Lecture 1:
An introduction to coordination

Contents

- Background and motivation
- What is a coordination language?
- Coordination mechanisms in Linda and derived languages

Specific references

Carriero, Gelernter "Coordination Languages and their Significance", CACM 35:2, 1992.
Carriero, Gelernter, *How to Write Parallel programs. A first course*, MIT Press, 1990
Carriero, *Implementing Tuple Space Machines*, PhD Thesis, Yale 1987

www.lindaspaces.com

Beyond sequential programming

Programming languages theory developed two basic paradigms useful for designing operating systems:

- shared data (semaphores, monitors, etc.)
- message passing (rendezvous, RPC, etc.)

They do not cover the whole range of coordination problems

- AI researchers developed a whole series of special-purpose architectures for multi-agent systems (e.g. blackboards, contract-nets, actors, etc.)
- Parallel programmers developed several *organizative techniques* that do not fit exactly in any of the two paradigms (e.g. master-worker, agenda, pipeline, etc.)
- Software engineers in designing distributed software architectures found a whole set of novel *integration* problems not easily solved within the classic paradigms (runtime interoperability, multiparadigm programming, associative invocation of services, dynamic multiclient-multiserver systems, mobile code, etc.)
- The Internet can be enhanced by coordination middleware to a programmable platform offering support for large-scale groupware, agent-based applications, and high-performance computational services (eg. the GRID)
- From a theoretical point of view there are several *properties of coordinated behaviour* (eg. locality, mobility, security, etc.) that are not easily analysable using formal semantics and logic developed for conventional concurrent languages

Coordination

Coordination is a key concept for studying the activities of complex dynamic systems

Coordination is managing dependencies between activities

Such a definition implies that all instances of coordination include agents performing activities that are interdependent [Malone and Crowston 94]

Due to its fundamentality, this notion covers a lot of facets, for instance in distributed artificial intelligence, robotics, biology, and organisational sciences

Here we see coordination from the viewpoint of programming languages and software engineering

Coordination is the process of building programs by gluing together active pieces [Carriero and Gelernter 92]

Active pieces here can mean processes, objects with threads, agents, or whole applications

programming = coordination + computation

Coordination programming

Coordination programming [CarGel90] is “more natural” than sequential programming for applications requiring explicit parallelism

To write a coordinated program:

1. choose the *conceptual class* that is most natural for the problem
2. write a program using the *software architecture* that is most natural for that conceptual class
3. if the resulting program is not acceptably efficient, *transform* it in a more efficient version by switching from a natural architecture to a more efficient one

Conceptual classes [Carriero Gelernter 1990]

- coordination by result
- coordination by specialisation
- coordination by agenda

These classes differ in the starting approach to design a program to solve the problem:

⇒ we can start from the *intended result*,
⇒ or from the *organisation* of computing agents,
⇒ or from the *list of subtasks* to be performed

Example: planning the building of a house
⇒ we can decompose the intended final layout, separately building the components and then putting them together
⇒ we can assign a special task to each available agent, aiming at exploiting each (specialist) agent in parallel given a list of building phases, or
⇒ we can try to parallelize the building process

Coordination by result

The intended result of a program can usually be decomposed in several subresults; all the components of the result can then be processed separately and simultaneously

We can design a parallel application around the data structure yielded as the ultimate result, and we get parallelism by computing simultaneously all the elements of the result

Result coordination focuses on the shape of the finished product: usually it has to be a complex structure whose elements can be computed in parallel

Typical examples:
When the program has to produce structured data and if we can specify precisely how each element of the resulting structure depends on the rest and on the input, then it is a good idea to attempt result parallelism

Examples:
- Given 2 n-element arrays A and B, compute their sum S
- Given 2 matrices M1 and M2, compute their product P
- Sort a list using parallel merge sort

Coordination by specialisation

Each available worker is assigned to perform one specified kind of work, and they all work in parallel (e.g. in pipeline) up to the natural restrictions imposed by the problem

We can plan an application around an ensemble of specialist programs connected into a logical network of some kind; parallelism results form all the nodes of the logical network being active simultaneously

Specialist coordination focuses on the makeup of the work crew (i.e. the "*software architecture*")

Examples:
- A number of servers in an operating system;
- A number of monitor/control processes in a realtime system
- A parallel compiler built as a pipeline of fine-grain tools (eg. scanner, parser, code generator, optimizer)

Coordination by agenda

Each worker is assigned to help out with the current item on the agenda, and they all work in parallel up to the natural restrictions imposed by the problem

We can plan an application around a particular agenda of activities and then assign several workers to each step

