Programação Concorrente(8)

José C. Cunha, DI/FCT/UNL

-- Processos Distribuídos (Brinch Hansen)

um processo = objecto com múltiplos threads

comunicação por invocação de métodos

-- Communicações Guardadas (Dijkstra)

um processo = um único thread sequencial

comunicação por send/receive síncronos

-- Modelo de Actores (Hewitt & Agha)

um processo = um único thread sequencial

comunicação por operações de envio assíncrono

-- Uma visão unificadora

Modelos de comunicação

1. **por espaços partilhados: com partilha de espaços de endereçamento, comuns a processos distintos;**
2. **por espaços distribuídos: com cópia de valores entre espaços de endereçamento (processos/objectos) diferentes:**
   - **PIPES Unix: fluxos de bytes (streams)**
   - **Mensagens: operações para enviar e receber;**
   - **Objectos distribuídos: invocação de pontos de entrada de processos ou de objectos: chamadas de procedimentos ou de métodos remotos (Remote Procedure Call - RPC ou Remote Method Invocation - RMI);**

Processos concorrentes

Quando comunicam por Memória Distribuída

-- enviar e receber mensagens (o mais básico)

ou invocar/retornar métodos remotos

-- com cópia de bytes

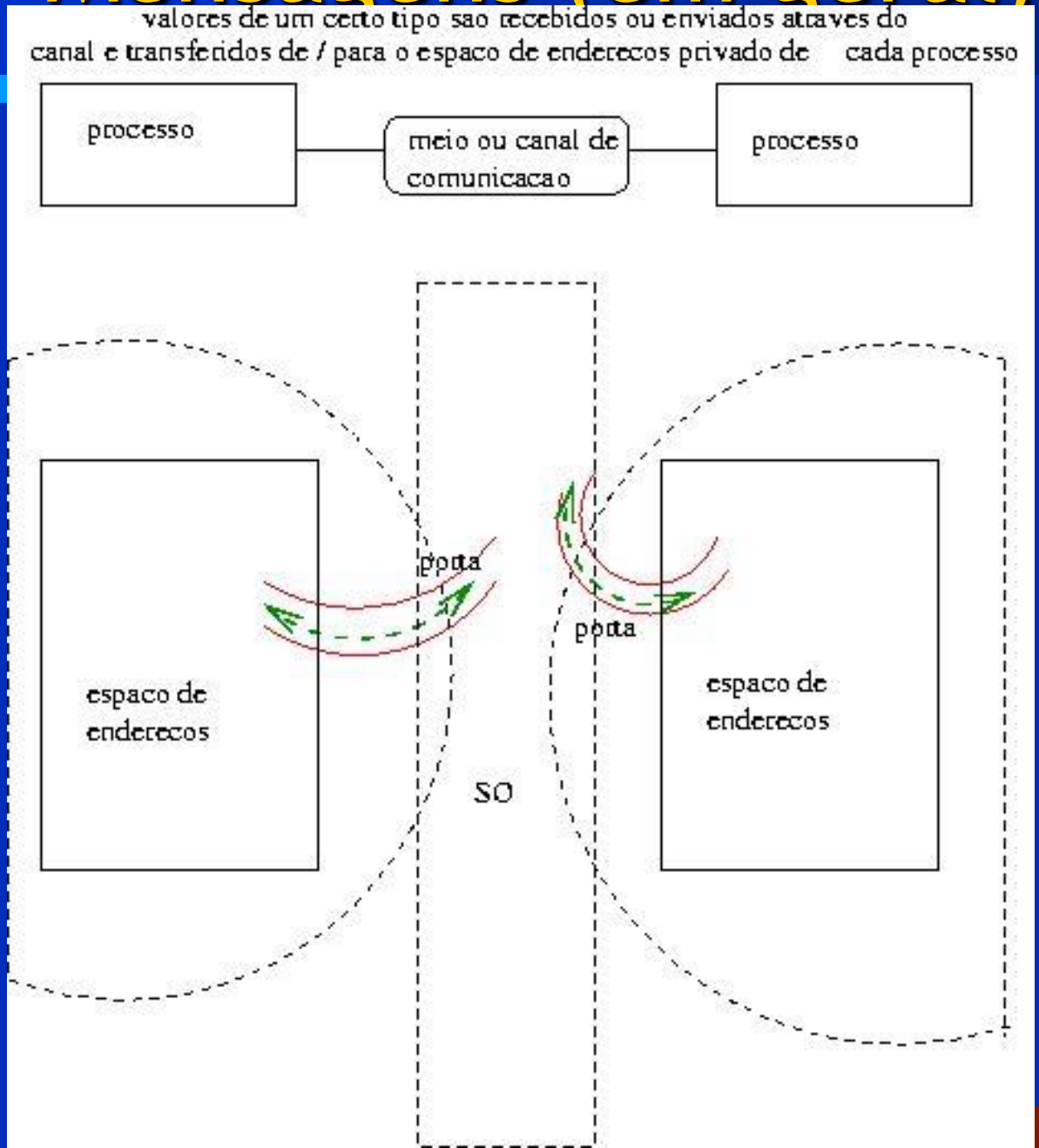
com sincronização implícita nas primitivas de comunicação

