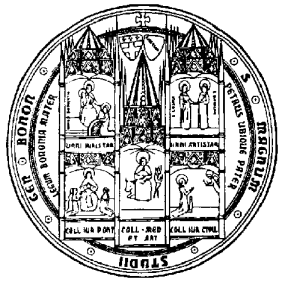


Diploma di Laurea in Ingegneria Informatica

Corso di Ingegneria del Software

Esercitazione n°2

Servizio di Prenotazione Web

La Progettazione: Moduli e Information Hiding

Cenni di Convalida e Verifica

ing. Paolo Bellavista

e-mail: pbellavista@deis.unibo.it tel.: (20) 93866

Principi di Progettazione

- Affidabilità (***robustezza***)
- Modificabilità (***estensibilità***)
- Comprensibilità del progetto (***leggibilità***)
- Rapida prototipazione (***riusabilità***)

Direzioni di soluzione:

Modularizzazione e Architettura :

* coesione
* disaccoppiamento

☑ ***Information Hiding***

Specifiche di Progetto

- Scelte architetturali (hw/sw)
  - quali moduli
  - quali interfacce per i moduli
  - interazione fra i moduli
- Protocolli di comunicazione
  - fra i differenti moduli
  - fra i vari servizi esistenti in sistemi distribuiti (Web)
- Linguaggio di Programmazione
  - si può scegliere solo a questo punto?
- Diagramma delle Classi

Notazioni per la Progettazione

Con differente ***livello di astrazione*** e differente ***granularità di dettaglio***

- **Textual Design Notation (TDN)**

```
module ArchivioPrenotazioni
        uses InterfacciaArchivio
        exports
          procedure aggiungi(in nuovoOrd:Ordine);
          procedure esisteCodice(in code:integer;
                 out esiste:boolean);
          ...
        Implementation ...
        composed of A, B;
        end ArchivioPrenotazioni;
```

- **Graphical Design Notation (GDN)**

Moduli e Astrazioni sui Dati

	Tipizzazione →	
Generi cità ↓	Oggetti	Tipi di dati astratti
	Oggetti generici	Tipi di dati astratti generici

La Programmazione Orientata agli Oggetti

- ***Separazione*** fra Interfaccia e Implementazione
- Codice + Stato (persistenza?)
- ***Ereditarietà*** semplice/multipla
- Polimorfismo
- Strong Typing (casting)
- ***Late Binding***

==> **UML** e progettazione OO (to be continued...)

3) PROGETTAZIONE & PROGRAMMAZIONE ('COME')

3.1) LINGUAGGIO DI PROGRAMMAZIONE

> Pensiero e linguaggio sono per l'artista strumenti di un'arte.
> Vizio e virtù sono per l'artista materiali di un'arte.
> (da "il ritratto di Dorian Gray" di Oscar Wilde)

Ho pensato di usare **Java** che, rispetto al C++, è un vero linguaggio orientato agli oggetti.
La scelta è quasi forzata, perché sarà necessario usare comunicazioni su socket via Internet, la mutua esclusione nell'accesso a alcuni oggetti, la sospensione di thread sui monitor associati agli oggetti e la possibilità di "serializzare" un oggetto e spedirlo via socket.
Attualmente il "JDK" è disponibile la versione 1.2.1.
Tuttavia c'è un vincolo sulle risorse disponibili: il programma dev'essere eseguibile nei laboratori della facoltà.
Perciò programmerò secondo lo standard dell JDK v **1.1** (leggansi: nei nostri laboratori non hanno ancora installato la nuova versione).

3.2) COMUNICAZIONI VIA SOCKET

> La scienza si può comunicare, ma la saggezza no. Si può trovarla, si può viverla [..] ma dirla e insegnarla non si può.
> (da "Siddharta" di Hermann Hesse)

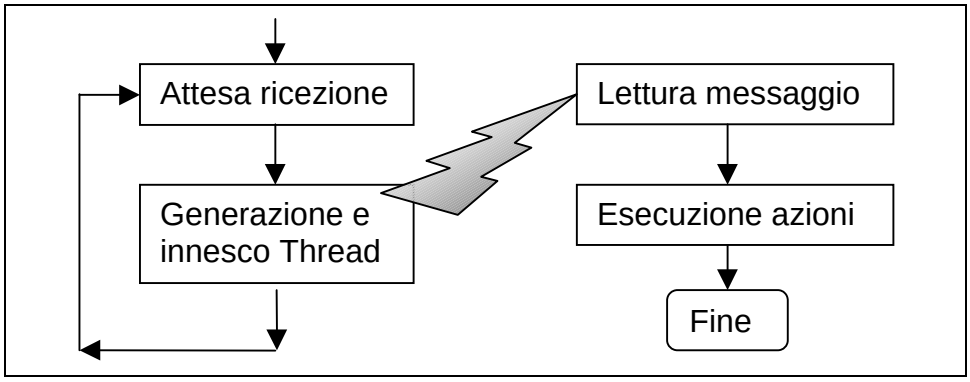
Il client resta in attesa finché tutti i server non hanno registrato la prenotazione; perciò le comunicazioni tra i server devono essere il più veloce possibile.
Tra client e server penso di usare una connessione basata su socket stream (in modo da poter trasferire anche lunghi flussi di dati, come una mappa del teatro con i posti liberi e quelli prenotati). Tale connessione è di sua natura affidabile.
Tra server e server preferisco usare una comunicazione a datagrammi, ossia si inviano pacchetti di dimensione limitata senza alcuna affidabilità (ossia il pacchetto può non arrivare a destinazione). Però si guadagna in velocità e quindi in efficienza.
Ovviamente si devono mettere in conto dei controlli aggiuntivi per rimediare alle situazioni di perdita di un pacchetto.
Usando i datagrammi, non si può' pensare di fare un protocollo di scambio di informazioni di tipo "negoziativo" (ossia un server invia i dati a un server e poi si aspetta una risposta a cui può ribattere) poiché non c'è distinzione tra un pacchetto e l'altro, e non è garantito nemmeno che venga mantenuto l'ordine di consegna di una sequenza pacchetti.
Ho pensato di costruire un protocollo in cui i vari server si scambiano messaggi "univoci", che contengono tutta l'informazione su cosa si debba fare.
Per es., un server manda al suo server successore la prenotazione (che lui ha appena fatto); questi può non essere in grado di fare la prenotazione (è in corso una prenotazione di qualcun altro su quel server) perciò deve dire al suo predecessore che non può farla! Invierà' a esso un messaggio di cancellazione della prenotazione appena ricevuta!
Il server predecessore non era in attesa di quel particolare messaggio (ma era in attesa di un messaggio qualsiasi), ma ricevendolo esegue ciecamente l'ordine ricevuto (di cancellare la prenotazione) e farà circolare il messaggio.
Ma quando devo connettere un nuovo server mi può essere più utile usare delle comunicazioni a "stream", altrimenti dovrei rifare nel programma tutti i controlli che fa a basso livello la "socket-stream"… e sarebbe molto dispendioso.
Inoltre l'aggancio di un nuovo server è un'operazione molto "rara" e quindi non è grave se si impiega un po' di più per fare una connessione a "stream".

Invece i messaggi di prenotazione e cancellazione devono muoversi più velocemente possibile (il client sta attendendo) e quindi la comunicazione via datagrammi è necessaria.
Quindi userò un'altra socket a stream da cui mi aspetto messaggi "di sistema", ossia quelli che servono a aggiungere un nuovo server, a controllare se un server è vivo, ecc..
Riassumendo, ogni server sarà in attesa di messaggi da tre socket (legate a tre porte differenti):

- stream – per le comunicazioni con un client;
- datagram – per comunicazioni di prenotazione con gli altri server;
- stream – per comunicazioni di sistema con gli altri server.

3.3) GESTIONE CONCORRENTE

Ogni server dev'essere in grado di "mettersi in ascolto" su due tipi di socket differenti.
Servono due "thread" concorrenti, ognuno dei quali si mette in ascolto su una differente socket, e lì resta bloccato fino all'arrivo di un messaggio.
Ogni messaggio ricevuto deve essere "capito" e l'ordine contenuto va "eseguito".
Tutta questa "gestione" non deve bloccare altri possibili clienti che vogliono prenotarsi. Perciò si dovrà fare un "server concorrente".
In pratica, ogni volta che si riceve un messaggio si genera un nuovo thread che pensa a gestirlo, la cui esecuzione avviene in parallelo a quella del thread che l'ha generato.
Il thread d'accettazione dei messaggi può così tornare subito a accettare un nuovo messaggio.
Schema di base:



3.4) PRENOTAZIONI

Ma che cos'è una prenotazione? Quando si prenota un posto a teatro, occorre indicare lo spettacolo, il posto e i propri dati anagrafici.
Ho pensato di racchiudere ognuna di queste tre caratteristiche in una classe e creare la classe "Prenotazione" che le usa tutte e tre (è una relazione di "**usa**", non di "eredita_da"!)