Agenda coordination focuses on the list of tasks to be performed; in this case, workers are not specialist: their structure is uniform (they input a task, solve it, and finally output the solution)

Two special cases of agenda coordination:
- Data parallelism (synchronous)
- Speculative parallelism (or-parallelism)

Coordination by agenda involves a series of transformations to be applied to all elements of some set in parallel: typically we have a *master-worker* structure

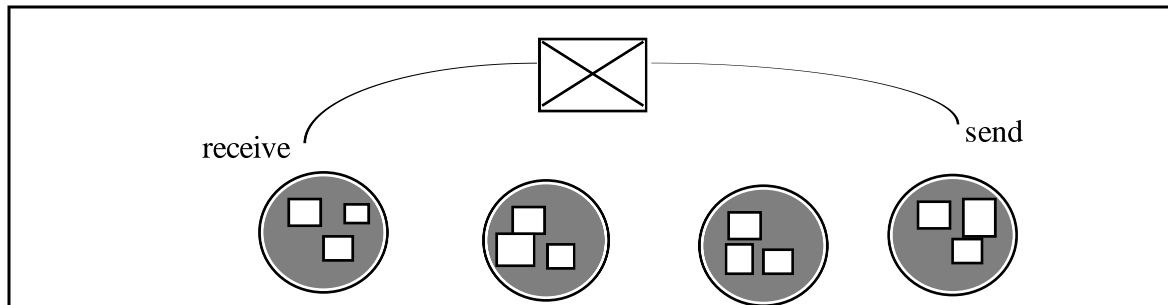
A master process initialises the computation and creates a collection of identical worker processes; each worker is capable of performing any step in the computation; the master waits for solutions computed by workers

Workers repeatedly seek in the agenda a task to perform, get a task, perform the selected task, output the solution, and repeat; when no task remains, the program terminates

Examples:
- A make utility which distributes sources to allow for parallel compilation
- A chess program which searches in parallel the game tree

Message passing

Message passing models allow to coordinate processes that can communicate with other processes through channels or ports on which messages are sent and received



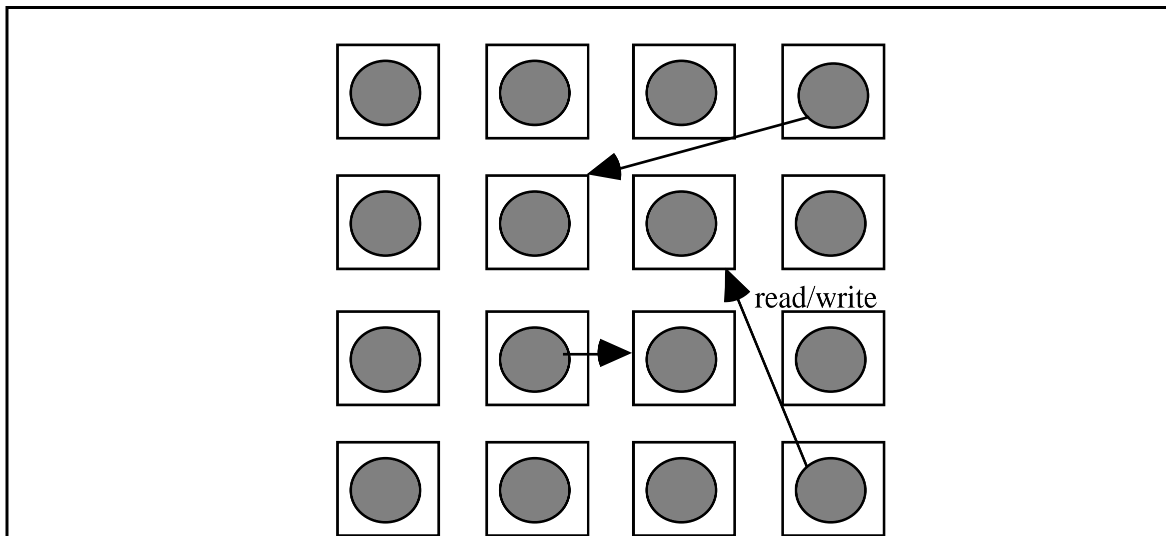
Typical languages of this class are Ada, CSP, Occam, POOL, concurrent logic languages, data flow languages

This programming model is the basis of most operating systems architectures that use the client-server model

It is also the basis of the actor (OO) model of computation

Live data structures

Live data structures coordination models allow to define data structures that contain active threads of computation; the threads can read/write other data structures under the control of other threads using synchronising primitives (e.g. semaphores, monitors, critical conditional regions, path expressions)

