Comunicazione con sincronizzazione estesa

Chiamate di procedura remota e rendez-vous

- Chiamata di procedura remota
- Rendez vous
- Linguaggio ADA

Chiamata di operazione remota

- Meccanismo di comunicazione e sincronizzazione tra processi in cui un processo che richiede un servizio ad un altro processo rimane sospeso fino al completamento del servizio richiesto (meccanismo sincrono).

- I processi rimangono sincronizzati durante l'esecuzione del servizio da parte del ricevente ed alla ricezione dei risultati da parte del mittente (rendez-vous esteso).

- Il meccanismo è noto, generalmente, con il nome di chiamata di operazione remota.

- Analogia semantica con una normale chiamata di funzione. Il programma chiamante prosegue solo dopo che l'esecuzione della funzione è terminata.

- La differenza sostanziale sta nel fatto che la funzione (servizio) viene eseguita **remotamente** da un processo diverso dal chiamante.

- Diverse modalità di esecuzione del servizio richiesto: due distinte notazioni :
  - **chiamata di procedura remota** (RPC -Remote Procedure Call)
  - **rendez-vous**.

Chiamata di procedura remota:

- Per ogni operazione che un processo client può richiedere viene dichiarata, lato server, una procedura e per ogni richiesta di operazione viene creato un nuovo processo (thread) servitore con il compito di eseguire la procedura corrispondente.

Rendez-vous:

- L'operazione richiesta viene specificata come un insieme di istruzioni che può comparire in un punto qualunque del processo servitore (es. linguaggio ADA) . Il processo servitore utilizza un'istruzione di input (*accept*) che lo sospende in attesa di una richiesta dell'operazione.
- All'arrivo della richiesta il processo esegue il relativo insieme di istruzioni ed i risultati ottenuti sono inviati al chiamante.

- RPC rappresenta solo un meccanismo di comunicazione tra processi; la possibilità che più operazioni siano eseguite concorrentemente comporta che si debba provvedere separatamente ad una sincronizzazione tra processi servitori.

- Rendez-vous combina comunicazione con sincronizzazione. Esiste, infatti, un solo processo servitore al cui interno sono definite le istruzioni che consentano di realizzare il servizio richiesto. Tale processo si sincronizza con il processo cliente quando esegue l'operazione di *accept*.

Chiamata di procedura remota
(Distibuted Processes, Brinch Hansen 1978)

L'insieme delle procedure remote sono definite all'interno di un componente sw (*modulo*), che contiene anche le variabili locali al server ed eventuali procedure e processi locali:

```
module  nome_del_ modulo
{
<dichiarazione delle variabili locali>;
<inizializzazione delle variabili locali>;
public void op1 (<parametri formali>){
<corpo della procedura op1>;}
……..
public void opn (<parametri formali>){
<corpo della procedura opn >;}

<dichiarazione di procedure locali>;
<dichiarazione di processi locali>;
}
```

I singoli moduli operano in spazi di indirizzamento diversi e possono quindi essere allocati su nodi distinti di una rete.

- La chiamata di una procedura remota verra` specificata dal client con uno statement del tipo:

call nome_del_modulo.op$_i$ (<parametri formali>);

il server crea un thread che esegue l'operazione richiesta

➔ Ad ogni istante e` possibile che piu` thread concorrenti all'interno del modulo accedano a variabili interne. Necessità di sincronizzazione -> monitor, semafori, ...

Esempio: servizio di sveglia

Si vuole realizzare tramite RPC un allarme che ha il compito di risvegliare un insieme di processi clienti che richiedono questo servizio dopo un tempo da loro prefissato.

SERVER:

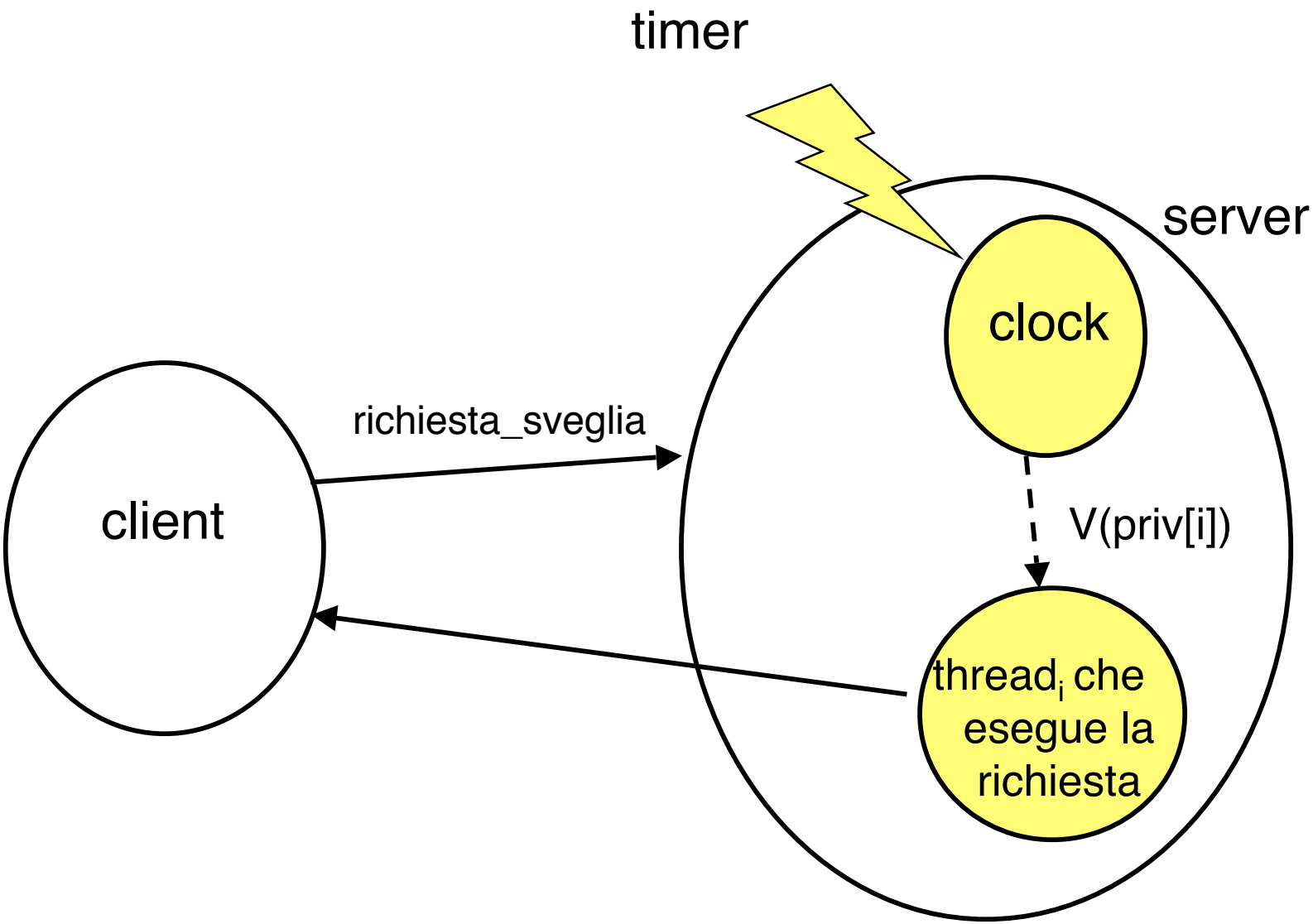
```
module allarme
{
int time;
semaphore mutex=1;
semaphore priv[N]=0; /*semafori privati per la sospensione dei proc.*/
coda_richieste coda; /* struttura contenente le richieste di
                  sveglia (sveglia, id) pervenute*/
public void  richiesta_sveglia(int timeout, int id)
{        int sveglia= time+timeout;
         P(mutex);
         <inserimento  sveglia e id  nella coda di risveglio in modo da
mantenere tale coda ordinata secondo valori  non decrescenti di sveglia>;
         V(mutex);
         P(priv[id]); /* attesa della sveglia..*/
}
```

```
process clock{
int tempo_di_sveglia ;
<avvia il clock>;
    while (true) {
    <attende per l'interruzione, quindi riavvia il clock>;
    time++;
    P(mutex);
    tempo_di_sveglia= < più piccolo valore di sveglia in coda>;
    while ( time>= tempo_di_sveglia) {
    <rimozione di tempo_di_ sveglia e id corrisp. dalla coda>;
    V(priv[id]); /* risveglio del processo id*/
    }
    V(mutex);
 }
}

}/* fine modulo */
```

