Processi e concorrenza in distribuito

SCD

Anno accademico 2014/15
Sistemi Concorrenti e Distribuiti

Tullio Vardanega, tullio.vardanega@math.unipd.it

Considerazioni di costo – 1

- Contesto di *processor*
  - I registri
- Contesto di *thread*
  - Il contesto del *processor* e la memoria che contiene lo stato del *thread* (*stack*, *heap*, ...)
- Contesto di *process*
  - Il contesto dei *thread* e lo stato della memoria virtuale del processo
  - I *thread* di un processo condividono lo stesso spazio di indirizzamento

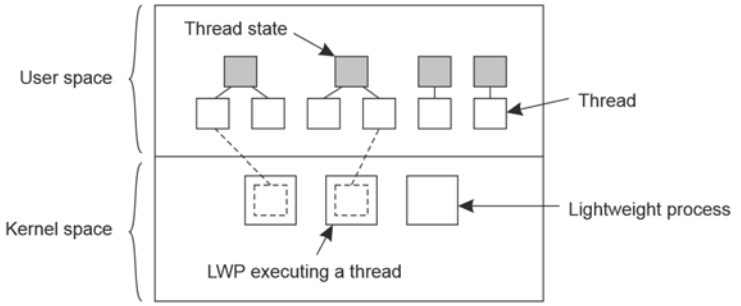
Considerazioni di costo – 2

- Il *context switch* a livello di *thread* non ha bisogno dell'intervento del S/O
- Il *context switch* a livello di *process* ha costo alto perché, operando a livello di virtualizzazione (memoria), deve coinvolgere il S/O
- Creare e distruggere *thread* costa molto meno che farlo con processi

Considerazioni di costo – 3

- Non è evidente che convenga supportare *thread* a livello di S/O (*kernel*)
  - Ogni operazione a livello *thread* (gestione di I/O bloccante e di eventi esterni) coinvolge il S/O
- Soluzioni nello spazio utente sono possibili
  - Ma gli interventi del S/O hanno effetto su tutti i *thread* del processo, riducendone il parallelismo
  - Serve un approccio «intelligente»: LWP (*light-weight process*) nato in Solaris e ritenuto 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

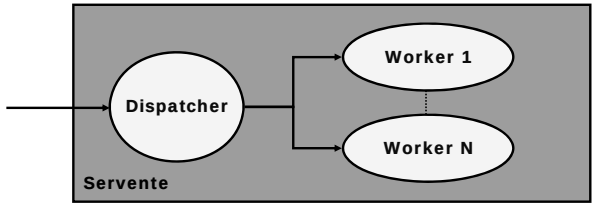
- **L'uso di *thread* nell'applicazione richiede un importante sforzo di logica concorrente**
  - **Oppure un uso massiccio di memoria: Apache crea un *thread* per ogni richiesta HTTP a prescindere dal rapporto costi-benefici rispetto ai dati da essa trasferiti**
- **Per applicazioni più «pragmatiche» può convenire un approccio a eventi**
  - **Node.js: un singolo *thread* per programma**
    - Esecuzione a «***event loop***» per ogni azione bloccante effettuata
    - Il *thread* serve la coda di *callback* del programma fino al suo esaurimento

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* (!)**
  - **Il cliente può supportare più sessioni parallele**
    - P.es., i "tab" di un browser?
  - **Uno dei presupposti su cui si basa AJAX**
    - P.es.: http://www.cmarshall.net/MySoftware/ajax/Threads/

