SCHOOL
FOR ADVANCED
STUDIES
LUCCA

Principles of Parallel and Distributed Programming

Linguistic Primitives for Distributed Systems

Rocco De Nicola

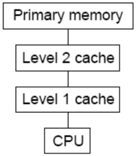
January 2026

GSSI - L'Aquila and IIT CNR - Pisa

Common Architectures

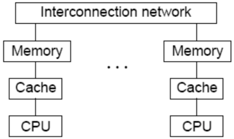
Hardware

Figures from (Andrews, 2000) Chapter 1

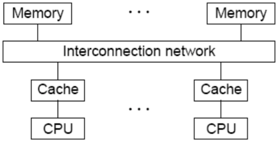


Single processor

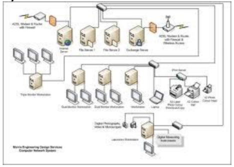
Concurrent programming models exist as an abstraction above hardware and memory architectures



Multicomputer - separate memories (Physically close to each other)



Multiprocessor - shared memory



Network o or cluster of workstations

Interaction in absence of shared memory

Shared Variables require Shared Memory

So far we have considered synchronisation mechanisms based on shared variables that can be executed on hardware in which processors share memory.

Not all architectures guarantee shared memory

There are multiprocessor architectures or networked (distributed) architectures that offer no memory to share.

Message passing

Programs interact by sending and receiving messages

- Processes can be located in different computers connected by a communication network with no need of shared memory
- Mutual exclusion is not an issue in message passing protocols
- Message passing is also used when processes are intended to run on a single computer

SCHOOL FOR ADVANCED STUDIES LUCCA

Syncronization and Naming

Important dimensions

- What form of synchronisation is required
- What form of process naming is involved in message passing

Synchronization of the sender

- Synchronous send: Send and wait for the message to be received (fax)
- Asynchronous send: Send and continue working (e-mail, SMS)
- Rendez-Vous / Remote invocation: send and wait for reply (phone call)

Naming or how do sender and receiver refer to each other

- Direct symmetric (A sends to B and B receives from A)
- Direct asymmetric (A sends to B and B receives from anybody)
- Indirect (A sends to C and B receives from C)

Synchronization

Synchronisation constructs are based upon explicit communication between sending processes and receiving processes.

We have different alternatives:

1. Syncronous Communication: the sender does not send the message (or does not continue) if it has no guarantee that somebody receives it simple Rendez-Vous. E.g., CSP, Occam
2. Asyncronous Communication: the sender assumes that someone will receive his message and continues. E.g., Unix sockets, Java.net
3. Extended Rendez-vous: There is a specific protocol to be executed: The sender asks for availability of the receiver and sends then message; it proceeds only after specific confirmations of reception. E.g., RPC, Rendez-Vous

Message Passing Models

Synchronous message passing

- blocking semantics of both, sender and receiver
- Send operations complete after the matching receive is posted and data are sent
- Receive operations complete after data has been received from the matching send
- Channels can be implemented without buffering

Asynchronous message passing

- non-blocking send and blocking receive
- Send is asynchronous w.r.t. Receive
- After Receive completes, the buffer may be reused
- Channels are unbounded FIFO queues of messages

Naming

1. Direct Symmetric Communication: Explicit identification of sender and receiver

```
SEND message TO process-id
RECEIVE message FROM process-id
```

2. Direct Asymmetric Communication: Receiver might not be interested about whom sent but only in the message itself.

```
SEND message TO process-id
RECEIVE message
```

3. Indirect Communication: Neither the sender nor the receiver are interested in knowing partners' names

```
SEND message TO mailbox
RECEIVE message FROM mailbox
```

4. Channel Based Communication: Partners are selected according to the channels they tune with. Channels are used to send and receive msgs.

```
SEND v on  ch   ||    RECEIVE x on  ch
```

Asynchronous message passing

Channel declaration

A Channel is unbounded queue of messages and is declared by:

```
chan name(id1: type1; ...; idN: typeN);
```

Channels Terminology

- Mailbox: a channel that have several process sending and several process receiving;
- Port: a channel that has exactly one receiver, it may have several senders
- Link: a channel with just one sender and one receiver

Asynchronous message passing

Sending

```
send name(expr1, ..., exprN)
```

- types and number of fields must match with channel decl.
- the effect is to evaluate the expressions, to produce a message M, and atomically to append M to the end of the named channel
- send is nonblocking(asynchronous) (queue is unbounded)

Receiving

```
receive name(var1, ..., varN)
```

- variables types and number must match with channel decl.
- the effect is to wait for a message on the named channel, atomically remove first message (at the front of the queue) and put the fields of the message into the variables.
- Receive is a blocking primitive since it might cause delay

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Synchronous vs Asynchronous Message Passing

Advantages of Synchrony

- There is a bound on the size of communications channels
- At most one message a time queued up on any channel
- The sending process can continue and send another message only after the message is received.

