

Distributed (Systems) Programming with SALSA (PDCS 9)

Universal Actors, Name Service, Actor Migration,
World-Wide Computer Theater (Run-time System)

Carlos Varela

Rensselaer Polytechnic Institute

November 14, 2014

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.

Overview of 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.

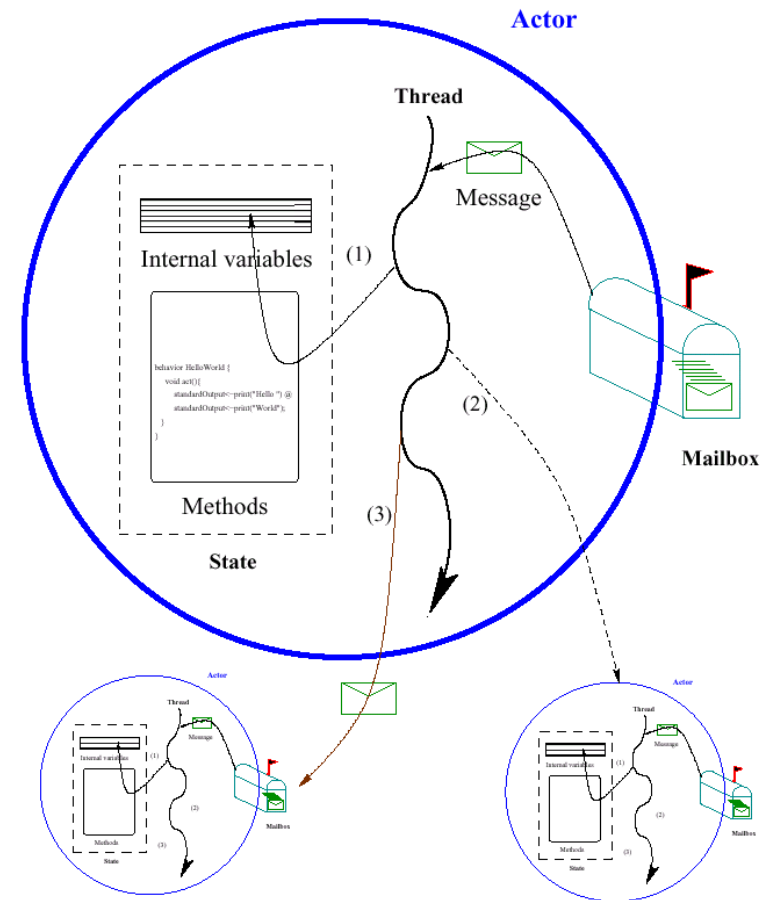
Actors/SALSA Revisited

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

G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

- 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

C. Varela and G. Agha, “Programming dynamically reconfigurable open systems with SALSA”, *ACM SIGPLAN Notices, OOPSLA 2001 Intriguing Technology Track*, 36(12), pp 20-34.



