Facoltà di Ingegneria - Università di Bologna - Stud. **Maurizio De Tommasi**

Sistema di prenotazione posto in sala prove musicale

Obiettivo

Il progetto mira a realizzare un sistema di prenotazione posti per una sala prove musicale e si basa sul modello Client-Server.

Autore

Maurizio De Tommasi

Ingegneria Informatica, n° matricola 2148 056705

Versione

2.0del 11/07/2001

~~ANALISI~~

L'idea di realizzare un sistema di prenotazione per una sala prove nasce da un problema realmente esistente per il quale mi è stato proposto di trovare una soluzione adeguata.
Per la realizzazione di tale sistema è previsto l'utilizzo di due calcolatori che svolgono la funzione di server. Il lato client è usato dagli utenti (eventualmente già registrati) che intendono verificare la disponibilità ed eventualmente l'occupazione di un posto nella sala.
La sala prove è disponibile in qualsiasi momento della giornata per tutta la settimana. In particolare le fasce orarie in cui poterla utilizzare sono quattro: MATTINA, POMERIGGIO, SERA, NOTTE.

Le ipotesi di lavoro sono:

1. Guasto singolo per i server.
2. Bassa sicurezza.
3. Traffico contenuto.
4. Richieste singole da parte dei client.
5. Assenza di guasti sulla linea di comunicazione tra i due server che impedirebbe la loro coordinazione.
6. Uno dei due server ha minor probabilità di crash rispetto all'altro.

Nell'ipotesi 1, la replicazione dei server (entrambi attivi ma su nodi differenti) garantisce affidabilità e continuità del servizio e recovery in caso di guasto.

~~PROGETTO~~

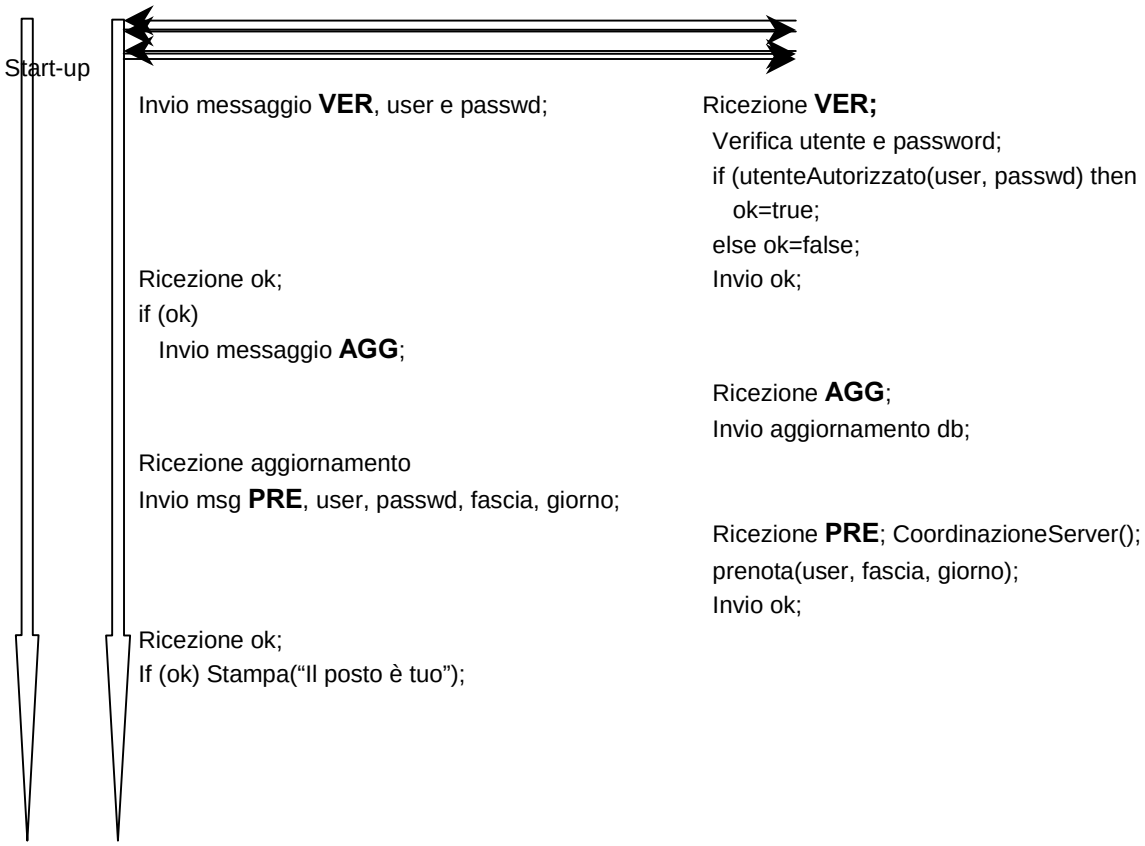
Data l'ipotesi 6, si era pensato inizialmente di strutturare il sistema in modo da avere un unico server master attivo che esegue le azioni sui dati e l'altro slave passivo utile solo in caso di guasto del server primario e che ottiene l'aggiornamento da questo a intervalli prefissati (checkpointing time-driven).Successivamente, per garantire *fairness* e *distribuzione del carico* sui due server, la soluzione adottata è stata un'altra: la disponibilità dei posti è divisa a metà fra i due server. In particolare un server è master per la disponibilità nelle fasce MATTINA e POMERIGGIO e slave nelle altre due fasce orarie, analogamente l'altro server è primario per le prenotazioni riguardanti la SERA e la NOTTE e secondario per le altre.
E' ragionevole pensare che le prenotazioni saranno più numerose nelle fasce di mattina e pomeriggio, per cui il carico sarà leggermente sbilanciato sul server master in quelle fasce orarie. Questa ipotesi tuttavia sarebbe perfettamente compatibile con l'ipotesi 6 se il master delle fasce più affollate fosse il server con minor probabilità di crash (d'ora in poi verrà nominato server1).