3.4.1) Spettacolo

Conterrà tutte le informazioni su una particolare rappresentazione teatrale, ossia:

- il nome dello spettacolo;
- la data (in formato giorno/mese/anno);
- il turno (per es. un codice numerico che indica il primo o il secondo spettacolo di una stessa giornata).

3.4.2) Posto

Conterrà solamente l'indicazione del posto.
Per semplicità ho ipotizzato i posti suddivisi in file (indicate da una lettera) e posizioni (indicate con un numero).
Ho aggiunto degli operatori di "scorrimento" dei posti, ossia i metodi per:

- rendere il primo posto;
- rendere il successivo di un posto;
- rendere l'ultimo posto possibile.
- controllare se un posto e' "corretto".

Così è possibile scorrere tutti i posti di un teatro indipendentemente da come è strutturato questo oggetto.

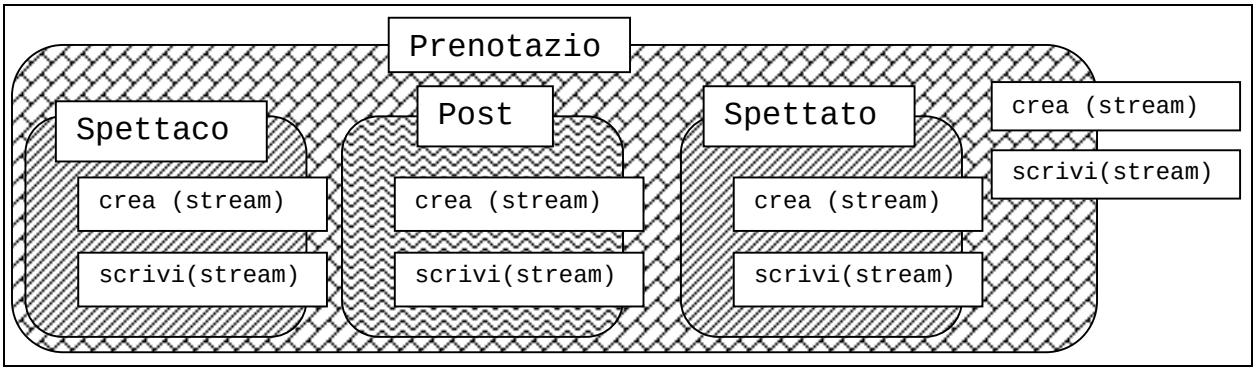
3.4.3) Spettatore

Contiene i dati anagrafici di uno spettatore; nel caso specifico contiene:

- nome e cognome dello spettatore.

3.4.4) Information Hiding

In questo modo ho nascosto i dettagli implementativi delle prenotazioni, prevedendo in anticipo possibili evoluzioni o modifiche future.
Un fatto importante è che l'oggetto "prenotazioni" dovrà poi essere inserito in un messaggio e quindi a sua volta convertito in un array di byte (e viceversa).
Devo scrivere anche per questo oggetto dei metodi di conversione, come fatto per "Mess", (uno che scrive l'oggetto su uno stream di dati e uno che lo costruisce leggendo da uno stream di dati).



3.5) MESSAGGI

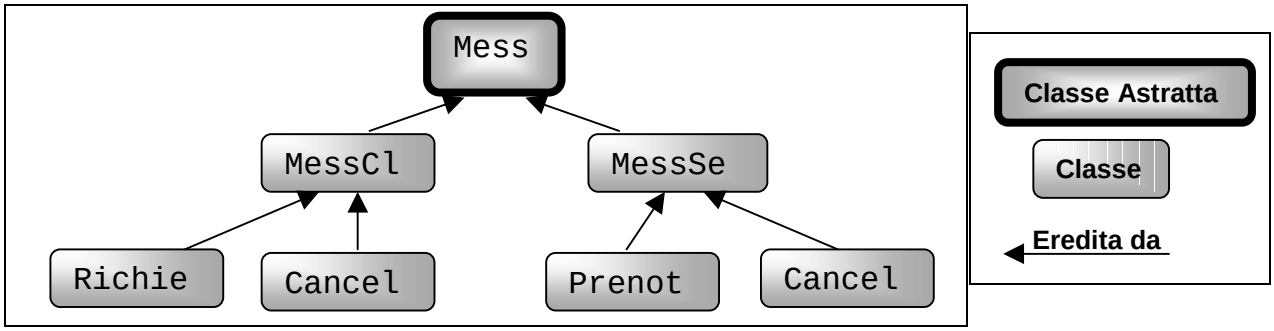
Per "messaggi" si intende quelli che vengono scambiati tra i client e i server.

3.5.1) Messaggi come Oggetti

Occorrerà distinguere i messaggi tra quelli generabili dai client e quelli generabili dai server. Creerò una classe astratta, per es. "Mess", da cui specializzerò due classi, per es. "MessCli" e MessSer", da ognuna delle quali farò discendere le classi dei messaggi che sono riconoscibili rispettivamente dal client e dal server.
In tal modo al client può essere negata la visibilità dei messaggi generabili dal server.
In particolare, i messaggi per i server devono contenere:

- l'indicazione della "direzione" in cui vanno inviati (avanti o indietro);
- un indicatore di "tempo di vita" per evitare un possibile circolo infinito di un messaggio.

Una prima possibile schematizzazione può essere:



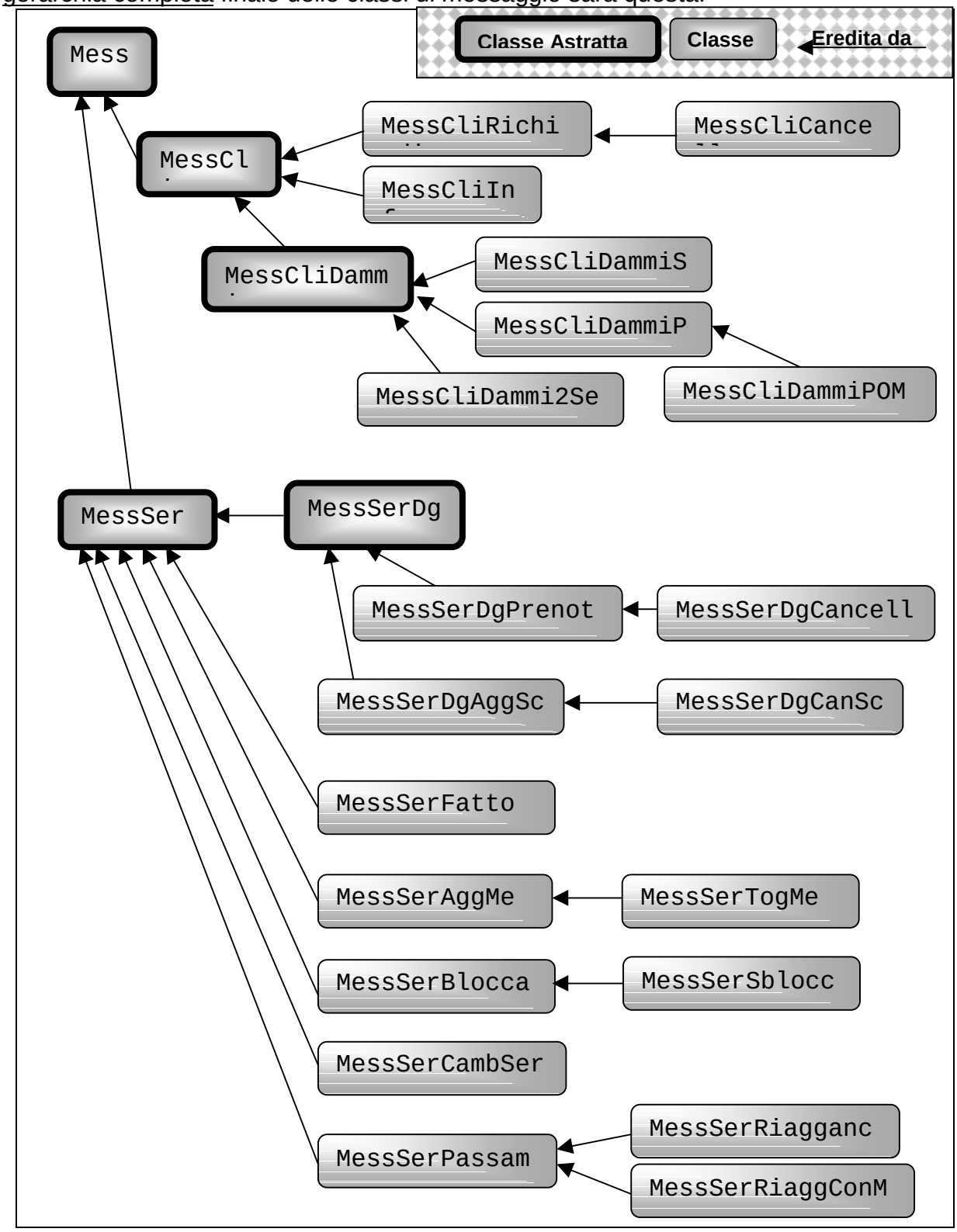
Poiché ogni messaggio è una classe a se e quindi dovrà essere scritto in un file ".java" separato, adotterò la nomenclatura di mettere il prefisso "Mess" a tutte le classi di messaggi, seguito da "Cli" o "Ser" per distinguere se è generabile rispettivamente da un client o da un server, seguito infine da una breve descrizione.
I messaggi inviabili tra i server possono essere ulteriormente classificati.
Infatti un server può inviare un messaggio al suo successore il quale lo manderà al suo successore e così via (ossia il messaggio viene fatto circolare nell'anello virtuale) oppure il messaggio è "specifico" per il successore (per esempio i messaggi che devono essere scambiati durante l'aggiunta di una nuova macchina).
I messaggi che girano nell'anello verranno incapsulati in datagrammi, perciò creerò una sottoclasse (astratta) di "MessSer" chiamata "MessSerDg" da cui erediteranno tali messaggi. Le classi astratte sono quattro, e alcune definiscono attributi e metodi comuni alle proprie sottoclassi. Esse sono:

- **Mess**
  - classe base di tutti i messaggi;
  - serve solamente come astrazione di un messaggio perciò non definisce nulla di comune a parte il metodo (astratto) "scriviInStream" che dovrà essere implementato da ogni sottoclasse (per scrivere i dati della sottoclasse in un DataOutputStream).
- **MessCli**
  - classe base di tutti i messaggi che possono essere scambiati tra client e server;
  - contiene l'attributo
    - "Tentativo" che serve per numerare i vari tentativi di invio di un messaggio da parte di un client; allo scadere del time-out il client ritrasmetterà lo stesso messaggio ma con questo attributo incrementato di uno.
  - a tale scopo si è aggiunto il metodo "incrementaTentativo()" e un costruttore senza argomenti (che inizializza l'attributo a uno).
- **MessSer**
  - superclasse di tutti i messaggi scambiabili tra server e server;
  - definisce le costanti:
    - "AVANTI" e "INDIETRO" per indicare le possibili direzioni di circolazione;
  - ha un metodo di classe "cambiaDirez" che rende la direzione opposta a quella data.
- **MessSerDg**
  - classe base di tutti i messaggi scambiabili tra server e server "lungo l'anello";
  - è sottoclasse di "MessSer"
  - contiene tre attributi:
    - "Direzione" per indicare la direzione in cui il messaggio va fatto "circolare";
    - "Vita", un indicatore del numero di server attraversabili prima che il messaggio venga "volutamente" perso;

- “Macchina”, è un’oggetto che contiene importanti informazioni sulla macchina che ha generato (e messo in circolazione) il messaggio.

– si sono aggiunti metodi per leggere e ridefinire tali attributi, in particolare un “decrementaVita()” per agevolare la modifica della vita.

La gerarchia completa finale delle classi di messaggio sarà questa:



3.5.2) Messaggi come Sequenza di Byte

Sulla rete i messaggi devono viaggiare in stream o in pacchetti, ossia come una sequenza di byte, mentre in memoria sono degli "oggetti".
Occorre prevedere una "conversione" tra oggetto e sequenza di byte.
Pensavo di incapsulare tale conversione direttamente nella classe dei messaggi.
Ogni classe avrà un costruttore aggiuntivo che servirà per creare un messaggio avendo in ingresso una sequenza di byte.
Un metodo, chiamato per es. "scriviInByte", farà l'operazione inversa, ossia renderà la rappresentazione del messaggio come sequenza byte (ossia renderà un array).
Posso incapsulare la sequenza di byte in uno stream di "dati" (per es., "DataInputStream", passando per "ByteArrayInputStream").
Nota: non uso la serializzazione su questi oggetti per non appesantire la trasmissione. La serializzazione introduce un ulteriore numero di informazioni nel messaggio (per es. per poter controllare che ci sia compatibilità tra ciò che viene serializzato e la classe che dovrà essere istanziata tramite tali dati).
Voglio che la trasmissione dei messaggi sia il più veloce possibile, perciò eviterò la serializzazione!

3.5.3) Conversione

Serve un oggetto che esegue le conversioni tra le due rappresentazioni di un messaggio, ossia deve svolgere le operazioni di:

- scrittura di un oggetto "Mess" su un "DataOutputStream";
- lettura di un oggetto "Mess" da un "DataInputStream".

Si deve ovviamente usare il "late-binding", ossia i due metodi definiscono il messaggio che usano "staticamente" di classe "Mess" mentre "dinamicamente" renderanno un messaggio che è un'istanza di una particolare sottoclasse di "Mess".
Si è così creato l'oggetto statico "TraslaMess".
Esso non fa altro che associare a ogni classe di messaggio un codice numerico da inserire nel flusso prima di scriverci il messaggio vero e proprio.

- Scrittura: si scrive nello stream il codice relativo alla classe del messaggio e poi si invoca il metodo del messaggio che ne esegue la scrittura su stream;
- Lettura: si legge il primo dato (il codice della classe del messaggio) e in base a questo si chiama il costruttore, della classe indicata dal codice, che ha come parametri un flusso di dati in input.

3.5.4) Operazioni fondamentali

Ogni classe deve avere due costruttori.
Il primo permette di costruire l'oggetto partendo dai dati elementari che lo costituiscono mentre l'altro esegue la costruzione leggendo i dati da uno stream di dati in input ("DataInputStream").
Per esempio, "MessCliInfo" è la classe dei messaggi usata dal server per inviare un messaggio (contenuto in una stringa) al client.
Il suo primo costruttore darà definito come "MessCliInfo (String Frase)", mentre il secondo sarà "MessCliInfo (DataInputStream DIS)".
Ogni classe dovrà avere dei metodi "selettori" per poter estrarre i dati dall'oggetto.
Per "MessCliInfo" avrò un metodo del tipo "String cheFrase ()" che serve per ottenere il messaggio portato dall'oggetto.
Per oggetti con più dati interni occorrerà prevedere un metodo "selettore" per ogni dato.

3.5.5) Semantica

Ai messaggi non viene associata nessuna semantica.
Questo vuol dire che al loro interno (nell'oggetto) non ci sarà nessun codice che indica cosa fare quando tale messaggio viene ricevuto.
Il solo scopo dei messaggi è di portare delle informazioni da una postazione a un'altra.
La loro interpretazione è lasciata al ricevente del messaggio, questo perché le operazioni che verranno fatte andranno a influenzare il server, il data base e altri oggetti del sistema.
Comunque, il nome di ogni messaggio ne indica lo scopo:

- **MessCliRichiedi** (Prenotazione) – per la richiesta da parte di un client di fare una prenotazione;
- **MessCliCancella**(Prenotazione) – idem ma per una cancellazione;
- **MessCliInfo** (Frase) – messaggio di informazioni reso dal server al client;
- **MessCliDammiSc** () – richiesta dell'elenco di tutti gli spettacoli in programma;
- **MessCliDammiPL**(Spettacolo) – richiesta dell'elenco di tutti i posti liberi per un certo spettacolo;
- **MessCliDammiPOMe** (Spettacolo, Spettatore) – idem, ma di tutti i posti occupati da un certo spettatore;
- **MessCliDammi2Ser** () – per richiedere al server gli indirizzi di altri due server;
- **MessSerDgPrenota** (Prenotazione) – ordine di una prenotazione da parte di un server verso un altro server;
- **MessSerDgCancella** (Prenotazione) – idem, ma di una cancellazione;
- **MessSerDgAggSc** (Spettacolo) – ordine di aggiungere il nuovo spettacolo;
- **MessSerDgCanSc** (Spettacolo) – idem, ma per la cancellazione;
- **MessSerFatto** (Sì/No) – risposta tra server per indicare che la precedente operazione è stata fatta oppure no;
- **MessSerAggMe** (IndirizzoServer) – richiesta di aggiunta di un nuovo server all'anello;
- **MessSerTogMe** () – idem, ma per la rimozione di un server (non serve l'indirizzo);
- **MessSerBlocca** (Direzione) – ordina di bloccare le transazioni in una certa direzione;
- **MessSerSblocca** (Direzione) – idem, ma per riabilitarle;
- **MessSerCambSer** (Direzione, NuovaMacchinaServer) – ordina di cambiare l'indirizzo del server vicino (secondo la direzione data); inoltre è passata anche la priorità.
- **MessSerPassami** (IndGener, Frase) – messaggio per controllare se l'anello funziana (ossia se c'è un server che è caduto); serve anche per passare un messaggio a video a tutti i server presenti.
- **MessSerRiaggancio** (IndCaduto, IndGener) – generato all'indietro dalla macchina che trova che il proprio successore è caduto; lo riceverà il successore del nodo caduto;
- **MessSerRiaggConMe** (IndGener) – inviato dal successore al predecessore di un nodo caduto per ripristinare l'anello.

