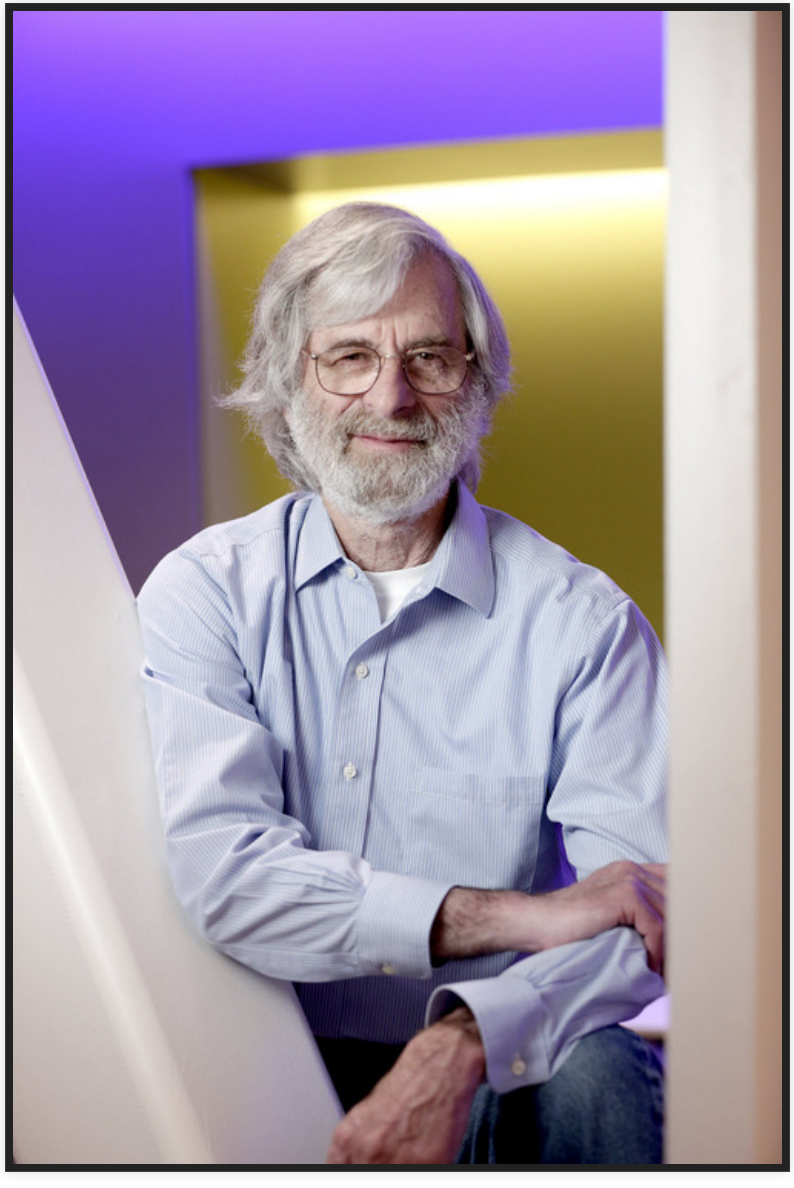
PCD2018 - 18

PROGRAMMAZIONE DISTRIBUITA

Teoria e tecniche per la gestione di più processi su macchine diverse che operano in modo coordinato allo svolgimento di un unico compito.

Un insieme di macchine che esegue un algoritmo distribuito è detto un sistema distribuito. In modo più preciso:



"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable" - Leslie Lamport, 2013 Turing Award

Abbiamo visto quanto sia complesso gestire più thread concorrenti all'interno di uno stesso processo. Perché andare verso difficoltà ancora maggiori e gestire processi coordinati su più macchine?

MOTIVAZIONI E CARATTERISTICHE

Le principali motivazioni che storicamente hanno portato ad ideare la distribuzione di un algoritmo su più nodi di calcolo sono:

AFFIDABILITÀ

Le attività possono proseguire sui nodi che sono (ancora) in linea, rispetto a quelli che sono fermi o in errore.

SUDDIVISIONE DEL CARICO

Una mole di lavoro più grande delle capacità di una sola macchina può essere suddivisa fra più nodi per essere eseguita in modo concorrente.