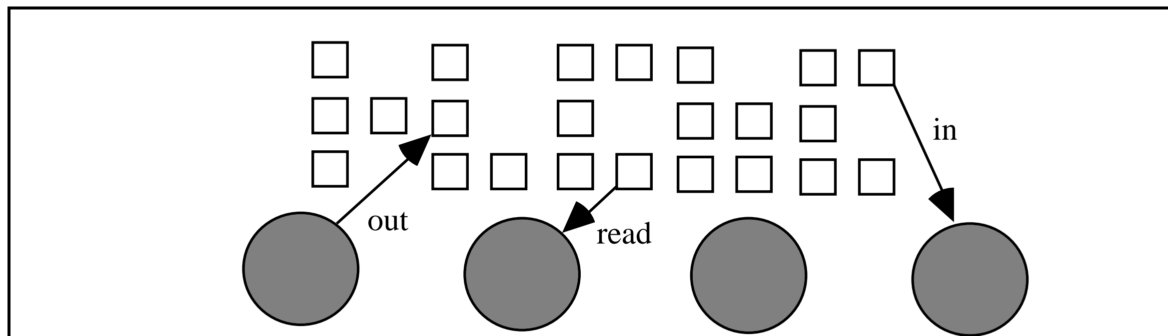


Typical languages of this class are Concurrent Pascal, DP, Edison, Argus, Modula, Mesa, Concurrent Euclid, Unity

Most ancient operating systems were designed using this kind of concurrency (e.g. Unix)

Distributed data structures (tuple space)

A *distributed data structure* is logically separated from processes that can manipulate it; in Linda, for instance, the distributed data structure is contained in a Tuple Space, that is a multiset of tuples; processes produce/consume tuples and create other processes



Linda is the most known language that provides a distributed data structure; other languages that offer distributed data structures are Orca, Shared Prolog, Gamma

Linda

Linda consists of a few simple operations that have to be embedded in a host sequential language to obtain a parallel programming language

programming = coordination + computation

Linda introduced a new paradigm: *generative coordination*

A Linda program refers to a (physically distributed) data structure called *Tuple Space*, that is a multiset of tuples; there are two kinds of tuples:

- passive tuples containing data
- active tuples containing processes

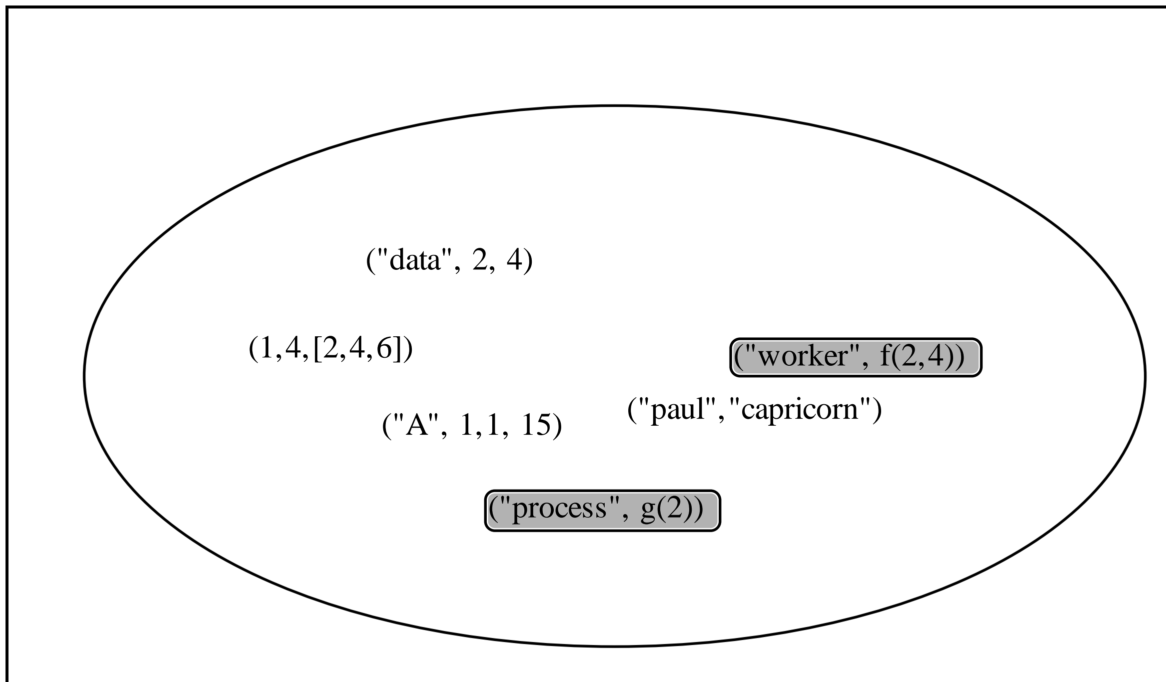
A *tuple* is a sequence of typed *fields*

