# Sistemi Operativi e Reti di Calcolatori (SOReCa)

## Corso di Laurea in *Ingegneria Informatica e Automatica (BIAR)*

## Terzo Anno | Primo Semestre

## A.A. 2024/2025

# Esercitazione [08] Server multi-process/thread

Riccardo Lazzeretti lazzeretti@diag.uniroma1.it
Paolo Ottolino paolo.ottolino@uniroma1.it
Edoardo Liberati e.liberati@diag.uniroma1.it

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

# Sommario

- Soluzioni precedente esercitazione
- C10K Problem
- Multi-Process:
  - Parallelismo dei Server (`fork()`)
  - Esercizio EchoServer
- Multi-Thread
  - Parallelismo dei Server (`pthread()`)
  - Esercizio EchoServer

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

# C10K Problem: 10.000 client

Dan Kegel, 1999
Solutions:

- Thread-based → Pooling
- Event-Driven → SW Architecture

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

# Obiettivi

1. Il problema C10K: origine, impatti ed evoluzione delle soluzioni.

2. Parallelismo (Pooling)
   - Muti-Process → Lab08, es.1
   - Multi-Thread → Lab08, es.2

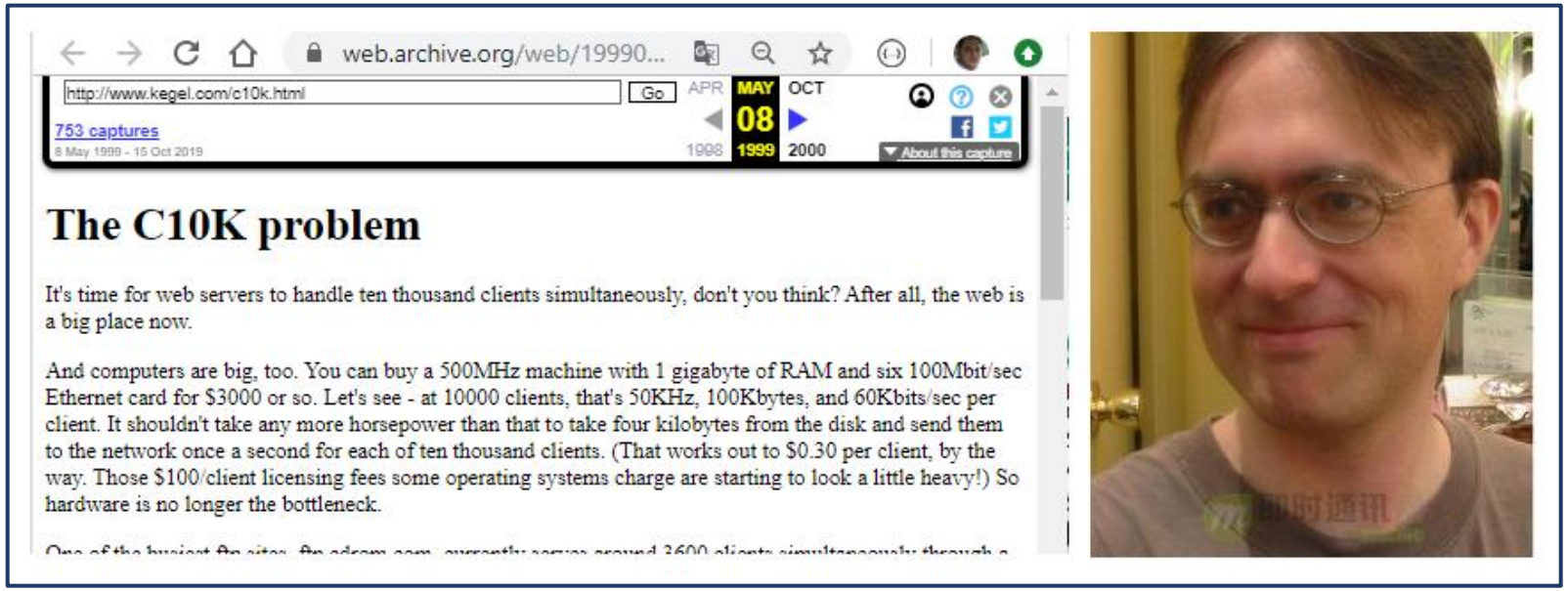
3. Altre soluzioni
   - Su base Evento (unparallel Thread)
   - I/O Multiplexing
   - Ottimizzazione dell'HW (Host e Network)

# C10K Problem

## Dan Kegel (1999)

Il problema **C10K**: **gestione simultanea di diecimila connessioni**. Cioè, il problema di ottimizzazione dei socket di rete per gestire un **gran numero** di client **contemporaneamente**.