Disadvantages of Synchrony

- Concurrency is reduced. When two processes communicate, at least one of them will have to block
- Programs are more prone to deadlock. The programmer has to be careful that all send and receive statements match up.

Asynchronous vs. Synchronous channels

- send and sync-send, are often interchangeable
- The trade-off is between having more concurrency and bounded communication buffers
- Most programmers prefer asynchronous message passing

CSP: Communicating Sequential Processes

CSP

- A simple programming language designed for multiprocessor machines
- Its key feature is its reliance on non-buffered message passing with explicit naming of source and destination processes
- CSP uses guarded commands to let processes wait for messages coming from different sources.

History

- Interest began in 1965 because of faster, more powerful computers (Edinburgh Multiple Access System) and Simula-67
- Started work by T. Hoare in 1975 – published in 1978
- Expanded (transformed) into a process algebra in 1985

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

CSP: Communicating Sequential Processes

Explicit naming

- process issuing a send( ) specifies the name of the process to which the message is sent
- A process issuing a receive( ) specifies the name of the process from which it expects a message

Indirect naming

- Most message passing architectures include an intermediary entity (port, mailbox, queue, socket, . . .)
- A process issuing a send( ) specifies the entity (e.g. the port number) to which the message is sent
- A process issuing a receive( ) specifies the entity on which expects the message and waits for the the first message that arrives there

CSP: Communication Statements

CSP processes P e Q exchange message by following the patterns below

- `process P { ...; Q!chan(expr); ...}`
  permits `P` to send `Q` the value of expression `expr` using channel `chan`
- `process Q { ...; P?chan(var);  ... }`
  permits `Q` to receive from `P` a value to associate to `var` using channel chan.

In general we have that

`Q!chan(expr)`: is an output statement that evaluates `expr` and sends the result to `Q` using channell `chan`, and blocks until the value is not received.

P?chan(var): is an input statement that blocks the executing process until a message is not received from `P` and bound to `var` to be used for later computations.

CSP: Guarded Commands

Basic CSP instruction have the following form:

```
B; C -> S;
```

where we have that

`B`	is a boolean expression that can be omitted when constantly true
`C`	is a communication command, i.e. input or output commands;
`S`	is the sequence of instruction to be executed after the premises hold.

The pair `B; C` is named guard and `B` is a boolean guard while `C` is a communication guard. A generic pair has the following possible behaviour: A generic pair of guards may give rise to the following behaviours:

- succeeds when `B` is true and `C` can be executed
- fails if `B` is false
- suspends if `B` is true but `C` cannot be executed

CSP: Conditional and Repetitive Commands

Guarded commands can be used within:

If-then commands

```
if B1; C1 -> S1;
   []
   .....
   []
   Bn; Cn -> Sn;
fi
```

while commands

```
do   B1; C1 -> S1;
     []
     .....
     []
     Bn; Cn -> Sn;
od
```

GCD in CSP

```
process GCD {
  int id, x, y;
   do true ->
         Client[*]?args(id, x, y); # input a "call"
         # repeat the following until x == y
         do x > y -> x = x - y;
            []
            x < y -> y = y - x;
         od
      Client[id]!result(x); # return the result
   od
}
```

```
Process i
... GCD!args(i,v1,v2); GCD?result(r); ...
```

n-position Buffer in CSP

```
process Copyn { # n character buffer
       char buffer[n];
       int front = 0, rear = 0, count = 0;
       do count < n; Left?buffer[rear] ->
              count = count+1; rear = (rear+1) mod n;
           []
           count > 0; Right!buffer[front] ->
              count = count-1; front = (front+1) mod n;
       od   }
```

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Exercises

- Semaphore in CSP
- Five Philosophers in CSP
- Consider asynchronous CSP and try the exercise above
- ...

