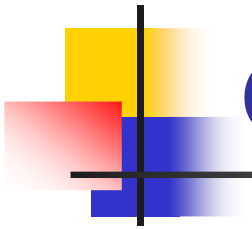
Klaim: *A Kernel Language for Agents Interaction and Mobility*

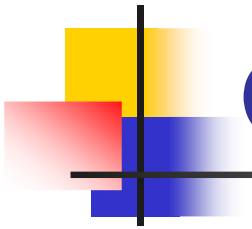
Rocco De Nicola
Dip. Sistemi e Informatica
Università di Firenze

denicola@dsi.unifi.it



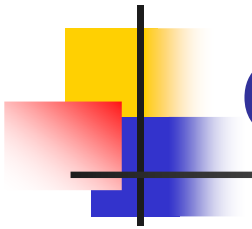
General Outline

- Motivations
- Calculi for mobility
- The Klaim Model
- Syntax and semantics
- Open Klaim
- X-Klaim
- Klava
- A logic for Klaim
- Systems Specifications
- Partial specification for Open Nets
- Secutity
- KryptoKlava
- Toward a Klaim calculus
- Core Klaim
- μ-Klaim
- Types for access control
- Types for μ-Klaim
- On going & future Work
- References



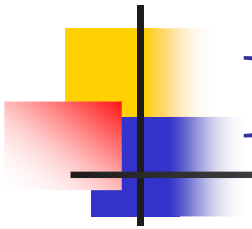
Outline for this lecture

- Motivations
- Calculi for mobility
- Linda
- The Klaim Model
- Syntax and semantics
- Mobility issues
- X-Klaim
- Example specifications



Global Systems

- Are *Distributed Systems* with distinguishing features:
  - Wide area distribution
  - Variable interconnection structures
  - (Physical and Logical) Mobility
  - Latency and bandwidth issues
  - Failures



Programming Global Systems

Explicit Primitives for

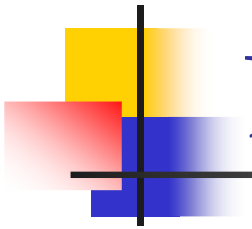
- Distribution

  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



Explicit Mechanisms for

- Distribution

  need to program distribution over localities

- Mobility

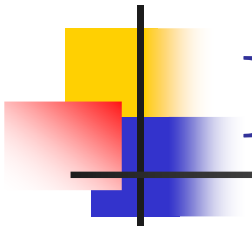
  need mobility of computations among localities

- Concurrency

  deal with parallel and non-deterministic comput.

- Access Rights

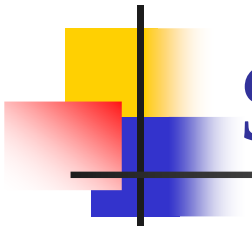
  need to maintain privacy and integrity of data



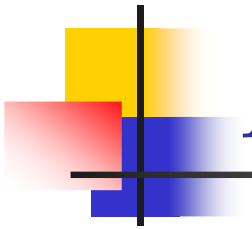
Formal Semantics

- We need verification techniques.
- We aim at developing a simple programming language for network aware and migrating applications with a tractable semantic theory that permits program verification.

Calculus of Communicating Systems

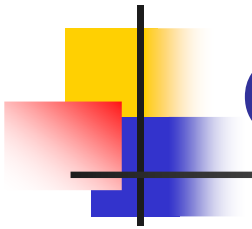


- CCS provides a small set of operators that may be used to construct system descriptions compositionally
- Basic blocks of system definitions are actions and the null process
- Actions represent atomic and uninterruptible execution steps



Actions in CCS

- Actions represent:
  - either signal inputs on *ports* (α)
  - or signal outputs on *ports* ($\underline{\alpha}$)
  - or internal computational steps (τ)



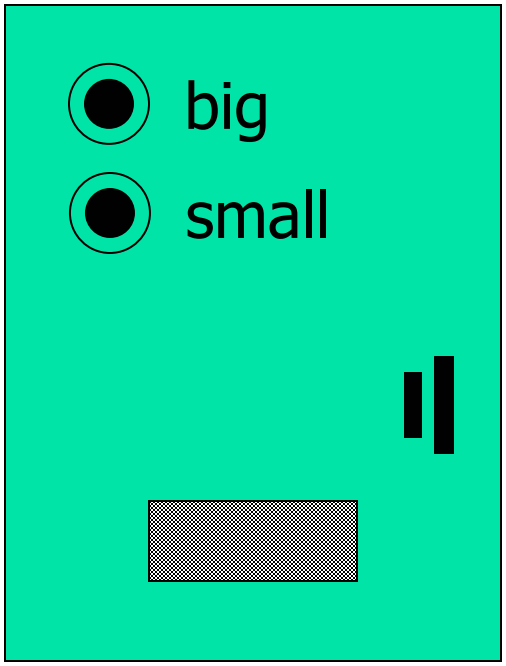
CCS Operators

- nil (terminated process)
- $a.P$ (action prefixing)
- P_1+P_2 (non deterministic choice)
- $P_1 \mid P_2$ (parallel composition)
- $P \setminus L$ (restriction)
- $P[f]$ (relabeling)
- $V=P$ (process definition)

A Simple Example

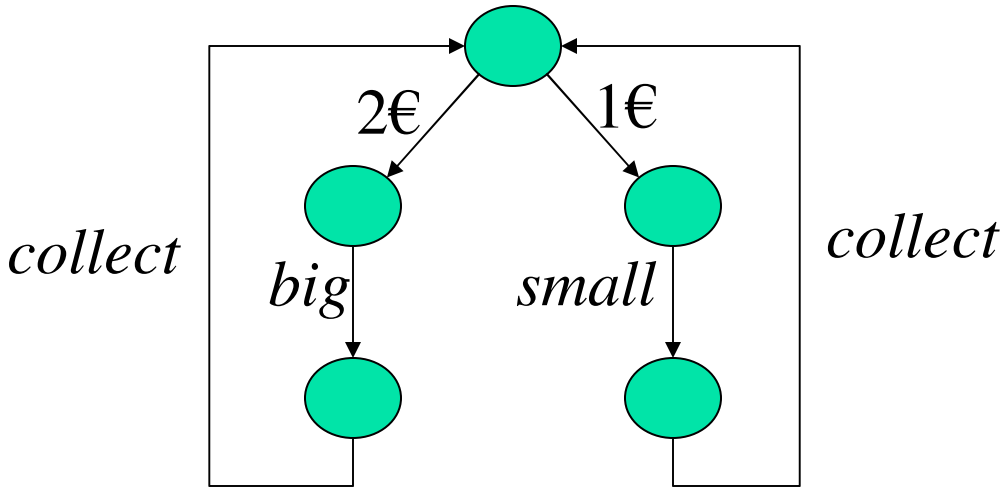
- A vending machine
  - A big chocolate costs 2€
  - A small chocolate costs 2€

$$V = 2€.big.collect.V + 1€.small.collect.V$$



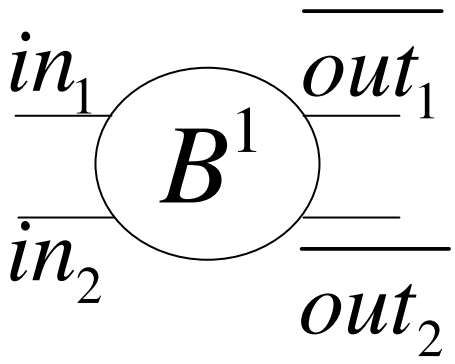
Operatinal Semantics

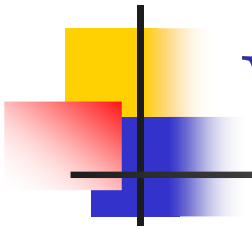
$$V = 2€.big.collect.V + 1€.small.collect.V$$