Types are inherited from the host sequential language (*eg*.: C-Linda types are C basic types or C arrays)

However, not all types are allowed
(eg. in C-Linda pointer fields are forbidden)

A coordination model

The Tuple Space is a global computing environment conceptually including both data, in form of *passive* tuples, and agents, in form of *active* tuples



All tuples are created by agents; they are atomic data (they can only be created, read or deleted)

Agents cannot communicate directly:

an agent can only read (`rd`) or consume (`in`) a tuple, write (`out`) a new tuple or create (`eval`) a new agent that when terminates becomes a data tuple

Operations

Tuples are created and manipulated by agents using the following operations:

`out(t)` puts a new passive tuple in the Tuple Space, after evaluating all fields; the caller agent continues immediately

`eval(t)` puts a new agent in the Tuple Space (each field containing a function to be computed starts a process); the caller agent continues immediately; when all active fields terminate the tuple becomes passive

`in(t)` looks for a passive tuple in the Tuple Space; if not found the agent suspends; when found, reads and deletes it

`rd(t)` looks for a passive tuple in the Tuple Space; if not found the agent suspends; when found, reads it

`inp(t)` looks for a passive tuple in the Tuple Space; if found, deletes it and returns TRUE; if not found, returns FALSE

`rdp(t)` looks for a passive tuple in the Tuple Space; if found, copies it and returns TRUE; if not found, returns FALSE

Matching rules

Operations `in`, `read`, `inp`, `readp` access tuples in the Tuple Space *associatively* (by pattern matching)

Their argument is *a tuple schemata*, namely a tuple containing formal fields used to search a tuple by pattern matching in the Tuple Space;

if a matching tuple is found, the operation is successful

Matching rules for original Linda:

A tuple T in the tuple space matches a tuple schemata S in a tuple operation if their arguments match in number and type, and

i) if argument Ti is actual and argument Si is formal, or
ii) if argument Ti is formal and argument Si is actual, or
iii) if argument Ti is actual and argument Si is actual, and Ti=Si

Matching tuples

Example:
`out("string", 10.1, 24, "another string")`

`real f; int i;`
`rd("string", ?f, ?i, "another string")`
succeeds
`in("string", ?f, ?i, "another string")`
succeeds
`rd("string", ?f, ?i, "another string")`
does NOT succeed

Example:
`out(1,2)`
`rd(?i,?i)` does not succeed

Example:
`eval("worker",7,exp(7))` creates an active tuple
`in("worker",?i,?f)` succeeds when `eval` terminates

Example:
`eval("double work", f(x), g(y))`
`in("double work", ?h, ?k)`
succeeds when both active fields terminate

Coordination patterns of tuple space

Master-worker

```
master(){
for all tasks {
  /* build task structure for this iteration */
  …
  out("task", task_structure);
  }
  for all tasks {
    in("result",?&task_id,?&result_structure);
    /* update total result using this result */
    …
  }
}

worker(){
  while(inp("task",?&task_structure){
    /*exec task*/
    …
    out("result,task_id,result_Structure);
}
```

Distributed data structures in C-Linda

Semaphors

```
out("sem"); out("sem"); out("sem");
```

This is equivalent to an integer semaphore initialized to "3"

Distributed records

```
out("Smith","Paul"); out("Smith",34);
out("Smith","professor");
```

Distributed arrays

```
out("A", 1,1, 15);
out("A", 1,2, 7);
out("A", 2,1, 10);
out("A", 2,2, 22);

for (next=1; next<2; next++)
   rd("A", 1, next, ?LocalA[next])
```

Distributed lists

```
out("A", "atom", value1)
out("B", "atom", value2)
out("A", "cons", ["A","B"])
```

Streams

```
out("strm",1,val1);
out("strm",2,val2);
out("strm",3,val3);
out("strm","tail",4)
```

To append a value to the stream:

```
in("strm","tail",?index);
out("strm","tail",index+1);
out("strm",index,NewElem);
```

A queue as a “live” data structure

```
init_queue(name)
char *name;
{
    out(“queue head ptr”,name,0);
    out(“queue tail ptr”,name,0);
}

add_to_tail(name,val)
char *name; int val;
{ long ptr;
    in(“queue tail ptr”,name,?&ptr);
    out(“queue tail ptr”,name,ptr+1);
    out(“queue”,name,ptr,val);
}

take_from_head(name)
char *name;
{ long ptr;
    in(“queue head ptr”,name,?&ptr);
    out(“queue head ptr”,name,ptr+1);
    in(“queue”,name,ptr,?&val);
    return val;
}
```