- ***Client***

Il client conosce entrambi i server e la prima operazione che compie è chiedere al server1 (se questo è attivo altrimenti al server2) l'aggiornamento del database locale e contemporaneamente la verifica dell'utente e della password.
Il database in questa versione può essere realizzato con un semplice file in cui si memorizza una matrice 4x7 (4 fasce e 7 giorni) di interi (0=posto libero, 1=posto occupato).
Dopo aver consultato la disponibilità dei posti, il client può inviare la richiesta di prenotazione al server. Ovviamente la richiesta non viene inoltrata qualora il posto sia trovato occupato già nella copia locale del database. Il client tenta di stabilire una connessione con il server master (Server1 per richieste di MATTINA e POMERIGGIO, Server2 altrimenti) e nel caso in cui non dovesse riuscirci tenta di connettersi con l'altro server che funge da master per tutte le fasce orarie.
Il *protocollo di comunicazione* col server prevede l'invio da parte del client di quattro tipi di messaggi:

- **AGG**: aggiornamento del database locale;
- **PRE**: richiesta di prenotazione di un posto;
- **REC**: richiesta di verifica di un posto (previsto solo in caso di recovery del client);
- **VER**: richiesta di verifica dell' utente e della password;

Ipotizzando una situazione di perfetto funzionamento del sistema, lo scambio di messaggi tra client e server avviene nel seguente modo:
CLIENT SERVER



La richiesta di prenotazione avviene dunque sulla base della copia locale del db che potrebbe quindi essere obsoleta. Ad esempio supponiamo che *client1* chieda l'aggiornamento del database al tempo t_0 e attenda qualche istante prima di effettuare la richiesta di prenotazione. Nel frattempo *client2* richiede anch'esso l'aggiornamento del proprio db al tempo t_1 ed effettua la prenotazione al tempo t_2 con successo. A questo punto *client1* possiede una copia non aggiornata del database e potrebbe richiedere l'occupazione dello stesso posto prenotato da *client2*. Perciò qualora il posto venga trovato occupato sul server il client ottiene nuovamente l'aggiornamento del db. Dal momento che il traffico non è elevato, si può ritenere che una situazione di incoerenza tra il db del client e quello del server sia comunque estremamente rara.
Tale situazione può essere descritta dal seguente grafico:

T_0 *client1* aggiorna db
T_1 *client2* aggiorna db
T_2 *client2* prenota per LUNEDI MATTINA (copia aggiornata)
T_3 *client2* ottiene conferma di successo
T_4 *client1* prenota per LUNEDI MATTINA (copia obsoleta)
T_5 *client1* trova occupato ma ottiene aggiornamento

La scelta di permettere una tale situazione di incoerenza, giustificata dall'ipotesi di basso traffico, è dettata anche da esigenze di semplicità della realizzazione. Una possibile estensione del sistema potrebbe prevedere l'invio da parte del server di continui aggiornamenti al client con intervalli regolari fino alla richiesta di quest'ultimo (che quindi avrebbe sempre una copia aggiornata, modello push con checkpointing time-driven). A mio avviso, tale situazione appesantirebbe troppo il sistema (soprattutto il server).

