Processi e concorrenza in distribuito

SCD

Anno accademico 2019/2020
Sistemi Concorrenti e Distribuiti

Tullio Vardanega, tullio.vardanega@unipd.it

Considerazioni di costo – 1

- Contesto di *processor*
  - I registri (e la semantica del loro contenuto)
- Contesto di *thread*
  - Il contesto del *processor*
  - La memoria contenente lo stato del *thread* (*stack*, *heap*)
- Contesto di *process*
  - Il contesto dei *thread*
  - La memoria virtuale assegnata al processo, il cui spazio di indirizzamento è condiviso dai suoi *thread*

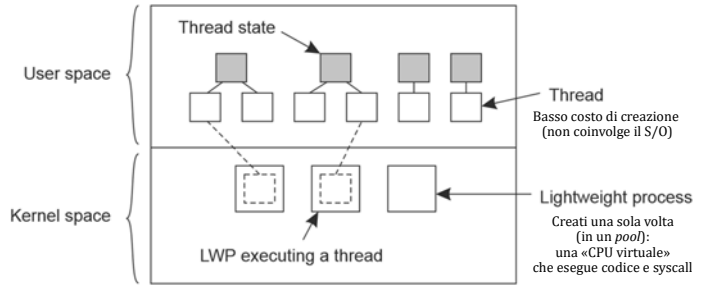
Considerazioni di costo – 2

- Al *context switch* tra *thread* è più veloce se non coinvolge il S/O
  - Può farlo perché si risolve all'interno dello stesso spazio di memoria virtuale di processo
  - All'estremo, i *thread* possono non essere noti al S/O
- Il *context switch* tra *process* ha costo alto
  - Perché coinvolge necessariamente il S/O
- Creare e distruggere processi di S/O costa molto
- Farlo con i *thread* costa meno ma non poco

Considerazioni di costo – 3

- Coinvolgere il S/O nella gestione dei *thread* non è conveniente
  - Ogni operazione a livello *thread* (p.es., gestione di I/O bloccante e di eventi esterni) coinvolgerebbe il S/O
- Soluzioni nello spazio utente sono possibili
  - Tuttavia, gli interventi del S/O, agendo sul processo, bloccano tutti i suoi *thread*, riducendone il parallelismo
- Serve un approccio più intelligente
  - LWP (*light-weight process*), nato in Solaris, poi acquisito in Linux

Considerazioni di costo – 4

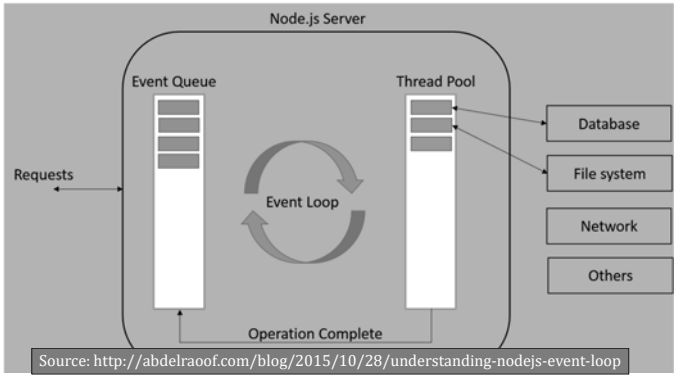


Le operazioni di S/O non rompono il legame tra *thread* (in *user space*) e LWP (in *kernel space*)

Considerazioni di costo – 5

- **L'uso di *thread* nell'applicazione richiede un importante sforzo di pensiero concorrente e di sua gestione**
- **La facilità di creazione (*new*) può indurre al consumismo**
  - **Esempio: Apache crea un *thread* per ogni richiesta HTTP senza curarsi del rapporto costi-benefici rispetto ai dati trasportati**
- **Per applicazioni *IO-bound* è preferibile usare eventi**
  - **Esempio: Node.js: un singolo *thread* per programma**
  - **Esecuzione a «*event loop*» per ogni azione bloccante pendente**
  - **Il *thread* serve la coda di *callback* («TODO») del programma fino a esaurimento**

