Chiamata di procedura 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 di tipo *sincrono* in cui processo mittente e processo ricevente *rimangono sincronizzati* per tutto il tempo necessario all'esecuzione del servizio da parte del ricevente ed alla ricezione dei risultati da parte del mittente (*rendez-vous esteso*).

Con gli strumenti gia` visti, si puo` realizzare come segue:
- **lato chiamante**: *send asincrona* immediatamente seguita da una *receive*
- **lato chiamato:** una *receive* seguita, al termine dell'azione richiesta, da una *send*

- *Analogia semantica* (e spesso anche sintattica) con una normale chiamata di procedura (*chiamata di procedura remota*).
- Prevede il *trasferimento di controllo* tra processi che operano in ambienti separati secondo il modello a scambio di messaggi (*modello cliente –servitore*).

```
call <servizio> (parametri - ingresso, parametri - uscita);
```

dove *servizio* identifica univocamente *l'insieme di istruzioni ed il processo* che deve eseguirle per soddisfare la richiesta.

```
remote procedure <servizio> (in<parametri-ingresso>,
out<parametri di uscita>)
{... }
```

Due modelli:

1) Per ogni richiesta di servizio viene creata una nuova *istanza del processo* servitore provvedendo, laddove necessario, ad una *sincronizzazione* per accesso a variabili comuni *(thread). (*es. linguaggio DP*)*
2) Il servizio richiesto viene specificato come un *insieme di istruzioni* che può comparire in un *punto qualunque* del processo servitore. (es. linguaggio ADA)

Modello 1: lato server

Ipotesi: l'insieme delle procedure remote invocabili dai client 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.

Modello 1

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

```
call nome_del_modulo.op_i (<parametri attuali>);
```

il server serve la chiamata creando un thread che esegue l'operazione richiesta (op_i).

Ad ogni istante e` possibile che piu` thread concorrenti all'interno del modulo accedano a variabili interne.

Necessita` 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

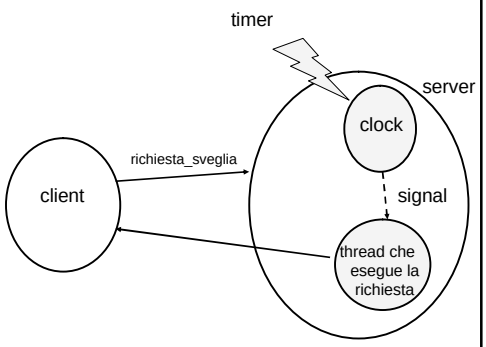
SERVER:

```
module allarme
{
int time;
semaphore mutex=1;
semaphore priv[N]=0;
coda_richieste coda; /* struttura contenente le richieste di
                        sveglia (sveglia, id) pervenute*/
    public void  richiesta_sveglia(int timeout, int id)
    {   int sveglia= time+timeout;
        wait(mutex);
    <inserimento  sveglia e id  nella coda di risveglio in modo da
    mantenere tale coda ordinata secondo valori  non decrescenti
    di sveglia>;
        signal(mutex);
        wait (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++;
   wait (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>;
   signal priv[id]; /* risveglio del processo id*/
   }
   signal(mutex);
 }
}/* fine modulo */
```

CLIENT:

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



Modello 2

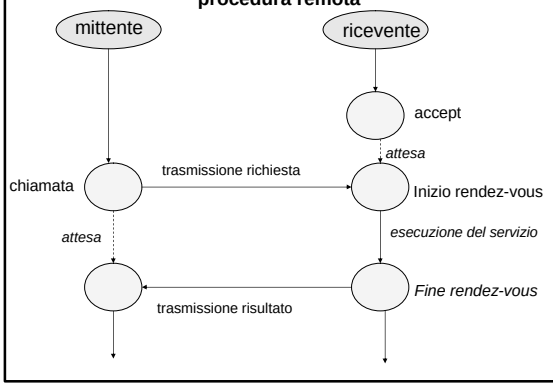
- Il servizio richiesto viene specificato come un *insieme di istruzioni* che può comparire in un *punto qualunque* del processo servitore (V. linguaggio ADA)

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

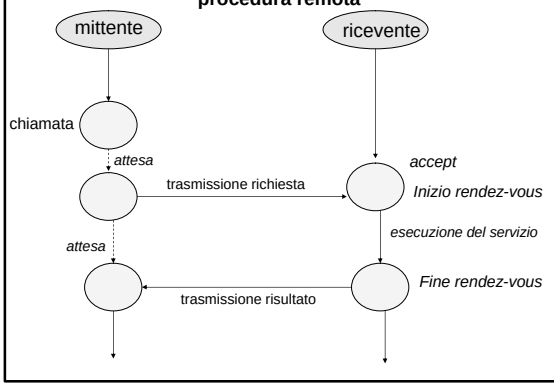
Accept

- 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 chiamata a procedura remota 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 secondo modello, il server puo` 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 puo` 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***) (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):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
{
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