Mobility, garbage collection, load balancing, visualization (SALSA) Fault-tolerance, hot code loading (Erlang) (PDCS 9; CPE 7*)

Carlos Varela
Rensselaer Polytechnic Institute

October 22, 2021

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

Advanced Features of Actor Languages

- SALSA and Erlang support the basic primitives of the actor model:
  - Actors can create new actors.
  - Message passing is asynchronous.
  - State is encapsulated.
  - Run-time ensures fairness.
- SALSA also introduces advanced coordination abstractions: tokens, join blocks, and first-class continuations; SALSA supports distributed systems development including actor mobility and garbage collection. Research projects have also investigated load balancing, malleability (IOS), scalability (COS), and visualization (OverView).
- Erlang introduces a selective receive abstraction to enforce different orders of message delivery, including a timeout mechanism to bypass blocking behavior of `receive` primitive. Erlang also provides error handling abstractions at the language level, and dynamic (hot) code loading capabilities.

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.

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:

`rmsp://wwc.cs.rpi.edu:4040`

Theater's IP address and (optional) port.

Migration

- Obtaining a remote actor reference and migrating the actor.

```
TravelAgent a = (TravelAgent)
    TravelAgent.getReferenceByName
                 (“uan://myhost/ta”);

a <- migrate( “yourhost:yourport” ) @
a <- printItinerary();
```

Agent Migration Example

```
module migrate;

behavior Migrate {

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

   void act( String[] args ) {

      if (args.length != 3) {
        standardError<-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 the code for the migrating actor's behavior.

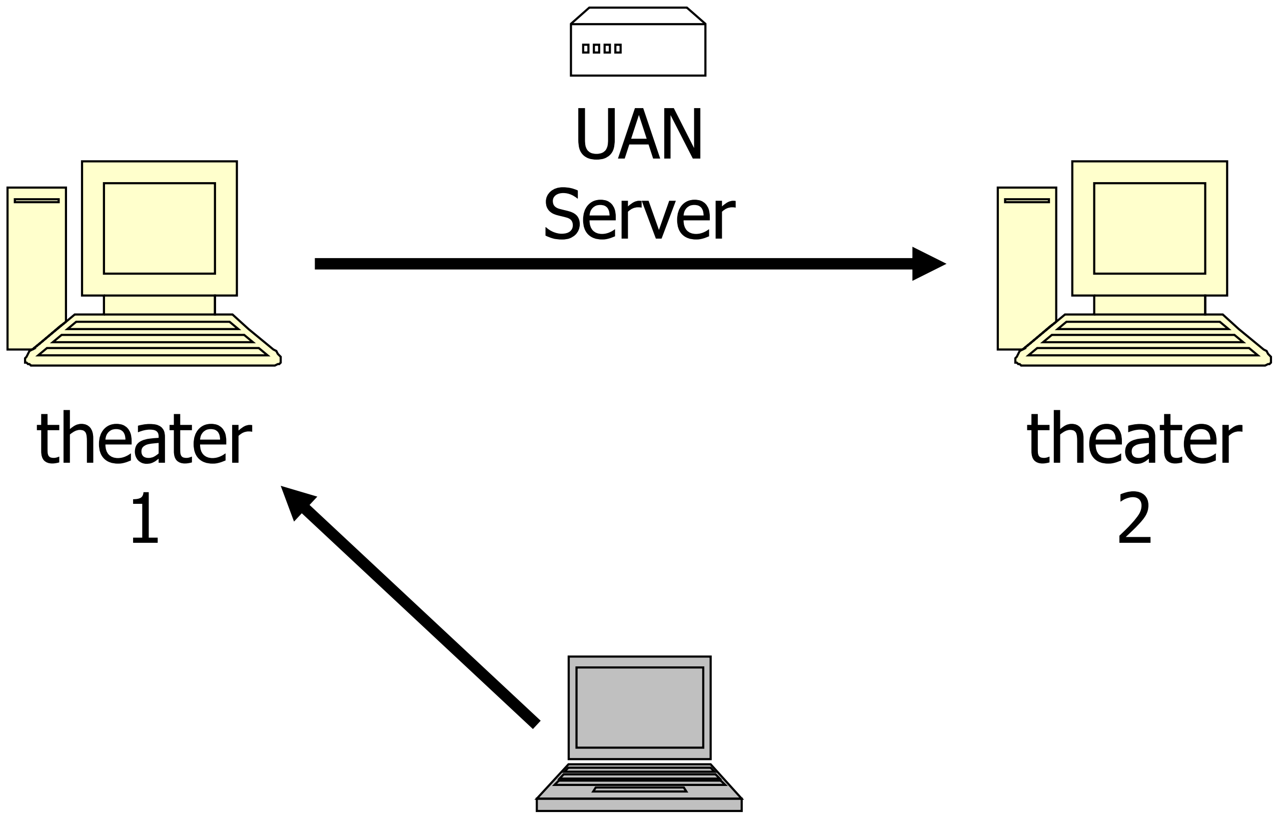
- After remotely creating the actor. It sends the `print` message to itself before migrating to the second theater and sending the message again.

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>
```

1. Compile `Migrate.salsa` file into `Migrate.java`, and then bytecode (`.class` files.)
2. Execute Name Server
3. Execute Theater 1 and Theater 2 (with access to migrate directory)
4. Execute Migrate in any computer with Internet access

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

Reference Cell Service Example

```
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;
    }
}
```

`implements ActorService` signals that actors with this behavior are not to be garbage collected.

Moving Cell Tester Example

```
module dcell;

behavior MovingCellTester {

    void act( String[] args ) {

      if (args.length != 3){
          standardError <- println(“Usage:
            salsa dcell.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 );
    }
}
```