DISTRIBUZIONE

Gli utenti di più macchine possono accedere ai risultati del lavoro da uno qualsiasi dei nodi.

Le caratteristiche di un Algoritmo distribuito sono le seguenti:

CONCORRENZA DEI COMPONENTI

I vari nodi di esecuzione operano contemporaneamente.

MANCANZA DI UN *GLOBAL CLOCK*

L'ordine temporale degli eventi non è strettamente condiviso, e se importante deve essere imposto con opportuni mezzi.

FALLIMENTI INDIPENDENTI

I nodi possono guastarsi o fallire indipendentemente uno dall'altro.

I nodi comunicano fra loro scambiandosi messaggi, in quanto non condividono altra risorsa che il collegamento alla rete.

Le principali astrazioni disponibili riguardano quindi l'invio di un messaggio e l'attesa della ricezione di un messaggio.

MESSAGGI E METODI

Fin dai primi tempi si è cercato di riportare il modello a messaggi a qualcosa di più simile ad una chiamata locale mascherando la distinzione fisica fra le due macchine chiamante e chiamato.

Il termine RPC - *Remote Procedure Call* indica un sistema per rendere trasparente la localizzazione del codice chiamato, rendendolo il più possibile simile ad una chiamata locale.

L'adattamento locale/remoto viene implementato da un componente detto *stub*.

Una RPC comporta i seguenti passi:

- Il client chiama lo *stub* locale
- Lo *stub* mette i parametri in un messaggio (*marshalling*)
- Lo *stub* invia il messaggio al nodo remoto

- Sul nodo remoto, un *server stub* attende il messaggio
- Il *server stub* estrae i parametri (*unmarshalling*)
- Il *server stub* chiama la procedura locale

La risposta segue il percorso inverso.

Nei linguaggi Object-Oriented questo meccanismo prende il nome di RMI - *Remote Method Invocation* in quanto deve includere anche l'indirizzamento dell'oggetto destinatario della chiamata.

Il componente server viene chiamato *Object Broker*

All'inizio degli anni '90 diventa popolare una tecnologia di RMI detta **CORBA**: *Common Object Request Brocker Architecture* che propone uno standard.

CORBA descrive gli oggetti tramite un linguaggio: IDL - *Interface Definition Language* che gli consente di far interagire tecnologie differenti.

Java, nascendo come linguaggio per sistemi embedded, implementa fin dalle prime versioni un sistema di RMI e molto presto anche la specifica CORBA per poter partecipare a sistemi distribuiti costruiti con questa tecnologia.

Una chiamata RMI comporta i seguenti passi:

- Il client chiama lo *stub* locale
- Lo *stub* prepara i parametri in un messaggio
- Lo *stub* invia il messaggio. Il client è bloccato.
- Il server riceve il messaggio
- Il server controlla il messaggio e cerca l'oggetto chiamato
- Il server recupera i parametri e chiama il metodo destinatario

- L'oggetto esegue il metodo e ritorna il risultato
- Il server prepara il risultato in un messaggio
- Il server invia il messaggio allo *stub*
- Lo *stub* riceve il messaggio di ritorno
- Lo *stub* recupera il risultato dal messaggio
- Lo *stub* ritorna al client il risultato

Domanda:

Quante cose possono andare male in questo (lungo) processo?

- cosa succede se uno dei messaggi non viene recapitato?
- cosa succede se il client ed il server hanno versioni diverse degli oggetti scambiati?

- cosa succede se il client ed il server sono su reti che non permettono l'uno di indirizzare l'altro?
- cosa succede se durante la chiamata il client od il server falliscono o diventano non più disponibili?

- come è possibile nascondere tutta questa complessità e renderla indistinguibile da una chiamata locale?

Nelle reti e negli ambienti di esecuzione moderni tutte queste problematiche sono comuni:

- le reti possono (per es. wireless) possono essere inaffidabili
- può essere difficile per sistemi di decine o centinaia di nodi essere tutti aggiornati allo stessa versione del software

- firewall, reti temporanee e wireless rendono impossibile indirizzare liberamente un singolo terminale
- i nodi entrano ed escono dalla rete con grande facilità; inoltre, più sono numerosi più è facile che qualcuno di essi fallisca

L'evoluzione del panorama delle reti ha reso poco pratico l'uso di tecnologie RMI, che sopravvivono solo in ambiti controllati (server/server), ed in forme molto diverse da quelle originarie.

