July 2002

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

Rensselaer Polytechnic Institute
Department of Computer Science
Worldwide Computing Lab
Carlos A. Varela

The Internet
A Language and Architecture for Distributed Computing over
Limitations of Java for Worldwide Computing

- only primitive synchronization mechanisms
- non-universal naming
- synchronous communication
- shared memory
- passive objects
Actor (Agent) Model

Actor

Message

Thread

Mailbox

Internal variables

State

Methods

standardOutput<−print("World");
standardOutput<−print("Hello ");

behavior HelloWorld {
void act() {
}
}
Inherent concurrency – threads are encapsulated in objects

Non-blocking, asynchronous communication

Coordination of actors in worldwide open systems remains difficult.

Universal naming

Distributed memory

Actor mobility is simpler than object mobility:

Actor (Agent) Model (continued)
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
The main goals of naming in worldwide computing are to provide:

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

Naming in Worldwide Computing (Requirements)
Proposed Solution: Universal Actor Names

Sample Universal Actor Locators for this WWC actor:

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

Sample UAN:

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

- UAN/UAL support transparent actor migration.
- Uniform Resource Identifiers (URI) syntax (Berners-Lee)
Universal Actor Naming Protocol

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>relative UAN, UAL</td>
<td>Updates the UAL entry in the database</td>
</tr>
<tr>
<td>GET</td>
<td>relative UAN</td>
<td>Deletes the entry in the database</td>
</tr>
<tr>
<td>DELETE</td>
<td>relative UAN</td>
<td>Returns the UAL entry in the database</td>
</tr>
<tr>
<td>PUT</td>
<td>relative UAN, UAL</td>
<td>Creates a new entry in the database</td>
</tr>
</tbody>
</table>

HTTP, UANP is text-based, and includes methods for the following actions:

- **GET**
- **PUT**
- **DELETE**
- **UPDATE**

Similar to the WWC Naming Service, the UANP defines how to interact with the WWC Naming Service.
A 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.

A Theater contains:

- A Universal Actor Runtime System
- A Listener Hashtable

Resources

World-Wide Computer Theaters

Universal Actor Runtime System

Listener

RMSP Server

Hashtable

Relative UAL

SALSA Reference
Remote Communication in Worldwide Computing

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

- Asynchronous, non-blocking communication
- An interface to the naming service for target actor location
- Data and code mobility

(Requirements)
Proposed Solution: Remote Message Sending Protocol

- RemoteMessageSendingProtocol (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
- More efficient than object/thread migration

Since actors do not have shared memory, actor migration is simpler and more efficient than object/thread migration.
Before migration of actor \( m \) from Theater 1 to Theater 2, its references to actors \( a \) and \( c \) are remote, while its reference to actor \( b \) is local.

Actor Migration
After migration of actor \( m \), its reference to actor \( a \) becomes remote and its reference to actor \( c \) remains unchanged.

Actor Migration (continued)
<table>
<thead>
<tr>
<th>Processor</th>
<th>OS-JVM</th>
<th>Location</th>
<th>Machine Name</th>
<th>Processor</th>
<th>OS-JVM</th>
<th>Location</th>
<th>Machine Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparc 20</td>
<td>Solaris 2.6-JDK 1.1.6</td>
<td>Tokyo, Japan</td>
<td>solaris.isr.co.jp</td>
<td>PII, 350MHz</td>
<td>Linux 2.2.5-JDK 1.2-mpress</td>
<td>Paris, France</td>
<td>vulcain.ecoledoc.lip6.fr</td>
</tr>
</tbody>
</table>

### Time Ranges (with SALSA 0.3.2)

<table>
<thead>
<tr>
<th>Event</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local actor creation time</td>
<td>386 ms</td>
</tr>
<tr>
<td>Local messages sending time</td>
<td>148 ms</td>
</tr>
<tr>
<td>LAN messages sending time</td>
<td>30-60 ms</td>
</tr>
<tr>
<td>WAN messages sending time</td>
<td>2-3 seconds</td>
</tr>
<tr>
<td>LAN actor migration time</td>
<td>150-160 ms (minimal actor)</td>
</tr>
<tr>
<td>LAN actor migration time</td>
<td>240-250 ms (actor with 100Kb of data)</td>
</tr>
<tr>
<td>WAN actor migration time</td>
<td>3-7 seconds (minimal actor)</td>
</tr>
<tr>
<td>WAN actor migration time</td>
<td>240-250 ms (actor with 100Kb of data)</td>
</tr>
<tr>
<td>WAN actor migration time</td>
<td>1-50-160ms (minimal actor)</td>
</tr>
</tbody>
</table>
Simple Actor Language, System and Architecture

SALSA is a dialect of Java, with support for:

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

Language Implementation

- SALSA Actor Library
- Join Continuation Code Generation
- RMSP and Migration

SALSA Architecture

Simple Actor Language, System and Architecture
**SALSA Architecture**

1. **Java Virtual Machine**
   - `javac Program.java` -> `Program.class`
   - `java Program`

2. **Actor Library**
   - `javac Program.salsa`
   - `Program.salsa`

3. **SALSA Source Code**
   - `Program.java`
   - `Program.salsa`

---

**A Language and Architecture for Distributed Computing over the Internet**

July 2002
module helloworld;

behavior HelloWorld {

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

}([])[]

module HelloWorld;

SALSA HelloWorld Example
memory or static variables are allowed in SALSA.

