

# Mobile Distributed Programming in X-KLAIM

Lorenzo Bettini and Rocco De Nicola

Dipartimento di Sistemi e Informatica, Università di Firenze  
`{bettini,denicola}@dsi.unifi.it`

SMF-Moby – Bertinoro, 27 April 2005

# Outline of the Lectures

## 1. An introduction to KLAIM - R. De Nicola

- Global Computing, Process Algebra and Linda
- KLAIM
- Implementation issues

## 2. X-KLAIM and its use - L. Bettini

- Main features of X-KLAIM
- Simple Programming Examples
- Programming a Chat

## 3. Expressiveness Results - R. De Nicola

- Encodings and their properties
- A stack of KLAIMs (... and  $\pi$ -calculus)
- Assessment of the encodings

These systems can be seen as Distributed Systems with a number of distinguishing features:

- **Wide area distribution**
- **Variability of interconnection structures**
- **(Physical and Logical) Mobility**
- **Possibility of Disconnections**
- **Indistinguishability of Failures from Slow reactions**
- **Importance of Quality of Service**

Programming Languages would definitely benefit from explicit primitives for

- **Distribution**
  - computing over different (explicit) localities
- **Mobility**
  - moving agents and computations over localities
- **Concurrency**
  - considering parallel and non-deterministic computations
- **Access Rights**
  - maintaining privacy and integrity of data

Languages for Global Computing would of course benefit from formal semantics and associated logics for reasoning on programs behaviour.

## Our Research Goals

Developing a simple *programming language* and associated tools **for network aware and migrating applications** with a tractable semantic theory that permits programs (services) verification.

## Our Starting Points (1980 — ...)

- Process Algebras
  - CCS, CSP, ...
- Calculi and languages for Mobility
  - Pi-calculus, Obliq, Ambients, ...
- Tuple Based Interaction Models
  - Linda
- Modal and Temporal Logics
  - HML, CTL, ACTL,  $\mu$ -calculus, ...

## Process Calculus Flavored

- Small set of basic combinator;
- Clean operational semantics.

## Linda based communication model

- Asynchronous communication;
- Shared tuple spaces;
- Pattern Matching

## Explicit use of localities

- Multiple distributed tuple spaces;
- Code and Process mobility.

# Tuples and Pattern Matching

## Tuples and Templates

`("foo", 10 + 5, !x)`

with

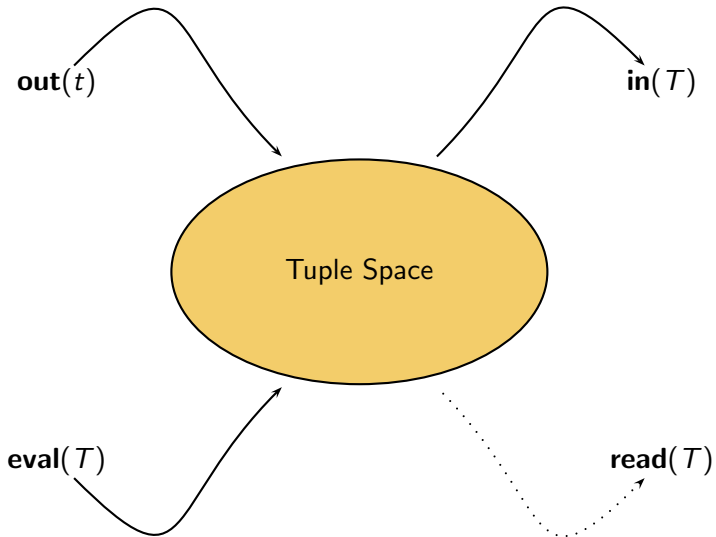
- Formal Fields
- Actual Fields

## Pattern Matching

- Formal fields match any field of the same type
- Actual fields match if identical

`("foo", 10 + 5, true)` matches `(!s, 15, !b)`

# Linda communication model





# Dining Philosophers in Linda

```
phil(i)
  int i;
{
  while(1) {
    think();
    in("room ticket");
    in("chopstick", i);
    in("chopstick", (i + 1) mod 5);
    eat();
    out("chopstick", i);
    out("chopstick", (i + 1) mod 5);
    out("room ticket");
  }
}
```

```
initialize()
{
  int i;
  for (i = 0; i < 5; i++) {
    out("chopstick", i);
    eval(phil(i));
    if (i < 4) out("room ticket");
  }
}
```

## *Localities* to model distribution

- *Physical Locality* (sites)
- *Logical Locality* (names for sites)
- A distinct name *self* (or *here*) indicates the site a process is on.

## *Allocation environment* to associate sites to logical localities

- This avoids the programmers to know the exact physical structure.

## *Process Algebras Operators*

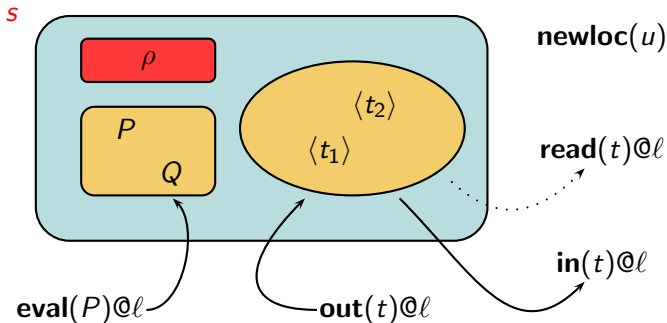
- Sequentialization
- Parallel composition
- Creation of new names

# KLAIM Syntax

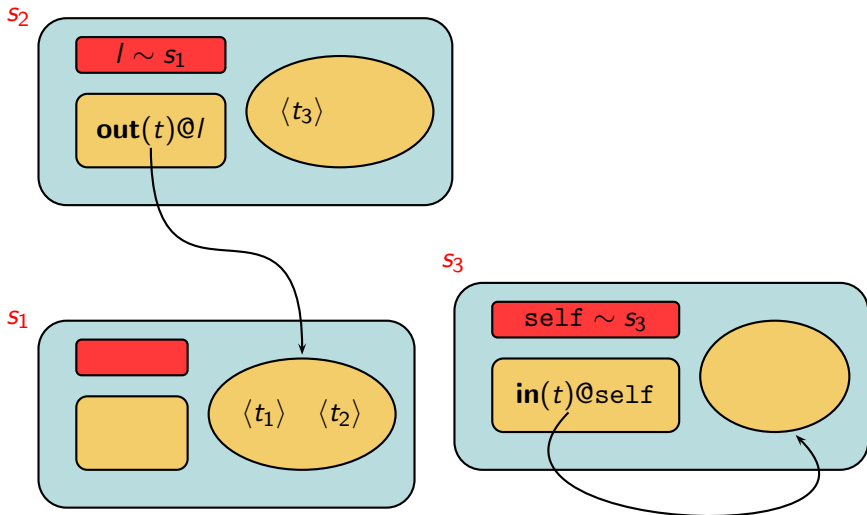
$P$	$::=$	<b>nil</b>	(null process)
		$a.P$	(action prefixing)
		$P_1 \mid P_2$	(parallel composition)
		$P_1 + P_2$	(choice)
		$X$	(process variable)
		$A\langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle$	(process invocation)
$a$	$::=$	<b>out</b> ( $t$ )@ $\ell$   <b>in</b> ( $t$ )@ $\ell$   <b>read</b> ( $t$ )@ $\ell$   <b>eval</b> ( $P$ )@ $\ell$   <b>newloc</b> ( $u$ )	
$t$	$::=$	$e \mid P \mid \ell \mid !x \mid !X \mid !u \mid t_1, t_2$	
$N$	$::=$	$s ::_{\rho} P$	(node)
		$N_1 \parallel N_2$	(net composition)