Non si cerca più di nascondere la complessità della chiamata remota, ma di rendere meno impegnativo possibile partecipare ad un servizio distribuito.

In Java 9, il modulo `java.corba` è stato ufficialmente deprecato, e rimosso in Java 11. Non è più presente nel classpath di default, e deve essere esplicitamente installato ed attivato.

SERIALIZZAZIONE

Un passo fondamentale evidenziato dai sistemi di RMI, ma che è necessario in generale, è la cosiddetta *serializzazione*, ovvero il metodo con cui un oggetto viene predisposto per la trasmissione in un messaggio.

Si dice *serializzare* usare un meccanismo di codifica che prende direttamente l'oggetto e lo traduce in messaggio per eseguire il passo di *marshalling*.

Al contrario, *deserializzare* corrisponde al passo di *unmarshalling*.

Mario Fusco

@mariofusco

Following

Serialization is the art of making sausages from a pig in a way that at some point you will be able to get the pig back from the sausages

Traduci il Tweet

23:39 - 23 ago 2018

Si tratta di un problema ingannevolmente semplice: in realtà è molto complesso e ricco di implicazioni, dalla storia, all'efficienza, alla sicurezza.

Java ha fin dalla versione 1.1 un meccanismo di serializzazione nativo, tramite l'interfaccia `java.io.Serializable`.

Tuttavia, la sua natura di "marker interface" e le cautele necessarie ad usarla fanno capire quanto il suo uso non sia semplice.

Le problematiche che la serializzazione deve affrontare sono:

- gestire il cambiamento strutturale delle classi
- serializzare grafi di oggetti

- indicare oggetti che non possono/non devono essere serializzati
- assicurare l'integrità e l'affidabilità dei dati serializzati
- rendere *marshalling*/*unmarshalling* efficienti in tempo e spazio

Per tutte queste motivazioni l'uso della serializzazione nativa di Java è sconsigliato nella pratica normale.

Non solo, è particolarmente sconsigliato per problematiche di sicurezza.

Inoltre, per il modo in cui funzionano reti e sistemi distribuiti oggi, sono diventati praticabili/preferibili in molte situazioni protocolli testuali trasportati da HTTP e umanamente leggibili.

I protocolli binari sono riservati ad ambienti controllati e dove ci sono particolari esigenze di efficienza.

Per questo motivo, non parleremo di serializzazione e useremo nei nostri esempi protocolli testuali semplici.

THE GOOD PARTS

Con queste premesse, quali sono le parti più importanti della libreria standard da usare per far comunicare fra loro nodi distribuiti?

Le classiche primitive del modello TCP/IP:

- Sockets (Connessioni TCP)
- Datagrams (Pacchetti UDP)

L'astrazione `Channel` per unificare le operazioni di I/O su canali differenti (file, rete, hardware).

Le implementazioni asincrone di `java.nio` per un'esecuzione più efficiente sfruttando a fondo le funzionalità fornite dal Sistema Operativo ospite.

La classe URL ed i suoi dintorni per fare (semplici) richieste HTTP.

Su queste basi, l'ecosistema Java ha a disposizione una grande scelta di robuste ed efficaci librerie per la realizzazione di topologie anche molto complesse.

OkHttp: semplice gestione di chiamate HTTP

Jackson: un/marshalling di dati JSON

Netty: I/O asincrono ad eventi

Thrift: RPC scalabile, efficiente, sicuro

Protobuf: RPC e serializzazione efficiente e sicura

Lo standard JEE introduce un framework per lo sviluppo di applicazioni client/server e web basato su astrazioni di livello più alto.

In generale, Java ha una posizione di primo piano nell'attuale mercato dello sviluppo di applicazioni di rete e web, con un ricchissimo panorama di soluzioni disponibili.

Un interessante punto di partenza per esplorare questo panorama sono i benchmark TechEnpower:

https://www.techempower.com/benchmarks/