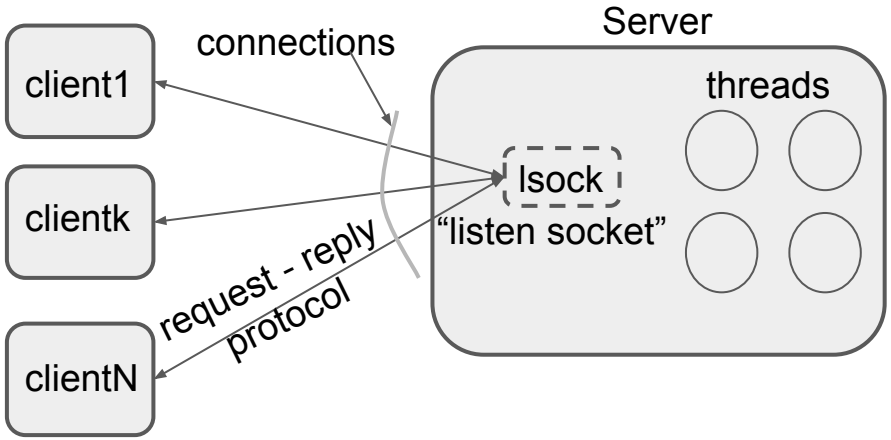
Note sull’architettura software di server multi-threaded

Server multi-threaded

- Un server multi-threaded è un processo internamente concorrente che serve richieste provenienti da client e fornisce uno o più servizi
  - Gli schemi architetturali che vedremo valgono per qualunque meccanismo IPC
  - Faremo riferimento ai socket (AF_UNIX) per convenienza
- Perchè un server internamente concorrente?
  - ✓ Per aumentare la reattività del servente
  - ✓ Diminuire i tempi di servizio delle richieste sfruttando concorrenza/parallelismo
  - ✓ Per utilizzare meglio le risorse di calcolo dei sistemi multi-cores
    - Sfruttare il parallelismo architetturale

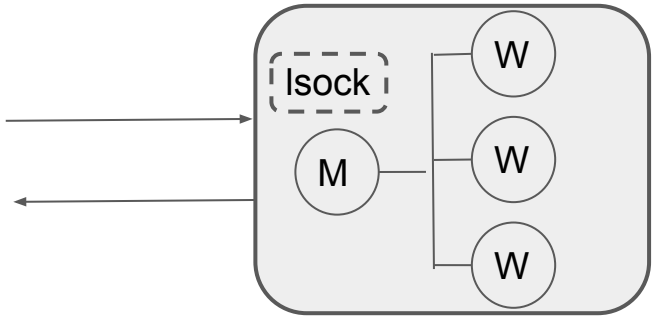
Server multi-threaded



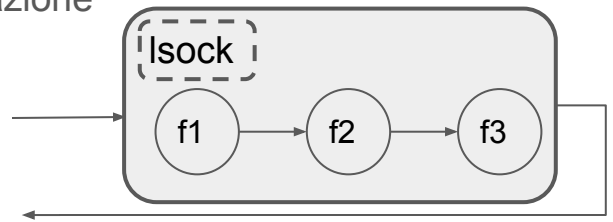
- Assumiamo che i client mandano una o più richieste sulla stessa connessione
- Assumiamo che il client prima di inviare la richiesta successiva aspetti di ricevere la risposta alla richiesta precedente (protocollo richiesta-risposta)
- Il server può avere uno stato interno (gestito opportunamente) che i client possono modificare sulla base del tipo di richieste che inviano

Design pattern tipici

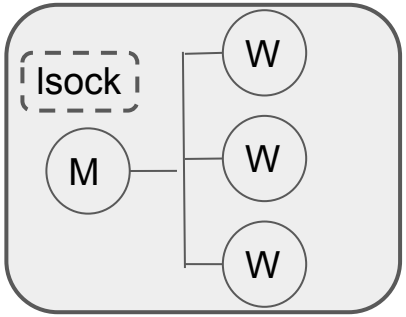
- **Master-Slave** (o Manager-Worker o Master-Worker)



- **Pipeline** (non lo vedremo in dettaglio)
  - Si applica quando la richiesta può essere soddisfatta dall'applicazione di un certo numero di funzioni distinte (F(x)=fn(fn-1(...f2(f1(x)))). Ogni fi (o un gruppo di fi) viene eseguita da un thread distinto.