# KLAIM Nodes

- Locality
- Processes
- Tuple Space
- Allocation Environment

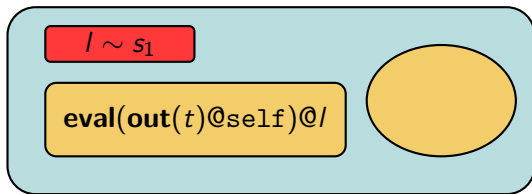


# KLAIM Nets

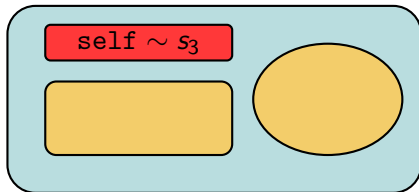


# Dynamic Scoping

$s_2$

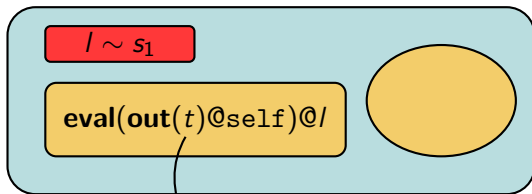


$s_3$

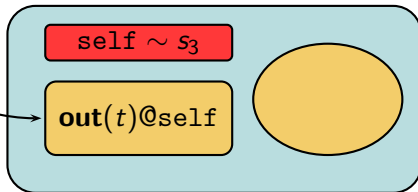


# Dynamic Scoping

$s_2$

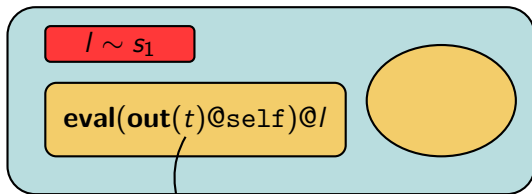


$s_3$

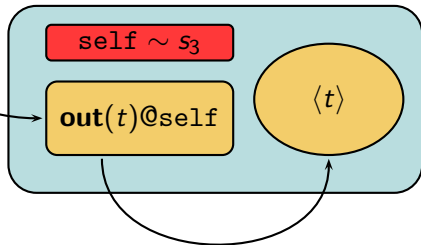


# Dynamic Scoping

$s_2$



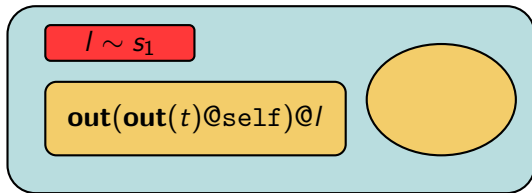
$s_3$



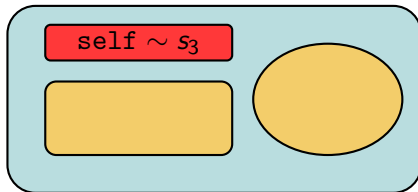


# Static Scoping

$s_2$

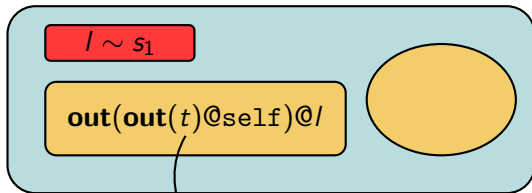


$s_3$

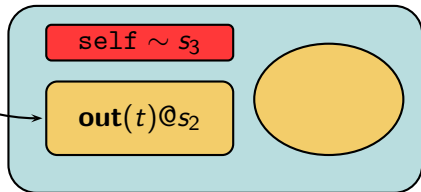


# Static Scoping

$s_2$

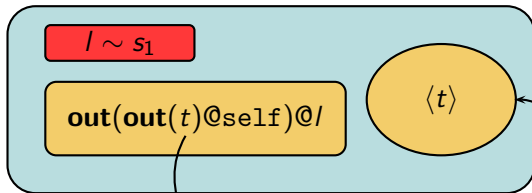


$s_3$

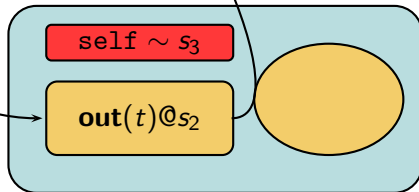


# Static Scoping

$s_2$



$s_3$



$$P_i =$$

*# think...*  
**in**(“chopstick”)@ $c_i$ .  
**in**(“chopstick”)@ $c_{(i+1) \bmod n}$ .  
*# eat...*  
**out**(“chopstick”)@ $c_i$ .  
**out**(“chopstick”)@ $c_{(i+1) \bmod n}$ .  
 $P_i$

$$\begin{aligned} c_0 &:: (\text{“chopstick”}) \parallel p_0 :: P_0 \parallel c_1 :: (\text{“chopstick”}) \parallel p_1 :: P_1 \parallel \\ c_2 &:: (\text{“chopstick”}) \parallel p_2 :: P_2 \parallel c_3 :: (\text{“chopstick”}) \parallel p_3 :: P_3 \parallel \\ c_4 &:: (\text{“chopstick”}) \parallel p_4 :: P_4 \end{aligned}$$

## ATTENTION

This system may deadlock!  
Tickets are needed as in the Linda case.

# Code Mobility in Network Programming

## Remote Evaluation

Any component of a distributed application can invoke services from other components by transmitting both the data needed to perform the service and the code that describes how to perform the service.

## Mobile Agent

A process (i.e., a program and an associated state of execution) on a given node of a network can migrate to a different node where it continues its execution from the current state.

## Code On-Demand

A component of a distributed application running on a given node, can dynamically download some code from a different component and link it to perform a given task.

# Remote Evaluation with KLAIM

Suppose process **Client** wants to require that process **Server** located at location  $l$  evaluates code  $A(x_1, \dots, x_n)$  with values  $v_1, \dots, v_n$  assigned to variables  $x_1, \dots, x_n$ . In KLAIM we would have:

Client

```
out(in(!y1, ..., !yn)@l.A⟨y1, ..., yn⟩, v1, ..., vn)@l
```

Server

```
in(!X, !x1, ..., !xn)@self.out(x1, ..., xn)@self.X
```

# Mobile Agents in K<sub>L</sub>AIM

Execution of process  $P$  at a remote location  $l$  can be implemented in either of the followings:

## Client

**eval**( $P$ )@ $l$

if a dynamic scoping discipline for resolving location names is adopted.

## Client

**newloc**( $u$ ).**out**( $P$ )@ $u$ .**in**(! $X$ )@ $u$ .**eval**( $X$ )@ $l$

when it is needed to guarantee the closure of  $P$  before moving it.

# Code On-Demand with K<sub>L</sub>AIM

A component of a networking application can dynamically download some code from a remote node and link it to perform the required task.

## Client

```
... .read(!X)@/.X
```

## Server

.... Offers in its data space the appropriate code that will be matched by X ...

Obviously, the query - !X - can be much more structured.



# A Simple Example: A Dynamic Newsgatherer

We would like to program a mobile agent that searches for specific instances of data distributed over a KLAIM net. We assume that each node contains:

- 1 a tuple of the form  $(item, v)$ , with the desired information, or
- 2 a tuple of the form  $(item, \ell_{next})$ , with details about the next node to search for the wanted information

If the agent finds the information it sends it back to a location specified by its owner; otherwise, it migrates to the next site.

$$P = \mathbf{newloc}(u).\mathbf{eval}(\mathit{gatherer}(item, u))@_{\ell_{item}}.((\mathbf{in}(!x)@u.P_1)|P_2)$$

$$\begin{aligned} \mathit{gatherer}(item, u) = & \\ & \mathbf{read}(item, !x)@self.\mathbf{out}(x)@u.\mathbf{nil} \\ & + \\ & \mathbf{read}(item, !u')@self.\mathbf{eval}(\mathit{gatherer}(item, u))@u'.\mathbf{nil} \end{aligned}$$

# Implementations

## Klava

A set of classes implementing

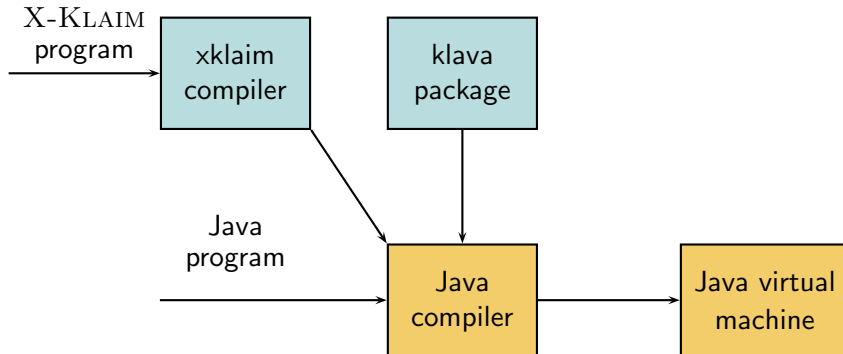
- `KLAIM` operations and concepts,
- Communications among processes and nodes
- Code mobility

## X-`KLAIM`

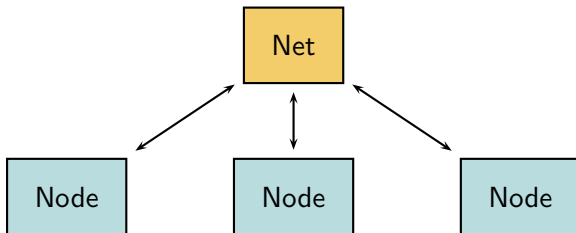
X-`KLAIM` is a full fledged programming language with `KLAIM` operations and a high-level syntax:

- Variable declarations, Conditionals, Assignments, ...
- Time-outs
- Hierarchical Nets
- Strong Mobility
- Object Orientation

# Programming Framework for KLAIM

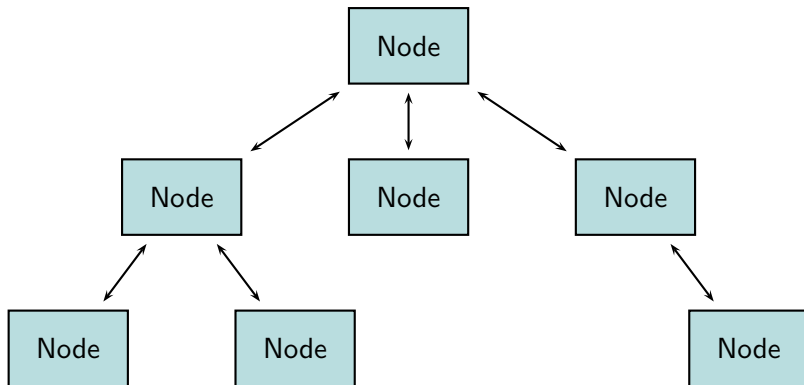


# A Flat Communication Model for KLAIM



- All nodes must be connected to a *Net server*
- All nodes are at the same level (flat)

# A Hierarchical Communication Model for KLAIM



# A Hierarchical Communication Model for KLAIM ctd.

- A node plays both the role of computational environment and *gateway*.
- Connections are managed explicitly.
- Allocations environment can be modified dynamically.
- Logical names are resolved taking into account hierarchies (like DNS).
- Two nodes can communicate if there is a connection path along the hierarchy tree.
- Connectivity actions can be performed only by privileged processes: *node coordinators*.