Dall'aggiornamento alla prenotazione potrebbe trascorrere molto tempo per cui il client chiude la connessione al termine dell' aggiornamento e la riapre per la richiesta di prenotazione. Alla fine della prenotazione il client deve essere disabilitato ad effettuare altre prenotazioni previo nuovo controllo di user e password.

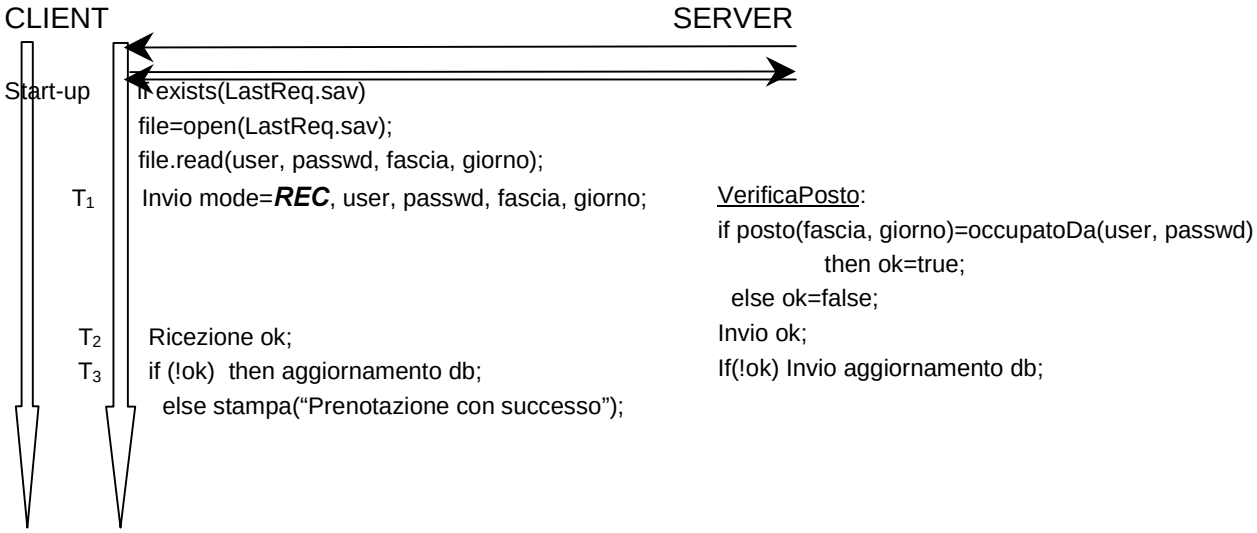
In caso di caduta del client

All'inizio della richiesta il client crea un file (LastReq.sav) in cui vengono salvate le informazioni riguardanti la prenotazione (utente, fascia oraria e giorno). Tale file viene distrutto soltanto quando la richiesta è stata soddisfatta dal server. In caso di caduta del client (che quindi mantiene lo stato della comunicazione) non è possibile sapere se l'operazione di prenotazione è andata a buon fine o meno sul server. Perciò al riavvio il client verifica la presenza del file, la cui esistenza implica appunto un precedente fallimento del client, e in caso affermativo recupera le informazioni necessarie da tale file e prova a verificare se la prenotazione precedente ha avuto successo sul server mandando il messaggio **REC**overy.

Protocollo per il recovery:

supponiamo che il client cada dopo aver inoltrato al server la richiesta e prima di aver ottenuto conferma.

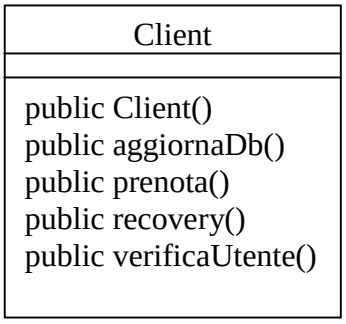
Al riavvio del client la situazione è la seguente:



La caduta del Client prima dell'aggiornamento e verifica utente iniziale o dopo la conferma del server di prenotazione con successo non provoca alcuna situazione critica.

Nel caso in cui il Client non riesca a stabilire una connessione con i Server si visualizza all'utente un messaggio di errore e di riprovare l'operazione in un secondo momento.

