Coordination & Security

Jan Vitek
S3Lab
Computer Sciences Department
PURDUE
UNIVERSITY

FOSAD 2004

Coordination

Coordination is the theory and practice of assembling software systems out of independently developed components

Coordination languages are intended to provide the glue around a set of, potentially heterogeneous, components

Coordination languages have been used successfully to compose applications parallel and distributed application for more than twenty years

Course Overview

"programming secure systems is easy; secure distributed systems are hard"

Goal: overview of security challenges and approaches in the field of coordination, focusing on data-centric languages of the Linda family

Two parts

Part 1

Coordination::

family of coordination languages; design space; use cases

Part 2

Security::

explore the space of security threats and potential responses

Part I
COORDINATION

Introducing Linda

Linda is a data-centric coordination language invented by *David Gelernter* (Yale) around 1982. The language has been widely successful sparking hundreds of papers, with a devoted conference, several commercial implementations and many research projects.

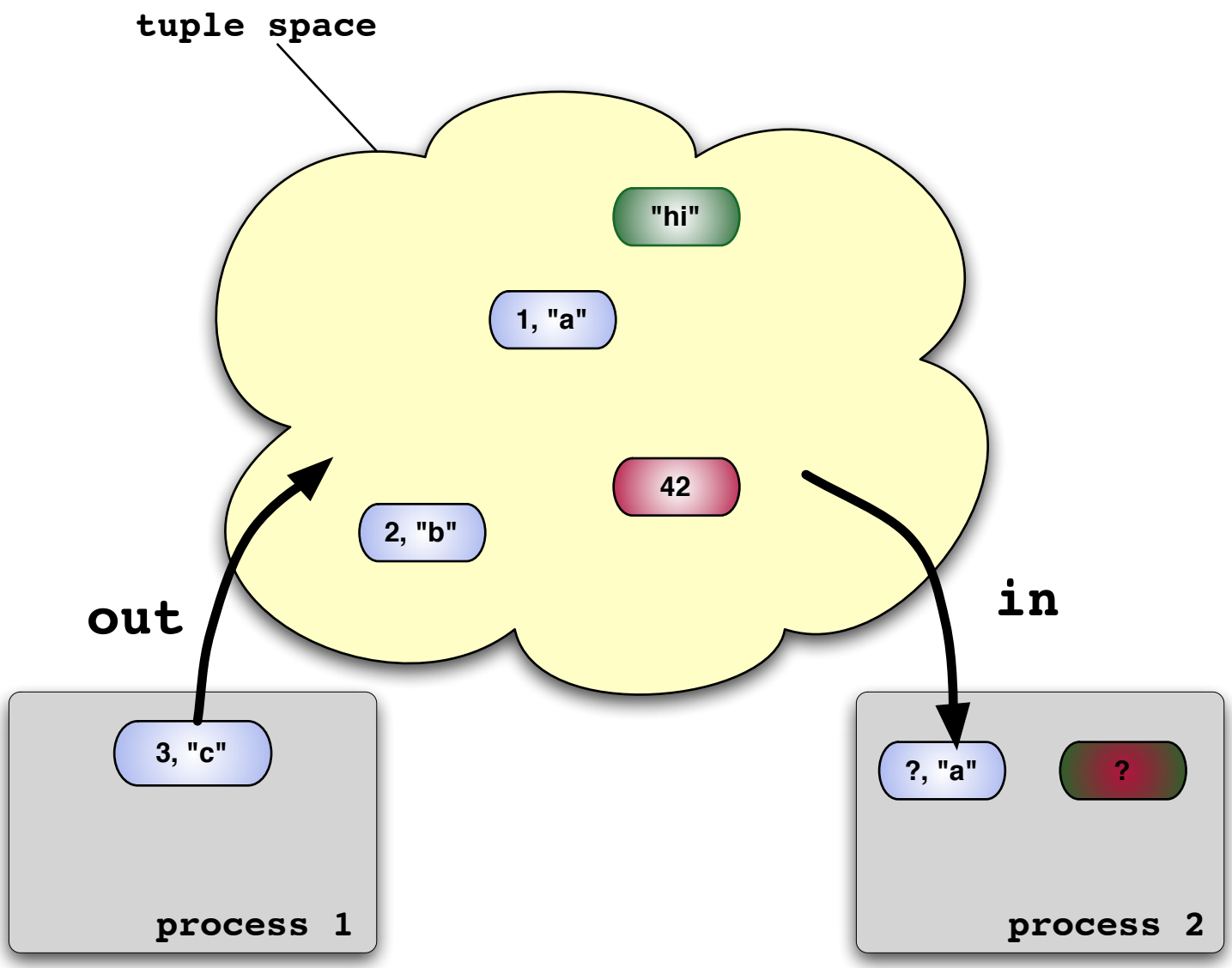
Originally designed for programming parallel computers, Linda has been adapted to weakly distributed systems, and more recently to wide area networks

The basic data structure, called ***tuple space***, underlying any Linda implementation is *a multiset of "tuples"*. *Tuples are ordered (finite) sequences of values.*

Linda must be embedded in an expression language. Originally the expression language was meant to express the sequential parts of a computation while Linda would specify the concurrent composition of the sequential fragments. Current hosts include C, C++, Fortran, Eiffel, Java.

The main innovation of Linda is the choice of a simple and elegant set of atomic operations for accessing tuple spaces based on pattern matching.

Linda



Middleware and Coordination

Middelware is the software that serves as the glue between two applications, sometimes called plumbing because it connects two sides of an application and passes data between them
www.cren.net

Middleware provides a layer insulating applications against change. In large and complex distributed environments: client and server codes are updated, interfaces evolve, implementation languages change, hosts are added, failures occur, connectivity fluctuates.

Linda-based coordination languages provides:

1. language neutrality (interoperability)
2. simple and stable interface (usability)
3. undirected communication (~anonymity)
4. uncoupled communication (persistent location transparent store)

Contrast with widely used message-based middleware such as CORBA Main differences are RPC style directed/synchronous communication (3, 4) and complex interface (2).

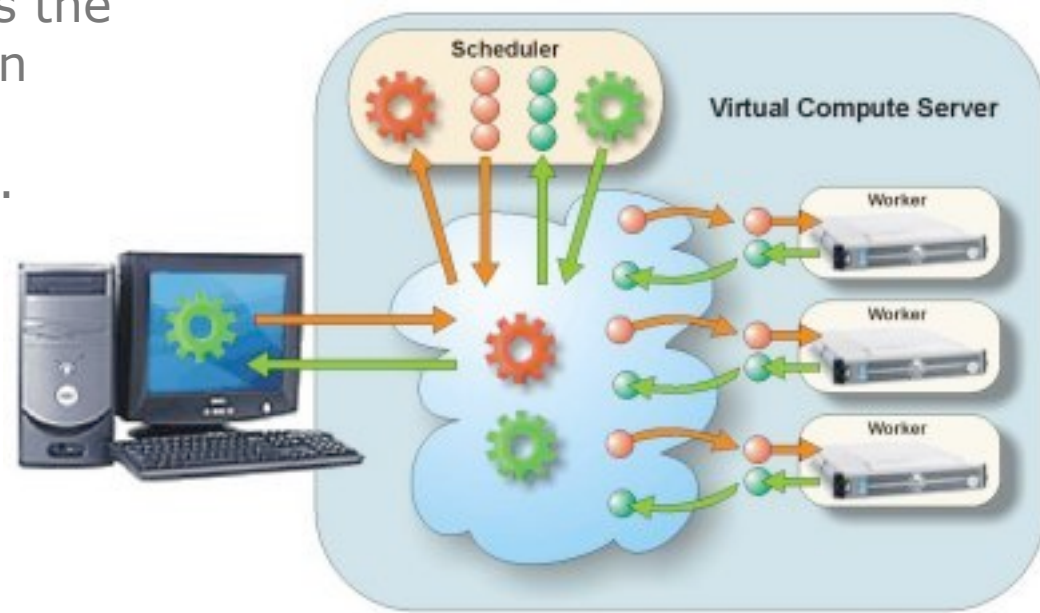
Case study 1: computational farms

A Computation Farm for Distributed Computing

Distributed computing is usually concerned with taking advantage of CPU, memory and other network resources by performing a computational task in parallel between all of them. It is a specific type of Peer to Peer application.

A user writes a set of commands into a space on the tuple spaces network. On the network are a number of workers that run on multiple computers and use their aggregated processing power. Each worker takes a task from the space and executes the related processing procedure. When processing is completed, the output is written back to the space. The final output is read by the mobile device from the space as a set of combined outputs.

