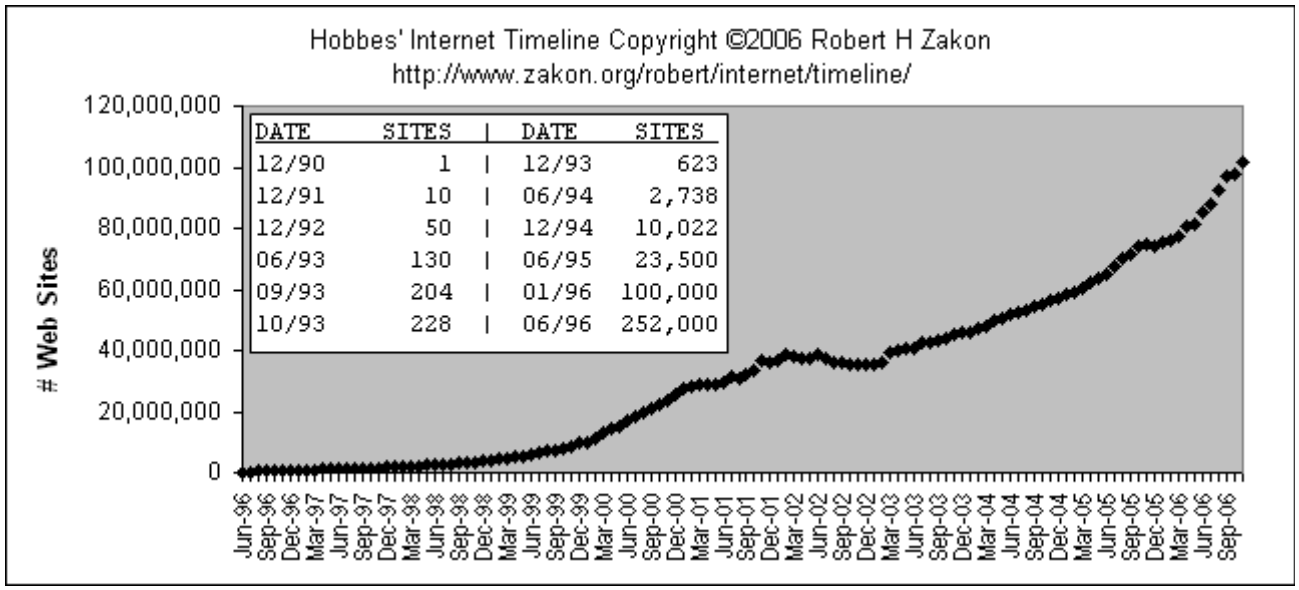
World Wide Web: introduzione e componenti

