Part 1:
An introduction to coordination languages

Contents

• Background and motivation
• What is a coordination language?
• A simple coordination language: Linda
• Linda-like languages
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 *organizational techniques* that do not fit exactly in any of the two paradigms (e.g. master-worker, agenda, pipeline, etc.)

- Software engineers in designing open distributed sw 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 WWW infrastructure can be enhanced to a distributed platform offering support for large-scale groupware and agent-based applications which need specific coordination architectures

- 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
Languages for distributed programming

Several languages for distributed programming exist; they differ widely in the models and in the communication mechanisms they offer to design distributed sw systems.

The most basic and primitive model is that of a static group of sequential processes running in time-sharing or in parallel, and communicating through msg passing. E.g.: CSP and occam.

Some languages support some kind of msg passing, mask communication errors, type checking the contents of msgs. E.g.: Ada.

Since imperative languages were thought to be intrinsically based on a global state, some researchers have explored logic (e.g. FCP, Parlog) and functional (eg. Multilisp, StarLogo) concurrent languages.

Parallelism can also be introduced into object-oriented languages by making objects active (agents; applets).

Most early languages for distributed programming are based on msg passing communication; however, there have been several efforts to abstract away from such a communication mechanisms, e.g. using a generalised form of procedure call (Remote Procedure Call, as RMI in Java).

A more fundamental break is achieved by introducing the high-level concepts of coordination language and coordination model for coordination programming.
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:

- choose the *conceptual class* that is most natural for the problem
- write a program using the *software architecture* that is most natural for that conceptual class
- 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 (*e.g.*: C-Linda types are C basic types or C arrays)

However, not all types are allowed (*e.g.* 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:

\textbf{out}(t)\) puts a new passive tuple in the Tuple Space, after evaluating all fields; the caller agent continues immediately.

\textbf{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.

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

\textbf{rd}(t)\) looks for a passive tuple in the Tuple Space; if not found the agent suspends; when found, reads it.

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

\textbf{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
Idiomatic uses 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**

