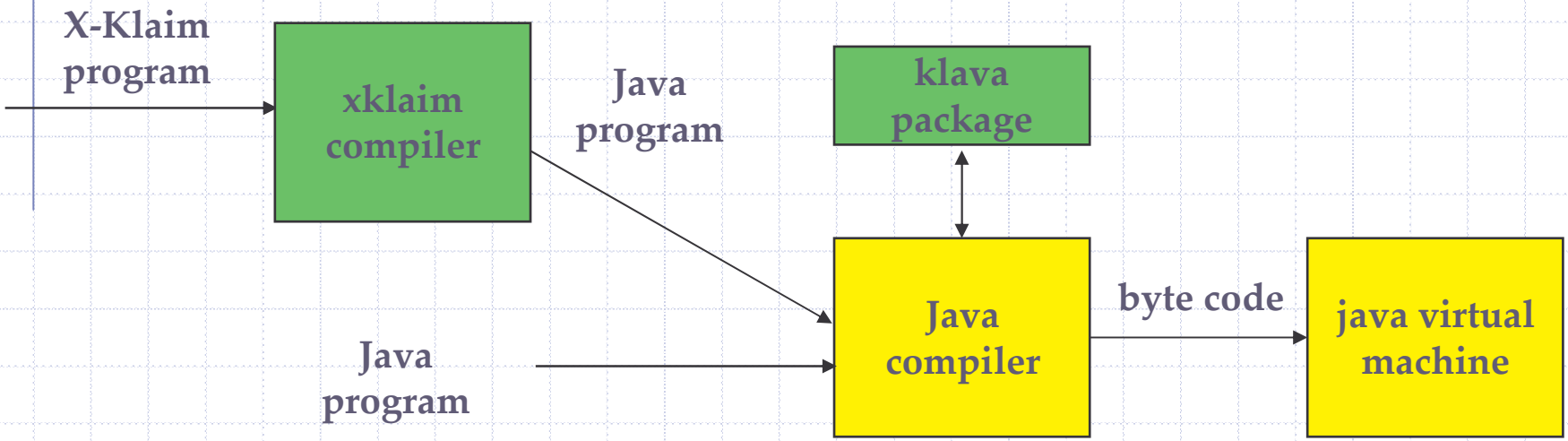
X-Klaim & Klava: Programming Object-Oriented Mobile Code in Open Nets

Lorenzo Bettini

Dip. di Sistemi e Informatica,
Univ. di Firenze

bettini@dsi.unifi.it

Framework for Klaim



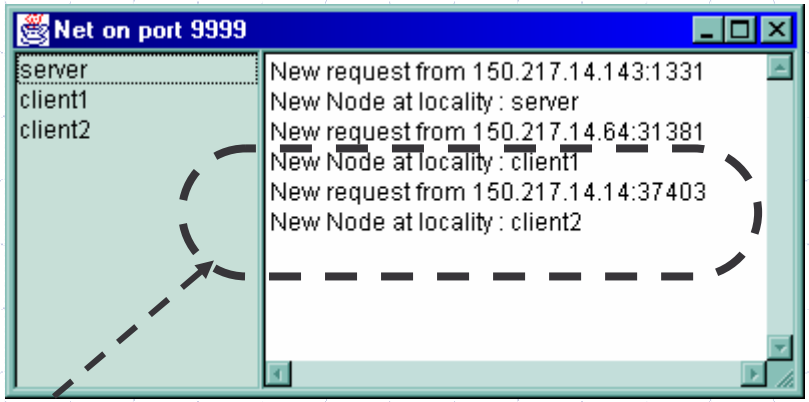
KlavaProcess

```
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
  }
}
```

Connecting to a Klava Net (old)

hostnet.dsi.unifi.it

```
$> java Klava.Net
Starting Net on port 9999
```



physical locality= IP:port

150.217.14.14

```
$> xklaim client2.xklaim
$> javac client2.java
$> java client2 hostnet.dsi.unifi.it 9999
Connecting to hostnet.dsi.unifi.it:9999 ...
Login as client2 ...
Login successful!
```

Communication Layer in Klaim

- is based on nodes' location knowledge,
- is influenced by:
  - Allocation environments;
  - Tuple and Process distribution.

Essentially flat:
non-structured, non-hierarchical

Dynamic Infrastructure

- The assumption that the underlying network is always available may be too strong;
- Permanent connection may not always be available;
- The knowledge of the address of a remote host may be not sufficient to communicate with it;
- Necessity of making *Node Connectivity* explicit in the language.