-- nas primitivas read/write (pipes) ou receive/send (mensagens) ou call/return (métodos)

Dimensões gerais da comunicação por mensagens

1. **Invocação** explícita ou implícita
2. **Nomeação** directa ou indirecta
3. **Interacção** 1-1, 1-N, N-1, N-M
4. **Sincronização**: operações bloqueantes/não
5. **Fluxo da informação** entre os processos
6. **Não-determinismo**: comunicações selectivas
7. **Papel dos participantes**: simétrico ou assimétrico: clientes e servidores

Mensagens (em geral)



Comunicar por Invocação de Métodos

- **Processes/Objects**
  - -- name, local memory, local threads
  - -- interface entry points: with associated procedures or methods
- **Communication: hides the message-passing**
  - -- based on the **invocation of remote entries of the interfaces defined in other processes**
  - -- with **parameter passing**
  - -- synchronous or asynchronous call semantics

A pioneer proposal: *Distributed Processes*

(Per Brinch Hansen, 1978)

On physically-shared memory systems:

-- Processes are active entities

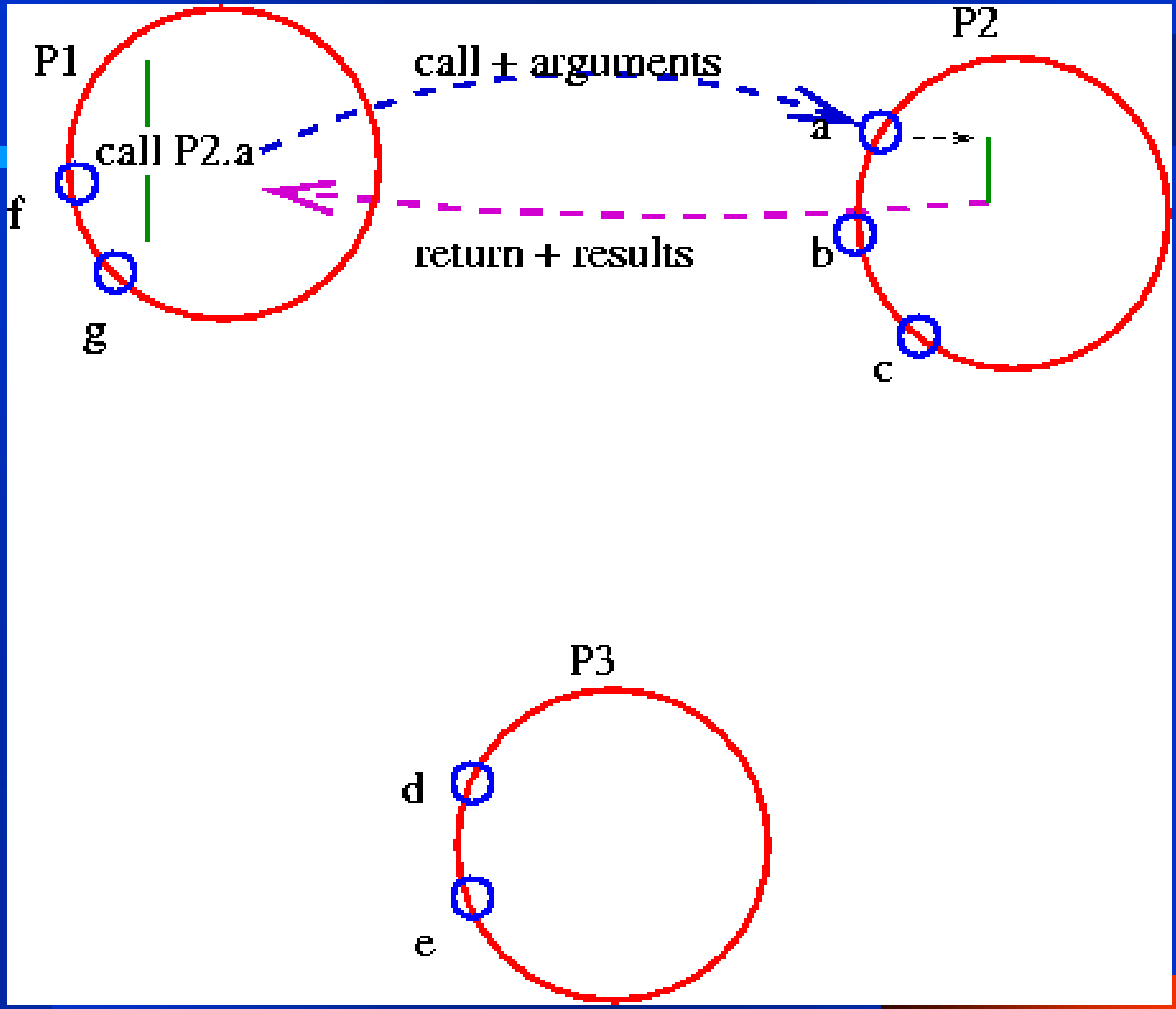
-- Monitors are passive modules

On distributed memory systems:

-- monitors evolve to processes or objects:

A system is built of multiple distributed processes, each process defines interface procedures, to be invoked from other processes.

- A process:
  - -- unique global name
  - -- local private memory/variables/state
  - -- a main process(thread body) for initialisations and for executing some program (or not): passive / pro-active
  - -- a set of interface procedures or public entry points that are invoked by other processes
- Processes communicate via invocations of interface procedures (cf. genesis of RPC or RMI models)
- Processes here are like Objects in OO models.