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 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));
    }
}
```

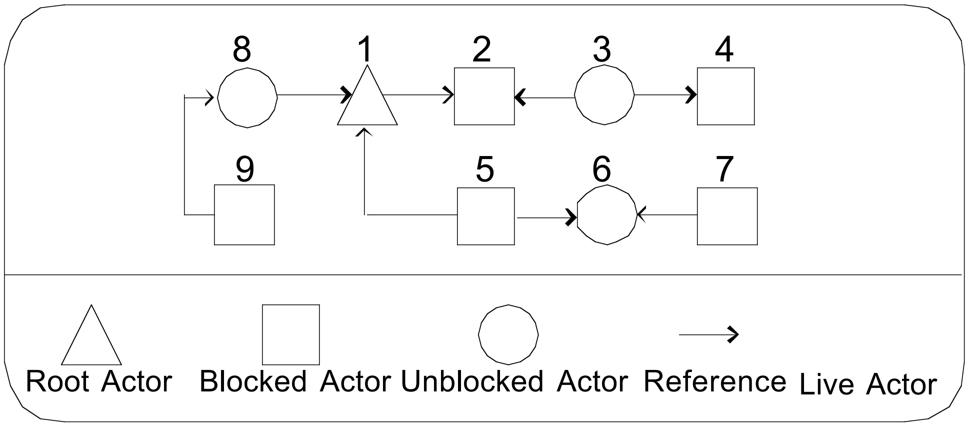
Actor Garbage Collection

- Implemented since SALSA 1.0 using *pseudo-root* approach.
- Includes distributed cyclic garbage collection.
- For more details, please see:

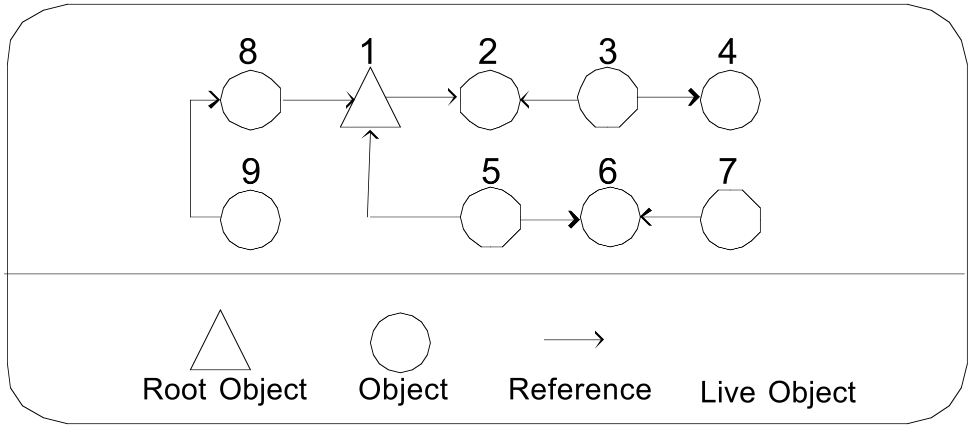
Wei-Jen Wang and Carlos A. Varela. Distributed Garbage Collection for Mobile Actor Systems: The Pseudo Root Approach. In *Proceedings of the First International Conference on Grid and Pervasive Computing (GPC 2006),* Taichung, Taiwan, May 2006. Springer-Verlag LNCS.

Wei-Jen Wang, Carlos Varela, Fu-Hau Hsu, and Cheng-Hsien Tang. Actor Garbage Collection Using Vertex-Preserving Actor-to-Object Graph Transformations. In *Advances in Grid and Pervasive Computing*, volume 6104 of *Lecture Notes in Computer Science*, Bologna, pages 244-255, May 2010. Springer Berlin / Heidelberg.

Challenge 1: Actor GC vs. Object GC



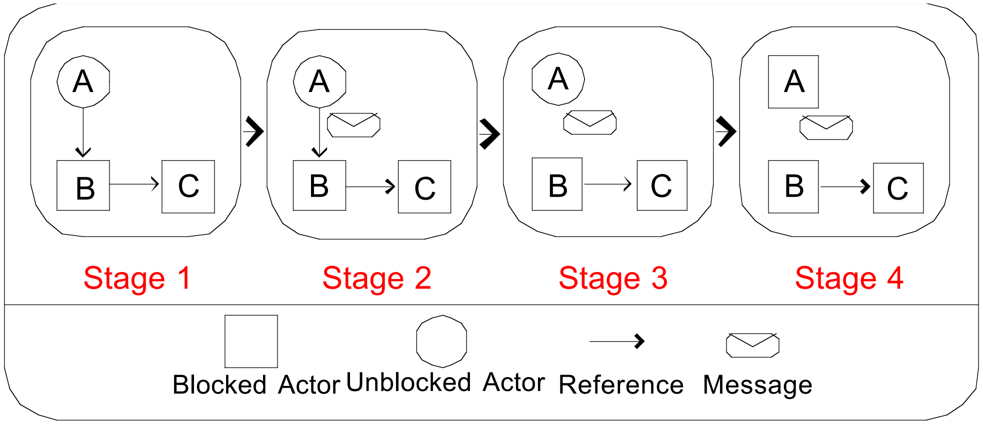
Actor Reference Graph



Passive Object Reference Graph

Challenge 2: Non-blocking communication

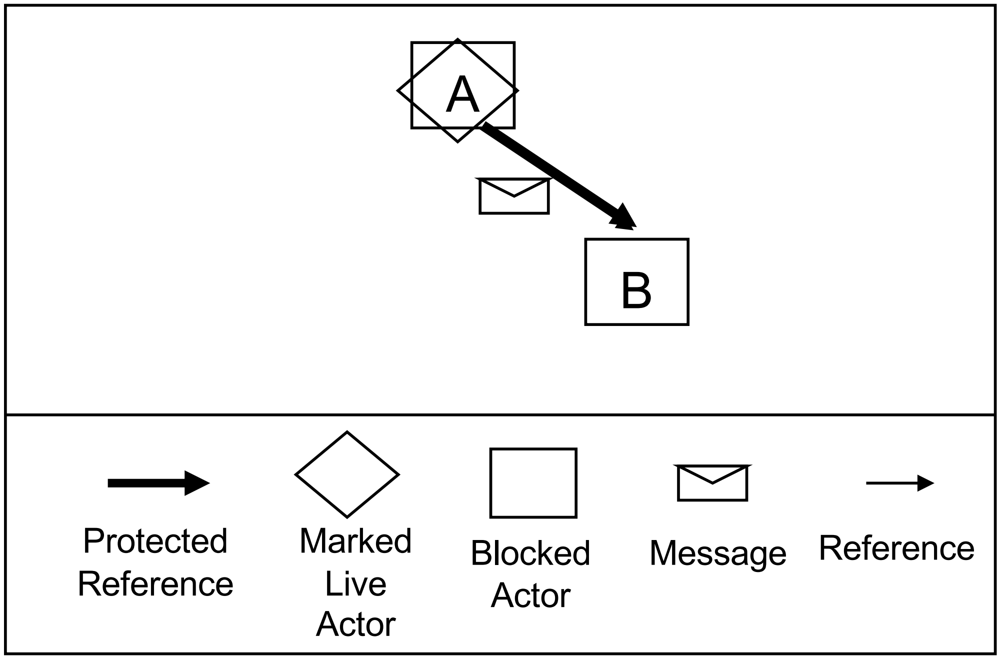
- Following references to mark live actors is not safe!



An example of mutation and asynchronous delivery of message

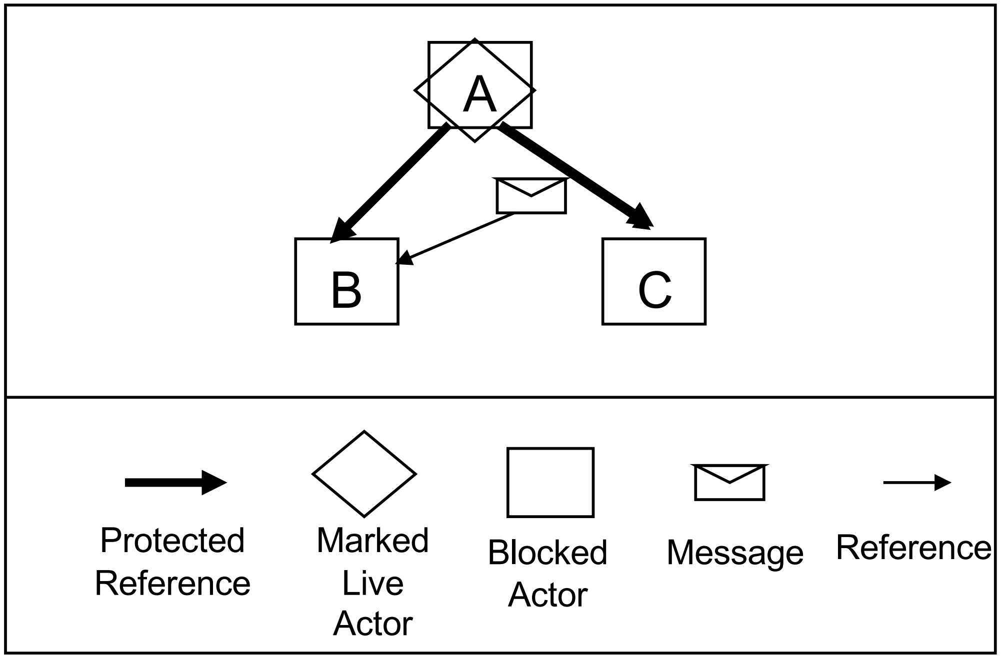
Challenge 2: Non-blocking communication

- Following references to mark live actors is not safe!
- What can we do?
  - We can ***protect the reference from deletion*** and ***mark the sender live*** until the sender knows the message has arrived



Challenge 2: Non-blocking communication (continued)

- How can we guarantee the safety of an actor referenced by a message?
- The solution is to ***protect the reference from deletion*** and ***mark the sender live*** until the sender knows the message has arrived



Challenge 3: Distribution and Mobility