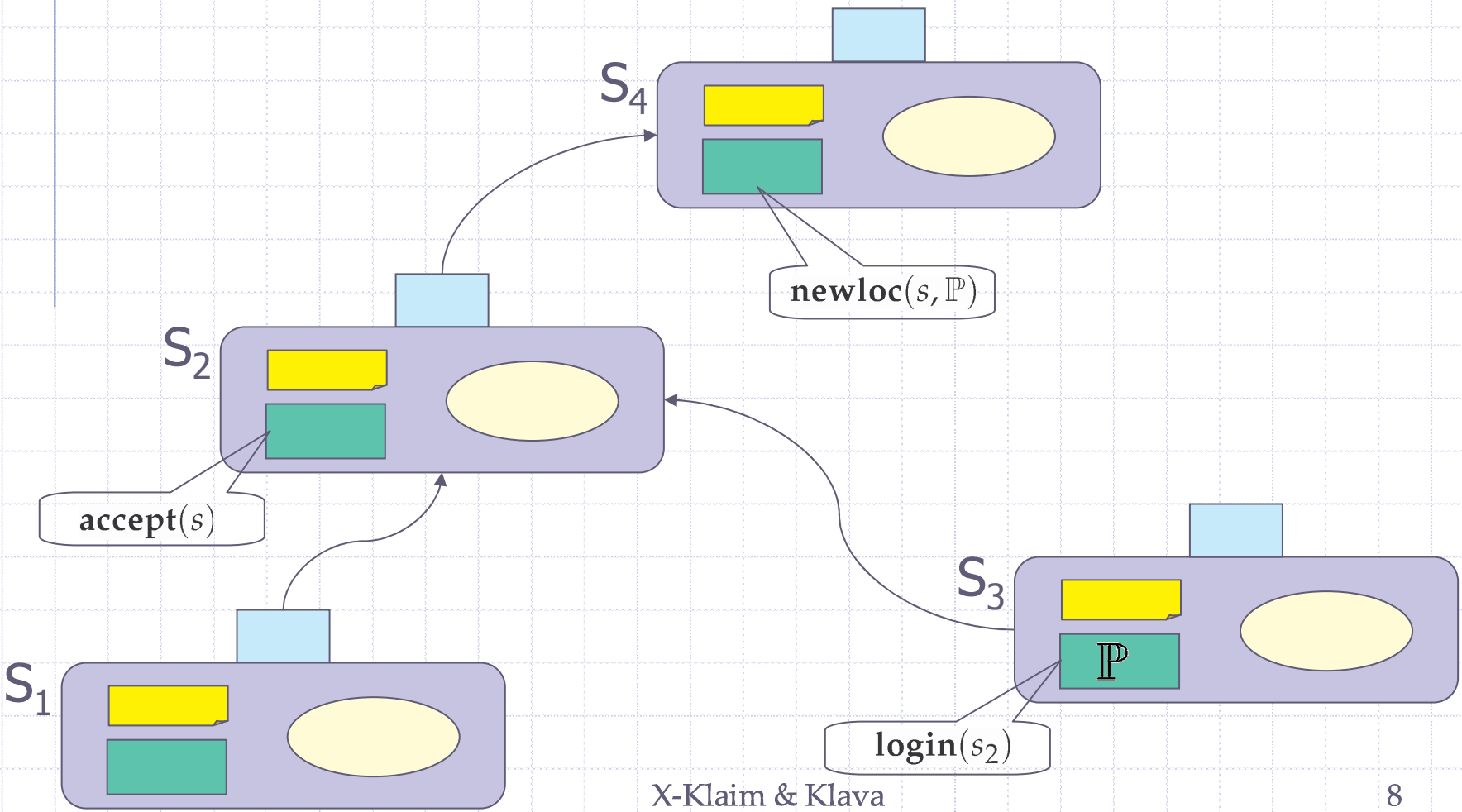
Open Nets

- New class of processes (*NodeCoordinators*) that:
  - can perform new *special* actions:

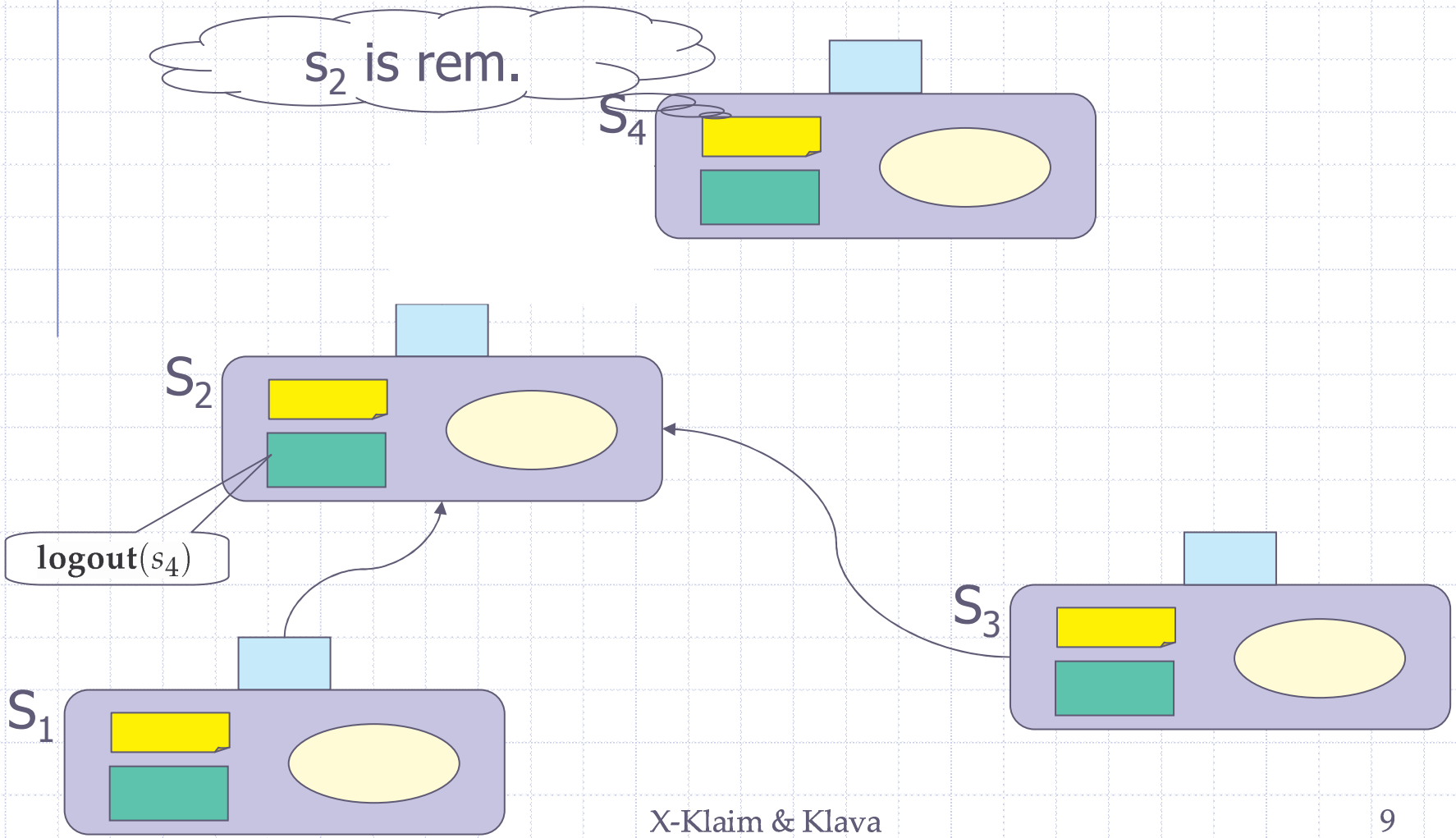
    $\mathbf{newloc}(s, \mathbb{P})$ $\mathbf{login}(\ell)$

    $\mathbf{logout}(\ell)$ $\mathbf{accept}(s)$
  - do not move
  - model the *network-interface* of a distributed system