# A Software framework for Implementing Mobile Calculi

IMC is a set of Java classes and interfaces that aims at supporting implementations of languages for network aware programming. It

- has been implemented within the EU **MIKADO** project
- is independent from the programming style and can be merged with different computing paradigms.
- implements recurrent mechanisms and patterns and permits concentrating on features specific of the language of interest
- can be customized to fit language-specific requirements

## Main Components of IMC

- 1 Network Topology
- 2 Naming and Binding
- 3 Communication Protocols
- 4 Code Mobility

# Main Components of IMC

## 1. Network topology

Provides primitives for

- Physical and logical connections and disconnections
- Node creation and deletion
- Management of the (*flat* or *hierarchical*) topology of the network

## 2. Protocols

Provides primitives for

- Implementing specific protocol layers and states
- Building protocols starting from small components
- Re-using components that are abstract and independent from specific communication layers.



## 3. Mobility

Provides primitives for

- Rendering code mobility transparent to the programmer
- code collection, marshalling, dispatching
- dynamic loading of code received from a remote site
- offering abstract interfaces and implementations for Java byte-code mobility

## 4. Naming and binding

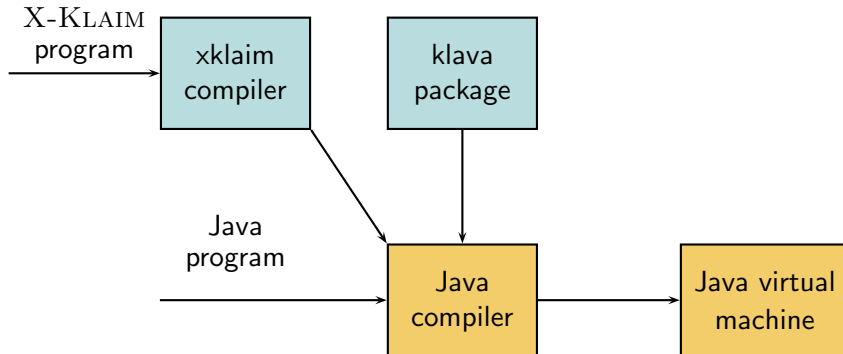
Provides primitives for

- Uniformly designating and interconnecting a set of objects
- name creation and deletion
- setting up policies for name resolution.

# IMC use and its perspectives

- IMC has been used at Firenze to implement JDPI and to re-engineer Klava and re-implement X-KLAIM run time support.
  - At University of Lisbon a research group is working on the implementation of MiKo (a calculus stemmed from the Mikado project) on the top of IMC
  - At France Telecom Research Labs in Grenoble have started investigating the implementation of security membranes on the top of an execution platform to enrich IMC.
- 
- We are investigating the possibility of enriching IMC with security policies based on the Java ones.
  - We are currently organizing the distribution of IMC as open source software under GPL licence. Please check <http://music.dsi.unifi.it>

# Programming Framework for KLAIM



# Variables

- **declare** for declaring a local variables
- basic types: **str**, **int**, **bool**  
**var** name : type
- constant types are automatically inferred:

**const** s := "foo" ; *# a string constant*

**const** b := **true** ; *# a boolean constant*

**const** i := 1971 ; *# an integer constant*

# Localities

- *Logical Localities* are names. Type **logloc**
- *Physical Localities* strings of the shape IP:port. Type **phyloc**
- **loc** generic locality type

**logloc** <: **loc**

**phyloc** <: **loc**

```
var l : loc;  
var output : logloc;  
var server : phyloc;  
output := "screen";  
server := "192.168.1.10:9999";  
l := output; # OK: a logical locality can be assigned to a locality  
l := server; # OK: a physical locality can be assigned to a locality
```

Constants logical localities:

```
locname screen, server;
```

# Locality Resolution

- Logical localities used as “destination” are always evaluated automatically
- Logical localities used as tuple fields:
  - Flat model: automatic
  - Hierarchical model: manual with operator \*

$l := *output; \#$  *retrieve the physical locality associated to output*  
**out**(\*output)@self;  $\#$  *insert the physical locality associated to output*

- **readp** and **inp**: non-blocking versions of **read** and **in**
- *Timeouts*:

```
if in(!x, !y)@l within 2000 then
  # ... success!
else
  # ... timeout occurred
endif
```

- combinations (*lazy evaluation*)

```
if readp(!i, !j)@l and (not in("foo", !k)@self within 3000) then
  out(i, j)@self # i and j are initialized
else
  out(k)@self # may not be initialized
endif
```

```
while readp(!i, !s)@self do  
  out(i + 1, s)@l  
enddo
```

- This may end up in an infinite loop, always modifying the same tuple.
- Repeatedly withdrawing such a tuple with **inp** does not solve the problem, since, in order not to be destructive on the original site, it would force to reinsert the withdrawn tuple, thus incurring in the same problem as above.



# Iterations: **forall**

**forall** guarantees that, given a template, each matching tuple is retrieved only once.

```
forall readp(!i, !s)@self do  
  out(i + 1, s)@l  
enddo
```

- Before: (10, "foo"), (10, "foo"), (20, "bar")
- After: (11, "foo"), (11, "foo"), (21, "bar")

The tuple space is not blocked when the execution of the **forall** is started, thus this operation is not atomic: the set of tuples matching the template can change before the command completes.

# Data Structures

A variable declared with type **ts** can be considered as a tuple space and can be accessed through standard tuple space operations

**declare**

**var** s : **str**;

**var** i : **int**;

**var** list : **ts**;

...

**forall inp**(!s, !i)@list **do**

**out**(i, s)@list

**enddo**

# Process Definition

- Process name
- formal parameters

```
rec ProcName[  
    param1 : paramtype, # call by value  
    ...  
    paramn : ref paramtype, # call by ref  
    ...  
]
```

- Local variables
- Process body

A process is accessible through its name.  
Similar to a procedure.

# Remote Evaluation

The operation **eval**( $P$ )@ $l$  starts the process  $P$  on the node at locality  $l$ ;  $P$  can be either a process name (and its arguments):

```
eval( MyProc("foo", 10) )@l
```

or the code (i.e., the actions) of the process to be executed:

```
eval( in(!i)@self; out(i)@l2 )@l
```

Processes can also be used as tuple fields:

```
out( P("foo", 10), in(!i)@self; out(i)@l2 )@l
```

These processes are not started automatically at *l*: they are simply inserted in its tuple space.

They must be retrieved (e.g., by another process executing at *l*) and explicitly evaluated:

```
in(!P1, !P2)@self;  
eval(P1)@self;  
eval(P2)@self
```

# Kinds of Mobility

- *weak mobility*: code coming from a different site can be dynamically linked;
- *strong mobility*: a thread can move its code and execution state to a different site and resume its execution on arrival;
- *full mobility*: in addition to strong mobility, the whole state of the running program is moved, and this includes all threads' stacks, namespaces (e.g., I/O descriptors, file-system names) and other resources, so that migration is completely transparent.

# Kinds of Mobility

- *weak mobility*: code coming from a different site can be dynamically linked;
- *strong mobility*: a thread can move its code and execution state to a different site and resume its execution on arrival;
- *full mobility*: in addition to strong mobility, the whole state of the running program is moved, and this includes all threads' stacks, namespaces (e.g., I/O descriptors, file-system names) and other resources, so that migration is completely transparent.

Full mobility can be considered orthogonal to mobile agents and requires a strong support from the operating system layer. Strong mobility is the best notion of mobility for mobile agents: automatic resumption of execution thread is one of the main features of mobile agents (it exalts their autonomy).

# Strong Mobility

X-KLAIM provides *strong mobility* by means of the action **go@l** that makes an agent migrate to *l* and resume its execution at *l* from the instruction following the migration action

```
in(!i, !j)@self;  
go@l;  
out(i, j)@self
```

The agent retrieves a tuple from the local tuple space, then it migrates to the locality *l* and inserts the retrieved tuple into the tuple space at locality *l*



- A process can execute only on a node (*execution engine*)
- A node is defined by specifying
  - a name
  - allocation environment
  - processes running on it
  - some options:
    - its port (default 9999)
    - its Java class
    - ...