Il termine *C10k* è stato coniato nel 1999 dall'ingegnere informatico Dan Kegel.




molte connessioni simultanee != molte richieste al secondo
pianificazione efficiente delle connessioni != throughput elevato


Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

# C10K Solutions

## Solve Scalability Bottleneck

Dall'emergere del problema c10k, sono state proposte e implementate varie soluzioni per gestire un gran numero di connessioni simultanee in modo più efficiente. Queste strategie includono:

1. **Pool-based**: questa tecnica prevede la creazione di un **pool** di processi e thread worker **all'avvio**, che può essere riutilizzato per gestire più connessioni, riducendo così il sovraccarico associato alla creazione e alla distruzione dei thread.

2. **Event-driven** programming: comporta l'organizzazione del server in modo che **reagisca agli eventi** (come l'arrivo di una **nuova connessione** o la **ricezione** di dati) anziché mantenere un thread per ogni connessione. Il server essenzialmente "aspetta" gli eventi e reagisce quando si verificano.

3. **I/O Multiplexing**: il multiplexing comporta la gestione di più input e output, utilizzando chiamate di sistema come `select()`, `poll()` ed `epoll()` che referenziano **multipli descrittori di file**.

4. **Ottimizzazioni SOReCa** (sistema operativo e rete): anche l'**hardware** di rete e i sistemi operativi sono stati migliorati per **supportare** meglio un gran numero di **connessioni simultanee**. Questi miglioramenti includono implementazioni TCP/IP migliorate, progressi nelle schede di interfaccia di rete e algoritmi di pianificazione migliori nel kernel del sistema operativo.

5. **Caching**: implementare strategie di caching **locale** per ridurre il carico del server servendo **contenuti statici** dalla cache. Esempi includono caching di oggetti, caching di pagine e caching di query di database.

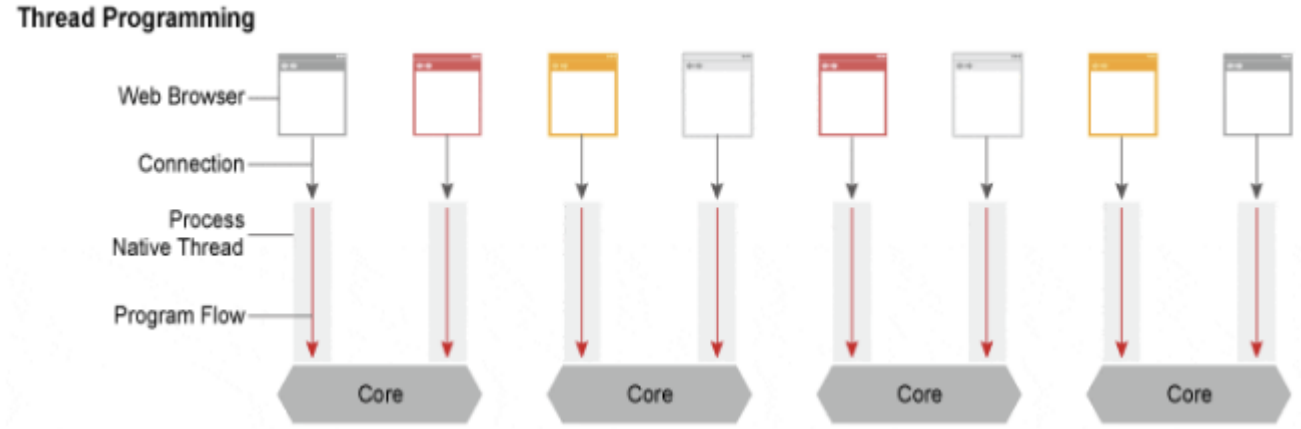
6. **Content Delivery Network** (CDN): utilizza una CDN per servire **contenuti statici** più vicini alla posizione geografica dell'utente, riducendo il carico sul server.