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:3030/cvarela/calendar

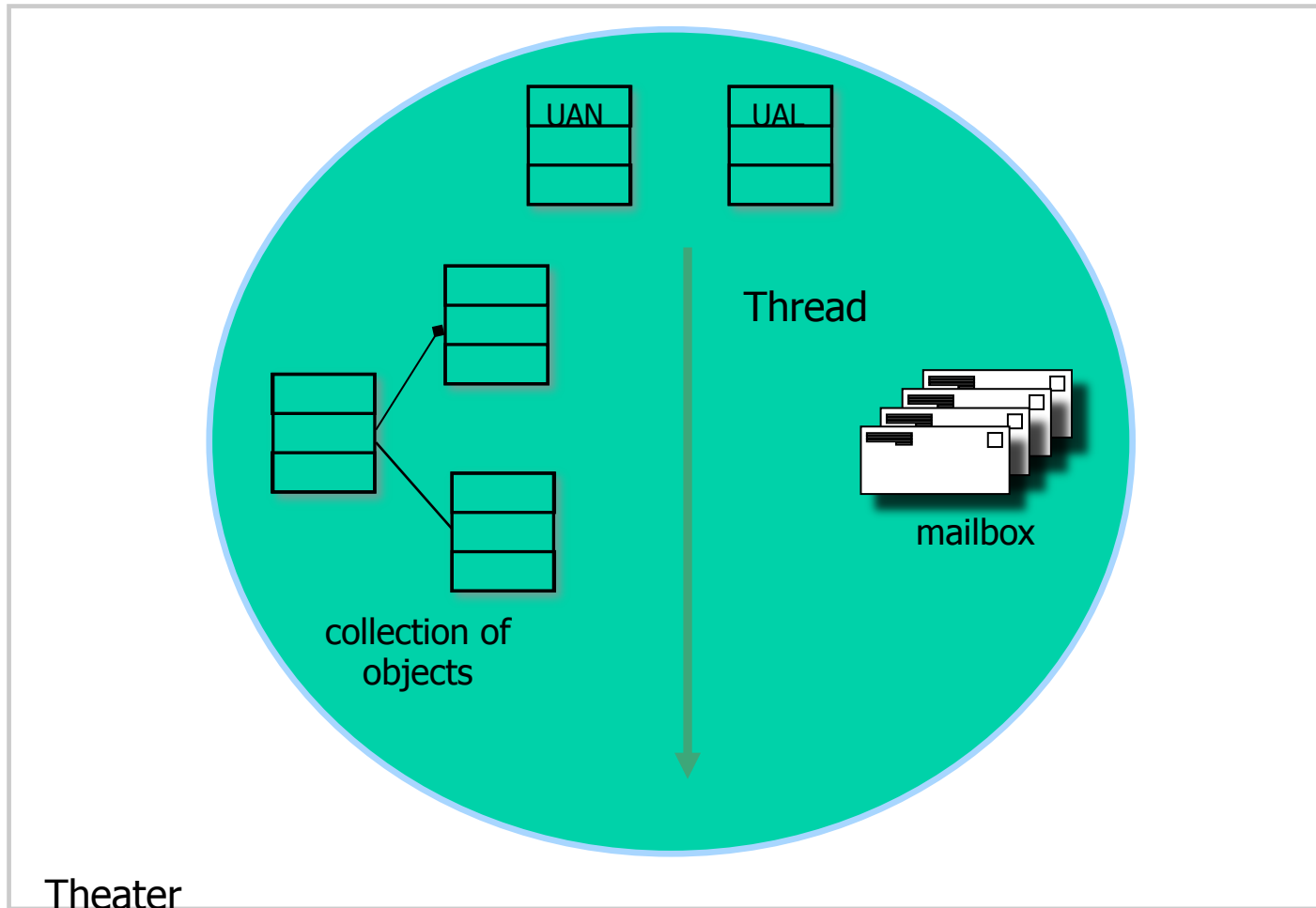
Name server
address and
(optional) port.

Unique
relative
actor name.

