Sistemi Distribuiti

Introduzione, Middleware, Client/Server, Messaggistica, RPC e Cluster

Corso: Sistemi di Calcolo 2
Docente: Riccardo Lazzeretti
Fonti indicate nelle slide: Coulouris *Distributed Systems: Concepts and Design* (cap. 1–2); Stallings *Operating Systems: Internals and Design Principles* (cap. 16)

Nota

Queste note traducono e riorganizzano fedelmente i contenuti delle slide del modulo *Distributed Systems*. Dove erano presenti figure, qui forniamo descrizioni testuali.

Indice

1 Introduzione ai sistemi distribuiti

1.1 Definizione

Un **sistema distribuito** è un insieme di entità spazialmente separate, ognuna con una certa capacità computazionale, in grado di *comunicare* e *coordinarsi* per raggiungere un obiettivo comune, apparendo agli utenti come un *unico sistema coerente*.

1.2 Perché svilupparli e cosa sono

Disponibilità di microprocessori potenti ed economici, e progressi continui nelle comunicazioni. In pratica: una collezione di calcolatori indipendenti che *appare* come un unico sistema.

1.3 Esempi e domini applicativi

- **Reti locali e intranet**, **Web**/Internet.
- **DBMS** distribuiti, **reti di sportelli ATM**.
- **Pervasive/Ubiquitous computing**, **SOA**, **virtual networks**.
- **Peer-to-peer (P2P)**, **Cloud Computing**, **Big Data**.

Domini d'esempio: eCommerce, motori di ricerca/wiki/social, gaming massivo online (MMOG), sanità (cartelle cliniche/monitoraggio), education (e-learning), logistica (GIS/GPS), *e-Science* (Grid), gestione ambientale (reti di sensori).[1]

1.4 MMOG

Richiedono risposte rapide, propagazione *real-time* degli eventi e visione *consistente* del mondo persistente (arena di gioco, sistemi sociali e finanziari).

2 Pro e contro

2.1 Vantaggi rispetto a sistemi centralizzati

Economia (miglior prezzo/prestazioni), velocità (potenza aggregata), distribuzione intrinseca (es. catene di supermercati), affidabilità/ disponibilità (il sistema sopravvive al guasto di un nodo), crescita incrementale (scalabilità modulare). Spinta ulteriore: proliferazione di PC/IoT e collaborazione.

2.2 Vantaggi rispetto a PC indipendenti

Condivisione dati/risorse (DB, periferiche costose), comunicazione (email/chat), flessibilità (bilanciamento carico).

2.3 Svantaggi

Software complesso; rete (saturazione, perdite); sicurezza (accesso non autorizzato).

Obiettivo primario Condivisione di dati/risorse con problemi di **sincronizzazione** e **coordinamento**.

3 Condizioni peculiari dei sistemi distribuiti

1. Concorrenza temporale e spaziale.
2. Assenza di **orologio globale**.
3. Guasti (nodi/rete).
4. Latenze imprevedibili.

Questi limiti restringono il set di problemi coordinabili in modo deterministico (richiamo al teorema CAP: *consistency vs availability vs partitions*).

4 Questioni di progetto

4.1 Eterogeneità e apertura

Eterogeneità di reti, HW, OS, linguaggi, implementazioni, codice mobile. **Openness**: interfacce pubbliche e meccanismo di comunicazione uniforme per accedere a risorse condivise; conformità agli standard.

[1] Le slide citano esempi quali Amazon/eBay, PayPal, Wikipedia, Facebook/MySpace, YouTube/Flickr, mappe Google, ecc.

4.2 Sicurezza

Confidenzialità (no disclosure), **integrità** (no corruption), **disponibilità** (no interferenza all'accesso).

4.3 Scalabilità

Controllo costi risorse fisiche e perdite prestazionali; prevenire esaurimento risorse software; evitare colli di bottiglia (componenti/tabelle/algoritmi centralizzati: *single mail server*, *URL address book*, routing "globale").

