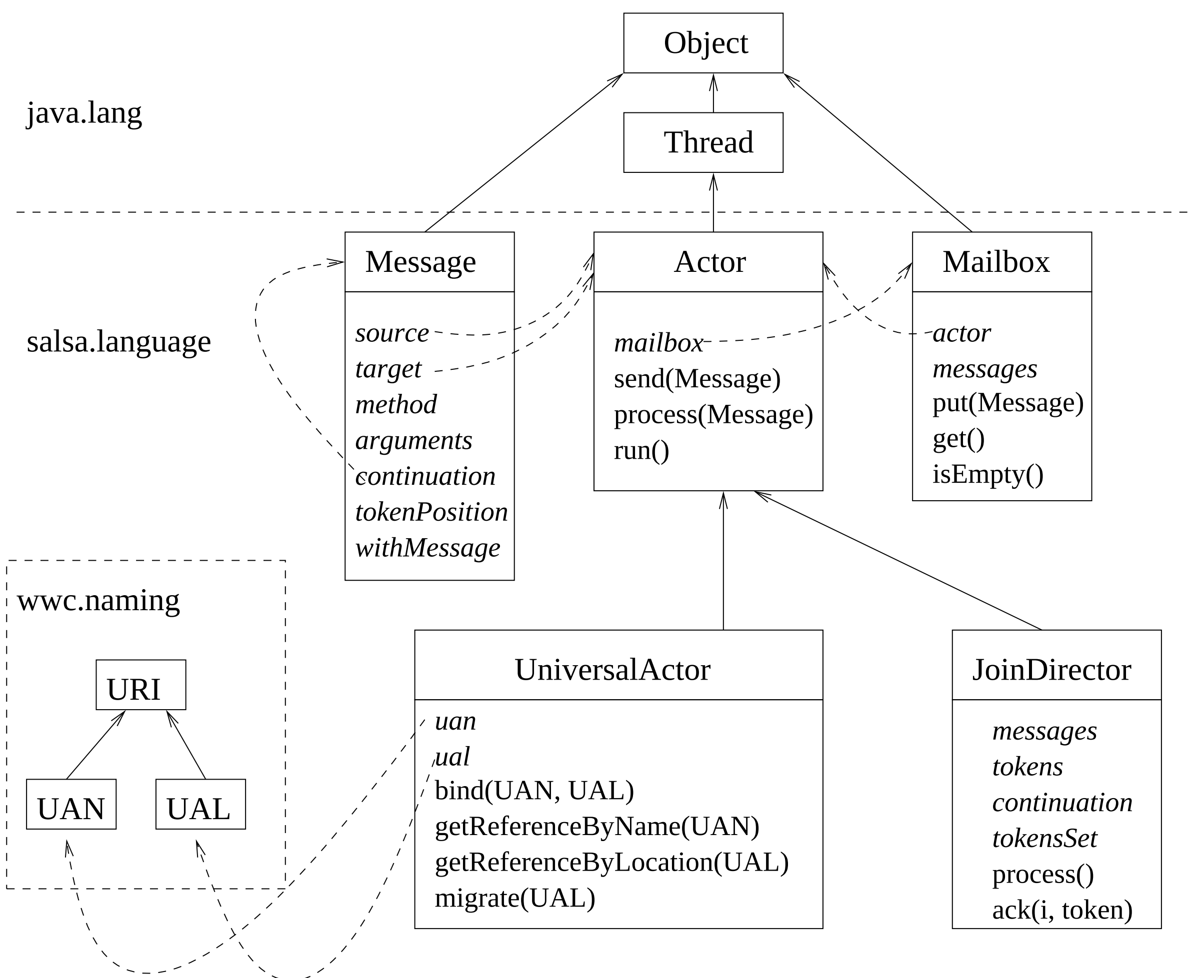



```
join ( a1<-m1(),
       a2<-m2(),
       a3<-m3() ) @
join ( a1<-ok(),
       a2<-ok(),
       a3<-ok() ) @ done();
```

Lines of Code Comparison for Multicast Protocols

	Foundry	SALSA	Java
Shared Code	40	10	34
Basic Multicast	146	27	115
Acknowledged Multicast	60	21	134
Group-knowledge Multicast	73	24	183
TOTAL	319	82	466

SALSA Actor Library



Join Continuation Code Generation