# C10K Solution – HTTP server

## Apache 2.x (2002) – Multi Processing Module

Lanciato per la prima volta nel 1995 (da una costola del progetto in stallo NCSA HTTPd), Apache dominò rapidamente il mercato e divenne il server web più diffuso al mondo.

Dal 2002 utilizza il **Multi Processing Module** (MPM) per implementare un server ibrido multi-processo/multi-thread. (ad esempio - **Prefork** e **Worker** MPM).




Le architetture server basate su thread, generalmente, associano ogni connessione in arrivo a un thread separato

Dal 2012 (Apache 2.4.x), MPM annovera anche la modalità Event-driven.


Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

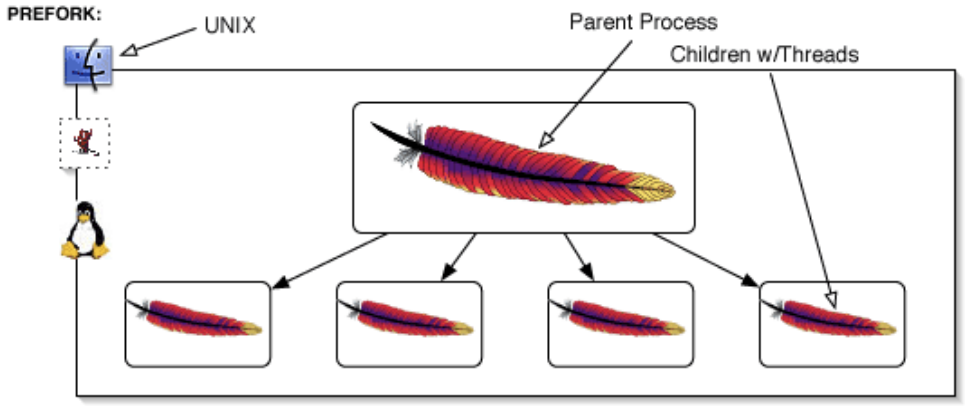
Sapienza Università di Roma

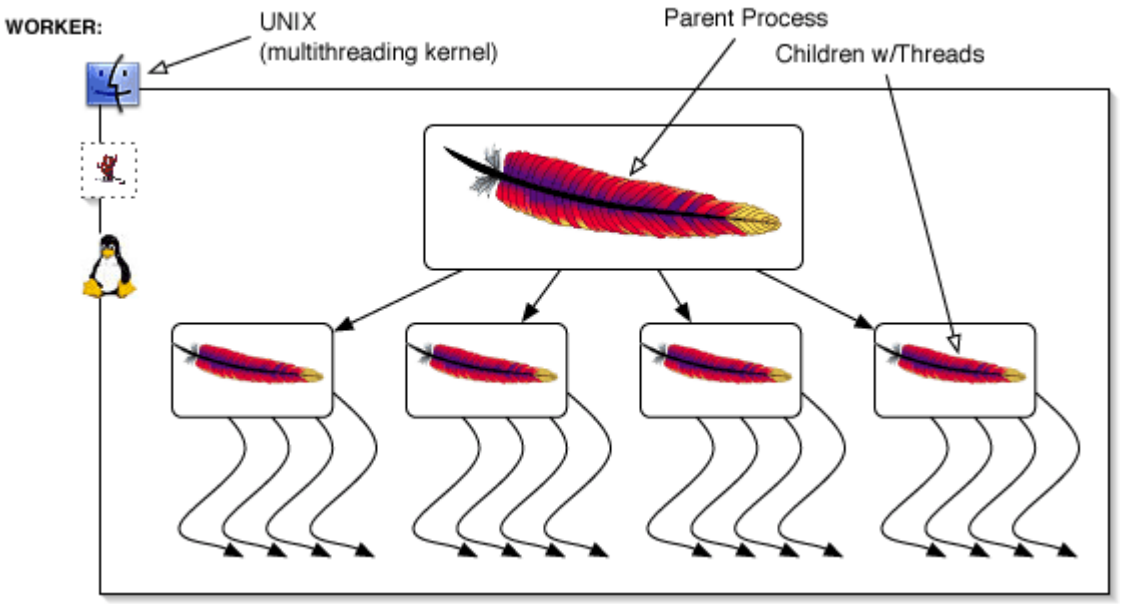
# C10K Solution – HTTP server

## Apache 2.x (2002) – Thread-based

**Prefork MPM**: server web **non-threaded**, **pre-forking**. Ciò significa che ogni processo figlio di Apache contiene un singolo thread e gestisce una richiesta alla volta. Per questo motivo, consuma più risorse degli MPM threaded: Worker ed Event.

- Prepara i thread figlio prima del processamento
- Un processo è connesso solo a un thread (massimo 1024)
- Non condivide la memoria quindi è stabile perché indipendente, tuttavia crea un **grande utilizzo di memoria**