La struttura del Client può essere descritta da una classe UML nel modo seguente:



Ogni metodo che il client mette a disposizione crea un Thread che effettua le relative operazioni.

- ***Server***

Per aumentare l'efficienza del sistema si può pensare di realizzare un server multi-threaded parallelo nella gestione delle richieste dei client.
All'avvio ogni server crea due Thread, uno per l'ascolto dei client e uno per la coordinazione con l'altro server. I due Thread non fanno altro che realizzare i seguenti protocolli:
Thread di ascolto dei client:

```
while(true)
begin
  accept();
  <nuovo Thread gestisci richiesta client>
end.
```

Thread di ascolto dell'altro server:

```
while(true)
begin
  accept();
  <nuovo Trhead coordinazione con altro server>
end.
```

La gestione della richiesta del client e la coordinazione tra i due server sono demandate ad altri processi leggeri.

Gestione richiesta del client

Il server (master per le fasce che gli competono) risponde alle richieste dei client per quelle fasce orarie secondo il modello di messaggi descritto in precedenza per il client. Nel caso in cui l'altro server sia down il server attivo è master anche per le altre fasce orarie.

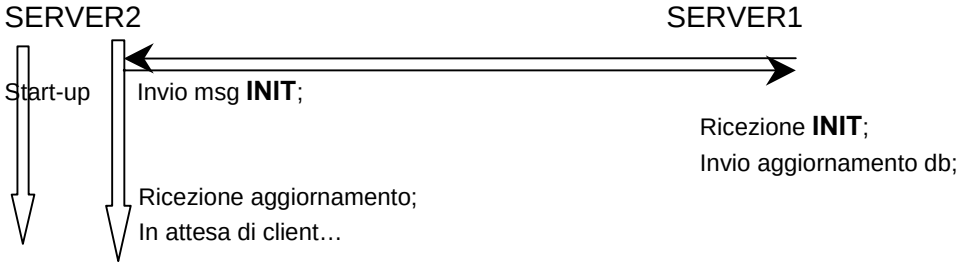
Protocollo di coordinazione tra i server

Il protocollo prevede quattro tipi di messaggi tra i server:

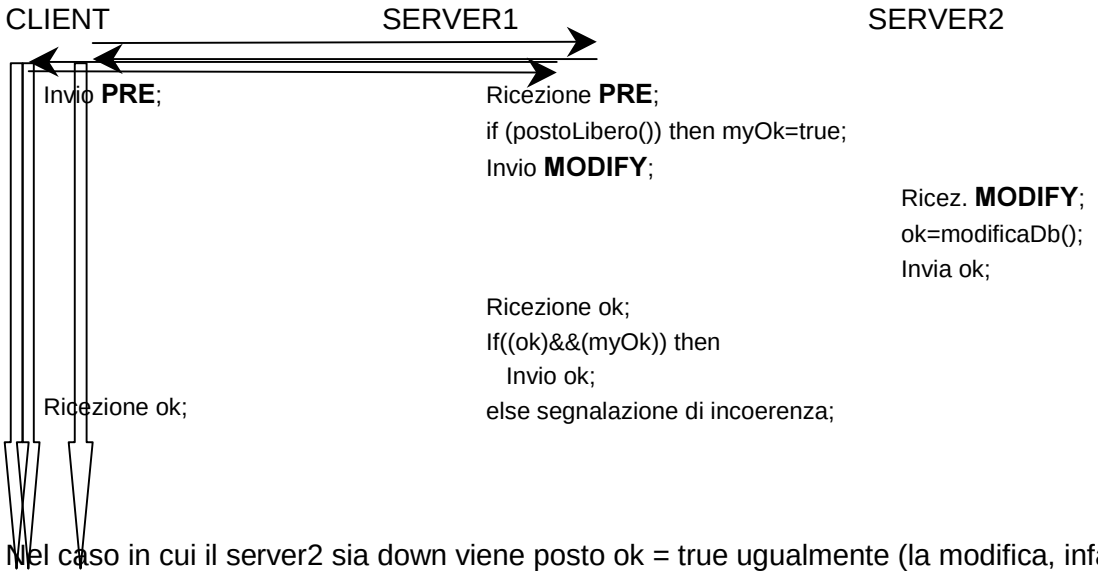
- **INIT**: il server comunica la propria attivazione all'altro;
- **MODIFY**: il server comunica all'altro di modificare il db;
- **SAVE**: il server chiede il salvataggio su disco all'altro server;
- **RESET**: il server chiede l'azzeramento del db all'altro server.