3.5.6) Dove?

Le "azioni" da fare sono quindi all'interno di vari oggetti usati dal server e dal client.
In alcuni oggetti ho inserito i metodi per iniziare a richiedere o a servire un particolare messaggio.
Per avere una facile lettura, ho distinto i vari metodi così:

- se iniziano con "Avvia_", vuol dire che servono per iniziare un protocollo, ossia contengono il codice che genera il messaggio di richiesta e eventualmente accetta l'arrivo dei successivi messaggi stabiliti dal particolare protocollo;

- se iniziano con “Servi_”, sono metodi di risposta a un particolare messaggio che servono per iniziare il protocollo dalla controparte, ossia contengono il codice che intraprende tutte le operazioni da dopo è arrivato il messaggio iniziale;
- tutti i metodi di queste due categorie terminano con il nome, per intero, del messaggio che è all’origine del protocollo, ossia il primo messaggio che viene scambiato.

Per esempio, “Avvia_MessSerCambSer” e “Servi_MessSerCambSer” sono i metodi da invocare quando si vuole iniziare il protocollo di “cambio nome server” rispettivamente dal lato di chi fa la richiesta e da quello di chi deve rispondere a essa.
Ho “sparpagliato” i vari metodi nella seguente maniera:

- i metodi per i messaggi inviabili da client verso un server sono in “**ClientApplet**”;
- quelli per rispondere (da parte del server) a tali messaggi sono in “**GestCli**”;
- quelli sia per inviare sia per rispondere a messaggi tra server su socket a datagrammi sono in “**GestSer**”;
- quelli sia per inviare sia per rispondere a messaggi tra server su socket a stream sono in “**Protocollo**”; da esso ereditano le classi “**Server**” e “**GestSys**”;

La suddivisione dell’invio e della risposta fatto tra “ClientApplet” e “GestCli” è necessario perché “GestiCli” dovrà essere in grado di generare un messaggio “MessSerDgPrenota” (da passare al “GestSer” che creerà), e perciò dovrà avere memorizzato su disco tale classe.
Poiché voglio che il client non abbia alcuna conoscenza dei messaggi generabili tra i server, ho dovuto per forza fare la separazione.
Invece il “GestSer” è l’unica classe a possedere tutti i metodi che lavorano sui messaggi che viaggiano in datagrammi. Nessuno all’infuori di un oggetto di tale classe può inviare o servire messaggi che sono sottoclassi di “MessSerDg”.
Un discorso diverso va fatto per la classe “Protocollo”. Essa contiene tutti i metodi che iniziano o servono i messaggi tra i server che viaggiano su socket-stream.
Ma nessuno crea un’istanza di tale classe! Infatti le due classi “Server” e “GestSys” sono sottoclassi di “Protocollo”, perciò conterranno entrambe tutti i metodi di invio e ricezione (ho fatto così per evitare la replicazione di codice nelle due classi).
A prima vista può sembrare che “GestSys” debba solo servire le richieste iniziate da “Server”, ma non è così, poiché alcuni protocolli prevedono al loro interno l’invocazione di metodi per l’inizio o il servizio di altri protocolli.
Si prenda per esempio il protocollo generale di “aggiunta di un server” (vedere descrizione molto più avanti).
Il nuovo server inizierà il protocollo, ma al suo interno deve attendersi e servire un messaggio per il “cambio del server”; il server esistente deve servire il messaggio di “aggiunta”, ma al suo interno deve anche poter iniziare richieste verso il terzo server.

3.6) MACCHINA

> Siamo su un treno che va a trecento km all’ora,
> non sappiamo dove ci sta portando e, soprattutto,
> ci siamo accorti che non c'è il macchinista.
> (Carlo Rubbia)

L’oggetto “Macchina” serve come astrazione di una macchina fisica nell’anello.
Esso contiene due attributi:

- una stringa contenente l’indirizzo della macchina (in formato numerico, con punti);
- un oggetto di classe Priorità, contenente la priorità della macchina.

In molti casi è utile usare questo oggetto e dev’essere scrivibile in un messaggio.

3.7) MEMORIA DELLE PRENOTAZIONI

> E` una ben povera memoria quella che funziona solo all'indietro. (Lewis Carroll)

Ogni server userà un oggetto (che per ora chiamo "MiePrenotazioni") che ha lo scopo di memorizzare tutte le prenotazioni o cancellazioni che il server stesso ha richiesto di fare agli altri. Sarà una tabella in cui a ogni prenotazione è associata l'informazione del thread che sta aspettando la risposta (che a sua volta passerà al client).
L'oggetto dev'essere unico (una sola copia all'interno del sistema) e accessibile da qualunque thread del server.
Penso che metterò l'istanza come campo "public static" nella classe "Server".
L'accesso dei thread deve essere in mutua esclusione, quindi tutti i metodi saranno dichiarati come "synchronized".
Le operazioni rilevanti sono:

- cancellazione della tabella;
- aggiunta di una coppia (Prenotazione, Thread) ;
- cancellazione di una coppia (Prenotazione, Thread) ;
- data la Prenotazione, rendere se c'è;
- data la Prenotazione, estrarre il Thread associato (rimuovendo la coppia).

A ben guardare tutte queste funzionalità sono già offerte dalla classe predefinita "Hashtable" (ha addirittura già tutti i metodi "synchronized").

3.8) DATA BASE

> Il passato è tutto ciò sul quale da un lato i documenti e dall'altro la memoria sono d'accordo. (da "1984" di George Orwell)

Il DataBase del teatro (che sarà incapsulato in un oggetto) conterrà tutti gli spettacoli possibili e i posti (prenotati e prenotabili) per ognuno di essi.
Come per l'oggetto "MiePrenotaz", anche l'oggetto "DataBase" dev'essere unico e accessibile in mutua esclusione da parte di qualunque thread del server.
A differenza di "MiePrenotazioni", voglio che l'intero archivio sia "serializzabile" poiché dovrò copiarlo per intero ai nuovi server che si aggiungono dinamicamente all'anello (penso che lo invierò brutalmente su una socket-stream appositamente creata tra server e server).
Per poter essere serializzabile, la classe DataBase non può essere definita come una classe con attributi e metodi "static" (infatti posso trasferire e ricevere un oggetto, ossia un'istanza di una classe, non una classe).
Devo crearla come una normale classe. Però in ogni server dev'essercene una sola istanza! Essa sarà memorizzata in un campo "public static" della classe "Server" e verrà creata dal thread "Server" all'avvio.
Ricordandosi però che il DataBase dovrà essere acceduto in mutua esclusione, occorre che tutti i suoi metodi siano definiti come "synchronized ".
Le operazioni rilevanti sul Data-Base sono:

- creazione della tabella vuota;
- aggiunta di un nuovo Spettacolo;
- cancellazione di un nuovo Spettacolo;
- inserimento di una terna (Spettacolo, Posto, Spettatore);
- idem, ma che non dà errore se inserisco lo stesso Spettatore;
- cancellazione di una terna (Spettacolo, Posto, Spettatore)
- idem, ma che non dà errore se libero un posto vuoto;
- dato Spettacolo e Posto, rendere se è libero;

- dato Spettacolo e Posto, rendere lo Spettatore;
- rendere una lista con tutti gli spettacoli presenti;
- dato Spettacolo, rendere una lista con tutti i posti liberi
- dato Spettacolo, rendere una lista di tutti i posti prenotati e da chi.