Cliente e servente concorrenti – 2

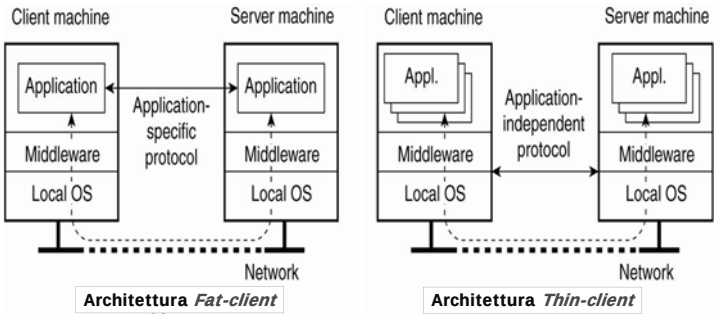
- ***Multi-threading* di lato servente**
  - **La concorrenza interna offre**
    - Maggiore efficienza prestazionale ancor più utile e desiderabile che nel cliente
    - Maggiore modularità (specializzazione, semplicità) architetturale



Problematiche di lato cliente – 1

Trasparenza	Ruolo del MW di lato cliente
Accesso	Fondamentale – a carico di *stub* (RPC) o *proxy* (RMI)
Collocazione	Fondamentale – tramite gestione delle corrispondenze nome-indirizzo (*naming*)
Migrazione / Spostamento	Desiderabile – serve *naming* a gestione dinamica
Replicazione	Utile per nascondere la possibile interazione con più repliche del servente
Transazione	Utile (ma molto di più dal lato servente)
Malfunzionamento	Desiderabile – p.es. il *caching* del *Web browser*
Persistenza	Non significativa (ma fondamentale dal lato servente)

Problematiche di lato cliente – 2



Architettura *Fat-client* **Architettura *Thin-client***

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

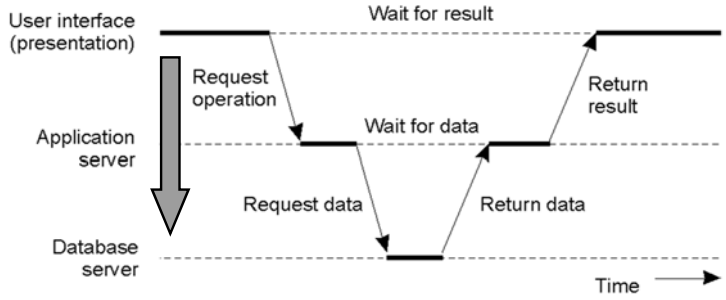
Problematiche di lato cliente – 2

- **Un *thin client* sa solo riflettere quello che riceve dal *server* tramite la rete**
  - **Non sa funzionare in assenza di comunicazioni dal *server***
  - **L'architettura dell'X-Window System (X11) era basata su questo paradigma**
- **Un *fat client* sa svolgere lavoro in proprio**
  - **Ha cose da fare anche in assenza di comunicazioni di rete**
  - **Quindi scarica di oneri il *server***

Problematiche di lato servente – 1

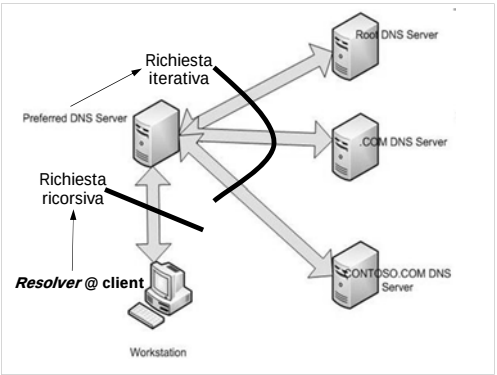
- **Due possibili organizzazioni**
  - **Iterativa o ricorsiva → distribuzione verticale**
    - Il servente utilizza i servizi di altri serventi (interni o esterni)
    - La richiesta successiva potrà essere accolta **solo dopo** il completamento di quella corrente
    - Per soddisfare più richieste in parallelo bisogna replicare l'intero servente
  - **Concorrente → distribuzione orizzontale**
    - Il *front-end dispatcher* del servente si occupa solo di accogliere richieste demandandone il soddisfacimento a un *thread* distinto
    - Nuove richieste possono essere accolte **appena** quella corrente sia stata affidata all'esecutore selezionato
    - Per soddisfare più richieste in parallelo basta replicare gli esecutori (1 *dispatcher* – N *worker*)

