Patrick Fogli

Vendita online di abbonamenti per lo stadio

Applicazione realizzata in 

Premessa

Aspetti e problematiche del progetto

Il sistema realizza la prenotazione e la vendita degli abbonamenti per lo stadio.

La vendita avviene tramite Internet gestita da postazioni che possono essere utilizzate da operatori a un botteghino o da utenti singoli.

E' possibile scegliere un preciso settore dello stadio, ottenendo quindi una mappa del settore stesso che visualizza i posti disponibili. Una volta effettuata la scelta la transazione viene gestita e si procederà al pagamento tramite carta di credito o a incasso diretto al botteghino. L'incasso può essere effettuato anche da un'apposito sportello presente nel luogo e che consegna il tagliando dopo aver ricevuto l'importo da corrispondere.

Ci si aspetta di ottenere un sistema affidabile che gestisca ogni fase della transazione, dalla scelta del posto alla vendita vera e propria, comprensiva della transazione bancaria necessaria a fronte di un pagamento via credit card.

Viene utilizzato un modello client-server responsabile dell'affidabilità. In relazione a ciò sono effettuate delle ipotesi di lavoro su cui si basa lo sviluppo.

E' chiaramente necessaria la presenza di un supporto fisico di memorizzazione delle vendite, utile al server per il proprio lavoro e all'azienda che gestisce la vendita per avere notizia delle transazioni effettuate ed effettuabili.

Eventuali miglioramenti possibili sulla base del progetto realizzato e di tecnologie più avanzate utilizzabili, verranno trattati alla fine di questa relazione nella sezione *Miglioramenti*.

POSTAZIONE DI VENDITA UTENTE (client)

E' l'applicazione che sta a contatto diretto con il pubblico, utilizzabile sia dall'utente finale sia dall'operatore al botteghino.

Deve:

- Visualizzare in forma grafica e quanto più possibile efficace ed intuitiva la disposizione generale dei settori dello stadio.
- Permettere in maniera semplice e veloce la scelta del posto nel settore prescelto
- Verificare che la scelta possa essere fatta soltanto a fronte di un posto effettivamente disponibile
- Gestire la transazione con il server dall'inizio al prodotto finale
- Rendere il più possibile affidabile il servizio
- Nel caso di accesso tramite operatore, l'operatore stesso può:

- Modificare i parametri di accesso al server in caso di variazione di indirizzo
- Scegliere a quale server collegarsi in caso di più postazioni disponibili (il progetto, per semplicità, ne prevede due)
- L'interfaccia che gestisce l'applicazione é grafica e gestita tramite bottoni di facile uso e comprensione.
- L'accesso all'applicazione avviene tramite password e user, sta quindi al sistema che gestisce l'accesso locale stabilire se si tratta di operatore o utente esterno e assegnare i relativi diritti

GESTIONE DELLA VENDITA (server)

E' l'applicazione remota con cui il client si collega per acquistare il biglietto.
Deve:

- Garantire l'accesso solo a utenti autorizzati
- Gestire l'accesso al database dei posti garantendo **affidabilità, consistenza, concorrenza, prevenzione e recupero dai guasti**
- Gestire i collegamenti con gli altri server in maniera affidabile ed efficace garantendo la consistenza dei dati anche su server differenti
- Permettere all'utilizzatore del server di conoscere ed eventualmente manipolare i parametri relativi a particolari caratteristiche dello stadio e a comunicazioni tra server.

IPOTESI DI LAVORO

- Per garantire la continuità del servizio è necessaria la presenza di risorse replicate. Il progetto viene svolto con *due server equipotenti* e la scelta di collegamento viene effettuata dall'applicazione client. Per entrambi i server è prevista una modalità master(in cui gestiscono la transazione con il client) e una slave (in cui ricevono le informazioni sulla transazione svolta dall'omologo). Eventuali estensioni verranno trattate nella sezione *Miglioramenti* al termine di questa relazione.
- Ipotesi di *guasto singolo* che garantisce almeno la presenza di un server attivo. Tutto ciò per mantenere la consistenza del database e garantire comunque la continuità del servizio.
- Un server conosce l'indirizzo e la porta del suo omologo. L'applicazione server conosce inoltre la struttura dello stadio.
- Il cliente conosce l'indirizzo e la porta di entrambi i server progettati.
- L'indirizzo e la porta del Bank Server sono noti solo ai server.
- L'applicazione si rivolge ad un impianto di cui siano noti il numero dei settori e il numero di posti che compongono ciascuna fila. **Non** è invece noto il numero delle file

che compongono ogni settore. Le modifiche in senso di espansione o contrazione della dimensione dei settori possono essere fatte aggiungendo o eliminando una fila intera.

- *Il Bank Server è immune da cadute*. Una volta attivato resta sempre presente.
- Per lo sviluppo dell'applicazione si è scelto *Java* in versione 2. L'applicazione è sviluppata su Macintosh 5500 con MacOS 9.1 utlizzando MacOS Runtime for Java Software Development Kit (MRJ SDK) versione 2.2 scaricata free da developer.apple.com/java/

Requisiti del sistema

- **Posizione fisica**

Le postazioni di lavoro client e server sono fisicamente distinte

- **Numero dei client**

Per come è strutturata l'applicazione non è necessario stabilire un numero minimo o massimo di processi client. E' tuttavia necessario che ognuno di essi sia dotato delle opportune informazioni necessarie al suo funzionamento.

- **Numero dei server**

L'applicazione è sviluppata con due server, uno clone dell'altro e privi di relazione master-slave. La duplicazione dei server è il requisito minimo (anche se probabilmente non ottimale) per lo sviluppo del progetto. Un possibile semplice miglioramento alla struttura indicata è analizzato nella sezione Miglioramenti di questa relazione.

- **Affidabilità**

E' uno dei requisiti fondamentali del sistema. L'utente deve **poter acquistare in ogni momento** il biglietto che desidera. Deve poterlo fare nel modo **più semplice** possibile e avere a **disposizione** l'intero elenco dei **posti disponibili** nello stadio. Compito dell'applicazione client è rendere snella ed efficace la comunicazione con il server. Compito dell'applicazione server è gestire la vendita, renderla **sempre** possibile, garantire la **consistenza delle informazioni** che gestisce e impedire la duplicazione delle prenotazioni per un medesimo posto.

- **Conoscenza client-server, server-server**

Per il corretto funzionamento dell'applicazione è necessario che il client conosca l'indirizzo di entrambi i server. Questo nel caso in cui uno dei due sia inutilizzabile per qualsiasi motivo.

Per quanto riguarda il server, è necessario che conosca l'indirizzo del suo omologo e del Bank Server con cui gestire la parte economica della transazione. Nell'applicazione in questione si suppone che il Bank Server sia unico.

- **Guasti**

Occorre prevedere quali politiche adottare nel caso in cui il server cada, dipendentemente dalla situazione in cui è avvenuto il crash. Occorre infatti fare in modo che il database resti sempre consistente a fronte di ogni inconveniente avvenuto al software, alla rete elettrica o alla rete di comunicazione.

Si fa in ogni caso l'ipotesi di guasto singolo, ossia si suppone sempre che almeno uno dei due server resti attivo. Il Bank Server non può mai cadere.

- **Gestione delle eventuali collisioni**

Trattandosi di risorse replicate e quindi della presenza in rete di più database (tanti quanti sono i server) e di un numero imprecisato di client, occorre gestire l'eventualità che uno stesso posto sia prenotato da due utenti diversi su server diversi in momenti diversi o addirittura contemporanei. Per risolvere il problema è necessario gestire un protocollo di comunicazione inter-server che tenga conto dell'assenza di priorità tra un server e l'altro. In pratica tutti i server sono non soltanto cloni uno dell'altro dal punto di vista delle azioni svolti e del codice, ma equiprioritari. Nessuno di loro ha cioè importanza maggiore degli altri, dal momento che non è prevista, in partenza, nessuna struttura di tipo master-slave nel senso classico del termine. Ogni transazione ha comunque un solo server responsabile della vendita al cliente e della comunicazione delle informazioni sulla transazioni al suo omologo. In pratica la relazione master-slave non esiste a livello di applicazione server, ma è presente all'interno dei processi singoli che gestiscono l'iterazione con il cliente.

- **Rapporti client-server e server server**

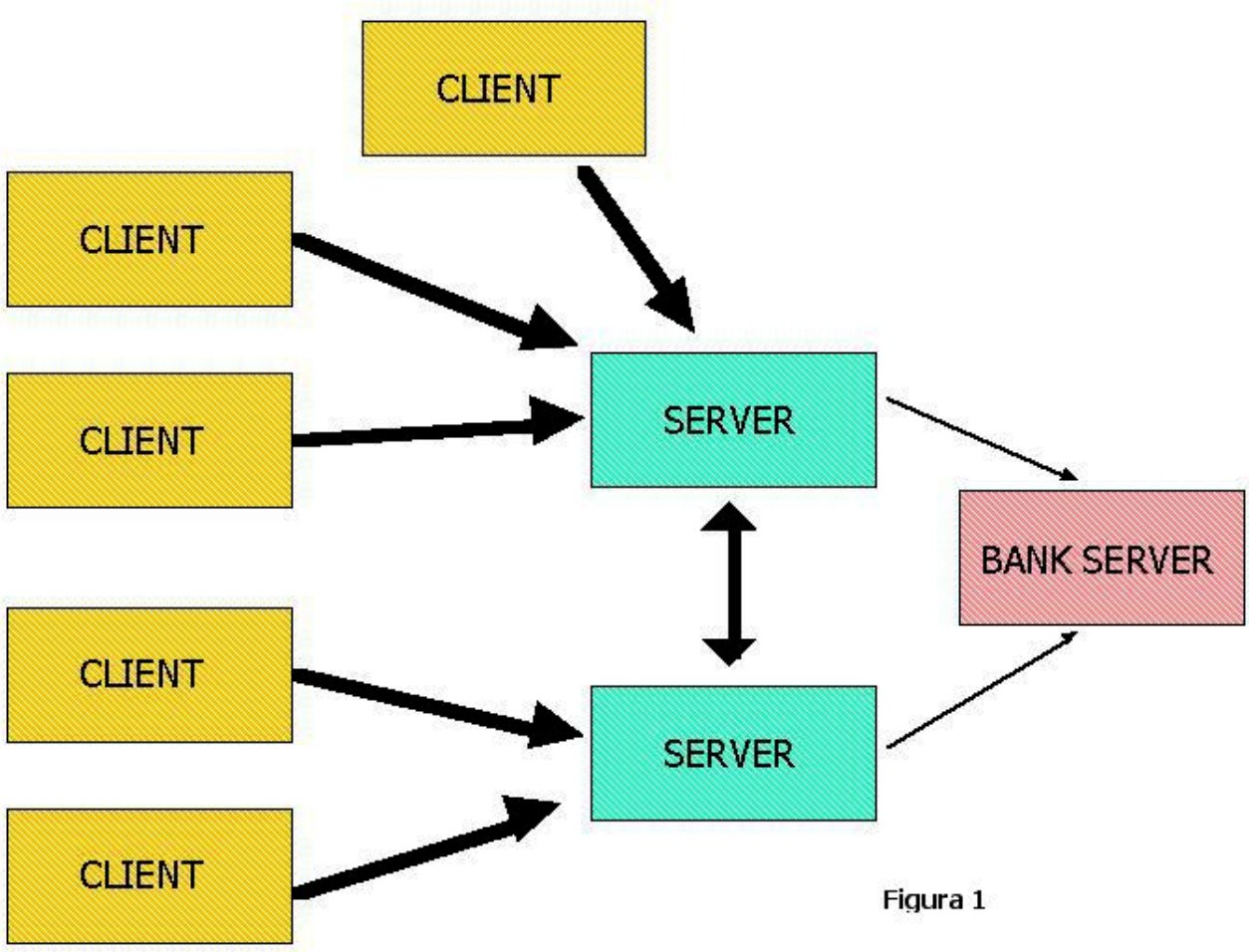


Figura 1

Come descritto in precedenza il progetto è stato realizzato con risorse replicate (in questo caso duplicate), senza relazione di precedenza tra i server e senza nessuna limitazione al numero delle applicazioni client che possono accedere al servizio.

La **Figura 1** illustra i rapporti intercorrenti fra server e client e fra server e Bank server, oggetto che verrà chiaramente messo in gioco solo in caso di transazione economica gestita tramite carta di credito.

Ogni client accede a uno specifico server di sua scelta e la consistenza dei dati è mantenuta tramite la comunicazione fra i due server, del tutto trasparente al client.

La comunicazione fra i server e fra il client e il server a cui è collegato segue specifici protocolli che tengono conto della particolare funzione che deve essere svolta. Tali protocolli basati sullo scambio di messaggi fra le applicazioni devono essere realizzati in maniera affidabile e in modo tale che la successione temporale dei messaggi sia garantita, dal momento che la cronologia è elemento fondamentale della comunicazione.

E' il server che fa da tramite fra client e Bank server in maniera tale da avere sempre la situazione economica della transazione sotto controllo. In caso di gestione del client da parte di un operatore al botteghino, la transazione finanziaria è sempre, dal punto di vista di chi incassa, sicura e non demandata alla responsabilità dell'acquirente.

Componenti e comunicazione

Progetto del client

- **Obiettivi**

Progettando il lato client è fondamentale tenere presente che ci riferiamo al lato dell'applicazione che starà a diretto contatto con il pubblico. Mentre il lato server può mantenere visibili alcune delle sue caratteristiche tecniche, da questa parte si deve tenere conto soprattutto del **cliente**, inteso in questo caso come la persona che acquista il servizio.

L'applicazione nasce infatti per poter essere utilizzata anche da un semplice acquirente non tenuto ad avere alcuna conoscenza tecnica. A questo tipo di utenza è necessario fornire un **utilizzo semplice, immediato e intuitivo**, nascondendo quindi ogni possibilità di modifica tecnica che invece sarà possibile all'operatore specializzato.

Occorre quindi una distinzione alla base, alla partenza dell'applicazione, per poter separare l'utenza esterna - con utilizzo diretto - da quella interna, che serve un acquirente come se fosse un normale negozio.

Si deve infatti tenere presente che qualunque modifica ai parametri tecnici del client mina non soltanto la qualità del servizio, ma la sua stessa esistenza. Il **problema** di **determinare l'identità** di chi accede è quindi il primo che si deve risolvere. In base a tale determinazione occorre quindi vincolare in un modo o nell'altro lo sviluppo dell'applicazione.

Successivamente per favorire la semplicità dell'utilizzo si è scelto di generare **un'interfaccia grafica** che guida l'operatore o l'utente attraverso la scelta del posto da acquistare, prevenendo il più possibile la generazione di un errore.

Come ultima istanza, trattandosi di una transazione commerciale, è necessario effettuare un **pagamento** per cui sono state previste due vie, una tradizionale, cash a un botteghino e una per carta di credito.

Del pagamento via carta di credito si parlerà più diffusamente nella parte relativa alla *sicurezza*.

- **Scelte progettuali**

Dal momento che, come si è visto, il tempo di vita dell'applicazione passa attraverso differenti fasi, si è scelto di creare oggetti diversi che rappresentano i momenti diversi dell'applicazione.

In pratica si tratta di due diverse *famiglie* di classi Java.

Una si occupa della parte grafica fondamentale per la facilità d'uso del sistema e l'altra svolge il lavoro sotterraneo che permette la visualizzazione dell'informazione.

Si tratta quindi di una **struttura a due livelli**. Il livello più basso effettua il collegamento fisico con il server, richiede e riceve le informazioni e le rende disponibili per

il livello superiore che si occupa di tramutarle in grafica gestibile facilmente dall'utente finale.

E' evidente come il livello più basso sia la parte fondamentale di questa struttura, mentre quello più alto crea soltanto *il vestito*, l'abito che il pubblico vede e su cui lavora. Tutto ciò che riguarda i protocolli di comunicazione, la forma in cui le informazioni vengono inviate e ricevute viene nascosto e non può essere modificato dall'utente finale. Solo alcuni parametri sono accessibili, ma solamente all'operatore e non all'acquirente singolo.

- **Chi è l'utente?**

L'identificazione dell'utente è quindi la prima procedura che il cliente deve effettuare una volta inizializzatosi. Per fare ciò si utilizza una **registrazione preventiva** degli operatori. All'immissione di user e password il primo controllo viene fatto localmente cercando gli identificativi all'interno di un file chiamato *access.txt*. E' prevista quindi una registrazione locale degli id degli operatori.

Ritrovando la registrazione nel file, l'utente è identificato come operatore, altrimenti come acquirente esterno. L'applicazione tiene quindi nota dell'identità per fornire l'esatta elaborazione degli oggetti che seguiranno.

- **Dov'è il server?**

L'applicazione nasce conoscendo gli indirizzi dei due server da cui può essere ascoltata. Chiaramente si tratta di una visione del mondo piuttosto ridotta. Per qualsiasi motivo il server può dover essere rilocato e quindi modificare il proprio indirizzo, la propria porta di ascolto o addirittura entrambi i parametri. E' necessario quindi che l'applicazione client possa trovare il server in ogni momento e che non sia necessario riscrivere il codice Java degli oggetti che si occupano del collegamento ogni qual volta uno dei due server (o entrambi) viene spostato.

Per far fronte a questo problema l'applicazione utilizza un **file di preferenze**.

Il file, denominato *preferences.txt*, contiene indirizzo e porta di entrambi i server e un criterio di selezione privilegiato che determina a quale dei due si deve tentare in prima istanza il collegamento. **Il file è modificabile dall'applicazione**.

Se il server deve migrare per qualche motivo è sufficiente entrare nell'applicazione client come operatore e modificare i valori da un'apposita finestra selezionabile da un menu. Tale finestra gestita dall'oggetto SetupWindow permette anche la modifica del criterio di scelta sull'accesso a quello che potremmo chiamare *main server* e che sarà il primo a essere cercato all'avvio del collegamento.

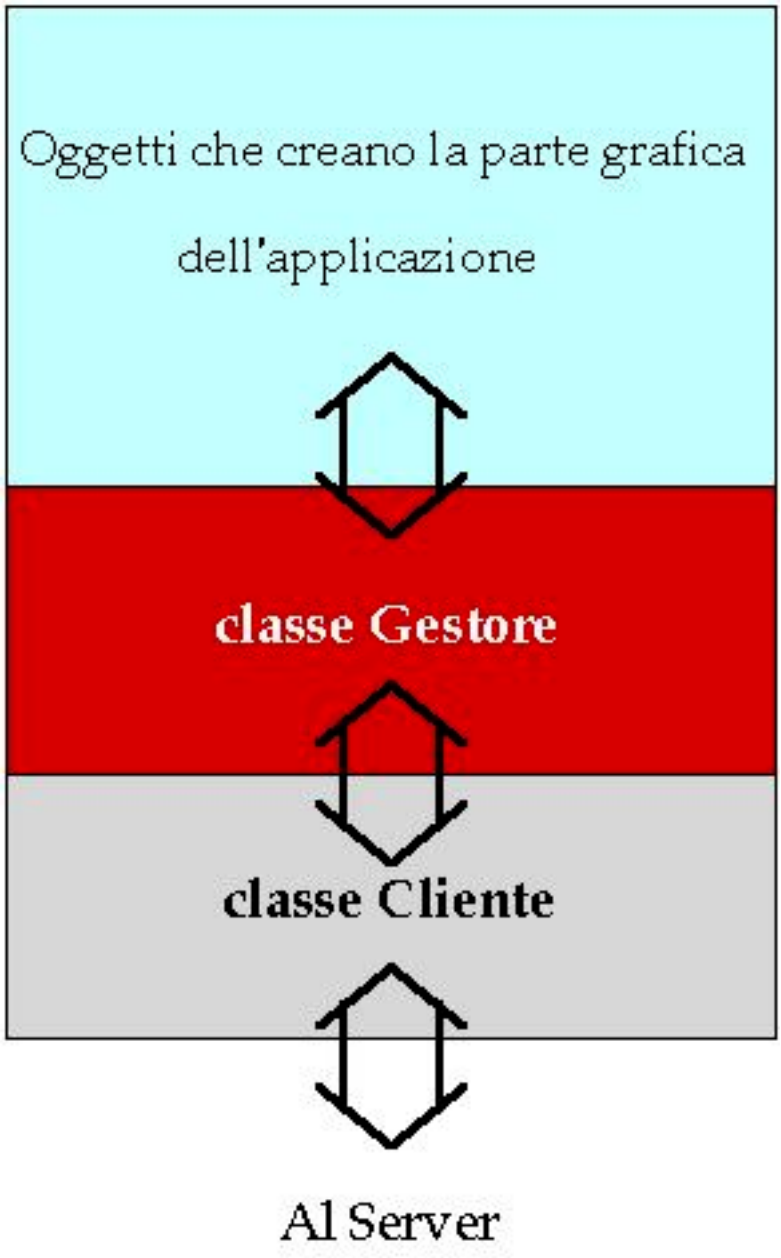
Naturalmente sono state prese misure di prevenzione che tendono ad assicurare la consistenza o addirittura la presenza stessa del file di preferenze. Queste procedure verranno discusse nella parte relativa all'inizializzazione del client.

- **Comunicazione esterne**

Abbiamo visto come il motore dell'applicazione sia in realtà l'oggetto Gestore. Continuando con un paragone motoristico si può dire che le ruote siano rappresentate dall'oggetto **Cliente**. Su questo oggetto infatti vengono svolte le operazioni primitive che coinvolgono i protocolli di comunicazioni: creazione e chiusura del collegamento con il server, spedizione e ricezione dei messaggi, richiesta di accesso e gestione della risposta.