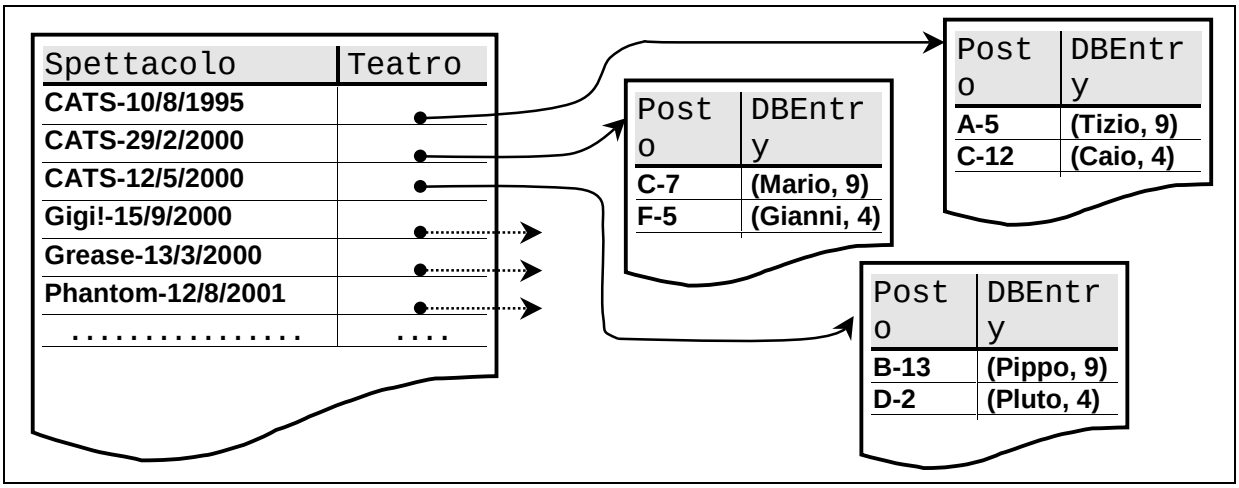
Penso di implementarla con una tabella Hash di tabelle Hash, ossia con una tabella Hash fatta su (Spettacolo, Teatro) dove Spettacolo è la chiave e Teatro è a sua volta una tabella Hash ma fatta su (Posto, Spettatore) dove Posto è la chiave.
Infatti le ricerche fondamentali (che è importante velocizzare) sono quelle basate sulla chiave "Spettacolo", eventualmente accoppiata alla chiave "Posto".
In questo modo si velocizzano gli accessi (la ricerca è più veloce).

3.8.1) Cosa inserire?

Per la precisione, la terza voce non è esattamente "Spettatore", ma un nuovo oggetto che chiamerò "DBEntry" che serve per contenere due oggetti:

- lo spettatore vero e proprio;
- la priorità del server che ha fatto la prenotazione;

Il salvataggio della priorità del server che ha generato la prenotazione si è reso necessario per far in modo che ogni server possa localmente stabilire se una nuova prenotazione possa o meno sovrascrivere una precedentemente memorizzata (l'uso di questo campo è descritto più avanti, in "collisione di prenotazioni").



3.8.2) Nuove Operazioni

Si devono perciò modificare le operazioni di inserimento:

- inserimento di una terna (Spettacolo, Posto, Spettatore) che accede allo spettacolo dato e:
  - se il posto è libero, vi inserisce lo spettatore;
  - se il posto è occupato ma dallo stesso spettatore passato, non genera un errore ma ritorna con successo, senza reinserire lo spettatore;
  - se il posto è occupato da qualcun altro, non può occuparlo (non si confronta alcuna priorità) e quindi ritorna con errore;
  - quando viene inserito, allo spettatore è associata la priorità del server attuale (il possessore del data base).

- inserimento di una quaterna (Spettacolo, Posto, Spettatore, Priorità) che accede allo spettacolo dato e:
  - se il posto è libero, lo occupa;
  - se il posto è occupato ma dallo stesso spettatore, non genera un errore ma ritorna con successo, senza reinserire lo spettatore;
  - se il posto e' occupato da qualcun altro, si **controlla la priorità** della prenotazione passata (contenuta in Priorità) con quella associata allo spettatore che occupa già il posto (nel data base) e perciò:
    - se il "nuovo arrivato" ha una priorità superiore al "vecchio", il vecchio spettatore viene sovrascritto dal nuovo e si ritorna con successo;
    - se invece ha priorità inferiore, non può essere inserito e quindi ritorna con errore;

La distinzione tra le due è molto importante!
Quando un server inserisce una prenotazione che gli è appena arrivata da un client non deve guardare la priorità; o il posto è libero (e quindi lo occupa) o è occupato (e allora non può farci niente).
Ogni server deve distinguere se la prenotazione che sta facendo è stata richiesta a lui da un client oppure è arrivata dall'anello (da un altro server).
Tutto sommato, questa distinzione veniva già fatta in "GestSer" tramite l'indicatore "GenOra".
Quindi, una volta implementata la variante dell'inserimento con priorità, tutto si riduce a aggiungere un "if GenOra" in GestSer, e quindi eseguire uno o l'altro inserimento.
Vedere il paragrafo sulle "collisioni di prenotazioni".

3.9) PRIORITÀ

> Siamo simili agli animali che vivono in comunità. [..] la nostra priorità sugli animali consiste prima di tutto nel nostro modo di vivere in società.
> (Albert Einstein)

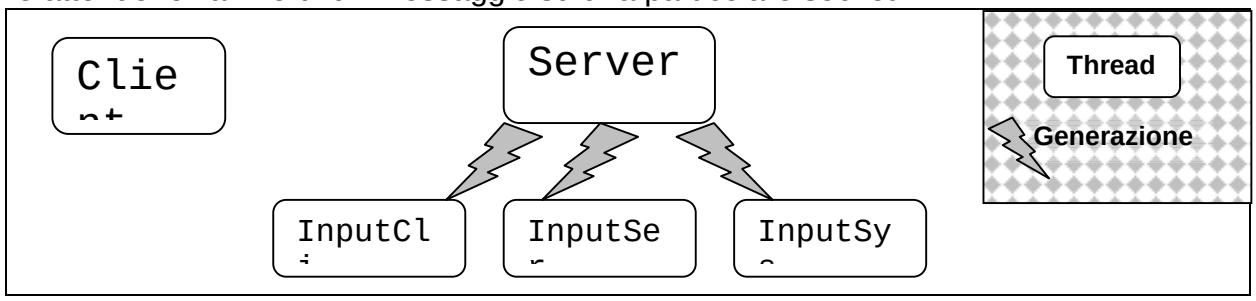
Anche la priorità l'ho incapsulata in un oggetto.
Potevo benissimo lasciarla come dato numerico, ma ho preferito fare così per restringere a un ambito "locale" la conoscenza della vera implementazione della "priorità".
In questo modo tutte le scelte implementative sono isolate all'interno di questa classe; all'esterno nessuna classe sa che la priorità è un numero, e neppure di che tipo (byte, intero o altro).
Poiché è memorizzata all'interno del data base, anche questo oggetto dovrà essere "serializzabile" e poiché dovrò inserirlo nei pacchetti (viaggeranno in datagrammi, insieme alle prenotazioni) occorre aggiungere il costruttore che legge da un DataInputStream e la "scriviInStream" che scrive l'oggetto priorità su un DataOutputStream.
All'interno della classe definirò due oggetti pubblici e "statici", chiamati "minPriorita" e "maxPriorita", che rappresenteranno la priorità minima e quella massima.
Ovviamente mi occorrerà almeno un metodo del tipo "greatestThan" per eseguire un confronto tra due priorità.
Vedere il paragrafo sulle "collisioni di prenotazioni" e sul "protocollo di aggiunta di un server".
La priorità è associata ai server, ma i server sono inseriti e rimossi "dinamicamente" nel sistema, perciò la priorità deve essere calcolata dinamicamente nel momento in cui avviene l'inserimento.
Per una descrizione dettagliata si veda il protocollo di "aggiunta di un server".

3.10) DIAGRAMMA DEGLI INNESCHI DEI PROCESSI NEL SERVER

Che me ne faccio di un libro senza figure e senza dialoghi?
(da "Alice nel paese delle meraviglie" di Lewis Carroll)

3.10.1) Situazione iniziale

Il programma server è "multiprocesso";
nel momento in cui è connesso nell'anello, ci sono tre processi (thread, per l'esattezza) che attendono l'arrivo di un messaggio su una particolare socket.



I tre thread generati aspettano per una particolare categoria di messaggi.