**Worker MPM**: server web **multi-processo** e **multi-thread**. A differenza di Prefork, ogni processo figlio sotto Worker può avere più thread. In quanto tale, Worker può gestire più richieste con meno risorse rispetto a Prefork. Worker è generalmente consigliato per server ad alto traffico che eseguono versioni di Apache precedenti alla 2.4.

- Un processo è connesso a **più thread** per gestire la richiesta.
- Poiché il processo condivide la memoria → **basso utilizzo di memoria** quindi è preferibile a Prefork per una istanza web con molto traffico.


Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

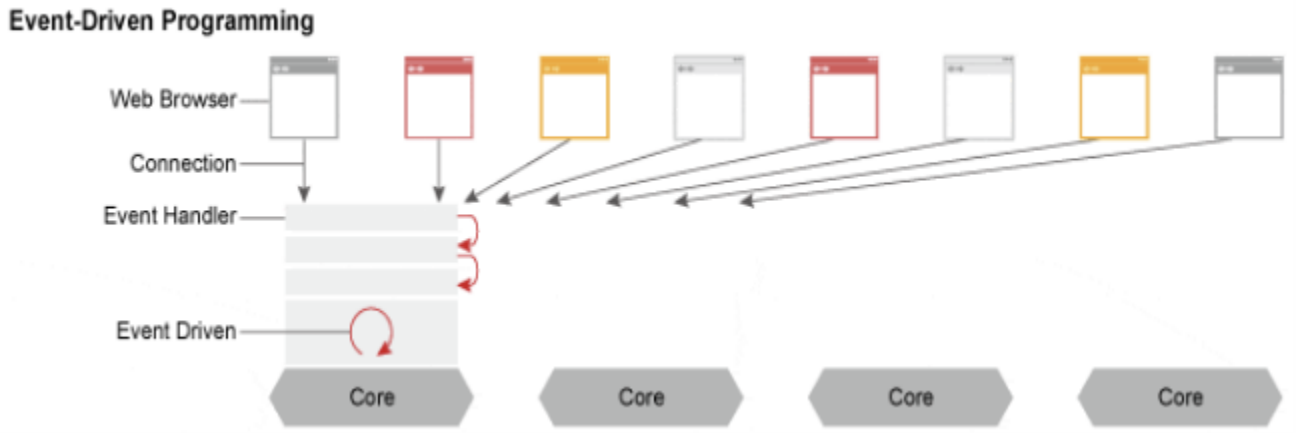
Sapienza
Università di Roma

# C10K Solutions – HTTP server

## Nginx (1999) – Event-Driven

I server web tradizionali, come Apache, hanno avuto difficoltà a gestire una concorrenza così elevata a causa del loro modello thread-per-connessione, che consumava enormi quantità di memoria e risorse della CPU.

NGINX è stato creato appositamente per affrontare le limitazioni di prestazioni dei server web Apache.




Le architecture Event-driven raccolgono le richieste in arrivo al sistema in una o più code di eventi centrali

L'architettura event-driven crea un solo processo fisso e controlla il flusso delle attività all'interno del processo in base agli eventi.