(from the Gigaspaces web site)

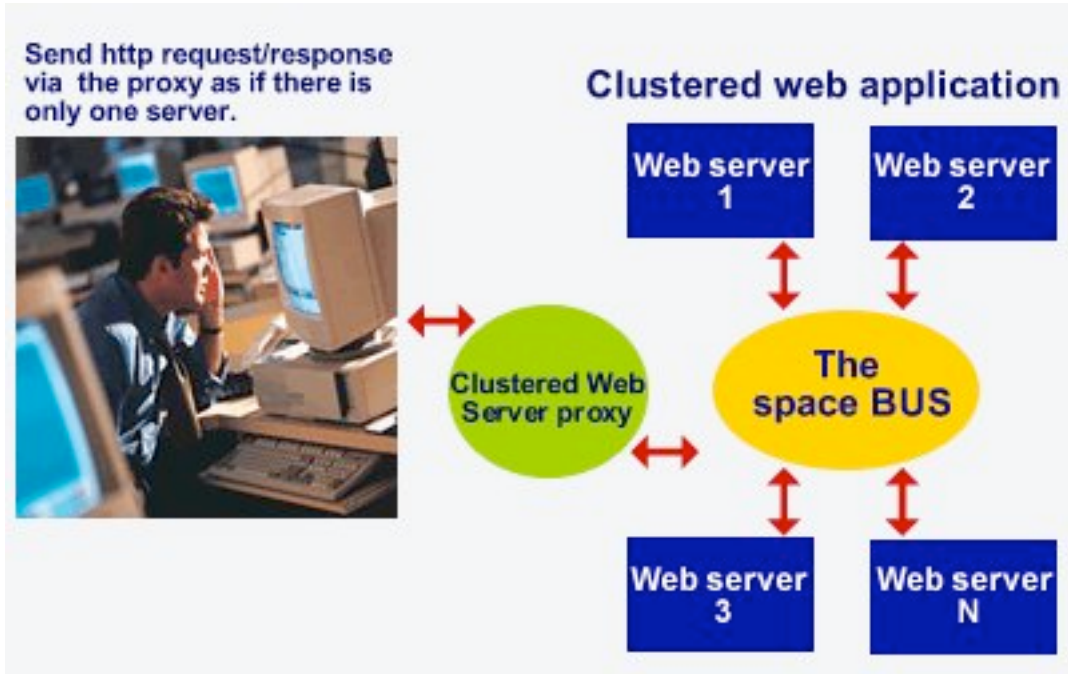


Case study II: message bus

A Messaging Bus for Clustering and Load Balancing

To support high availability and fault tolerance, the ideal solution is to build an adaptive server that can shrink or expand according to specific needs and budgets. This can be done by creating a network bus through which servers are hooked together and act as one to the outside world. Load is balanced simply by adding or removing servers.

The tuple space acts as a communication bus. Each server takes the appropriate packet, executes it and responds. Since the space provides the means for associative retrieval, different types of load-balancing topologies can be easily applied. For example, in one topology, all the servers are mirrored so that the first one available takes the request. This provides automatic load balancing; each request is sent to the most available server ....

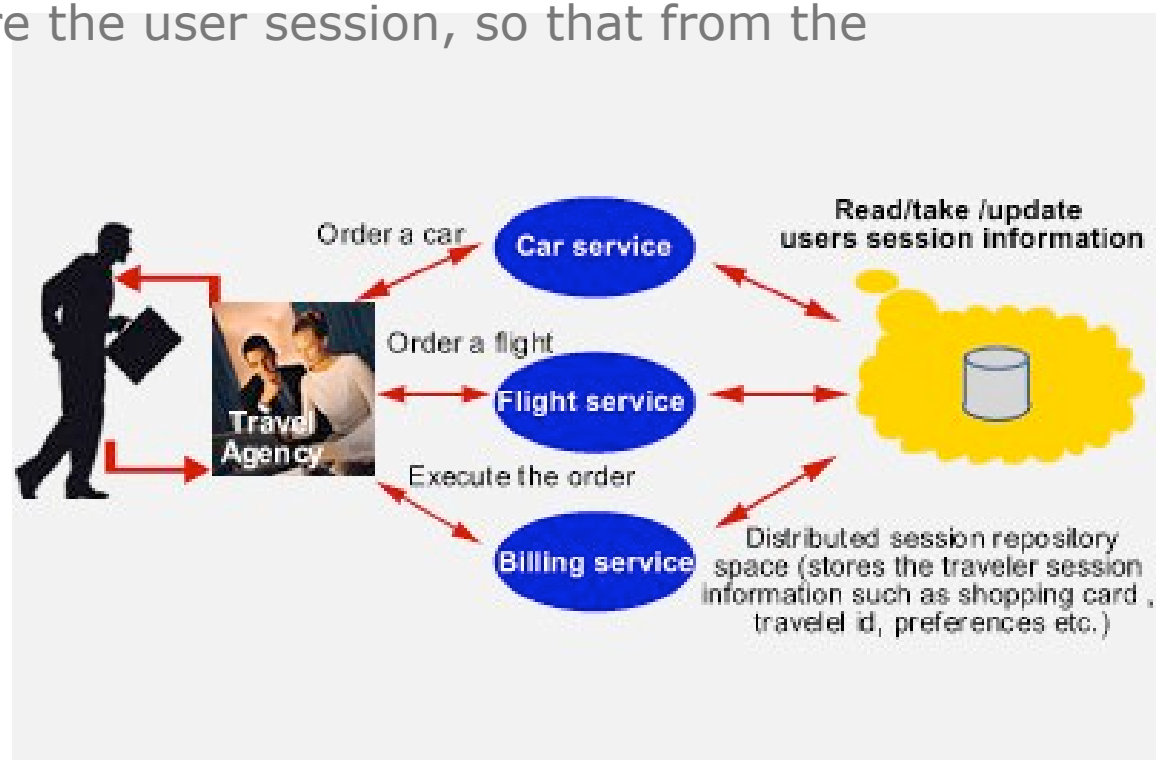


Case Study III: distributed repository

A Distributed Repository for Distributed Session Sharing

A distributed repository stores and retrieves data for distributed components. The data may be configuration data and/or session information.
vices.

A customer wants to order a car from one agency and a flight from another. The car and flight reservation systems are located in separate servers. Both systems need to be able to share the user session, so that from the customer's point of view it is a single transaction. Since these services are distributed, we cannot use the session information in each of the servers, but instead use the space as a distributed session server accessible to both the reservation services.



Historical perspective

A subjective, partial view of the history of coordination with a bias towards papers and systems that will be covered in the course.

1982 - First documented version of Linda

1985 - Gelernter, Generative communication in Linda, TOPLAS

1993 - Arbab. Herman, Spilling, An Overview of Manifold and its Implementation, C - P&E (control-centric coordination)

1996 - Ciancarini, Knoche, Tolksdorf, Vitali, PageSpace: An Architecture to Coordinate Distributed Applications on the Web, WWW

1996 - First COORDINATION conference

1998 - de Nicola, Ferrari, Pugliese, Klaim: a Kernel Language for Agents Interaction and Mobility, TSE

1998 - Busi, Gorrieri, Zavattaro, A process algebraic view of Linda coordination primitives, TCS

1999 - JavaSpaces

1999 - Bryce, Oriol, Vitek, A Coordination Model for Agents Based on Secure Spaces, Coordination

1999 - Picco, Murphy Roman, Lime: Linda Meets Mobility, MA

2005 (february?) - next COORDINATION -- Submit, submit early, submit often

Linda

Linda's simplicity can be contrasted to the complexity of language such as Ada.

Little know fact: Does Linda means *L*inda *I*s *N*ot a*DA*? No. The name of the language and some of the primitive was inspired by the 'actress' ***Linda*** *Lovelace* with no relation to *Ada Lovelace*, Byron's daughter and "founder of scientific computing".

Linda has only four atomic primitive operations:

- **`in(t)`** find a tuple matching **`t`**, remove it from the space, perform any bindings of formal to actuals
- **`out(t)`** output tuple **`t`** into the space
- **`rd(t)`** non destructive version of **`in`**
- **`eval(P)`** start a new parallel task to evaluate **`P`**

Plus two non blocking variants **`inp`** and **`rdp`**

Linda

Access to the tuple space is termed *associative* because tuples are retrieved using a simple but powerful form of pattern matching

A ***tuple*** is a sequence of values (*actuals*) of type int, bool, char, string, ...

```
out("string", 42, true)
```

A ***template*** is a tuple where some fields are unbound variables (*formals*)

```
in("string", ?x, ?b)
```

Matching a tuple to a template succeeds if the tuples have the same length, the actuals of the template are equal to the corresponding actuals in the tuple

The result of a successful match binds the formals in the template, e.g.

```
out("string", 42, true);
in ("string", ?x, ?b  );
if ( x == 42 && b)
        print("okay");
```

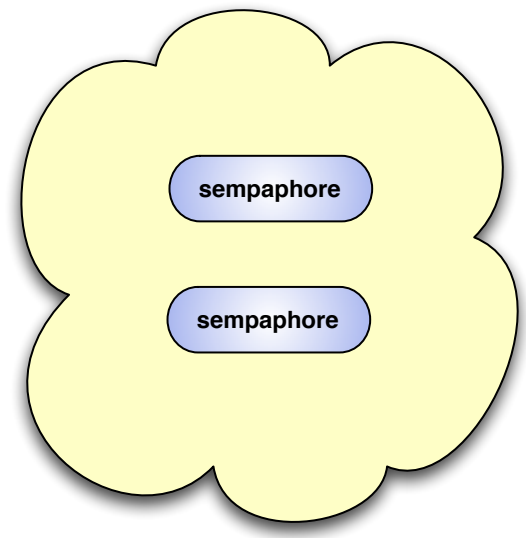
Linda example: sempaphores

A simple counting semaphore can be implemented simply.

Thread 1:

```
out( "semaphore" );

out( "semaphore" );
```



Thread 2:

```
in( "semaphore" );

proceed_to_do_something_meaningful();
```

Linda example: Linked list

```
out( "list", 0, 1,
     "this is the head");

out( "list", 1, -1,
     "this is the end of the list");

next = 0;

while ( next != -1 ) {
   in( "list", next, ?tail, ?val);
   print val;
   next = tail;
}
```

possibly not the best way to construct data structures

Implementing Linda

Obtaining an efficient implementation requires optimizing the storage requirements of tuple, and ensuring fast associative access.