Buffer (n=1)

$$B^1 = in_1.\overline{out_1}.B^1 + in_2.\overline{out_2}.B^1$$





Value passing

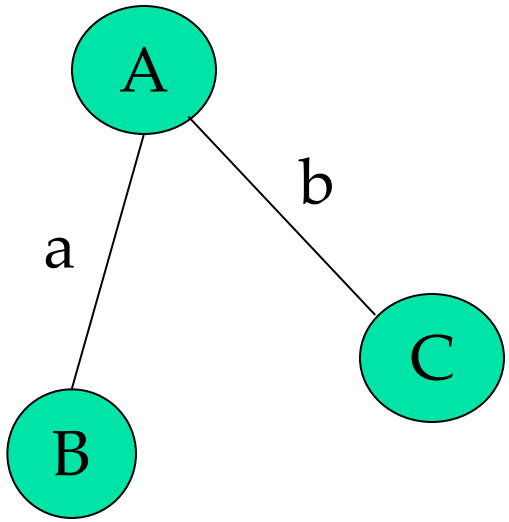
- Explicit values instead of signals:
  - *in*(x).P
  - *out*(v).P

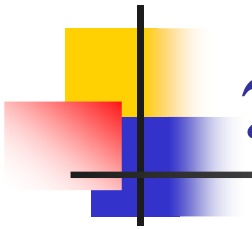
- Example:

$$B^1 = in_1(x).\overline{out_1}(x).B^1$$

Static Port

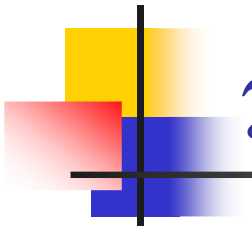
- There is no mobility
- B and C cannot directly interact (no common channel)





π-calculus

- There is a set of names X ($x,y,z \in X$)
- Action prefixes of π-calculus are:
  - $x(y)$ (receive y along x)
  - $\overline{x}\langle y \rangle$ (send y along x)
  - τ (unobservable action)



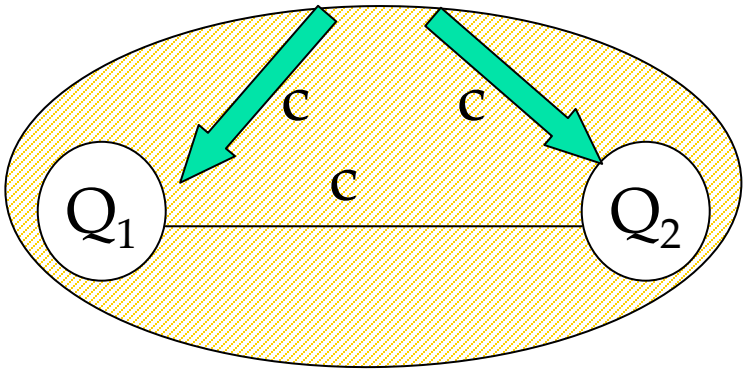
π-calculus

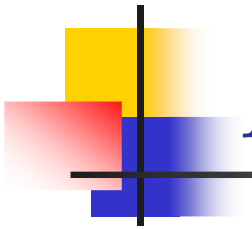
- *nil* (terminated process)
- (νa)P (new name)
- π.P (action prefixing)
- $P_1 | P_2$ (parallel composition)
- $P_1 + P_2$ (non deterministic choice)
- !P (replication)

Mobility in π-calculus

$$P_1 = a(x).Q_1 \qquad P_2 = b(x).Q_2$$

$$Q = (\nu c)a\langle c\rangle.b\langle c\rangle.nil$$





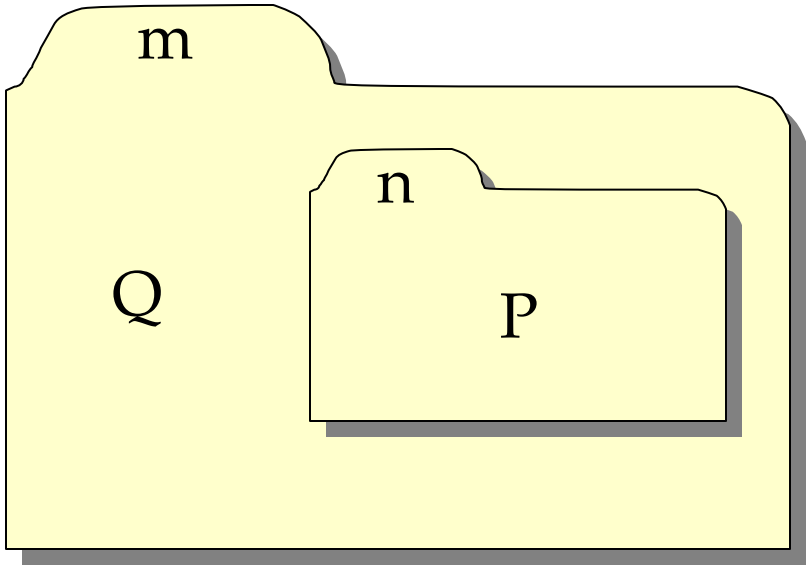
Ambients

- An ambient is a *bounded* place where computation happens;
- Ambients can be nested within other ambients, forming a tree structure;
- Each ambient:
  - has a collection of running processes;
  - moves as a whole with all its subcomponents;
  - has a name.

Ambients mobility

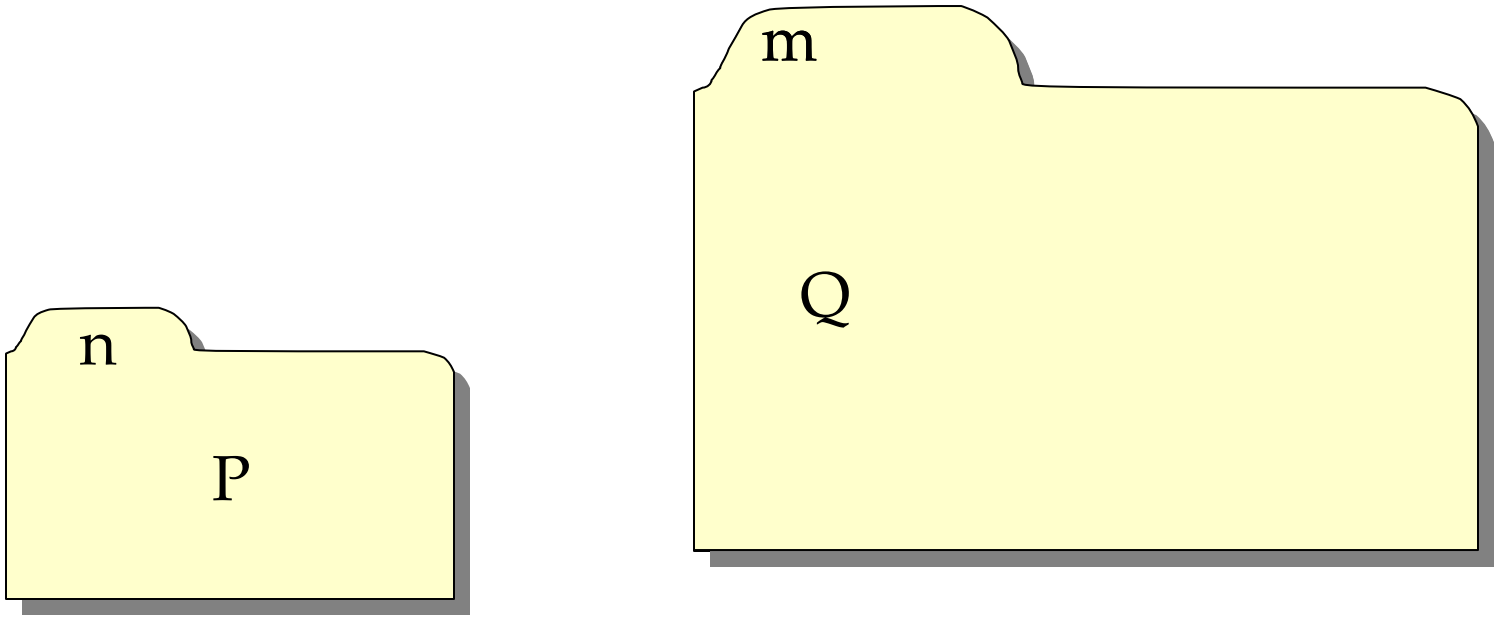
- The enter reduction:

$$n[in(m).\ P] \mid\mid m[Q]$$



Ambients mobility

- The exit reduction: m[n[out(m). P] ||Q]