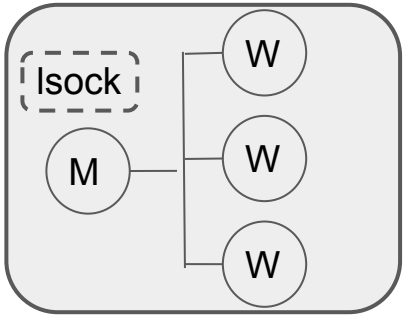


Master-Worker



- Un Manager thread (M) e “molti” Worker threads (Ws).
- M esegue il dispatching delle richieste provenienti dai clienti ai Workers
- I Ws servono le richieste. Se il client C invia la richiesta ‘x’, W esegue y=F(x) ed invia un messaggio di risposta contenente y a C
- Il messaggio da M ai Ws può avere diversa natura:
  - un’intera connessione di un client appena connesso viene fatta gestire da uno dei Ws,
  - una singola richiesta di un client già connesso viene assegnata ad uno dei Ws
- I thread Ws
  - possono essere lanciati dinamicamente per ogni nuova richiesta di connessione (“un thread per connessione”)
  - possono far parte di un pool di thread predefinito (“pool di thread” vuol dire un gruppo di thread già spawnati e pronti per servire richieste, al termine del servizio il thread ritorna)
  - mix delle due soluzioni precedenti

Master-Worker



- Il dispatching delle richieste può essere fatto attraverso una coda concorrente condivisa da tutti i Ws, oppure, alternativamente, con k-code distinte (k<=#Ws)
- I Workers possono essere eterogenei, cioè eseguire richieste di tipo diverso (F, G, H, …).
  - In questo caso, si possono usare più pool distinti di threads una per ogni funzione, oppure ogni W è in grado di eseguire tutte le richieste

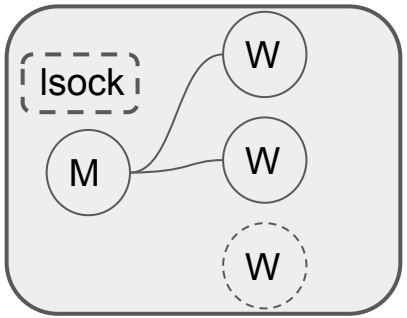
- Variante (detta anche **Peer-Threads**)
  - M è un Worker come gli altri
  - M è un thread che si comporta anche come Worker oltre a fare il dispatching

Master-Worker: soluzione base

- Un “thread per connessione”: W serve tutte le richieste di un client
- M esegue continuamente la SC **accept** sul “listen socket”
- Il descrittore ritornato dall’accept viene passato come parametro ad un W thread (tipicamente spawnato in modalità *detached*)

(pseudo-)codice del Manager

```
while(!terminate) {
  int fd = accept(lsock, NULL,NULL);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, ...);
  pthread_create(&thr, &attr, F, (void*)fd);
}
```



Master-Worker: soluzione base

PRO

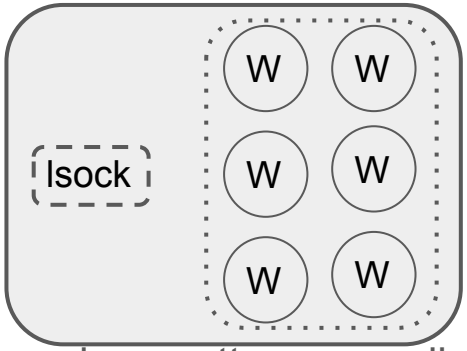
- Soluzione molto semplice da implementare
- Funziona molto bene per carichi non troppo elevati

CONTRO

- Overhead legato allo spawning dinamico dei threads
- Ci possono essere molti worker threads inattivi

Variante “Peer-Threads”: pool di soli Worker

- Non c’è un Manager thread, ma un thread pool di Ws sempre attivi
- A turno i Ws eseguono accept in *mutua esclusione* e servono la nuova connessione
  - **NOTA**: la SC accept può essere eseguita in modo concorrente da tutti i Ws, ma è più efficiente se eseguita da uno solo alla volta.
- I Ws scarichi (tranne uno, *W’*), che non stanno servendo richieste sono sospesi su una variabile di condizione. Uno di loro esegue accept all’interno della sezione critica. Quando accept si sblocca, *W’* fa una *signal* sulla variabile di condizione e serve per intero la nuova connessione.



- Si può prevedere che, quando i thread del pool che sono scarichi scendono sotto una soglia minima, ne vengono fatti partire un certo numero in modalità detached

Variante “Peer-Threads”: pool di soli Worker

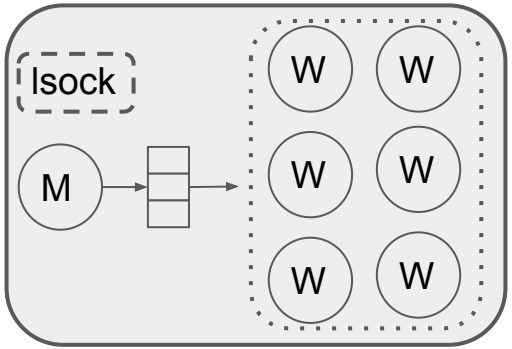
PRO

- Soluzione semplice da implementare
- Funziona molto bene per carichi non troppo elevati
- Minore overhead legato allo spawning dinamico dei threads (presente solo per alti carichi)

CONTRO

- Per alti carichi l’overhead può essere molto alto
- Ci possono essere thread facenti parte del pool che sono inattivi anche se dobbiamo spawnare nuovi thread per far fronte a nuove connessioni

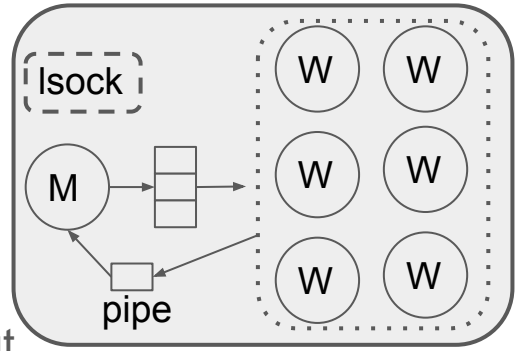
Master-Workers: soluzione con thread pool



- Pool di Worker threads sempre attivi
- W gestisce una/più richiesta/e di qualsiasi client
  - W non è associato ad una specifica connessione
- Il thread M ha il compito di
  - accettare nuove connessioni
  - controllare quali, tra i descrittori di connessioni già aperte, sono pronti in lettura
    - utilizzando ***select***, ***poll*** o ***epoll*** (man 7 epoll -- Linux-specific)
- M -> Ws comunicano attraverso una coda concorrente (single-producer multi-consumer) contenente i descrittori pronti in lettura (cioè, sui quali la read non si blocca)
  - I descrittori inviati ai thread non vengono più “ascoltati” dal manager thread
- Problema: quando la richiesta è stata servita da uno dei Worker del pool come viene detto ad M di ri-ascoltare nuovamente il descrittore?

Master-Workers: soluzione con thread pool

- Ws -> M comunicano tramite
  - una pipe senza nome il cui endpoint di lettura è registrato sulla select/poll/epoll
    - La pipe contiene descrittori che potrebbero bloccare una *read*
    - La scrittura sulla pipe è atomica, mutex non necessaria
  - oppure, attraverso variabili condivise (o coda multi-producer single-consumer) protette da mutex
    - In questo caso select/poll/epoll devono gestire un timeout



- Se i thread del pool non bastano, si lanciano fino k>1 Ws in modalità detached per far fronte al carico di richieste

Estensione con AIO: W può servire più richieste dal descrittore che ha estratto dalla coda fintantoché la *read* non si blocca (fcntl, O_NONBLOCK)

Master-Workers: soluzione con thread pool

PRO

- Funziona molto bene per carichi medio-alti
- Basso overhead di gestione
- Con alti carichi non ci sono thread inattivi

CONTRO

- Soluzione più complessa da implementare/mantenere

Asynchronous I/O (AIO)/Non-blocking I/O

- **Tecnica alternativa a quanto visto fino ad ora**. Usata per diminuire (anche eliminare) l'uso di threads rendendo tutte le operazioni di lettura e scrittura sui socket (e più in generale tutte le operazioni) **non bloccanti**
- Motivazioni: avere molti thread/processi costa sia in termini di risorse di sistema che in termini di overhead legato al context-switch in **workload I/O bound**
- Con i moderni sistemi multi-cores è (*quasi*) *sempre* preferibile un approccio multi-threaded/process piuttosto che single-thread/process ed AIO
- Architettura del server ad eventi: più complessa da implementare.

Non-blocking I/O + multi-threading

- E’ possibile **combinare le due tecniche** multi-threading/processing ed AIO
  - per ragioni di scalabilità verticale (un singolo server in grado di gestire alcune decine di migliaia di connessioni contemporanee)
- Tanti threads quanti sono i cores del sistema
- Ogni thread gestisce un set di connessioni in modalità non-blocking
- Gestione molto complessa
- Per la gestione degli eventi è bene utilizzare librerie specifiche che garantiscono la portabilità su diversi sistemi
  - libevent, libev, ...

Per approfondire:
“C10K problem”: http://www.kegel.com/c10k.html link datato ma interessante
Architettura di NGINX: https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/