P1
call + arguments
P2
call P2.a
a
f
return + results
b
g
c
P3
d
e

The Process Address Space

- **Internal to each process, individual threads execute on interface procedure invocation**
- **Each thread runs on a distinct execution context (CPU registers) and stack, but all threads share the same address space (code and data)**
- **On a monoprocessor: a single thread running at each instant in time**
- **On a shared-memory multiprocessor: multiple threads may be active simultaneously**
- **Thread models: in a high-level language (*eg* Java) or as system calls (a standard defined by POSIX Pthreads)**

- **A process/object becomes:**
  - **-- a unit of protection**
  - **-- a unit of spatial distribution (possible mobile…)**
  - **-- a unit for providing a ‘service’**
  - **-- a capsule for internal concurrency and parallelism**
- **Multiple threads inside a process:**
  - **-- communicate via the shared address space (through physically-shared memory)**
  - **-- may need to synchronise through:**
    - **Semaphores and locks (called *mutex*)**
    - **Monitor style: condition variables**

Hipótese simplificadora:

faz sentido p/ sistemas 1 CPU

A single thread is active at each instant in a process and the scheduler is non-preemptive

Deste modo o thread activo só perde o controlo da execução:

- quando termina
- quando se bloqueia aguardando uma condição

E o modelo de programação fica mais simples...

Non-determinism internal to each process

Guarded programming constructs:

-- if b1→S1 or b2→S2 or …. bn→Sn end

(executes only one; error if none is true;)

-- do b1→S1 or b2→S2 or …. bn→Sn end

(repeatedly executes one, until all false)

-- when b1→S1 or b2→S2 or …. bn→Sn end

(wait until one is true; executes only one)