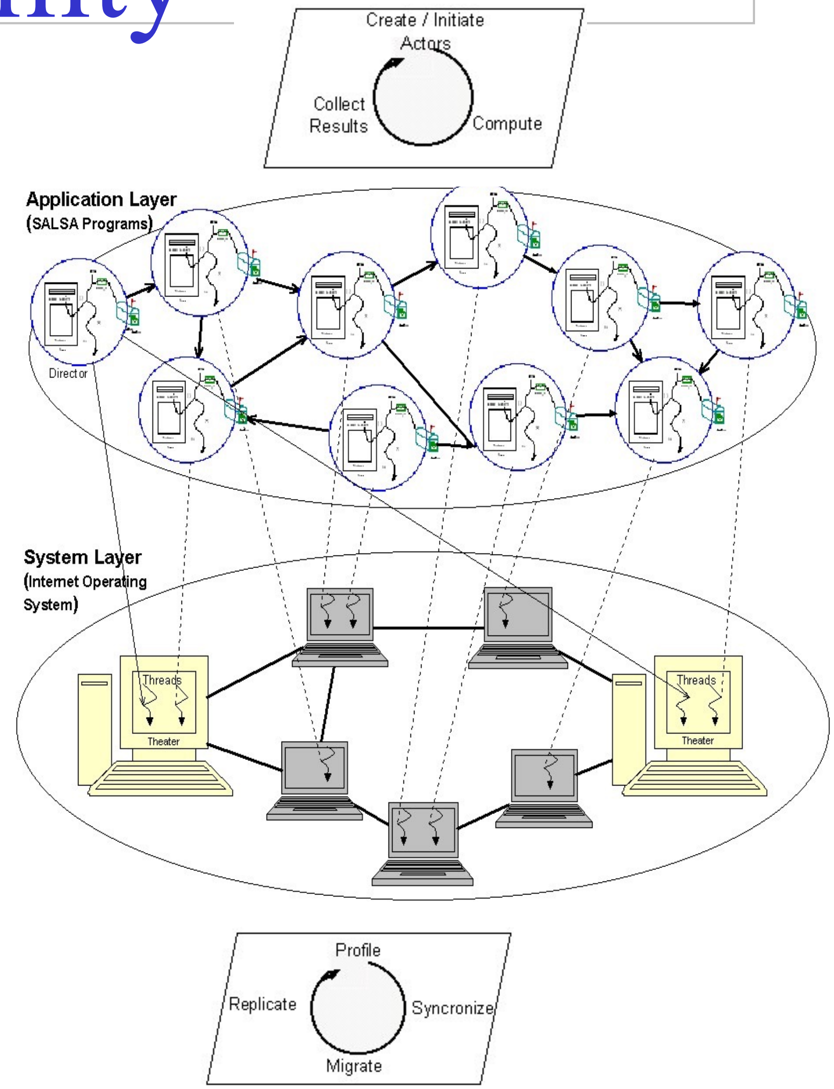
- What if an actor is remotely referenced?
  - We can ***maintain an inverse reference list*** (only visible to the garbage collector) to indicate whether an actor is referenced.
  - Three operations change inverse references: ***actor creation***, ***reference passing***, and ***reference deletion***.
  - The inverse reference registration must be based on ***non-blocking*** and ***non-First-In-First-Out*** communication!

The Pseudo Root Approach

- Pseudo roots:
  - Treat unblocked actors, migrating actors, and roots as pseudo roots.
  - Map ***in-transit messages and references*** into ***protected references*** and ***pseudo roots***
  - Use inverse reference list (only visible to garbage collectors) to identify remotely referenced actors
- Actors which are not reachable from any pseudo root are garbage.

IOS: Load Balancing and Malleability

- Middleware
  - A software layer between distributed applications and operating systems.
  - Alleviates application programmers from directly dealing with distribution issues
    - Heterogeneous hardware/O.S.s
    - Load balancing
    - Fault-tolerance
    - Security
    - Quality of service
- Internet Operating System (IOS)
  - A decentralized framework for adaptive, scalable execution
  - Modular architecture to evaluate different distribution and reconfiguration strategies

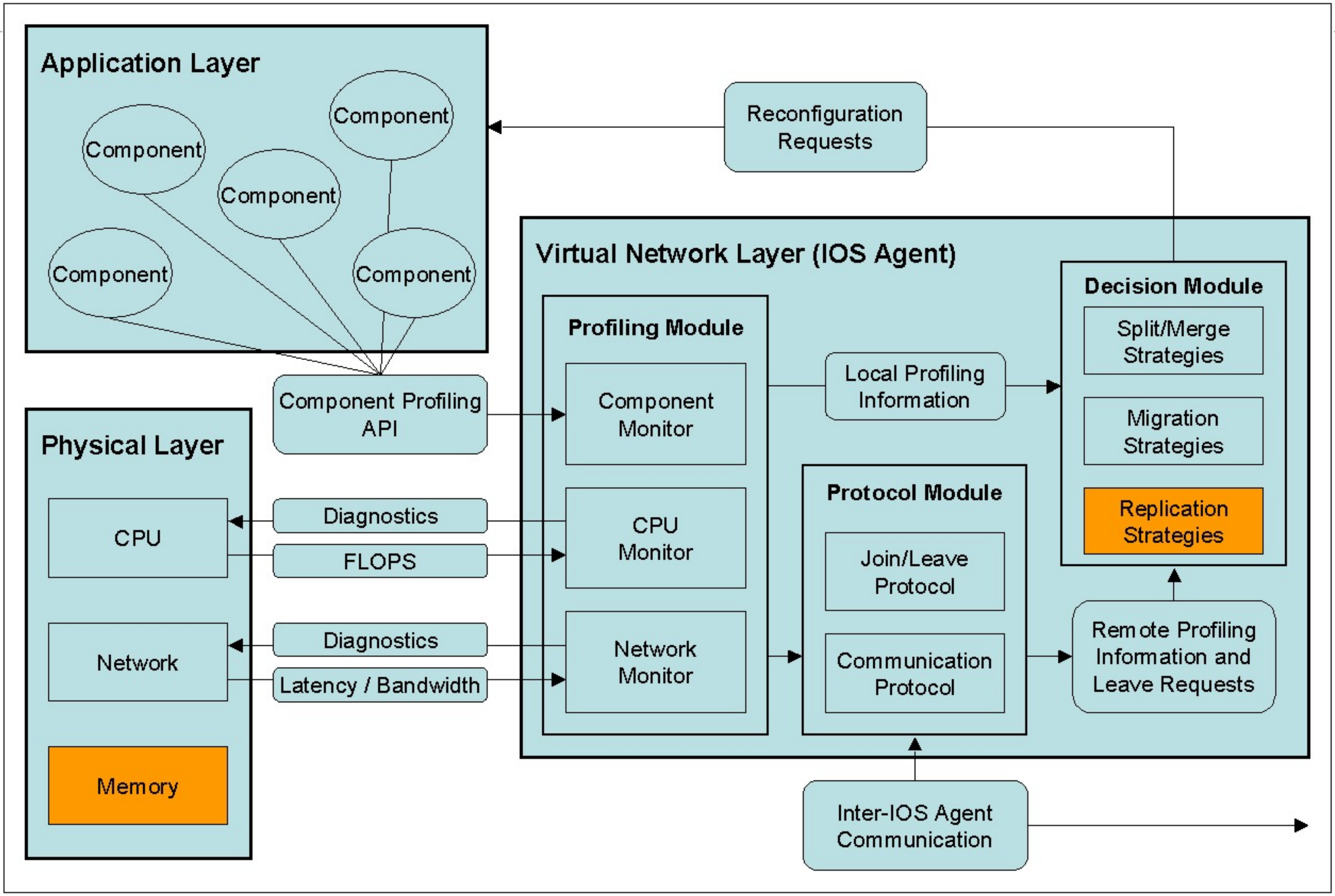


K. El Maghraoui, T. Desell, B. Szymanski, and C. Varela, "The Internet Operating System: Middleware for Adaptive Distributed Computing", ***International Journal of High Performance Computing and Applications***, 2006.

K. El Maghraoui, T. Desell, B. Szymanski, J. Teresco and C. Varela, "Towards a Middleware Framework for Dynamically Reconfigurable Scientific Computing", ***Grid Computing and New Frontiers of High Performance Processing***, Elsevier 2005.