In pratica, ripensando all'organizzazione a livelli di cui abbiamo parlato sopra, il Cliente rappresenta il livello più basso, quello più vicino al protocollo di comunicazione, ignaro dell'utilizzo che verrà fatto delle informazioni che reperisce e smista. E' infatti il Gestore che dà un significato logico in termini del nostro progetto alle informazioni del Cliente.

Graficamente la struttura è quindi rappresentabile come in figura.



- **Comunicazioni esterne, oggetto Cliente**

Come si vede dalla figura è la parte più bassa della "pila" che gestisce il client.
Concettualmente è un oggetto piuttosto semplice e quasi per nulla evoluto, contenendo soltanto pochi metodi che servono per creare la connessione con il server (qualunque esso sia), per spedire e ricevere messaggi e autenticare l'accesso. In pratica quest'ultimo è l'unico metodo di più alto livello dell'oggetto che, per il resto, è passivo nelle mani del Gestore. In pratica fornisce le parole, ma non il loro significato.

- **Classe Gestore**

Diamo qui solo alcuni brevi cenni sull'oggetto Gestore, che sarà più ampiamente trattato descrivendo i protocolli di comunicazione.
In pratica la funzione del Gestore è una soltanto, ma fondamentale. Coordina tutto il lavoro dell'applicazione. E' a lui che si rivolgono tutti gli oggetti grafici per richiedere le informazioni necessarie al loro funzionamento (i prezzi dei settori, le mappe dei settori, l'accesso al server…) ed è quindi l'unico oggetto Java presente per tutto il ciclo di vita dell'applicazione. Viene distrutto insieme agli altri nel momento in cui si decide di uscire dall'applicazione stessa. Per ogni passaggio tra un oggetto e l'altro (dalla scelta del settore alla scelta del posto, ad esempio) viene passato il riferimento all'oggetto Gestore che deve essere noto a ogni classe che si occupa dell'elaborazione.

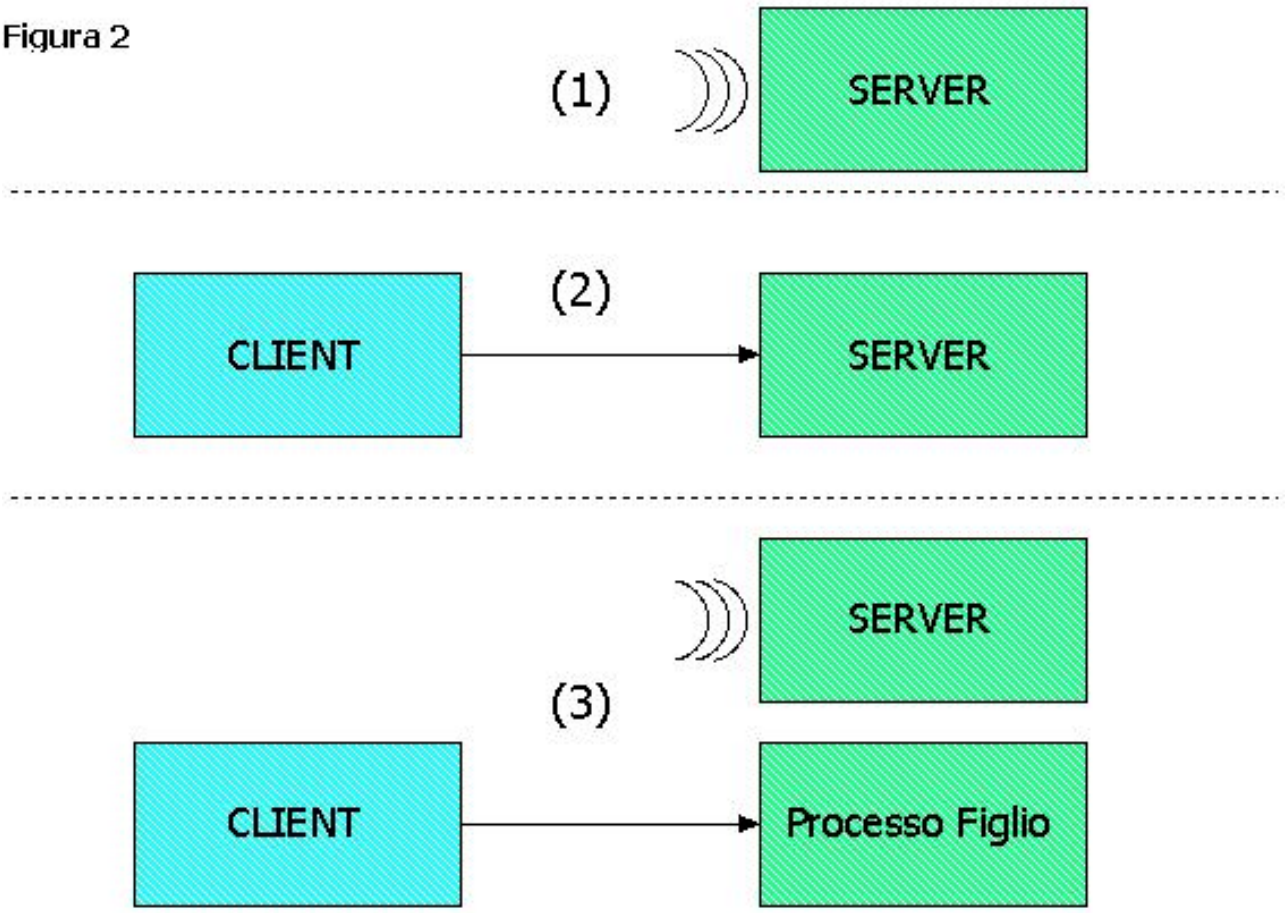
Progetto del server

- **Problemi di partenza e risoluzioni**

Dato il numero illimitato di applicazioni client e due sole applicazioni server come è possibile gestire un rapporto di 2:N ?

E' necessario, anche per garantire un'azione snella ed efficace della procedura di prenotazione, progettare un **server parallelo**.
Ogni applicazione server gestisce le richieste provenienti da un client generando un **processo figlio (serverThread)** che relazionerà con il client che ha richiesto l'accesso risolvendone le richieste nella maniera appropriata.

Figura 2

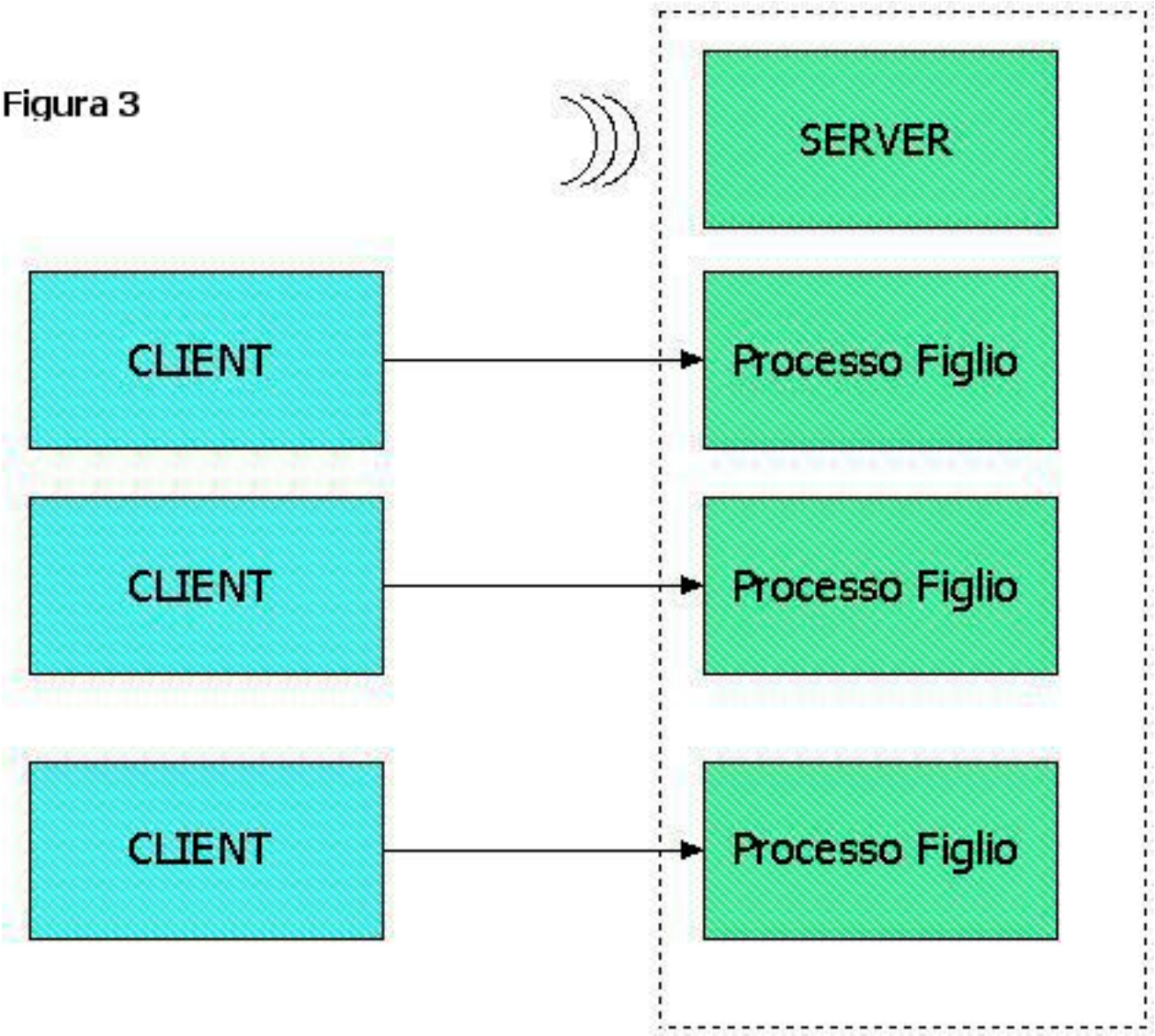


La **figura 2** mostra le tre fasi con cui avviene l'accesso del client al servizio.

In **(1)** il server è in ascolto, in attesa di ricevere la richiesta. In **(2)** giunge la richiesta del client che viene gestita assegnandola a un **processo figlio** che la gestirà, creando così la situazione indicata in **(3)**. Da quel punto in avanti sarà il processo generato ad occuparsi della gestione di ognuna delle richieste del client, mentre il server si porrà di nuovo in ascolto in attesa di ulteriori richieste in arrivo.

Ad un tempo t_n imprecisato, potranno esserci più processi client collegati sul medesimo server, creando la situazione illustrata nella **figura 3**.

Figura 3



Le loro richieste sono servite ognuna da un processo il cui tempo di vita comincia e finisce con quello della transazione.

Contemporaneamente il server resta in ascolto aspettando nuove richieste da poter servire allo stesso modo.

Come gestire la contemporaneità delle richieste e non intaccare la consistenza del database ?

E' uno dei problemi fondamentali dell'applicazione dal momento che dalla consistenza delle informazioni dipende l'esatta fornitura del servizio. Il problema non si pone soltanto nella gestione della concorrenza fra i due server, ma anche fra due processi figli che lavorano in parallelo all'interno dello stesso server.

Deve esistere un meccanismo di protezione delle informazioni che impedisca inconsistenza dei dati. L'accesso al database **deve** perciò avvenire in **mutua esclusione**. Non può quindi esservi accesso contemporaneo alla risorsa da parte di due processi diversi ed è necessario dotare il server di un sistema di sincronizzazione efficace.

All'interno di ogni processo figlio verrà gestita l'intera transazione, dalla richiesta di accesso alla risorsa fino al pagamento del posto acquistato, a seguito di una positiva conclusione della trattativa con il Bank server o di un pagamento cash al botteghino.

- **Comunicazioni esterne**

Fondamentalmente il server deve relazionare con quattro differenti oggetti: il **database** da cui reperisce le informazioni, il **cliente** che richiede le informazioni, un eventuale altro **serve**r con cui garantire la consistenza e il **Bank Server** in caso di transazione economica.

In ognuno di questi casi, occorre interfacciare la struttura scelta per il server con i requisiti richiesti dal particolare tipo di rapporto. Per ogni soggetto con cui il server si trova a relazionare viene creato un oggetto che funge da "porta".

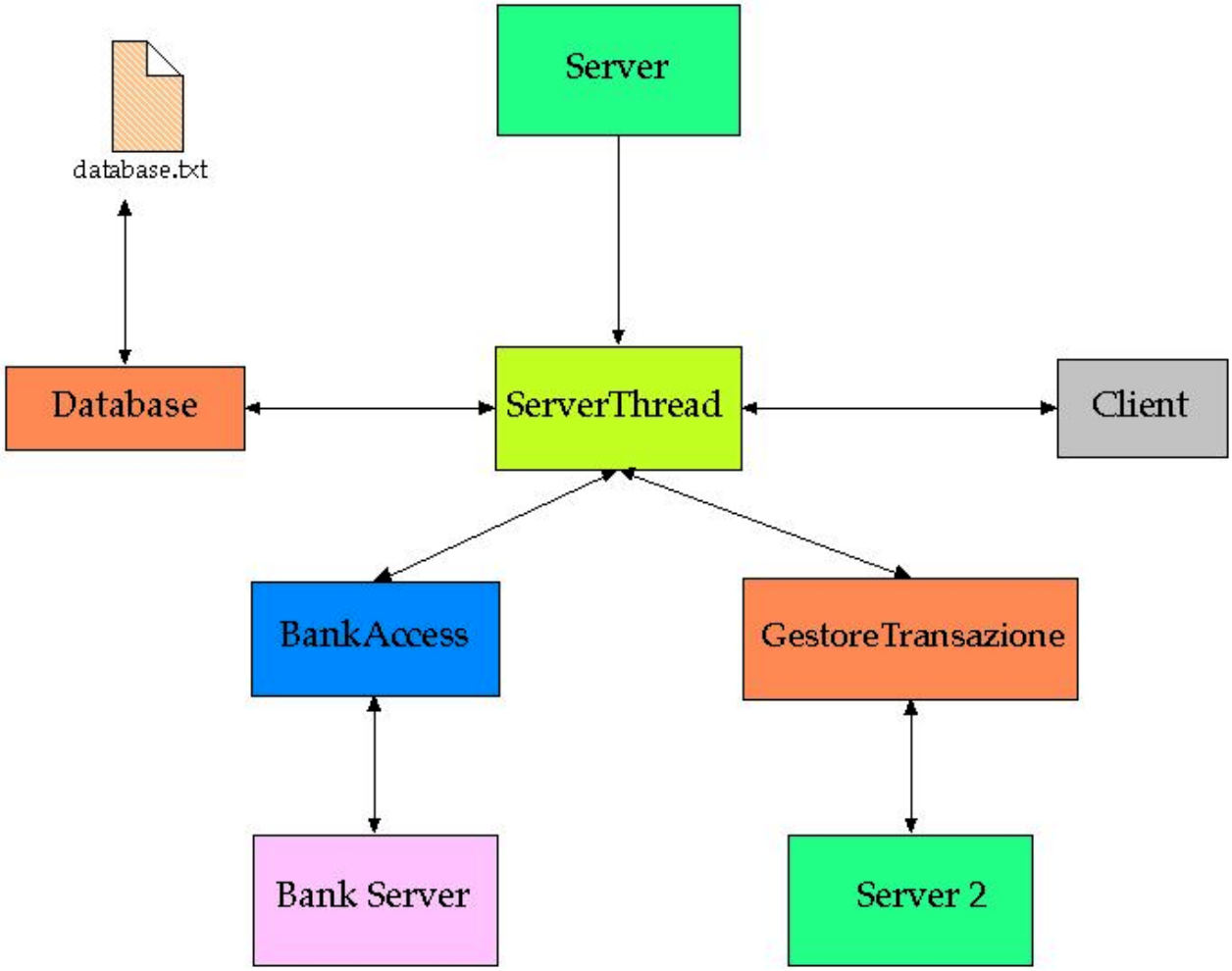
La classe **Database** rapporta il server con il database fisico e logico.

La classe **ServerThread** rapporta il server con ognuno dei client che ne richiede il servizio.

La classe **BankAccess** rapporta il server con il Bank Server.

La classe **GestoreTransazione** rapporta il server con il suo omologo.

La figura qui sotto rappresenta, per un solo client, la situazione descritta.



- **Comunicazioni esterne, classe Database**

Della classe Database parleremo nel dettaglio nella sezione a lei dedicata. Sottolineiamo ora solamente il fatto che è monoistanziata, per garantire che l'accesso alle informazioni avvenga da una sola porta.

- **Comunicazioni esterne, classe ServerThread**

E' il processo figlio a cui il server delega lo svolgimento delle funzioni richieste dal client. Chiaramente viene generata un'istanza di classe per ogni processo client che richiede l'accesso. Le funzioni svolte dal thread e i protocolli di comunicazione con il client occuperanno una sezione successiva di questa relazione.

- **Comunicazioni esterne, classe BankAccess**

E' l'oggetto che si preoccupa di creare, mantenere e gestire la connessione con il Bank Server. Viene creato naturalmente solo a fronte di una transazione economica gestita con carta di credito. E' una risorsa replicata, nel senso che ogni ServerThread ne crea uno che gestisce la transazione economica al suo posto e gli comunica il risultato.

- **Comunicazioni esterne, classe GestoreTransazione**

E' l'oggetto che si occupa di gestire la comunicazione con l'omologo del server in uso. In pratica, dal momento che i server sono equipotenti, in ognuno di loro, in ogni momento della loro attività possono essere in corso transazioni di tipo **master** e transazioni di tipo **slave**. Con transazioni di tipo **master** si intende quel tipo di comunicazione in cui un server gestisce il client e interroga l'omologo per avere informazioni sullo stato del posto che gli serve (gestione della concorrenza) e/o comunicare l'avvenuta chiusura (positiva o negativa) della transazione (gestione della consistenza). All'interno del server quindi vi possono essere più thread di tipo slave e master contemporaneamente e proprio per questo motivo la gestione di concorrenza e consistenza va tenuta particolarmente in attenzione. Del GestoreTransazione si parlerà più dettagliatamente all'interno della parte sui protocolli di comunicazione.

Progetto del BankServer

E' l'oggetto che si occupa di gestire la parte puramente economica della transazione finanziaria. La sua progettazione è del tutto speculare a quella del server, se si esclude il fatto che è notevolmente più semplice.

Ricordiamo inoltre che una delle ipotesi di lavoro fatte è che **il Bank Server non cada mai**.

La struttura dell'oggetto è quella di un server parallelo. Ogni richiesta che giunge viene servita da un thread separato che resta in vita fino alla fine del proprio servizio. Allo stesso modo in cui il server deve tenere consistenti le informazioni del database, il Bank Server deve occuparsi di mantenere aggiornate in maniera precisa, puntale e priva di errori la situazione economica dei conti correnti di cui si occupa. Viene da sé che se un errore del server che gestisce il database può creare un disguido anche importante sulla vendita di

un posto, un errore nella gestione del Bank Server crea una perdita economica al cliente, danno molto più importante e decisamente meno *digeribile*.

Occorre quindi creare una gestione attenta ed affidabile.

Anche in questo caso, come per l'oggetto Database, creiamo una classe ad hoc che si occupa della gestione dei conti correnti. In pratica l'oggetto BankServer si occupa solamente della parte di rete della transazione economica, gestendo la comunicazione con il server che fa richiesta di accesso e restituendo un valore indicativo dell'esito della transazione. Un puro e semplice scambio di messaggi.

Tutta la transazione viene invece svolta da un altro oggetto chiamato **GestoreBankServer** che si occupa della parte economica vera e propria e che, ancora in anaologia con la classe Database, viene creato dal server al suo avvio in un'unica istanza e passato ad ogni processo figlio che lo utilizza.