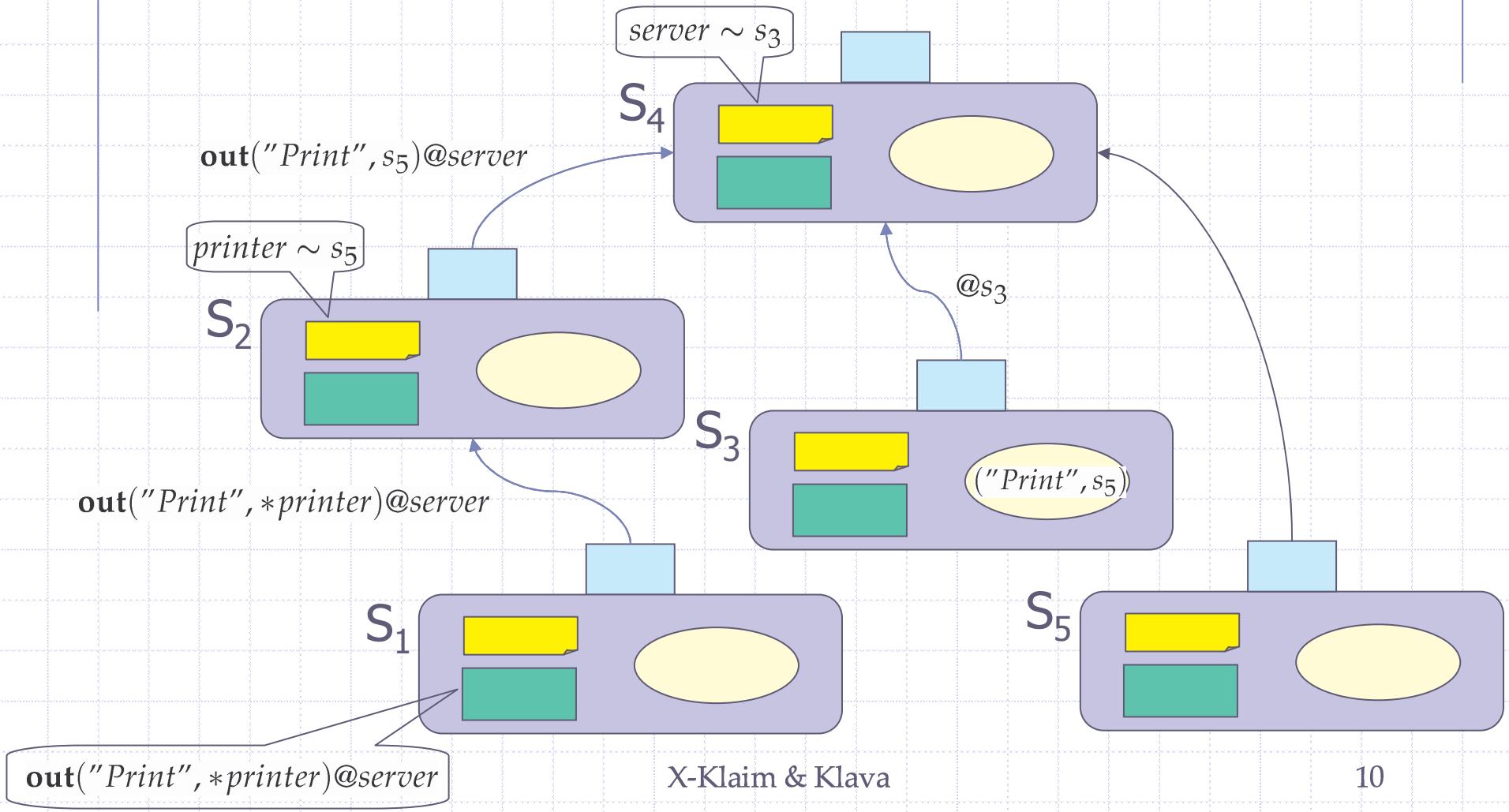
Dynamic evolution.... (1)



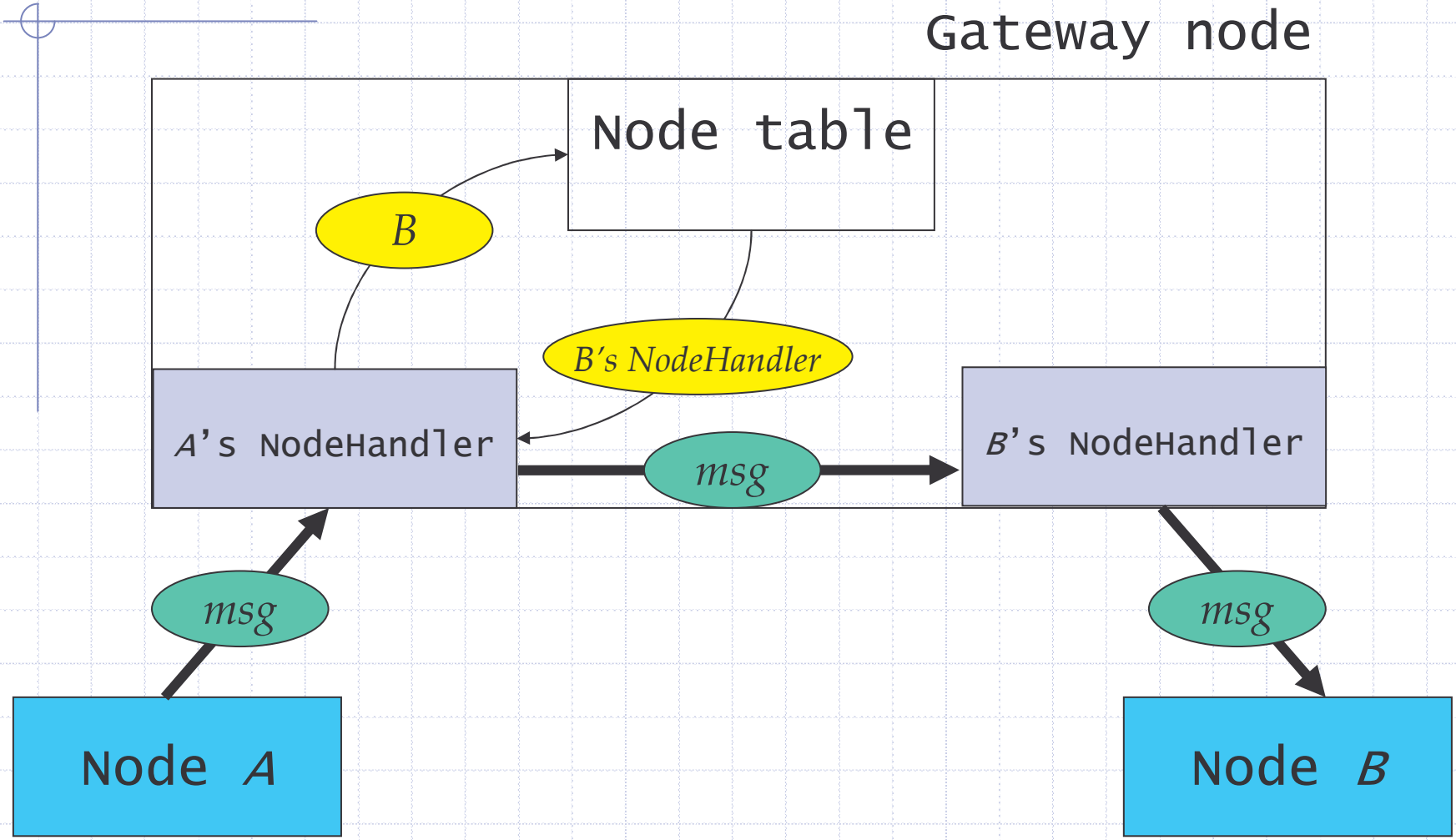
Dynamic evolution.... (2)