T. Desell, K. El Maghraoui, and C. Varela, "*Load Balancing of Autonomous Actors over Dynamic Networks*", HICSS-37 Software Technology Track, Hawaii, January 2004. 10pp.

Middleware Architecture



IOS Architecture

- IOS middleware layer
  - A Resource Profiling Component
    - Captures information about actor and network topologies and available resources
  - A Decision Component
    - Takes migration, split/merge, or replication decisions based on profiled information
  - A Protocol Component
    - Performs communication with other agents in virtual network (e.g., peer-to-peer, cluster-to-cluster, centralized.)

A General Model for Weighted Resource-Sensitive Work-Stealing (WRS)

- Given:
  A set of resources, $R = \{r_0 \ldots r_n\}$
  A set of actors, $A = \{a_0 \ldots a_n\}$
  ω is a weight, based on importance of the resource r to the performance of a set of actors A

  $0 \leq \omega(r,A) \leq 1$
  $\Sigma^{\text{all } r}\, \omega(r,A) = 1$

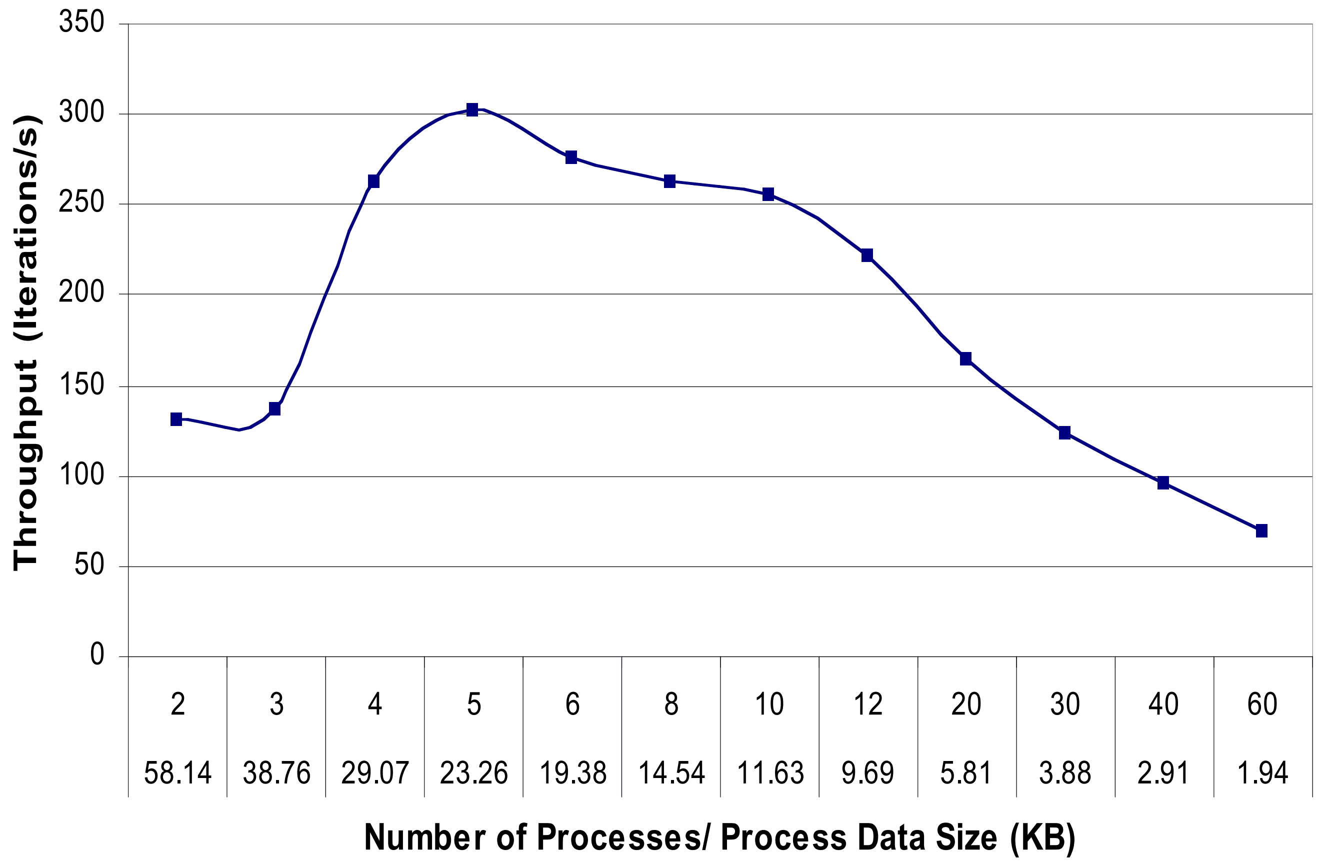
  $\alpha(r,f)$ is the amount of resource r available at foreign node f
  $\upsilon(r,l,A)$ is the amount of resource r used by actors A at local node l
  $M(A,l,f)$ is the estimated cost of migration of actors A from l to f
  $L(A)$ is the average life expectancy of the set of actors A

- The predicted increase in overall performance Γ gained by migrating A from l to f, where $\Gamma \leq 1$:

  $\Delta(r,l,f,A) = (\alpha(r,f) - \upsilon(r,l,A)) / (\alpha(r,f) + \upsilon(r,l,A))$
  $\Gamma = \Sigma^{\text{all } r}\, (\omega(r,A) * \Delta(r,l,f,A)) - M(A,l,f)/(10+\log L(A))$

- When work requested by f, migrate actor(s) A with greatest predicted increase in overall performance, if positive.

Impact of Process/Actor Granularity



Experiments on a dual-processor node (SUN Blade 1000)

Component Malleability

- New type of reconfiguration:
  - Applications can dynamically change component granularity