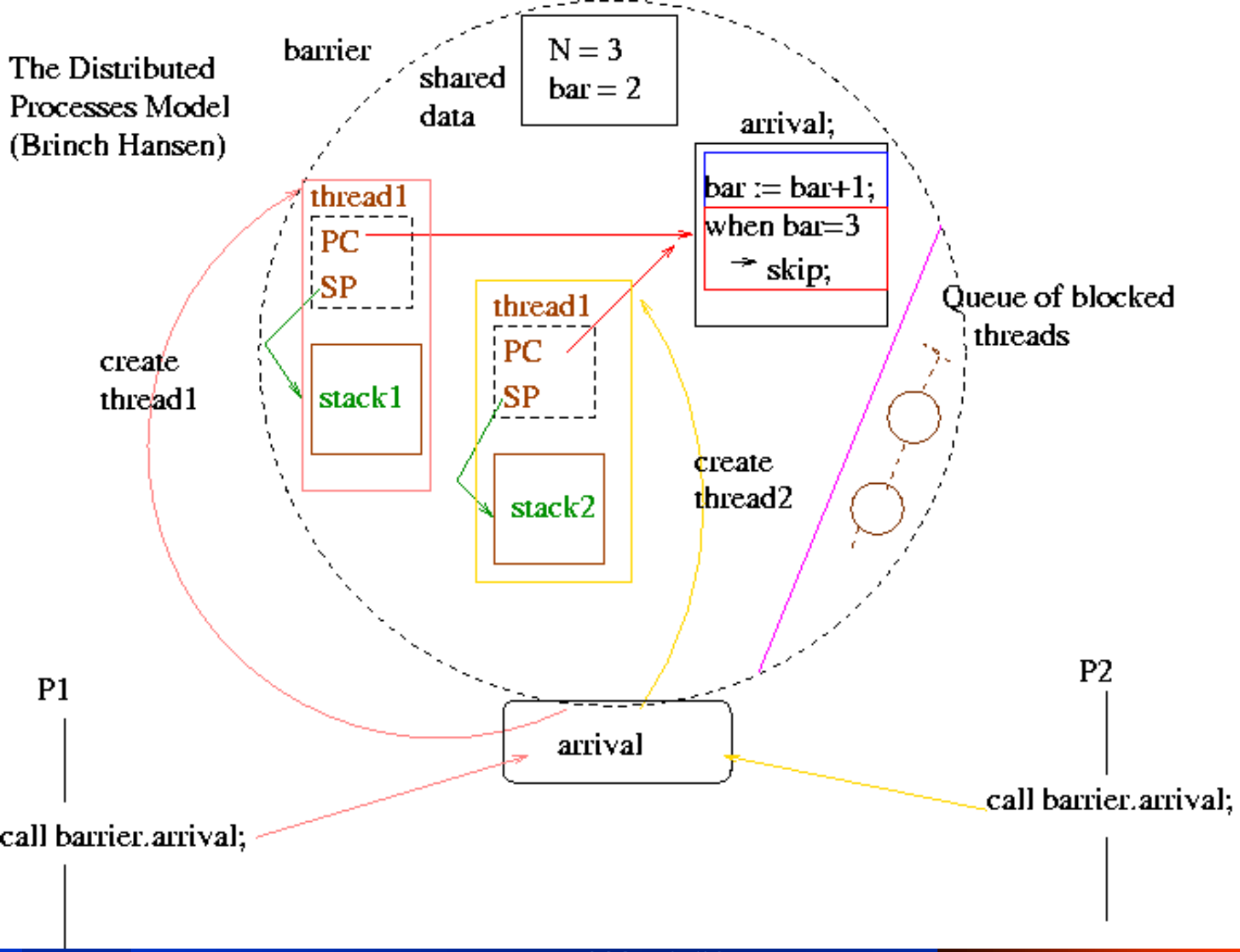
-- cycle b1→S1 or b2→S2 or …. bn→Sn endif

(repeatedly wait until one is true; forever)

A single thread is active at each instant in a process and the scheduler is non-preemptive

Distributed Processes (*B. Hansen*)

```
Process barrier;
{ var bar: integer;
  interface procedure arrival;
    { bar = bar + 1;
      WHEN bar == N → skip END;
            -- bloqueia até a condição ser True
}
  { bar = 0;}
 invoked by:   Call barrier.arrival;
```


The Distributed
Processes Model
(Brinch Hansen)
barrier
shared
data
N = 3
bar = 2
arrival;
bar := bar+1;
when bar=3
→ skip;
thread1
PC
SP
stack1
thread1
PC
SP
stack2
create
thread1
create
thread2
Queue of blocked
threads
P1
P2
arrival
call barrier.arrival;
call barrier.arrival;

Distributed Processes (*B. Hansen*)

```
Process critical_resource;
{ var free: boolean;
  interface procedure request;
    {WHEN free → free = false END;}
  interface procedure release;
    {if not free → free = true END;}
  { free = true;}
}
 invoked by:   Call critical_resource.request;
    or              Call critical_resource.release;
```

Distributed Processes (*B. Hansen*)