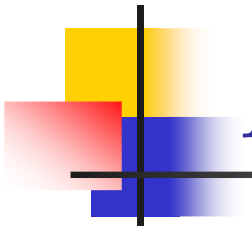


Ambients mobility

- The open reduction:

  open(n).Q || n[P]

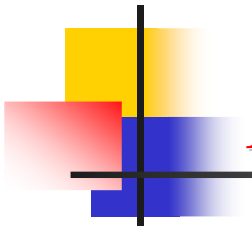
Q P



Ambients Communication

- A message: ⟨M⟩
- Read action: (x).P

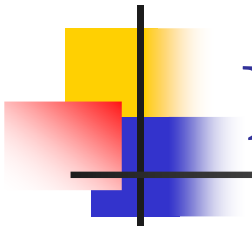
P[M/x]



Klaim

Kernel Language for Agent Interaction and Mobility

- Process Algebra Flavored
- Linda based communication model:
  - Asynchronous communication;
  - Via tuple space.
- Explicit use of *localities*:
  - Multiple distributed tuple spaces.
- Code mobility.

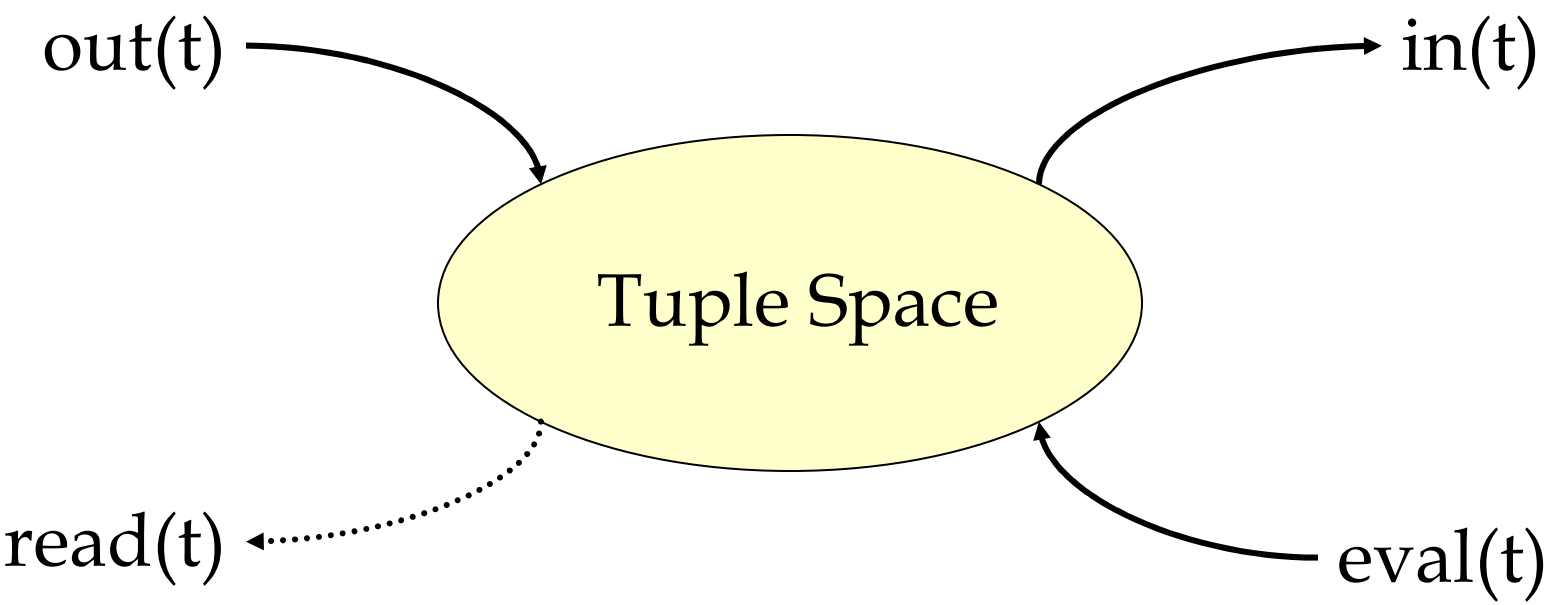


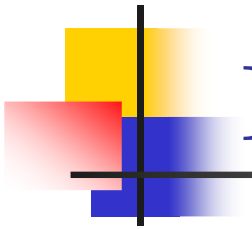
Linda Communication Model

- Tuples (“foo”, 10+5, !x)
  - 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)

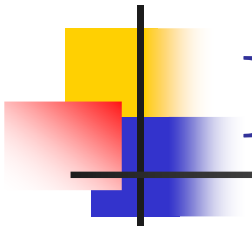
Linda Communication Model





Philosophers dining with Linda

```
Phil(int i) { while true {
        think();
        in("ticket"); in("fork", i); in("fork", i+1%5);
        eat();
        out("fork", i); out("fork", i+1%5); out("ticket")}}

main() {
        int I; for{ (i=0; i<5,i++)
        out("fork", i); eval(Phil(i));
        if (i<4) out("ticket")}}
```