Il *server* in Node.js

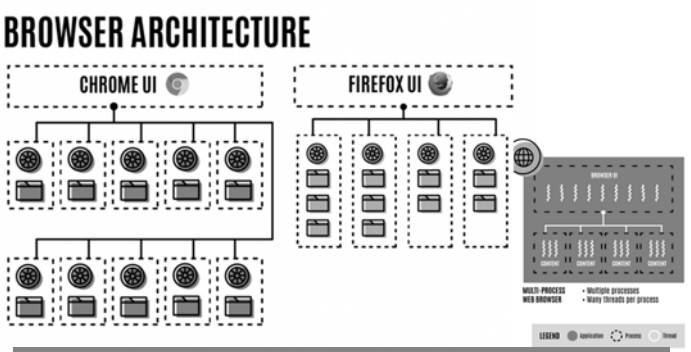


Source: http://abdelraoof.com/blog/2015/10/28/understanding-nodejs-event-loop

Cliente e servente concorrenti – 1

- ***Multi-threading* di lato cliente**
  - **La concorrenza interna mitiga l'effetto del ritardo di rete**
    - In un *Web browser* (lato cliente) conviene eseguire in parallelo
    - L'attivazione della connessione TCP/IP è operazione bloccante
    - Lettura ed elaborazione dei dati in ingresso sono eseguibili in *pipeline*
    - Il trasferimento su video è eseguibile in *pipeline*
  - **Google Chrome (2008) primo *browser multi-threaded* (!)**
    - Firefox lo è diventato dalla versione 54 (2017)
  - **Il cliente può supportare più sessioni parallele**
    - In Chrome, un processo (con *browser engine*) per *tab*
    - In Firefox, un processo (con *browser engine*) per ciascuno dei primi 4 *tab*, poi *thread*
  - **Uno dei presupposti su cui si basa AJAX**
    - Vedi p.es.: http://www.cmarshall.net/MySoftware/ajax/Threads/

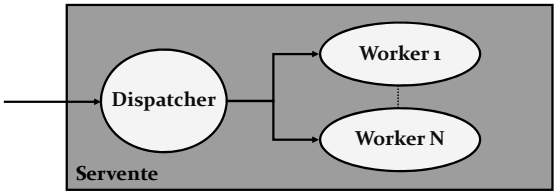
Chrome *vs.* Firefox



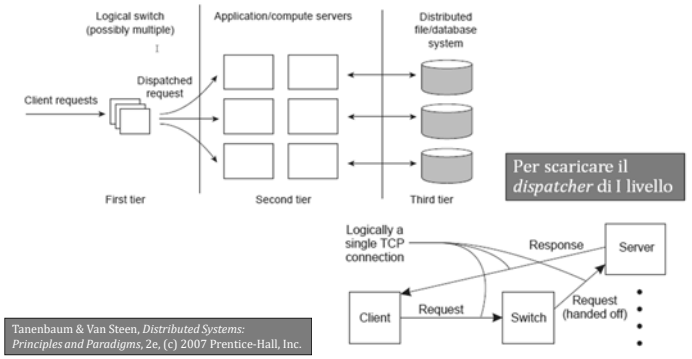
www.extremetech.com/internet/250930-firefox-54-finally-supports-multithreading-claims-higher-ram-efficiency-chrome

Cliente e servente concorrenti – 2

- ***Multi-threading*** **di lato servente**
  - **La concorrenza interna offre**
    - Maggiore efficienza prestazionale ancor più utile e desiderabile che nel cliente
    - Maggiore modularità (specializzazione, semplicità) architetturale