Distribuzione verticale – 1



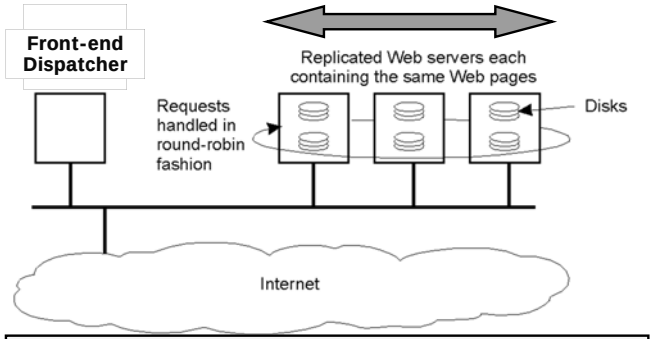
Nell'architettura a **distribuzione verticale** il servente visto dal cliente può essere esso stesso cliente di un componente servente cui sia stata demandata parte del servizio

Distribuzione verticale – 2



- **Paradigma di *name resolution* del DNS**
- **La richiesta iterativa sposta l'onere sul cliente**
- **La richiesta ricorsiva sposta l'onere sul servente**

Distribuzione orizzontale

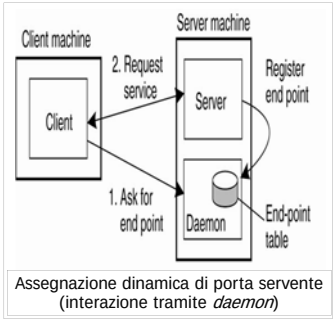


Nell'architettura a **distribuzione orizzontale** la parte più onerosa del servizio può essere completamente replicata su più elaboratori distinti operanti in parallelo

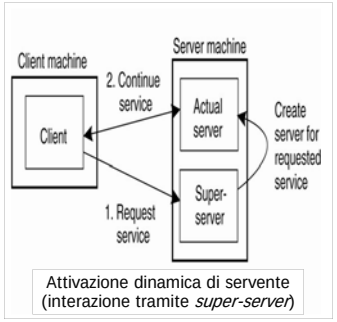
Problematiche di lato servente – 2

- **Localizzazione del servente**
  - **Al servente corrisponde una porta (*end-point*) del nodo sulla quale sta in ascolto un processo dedicato**
  - **Porta preassegnata**
    - **IANA** (***Internet Assigned Numbers Authority***) attribuisce porte a protocolli – p.es.: HTTP:80, FTP:20-1, SMTP:25, …
  - **Porta assegnata dinamicamente**
    - Un *daemon* ascolta su porta preassegnata le richieste in arrivo
    - Le richieste per lo stesso servizio sono girate su porta assegnata dinamicamente di cui viene informato il servente corrispondente
    - Se le richieste per un insieme di servizi sono rare si usa un Super-Servente che ascolta tutte le porte di quell'insieme e per ogni richiesta in arrivo risveglia (o crea dinamicamente) il servente corrispondente – p.es.: `inetd` di UNIX

Localizzazione del servente



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



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

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

Problematiche di lato servente – 3

- **Interrompibilità del servente**
  - **Modello TCP/IP**
    - La rottura della connessione (p.es. per abbandono del cliente) comporta interruzione del servizio
    - Non immediata ma garantita senza confusione con richieste successive
  - **Dati "*out-of-band*"**
    - Il cliente può chiedere di dare precedenza a dati fuori sequenza ma di maggiore urgenza
    - Designazione di urgenza nell'intestazione dei pacchetti dati (p.es. TCP)
    - Cliente e servente devono intrattenere più di una sottoconnessione logica entro la stessa connessione di servizio
    - Con porta distinta per ogni sottoconnessione