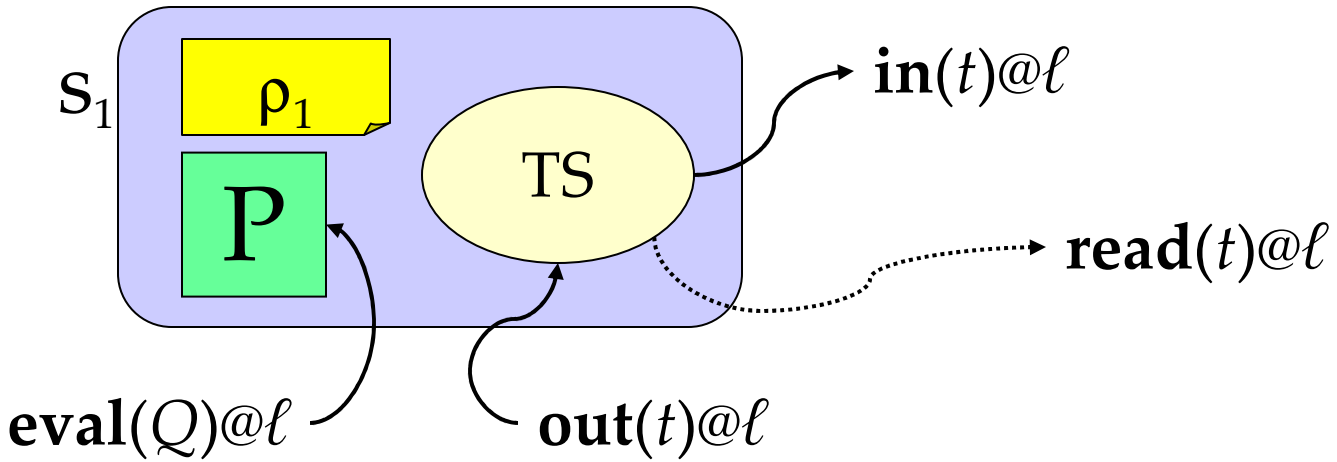


From Linda to Klaim

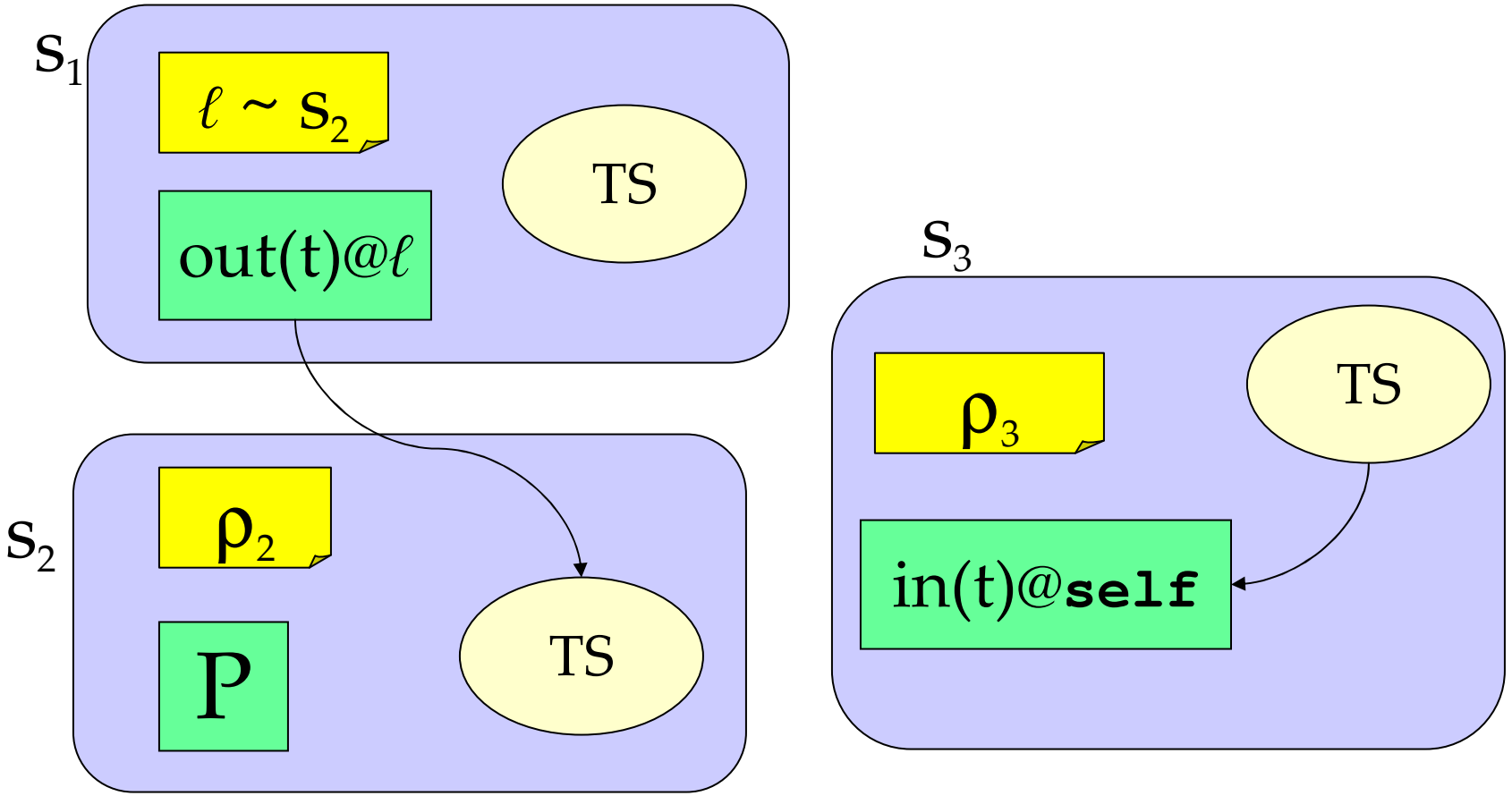
- Localities to model distribution
  - Physical Locality (sites)
  - Logical Locality (names for sites)
  - A distinct name *self* indicates the site a process is on.
- Allocation Environment to associate site to logical locality
  - This avoids the programmers to know the exact physical structure.

Klaim Nodes

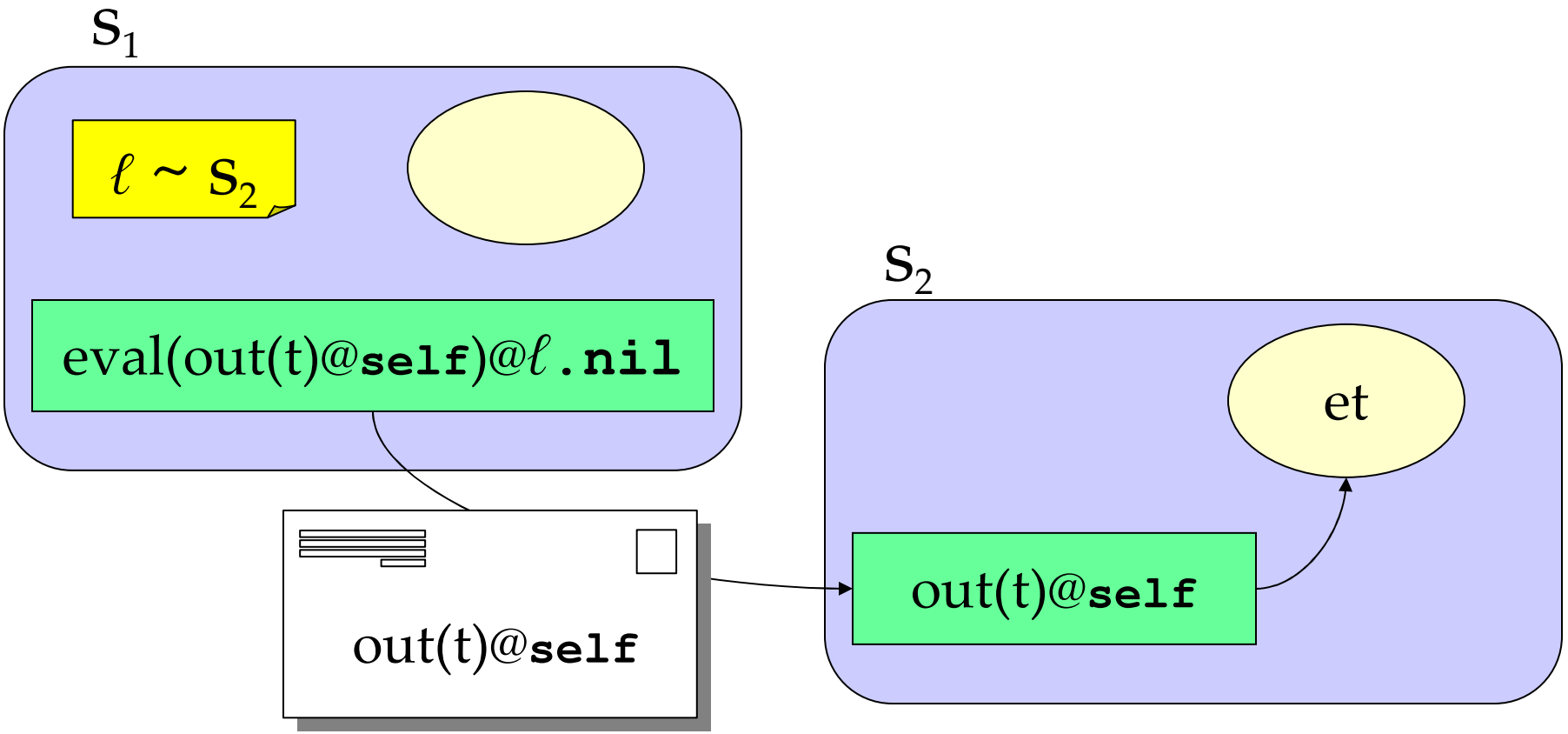
- Name (phys. loc.)
- Tuple space
- Processes
- Environment