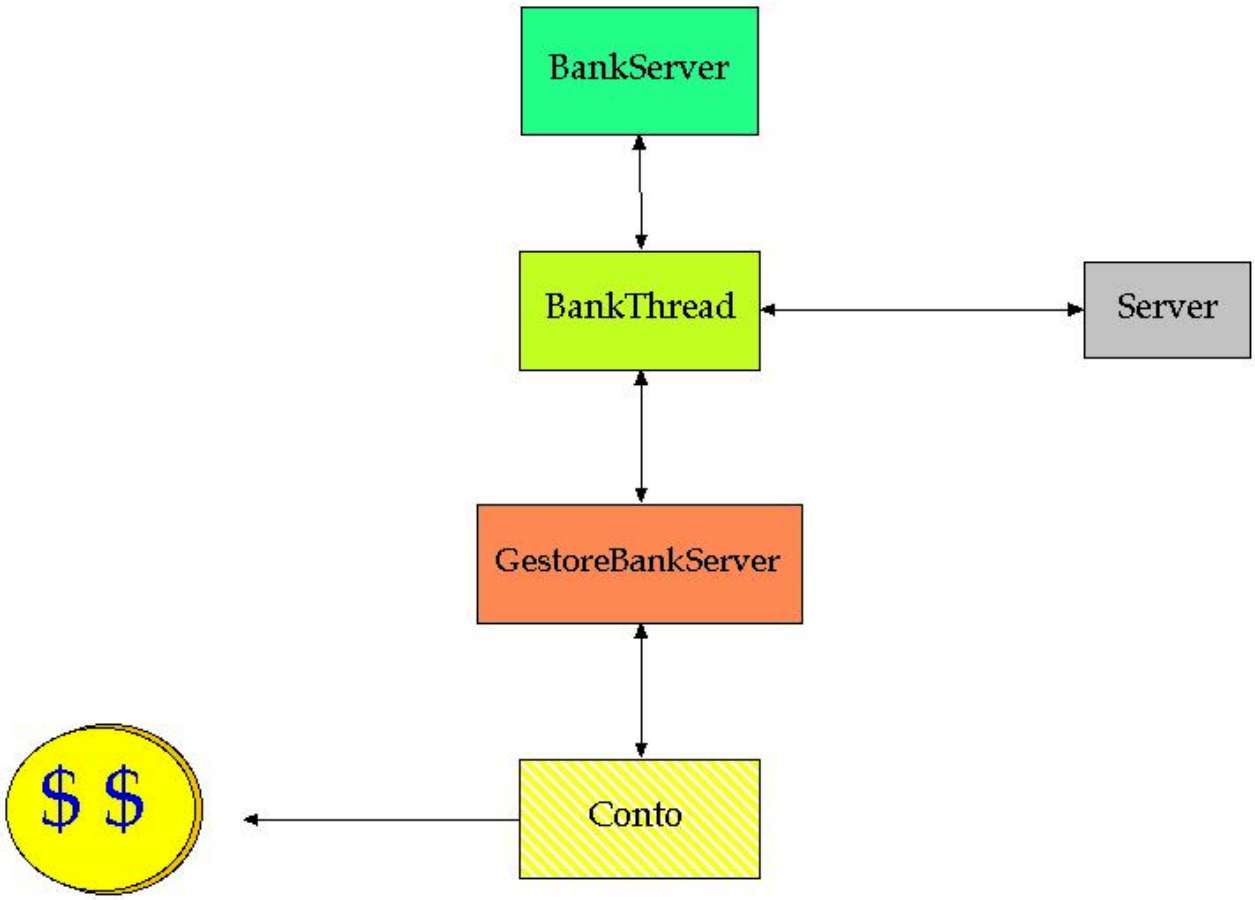
In pratica, così come la classe Database era la porta di accesso all'informazione logica e fisica, l'oggetto GestoreBanKServer è la porta d'ingresso alla nostra banca virtuale e gestisce gli addebiti e gli accrediti necessari alla nostra operazione.

E' evidente come qui, più che in ogni altra applicazione del nostro progetto, diventi largo il gap con la situazione reale, sia in termini di applicazione di gestione, sia, e lo vedremo più avanti, in termini di sicurezza.

Nella gestione della parte economica diventa fondamentale anche l'astrazione che si crea del conto corrente, qui diventato un oggetto Java chiamato **Conto** su cui vengono definiti i metodi necessari allo svolgimento delle operazioni.

Schematicamente la situazione si può quindi visualizzare come nella figura.

Chiaramente si tratta di una semplificazione valida *per ogni server che richiede la*



transazione. Ognuno dei server riceverà il servizio da un'istanza della classe **BankThread**

mentre il GestoreBankServer sarà sempre e soltanto uno, creato all'inizializzazione del BankServer.

Vediamo ora nel dettaglio come sono gestite le classi.

- **Classe Conto**

E' l'oggetto che crea il nostro conto corrente virtuale.

Ogni conto è caratterizzato dai seguenti parametri che lo **identificano univocamente**:

- **Nome** del titolare del conto
- **Cognome** del titolare del conto
- **Numero** del conto corrente
- **Disponibilità (campo disponibile)** attuale del conto
- **Password** per avere l'accesso in rete
- **Un campo boolean** che indica se il conto è disponibile per l'elaborazione o se è in atto una transazione.
- **Un vettore Operazioni** in cui vengono memorizzate nel dettaglio tutte le operazioni effettuate sul conto

Per semplicità - e soprattutto perché per la nostra applicazione non ne servono altre - sono stati creati due soli metodi che rappresentano le due operazioni principali applicabili al conto corrente: prelievo e deposito, entrambe previste *void* e quindi senza ritorno.

L'operazione di **prelievo** decrementa il campo *disponibile* della cifra richiesta, crea un oggetto *Date* che rappresenta la data dell'operazione e aggiorna il vettore *operazion*i con la transazione effettuata.

L'operazione di **deposito** è totalmente speculare ma, chiaramente, incrementa il valore del campo *disponibile*.

Da notare che non vengono effettuati controlli sulla disponibiltà dei conti prima del prelievo. Tali controlli vengono demandati direttamente alla supervisione del Gestore.

- **GestoreBankServer, la nostra banca virtuale**

E' la classe che crea l'astrazione dello sportello bancario, la nostra banca virtuale.

Il database dei conti è rappresentato in un **vettore di oggetti Conto** denominato appunto *banca,* ancora una volta in completa analogia con la classe Database. Qui tuttavia entra in gioco un'importante semplificazione. Dal momento che un crash del BankServer non è previsto, non è necessario un aggiornamento fisico del DB dei conti correnti. Ci limitiamo perciò alla creazione del database logico a partire da dati fittizi che l'applicazione conosce. Chiaramente si tratta di una semplificazione notevole e limitante, ma quello che interessa maggiormente è il rapporto fra i server per la gestione della transazione economica e non la creazione di un vero e proprio sportello bancario virtuale.

L'inizializzazione del server è affidata proprio all'oggetto GestoreBankServer che dai dati di cui è in possesso crea l'archivio dei conti correnti. Altri due metodi completano la gestione delle operazioni.

Il metodo **preleva** riceve in ingresso i dati relativi al conto corrente di interesse, verifica la disponibilità del conto e se possibile effettua il prelievo grazie al metodo corrispondente

della classe Conto, occupando il conto corrente relativo per tutta la durata dell'operazione e liberandolo alla fine. E' previsto un ritorno intero (int) che informa sull'avvenuta esecuzione o sui motivi dell'impossibilità ad eseguire.

Il metodo **ripristina** compie funzioni analoghe ma simmetriche. Chiaramente non è necessario nessun controllo sulla disponibiltà, dal momento che si effettua il deposito e non il prelievo. Sfrutta il metodo deposito della classe Conto e ritorna un valore intero (int) a garanzia dell'avvenuta o non avvenuta transazione.

- **Comunicazioni esterne, classe BankThread**

E' il processo figlio chiamato dal BankServer ad occuparsi della richiesta.

E' lui che si occupa dello scambio di messaggi con il server che richiede il servizio e dei meccanismi di protezione della consistenza dei conti correnti.

Per i protocolli di comunicazione utilizzati si veda l'apposita sezione della relazione.

Progetto di comunicazione

Prima di passare alla descrizione vera e propria dell'attività svolta dall'applicazione, occorre definire i modi e gli aspetti tecnici in cui gli oggetti in gioco relazionano fra loro. Rivestono un ruolo importante il **tipo di socket** utilizzata, il modo in cui vengono scambiati i **messaggi**, le primitive che servono a **trasmissione e ricezione** e la validità di tali messaggi, ossia cosa ci aspettiamo di ricevere ogni volta che un mittente ne spedisce uno.

- **Socket**

In Java, come nella maggior parte dei linguaggi che si occupano di comunicazione in rete, abbiamo a disposizione due differenti tipi di socket relativi a due protocolli disponibili su Internet: TCP e UDP. Le classi relative alle socket di entrambi i tipi sono comprese nel package **java.net** che va quindi *importato* da ogni classe che le utilizza.

La differenza fra i due protocolli citati è fondamentale e va tenuta conto nel momento di effettuarla scelta di una a scapito dell'altra. Si tratta di un protocollo orientato alla connessione e di uno senza connessione.

Il **protocollo TCP** è orientato alla connessione e stabilisce un collegamento di comunicazione tra un indirizzo di origine e uno di destinazione che rimane valido fino alla chiusura del collegamento.

Il **protocollo UDP** è senza connessione ed è quindi meno affidabile del TCP per quanto riguarda garanzia di consegna e rilevazione (e correzione) degli errori. La differenza è un po' quella che passa tra una lettera e una telefonata.

Si tratta quindi di scegliere fra i due protocolli.

Nel nostro tipo di applicazione vi sono alcuni requisiti fondamentali che non possono essere trascurati.

La **cronologia** dei messaggi è importante ai fini della corretta interpretazione degli stessi. I messaggi devono arrivare nell'ordine in cui sono stati spediti. Si pensi ad esempio alla ricezione invertita delle informazioni relative a due posti differenti, che andrebbero quindi confusi uno con l'altro.

L'**esattezza** dei messaggi deve essere garantita per essere certi che l'informazione ricevuta e spedita sia quella corretta.

E' necessario un **canale di comunicazione** che tenga legati gli oggetti che compongono l'applicazione dal momento che le relazioni che li legano sono continue durante tutto il tempo di vita della transazione.

Viene da sé che la **scelta della socket TCP** è necessaria alle nostre esigenze.

- **Messaggi**

Lo scambio di messaggi è il cuore della comunicazione fra i server e fra server e client. Dalla precisione di questo scambio dipenda la corretta interpretazioni delle notizie ricevute e quindi il giusto svolgimento dell'applicazione.

Si è scelto di scambiare messaggi sotto forma di stringhe di testo. Le informazioni vengono convertite prima di essere spedite e riconvertite all'atto di ricezione in istanze della classe String su cui poi viene fatta l'esatta elaborazione che serve a interpretarle.

Si è cercato di limitare il numero e la dimensione dei messaggi necessari al chatting fra i vari oggetti remoti al fine di evitare il più possibile gli errori e di rendere snella e veloce la comunicazione.

Chiaramente esiste un preciso **vocabolario** di messaggi consentiti e una politica da eseguire in caso di ricezione di un messaggio errato o non compreso nel vocabolario. Il vocabolario è esaustivo per quello che riguarda la nostra applicazione.

- **Spedizione e ricezione dei messaggi**

Ogni oggetto che svolge direttamente un'attività di rete deve possedere le primitive necessarie alla ricezione e alla spedizione dei messaggi. Tali primitive sono universali all'interno del nostro progetto, nel senso che tutti gli oggetti che le utilizzano hanno lo stesso tipo di primitive.

Per motivi ovvi i due metodi che implementano tali primitive sono **sendMessage** e **receiveMessage**. Entrambe prevedono un valore di ritorno. Mentre per la receive è necessario, in quanto se ricevo un messaggio è logico che voglia sapere cosa contiene, per la send non è così scontato dal momento che si potrebbe benissimo lasciarla void e quindi invocarla senza che ci si aspetti qualche tipo di risposta.

Si è preferito invece definirla *int*.

La send pertanto rilascia un intero come valore di ritorno, a conferma della spedizione (1) o dell'occorrenza di un'eccezione (0) che ne ha impedito la spedizione. Tutto questo permette di verificare la presenza del collegamento e la nascita di eventuali problemi. In base a tale valore di ritorno gli oggetti che manipolano i messaggi sanno quali azioni compiere, in che modo ripristinare la consistenza dei dati e quali politiche di concorrenza attuare per la corretta prosecuzione della transazione.

Inoltre, come ulteriore meccanismo di prevenzione, in caso di problemi sulla prima spedizione, la send provvede a tentarne una seconda prima di fallire.

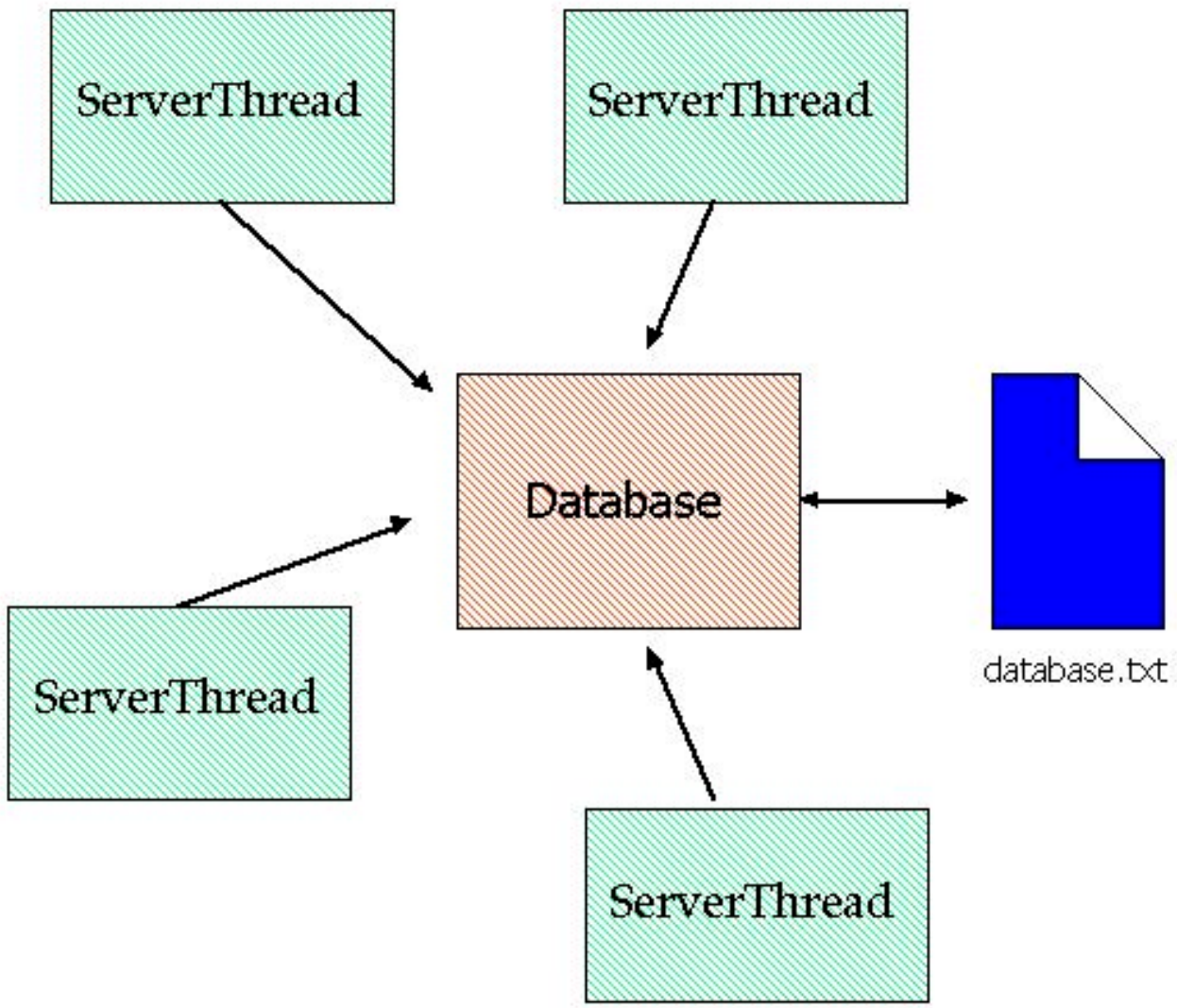
L'oggetto database

Tutta la parte fondamentale dell'applicazione ruota intorno al database dei posti. Occorre quindi porre particolare attenzione al modo in cui il database stesso viene creato e gestito dall'applicazione server.

Chiaramente bisognerà avere una copia fisica del database, salvata su un supporto di memoria di massa. Questa copia, oltre ad essere strettamente necessaria all'applicazione, rappresenta un requisito obbligatorio per ogni servizio che voglia espandere l'applicazione di cui ci stiamo occupando, sia essa contabile, pubblicitaria o semplicemente statistica.

Tuttavia resta quasi impossibile servirsi del database fisico per gli scopi che ci siamo prefissati. Occorre infatti un accesso rapido sia in fase di lettura che in fase di aggiornamento e i tempi necessari all'I/O da dispositivo fisico risulterebbero troppo elevati per le nostre esigenze. Inoltre occorre poter accedere in maniera facile ed efficace alle informazioni senza lunghe ricerche che rallenterebbero ulteriormente l'elaborazione.

Utilizziamo perciò un oggetto Java che sostituisca il database e che implementi le funzioni che servono al nostro scopo. Il problema della consistenza di cui si è discusso, viene risolto creando un'unica istanza dell'oggetto in questione. Nella pratica il nostro oggetto sarà la copia virtuale del database fisico e su di esso opereremo. A questo punto



perciò, quando ci riferiremo al database sarà al **database logico** creato da questa classe e non a quello fisico residente sulla macchina.

La figura illustra il meccanismo utilizzato.

Chiaramente l'ipotesi di lavoro fatta è che il database sia semplificato con un tradizionale file di testo. (La realtà progettuale dell'applicazione potrebbe e dovrebbe essere molto più complessa. La tratteremo nella sezione *Miglioramenti*.**)**

Così come la copia fisica esiste in un'unica istanza, allo stesso modo l'oggetto database sarà unico, creato dal server all'inizio della sua esecuzione. Il riferimento a tale oggetto sarà poi passato a ognuno dei processi figli che verranno creati per gestire le richieste del client. E' all'interno della classe Database che verrà implementato il meccanismo di mutua esclusione che garantisce la consistenza del database e la concorrenza di scrittura e lettura da parte di processi diversi.

Per fare ciò si usa una caratteristica di Java. La sincronizzazione nell'accesso alle risorse è infatti garantita dalla possibilità di definire **monitor**. I metodi del database vengono perciò definiti utilizzando la keyword **synchronized** che associa a ogni metodo un lock. Questo tipo di definizione dei metodi di accesso al database garantisce che *due thread diversi non possano accedere a due sezioni synchronized dello stesso oggetto*.

- **Inizializzazione delle funzioni**

Per garantire lo svolgimento delle sue funzioni, l'oggetto Database ha bisogno della copia fisica del file di database su memoria di massa. E' compito dell'applicazione server creare il file di database di cui l'oggetto necessita per un corretto funzionamento.

Successivamente, dalla copia fisica, creiamo la copia logica in un vettore di oggetti di tipo Ticket. Su questo DB logico effettueremo tutte le operazioni che coinvolgono l'applicazione.

Per quanto riguarda le informazioni sui proprietari dei posti, l'oggetto si preoccupa di verificare l'esistenza del file di dati e da esso recupera le informazioni con cui aggiorna la propria base di dati.

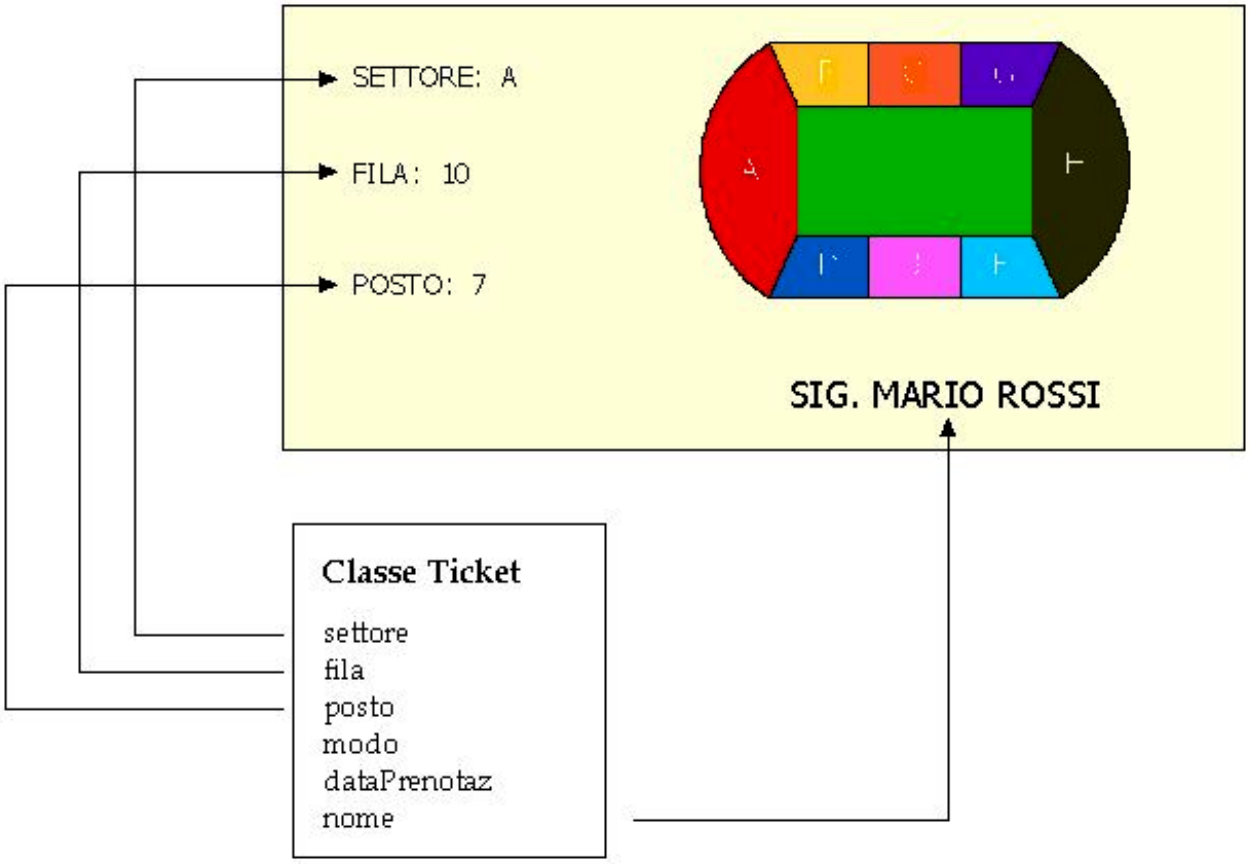
- **Classe Ticket**

Abbiamo spiegato poco fa che la nostra astrazione del database è composta da un vettore di oggetti di tipo Ticket.