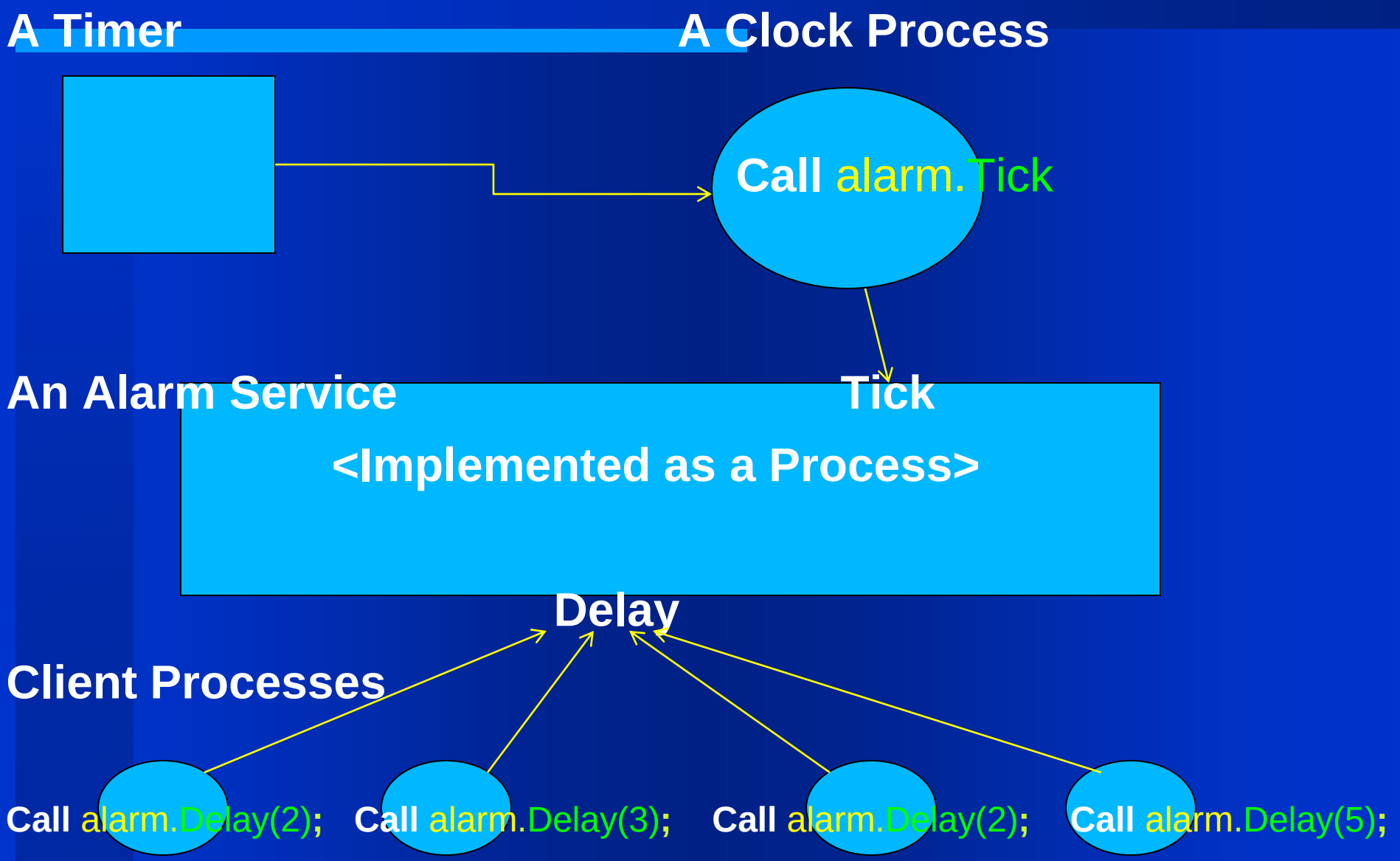
```
Process resource_pool;
{ var total: integer;
  interface procedure allocate(n: integer);
    {WHEN n <= total → total = total - n END;}
 interface procedure free(n: integer);
    { total = total + n;}
  { total = N;}
}
 invoked by:   Call resource_pool.allocate();
    or              Call resource_pool.free();
```

Distributed Processes (*B. Hansen*)

```
Process buffer;
{ var buf: array[0..Bufsize-1]  of T; ... ;
  interface procedure  Put (t: T);
    {WHEN not full(buf) →
                        insert(t, buf) END;}
 interface procedure Get (var t: T);
    {WHEN not empty(buf) →
                        t = remove(buf) END;}
  { initially, buf is empty;}
```

invoked by: Call buffer.Put(); or Call buffer.Get();

Distributed Processes (*B. Hansen*)



Distributed Processes (*B. Hansen*)

```
Process alarm;
{ var Time: integer;
  interface procedure  Delay(interval: integer);
    {var deadline: integer;
        deadline = Time + interval;
        WHEN Time == deadline → SKIP END;}
 interface procedure Tick;
    {Time = Time + 1;}
  {Time = 0;}
```

Process Clock: ... Call alarm.Tick; **// Client Call** alarm.Delay(5);