Allo start-up il server richiede all'altro il download del database(messaggio INIT) per poter essere in grado di servire i propri clienti con una copia aggiornata dei dati. Solo nel caso in cui l'altro server non risponde (situazione che non dovrebbe mai verificarsi a regime per l'ipotesi 1) il server tenta di aggiornare il database da file.
Quando il server master soddisfa la richiesta di un client modificando il database invia un messaggio (il messaggio MODIFY) e i dati modificati (user, fascia oraria e giorno in cui è stata effettuata la prenotazione) all'altro server richiedendogli di aggiornare il database (*modello PUSH*). Ciò avviene anche quando su uno dei due server viene effettuata l'operazione di *reset* dei dati

(azzeramento della matrice dei posti a fine settimana) e l'operazione di salvataggio su disco. In questo modo viene garantita la consistenza dei dati duplicati.
Più in dettaglio, supponiamo che il server1 sia già attivo e venga attivato il server2. Il server2 invia il messaggio INIT al server1 che risponde inviando l'aggiornamento del database:

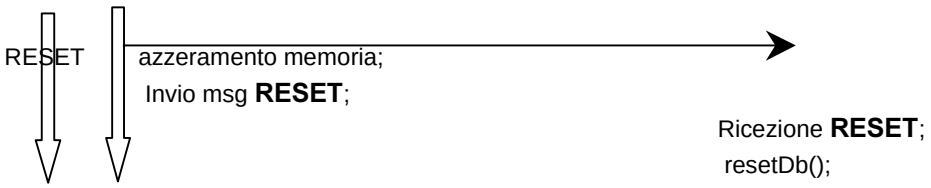


Quando poi un server riceve il messaggio di PRE (richiesta di prenotazione) da un client, se il posto è già occupato viene inviato l'aggiornamento al client altrimenti, prima di fornire a questo una risposta, il server deve aggiornare l'altro server attraverso l'invio del messaggio MODIFY.
La situazione successiva al messaggio di PRE del client è la seguente (supponiamo che il posto sia libero):



Nel caso in cui il server2 sia down viene posto ok = true ugualmente (la modifica, infatti, avverrà all'avvio del server2 con INIT).
Operazioni critiche sono il reset e il salvataggio su disco da parte del server. In questo caso il server sul quale viene chiesto il salvataggio dei dati o il reset deve forzare il salvataggio anche sull'altro server per non avere copie inconsistenti su disco.
Per questo scopo è predisposto un Thread che tenta di notificare l'operazione all'altro server attraverso i messaggi RESET e SAVE. La gestione dei due messaggi è identica, quindi verrà descritto il comportamento del sistema nel caso di reset.

SERVER1 SERVER2

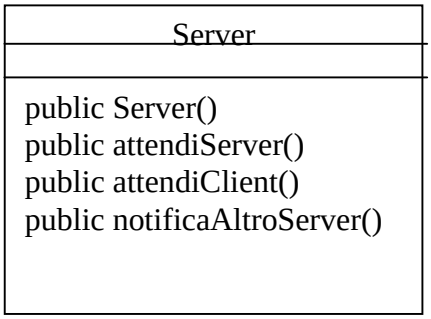


Si noti che il protocollo per il reset e per il salvataggio non prevedono un acknowledge da parte dell'altro server, cioè la comunicazione è a senso unico. Se il server2 in quel momento è down non ci sono problemi (il server2 aggiornerà all'avvio). Ma nel caso in cui il server2 è attivo il server1 non saprà mai se l'operazione ha avuto successo sull'altro server. Teoricamente dunque si potrebbe verificare una grave situazione di incoerenza nel caso del reset fallito sul server2. Questa scelta è giustificata dal fatto che, qualunque sia l'esito dell'operazione sul server2, il server1 è comunque tenuto a portare a termine la sua operazione (le operazioni di reset e salvataggio sono infatti richieste sul server dall'esterno e non possono essere rimandate). Si potrebbe pensare ad un miglioramento della soluzione ad esempio predisponendo il server1 a ricevere ok dal server2 (comunicazione nei due sensi). In caso di insuccesso (ok=false) il server1 potrebbe effettuare ugualmente l'operazione ignorando il fallimento dell'altro server e ripetere la richiesta per un certo numero di volte ad intervalli di tempo regolari. Nel caso in cui il server2 sia down allora ok viene posto ugualmente true.
Si noti che un server suppone che l'altro server sia down solo se non riesce ad effettuare la connessione con quest'ultimo (ipotesi 5).