CLIENT:

```
call allarme.richiesta_sveglia(60);
```


timer
server
clock
client
richiesta_sveglia
V(priv[i])
thread$_i$ che esegue la richiesta

Rendez vous

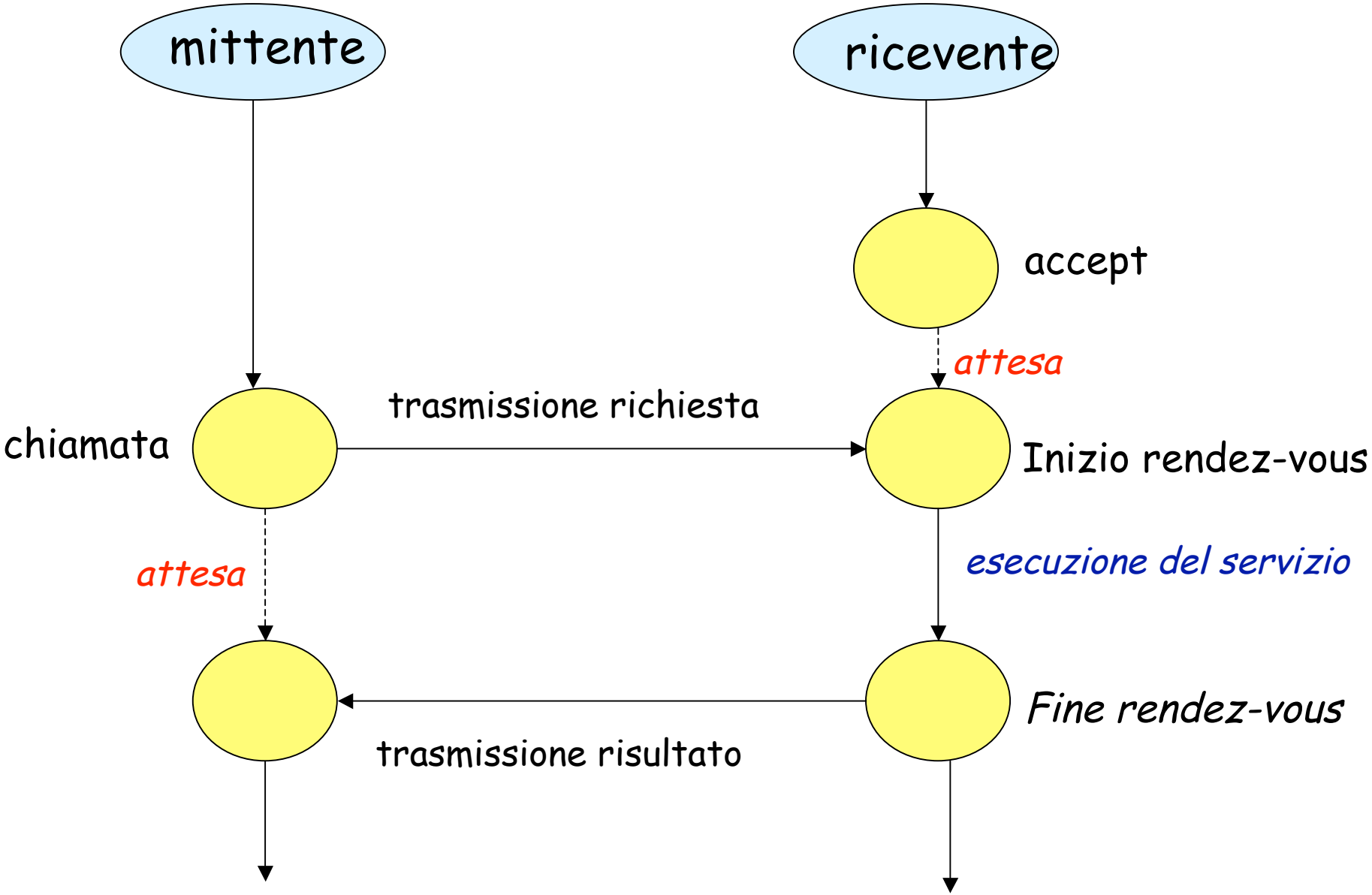
- Il servizio richiesto viene specificato come un insieme di istruzioni che può comparire in un punto qualunque del processo servitore (v. linguaggio ADA, 1976)

accept<servizio>(in <par-ingresso>, out<par-uscita>); -> {S1,..,Sn};