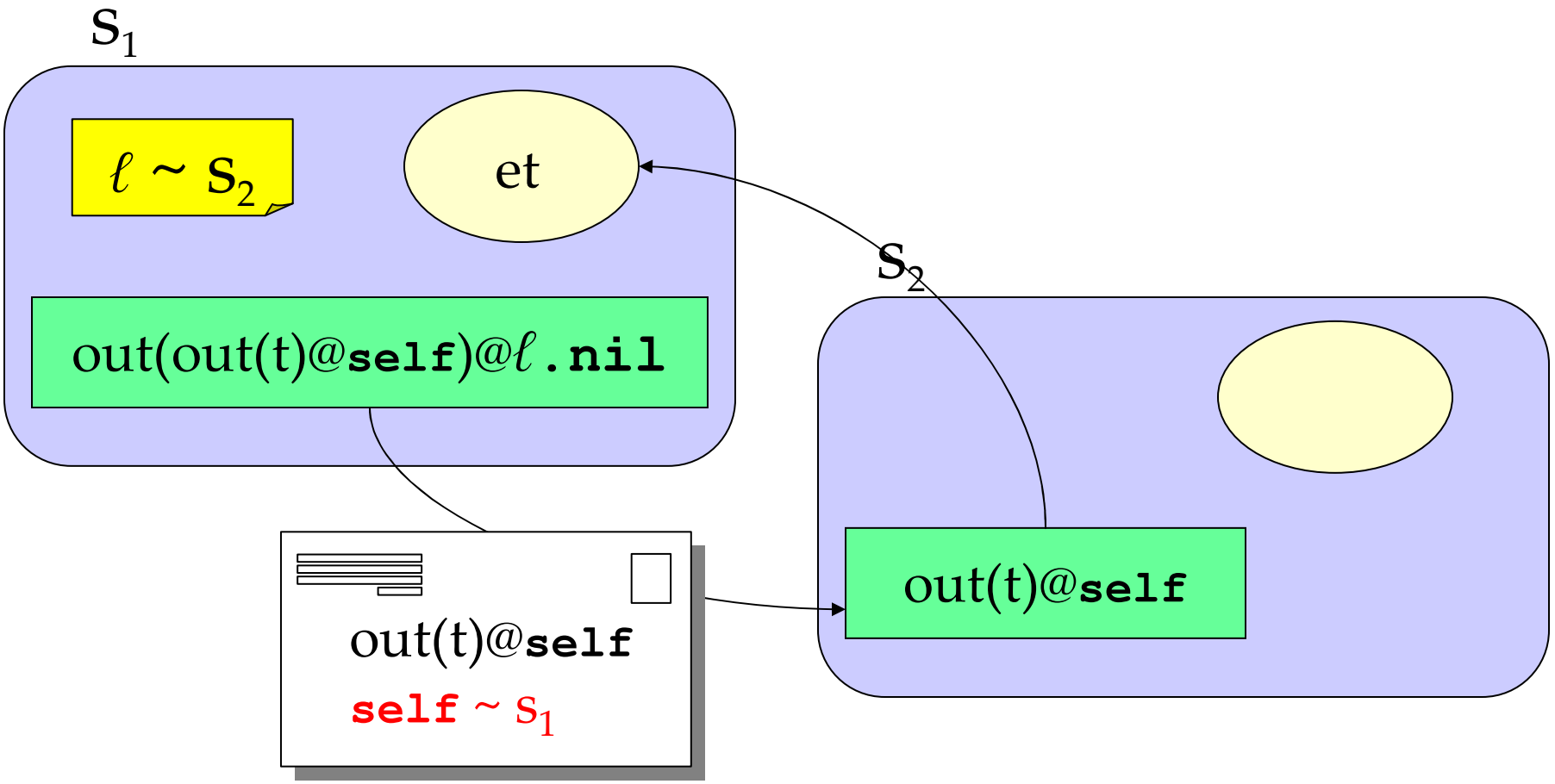
Klaim Nets



Dynamic Scoping



Static Scoping



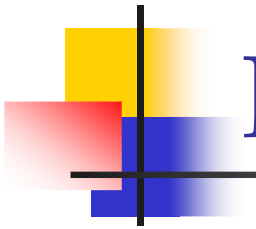
Klaim Processes

$$
\begin{array}{lcll}
P & ::= & \mathbf{nil} & \text{(null process)} \\
 & | & a.P & \text{(action prefixing)} \\
 & | & P_1 \mid P_2 & \text{(parallel composition)} \\
 & | & X & \text{(process variable)} \\
 & | & A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{e} \rangle & \text{(process invocation)}
\end{array}
$$

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

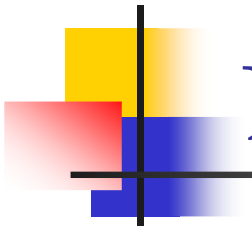
$$
t ::= f \mid f, t
$$

$$
f ::= e \mid P \mid \ell \mid !x \mid !X \mid !u
$$



Nets

$$
\begin{array}{lcll}
N & ::= & s ::_{\rho} P & \text{(node)} \\
 & | & N_1 \parallel N_2 & \text{(net composition)}
\end{array}
$$



Labelled Operational Semantics

It is given in two steps:

- *Local* rules (Processes):
  - availability of resources;
  - resources request.
- *Global* rules (Nets):
  - system evolution
  - actual use of resources.

Transition Labels

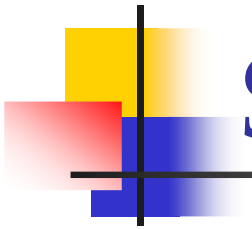
Labels reflect:

- *Information* transmitted over the net:

$$\xrightarrow{\mathbf{i}(s_1, t, s_2)}$$

- *Resources* available:

$$\xrightarrow{et@s} \qquad \xrightarrow{\rho@s}$$



Structural Congruence

$$N_1 \parallel N_2 \;=\; N_2 \parallel N_1$$

$$(N_1 \parallel N_2) \parallel N_3 \;=\; N_2 \parallel (N_1 \parallel N_3)$$

$$s ::_\rho (P_1 \mid P_2) \;=\; s ::_\rho P_1 \parallel s ::_\rho P_2$$

Local rules

$$s ::_\rho P \xrightarrow{\rho@s} s ::_\rho P \qquad s ::_\rho \mathbf{out}(et) \xrightarrow{et@s} s ::_\rho \mathbf{nil}$$

$$s ::_\rho \mathbf{out}(t)@\ell.P \xrightarrow{\mathbf{o}(s, \mathcal{T}[\![\, t \,]\!]_\rho, \rho(\ell))} s ::_\rho P$$