In caso di caduta di un server

Nel caso in cui uno dei due server cada, l'altro diventa master anche per le fasce orarie per le quali era slave. Tutte le modifiche sul database locale non avranno ovviamente effetto sul database remoto, ma al riavvio del server precedentemente caduto la procedura di inizializzazione aggiornerà il proprio database.
Il sistema non è in grado di far fronte a recovery di guasti multipli.

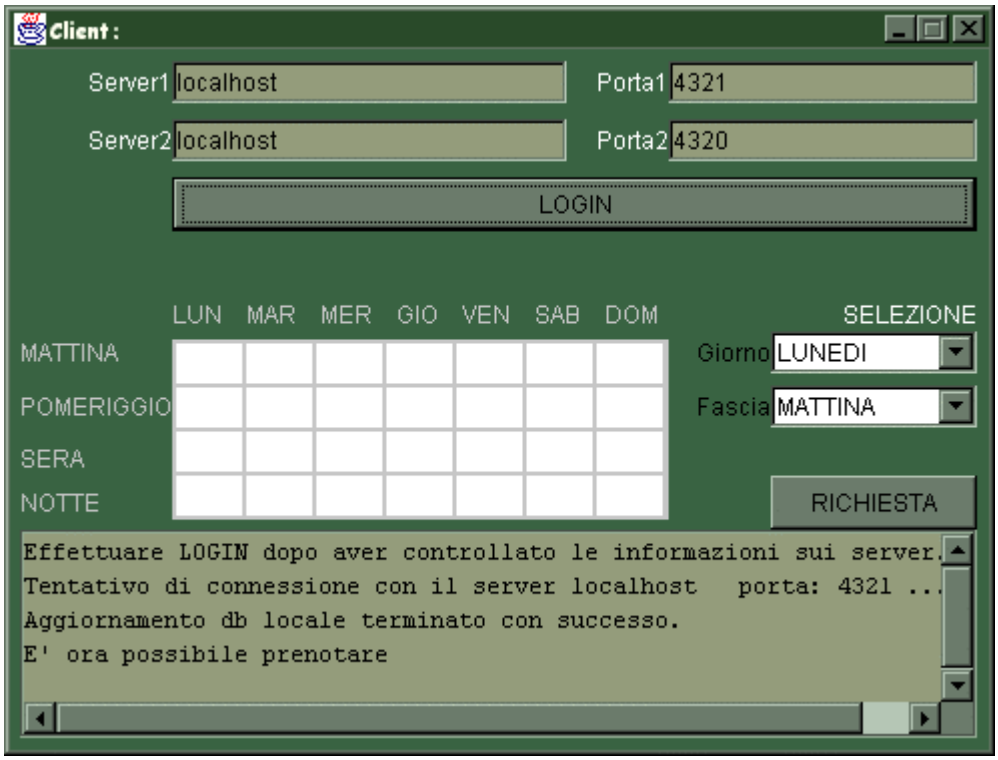
La struttura del Server può essere descritta nel modo seguente:



In cui *attendiServer* e *attendiClient* attivano i Thread di ascolto rispettivamente per l'altro server e per i client e *notificaAltroServer* forza il salvataggio su disco e il reset sull'altro server secondo i protocolli descritti in precedenza.

