Concurrent Distributed Mobile (Systems) Programming

Universal Actors, SALSA, Coordination Abstractions

Carlos Varela

RPI

November 9, 2010

Programming distributed systems

- It is harder than concurrent programming!
- Yet unavoidable in today’s information-oriented society, e.g.:
  - Internet
  - Web services
  - Grid/cloud computing
- Communicating processes with independent address spaces
- Limited network performance
  - Orders of magnitude difference between WAN, LAN, and single machine communication.
- Localized heterogeneous resources, e.g., I/O, specialized devices.
- Partial failures, e.g. hardware failures, network disconnection
- Openness: creates security, naming, composability issues.
Actors/SALSA

- **Actor Model**
  - A reasoning framework to model concurrent computations
  - Programming abstractions for distributed open systems


- **SALSA**
  - Simple Actor Language System and Architecture
  - An actor-oriented language for mobile and internet computing
  - Programming abstractions for internet-based concurrency, distribution, mobility, and coordination


SALSA and Java

- SALSA source files are compiled into Java source files before being compiled into Java byte code.
- SALSA programs may take full advantage of the Java API.
Hello World Example

module examples.helloworld;

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

Hello World Example

• The act( String[] args ) message handler is similar to the main(...) method in Java and is used to bootstrap SALSA programs.

• When a SALSA program is executed, an actor of the given behavior is created and an act(args) message is sent to this actor with any given command-line arguments.

• References to standardOutput, standardInput and standardError actors are available to all SALSA actors.
SALSA Support for Actors

- Programmers define behaviors for actors.
- Messages are sent asynchronously.
- State is modeled as encapsulated objects/primitive types.
- Messages are modeled as potential method invocations.
- Continuation primitives are used for coordination.

Reference Cell Example

```
module examples.cell;

behavior Cell {
    Object content;

    Cell(Object initialContent) {
        content = initialContent;
    }

    Object get() { return content; }

    void set(Object newContent) {
        content = newContent;
    }
}
```
Actor Creation

• To create an actor:

\[
\text{TravelAgent } a = \text{new TravelAgent();}
\]

Message Sending

• To create an actor:

\[
\text{TravelAgent } a = \text{new TravelAgent();}
\]

• To send a message:

\[
a <- \text{book(flight);}\]
### Causal order

- In a sequential program all execution states are totally ordered.
- In a concurrent program all execution states of a given actor are totally ordered.
- The execution state of the concurrent program as a whole is partially ordered.

### Total order

- In a sequential program all execution states are totally ordered.

```
sequential execution

| computation step |
```
Causal order in the actor model

• In a concurrent program all execution states of a given actor are totally ordered
• The execution state of the concurrent program is partially ordered

Nondeterminism

• An execution is nondeterministic if there is a computation step in which there is a choice what to do next
• Nondeterminism appears naturally when there is asynchronous message passing
  – Messages can arrive or be processed in an order different from the sending order.
Example of nondeterminism

Actor 1

\[ a \leftarrow m_1(); \]

time

Actor 2

\[ a \leftarrow m_2(); \]

time

Actor a

time

Actor \( a \) can receive messages \( m_1() \) and \( m_2() \) in any order.

Coordination Primitives

- SALSA provides three main coordination constructs:
  - Token-passing continuations
    - To synchronize concurrent activities
    - To notify completion of message processing
    - Named tokens enable arbitrary synchronization (data-flow)
  - Join blocks
    - Used for barrier synchronization for multiple concurrent activities
    - To obtain results from otherwise independent concurrent processes
  - First-class continuations
    - To delegate producing a result to a third-party actor
Token Passing Continuations

- Ensures that each message in the continuation expression is sent after the previous message has been processed. It also enables the use of a message handler return value as an argument for a later message (through the token keyword).

  - Example:

    ```
    a1 <- m1() @
    a2 <- m2(token);
    ```

    Send m1 to a1 asking a1 to forward the result of processing m1 to a2 (as the argument of message m2).

Named Tokens

- Tokens can be named to enable more loosely-coupled synchronization

  - Example:

    ```
    token t1 = a1 <- m1();
    token t2 = a2 <- m2();
    token t3 = a3 <- m3(t1);
    token t4 = a4 <- m4(t2);
    a <- m(t1,t2,t3,t4);
    ```

    Sending m(...) to a will be delayed until messages m1()..m4() have been processed. m1() can proceed concurrently with m2().
Cell Tester Example

```java
module examples.cell;

behavior CellTester {
    void act( String[] args ) {
        Cell c = new Cell("Hello");
        standardOutput <- print( "Initial Value:" ) @ c <- get() @
        standardOutput <- println( token ) @
        c <- set("World") @
        standardOutput <- print( "New Value:" ) @
        c <- get() @
        standardOutput <- println( token );
    }
}
```
Join Blocks

- Provide a mechanism for synchronizing the processing of a set of messages.
- Set of results is sent along as a \textit{token} containing an array of results.
  - Example:

\[
\text{Actor[]}\text{actors} = \{\text{searcher0}, \text{searcher1}, \text{searcher2}, \text{searcher3}\};
\]

\[
\text{join}\lbrace
\begin{align*}
\text{for} \ (\text{int} \ i=0; \ i < \text{actors.length}; \ i++) \\
\quad \text{actors}[i] & \leftarrow \text{find( phrase );}
\end{align*}
\rbrace
\]

\[
\text{resultActor} \leftarrow \text{output( token });
\]

Send the \texttt{find( phrase )} message to each actor in \texttt{actors[]} then after all have completed send the result to \texttt{resultActor} as the argument of an \texttt{output( … )} message.

Example: Acknowledged Multicast

\[
\text{join}\lbrace \text{al} \leftarrow \text{m1()}; \text{a2} \leftarrow \text{m2();} \ldots \text{an} \leftarrow \text{mn()}; \rbrace \quad \text{@}
\]

\[
\text{cust} \leftarrow \text{n(token)};
\]

C. Varela

C. Varela
### Lines of Code Comparison

<table>
<thead>
<tr>
<th>Acknowledged Multicast</th>
<th>Java</th>
<th>Foundry</th>
<th>SALSA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>168</td>
<td>100</td>
<td>31</td>
</tr>
</tbody>
</table>

### First Class Continuations

- Enable actors to delegate computation to a third party independently of the processing context.

- For example:

```java
int m(...) {
  b <- n(...) @ currentContinuation;
}
```

*Ask (delegate) actor b to respond to this message m on behalf of current actor (see 12) by processing its own message n.*
Delegate Example

```java
module examples.fibonacci;

behavior Calculator {
    int fib(int n) {
        Fibonacci f = new Fibonacci(n);
        f <- compute() @ currentContinuation;
    }
    int add(int n1, int n2) { return n1+n2; }

    void act(String args[]) {
        fib(15) @ standardOutput <- println(token);
        fib(5) @ add(token,3) @ standardOutput <- println(token);
    }
}
```

C. Varela

Fibonacci Example

```java
module examples.fibonacci;

behavior Fibonacci {
    int n;
    Fibonacci(int n) { this.n = n; }
    int add(int x, int y) { return x + y; }
    int compute() {
        if (n == 0) return 0;
        else if (n <= 2) return 1;
        else {
            Fibonacci fib1 = new Fibonacci(n-1);
            Fibonacci fib2 = new Fibonacci(n-2);
            token x = fib1.compute();
            token y = fib2.compute();
            add(x,y) @ currentContinuation;
        }
    }

    void act(String args[]) {
        n = Integer.parseInt(args[0]);
        compute() @ standardOutput<-println(token);
    }
}
```

C. Varela
Fibonacci Example 2

module examples.fibonacci2;

behavior Fibonacci {

    int add(int x, int y) { return x + y; } 

    int compute(int n) {
        if (n == 0) return 0;
        else if (n <= 2) return 1;
        else {
            Fibonacci fib = new Fibonacci();
            token x = fib <- compute(n-1);
            compute(n-2) @ add(x, token) @ currentContinuation;
        }
    }

    void act(String args[]) {
        int n = Integer.parseInt(args[0]);
        compute(n) @ standardOutput <- println(token);
    }
}

Execution of salsa Fibonacci 6

Create new actor

Synchronize on result

Non-blocked actor
Worldwide Computing

- Distributed computing over the Internet.
- Access to *large number* of processors offsets slow communication and reliability issues.
- Seeks to create a platform for many applications.

World-Wide Computer (WWC)

- Worldwide computing platform.
- Provides a run-time system for universal actors.
- Includes naming service implementations.
- Remote message sending protocol.
- Support for universal actor migration.
Abstractions for Worldwide Computing

- *Universal Actors*, a new abstraction provided to guarantee unique actor names across the Internet.

- *Theaters*, extended Java virtual machines to provide execution environment and network services to universal actors:
  - Access to local resources.
  - Remote message sending.
  - Migration.

- *Naming service*, to register and locate universal actors, transparently updated upon universal actor creation, migration, recollection.

Universal Naming

- Consists of *human readable* names.
- Provides location transparency to actors.
- Name to location mappings efficiently updated as actors migrate.
Universal Actor Naming

• UAN servers provide mapping between static names and dynamic locations.
  – Example:

  uan://wwc.cs.rpi.edu/cvarela/calendar

  ![Diagram of Universal Actor Naming]

Universal Actors

• Universal Actors extend the actor model by associating a universal name and a location with the actor.

• Universal actors may migrate between theaters and the name service keeps track of their current location.
Universal Actor Implementation

WWC Theaters
WWC Theaters

- Theaters provide an execution environment for actors.
- Provide a layer beneath actors for message passing and migration.
- Example locator:
  
  \[ \text{rmosp://wwc.cs.rpi.edu/calendarInstance10} \]

  \underline{Theater address and port.} \underline{Actor location.}

Environment Actors

- Theaters provide access to environment actors.
- Environment actors perform actions specific to the theater and are not mobile.
- Include standard input, output and error stream actors.
Remote Message Sending Protocol

- Messages between remote actors are sent using the Remote Message Sending Protocol (RMSP).
- RMSP is implemented using Java object serialization.
- RMSP protocol is used for both message sending and actor migration.
- When an actor migrates, its locator (UAL) changes but its name (UAN) does not.

Universal Actor Naming Protocol
Universal Actor Naming Protocol

- UANP includes messages for:
  - Binding actors to UAN, UAL pairs
  - Finding the locator of a universal actor given its UAN
  - Updating the locator of a universal actor as it migrates
  - Removing a universal actor entry from the naming service

- SALSA programmers need not use UANP directly in programs. UANP messages are transparently sent by WWC run-time system.

UANP Implementations

- Default naming service implementation stores UAN to UAL mapping in name servers as defined in UANs.
  - Name server failures may induce universal actor unreachability.

- Distributed (Chord-based) implementation uses consistent hashing and a ring of connected servers for fault-tolerance. For more information, see:


SALSA Language Support for Worldwide Computing

• SALSA provides linguistic abstractions for:
  - Universal naming (UAN & UAL).
  - Remote actor creation.
  - Message sending.
  - Migration.
  - Coordination.

• SALSA-compiled code closely tied to WWC run-time platform.

Universal Actor Creation

• To create an actor locally

\[
\text{TravelAgent } a = \text{new } \text{TravelAgent}();
\]

• To create an actor with a specified UAN and UAL:

\[
\text{TravelAgent } a = \text{new } \text{TravelAgent()} \text{ at } (\text{uan}, \text{ual});
\]

• At current location with a UAN:

\[
\text{TravelAgent } a = \text{new } \text{TravelAgent()} \text{ at } (\text{uan});
\]
Message Sending

TravelAgent a = new TravelAgent();

    a <- book(flight);

Remote Message Sending

• Obtain a remote actor reference by name.

TravelAgent a = (TravelAgent)
    TravelAgent.getReferenceByName("uan://myhost/ta");

    a <- printItinerary();
Reference Cell Service Example

```java
module examples.cell;

behavior Cell implements ActorService{
    Object content;

    Cell( Object initialContent) {
        content = initialContent;
    }

    Object get() {
        standardOutput <- println("Returning:"+content);
        return content;
    }

    void set( Object newContent) {
        standardOutput <- println("Setting:"+newContent);
        content = newContent;
    }
}
```

Reference Cell Client Example

```java
module examples.cell;

behavior GetCellValue {
    void act( String[] args) {
        if (args.length != 1){
            standardOutput <- println("Usage:
            salsa examples.cell.GetCellValue <CellUAN>");
            return;
        }

        Cell c = (Cell)
        Cell.getReferenceByName(new UAN(args[0]));

        standardOutput <- print("Cell Value") @
        c <- get() @
        standardOutput <- println(token);
    }
}
```
Migration

- Obtaining a remote actor reference and migrating the actor.

TravelAgent a = (TravelAgent)
    TravelAgent.getReferenceByName
    ("uan://myhost/ta");
a <- migrate("rmsp://yourhost/travel") @
a <- printItinerary();

Moving Cell Tester Example

module examples.cell;

behavior MovingCellTester {
    void act( String[] args ) {
        if (args.length != 3)
            standardOutput <- println("Usage:
                salsa examples.cell.MovingCellTester <UAN> <UAL1> <UAL2>");
            return;
    }

    Cell c = new Cell("Hello") at (new UAN(args[0]), new UAL(args[1]));
    standardOutput <- print("Initial Value:"); c <- get(); standardOutput <- println( token );
    c <- set("World"); standardOutput <- print("New Value:"); c <- get(); standardOutput <- println( token );
    c <- migrate(args[2]);
    standardOutput <- print("New Value at New Location:"); c <- get(); standardOutput <- println( token );
    }
}
Agent Migration Example

behavior Migrate {
    void print() {
        standardOutput<-println( "Migrate actor is here."");
    }
    void act( String[] args ) {
        if (args.length != 3) {
            standardOutput<-println("Usage: salsa migration.Migrate <UAN> <srcUAL>
<destUAL>");
            return;
        }
        UAN uan = new UAN(args[0]);
        UAL ual = new UAL(args[1]);
        Migrate migrateActor = new Migrate() at (uan, ual);
        migrateActor<-print() @
migrateActor<-migrate{ args[2] } @
migrateActor<-print();
    }
}

Migration Example

- The program must be given valid universal actor name and locators.
  - Appropriate name services and theaters must be running.

- After remotely creating the actor. It sends the print message to itself before migrating to the second theater and sending the message again.
Compilation and Execution

```
$ java salsac.SalsaCompiler Migrate.salsa
SALSA Compiler Version 1.0: Reading from file Migrate.salsa . . .
SALSA Compiler Version 1.0: SALSA program parsed successfully.
SALSA Compiler Version 1.0: SALSA program compiled successfully.
$ javac Migrate.java
$ java Migrate
$ Usage: java Migrate <uan> <ual> <ual>
```

- Compile Migrate.salsa file into Migrate.java.
- Compile Migrate.java file into Migrate.class.
- Execute Name Server
- Execute Theater 1 and Theater 2 Environments
- Execute Migrate in any computer

Migration Example

The actor will print "Migrate actor is here." at theater 1 then at theater 2.
World Migrating Agent Example

<table>
<thead>
<tr>
<th>Host</th>
<th>Location</th>
<th>OS/JVM</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>yangtze.cs.uic.edu</td>
<td>Urbana IL, USA</td>
<td>Solaris 2.5.1/SDK 1.1.6</td>
<td>Ultra 2</td>
</tr>
<tr>
<td>vulcan.ecoledoc.lip6.fr</td>
<td>Paris, France</td>
<td>Linux 2.2.5/SDK 1.2pre2</td>
<td>Pentium II 350MHz</td>
</tr>
<tr>
<td>solar.isr.co.jp</td>
<td>Tokyo, Japan</td>
<td>Solaris 2.6/SDK 1.1.6</td>
<td>Sparc 20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Processing Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local actor creation</td>
<td>386ms</td>
</tr>
<tr>
<td>Local message sending</td>
<td>148ms</td>
</tr>
<tr>
<td>LAN message sending</td>
<td>30-60</td>
</tr>
<tr>
<td>WAN message sending</td>
<td>2-3s</td>
</tr>
<tr>
<td>LAN minimal actor migration</td>
<td>100-300ms</td>
</tr>
<tr>
<td>WAN minimal actor migration</td>
<td>240-250ms</td>
</tr>
<tr>
<td>WAN 100Kb actor migration</td>
<td>5-7s</td>
</tr>
<tr>
<td>WAN 100Kb actor migration</td>
<td>25-30s</td>
</tr>
</tbody>
</table>

Address Book Service

```java
module examples.addressbook;

behavior AddressBook implements ActorService {
    Hashtable name2email;
    AddressBook() {
        name2email = new HashTable();
    }
    String getName(String email) { ... }
    String getEmail(String name) { ... }
    boolean addUser(String name, String email) { ... }

    void act(String[] args) {
        if (args.length != 0) {
            standardOutput.println("Usage: salsa -Duan=<uan> -Dual=<ual>
examples.addressbook.AddressBook");
        }
    }
}
```

C. Varela 56
module examples.addressbook;

behavior AddUser {
    void act(String[] args) {
        if (args.length != 3) {
            standardOutput<println("Usage: salsa examples.addressbook.AddUser <BookUAN> <Name> <Email>");
            return;
        }
        AddressBook book = (AddressBook)
            AddressBook.getReferenceByName(new UAN(args[0]));
        book<addUser(args(1), args(2));
    }
}

C. Varela

module examples.addressbook;

behavior GetEmail {
    void act(String[] args) {
        if (args.length != 2) {
            standardOutput<println("Usage: salsa examples.addressbook.GetEmail <BookUAN> <Name>");
            return;
        }
        getEmail(args(0), args(1));
    }

    void getEmail(String uan, String name) {
        AddressBook book = (AddressBook)
            AddressBook.getReferenceByName(uan);
        standardOutput<print(name + "’s email: ") @
            book<getEmail(name) @
            standardOutput<println(token);
    }
}

C. Varela
module examples.addressbook;

behavior MigrateBook {
    void act(String[] args) {
        if (args.length != 2) {
            standardOutput.println("Usage: salsa examples.addressbook.Migrate <BookUAN> <NewUAN>");
            return;
        }
        AddressBook book = (AddressBook)
            AddressBook.getReferenceByName(new UAN(args[0]));
        book.migrate(args[1]);
    }
}

Exercises

1. How would you implement the join continuation linguistic abstraction in terms of message passing?

2. Download and execute the CellTester.salsa example.

3. Write a solution to the Flavius Josephus problem in SALSA. A description of the problem is at Van Roy and Haridi’s book, Section 7.8.3 (page 558).
4. How would you implement the join continuation linguistic abstraction considering different potential distributions of its participating actors?

5. Download and execute the *Agent.salsa* example.

6. Modify the lock example in the SALSA distribution to include a wait/notify protocol, as opposed to “busy-waiting” (or rather “busy-asking”).

7. Van Roy and Haridi’s Book Exercise 11.11.3 (pg 746). Implement the example using SALSA/WWC.