Università di Messina — CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

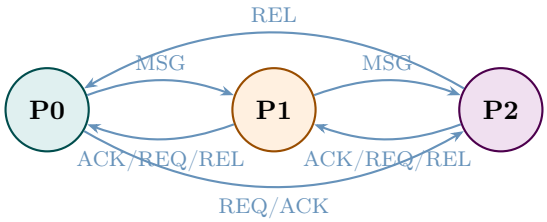
Hands-On #17

Clock logici e mutex distribuito in C

Last Update: 14 maggio 2026

Scenario

In questo HO implementiamo in C i tre meccanismi studiati a lezione, partendo dalla struttura più semplice e aggiungendo complessità ad ogni esercizio.



Ogni *nodo* è un **thread POSIX**. I nodi non condividono variabili di stato: comunicano esclusivamente depositando messaggi nella coda di ingresso del destinatario. Questa scelta forza il rispetto del modello a scambio di messaggi e rende impossibile "sbirciare" il clock del vicino.

Infrastruttura condivisa (`infra.h`)

Il file `infra.h` è già pronto e non va modificato. Fornisce:

- la struttura `Msg` (tipo, mittente, timestamp Lamport, timestamp vettoriale, payload);
- la struttura `Queue`: coda FIFO protetta da mutex e condition variable;
- l'array globale `queues[N]`: una coda per nodo;
- `send_msg(dst, m)` — deposita `m` nella coda di `dst` (non bloccante salvo coda piena);
- `recv_msg(id)` — preleva il prossimo messaggio dalla coda del nodo `id` (**bloccante** se la coda è vuota).

Listing 1: Estratto di infra.h — tipi e API pubblica

```c
#define N       3         /* numero di nodi                    */
#define MSG_DATA    1     /* messaggio dati (es. 1 e 2)        */
#define MSG_REQUEST 2     /* richiesta SC   (es. 3)            */
#define MSG_ACK     3     /* ACK a REQUEST  (es. 3)            */
#define MSG_RELEASE 4     /* rilascio SC    (es. 3)            */

typedef struct {
    int type;             /* MSG_*                             */
    int from;             /* ID mittente                       */
    int ts;               /* timestamp Lamport                 */
    int vec[N];           /* timestamp vettoriale              */
    int payload;          /* dato opzionale                    */
} Msg;

void send_msg(int dst, Msg m);   /* invia m al nodo dst        */
Msg  recv_msg(int id);           /* ricevi (bloccante)          */
```

▷ **Perché `recv_msg` è bloccante?**

Se la coda è vuota, il thread si sospende sulla condition variable e riprende solo quando arriva un messaggio. Non si spreca CPU in attesa attiva (*busy waiting*). Negli esercizi 1 e 2 ogni `recv` corrisponde esattamente a un `send` precedente, quindi non c'è rischio di deadlock. Nell'esercizio 3 la stessa proprietà vale perché ogni REQUEST genera esattamente un ACK per mittente.

Come funziona la coda internamente

Vale la pena leggere `infra.h` riga per riga almeno una volta, perché la struttura della coda riappare in molti contesti reali (pool di thread, broker di messaggi, driver di dispositivo).

Il buffer circolare. La coda è un array fisso `buf[QSIZE]` con tre indici: `head` punta al prossimo da leggere, `tail` a dove scrivere il prossimo, `count` tiene il conto degli elementi presenti. Quando `tail` raggiunge la fine dell'array riparte da zero con `% QSIZE`.

Listing 2: `q_push` — deposita un messaggio in coda

```
pthread_mutex_lock(&q->lock);
while (q->count == QSIZE)          /* coda piena? aspetta        */
    pthread_cond_wait(&q->ready, &q->lock);
q->buf[q->tail] = m;
q->tail = (q->tail + 1) % QSIZE;   /* avanza in circolare        */
q->count++;
pthread_cond_broadcast(&q->ready);  /* sveglia chi era in attesa */
pthread_mutex_unlock(&q->lock);
```

Il `while` al posto di `if` è fondamentale: quando un thread si risveglia da `cond_wait` non è garantito che la condizione sia ancora vera — un altro thread potrebbe aver riempito di nuovo la coda nell'intervallo. Bisogna ricontrollare.

