Comunicazione con sincronizzazione estesa

Comunicazione con sincronizzazione estesa

- Semantica
- Implementazione:
  - Chiamata di procedura remota
  - Rendez vous
- Strumenti linguistici:
  - Linguaggio ADA

Sincronizzazione estesa: semantica

Modello di comunicazione/sincronizzazione in cui:

- il processo mittente richiede l'esecuzione di un **servizio** al processo destinatario
- il processo mittente rimane **sospeso** fino al completamento del servizio richiesto.

👉 I processi rimangono **sincronizzati** durante l'esecuzione del servizio da parte del ricevente **fino alla ricezione dei risultati** da parte del mittente.

Il meccanismo è noto anche con il nome di **chiamata di operazione remota**.

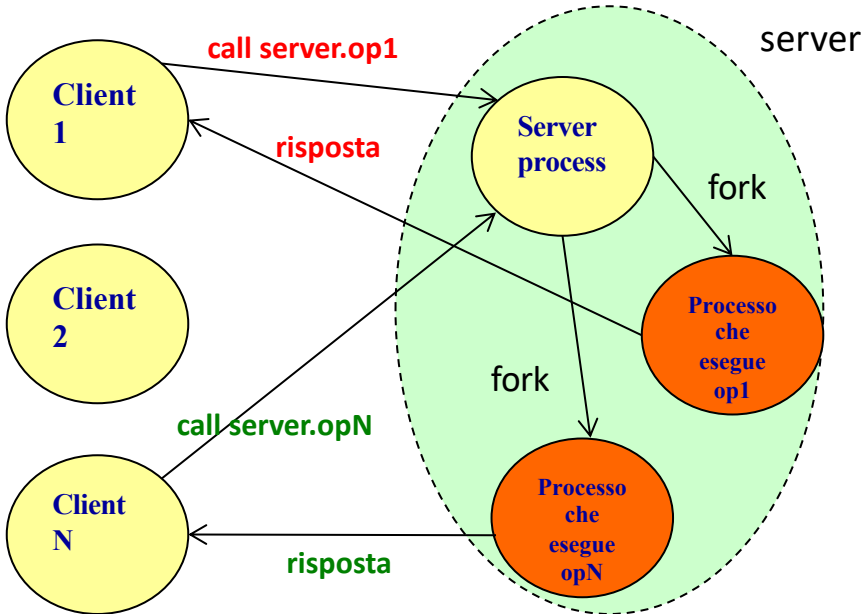
- C’è una forte **analogia semantica** con una normale **chiamata di funzione**: il **chiamante** prosegue solo dopo che l'esecuzione della funzione è terminata.

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

👉 **Due diverse modalità di implementazione lato ricevente (server):**
  - **chiamata di procedura remota** (RPC -Remote Procedure Call)
  - **rendez-vous esteso**.

Chiamata di procedura remota (RPC):

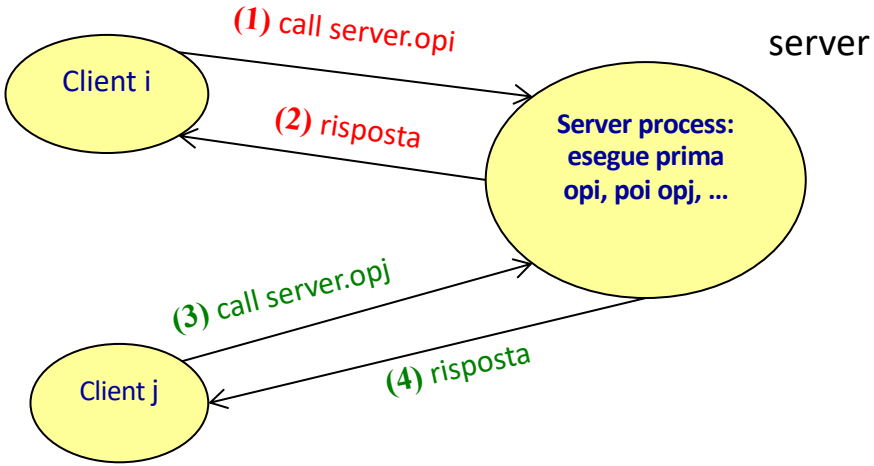
- Per ogni operazione che un processo client può richiedere viene dichiarata, lato server, una **procedura**;
- per ogni richiesta di operazione al server viene creato un **nuovo processo** (thread) che ha il compito di eseguire la procedura corrispondente.