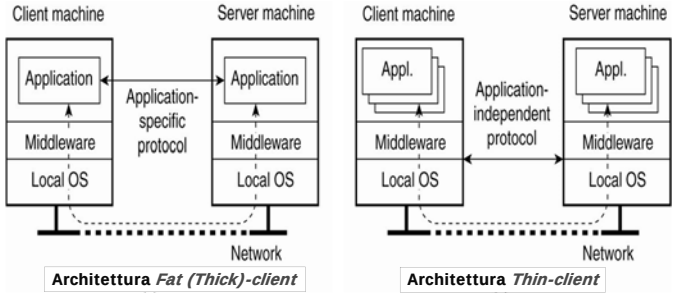


Corto-circuitazione: TCP *hand-off*



Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Problematiche di lato cliente – 1



Architettura *Fat (Thick)-client* **Architettura *Thin-client***

Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

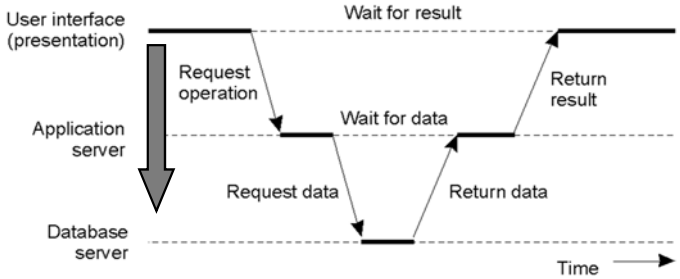
Problematiche di lato cliente – 2

- Un *thin client* sa solo riflettere quello che riceve dal *server* tramite la rete
  - Non sa cosa fare in assenza di comunicazioni dal *server*
  - L'architettura dell'X-Window System (X11, oggi l'X.org di Linux) era basata su questo paradigma
- Un *fat (thick) client* sa svolgere lavoro in proprio
  - Ha cose da fare anche in assenza di comunicazioni di rete
  - Quindi scarica di oneri il *server*
- Come classificare le *single-page web app*?

Lato servente: organizzazione

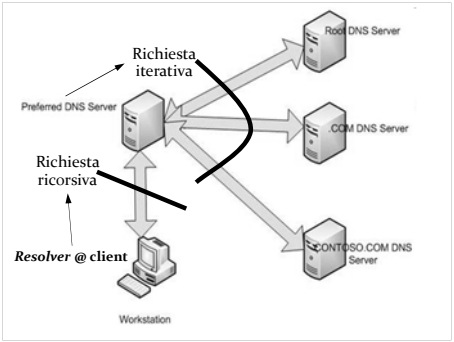
- Organizzazione iterativa o ricorsiva → distribuzione verticale
  - Il servente utilizza i servizi di altri serventi (interni o esterni)
  - La richiesta seguente viene servita dopo la fine di quella corrente
  - Per avere maggior *throughput* bisogna replicare l'intero servente
- Organizzazione concorrente → distribuzione orizzontale
  - Il *front-end dispatcher* del servente si limita a inoltrare le richieste a un *worker thread* distinto secondo qualche politica
  - Le richieste seguenti devono aspettare solo la fine inoltro
  - Per avere maggior *throughput* basta replicare gli esecutori, facendo però attenzione alle problematiche «Apache»

Distribuzione verticale – 1



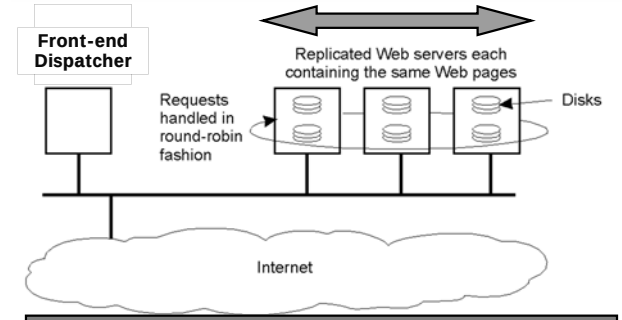
Nell'architettura a **distribuzione verticale** il servente visto dal cliente può essere esso stesso cliente di un componente servente cui sia stata demandata parte del servizio