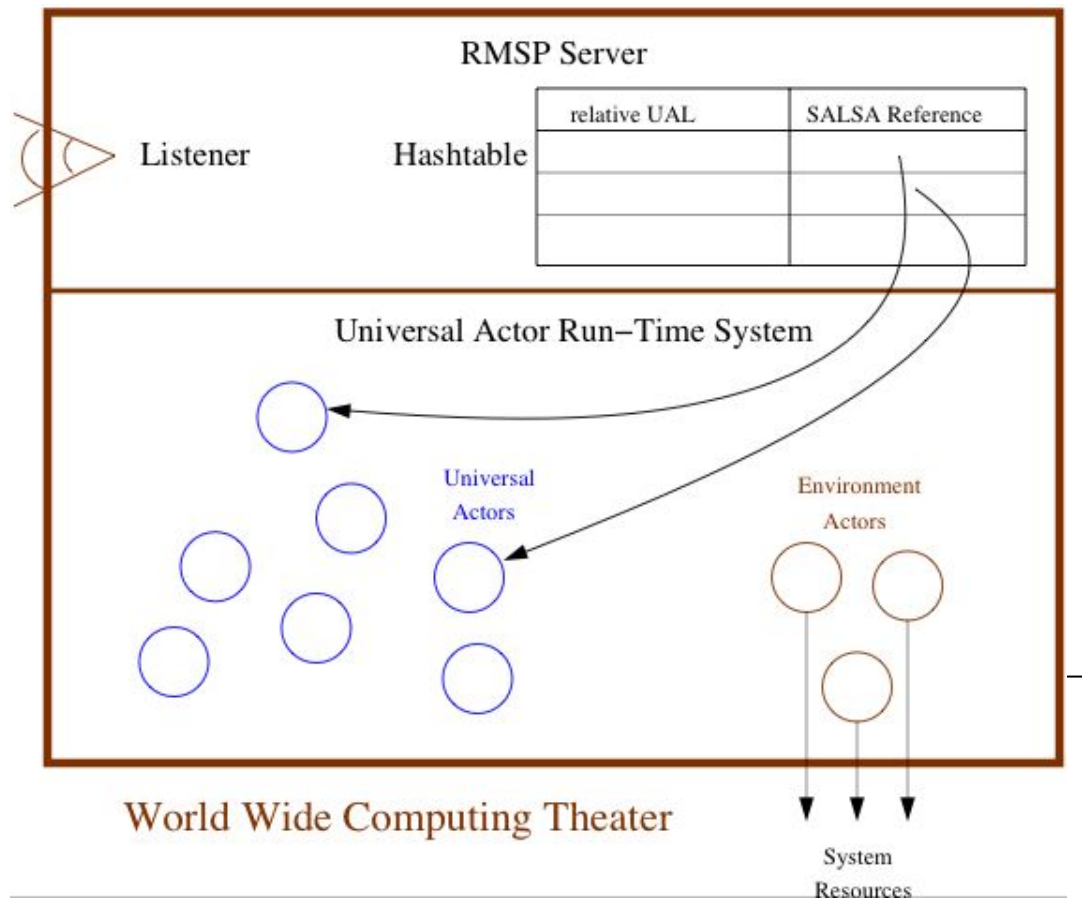
Universal Actors

- 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



WWC Theaters



WWC Theaters

- Theaters provide an execution environment for actors.
- Provide a layer beneath actors for message passing and migration.
- Example theater reference, called Universal Actor Locator:

rmisp://wwc.cs.rpi.edu:4040

Theater's IP
address and port.