4.4 Affidabilità e tolleranza ai guasti

Più affidabile di un singolo sistema; tollerare/mascherare/rilevare/recuperare dai guasti; **ridondanza**. Esempio: 3 macchine con disponibilità $0.95 \Rightarrow 1 - 0.05^3$ disponibilità complessiva.

4.5 Concorrenza e prestazioni

Concorrenza spaziale/temporale; prestazioni affette da ritardi di comunicazione (fine-grain vs coarse-grain), e dai meccanismi di fault tolerance.

4.6 Flessibilità (microkernel vs monolitico)

Kernel monolitico (tutte le system call nel kernel) vs **microkernel** (minimi servizi: IPC, una parte di MM e scheduling, basso livello I/O).

5 Trasparenza

Accesso: stesse operazioni per risorse locali/remote. **Locazione**: posizione non percepita dall'utente. **Concorrenza**: accesso concorrente gestito (lock/unlock, ME). **Replica**: copie addizionali trasparenti. **Guasto**: fault nascosti agli utenti. **Migrazione**: risorse/client possono spostarsi senza cambiare nome. **Prestazioni**: riconfigurabile al variare del carico. **Scalabilità**: espandibile senza cambiare architetture/algoritmi. **Parallelismo**: uso automatico del parallelismo (obiettivo ambizioso).
Nota: non sempre si desidera trasparenza completa (trade-off).

6 Modelli di interazione e middleware

6.1 Client/Server e Peer-to-Peer

Due modelli base di interazione: **client/server** e **P2P**.

6.2 Stratificazione HW/SW e middleware

Descrizione: middleware che si estende su più macchine, tra applicazioni e OS/rete, fornendo astrazioni uniformi.

6.3 Problemi del middleware

Eterogeneità (OS/clock/dati/HW), asincronia locale (carico/interrupt), mancanza di conoscenza globale (propagazione lenta), asincronia di rete (tempi imprevedibili), guasti/partizionamenti, assenza di ordine globale, **trade-off consistenza–disponibilità–partizioni**.

6.4 Ruolo del middleware

Interfacce/protocolli standard tra applicazioni e software di comunicazione, per accesso uniforme alle risorse dovunque siano.

7 Client/Server Computing

7.1 Caratteristiche

Client: PC/workstation *user-friendly*. Server: servizi condivisi (DB, gestione risorse). Sfrutta TCP/IP come *substrato* di interoperabilità.

7.2 Componenti applicativi

Presentation logic (UI), **I/O logic** (validazione), **Business logic** (regole/calcoli), **Data storage logic** (vincoli, integrità referenziale, query). Questi componenti possono risiedere su programmi/host/linguaggi differenti.

7.3 Domande progettuali

Linguaggio, piattaforma, frequenza di cambiamento, responsabilità di manutenzione, durata dell'applicazione.

7.4 Classi di applicazioni C/S

Host-based (mainframe; non vero C/S). **Server-based** (server fa tutto; client = GUI). **Client-based** (app al client, DB al server). **Cooperative** (distribuzione ottimizzata; *fat vs thin client*).

7.5 Architettura a tre livelli

Client sottile; **middle-tier** come gateway/conversione protocolli/integrazione da fonti multiple; **backend** dati. *Descrizione*: figura con utente ↔ middle-tier ↔ backend.

7.6 Service Oriented Architecture (SOA)

Organizza funzioni in *servizi* riusabili (interni/esterni). Interfacce standard (più citate: XML/-HTTP, *Web services*). *Esempi d'uso*: composizione di servizi interni e di partner.

8 Messaggio e RPC

8.1 Message Passing

Primitive di base invio/ricezione; semantica di buffer/coda, blocco/non-blocco; indirizzamento diretto/indiretto. *Descrizione*: figura con primitive elementari.

8.2 Remote Procedure Calls (RPC)

Astrazione che consente di invocare procedure remote con semantica *call/return*. Ampiamente adottata; comunicazione generabile automaticamente; portabilità di client/server su piattaforme/OS diversi.