Distribuzione verticale – 2



- Paradigma di *name resolution* del DNS
- La richiesta iterativa sposta l'onere chi la emette
- La richiesta ricorsiva sposta l'onere su chi la riceve

Distribuzione orizzontale

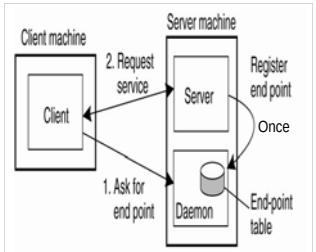


Nell'architettura a **distribuzione orizzontale** la parte più onerosa del servizio può essere completamente replicata su più elaboratori distinti operanti in parallelo

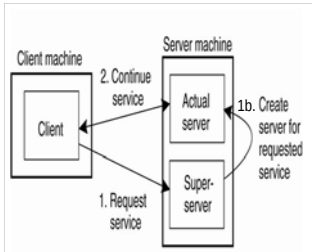
Lato servente: localizzazione – 1

- **Al servente corrisponde una porta (*end-point*) del suo nodo di residenza**
  - **Su di essa ascolta un processo dedicato**
- **Per serventi base, la porta può essere preassegnata da IANA (*Internet Assigned Numbers Authority*)**
- **Per gli altri, viene assegnata dinamicamente**
  - **Un *daemon* ascolta su porta convenzionale le richieste in arrivo e poi ne assegna una dinamicamente per ogni servizio gestito**
  - **Un Super-Servente (p.es. `inetd` in UNIX) ascolta tutte le porte dei serventi gestiti e li crea «al volo» secondo bisogno**

Lato servente: localizzazione – 2



Assegnazione dinamica di porta servente (interazione tramite *daemon*)



Attivazione dinamica di servente (interazione tramite *super-server*)

Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Lato servente: stato – 1

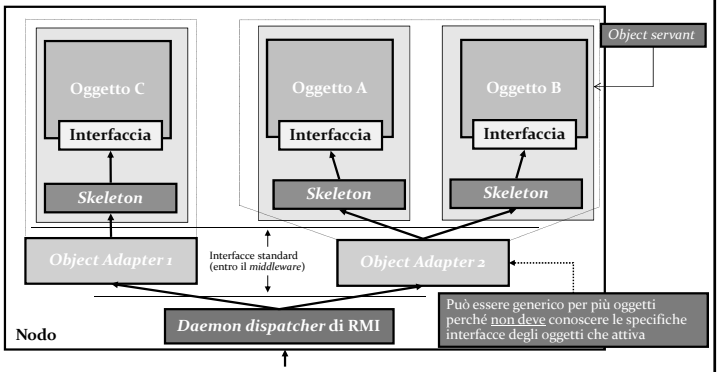
- **Servente *stateless* → lo stato non è nel servizio**
  - **Non ricorda lo stato di servizio del cliente, e non deve informarlo di eventuali suoi cambi di stato**
  - **Questa scelta è all'origine dei *cookies* !**

- **L'esempio storico è NFS**
  - **Il cliente opera localmente su *virtual inode*, con *cache write-through* (con scrittura asincrona), non coerente tra clienti diversi**
  - **Il servente tratta ogni operazione in sessione distinta**
  - **Il *file system* di lato servente può cambiare locazione, stato ed esistenza dei propri *file* senza doverne informare alcun cliente**
- **La *statelessness* è la base della scalabilità elastica**

Lato servente: stato – 2

- **Servente *stateful* → lo stato è nel servizio**
  - **Ricorda lo stato di servizio del cliente e offre sempre stato di servizio coerente**
  - **L'esempio classico è nei sistemi transazionali**
    - $begin\ (Op_1, Op_2, \ldots, Op_n)\ commit$
    - ***Atomicity***: gli effetti sullo stato sono di tipo *all-or-nothing*
    - ***Consistency***: lo stato di lato servente è sempre consistente (risultante da transazioni eseguite in un qualche ordine totale)
    - ***Independence*** (***isolation***): le transazioni non interferenti possono eseguire in parallelo
    - ***Durability***: gli effetti delle transazioni terminate con successo sono persistiti
- **Promesse non scalabili elasticamente al variare dello stato**