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

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

**Distributed records**

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

**Distributed arrays**

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

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

**Distributed lists**

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

**Streams**

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

To append a value to the stream:

```c
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;
}
#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()` and `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
- Protocol implementation for transfer of tuples
- Configuration of the processor network
Distributing the tuple space

The search for a tuple in a distributed memory implementation can follow one of two general schemata:

• **hash model**: a distributed hash table is used; if no broadcast primitive is available on the system, we have to minimize communication to reach the node storing the tuple

• **uniform distribution**: every node knows two predefined node sets: the “out-set” and the “in-set”; every in-set has a non-empty intersection with all out-sets (this model offers opportunities for load balancing).

Possible choices for in-sets and out-sets are:

1) for every node n,
   \[ \text{out-set}(n) = \text{all the network}, \text{in-set}(n) = \{n\} \]
   (this needs reliable broadcasting)

2) for every node n,
   \[ \text{out-set}(n) = \{n\}, \]
   \[ \text{in-set}(n) = \text{all the network} \]
   (this does not need broadcasting)

3) for every node \((i,j)\) belonging to a square grid,
   \[ \text{out-set}(n) = \text{nodes in row } i, \]
   \[ \text{in-set}(n) = \text{nodes in column } j \]
Implementation of `eval`

The language does not impose any restriction for creating active tuples anywhere on the network.

Moreover, the language semantics is ambiguous concerning the environment of a new process: it is not clear if the new process shares the global variables of its parent.

There are two main strategies to implement `eval`:

• deliver the code at runtime to remote node; however, such a scheme is attractive neither for very large code, nor for small active tuples representing threads

• each node has a demon `eval` server; each eval server should have the code of all functions that have to be evaluated in parallel
Load balancing

When `eval` contains several active fields to be evaluated in parallel, it becomes necessary to distribute tasks.

The load balancing scheme depends on the availability of following data:

- number of `eval` procedures
- number of times each `eval` is evaluated
- computationas resources needed by an active tuple
- number of elements in in_set and out_set for all `eval()`
- tuple distribution scheme
Linda friends

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
- GLENDIA
- Laura (TU Berlin)

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

POSYBL [Shoinas 91] is a Linda implementation based on a local area network of Unix workstations; it uses RPC extended with broadcasting and answer delegation.

There is no compiler; a daemon process, the Tuple Manager, is installed in every node; processes can invoke procedures in a special library.

A subset of Linda syntax is allowed in tuple operations:
- types have to be explicitly defined,
- the first field has to be an actual,
- eval can only handle one executable file,
- inp and rdp are not supported,
- non formals in out are allowed.
LIPS

Networks of workstations and personal computers are a cheap resource available in several environments

There is a large amount of computing power that is lost simply because the workstations remain idle, especially when they are used for data input

LiPS [Setz96] was designed to implement a tuple space over a network of workstations, stealing idle CPU time

LiPS includes three components:
- a service (in.lipsd) running on each workstation
- mechanisms enabling the user to distribute his program
- monitoring of applications

The service is designed with three goals in mind:
- **security** against unauthorized use: every message is marked by an authorization pattern
- **minimizing maintenance**: the network reconfigures itself every time a workstation is booted (using cron and at)
- **reliability** of the distributed services (but no fault tolerance is guaranteed)
There are several functions to start, stop and coordinate distributed execution; every process calls first the function initlips(), to create a new daemon which handles all msgs with the help of local and remote in.lipsd; the user process communicates with its daemon via sockets.

The daemon:
- waits for communication requests from the program or from other daemons
- processes messages from daemons or from the program
- checks if the user has permission to compute at this time

The user can grant permissions as follows:
- permission for a whole day
- permission for a period in the night
- permission when less than a specified number of users are logged in, or all present users are idle on input

These permissions are checked every 30 seconds. LIPS processes remain in the run queue needing memory at least in the swap area (so the workstation owner actually could be disturbed).

The monitor offers commands to:
- start/stop program; show or kill processes on any hosts
- show available hosts, get info about a host or all hosts
- restart the whole LIPS network
- transmit messages (non blocking send and receive to machine addresses)
- copying files across machines (not based on rcp)
- error handling (as in Unix)
Piranha™

Piranha [Kaminsky, 1994; SCA™ 1996] is a distributed runtime system that offers standard Linda coordination over a network of workstations, whose computing power is “stolen” when they are idle.

This system allows for “adaptive parallelism”, i.e. computations that dynamically change the set of processors they use: processors may join or withdraw from the computations as it proceeds.

A Piranha program consists of two types of processes: one *feeder* and several *piranhas*; the feeder runs on the home node, that has to be always available; it distributes computations and gather results; it can also consume tasks.

A piranha function typically executes a loop that inputs a task and the corresponding input data, performs the task, creates 0 or more task, and outputs data. A piranha can also *retreat* when its node is claimed back and withdrawn from the set of available nodes.

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);

Piranhas do not migrate; they communicate through a tuple space that persists irrespectively of which processors are available.
Paradise™ [Kaminsky 93; SCA™ 1996] embeds the PoliS model of coordination based on 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 \texttt{eval}, plus several other coordination operators for tuple space manipulation.

```c
#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
",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 `roots` and temporary `tmpts` tuple spaces
- placing tuples into roots, including a handle for `tmpts`.
- starting the handle server, which immediately registers `roots` and `tmpts`
- maintaining the time tuple in the format (“time”, host, sec, usec)

Operations:

\[
\begin{align*}
\text{in@ts(“coord”,?x)} & \quad \text{inp@ts(“coord”,?x)} \\
\text{rd@ts(“coord”,?x)} & \quad \text{rdp@ts(“coord”,?x)} \\
\text{out @ ts(“coord”,3.0)}
\end{align*}
\]

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

`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 whether 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

sender

out

sender

out

sender

out

Post office tuple space

in

broker

out

out

out

employee 1
tuple space

in

employee 1

employee 2
tuple space

in

employee 2

employee 3
tuple space

in

employee 3
Example

Create persistent PostOffice and LetterBox tuple spaces

```c
#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

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

Broker

```c
#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();
    }
  }
}```
Bonita

Bonita [Rowstron and Wood 96] takes a simple approach to multiple tuple spaces:

\[ \text{out}(ts, \text{tuple}) \]
\[ \text{in}(ts, \text{template}) \]
\[ \text{rd}(ts, \text{template}) \]
\[ \text{eval}(ts, \text{active_tuple}) \]

Bonita also offers new special primitives for bulk retrieval of tuples. The York group first proposed two primitives:

\[ \text{collect}(ts1, ts2, \text{template}) \]
- moves all tuples matching template from ts1 to ts2, returning a count of tuples moved
\[ \text{copy-collect}(ts1, ts2, \text{template}) \]
- copies all tuples matching template from ts1 to ts2, returning a count of tuples moved