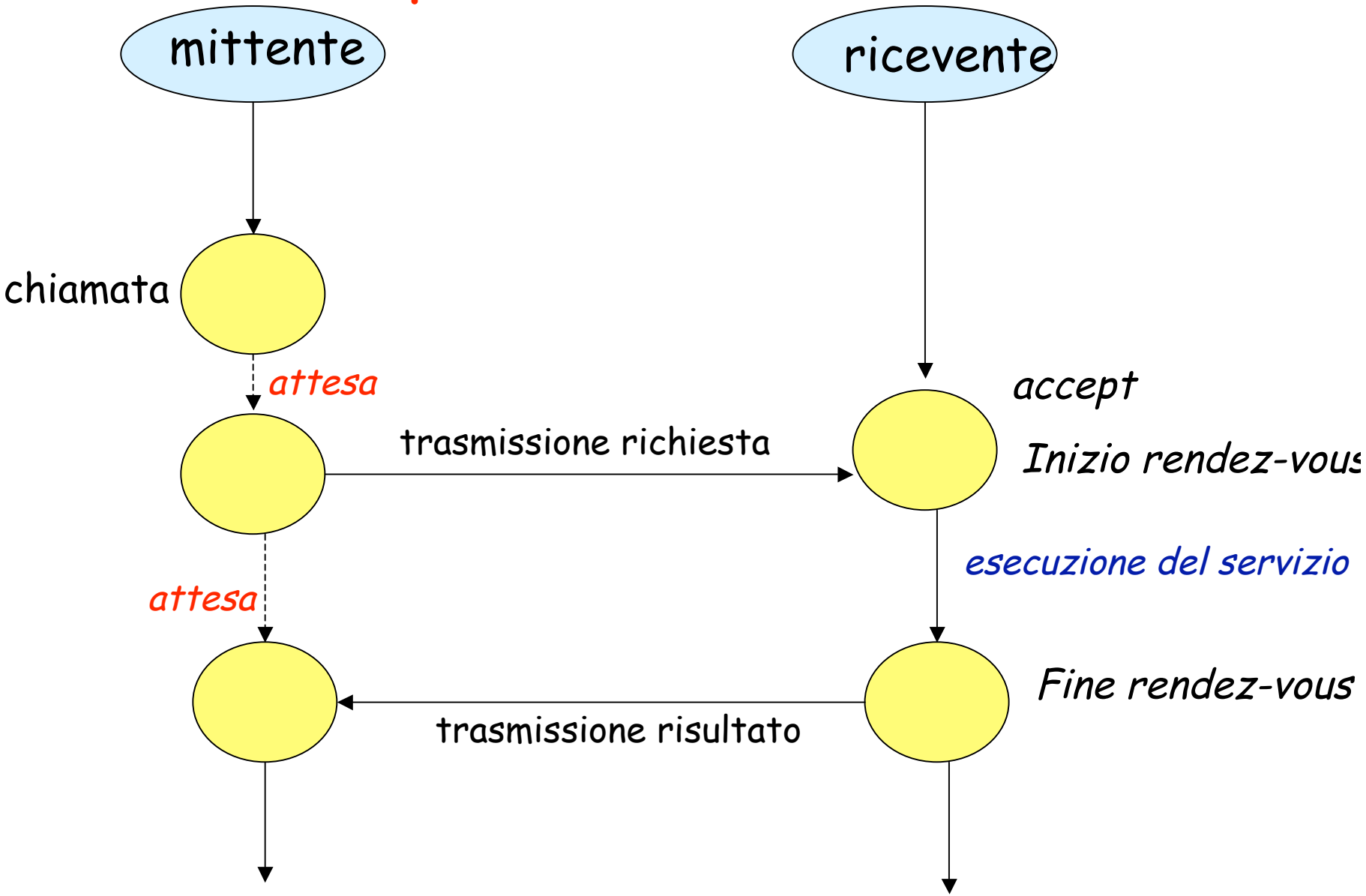
Accept

- Se non sono presenti richieste di servizio l'esecuzione di accept provoca la sospensione del processo servitore.
- Se lo stesso servizio è richiesto da più processi prima che il servitore esegua la accept, le richieste vengono inserite in una coda associata al servizio gestita, normalmente, FIFO.
- Ad uno stesso servizio possono essere associate più accept: ad una richiesta possono corrispondere azioni diverse in funzione del punto di elaborazione del processo che la definisce.
- Lo schema di comunicazione realizzato dal meccanismo di rendez vous è di tipo asimmetrico da molti a uno.

Possibili sequenze di eventi in una chiamata di procedura remota



Possibili sequenze di eventi in una chiamata di procedura remota



Accept: selezione delle richieste

- Nel modello rendez-vous, il server può selezionare le richieste da servire in base al suo stato interno (es. lo stato delle risorse gestite), utilizzando i comandi con guardia:

```
if
    []<stato1>; accept<servizio1>(in <par-ingresso>, out<par-
      uscita>);
        -> {S11,..,S1n}; ...
    []<stato2>; accept<servizio2>(in <par-ingresso>, out<par-
      uscita>);
        -> {S21,..,S2n}; ...
    ...
end;
```

Esempio: produttore e consumatore

```
process buffer
{  messaggio buff[N];
   int testa=0,coda=0;
   int cont=0;
   do {
        [](cont<N);accept inserisci(in dato:messaggio)->
        {       buff[coda] = dato;}/* fine rendez-vous*/
                cont++;
                coda= (coda+1)%N;
        [](cont>0);accept preleva(out dato:messaggio)->
        {       dato=buff[testa];} /* fine rendez-vous*/
                cont--;
                testa=(testa+1)%N;
}
```

NB: la sincronizzazione tra processo chiamante e processo chiamato sia limitata alle sole istruzioni comprese nel blocco di accept (cioe` quelle comprese in -> {..})

```
process produttore-i{
   messaggio dati;
   for(; ;)
    {   <produci dati>;
        call buffer.inserisci(dati);
    }
}

process consumatore-j{
   messaggio dati;
   for(; ;)
   {    call buffer.preleva(dati);
        <consuma dati>;
   }
}
```

Selezione delle richieste in base ai parametri di ingresso

- La decisione se servire o no una richiesta può dipendere, oltre che dallo stato della risorsa, anche dai parametri della richiesta stessa. In questo caso infatti, la guardia logica che condiziona l'esecuzione dell'azione richiesta deve essere espressa anche in termini dei parametri di ingresso.
- E' pertanto necessaria una doppia interazione tra processo cliente e processo servitore; la prima per trasmettere i parametri della richiesta e la seconda per richiedere il servizio.

Vettore di operazioni di servizio

- Nell'ipotesi di un numero limitato di differenti richieste si può ottenere una semplice soluzione al problema associando ad ogni richiesta una differente operazione di servizio (**vettore di operazioni di servizio**) (linguaggio Ada).

Esempio: sveglia

- Si consideri ad esempio il caso del processo (server) allarme il cui compito sia di inviare una segnalazione di sveglia ad un insieme di processi che richiedono questo servizio dopo un tempo da essi stabilito.
- Il processo allarme interagisce periodicamente con un processo clock per tenere traccia del tempo.
- Server: 3 tipi di richieste
  - tick: aggiornamento del tempo (da clock a allarme)
  - richiesta_di_sveglia(T): impostazione della sveglia per il cliente mittente (da cliente generico ad allarme)
  - svegliami[T] (da cliente generico ad allarme):invio del segnale di allarme al tempo specificato

➔ L'ordine con cui il processo allarme risponde alle richieste del tipo **svegliami** dipende solo dal parametro T (intervallo di attesa) trasferito con la richiesta.

Struttura del generico processo cliente:

```
process cliente_i
{  ...
   allarme.richiesta_di_sveglia (T);
   allarme.svegliami[T];
   ...
}
```

Vettore di operazioni di servizio

- possiamo associare ad ogni richiesta di sveglia, un diverso elemento di un vettore:

```
typedef struct
{  int risveglio;
   int intervallo;
}dati_di_risveglio;

/*vettore delle richieste  di servizio: */
dati_ di_risveglio tempo_di_sveglia[N];
```

Server:

```
process allarme
{  entry tick;
   entry richiesta di sveglia(in int intervallo;
   entry svegliami[first..last]; //vettore di entry
int tempo;
typedef struct
{  int risveglio;
   int intervallo;}dati_di_risveglio;
dati_ di_risveglio tempo_di_sveglia[N];

do {
   []accept tick;-> {tempo++;} /* dal processo clock*/
   []accept richiesta di sveglia (in int intervallo)
   -> {<inserimento tempo + intervallo ed intervallo in tempo di
   sveglia in modo da mantenere tale vettore ordinato secondo valori
   non decrescenti di risveglio>;}
   [](tempo==tempo_di_sveglia[1].risveglio);
   accept svegliami [tempo_di_sveglia[1].intervallo];
   -> {<riordinamento del vettore tempo_di_ sveglia>;}
   }
}
```

Ipotesi: bassa frequenza di aggiornamento del clock (periodo>>tempo di servizio)

Linguaggio ADA

- Sviluppato per conto del DOD (Department Of Defense) degli Stati Uniti.
- Applicazioni tradizionali ed in tempo reale.
- Adotta come metodo d'interazione tra i processi (*task*) il *rendezvous*.
- Successiva introduzione dei **protected type** (simili al monitor) e l'istruzione requeue per dare al programmatore maggiore controllo sulla sincronizzazione e sullo scheduling.

- Comunicazione di tipo asimmetrico a rendez vous.

- Il rendez-vous tra due task viene stabilito quando entrambi esprimono la volontà di eseguire una stessa operazione (*entry*).

- Una entry definita in un task P e resa visibile all'esterno di P, può essere chiamata da un altro task Q.

Lo schema base di un task contiene una parte di specifica che definisce le operazioni entry ed una parte body che consente la realizzazione di tali operazioni.

```
task name is
     <dichiarazione delle entry>;
end;

task body name is
     <dichiarazioni locali>;
begin
     <istruzioni del task>;
end name;


entry identifier (parametri formali);
```

Rendez-vous:

- Una entry dichiarata in un task P e resa visibile all'esterno di P, può essere chiamata dal task Q mediante:

```
call P.entryname (<parametri effettivi>);
```

- La comunicazione tra P e Q avviene quando P esprime la volontà di eseguire la entryname mediante:

```
accept entryname (in <param-ingresso>,out<param-uscita>);
do S1; S2;..Sn; end;
```

- Durante l'esecuzione di Si..Sn P e Q rimangono sincronizzati.

- L'esecuzione di accept entryname da parte di P sospende il task fino a quando non vi è una chiamata di entryname.

- In quel momento i parametri effettivi sono **copiati** nei parametri formali di ingresso e viene eseguita la lista di istruzioni.

- Al loro completamento i risultati sono **copiati** nei parametri di uscita e termina la sincronizzazione tra P e Q.

  Coda associata ad ogni entry (gestita *FIFO*): una stessa entry può essere chiamata da più task prima che il task che la definisce esegua la corrispondente accept.

- Ad una stessa entry possono essere associate più accept. Ad esse possono corrispondere azioni diverse a seconda del punto di elaborazione del task.

Comando con guardia (forma semplice):

```
select
        accept entry1 do..end;
   or   accept entry2 do..end;
   ….
   or   accept entryn do..end;
end select;
```

Comando con guardia (forma completa):

```
select
        when cond1-> accept entry1 do..end;
   or         when cond2-> accept entry2
do..end;
   ….
   or   when condn-> accept      entryn
do..end;
```

- Per consentire la realizzazione di politiche dipendenti dal tipo di richiesta, dai parametri associati al task si può usare il concetto di *famiglie di entry*:

```
entry entryname (first..last) (in..out);
```

- L'accoppiamento tra una chiamata ad una entry priva di parametri ed una istruzione accept priva di corpo rappresenta la trasmissione ed il relativo riconoscimento di un segnale di sincronizzazione tra due task.
- Conditional entry call:

```
select
      <chiamata ad una entry>;
      else <comandi>:
end select;
```

```
select
      <chiamata di una entry>
or
      delay <intervallo di tempo>;
end select;
```

Il task chiamante attende, per l'esecuzione del comando, al più un tempo pari ad un intervallo fisso.