Dining philosophers

```
#define NUM 5

phil(i)
    int i;
{
    while(1) {
        think();
        in("room ticket");
        in("fork", i);
        in("fork", (i+1)%NUM);
        eat();
        out("fork", i);
        out("fork", (i+1)%NUM);
        out("room ticket");
    }
}

real_main()
{
    int i;
    for (i=0, i<NUM, i++){
        out("fork", i);
        eval(phil(i));
        if (i<(NUM-1)) out("room ticket");
    }
}
```

Implementations of Linda

Linda has been proved to be efficiently implementable on a wide set of hardware architectures, even if efforts have to be devoted to exploit the features of specific machinery

There are two approaches to Linda implementation:

⇒ build a library for Linda primitives, and include it in a host sequential language; the resulting run time system extends the sequential one with distributed programming capabilities based on the Tuple Space abstraction.
This approach is used in some simple prototype implementations (e.g. POSYBL, LiPS, Minix Linda) and in some programming systems for Linda-like high-level coordination languages

⇒ build a new compiler for a sequential language extended with Linda primitives; the compiler optimizes the implementation of Linda primitives and data structures for specific architectures (works by Carriero and others)

A Linda implementation has to consider if:

1) the underlying hw includes shared memory
2) the hw does not include shared memory (eg. network)

The main problems to be solved by the Linda run time system are how to find a tuple and where to store a tuple

Linda™ run time

The (commercial) Linda run time has the goal to implement the Tuple Space abstraction, offering support for tuple search, matching, communication, and synchronization

The Linda compiler is machine independent for all that concerns tuple parsing and active tuple analysis

Instead, the implementation of tuple operations like `in()/rd()` are typically machine dependent

1) where are stored tuples? in which host? in which memory level?
2) when the area has been found, how it is searched?

The Tuple_Space can be partitioned inserting tuples generated by `out()` and `eval()` in different "signature" sets

Each set is then again partitioned in subsets,
one for all tuples with the same constant fields

Issues in distributed implementations

⇒Tuple space organization and distribution
⇒Eval implementation
⇒Load balancing scheme used for process creation
⇒Procotol implementation for transfer of tuples
⇒Configuration of the processor network

The Linda concept has been implemented in a number of flavors

Original Linda™ by SCA™
VAX Linda by Leichter
Unix Linda
Minix Linda

Network implementations:
- POSYBL
- LIPS
- Pinakis' Tuple Server
- GLENDA
- Laura (TU Berlin)

Multiple tuple spaces:
- Piranha™ by SCA™ (Yale)
- Paradise™ by SCA™ (Yale)
- Bonita (York Univ.)

The master worker model

Linda programs exhibit automatic load balancing; in the “Piranha” environment, running over a network of workstations, programs “steal” computing power when the processors are idle

Linda/Piranha also allows for “adaptive parallel programs”, i.e. computations that dynamically change the set of processors they use: processors may join or withdraw from the computations as it proceeds

A typical program consists of two types of processes: one master and several workers

• The *master* manages the task agenda, that has to be always available; it distributes computations and gathers results; it can also consume tasks

• A *worker* typically executes a loop that inputs a task and the corresponding input data, performas the task, creates 0 or more task, and outputs data