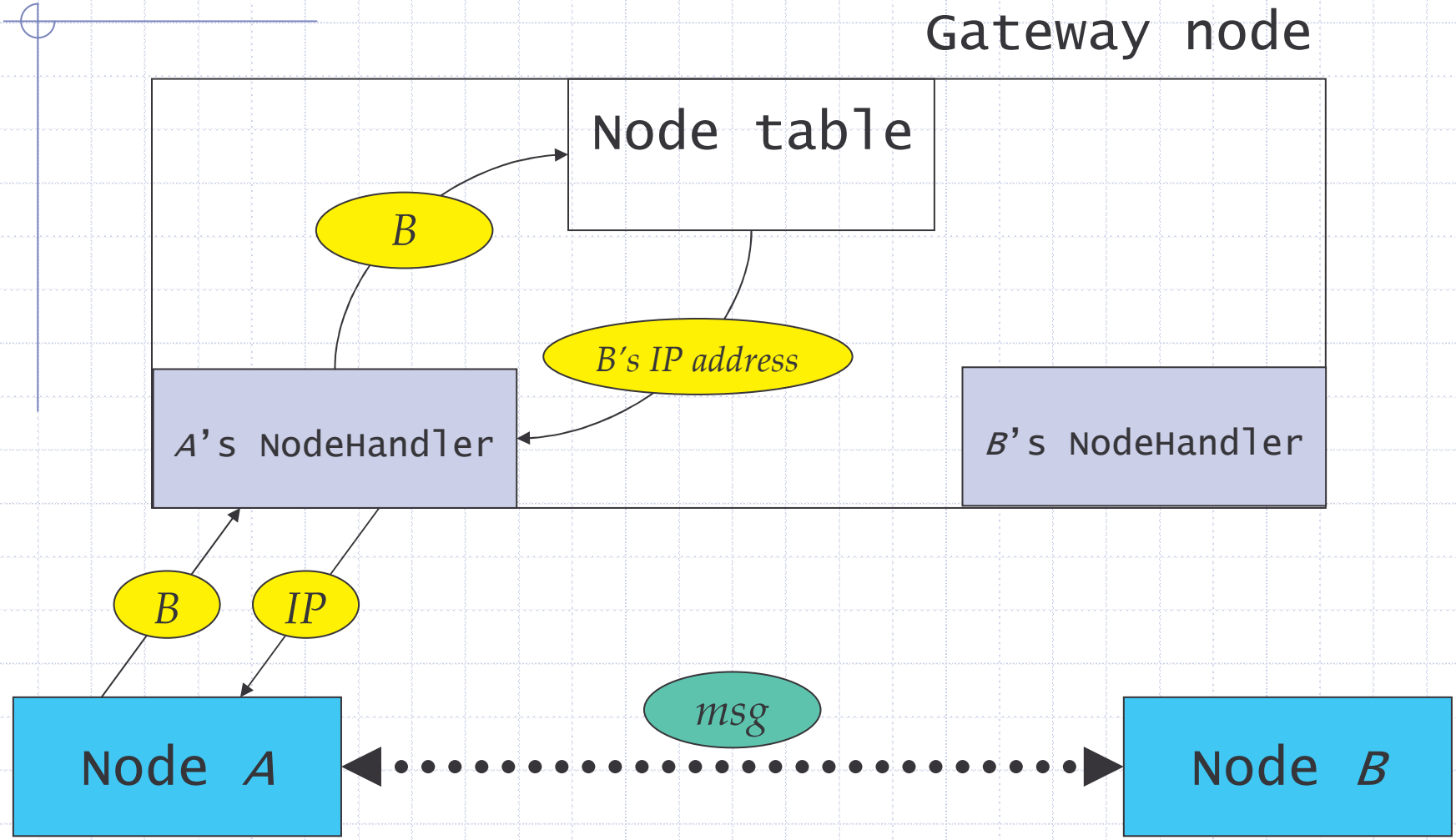
Dynamic name resolution



Communications (indirect)



Communications (direct)



Code Mobility in Klava

- Use *Java Reflection* to collect classes used by a process for:
  - Fields
  - Method signatures
  - Implemented interfaces and base class
  - Inner classes
- Send classes together with a migrating process (through *Java Serialization*)
- *NodeClassloader* of the receiving node stores these classes before deserializing a process

Code Mobility in Klava

```
class P extends KlavaProcess
implements I {
  Q q;
  R m(S s) throws E {
    …
  }
  …
}
…
P p = new P();
… use p …
eval(p, loc);
```

On the sender site

KlavaPacket

```
<serialized p>
[P,I,Q,R,S,E]
```

Code Mobility in Klava

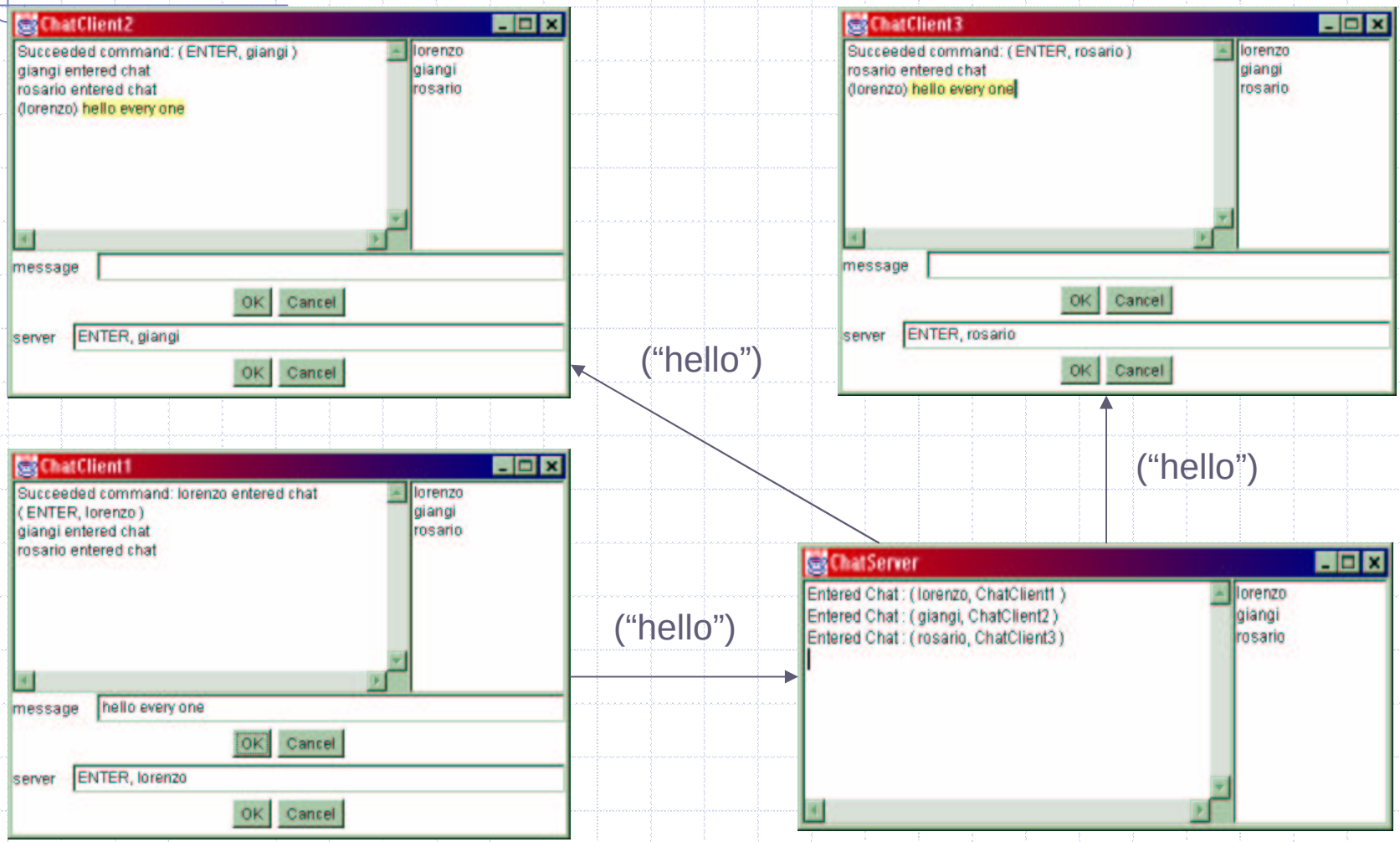
```
<serialized p>
[P,I,Q,R,S,E]
```

On the receiver site

1. Add [P,I,Q,R,S,E] to NodeClassloader's class table
2. Deserialize p
3. When P,I,Q,R,S,E are needed, the NodeClassloader will load them from its local table