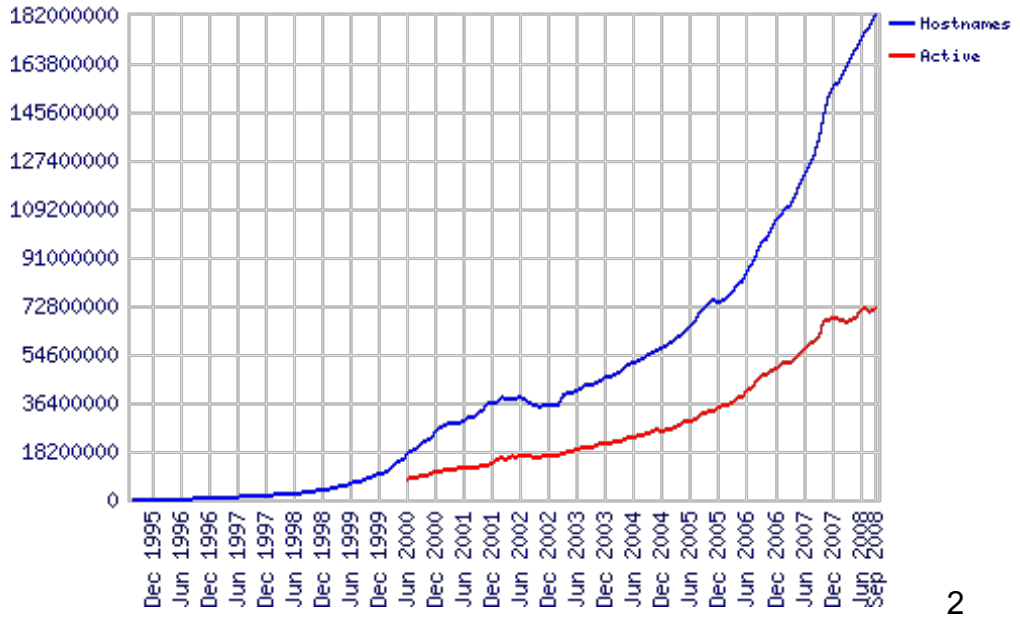
I segnali del successo del Web



DATE	SITES	DATE	SITES
12/90	1	12/93	623
12/91	10	06/94	2,738
12/92	50	12/94	10,022
06/93	130	06/95	23,500
09/93	204	01/96	100,000
10/93	228	06/96	252,000

Dal 2007 aumento esponenziale del numero di siti presso fornitori di servizi di blogging e social networking (MySpace, Live Spaces, Blogger, ..)

Fonte: Netcraft Web Server Survey (http://news.netcraft.com/archives/web_server_survey.html)



I segnali del successo del Web (2)

- Fino all'introduzione dei sistemi P2P, il Web è stata l'applicazione killer di Internet (75% del traffico di Internet nel 1998)

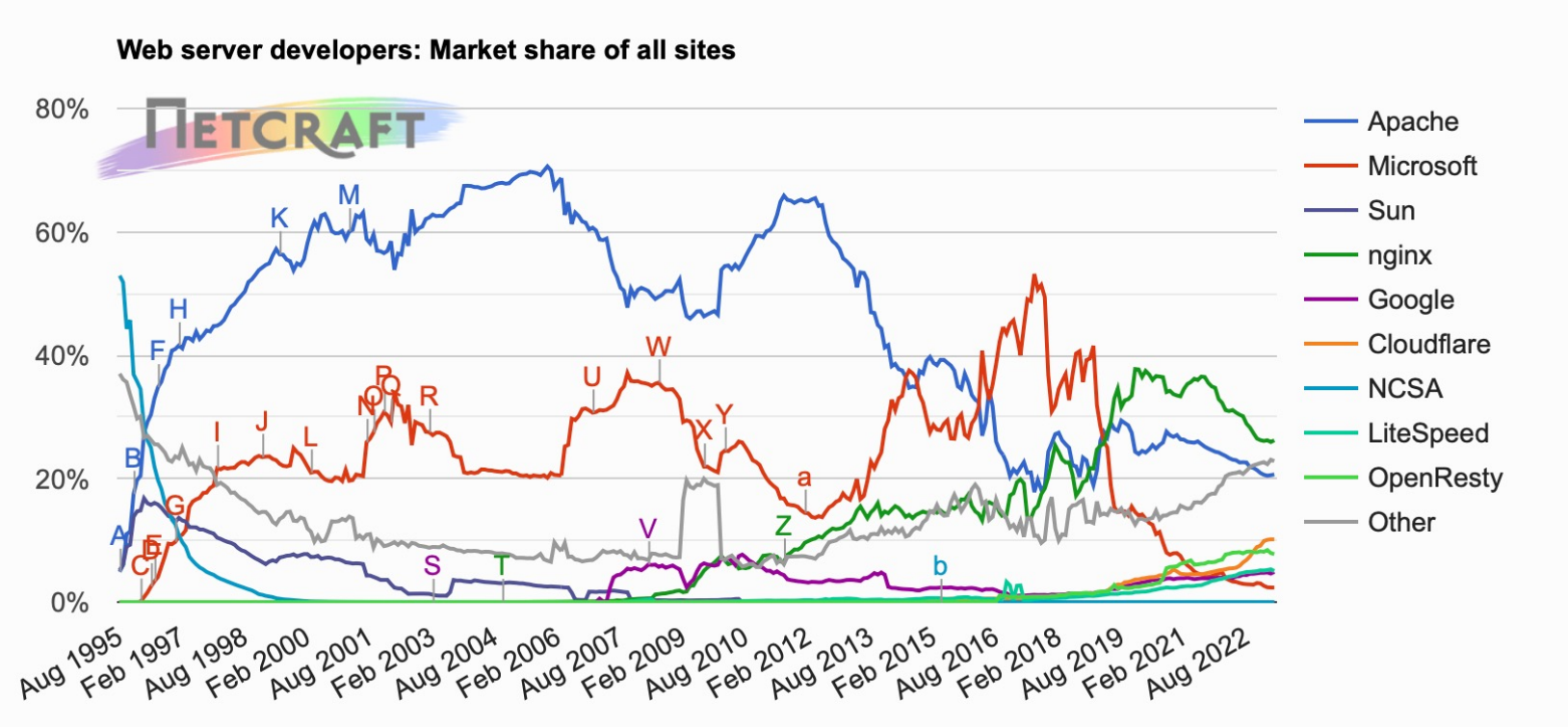
Event	*Period*	*Peak day*	*Peak minute*
NCSA server (Oct. 1995)	-	2 Million	-
Olympic Summer Games (Aug. 1996)	192 Million (17 days)	8 Million	-
Nasa Pathfinder (July 1997)	942 Million (14 days)	40 Million	-
Olympic Winter Games (Feb. 1998)	634.7 Million (16 days)	55 Million	110,000
Wimbledon (July 1998)	-	-	145,000
FIFA World Cup (July 1998)	1,350 Million (84 days)	73 Million	209,000
Wimbledon (July 1999)	-	125 Million	430,000
Wimbledon (July 2000)	-	282 Million	964,000
Olympic Summer Games (Sept. 2000)	-	875 Million	1,200,000