The physical locality of a node consists of:

- the IP address of the computer on which the node is executed
- the specified port number

# Hello World (local)

```
# HelloWorld.xklaim  
nodes  
hello_world :: {}  
  begin  
    print "Hello World!"  
  end  
endnodes
```

# Hello World (local)

```
# HelloWorld.xklaim  
nodes  
hello_world :: {}  
  begin  
    print "Hello World!"  
  end  
endnodes
```

## Compilation & Execution

- 1 `xklaim HelloWorld.xklaim`
- 2 `javac HelloWorld.java`
- 3 `java HelloWorld`

# Hello World (local) - version 2

```
# HelloWorld2.xklaim  
rec HelloProc[]  
  begin  
    print "Hello World!"  
  end  
  
nodes  
hello_world2 :: {}  
  begin  
    eval(HelloProc())@self  
  end  
endnodes
```

# Hello World (local) - version 2

```
# HelloWorld2.xklaim  
rec HelloProc[]  
  begin  
    print "Hello World!"  
  end  
  
nodes  
hello_world2 :: {}  
  begin  
    eval(HelloProc())@self  
  end  
endnodes
```

## Code Generation

A different Java source is created for each process  
(HelloProc.java).

# Hello World (local) - version 3

```
rec HelloProc[] extern  
  
nodes  
hello_world3::{}  
  begin  
    eval(HelloProc())@self  
  end  
endnodes
```

The code of HelloProc can be reused in another program.

# Hello World (distributed)

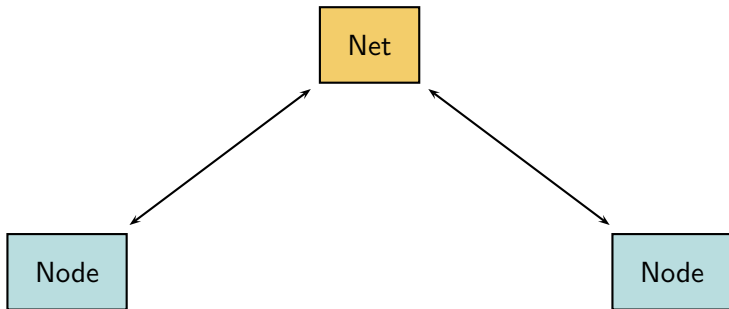
```
# HelloSender.xklaim
rec HelloSenderProc[ dest : loc ]
  begin
    out("Hello World!")@dest
  end

nodes
hello_sender::{receiver ~ localhost:11000}
  port 10000
  begin
    eval(HelloSenderProc(receiver))@self
  end
endnodes
```

```
# HelloReceiver.xklaim
nodes
hello_receiver::{}
  port 11000
  declare
    var msg : str
  begin
    in(!msg)@self;
    print "received: " + msg
  end
endnodes
```

```
xklaim -T 1 HelloSender.xklaim
To enable flat model.
```

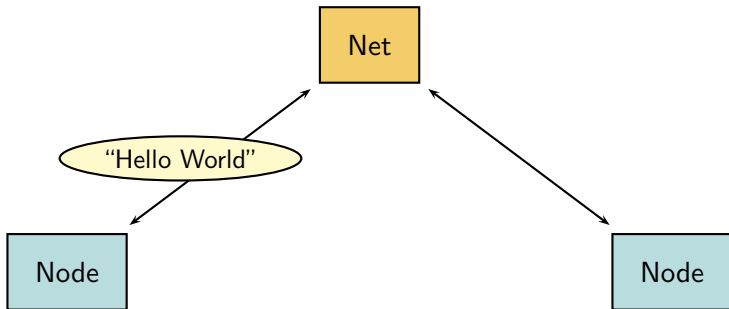
# Distributed Hello World



All communications pass through the Net server (*indirect communications*)

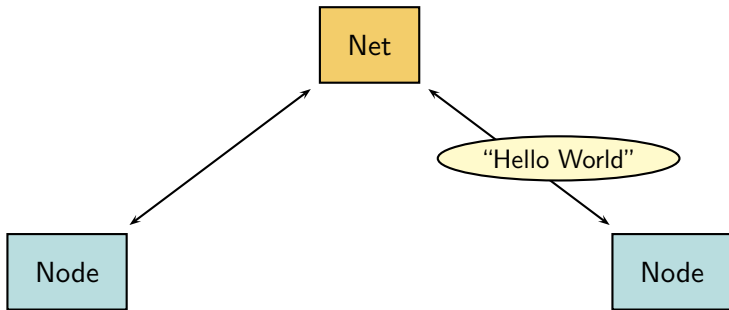


# Distributed Hello World



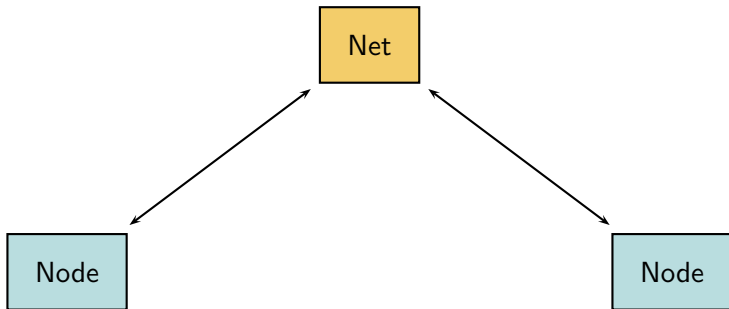
All communications pass through the Net server (*indirect communications*)

# Distributed Hello World



All communications pass through the Net server (*indirect communications*)

# Distributed Hello World



All communications pass through the Net server (*indirect communications*)

## Execution

- 1 `java Klava.Net 9999`
- 2 `java HelloReceiver localhost 9999`
- 3 `java HelloSender localhost 9999`

# News Gathering

```
rec NewsGatherer[ item : str, itemVal : str, finish : bool, retLoc : loc ]
  declare
    var itemVal : str ;
    var nextLoc : loc
  begin
    if not finish then
      if read( item, !itemVal )@self within 10000 then
        eval( NewsGatherer( item, itemVal, true, retLoc ) )@retLoc
      else
        if readp( item, !nextLoc )@self then
          eval( NewsGatherer( item, "", false, retLoc ) )@nextLoc
        else
          eval( NewsGatherer( item, "", true, retLoc ) )@retLoc
        endif
      endif
    else
      if itemVal != "" then print "found " + itemVal
      else print "search failed" endif
    endif
  end
```

# News Gathering (strong mobility)

```
rec NewsGatherer[ item : str, retLoc : loc ]
  declare var itemVal : str ; var nextLoc : loc ; var again : bool
  begin
    again := true;
    while again do
      if read( item, !itemVal )@self within 10000 then
        go@retLoc;
        print "found " + itemVal;
        again := false;
      else
        if readp( item, !nextLoc )@self then
          go@nextLoc
        else
          go@retLoc;
          print "search failed";
          again := false
        endif
      endif
    enddo
  end
```

Market place scenario:

- suppose that someone wants to buy a specific product at a market made of geographically distributed shops;
- to decide at which shop to buy, he activates a migrating agent which is programmed to find and return the name of the closest shop (i.e., the shop within the chosen area, determined by a maximal distance parameter) with the lowest price.

# News Gathering (strong mobility)

```
rec MarketPlaceAgent[ ProductMake : str, retLoc : loc, distance : int ]  
  declare  
    var shopList : TS ; var nextShop, CurrentShop, thisShop : loc ;  
    var CurrentPrice, newCost : int ; locname screen  
  begin  
    out( "cshop", distance )@self; # ask for a list of shops within a distance  
    in( "cshop", !shopList )@self;  
    out( "retrieved list: ", shopList )@screen;  
    CurrentPrice := 0 ;  
    CurrentShop := self ;  
    forall inp( ! nextShop )@shopList do # while there are shops to visit  
      thisShop := nextShop ;  
      go@nextShop ; # migrate to the next shop ;  
      out( "AgentClient: searching for ", ProductMake )@screen ;  
      if read( ProductMake, ! newCost )@self within 10000 then  
        if ( CurrentPrice = 0 OR newCost < CurrentPrice ) then  
          CurrentPrice := newCost; # update the best price  
          CurrentShop := thisShop  
        endif  
      endif  
    enddo ;  
    out( ProductMake, CurrentShop, CurrentPrice )@retLoc # OK, let's send the results  
  end
```

# Connectivity: Login

- **login**(*loc*) logs the node where the node coordinator is executing at the node at locality *loc*; it must synchronize with **accept**.
- **accept**(*l*) waits for a connection. It initializes the variable *l* to the physical locality of the node that is logging.
- **logout**(*loc*) logs the node out from the net managed by the node at locality *loc*.
- **disconnected**(*l*) notifies that a node has disconnected from the current node; the physical locality of such node is stored in the variable *l*. **disconnected** also catches connection failures.