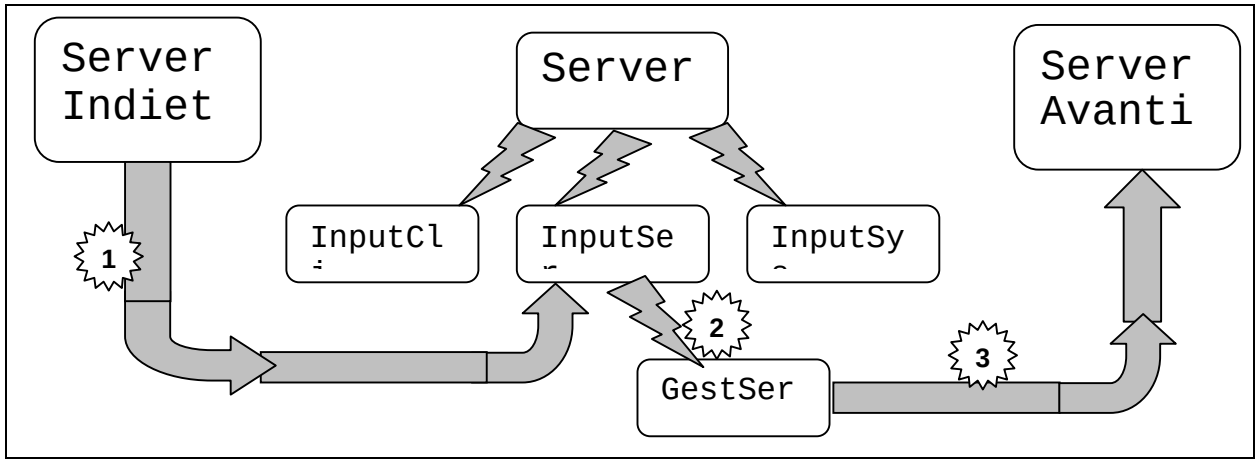
- **InputCli** attende su una socket-stream l'arrivo di messaggi da un client;
- **InputSer** attende su una datagram-stream l'arrivo di messaggi da un server;
- **InputSys** attende su una socket-stream l'arrivo di messaggi "di sistema" da un server.

Quando arriva un messaggio, esso non viene gestito direttamente dal thread di "Input", ma quest'ultimo genera un thread apposta a cui ne delega la gestione.
Il nuovo thread si chiama "Gest" seguito da "Cli", "Ser" o "Sys", a seconda di chi l'ha generato. Ognuna di queste classi di "Gest" può gestire particolari tipi di messaggi.
Nota: per le socket-stream, quando arriva un messaggio, la comunicazione viene fatta usando una "nuova socket" resa dalla primitiva di "accept" (quindi sarà il nuovo thread che si occuperà della lettura) mentre per la "datagram-stream" non c'è tale distinzione (sarà il processo "Input" che dovrà leggere il messaggio e passarlo quindi al "Gest" creato).

Più lontani si va e meno si apprende.
Per questo l'uomo saggio non cammina e arriva. (Lao Tze)

3.10.2) Arrivo di una Prenotazione da un Server

Al server arriva un messaggio di fare una prenotazione da parte di un altro server (ossia arriva un messaggio MessSerPrenota).
Tale messaggio viaggia in un datagramma e arriva perciò a "InputSer".
Viene generato un "GestSer" che proverà a scrivere la prenotazione nel DataBase, se ci riesce passerà il messaggio arrivato al suo server successore; altrimenti creerà un messaggio di "cancellazione" contenente la stessa prenotazione e lo passerà al suo predecessore (così tornerà a chi l'aveva generato, cancellandosi).

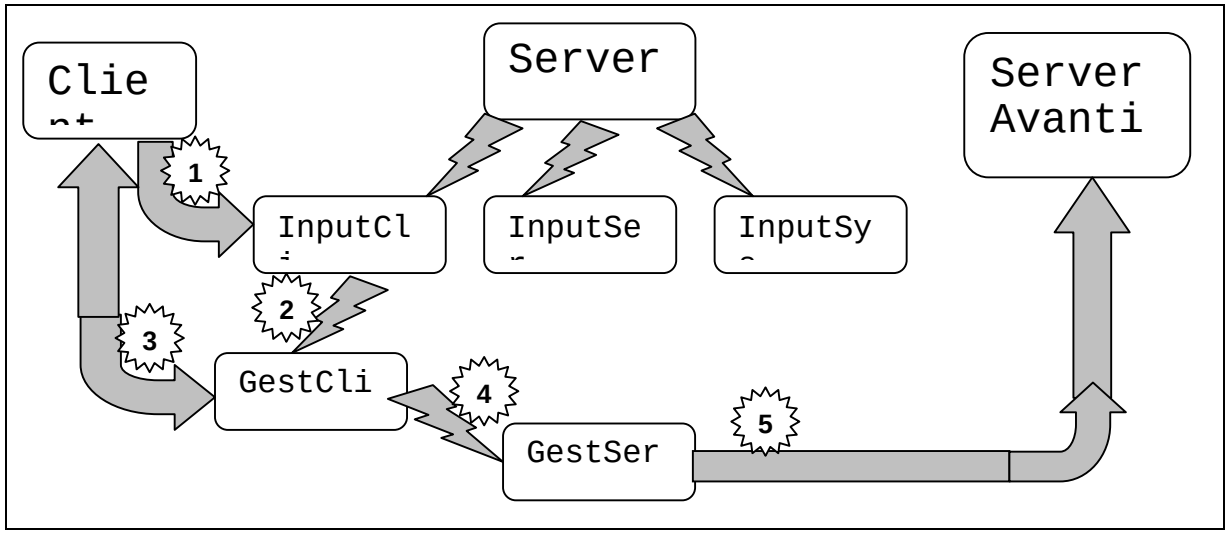


Da notare che il “GestSer” creato si occupa solo di trasmettere in avanti (se possibile) la prenotazione; dopo che ha inviato il messaggio (tramite un datagramma) al server “avanti” non si preoccuperà nemmeno di verificare se il destinatario l’ha ricevuto (vedere paragrafo sulle “ritrasmissioni”).

3.10.3) Arrivo di una Richiesta di Prenotazione da un Client

3.10.3.1) Parte Prima

Se il client invia una richiesta di prenotazione, sarà il thread “GestCli” a accettarla e a generare il thread “GestCli” che la servirà.
Nell’ipotesi che la prenotazione sia corretta e eseguibile, il “GestCli” dovrebbe fare la prenotazione e inviare un messaggio di prenotazione (non di “richiesta” ma di “prenotazione”) al suo server successore e attendere l’arrivo dello stesso dalla parte opposta (o dalla direzione stessa come “cancellazione”).
Poiché il passaggio di questo messaggio è identico a quanto deve fare un eventuale “GestSer” (generato all’arrivo di un messaggio da un server), ho pensato che anche “GestCli” possa generare un “GestSer” a cui passa un messaggio di “prenotazione” che ha appena creato (tale messaggio è diverso da quello di richiesta del client, ma ha gli stessi dati; in pratica uno è il MessCliRichiedi e l’altro il MessSerPrenota).
Occorre però che “GestSer” sappia se il messaggio che deve elaborare è stato appena generato da un “GestCli” oppure è arrivato dalla rete.
A tale scopo il suo costruttore, oltre al messaggio, dovrà avere un’indicatore che distingue la provenienza del messaggio (lo chiamerò “GenOra”): è stato generato ora da un GestCli oppure è arrivato dall’anello (tramite un InputSer).