- Malleability can provide many benefits for HPC applications:
  - Can more adequately reconfigure applications in response to a dynamically changing environment:
    - Can scale application in response to dynamically joining resources to improve performance.
    - Can provide soft fault-tolerance in response to dynamically leaving resources.
  - Can be used to find the ideal granularity for different architectures.
  - Easier programming of concurrent applications, as parallelism can be provided transparently.

Component Malleability

- Modifying application component granularity dynamically (at run-time) to improve scalability and performance.
- SALSA-based malleable actor implementation.
- MPI-based malleable process implementation.
- IOS decision module to trigger split and merge reconfiguration.
- For more details, please see:

El Maghraoui, Desell, Szymanski and Varela, "Dynamic Malleability in MPI Applications", *CCGrid 2007*, Rio de Janeiro, Brazil, May 2007, **nominated for Best Paper Award**.

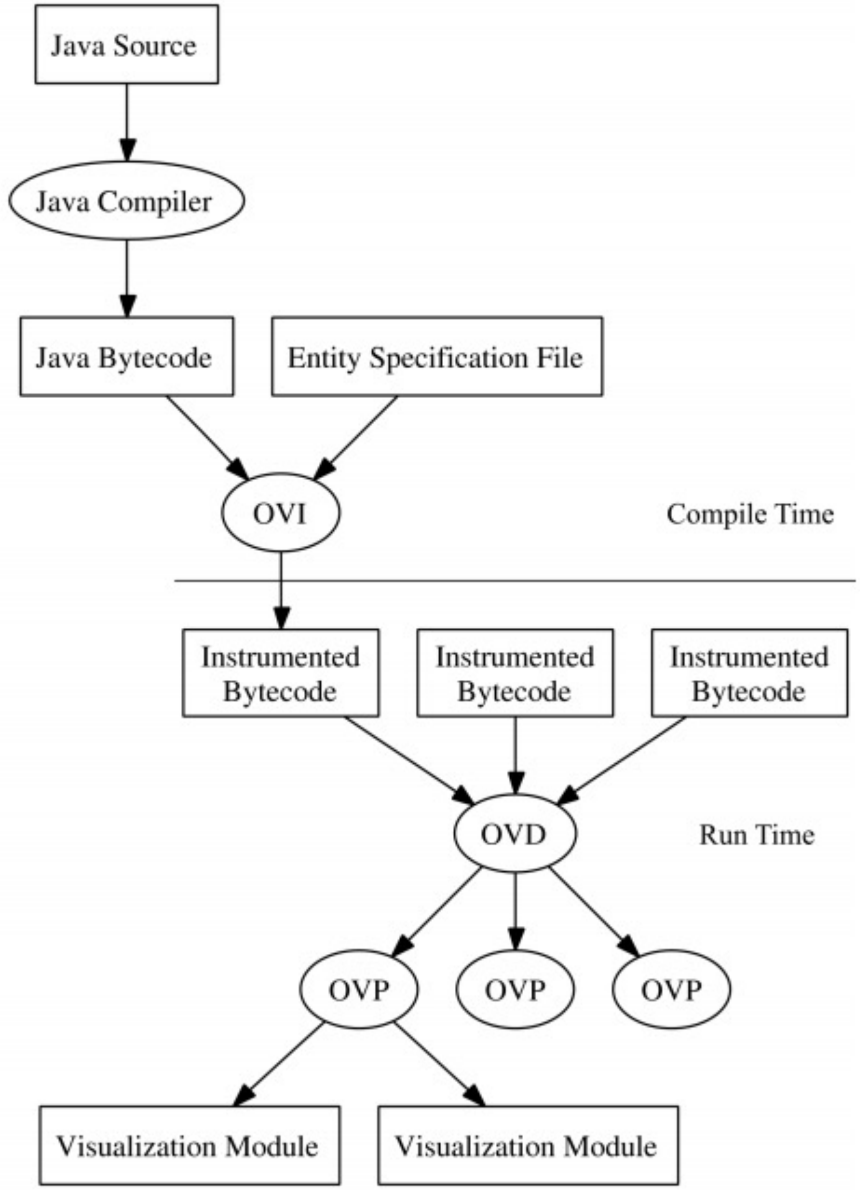
Distributed Systems Visualization

- Generic online Java-based distributed systems visualization tool
- Uses a declarative Entity Specification Language (ESL)
- Instruments byte-code to send events to visualization layer.
- For more details, please see:

T. Desell, H. Iyer, A. Stephens, and C. Varela. OverView: A Framework for Generic Online Visualization of Distributed Systems. In *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS 2004), eclipse Technology eXchange (eTX) Workshop*, Barcelona, Spain, March 2004.

Gustavo A. Guevara S., Travis Desell, Jason Laporte, and Carlos A. Varela. Modular Visualization of Distributed Systems. *CLEI Electronic Journal*, 14:1-17, April 2011. Note: **Best papers from CLEI 2010.**

Overview Architecture

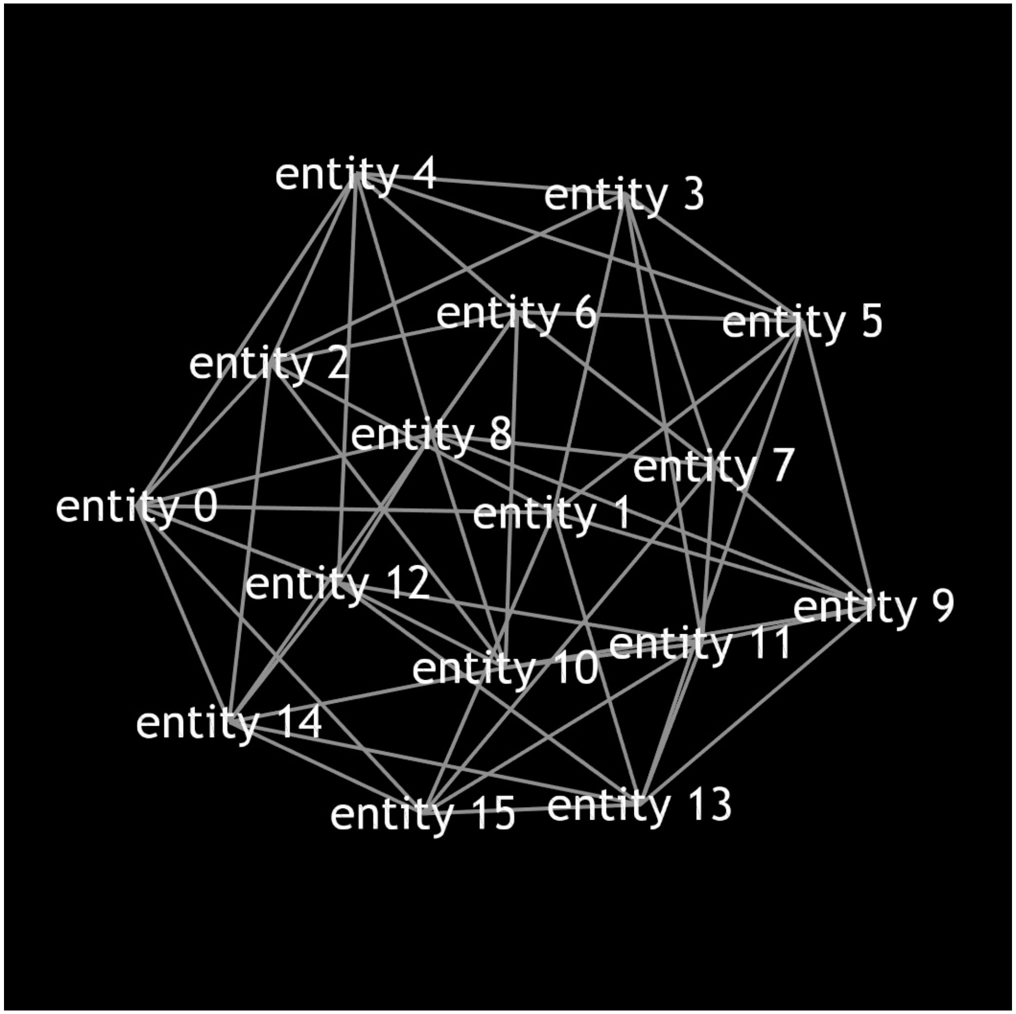


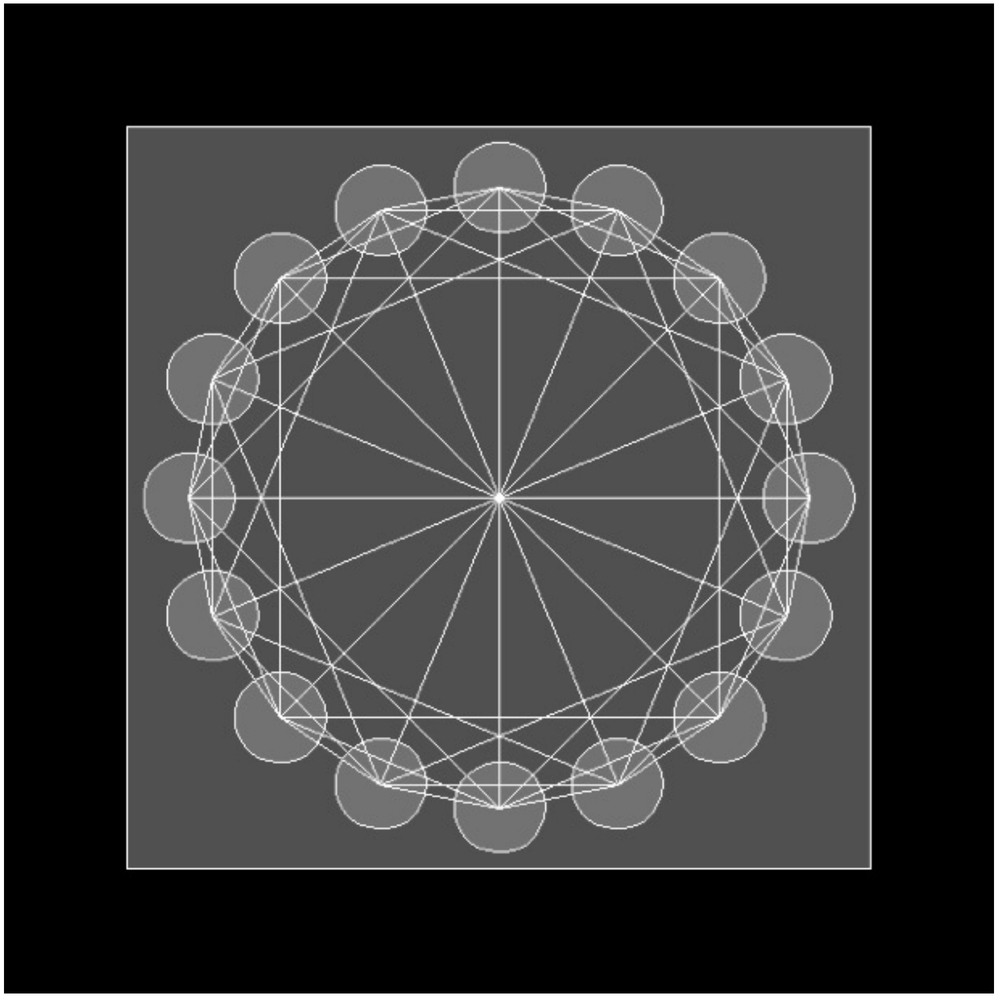
Example Specifications for SALSA