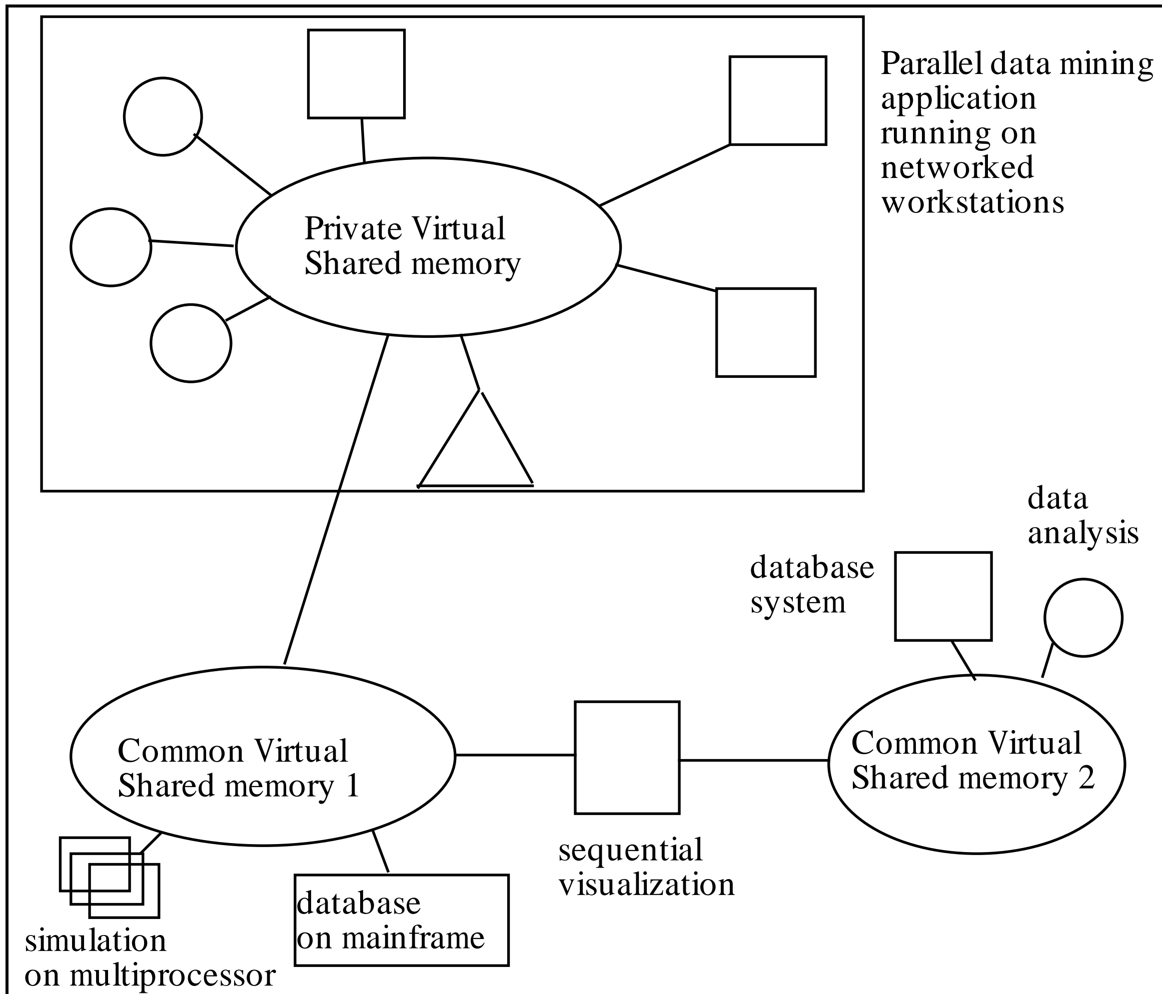
The set of processors is partitioned in available and withdrawn processors: only available processors run piranhas; if an available processor withdraws, the local piranha is destroyed (the user program does not creates processes);

Workers do not migrate; they communicate through the tuple space that persists irrespectively of which processors are available

Typical applications for Linda are:
- Montecarlo simulations
- LU decompositions
- domain decomposition (simulations)

Paradise™

Paradise™ [Kaminsky 93; SCA™ 1996] enhances the Linda model of coordination with multiple tuple spaces



Paradise

Paradise is a flat multiple tuple spaces extension of Linda, implemented over a network of heterogeneous workstations (both Sun and Intel)

It includes the standard Linda operators, except `eval`, plus several other coordination operators for tuple space manipulation

```
#include <paradise.h>  % producer
main()
{ TSHANDLE myts, rmyts;
  open @ rootts(); % default root ts
  myts = create @ rootts(PARADISE_PERSISTENT,
                   PARADISE_LABEL("my TS"));
  out @ myts("message", "Hello world!");
  rmyts = restrict @ myts(PERM_RD);
  register_handle("info","gen","xxxx",&rmyts);
  close @ myts();
  close @ rootts();
}

#include <paradise.h>  % consumer
main()
{ TSHANDLE myts;
  char s[1024]; int stat;

  stat = lookup_handle("info","gen",&myts);
  open @ myts();
  rd @ myts("message",?s);
  printf("Message is: %s\n",s);
  close @ myts();
}
```

Paradise primitives

The Paradise system is controlled by a server process which provides Paradise's services to client programs

The `paradise` command is used to boot the system:
- creating the root `rootts` and temporary `tmpts` tuple spaces
- placing tuples into roots, including a handle for tmpts.
- starting the handle server, which immediately registers `rootts` and `tmpts`
- maintaining the time tuple in the format `("time",host,sec,usec)`