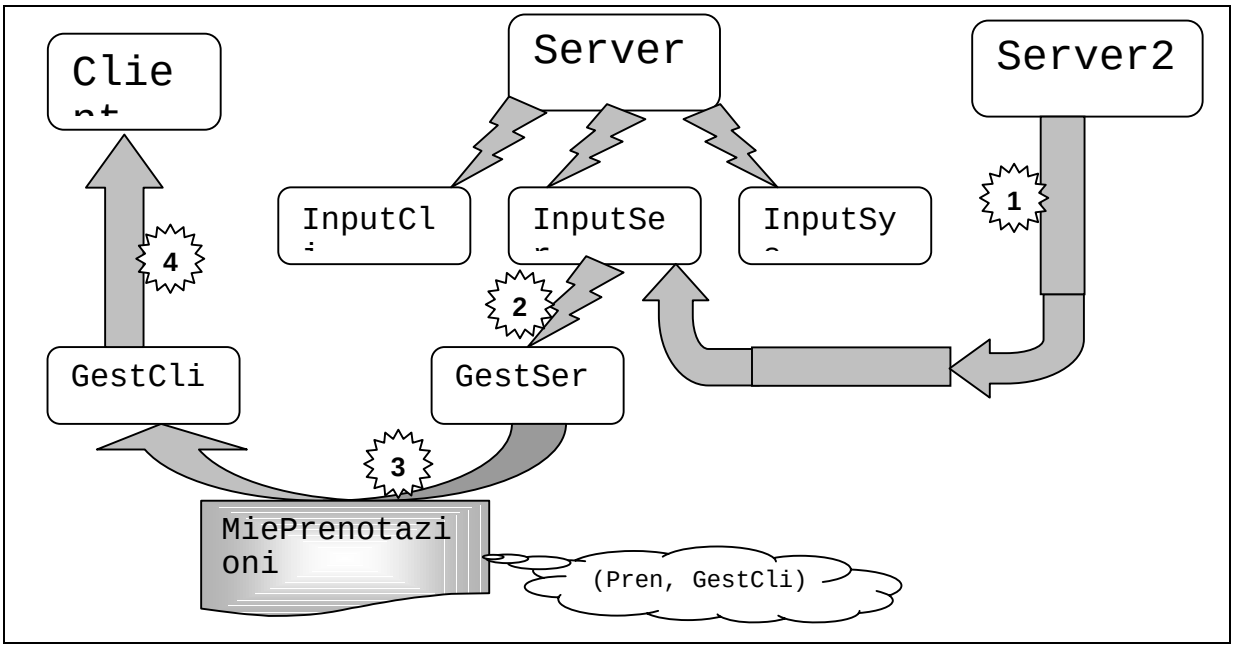


Facendo in questo modo, solo i thread di classe “GestSer” accederanno (in scrittura) al Data-Base!
Il GestSer creato trasmetterà il messaggio al server successore e si terminerà.
Il GestCli invece non termina subito, ma resta in attesa (si sospende) fino al ritorno del messaggio inviato.
In pratica, GestCli inserisce in “MiePrenotazioni” una coppia formata dalla prenotazione e da un riferimento a se stesso (come thread), genererà GestSer e si sospenderà in attesa di essere risvegliato.

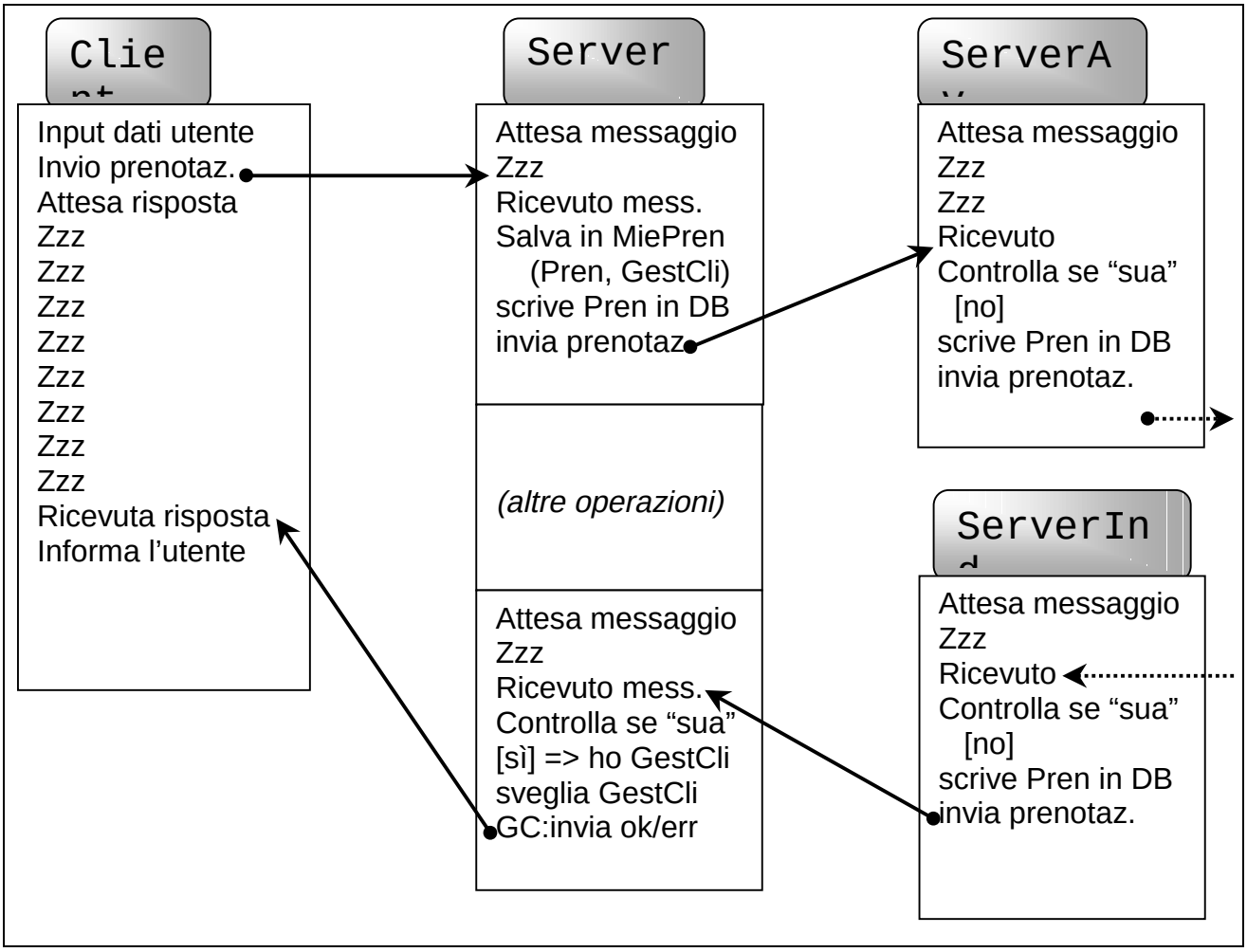
Nota: nel messaggio inviato da GestSer non c'è soltanto la prenotazione, ma ci sarà anche un campo che conterrà l'indirizzo della macchina server che ha generato il messaggio.
In tal modo ogni server può riconoscere la propria paternità su un messaggio (e quindi interromperne la circolazione).

3.10.3.2) Parte Seconda

Al ritorno del messaggio (dopo che ha fatto il giro dell'anello o al suo ritorno indietro come cancellazione) viene generato un nuovo GestSer.
Esso andrà innanzitutto a verificare la paternità del messaggio.
Se è stato in origine generato da lui, significa che c'è un "GestCli" che sta attendendo la risposta della fine della transazione.
Il GestSer accederà all'elenco "MiePrenotazioni" cercando una prenotazione uguale a quella che è contenuta nel messaggio appena arrivato.
La toglierà dopo aver letto il riferimento al thread "GestCli" associato.
Risveglierà tale "GestCli" comunicandogli il buono o cattivo esito della prenotazione.
La prenotazione è stata fatta se il messaggio è arrivato "in avanti" nell'anello; è andata male se è arrivata dalla direzione "indietro".
Per controllare ciò basta leggere il campo "direzione" contenuto nel messaggio.
Il GestCli comunicherà al Client l'esito della prenotazione.



3.11) PROTOCOLLO DI PRENOTAZIONE



3.12) PROTOCOLLO DI CANCELLAZIONE DI UNA PRENOTAZIONE

È essenzialmente identico al protocollo di prenotazione, eccetto che una cancellazione non può fallire. La cancellazione cancella uno spettatore solo se questo occupa effettivamente quel posto; se il posto è libero o occupato da qualcun altro, non si fallisce, ma si continua.