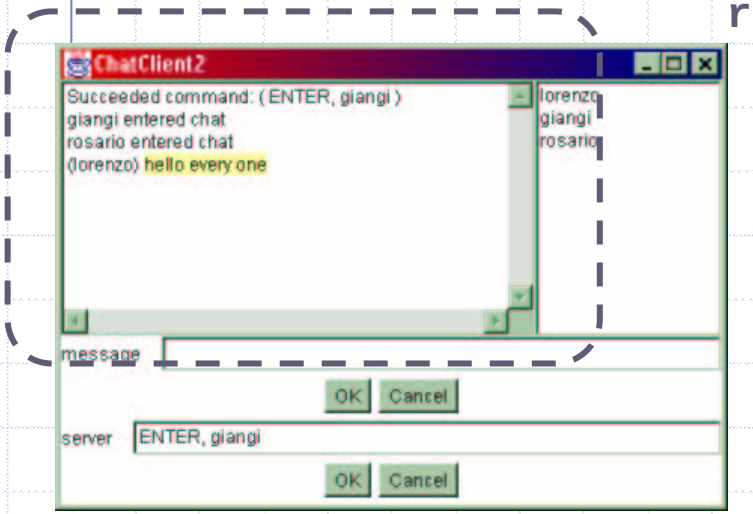
Programming Examples

A Simple Chat System



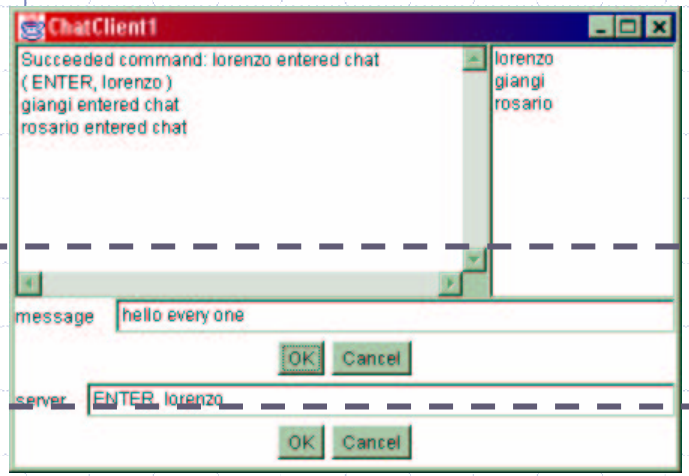
Chat Client

screen




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

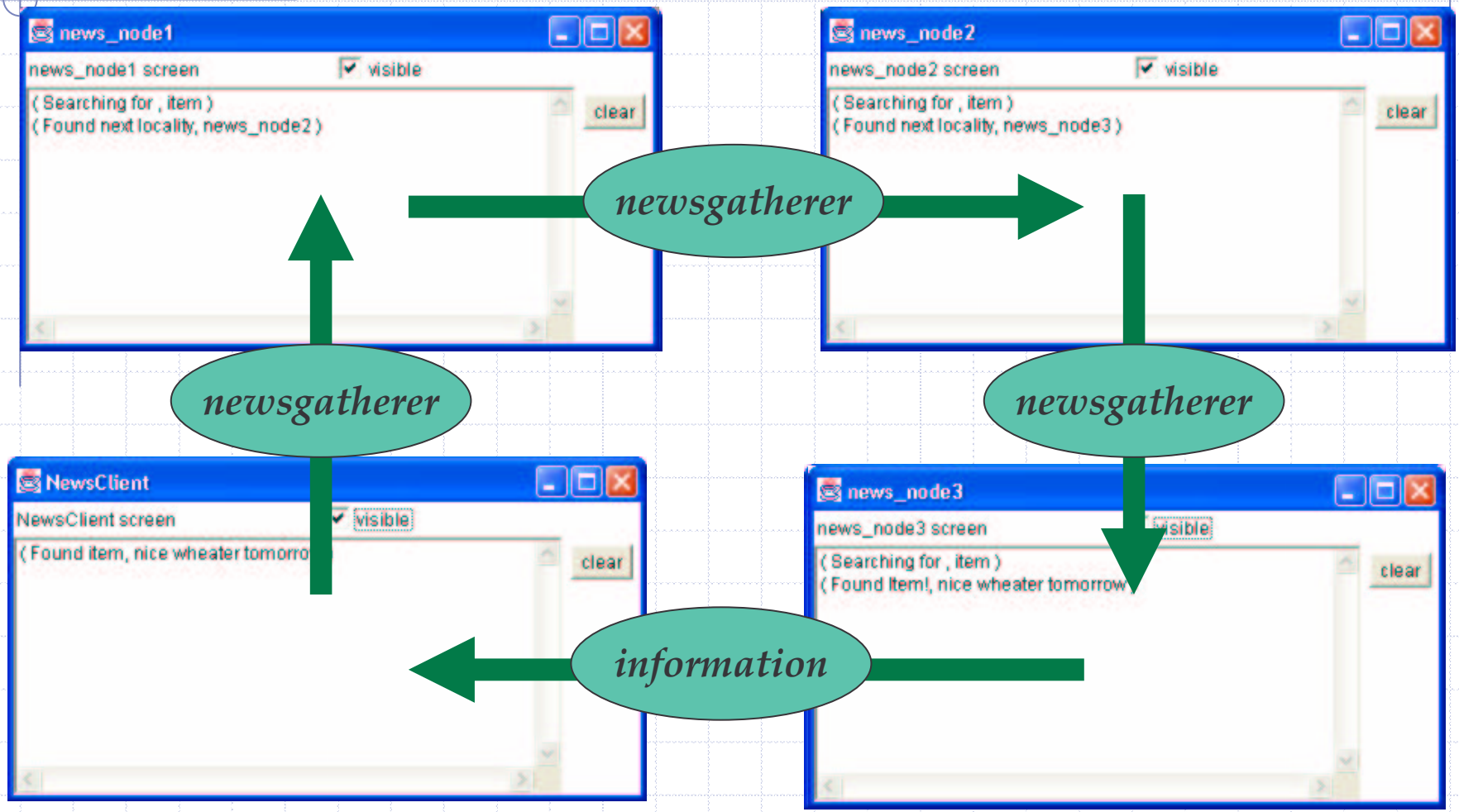
Chat Client



messageKeyb

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

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 ;
    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