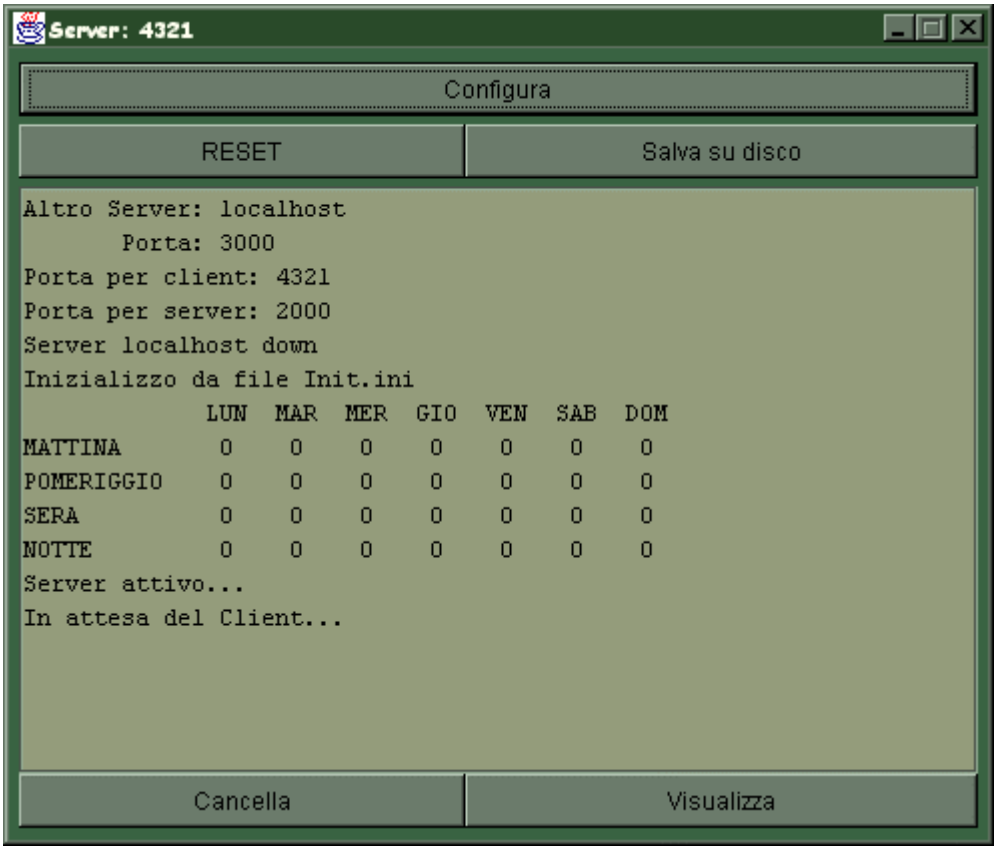
IMPLEMENTAZIONE

La tecnologia con cui si vuole realizzare il sistema è Java.
Le interfacce client e server sono classi estensioni della classe Jframe e implementano l'interfaccia ActionListener (sono le classi CliIODevice.java e ServIODevice.java). Non mi soffermerò ulteriormente sulla descrizione del loro funzionamento. L'interfaccia grafica del lato client è la seguente:



All'avvio dell'applicazione si chiedono user e password, si chiede di settare i parametri necessari per la comunicazione con i server e quindi si può proseguire con il controllo di user e password (tasto LOGIN). Se il controllo ha avuto successo si aggiorna il db (griglia al centro dell'interfaccia). Il tasto RICHIESTA inoltra la richiesta di prenotazione.
Per quanto riguarda il lato server, all'avvio viene chiesto di configurare i parametri della comunicazione con i client e con l'altro server:

L'interfaccia del server appare come segue:



Il tasto *Cancella* effettua l'operazione jTextArea.setText(""); mentre il tasto *Visualizza* visualizza la matrice (0=libero, 1=occupato). Il tasto *Reset* azzera la matrice (pone tutti gli elementi a 0) e il tasto *Salva su disco* permette il salvataggio secondo le modalità descritte in precedenza.
Le classi principali sono la classe *Client.java* e la classe *Server.java* descritte in fase di progetto. I metodi della classe Client.java sono implementati come segue:

```
public void aggiornaDb(){

    RichiestaClient aggThr = new RichiestaClient(CliDev, "AGG");

    aggThr.start();

  }


public void prenota(){

   RichiestaClient prenotaThr = new RichiestaClient(CliDev, "PRE");

   prenotaThr.start();

}


public void recovery(){

   RichiestaClient recoveryThr = new RichiestaClient(CliDev, "REC");

   RecoveryThr.start();

}
```

```
public void verificaUtente(){
   RichiestaClient verifThr = new RichiestaClient(CliDev, "VER");
   verifThr.start();
}
```
in cui *RichiestaClient* è una classe che estende la classe Thread ed esegue le operazioni del client. Il metodo run() è implementato come segue:
```
public void run(){
    if (mode.equals("AGG")) aggiornaDb();
     else if (mode.equals("PRE")) prenota();
       else if (mode.equals("REC")) recovery();
            else if (mode.equals("VER")) verificaUtente();
  }
```
Per quanto riguarda il server, la classe *Server.java* contiene i seguenti metodi:
```
public void attendiServer(){
     InAscolto attesaServerThr = new InAscolto(servDev, portaServer,"SER");
     attesaServerThr.start();
   }

   public void attendiClient(){
     InAscolto attesaClientThr = new InAscolto(servDev, portaClient, "CLI");
     attesaClientThr.start();
   }

   public void notificaAltroServer(String mode){
          Notifica notificaThr = new Notifica(servDev, mode);
          notificaThr.start();
   }
```
I primi due metodi attivano un thread (classe *InAscolto.java*) che si mette in ascolto dei client o del server e il metodo run() è implementato come segue:
```
public void run(){
    Socket newSoc=null;
    while (true) {
         if (ascolto==null) return;
         try{
              newSoc=ascolto.accept();
```

```
        }catch(IOException e){
                System.err.println("Impossibile connettermi al socket client: "+e.toString());
                System.exit(1);
        }
         if (mode.equals("SER")){
                    CoordinaServer coordServThr = new CoordinaServer(servDev, newSoc);
                    coordServThr.start();
         }else if (mode.equals("CLI")){
                     ServiClient servCliThr = new ServiClient(servDev, newSoc);
                    servCliThr.start();
         }
      }//fine while
   }//fine run
```

