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

Hands-On #18

Algoritmi distribuiti in C: shortest-path-tree e broadcast

Last Update: May 17, 2026

Scenario

In questo HO implementeremo in C i due algoritmi studiati a lezione — **Flooding** e **Bellman-Ford distribuito** — simulando fedelmente il modello di rete sincrona visto in aula.

La traduzione dal modello teorico al codice segue tre scelte di design precise:

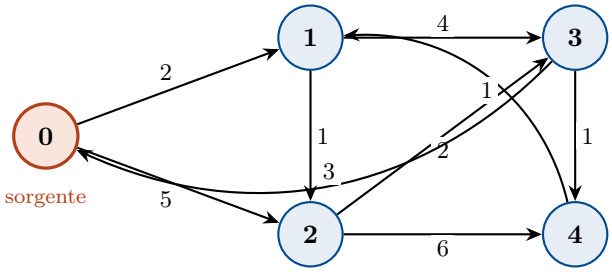
- **Un processo per ogni processore.** Ogni nodo della rete è un processo Unix creato con `fork()`. I processi non condividono memoria, esattamente come nel modello distribuito.
- **Una pipe per ogni arco.** Per ogni arco $(i \to j)$ del grafo esiste una pipe anonima `msg_pipe[i][j]` su cui il processo i scrive i messaggi e il processo j li legge.
- **Sincronizzazione tramite pipe di ack.** La semantica sincrona — tutti i processori completano il round ℓ prima di iniziare il round $\ell+1$ — emerge dalle `read()` bloccanti stesse, senza semafori né coordinatori esterni. Il meccanismo è spiegato nella Sezione 4.

Il programma esegue i due algoritmi in sequenza sullo stesso grafo e stampa il risultato di ciascuno. I file da consegnare sono **un unico sorgente C** compilabile con:

```
gcc -Wall -Wextra -o ho18 ho18.c -lm && ./ho18
```

La rete

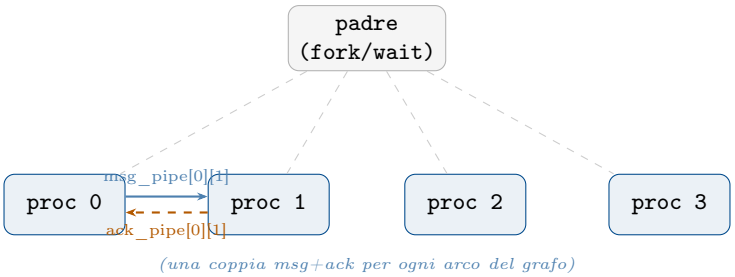
Il grafo su cui lavorerete ha **5 nodi** (numerati da 0 a 4) e **9 archi diretti** con pesi strettamente positivi. Il nodo **0** è la sorgente per entrambi gli algoritmi.



Architettura del simulatore

Processi e pipe

Il padre chiama `create_pipes()` e poi `fork()` N volte. Ogni figlio riceve l'indice `id` $\in \{0, \ldots, N-1\}$, chiude i file descriptor che non usa, ed esegue `run_flooding(id)` o `run_bf(id)`.



Per ogni arco $(i \to j)$ del grafo esistono **due pipe**:

Pipe	Chi scrive	Chi legge
`msg_pipe[i][j]`	processo i (lato `[1]`)	processo j (lato `[0]`)
`ack_pipe[i][j]`	processo j (lato `[1]`)	processo i (lato `[0]`)

`ack_pipe[i][j]` scorre in direzione opposta a `msg_pipe[i][j]`: dopo aver letto il messaggio di i, il processo j invia un ack a i per segnalare che la pipe è pronta per il round successivo.

Il meccanismo di sincronizzazione

Ogni round è diviso in **quattro fasi** eseguite da tutti i processi nello stesso ordine:

```
Fase 1  MSG   ogni processo scrive il proprio messaggio su msg_pipe[id][j]
              per ogni out-neighbor j
Fase 2  STF   ogni processo legge da msg_pipe[i][id]
              per ogni in-neighbor i, poi chiama stf()
Fase 3  ACK   ogni processo scrive un ack su ack_pipe[i][id]
              per ogni in-neighbor i ("ho letto il tuo messaggio")
Fase 4  WAIT  ogni processo legge un ack da ack_pipe[id][j]
              per ogni out-neighbor j (attende che tutti abbiano letto)
```

La sincronizzazione emerge automaticamente dalle `read()` bloccanti: la Fase 2 del processo j non può avanzare finché il processo i non ha completato la Fase 1 (la `read()` su `msg_pipe[i][j]` blocca j fino a quando i non scrive). La Fase 4 impedisce che i sovrascriva la pipe con i messaggi del round $\ell{+}1$ prima che j abbia consumato quelli del round ℓ.

▷ Perché servono entrambe le pipe?

Senza la pipe di ack, il processo i potrebbe scrivere il messaggio del round $\ell{+}1$ su `msg_pipe[i][j]` prima che j abbia letto quello del round ℓ, corrompendo la comunicazione. La `read()` bloccante in Fase 4 garantisce che i parta solo quando tutti i suoi out-neighbor hanno effettivamente consumato il messaggio corrente.

Il codice fornito

Trovate nel file `ho18_skeleton.c` le seguenti funzioni già implementate. Leggetele con attenzione prima di iniziare i task.

`create_pipes()` — crea tutte le pipe

Listing 1: `create_pipes()`: alloca NxN coppie di pipe

```
1 void create_pipes(void) {
2     for (int i = 0; i < N; i++)
```

```
        for (int j = 0; j < N; j++) {
            if (i == j) continue;           /* niente self-loop */
            if (pipe(msg_pipe[i][j]) < 0) { perror("pipe msg"); exit(1); }
            if (pipe(ack_pipe[i][j]) < 0) { perror("pipe ack"); exit(1); }
        }
}
```

Crea $N \times (N-1)$ coppie di pipe, una per ogni possibile arco diretto. Le pipe vengono create per tutti i potenziali archi, non solo per quelli presenti nel grafo: le pipe inutilizzate vengono semplicemente chiuse da `close_unused_fds()`.

`close_unused_fds()` — ogni figlio chiude i capi che non usa

Listing 2: `close_unused_fds(id)`: gestione dei descrittori dopo `fork()`

```
void close_unused_fds(int id) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (i == j) continue;
            if (i == id) {
                /* id e' il mittente su msg_pipe[id][j]:
                 * tiene il lato scrittura [1], chiude il lato lettura [0].
                 * id e' il destinatario di ack_pipe[id][j]:
                 * tiene il lato lettura [0], chiude il lato scrittura [1]. */
                close(msg_pipe[id][j][0]);
                close(ack_pipe[id][j][1]);
            } else if (j == id) {
                /* id e' il destinatario su msg_pipe[i][id]:
                 * tiene il lato lettura [0], chiude il lato scrittura [1].
                 * id e' il mittente di ack su ack_pipe[i][id]:
                 * tiene il lato scrittura [1], chiude il lato lettura [0]. */
                close(msg_pipe[i][id][1]);
                close(ack_pipe[i][id][0]);
            } else {
                /* pipe tra altri due processi: id chiude entrambi i lati */
                close(msg_pipe[i][j][0]); close(msg_pipe[i][j][1]);
                close(ack_pipe[i][j][0]); close(ack_pipe[i][j][1]);
            }
        }
    }
}
```

[!] Dimenticare di chiudere i capi inutilizzati

Se un processo tiene aperto il lato di *scrittura* di una pipe da cui non scriverà mai, la `read()` dell'altro processo non restituirà mai EOF e si bloccherà per sempre. Questo è il bug più comune con le pipe in C. `close_unused_fds()` lo previene chiudendo sistematicamente ogni file descriptor non necessario.

`send_double()`, `recv_double()`, `send_ack()`, `recv_ack()`

Listing 3: Primitive di I/O sulle pipe

```
void send_double(int fd, double val) {
    /* Scrive sizeof(double) = 8 byte atomicamente sulla pipe.
```

```
     * Le pipe Linux garantiscono atomicita' per scritture <= PIPE_BUF
       (4096 B). */
    if (write(fd, &val, sizeof(double)) != sizeof(double))
        { perror("write"); exit(1); }
}

double recv_double(int fd) {
    double val;
    /* Blocca finche' il mittente non ha scritto 8 byte. */
    if (read(fd, &val, sizeof(double)) != sizeof(double))
        { perror("read"); exit(1); }
    return val;
}

void send_ack(int fd) {
    char c = 1;
    write(fd, &c, 1);  /* 1 byte e' sufficiente come segnale di
        sincronizzazione */
}

void recv_ack(int fd) {
    char c;
    read(fd, &c, 1);   /* blocca finche' l'ack non arriva */
}
```

I messaggi sono trasmessi come valori `double` a 8 byte in formato binario nativo, senza serializzazione. Il valore sentinella `NULL_MSG = -1.0` segnala l'assenza di messaggio (tutti i pesi sono positivi, quindi -1 è inequivocabilmente una sentinella).

`run_flooding()` e `run_bf()` — il loop dei round

Queste funzioni implementano il loop dei round per i due algoritmi. La struttura è identica: le quattro fasi si ripetono per `ROUNDS` round. I dettagli dell'algoritmo sono delegati alle funzioni `msg` e `stf` che voi implementerete.

Listing 4: `run_flooding()`: loop dei round con le quattro fasi

```
void run_flooding(int id) {
    /* Stato iniziale secondo il modello formale */
    FloodState s;
    s.parent   = (id == SOURCE) ? SOURCE : NO_PAR;
    s.data     = (id == SOURCE) ? 1 : 0;      /* 1 = ha il token */
    s.snd_flag = (id == SOURCE) ? 1 : 0;

    for (int r = 0; r < ROUNDS; r++) {
        /* Fase 1: ogni out-neighbor riceve il risultato di flood_msg() */
        for (int j = 0; j < N; j++) {
            if (j == id || adj[id][j] == 0.0) continue;
            send_double(msg_pipe[id][j][1], flood_msg(id, &s, j));
        }
        /* Fase 2: raccogli messaggi dagli in-neighbor, poi aggiorna stato
           */
        double in_msgs[N];
        for (int i = 0; i < N; i++) in_msgs[i] = NULL_MSG;
        for (int i = 0; i < N; i++) {
            if (i == id || adj[i][id] == 0.0) continue;
            in_msgs[i] = recv_double(msg_pipe[i][id][0]);  /* bloccante */
        }
        flood_stf(id, &s, in_msgs);
        /* Fase 3: ack verso ogni in-neighbor ("ho letto il tuo messaggio
            ") */
        for (int i = 0; i < N; i++) {
            if (i == id || adj[i][id] == 0.0) continue;
```

```
            send_ack(ack_pipe[i][id][1]);
        }
        /* Fase 4: attendi ack da ogni out-neighbor prima del round
           successivo */
        for (int j = 0; j < N; j++) {
            if (j == id || adj[id][j] == 0.0) continue;
            recv_ack(ack_pipe[id][j][0]);
        }
    }
    printf("[FLOOD] nodo %d: parent=%2d  token=%s\n",
           id, s.parent, s.data ? "SI" : "NO");
    fflush(stdout);
}
```

`run_bf()` ha la stessa struttura: state iniziale diverso (`dist = 0` per la sorgente, `INF` per gli altri) e chiama `bf_msg()` e `bf_stf()` al posto delle controparti per il flooding.

`main()` — fork, wait e sequenziamento

Listing 5: `main()`: esegue flooding poi Bellman-Ford

```
int main(void) {
    /* ?? Parte 1: Flooding ?????????????????????????????????????????? */
    printf("=== Flooding Algorithm (sorgente: nodo %d) ===\n", SOURCE);
    create_pipes();
    for (int id = 0; id < N; id++) {
        pid_t pid = fork();
        if (pid < 0)  { perror("fork"); exit(1); }
        if (pid == 0) {                     /* figlio */
            close_unused_fds(id);
            run_flooding(id);
            exit(0);
        }
    }
    /* Il padre chiude tutte le sue copie delle pipe e aspetta i figli */
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            if (i == j) continue;
            close(msg_pipe[i][j][0]); close(msg_pipe[i][j][1]);
            close(ack_pipe[i][j][0]); close(ack_pipe[i][j][1]);
        }
    for (int i = 0; i < N; i++) wait(NULL);

    /* ?? Parte 2: Bellman-Ford ????????????????????????????????????? */
    printf("\n=== Distributed Bellman-Ford (sorgente: nodo %d) ===\n",
        SOURCE);
    create_pipes();
    for (int id = 0; id < N; id++) {
        pid_t pid = fork();
        if (pid < 0)  { perror("fork"); exit(1); }
        if (pid == 0) {
            close_unused_fds(id);
            run_bf(id);
            exit(0);
        }
    }
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            if (i == j) continue;
            close(msg_pipe[i][j][0]); close(msg_pipe[i][j][1]);
            close(ack_pipe[i][j][0]); close(ack_pipe[i][j][1]);
        }
    for (int i = 0; i < N; i++) wait(NULL);
```

```
    return 0;
}
```

▷ Perché il padre chiude tutte le pipe?

Dopo il `fork()`, il padre ha ereditato tutte le copie dei file descriptor. Se non le chiudesse, i lati di scrittura resterebbero aperti anche dopo che tutti i figli hanno terminato: le `read()` dei figli non riceverebbero mai EOF e il programma si bloccherebbe.

Task 1 — Flooding Algorithm

1.1 — Topologia: la matrice di adiacenza

Task 1.1 — Definire la matrice di adiacenza

Completate la definizione di `adj[N][N]` in `ho18.c` in modo da rappresentare il grafo della Sezione 2. Il valore `adj[i][j]` deve essere il peso dell'arco $i \to j$, oppure `0.0` se l'arco non esiste. Non esistono self-loop (`adj[i][i] = 0.0`).

Elenco degli archi: $0{\to}1:2$, $0{\to}2:5$, $1{\to}2:1$, $1{\to}3:4$, $2{\to}3:2$, $2{\to}4:6$, $3{\to}4:1$, $3{\to}0:3$, $4{\to}1:1$.

1.2 — La funzione `flood_msg()`

Nel modello formale, `msg`(w, i) restituisce il messaggio da inviare al vicino i dato lo stato corrente $w = (\texttt{parent}, \texttt{data}, \texttt{snd_flag})$. Per il Flooding:

- se `snd_flag = 1` e il vicino i *non è* il `parent`, restituisce `1.0` (trasmette il token);
- altrimenti restituisce `NULL_MSG`.

Il nodo non ritrasmette verso il proprio parent per non creare cicli nell'albero BFS. La sorgente ha `parent = SOURCE = 0`, quindi trasmette a tutti i suoi out-neighbor.

Task 1.2 — Implementare `flood_msg()`

```
/* Restituisce il messaggio da inviare al vicino 'neighbor'.
 * Usate NULL_MSG (-1.0) per indicare "nessun messaggio". */
double flood_msg(int id, FloodState *s, int neighbor) {
    /* DA IMPLEMENTARE */
}
```

1.3 — La funzione `flood_stf()`

`stf`(w, y) riceve il vettore dei messaggi `in_msgs[N]` (dove `in_msgs[i] = NULL_MSG` se i non è un in-neighbor o non ha inviato nulla) e aggiorna lo stato del processore. I tre casi del modello formale:

1. `data = 0` **e tutti** `NULL_MSG`: il token non è ancora arrivato; lo stato rimane invariato.
2. `data = 0` **e almeno un msg** $\neq$ `NULL_MSG`: token ricevuto per la prima volta. Il `parent` diventa il mittente con **UID minore** tra tutti quelli che hanno inviato un messaggio non-null. Si pone `data = 1` e `snd_flag = 1`.
3. `data = 1`: token già posseduto; si pone solo `snd_flag = 0` (non ritrasmettere più).

Task 1.3 — Implementare `flood_stf()`

```c
/* Aggiorna lo stato in base ai messaggi ricevuti.
 * in_msgs[i] == NULL_MSG se i non ha inviato nulla. */
void flood_stf(int id, FloodState *s, double in_msgs[N]) {
    /* DA IMPLEMENTARE */
}
```

▷ **Come trovare il parent con UID minore**

Scorrete `in_msgs` da `i = 0` a `N-1`: il primo indice con `in_msgs[i] != NULL_MSG` che trovate ha già l'UID minore tra quelli che hanno inviato, perché state scorrendo in ordine crescente. In alternativa, inizializzate `new_parent = NO_PAR` e aggiornatelo ogni volta che trovate un mittente con UID minore di quello attuale.

Task 2 — Distributed Bellman-Ford

2.1 — La funzione `bf_msg()`

Nel Bellman-Ford distribuito, `msg`(w, i) è indipendente dal vicino i: ogni processore trasmette semplicemente la propria stima `dist` a tutti gli out-neighbor, per i primi N round. Dopo N round invia `NULL_MSG`. Poiché `run_bf()` esegue esattamente `ROUNDS = N` round, in pratica la condizione su `round` è sempre vera; va comunque codificata per fedeltà al modello formale.

Task 2.1 — Implementare `bf_msg()`

```c
/* Restituisce il messaggio da inviare al vicino 'neighbor' al round '
    round'.
 * Trasmette dist se round < N, altrimenti NULL_MSG. */
double bf_msg(int id, BFState *s, int round, int neighbor) {
    /* DA IMPLEMENTARE */
}
```

2.2 — La funzione `bf_stf()`

`stf`(w, y) implementa il rilassamento degli archi distribuito. Per ogni in-neighbor i che ha inviato un valore non-`NULL_MSG`:

$$\text{candidato} = \texttt{in_msgs[i]} + \texttt{adj[i][id]}$$

Se `candidato < s->dist`, aggiornate `dist` e `parent`. Il processo compara *tutti* gli in-neighbor nel medesimo round e tiene il candidato migliore.

Task 2.2 — Implementare `bf_stf()`

```c
/* Aggiorna dist e parent se un in-neighbor offre un percorso piu'
    corto.
 * adj[i][id] e' il peso dell'arco i -> id. */
void bf_stf(int id, BFState *s, double in_msgs[N]) {
    /* DA IMPLEMENTARE */
}
```

▷ Gestire correttamente `INF`

La sorgente inizia con `dist = 0.0`; tutti gli altri con `dist = INF = 1e18`. Nel primo round, ogni out-neighbor della sorgente riceve `0.0 + peso_arco` e aggiorna la propria stima. Verificate che `in_msgs[i] != NULL_MSG` *prima* di sommare il peso: un `NULL_MSG = -1.0` sommato a un peso darebbe un candidato negativo scorretto.

[!] Non aggiornare in-place durante la fase STF

`bf_stf()` viene chiamata *dopo* che tutti i messaggi del round corrente sono già stati ricevuti (Fase 2 è completata). Non dovete preoccuparvi di aggiornamenti in-place: ogni processore aggiorna solo il proprio stato locale `s->dist` e `s->parent`, che il processore stesso leggerà solo nel round successivo. La sincronizzazione garantisce che il round $r+1$ non inizi prima che tutti abbiano finito il round r.

Compilazione ed esecuzione

```
gcc -Wall -Wextra -o ho18 ho18.c -lm
./ho18
```

Output atteso (l'ordine delle righe può variare perché i processi sono paralleli):

```
=== Flooding Algorithm (sorgente: nodo 0) ===
[FLOOD] nodo 0: parent= 0  token=SI
[FLOOD] nodo 1: parent= 0  token=SI
[FLOOD] nodo 2: parent= 0  token=SI
[FLOOD] nodo 3: parent= 1  token=SI
[FLOOD] nodo 4: parent= 2  token=SI

=== Distributed Bellman-Ford (sorgente: nodo 0) ===
[BF] nodo 0: dist=  0  parent= -
[BF] nodo 1: dist=  2  parent= 0
[BF] nodo 2: dist=  3  parent= 1
[BF] nodo 3: dist=  5  parent= 2
[BF] nodo 4: dist=  6  parent= 3
```

▷ Ordine non deterministico dell'output

I processi figli vengono schedulati dal kernel in ordine non deterministico. L'output del nodo 3 può apparire prima di quello del nodo 1 senza che il programma sia sbagliato. La correttezza si verifica sui valori, non sull'ordine delle righe.

Debug: cosa controllare se il programma si blocca

Se `./ho18` non termina entro pochi secondi, il programma è in deadlock. Le cause più comuni:

1. **Pipe con lato scrittura ancora aperto.** Un processo ha dimenticato di chiudere un lato di scrittura che non usa: la `read()` dell'altro processo attende un EOF che non arriverà mai. Verificate che `close_unused_fds()` venga chiamata all'inizio di ogni figlio e che il padre chiuda tutte le sue copie dopo i `fork()`.
2. **Asimmetria msg/ack.** Se un processo invia un ack senza che nessuno lo attenda, la corrispondente `recv_ack()` non verrà mai eseguita e il mittente del round successivo si bloccherà in Fase 4. Verificate che le fasi 3 e 4 iterino sulle stesse condizioni della rete (`adj[i][id] > 0` e `adj[id][j] > 0` rispettivamente).

3. **NULL_MSG non gestito in `stf()`.** Se `bf_stf()` aggiorna `dist` anche quando `in_msgs[i] == NULL_MSG`, il calcolo è scorretto ma il programma non si blocca. Controllate con `-fsanitize=address` eventuali accessi a pipe non inizializzate.