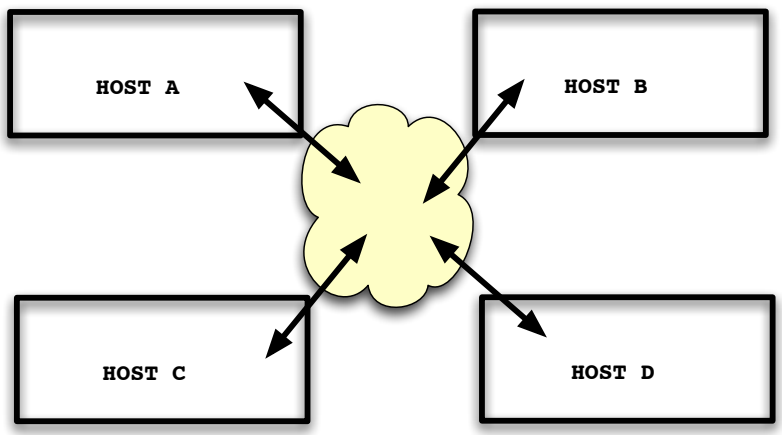
Unlike a database, Linda has no fixed index fields - any combination of fields can be used as a key to retrieve values. Moreover the values of the actuals of a key are typically not know at compile time. Databases are made up of table which can have different implementations, a Linda system is made up of a single tuple space shared by many applications for different purposes.

Parallel Linda can be optimized under *closed world assumption*. E.g. it is possible analyze the program to discover all tuple types and the signature of all templates; different tuple types can be treated differently. The same is not possible for distributed systems.

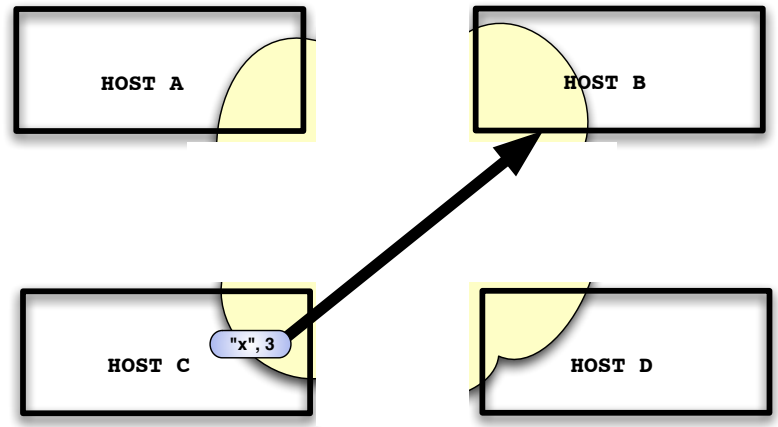
In distributed implementations, the matching is not costly.
Synchronization and communication are slower by orders of magnitude.
Adaptive optimizations such as proactive routing and caching can win big.

Basic Architectures

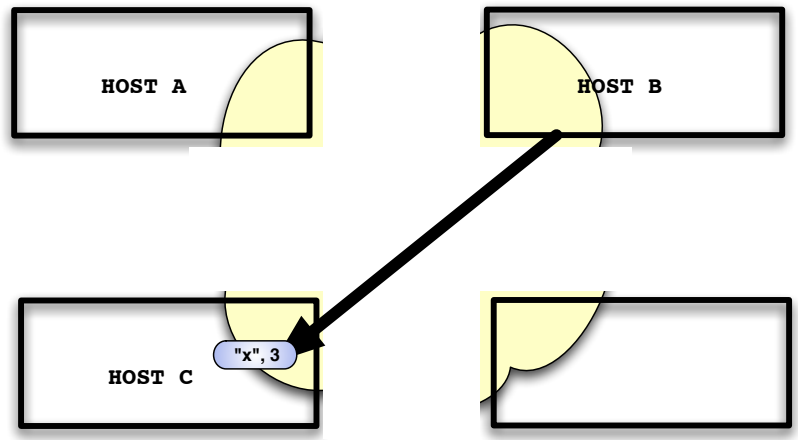
Implementations of the tuple space abstraction have some basic infrastructures. Tradeoffs are related to the choice of location of the tuple space (centralized implies a bottleneck and single point of failure, while distributed means high synchronization costs)



Centralized



Push



Pull

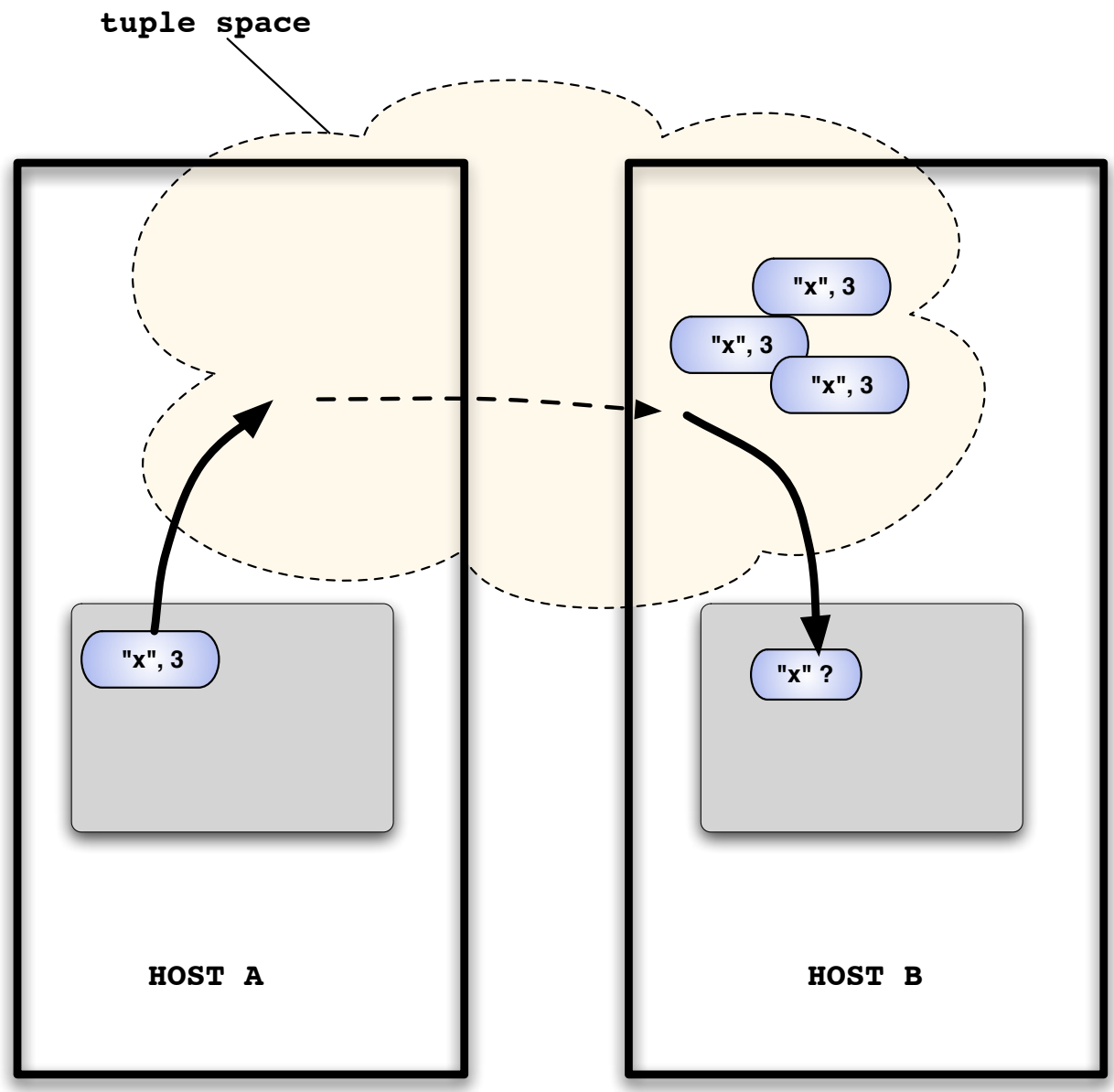
Implementing Linda

Assume a client-provider relationship between processes located on two distinct nodes for tuple of shape (”x”, ?).

A ***proactive routing*** optimization tries to push the tuples to the local store on host B. The system can adapt to changes in communication patterns by discontinuing the routing.