## Blocking and Non-blocking

Notice that both **accept** and **disconnected** are blocking in that they block the running process until the event takes place. Instead, **logout** does not have to synchronize with **disconnected**.



# Connection example

```
rec nodecoord SimpleLogin[ server : loc ]
begin
  print "try to login to " +
    server + "...";
  if login( server ) then
    print "login successful";
    out("logged", true)@self
  else
    print "login failed!"
  endif
end
```

```
rec nodecoord SimpleLogout[ server : loc ]
begin
  in("logged", true)@self;
  print "logging off from " +
    server + "...";
  logout(server);
  print "logged off."
end
```

```
rec nodecoord SimpleAccept[]
declare
  var client : phyloc
begin
  print "waiting for clients...";
  accept(client);
  print "client " + client + " logged in"
end
```

```
rec nodecoord SimpleDisconnected[]
declare
  var client : phyloc
begin
  print "waiting for disconnections...";
  disconnected(client);
  print "client " + client +
    " disconnected."
end
```

- **subscribe**(*loc*, *logloc*) is similar to **login**, but it also permits specifying the logical locality with which a node wants to become part of the net coordinated by the node at locality *loc*; this request can fail also because another node has already subscribed with the same logical locality at the same server.
- **unsubscribe**(*loc*, *logloc*) performs the opposite operation.
- **register**(*pl*, *ll*), is the complementary action of **subscribe** that has to be performed on the server; if the subscription succeeds *pl* and *ll* will respectively contain the physical and the logical locality of the subscribed node. The allocation environment is updated automatically.
- **unregister**(*pl*, *ll*) records the unsubscriptions.

# Creating a new node: **newloc**

Apart from standard **newloc** the hierarchical model introduces two variants:

- **newloc**( $\ell$ , *NodeCoord*) also installs a node coordinator into the new node
- **newloc**( $\ell$ , *NodeCoord*, *port*, *JavaClass*) also specifies the port number (i.e., the physical locality) and the Java class implementing the node

# Direct Connections

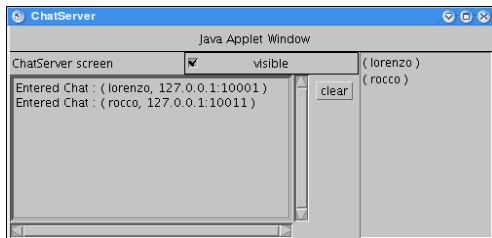
- **dirconnect**(*loc*) and **acceptconn**(*l*) that allow to create a unidirectional direct communication channel
- if a node  $n_1$  establishes a direct connection with the node  $n_2$  every time  $n_1$  sends a message to  $n_2$  it will do this directly, i.e., without passing through a possible common server

A direct connection is unidirectional: a bidirectional *peer to peer* communication has to be programmed explicitly so that upon accepting a direct connection from a node, also the other way direction is established.

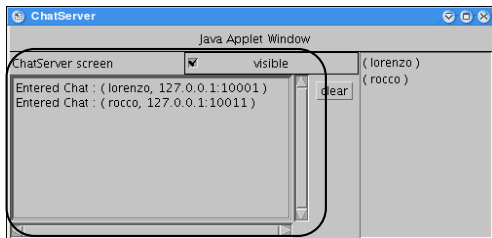
# A Chat System in X-KLAIM

- The system consists of a ChatServer and many ChatClients.
- The system is dynamic because new clients can enter the chat and existing clients may disconnect.
- The server represents the gateway through which the clients can communicate, and the clients logs in the chat server by specifying their “nickname”, represented here by a logical locality.
- A client that wants to enter the chat must subscribe at the chat server.
- The server must keep track of all the registered clients and, when a client sends a message, the server has to deliver the message to every connected client.

# ChatServer window

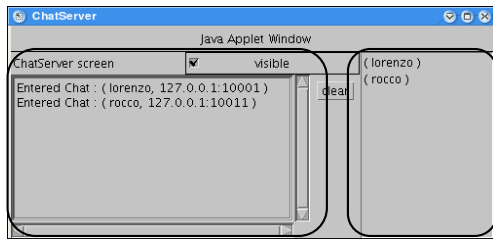


# ChatServer window



screen tuple space

# ChatServer window

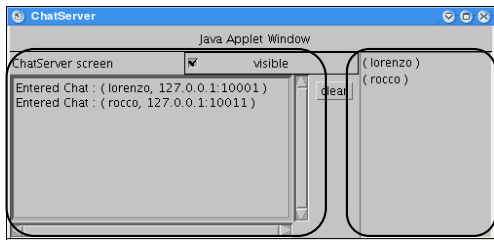


screen tuple space

list tuple space



# ChatServer window



screen tuple space

list tuple space

Input/Output in X-KLAIM is provided through tuple space abstraction.

# The server

```
rec nodecoord HandleLogin[ usersDB : ts ]  
  declare  
    var nickname : logloc ;  
    var client : phyloc ;  
    locname users, screen, server  
  begin  
    while ( true ) do  
      if register( client, nickname ) then  
        out( nickname, client )@usersDB ;  
        out( true )@client ;  
        SendUserList( client, usersDB ) ;  
        out( nickname )@users ;  
        out( "Entered Chat : " )@screen ;  
        out( nickname, client )@screen ;  
        BroadCast( "USER", "ENTER",  
          nickname, server, usersDB )  
      endif  
    enddo  
  end
```

# The server

```
rec nodecoord HandleLogin[ usersDB : ts ]
  declare
    var nickname : logloc ;
    var client : phyloc ;
    locname users, screen, server
  begin
    while ( true ) do
      if register( client, nickname ) then
        out( nickname, client )@usersDB ;
        out( true )@client ;
        SendUserList( client, usersDB ) ;
        out( nickname )@users ;
        out( "Entered Chat : " )@screen ;
        out( nickname, client )@screen ;
        BroadCast( "USER", "ENTER",
                    nickname, server, usersDB )
      endif
    enddo
  end
```



data base of clients

# The server

```
rec nodecoord HandleLogin[ usersDB : ts ]
  declare
    var nickname : logloc ;
    var client : phyloc ;
    locname users, screen, server
  begin
    while ( true ) do
      if register( client, nickname ) then
        out( nickname, client )@usersDB ;
        out( true )@client ;
        SendUserList( client, usersDB ) ;
        out( nickname )@users ;
        out( "Entered Chat : " )@screen ;
        out( nickname, client )@screen ;
        BroadCast( "USER", "ENTER",
                    nickname, server, usersDB )
      endif
    enddo
  end
```

data base of clients

graphical list of clients

# The server

```
rec nodecoord HandleLogin[ usersDB : ts ]
  declare
    var nickname : logloc ;
    var client : phyloc ;
    locname users, screen, server
  begin
    while ( true ) do
      if register( client, nickname ) then
        out( nickname, client )@usersDB ;
        out( true )@client ;
        SendUserList( client, usersDB ) ;
        out( nickname )@users ;
        out( "Entered Chat : " )@screen ;
        out( nickname, client )@screen ;
        BroadCast( "USER", "ENTER",
                    nickname, server, usersDB )
      endif
    enddo
  end
```

data base of clients

graphical list of clients

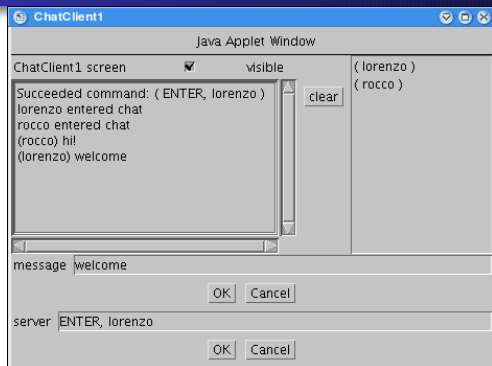
screen for messages

# Server: handling messages

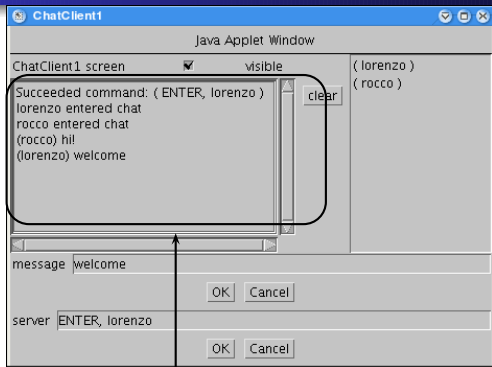
```
rec HandleMessage[ usersDB : ts ]
  declare
    var message : str ;
    var sender : logloc ;
    var from : phyloc
  begin
    while ( true ) do
      in( "MESSAGE", !message, !from )@self ;
      if readp( !sender, from )@usersDB then
        Broadcast( "MESSAGE", "ALL", message, sender, usersDB )
      endif # ignore errors
    enddo
  end

rec Broadcast[ communication_type : str, message_type : str,
              message : str, from : logloc, usersDB : ts ]
  declare
    var nickname : logloc ;
    var user : phyloc
  begin
    forall readp( !nickname, !user )@usersDB do
      out( communication_type, message_type, message, from )@user
    enddo
  end
```