La **classe Ticket** è creata appositamente per gestire ogni componente del biglietto utile al server. L'informazione di cui il server ha bisogno è differente infatti da quella necessaria al client, in cui la gestione del biglietto si riduce alle caratteristiche identificative (settore, fila, posto) e a una componente grafica che serve a rappresentarlo all'utente.

La classe Ticket è stata creata appositamente per il lato server dell'applicazione.

La figura nella pagina successiva illustra il rapporto fra la classe Java creata e l'emissione reale del biglietto.



Come si vede, oltre a tenere conto dei parametri reali necessari all'acquirente per la posizione fisica del posto, la classe presenta due campi ulteriori che l'applicazione utilizza durante l'elaborazione e la fase di transazione necessarie all'acquisto.

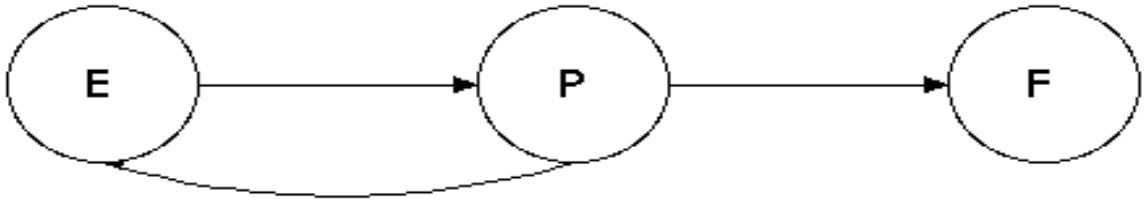
Il campo **modo** rappresenta lo stato in cui il posto rappresentato dal biglietto può esistere durante l'elaborazione. In figura sono rappresentati i possibili stati:

E: Posto libero

P: Posto prenotato
ma non ancora venduto

F: Posto occupato
e quindi venduto

L'evoluzione dello stato in questione è quindi:



Il posto nasce come *libero* (E), diventa *prenotato* (P) per tutta la durata della transazione e infine *occupato* (F), nel momento in cui è assegnato e l'eventuale conflitto sulla proprietà è stato risolto. Chiaramente può ritornare libero, dallo stato di prenotato, ma mai dallo stato di occupato. L'assegnazione del posto è permanente e l'applicazione non prevede ritorno. In pratica significa che **non è possibile vendere o cedere un biglietto acquistato**.

Il ritorno allo stato di libero può avvenire a causa di un'attribuzione persa con un altro server oppure a seguito della caduta del client durante la transazione o a seguito della caduta del server che ne aveva la proprietà prima che la conferma del posto venisse data al client acquirente.

Trattandosi, come detto, di applicazione commerciale non è possibile cambiare lo stato del biglietto **dopo** aver dato conferma al cliente della vendita. Quando il cliente ha la sicurezza di aver acquistato il biglietto l'informazione contenuta nel database **deve** essere consistente.

Il campo **dataPrenotaz** della classe Ticket assegna al biglietto un timestamp che lo identifica temporalmente all'atto della prenotazione. Il dato temporale non è necessario alla vendita in sé stessa, ma è fondamentale nella risoluzione delle controversie con altri server, secondo un protocollo che vedremo più avanti.

In realtà non si tratta di una data di prenotazione ma più precisamente di un'ora di prenotazione. Questo perché il giorno in cui il biglietto viene riservato non è determinante al fine della risoluzione della controversia con gli altri server.

• Oggetto database: funzioni realizzate

Dal momento che il nostro oggetto Java è facente funzioni di un database reale occorre che su di esso siano implementate tutte le funzioni principali di una base di dati. Mentre in un database reale la mutua esclusione nell'acceso ai record è garantita dall'applicazione che gestisce la base dei dati, nel nostro caso utilizziamo il monitor nativo di Java, espresso mediante la keyword synchronized prefissa alla definizione dei metodi.

Abbiamo così la sicurezza che due processi differenti che necessitano di eseguire la medesima operazione sul nostro database, possano effettuarla in mutua esclusione.

Ricordiamo che la nostra base di dati è composta da record rappresentati da istanze diverse della classe Ticket e su di esse saranno effettuate le operazioni necessarie all'applicazione. In particolare:

- getPrices: che permette di ricavare le informazioni riguardanti i pressi degli abbonamenti nei vari settori dello stadio
- getMap: che permette di ricavare la mappa di un determinato settore dello stadio
- readAndUpdateDB: che dato un posto (settore, fila, posto) permette di verificarne la disponibilità, di prenotarlo e di creare il timestamp necessario all'elaborazione
- checkPlace: che dato un posto (settore, fila, posto) permette di verificare il campo *modo* del biglietto in questione
- getDataPrenotaz: che ritorna il valore del campo *dataPrenotaz* del biglietto in questione
- setTicket: che dato un posto (settore, fila, posto) e uno stato permette di settare il posto suddetto allo stato voluto
- setName: che dato un posto (settore, fila, posto) setta il nome del proprietario del biglietto corrispondente

- getName: che dato un posto (settore,fila,posto) ritorna il nome del proprietario
- postiFree: che ritorna il numero di posti liberi insieme a un vettore che li contiene tutti come differenti istanze della classe Ticket
- incasso: che determina l'incasso totale determinato dalla vendita dei posti.

Aggiornamento fisico dei dati

Tutto questo per quanto riguarda l'interazione del database con l'applicazione che lo utilizza. Un problema a parte è dovuto all'aggiornamento della parte fisica della base di dati. Ricordiamo infatti che utilizziamo un'astrazione logica di una base di dati reali, ma che - anche per la nostra applicazione - è necessario avere una *pezza d'appoggio* fisica che quindi andrà aggiornata in base alle modifiche effettuate.

In questo caso l'astrazione scelta di realizzare il database fisico come un tradizionale file di testo (suffisso txt) non aiuta particolarmente. E' evidente infatti che l'accesso contemporaneo di molti client all'applicazione server rischierebbe di comportare una coda di aggiornamenti lunga da gestire e che comporta alcuni rischi, anche di consistenza dei dati. D'altra parte l'ipotesi di guasto singolo mette al riparo quasi da ogni possibile perdita dei dati a fronte di una caduta di un server.

Il database fisico e logico è infatti in copia duplicata (una per server) e la procedura di inizializzazione richiede -nel caso esista attivo un altro server- la copia remota del file fisico di database, risolvendo in tal modo l'eventuale problema di inconsistenza.

Il problema dell'aggiornamento fisico si riduce quindi a un puro problema di utilità e di facilità d'uso. Occorre realizzare una procedura che non blocchi l'elaborazione creando ritardi inaccettabili dal lato del server e, di conseguenza, dal lato client.

Si utilizza pertanto una variabile di stato chiamata **COMMIT**. Essa tiene conto del numero delle transazioni portate a buon fine sul database (il numero dei commit effettuati, appunto). Al termine di una transazione, tale variabile viene aggiornata e quindi verificata.

L'applicazione nasce scrivendo su file fisico ogni volta che una transazione è conclusa, ma l'esistenza di tale variabile permette di modificare la frequenza di aggiornamento nella maniera desiderata.

Naturalmente la scrittura fisica sarà parallela all'elaborazione del server che sfrutta la copia logica del database, sempre aggiornata.

Gestione della concorrenza

Una parte rilevante dei problemi da affrontare nella creazione dell'applicazione riguardano la gestione dell'informazione sui posti dello stadio. Essendo un dettaglio fondamentale per la vendita dell'abbonamento, lo stato reale di ognuno dei posti dello stadio va gestita in maniera **affidabile e esente da errori**. Il problema nasce dalla molteplicità dei client a fronte di due applicazioni server e va risolto creando una fase di trattativa fra i due server sull'assegnazione del posto. E' facile capire che in caso di richiesta di posti differenti da parte di due clienti differenti la trattativa è scontata e di facile risoluzione, sempre che il database sia mantenuto consistente.

Il problema diventa più complesso se due clienti fanno richiesta del medesimo posto. E' infatti possibile che entrambi lo trovino libero. Occorre quindi creare un protocollo di

comunicazione fra i due server che consenta ad uno solo di ottenere il privilegio di acquisto.

Ricordiamo che è stato convenuto di **non** creare un rapporto master-slave fra i due server, quindi nell'attribuzione non esiste un privilegio d'acquisto.

Si è ritenuto più efficace e soprattutto più giusto pensare l'acquisto remoto, fatto da un qualunque client su un qualunque server che fornisce il servizio, come parte di un'unica fila immaginaria allo stesso botteghino. Come nella realtà il momento dell'acquisto determina l'ampiezza della scelta (vale il detto *chi prima arriva meglio alloggia*), così nell'astrazione creata dall'applicazione risulta determinante l'istante di tempo in cui il cliente prenota il biglietto.

In caso di concorrenza la questione viene risolta assegnando la priorità a chi ha fatto prima la richiesta. In caso di perfetta concomitanza si procede a un democratico sorteggio. Tutto ciò, chiaramente, in maniera del tutto trasparente al cliente che richiede il servizio.

- **Struttura del Database FISICO**

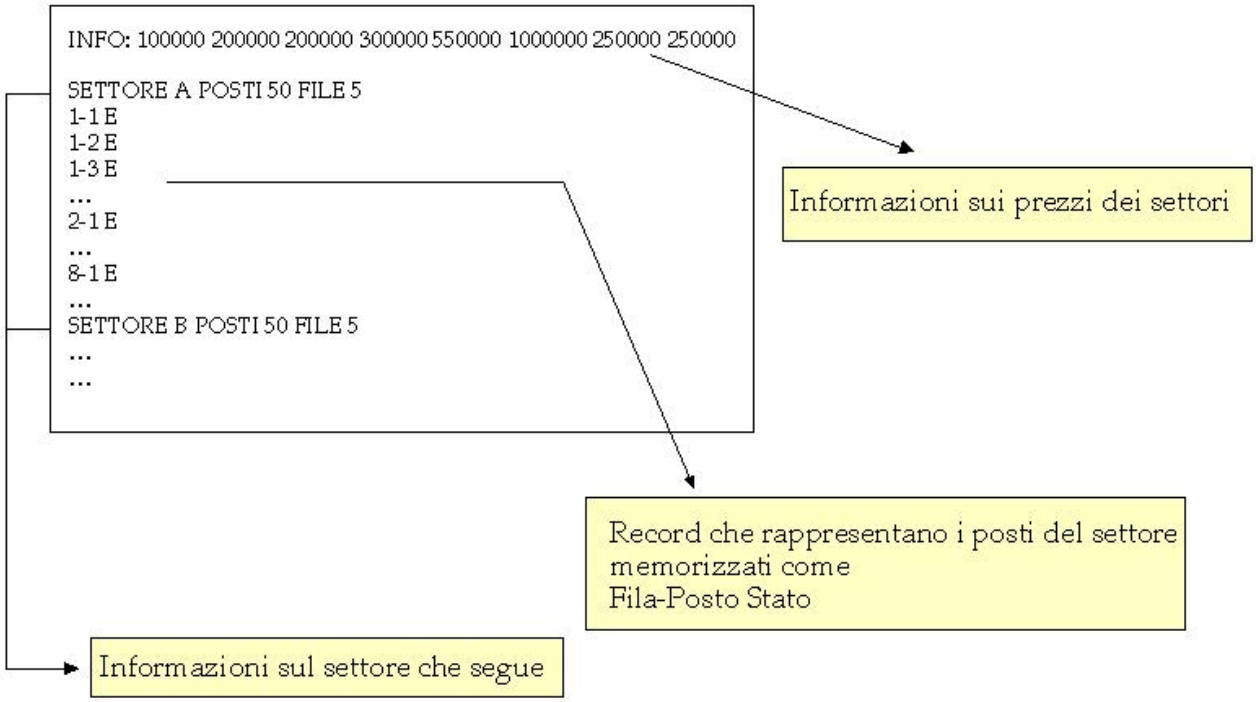
Diamo ora uno sguardo all'astrazione che abbiamo scelto per rappresentare il **database fisico**.

Precisiamo di nuovo che si tratta di una pura semplificazione dal momento che qualsiasi applicazione reale sfrutterebbe le caratteristiche di un database relazionale (ad es. MS Access o FileMaker per il mondo Macintosh) che andrebbe in qualche modo collegato alle necessità del nostro progetto (vd. sezione*Miglioramenti*). Tuttavia, per non complicare troppo le cose abbiamo scelto una semplificazione a puro file di testo della base di dati, meno efficace dal punto di vista economico, ma più snella da relazionare con l'applicazione creata dato che non richiede l'utilizzo di classi Java ad hoc per essere manipolata.

Il file di database fisico si chiama **database.txt** .

Ogni **record** è rappresentato da una linea del suddetto file e le informazioni necessarie alla creazione del database logico su cui opereremo sono contenute all'inizio del file e all'inizio di ogni settore secondo lo schema rappresentato nella figura della pagina successiva.

Come si vede all'inizio del file è presente una linea che indica il prezzo dei biglietti. Essa sarà necessaria all'avvio dell'applicazione client. Naturalmente deve essere possibile modificare l'importo di tali prezzi da parte di chi effettua la vendita (nel nostro caso il server). Di tale possibile modifica bisognerà tenere conto sia nella progettazione del server che in quella del client. Dal momento che il numero delle file che compongono ogni settore è variabile e sconosciuto al client, è necessaria la linea che precede le informazioni sui settori e che contiene, oltre al nome del settore stesso, anche il numero dei posti che lo compongono. Come sappiamo tale dato è superfluo, ma serve all'applicazione client per non essere appesantito di un ulteriore calcolo. I record che rappresentano le istanze della classe Ticket sono la parte successiva del file di database. Come si vede in figura ogni posto all'interno di un settore è rappresentato dal suo numero di fila e dal numero del posto, oltre naturalmente alla modalità che illustra se sia pieno, vuoto o in prenotazione.



E' chiaro che la presenza di un posto denominato P nel file di database fisico è un problema di non poco conto, dal momento che il salvataggio su disco della base di dati avviene solo al termine della transazione e quindi in una fase in cui il posto non può essere prenotato. Vedremo che la presenza di un posto P nel database fisico risulta da un problema nella fase di inizializzazione del server che può essere risolto con un opportuno protocollo di comunicazione.

Schema del funzionamento

Funzioni svolte dall'applicazione client

- **Inizializzazione dell'applicazione**
  - Autenticazione dell'utente locale e distinzione fra operatore e utente singolo
  - Lettura dei parametri di connessione contenuti nel file di preferenze ed eventuale ripristino del file stesso a seguito di problemi.

- **Creazione del collegamento**
  - Ricerca di un server disponibile
  - Creazione di un collegamento affidabile con il server
  - Autenticazione remota
  - Richiesta e ricezione dei parametri necessari alla partenza del servizio

- **Scelta del biglietto**
  - Selezione del settore desiderato
  - Ricezione della mappa del settore desiderato
  - Selezione del posto desiderato
  - Ricezione di conferma della disponibilità del posto scelto

4 **Pagamento**
  - Scelta del tipo di pagamento
  - Conferma dell'avvenuto pagamento
  - Conferma e chiusura della transazione

- **Chiusura del collegamento**
  - Comunicazione al server della chiusura del collegamento
  - Scelta da parte dell'utente se ripartire con una nuova prenotazione o chiudere l'applicazione

Ognuna di queste fasi richiede un colloquio con il server svolto tramite scambio di messaggi. Come detto, tale scambio deve essere affidabile e cronologicamente esatto.

Andranno gestiti in maniera efficiente i tempi di attesa e interpretate nella maniera adeguata le eventuali mancate risposte da parte del server.

Funzioni svolte dall'applicazione server

- **Inizializzazione**
  - Distinzione fra avvio come primo server o come secondo server e differenti procedure di avviamento
  - Attesa delle richieste provenienti da un client

Per risolvere le richieste proveniente dal client il server stesso deve svolgere diverse funzioni.

- **Accettazione della richiesta**
  - Creazione di un processo figlio che gestisca la richiesta del client
  - Autenticazione del client
  - Spedizione al client dei parametri necessari al suo svolgimento

- **Transazione**
  - Attesa e gestione della richiesta del settore
  - Attesa e gestione della richiesta del biglietto
  - Comunicazione dell'attesa di pagamento

- **Pagamento**
  - Gestione delle due differenti modalità di pagamento possibili
    - Se il cliente sceglie il pagamento via credit card gestione della transazione economica con il Bank Server

- **Chiusura del collegamento**
  - Attesa della ricezione da parte del client dell'avvenuta transazione
  - Chiusura del collegamento
  - Uccisione del processo che serviva il client

ServerThread

E' il processo che si **occupa di gestire le richieste proveniente dal client** ed è presente nell'applicazione in un numero di istanze pari al numero di client che accedono al servizio.

La generazione del thread è opera della classe ServerApp che rappresenta l'oggetto principale del lato server.

Alla chiamata del metodo run(), necessario per ogni thread Java, vengono creati flussi di comunicazione con il client e quindi si processa la richiesta ricevuta dall'esterno.

Esiste un metodo che si incarica di distinguere se la richiesta giunge da un server o da un client e che prende le necessarie decisioni per il proseguimento dell'applicazione. E' in

questo punto dell'elaborazione che si decide quindi se il processo è **master o slave**. Se la richiesta da processare giunge da un cliente il processo deve proseguire l'esecuzione gestendola e quindi si comporterà da *master* nei confronti degli altri server. Se la richiesta giunge da un altro server - per una qualunque necessità di informazioni - il processo si comporterà da *slave*.

La distinzione fra master e slave non è quindi definita a priori sull'identità del server ed **è relativa alla transazione**. Un server può svolgere contemporaneamente funzioni di master e di slave gestendo due transazioni diverse. In sintesi:

- **Master è il thread che gestisce direttamente la richiesta del client**
- **Slave è il thread che gestisce le richieste dell'altro server**

Entrambe le identità del server dovranno gestire le politiche di protezione dei dati e di mantenimento della consistenza.

Come master il thread gestisce tutta la transazione occupandosi di informare il suo omologo sullo stato della transazione stessa e svolgendo i passi necessari alla risoluzione della parte economica del pagamento, tramite colloquio con il Bank Server, in modo da lasciare tutto quanto trasparente al client che ne fa richiesta.

Come slave il thread risolve le richieste ricevute dall'altro server sullo stato dei posti, sulla procedura di inizializzazione, sullo stato della transazione, sulla necessità di aggiornamento dei posti o di abort della transazione.

Protocolli di comunicazione

Costituiscono la parte fondamentale dell'applicazione. Servono a definire un vocabolario comune a tutte le parti del progetto e determinano l'accuratezza dello scambio di messaggi (e quindi di informazione) necessari al corretto svolgimento di ogni parte della transazione.

Si tratta di tre protocolli differenti che gestiscono tre parti differenti della transazione:

1. **Protocollo di comunicazione server/server**: gestisce la parte che riguarda concorrenza e consistenza dell'informazione. Fondamentale per il corretto mantenimento della base di dati.
2. **Protocollo di comunicazione server/client**: gestisce la richiesta da parte del client. Fondamentale per il lato commerciale dell'applicazione, la vendita del biglietto all'utente finale.
3. **Protocollo di comunicazione server/Bank Server**: gestisce la transazione economica e il pagamento del biglietto via credit card.

Ognuno di essi si serve di un differente oggetto Java che lo gestisce. Con riferimento ai numeri citati sopra:

1. ServerThread/GestoreTransazione
2. ServerThread/Gestore (lato client)
3. BankAccess/BankServer

I colloqui fra le varie applicazioni avvengono tramite scambio di messaggi di testo (classe Java String) che vengono adeguatamente interpretati dal ricevente in funzione del momento cronologico in cui sono ricevuti e della funzione che devono svolgere.

Le funzioni svolte dai tre protocolli sono intrecciate cronologicamente e spesso dipendono l'una dall'altra. E' chiaro infatti che la chiusura della transazione con il client può dipendere dall'avvenuto pagamento del biglietto durante la transazione economica. Allo stesso modo la trasmissione di una conferma o di una disdetta di una prenotazione all'altro server dipenderà dalla conferma data dal cliente o dalla disponibilità locale o remota del posto.

Ad ogni modo, per una migliore e più semplice conviene vedere in dettaglio i tre protocolli.

Intrecciando la conoscenza dei protocolli con il funzionamento delle applicazioni client e server si ottiene una comprensione totale delle modalità di realizzazione del progetto.

Protocollo di comunicazione SERVER/CLIENT

E' il protocollo fondamentale.

Dalla comunicazione fra il server e il client infatti dipende tutto lo sviluppo dell'applicazione. Il progetto, in parole povere, altro non è che una richiesta di servizio da parte di un cliente ad un server che lo offre. All'interno di tale comunicazione dovranno poi trovare posto le relazioni fra i server e quella fra il server e il Bank Server. Ricordiamo inoltre che il protocollo di comunicazione fra client e server è l'unico che può esistere a sé stante, dal momento che non ha bisogno dell'esistenza degli altri due protocolli per essere portato a termine.