$$s ::_\rho \mathbf{in}(t)@\ell.P \xrightarrow{\mathbf{i}(s, \mathcal{T}[\![\, t \,]\!]_\rho, \rho(\ell))} s ::_\rho P \qquad s ::_\rho \mathbf{read}(t)@\ell.P \xrightarrow{\mathbf{r}(s, \mathcal{T}[\![\, t \,]\!]_\rho, \rho(\ell))} s ::_\rho P$$

$$\frac{s' = sup(\mathbf{succ}(s), s ::_\rho P')}{s ::_\rho \mathbf{newloc}(u).P \xrightarrow{\mathbf{n}(s, -, s')} s ::_\rho P[s'/u]}$$

$$\frac{s ::_\rho P[\widetilde{P}/\widetilde{X}, \widetilde{\ell}/\widetilde{u}, \widetilde{e}/\widetilde{x}] \xrightarrow{a} N'}{s ::_\rho A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{e} \rangle \xrightarrow{a} N'} \quad A(\widetilde{X}, \widetilde{u}, \widetilde{x}) \stackrel{def}{=} P$$

$$\frac{s ::_\rho P_1 \xrightarrow{a} s ::_\rho P_1'}{s ::_\rho P_1 + P_2 \xrightarrow{a} s ::_\rho P_1'} \qquad \frac{s ::_\rho P_1 \xrightarrow{a} s ::_\rho P_1'}{s ::_\rho P_1 | P_2 \xrightarrow{a} s ::_\rho P_1' | P_2}$$

$$\frac{N_1 \xrightarrow{\mathbf{n}(s_1, -, s_2)} N_1' \quad s_3 = sup(s_2, N_1 \parallel N_2)}{N_1 \parallel N_2 \xrightarrow{\mathbf{n}(s_1, -, s_3)} N_1'[s_3/s_2] \parallel N_2} \qquad \frac{N_1 \xrightarrow{a} N_1' \quad a \neq \mathbf{n}(s_1, -, s_2)}{N_1 \parallel N_2 \xrightarrow{a} N_1' \parallel N_2}$$

Global rules

$$\frac{N_1 \xrightarrow{\mathbf{o}(s_1,et,s_2)} N_1' \quad N_1' \xrightarrow{\rho@s_2} N_2}{N_1 \succ\!\xrightarrow{\mathbf{o}(s_1,et,s_2)} N_2 \parallel s_2 ::_\rho \mathbf{out}(et)} \qquad \frac{N_1 \xrightarrow{\mathbf{e}(s_1,P,s_2)} N_1' \quad N_1' \xrightarrow{\rho@s_2} N_2}{N_1 \succ\!\xrightarrow{\mathbf{e}(s_1,P,s_2)} N_2 \parallel s_2 ::_\rho P}$$

$$\frac{N_1 \xrightarrow{\mathbf{i}(s_1,et_1,s_2)} N_1' \quad N_1' \xrightarrow{et_2@s_2} N_2 \quad match(et_1,et_2)}{N_1 \succ\!\xrightarrow{\mathbf{i}(s_1,et_2,s_2)} N_2[et_2/et_1]}$$

$$\frac{N_1 \xrightarrow{\mathbf{r}(s_1,et_1,s_2)} N_1' \quad N_1' \xrightarrow{et_2@s_2} N_2 \quad match(et_1,et_2)}{N_1 \succ\!\xrightarrow{\mathbf{r}(s_1,et_2,s_2)} N_1'[et_2/et_1]}$$

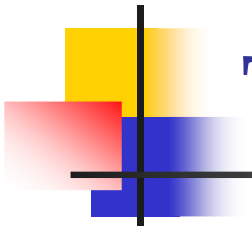
$$\frac{N_1 \xrightarrow{\mathbf{n}(s_1,-,s_2)} N_1' \quad N_1' \xrightarrow{\rho@s_1} N_2}{N_1 \succ\!\xrightarrow{\mathbf{n}(s_1,-,s_2)} N_2 \parallel s_2 ::_{[s_2/\mathtt{self}]\bullet\rho} \mathbf{nil}}$$

Zooming on rules

$$s ::_\rho \mathbf{in}(t)@\ell.P \xrightarrow{\mathbf{i}(s, \mathcal{T}[\![t]\!]_\rho, \rho(\ell))} s ::_\rho P$$

$$s ::_\rho \mathbf{out}(et) \xrightarrow{et@s} s ::_\rho \mathbf{nil}$$

$$\frac{N_1 \xrightarrow{\mathbf{i}(s_1, et_1, s_2)} N_1' \quad N_1' \xrightarrow{et_2@s_2} N_2 \quad match(et_1, et_2)}{N_1 \xrightarrow{\mathbf{i}(s_1, et_2, s_2)} N_2[et_2/et_1]}$$



Tuple evaluation

$$\mathcal{T}[\![\, e \,]\!]\rho = \mathcal{E}[\![\, e \,]\!]$$
$$\mathcal{T}[\![\, P \,]\!]\rho = P\{\rho\}$$
$$\mathcal{T}[\![\, \ell \,]\!]\rho = \rho(\ell)$$
$$\mathcal{T}[\![\, t_1, t_2 \,]\!]\rho = \mathcal{T}[\![\, t_1 \,]\!]\rho, \mathcal{T}[\![\, t_2 \,]\!]\rho$$

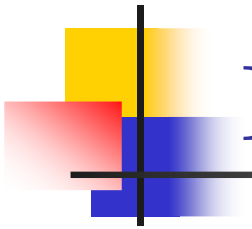
$$\mathcal{T}[\![\, !\, x \,]\!]\rho = !\, x$$
$$\mathcal{T}[\![\, !\, X \,]\!]\rho = !\, X$$
$$\mathcal{T}[\![\, !\, u \,]\!]\rho = !\, u$$

Matching Rules

$$match(v, v) \qquad match(P, P) \qquad match(s, s)$$

$$match(!\,x, v) \qquad match(!\,X, P) \qquad match(!\,u, s)$$

$$\frac{match(et_1, et_2)}{match(et_2, et_1)} \qquad \frac{match(et_1, et_2) \quad match(et_3, et_4)}{match((et_1, et_3), (et_2, et_4))}$$



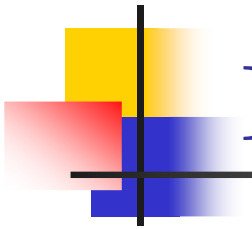
Paradigms for Mobile Code - 1

Code on demand

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

With Klaim:

…. **read**(!X)@l.X



Paradigms for Mobile Code - 2

Remote Evaluation

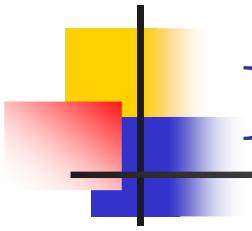
A component (*Client*) can require services from other components over the net *(Server),* by transmitting both the data needed to perform the task and the code that describes how to perform the service.