Remote invocation

In distributed systems, a process may need to invoke computation located on a remote machine. The goal is to make remote interactions appear as close as possible to local ones, while:

- hiding communication details,
- preserving a clear programming model.

Remote invocation provides a two way communication channel from the caller to the process servicing the call and back and combines aspects of monitors and synchronous message passing:

- as with monitors interaction is via public procedures
- as with synchronous send, calling a procedure delays the caller

Two classical abstractions are:

- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI)

Remote Procedure Call (RPC)

RPC extends the notion of a *procedure call* to a distributed setting.

- The remote entity exports a set of procedures (functions).
- The caller invokes a procedure as if it were local.
- Arguments and return values are passed by value (marshalling).
- Interfaces are typically specified using an IDL (Interface Definition Language).
- RPC systems are often language-neutral.

Typical view: client-server interaction via stateless function calls.

Remote Method Invocation (RMI)

RMI extends the notion of a *method call on an object* to a distributed setting.

- The remote entity is a remote object.
- Clients invoke methods on that object.
- Object identity is preserved across the network.
- Parameters may be passed:
  - by value, or
  - by reference (remote object references).
- RMI is usually language-specific and integrated with the OO type system.

Typical example: Java RMI.

RPC

RPC permits to a program running on a host to start the execution of a program running on another host. The connection is set up automatically and the programmer is not in charge of setting up the necessary links.

There are two processes involved

- the caller or client
- the called or server.

Each RPC is executed as a distinguished process on server.

Communication is synchronous, and the client suspends till he gets the answer from the server.

1. The client calls the procedure on the server
2. The server does the job
3. The client waits for the result.

RPC

```
     Client                          Server
               .
               .                        WAIT
               .
        1. Call
               .                   2. Receive call
          WAIT                     3. execute
               .                   4. send result
        5. receive result
               .                        WAIT
               .
               .
```

The client calls the servers in the following way:

```
call opname(actual identifiers) # (arguments, result)
```

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Modules and Procedures

Modules

```
module mname
   headers of exported operations; \# exported interfaces
   body
      variable declarations;
      initialization code;
      body of exported procedures;
      local procedures
      background processes;
end mname
```

Procedures

```
     proc opname(formals) returns result
          declarations of local variables;
          statements for proc body
     end
```

Procedure Calls

Client Side:

The client invokes a local stub for the remote procedure that:

1. accepts the parameters and builds up a message with the appropriate data to be sent to remote server (marshalling);
2. sends a message and waits for the answer from the stub on the server or remote stub;
3. extracts the results from the answer (un-marshalling) and gives them to the caller.

Server Side:

The server refers to the *remote stub* of the procedure that is located on the same host of the server and that:

1. receives the message from the client's local stub and sets up the (local) call to the desired;
2. makes the call and receives the results
3. prepares the answers and sends it back to the stub of the client.

An Example: A time server with RPC

```
module TimeServer
       op get_time() returns int; # retrieve time of day
       op delay(int interval);    # delay interval ticks body
       int tod = 0;               # the time of day
       sem m = 1;                 # mutual exclusion sem.
       sem d[n] = ([n] 0);        # private delay sem.
       queue of (int waketime, int process_id) napQ;
       ## when m == 1, tod < waketime for delayed processes
   proc get_time() returns time {
        time = tod;       }
   proc delay(interval) {      # assume interval > 0
        int waketime = tod + interval;
        P(m);
        insert (waketime, myid) at appropriate place on napQ;
        V(m);
        P(d[myid]);  }           # wait to be awakened
```

An Example ctd.

..... CONTINUING .....

```
 module TimeServer
        .......
        process Clock {
            start hardware timer;
            while (true) {
            wait for interrupt, then restart hardware timer;
            tod = tod+1;
            P(m);
            while (tod >= smallest waketime on napQ) {
                remove (waketime, id) from napQ;
                V(d[id]); # awaken process id
            }
            V(m);    }
 }
end TimeServer
```

Rendez-Vous

The basic idea of Rendez-Vous is that of an active process and a client that occasionally requires services to this process that, if available, performs the operation requested by the customer, then it continues its activities.

Compared to RPC, the Rendez-Vous does not allow concurrent execution of procedures and avoids the potential problems in this committed.

