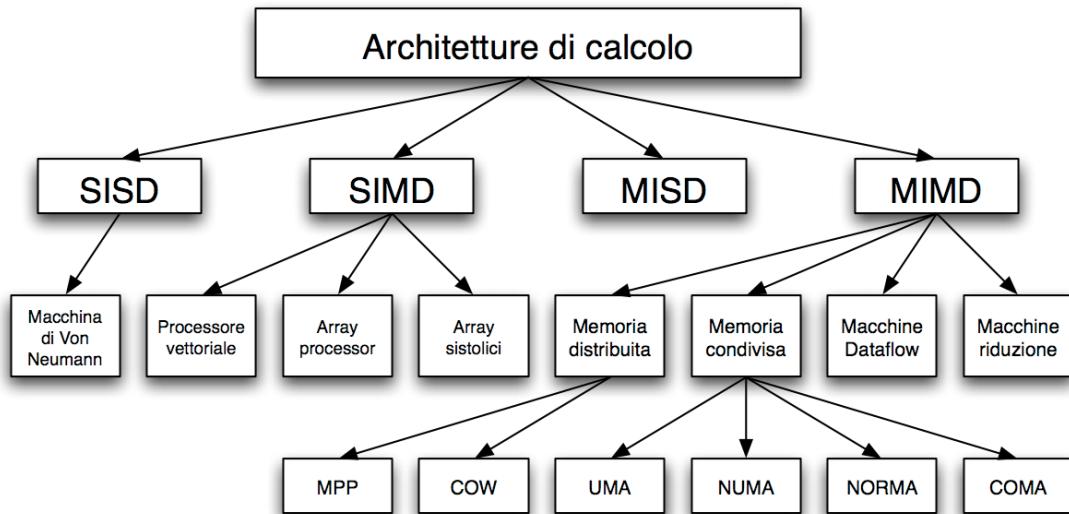


# SDC 2

## Tassonomia di Flynn

Classificazione dei calcolatori a seconda della molteplicità del flusso di dati e istruzioni che possono gestire contemporaneamente:

- **SISD** (single instruction single data) => nessun parallelismo, istruzioni sequenziali.
- **SIMD** (single instruction multiple data) => più processori eseguono la stessa istruzione su insiemi di dati differenti.
- **MISD** (multiple instruction single data) => più processori eseguono diverse istruzioni su stessi dati. (inutilizzata)
- **MIMD** (multiple instruction multiple data) => più processori eseguono diverse istruzioni su insiemi di dati differenti o su stessi dati, ovvero ho più processi indipendenti su processori differenti. (vedere Asymmetric multiprocessing e SMP, pagina 15)



## Logical clock

Il logical clock è un algoritmo di sincronizzazione che rende possibile la coordinazione nei sistemi distribuiti tenendo conto che in molte applicazioni non è importante conoscere quando accadono determinati eventi ma l'ordine in cui questi si susseguono.

La versione implementata da Lamport utilizza un registro (software) monotono crescente assegnato ad ogni processo per individuare il susseguirsi degli eventi.

Dato e evento ed  $L(e)$  il suo tempo logico, allora

se  $L(e) < L(e')$  =>  $e$  è avvenuto prima di  $e'$

## Algoritmi di sincronizzazione

Dati N processi questi ripeteranno il seguente schema di codice:

<non in sezione critica>

<trying protocol>

sezione critica

<exit protocol>

<non in sezione critica>

Uscito dal “exit protocol” il processo può rientrare infinite volte nel “trying protocol”.

Gli algoritmi che gestiscono l’accesso alla SC devono rispettare le proprietà:

- **mutua esclusione (ME)** => due processi non possono essere in sezione critica contemporaneamente
- **No deadlock (ND)** => se un processo rimane bloccato nella trying section è perché uno o altri processi stanno uscendo dalla sezione critica (non ci sono stalli)
- **No starvation (NS)** => nessun processo rimane bloccato nella trying section (extra)  
OSS NS => ND

## Algoritmo di Dijkstra

Shared variables

x[1..n]: array of Boolean, initially all false

y[1..n]: array of Boolean, initially all false

k: integer in range 1..N, initially any value in its range

Local variables

j: integer in range 1..N

repeat

```
1      NCS
2      y[i]:= true           % inizio trying protocol %
3      x[i]:= false
4      while k≠i do          % ciclo della sentinella %
5          if not y[k] then k:=i
6          x[i]:= true
7          for j:= 1 to n do
8              if i≠j and x[j] then goto 3    % fine trying protocol %
9          CS
10         y[i]:=x[i]:= false;           % exit protocol %
```

forever

---

### **Assunzioni:**

- I processi comunicano leggendo e scrivendo variabili comuni
  - Lettura e scrittura di una variabile sono azioni atomiche
- Non ci sono assunzioni sul tempo richiesto per un’operazione

## Protocollo:

1. Ciclo while (sentinella) => il processo **i** tenta di impostare **k** al suo id **i**, essendo **k** variabile condivisa tra i processi vale solo l'ultimo cambiamento effettuato dell'ultimo processo che vi ha acceduto => **k** indica l'ultimo valore, corrispondente al numero del processo, uscito dal ciclo.
2. Ciclo for => controlla che non ci siano stati passaggi concorrenti attraverso il primo ciclo testando che  $x[j] = \text{false}$  per ogni altro processo

## Dimostrazioni:

ME => si suppone per contraddizione che **i** e **j** sono in SC contemporaneamente.

Se **i** è entrato in SC vuol dire che ha trovato  $x[j] = \text{false}$  in linea 8, ovvero l'ha eseguita prima che **j** eseguisse la linea 6 e cambiasse il suo stato.

(a -> b indica che a è avvenuto prima di b)

Quindi  $i.8 \rightarrow j.6$  e dato che  $i.6 \rightarrow i.8$  implica  $i.6 \rightarrow j.6$ , per avere lo stesso risultato per **j** si dovrebbe avere  $j.6 \rightarrow i.6$  e quindi  $i.6 \rightarrow i.6 \Rightarrow \text{CONTRADDIZIONE}$

ND => si suppone per contraddizione di avere un insieme D di processi in deadlock.