(This is meant to be a semantics preserving optimization, but is it? See the next slide on mem models)

NB: Synchronization is required whenever a destructive operation is performed (in())



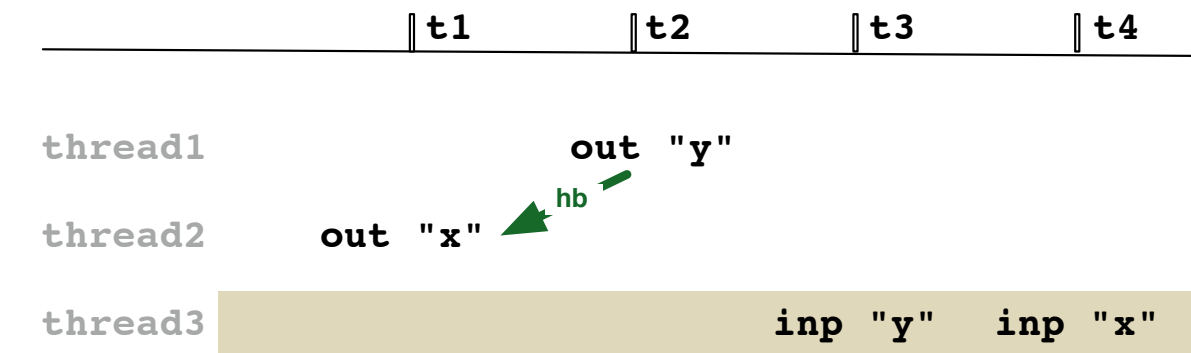
Implementation: Memory Model

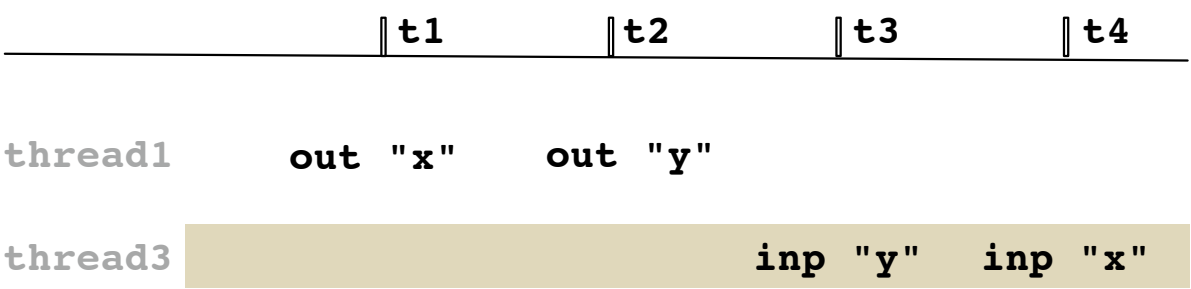
What kind of memory model should be exposed to the programmer? When is the tuple space implementation obligated to
make an operation visible? There is tension between performance and being faithful to the semantics.

Provide some form of ***global sequential consistency***? Then if "y" is visible so must "x". (this prevents any form of "active routing" which breaks this illusion)

Looser consistency (e.g. within a thread)? Then if "y" is visible so must "x". (This only prevents active routing within a thread, but allows it across threads. A reasonable choice. But high-level semantics is breached)

Ideally we want some form a serializability... (as discussed by Jagannathan in transactional linda [COORDINATION04]





Core Linda

Formal semantics will help us gain a better understanding of the design space of coordination language. Many security vulnerabilities are apparent in the semantics!

Core Linda is a simple extension of Milner's polyadic asynchronous pi-calculus with a small-step operational semantics

Difference with Linda are:

- no eval(), it is subsumed by the inherent parallelism in pi
- inp() and rdp() operators are omitted for simplicity; they introduce interesting complication which are out of the scope of this talk

Core Linda

$$
\begin{array}{rcl}
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{v} \mid \langle\, \overline{\mathtt{e}}\, \rangle \mid \mathtt{e}.i \\
\mathtt{v} & ::= & ? \mid \mathtt{x} \\
P & ::= & \mathbf{0} \mid \mathbf{!}\, P \mid P \mid Q \mid \mathbf{in}\, \mathtt{e}\, \mathtt{x}\, \mathbf{.}\, P \mid \mathbf{out}\, \mathtt{e} \mid (\nu\, \mathtt{x})P
\end{array}
$$

Table 1: Core Language Syntax.

$$
\begin{array}{c}
fn(\mathbf{0}) = fn(?) = \{\}, \quad fn(\mathtt{x}) = \{x\}, \quad fn(\langle\, \overline{\mathtt{e}}\, \rangle) = fn(\mathtt{e}_1) \cup \cdots \cup fn(\mathtt{e}'_n) \\
fn(P \mid Q) = fn(P) \cup fn(Q), \quad fn(\mathbf{!}\, P) = fn(P), \\
fn(\mathbf{out}\, \mathtt{e}) = fn(\mathtt{e}), \quad fn(\mathbf{in}\, \mathtt{e}\, \mathtt{x}\, \mathbf{.}\, P) = fn(\mathtt{e}) \cup fn(P) - \{\mathtt{x}\}, \\
fn((\nu\, \mathtt{x})P) = fn(P) - \{\mathtt{x}\}
\end{array}
$$

Table 3: Free names.

Core Linda

Structural Congruence Rules

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P \qquad !\,P \equiv P \mid !\,P$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad (\nu\,\mathtt{x})(\nu\,\mathtt{y})P \equiv (\nu\,\mathtt{y})(\nu\,\mathtt{x})P$$

$$(\nu\,\mathtt{x})(P \mid Q) \equiv P \mid (\nu\,\mathtt{x})Q \quad \textit{if } x \notin \textit{fn}(P)$$

Pattern Matching Rules

$$? \leq \mathtt{v} \qquad \mathtt{v} \leq \mathtt{v} \qquad \frac{\mathtt{v}_1 \leq \mathtt{v}'_1 \;\ldots\; \mathtt{v}_n \leq \mathtt{v}'_n}{\langle\, \overline{\mathtt{v}} \,\rangle \leq \langle\, \overline{\mathtt{v}}' \,\rangle}$$

Core Linda

Reduction

$$\frac{P \to Q}{(\nu\, \mathtt{x})P \to (\nu\, \mathtt{x})Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \frac{P \equiv P' \quad P' \to Q}{P \to Q}$$

$$\frac{\mathtt{e} \downarrow \langle\, \overline{\mathtt{v}}\, \rangle \quad \mathtt{e}' \downarrow \langle\, \overline{\mathtt{v}'}\, \rangle \quad \langle\, \overline{\mathtt{v}'}\, \rangle \leq \langle\, \overline{\mathtt{v}}\, \rangle}{\mathbf{out}\, \mathtt{e} \mid \mathbf{in}\, \mathtt{e}'\, x\, \mathbf{.}\, P \to P\{^{\langle \overline{\mathtt{v}} \rangle}/_{\mathtt{x}}\}}$$

Evaluation

$$\mathtt{v} \downarrow \mathtt{v} \qquad \frac{\mathtt{e}_1 \downarrow \mathtt{v}_1 \;\ldots\; \mathtt{e}_n \downarrow \mathtt{v}_n}{\langle\, \mathtt{e}_1 \ldots \mathtt{e}_n \,\rangle \downarrow \langle\, \mathtt{v}_1 \ldots \mathtt{v}_n \,\rangle} \qquad \frac{\mathtt{e} \downarrow \langle\, \mathtt{v}_1 \ldots \mathtt{v}_i \ldots \mathtt{v}_n \,\rangle}{\mathtt{e}.i \downarrow \mathtt{v}_i}$$

Core Linda: Semaphores

Semaphores can be represented by a tuple of the shape out(). The following gives the reduction sequence of a program with two semaphores.

```
   out() | out() | in()x.out(a) | in(a)x.in()y.0
=  out() | in()x.out(a) | out() | in(a)x.in()y.0
→  out() | out(a) | in(a)x.in()y.0
=  out(a) | in(a)x.in()y.0 | out()
→  out() | in().0
→  0
```

Notice that the state and control are merged; there is no difference between the command out(a) and the tuple (a) in the tuple space.

Core Linda: Lists

A list can be represented by a tuple of the shape out(hd,tl,val). The following gives the reduction sequence of a program with a list of length two.

```
   ((new a) out(0,1,a)) | out(1,0,b) | in(0,?,?)x.in(x.2,?,?)z.0
=
   (new a)( out(0,1,a)  | out(1,0,b) | in(0,?,?)x.in(x.2,?,?)z.0)
=
   (new a)( out(0,1,a)  | in(0,?,?)x.in(x.2,?,?)z.0 | out(1,0,b))
→
   (new a)( in((0,1,a).2,?,?)z.0 | out(1,0,b))
=
   (new a)( in(1,?,?)z.0 | out(1,0,b))
→
   (new a)( [(1,0,b)/z]0 )
=
  (new a)( 0 )
```

Core Linda: Observations

Unlike in Linda, the definition of pattern matching in CoreLinda treats tuples and templates uniformly. Thus we have the following, where a template is output:

```
    out( a,?,1 ) | in( ?,b,1 )x . out( x.1 )

➞   out( (a,?,1).1 )

=   out( a
```

This treatment of templates was adopted in SECOS [Bryce,Vitek02] and simplifies the implementation. Pattern matching is implemented by unification of the two tuples.

Core Linda: Observations

Non blocking operations (inp/rdp) can be specified by adding rules that fire in the absence of matching tuples.

These operations introduce interesting complications as it is possible to observe the absence of a value.

Without non-blocking operations, the discussion of memory model would be irrelevant.

Core Linda: Semaphores

Semaphores can be represented by a tuple of the shape out(). The following gives the reduction sequence of a program with two semaphores.

```
   out() | out() | in()x.out(a) | in(a)x.in()y.0
=  out() | in()x.out(a) | out() | in(a)x.in()y.0
→  out() | out(a) | in(a)x.in()y.0
=  out(a) | in(a)x.in()y.0 | out()
→  out() | in().0
→  0
```

Deja vu, all over again. Consider using this implementation of semaphores as component all over a large system. Problem? (hint: how can we have two distinct semaphores)

Interference

The shared tuple space's main advantage is that independent applications can coordinate. The problem is that it is very easy for applications to interfere with one another by accident (or malice).