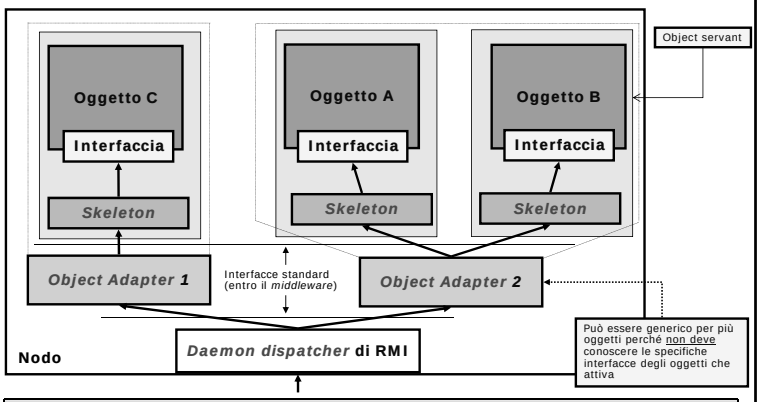
Problematiche di lato servente – 4

- **Servente senza stato (*stateless*)**
  - **Non ricorda lo stato di servizio del cliente e non deve informarlo di eventuali cambi di stato di lato servente**
  - **L'esempio classico è NFS**
    - Il cliente opera localmente su *virtual inode*
    - La *cache* di lato cliente è *write-through* ma la scrittura è asincrona
    - La *cache* cliente può essere invalidata da modifiche che originano altrove
    - Il servente tratta ogni operazione in una 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**

Problematiche di lato servente – 5

- **Servente con stato (*stateful*)**
  - **Ricorda lo stato di servizio del cliente**
  - **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 (come modificato da transazioni eseguite in un qualche ordine totale)
    - ***Independence***: le transazioni non interferenti possono eseguire in parallelo
    - ***Durability***: gli effetti delle transazioni terminate con successo sono persistiti
  - **Promesse totalmente non scalabili al crescere elastico dello stato**

Servente di oggetto – 1



Servente di oggetto – 2

- **È il tramite di invocazione locale per conto di clienti remoti**
  - **Non fornisce alcun servizio di per se**
- **La sua implementazione concreta determina come l'interfaccia e lo stato dell'oggetto remoto a lui associato possano essere separati**
- **Il servente di oggetto può supportare diverse «politiche di attivazione» dell'oggetto remoto**
  - **Più evoluto rispetto alla «disponibilità permanente» di RPC**

Servente di oggetto – 3

- **Le politiche di attivazione fissano le modalità (il ciclo di vita) con cui un oggetto remoto può essere invocato**
  - **Creazione e distruzione dell'*object reference***
    - Ciò che rende disponibile al cliente l'entità che realizza le operazioni dell'interfaccia
  - **Attivazione e de-attivazione del *servant***
    - L'insieme di risorse (CPU, memoria) che realizzano l'oggetto nel servente

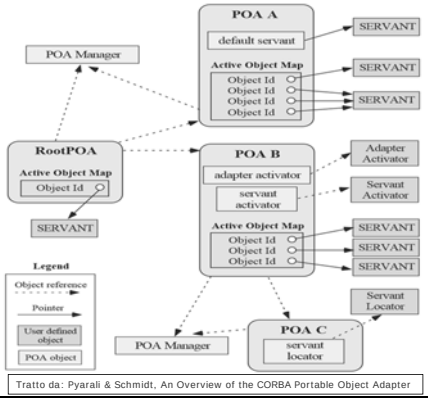
Servente di oggetto – 4

- **Politiche di attivazione comuni per nodo sono attuate da un singolo *object adapter***
  - **Un 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**
    - Servizio funzionale
  - **Registrare o rimuovere *servant* e influenzare le politiche di servizio**
    - Servizio amministrativo

Servente di oggetto – 5

- **I compiti dell'OA**
  - 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



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