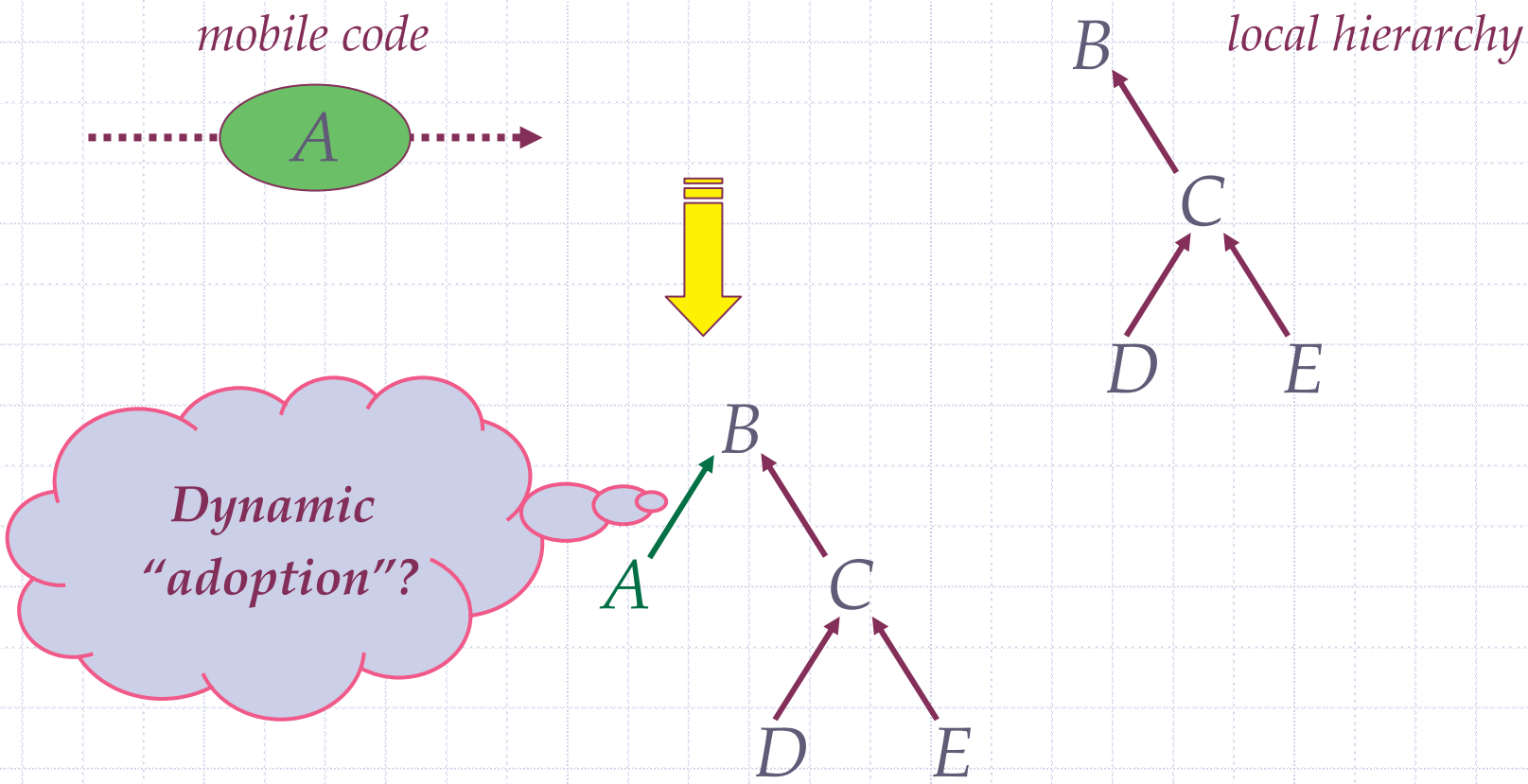
```
rec NewsGatherer[ item : str, retLoc : loc ]
  declare
    var itemVal : str ;
    var nextLoc : loc ;
    var found : bool ;
    locname screen
  begin
    found := false ;
    while not found do
      out( "Searching for ", item )@screen ;
      if read( item, !itemVal )@self within 2000 then
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

Object-Oriented Mobile Code

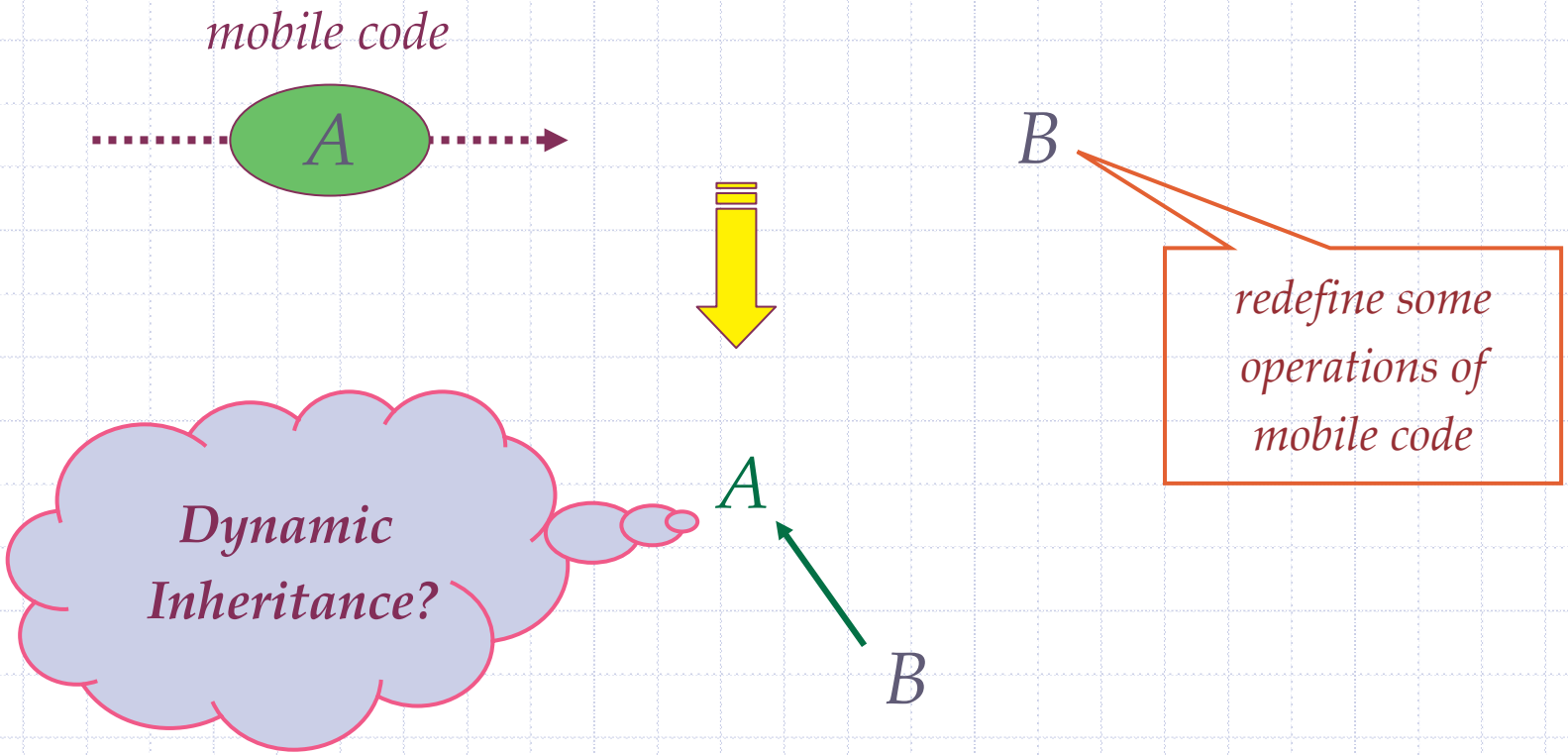
Object-Oriented: Goals

- Add object-oriented features to a language for distributed applications and mobile code, such as, e.g., Klaim
- Make object-oriented & mobile features integrate smoothly
- Exploit benefits of both

Remote Evaluation



Code On-Demand



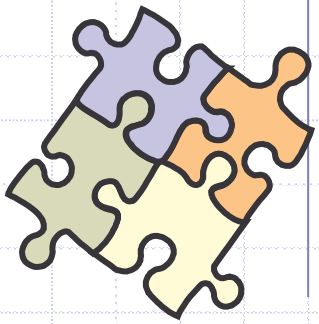
What we would like

- Dynamically inserting a class in an existing hierarchy <u>without</u>:
  - Modifying the source
    - Local code
    - Mobile code
  - Recompiling the whole hierarchy
- Having a uniform mechanism:
  - Mobile and local code can be used both as base class and as potential derived classes

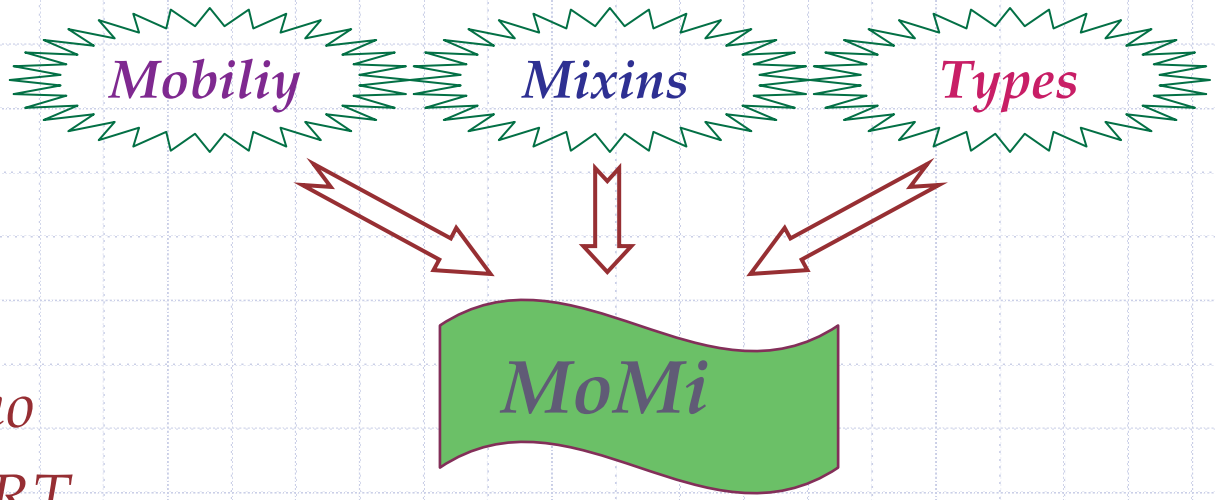
Mixin-based Inheritance

- *Mixin*: (Bracha '90) a class parameterized over the superclass
- A subclass is created by "applying" a mixin to a (base) class
- A subclass can be implemented before its superclass
- The same mixin can be applied to several base classes

Our Approach



- Mixins seem the right mechanism to structure OO mobile code
- We use mixins in a new environment: distributed mobile applications.



Cooperation:

Firenze – Torino

MIKADO - DART

A mixin example

M = mixin
  expect $[n : \tau]$
  redef $[m_2 : \tau_2$ with … next() …$]$
  def $[m_1 : \tau_1 = \ldots n() \ldots]$
end

C = class
  $[n : \tau = \ldots$
  $m_2 : \tau_2 = \ldots]$
end

Mixin application

$$(\text{new } (\text{M} \diamond \text{C})) \Leftarrow m_1()$$

MoMi & O'Klaim

- MoMi abstracts away from the actual underlying concrete language (for writing method bodies)
- MoMi can be "instantiated" with a specific coordination (mobile code) language
- O'Klaim is such an instantiation

O'Klaim (OO Klaim)

$$
\begin{array}{rcll}
P & ::= & \mathbf{nil} & \text{(null process)} \\
 & | & act.P & \text{(action prefixing)} \\
 & | & P_1 \mid P_2 & \text{(parallel composition)} \\
 & | & P_1 + P_2 & \text{(non-deterministic choice)} \\
 & | & X & \text{(process variable)} \\
 & | & A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{e} \rangle & \text{(process invocation)} \\
 & | & exp & \text{(object-oriented expression)} \\
 & | & \mathsf{let}\ x = exp\ \mathsf{in}\ P & \text{(let)} \\
act & ::= & \mathbf{out}(t)@\ell \mid \mathbf{in}(t)@\ell \mid \mathbf{read}(t)@\ell \mid \mathbf{eval}(P)@\ell \mid \mathbf{newloc}(u) & \\
t & ::= & f \mid f, t & \\
f & ::= & e \mid P \mid \ell \mid !x \mid !X \mid !u \mid !m : \tau &
\end{array}
$$

Exchanging OO mobile code

- All programs are statically type checked in their own local sites
- Type information is dispatched together with migrating code
- Communications are (dynamically) type checked using statically inferred types
- Integrating mobile code in a local class hierarchy does not compromise its correctness
- No further recompilation and type checking is required.

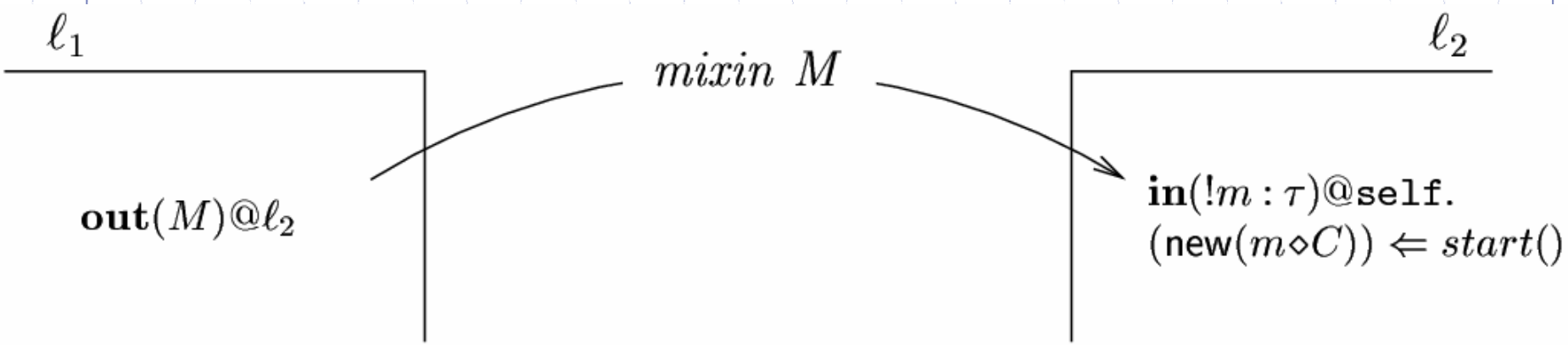
Subtyping class and mixin types

- Classes and mixins become polymorphic objects during mobile code exchange: *subtyping as a flexible communication pattern*

Es: **in**(x : **mixin**<…>). … $x \lozenge C$

The process can accept any mixin with a subtype.

O'Klaim (simple example)



- ℓ_1 sends M together with its (statically inferred) type τ'