Poiché il programma si basa sulla gestione asincrona degli eventi, l'utilizzo di thread/CPU è basso. L'I/O viene elaborato in modo più flessibile, consentendo un cambio di contesto più fluido.

# Multi-Process PoC

Lab08, es1 EchoServer

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

# Multi-Process

## Parallelismo lato server

Nelle scorse esercitazioni abbiamo visto server «seriali»:

- Viene servita una connessione alla volta
- Connessioni che arrivano nel mentre vengono messe in coda…
- …e verranno processate sequenzialmente al termine della connessione attualmente servita
- Questo comporta dei tempi di attesa crescenti all'aumentare del numero di connessioni in coda!

➢ La soluzione consiste nel disaccoppiare l'accettazione delle connessioni dalla loro elaborazione

➢ Una volta accettata, una connessione viene elaborata in un processo o thread dedicato, così il server può subito rimettersi in attesa di altre connessioni da accettare

# Multi-Process

## Parallelismo lato server

Per ogni connessione accettata, viene lanciato un nuovo processo figlio tramite `fork()`

- Il figlio deve chiudere il descrittore della socket usata dal server per accettare le connessioni
- Analogamente, il padre deve chiudere il descrittore della socket relativa alla connessione appena accettata
- Una volta completata l'elaborazione della connessione, il processo figlio esce

Elevato overhead legato alla creazione di nuovo processo per ogni connessione

Complessa gestione di eventuali strutture dati condivise (tramite file, pipe, memoria condivisa oppure anche socket)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

# Multi-Process

## Parallelismo lato server

```
while (1) {
      int client = accept(server, ……);
      <gestione errori>

      pid_t pid = fork();
      if (pid == -1) {
            <gestione errori>
      } else if (pid == 0) {
            close(server);
            <elaborazione connessione client>
            _exit(0);
      } else {
            close(client);
      }
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

# Multi-Process: EchoServer

## Lab08, es1

Completare il codice dell'EchoServer in modalità multi-process

Sorgenti

- `Makefile`
- Client: `client.c`
- Server: `server.c`
- compilazione: `-DSERVER_MPROC` vs `-DSERVER_SERIAL`

Suggerimento: seguire i blocchi di commenti inseriti nel codice

Altro suggerimento:

Per monitorare a runtime il numero di istanze di processi attivi in un certo momento, lanciare da terminale il comando:

```
ps -e -O ppid | head -1; ps -e -O ppid | grep multiprocess
```

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

# Multi-Thread PoC

Lab08, es2 EchoServer «Event-driven»

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

# Multi-Thread

## Parallelismo lato server

Per ogni connessione accettata, viene lanciato un nuovo thread tramite `pthread_create()`

- Oltre ai parametri application-specific, il nuovo thread avrà bisogno del descrittore della socket relativa alla connessione appena accettata
- A differenza del server multi-process, non è necessario chiudere alcun descrittore di socket (perché?)
- Una volta completata l'elaborazione della connessione, il thread termina
- Il main thread può voler fare detach dei thread creati

Minore overhead rispetto al server multi-process (no memoria duplicata, no syscall etc.)

Gestione più semplice di eventuali strutture dati condivise

Un crash in un thread causa un crash in tutto il processo!

# Multi-Thread

## Parallelismo lato server

```
while (1) {
      int client = accept(server, ……);
      <gestione errori>

      pthread_t t;
      t_args = ……
      <includere client in t_args>
      pthread_create(&t,NULL,handler,(void*)t_args);
      <gestione errori>
      pthread_detach(t);
}
```

# Multi-Thread: EchoServer

## Lab08, es2

Completare il codice dell'EchoServer in modalità multi-thread

Sorgenti

- `Makefile`
- Client: `client.c`
- Server: `server.c`
- compilazione: `-DSERVER_MTHREAD` vs `-DSERVER_SERIAL`

Suggerimento: seguire i blocchi di commenti inseriti nel codice

Altro suggerimento:

Per monitorare a runtime il numero di istanze di processi/thread attivi in un certo momento, lanciare da terminale il comando:

```
ps -e -T | head -1 ; ps -e -T | grep multithread
```

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma