Distributed systems abstractions
(PDCS 9, CPE 6*)

Carlos Varela
Rensselaer Polytechnic Institute

October 18, 2019

* Concurrent Programming in Erlang, by J. Armstrong, R. Virding, C. Wikström, M. Williams

C. Varela
Overview of programming distributed systems

• It is harder than concurrent programming!
• Yet unavoidable in today’s information-oriented society, e.g.:
  – Internet, mobile devices
  – Web services
  – Cloud computing
• Communicating processes with independent address spaces
• Limited network performance
  – Orders of magnitude difference between WAN, LAN, and intra-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.
SALSA Revisited

- **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


- **Advantages for distributed computing**
  - Actors encapsulate state and concurrency:
    - Actors can run in different concurrency machines.
    - Actors can change location dynamically.
  - Communication is asynchronous:
    - Fits real world distributed systems.
  - Actors can fail independently.
World-Wide Computer (WWC)

- Distributed 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, garbage collection.
Universal Actor Names (UAN)

- Consists of *human readable* names.
- Provides location transparency to actors.
- Name to locator mapping updated as actors migrate.
- UAN servers provide mapping between names and locators.
  - Example Universal Actor Name:

```
uan://wwc.cs.rpi.edu:3030/cvarela/calendar
```

- Name server address and (optional) port.
- Unique relative actor name.
WWC Theaters

Listener

Hashtable

RMSP Server

<table>
<thead>
<tr>
<th>relative UAL</th>
<th>SALSA Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Universal Actor Run-Time System

World Wide Computing Theater

Universal Actors

Environment Actors

System Resources

C. Varela
Universal Actor Locators (UAL)

- Theaters provide an execution environment for universal actors.
- Provide a layer beneath actors for message passing and migration.
- When an actor migrates, its UAN remains the same, while its UAL changes to refer to the new theater.
- Example Universal Actor Locator:
  
  \[\text{rmsp://wwc.cs.rpi.edu:4040}\]
  
  Theater’s IP address and (optional) port.

C. Varela
SALSA Language Support for Worldwide Computing

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

- SALSA-compiled code closely tied to WWC run-time platform.
Universal Actor Creation

• To create an actor locally

   TravelAgent a = new TravelAgent();

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

   TravelAgent a = new TravelAgent() at (uan, ual);

• To create an actor with a specified UAN at current location:

   TravelAgent a = new TravelAgent() at (uan);
Message Sending

TravelAgent a = new TravelAgent();

a <- book( flight );

Message sending syntax is the same (<-), independently of actor’s location.
Remote Message Sending

- Obtain a remote actor reference by name.

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

a <- printItinerary();
```
module dcell;

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;
    }
}

implments ActorService signals that actors with this behavior are not to be garbage collected.
module dcell;

behavior CellTester {

    void act( String[] args ) {

        if (args.length != 2){
            standardError <- println("Usage: salsa dcell.CellTester <UAN> <UAL>") ;
            return;
        }

        Cell c = new Cell(0) at (new UAN(args[0]), new UAL(args[1]));

        standardOutput <- print("Initial Value:" ) @
                       c <- get() @ standardOutput <- println( token );

    }

}
module dcell;

behavior GetCellValue {

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

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

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

}
module addressbook;
import java.util.*

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> addressbook.AddressBook");
        }
    }
}
module addressbook;

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

Address Book Get Email Example

```java
module addressbook;

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

    void getEmail(String uan, String name){
        try{
            AddressBook book = (AddressBook) AddressBook.getReferenceByName(new UAN(uan));
            standardOutput <- println(name + "'s email: ") @
            book <- getEmail(name) @
            standardOutput <- println(token);
        } catch (MalformedUANException e){
            standardError<-println(e);
        }
    }
}
```
Erlang Language Support for Distributed Computing

- Erlang provides linguistic abstractions for:
  - Registered processes (actors).
  - Remote process (actor) creation.
  - Remote message sending.
  - Process (actor) groups.
  - Error detection.

- Erlang-compiled code closely tied to Erlang node run-time platform.
Erlang Nodes

• To return our own node name:

```
node()
```

• To return a list of other known node names:

```
nodes()
```

• To monitor a node:

```
monitor_node(Node, Flag)
```

If `flag` is true, monitoring starts. If false, monitoring stops. When a monitored node fails, `{nodedown, Node}` is sent to monitoring process.
Actor Creation

- To create an actor locally
  
  \[
  \text{Agent} = \text{spawn}(\text{travel, agent, []});
  \]

- To create an actor in a specified remote node:
  
  \[
  \text{Agent} = \text{spawn}(\text{host, travel, agent, []});
  \]

- travel is the module name,
- agent is the function name,
- Agent is the actor name.
- host is the node name.
Actor Registration

• To register an actor:

\[
\text{register}(ta, \text{Agent})
\]

• To return the actor identified with a registered name:

\[
\text{whereis}(ta)
\]

• To remove the association between an atom and an actor:

\[
\text{unregister}(ta)
\]

ta is the registered name (an atom), Agent is the actor name (PID).
Agent = `spawn(travel, agent, [])`,
register(ta, Agent)

Agent ! {book, Flight}
ta ! {book, Flight}

Message sending syntax is the same (!) with actor name (Agent) or registered name (ta).
Remote Message Sending

• To send a message to a remote registered actor:

\{ta, host\} ! \{book, Flight\}
Reference Cell Service Example

-module(dcell).
-export([[cell/1,start/1]]).

cell(Content) ->
    receive
        {set, NewContent} -> cell(NewContent);
        {get, Customer}   -> Customer ! Content,
                             cell(Content)
    end.

start(Content) ->
    register(dcell, spawn(dcell, cell, [Content]))
-module(dcellTester).
-export([main/0]).

main() -> dcell:start(0),
          dcell!{get, self()},
          receive
              Value ->
                  io:format("Initial Value:~w~n", [Value])
          end.
Reference Cell Client Example

-module(dcellClient).
-export([getCellValue/1]).

getCellValue(Node) ->
    {dcell, Node}!{get, self()},
    receive
    Value ->
        io:format("Initial Value:~w~n", [Value])
    end.
Address Book Service

-module(addressbook).
-export([start/0,addressbook/1]).

start() ->
    register(addressbook, spawn(addressbook, addressbook, [[]])).

addressbook(Data) ->
    receive
    {From, {addUser, Name, Email}} ->
        From ! {addressbook, ok},
        addressbook(add(Name, Email, Data));
    {From, {getName, Email}} ->
        From ! {addressbook, getname(Email, Data)},
        addressbook(Data);
    {From, {getEmail, Name}} ->
        From ! {addressbook, getemail(Name, Data)},
        addressbook(Data)
    end.

add(Name, Email, Data) -> ...
getName(Email, Data) -> ...
getemail(Name, Data) -> ...
Address Book Client Example

-module(addressbook_client).
-export([getEmail/1,getName/1,addUser/2]).

addressbook_server() -> 'addressbook@127.0.0.1'.

getEmail(Name) -> call_addressbook({getEmail, Name}).
getName(Email) -> call_addressbook({getName, Email}).
addUser(Name, Email) -> call_addressbook({addUser, Name, Email}).

call_addressbook(Msg) ->
    AddressBookServer = addressbook_server(),
    monitor_node(AddressBookServer, true),
    {addressbook, AddressBookServer} ! {self(), Msg},
    receive
        {addressbook, Reply} ->
            monitor_node(AddressBookServer, false),
            Reply;
        {nodedown, AddressBookServer} ->
            no
    end.
51. How would you implement the join block linguistic abstraction considering different potential distributions of its participating actors?
52. CTM Exercise 11.11.3 (page 746). Implement the example using SALSA/WWC and Erlang.
53. PDCS Exercise 9.6.3 (page 203).
54. PDCS Exercise 9.6.9 (page 204).
55. PDCS Exercise 9.6.12 (page 204).
56. Write the same distributed programs in Erlang.