Analizziamolo nel dettaglio, partendo dalla situazione in cui il server, in ascolto in attesa di un client, riceve una richiesta di connessione e genera il thread che la serve. Occorre tenere presente che qui non vengono trattati i meccanismi di recupero da attuare in caso di errore in ricezione o di mancata spedizione dei messaggi. Tali procedure verranno trattate alla fine della descrizione dei protocolli.

- Il **client** spedisce un messaggio di **AC** (Access Client). Il **server** risponde con **OK** e si pone in attesa di ricevere.

- Il **client** spedisce in successione **user e password** necessarie all'autenticazione remota. Il **server** riceve i due messaggi e verifica l'attendibilità della richiesta. Se il client è autorizzato a entrare spedisce **OK** altrimenti spedisce **NO**. In caso di autenticazione eseguita il server si pone in attesa del client.

- Il **client,** gestita la sua elaborazione locale, spedisce un messaggio di ready **RDY**. Il **server** ricava dal database l'informazione sui prezzi dei settori e la spedisce al client. Quindi si ripone in attesa di ricevere.

- Il **client** spedisce al server la lettera che rappresenta il settore in cui desidera acquistare il biglietto. Il **server** estrae dal database la mappa del settore richiesto e la spedisce in un solo messaggio al client. Quindi si ripone in attesa.

- Il **client** spedisce al server l'informazione relativa al posto che desidera ricevere. Tale informazione è spedita in una stringa nel formato S.F-P (S: settore, F: fila, P:posto). Il **server** verifica sul proprio database se il posto è libero. In caso affermativo interroga il suo omologo. Se il posto è libero anche dall'altra parte, o se risolve positivamente la controversia sul possesso, spedisce al client il messaggio di attesa di pagamento **PAY** e si pone in attesa del client. In caso contrario spedisce il messaggio di rifiuto **NO** e chiude la connessione con il client.

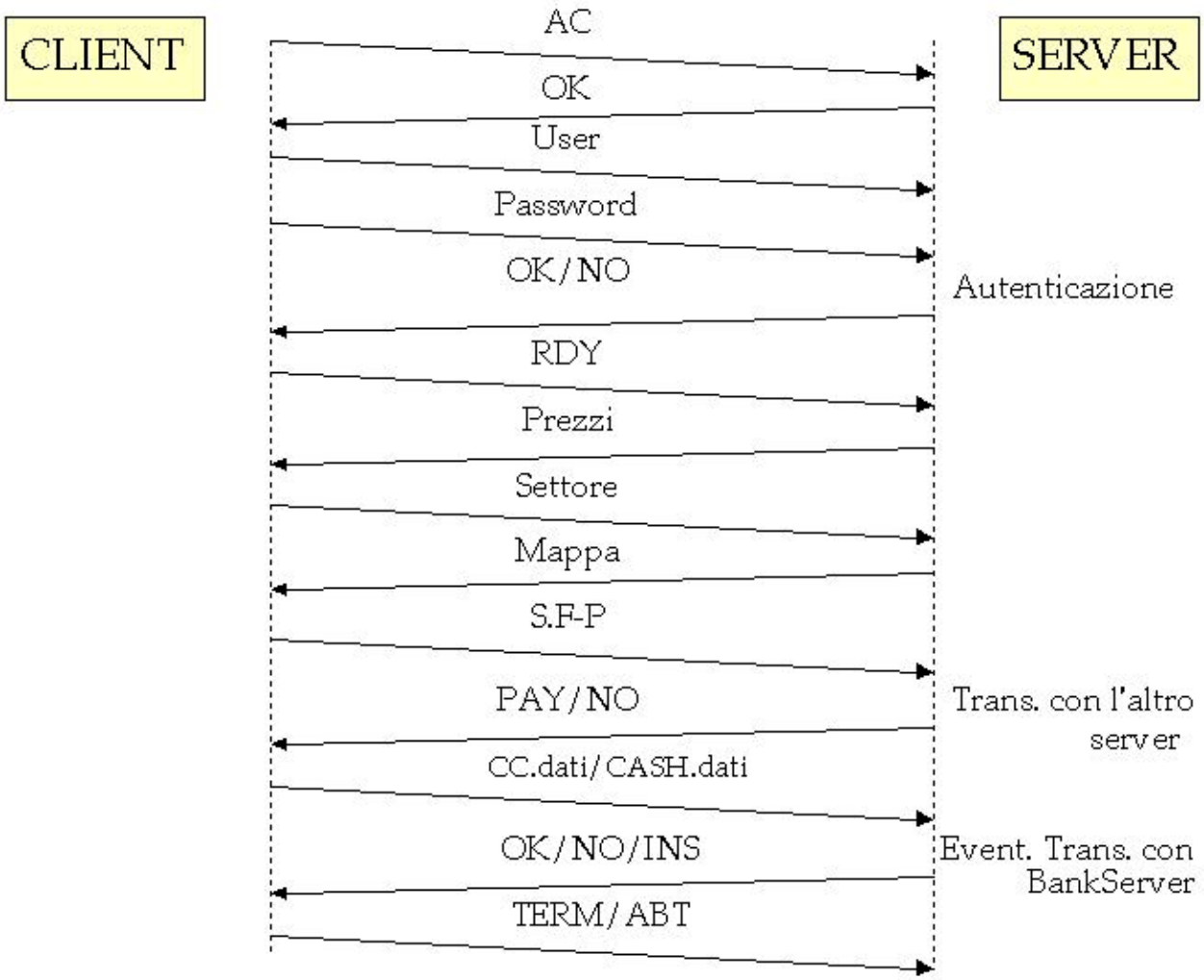
- Il **client** spedisce al server comunicazione del modo in cui vuole effettuare il pagamento. Il messaggio prevede due alternative:

    - **CASH.Nome#Cognome** se il pagamento è stato effettuato in contanti al botteghino in cui il client si trova. Il **server** preleva le informazioni anagrafiche le comunica all'altro server insieme alla conferma del posto, blocca il posto sul proprio

database logico, aggiorna il campo anagrafico del posto e spedisce conferma al client. Il **client** riceve la conferma e manda al server un messaggio di terminazione del collegamento **TERM** o di aborto della transazione **ABT**.
Nel *primo caso* il **server** conferma l'avvenuta terminazione della transazione al suo omologo, quindi chiude la transazione aggiornando il database fisico, chiudendo il collegamento con il client e uccidendo il processo che lo serviva.
In *caso di abort* il **server** comunica al proprio omologo che la transazione va annullata, libera di nuovo il posto sul proprio database logico, chiude la connessione con il cliente e uccide il processo che lo serviva.

- **CC.dati_sul_conto** se il pagamento va effettuato via credit card. Il **server** gestisce la transazione con il BankServer mediante la creazione dell'oggetto BankAccess. Se la transazione è positiva preleva le informazioni anagrafiche le comunica all'altro server insieme alla conferma del posto, blocca il posto sul proprio database logico, aggiorna il campo anagrafico del posto, spedisce conferma al client e avverte il BankServer che la transazione sta per concludersi positivamente. Il **client** riceve la conferma e manda al server un messaggio di terminazione del collegamento **TERM** o di aborto della transazione **ABT**.
  Nel *primo caso* il **server** conferma l'avvenuta terminazione della transazione al BankServer, al suo omologo, quindi chiude la transazione aggiornando il database fisico, chiudendo il collegamento con il client e uccidendo il processo che lo serviva.
  In *caso di abort* comunica che la transazione è saltata al suo omologo e al Bank Server, libera il posto sul proprio database logico, chiude la connessione con il cliente e uccide il processo che lo serviva.
  Se la transazione con il BankServer non si conclude positivamente il **server** comunica il problema al client mediante un messaggio (**NO** conto inesistente, **INS** credito insufficiente) e si pone di nuovo in ascolto per conoscere i nuovi termini di pagamento.

Lo scambio dei messaggi è schematizzato nella figura della pagina seguente.



La figura evidenzia un particolare già messo in risalto all'inizio di questa sezione. Il protocollo di comunicazione fra il client e il server è l'unico che può esistere da solo. Come si vede, infatti, la transazione con l'altro server è relativa alla ricerca delle informazioni riguardanti la disponibilità del posto scelto dal client e va in esecuzione solo nel caso in cui esistano due server.

D'altra parte l'esistenza della transazione con il BankServer dipende dalla scelta di pagamento fatta dal client.

Entrambi questi protocolli, di cui vedremo le specifiche, sono comunque utili per il completamento dello scambio di informazioni fra il client e il server e per ottenere un servizio più efficiente riguardo alla disponibilità del servizio e alla completezza del servizio stesso.

Protocollo di comunicazione SERVER/SERVER
Consistenza e Concorrenza

E' la parte del protocollo fra i server fondamentale per la consistenza dei dati e la concorrenza degli accessi al database dei posti. In questa sezione ci occuperemo di descrivere il metodo con cui i due server si scambiano informazioni sullo stato del posto durante tutta la transazione e le politiche con cui vengono risolti eventuali conflitti di competenza sullo stesso posto.

E' chiaro che questo tipo di comunicazione viene messa in atto solo in presenza di entrambi i server. E' quindi necessario un controllo preventivo che il chiamante fa dell'esistenza del suo omologo.

Pensiamo, con la solita terminologia, come **server master** colui che interroga e **server slave** chi la riceve e ricordiamo che dal lato del master la comunicazione è gestita dal GestoreTransazione mentre dal lato dello slave dallo stesso ServerThread.

Il **GestoreTransazione** è un oggetto Java che viene creato solo nel caso in cui il master riconosca la presenza dello slave. Serve al thread del server per gestire la transazione con il suo omologo e fornire le informazioni necessarie allo stato della vendita, allo scopo di tenere consistente ed aggiornato il database sullo slave. Contiene metodi necessari alla verifica del posto sul server slave, a richiedere la risoluzione della controversie sull'attribuzione del posto, a confermare l'avvenuto pagamento, a chiudere e ad abortire la transazione, oltre naturalmente alla sendMessage e alla receiveMessage.

Vediamo ora nel dettaglio le operazioni svolte, ricordando che il protocollo va in atto quando il client richiede al server di prenotare un posto, mandando il messaggio di S.F-P (vd protocollo server/client).

- Una volta creata la connessione **il master** spedisce un messaggio di AS (Access Server) a cui **lo slave** risponde con OK e si pone in ascolto.

- **Il master** spedisce **CHECK** per informare che desidera informazioni su un posto. **Lo slave** conferma di nuovo con un **OK** e si pone di nuovo in ascolto.

- **Il master** spedisce le indicazioni del posto di cui vuole informazioni nella consueta forma S.F-P. Lo **slave** va a verificare sul proprio database locale qual è lo stato del posto richiesto e spedisce al suo omologo:
  - **OK** se il posto è libero
  - **NO** se il posto è stato venduto
  - **OCC** se il posto è prenotato

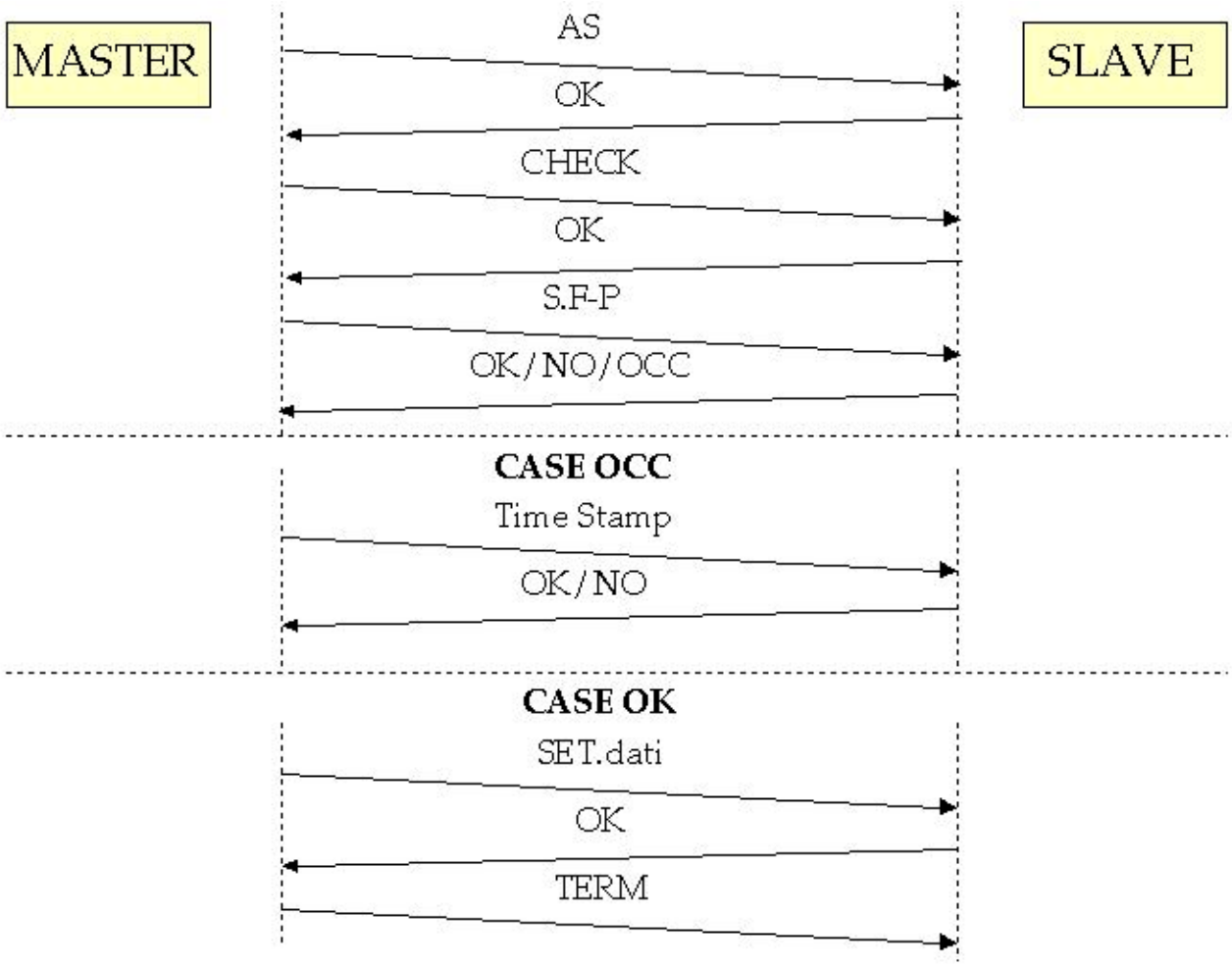
  Nel *secondo caso* la transazione è conclusa

  Nel *primo caso* **il master** manda l'informazione al client e attende comunicazioni sul metodi di pagamento. Una volta ricevuta la conferma del pagamento spedisce allo slave i dati del proprietario del posto nella forma **SET.dati**. **Lo slave** aggiorna il proprio database locale e spedisce al master un messaggio di **OK**, quindi si pone in attesa. Quando **il master** ha ricevuto conferma dal client spedisce allo slave il messaggio di

TERM che chiude la transazione. In ogni momento la transazione può essere abortita spedendo allo slave il messaggio di **ABT**.

Il caso in cui il posto sia allo stato di P anche sull'altro server denota l'esistenza di una richiesta contemporanea. Due client diversi sui due server hanno chiesto lo stesso posto in un istante confrontabile temporalmente. La controversia va quindi risolta.
Se il **master** riceve un messaggio di OCC recupera il campo dataPrenotaz del posto che ha scelto e lo spedisce allo **slave** che lo confronta con il medesimo campo del medesimo posto presente nel suo database e decide spedendo **OK** (il master ha precedenza sul posto) oppure **NO** (lo slave ha precedenza sul posto). In quest'ultimo caso la transazione è conclusa. Altrimenti lo slave libera il posto sul suo database e la comunicazione procede come se il posto fosse sempre stato libero sul server slave.

Lo schema in figura riassume le varie fasi.



Abbiamo così visto la prima delle due fasi che coinvolgono la comunicazione fra i server. Prima di vedere la seconda occorre soffermarsi un momento sulla parte relativa alla risoluzione della controversia sulla priorità della prenotazione.

- **Gestione della concorrenza, timestamp**

Per garantire la consistenza dei dati e la concorrenza dell'accesso occorre la procedura di sincronizzazione che abbiamo illustrato precedentemente. Tale procedura deve garantire ad ogni client un accesso giusto e corretto alla risorsa stadio. Ciò significa che deve essere applicata una politica di uguaglianza fra i vari client che assegni il posto a chi ne ha realmente diritto secondo una politica che va determinata a priori.

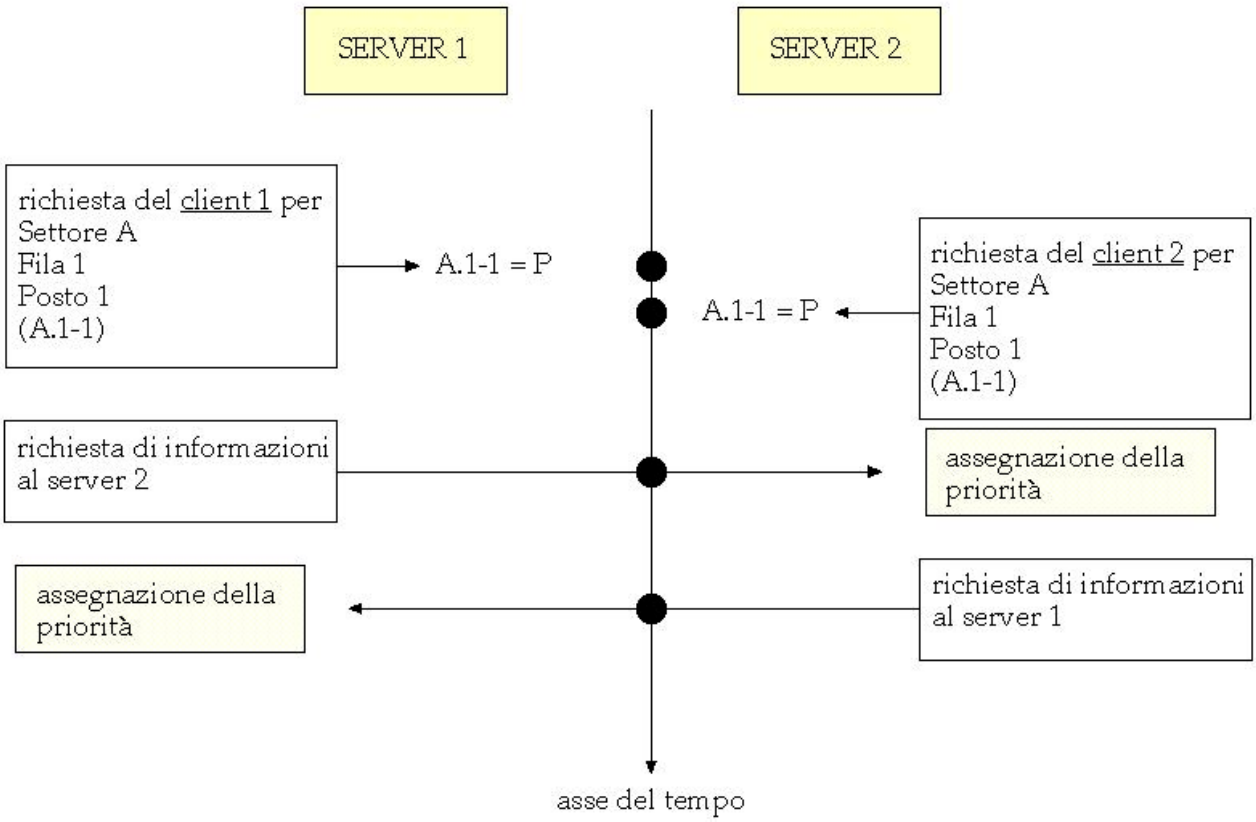
Abbiamo deciso, come indicato nei requisiti di sistema, di trattare la vendita come eseguita su un'unica fila a un unico botteghino. Il momento dell'acquisto é perciò il fattore determinante. Chiaramente la replicazione delle risorse obbliga al chatting fra i server per fare in modo che ognuno di essi possieda lo stesso bagaglio di informazioni.

In quest'ambito occorre tenere presente che due client connessi a due server differenti potrebbero trovare contemporaneamente un posto prenotato (N.B. non occupato - F - ma prenotato - P -) nello stesso momento. Questo problema, uno dei più spinosi da risolvere, si verifica quando la seconda prenotazione avviene prima che il server slave sia informato della prima.

Sappiamo infatti dalla procedura illustrata sopra che una volta prenotato il posto sul suo database, il server master chiede allo slave notizie del posto che gli è stato richiesto. Se nel lasso di tempo che intercorre fra le due azioni sullo slave viene fatta una prenotazione per il medesimo posto, l'abbonamento in questione sarà allo stato di P su entrambi i server. E' chiaro che bisogna dirimere la questione.

Si è scelto di far valere il detto *chi prima arriva meglio alloggia* e quindi a ogni ticket è associato un valore temporale che identifica il momento in cui la prenotazione è stata fatta. Il master lo spedisce allo slave che confrontandolo con il valore contenuto nel ticket del suo database decide se ha o no la priorità del posto. E' chiaro che in una controversia del genere la procedura si svolge contemporaneamente su entrambi i server. Ciò è dovuto al fatto che ogni thread non conosce durante la sua esecuzione l'attività dei *fratelli* presenti insieme al suo nell'applicazione server.

Vediamo una figura per capire meglio come funziona.