in cui *CoordinaServer* è il thread per il coordinamento tra i due server mentre *ServiClient* è il thread per la gestione della richiesta del client.
La classe *Server.java* contiene inoltre un metodo *notificaAltroServer*() per la notifica all'altro server del RESET e del SALVATAGGIO SU DISCO.
Una nota implementativa importante riguarda il controllo di user e password. L'utilizzo delle password nel progetto è del tutto esemplificativo perciò il corpo dei metodi per la verifica di user e password non fa altro che abilitare qualsiasi utente senza effettuare alcun controllo. In sostanza l'applicazione è accessibile da chiunque, nonostante l'applicazione chieda che vengano immessi user e password.

CODIFICA

I file .java relativi al client sono i seguenti:

CliIODevice.java
(interfaccia grafica client)

Client.java
(classe principale)

RichiestaClient.java
(thread per l'invio della richiesta)

finPass.java
(JDialog per l'immissione di user e password)

Selezione.java
(classe della selezione giorno e fascia)

Utente.java
(memorizza user e password dell'utente)

remoteAddr.java

(memorizza indirizzi remoti dei server)

creaGriglia.java
(applet per la creazione dell'immagine griglia)

I file relativi al lato server sono i seguenti:

ServIODevice.java
(interfaccia grafica server)

Server.java
(classe principale)

InAscolto.java
(Thread d'ascolto per i client e per l'altro server)

ServiClient.java
(Thread per la gestione delle richieste dei client)

CoordinaServer.java
(Thread per il coordinamento tra i due server)

Notifica.java
(Thread per la notifica all'altro server del reset e del salvataggio su disco)

Matrice.java
(classe per memorizzare la matrice di 0 e 1 riguardante l'occupazione dei posti)

finConfigServer.java
(JDialog per la configurazione dei parametri della comunicazione)

FinReset.java
(JDialog per la conferma del reset)

FinSave.java
(JDialog per la conferma del salvataggio su disco)

TESTING

La fase di test prevede alcune prove con un numero di client collegati sempre maggiore. Il test è effettuato in due contesti diversi:

- Test in locale;
- Test in rete;

TEST IN LOCALE

Per quanto riguarda il test in locale sono state effettuate prove su una macchina Windows98 e su una macchina Windows NT. In entrambi i casi il comportamento è stato simile e soprattutto non si è verificato nessun peggioramento delle prestazioni all'aumentare del numero di client. Infatti il sistema è progettato in modo tale che il numero di connessioni attive in un istante sono limitate (il client apre una connessione e la chiede immediatamente dopo la risposta del client). E' perciò impossibile verificare il parallelismo del server (nel caso di Windows NT) in quanto in locale non è possibile inoltrare contemporaneamente più richieste da più client.
In conclusione all'aumentare del numero dei client il sistema rimane efficiente.

Suppongo che, soprattutto nel caso di Windows98, i server non riuscirebbero a gestire contemporaneamente un numero elevato di connessioni con conseguente caduta di alcune socket, ma non è stato possibile rilevare tale anomalia.

TEST IN RETE

Il sistema operativo installato sulle singole macchine della rete è Windows NT. Il comportamento del sistema in questo caso è del tutto analogo al caso precedente

E' stato verificato inoltre il comportamento del sistema in caso di fallimento di uno dei due server. Il peggioramento riscontrato (inevitabile) è comunque da ritenersi trascurabile (maggiore attesa solo nelle operazioni riguardanti il server down in quel momento).