- ℓ_2 awaits for a mixin of type τ
- $\tau' <: \tau$ thus pattern matching succeeds
- at ℓ_2: `(new (m ◇ C))←start()` was previously type checked with τ, and it is still type correct

Example (*OO mobile agent*)

- An agent migrates to a site where it has to print using the local printer
- The printer is a resource that depends on the local execution environment
- The agent is a mixin that expects a method *print*
- The mixin is completed on the local site with a class providing *print*

Implementation

in(!*mob_agent* : `agent`)@**`self`**.

let *PrinterAgent* = *mob_agent* ◇ *printer* in

(**new** *PrinterAgent*) ⟵ *start*()

`agent` = mixin<defines *start*, expects *print*>

type of *printer* = class<defines *print*>

Example (*OO mobile agent*) II

- An agent migrates to a site where it has to access the local file system
- The local site wants to avoid its own file system to be modified
- The local site redefines critical operations of the mobile agent
- The mobile agent is expected as a class and a local mixin redefines the critical method
- An instance of this "derived" agent is executed instead of the original one

Implementation II (*sandbox*)

in(!*mob_agent* : `agent`)@**`self`**.

let *RestrictedAgent* = *restricted* ◇ *mob_agent* in

(**new** *RestrictedAgent*) ⟵ *start*()

`agent` = class<defines *start*, *access*>

type of *restricted* = mixin<redefines *access*>

Implementation of O'Klaim

- A Java package, `momi`, implementing the virtual machine for MoMi objects, classes and mixins
- The X-Klaim compiler handles classes, mixins and objects, and generates code for `momi`
- Klava moves MoMi objects, classes and mixins

Privacy in Distributed Tuple Spaces

Security Problems

- Linda provides no access protection to a tuple space
- No way to determine the issuer of an operation to the tuple space
- A process may retrieve/remove data that do not belong to it
- Shared data can be easily modified and corrupted

Our Proposal

- Extend Linda operations with cryptography:
  - Tuples can contain encrypted data
  - Primitives for encryption/decryption
  - Do not completely change the original Linda model
- Suitable for distributed application and mobile agent based application

Privacy, not Security

- Our principal aim is not to avoid that wrong data be retrieved
- Our aim is that even if data is eavesdropped or stolen, still it cannot be read
- A sort of PGP for Linda
- Smooth extension of Linda
  - The impact on the Linda model is minimal
  - Previous applications continue to work

ink & readk

1. look for and possibly retrieve a matching tuple,
2. attempt a decryption of the encrypted fields of the retrieved tuple
3. if the decryption fails:
    1. if the operation was an ink then put the retrieved tuple back in the tuple space,
    2. look for alternative matching tuples,
4. if all these attempts fail, then block until another matching tuple is available.

Extended Pattern Matching

- The original pattern matching is not modified, but only extended
- Two stage pattern matching:
  - In the first stage an encrypted field is seen as an ordinary field with the type “encrypted” and it can match only with another “encrypted” field
  - In the second stage the decryption takes place, and a further matching is performed with the decrypted clear-text fields

CryptoKlava

- A subpackage of Klava providing these new modular extensions
- Based on Sun JCE (Java Cryptography Extension) providing basic interfaces and API for encryption
- Extended classes and extended operations

Basic Features

Wrapper class for encrypted tuple fields