[Carico misurato in contatti]

- Inoltre: google.com, msn.com, yahoo.com (> 200 milioni di contatti al giorno)

Software per server Web

- I server Web più diffusi sono:
  - Apache (http://httpd.apache.org/)
  - Microsoft Internet Information Server (http://www.microsoft.com/)
  - Google Web server



From SD - Valeria Cardellini, A.A. 2008/09

Modelli architetturali di server Web

- Diversi approcci per l'architettura del server
  - Basato su processi
    - Fork e preforking
    - Esempio: Apache 1.X e Apache 2.X con mpm_prefork
  - Basato su thread
    - Esempio: Microsoft IIS
  - Ibrido (processi e thread)
    - Apache 2.X con mpm_worker
  - Basato su eventi
    - Esempio: Flash, Zeus, nginx, mpm_event
  - Interno al kernel
    - Esempio: Tux
- Nella scelta del modello tradeoff tra:
  - Prestazioni, robustezza, protezione, estensibilità, …

Server multi-process: fork

- Server basato su processi
- Per ogni nuova richiesta che arriva il processo server (padre):
  - Crea una copia di se stesso (un processo child), alla quale affida la gestione della richiesta tramite fork()
  - Si pone in attesa di nuove richieste
  - Il processo child si occupa di soddisfare la richiesta e poi termina
    - Un processo child per ogni client
  - Con fork():
    - Copia di dati, heap e stack; condivisione del segmento testo
    - No copia completa ma *copy-on-write*: è una ottimizzazione

Server multi-process: fork (3)

- Vantaggi:
  - Il codice del server rimane semplice, poiché la copia è demandata in toto al sistema operativo
- Svantaggi:
  - Overhead di fork() può penalizzare l'efficienza del sistema
    - Il tempo di generazione del processo child può non essere trascurabile rispetto al tempo di gestione della richiesta
  - In mancanza di un limite superiore al numero di richieste che possono essere gestite concorrentemente, nel caso di un elevato numero di richieste i processi child possono esaurire le risorse del sistema
  - Meccanismo di IPC per la condivisione di informazioni tra padre e figli successivamente a fork()
    - Ad esempio memoria condivisa