Distributed Processes (*B. Hansen*)

```
Process semaphore;
{ var sem_value: integer;
  interface procedure  P;
    {WHEN sem_value > 0 →
                  sem_value = sem_value -1 END;}
 interface procedure V;
    {sem_value = sem_value + 1;}

 {sem_value = ... Valor inicial;}
... Call semaphore.P; //  ... Call semaphore.V;
```

Several alternative models

- **A single thread active in each process (non-preemptive) → Brinch Hansen´s model!!!**

but there are other possibilities:

- **Multiple threads concurrently active in each process, preempted by events**
  and synchronised via mutexes and conditions

- **On each interface procedure invocation, a thread is created and runs concurrently with other threads: a preemptive scheduler inside each process**

Dimensões gerais da comunicação por mensagens

1. Invocação explícita ou implícita
2. Nomeação directa ou indirecta
3. Interacção 1-1, 1-N, N-1, N-M
4. **Sincronização: operações bloqueantes ou não**
5. Fluxo da informação entre os processos
6. Não-determinismo: comunicações selectivas
7. Papel dos participantes: simétrico ou assimétrico: clientes e servidores

Guarded communication – handles nondeterminism

Objectivo

- **Programação de autómatos que comunicam entre si e reagem às mensagens recebidas de forma não-determinística**
- **Cada autómato representado por um processo sequencial**
- **Em diversos modelos:**

  os processos comunicam entre si por operações de *send* e *receive* síncronas

Comandos guardados (E.W.Dijkstra)

B → A

Extensão com primitivas de comunicação:

G → A

em que G é uma guarda da forma G: B; C

B - expressão 'Booleana' –condição a cumprir

A – instruções – acção a executar

C - uma operação 'enviar' / 'receber' síncrona

- **Avaliação de uma guarda** **G: B, C**
- **1º passo: avaliação das guardas**
- **2º passo: execução de uma acção atómica**

B expressão booleana

C primitiva síncrona: Send ou Receive

-- B true e C não bloqueia: G com sucesso

-- B true e C bloqueia: G suspende

-- B falsa: G falha

- **Tipos de comandos:**
  - **-- Alternativa (generaliza o IF THEN ELSE)**
  - **-- Iterativo (realiza ciclos)**