Also for the Rendez-Vous we need:

1. synchronization between a client and a server;
2. the execution of operations by the server;
3. returning the results to the client.

Every remote call is served by the same server process and at any given time there is a single server process active. If the server is not available, the caller is blocked until the call is accepted. Anyway it waits until the results provided.

Rendez-Vous

```
in
     op1(formal-s1) and B1 by e1 -> S1;
     []
     ....
     []
     opn(formal-sn) and Bn by en -> Sn;
ni
```

Guards are structured as follows:

opi is the name of the operation to be called; it represents the meeting point of the Rendez-Vous (operation);

Bi is the condition that determines the synchronization; this is possible only if the condition is true(synchronization condition);

ei is an expression used to resolve nondeterminism when more than one guard is successful (scheduling expression).

Rendez-Vous

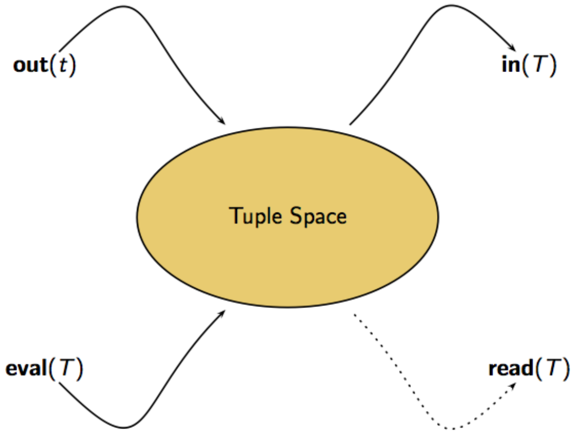
```
Node A                  Node B

Client                  Server
      .                                     .
      .                                  .
1. call op.                         .
                        2. receives on a in command
   WAIT                 3. executes one alternative
                        4. sends results and end
5. receives res.                 .
     .                                   .
     .                                   .
```

An Example: A bounded buffer with Rendez-Vous

```
module BoundedBuffer
      op deposit(typeT), fetch(result typeT);
body  process Buffer {
          typeT buf[n];
          int front = 0, rear = 0, count = 0;
          while (true)
              in
              deposit(item) and count < n ->
                  buf[rear] = item;
                  rear = (rear+1) mod n; count = count+1;
              []
              fetch(item) and count > 0 ->
                  item = buf[front];
                  front = (front+1) mod n; count = count-1;
              ni  }
end BoundedBuffer
```

The Linda Model



Processes coordinate indirectly via a shared *tuple space*

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Tuples, Templates, and Matching

Tuples

A tuple contains only actual fields:

$$("foo", 15, \texttt{true})$$

Templates

A template may contain formal fields (variables):

$$("foo", 15, !u)$$

Pattern Matching

- Formal fields match any value of the same type
- Actual fields must be identical
- Example:

$$("foo", 15, \texttt{true}) \text{ matches } (!s, 15, !b)$$

SCHOOL FOR ADVANCED STUDIES LUCCA

Linda Operations

OUT(exp1, . . . ,expn)

Evaluates expressions exp1, . . . , expn and produces a tuple t that atomically adds to the tuple space. All fields evaluated by the outing process

IN(T)

Either finds and removes a matching tuple or blocks until a tuple t in the tuple space matches the pattern T. After matching it assigns to the variables in T the values in t according to the match. Tuple t is atomically removed from the tuple space.

RD(P)

It is similar to IN(T) except that the matched tuple t is left in the tuple space

Linda Operations (Advanced)

EVAL(exp1,. . . ,expn)

Concurrently evaluates separately the expressions exp1, . . . , expn and produces a tuple t that is atomically added to the tuple space. Spawns a "live tuple" that evolves into a normal tuple; the caller does not wait.

`INP(T)`

Non-blocking version of `IN`. Returns `false` if no matching tuple is found.

`RDP(T)`

Non-blocking version of `RD`. Returns `false` if no matching tuple is found.

Why Linda is *Generative*

Generative Communication

In Linda, communication is performed by *generating data* in a shared space, rather than by sending messages between processes.

- `OUT` and `EVAL` *create* tuples that persist independently of the producing process
- Tuples exist autonomously in the tuple space until explicitly removed
- Producers and consumers are:
  - decoupled in time
  - decoupled in space
  - decoupled in synchronization

Key Consequence

Coordination emerges from the *generation and consumption of shared data*, not from direct process interaction.