Come si vede chiaramente la procedura di decisione sulla priorità scatta in entrambi i server (nella loro parte slave), ma non vi sono problemi di conflitto di competenza, poiché la decisione che prenderanno entrambi sarà la stessa, dal momento che sono gli stessi i campi dataPrenotaz che vengono confrontati. In questo caso la ridondanza del codice è evidente, ma la presenza della doppia operazione permette di non dover creare complessi meccanismi interni di comunicazione fra le varie parti del server e di garantire l'efficienza e la consistenza dei dati in possesso di entrambe le applicazioni server.

Protocollo di comunicazione SERVER/SERVER
Inizializzazione del server

Non meno importante è la procedura con cui il server si inizializza.

E' chiaro che dal momento che l'unico vincolo che abbiamo posto è "l'esistenza in vita" di almeno un server, l'ingresso del secondo server può avvenire in qualunque momento. E' importante quindi che ogni nuova risorsa che entra a far parte del progetto abbia le stesse caratteristiche informative di quella già esistente.

E' altresì importante che il primo server che si avvia possa creare le condizioni per lo svolgimento del servizio e quindi il database dei posti. Occorre perciò che al suo avvio il server sappia se è il primo o se qualcun altro è già avviato da qualche parte.

Ciò viene facilmente determinato tentando un collegamento con il suo omologo, il cui indirizzo è contenuto nel file di preferenze del server. In caso di mancata risposta si conclude per una situazione di solitudine e scatta la procedura di inizio del servizio.

Se invece esiste in linea un altro server, il servizio può essere già iniziato e occorre una procedura che metta il nuovo arrivato al passo con chi già c'era.

Vediamo di cosa si tratta.

Per spiegare la procedura ci riferiremo ai server come **nuovo e vecchio** con evidente significato dei termini.

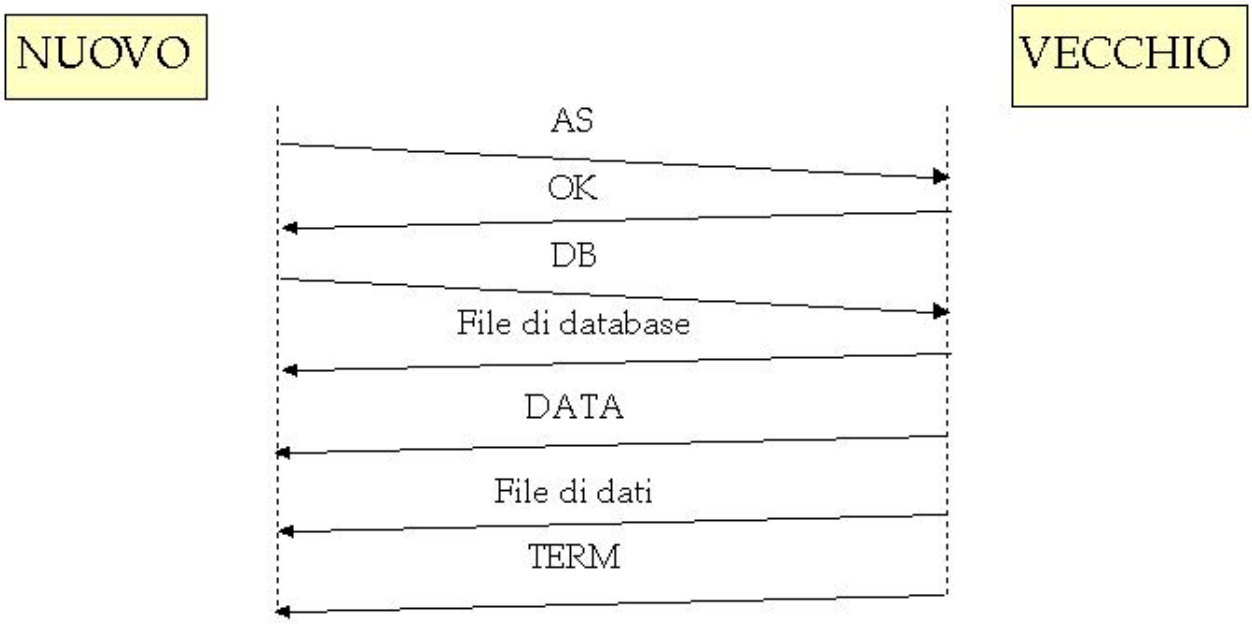
- Il **nuovo** server spedisce **AS** (Access Server) al **vecchio** server che risponde con **OK** e si mette in attesa della richiesta

- Il **nuovo** server spedisce il messaggio **DB** per indicare che richiede il database del vecchio server. Il **vecchio** server preleva le informazioni dal database logico (dal momento che è il più aggiornato) e le spedisce al nuovo server nella forma che avrebbero se si trattasse di database fisico. In pratica dal vettore di oggetti Ticket si prelevano i campi e li si trasformano in stringhe da spedire al nuovo server come se fossero le linee del file *database.txt* . Si è scelta questa procedura per poter inviare a chi ne fa richiesta la versione più aggiornata del database e in maniera tale che possa direttamente scriverla sul suo file di database fisico senza ulteriori elaborazioni.

- Il **nuovo** server riceve le informazioni e le salva sul file *database.txt* che ha appena creato

- A conclusione della spedizione del file di database, il **vecchio** server verifica se possiede il file anagrafico *dati.txt* .Se esiste spedisce al nuovo server il messaggio **DATA** per indicare la fine della trasmissione del database e l'inizio della trasmissione del file anagrafico con i nomi dei posti venduti associati a quelli dei proprietari. Il **nuovo** server riceve il file e lo salva nella stessa modalità con cui ha creato il precedente.
  Se il file anagrafico non esiste (nessuna transazione conclusa, o inizializzazione quasi contemporanea dei due server) il **vecchio** server spedisce il messaggio di **TERM** a indicare la chiusura della comunicazione.

La procedura è indicata nella figura che segue.



Dal file di database e da quello di dati l'inizializzazione del server procede a ricavare le informazioni necessarie alla creazione del vettore di database formato dalle varie istanze della classe Ticket.

A questo punto è perciò necessario focalizzare la nostra attenzione su un nuovo oggetto, la classe **serverInit** che si occupa dell'inizializzazione del server e che quindi è responsabile, dal lato del server nuovo, del protocollo appena descritto.

- **Classe serverInit**

E' l'oggetto che si occupa dell'inizializzazione del server.

Come per il client, anche qui ci si serve di un file di preferenze chiamato *serverPref.txt* in cui sono contenuti i campi indirizzo e porta del server omologo e del Bank Server. Essi vengono letti all'avvio e memorizzati in opportuni campi che poi verranno letti dal ServerThread quando ne avrà bisogno. Stabilita l'identità del server da contattare la serverInit stabilsce la connessione ed esegue il protocollo visto sopra.

Nel caso di solitudine e quindi di inizializzazione del servizio occorre creare ex-novo il file di database. Per fare ciò le informazioni necessarie sono il prezzo dei biglietti e il numero di file che compongono ogni settore. Tali informazioni sono contenute in un ulteriore file di preferenze denominato *preferences.txt*. Letto tale file la serverInit crea il database fisico da cui poi verrà creato quello logico. Trattandosi di inizializzazione del servizio non viene creato il file di dati.

Chiaramente - anche qui in perfetta analogia con il client - è necessaria una politica di prevenzione contro eventuali errori contenuti in uno dei due file di preferenze. In caso di eccezione occorsa durante la lettura di uno dei due, il file viene riscritto utilizzando parametri di default noti alla serverInit.

- **Problemi derivati dall'inizializzazione**

Avvengono solamente se il server che si inizializza non è solo.

Dal momento che il nuovo server riceve da quello già esistente la copia del database logico è probabile che alcuni dei posti che gli verranno inviati si trovino allo stato di prenotazione. Ciò è dovuto al fatto che l'invio delle informazioni al server che si inizializza può avvenire contemporaneamente a transazioni e quindi in un istante in cui le transazioni stese non si siano concluse.

Il database creato si trova quindi ad avere dei biglietti prenotati di cui non conosce il destino ultimo, in una situazione di grave inconsistenza.

Per eliminare tale problema viene creato un nuovo oggetto, la classe **updateObj**.

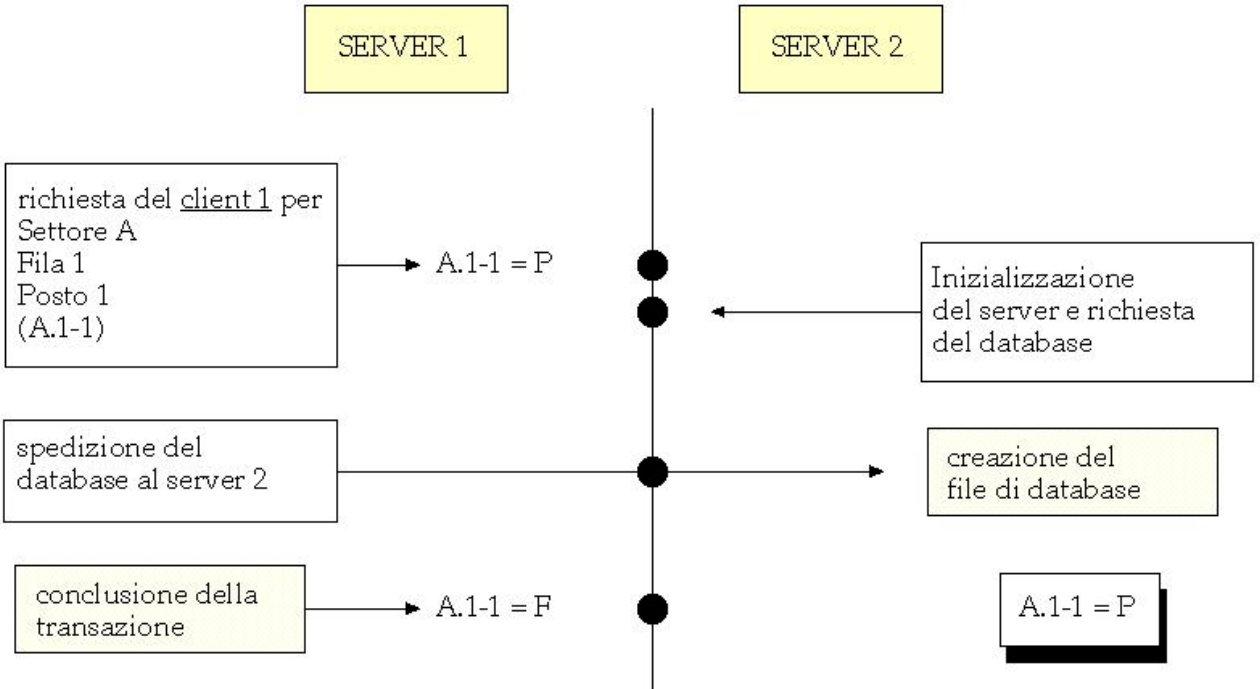
Tale classe viene interfacciata con l'oggetto Database. Quest'ultimo, durante la creazione del database logico, aggiorna il campo UPD dell'oggetto updateObj ogni volta che incontra un posto denominato con la P.

Conclusa la creazione del database logico, il campo UPD viene verificato e se non è nullo si avvia un preciso protocollo che tenta di rendere consistente l'informazione contenuta nella nuova base di dati. Tutto ciò chiaramente poiché se un nuovo server si aggiunge mentre il vecchio sta svolgendo una transazione, quest'ultimo non ne registrerà la presenza che alla transazione successiva e quindi non si preoccuperà di comunicare gli aggiornamenti delle transazioni in corso.

Per risolvere questo problema viene quindi in aiuto il **protocollo di update**.

Protocollo di comunicazione SERVER/SERVER Update

Elimina le inconsistenze dovute alla trasmissione al nuovo server dei dati ancora non consistenti riferiti alle transazioni in corso sul server vecchio. La figura qui sotto illustra il



problema.

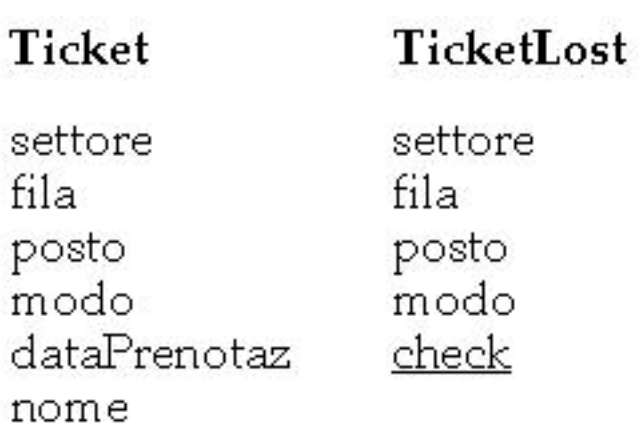
Come si vede la conclusione della transazione non viene comunicata al nuovo server che si è aggiunto al sistema. Così avverrebbe per ogni transazione in corso durante la trasmissione del file di database.

La scelta alternativa era quella di spedire il database fisico, ma l'inconsistenza creata sarebbe stata ancora più pericolosa. Infatti il posto A.1-1 indicato nell'esempio e oggetto della transazione, risulta libero (E) sul database fisico fino al commit finale. Ci troveremmo quindi in una situazione in cui il server vecchio vende un posto che risulta libero sul nuovo server.

Una possibile soluzione è rappresentata dalla costruzione di un link fra server nuovo e server vecchio durante la transazione. Non è tuttavia possibile prevedere il momento in cui il nuovo server *nasce* e sarebbe necessario una sorta di attesa attiva da parte del server vecchio che testa a ogni passo della transazione l'esistenza del suo omologo, con conseguente perdita di efficienza nella consegna dei risultati al client e, in buona parte dei casi, sovraccarico inutile del codice.

La soluzione pensata con la creazione dell'oggetto **updateObj** effettua un aggiornamento mirato e ricorsivo.

Durante la creazione del database logico, l'oggetto Database si incarica di aggiornare il campo UPD dell'oggetto updateObj per tenere conto del numero dei posti pendenti. Per ognuno di tali posti viene creato un posto in un apposito vettore, utilizzando una modifica della classe Ticket, denominata **TicketLost**. Come evidenziato in figura le informazioni fondamentali sul posto vengono confermate, mentre viene aggiunto un campo intero **check**, inizialmente azzerato.Il vettore creato dall'oggetto Database, parallelamente all'aggiornamento di UPD, contiene un'istanza della classe TicketLost per ogni biglietto ricevuto allo stato di prenotazione P.



La procedura di update svolge quindi un'interrogazione mirata sullo stato dei posti contenuti nel vettore, secondo un metodo che ora vediamo nel dettaglio, di cui s'incarica l'oggetto **updateObj.** Ci riferiremo ai due server ancora con i termini di vecchio e nuovo.

- Viene creata la connessione e quindi il **nuovo** richiede l'accesso come server (messaggio **AS**) a cui il **vecchio** risponde con **OK** e si pone in attesa.

- Il **nuovo** manda richiesta di update (messaggio **UPD**) e il **vecchio** risponde di nuovo confermando (messaggio **OK**) e si pone in attesa.

- Il **nuovo** cicla sul vettore dei posti persi e per ogni istanza manda un messaggio che contiene settore, fila e posto (S.F-P) del posto di cui richiede lo stato. Il **vecchio** risponde con lo stato del posto sul suo database.
  Se è consistente (modo E o modo F) il **nuovo** aggiorna il proprio database e rimuove il posto dal vettore dei posti persi. Se il posto è F il nuovo riceve insieme allo stato anche il nome del proprietario.

Se non è consistente (ancora modo P) il **nuovo** incrementa il valore della variabile check per tenere nota del numero dei tentativi effettuati.

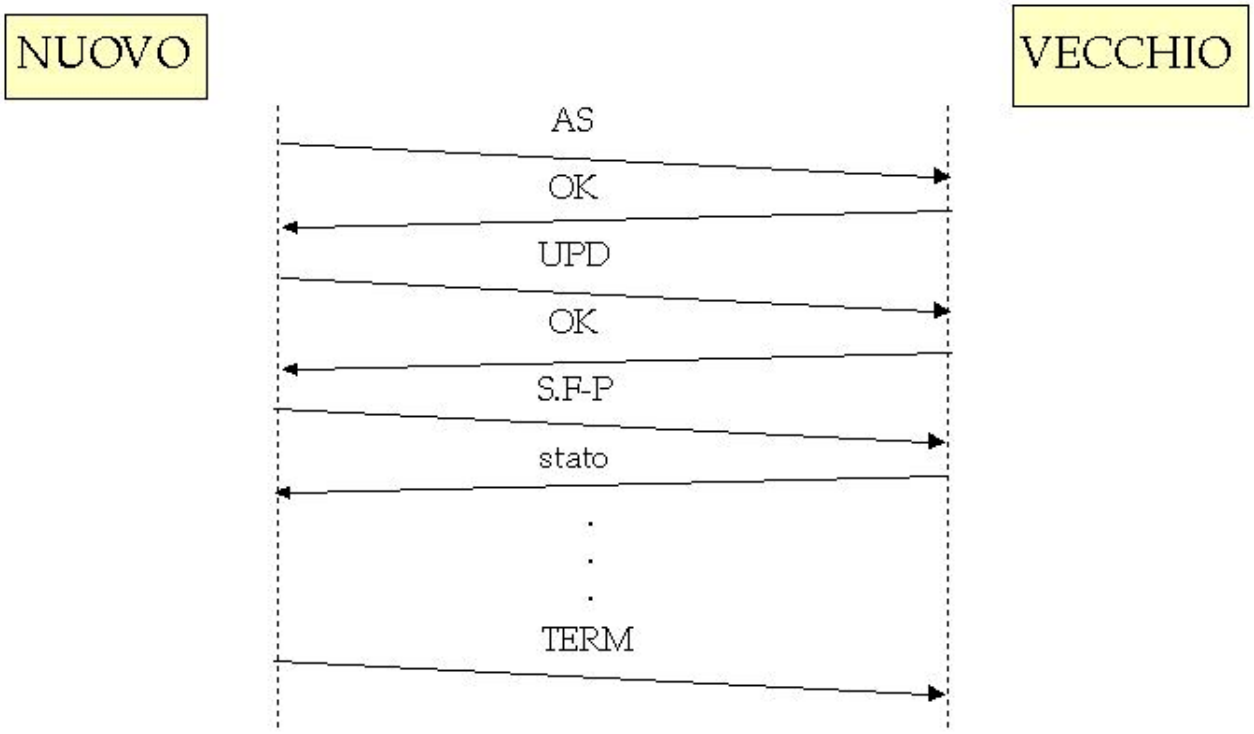
- Quando sono stati controllati tutti i posti viene aggiornata la variabile UPD e spedito dal **nuovo** al **vecchio** la chiusura della connessione (messaggio **TERM**).

Tale procedura di update viene effettuata (chiaramente se necessaria) una prima volta all'inizializzazione del nuovo server e quindi ogni volta che viene aggiornato il database fisico, a fronte della chiusura di una transazione, sia essa stata effettuata direttamente (in condizione di server master) o indirettamente (in condizione di server slave).

Data la durata media di una transazione si può affermare con buona certezza che in pochissimi cicli il database dei due server è di nuovo perfettamente identico.

Il significato della variabile check e del suo aggiornamento sarà spiegato quando ci occuperemo del recupero dai crash.

Il protocollo è illustrato graficamente nella figura che segue.



Protocollo di comunicazione SERVER/BANK SERVER

E' l'ultimo dei protocolli e il più semplice. Gestisce la transazione finanziaria necessaria al pagamento con carta di credito tramite il BankThreadr (dal lato del BankServer) e il BankAccess (dal lato del server che fa richiesta).

Il protocollo è chiaramente strettamente legato a quello tra server e client dal momento che non fa altro che fornire un servizio aggiuntivo all'acquirente.

Vediamolo nel dettaglio.

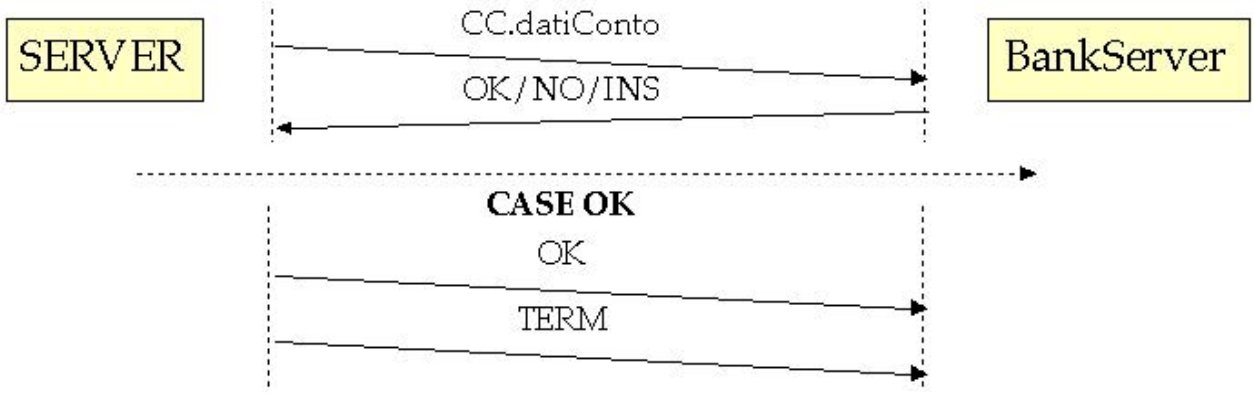
- Il **BankServer** accetta il collegamento del server e si pone in ascolto.