```
entity UniversalActor is salsa.language.UniversalActor$State {
        when start putMessageInMailbox(salsa.language.Message message)
                -> communication(message.getSource().getId(),
                                     message.getTarget().getId());
        when finish finalize()
                -> deletion(this.getId());
}
```

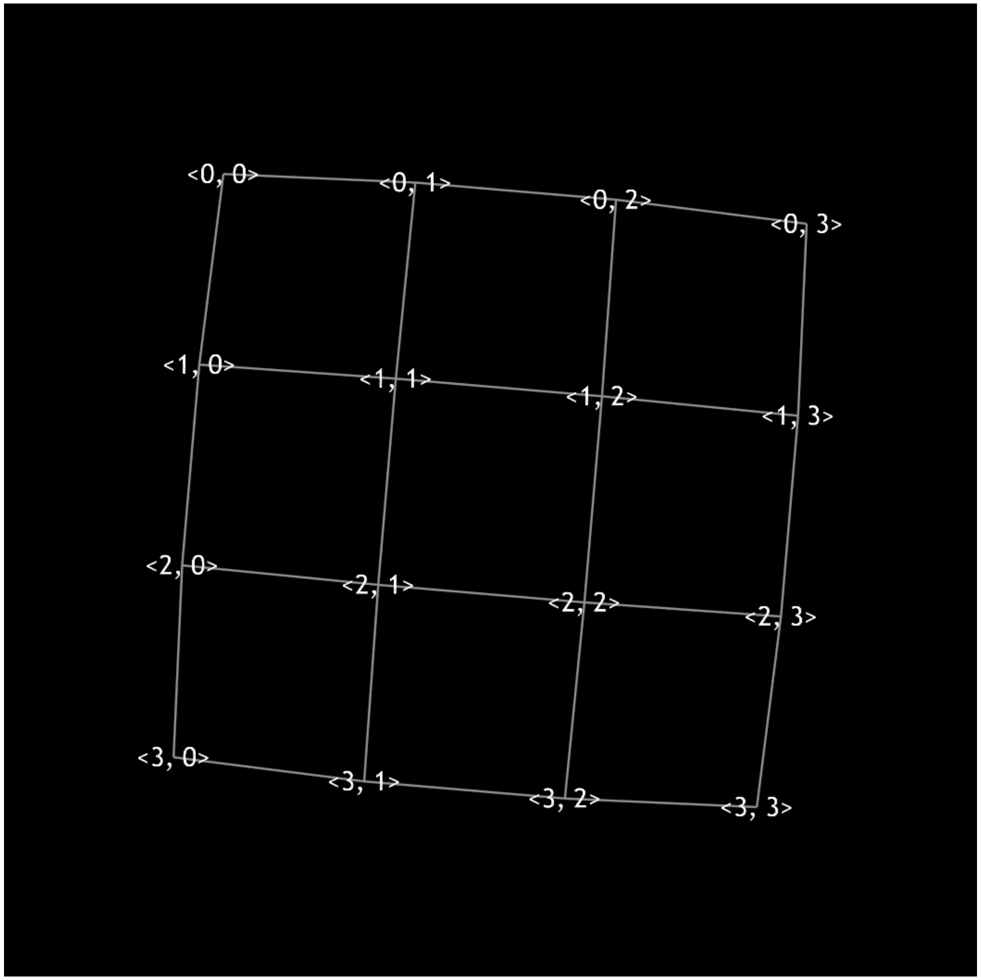
```
entity WWCSystem is wwc.messaging.WWCSystem$State {
        when start createActor(salsa.naming.UAN uan,
                               salsa.naming.UAL ual,
                               java.lang.String className)
                -> creation(uan.getId(), ual.getHostAndPort());