Rendez-vous esteso:

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 v. Rendez Vous esteso

- **RPC** rappresenta solo un meccanismo di **comunicazione** tra processi: la possibilità che più operazioni siano eseguite concorrentemente implica la necessità di sincronizzazione tra i vari processi servitori. La sincronizzazione è a carico del programmatore. (ES: Java RMI, Distributed Processes)

- **Rendez-vous esteso** combina **comunicazione** con **sincronizzazione**. Esiste, infatti, un solo processo servitore al cui interno sono definite le istruzioni che consentono di realizzare il servizio richiesto. Il processo servitore si sincronizza con il processo cliente quando esegue l'operazione di accept (es: ADA).

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

L'insieme delle procedure remote è 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                    // server
{
<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 verrà specificata dal client con uno statement del tipo:

```
module client {
        ...
        call nome_del_modulo.op_i (<parametri attuali>);
        ...
}
```

- il server crea un thread che esegue l'operazione richiesta (op_i)

- Ad ogni istante è possibile che **più thread concorrenti** all'interno del modulo server 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.

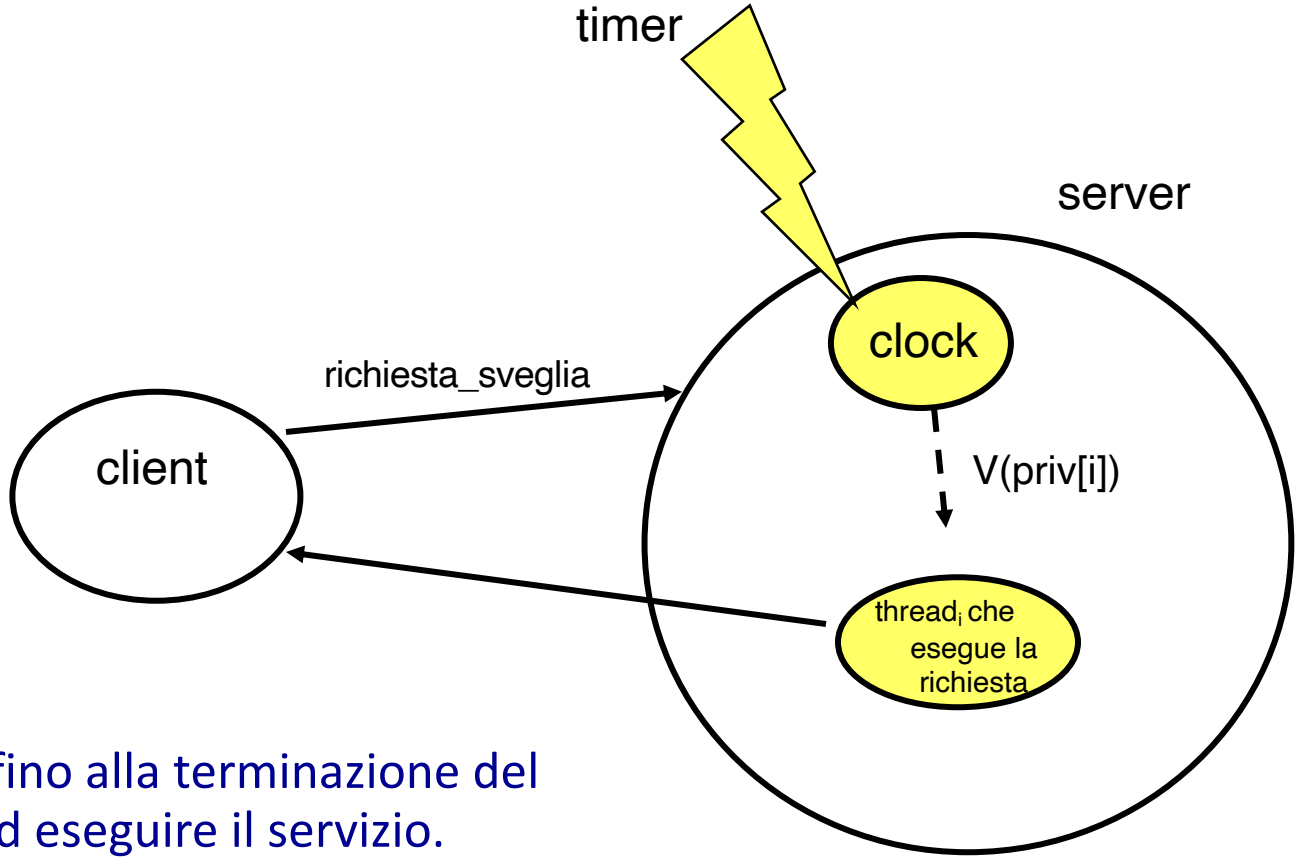
SERVER:

```
module allarme
{
int time;//tempo corrente
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) //servizio di impostazione sveglia
{         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{          // "demone"
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); //crea nel server il thread dedicato
```