- Il**server** invia i dati del conto corrente come li ha ricevuti dal cliente(messaggio CC.Numero#Nome$Cognome@pwd&cifraDaPrelevare) e il **BankServer** preleva le informazioni e si accinge a fare il prelievo.

  Se il prelievo è stato effettuato restituisce la conferma (messaggio OK) e attende il server. Questi gli manda comunicazione che la transazione sta andando avanti positivamente (messaggio OK) e quando si è conclusa spedisce al BankServer la chiusura del collegamento (messaggio TERM).

  Se invece il prelievo non è stato effettuato il **BankServer** restituisce il problema occorso (messaggio **INS** o **NO**) e chiude il collegamento.

  Il protocollo è illustrato nella figura che segue.



Come si vede bene dalla figura - e come era stato illustrato sopra - il BankServer riceve una doppia conferma dal server a fronte di una transazione positiva. Questa scelta è dovuta alla necessità di recuperare nella maniera più efficace possibile un eventuale problema avvenuto durante gli altri protocolli che rischierebbe di guastare in maniera traumatica la transazione finanziaria. Si pensi ad un prelievo sul conto senza nessun acquisto del biglietto e si immaginino le conseguenze…

Risoluzione dei problemi e recupero dai guasti

Uno degli obiettivi del nostro progetto è garantire una certa qualità del servizio a fronte dei problemi che possono nascere durante lo svolgimento dell'applicazione. Occorre perciò decidere quali politiche implementare per evitare al massimo i guasti (approccio pessimistico) e poter reagire a fronte di situazioni inaspettate (approccio ottimistico). Vedremo che una miscela delle due azioni garantirà lo svolgimento del servizio nella quasi totalità dei casi e descriveremo per quanto possibile nel dettaglio l'unico caso di inconsistenza che la progettazione a due server può mettere in gioco. Si vedrà che è un limite dovuto in parte al basso numero dei server, ma soprattutto a un non corretto - per uso di risorse non ad hoc - interfacciamento con il database.

Ricordiamo che per ipotesi abbiamo definito tra le condizioni di lavoro il **guasto singolo**, il che implica che il tempo di identificazione e di recovery del guasto sia inferiore al tempo che intercorre fra due guasti. Il Bank Server, inoltre, non *crasha* mai.

- **Comportamento del client**

Come punto di partenza ci poniamo nella situazione utente, alle prese con l'applicazione client in una qualsiasi delle sue fasi. Ricordiamo che il client può servire anche un utente singolo e non solo fare da tramite tra il pubblico e il botteghino e quindi occorre progettare una semplicità d'uso che oltre a guidare l'acquirente passo passo, lo porti fuori senza particolari difficoltà dalle *sabbie mobili* di un possibile crash.

I **problemi** che possono incorrere durante l'applicazione sono fondamentalmente di due tipi: la *caduta del server* a cui l'applicazione è collegata o il ricevimento di *un messaggio non appartenente al vocabolario* stabilito. Quest'ultima operazione, data la progettazione del server, rappresenta comunque un problema incorso nella chat con il server e viene quindi trattato come se fosse un problema di comunicazione.

In ogni caso si attua una politica unica: **in caso di fallimento del server si procede al restart del servizio al client.**

Verrà effettuato un nuovo collegamento alla ricerca di un nuovo server - che in caso di ripristino può anche essere lo stesso - e si effettuerà una nuova transazione. Il problema del mantenimento della consistenza dell'informazione è naturalmente del tutto a carico del server.

- **Comportamento del server**

Dal lato del server la politica è più complessa dal momento che il server mantiene contatti con tre parti: i client, il suo omologo e il Bank Server. Con il Bank Server il problema si risolve da solo grazie delle ipotesi di lavoro.

Per quanto riguarda il link fra i due server occorre vedere chi dei due *tiene il coltello dalla parte del manico*. Sappiamo infatti che il server, durante una transazione, può trovarsi in condizione di master o di slave e quindi occorrerà vedere come comportarsi nelle due

situazioni. In maniera del tutto analoga il comportamento da tenere dipenderà dal momento della transazione in cui avviene il crash del server omologo. Tutto ciò per garantire la consistenza dei dati.

Vediamo i vari casi.

1. Crash del Master server e comportamento dello slave server

Momento in cui occorre il guasto	Meccanismo di recovery
Durante la richiesta di informazione sullo stato del posto, prima della ricezione del messaggio di SET	Chiusura del collegamento
Dopo che il posto è stato settato a P e prima della conferma della chiusura della transazione	Ripristino della condizione iniziale (posto settato di nuovo a E) e chiusura del collegamento
Durante la chat per la priorità dell'assegnamento prima della ricezione del messaggio di SET	Ripristino della condizione iniziale (posto settato di nuovo a E) e chiusura del collegamento
In ogni tipo di transazione, tra la ricezione del messaggio di SET e la chiusura della transazione	Ripristino della condizione iniziale (posto settato di nuovo a E) e chiusura del collegamento

- **Crash dello slave server e comportamenti del Master server**

Ricordiamo che lo slave server entra in gioco nella transazione dopo che il cliente ha comunicato al master la scelta del posto che ha effettuato.

Qui la gestione degli eventi è molto più semplice dato che, in fondo, il master è totalmente disinteressato al crash dello slave. Nel caso in cui lo slave stesso cada durante la transazione sarà compito dello slave stesso ripristinare lo stato delle proprie informazioni al riavvio, senza nessuna spesa aggiuntiva da parte del master che già ha il controllo della transazione. Quindi: **in caso di caduta dello slave il master si comporta esattamente come se fosse solo, ignorando la presenza dell'altro.**

- **Comportamento del Bank Server**

Abbiamo visto nel protocollo di comunicazione con il Bank Server che il server conferma la chiusura della transazione al Bank Server in due fasi. E' evidente che dopo la seconda conferma non c'è più nulla da fare e comunque il meccanismo garantisce che dopo la

seconda conferma il cliente abbia ricevuto anch'esso la conferma della vendita e quindi il biglietto si debba ritenere a tutti gli effetti emesso.

Nel caso in cui la transazione venga abortita prima a seguito del crash del server che la gestiva, il Bank Server ha tenuto traccia dell'onere economico e ripristina lo stato del conto corrente in oggetto riportando il saldo al valore corretto.

- **Comportamento durante l'inizializzazione**

Non consideriamo possibile la caduta del server che spedisce le informazioni al nuovo dal momento che nelle ipotesi abbiamo inserito il guasto singolo e la presenza di almeno un server sempre attivo.

- **In conclusione…**

Sommando quanto visto fin qui con la descrizione fatta dell'updateObj si vede come in ogni caso è garantita la consistenza e la concorrenza nella gestione della nostra applicazione. Resta da trattare il caso particolare (l'unico) che porta a una forma di inconsistenza non gestibile con i mezzi scelti per l'applicazione. Per poterla risolvere occorrerebbero sistemi per la gestione di basi di dati più complessi e non trattati qui (ne discuteremo nei *Miglioramenti*) e che comporterebbero l'interfacciamento della nostra applicazione con un database relazionale, ad esempio MS Access.

- **Crash non recuperabile**

E' l'unico caso di possibile inconsistenza e deriva in parte dall'interfacciamento con il database e in parte dal numero ridotto dei server progettati.

Per descriverlo partiamo dalla successione temporale delle conferme che il master Server dà alla chiusura della transazione, nell'**ipotesi** che sia attivo in quel momento **un solo server**.

1.Conferma sul proprio database locale (settaggio del posto a F)
2.Conferma al client con attesa di conferma
3.Update fisico

Non consideriamo la transazione finanziaria perché non necessaria per la nostra trattazione.

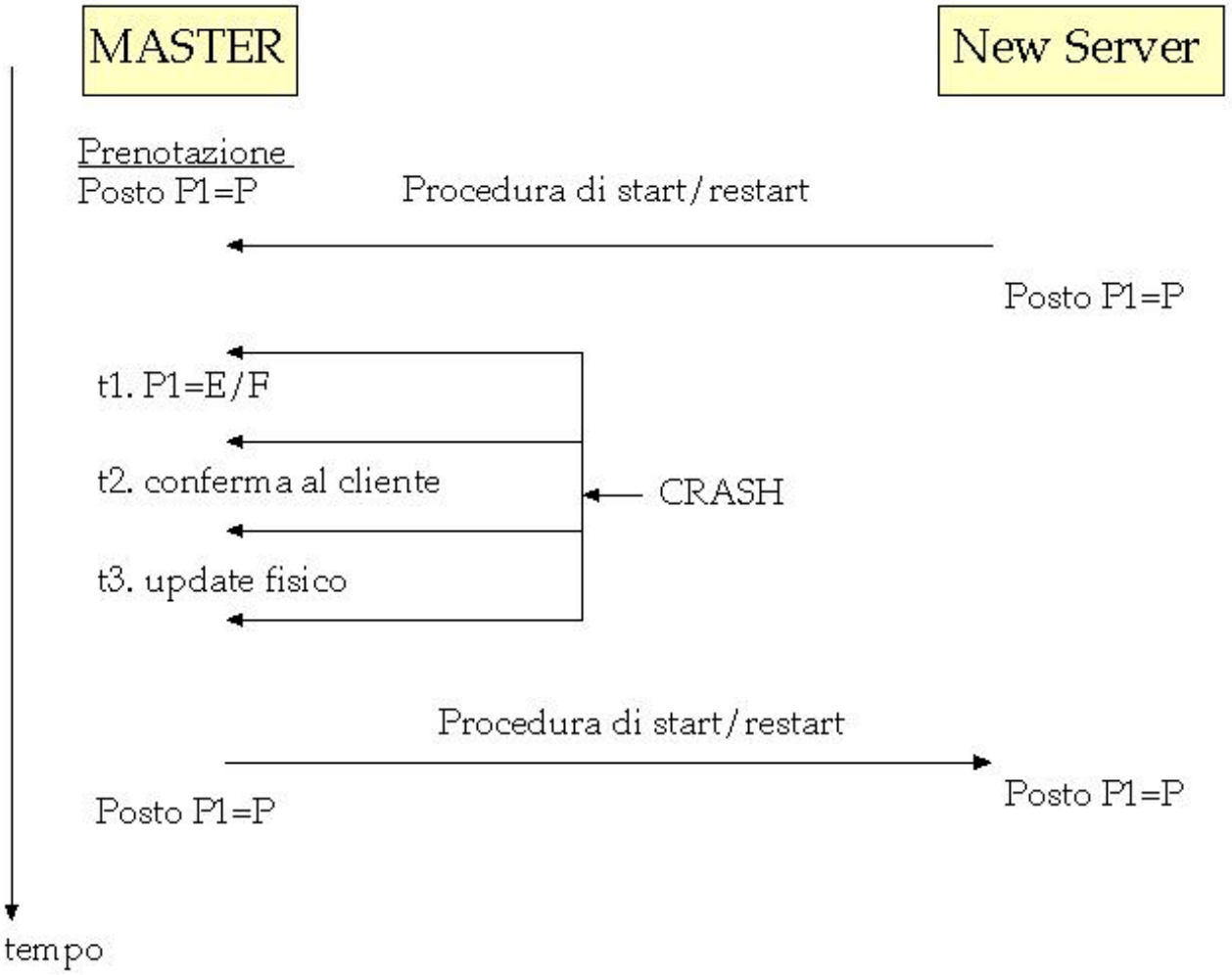
Supponiamo ora che dopo la prenotazione (settaggio del posto a P), cioè in un istante antecedente a 1. un altro server si aggiunga al nostro sistema. Ricevendo il database del Master server avrà il posto prenotato (P) anche sul proprio database locale. Se la transazione del master server va a buon fine non ci saranno problemi dal momento che nella procedura di update il nuovo server leggerà lo stato nuovo del posto (sia esso E oppure F).

Se il Master server cade possono insorgere problemi.

Se il crash avviene dopo il tempo 1. e il nuovo server ha già fatto una procedura di update, la consistenza sarà garantita dalla presenza del dato aggiornato sul secondo server e tutto sarà sistemato.

Se il master server cade senza che il nuovo server abbia modificato lo stato P del posto ricevuto, esso stesso lo invierà di nuovo alla Master server durante la sua procedura di restart e lo stato reale del posto sarà perduto.

Vediamo una figura per illustrare meglio questo percorso.



Come si vede, in mancanza di procedura di update del new server lo stato del posto resta inconsistente.

Per questo motivo l'oggetto updateObj teneva il conto con una variabile del numero di volte in cui tentava di risolvere l'eventuale inconsistenza di uno dei suoi posti, cercando un infinito teorico che gli facesse capire quando non fosse più il caso di provare.

Al raggiungimento dell'infinito teorico non viene più tentato l'aggiornamento del posto e lo si lascia a P, preferendo **non vendere un posto libero che vendere di nuovo un posto occupato.**

Un possibile rimedio a tale situazione sarebbe la presenza di più server attivi e sarà presa in considerazione nei *Miglioramenti.*

E' chiaro che il posto potrebbe essere prenotato (stato P) a causa di una transazione in corso del new Server: la procedura di update su quello che abbiamo indicato come Master server ripristinerà dopo un breve transitorio la situazione corretta.

Sicurezza

E' il tallone d'Achille dell'applicazione, se per sicurezza si intende la trasmissione di dati sicura attraverso la Rete.

Non è prevista nessuna forma di crittografia dei dati e se per alcune informazioni ciò può essere irrilevante (a chi può interessare sapere che posto sto acquistando ? e in che modo può danneggiarmi chi altri lo sappiano?) per altre non lo è affatto. Esiste infatti una possibilità di pagamento con carta di credito con conseguente passaggio di informazioni riservate sui numeri di conto e di carta che sono a disposizione di un'eventuale intrusione sulla linea di comunicazione.

L'unica procedura di controllo che viene eseguita riguarda la fornitura dell'accesso e i diritti di accedere al servizio stesso, verificati tramite user e password. Le stesse user e password sono peraltro trasmesse in chiaro lungo la rete e quindi si tratta di un'operazione di sicurezza piuttosto effimera.

Vedremo nella successiva sezione *Miglioramenti* quali procedure sarebbe più sicure attuare per garantire la sicurezza del nostro sistema.

Miglioramenti

Trattandosi di un progetto e non di un'applicazione di tipo commerciale, il nostro sistema possiede dei limiti piuttosto precisi soprattutto dovuti alla semplificazione fatta sull'uso del database e sul numero dei server.

I miglioramenti che possono essere apportati per rendere l'applicazione più reale e meglio funzionale vanno soprattutto in tre direzioni:

- Maggior numero di server
- Diversa interfaccia con il database
- Sicurezza

Vediamo di affrontare tutte e tre le questioni nel dettaglio.

Gestione dei server

Nella parte relativa alla risoluzione dei problemi abbiamo visto l'esistenza di una possibile situazione di inconsistenza non recuperabile, in cui anche il protocollo di update che funziona in maniera efficace nella totalità degli altri casi, risulta inutile. Tale situazione è in parte dovuta all'esiguo numero dei server che si è deciso di utilizzare.

Si può dire che anche in un sistema client/server come in un normale sistema commerciale, una delle leggi che dominano sia quella della domanda e dell'offerta, nel senso che è necessario essere sempre pronti a soddisfare le richieste dei client in maniera esauriente. Tutto ciò, naturalmente, incide anche sul numero dei server che si devono utilizzare. Tuttavia, nel nostro tipo di transazioni, occorre tenere presente anche che si deve garantire una continuità di servizio e equiparare le risorse disponibili per tutti i server, in modo che siano sempre le stesse nello stesso istante. Non è sufficiente infatti fornire un servizio, ma occorre gestirlo.

Per questo motivo la struttura a due server è da un lato efficiente dal punto di vista dello svolgimento del servizio, ma dall'altro risulta limitata per un'eventuale necessità di ampliamento dello stesso a fronte della crescita delle richieste. Non è prevista infatti nessuna procedura di ridistribuzione del carico fra i server e l'eventuale spostamento di uno dei server richiede una modifica - seppur semplice e limitata a un file di preferenze - dell'applicazione client. Inoltre non è possibile, se non modificando il codice, aggiungere un ulteriore server al sistema.

In poche parole il nostro sistema gestisce bene la situazione per cui è stato creato, ma potrebbe (?!) risultare inefficiente a fronte di un servizio reale. Si tratta di un sistema **statico** e la modifica che si dovrebbe portare è mirata a renderlo **dinamico**.

La soluzione pensata distingue fra server di **accesso al servizio** e server di **svolgimento del servizio**.

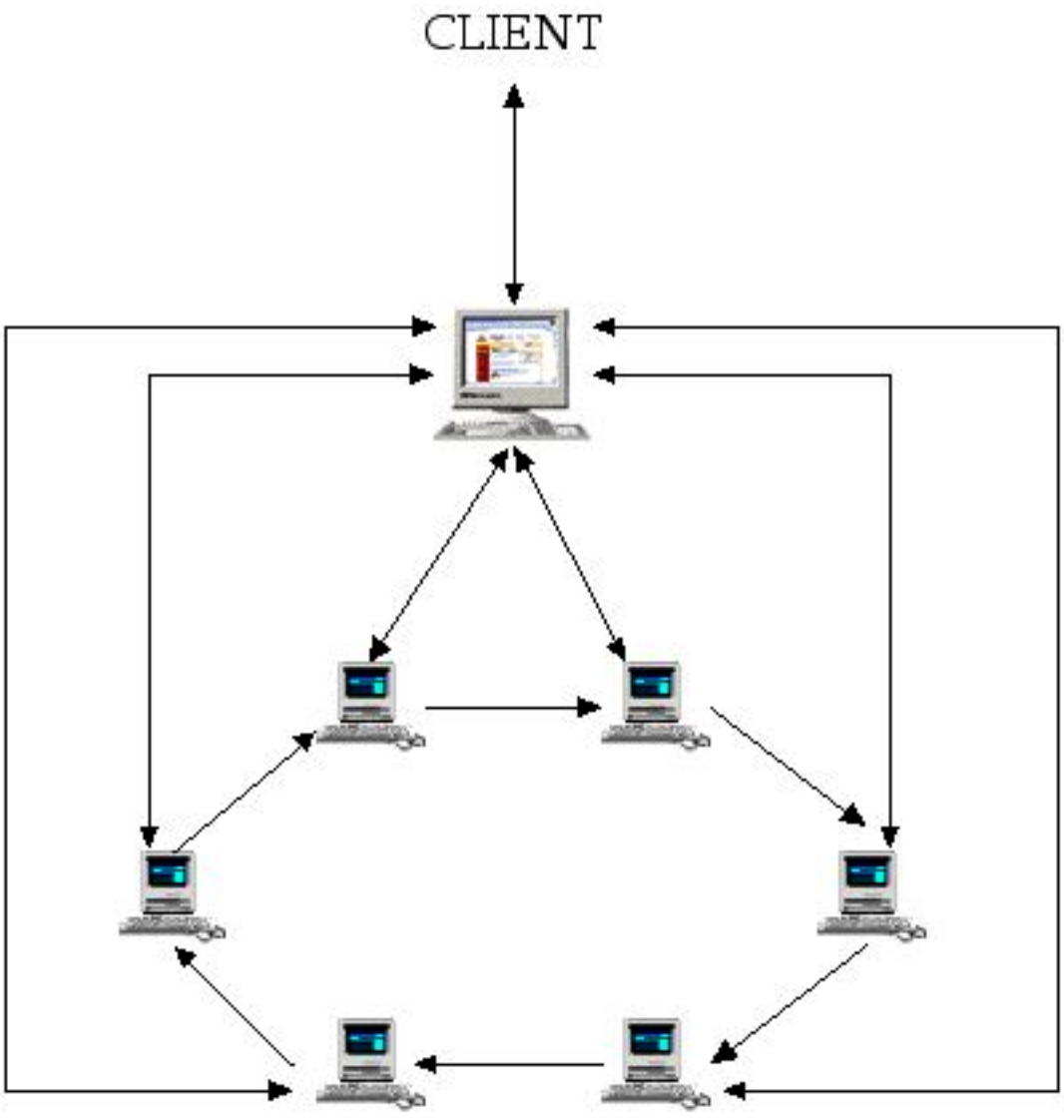
Il cliente accede al server di accesso che lo autentica e gli comunica i parametri del server di svolgimento a cui può collegarsi per effettuare l'acquisto. Tale sistema permette al server di accesso di tenere traccia di ogni collegamento aperto dai server di svolgimento ed effettuare quindi opportune operazioni di ridistribuzione del carico. Può, ad esempio,

decidere di comunicare al client ogni volta il *nome* del server con meno connessioni attive. In questo modo il client conosce solo il nome del server di accesso al servizio ed è quest'ultimo che si preoccupa di permettere la ridirezione del collegamento, funzionando in pratica un po' anche da name server. Solo il server di accesso conosce i server di svolgimento attivi e disponibili e l'informazione diventa perciò trasparente al client.

Chiaramente quando ci riferiamo al server di accesso intendiamo la risorsa server e nulla vieta che sia replicata per ottenere una migliore gestione delle risorse.

Per quanto riguarda la gestione vera e propria del servizio occorre prevedere un meccanismo che permetta di aggiungere e rimuovere un server in qualunque momento. La struttura che si adatta meglio a questo tipo di modello e **l'anello**.