Listing 3: `q_pop` — preleva un messaggio dalla coda

```
pthread_mutex_lock(&q->lock);
while (q->count == 0)              /* coda vuota? aspetta        */
    pthread_cond_wait(&q->ready, &q->lock);
Msg m = q->buf[q->head];
q->head = (q->head + 1) % QSIZE;
q->count--;
pthread_cond_broadcast(&q->ready);
pthread_mutex_unlock(&q->lock);
return m;
```

`q_pop` è speculare: se la coda è vuota il thread si addormenta e viene svegliato dalla prima `q_push` successiva.

I mutex violano le ipotesi dei sistemi distribuiti?

La risposta è no. Nei sistemi distribuiti reali i nodi sono macchine separate collegate da una rete: non esiste memoria condivisa tra loro. Qui usiamo thread sullo stesso processo, quindi la memoria condivisa esiste fisicamente. I mutex in `infra.h` proteggono la struttura interna della coda — `head`, `tail`, `count` — dagli accessi simultanei. Senza di loro, due thread che chiamano `send_msg` nello stesso istante corromperebbero il buffer.

I mutex sono un dettaglio implementativo del **mezzo di trasporto**, non del modello applicativo. Pensali come l'elettronica di una scheda di rete: garantisce l'integrità dei bit sul cavo senza che il programmatore lo sappia.

Il modello distribuito rimane integro finché ogni nodo rispetta una regola sola: accede solo al proprio stato locale e comunica con gli altri esclusivamente tramite `send_msg` e `recv_msg`.

[!] Cosa violerebbe davvero il modello

Queste operazioni sono fisicamente possibili con i thread ma concettualmente vietate, perché non sarebbero realizzabili su macchine separate:

```
/* VIETATO: sbirciare il clock di un altro nodo */
if (clk[1] > clk[0]) { ... }

/* VIETATO: leggere la coda di un altro nodo */
if (pq[1].size > 0) { ... }
```

Non farlo. Il protocollo corretto non ne ha bisogno.

▷ Il confine è portabile

Se sostituissi le code con socket TCP tra processi su macchine diverse, il codice degli esercizi — `lc_event`, `lc_send`, `lc_recv`, `request_sc` e così via — non cambierebbe di una riga. Cambierebbe solo `infra.h`. Questo è il principio di separazione tra logica applicativa e strato di trasporto.

Esercizio 1: Clock di Lamport

Obiettivo

Implementare le tre regole del clock di Lamport e verificare sperimentalmente che la proprietà $a \to b \;\Rightarrow\; C(a) < C(b)$ sia soddisfatta per le coppie causali dello scenario.

Scenario di comunicazione

Il file `es1_lamport.c` contiene già le funzioni `node0`, `node1`, `node2` con il seguente scenario fisso:

Nodo	Sequenza di eventi
P0	evento interno a → **send a P1** → evento interno b
P1	evento interno c → **recv da P0** → **send a P2**
P2	**recv da P1** → evento interno d

La catena causale garantita è: $a \to send_{0\to1} \to recv_1 \to send_{1\to2} \to recv_2$. L'evento c di P1 è *concorrente* con a (nessun messaggio li collega).

Cosa implementare

Compito — `es1_lamport.c`

1. `lc_event(id)` — regola R1. Incrementa `clk[id]` e stampa il nuovo valore. Restituisce il nuovo clock.
2. `lc_send(id, dst, payload)` — regola R2. Incrementa `clk[id]`, costruisce un `Msg` con `type=MSG_DATA`, `from=id`, `ts=clk[id]`, lo deposita con `send_msg`. Restituisce il timestamp allegato.
3. `lc_recv(id)` — regola R3. Preleva un messaggio con `recv_msg(id)`, aggiorna `clk[id] = max(clk[id], msg.ts) + 1`. Restituisce il messaggio.