-- Alternativa - só executa 1 vez

If G1→ A1 or G2→ A2 or … Gn→ An fi

executa um Ai que tenha um Gi com sucesso

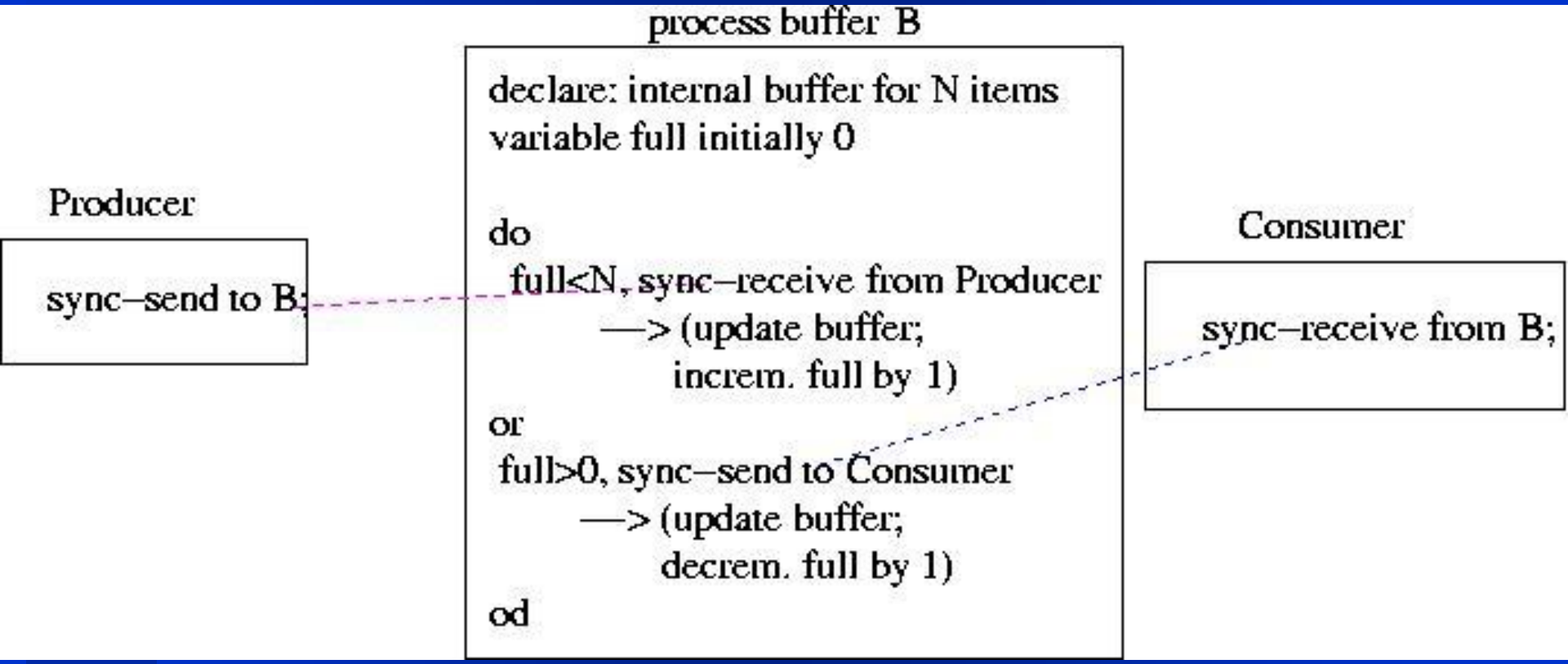
erro se todos os Gi falham

-- Iterativo - executa repetidas vezes

do G1→ A1 or G2→ A2 or … Gn→ An od

repetidamente executa um Ai que tenha um Gi com sucesso,

até que todos os Gi falhem


process buffer B
declare: internal buffer for N items
variable full initially 0
do
full<N, sync–receive from Producer
—> (update buffer;
increm. full by 1)
or
full>0, sync–send to Consumer
—> (update buffer;
decrem. full by 1)
od
Producer
sync–send to B;
Consumer
sync–receive from B;

Noutros modelos:

- **os processos comunicam por operações de envio assíncrono de mensagens**

Asynchronous Actor Models

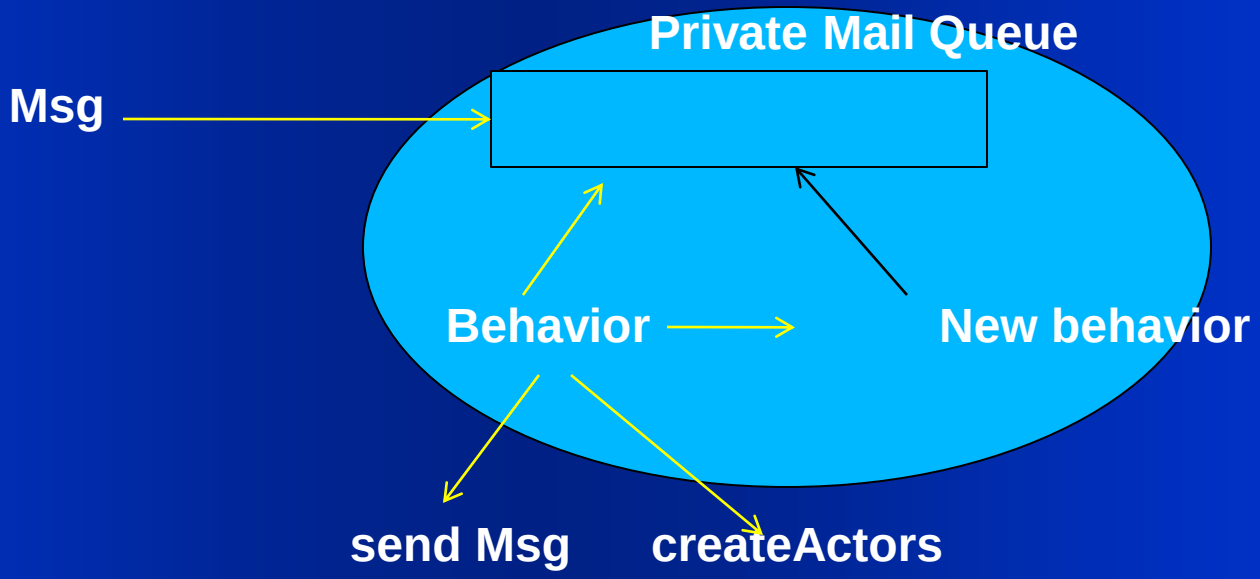
Distributed Object-Oriented

Active Objects:

- **focus on asynchronous communication**
- **encapsulate: data+methods+multiple processes**

origins in Actor model (Hewitt and Agha) 1986

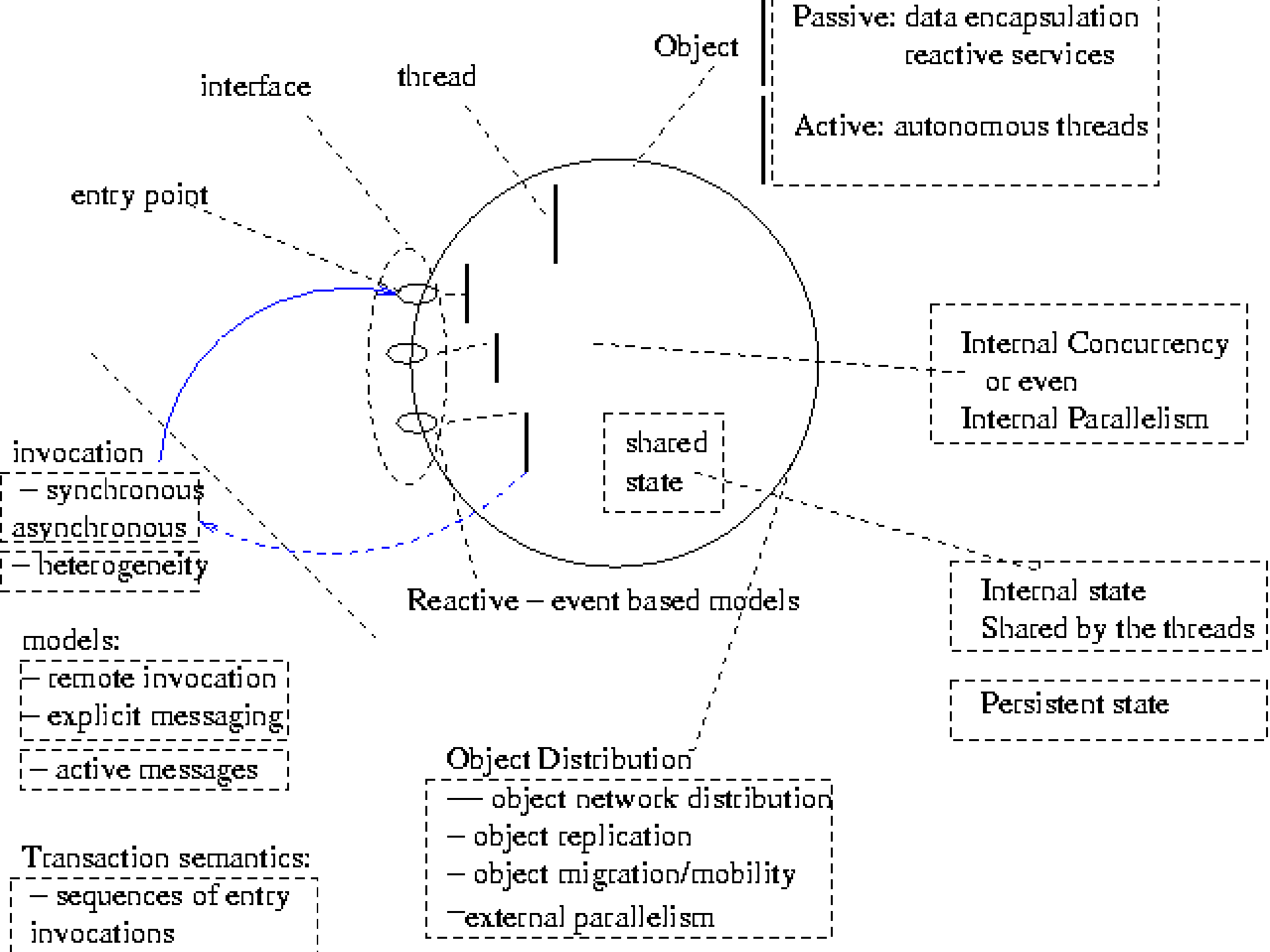
- **a reactive entity**



An Unifying Model

All dimensions centered around an abstraction for a computational entity:

process / agent / object / component / service ...


Passive: data encapsulation
reactive services
Active: autonomous threads
Object
thread
interface
entry point
Internal Concurrency
or even
Internal Parallelism
shared
state
invocation
– synchronous
asynchronous
– heterogeneity
Reactive – event based models
Internal state
Shared by the threads
Persistent state
models:
– remote invocation
– explicit messaging
– active messages
Object Distribution
– object network distribution
– object replication
– object migration/mobility
– external parallelism
Transaction semantics:
– sequences of entry
invocations