Il cliente attende fino alla terminazione del thread delegato ad eseguire il servizio.

Rendez vous esteso

Il servizio richiesto viene specificato dal **server** come un insieme di istruzioni che può comparire in un punto qualunque del processo server tramite l'istruzione accept (v. linguaggio ADA, 1976):

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

dove :

in: parametri di input,

out: parametri di output (risultato del servizio)

Il **client** richiede il servizio con l'istruzione:

call <server>.<servizio>(<parametri attuali>);

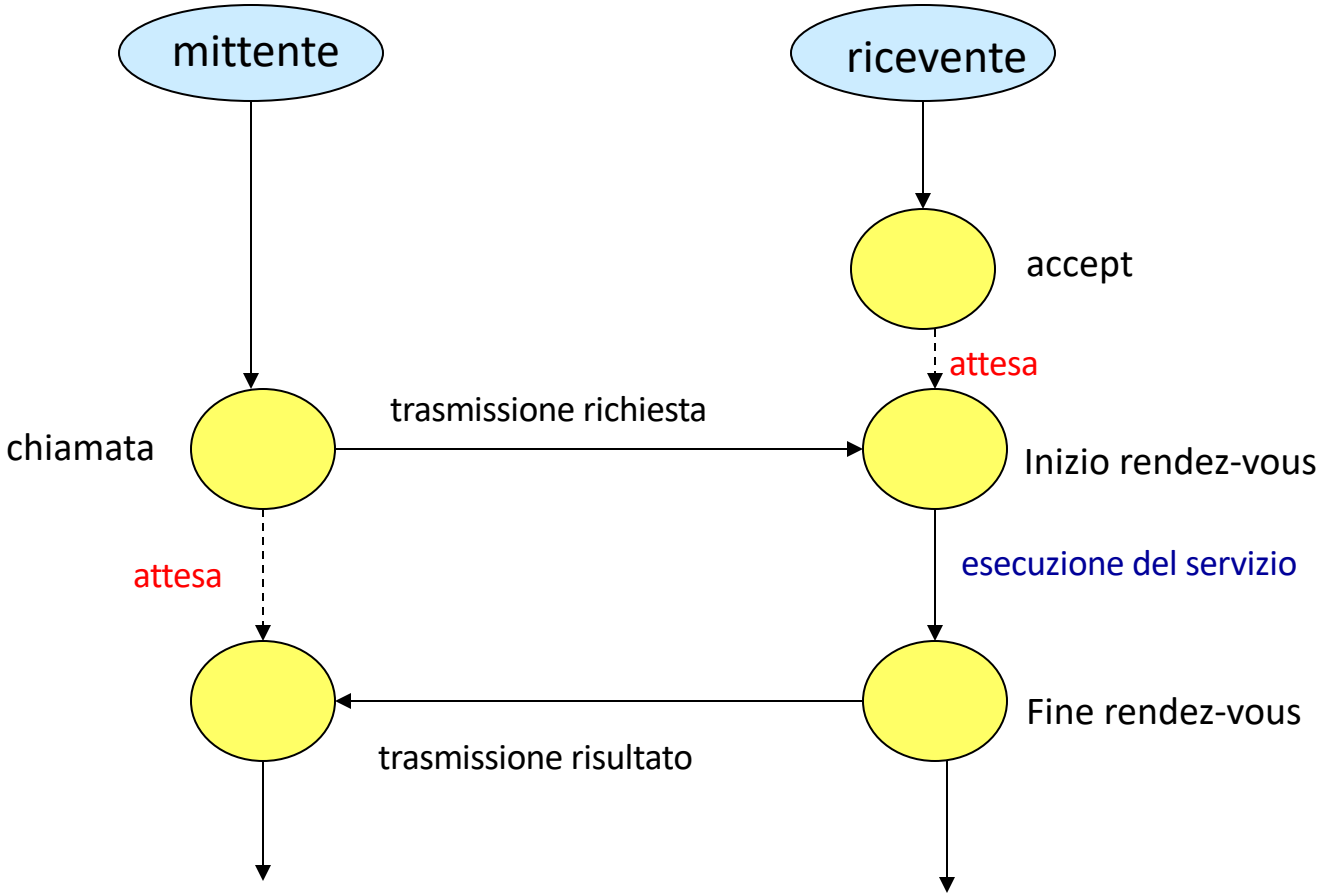
Accept

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

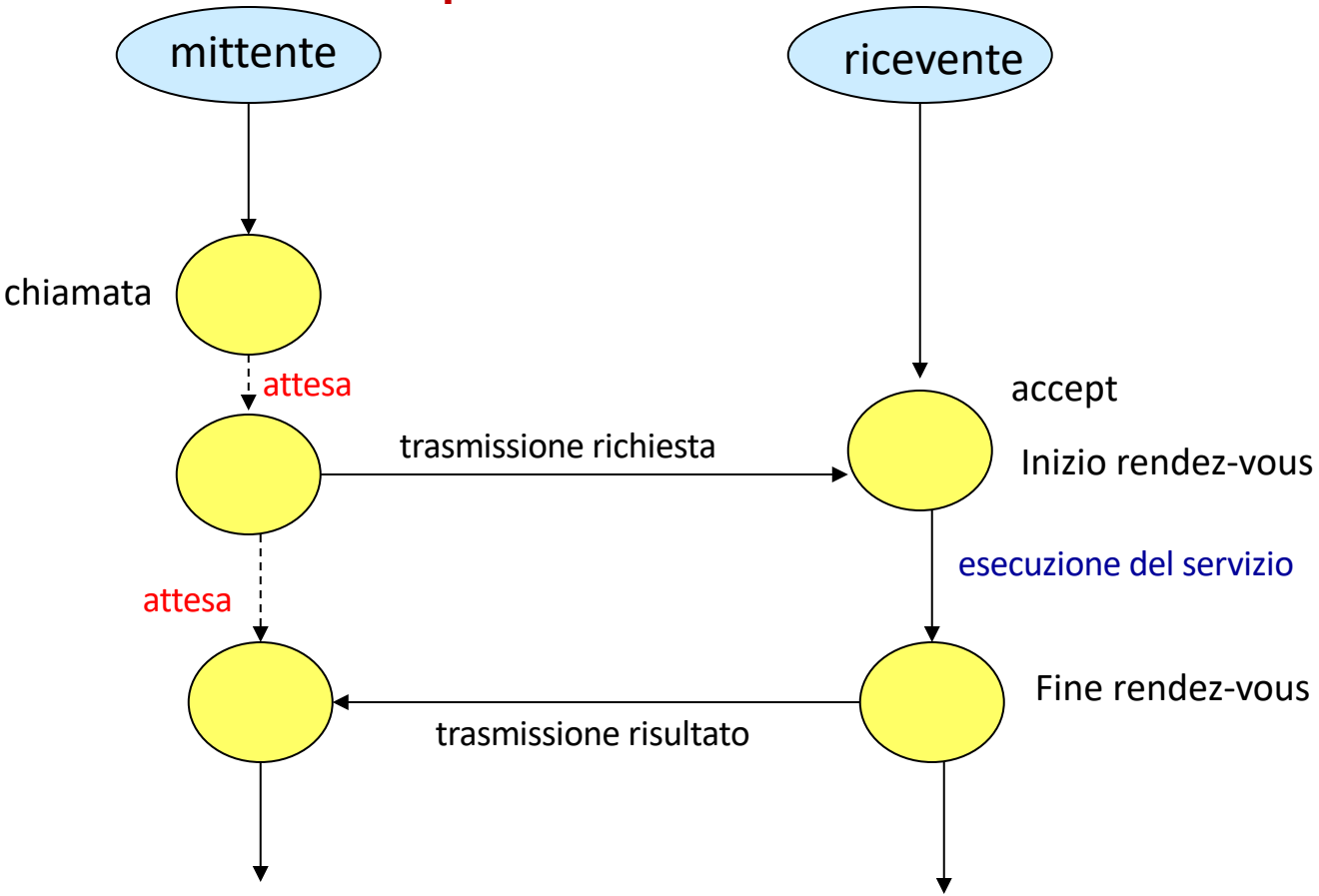
Semantica:

- Se non sono presenti richieste del servizio S l'accept è sospensiva.
- Se lo stesso servizio S è richiesto da più processi prima che il servitore esegua la accept, le richieste vengono inserite in una **coda** associata al servizio, gestita con politica FIFO.
- Ad uno stesso servizio possono essere associate più accept nel codice eseguito dal server -> ad una richiesta possono corrispondere azioni diverse.
- Lo schema di comunicazione realizzato dal meccanismo di rendez vous è di tipo asimmetrico da molti a uno.

Possibili sequenze di eventi in una comunicazione di tipo rendez-vous



Possibili sequenze di eventi in una comunicazione di tipo rendez-vous



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**:

```
select
   []<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, buffer di capacità N

```
process buffer          //server
{  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 (client) e processo chiamato (server) è 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>;
   }
}
```

Rendez-vous esteso: 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 (esempio: politiche di accettazione delle richieste basate su priorità). In questo caso, infatti, condizione per l'esecuzione dell'azione richiesta deve essere espressa anche in termini dei parametri di ingresso.

Ciò non è possibile tramite la guardia di ingresso: è 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 possibile richiesta una differente operazione (operazione di servizio)

Ciò è reso possibile in alcuni linguaggi dalla possibilità di aggregare le richieste possibili in strutture di tipo vettore: vettore delle operazioni di servizio (v. 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):attesa 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); //impost. preliminare dell'allarme
                                     //(non sospensiva)
   …
   allarme.svegliami[T]; //attesa del timeout
   ...
}
```

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 operazioni di servizio
  int tempo;
  typedef struct
  {     int risveglio;
        int intervallo;}dati_di_risveglio;
  dati_ di_risveglio tempo_di_sveglia[N];  // range indici [1,..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>;}
  }}
```

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.

- Comunicazione di tipo asimmetrico a rendez vous esteso.
- Ogni task può definire delle operazioni pubbliche (entry) visibili da altri task.
- Una entry definita in un task P e resa visibile all'esterno di P, può essere chiamata da un altro task Q.
- Il rendez-vous tra i due task viene stabilito quando un processo Q chiama una operazione entry di P.