Distributed systems abstractions
(PDCS 9, CPE 6*)

Carlos Varela
Rensselaer Polytechnic Institute

October 18, 2016

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

- Actors are concurrent entities that react to messages.
  - State is completely encapsulated. There is no shared memory!
  - Message passing is asynchronous.
  - Actors can create new actors. Run-time has to ensure fairness.
- AMST extends the call by value lambda calculus with actor primitives. State is modeled as function arguments. Actors use ready to receive new messages.
- SALSA extends an object-oriented programming language (Java) with universal actors. State is explicit, encapsulated in instance variables. Control loop is implicit: ending a message handler, signals readiness to receive a new message. Actors are garbage-collected.
- Erlang extends a functional programming language core with processes that run arbitrary functions. State is implicit in the function’s arguments. Control loop is explicit: actors use receive to get a message, and tail-form recursive call to continue. Ending a function denotes process (actor) termination.
Tree Product Behavior in AMST

\[ B_{\text{treeprod}} = \]
\[ \text{rec}(\lambda b. \lambda m. \]
\[ \quad \text{seq}(\text{if(isnat(tree(m))}, \]
\[ \quad \quad \text{send(cust(m),tree(m))}, \]
\[ \quad \quad \text{let newcust=new}(B_{\text{joincont}}(\text{cust}(m))), \]
\[ \quad \quad \quad \text{l}p = \text{new}(B_{\text{treeprod}}), \]
\[ \quad \quad \quad \text{r}p = \text{new}(B_{\text{treeprod}}) \text{ in} \]
\[ \quad \text{seq}(\text{send}(l)p, \]
\[ \quad \quad \text{pr(left(tree(m)),newcust))}, \]
\[ \quad \quad \text{send}(r)p, \]
\[ \quad \quad \quad \text{pr(right(tree(m)),newcust))}), \]
\[ \quad \text{ready}(b)) \]
Join Continuation in AMST

\[ B_{\text{joincont}} = \lambda \text{cust} . \lambda \text{firstnum}. \text{ready}(\lambda \text{num}. \]
\[ \quad \text{seq}(\text{send}(\text{cust}, \text{firstnum} \times \text{num}), \]
\[ \quad \quad \text{ready}(\text{sink})) \]
module jctreeprod;

import tree.Tree;

behavior TreeProduct {

    void compute(Tree t, UniversalActor c) {
        if (t.isLeaf()) c <- result(t.value());
        else {
            JoinCont newCust = new JoinCont(c);
            TreeProduct lp = new TreeProduct();
            TreeProduct rp = new TreeProduct();
            lp <- compute(t.left(), newCust);
            rp <- compute(t.right(), newCust);
        }
    }

}
module jctreeprod;

behavior JoinCont {

  UniversalActor cust;
  int first;
  boolean receivedFirst;

  JoinCont(UniversalActor cust){
    this.cust = cust;
    this.receivedFirst = false;
  }

  void result(int v) {
    if (!receivedFirst){
      first = v; receivedFirst = true;
    }
    else // receiving second value
      cust <- result(first*v);
  }
}
-module(treeprod).
-export([treeprod/0, join/1]).

treeprod() ->
    receive
        {Left, Right}, Customer} ->
            NewCust = spawn(treeprod, join, [Customer]),
            LP = spawn(treeprod, treeprod, []),
            RP = spawn(treeprod, treeprod, []),
            LP!{Left, NewCust},
            RP!{Right, NewCust};
        {Number, Customer} ->
            Customer ! Number
    end,
    treeprod().

join(Customer) -> receive V1 -> receive V2 -> Customer ! V1*V2 end end.
Concurrency Control in SALSA

- 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

- `@` syntax using `token` as an argument is syntactic sugar.
  - Example 1:
    ```plaintext
    a1 <- m1() @
    a2 <- m2( token );
    ```
    is syntactic sugar for:
    ```plaintext
    token t = a1 <- m1();
    a2 <- m2( t );
    ```
  
  - Example 2:
    ```plaintext
    a1 <- m1() @
    a2 <- m2();
    ```
    is syntactic sugar for:
    ```plaintext
    token t = a1 <- m1();
    a2 <- m2() : waitfor( t );
    ```
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()`.
Join Blocks

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

```java
UniversalActor[] actors = { searcher0, searcher1, searcher2, searcher3 };
join {
    for (int i=0; i < actors.length; i++){
        actors[i] <- find( phrase );
    }
} @ resultActor <- output( token );
```

*Send the find( phrase ) message to each actor in actors[] then after all have completed send the result to resultActor as the argument of an output( ... ) message.*
First Class Continuations

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

- For example:

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

*Ask (delegate) actor b to respond to this message m on behalf of current actor (self) by processing b’s message n.*
module treeprod;
import tree.Tree;

behavior JoinTreeProduct {

    int multiply(Object[] results) {
        return (Integer) results[0] * (Integer) results[1];
    }

    int compute(Tree t) {
        if (t.isLeaf()) return t.value();
        else {
            JoinTreeProduct lp = new JoinTreeProduct();
            JoinTreeProduct rp = new JoinTreeProduct();
            join {
                lp <- compute(t.left());
                rp <- compute(t.right());
            } @ multiply(token) @ currentContinuation;
        }
    }
}
Concurrency control in Erlang

- Erlang uses a *selective receive* mechanism to help coordinate concurrent activities:
  - **Message patterns and guards**
    - To select the next message (from possibly many) to execute.
    - To receive messages from a specific process (actor).
    - To receive messages of a specific kind (pattern).
  - **Timeouts**
    - To enable default activities to fire in the absence of messages (following certain patterns).
    - To create timers.
  - **Zero timeouts** *(after 0)*
    - To implement priority messages, to flush a mailbox.
Selective Receive

receive
  MessagePattern1 [when Guard1] ->
    Actions1 ;
  MessagePattern2 [when Guard2] ->
    Actions2 ;
...
end

receive suspends until a message in the actor’s mailbox matches any of the patterns including optional guards.