Coordinating Parallel Processes

Sequentialization

Two parallel processes:

```
P = P1; P2
Q = Q1; Q2
```

Ensure that, when `P` and `Q` are run in parallel, `Q2` executes only after `P1`:

```
P = P1; out(go); P2
Q = Q1; in(go);  Q2
```

Semaphores

```
sem s = N
for(int x=0; x<N; x++) OUT("s");

P(s);   IN("s");
V(s);   OUT("s");
```

Linda vs. CSP

Dimension	**Linda**	**CSP**
Comm. model	Generative (tuple space)	Message passing
Interaction style	Indirect	Direct
Synchronization	Decoupled	Synchronous rendezvous
Coupling	Time and space decoupled	Strongly coupled
Comm. medium	Shared associative space	Explicit channels
Data persistence	Tuples persist until consumed	Messages are transient
Blocking behavior	Optional (`IN` / `INP`)	Mandatory on send/receive
Coord. paradigm	Data-driven	Control-driven
Formal nature	Coordination language	Process algebra

Key Difference

Linda coordinates processes by *generating shared data*, while CSP coordinates them through *synchronous interaction*.

Kernel Language for Agent Interaction and Mobility

Process Calculus Flavored

- Small set of basic combinator;
- Clean operational semantics.

Linda based communication model

- Asynchronous communication;
- Shared tuple spaces;
- Pattern Matching

Explicit use of localities

- Multiple distributed tuple spaces;
- Code and Process mobility.

From Linda and Process Algebras to Klaim

Explicit Localities to model distribution

- *Physical Locality* (sites)
- *Logical Locality* (names for sites)
- A distinct name *self* (or *here*) indicates the site a process is on.

Allocation environment to associate sites to logical localities

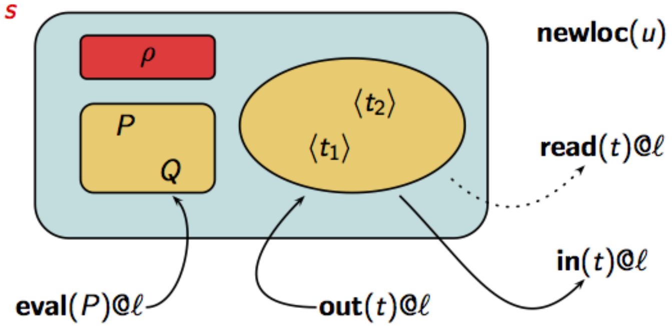
- This avoids the programmers to know the exact physical structure.

Process Algebras Operators to compose programs

- Sequentialization
- Parallel composition
- Creation of new names

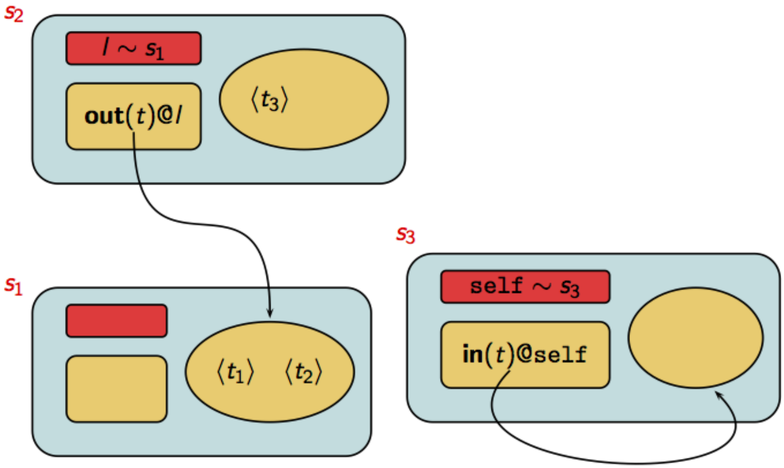
Klaim Nodes

- Locality
- Processes
- Tuple Space
- Allocation Environment



IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Klaim Nets



s_2
$l \sim s_1$
$\mathbf{out}(t)@l$
$\langle t_3 \rangle$
s_1
$\langle t_1 \rangle$
$\langle t_2 \rangle$
s_3
$\mathtt{self} \sim s_3$
$\mathbf{in}(t)@\mathtt{self}$