Only an actor itself can change its internal state by assigning to its instance variables. No shared assignments to static variables are allowed in SALSA. 

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

To create a new actor:

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

To send a message to an acquaintance actor:

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

For example:

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

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

To send messages to acquaintance actors:

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

Instance:

```java
UniverseActor new HelloWorld = new HelloWorld();
```

All SALSA behaviors inherit from the

Actor Model Support
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):

\[
\text{a1} \rightarrow \text{m1}() \& \text{a2} \rightarrow \text{m2}() \& \text{a3} \rightarrow \text{m3}(\text{token}, 10)
\]

It is possible to "chain" continuations. For example:

\[
\text{fractal} \rightarrow \text{computePixel}() \& \text{screen} \rightarrow \text{draw}(\text{token})
\]

The return type of \text{computePixel()} needs to match the formal argument type of \text{draw}(). SALSA allows method overloading and will choose the most specific method according to the token run-time type.

\[
a \rightarrow \text{m} \text{ with no arguments, is syntactic sugar for } a \rightarrow \text{m}(\text{token})
\]

For example:

\[
\text{fractal} \rightarrow \text{computePixel}() \& \text{screen} \rightarrow \text{draw}(\text{token})
\]

It is possible to "chain" continuations. For example:

\[
\text{customer} \rightarrow \text{m2}(\text{arg0}, \ldots, \text{token}, \ldots, \text{argn}) \& \text{acquaintance} \rightarrow \text{m1}(\text{args})
\]

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

**Token-Passing Continuations**
A join statement can be set up 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;
```

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

A join statement allows to provide a synchronization barrier for multiple actor computations into a single continuation.
Universal Actor Naming in SALSA

```java
behaviorAgent {
    void printItinerary() { ... }
    void act(String[] args) {
        Agent a = new Agent();
        try {
            a.bind("rmsp://yangtze.cs.uiuc.edu:4040/agent",
                   "uan://yangtze.cs.uiuc.edu:3030/agent")
        } catch (Exception e) {
            standardoutput.println(e);
        }
    }
}
```
Getting a remote actor reference by name and sending a message:

\[ \text{RMSP and Actor Migration in SALSA} \]
Example 1: Multicast Protocols

Acknowledged multicast

Multicast

A Language and Architecture for Distributed Computing over the Internet July 2002
Group knowledge multicast (Fagin, Halpern, Moses and Vardi, 1995)

Beginning with a multicast service, one can implement the join protocol:

\[
\text{join (a1<-m1(), a2<-m2(), a3<-m3()) @ a3<-ok() @ done();}
\]

which involves:

- Acknowledging the reception of the messages
- Sending an Acknowledgment
- Notifying the other members

The following diagram illustrates the sequence of messages and acknowledgments:

![Diagram showing the sequence of messages and acknowledgments in the join protocol.]
<table>
<thead>
<tr>
<th>Language/Architecture</th>
<th>Basic Multic和平</th>
<th>Acknowledged Multicast</th>
<th>Group-Knowledge Multicast</th>
<th>Shored Code</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>183 24</td>
<td>73 60</td>
<td>73 60</td>
<td>100 40</td>
<td>319</td>
</tr>
<tr>
<td>SALSA</td>
<td>134 21</td>
<td>60 24</td>
<td>60 24</td>
<td>84 44</td>
<td>134</td>
</tr>
<tr>
<td>Foundry</td>
<td>115 27</td>
<td>73 60</td>
<td>73 60</td>
<td>115 40</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>34 10</td>
<td>40</td>
<td>40</td>
<td>34 10</td>
<td>34</td>
</tr>
</tbody>
</table>

Lines of Code Comparison for Multicast Protocols
SALSA Actor Library

Source

Target

Method

Arguments

Continuation

Token Position

Send (Message)

Process (Message)

Run()

Actor

Messages

Put (Message)

Get()

IsEmpty()

Bind (UAN, UAL)

GetReferenceByLocation (UAL)

GetReferenceByLocation (UAN)

JoinDirector

Mailbox

Thread

Object

Thread

JoinDirector

Actor

Mailbox

A Language and Architecture for Distributed Computing over the Internet July 2002

35
Join Continuation Code Generation