4. `verify()` — dopo il join di tutti i thread, controlla che `saved.send_p0_p1 < saved.recv_p1` e `saved.send_p1_p2 < saved.recv_p2`. Stampa `PASS` o `FAIL` per ciascuna coppia.

Output atteso

```
[P0] evento interno    C = 1
[P1] evento interno    C = 1
[P0] send -> P1        C = 2
[P0] evento interno    C = 3
[P1] recv <- P0        C = 3
[P1] send -> P2        C = 4
[P2] recv <- P1        C = 5
[P2] evento interno    C = 6

=== Verifica proprieta' C(send) < C(recv) ===
  send(P0->P1) = 2,  recv(P1) = 3  ->  PASS
  send(P1->P2) = 4,  recv(P2) = 5  ->  PASS
```

▷ Suggerimento

La regola R3 applica **prima** il massimo, **poi** il `+1`. L'ordine è fondamentale: senza `+1` non sarebbe garantito $C(\text{send}) < C(\text{recv})$.

[!] Attenzione

L'ordine delle stampe può variare da un'esecuzione all'altra perché i thread girano in parallelo. Ciò che deve essere *sempre* uguale sono i valori numerici nelle ultime due righe del verify.

Esercizio 2: Vector Clock

Obiettivo

Estendere il clock da un intero a un vettore di N interi e verificare che il vector clock *caratterizzi completamente* la relazione *happened-before*: individua sia le coppie causali sia le coppie concorrenti.

Scenario

Identico all'esercizio 1. Il file `es2_vector.c` contiene già `node0`, `node1`, `node2` e salva automaticamente i timestamp degli eventi rilevanti per la verifica.

Cosa implementare

Compito — `es2_vector.c`

1. `vc_event(id)` — R1: incrementa solo la componente propria `vc[id][id]`.
2. `vc_send(id, dst, payload)` — R2: incrementa `vc[id][id]`, allega l'intero vettore `vc[id]` al messaggio, lo deposita con `send_msg`.
3. `vc_recv(id)` — R3: per ogni `k`: `vc[id][k] = max(vc[id][k], msg.vec[k])`; poi `vc[id][id]++`.
4. `vc_cmp(a, b)` — confronto tra due vettori: restituisce `-1` se $a < b$, `1` se $a > b$, `0`

se $a = b$, 2 se $a \parallel b$.

5. `verify()` — chiama `vc_cmp` sulle tre coppie seguenti e stampa `PASS`/`FAIL`:
   - `ts_a` vs `ts_c` → atteso: **CONCORRENTI**
   - `ts_send_p0` vs `ts_recv_p1` → atteso: **CAUSALE**
   - `ts_send_p1` vs `ts_recv_p2` → atteso: **CAUSALE**

Output atteso

```
[P0] evento interno   V = [1,0,0]
[P1] evento interno   V = [0,1,0]
[P0] send -> P1       V = [2,0,0]
[P0] evento interno   V = [3,0,0]
[P1] recv <- P0       V = [2,2,0]
[P1] send -> P2       V = [2,3,0]
[P2] recv <- P1       V = [2,3,1]
[P2] evento interno   V = [2,3,2]

=== Confronti vector clock ===
  ts_a [1,0,0]  vs  ts_c [0,1,0]       -> CONCORRENTI  PASS
  send(P0->P1) [2,0,0] vs recv(P1) [2,2,0] -> CAUSALE  PASS
  send(P1->P2) [2,3,0] vs recv(P2) [2,3,1] -> CAUSALE  PASS
```

▷ **Implementazione di `vc_cmp`**

Calcola separatamente se $a \leq b$ e se $b \leq a$ (un loop ciascuno). I quattro casi sono: $a \leq b$ e $b \leq a \Rightarrow$ uguali; $a \leq b$ e $b \not\leq a \Rightarrow a < b$; $b \leq a$ e $a \not\leq b \Rightarrow a > b$; nessuno dei due $\Rightarrow$ concorrenti.

Esercizio 3: Mutex Distribuito di Lamport

Obiettivo

Implementare il protocollo di Lamport (1978) per la mutua esclusione distribuita senza coordinatore centrale. Verificare *safety* (un solo nodo per volta in SC) e *fairness* (l'ordine di ingresso rispecchia la coppia $\langle ts, id \rangle$).