        when start addActor(salsa.language.Actor actor)
                -> migration(actor.getUAN().getId(),
                                 actor.getUAL().getHostAndPort());
}
```

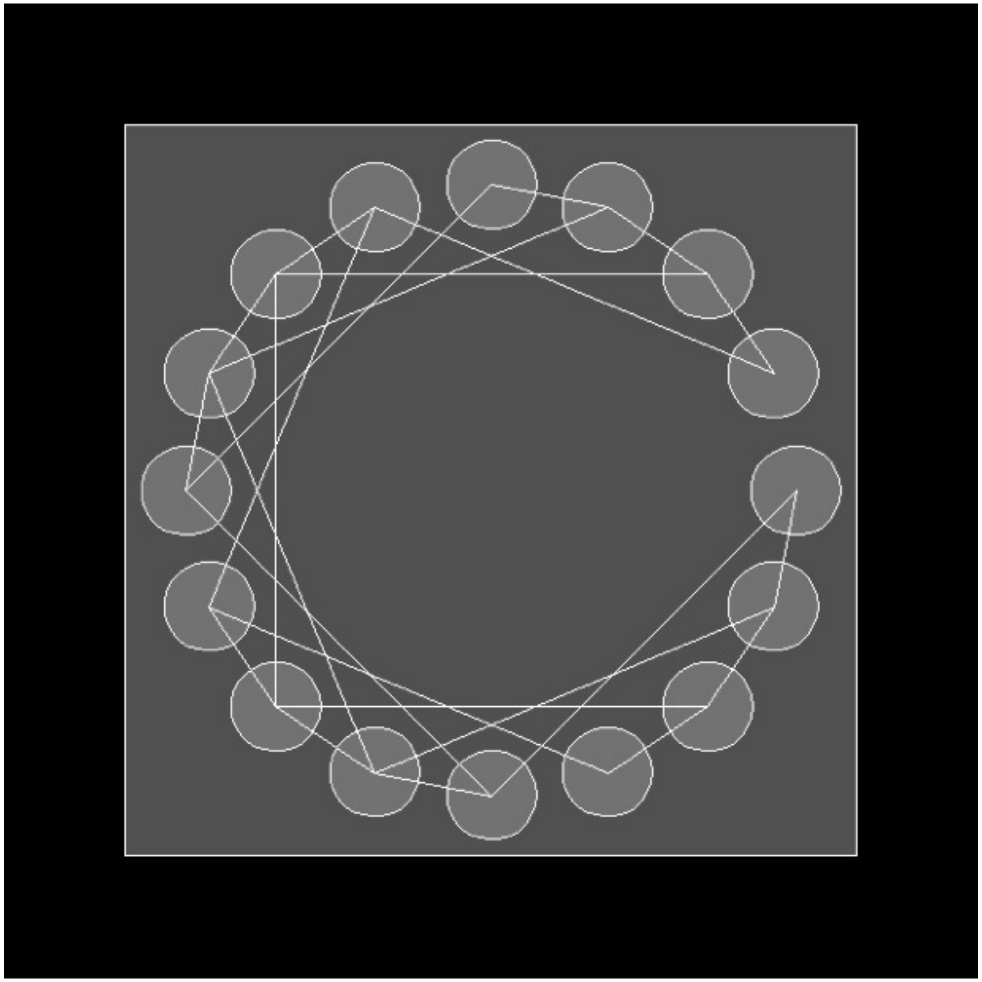
Chord application topology



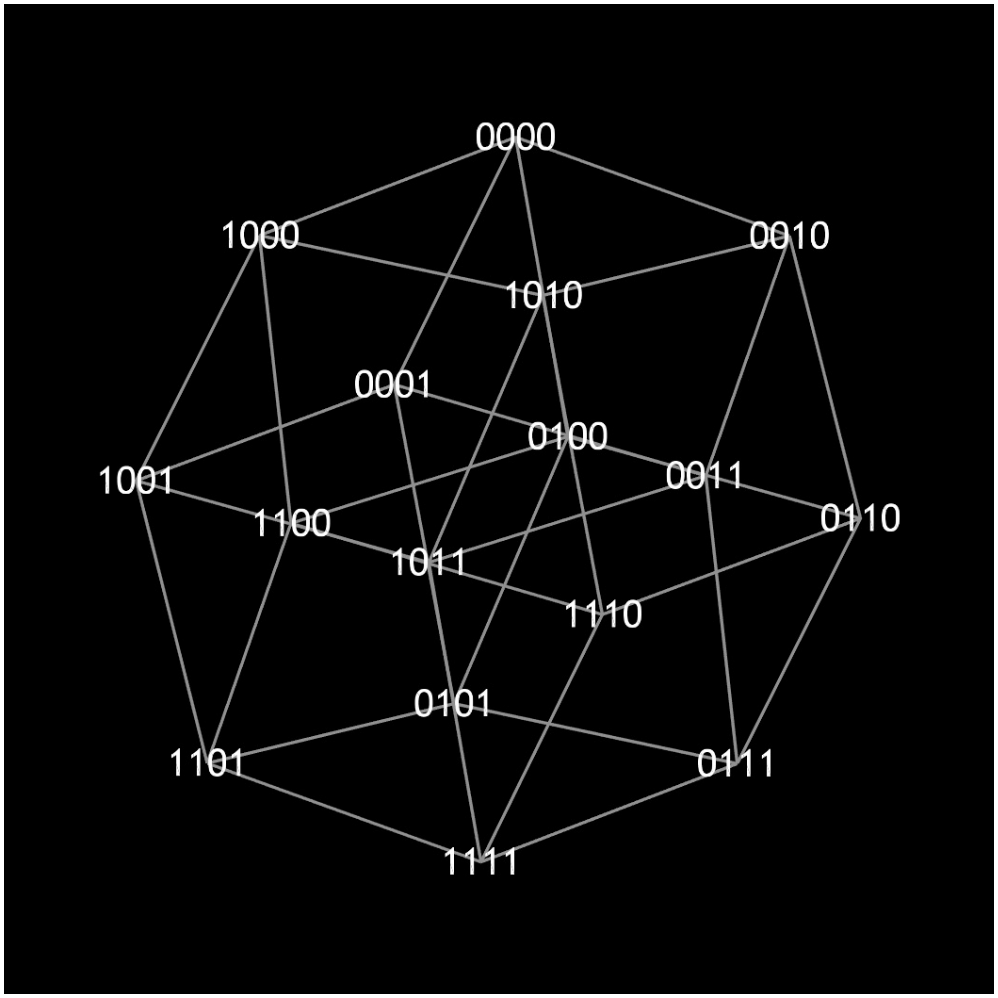


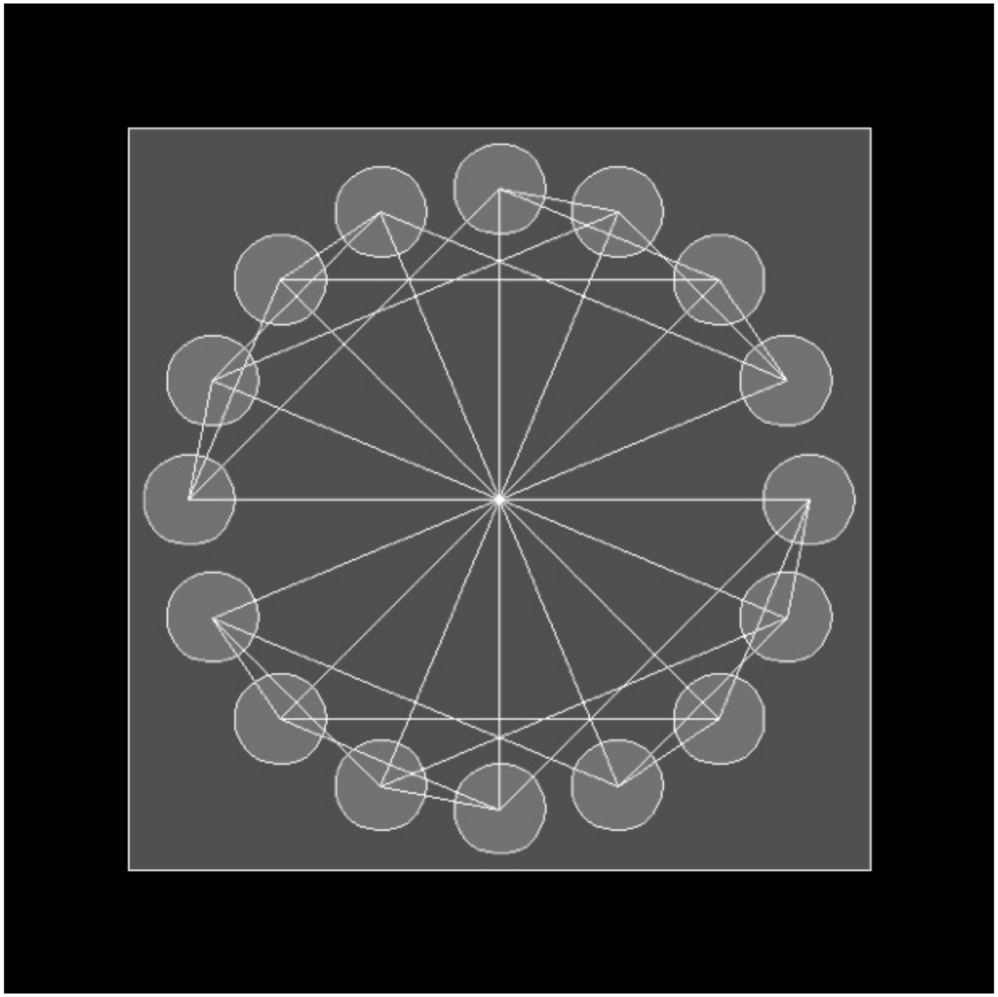
2D-Mesh application topology



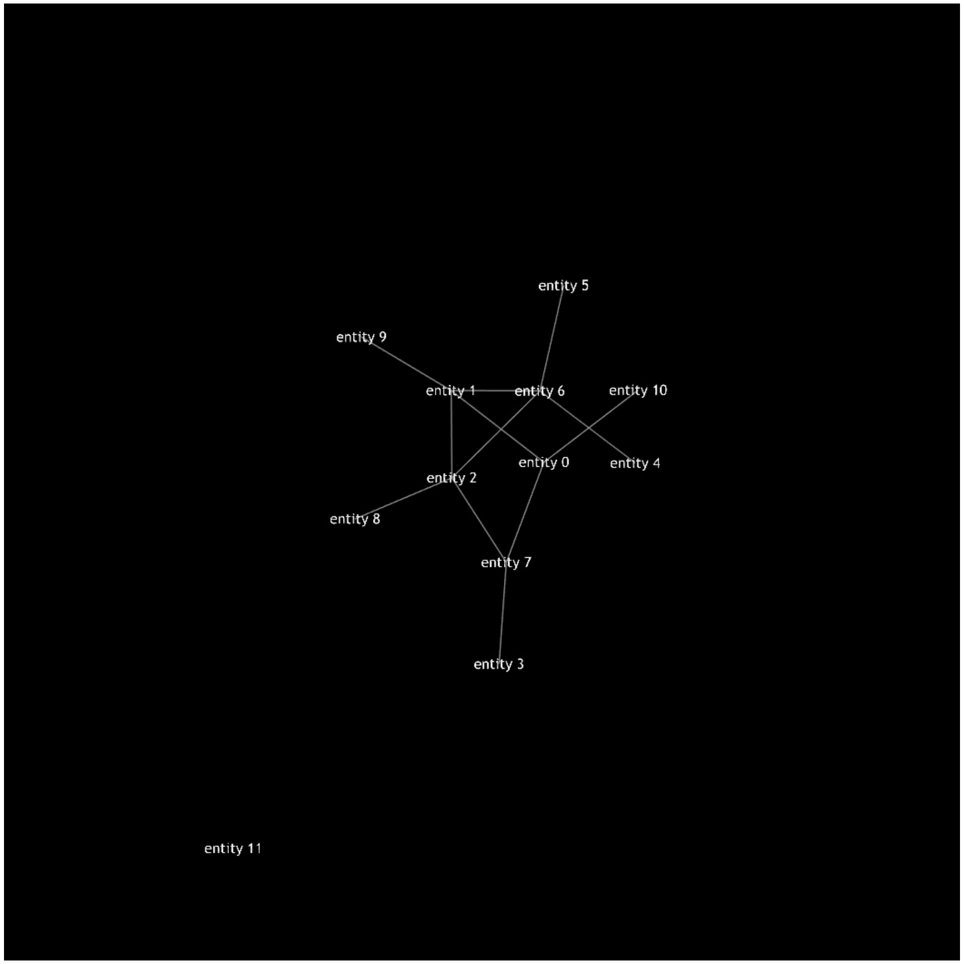


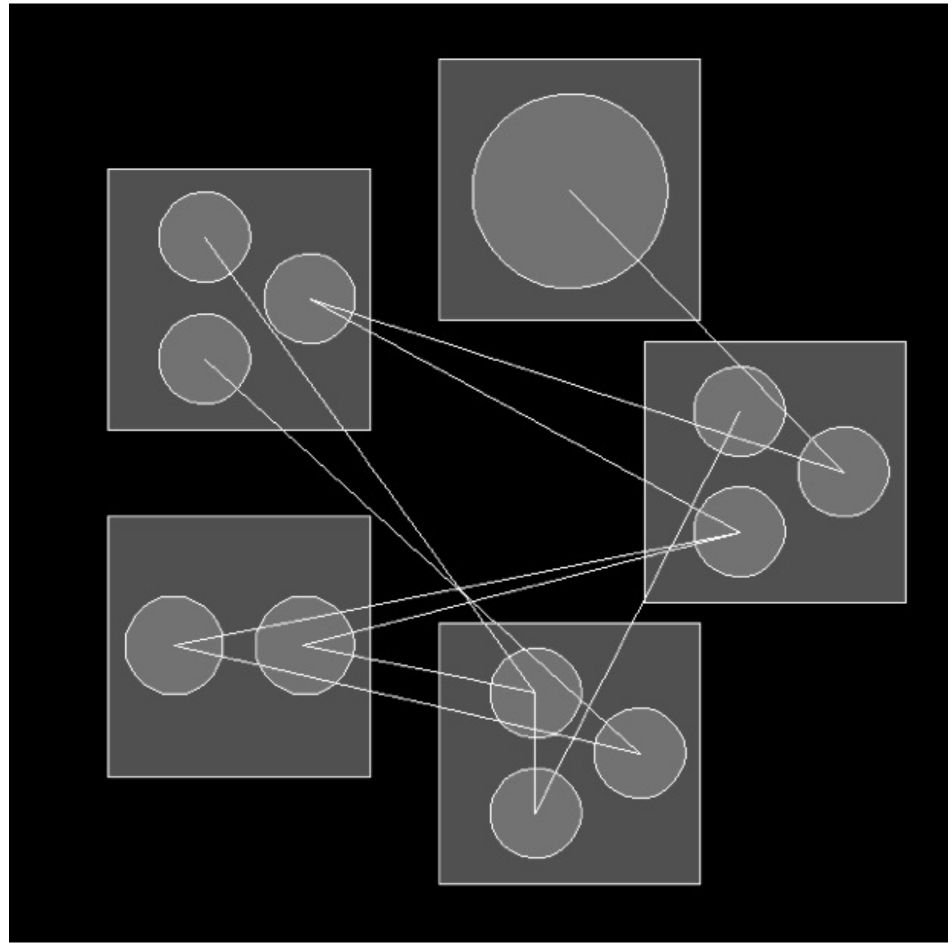
Hypercube application topology





Visualizing distribution and mobility





Open Source Code

- Consider to contribute!
- Visit our web pages for more info:
  - SALSA: http://wcl.cs.rpi.edu/salsa/
  - IOS: http://wcl.cs.rpi.edu/ios/
  - OverView: http://wcl.cs.rpi.edu/overview/
  - COS: http://wcl.cs.rpi.edu/cos/
  - PILOTS: http://wcl.cs.rpi.edu/pilots/
  - MilkyWay@Home: http://milkyway.cs.rpi.edu/

Erlang Language Support for Fault-Tolerant Computing

- Erlang provides linguistic abstractions for:
  - Error detection.
    - Catch/throw exception handling.
    - Normal/abnormal process termination.
    - Node monitoring and exit signals.
  - Process (actor) groups.
  - Dynamic (hot) code loading.

Exception Handling

- To protect sequential code from errors:

```
catch Expression
```

If failure does not occur in **`Expression`** evaluation, **`catch Expression`** returns the value of the expression.

- To enable non-local return from a function:

```
throw({ab_exception, user_exists})
```

Address Book Example

```
-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));
    …
end.

add(Name, Email, Data) ->
    case getemail(Name, Data) of
       undefined ->
         [{Name,Email}|Data];
       _ ->  % if Name already exists, add is ignored.
         Data
    end.
getemail(Name, Data) -> …
```

Address Book Example with Exception

```
addressbook(Data) ->
  receive
    {From, {addUser, Name, Email}} ->
       case catch add(Name, Email, Data) of
         {ab_exception, user_exists} ->
            From ! {addressbook, no},
            addressbook(Data);
         NewData->
            From ! {addressbook, ok},
            addressbook(NewData)
       end;
    …
end.

add(Name, Email, Data) ->
    case getemail(Name, Data) of
       undefined ->
         [{Name,Email}|Data];
       _ ->       % if Name already exists, exception is thrown.
         throw({ab_exception,user_exists})
    end.
```

Normal/abnormal termination

- To terminate an actor, you may simply return from the function the actor executes (without using tail-form recursion). This is equivalent to calling:

```
exit(normal).
```

- Abnormal termination of a function, can be programmed:

```
exit({ab_error, no_msg_handler})
```

equivalent to:

```
throw({'EXIT',{ab_error, no_msg_handler}})
```

- Or it can happen as a run-time error, where the Erlang run-time sends a signal equivalent to:

```
exit(badarg)            % Wrong argument type
exit(function_clause)   % No pattern match
```

Address Book Example with Exception and Error Handling

```
addressbook(Data) ->
  receive
    {From, {addUser, Name, Email}} ->
       case catch add(Name, Email, Data) of
         {ab_exception, user_exists} ->
            From ! {addressbook, no},
            addressbook(Data);
         {ab_error, What} -> …  % programmer-generated error (exit)
         {'EXIT', What} -> …    % run-time-generated error
         NewData->
            From ! {addressbook, ok},
            addressbook(NewData)
       end;
    …
end.
```

Node monitoring

- 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.

Address Book Client Example with Node Monitoring

```
-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.
```

Process (Actor) Groups

- To create an actor in a specified remote node:

```
Agent = spawn(host, travel, agent, []);
```

- To create an actor in a specified remote node and create a link to the actor:

```
Agent = spawn_link(host, travel, agent, []);
```

An `'EXIT'` signal will be sent to the originating actor if the host node does not exist.

Group Failure

- Default error handling for linked processes is as follows:
  - Normal exit signal is ignored.
  - Abnormal exit (either programmatic or system-generated):
    - Bypass all messages to the receiving process.
    - Kill the receiving process.
    - Propagate same error signal to links of killed process.

- All linked processes will get killed if a participating process exits abnormally.

Dynamic code loading

- To update (module) code while running it:

```
-module(m).
-export([loop/0]).

loop() ->
    receive
        code_switch ->
            m:loop();
        Msg -> ...
            loop()
    end.
```

code_switch message dynamically loads the new module code. Notice the difference between **m:loop()** and **loop()**.

Exercises

57. Download and execute the `Migrate.salsa` example.
58. Download OverView and visualize a Fibonacci computation in SALSA. Observe garbage collection behavior.
59. Download social networking example (PDCS Chapter 11) in SALSA and execute it in a distributed setting.
60. PDCS Exercise 11.8.2 (page 257).
61. Create a ring of linked actors in Erlang.
    a. Cause one of the actors to terminate abnormally and observe default group failure behavior.
    b. Modify default error behavior so that upon an actor failure, the actor ring reconnects.
62. Modify the cell example, so that a new "get_and_set" operation is supported. Dynamically (as cell code is running) upgrade the cell module code to use your new version.