The semaphore is a prototypical example where there is a need for controlled sharing.

Assume to different components want to uses semaphores independently. With the current definition of semaphores, they would modify each other's semaphores.

The solution is to add information to differentiate them. (one extra level of indirection)

```
((new a) out(semaphore, a) |  P )  | ((new b) out(semaphore, b) |  P )
```

This only helps a bit, what if a third process does

```
in(?,?)x.P
```

this results in random corruption of the state space of unrelated applications

Labeled Linda

Labeled Linda is designed to address the problem of interference by allowing users to partition the tuple space without loosing the desirable sharing properties.

SECOS implements most of the semantics of Labeled Linda.

In Labeled Linda a ***tuple*** is a map from labels to values. *Labels* can be names or integers. *Values* include tuples, labels and the distinguished ?.

The lexical order of appearance of labels is irrelevant.

Tuples with disjoint label sets can be composed into a new tuple that is the union of the two original ones.

Positional notation can be encoded by using integer labels, a Core Linda tuple **`("a", 1, true)`** can be encoded as **`(1:"a", 2:1, 3:true)`**

Labeled Linda

$$
\begin{array}{l}
\mathtt{e} ::= \mathtt{x} \mid \mathtt{v} \mid \langle \overline{\mathtt{e}} : \overline{\mathtt{e}} \rangle \mid \mathtt{e.e} \mid \mathtt{e} \oplus \mathtt{e} \\
\mathtt{v} ::= ? \mid \mathtt{l} \mid \langle \overline{\mathtt{l}} : \overline{\mathtt{v}} \rangle \\
\mathtt{l} ::= \mathtt{x} \mid i
\end{array}
$$

Table 4: Core Language Syntax.

Labeled Linda

Reduction

$$\frac{\mathtt{e} \downarrow \mathtt{v} \quad \mathtt{e}' \downarrow \mathtt{v}' \quad \mathtt{v}' \leq \mathtt{v}}{\mathbf{out}\,\mathtt{e} \mid \mathbf{in}\,\mathtt{e}'\,x\,.\,P \rightarrow P\{{}^{\mathtt{v}}/_{\mathtt{x}}\}}$$

Evaluation

$$\mathtt{v} \downarrow \mathtt{v} \qquad \frac{\mathtt{e}_1 : \mathtt{e}'_1 \downarrow \mathtt{l}_1 : \mathtt{v}_1 \;\ldots\; \mathtt{e}_n : \mathtt{e}'_n \downarrow \mathtt{l}_n : \mathtt{v}_n}{\langle\, \mathtt{e}_1 : \mathtt{e}'_1 \ldots \mathtt{e}_n : \mathtt{e}'_n \,\rangle \downarrow \langle\, \mathtt{l}_1 : \mathtt{v}_1 \ldots \mathtt{l}_n : \mathtt{v}_n \,\rangle}$$

$$\frac{\mathtt{e} \downarrow \langle\, \overline{\mathtt{l}} : \overline{\mathtt{v}}\; \mathtt{l} : \mathtt{v} \,\rangle}{\mathtt{e} \downarrow \langle\, \mathtt{l} : \mathtt{v}\; \overline{\mathtt{l}} : \overline{\mathtt{v}} \,\rangle} \qquad \frac{\mathtt{e} \downarrow \langle\, \mathtt{l} : \mathtt{v}\; \overline{\mathtt{l}} : \overline{\mathtt{v}} \,\rangle \quad \mathtt{e}' \downarrow \mathtt{l}}{\mathtt{e}.\mathtt{e}' \downarrow \mathtt{v}}$$

$$\frac{\mathtt{e} \downarrow \langle\, \overline{\mathtt{l}} : \overline{\mathtt{v}} \,\rangle \quad \mathtt{e}' \downarrow \langle\, \overline{\mathtt{l}}' : \overline{\mathtt{v}}' \,\rangle \quad \overline{\mathtt{l}} \cap \overline{\mathtt{l}}' = \emptyset}{\mathtt{e} \oplus \mathtt{e}' \downarrow \langle\, \overline{\mathtt{l}} : \overline{\mathtt{v}}\; \overline{\mathtt{l}}' : \overline{\mathtt{v}}' \,\rangle}$$

Pattern Matching Rules

$$? \leq \mathtt{x} \qquad \mathtt{v} \leq \mathtt{v} \qquad \frac{\mathtt{v}_1 \leq \mathtt{v}'_1 \;\ldots\; \mathtt{v}_n \leq \mathtt{v}'_n}{\langle\, \overline{\mathtt{l}} : \overline{\mathtt{v}} \,\rangle \leq \langle\, \overline{\mathtt{l}} : \overline{\mathtt{v}}' \,\rangle}$$

Labeled Linda: Semaphores

Semaphores can be represented by a tuple of the shape out(sem:a) where a is the name of the component owning the semaphore. The following gives the reduction sequence of a program with two components.

```
(new a)(out(sem:a) | in(sem:a)x.out(42)) | (new b)(out(sem:b)|P)
```

→

```
(new a)(out(42)) | (new b)(out(sem:b)|P)
```

Strict Match

Even with labels it is still possible to write **`in(sem:?)x.P`** which defeats the purpose of Labeled Linda.

Extend Label linda with a ***strict match*** constructor #, such that
l#v < l'#v' only if l = l' and v = v'

With the strict match we have the guarantee that if we write

`out(sem#a) | in(sem: ?)x.out(42)`

the term will not be reduced further.

The syntax and semantic extensions are omitted (small changes to the definition of match)

Multiple tuple spaces

JavaSpaces, and most distributed implementations of Linda allow for multiple distinct tuple spaces.

Multiple tuple spaces have advantages as they can improve locality (e.g. it is more efficient for two colocated processes to communicate over a shared tuple space located on the same host rather than have to go on the network.)

Multiple tuples limit the potential for unwanted interferences between applications.

Locations can be expressed with labels:

```
out( loc# here, 1: 59, ...)
```

Nested tuple spaces simply use several locations fields:

```
out( loc# here, subloc# x, 1: 59, ...)
```

Creation of a tuple space in Labeled Linda boils down to creating a fresh name **((new a)out(loc# a))**

Example: Reactive Programming

Suppose that you want to ensure that every tuple which is emitted has a default location appended. Can we express this in Labeled Linda?

```
! ( in( 1:? )x. out x+(loc#here) )
```

What about ensuring that all input request be located?

Reactive Linda

Reactive Linda allows to specify actions that are performed *as soon* and *every time* a particular tuple is inserted in the space.

Reactive capacities are part of GigaSpaces, JavaSpaces, Lime, etc.

A ***reaction*** is a process which is registered with a template with a guarantee that the process is run when the tuple is inserted in the space.

Some design choices:

- reaction allow modifications of the tuple (i.e. interposition)
- reactions are atomic
- reactions are executed once / multiple times
- reactions can be chained
- reactions can be registered for all tuple space operations

Reactions reify the tuple space, same as Aspects.

Reactions generalize the concept of event handlers.

Reactive Linda

Reactive Linda is a simplified reactive langauge. Reactions can be registered only on output operations.

Implemented with the following convention. Two virtual tuple spaces are used. All input operations operate over tuples label with (r:r).

Reactions operate over the unadorned space.

A reaction is triggered whenever an unadorned tuple is inserted in the space. It consumes the tuple and can output it as a labeled tuple (or not).

$$P ::= \ldots \mid \mathbf{react}\ \mathtt{e}\ \mathtt{x}\ \mathbf{.}\ P$$

Table 6: Core Language Syntax.

Reactive Linda

Reduction

$$\frac{\mathtt{e} \downarrow \mathtt{v} \quad \mathtt{e}' \downarrow \mathtt{v}' \quad \mathtt{v}' \leq \mathtt{v} \quad \mathtt{r} \notin \mathit{labels}(\mathtt{v})}{\mathbf{react}\,\mathtt{e}\,\mathtt{x}\,\mathbf{.}\,P \mid \mathbf{out}\,\mathtt{e}' \rightarrow P\{{}^{\mathtt{v}}/_{\mathtt{x}}\}} \qquad \frac{\mathtt{e} \downarrow \mathtt{v} \quad \mathtt{e}' \downarrow \mathtt{v}' \quad \mathtt{v}' \oplus \langle\, \mathtt{r} : \mathtt{r} \,\rangle \leq \mathtt{v}}{\mathbf{out}\,\mathtt{e} \mid \mathbf{in}\,\mathtt{e}'\,x\,\mathbf{.}\,P \rightarrow P\{{}^{\mathtt{v}}/_{\mathtt{x}}\}}$$

Table 7: Operational Semantics.

Reactive Linda: Observations

```
out( 1:a ) | react( 1:?)x.out x+(r#r) | in( 1:? )z.P
```

→

```
out (1:a)+(r#r) | in( 1:? )z.P
```

→

```
[(1:a,r#r)/z]P
```

Note that this is a cooperative model, anyone can output a tuple adorned wit (r#r).

Not really usable because there is no chaining and no defaulting

Also not that reactions are not atomic, thus there can be out of order delivery of out()s

NB: reactive Linda falls short of a full reactive system; there is no major difficulty in extending the semantics to support, e.g. Lime reactions.

Distribution

Labeled Linda is not terribly convenient to express distributed process.