# ChatClient window



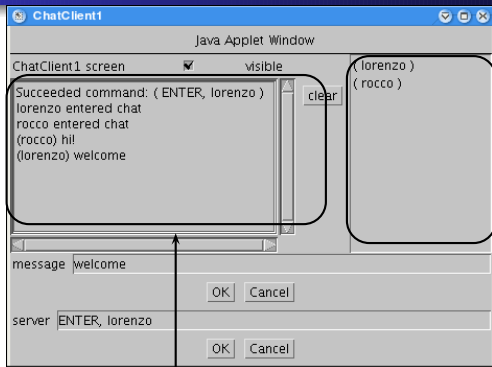
# ChatClient window



screen tuple space



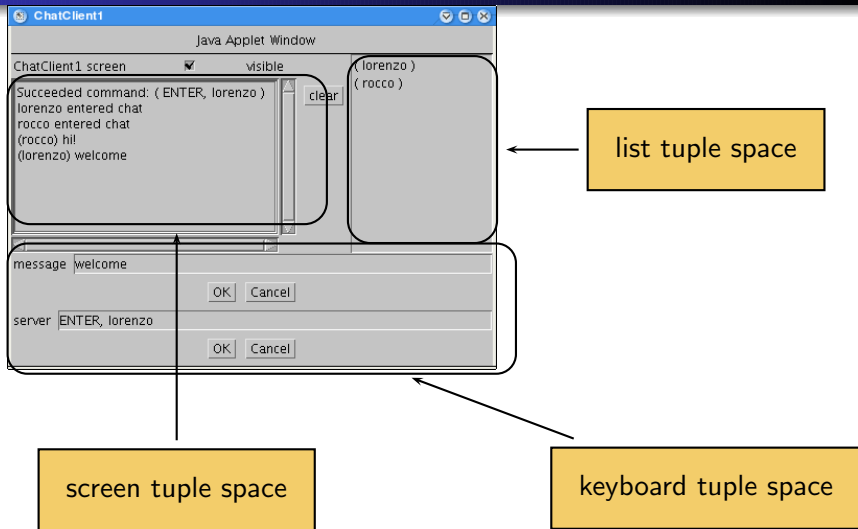
# ChatClient window



list tuple space

screen tuple space

# ChatClient window



# Client: handing incoming messages

```
rec HandleMessages[]
  declare
    locname screen ;
    const standard_message := "MESSAGE";
    var message, message_type : str ;
    var from : logloc
  begin
    while ( true ) do
      in( standard_message, !message_type, !message, !from )@self ;
      out( "(" )@screen ;
      out( (str)from )@screen ;
      out( ")" " )@screen ;
      out( message )@screen ;
      out( "\n" )@screen
    enddo
  end
```

# Client: handing outgoing messages

```
rec HandleMessageKeyboard[]  
  declare  
    var message : str ;  
    locname messageKeyb, server  
  begin  
    while ( true ) do  
      in( !message )@messageKeyb ;  
      out( "MESSAGE", message, *self )@server  
    enddo  
  end
```

## Main aims

- Evaluate the theoretical impact of tuple-based communication
- Test the expressive power of  $\text{KLAIM}$  constructs
- Test expressiveness relatively to asynchronous  $\pi$ -calculus

by

- Distilling from  $\text{KLAIM}$  a few, more and more, foundational formalisms
- Studying the possibility of encoding each of the languages in a simpler one

- Encodings and their evaluation
- A stack of four increasingly simpler and simpler languages:
  - 1 KLAIM
  - 2 micro-KLAIM
  - 3 core-KLAIM
  - 4 Local core-KLAIM
- Three encodings along the stack
- Encoding synchronous  $\pi$ -calculus in CKLAIM (and L-CKLAIM in asynchronous  $\pi$ -calculus)
- Assessment of the encodings

# Criteria for an encoding to be reasonable

Any 'reasonable' [Palamidessi:MSCS2003] encoding  $enc(\cdot)$  should preserve the semantics, guarantee the same degree of parallelism, and keep a close correspondence between the used names, i.e., it should:

- 1 **preserve basic observations**: it must not change the visible behaviours of the encoded terms;
- 2 **preserve termination**: it must turn each terminating term in a terminating one;
- 3 **be homomorphic w.r.t. the parallel operator**:  
$$enc(N \parallel M) = enc(N) \parallel enc(M);$$
- 4 **preserve renaming**: for every permutation of names  $\sigma$  in the source language there has to exist a permutation of names  $\theta$  in the target language such that  $enc(P\sigma) = (enc(P))\theta$

# Evaluation criteria for Behavioural Correspondence

## Operational Correspondence:

- if  $X_1 \rightarrow_{\mathcal{X}} X_2$  then  $enc(X_1) \Rightarrow_{\mathcal{Y}} enc(X_2)$
- if  $enc(X) \rightarrow_{\mathcal{Y}} Y$  then  $X \Rightarrow_{\mathcal{X}} X'$  with  $Y \Rightarrow_{\mathcal{Y}} enc(X')$

## Full Abstraction w.r.t. $EQ$ :

- $X_1 EQ_{\mathcal{X}} X_2$  if and only if  $enc(X_1) EQ_{\mathcal{Y}} enc(X_2)$

## Semantical Equivalence w.r.t. $EQ$ :

- $X EQ_{\mathcal{Z}} enc(X)$ , for a language  $\mathcal{Z}$  containing both  $\mathcal{X}$  and  $\mathcal{Y}$

## NOTICE:

- $EQ$  is not a precise equivalence but a *family* of equivalences and  $EQ_{\mathcal{L}}$  is the equivalence  $EQ$  in the language  $\mathcal{L}$
- A stronger equivalence guarantees a better encoding
- Sem. equiv. w.r.t.  $EQ$  implies full abstraction w.r.t.  $EQ$



# A language-independent notion of equivalence

Observation predicate, or **barb**:

$N \Downarrow I$  iff  $N \equiv (\nu \tilde{I})(N' \parallel I ::_{\rho} \langle t \rangle)$  for some  $\tilde{I}$ ,  $N'$ ,  $\rho$  and  $t$  s.t.  $I \notin \tilde{I}$

A binary relation  $\mathcal{R}$  between nets is

- **barb preserving**, if  $N \mathcal{R} M$  and  $N \Downarrow I$  imply  $M \Downarrow I$
  - **reduction closed**, if  $N \mathcal{R} M$  and  $N \mapsto N'$  imply  $M \mapsto^* M'$  and  $N' \mathcal{R} M'$
  - **context closed**, if  $N \mathcal{R} M$  implies  $\mathcal{C}[N] \mathcal{R} \mathcal{C}[M]$  for every context  $\mathcal{C}[\cdot]$
- 
- **Barbed bisimilarity**,  $\dot{\cong}$ , is the largest symmetric, barb preserving and reduction closed relation between nets
  - **Barbed congruence**,  $\cong$ , is the largest symmetric, barb preserving, reduction and context closed relation between nets

# KLAIM: Syntax

**Nets:**  $N ::= \mathbf{0} \mid I ::_{\rho} C \mid N_1 \parallel N_2 \mid (\nu I)N$

**Components:**  $C ::= \langle t \rangle \mid P \mid C_1 \mid C_2$

**Processes:**

$$P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid X \mid \mathbf{rec}X.P$$

**Actions:**

$$a ::= \mathbf{in}(T)@u \mid \mathbf{read}(T)@u \mid \mathbf{out}(t)@u \mid \mathbf{eval}(P)@u \mid \mathbf{newloc}(l)$$

**Tuples:**  $t ::= u \mid P \mid t_1, t_2$

**Templates:**  $T ::= u \mid !x \mid !X \mid T_1, T_2$

# Structural Congruence

## Monoid Laws for $\parallel$

$$N \parallel \mathbf{0} \equiv N$$

$$N_1 \parallel N_2 \equiv N_2 \parallel N_1$$

$$(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$$

## Congruence Laws

$$(\text{ALPHA}) \quad N \equiv N' \quad \text{if } N =_{\alpha} N'$$

$$(\text{RCOM}) \quad (\nu l_1)(\nu l_2)N \equiv (\nu l_2)(\nu l_1)N$$

$$(\text{EXT}) \quad N_1 \parallel (\nu l)N_2 \equiv (\nu l)(N_1 \parallel N_2) \quad \text{if } l \notin \text{fn}(N_1)$$

$$(\text{ABS}) \quad l ::_{\rho} C \equiv l ::_{\rho} (C \mid \mathbf{nil})$$

$$(\text{CLONE}) \quad l ::_{\rho} C_1 \mid C_2 \equiv l ::_{\rho} C_1 \parallel l ::_{\rho} C_2$$

$$(\text{REC}) \quad l ::_{\rho} \mathbf{rec}X.P \equiv l ::_{\rho} P[\mathbf{rec}X.P/X]$$