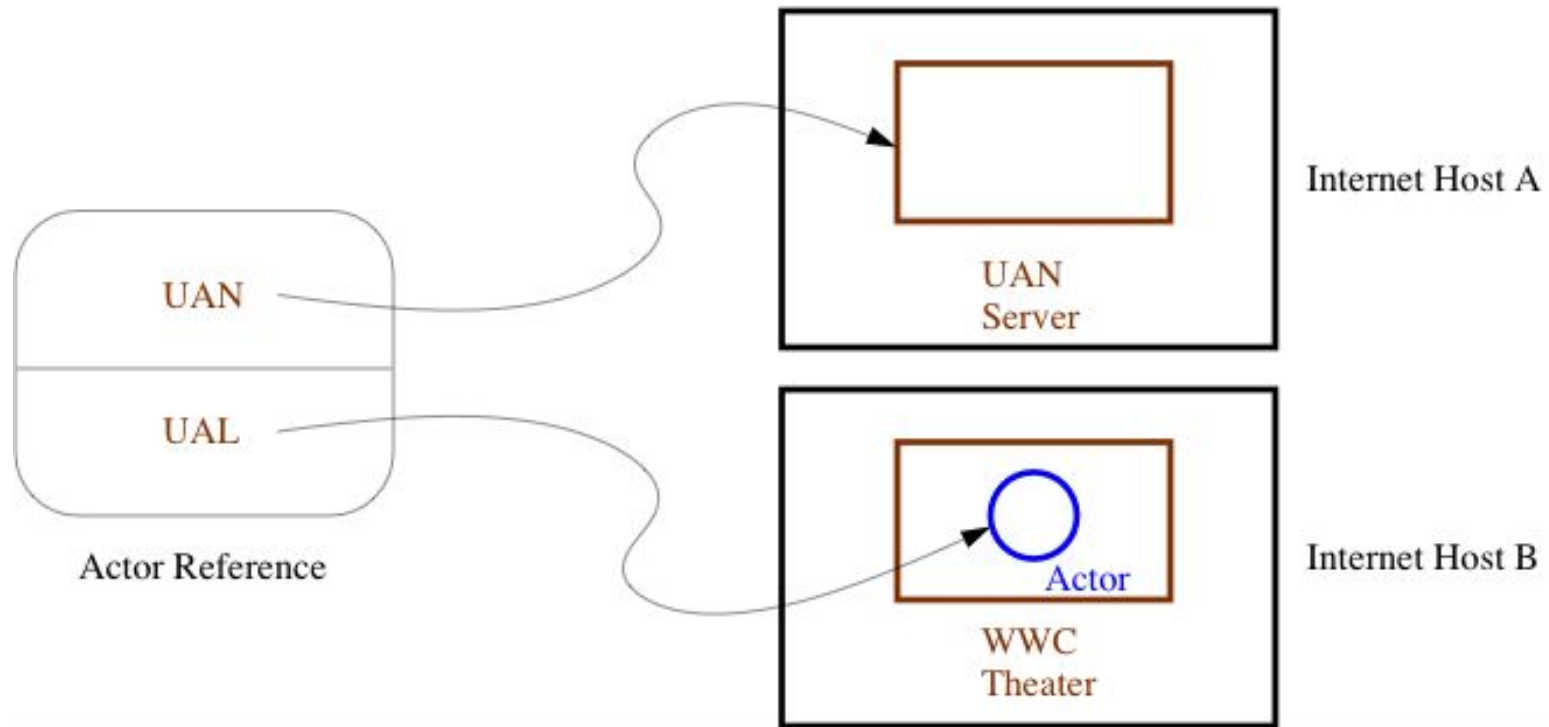
Environment Actors

- 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

- Theaters use the Remote Message Sending Protocol (RMSP) to send messages between remote actors.
- RMSP is implemented using Java object serialization.
- RMSP protocol is used for both message sending and actor migration.
- When an actor migrates, its universal actor locator (UAL) changes but its universal actor name (UAN) does not.
- RMSP is transparent to SALSA programmers. Messages are delivered in a location-transparent manner.

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
- UANP is transparent to SALSA programmers. UANP messages are sent by the WWC run-time system, upon actor creation, lookup, migration, and garbage collection.

UANP Implementations

- Default naming service implementation stores UAN to UAL mapping in name servers specified by 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:

Camron Tolman and Carlos Varela. *A Fault-Tolerant Home-Based Naming Service For Mobile Agents*. In Proceedings of the XXXI Conferencia Latinoamericana de Informática (CLEI), Cali, Colombia, October 2005.

Tolman C. *A Fault-Tolerant Home-Based Naming Service for Mobile Agents*. Master's Thesis, Rensselaer Polytechnic Institute, April 2003.

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

```
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

```
module mcell;

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

```
module mcell;

behavior GetCellValue {

    void act( String[] args ) {
        if (args.length != 1){
            standardOutput <- println("Usage:
                salsa mcell.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 (“yourhost:yourport” ) @
a <- printItinerary();
```


Moving Cell Tester Example

```
module mcell;

behavior MovingCellTester {

    void act( String[] args ) {

        if (args.length != 3){
            standardOutput <- println("Usage:
                salsa mcell.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 );
    }
}
```