In a distributed setting, processes have locations

Design choice: location *awareness* vs. location *transparency*

(old debate in distributed systems: everyone wants transparency, but no one wants to pay for it)

Located Linda

Located Linda is a distributed language where processes and tuples have locations

The semantics enforces multiple disjoint tuple spaces

Changes to Located Linda include a new syntactic category for named location executing processes, and tuples

Top level tuples are adorned with a location label

in() and out() are extended with location annotations

$$L ::= \mathbf{0} \mid L \mid L \mid \mathtt{a}[P] \mid \mathtt{v}$$

$$P ::= \ldots \mid \mathbf{in}\ \mathtt{e@e}\ \mathtt{x}\ \mathbf{.}\ P \mid \mathbf{out}\ \mathtt{e@e}$$

Table 8: Core Language Syntax.

Located Linda

Reduction

$$\frac{P \to P'}{a[P] \mid L \to a[P'] \mid L}$$

$$\frac{\begin{array}{c} L \equiv a[\,\mathbf{in}\,\mathtt{e}@\mathtt{e}x\,.\,P \mid P'\,] \mid v \oplus \langle\,\mathtt{loc} : \mathtt{b}\,\rangle \mid L' \\ \mathtt{e} \downarrow \mathtt{v}' \qquad \mathtt{e}' \downarrow b \qquad \mathtt{v}' \leq \mathtt{v} \\ L'' = a[\,P\{^{\langle \overline{\mathtt{v}} \rangle}/_{\mathtt{x}}\} \mid P'\,] \mid L' \end{array}}{L \to L''}$$

$$\frac{\mathtt{e} \downarrow \mathtt{v} \quad \mathtt{e}' \downarrow \mathtt{b} \qquad L' = v \oplus \langle\,\mathtt{loc} : \mathtt{b}\,\rangle \mid L}{a[\,\mathbf{out}\,\mathtt{e}@\mathtt{e}' \mid P\,] \mid L \to a[\,P\,] \mid L'}$$

Table 9: Operational Semantics.

Summary: JavaSpace

JavaSpace is the SUN implementation of Linda which has been incorporated in the JINI infrastructure; GigaSpace and TSpaces (IBM) are commercial products with similar features

JavaSpaces has:

- locations and multiple tuple spaces
- concurrency is provided by the host language (e.g. no eval)
- reactions
- transactions (did not discuss it here see Busi et.al, and Jagannathan)

Mobile Linda

Mobility allows to express computations that adapt to locality

Better usage of resources as the computation can be co-located to the resource, where ever the resource happens to be

Inspired by Ambients [CardelliGordon] and the work on Mobile Agents

Mobile Linda is an extension of Located Linda in which agents have names and execute on a single location

Explicit move operation

$$L ::= \ldots \quad | \quad \mathtt{a}_{\mathtt{h}}[P]$$

$$P ::= \ldots \quad | \, \mathbf{move}\, \mathtt{a}$$

Table 10: Core Language Syntax.

Mobile Linda

Reduction

$$\frac{L \equiv \mathtt{a_h}[\,\mathbf{in}\,\mathtt{e}@\mathtt{e}x\,.\,P \mid P'\,] \mid v \oplus \langle\,\mathtt{loc}:\mathtt{b}\ \mathtt{own}:\mathtt{g}\,\rangle \mid L' \qquad \mathtt{e} \downarrow \mathtt{v}' \qquad \mathtt{e}' \downarrow b \qquad \mathtt{v}' \leq \mathtt{v} \qquad L'' = \mathtt{a_h}[\,P\{^{\langle \overline{\mathtt{v}} \rangle}/_{\mathtt{x}}\} \mid P'\,] \mid L'}{L \rightarrow L''}$$

$$\frac{\mathtt{e} \downarrow \mathtt{v} \quad \mathtt{e}' \downarrow \mathtt{b} \qquad L' = v \oplus \langle\,\mathtt{loc}:\mathtt{b}\ \mathtt{own}:\mathtt{h}\,\rangle \mid L}{\mathtt{a_h}[\,\mathbf{out}\,\mathtt{e}@\mathtt{e}' \mid P\,] \mid L \rightarrow \mathtt{a_h}[\,P\,] \mid L'}$$

$$\frac{L' = L\{^{\mathtt{v} \oplus \langle\,\mathtt{loc:b\ own:h}\,\rangle}/_{\mathtt{v} \oplus \langle\,\mathtt{loc:a\ own:h}\,\rangle}\}}{\mathtt{a_h}[\,\mathbf{move}\,\mathtt{b} \mid P\,] \mid L \rightarrow \mathtt{b_h}[\,P\,] \mid L'}$$

Table 11: Operational Semantics.

Roaming Consumer

```
a_h[ in(1:?)x.(move j|in(1:?)y) ] | (1:1,loc#h) | (1:2,loc#j)
```

→

```
a_h[ move j | in(1:?)y ] | (1:2,loc#j)
```

→

```
a_j[ in(1:?)y.0 ] | (1:2,loc#j)
```

→

```
a_j[ 0 ]
```

Summary: Lime

Lime was developed by Picco, Murphy and Roman at WUSTL

It is a Java-based mobile agent/coordination language

Lime has:

- locations and mobility
- global and local tuple spaces
- atomic reaction
- engagement/disengagement

- CoreLime [Carbunar,Valent,Vitek] a simplified semantics

Part II
SECURITY

Threats

Threats can be categorized with respect to their target and their initiator.

Malicious Users

Attacker is a process with access to the tuple space, targets are other processes and the coordination middleware itself

Secrecy & Integrity: subvert the coordination language to obtain secrets or damage other computations

Quality of Service: affect non-functional characteristics of the coordination middleware in order to degrade performance

Network

Secrecy & Integrity: traditional distributed system issues which can be addressed with cryptographic techniques; not discussed further as they not specific to coordination languages

Malicious Host

Attacker is the environment, either at the level of the machine, operating system, runtime system or coordination middleware; targets are the programs that are sent to execute on the compromised environment

Secrecy

Secrecy can be compromised by a process accessing another's process data

Basically any input primitive can be used:

`rd(?,?)x.P` lets an attacker non-deterministically poke in the state of the tuple space, and retrieve data that does not belong to it

`in( ? )x.P` similar to rd, except the data is removed from the space

`rdp( ? )x.P` similar to rd, and lets the attacker observe the absence of a datum, and witness intermediate steps in a protocol

`react( ? )x.P` stronger than rd/in because it guarantees that all tuple will be witness

```
out(x) | in(?)y.P | in(?)y.Attack

out(x) | in(?)y.P | react(?)y.Attack
```

In the first line, the attacker is picked non-deterministically, while in the second the reaction is always triggered

Integrity

The integrity of a computation can be attacked with any operation that modifies the tuple space

`in(sem:a)x.P` can be used to remove a datum, here a semaphore, used by another computation

`out(sem:a)` can be used to corrupt another computation's state, here by misleading it about the availability of a resource

`in(2,1)x.out(2,3)` can be used to modify data in arbitrary ways; note however that due to non-determinism the attacker is not guaranteed to succeed in grabbing the tuple, and that the attack can be observed by the victim with rdp(2,?), as the non-blocking read may observed the disappearance of (2,1).

`transact[in(2,1)x.out(2,3)]` the transactional facilities of JavaSpaces can be used to ensure that victim can not observe the intermediary state during the attack

`react(2,1)x.out(2,3)` ensure that the attacker will get access to the tuple; depending on the semantics of react this may happen atomically

Quality of Service

Denial of Service Attacks:

```
while ( true )
   out( i++ )
```

A runaway task may easily fill the tuple space with garbage. An associative communication model has no provisions for garbage collection of tuples, because, it is, in general, not possible to determine that a tuple is not "reachable".

JavaSpaces supports the specification of tuple "lifetimes" to prevent tasks from accidentally generating garbage.

Possible Solutions:

- Resource limits – these may limit legitimate applications and do not provide strong guarantees against a number of tasks colluding to fill up the space.
- Mandatory tuple expiration – complicates the programming model as programmers must worry about relative timings of **`in`**s and **`out`**s.
- Static space analysis – impose restrictions on untrusted code via types and or static analysis.

Quality of Service

Implementation Knowledge:

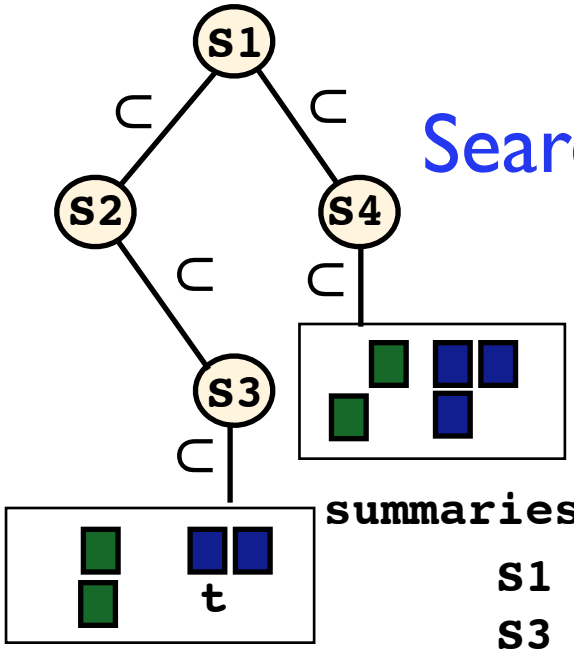
Denial of service attacks can be mounted with knowledge of the implementation of the tuple space.