# Process Reduction Rules

$$\text{(RED-OUT)} \quad \frac{\rho(u) = l' \quad \mathcal{E}[\![t]\!]_{\rho} = t'}{l ::_{\rho} \mathbf{out}(t)@u.P \parallel l' ::_{\rho'} \mathbf{nil} \mapsto l ::_{\rho} P \parallel l' ::_{\rho'} \langle t' \rangle}$$

$$\text{(RED-EVAL)} \quad \frac{\rho(u) = l'}{l ::_{\rho} \mathbf{eval}(P_2)@u.P_1 \parallel l' ::_{\rho'} \mathbf{nil} \mapsto l ::_{\rho} P_1 \parallel l' ::_{\rho'} P_2}$$

$$\text{(RED-IN)} \quad \frac{\rho(u) = l' \quad \text{match}(\mathcal{E}[\![T]\!]_{\rho}, t) = \sigma}{l ::_{\rho} \mathbf{in}(T)@u.P \parallel l' ::_{\rho'} \langle t \rangle \mapsto l ::_{\rho} P\sigma \parallel l' ::_{\rho'} \mathbf{nil}}$$

$$\text{(RED-READ)} \quad \frac{\rho(u) = l' \quad \text{match}(\mathcal{E}[\![T]\!]_{\rho}, t) = \sigma}{l ::_{\rho} \mathbf{read}(T)@u.P \parallel l' ::_{\rho'} \langle t \rangle \mapsto l ::_{\rho} P\sigma \parallel l' ::_{\rho'} \langle t \rangle}$$

$$\text{(RED-NEW)} \quad l ::_{\rho} \mathbf{newloc}(l').P \mapsto (\nu l')(l ::_{\rho} P \parallel l' ::_{\rho[l'/\mathbf{self}]} \mathbf{nil})$$

# Nets Reduction Rules

$$\text{(RED-PAR)} \quad \frac{N_1 \mapsto N'_1}{N_1 \parallel N_2 \mapsto N'_1 \parallel N_2}$$

$$\text{(RED-RES)} \quad \frac{N \mapsto N'}{(\nu l)N \mapsto (\nu l)N'}$$

$$\text{(RED-STRUCT)} \quad \frac{N \equiv M \mapsto M' \equiv N'}{N \mapsto N'}$$

# Matching Function

$$\text{match}(l, l) = \epsilon$$

$$\text{match}(!x, l) = [l/x]$$

$$\text{match}(!X, P) = [P/X]$$

$$\frac{\text{match}(T_1, t_1) = \sigma_1 \quad \text{match}(T_2, t_2) = \sigma_2}{\text{match}(T_1, T_2, t_1, t_2) = \sigma_1 \circ \sigma_2}$$

We strip off KLAIM:

- 1 higher-order data (i.e. tuples containing process code)
- 2 allocation environments (i.e. the distinction between logical and physical localities)

$\mu$ KLAIM is the largest sub-calculus of KLAIM such that:

- tuples do not contain processes,
- templates do not contain process variables,
- allocation environments are empty
- processes do not have free locality variables

$$N ::= \mathbf{0} \mid I :: C \mid N_1 \parallel N_2 \mid (\nu I)N$$
$$C ::= \langle t \rangle \mid P \mid C_1 \mid C_2$$
$$P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid X \mid \mathbf{rec}X.P$$
$$a ::= \mathbf{in}(T)@u \mid \mathbf{read}(T)@u \mid \mathbf{out}(t)@u \mid \mathbf{eval}(P)@u \mid \mathbf{newloc}(l)$$
$$t ::= u \mid t_1, t_2$$
$$T ::= u \mid !x \mid T_1, T_2$$

$\mathcal{N}$  set of *names* ranged over by  $l, l', \dots, u, \dots, x, y, \dots, X, Y, \dots$



# Encoding Higher Order Messages

In  $\pi$ -calculus, higher-order data are handled by means of *triggers* [Sangiorgi 1996].

$$\bar{a}\langle P \rangle \mid a(X).X$$

is translated to

$$(\nu c)(\bar{a}\langle c \rangle \mid * c().P') \mid a(x).\bar{x}\langle \rangle$$

where  $P'$  is the translation of  $P$  and  $*$  denotes replication

The address of  $P$  is sent over to be used by the interested processes to activate as many copies of  $P$  as needed

# Process Mobility in $\mu$ KLAIM

A fresh node is used to deposit  $P$  (the process to migrate). The address of the new node is distributed to be used by the interested processes to activate as many copies of  $P$  as needed

. Thus

$$l_1 :: \mathbf{out}(P)@l \parallel l_2 :: \mathbf{in}(!X)@l.X$$

is translated to

$$\begin{aligned} l_1 &:: \mathbf{newloc}(l').\mathbf{eval}(\mathbf{rec}X.\mathbf{in}(!y)@l'.\mathbf{eval}(P)@y.X)@l'.\mathbf{out}(l')@l \\ \parallel l_2 &:: \mathbf{in}(!x)@l.\mathbf{out}(l_2)@x \end{aligned}$$

that evolves to (a net “equivalent” to)

$$(\nu l')(l_1 :: \mathbf{nil} \parallel l_2 :: P)$$

# Modelling the allocation environment in $\mu\text{KLAIM}$

Binding of free locality variables to names can be modelled by:

- storing the allocation environment of each node in the tuple space of a fresh node `env`
- prefixing each localized action with the "capture" of the free variables occurring in its argument and in its target by exploiting `env`

$l ::_{\rho} \mathbf{out}(u_1, l, u_2)@l'.P$     **where**     $\rho$  maps  $u_1$  to  $l_1$  and  $u_2$  to  $l_2$ ,

is translated to

$\mathbf{env} :: \langle l, u_1, l_1 \rangle \mid \langle l, u_2, l_2 \rangle$

$\parallel l :: \mathbf{read}(l, u_1, !y_1)@env.\mathbf{read}(l, u_2, !y_2)@env.\mathbf{out}(y_1, l, y_2)@l'.P'$

where  $P'$  is the translation of  $P$ , and  $y_1$  and  $y_2$  are fresh names

We strip off  $\mu$ KLAIM:

- actions **read** for accessing data without removing them from tuple spaces

and

- *polyadic* tuples, i.e. all tuples in CKLAIM have just one field

# cKLAIM: Syntax

$$N ::= \mathbf{0} \mid I :: C \mid N_1 \parallel N_2 \mid (\nu I)N$$

$$C ::= \langle t \rangle \mid P \mid C_1 \mid C_2$$

$$P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid X \mid \mathbf{rec}X.P$$

$$a ::= \mathbf{in}(T)@u \mid \mathbf{out}(t)@u \mid \mathbf{eval}(P)@u \mid \mathbf{newloc}(l)$$

$$t ::= u$$

$$T ::= u \mid !x$$

cKLAIM is a sub-calculus of  $\mu$ KLAIM and inherits its operational semantics

# Encoding $\mu$ KLAIM in cKLAIM (1)

Action **read** can be naturally encoded by performing action **in** followed by action **out** on the tuple that has been accessed

For example,

**read**(!x, l', !y)@l

is translated to

**in**(!x, l', !y)@l.**out**(x, l', y)@l

Due to asynchrony, lack of atomicity does not lead to problems.  
This issue is somehow related to the equation

$$a(x).\bar{a}\langle x \rangle \approx \mathbf{0}$$

established for the  $\pi_a$ -calculus

(see [Amadio, Castellani and Sangiorgi 1998])

# Encoding Polyadic into Monadic calculi (1)

For *Synchronous  $\pi$ -calculus*, Milner's encoding transforms:

$$\bar{a}\langle b, c \rangle \quad | \quad a(x, y)$$

into

$$(\nu n)(\bar{a}\langle n \rangle . \bar{n}\langle b \rangle . \bar{n}\langle c \rangle \quad | \quad a(z) . z(x) . z(y))$$

A fresh name ( $n$ ) is exchanged by exploiting a common channel ( $a$ ). Name  $n$  is then used to pass the sequence of values.

# Encoding Polyadic into Monadic calculi (2)

For *Asynchronous  $\pi$ -calculus* a possible encoding, adapted from [Honda-Tokoro 1991], would transform:

$$\bar{a}\langle b, c \rangle \mid a(x, y)$$

into

$$\begin{aligned} &(\nu n)(\bar{a}\langle n \rangle \mid n(n_1).(\bar{n}_1\langle b \rangle \mid n(n_2).\bar{n}_2\langle c \rangle)) \\ &\mid (\nu n_1, n_2)a(z).(\bar{z}\langle n_1 \rangle \mid n_1(x).(\bar{z}\langle n_2 \rangle \mid n_2(y))) \end{aligned}$$

The schema is similar to the one for the synchronous calculus. However, since output sequentialization is not possible, different channels are needed to send the different values in the sequence.