Then another set of primitives were defined, especially suited for “open” (multiagent) coordination architectures

\[ \text{rqid=dispatch}(ts, \text{tuple}) \]
- puts a tuple in ts; non-blocking, returns a req id
\[ \text{rqid=dispatch}(ts, \text{template}, d|p) \]
- retrieves a tuple in ts matching the template, removing (d) or copying (p) it; non-blocking, returns a req id used by other primitives to retrieve the matched tuple
\[ \text{rqid=dispatch_bulk}(ts1, ts2, \text{template}, d|p) \]
- non-blocking; moves/copies from ts1 to ts2 all tuples matching template, returning a count of tuples moved
\[ \text{arrived}(\text{rqid}) \]
- non blocking; tests if a tuple or result associated with rqid is available
\[ \text{obtain}(\text{rqid}) \]
- blocking; waits for tuple associated with rqid to arrive
Implementing Linda ops using Bonita

Bonita operations in some sense are more primitive that Linda operations

\[
\text{in}(ts, ?X) \quad \text{int id, x;}
\]

\[
\text{id} = \text{dispatch}(ts, ?x, \text{destructive});
\]

\[
\text{obtain}(\text{id});
\]

\[
\text{out}(ts, 10) \quad \text{dispatch}(ts, 10);
\]

\[
\text{count} = \text{collect}(t1, t2, ?\text{int})
\]

\[
\text{rqid} = \text{dispatch\_bulk}(ts1, ts2, ?\text{int}, d);
\]

\[
\text{count} = \text{obtain}(\text{rqid});
\]

Solving the non-deterministic choice (multiple rd) problem

C-Linda:

\[
\text{while}(1)
\]

\[
\{ \\
\quad \text{if (inp(ts1, "FIRST"))} \{ \text{do\_first(); break; } \} \\
\quad \text{if (inp(ts1, "SECOND"))} \{ \text{do\_second(); break; } \}
\}
\]

C-Bonita:

\[
\text{int rqid1, rqid2;}
\]

\[
\text{rqid1} = \text{dispatch}(ts1, \text{"FIRST"}, d);
\]

\[
\text{rqid2} = \text{dispatch}(ts2, \text{"SECOND"}, d);
\]

\[
\text{while}(1)
\]

\[
\{ \\
\quad \text{if (arrived(rqid1))} \{ \text{do\_first(rqid1); break; } \} \\
\quad \text{if (arrived(rqid2))} \{ \text{do\_second(rqid2); break; } \}
\}
\]
Laura

Laura (Tolksdorf, TU Berlin) introduces an abstraction of service space which is a tuple space containing a collection of typed forms, shared by all agents; a form can contain a description of a service-offer, a service-request with arguments or a service-result with results.

The service space is monolithic and fault-tolerant (in the original version it is based on ISIS)

SERVE is used by clients to ask for service
SERVICE is used by servers to offer service
RESULT is used by servers to produce a result-form
Laura

In Laura, a service is the result of an interaction between a service-provider and a service-user.

SERVE large-agency operation
(getflightticket: cc * <day,month,year> * dest ->
  ack * <dollar,cent>);
getbusticket  : cc * <day,month,year>  * dest ->
  ack * <dollar,cent> * line;
gettrainticket : cc * <day,month,year> * dest ->
  ack * <dollar,cent>);
SERVE

This states that a service with interface large-agency is offered and that a code for the selected operation should be bound to the var operation.

SERVICE small_agency
(getflightticket: cc * <date.day,date.month,date.year> * dest ->
  ack * <dollar,cent>);
SERVICE

This states that a client requests a service with interface small-agency, to perform op getflightticket.
Conclusions

The Linda coordination model (Tuple Space) has been implemented on a wide range of parallel or distributed hw architectures.

Several experiences have proven that an optimized implementation (eg. commercial SCA™ Linda) has an efficiency not worse than more traditional libraries for distributed programming (eg. PVM).

Even if the tuple space is an embodiment of agenda parallelism, the Linda concept is not bound to a specific compiler or programming environment, but can easily be implemented by a client/server distributed run-time system: thus, it is suitable to implement multi-agent systems over a network.

The most interesting extension to the tuple space concept is coordination by *multiple tuple spaces*, to add modularity and improve protection and security of computations.