Ognuno dei server conosce soltanto colui che lo segue e colui che lo precede e solo con essi mantiene rapporti, oltre naturalmente al server di accesso. Le informazioni sullo stato dei posti, sulla consistenza, sull'eventuale contemporaneità delle transazioni viaggeranno lungo l'anello in una direzione e saranno disponibili al server che le richiede al termine di un giro completo. In caso di problemi, ad esempio un guasto di un server, sarà il server di accesso a funzionare da by-pass consentendo che le informazioni giungano a destinazione nei temi richiesti dall'applicazione e senza errori. La struttura completa è evidenziata nella figura che segue.



Come si vede i server sono collegati fra loro ad anello, con una direzione privilegiata di circolazione delle informazioni. Ognuno di loro mantiene un link bidirezionale con il server di accesso a cui si collega il client e che conterrà in un'apposita tabella lo stato dei collegamenti attivi dei server dell'anello.

L'informazione sulla transazione in corso sui server sarà comunicata anche al server di accesso che terrà nota delle transazioni in corso, del loro stato e della loro evoluzione, ma non avrà copia del database, presente invece su ognuno dei server di servizio. In questo modo a fronte della situazione di crash non recuperabile vista precedentemente sarà molto più semplice, data la molteplicità delle informazioni presenti, reperire quella necessaria a risolvere il problema dello stato del posto.

E' chiaro come, in caso di inserimento di un nuovo server o di eliminazione di un server esistente, è sufficiente da parte del server di accesso comunicare ai componenti dell'anello interessati la modifica del loro antecedente o conseguente.

Gestione del database

Sin da quando abbiamo introdotto l'oggetto Database abbiamo evidenziato come sia una pura semplificazione considerarlo un file di testo. La semplificazione, che ci permette di manipolare agevolmente i dati senza appesantire il codice, diventa insostenibile se ci riferiamo a una situazione reale. E' evidente che una società che voglia effettuare realmente il servizio che abbiamo studiato non può limitarsi alla nostra semplificazione.

La base di dati va gestita in maniera più complessa, più efficace e con strumenti che, al di fuori dell'applicazione creata, ne consentano una più semplice manipolazione.

Molti server di database utilizzano protocolli specifici dei loro produttori, ma come sempre accade, sono stati definiti degli standard (ad esempio ODBC di Microsoft).

Java utilizza **JDBC** per la connessione a un database, per esempio MS Access.

Per connettere un'applicazione Java a un database relazionale come Access occorre un server di database (ad esempio Access appunto) e un driver di database che fornisca il collegamento fra il pacchetto JDBC e il proprio database. JDBC stesso è provvisto di un *bridge* JDBC-ODBC che consente di accedere al database tramite opportune API.

Il pacchetto per la gestione del database necessita l'importazione di java.sql che fornisce i metodi per la connessione al database, prevedendo anche la possibilità di accesso con user e password. Per il reperimento dell'informazione necessaria si usa una query SQL immessa sotto forma di stringa come argomento di un metodo dedicato.

Resta da definire il problema del commit, la parte più *sentita* per quello che riguarda una transazione. Anche per questo esiste un apposito metodo, *commit()* appunto e la possibilità di utilizzare livelli diversi di isolamento delle transazioni.

In questo modo posso garantire la correttezza delle transazioni e effettuare l'accesso su un applicazione di *database reale*.

Ognuno dei server di gestione del servizio si interfaccerà al suo database in questo modo, utilizzando un proprio client di database che terrà conto dell'implementazione reale della base di dati e fornirà all'esterno (cliente) i risultati del servizio.

Sicurezza della transanzione

E' il problema meno implementato nella nostra applicazione, così come abbiamo avuto modo di spiegare precedentemente.

I dati che viaggiano su una rete sono accessibili da chiunque abbia l'intenzione (e la capacità) di intercettarli. Chiaramente quanto più questi dati sono importanti, quanto più l'informazione che recano possiede una qualche utilità (quanto più conduce direttamente o indirettamente a denaro), tanto più occorre prendere le necessarie precauzioni affinché i dati stessi non vengano intercettati o manipolati da chiunque non sia autorizzato a farlo. Chiaramente la nostra applicazione necessita, se inserita nel mondo reale, di un tale accorgimento.

Nella standard edition del Java 2 SDK è stata inserita la **Java Secure Socket Extension (JSSE)** che implementa una versione Java di SSSL e TLS con possibilità di crittare dati, di autenticazione sia dal lato server che dal lato client, permettendo il passaggio sicuro dei dati da client a server sia HTTP, Telnet o TCP/IP.

Sono previsti metodi per creare SocketSSL e l'implementazione di metodi per l'utilizzo di algoritmi di crittografia quali RSA, RC4, DES, Triple DES, al fine di rendere il più possibile sicura la transazione e immunizzare i nostri dati da un attacco esterno.

Un breve manuale utente

Cerchiamo di capire come funziona esattamente il nostro sistema. Per farlo ci serviremo di screenshots che illustrino esattamente l'ambiente che l'utente si troverà a gestire. Dal momento che il sistema è stato creato in ambiente MacOs gli screenshots potranno avere un aspetto lievemente differente su un sistema Windows o Linux, fermo restano comunque il requisito di portabilità, caratteristica fondamentale di Java e uno motivo per cui si è scelto di sviluppare il progetto con questo linguaggio.

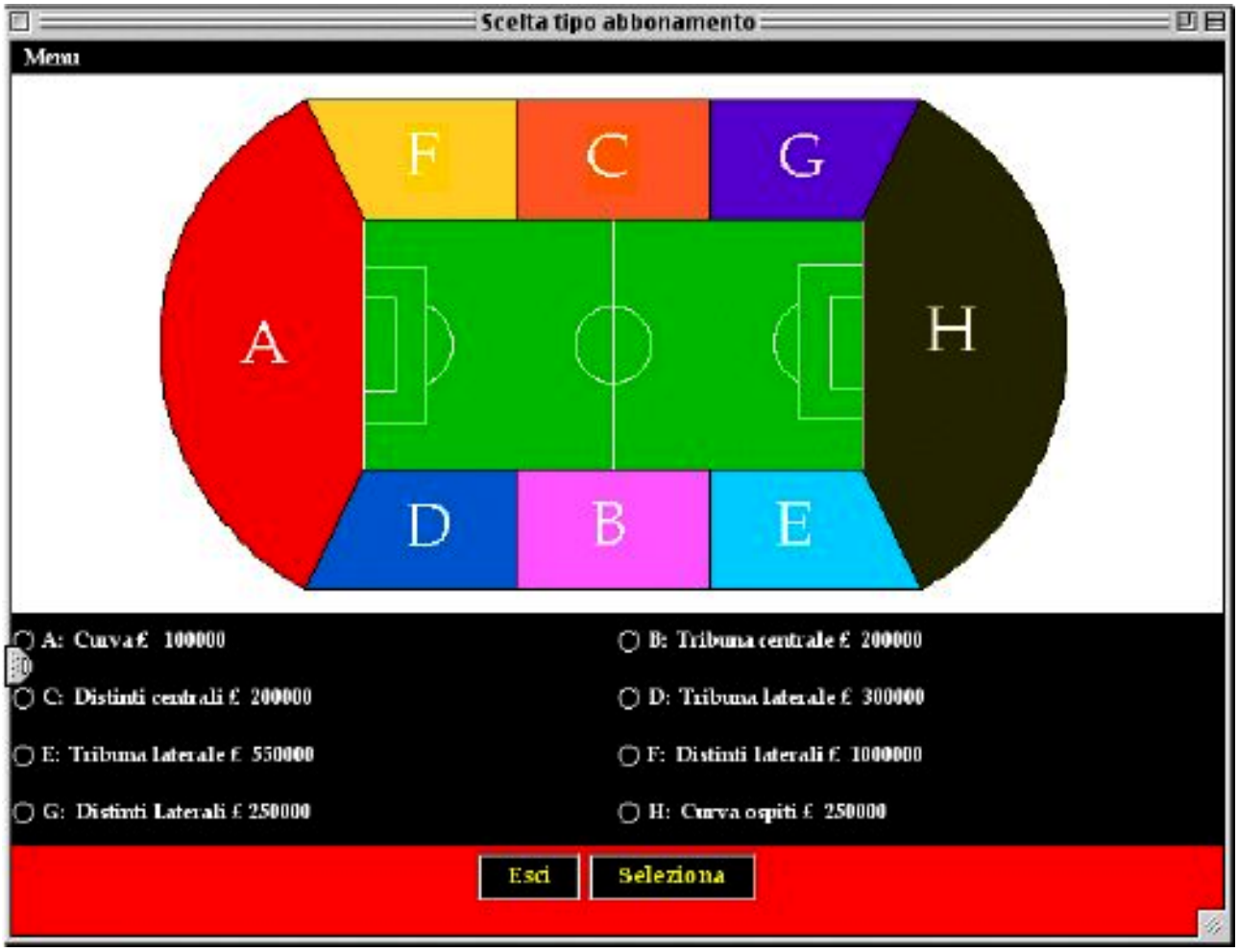
- **Lato client**

Come prima cosa viene richiesta l'autenticazione.

L'indicazione di user e password servono sia al sistema locale per distinguere tra utente singolo e operatore sia al sistema remoto per l'autenticazione al servizio.

In caso di autenticazione corretta il sistema riceve i prezzi dei biglietti e l'applicazione visualizza la mappa dello stadio, in modo che l'utente possa orientarsi nella scelta della sua posizione preferita.

In alto a sinistra (some indicato nella figura sottostante) compare l'indicazione del menu, presente solo nel caso di operatore al botteghino autorizzato alle modifiche sui parametri di collegamento.



La finestra permette di scegliere il settore desiderato. Come meccanismo di prevenzione nei confronti di una selezione errata viene richiesta conferma prima di processare la richiesta

e quindi il client richiede la mappa del settore indicato dall'utente.

La scelta del posto viene quindi determinata visualizzando la mappa completa del settore desiderato, con l'indicazione del numero e della fila e la posizione rispetto al terreno di gioco. Naturalmente lo sviluppo è su pianta e si deve intendere, nella realtà, sviluppato in altezza.

Su ogni posto è indicato il numero. E' sufficiente cliccare sopra al posto desiderato per selezionarlo. E' richiesta una selezione per poter continuare. Esiste un meccanismo di controllo che previene la possibilità di selezione nulla.

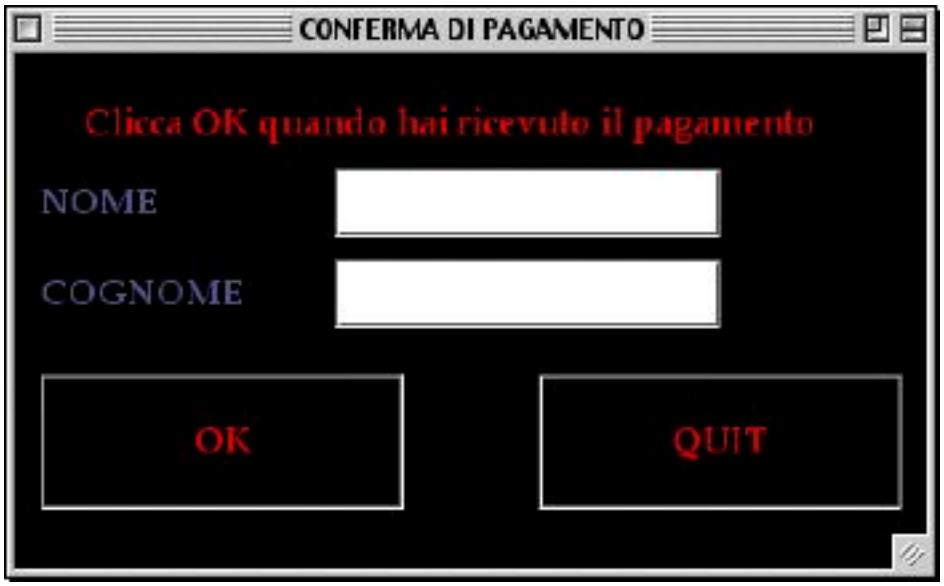
Nella figura portata da esempio il settore (indicato nel titolo della finestra) è interamente libero. Eventuali posti occupati risulterebbero rossi e non sarebbe possibile la selezione. In caso di posti non disponibili - perché in quel momento gestiti da un altro server o da un'altra transazione parallela - la casella apparirebbe grigia e non sarebbe possibile selezionarla.

Anche in questo caso è richiesta una conferma della selezione.

A conferma avvenuta viene gestita la prima parte della transazione si passa all'aspetto economico.

Qui è possibile effettuare una scelta sul metodo di pagamento.

Scegliendo di pagare in contanti sono richiesti i dati dell'acquirente che andranno stampati sull'abbonamento e registrati da chi fornisce il servizio.



La pressione del tasto OK chiude la transazione.

Scegliendo il pagamento con carta di credito invece la transazione è più complessa.

I dati richiesti sono quelli tradizionali utilizzati da un bancomat - si potrebbero implementare mediante un lettore della carta di credito stessa - oltre alla parte anagrafica necessaria alla registrazione.

A pagamento effettuato, in uno qualunque dei due metodi, una finestra indica l'esito della transazione

e permette di scegliere e se lasciare il sistema o effettuare una nuova prenotazione.

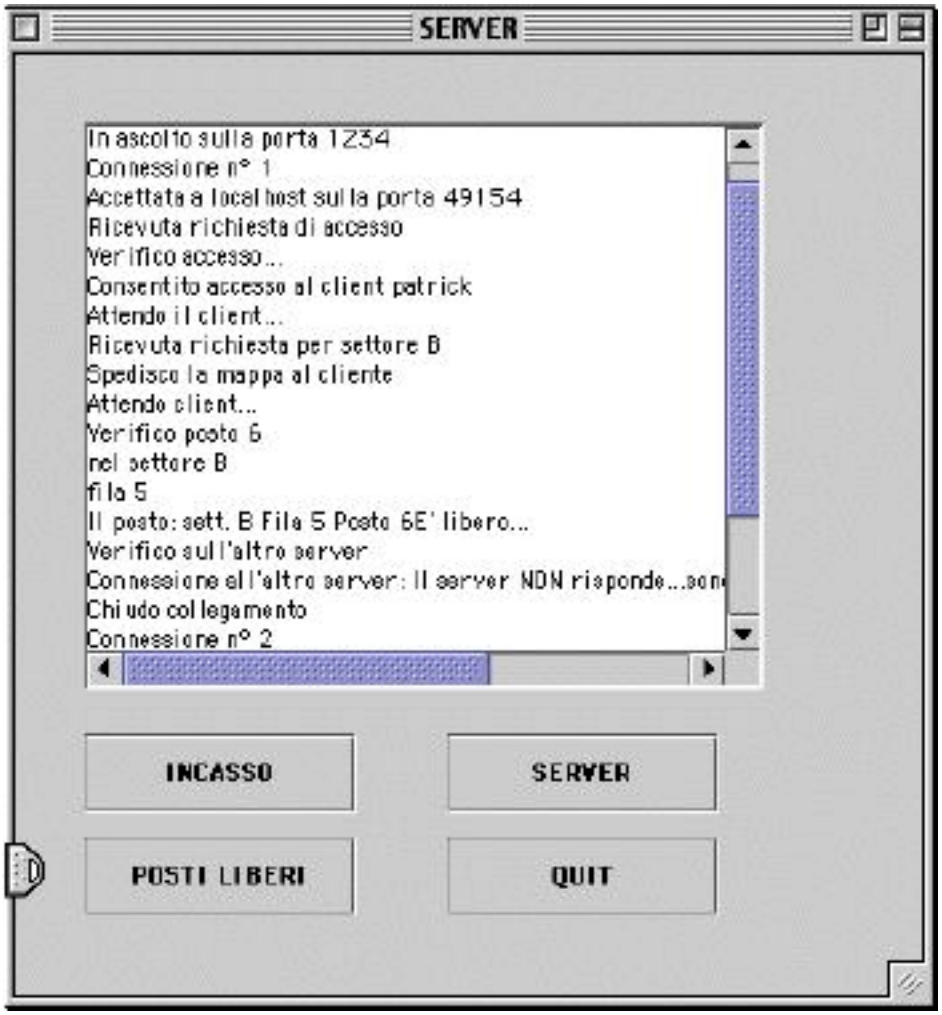
Per quanto riguarda la parte di preferenze del client, possono essere immesse da una finestra separata

Dalla finestra è possibile scegliere l'indirizzo e la porta dei due server e l'ordine di connessione.

- **Lato server**

Da questo lato tutto è naturalmente più semplice e meno ricercato nell'aspetto dell'ambiente. Quello che conta è la gestione dell'informazione, mentre per l'utente anche l'aspetto con cui vengono fornite è un dettaglio non trascurabile.

L'applicazione server ha una finestra fondamentale, composta da un campo di testo su cui vengono visualizzate le informazioni relative alle connessioni e alle operazioni in svolgimento.

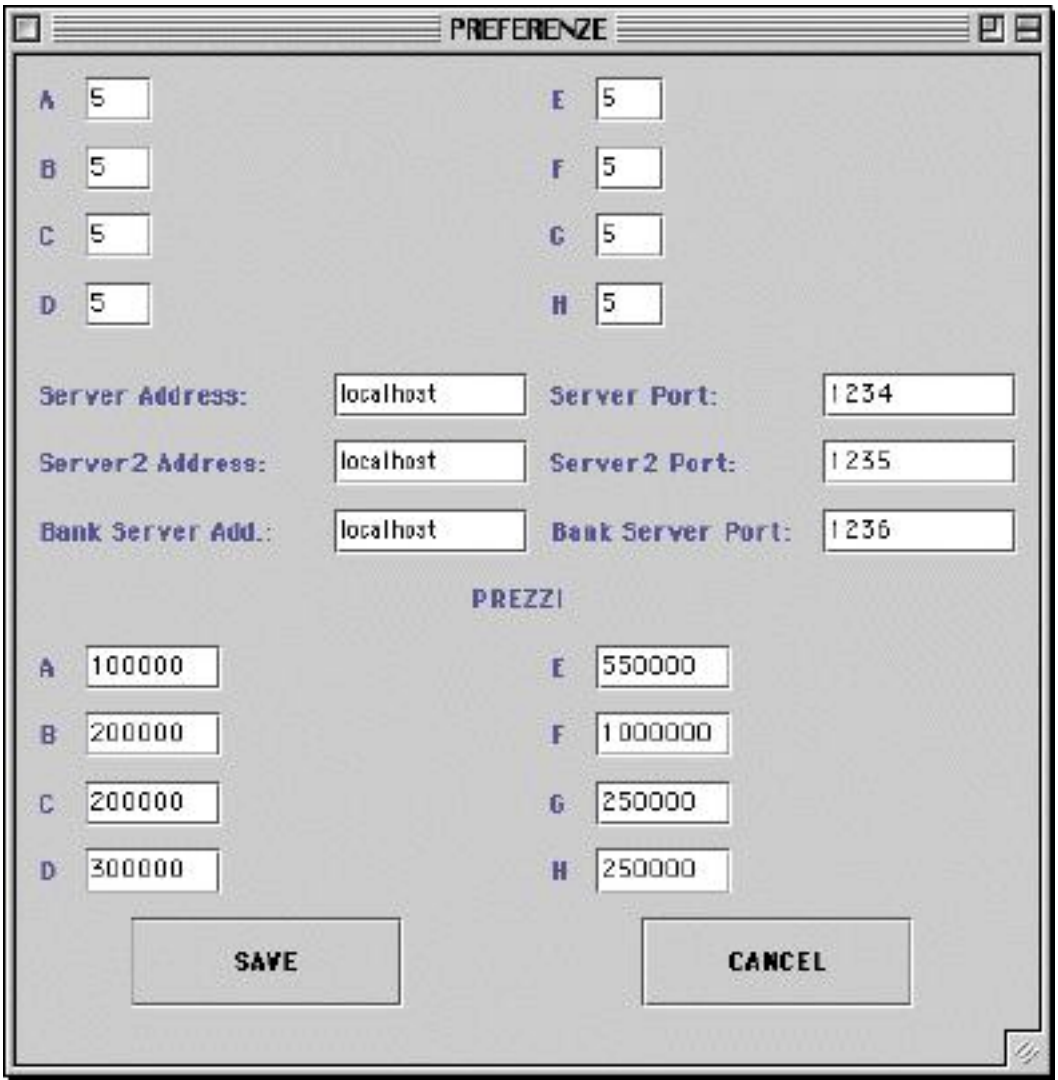


I tre pulsanti presenti sul fondo della finestra (più il quarto di quit) servono a visualizzare alcune informazioni sullo stato delle transazioni.

L'incasso effettuato dal sistema globale (indipendentemente dal server)

il numero dei posti liberi (il sistema in questo caso possiede anche la lista dei posti liberi, la cui visualizzazione non è implementata)

e le preferenze del sistema ottenibili al pulsante SERVER



in cui è possibile modificare il prezzo dei biglietti, oltre naturalmente ai parametri della connessione al server omologo e al Bank Server.

Dati per l'utilizzo

Per utilizzare l'applicazione è necessario conoscere user e password per l'accesso al servizio e i dati dei numeri di conto disponibili nella simulazione. Tali valori sono indicati nella tabella sottostante.

Nome	*N° conto*	*password*
Patrick Fogli	12345	alex
Antonio Corradi	11111	unibo

User	*Password*
patrick	alex
pippo	pluto
corradi	unibo