Server multi-process: helper

- Server basato su processi
- Un processo *dispatcher* (o listener) ed alcuni processi per il servizio delle richieste, detti processi *helper* (o worker)
  - All'avvio del servizio, il dispatcher effettua il *preforking* dei processi helper (*pool* di processi helper)
  - Il dispatcher rimane in ascolto delle richieste di connessione
  - Quando arriva una richiesta di connessione, il dispatcher la trasferisce ad un helper per la gestione
    - Occorre usare una forma di passaggio di descrittori tra processi distinti
  - Quando l'helper termina la gestione della richiesta, si rende disponibile per gestire una nuova richiesta
  - A regime, il dispatcher svolge compiti di supervisione e controllo

Server multi-process: helper (2)

- Vantaggi
  - Processi helper creati una sola volta e poi riusati
    - Si evita l'overhead dovuto alla fork() all'arrivo di ogni nuova richiesta
  - Maggiore robustezza (separazione dello spazio di indirizzi) e portabilità rispetto al server multi-threaded
  - Maggiore semplicità rispetto al server basato su eventi (linearità nel modo di pensare del programmatore)
- Svantaggi
  - Processo dispatcher potenziale collo di bottiglia
  - Gestione del numero di processi helper nel pool
  - Maggior uso di memoria rispetto a server multi-threaded
  - Gestione della condivisione di informazioni tra i processi helper (uso di lock)

Schemi per preforking

- Schema con preforking considerato finora:
  - Il dispatcher effettua accept() e passa il descrittore del socket di connessione ad un helper
  - Anche noto come schema *job-queue*
    - Il dispatcher è il produttore; gli helper sono i consumatori
    - Il dispatcher accetta le richieste e le pone in una coda
    - Gli helper leggono le richieste dalla coda e le servono
- In alternativa, si può usare lo schema *leader-follower*

Schema leader-follower

- Dopo il preforking, ogni helper chiama accept() sul socket di ascolto
- Soluzione 1: nessuna forma di locking per accept
  - Problema a: *thundering herd*
  - Problema b: funziona correttamente su kernel Unix derivati da Berkeley (accept implementata nel kernel), ma non su kernel Unix derivati da System V (accept come funzione di libreria)
    - Il socket d'ascolto è una risorsa a cui accedere in mutua esclusione
- Soluzione 2: file locking per accept
  - Dopo il preforking, ogni helper chiama accept(), effettuando un *file locking* prima dell'invocazione di accept()
  - Vedi esempio: file locking Posix con funzione fcntl()
- Gli helper idle competono per accedere al socket d'ascolto
  - Al più uno (detto *leader*) si può trovare in ascolto, mentre gli altri (detti *follower*) sono accodati in attesa di poter accedere alla sezione critica per il socket d'ascolto

Preforking con file locking

```
static int              nchildren;
static pid_t            *pids;
int main(int argc, char **argv)
{
    int                 listenfd, i;
    socklen_t           addrlen;
    void                sig_int(int);
    pid_t               child_make(int, int, int);

    …                   /* creazione del socket di ascolto, bind() e listen() */
    pids = calloc(nchildren, sizeof(pid_t));
    my_lock_init("/tmp/lock.XXXXXX"); /* un file di lock per tutti i processi child */
    for (i = 0; i < nchildren; i++)
            pids[i] = child_make(i, listenfd, addrlen);
    …
}

pid_t child_make(int i, int listenfd, int addrlen)
{
    pid_t pid;
```

Preforking con file locking (2)

```
    if ( (pid = fork()) > 0)
        return(pid);                    /* processo padre */
    child_main(i, listenfd, addrlen);   /* non ritorna mai */
}

void child_main(int i, int listenfd, int addrlen)
{
    int     connfd;
    socklen_t clilen;
    struct sockaddr *cliaddr;

    cliaddr = malloc(addrlen);
    printf("child %ld starting\n", (long) getpid());
    for ( ; ; ) {
        clilen = addrlen;
        my_lock_wait();                 /* my_lock_wait() usa fcntl() */
        connfd = accept(listenfd, cliaddr, &clilen);
        my_lock_release();
        web_child(connfd);              /* processa la richiesta */
        close(connfd);
    }
}
```