Case Study: SECOS [Bryce,Vitek02] an implementation of Labeled Linda

Tuples are sequences of label - value pairs, `(l:v,...,l':v')`. A *fingerprint* is a `(K,V)` is a 128 bit vector such that for tuples `t` and `t'` we have $\texttt{fp(t)} \not\subset \texttt{fp(t')} \Rightarrow \texttt{t} \leq \texttt{t}'$. A *summary* is a 16 bit vector computed as the union of K and V.

Associative search is performed over a lazily computed binary search tree, leaves contain tuples and templates and are searched exhaustively in FIFO order.

Quality of Service



Search Tree

summaries:

S1 ⊂ S2, ...

S3 ⊆ sum(t)...

computing fingerprints:

```
for (int j = 0; j < keys.length; j++) {
    keysFP = keysFP | (1 << (keys[j].hashCode() %64
    if ( val[j] != null)
        valsFP = valsFP | (1 << (val[j].hashCode()
}
```

computing summaries:

```
short summary(long vfp, long kfp) {
    short sum = 0;
    for (int i = 0, j = 0; i < 64; i += 4, j++)
        sum = sum | (((7<<i)& vfp) << j) |
                    (((7<<i)& kfp) << j);
    return sum;
}
```

Exploits include:

large numbers of different fingerprints to artificially increase the size of the tree; degenerate cases will search times due to implementation restrictions

large numbers of identical fingerprints and issuing queries that perform exhaustive searches on buckets, blocking legitimate queries due to synchronization

Information Flow: knowledge of lookup algorithm allows timing information leaks

Quality of Service: React

In Lime and JavaSpaces reactions are performed atomically as part of the procedure of inserting a tuple in the space.

A non-terminating reaction is an example of denial of service attack in which an unprivileged users gets to execute with privileges and consume a finite resource. (Similar attacks can be mounted on Java virtual machines, e.g. with finalizers)

The only good response is to execute reaction in a non-atomic fashion, but thus potentially loosing ordering of output events.

Alternatively, reactions can be restricted to trusted processes.

Quality of Service: Eq

Value comparison is at the heart of the pattern matching process. In many object-oriented Linda languages, arbitrary objects can be compared.

While this is desirable from the point of view of expressiveness, the question the security issue is how does one compare them.

Using equals() and hashCode() amounts to running user code within the kernel and is subject to denial of service attacks.

Alternatives:

- use built trusted in functions such as identityHashCode() does not work as they are collision prone.

- replace equals() with comparison of UIDs as in GigaSpaces where a UID is a value provided by the user and computed outside of the kernel. This runs the risk of collisions.

- restrict matching to primitive values.

Threats: Summary

To summarize:

secrecy and integrity attacks can be mounted by taking advantage of the open nature of tuple space communication; these threats are obvious in the semantics and require changes to the programming model

quality of service attacks can be mounted by misusing particular implementation details, these threats are typically not apparent in the semantics; to address them one needs to be able to

- bound resource usage of processes

- ensure that untrusted code is not run with kernel privileges

- avoid optimizations that are based on some expected characteristics of computations if these can be used to degrade system performance

Capabilities

Discretionary access control (DAC) mechanisms restrict a subject's access to an object. It is generally used to limit a user's access to resources such as files. In this type of access control it is the owner of the file who controls other users' accesses to the resource.

In coordination language using capabilities for DAC is quite natural and has been explored by many systems (JavaSpace, SECOS, Klaim,...).

A capability is a software token describing the rights of its owner; any entity in possession of a given capability may exercise that capability.

Capabilities are managed by a reference monitor which enforces the associated access rights.

Assumptions:

- the reference manager is part of the trusted base, thus we require trust in the middleware, operating system and hardware.

- If capabilities are allowed to span hosts on the network the reference manager must either use encryption or assume a trusted network.

Design Space (1/2)

1) Explicit v. implicit policies

An explicit policy is expressed declaratively outside of the program, while an implicit policy is part of the program code itself.

Explicit policies are easier to reason about, but typically limited by the expressiveness of the policy language (for example a type system), implicit policies are turing complete (they are embodied by the code) but much harder to validate.

2) Capability transfer

Can capabilities be given away or copied? (e.g. nothing prevents a user from revealing a password)

Transfer is needed to implement many applications, e.g. a user level load balancing task manager which needs to give appropriate access to the tasks.

Transfer makes reasoning about the overall system mode difficult.

Design Space (2/2)

3) Capability revocation.

Can the access rights associated to a capability be revoked, if yes can it happen while a task is exercising that capability?

Revocation requires tracking of capabilities which can be cause added runtime overhead.

4) Static v. dynamic reference monitors

A dynamic reference monitor will flag access violation at runtime, while a static monitor will prevent the release of a task that can potentially violate the security policy

Static checking is preferable and there is no runtime cost and errors are caught earlier, but it is limited by the expressive power of the checker (types or static analysis), dynamic checks are less restrictive (i.e. more correct programs are allowed under dynamic checks than static checks)

Coarse-grained Capabilities

Coarse-grained Capabilities (CgC) control access to a tuple space via password or digital signatures. The infrastructure grants access to a program by releasing a reference to a tuple space object.

With a CgC a program gains full access to the tuple space. No further restriction can be applied.

Revocation can be implemented by deactivating a CgC if it has an identity.

Transfer is usually not allowed - most implementation do not allow CgCs to be transmitted on the network thus eschewing the need for trusted network communication.

- JavaSpaces, GigaSpaces

Coarse-grained Capabilities

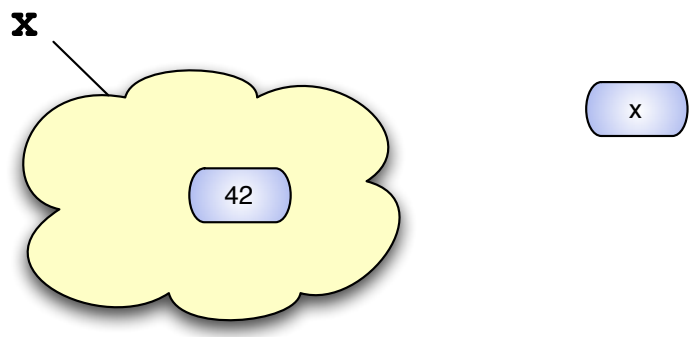
Labeled Linda supports CgC in a straightforward and natural way:

```
((new x) ( out( 1:x ) | out( loc#x , 2:42 ) )) |
```

defines a new tuple space named x and outputs 42 to it.

```
in (1:?)v. in(loc#(v.1), 1:?)v'.P
```

reads a one elem tuple, use first field as tuple space name



Reductions:

```
((new x)(out(1:x)|out(loc#x,2:42)))| in(1:?)v.in(loc#(v.1),1:?)v'.P
```

➞ `(new x)(out(loc#x,2:42))| in(loc#((1:x).1),1:?)v'.P)`

➞ `(new x)([(loc#x,2:42)/v']P)`

Coarse-grained Capabilities

CgCs in JavaSpaces are implemented by object references, the following

```
JavaSpaces jspace = SpaceFinder.find("jini://"+host+"/Js");
```

returns a reference to a named space.

The **`jspace`** variable acts as a capability and the Java VM is the reference monitor.

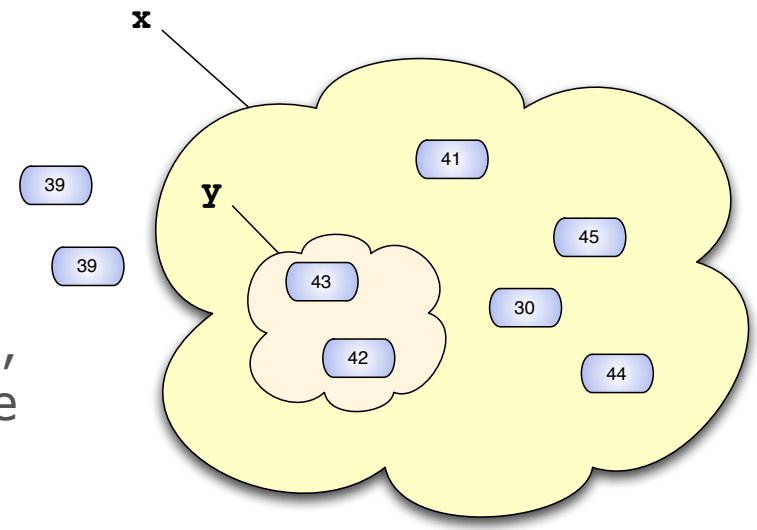
Note however that in above example (just as in Labeled Linda), there are no restrictions on capability transfer and revocation of individual capabilities is not possible.

Coarse-grained Capabilities

CgCs can be viewed as partitioning the tuple space.

Structured CgCs provide nested partitions for finer granularity control. In Labeled Linda this is simply achieved by multiple location fields, e.g.

```
out(loc#x, 1:41),
out(loc#x, subloc#y, 1:42),...
```



The labels **loc** and **subloc** are user defined, any label can be used thus allowing multiple levels of nesting.

Asymmetric CgC

Labeled Linda CgCs grant full access to a tuple space to all users.

SECOS introduced "co-names" as a simple way to get more control on the use of spaces.

The expression (new a,b) creates two fresh names such that, when used as labels, a < b and b < a, but but b (resp. a) does not match b (resp. a).