Operations:

```
in@ts("coord",?x)    inp@ts("coord",?x)
rd@ts("coord",?x)    rdp@ts("coord",?x)
out @ ts("coord",3.0)
```

matching is complicated in case of heterogeneous hw
- big endian vs little endian
- floating point representation
- structure alignement

`create @ ts()` creates a new tuple space at the same Paradise server as the specified handle

`open @ ts()` opens a tuple space handle

`close @ ts()` closes a tuple space handle, cancelling any pending transactions

`catch @ ts(handler)`
specifies a ts handle-specific error handler

Protection scheme

`allow @ ts(pmask)`
checks handle privileges and determine wheter the specified handle has all of the specified permissions in pmask

`restrict @ ts(pmask)`
creates a less (or equally) privileged ts handle, setting its permissions as specified in pmask

Paradise allows the admin to limit access to the system:

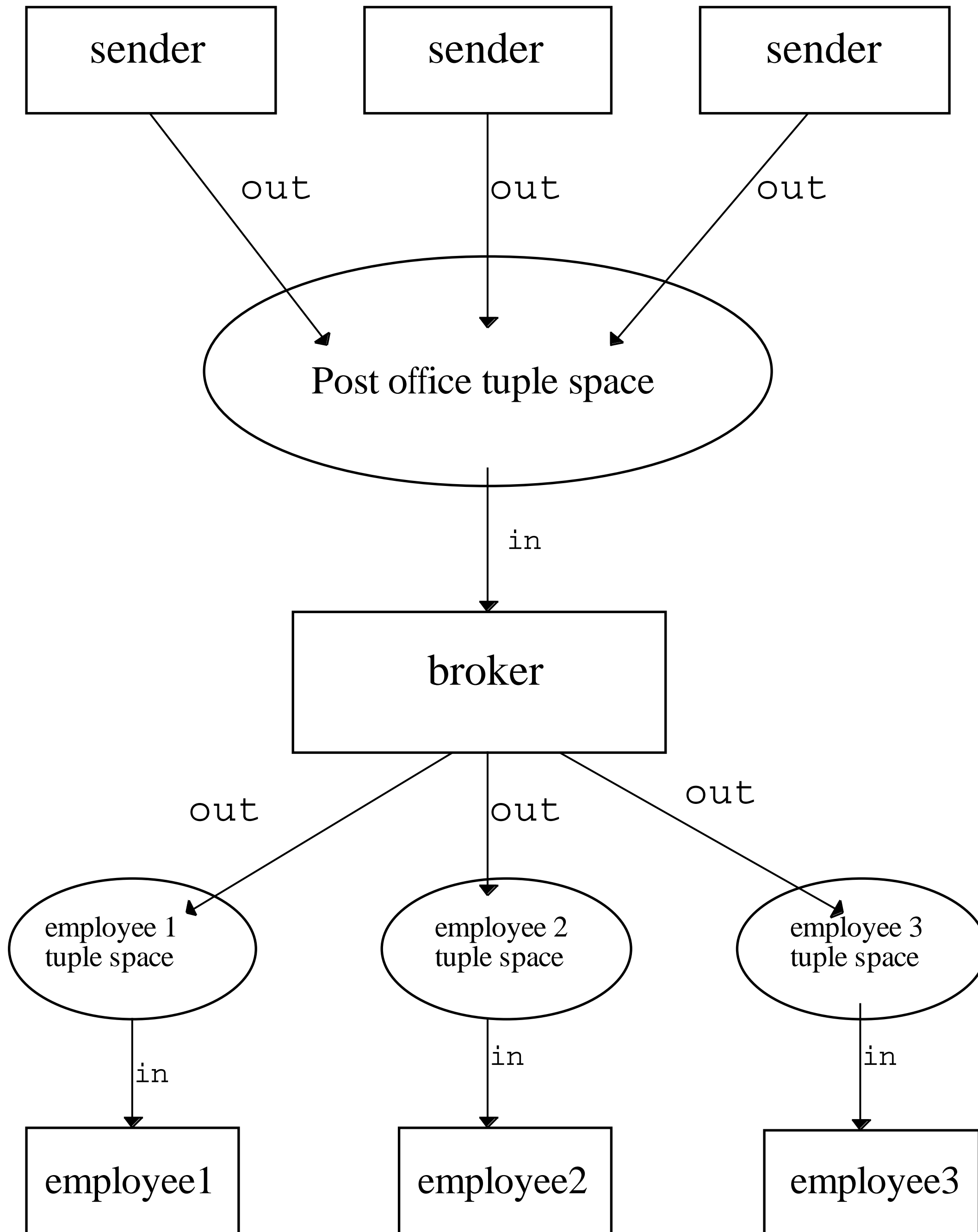
`nodelist` lists the hosts which can connect to the Paradise server
`uidlist` specifies the users (by userid) who are permitted to use the Paradise server
`gidlist` specifies the UNIX user groups (by gid) who are permitted to use the Paradise server