Preforking con file locking (3)

```
#include <fcntl.h>
#include "basic.h“
static struct flock  lock_it, unlock_it;
static int  lock_fd = -1;
     /* fcntl() will fail if my_lock_init() not called */

void my_lock_init(char *pathname)
{
  char  lock_file[1024];  /* must copy caller's string, in case it's a constant */

  strncpy(lock_file, pathname, sizeof(lock_file));
  if ( (lock_fd = mkstemp(lock_file)) < 0) {
    fprintf(stderr, "errore in mkstemp");
    exit(1);
  }
  if (unlink(lock_file) == -1) {              /* but lock_fd remains open */
    fprintf(stderr, "errore in unlink per %s", lock_file);
    exit(1);
  }
```

Preforking con file locking (4)

```
  lock_it.l_type = F_WRLCK;
  lock_it.l_whence = SEEK_SET;
  lock_it.l_start = 0;
  lock_it.l_len = 0;

  unlock_it.l_type = F_UNLCK;
  unlock_it.l_whence = SEEK_SET;
  unlock_it.l_start = 0;
  unlock_it.l_len = 0;
}
```

Preforking con file locking (5)

```
void my_lock_wait()
{
  int rc;
  while ( (rc = fcntl(lock_fd, F_SETLKW, &lock_it)) < 0) {
    if (errno == EINTR) continue;
    else {
      fprintf(stderr, "errore fcntl in my_lock_wait");
      exit(1);
    }
  }
}

void my_lock_release()
{
  if (fcntl(lock_fd, F_SETLKW, &unlock_it) < 0) {
    fprintf(stderr, "errore fcntl in my_lock_release");
    exit(1);
  }
}
```

Server multi-threaded

- Server basato su thread
- Una sola copia del server che genera thread multipli di esecuzione
  - Il thread principale rimane sempre in ascolto delle richieste
  - Quando arriva una richiesta, esso genera un nuovo thread (un *request handler*), che la gestisce e poi (eventualmente) termina
  - Ogni thread possiede una copia privata della connessione gestita, ma condivide con gli altri thread uno spazio di memoria (codice del programma e variabili globali)
- In alternativa alla creazione del thread all'arrivo della richiesta, il pool di thread può essere pre-creato (*prethreading*)

Server multi-threaded (2)

- Vantaggi
  - Creazione di un thread più veloce della creazione di un processo
    - Da 10 a 100 volte
  - Minore overhead per il context switching
  - Condivisione delle informazioni per default
  - Mantiene una maggiore semplicità rispetto a server basato su eventi
- Svantaggi
  - Maggiore complessità del codice del server (gestione della sincronizzazione tra thread)
  - Minore robustezza rispetto al server multi-process: i thread non sono protetti uno dall'altro
  - Supporto da parte del sistema operativo al multithreading (ad es., Linux/Unix, Windows)
  - Maggiori limitazioni sul numero di risorse rispetto al preforking (ad es. numero di descrittori aperti)

Preforking e prethreading

- Riassumiamo le possibili alternative per realizzare un server con preforking o prethreading
- Preforked server
  - Dispatcher/listener
    - Il processo padre invoca accept; occorre passare da padre a figlio il descrittore del socket di connessione
  - Leader/follower
    - Nessuna forma di locking per accept
    - File locking per accept
    - Mutex per proteggere accept
- Prethreaded server
  - Dispatcher/listener
    - Il thread principale invoca accept; non occorre passare il descrittore da un thread all'altro, perché i thread condividono tutti i descrittori
  - Leader/follower
    - Mutex per proteggere accept

Dimensione del pool

- Comportamento della dimensione del pool di processi o thread
  - Scelta significativa nell'architettura software di un server basato su processi o thread
- Alternative: dimensione statica o dinamica

- Pool di dimensione *statica* (ad es. *p*)
  - Carico alto: se i *p* processi o thread del pool sono occupati, una nuova richiesta deve attendere
  - Carico basso: la maggior parte dei processi o thread sono idle (spreco di risorse)