8.3 Architettura e meccanismo RPC

Descrizione: client stub ↔ runtime ↔ rete ↔ runtime ↔ server stub; marshaling/ unmarshaling parametri; binding; gestione errori. *Figura*: pipeline del meccanismo (come in slide).

8.4 Passaggio parametri e rappresentazione

Per *by value* è semplice; *by reference* richiede puntatori globali/identificatori e può non valere la pena; la rappresentazione/ formato può variare con i linguaggi, va definita (es. XDR).

8.5 Binding (associazione) client/server

Non persistente: connessione istituita alla chiamata e chiusa al ritorno (overhead elevato per chiamate frequenti). **Persistente**: connessione mantenuta per più chiamate e chiusa dopo inattività.

8.6 Sincrono vs asincrono

Sincrono: come una subroutine; prevedibile ma non sfrutta il parallelismo. **Asincrono**: non blocca il chiamante; le risposte arrivano quando pronte; consente parallelismo locale al client.

8.7 Semantiche di chiamata RPC sotto guasti

At-least-once: eseguita una o più volte; semplice (ritrasmissioni a timeout), adatta a operazioni idempotenti.
At-most-once: eseguita al più una volta; serve rilevamento duplicati; ok per non-idempotenti.
Exactly-once: local-call semantics anche con crash e riavvio; implica tracciamento/adoption di orfani; implementazione complessa.

9 Cluster

9.1 Definizione e benefici

Alternativa a SMP per alte prestazioni/alta disponibilità. Gruppo di *computer completi* interconnessi e coordinati come un'unica risorsa. Benefici: **scalabilità assoluta** (cluster enormi), **scalabilità incrementale** (aggiunta modulare), **alta disponibilità** (il guasto di un nodo non è critico), **miglior prezzo/prestazioni**.

9.2 Configurazioni di cluster

Descrizione delle varianti: *passive standby* (secondario prende il posto del primario; semplice ma costoso); *active secondary* (secondario anche per lavoro; meno costoso, più complesso); *server separati* (dischi separati con copia continua; alta disponibilità, overhead di rete); *server connessi ai dischi* (stessi dischi cablati, takeover; meno overhead, richiede mirroring/RAID); *server che condividono dischi* (accesso concorrente, basso overhead, serve lock manager, spesso con RAID).

9.3 Gestione guasti a livello OS

High availability: al guasto, le query in corso si perdono; se ritentate verranno servite da altro nodo; stato delle transazioni parziali non garantito; alta probabilità che le risorse restino in servizio.
Fault-tolerant: risorse sempre disponibili tramite dischi condivisi ridondanti, *rollback* di transazioni non committate e *commit* di quelle completate.

9.4 Fallover e fallback

Fallover: commutazione di applicazioni/dati dal sistema guasto a un altro nodo. **Fallback**: ripristino sul nodo originale quando riparato. Il fallback automatico è desiderabile solo se il problema è risolto; altrimenti rischio di "rimbalzo" delle risorse tra nodi.

9.5 Bilanciamento del carico

Capacità di distribuire il carico tra i nodi, includendo l'inserimento automatico di nuovi nodi nella schedulazione. Il middleware deve riconoscere che i servizi possono apparire/migrare su membri diversi del cluster.

9.6 Parallelizzare il calcolo

Parallelizing compiler: decide a *compile-time* le parti parallele (dipende dal problema e dal compilatore).
Parallelized application: il programmatore usa passaggio di messaggi per muovere i dati (massimo controllo, maggiore onere).
Parametric computing: eseguire la stessa applicazione molte volte cambiando condizioni iniziali/parametri; servono tool per organizzare, eseguire e gestire i job.

10 Appendice: viste logiche e modelli

10.1 Vista logica del middleware

Descrizione: schema a blocchi con API/applicazioni sopra, middleware al centro, OS/rete sotto; il middleware fornisce naming, security, RPC, code di messaggi, transazioni, directory, ecc.

10.2 Layering HW/SW

Descrizione: stratificazione dei livelli software e hardware, con interfacce tra livelli adiacenti.