Questo vuol dire che ogni processo **d** appartenente a D ha  $y[d] = \text{true}$ .

Supponendo che **i** sia l'ultimo processo ad aver assegnato **k**, per mantenerlo deve appartenere a D, altrimenti un altro processo **j** trovando  $y[i] = y[k] = \text{true}$  settierebbe nuovamente **k**.

Così facendo prima o poi ogni processo **d** in  $D/\{i\}$  avrà  $x[d] = \text{false}$  e si bloccherà nel ciclo while, ma questo implica che il processo **i**, saltato il while poiché vale ancora  $i = k$ , entrerà sicuramente in SC poiché tutti i valori  $x[d] = \text{false}$ .

MA se **i** entra in SC non può appartenere a D => CONTRADDIZIONE

NS => non supportata da questo algoritmo

## note:

Questo algoritmo può essere realmente implementato solo su architetture a singolo processore poiché nel caso di architetture multiprocessore non si può garantire che le operazioni di lettura e scrittura e siano operazione atomiche

## Algoritmo del panettiere di Lamport

### Shared variable

`num[1..n]`: array of integer, initially all 0

choosing[1..n]: array of Boolean, initially all false

%process  $i$  owns  $num[i]$  and  $choosing[i]$ %

Local variable

j: integer in range 1..N

repeat

1 NCS

2 choosing[i]:= true

3        num[i]:= 1+ max {num[j] : n≥ j ≥ 1}

4 choosing[i]:= false

5       for j:= 1 to n do begin

6           **while** choosing[i] **do** s

8 end

9 CS

10

Assunzioni:

- lettura e scrittura NON sono operazioni atomiche
  - Ogni variabile condivisa è di proprietà di un processo e solo lui può scriverci, gli altri possono solo leggerla
  - Nessun processo può emettere 2 scritture insieme
  - Le velocità dei processori non sono correlate

## Protocollo:

1. DOORWAY => il processo **i** segnala agli altri processi di esser entrato nella Doorway impostando choosing[i] = true, quindi verifica il numero di attesa maggiore ed imposta il suo ad uno di più, dopodiché inizializza nuovamente choosing prima di uscire  
OSS simile ad una panetteria dove l'ultimo arrivato prende l'ultimo numeretto
  2. BAKERY => il ciclo for permette di assicurarsi che il processo **i** sia il prossimo a dover accedere alla sezione critica, tramite:
    - While L6 => controlla che nessun altro processo stia eseguendo operazioni di scrittura, permette quindi a tutti i processi di terminare la Doorway correttamente. Serve per assicurarsi che nessun processo **j** sia stato schedulato tra le L3 e la L4 e quindi abbia in realtà un numero di attesa minore di quello di **i**.
    - While L7 => pone in attesa il processo **i** finché questo non arriva ad avere il numero d'attesa più piccolo, se due processi hanno lo stesso numero passa prima quello con ID più piccolo

## Dimostrazioni:

ME => dati 2 processi **i** e **j**, se **i** è nella Bakery e **j** nella Doorway allora vale la proprietà  $\{num[i],i\} < \{num[j],j\}$ , quindi ipotizzando per assurdo che entrambi si trovino in SC si avrebbe che  $\{num[i],i\} < \{num[j],j\}$  e  $\{num[j],j\} < \{num[i],i\}$  contemporaneamente => ASSURDO

ND => data da NS => ND

NS => num garantisce che prima o poi ogni processo entrerà in SC

## Note:

Questo algoritmo inoltre gode della proprietà SCFS (first-come-first-served)

## Algoritmo di Lamport per sistemi distribuiti

Local variable

```
j: integer in range 1,..N; num: integer, initially all 0;  
choosing: Boolean, initially false;  
repeat  
    1      NCS  
    2      choosing:= true          %inizio doorway%  %inizio trying%  
    3      for j≠i do  
        4          send num to pj  
        5          receive reply(v) from pj  
        6          num:=max(v,num)  
        7          num:= num+1  
    8      choosing:= false          %fine doorway%  
    9      for j:= 1 to n do begin  
    10         repeat  
    11             send choosing to pj  
    12             receive reply(v) from pj  
    13             until v  
    14             repeat  
    15                 send num to pj  
    16                 receive reply(v) from pj  
    17                 until v=0 or {num,i}>{v,j }  
    18     end                      %fine bakery%  %fine trying%  
    19     CS  
    20     num:=0;                  %exit protocol%  
forever
```

---

Upon the arrival of a num message from process j

send reply(num) to process j

Upon the arrival of a choosing message from process j

send reply(choosing) to process j

---

### Assunzioni:

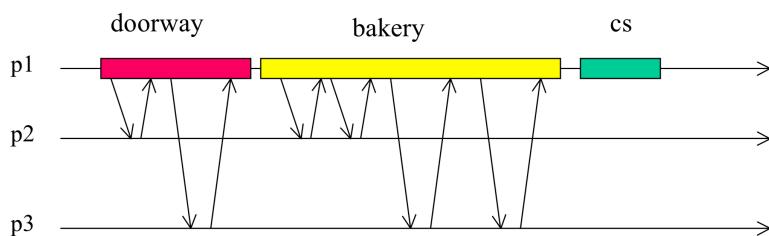
- lettura e scrittura NON sono operazioni atomiche
- Ogni variabile condivisa è di proprietà di un processo e solo lui può scriverci, gli altri possono solo leggerla
- Nessun processo può emettere 2 scritture insieme
- Le velocità dei processori non sono correlate
  - I processi leggono e scrivono messaggi attraverso il message passing con un ritardo indefinito ma comunque finito
  - I canali sono affidabili, ogni messaggio arriva a destinazione