With Klaim:
Client:

out(**in**(<!y>@l. A(y)), v)@l

Server (@l)

in(!X, !x>)@self. **out**(x)@self. X



Paradigms for Mobile Code - 3

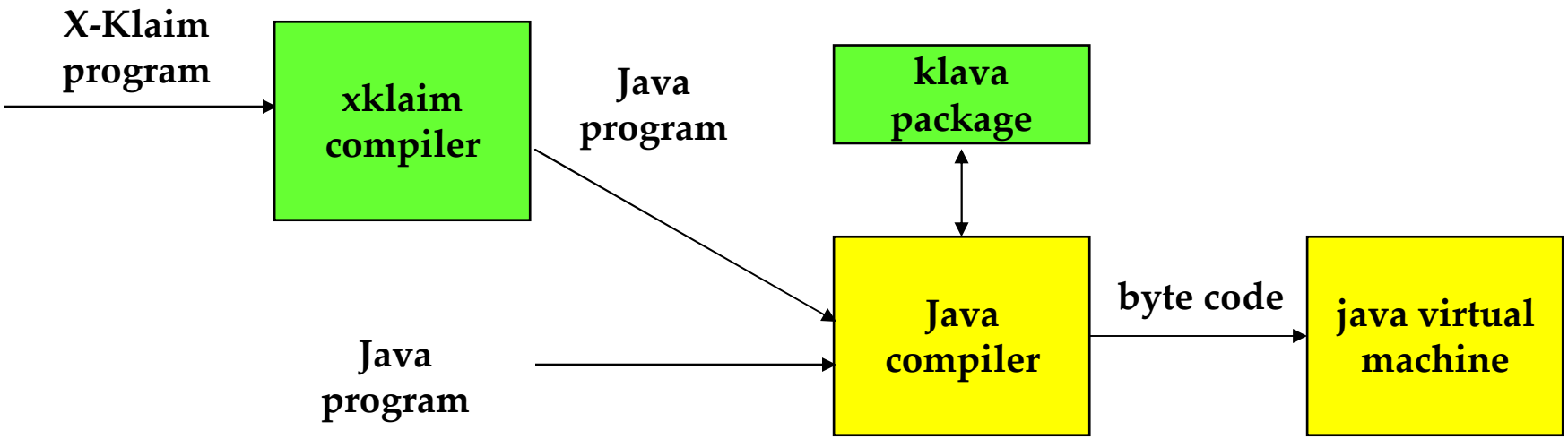
Mobile Agents

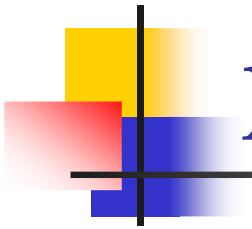
A process at a given can migrate to a different node where it will continue its execution (weak and strong mobility)

With Klaim:

eval(Q)@l.nil

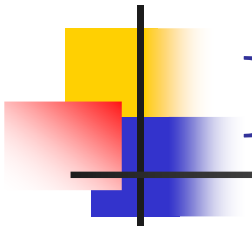
Framework for Klaim





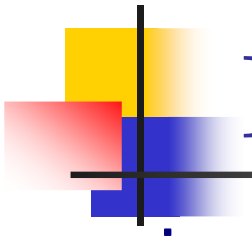
X-Klaim programming language

- Klaim operations
- High-level syntax:
  - Variable declarations
  - Conditionals
  - Assignments
  - Time-outs
- Strong Mobility



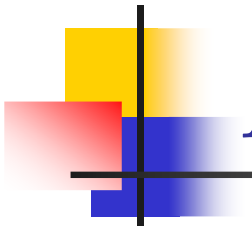
Package Klava

- Classes implementing Klaim operations and concepts
- Communications among processes and nodes
- Code mobility



KlavaProcess

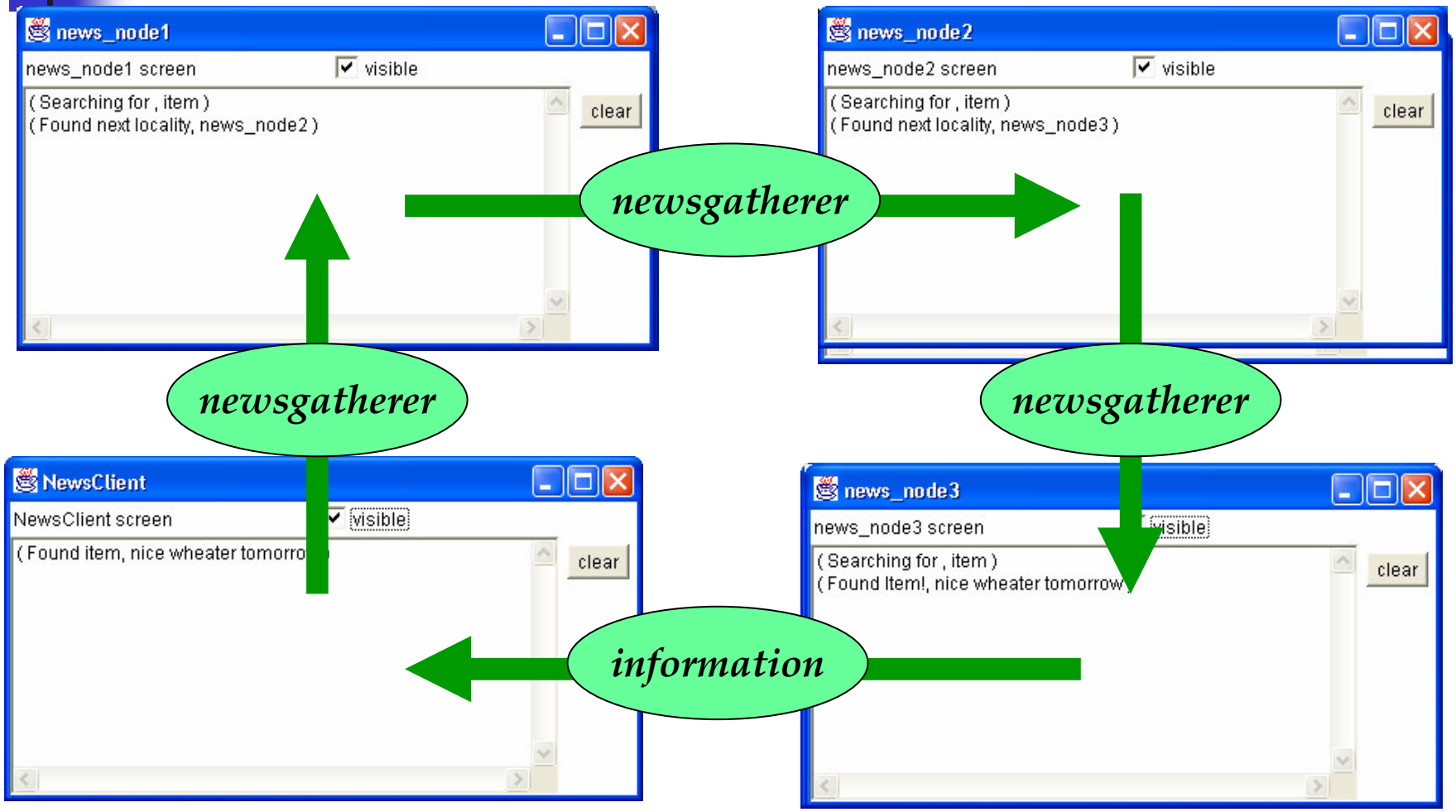
```
import Klava.*;

class MyProc extends KlavaProcess {
  public void execute() throws KlavaException {
    KString s = new KString() ;
    KInteger i = new KInteger() ;
    Locality loc = new Locality() ;

    read( s, i, loc, self ) ;
    // read( !s, !i, !loc )@self
    out( i, new KString("result"), loc ) ;
    // out( i, "result" )@loc
  }
}
```

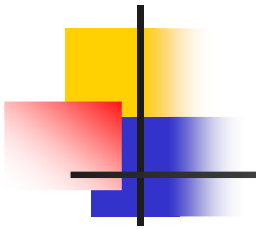


A NewsGatherer

- Some data are distributed over some nodes in a Klaim net
- Each node contains:
  - The information we are searching for, or
  - The locality of the next node to visit
- If the agent finds the information it sends it back to its owner
- Otherwise it migrates to the next site