Thus it is possible to mimic asymmetric crypto with:

```
((new a,b) out(1:a) | out(b:2) | out(a:1))

                                  | in(1:?)x.in( (x.1) : ?)
```

Similarly, we can use co-names to implement read/write restricted tuple space. Given (new a,b), if we pick a to be the name of the tuple space, then a is the capability needed for writing and b is the one needed for reading the new tuple space

Crypto Linda

Using cryptographic operation in a coordination middleware allows to protect data from inspection while in transfer. CryptoKlava and SecureSpaces provide support for encrypted tuples. CryptoKlava allows:

- encrypt tuple fields with specific keys
- use variants of the operations in and read (ink and readk) to atomically retrieve a tuple and decrypt its contents.

The modified versions of the retrieving operations, ink and readk, are based on the following procedure:

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: (a) if the operation was an ink then put the retrieved tuple back in the tuple space, (b) look for alternative matching tuples
4. if all these attempts fail, then block until another matching tuple is available.

Basically the original Linda pattern matching mechanism is not modified: encrypted fields are seen as ordinary fields that have type KCipher. For efficiency reasons, could one use something like the SECOS fingerprints to avoid spurious decryption attempts? (what about info leakage). In a mobile agent context, agents should not roam the network with keys (cf. “clueless agents”).

Fine grained Capabilities: Roles

A role is a capability that grants restricted access to a tuple space.

Multiple distinct roles can be created for the same tuple space.

Roles can be revoked.

```
class Role_IR implement In, Rd {
   JavaSpace space;
   Role_IR(JavaSpace s ) {...}
   Object in(Pattern p) {...}
   Object rd(Pattern p) {...}
   void revoke() { space = null; }
}
```

Fine grained Capabilities: Roles w. Filters

A fine grained access mechanism could extend the notion of roles with filters.

A filter pair (t,f) such that t is a template and f is a function that take a tuple and returns a tuple.

The semantics of Role is that before performing any of the allowed operations, all filters will be executed. The value delivered is that returned by the filter

```
InFilter in = (”list”, ?, ?, ?),
              fun(s,i,j,val) = if val=“foo” then (s,i,j,“fum”)
                                            else (s,i,j,val)
Role_I list_role = new Role_IT( space, in )
...
list_role.in(?, 0, ?, ?);
```

Types as Capabilities

Klaim supports agent mobility; there is a need to prevent agents from moving to a host and misusing the local resources

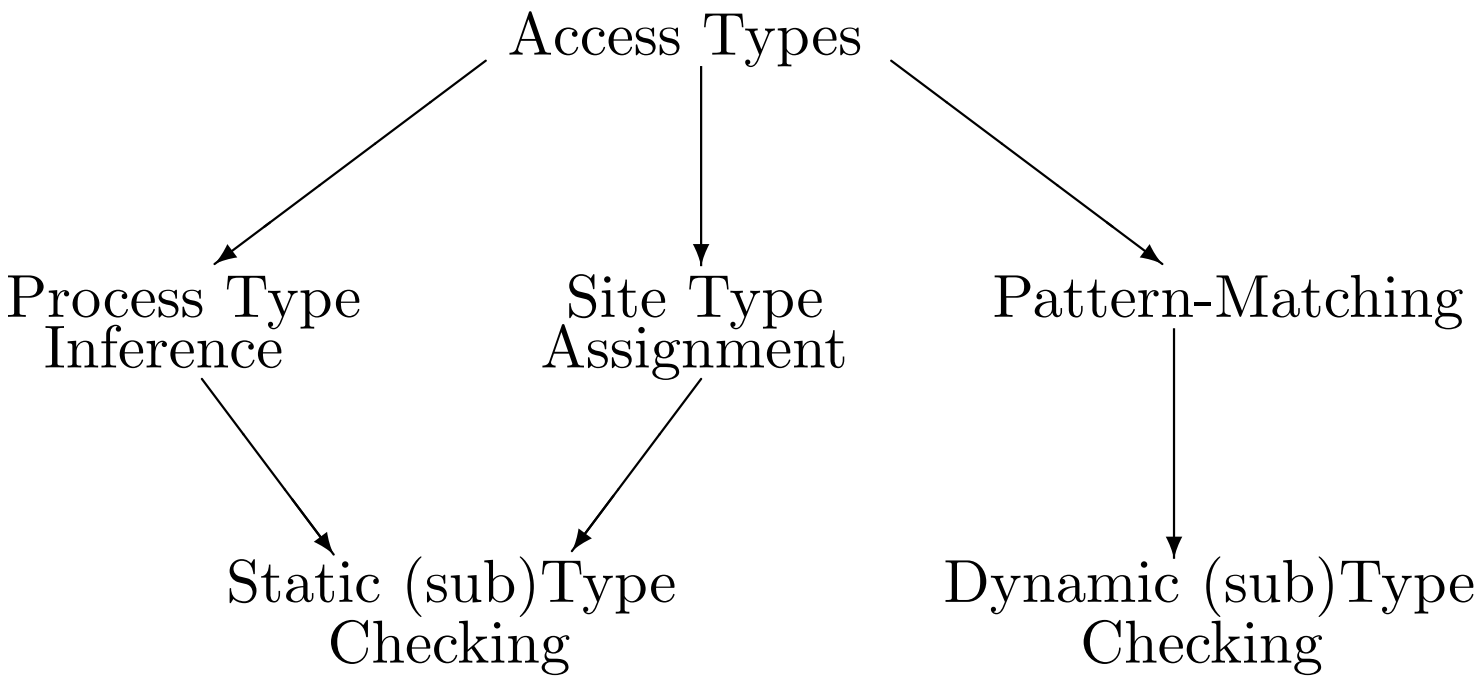
SecureKlaim is an example of a coordination language with a mixture of static and dynamic reference monitor.

Agents are statically typed, which has the advantage of making security policies explicit and catching errors early.

The main drawback of the approach is the expressiveness of the type system

(following slides from a Klaim presentation)

Klaim Overview

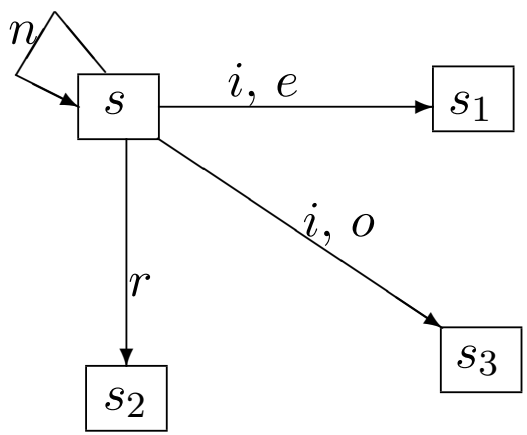


Klaim: Example

- The access policy specification δ_s of node s

$$\begin{aligned} \delta_s \quad = \quad & s : \{n\} \mapsto \bot, \\ & s1 : \{i, e\} \mapsto \delta_{\mathtt{s_1}}, \\ & s2 : \{r\} \mapsto \bot, \\ & s3 : \{i, o\} \mapsto \bot \end{aligned}$$

A graphical interpretation: *access types are graphs*



- *Capabilities:*

r stands for **read**

i stands for **in**

o stands for **out**

e stands for **eval**

n stands for **newloc**

Klaim

Actions $a ::= \mathbf{out}(t)@\ell \mid \mathbf{in}(t)@\ell \mid \mathbf{read}(t)@\ell$
$\mid \mathbf{eval}(P)@\ell \mid \mathbf{newloc}(u : \widetilde{\langle \lambda, \delta \rangle})$

Types

$$
\begin{array}{rcll}
\delta & ::= & \bot & (\textit{empty type}) \\
& \mid & \top & (\textit{universal type}) \\
& \mid & \ell : \pi \mapsto \delta & (\textit{locality-labelled arrow type}) \\
& \mid & \delta_1, \delta_2 & (\textit{union type}) \\
& \mid & \nu & (\textit{type variable}) \\
& \mid & \mu\nu.\delta & (\textit{recursive type})
\end{array}
$$

$\pi \subseteq \{r, i, o, e, n\} \quad (\pi \neq \emptyset)$ set of *capabilities*

Information Flow

- Mandatory access control tries to enforce constraint on the flow of information
- Programming language-based information techniques based on type systems have been investigated for Java, different of the pi-calculus have been studied for info flow in a concurrent language
- Coordination Languages have two specific problems over and above known issues in the pi-calculus (implicit flows, timing channels, etc)
  - rdp/inp allow low level programs to observe the absence of values; pi does not have equivalent operations
  - type systems for associative matching remain an open issue. Consider the following:

    in(y,?)x:T.P

  should the type of x be high or low? If we have

    out(12, h) : H | out(11, l) : L

  then it could be either depending on the value of "y". In general to be safe one should use the least upper bound of all two tuple output statements. But this will quickly push everything to high.

Conclusions

- Linda is a powerful coordination language; it's power comes from a flexible associative access to memory which enables independently developed applications to communicate easily
- Security, in the presence of untrusted processes, is endangered by the very nature of coordination languages
- Most approaches to secure tuple spaces are based on restrictions to the associative access primitives; capabilities can be used to restrict the set of primitives allowed to a process as well as the portion of the space that is addressable
- Static type systems show promise for explicit and verifiable security policies, the expressiveness of existing system must be extended to allow checking of protocols rather than simple access