Agent Migration Example

```
module migrate;

behavior Migrate {

    void print() {
        standardOutput<-println( "Migrate actor is here." );
    }

    void act( String[] args ) {

        if (args.length != 3) {
            standardOutput<-println("Usage: salsa migrate.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

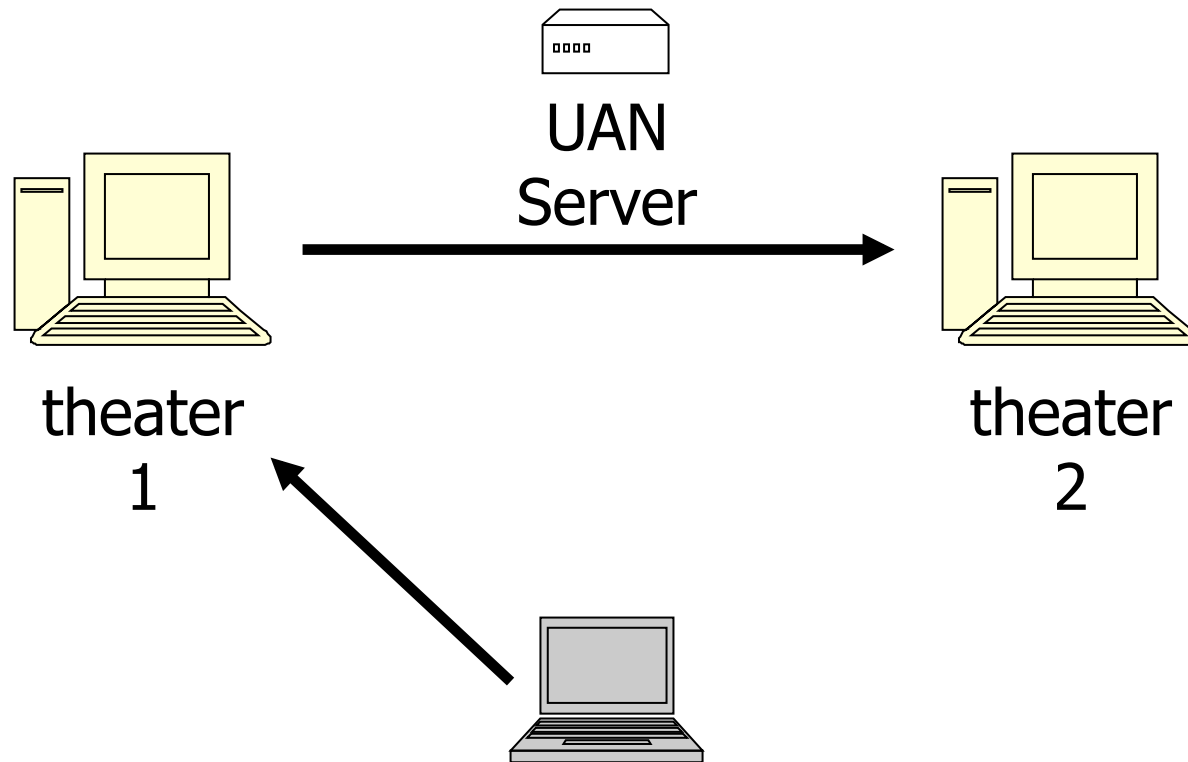
- The program must be given *valid* universal actor name and locators.
 - Appropriate name server and theaters must be running.
 - Theater must be run from directory with access to migrating actor behavior code.
- After remotely creating the actor. It sends the `print` message to itself before migrating to the second theater and sending the message again.

Compilation and Execution

```
$ salsac migrate/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.
$ salsa migrate.Migrate
Usage: salsa migrate.Migrate <UAN> <srcUAL> <destUAL>
```

- Compile Migrate.salsa file into Migrate.class.
- Execute Name Server
- Execute Theater 1 and Theater 2 (with access to migrate directory)
- 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

Host	Location	OS/JVM	Processor
yangtze.cs.uiuc.edu	Urbana IL, USA	Solaris 2.5.1 JDK 1.1.6	Ultra 2
vulcain.ecoledoc.lip6.fr	Paris, France	Linux 2.2.5 JDK 1.2pre2	Pentium II 350Mhz
solar.isr.co.jp	Tokyo, Japan	Solaris 2.6 JDK 1.1.6	Sparc 20

Local actor creation	386us
Local message sending	148 us
LAN message sending	30-60 ms
WAN message sending	2-3 s
LAN minimal actor migration	150-160 ms
LAN 100Kb actor migration	240-250 ms
WAN minimal actor migration	3-7 s
WAN 100Kb actor migration	25-30 s

Address Book Service

```
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");
    }
  }
}
```

Address Book Add User Example

```
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

```
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 <- print(name + "'s email: ") @
      book <- getEmail(name) @
      standardOutput <- println(token);
    } catch (MalformedUANException e){
      standardError<-println(e);
    }
  }
}
```

Address Book Migrate Example

```
module addressbook;

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

Exercises

78. How would you implement the join continuation linguistic abstraction considering different potential distributions of its participating actors?
79. Download and execute the `Migrate.salsa` example.
80. Modify the lock example in the SALSA distribution to use a wait/notify protocol, as opposed to “busy-waiting” (or “busy-asking”).
81. CTM Exercise 11.11.3 (page 746). Implement the example using SALSA/WWC.
82. PDCS Exercise 9.6.9 (page 204).