Strutture dati fornite

Il file `es3_mutex.c` fornisce già:

- `PQueue` — coda di priorità delle richieste pendenti, ordinata per $\langle ts, from \rangle$;
- `pq_insert`, `pq_remove`, `pq_is_top`, `pq_print` — operazioni sulla coda;
- `lc_tick(id)` — R1/R2: incrementa `clk[id]`, restituisce il nuovo valore;
- `lc_update(id, ts)` — R3: aggiorna il clock alla ricezione.

Cosa implementare

Compito — `es3_mutex.c`

1. `process_message(id, m)` — gestisce un messaggio in arrivo per il nodo `id`:
   - **REQUEST**: aggiorna il clock, inserisce la richiesta in `pq[id]`, invia ACK al mittente.
   - **ACK**: aggiorna il clock, incrementa `ack_cnt[id]`.
   - **RELEASE**: aggiorna il clock, rimuove la richiesta del mittente da `pq[id]`.

2. `request_sc(id)`:
   a. `my_ts[id] = lc_tick(id)`
   b. Invia `REQUEST` a tutti i nodi $j \neq i$.
   c. Inserisce la propria richiesta in `pq[id]`.
   d. Azzera `ack_cnt[id] = 0`.
   e. Cicla su `recv_msg` fino a che `pq_is_top` $== 1$ *e* `ack_cnt[id] == N-1`; per ogni messaggio ricevuto chiama `process_message`.
3. `release_sc(id)`: rimuove la propria richiesta da `pq[id]`, incrementa il clock, invia `RELEASE` a tutti i nodi $j \neq i$.

Output atteso

Con tre thread che partono quasi simultaneamente (timestamp=1 per tutti), l'ordine deterministico è P0, P1, P2 grazie allo spareggio sull'ID.

```
[P0] >>> ENTRO in sezione critica  (ts richiesta = 1)
[P0] <<< ESCO  da sezione critica
[P1] >>> ENTRO in sezione critica  (ts richiesta = 1)
[P1] <<< ESCO  da sezione critica
[P2] >>> ENTRO in sezione critica  (ts richiesta = 1)
[P2] <<< ESCO  da sezione critica

Terminazione pulita     safety verificata se ENTRO appare
una sola volta per volta, fairness se l'ordine rispetta <ts,id>.
```

▷ Il loop di attesa in `request_sc`

Il loop *non* deve usare *busy waiting*: chiama semplicemente `recv_msg(id)` che blocca finché arriva un messaggio. Ogni messaggio ricevuto aggiorna la coda o gli ACK, poi si ricontrolla la condizione. La CPU non viene sprecata.

▷ Perché `ack_cnt` si azzera in `request_sc`?

Ogni nuova richiesta è indipendente: gli ACK ricevuti per una richiesta precedente non valgono per quella corrente. In questo esercizio ogni nodo fa una sola richiesta, ma l'azzera garantisce correttezza anche nel caso generale.

[!] Ordine delle operazioni in `release_sc`

Rimuovi prima la tua richiesta da `pq[id]`, poi invia RELEASE. Se lo fai al contrario, un altro nodo potrebbe ricevere il RELEASE, rimuovere la tua entry, e trovare una coda in stato incoerente.

Compilazione ed esecuzione

```
# Esercizio 1
gcc -Wall -Wextra -o es1 es1_lamport.c -lpthread
./es1

# Esercizio 2
gcc -Wall -Wextra -o es2 es2_vector.c -lpthread
./es2

# Esercizio 3
gcc -Wall -Wextra -o es3 es3_mutex.c -lpthread
```

```
./es3
```

I tre esercizi sono file **indipendenti**: ciascuno include `infra.h` direttamente e si compila da solo. Non è necessario un `Makefile` separato.

[!] Sanitizer consigliato per l'esercizio 3

Per rilevare data race aggiungi `-fsanitize=thread` alla compilazione di `es3`. Il thread sanitizer segnala qualsiasi accesso concorrente non protetto a variabili condivise.