```
KInteger i = new KInteger();
KCipher ki = new KCipher(ki);
ink("foo", ki, l2);
Print("retrieved: " + i);
```

```
KInteger num = new KInteger(150);
KCipher knum = new KCipher(num);
out("foo", knum, self);
```

Dealing with Mobile Agents

- Symmetric and asymmetric key encryption techniques rely on the secrecy of the private key
- Mobile code and mobile agents must not carry private keys when migrating to a remote site

Finer Grain Mechanisms

- Explicit operations: **enc** & **dec** acting on single tuple fields
- Mobile agents retrieve encrypted tuples with standard Linda operations (e.g. without decrypting them)
- Actual decryption will take place only at the home site (where the private key is stored) by stationary agents
- Wrong tuples retrieved by mistake have to be explicitly put back

Information Retrieval Agents

- Mobile agents can safely transport and use public keys also on remote sites
- Intermediate results can be encrypted so that they cannot be eavesdropped by other sites
- They can be decrypted only by the home site

References

Klaim site: **http://music.dsi.unifi.it**

- Papers
- Current Implementations:
  - X-Klaim
  - Klava
  - MoMi

Operational Semantics

$$\frac{\tau_1 <: \tau_2}{\ell_1 :: \mathsf{send}(A^{\tau_1}, \ell_2).P' \parallel \ell_2 :: \mathsf{receive}(id : \tau_2).Q \rightarrowtail \ell_1 :: P' \parallel \ell_2 :: Q[A^{\tau_1}/id]}$$

- The argument of a send is delivered annotated with its type
- This type has to be compliant with the one expected
- Communication takes place
- No further check is needed

Types

$$\begin{array}{lcl} \Sigma & ::= & \{m_i : \tau_{m_i}{}^{\,i \in I}\} \\ \tau & ::= & \Sigma \\ & | & \mathsf{class}\langle \Sigma \rangle \\ & | & \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \end{array}$$

width subtyping

$$\frac{\Sigma_2 \subseteq \Sigma_1}{\Sigma_1 <: \Sigma_2}$$

defined by the mixin

redefined by the mixin

expected from the superclass

Subtyping classes & mixins

$$\frac{\Sigma' <: \Sigma}{\mathsf{class}\langle \Sigma' \rangle \sqsubseteq \mathsf{class}\langle \Sigma \rangle}$$

$$\frac{\Sigma'_{new} <: \Sigma_{new} \qquad \Sigma_{exp} <: \Sigma'_{exp} \qquad \Sigma'_{red} = \Sigma_{red}}{\mathsf{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle \sqsubseteq \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle}$$

- The subtype can define more new methods and require less methods
- Cannot override more methods

An "Encrypted" Chat System

- Messages are exchanged via tuples with a specific structure
- Possible problems in a chat system:
  - An eavesdropper can intercept the messages and read their contents
  - A misbehaving chat server can examine clients' messages

An “Encrypted” Chat System

- When the client wants to send a private message to a specific receiver, it encrypts the body of the message with a key;
- The server receives the message and simply forwards it to the receiver;
- The receiver will receive the message with the encrypted body and it can decrypt it with the appropriate key.
- Both symmetric and asymmetric encryption schemes can be employed