

Sistemi distribuiti

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

Indice

1	Introduzione	2
2	Lezione 1	3
2.0.1	Il modello Client-Server	4

Capitolo 1

Introduzione

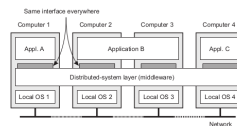
Questi appunti sono presi a le lezioni. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: <https://github.com/dlccgold/Appunti>.

Grazie mille e buono studio!

Capitolo 2

Lezione 1

Un sistema distribuito è un sistema nel quale componenti hardware e software, collocati in computer connessi alla rete, comunicano e coordinano le loro azioni solitamente col passaggio di messaggi (a differenza delle chiamate di procedura che si hanno col passaggio di parametri su memoria condivisa). Ogni processo ha quindi una parte di logica applicativa e una parte di coordinamento. Altrimenti si ha questa definizione. un sistema distribuito è un insieme di elementi autonomi di computazione che si interfacciano agli utenti come un singolo sistema "coerente".



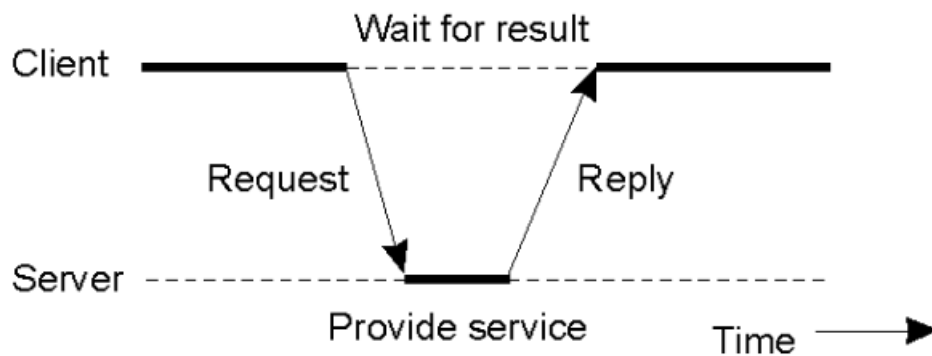
I sistemi distribuiti sono quindi sistemi complessi. Si hanno le seguenti caratteristiche:

- le unità autonome di computazione si chiamano **nodi** e possono essere device hardware o singoli processi software
- ogni nodo "fa quello che vuole", ogni nodo è autonomo, e vanno tra loro sincronizzati e coordinati (programmazione concorrente). Ogni nodo ha la sua "nozione di tempo"
- utenti e applicazioni vedono un singolo sistema
- si possono aprire e chiudere gruppi di nodi

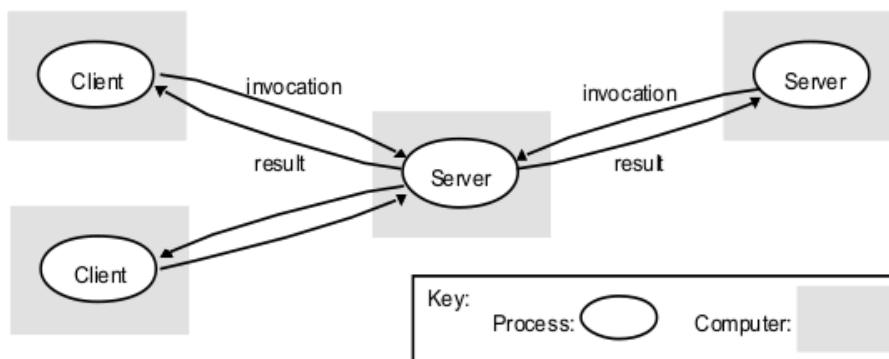
La parola chiave è **trasparenza di distribuzione (distribution transparency)**. Trasparenza significa nascondere dettagli agli utenti che possono ignorare ciò che succede e che non possono modificare il servizio. Si ha che

il sistema non va in errore se un solo nodo va in errore in quanto i nodi sono indipendenti ma è difficile occultare gli errori parziali dei singoli nodi ed è difficile sistemare gli eventuali errori del singolo nodo. Ovviamente non si ha memoria condivisa e non c'è uno stato globale. In un sistema distribuito non si ha un clock globale e non si può controllare globalmente o avere uno scheduling globale.

2.0.1 Il modello Client-Server



Si ha che un client fa una richiesta e il server risponde con un certo risultato (con il conseguente ritardo, a differenza del modello a chiamata di procedura).



Si può accedere a server multipli (cluster con anche bilanciamento del carico) e si può accedere via proxy (dei server "finti" che fungono da concentratori). Un sistema distribuito ha 4 problemi da fronteggiare:

1. **identificare la controparte**, che si risolve assegnando un nome, è la procedura di **naming**

2. **accedere alla controparte**, che si risolve con una reference, un **access point**
3. **comunicare (parte 1)**, che si risolve accettando e condividendo un formato, un **protocollo**, "**protocol**"
4. **comunicare (parte 2)**, che si risolve concordando *sintassi e semantica* per l'informazione da condividere (**quest'ultimo è però ancora un problema aperto**)

Si hanno le seguenti definizioni per quanto riguarda la trasparenza:

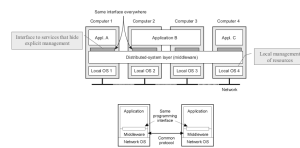
- **naming**, usare nomi simbolici per identificare le risorse che sono parte del sistema distribuito
- **access transparency**, nascondere le differenze nella rappresentazione delle informazioni e nell'accedere ad un'informazione locale o remota
- **location transparency**, nascondere dove è collocata una risorsa sulla rete
- **relocation or mobility transparency**, nascondere che una risorsa può essere stata trasferita ad un'altra locazione mentre è in uso
- **migration transparency**, nascondere che una risorsa può essere trasferita
- **replication transparency**, nascondere che una risorsa può essere replicata
- **concurrency transparency**, nascondere che una risorsa può essere condivisa da molti utenti indipendenti
- **failure transparency**, nascondere fallimenti e recovery di una risorsa
- **persistence transparency**, nascondere se una risorsa è volatile o memorizzata permanentemente

non si possono però nascondere:

- **ritardi e latenze di comunicazione**
- **nascondere completamente i failure della rete e dei nodi**, non puoi neanche distinguere bene rallentamenti e errori. Ovviamente non puoi sapere se sta per accadere un **crash**

Una trasparenza completa, oltre ad essere quasi impossibile a livello teorico, è anche estremamente "cara" a livello di performances e tempistiche (causa scrittura costante su dischi e mantenimento delle repliche).

Nascondere le informazioni è alla base dell'ingegneria del software. Bisogna separare il *cosa* si fa e il *come* lo si fa. Il *cosa* si fa mediante la definizione dell'interfaccia, *Interface Definition Languages (IDL)*, e il *come* mediante l'implementazione delle classi e dei metodi. Le interfacce sono definite mediante principi standard, sono complete e sono neutrali (indipendenti dall'implementazione).



Tra i vari componenti si ha:

- **indipendenza logica**, con i vari componenti che lavorano autonomamente
- **composizione**, con la collaborazione dei vari processi

Si separano:

- **meccanismi**, ciò che è fatto dai componenti (esempio il context switch)
- **politiche**, come vengono applicati le varie funzionalità del sistema (esempio lo scheduling Round Robin RR)

Bisogna separare e bilanciare politiche e meccanismi



Ricapitolando il concetto di protocollo:

- per poter capire le richieste e formulare le risposte i due processi devono concordare un protocollo
- i protocolli (come *HTTP*, *FTP* e *SMTP*) definiscono il formato, l'ordine di invio e di ricezione dei messaggi tra i dispositivi, il tipo dei dati e le azioni da eseguire quando si riceve un messaggio
- le applicazioni su TCP/IP:

- si scambiano stream di byte di lunghezza infinita (il meccanismo)
- che possono essere segmentati in messaggi (la politica) definiti da un protocollo condiviso

Vediamo un esempio di codice. Partiamo dall'*header.h*

```
// definizioni necessarie a client e server

#define TRUE 1
#define MAX_PATH 255 // lunghezza massima del nome di un file
#define BUF_SIZE 1024 // massima grandezza file trasferibili per volta
#define FILE_SERVER 243 // indirizzo di rete del file del server

// operazioni permesse

#define CREATE 1 // crea un nuovo file
#define READ 2 // legge il contenuto di un file e lo restituisce
#define WRITE 3 // scrive su un file
#define DELETE 4 // cancella un file

// errori

#define OK 0 // nessun errore
#define E_BAD_OPCODE -1 // operazione sconosciuta
#define E_BAD_PARAM -2 // errore in un parametro
#define E_IO -3 // errore del disco o errore di I/O

// definizione del messaggio

struct message{
    long source; // identità del mittente
    long dest; // identità del ricevente
    long opcode; // operazione richiesta
    long count; // numero di byte da trasferire
    long offset; // posizione sul file da cui far partire l'I/O
    long result; // risultato dell'operazione
    char name[MAX_PATH]; // nome del file
    char data[BUF_SIZE]; //informazione da leggere o scrivere
};
```


vediamo la struttura di un semplice server che realizza un semplice file server remoto:

```
#include <header.h>
void main(void){
    struct message m1, m2; // messaggio in entrata e uscita
    int r; // risultato

    while(TRUE){ // il server è sempre in esecuzione
        receive(FILE_SERVER, &m1); // stato di wait in attesa di m1
        switch(m1.code){ // vari casi in base alla richiesta
            case CREATE:
                r = do_create(&m1, &m2);
                break;
            case CREATE:
                r = do_read(&m1, &m2);
                break;
            case CREATE:
                r = do_write(&m1, &m2);
                break;
            case CREATE:
                r = do_delete(&m1, &m2);
                break;
            default:
                r = E_BAD_OPCODE;
        }

        m2.result = r; // ritorna il risultato al client
        send(m1.source, &m2); // manda la risposta
    }
}
```

vediamo ora un client che usa il servizio per trasferire un file:

```
#include <header.h>

int copy(char *src, char *dst){ // copia file usando il server
    struct message m1; // buffer del messaggio
    long position; // attuale posizione del file
    long client = 110; // indirizzo del client

    initialize(); // prepara l'esecuzione
    position = 0;
    do{
        m1.opcode = READ; // operazione settata su READ
        m1.offset = position; // scelta la posizione nel file
        m1.count = BUF_SIZE; // byte da leggere
        strcpy(&m1.name, src); // nome file copiato in m1
        send(FILESERVER, &m1); // manda il messaggio al file server
        receive(client, &m1); // aspetta la risposta

        // scrive quanto ricevuto su un file di destinazione
        m1.opcode = WRITE; // operazione settata su WRITE
        m1.offset = position; // scelta la posizione nel file
        m1.count = BUF_SIZE; // byte da leggere
        strcpy(&m1.name, dst); // nome del file sul buffer
        send(FILESERVER, &m1); // manda il messaggio al file server
        receive(client, &m1); // aspetta la risposta
        position += m1.result // il risultato sono i byte scritti
    }while(m1.result > 0); // itera fino alla fine
    return(m1.result >= 0 ? OK : m1.result); // ritorna OK o l'errore
}
```