- Pool di dimensione *dinamica*
  - Il numero di processi o thread varia con il carico: cresce se il carico è alto, diminuisce se il carico è basso
  - Tipicamente, c'è un minimo numero di processi o thread idle
  - Esempio: Apache

Server ibrido

- Server basato su processi e thread
- Molteplici processi, ciascuno dei quali è multi-threaded
  - Un singolo processo di controllo (processo padre) lancia i processi figli
  - Ciascun processo figlio crea un certo numero di thread di servizio ed un thread listener
  - Quando arriva una richiesta, il thread listener la passa ad un thread di servizio che la gestisce
- Combina i vantaggi delle architetture basata su processi e basata su thread, riducendo i loro svantaggi
  - In grado di servire un maggior numero di richieste usando una minore quantità di risorse rispetto all'architettura basata su processi
  - Conserva in gran parte la robustezza e stabilità dell'architettura basata su processi

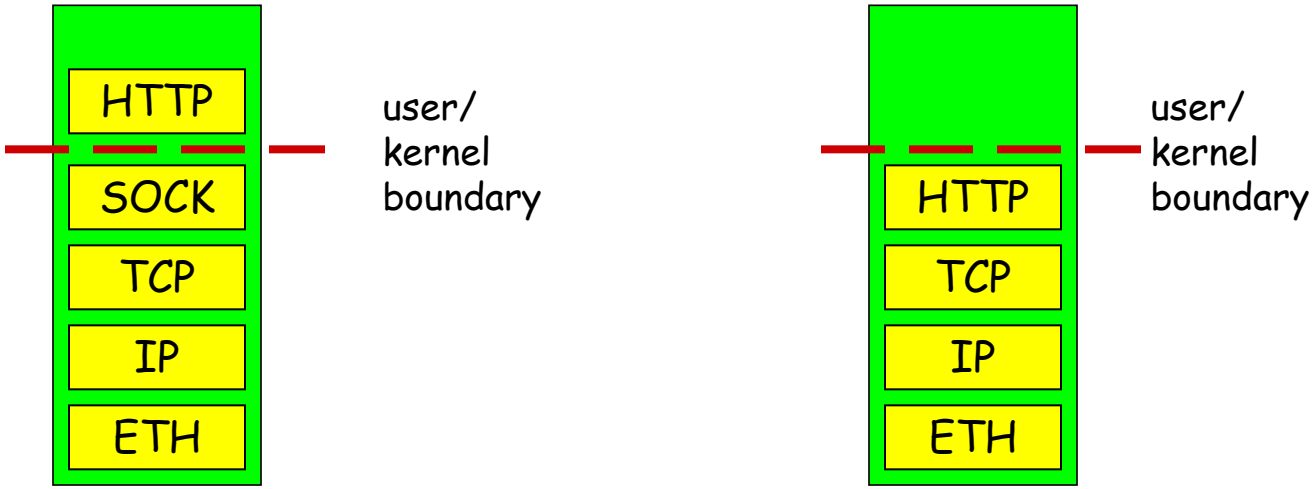
Server basato su eventi

- Un solo processo che gestisce le richieste in modo event-driven
  - Anziché servire una singola richiesta nella sua interezza, il server esegue una piccola parte di servizio per conto di ciascuna richiesta
- Uso di select(), opzioni non bloccanti sui socket, gestione asincrona dell'I/O
  - Il server continua l'esecuzione mentre aspetta di ricevere una risposta alla chiamata di sistema da parte del sistema operativo

Server basato su eventi (2)