A tuple space can be checkpointed periodically, to recover from unexpected events

`xaction @ ts()` initiate a transaction
`commit @ ts()` commit a pending transaction
`cancel @ ts()` cancel the current transaction

Example

A simple e-mail system



Example

Create persistent PostOffice and LetterBox tuple spaces

```
#include <paradise.h>

main()
{ TSHANDLE myts,rmyts;
  open @ rootts(); /* this is the root ts */

  /* create Post Office ts */
  myts = create @ rootts(PARADISE_PERSISTENT,
     PARADISE_LABEL("PostOffice"));

  rmyts= restrict @ myts(PERM_OUT);
  register_handle("PostOffice","Sender","xxxx",&rmyts);
  rmyts= restrict @ myts(PERM_IN);
  register_handle("PostOffice","Broker","xxxx",&rmyts);
  close @ myts();

/* Create a LetterBox for each employee */
  myts = create@roots(PARADISE_PERSISTENT,
    PARADISE_LABEL("LetterBox Employee 1"));

  rmyts= restrict @ myts(PERM_OUT);
  register_handle("Employee 1","Broker","xxxx",&rmyts);
  rmyts= restrict @ myts(PERM_IN);
  register_handle("Employee 1","Employee","xxxx",&rmyts);
  close @ myts();

…
  close @ rootts();
}
```

Example

Insert a new msg into PostOffice

```
#include <paradise.h>
main()
{ TSHANDLE myts;
    lookup_handle("PostOffice","Sender",&myts);
    open @ myts();
    out @ myts("Employee 1","Hello!");
    close @ myts();
}
```

Broker

```
#include <paradise.h>
main()
{ TSHANDLE myts;
    char addressee[100]; char message[256]; int stat;
    while(TRUE){
    lookup_handle("PostOffice","Broker",&myts);
    open @ myts();
    in @ myts(?addressee,?message);
    close @ myts();

    if (lookup_handle(addressee,"Broker",&myts)==1){
        open @ myts();
        out @ myts(addressee,message);
        close @ myts();
    }else{
        lookup_handle("Unknown","Broker",&myts)
        open @ myts();
        out @ myts(addressee,message);
        close @ myts();

}
```

Paradise

Paradise implements a simple “flat” multiple tuple space concept: the tuple spaces are named and they cannot be nested

Tuple spaces are protected under an access control scheme

Tuple spaces can be used as shared relational databases, and transactions can be defined

A Paradise client can use tuple spaces under control of different TS servers

There are several ways that programs may share data using Paradise

- one program may visualize or analyze data generated by other programs
- a program may compare and cross-analyze data generated by other programs
- a program may use the data from another program to perform its own computations or simulations

Coordination

Coordination is a key concept for studying the activities of complex dynamic systems

⇒*Coordination is managing dependencies between activities* [Malone&Crowston 94]. All instances of coordination include *agents* performing *activities* that are *interdependent*

⇒*Coordination is the process of building programs by gluing together active pieces; a coordination model is the glue that binds separate activities into an ensemble*" [Carriero&Gelernter 92]

A coordination model is the formal basis (semantics) for a coordination language; usually a coordination language has to be combined with a conventional programming language to obtain a fully-fledged programming language

A number of co-ordination languages have been defined and studied in the last ten years; however the field is far from being exhausted, especially because the concept of "co-ordinable entity", or agent, has still to be fully understood

Conclusions

What is a coordination model?

Historically, Linda was introduced as a new model for parallel programming, more flexible and high level wrt its competitors (“Linda Is Not aDA”)

It proved that it is possible to “think in a coordinated way” abstracting from low level mechanisms for concurrent, parallel, or distributed programming

It showed that a careful design can avoid to pay a performance price to use a high-level coordination model

It opened the way for a new reasearch area:
Coordination Languages and Models

A *coordination model* is an abstract (semantic) framework useful to study and understand problems in designing concurrent and distributed programs

“A coordination model is the glue that binds separate activities into an ensemble”' [CarGel92]

In other words, a coordination model provides a *minimalistic* framework in which the interaction of individual agents can be expressed.

This covers the issues of dynamic agent creation and destruction, control of communication flows among agents, control of spatial distribution and mobility of agents, as well as synchronization and distribution of actions over time