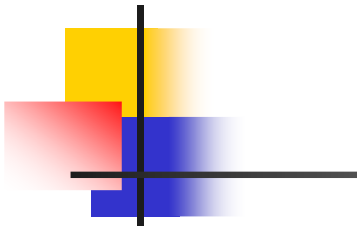
A NewsGatherer





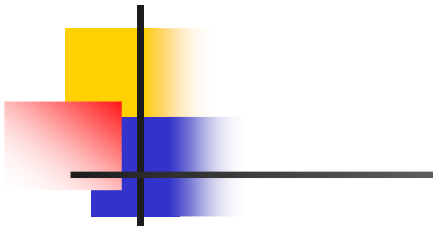
X-Klaim code

```
rec NewsGatherer[ item : str, retLoc : loc ]
  declare
    var itemVal : str ;
    var nextLoc : loc ;
    locname screen
  begin
    out( "Searching for ", item )@screen ;
    if read( item, !itemVal )@self within 2000 then
      out( "Found Item!", itemVal )@screen ;
      out( itemVal )@retLoc ;
      found := true
    else
      read( item, !nextLoc )@self ;
      out( "Found next locality", nextLoc )@screen;
      eval( NewsGatherer( item, retLoc ) )@nextLoc
    endif
  end
```



Java code

```
import Klava.*;
class NewsGatherer extends KlavaProcess {
  protected KString itemVal ;
  protected KString item ;
  protected Locality retLoc ;

  public NewsGatherer( KString item, Locality retLoc ) {
    this.item = item ;
    this.retLoc = retLoc ;
  }

  public void execute() throws KlavaException {
    itemVal = new KString() ;
    Print( "Searching for ", item ) ;
    try {
      read( item, itemVal, self, 2000 ) ;
      Print( "Found Item!", itemVal ) ;
      out( itemVal, retLoc ) ;
    } catch (KlavaTimeOutException e) {
      Locality nextLoc = new PhysicalLocality() ;
      read( item, nextLoc, self ) ;
      Print( "Found next locality", nextLoc ) ;
      eval( new NewsGatherer( item, retLoc ), nextLoc ) ;
    }
  }
}
```

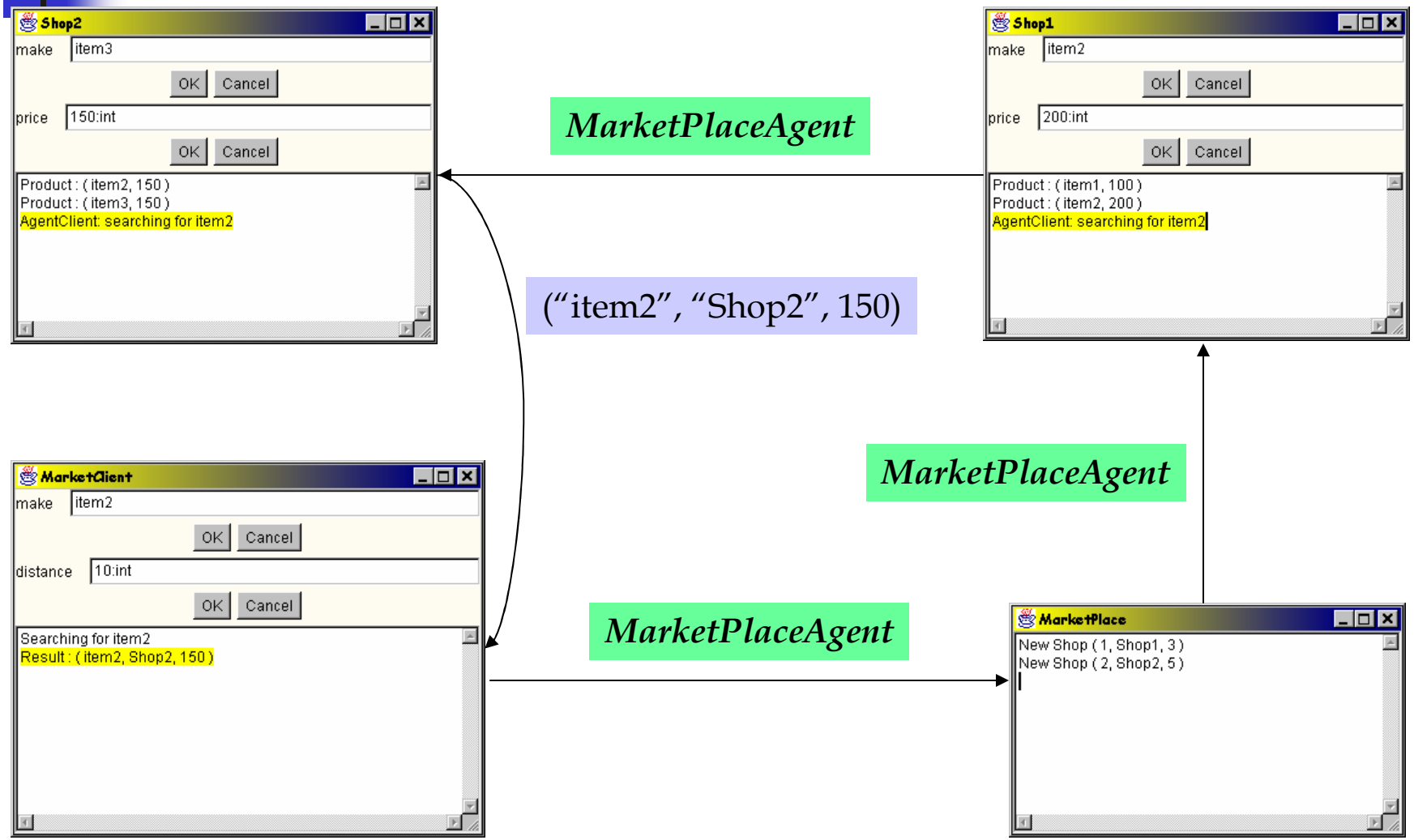


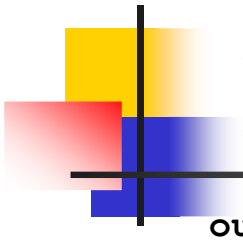
X-Klaim code

strong mobility

```
rec NewsGatherer[ item : str, retLoc : loc ]
  declare
    var itemVal : str ;
    var nextLoc : loc ;
    var found : bool ;
    locname screen
  begin
    found := false ;
    while not found do
      out( "Searching for ", item )@screen ;
      if read( item, !itemVal )@self within 2000 then
        out( "Found Item!", itemVal )@screen ;
        out( itemVal )@retLoc ;
        found := true
      else
        read( item, !nextLoc )@self ;
        out( "Found next locality", nextLoc )@screen;
        go@nextLoc
      endif
    enddo
  end
```

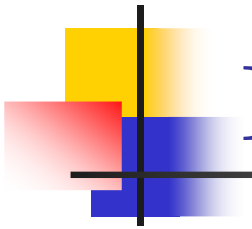
An Electronic MarketPlace





MarketPlaceAgent

```
out( "cshop", distance )@self;
in( "cshop", !shopList )@self;
again := true ; CurrentPrice := 0 ; CurrentShop := self ;
while ( again ) do
  if in( ! nextShop )@shopList within 0 then
     thisShop := nextShop ;
     go@nextShop ; # migrate to the next shop
     out( "AgentClient: searching for " )@screen ;
     out( ProductMake )@screen ;
     if read( ProductMake, ! newCost )@self within 10000 then
       if ( CurrentPrice = 0 OR newCost < CurrentPrice ) then
         CurrentPrice := newCost; CurrentShop := thisShop
       endif
     endif
  else
     again := false ;
     out( ProductMake, CurrentShop, CurrentPrice )@retLoc
  endif
enddo
```



Klaim site

http://music.dsi.unifi.it/klaim

- A few papers
- Current Implementation:
  - X-Klaim
  - Klava