

# Server REST Concorrente in C

## Architettura ad Alte Prestazioni con Epoll e Thread Pool

Sandi Russo, 553675

Università degli Studi di Messina  
Reti di Calcolatori e Sistemi Distribuiti

A.A. 2024/2025

- 1 Introduzione
- 2 Definizione del Problema
- 3 Metodologia
- 4 Presentazione dei Risultati
- 5 Conclusione

## Server RESTful in C

Implementare un server HTTP capace di gestire operazioni CRUD su una risorsa `/users`

## Focus: Gestione della Concorrenza

- Gestire **migliaia di connessioni simultanee**
- Architettura **scalabile ed efficiente**
- Evitare blocchi e race condition

## Principi Chiave

## REpresentational State Transfer

### Metodi HTTP

- GET: Leggi
- POST: Crea
- DELETE: Cancella

### Caratteristiche

- Stateless
- Client-Server
- Interfaccia Uniforme

GET /users/1 → Restituisce utente con ID 1

**C (Standard C11)**

**POSIX Threads (pthread)**

**Epoll (Linux I/O Multiplexing)**

**SQLite3 (Database Embedded)**

## Architettura Iterativa

Un server iterativo accetta ed elabora una connessione alla volta mediante un ciclo `accept()`  $\rightarrow$  `process()`  $\rightarrow$  `close()`.

## Problematiche

- **Serializzazione forzata:** impossibilità di gestire richieste concorrenti
- **Blocco su I/O:** operazioni disco/rete bloccano l'intero server
- **Throughput limitato:** latenza cumulativa proporzionale al numero di client
- **Denial of Service:** un singolo client lento blocca tutti gli altri

# Strategie di Concorrenza

## 1. Modello Thread-per-Connection

- + Implementazione immediata
- - Overhead di creazione/distruzione thread
- - Non scala: C10K problem

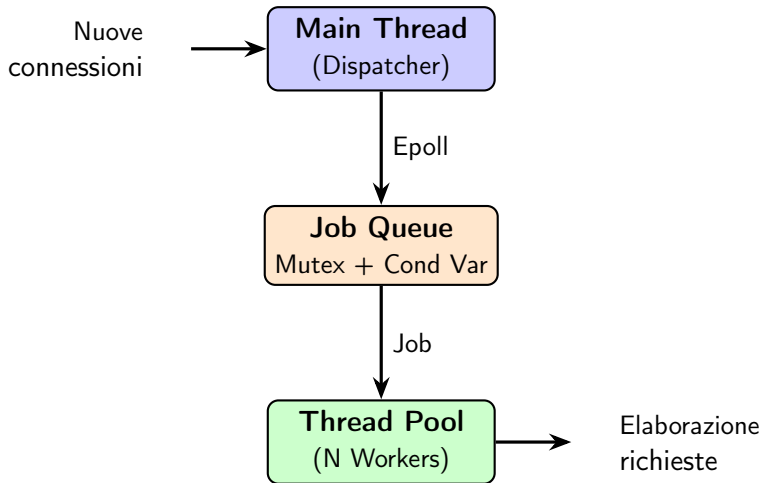
## 2. Modello Select/Poll

- + I/O multiplexing senza thread multipli
- -  $O(n)$ : scansione lineare dei descriptor
- - Limitazione nel numero di file descriptor

## 3. Modello Epoll + Thread Pool (Soluzione Adottata)

- +  $O(1)$ : efficiente indipendentemente dal numero di connessioni
- + Pool fisso: overhead controllato
- + Separazione dispatcher/worker

# Architettura: Dispatcher + Worker



## Meccanismo Epoll

API del kernel Linux per notifica asincrona di eventi su file descriptor

## Vantaggi Rispetto a Select/Poll

- **Efficienza  $O(1)$** : tempo di attesa indipendente dal numero di FD monitorati
- **Edge-triggered mode**: notifica solo su cambiamenti di stato
- **Scalabilità**: supporto nativo per decine di migliaia di connessioni
- **Event-driven**: eliminazione del polling attivo

Un singolo thread monitora tutte le connessioni senza overhead

# Loop Principale del Dispatcher

```
1 epoll_fd = epoll_create1()
2 epoll_ctl_add(epoll_fd, server_fd, EPOLLIN)
3
4 while (true) {
5     num_events = epoll_wait(epoll_fd, events, ...)
6     for (i = 0; i < num_events; i++) {
7         if (events[i].fd == server_fd) {
8             // Nuova connessione
9             client_fd = accept(server_fd)
10            epoll_ctl_add(epoll_fd, client_fd, ...)
11        } else {
12            // Dati pronti -> crea job
13            pool_submit_job(pool, handle_request, ctx)
14        }
15    }
16 }
```

## Motivazione

La creazione/distruzione dinamica di thread introduce overhead significativo (context switch, allocazione stack)

## Caratteristiche

- Pool di N worker pre-allocati
- Stato dormiente quando idle
- Attivazione on-demand

## Benefici

- Overhead ammortizzato
- Utilizzo CPU ottimizzato
- Controllo risorse

# Logica Worker Thread

```
1 function worker_loop(pool) {  
2     while (true) {  
3         lock(pool.mutex)  
4         while (queue_empty && !shutdown)  
5             wait(pool.cond, pool.mutex)  
6         if (shutdown) break  
7         job = dequeue_job(pool)  
8         unlock(pool.mutex)  
9         // Esegue il lavoro FUORI dal lock  
10        job.function(job.arg)  
11    }  
12 }
```

# Sincronizzazione: Pattern Produttore-Consumatore

## Problema

Accesso concorrente alla coda condivisa tra dispatcher (produttore) e worker (consumatori)

## Soluzione: Mutex + Condition Variable

`pthread_mutex_t` Garantisce mutua esclusione nell'accesso alla coda (enqueue/dequeue atomici)

`pthread_cond_t` Segnalazione efficiente: worker si bloccano su `wait()` e vengono risvegliati dal dispatcher con `signal()`

**Elimina sia race condition che busy-waiting**

## Caratteristiche SQLite

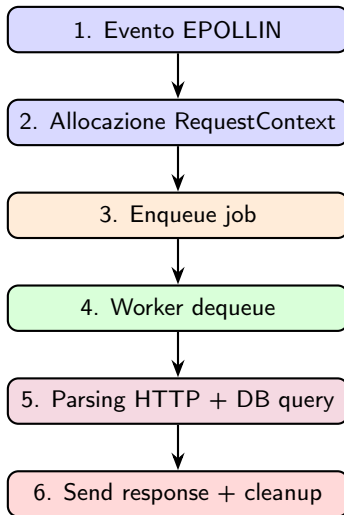
- **Embedded:** libreria C linkata direttamente, non server separato
- **Zero configurazione:** database self-contained in singolo file
- **Transazionale:** garanzie ACID complete
- **Thread-safe:** modalità serialized/multi-thread

## Design Modulare

Isolamento completo della logica SQL in `db_handler.c`

`db_get_all_users()`, `db_create_user()`, `db_delete_user()`

# Flusso di Elaborazione Richiesta



# Test Funzionali con curl

## GET /users

```
$ curl http://localhost:8080/users
```

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
[{"id":1,"name":"Mario"}, {"id":2,"name":"Luigi"}]
```

## POST /users

```
$ curl -X POST -d "name=Peach" http://localhost:8080/users
```

```
HTTP/1.1 201 Created
```

```
{"status":"created","id":3}
```

## Scenario: 50 richieste simultanee

```
#!/bin/bash
for i in {1..50}; do
    curl http://localhost:8080/users &
done
wait
```

## Risultati

- ✓ Tutte le 50 richieste completate con successo
- ✓ Worker diversi processano in parallelo
- ✓ Server rimane reattivo

# Vantaggi dell'Architettura

## Performance

- Basso consumo CPU
- Overhead minimo
- Scalabilità  $O(1)$

## Robustezza

- Thread-safe
- Modulare
- Gestione risorse

**Architettura production-ready per server ad alte prestazioni**

## Limitazioni Implementative

- **Sicurezza:** Comunicazione in chiaro (HTTP)
- **Autenticazione:** Assenza di meccanismi di controllo accessi
- **Parsing HTTP:** Gestione limitata ai casi d'uso principali
- **Bounded queue:** Coda illimitata espone a rischio OOM

## Roadmap Sviluppi Futuri

- Integrazione **TLS/SSL** (OpenSSL) per HTTPS
- Sistema di autenticazione basato su **JWT**
- Parser HTTP robusto (`http-parser`, `picohttpparser`)
- Integrazione **Redis** per caching distribuito
- Implementazione **bounded queue** con backpressure

- **Programmazione di Rete:** Socket API, TCP/IP
- **Programmazione di Sistema:** Epoll, I/O non bloccante
- **Concorrenza:** Pthreads, Mutex, Condition Variables
- **Pattern:** Thread Pool, Produttore-Consumatore
- **Protocolli:** HTTP/1.1, REST
- **Database:** SQLite, API C

**Grazie per l'attenzione!**