Per lo più uguali a versione base

### Protocollo:

- I. DOORWAY => il processo i segnala agli altri sistemi di esser entrato nella Doorway impostando il suo choosing = true, quindi richiede a tutti gli altri sistemi i numeri d'attesa ed imposta il suo ad uno di più del maggiore, dopodiché inizializza nuovamente choosing prima di uscire
- II. BAKERY => il ciclo for permette di assicurarsi che il processo i sia il prossimo a dover accedere alla sezione critica, tramite:
  - L10 - L13 => controlla che nessun altro processo stia eseguendo operazioni di scrittura, permette quindi a tutti i processi di terminare la Doorway correttamente. Serve per assicurarsi che nessun processo j sia stato schedulato tra le L3 e la L4 e quindi abbia in realtà un numero di attesa minore di quello di i.
  - L14 - L17 => pone in attesa il processo i finché questo non arriva ad avere il numero d'attesa più piccolo, se due processi hanno lo stesso numero passa prima quello con ID più piccolo

OSS Concettualmente uguale a versione base ma con l'aggiunta di funzioni send e receive per effettuare lo scambio dei messaggi che potrebbero rallentare notevolmente le interazioni, inoltre implica la creazione di thread concorrenti per le interazioni multiple



### Note:

Detto non cooperante poiché ad ogni interazione il processo generico i (server), che vuole definire il proprio numero d'attesa, invia e riceve dati da ogni altro processo nel sistema (clients) che però non hanno alcun ruolo attivo nella scelta.

Questo scambio non cooperante implica un rallentamento notevole dell'algoritmo, può essere quindi ottimizzato tramite una versione decentralizzata peer-to-peer come l'algoritmo di Ricart-Agrawala

## Algoritmo di Ricart-Agrawala

---

Local variable

```
#replies: integer initially set to zero  
state: in set {requesting, CS, NCS} initially set to NCS  
Q: queue of pending request {T,i} initially set to empty  
Last_req: integer ; initially set to maxint  
Num: integer initially set to zero  
  
begin  
    1 state:= requesting  
    2 num:=num+1; last_req:=num;  
    3 send REQUEST(last_req) to p1..pn  
    4 wait until #replies=(n-1)  
    5 state:= CS  
    6 CS  
    7 send REPLY to any request in Q; Q:=∅; state:=NCS; #replies:=0; lastreq:=maxint;  
end
```

Upon the receipt of REQUEST(t) from process j

```
8     num:=max(t,num)  
9     if state=CS or (state=requesting and {last_req, i } < {t,j})  
10    then insert.Q({t, j})  
11    else send REPLY to Pj
```

Upon the receipt of REPLY from process j

```
12. #replies++;
```

---

### Assunzioni:

- lettura e scrittura NON sono operazioni atomiche
- Ogni variabile condivisa è di proprietà di un processo e solo lui può scriverci, gli altri possono solo leggerla
- Nessun processo può emettere 2 scritture insieme
- Le velocità dei processori non sono correlate
- I processi leggono e scrivono messaggi attraverso il message passing con un ritardo indefinito ma comunque finito
- I canali sono affidabili, ogni messaggio arriva a destinazione
- La linea 2 viene eseguita atomicamente

Una sola assunzione in più rispetto alla versione non cooperante

### **Protocollo:**

In questa versione ogni processo invece di chiedere a tutti qual'è il numero più grande ogni volta che vuole entrare in SC per poi calcolare il suo, trasmette direttamente il suo numero agli altri processi utilizzando un contatore interno.

Questo è possibile poiché ad ogni richiesta da parte di un processo tutti i sistemi aumentano il loro contatore interno.

1. L1 - L3 => una volta modificate le variabili interne il sistema invia la richiesta contenente il suo numero agli altri sistemi  
*OSS aumenta num subito per evitare di perdere il posto nel caso dovesse ricevere una richiesta da un altro processo prima di completare il suo accesso alla SC*
2. L4 => il processo quindi si mette in attesa finché non riceve un REPLY da ogni altro sistema, una volta collezionati tutti gli N-1 REPLY potrà accedere alla SC

La mutua esclusione è quindi possibile poiché nel caso un cui un processo è già in sezione critica oppure ha un numero inferiore ritarda il REPLY finché il suo turno non è finito, le richieste vengono inserite nella coda Q

### **Dimostrazione:**

ME => si suppone per contraddizione che **i** e **j** sono in SC contemporaneamente, quindi **i** ha inviato un REPLY a **j** e viceversa. Sono quindi possibili 3 casi:

1. **i** spedisce REPLY a **j** prima di scegliere il proprio numero in L2, poi invia una richiesta che quindi avrà un numero con valore più alto rispetto a quella di **j**.  
Quando la richiesta di **i** arriva a **j** questo la inserisce nella sua coda Q, poiché il suo numero ha un valore più basso (oppure è in già in SC).  
Di conseguenza **i** e **j** non possono essere entrambi in SC insieme.
2. identico invertendo **i** e **j**
3. sia **i** che **j** inviano un REPLY all'altro dopo aver scelto il num, a questo punto uno dei due invia un REPLY all'altro e l'altro lo mette in coda.  
Di conseguenza **i** e **j** non possono essere entrambi in SC insieme.

ND => data da NS => ND

NS => si suppone per contraddizione che **i** dopo aver inviato la richiesta con il suo num attenda indefinitivamente l'accesso alla SC.

Poiché i messaggi arrivano a destinazione in tempo finito questo vuol dire che un set non vuoto di processi **S** non invia il messaggio REPLY, ma questo può accadere solo se esiste almeno un processo **j**  $\in$  **S** che si trova in SC oppure ha un numero d'attesa inferiore.