Servente di oggetto – 1



Servente di oggetto – 2

- **Ospita la concretizzazione dell'oggetto distribuito**
  - **Non fornisce alcun servizio in proprio**
  - **È il tramite per l'invocazione locale per conto del vero chiamante remoto**
- **La sua implementazione determina la separazione effettiva tra l'interfaccia e lo stato dell'oggetto**
- **Può supportare diverse politiche di attivazione dell'oggetto distribuito**
  - **Modello più potente ed espressivo di RPC**

Politiche di attivazione – 1

- **Fissano il ciclo di vita dell'oggetto remoto**
- **Creazione / distruzione dell'*object reference***
  - **Ciò che rende disponibile al cliente l'entità che realizza le operazioni dell'interfaccia remota**
- **Attivazione / de-attivazione del *servant***
  - **L'insieme di risorse (CPU, memoria) che realizzano l'oggetto nel servente**

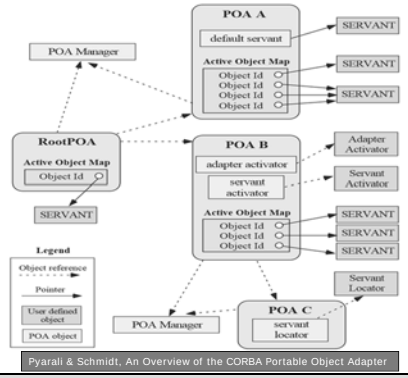
Politiche di attivazione – 2

- Politiche comuni per nodo sono attuate da un singolo *object adapter*
  - Noto *design pattern* della GoF
  - "*A reusable class that cooperates with unrelated or unforeseen classes*"
- Un OA fornisce metodi per
  - Ricevere invocazioni remote in arrivo dal MW e inviarle al *servant* destinatario [ruolo funzionale]
  - Registrare/rimuovere *servant* e abilitare politiche di servizio [ruolo amministrativo]

Compiti dell'*object adapter*

- Registrare implementazioni di interfacce
- Mappare, generare, interpretare riferimenti a tali implementazioni
- Attivare e deattivare *servant* e relativa implementazione
- Inoltrare invocazioni di metodi ai *servant* corrispondenti
- Partecipare a garantire la sicurezza delle interazioni con il cliente

Portable Object Adapter



Pyarali & Schmidt, An Overview of the CORBA Portable Object Adapter

Organizzazione a micro-servizi

In Pursuit of Architectural Agility: Experimenting with Microservices

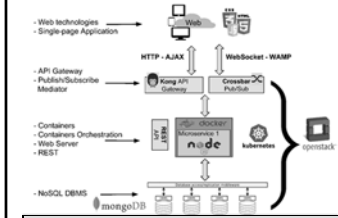
Published in: 2018 IEEE International Conference on Services Computing (SCC)

Date of Conference: 2-7 July 2018 **INSPEC Accession Number:** 18076512

Date Added to IEEE *Xplore*: 06 September 2018 **DOI:** 10.1109/SCC.2018.00022

ISBN Information: **Publisher:** IEEE

Electronic ISSN: 2474-2473 **Conference Location:** San Francisco, CA, USA



II. The Microservices Approach: A Short Recap

The term "microservices" designates an architectural style that yields a single application from the coordination of a suite of unitary services [5]. Such services expose an Application Program Interface (API) *outside* of their codebase (a central trait of their specific composition style), which the user invokes using *asynchronous* (crucial to loose coupling) *web-based* service requests (key to reachability).

A microservice is understood as a small self-contained application that has a single responsibility, a lightweight stack, and can be deployed, scaled and tested independently