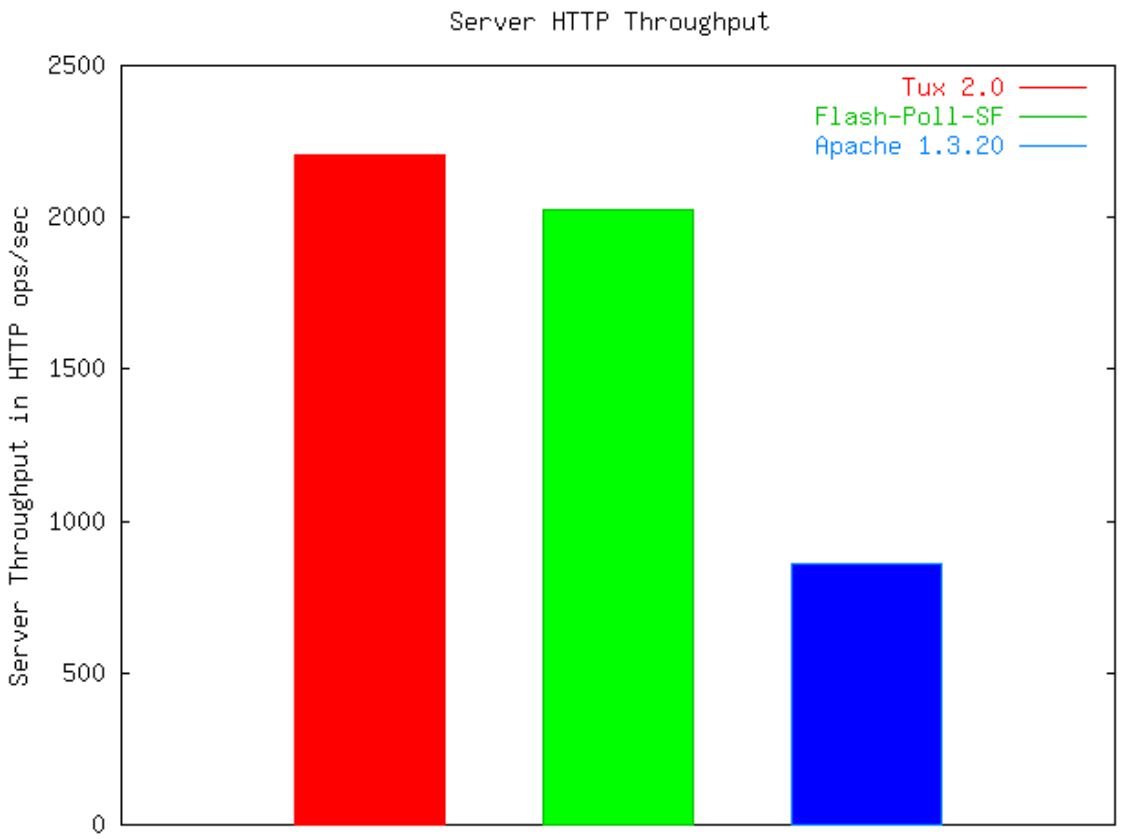
- Vantaggi
  - Molto veloce
  - La condivisione è intrinseca: un solo processo
  - Non c'è bisogno di sincronizzazione come nel server multi-threaded
  - Non ci sono overhead dovuti al context switching o consumi extra di memoria
- Svantaggi
  - Maggiore complessità nella progettazione ed implementazione
  - Meno robusto: una failure può fermare l'intero server
  - Limiti delle risorse per processo (es. descrittori di file)
  - Supporto in tutti i sistemi operativi di I/O asincrono

Server interno al kernel



- Thread del kernel dedicato per richieste HTTP:
  - Prima alternativa: tutto il server nel kernel
  - Seconda alternativa: gestione delle richieste GET statiche nel kernel, mentre richieste dinamiche gestite da un server (es. Apache) nello spazio utente
- Tux rimosso dal Linux kernel 2.6
  - Tuttavia, alcune distribuzioni (es. Fedora) lo hanno rimesso nel kernel 2.6

Confronto delle prestazioni



Tux: interno al kernel
Flash: basato su eventi
Apache 1.3: preforking

Fonte: E. Nahum, “Web servers: Implementation and performance”

- Il grafo mostra il throughput per Tux, Flash e Apache
- Esperimenti su P/II 400 MHz, gigabit Ethernet, Linux 2.4.9-ac10, 8 macchine client, WaspClient come generatore di carico