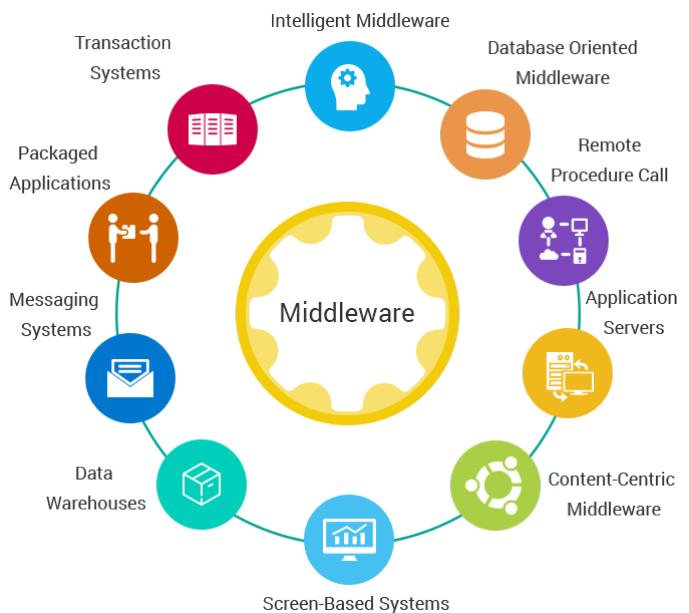
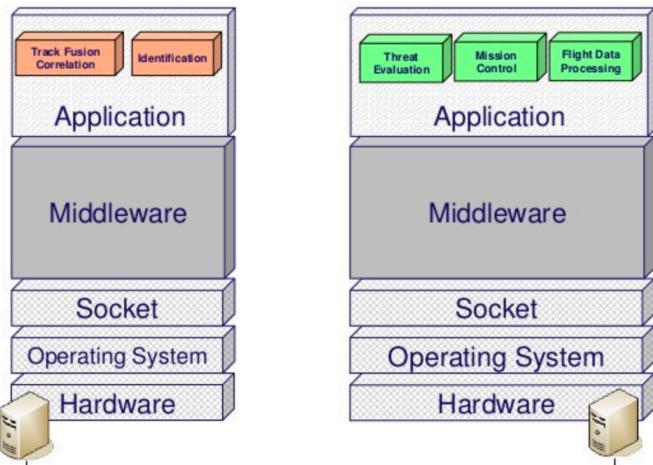
Quindi si possono avere 2 casi:

1. Nel primo caso **j** esce dalla SC ed invia il REPLY e sbloccando **i**.
2. Nel secondo esiste un altro processo **k**  $\in$  **S**/**{j}** che blocca **j**, quindi si crea una versione ricorsiva del primo caso

## La rete

### Middleware

Il middleware consiste in un set di strumenti software (API) interposti tra sistema operativo e applicazioni che forniscono le astrazioni per il dialogo tra sistemi eterogenei<sup>1</sup>

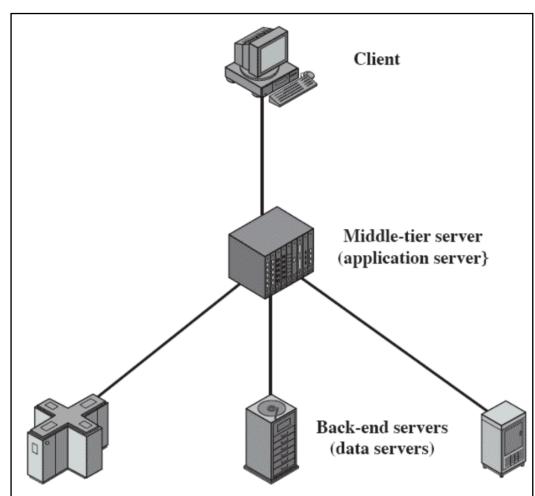
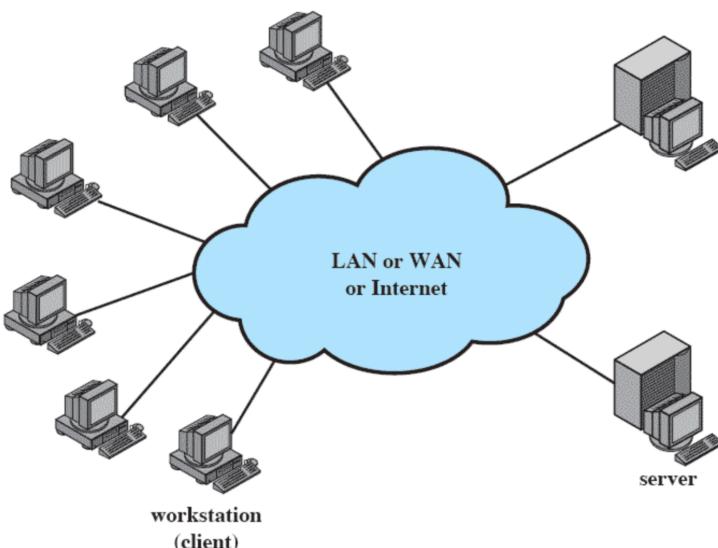


### Client/server computing and application

Interazione generalmente usata dai database, i dispositivi connessi sono di 2 tipi:

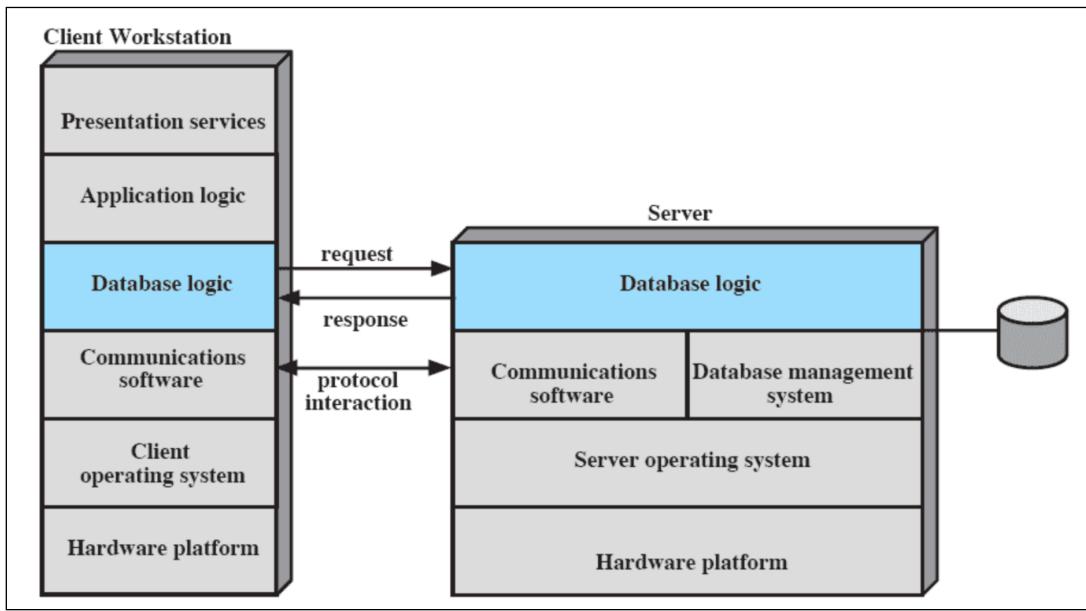
- **Server** => high performance computer che provvede allo scambio dei dati e permette la connessione tra i client, esegue le operazioni di computazione sui dati e poi li invia al server.
- **Clients** => single-user PCs che si collegano al server al quale inviano e ricevono dati, permettono interfaccia user-friendly

OSS in alcuni casi si può avere una struttura a 3 livello che aggiunge anche un Middle-tier server che funge da middleware e si occupa della distribuzione del carico su più server e la protezione degli accessi

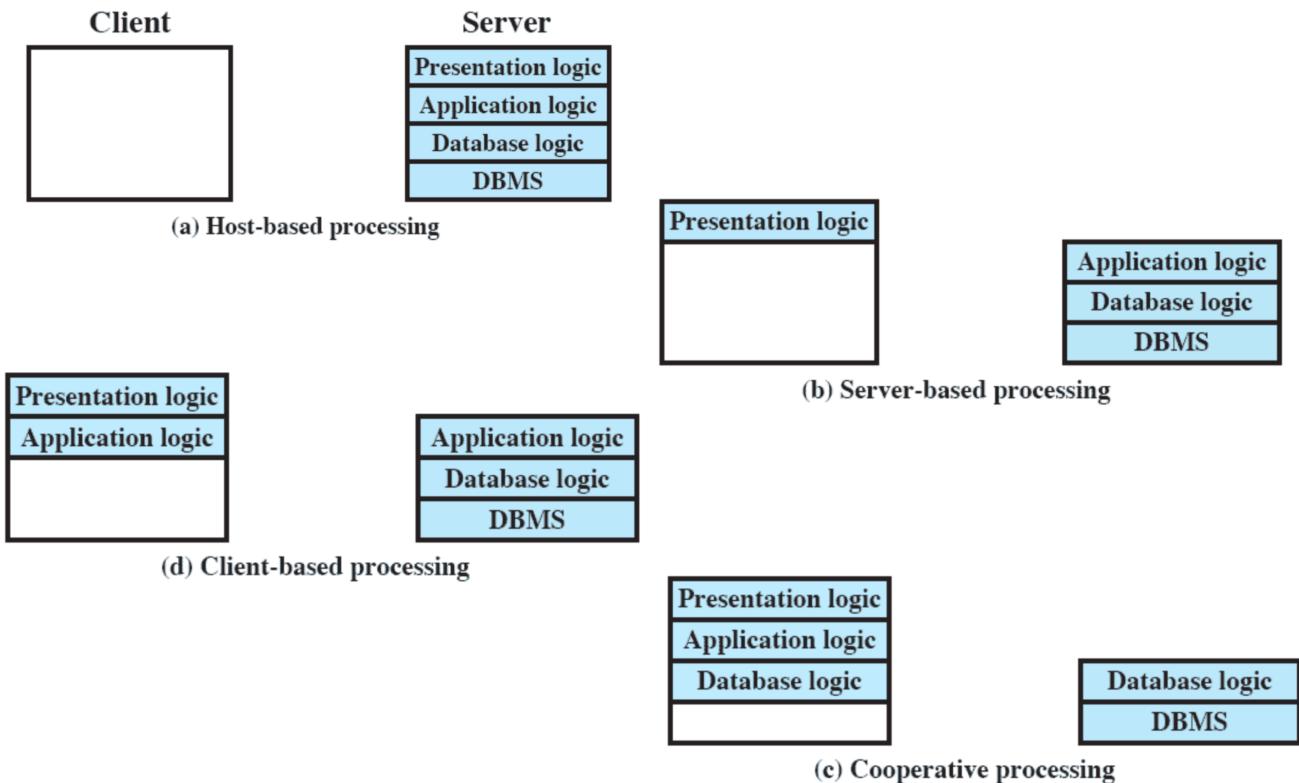


<sup>1</sup> sistemi eterogenei = sistemi con hardware, sistema operativo e protocolli differenti

## Classi applicazioni client/server



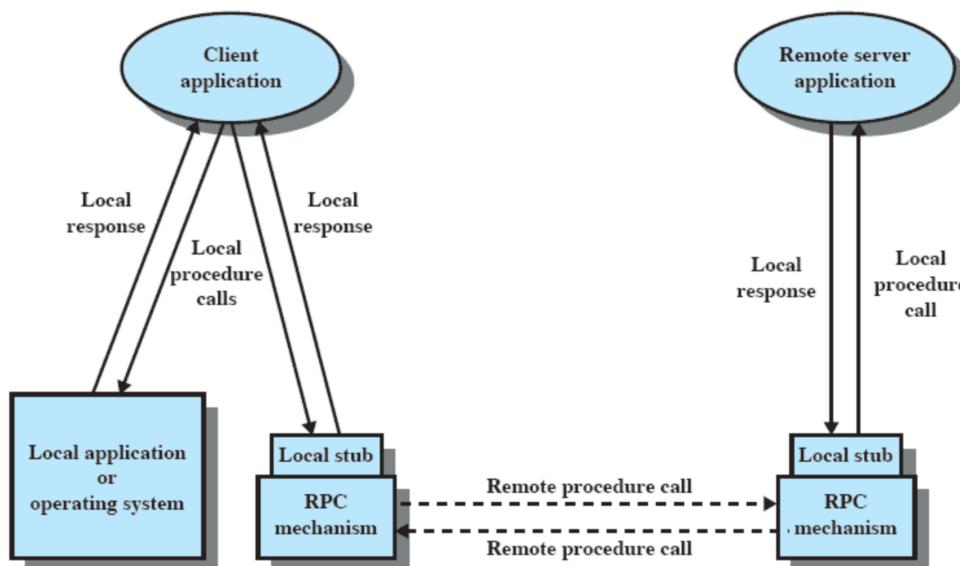
- **Host-based** => il client serve solo per mostrare le informazioni, tutta la logica è affidata al server
- **Server-based** => il client in questo caso si occupa anche della componente grafica ma il resto delle operazioni viene gestito solo dal server
- **Client-based** => il client si occupa della grafica e delle operazioni base che però devono comunque essere ampliate dalla componente server
- **Cooperative** => parte del server viene caricato sul client rendendo operazioni più veloci



## RPC

Trasforma l'interazione client/server in una chiamata a procedura nascondendo al programmatore i meccanismi implementativi che la compongono, come:

- Interscambio dei messaggi
- Localizzazione del server
- Possibili protocolli differenti tra le macchine

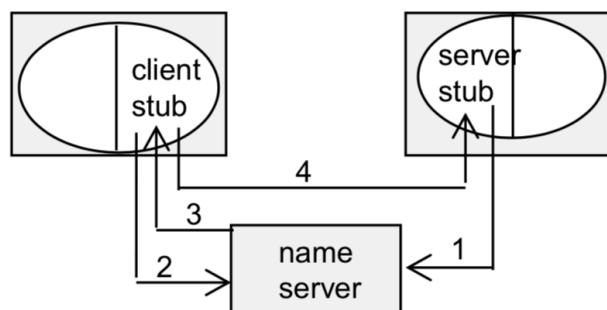


Il **mascheramento** avviene in 3 fasi:

1. Tempo di **scrittura del codice** => RPC usate vengono dichiarate esplicitamente dal programmatore attraverso import/export dalle interfacce
2. Tempo di **compilazione** => durante la compilazione vengono associate le linee di codice dello stub che permettono le operazioni standard sui dati e le chiamate al **RPC runtime-support**
3. Tempo di **esecuzione** => tramite RPC runtime-support vengono eseguite le funzioni nascoste come localizzazione del server e registrazione nuovi servizi

## Metodi per la localizzazione del server

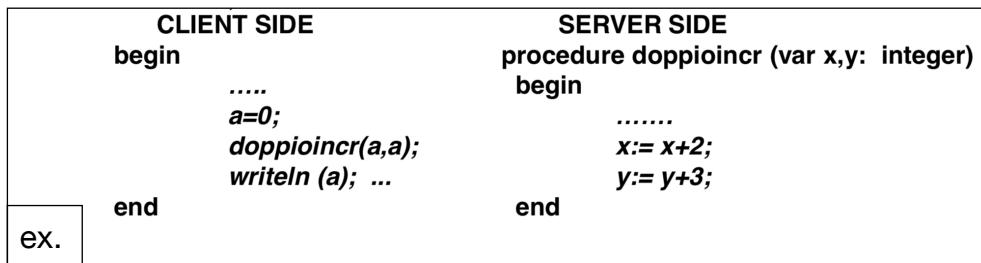
- Metodo **statico** => IP del server cablato nel client
- Metodo **dinamico** => lo stub del client, mentre impacchetta i dati, invia concorrentemente un broadcast richiedendo l'indirizzo di una macchina in grado di eseguire la RPC desiderata
- **Name server** => il collegamento tra client e server viene effettuato mediante il name server che alla richiesta di una RPC del client consulta la sua tabella interna e gli fornisce un server in grado di eseguirla



## Passaggio dei parametri

Effettuato dallo stub

- **Call by reference** => passaggio di un puntatore (**SCONSIGLIATO**)
- **Call by copy/restore** => copia diretta dei valori



Con "call by ref" si ha "a = 5" poiché sullo stesso calcolatore si ha che x = a e y = a quindi a = 0 + 2 + 3 = 5

Con "call by copy/restore" il risultato è randomico poichè si ha che solo x oppure y verrà usato come valore di ritorno di a

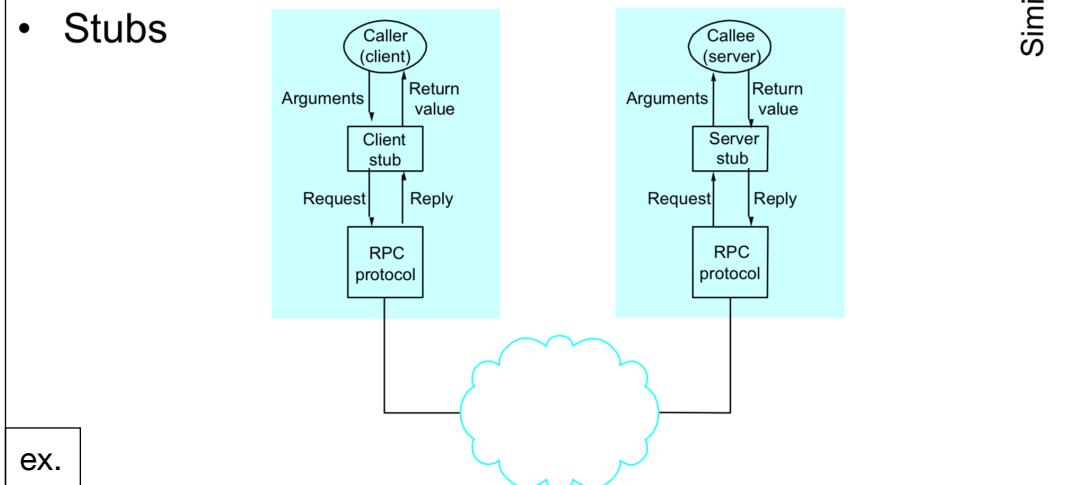
## Possibili semantiche delle RPC

- **At least once** => il client esegue n volte la richiesta, il server ne riceve n
- **At most once** => il client esegue la richiesta al massimo una volta e non è detto che server la riceva, nel caso di più tentativi ritorna codice d'errore
- **Exactly once** => equivalente a chiamata locale, implica:
  - Server => immagazzina tutti i risultati nel logger, se la chiamata è già stata ricevuta riprende il risultato dal logger
  - Client => numera tutte le richieste e ne tiene il conto tramite il numero di reincarnazione che viene inviato in caso di guasto della RPC

## Sottosistema di comunicazione

Il protocollo usato a livello di trasporto dalle RPC è UDP, il quale però non implementa controlli sulla consegna dei pacchetti rendendo necessario l'utilizzo di protocolli supplementari. (ex. BLAST, CHAN e SELECT)

- **Protocol Stack**
  - BLAST: **fragments** and reassembles large messages
  - CHAN: synchronizes request and reply **messages** (at most once semantic)
  - SELECT: dispatches request to the **correct process**
- **Stubs**



## Protocollo BLAST

Divide (e riassembra) i pacchetti e li invia tramite UDP gestendo i riscontri.

È un protocollo persistente, ovvero continua a richiedere la trasmissione dei frammenti mancanti, nel complesso però non si occupa di garantire l'invio dell'intero messaggio poiché invia (e riceve) solo le componenti dettate da CHAN.

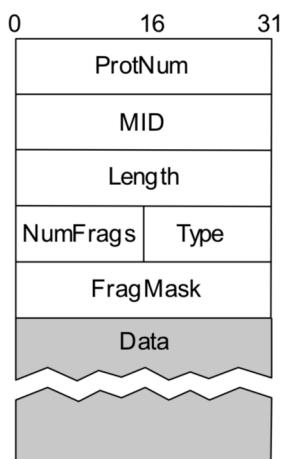
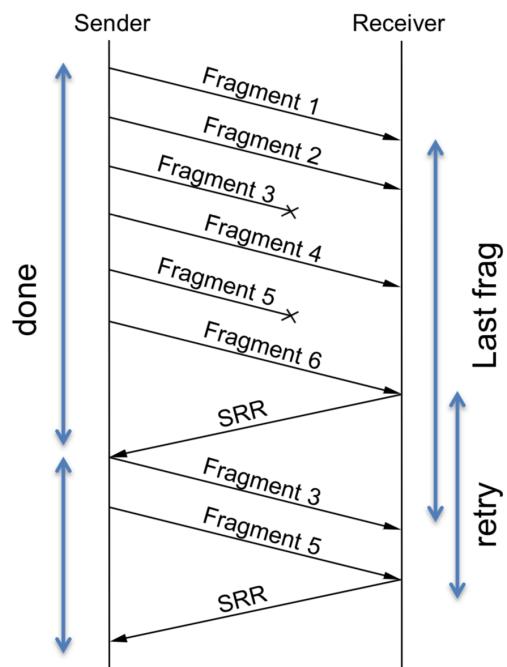
### protocollo:

#### Lato sender (client):

1. Salva i frammenti nella memoria locale
2. Invia tutti i frammenti e setta timer = DONE
- IF riceve SRR invia pacchetti mancanti e setta timer = DONE
- IF riceve SRR speciale "all fragments received" setta timer = DONE
- IF timer scade rinuncia, libera la memoria ed avvisa CHAN

#### Lato receiver (server):

1. Al primo frammento setta timer = LAST\_FRAG
- IF riceve tutti i dati, rilascia e va a strato superiore
- ELSE eccezioni:
  - Ultimo frammento arrivato, messaggio incompleto  
=> invia SRR e timer = RETRY
  - LAST\_FRAG scade  
=> invia SRR e timer = RETRY
  - RETRY scade per la seconda volta  
=> invia SRR e timer = RETRY
  - RETRY scade per la terza volta  
=> rinuncia e libera la memoria



#### Formato HEADER:

MID => intestazione, la stessa per ogni pacchetto  
 NumFrags => numero dei frammenti  
 TYPE => dati o SRR  
 FragMask => identifica i frammenti

## Protocollo CHAN

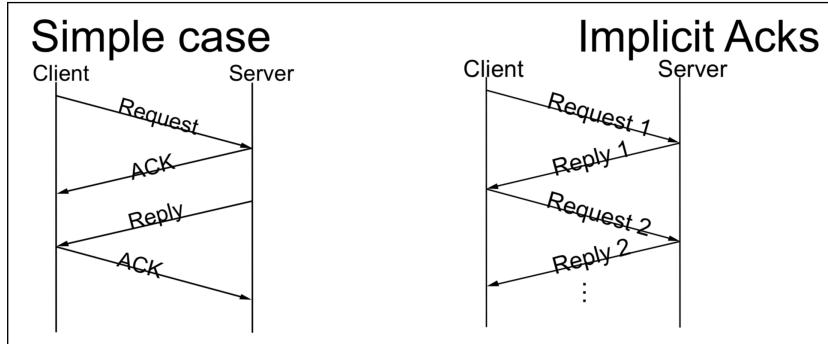
Serve per garantire l'invio dei messaggi (richiamando BLAST) e sincronizzare il client con il server, supporta semantica "at most once".

Gestisce la perdita dei messaggi memorizzando ognuno di essi fino all'arrivo dell'ACK corrispondente per il riscontro

### protocollo:

1. Invia messaggio e setta timer = RETRANSMIT
2. IF timer scade ed ACK non è arrivato  
=> rinvia il messaggio e setta nuovamente timer = RETRANSMIT

OSS questo implica che il ricevente deve controllare i casi di duplicazione dei messaggi tramite il campo MID



## Protocollo SELECT

Dispatcher, ovvero assegna i dati alla giusta RPC.

La distinzione tra le chiamate si può fare tramite:

- **flat** => id unico per ogni procedura
- **Hierarchical** => id programma + numero della procedura

## Formattazione dei dati

Avendo che chiamante e chiamato lavorano su macchine differenti si potrebbero avere problemi per la formattazione dei dati, del tipo:

- Interi => little endian / big endian
- Floating point => IEEE754 / non standard

Si usano quindi Forme canoniche (standard) di invio dei pacchetti:

- eXternal Data representation (XDR) => usata per le prime RPC (SunRPC), untagged<sup>2</sup>
- Abstract syntax notation (ASN) => usato per ISO standard, tagged<sup>3</sup>
- Network data representation (NDR) => utilizza il receiver-makes-right, ovvero è il ricevitore ad interpretare la codifica

0	4	8	16	24	31
IntegrRep	CharRep	FloatRep	Extension 1	Extension 2	

- IntegerRep
  - 0 = big-endian
  - 1 = little-endian
- CharRep
  - 0 = ASCII
  - 1 = EBCDIC
- FloatRep
  - 0 = IEEE 754
  - 1 = VAX
  - 2 = Cray
  - 3 = IBM

<sup>2</sup> untagged => non esiste una struttura prestabilita del pacchetto

<sup>3</sup> tagged => nel pacchetto vengono specificate: TYPE, LEN, VALUE

# Sicurezza informatica

## 3 principi sicurezza informatica

- **confidenzialità** => riservatezza dei dati privati o informazioni confidenziali che devono essere accessibili solo se strettamente necessario ad individui autorizzati
- **integrità** => le informazioni devono poter essere modificabili solo da persone autorizzate
- **disponibilità** => il servizio deve essere sempre disponibile agli utenti autorizzati

EXTRA

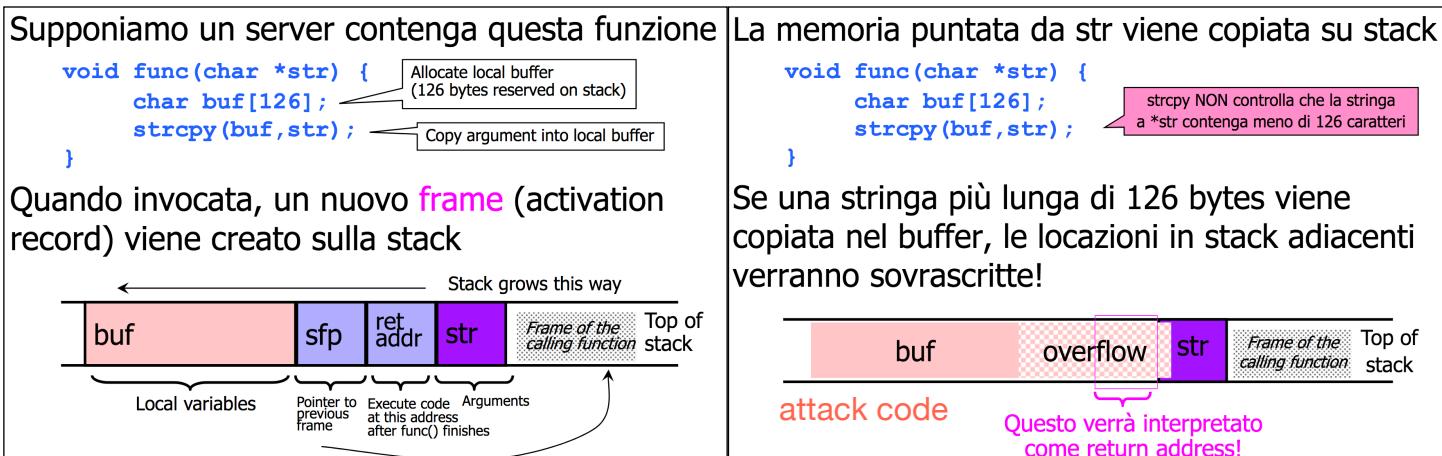
- **Autenticità** => verifica delle credenziali degli utenti
- **Tracciabilità** => possibilità di tracciare le azioni svolte dagli utenti

## + Minacce principali

- **Exploit** => codice malevolo eseguito sfruttando vulnerabilità del sistema
- **Malware** => malicious software, termine generale per indicare codice malevolo
- **Backdoor** => punto di accesso segreto che permette accesso non autorizzato
- **Logic bomb** => esplode in determinate condizioni, come esecuzione di applicazione
- **Trojan horse** => codice malevolo nascosto in programmi normali
- **Mobile code** => codice trasmesso in remoto ed eseguito senza autorizzazione esplicita dell'utente ricevente
- **Bots / zombie** => programmi che eseguono attacchi in automatico attraverso internet
- **Virus** => software che infetta altro software modificandone le funzioni, ha 4 fasi:  
Dormiente => Propagazione => Attivazione => Esecuzione
- **Worms** => malware in grado di autoreplicarsi attraverso la rete, generalmente con mail

## + Buffer overflow (ex. di exploit)

I buffer sono porzioni di memoria per immagazzinare dati, il buffer overflow sfrutta le vulnerabilità di sistemi che li usano per fornire codice malware al posto dei dati richiesti.



Conoscendo la lunghezza del buffer è facile intuire la posizione esatta dell'indirizzo di ritorno della funzione.

A questo punto, tramite un overflow sul buffer, si potrà eseguire un **override** dell'indirizzo di ritorno della funzione con l'indirizzo iniziale del buffer nel quale è stato precedentemente scritto il codice "**attack code**" del malware.

Per ovviare a questo sistema si può usare la funzione **strncpy**, che al contrario di strcpy (o gets, scanf, ...) controlla anche la dimensione del buffer prima di effettuare la copia.