- Patterns are tried in order. On a match, the message is removed from the mailbox and the corresponding pattern’s actions are executed.

- When a message does not match any of the patterns, it is left in the mailbox for future receive actions.
Selective Receive Example

Example program and mailbox (head at top):

```
receive
    msg_b  --> ...
end

receive  tries to match `msg_a` and fails. `msg_b` can be matched, so it is processed. Suppose execution continues:

receive
    msg_c  --> ...
    msg_a  --> ...
end
```

The next message to be processed is `msg_a` since it is the next in the mailbox and it matches the 2nd pattern.
Receiving from a specific actor

Actor ! {self(), message}

self() is a Built-In-Function (BIF) that returns the current (executing) process id (actor name). Ids can be part of a message.

receive
    {ActorName, Msg} when ActorName == A1 ->
    ...
end

receive can then select only messages that come from a specific actor, in this example, A1. (Or other actors that know A1’s actor name.)
Receiving a specific kind of message

counter(Val) ->
    receive
        increment -> counter(Val+1);
        {From, value} ->
            From ! {self(), Val},
            counter(Val);
        stop -> true;
        Other -> counter(Val)
    end.

increment is an atom whereas Other is a variable (that matches anything!).

counter is a behavior that can receive increment messages, value request messages, and stop messages. Other message kinds are ignored.
Order of message patterns matters

receive

  {{Left, Right}, Customer} ->
    NewCust = spawn(treeprod, join, [Customer]),
    LP = spawn(treeprod, treeprod, []),
    RP = spawn(treeprod, treeprod, []),
    LP!{Left, NewCust},
    RP!{Right, NewCust};

{{Number, Customer} ->
    Customer ! Number
}

end

In this example, a binary tree is represented as a tuple

  {Left, Right}, or as a Number, e.g.,

  {{{5, 6}, 2}, {3, 4}}

{Left, Right} is a more specific pattern than Number is (which matches anything!). Order of patterns is important.
Selective Receive with Timeout

receive
    MessagePattern1 [when Guard1] ->
        Actions1 ;
    MessagePattern2 [when Guard2] ->
        Actions2 ;
    ...
    after TimeOutExpr ->
        ActionsT
end

TimeOutExpr evaluates to an integer interpreted as milliseconds.

If no message has been selected within this time, the timeout occurs and ActionsT are scheduled for evaluation.

A timeout of infinity means to wait indefinitely.
sleep(Time) ->
    receive
        after Time ->
            true
    end.

sleep(Time) suspends the current actor for Time milliseconds.
Timeout Example

receive
  click ->
  receive
    click ->
      double_click
    after double_click_interval() ->
      single_click
  end
end
...
end

double_click_interval evaluates to the number of milliseconds expected between two consecutive mouse clicks, for the receive to return a double_click. Otherwise, a single_click is returned.
receive
    MessagePattern1 [when Guard1] ->
       Actions1 ;
    MessagePattern2 [when Guard2] ->
       Actions2 ;
...
    after 0 ->
       ActionsT
end

A timeout of 0 means that the timeout will occur immediately, but Erlang tries all messages currently in the mailbox first.
flush_buffer() ->
    receive
        AnyMessage ->
            flush_buffer()
        after 0 ->
            true
    end.

flush_buffer() completely empties the mailbox of the current actor.
priority_receive() will return the first message in the actor’s mailbox, except if there is an interrupt message, in which case, interrupt will be given priority.
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 machines.
    - Actors can change location dynamically.
  - Communication is asynchronous:
    - Fits real world distributed systems.
  - Actors can fail independently.

C. Varela
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:

  \[
  \text{uan://wwc.cs.rpi.edu:3030/cvarela/calendar}
  \]

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

  \texttt{rmsp://wwc.cs.rpi.edu:4040}

  Theater’s IP address and (optional) port.
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);
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;
  }
}
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");
        }
    }
}

C. Varela
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));
    }
}
module addressbook;

behavior GetEmail {
  void act( String[] args ) {
    if (args.length != 2){
      standardOutput <- println("Usage: salsa");
      return;
    }
    getEmail(args(0),args(1));
  }
  void getEmail(String uan, String name){
    try{
      AddressBook book = (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

  \[ \text{Agent} = \text{spawn}(\text{travel}, \text{agent}, []) \];

- To create an actor in a specified remote node:

  \[ \text{Agent} = \text{spawn}(\text{host}, \text{travel}, \text{agent}, []) \];

  \text{travel} is the module name, \text{agent} is the function name, \text{Agent} is the actor name.

  \text{host} is the node name.
Actor Registration

- To register an actor:
  \[
  \text{register}(\text{ta}, \text{Agent})
  \]

- To return the actor identified with a registered name:
  \[
  \text{whereis}(\text{ta})
  \]

- To remove the association between an atom and an actor:
  \[
  \text{unregister}(\text{ta})
  \]

\[\text{ta} \text{ is the registered name (an atom),}\]
\[\text{Agent} \text{ is the actor name (PID).}\]
Message Sending

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]))
Reference Cell Tester

-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) -> ...

C. Varela
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.
Exercises

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.