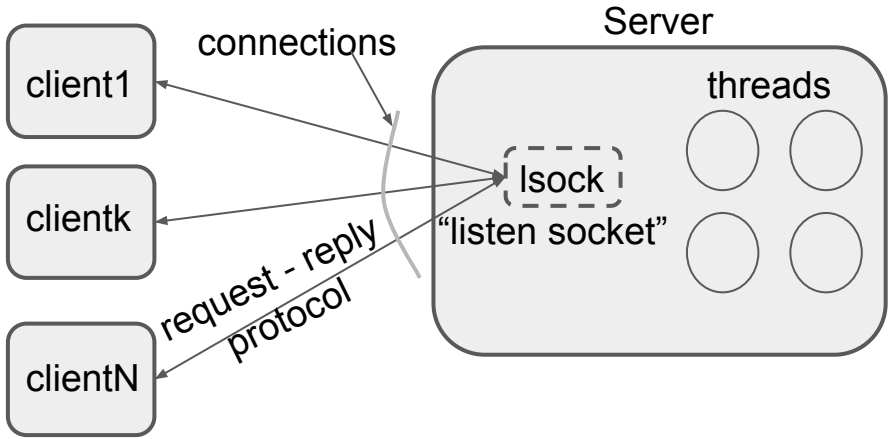
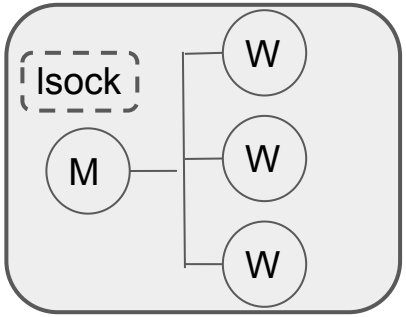
Note sull’architettura software di server multi-threaded

Server multi-threaded

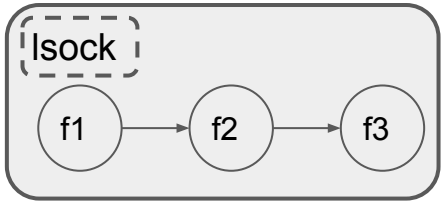
- Un server multi-threaded è un processo internamente concorrente che serve richieste provenienti da client
  - Gli schemi architetturali che vedremo valgono per qualunque meccanismo IPC
  - Faremo riferimento ai socket (AF_UNIX/AF_INET) per convenienza
- Perchè un server concorrente?
  - ✓ Per aumentare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



Design pattern tipici



- **Master-Slave** (o Manager-Workers)
  - Un Manager thread (M) e “molti” Worker threads (Ws). M esegue il dispatching delle richieste dei clienti ai Workers (‘x’ è una richiesta).
    I Ws servono le richieste ($\forall$ x: F(x)). Il messaggio da M ai Ws può avere diversa natura: 1) un’intera connessione di un client appena connesso, 2) una singola richiesta di un client già connesso. I thread Ws
    - possono essere lanciati dinamicamente per ogni nuova richiesta
    - possono far parte di un pool di thread predefinito
    - mix delle due soluzioni precedenti
  - Il dispatching delle richieste può essere fatto attraverso una coda condivisa da tutti i Ws (shared queue) oppure con k-code distinte (k<=#Ws)
  - I Workers possono essere eterogenei (cioè eseguire richieste di tipo diverso)
  - Variante (**Peer**): M è anche un Worker, cioè esegue alcune richieste oltre a fare il dispatching
- **Pipeline** (non lo descriviamo in queste note)
  - Si applica quando la richiesta può essere soddisfatta dall’uso di un certo numero di funzioni distinte (F(x)=fn(fn-1(...f2(f1(x)))).
    Ogni fi viene eseguita da un thread distinto.

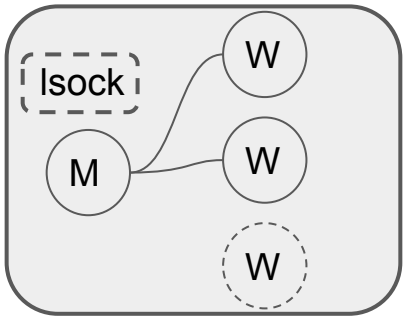


Manager-Workers: soluzione base

- Un Worker 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 lanciato in modalità *detached*

(pseudo-)codice del Manager

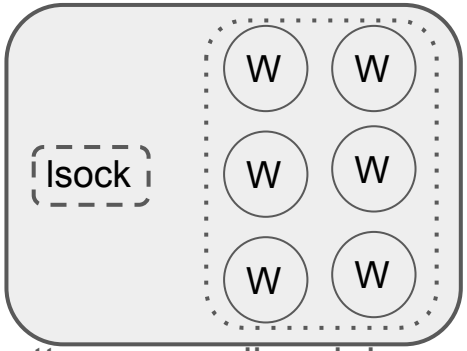
```
while(!terminate) {
  int fd = accept(lsock, NULL,NULL);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, ...);
  pthread_create(&thr, &attr, F, (void*)fd);
}
```



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

Peer: pool di soli Workers

- Non c'e' 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.

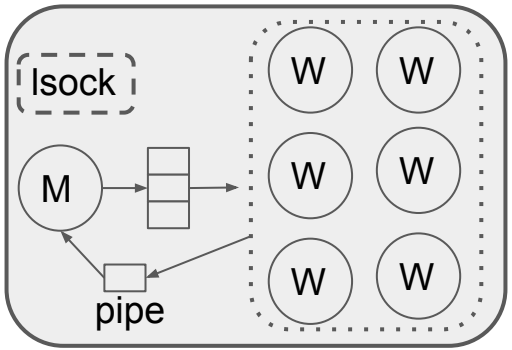


- Si può prevedere che, quando i thread del pool scarichi scendono sotto una soglia minima, ne vengono fatti partire un certo numero in modalità detached

- Meno overhead della soluzione base quando il il sistema è scarico
- Ci possono essere molti Ws inattivi

Manager-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 un client)
- M accetta nuove connessioni e controlla quali, tra i descrittori di connessioni già aperte, sono pronti in lettura (utilizzando ***select***, ***poll*** o ***epoll***)
- M -> Ws comunicano attraverso una coda concorrente (single-producer multi-consumer) contenente i descrittori pronti in lettura
- 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
  - Estensione: W può servire più richieste dal descrittore che ha estratto dalla coda fintantoché la *read* non si blocca (fcntl, O_NONBLOCK)



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

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

- Tecnica alternativa, 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**
- Razionale: avere molti thread/processi costa sia in termini di risorse di sistema che in termini di overhead legato al context-switch in sistemi I/O bound (cioè che fanno poco calcolo)
- Con i moderni sistemi multi-cores è (*quasi*) *sempre* preferibile un approccio multi-threaded/process piuttosto che single-thread/process ed AIO
- E' possibile **combinare le due tecniche** multi-threading/processing ed AIO (anche se più complicato) per ragioni di scalabilità verticale (un singolo server in grado di gestire diverse decine di migliaia di connessioni):
  - tanti threads/processi quanti sono i cores (logici o fisici)
  - ogni threads/processo gestisce le connessioni in modalità non bloccante
- 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
Architettura di NGINX:
https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/