# Encoding $\mu$ KLAIM in cKLAIM

In our setting, pattern-matching of values performed while retrieving tuples complicates the encoding. The key ideas are the following:

- A polyadic tuple is implemented by a process that sequentially produces the fields of the tuple
- The receiving process accesses these (monadic) fields in an exclusive and ordered way
- If the  $i$ -th tuple field matches against the  $i$ -th template field, the retrieving procedure goes on; otherwise, it is stopped and the involved processes are rolled back

# Encoding $\mu$ KLAIM in cKLAIM (2)

We translate the interaction between the consuming action  $\text{in}(T)@l$  and a tuple  $t$  by exploiting the following protocol:

TUPLE CONSUMER		TUPLE HANDLER
Acquire the <b>lock</b> over a tuple		
Ask for tuple's <b>length</b>	 	Provide $t$ 's length $k$
If $k =  T $ proceed, otherwise release the lock and roll back the tuple handler		
Ask for tuple's <b>first field</b>	 	Provide $t$ 's first field $f_1$
If $T$ 's first field matches $f_1$ proceed, otherwise release the lock and roll back the tuple handler		
...	...	...
Require tuple's <b>last field</b>	 	Provide $t$ 's last field $f_k$
If $T$ 's last field matches $f_k$ FINISH, otherwise release the lock and roll back the tuple handler		

To obtain a "local" variant of CKLAIM, we strip away remote inputs and remote outputs:

- communication is only local
- process migration is exploited to use remote resources

New syntax of actions:

$$a ::= \text{in}(T) \mid \text{out}(t) \mid \text{eval}(P)@u \mid \text{newloc}(l)$$

The rest of syntax (and semantics) is unchanged

# Encoding L-CKLAIM in CKLAIM

- action **out**( $l$ )@ $l'$ . $P$  is implemented as

$$\mathbf{eval}(\mathbf{out}(l))@l'.P'$$

where  $P'$  is the encoding of  $P$

- action **in**( $T$ )@ $l'$ . $P$  at site  $l$  is implemented as

$$\mathbf{eval}(\mathbf{in}(T).\mathbf{eval}(P')@l)@l'$$

where  $P'$  is the encoding of  $P$

# Encoding asynch. $\pi$ -calculus into CKLAIM

Syntax:

$$p ::= \mathbf{0} \mid \bar{a}\langle b \rangle \mid a(b).p \mid p_1 \mid p_2 \mid (\nu a)p \mid [a = b]p \mid !p$$

Encoding  $\pi_a$ -calculus in CKLAIM:

- each channel  $a$  is translated in the TS of a node named  $a$
- the basic constructs are translated as follows:

$$\llbracket (\nu a)p \rrbracket = \mathbf{newloc}(a).\llbracket p \rrbracket$$

$$\llbracket \bar{a}\langle b \rangle \rrbracket = \mathbf{out}(b)@a.\mathbf{nil}$$

$$\llbracket a(b).p \rrbracket = \mathbf{in}(!b)@a.\llbracket p \rrbracket$$

$$\llbracket [a = b]p \rrbracket = \mathbf{newloc}(l).\mathbf{out}(a)@l.\mathbf{in}(b)@l.\llbracket p \rrbracket$$

# Encoding L-CKLAIM into asynch. $\pi$ -calculus

## Encoding Nets:

$$([0]) \triangleq !\overline{ex}()$$

$$((\nu l)N) \triangleq (\nu l)([N] \mid !\overline{ex} l)$$

$$([N_1 \parallel N_2]) \triangleq ([N_1]) \mid ([N_2])$$

$$([I :: C]) \triangleq ([C])_I \mid !\overline{ex} I$$

## Encoding Processes:

$$([nil])_u \triangleq 0$$

$$([X])_u \triangleq X$$

$$([\langle l \rangle])_u \triangleq \bar{u} l$$

$$([\text{rec } X.P])_u \triangleq \text{rec } X.([P])_u$$

$$([C_1 \mid C_2])_u \triangleq ([C_1])_u \mid ([C_2])_u \quad ([\text{newloc}(l).P])_u \triangleq (\nu l)([P]_u \mid !\overline{ex} l)$$

$$([\text{out}(u').P])_u \triangleq \bar{u} u' \mid ([P])_u \quad ([\text{in}(!x).P])_u \triangleq u(x).([P])_u$$

$$([\text{in}(u').P])_u \triangleq \text{rec } X.u(x).(\nu c)(\bar{c} \mid [x = u']c.([P])_u \mid c.(\bar{u} x \mid X))$$

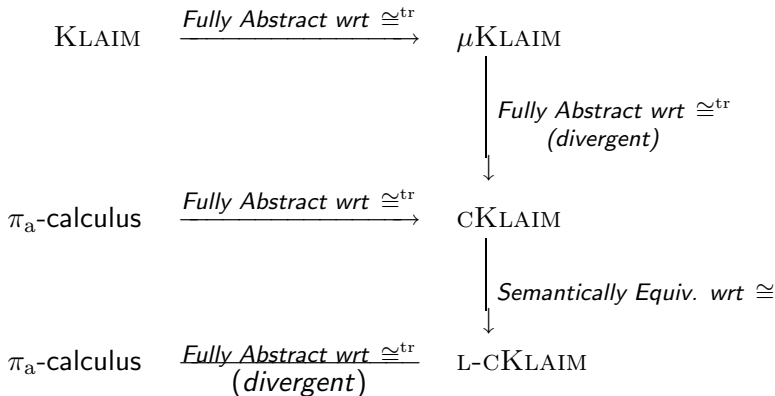
$$([\text{eval}(Q)@u'.P])_u \triangleq \text{rec } X.\text{ex}(x).(\nu c)(\bar{c} \mid [x = u']c.([P])_u \mid ([Q])_{u'} \mid c.X)$$

$x, X$  fresh in the last two cases

# Encoding L-CKLAIM into asynch. $\pi$ -calculus

- Locality names are rendered as channels
- To distinguish between addresses of network nodes and usual names, the encoding uses a reserved channel  $ex$  to record addresses
- The encoding of processes of the form  $([P])_u$  is parameterized with the site  $u$  where  $P$  is located
- process  $\mathbf{in}(l').P$  running at  $l$  is rendered as a process that first retrieves a datum at  $l$  and then checks if it is  $l'$ ; if the check succeeds, the process proceeds, otherwise it places back the retrieved datum and looks for another one

# Properties of the Encodings



- $\cong^{\text{tr}}$  is  $\cong$  closed for all contexts resulting from the translation (via the encoding) of source contexts
- Translated contexts represent opponents conforming to the protocol, i.e. to the precise exchange of messages.



- The encoding of  $\text{CKLAIM}$  in  $\text{L-CKLAIM}$  is the best we can imagine: it does not introduce divergence and translates nets into  $\text{barb.congr. ones}$ .

The two calculi have exactly the same expressive power

- The encodings of  $\text{KLAIM}$  in  $\mu\text{KLAIM}$  and of  $\pi_a$ -calculus in  $\text{CKLAIM}$  are satisfactory (and simple): they enjoy the four properties pointed out in [Palamidessi 2003].

Source and target calculi have similar expressive power

- The encodings of  $\mu\text{KLAIM}$  in  $\text{CKLAIM}$  and of  $\text{L-CKLAIM}$  in  $\pi_a$ -calculus are less satisfactory: they may introduce divergence. The encoding of polyadic communication in monadic communication is neither simple nor efficient.

$\mu\text{KLAIM}$  is more expressive than  $\text{CKLAIM}$

A key design issue breaks full abstraction:

- In  $\pi_a$ -calculus, knowing a name implies that communication actions can be performed along a channel with that name and these actions succeed whenever a parallel component performs a complementary action
- In cKLAIM it is *not* necessarily the case that each free name is associated to a locality

## Claim:

No 'reasonable' encoding of  $\dagger$ -cKLAIM in  $\pi_a$ -calculus can be given: checking existence of nodes before firing an output is a too low-level feature that cannot be implemented in such an abstract setting as the  $\pi_a$ -calculus and  $\pi$ -calculus.

# Conclusions

## We have presented:

- a family of process languages based on  $\text{KLAIM}$
- a number of encodings to study their expressive power
- a comparison of tuple based calculi with asynchronous  $\pi$ -calculus

## Our results might be used to:

- investigate the expressive power of pattern-matching
- propose 'reasonable' encodings of pattern-matching based communication in channel-based one
- settle issues connected to remote/local operations in global computing
- understand the relative convenience of dynamic and static naming disciplines

The KLAIM project is a collective effort:

- Three Ph.D. Thesis
  - 1 Lorenzo Bettini: X-KLAIM and KLAVA
  - 2 Michele Loreti: Logics for KLAIM
  - 3 Daniele Gorla: Types and Expressivity in Klaim
- Many papers with them, but also with Gianluigi Ferrari, Jost Peter Katoen, Diego Latella, Mieke Massink, Eugenio Moggi, Ugo Montanari, Rosario Pugliese, Emilio Tuosto, ....

MANY THANKS TO ALL OF THEM

PLEASE VISIT:

<http://music.dsi.unifi.it>

THANK YOU ALL FOR YOUR ATTENTION