Distributed and Mobile Systems Programming

Universal Actors, Naming Service, Theaters

Carlos Varela
RPI

March 18, 2019
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.
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

  Name server address and port.  Actor name.
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

Theater

collection of objects

UAN

UAL

Thread

mailbox

C. Varela
WWC Theaters
WWC Theaters

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

  \[\text{Theater address and port.}\]  \[\text{Actor location.}\]
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

    TravelAgent a = new TravelAgent();

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

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

• At current location with a UAN:

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

```java
TravelAgent a = new TravelAgent();

a <- book( flight );
```
Remote Message Sending

- Obtain a remote actor reference by name.

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

a <- printItinerary();
```
References Cell Service Example

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;
    }
}
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);
    }
}

C. Varela
Migration

- Obtaining a remote actor reference and migrating the actor.

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

a <- migrate("rmsp://yourhost/travel") @
a <- printItinerary();
```
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]) @
        c <- set(”New World”) @
        standardOutput <- print(”New Value at New Location:”) @
        c <- get() @ standardOutput <- println( token );
    }
}
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
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.uiuc.edu</td>
<td>Urbana IL, USA</td>
<td>Solaris 2.5.1 JDK 1.1.6</td>
<td>Ultra 2</td>
</tr>
<tr>
<td>vulcain.ecoledoc.lip6.fr</td>
<td>Paris, France</td>
<td>Linux 2.2.5 JDK 1.2pre2</td>
<td>Pentium II 350Mhz</td>
</tr>
<tr>
<td>solar.isr.co.jp</td>
<td>Tokyo, Japan</td>
<td>Solaris 2.6 JDK 1.1.6</td>
<td>Sparc 20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local actor creation</th>
<th>386(\mu)s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local message sending</td>
<td>148 (\mu)s</td>
</tr>
<tr>
<td>LAN message sending</td>
<td>30-60 ms</td>
</tr>
<tr>
<td>WAN message sending</td>
<td>2-3 s</td>
</tr>
<tr>
<td>LAN minimal actor migration</td>
<td>150-160 ms</td>
</tr>
<tr>
<td>LAN 100Kb actor migration</td>
<td>240-250 ms</td>
</tr>
<tr>
<td>WAN minimal actor migration</td>
<td>3-7 s</td>
</tr>
<tr>
<td>WAN 100Kb actor migration</td>
<td>25-30 s</td>
</tr>
</tbody>
</table>
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");
        }
    }
}
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.getReferenceByUAN(new UAN(args[0]));
        book<-addUser(args(1), args(2));
    }
}
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 <- println(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> <NewUAL>");
            return;
        }
        AddressBook book = (AddressBook)
AddressBook.getReferenceByName(new UAN(args[0]));
book<‐migrate(args(1));
    }
}
Exercises

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.