3.13) PRIMA DESCRIZIONE DEI PROCESSI

> Il vero inizio dell'attività scientifica consiste nel descrivere i fenomeni e quindi nel raggrupparli, classificarli e correlarli. Già allo stadio di descrizione non è possibile evitare di applicare certe idee astratte [..] che derivano da qualche parte, ma che certamente non sono basate unicamente sulle nuove osservazioni. (Sigmund Freud)

I vari processi in esecuzione faranno qualcosa come:

3.13.1) Client [processo principale cliente]

```
repeat
  inserimento dati utente
  inserimento prenotazione/cancellazione
  invio richiesta al server
  attesa conferma prenotazione fatta
  (con Time-Out: dopo di che, ritrasmette)
  (dopo 3 o 4 ritrasmissioni indica "errore")
  informa l'utente che Ok/Errore
until FinePrenotazioni
```

3.13.2) Server [processo principale server]

```
crea e innesca "InputCli" e "InputSer"
```

3.13.3) InputCli [Thread unico]

```
crea socket Stream
repeat
  accetta connessione [con blocco, crea newSock]
  crea e innesca "GestCli (newSock)"
forever
```

3.13.4) InputSer [Thread unico]

```
crea socket Datagram
repeat
  ricezione messaggio [con blocco]
  controllo mittente corretto (e' un "vicino" o sono io)
  Pacchetto ==> Mess
  crea e innesca "GestSer (Mess, GenOra=FALSE)"
forever
```

3.13.5) GestCli (newSock) [Thread multipli]

```
Stream<newSock> ==> MessCli
Se MessCli = Richiesta/Cancella Prenotaz.,
  estrai Prenotaz da MessCli
  crea Mess per i server (prenota/canc.Prenotaz; dir=Avanti)
  salva in MiePrenotaz. le info "Prenotaz, GestCli"
  TransazioneFatta := Errore
  ripeti
    crea e innesca "GestSer (Mess, GenOra=TRUE)"
    attesa (sospensione) di TransazioneFatta<>Errore[*] o TimeOut
  finché TransazioneFatta<>Errore
         o TimeOut
         o ho eseguito 3 volte il ciclo.
  se TransazioneFatta=SI', invia OK a client
  se TransazioneFatta=NO,  invia ERRORE a client
  se TransazioneFatta=Errore,  inizia controllo server caduto.
```

[*] Nota: sarà un "GestSer" (creato quando la prenotaz. ritorna) che comunicherà con tale thread invocando un suo metodo di "operazFatta" dando OK o Errore.

Nota2: in “TransazioneFatta” distinguo tre casi: SI’ e NO’ è la risposta data dal “GestSer” al ritorno del messaggio; Errore significa che il messaggio non è tornato!

```
Se MessCli = DammiSc
  costuisci elenco spettacoli (leggendoli dal DB)
  Stream<newSock> <== elenco spettacolo
```

3.13.6) GestSer (Mess, GenOra) [Thread multipli]

Nota: GenOra indica se questo thread è stato generato da un GestCli o da un InputSer, e quindi indica la provenienza del messaggio]

```
Se Mess = prenota/cancella
  Blocco finché le transazioni non sono “abilitate”
  preleva Prenotaz. da Mess
  Se GenOra = FALSE   (è arrivato da un ser.)
    controlla se Mess contiene indirizzo uguale al “locale”
    cerca Prenotaz in MiePrenotaz
    Se trovato,
      legge ThreadCli associato a Prenotaz
      StatoOp:= SI’ se direzione Mess è Avanti
                NO  se direzione Mess è Indietro
                (la prenotaz. è tornata come cancellazione)
      invoca ThreadCli.operazFatta (StatoOp)
      cancella Prenotaz da MiePrenotaz
      Termina thread (coorente).
  In ogni altro caso (ma sempre prenota/cancella)
  inserisce/cancella Prenotaz nel DataBase locale
```

Nota: OK se aggiungo una già esistente ma con stesso spettatore; ERR se cancello una che contiene un differente spettatore

Nota2: occorre distinguere anche l’inserimento con “priorità” (vedere paragrafo “collisione di prenotazioni”)

```
  Se OK o direz=Indietro,
    Passa (Mess)  [si usa la stessa direzione]
                  [ma si decrementa la “vita”]
  altrimenti
    MessBis:= cancella Prenotaz, con direz opposta a Mess
    Passa (MessBis)
Se Mess = aggancio
  se direz=Avanti e MioAvanti=ServerCaduto
    MioAvanti:=NuovoSer
    Mess2:=agganciatiAMe(MioIndir, conIndietro) con dir=Avanti
    Passa (Mess2)   [ora Avanti è il nuovo server]
  [duale per direz=Indietro]
Se Mess = agganciatiAMe
  se conDirez=conIndietro, mioIndietro:=Mess.MioIndir
  se conDirez=conAvanti,   mioAvanti  :=Mess.MioIndir
  riabilita passaggio transizioni
```

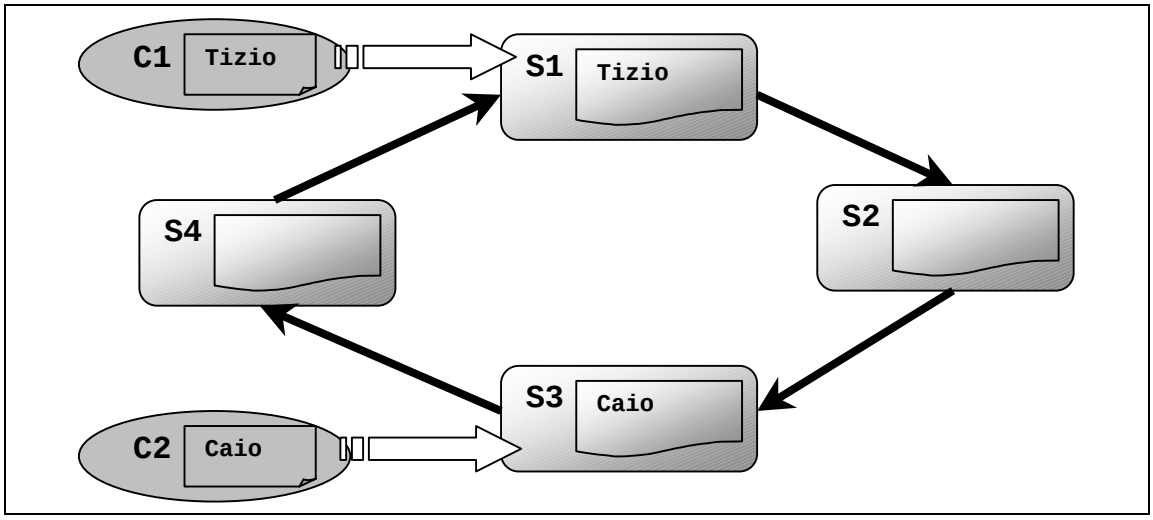
NB: il metodo “Passa (Mess)” invia il messaggio Mess al server successore controllando che non sia caduto;
in tal caso disabilita tutte le transazioni verso tale server e invia un messaggio di “aggancio” all’indietro verso il server che comunicava con il server caduto.
Dopo di cio’ non si ritorna subito, ma si attende finchè le transazioni non vengono riabilitate (dopo l’arrivo di un “agganciati a me”); dopo un certo Time-Out si ritrasmette il messaggio di aggancio.

3.14) COLLISIONE DI PRENOTAZIONI

> Più lontano accade una catastrofe o un incidente, più alto deve essere il numero dei morti e feriti perché faccia notizia.
> (Arthur Bloch)

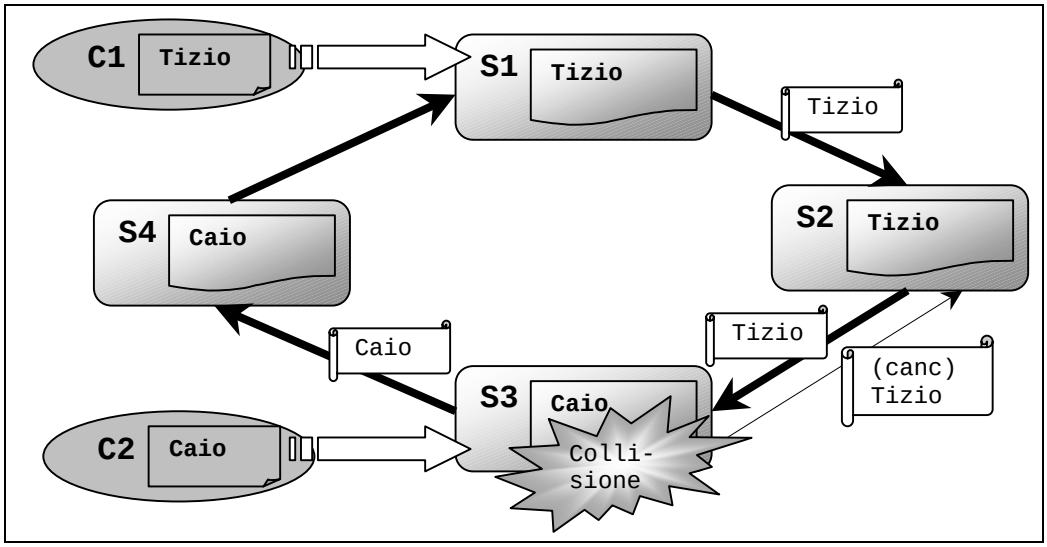
3.14.1) Il Problema

Può accadere che due o più client richiedano contemporaneamente di fare una prenotazione per lo stesso spettacolo e lo stesso posto ma per uno spettatore differente.
Ovviamente, solo uno dei due spettatori dovrà essere registrato mentre all'altro dovrà essere segnalato l'errore.
Se le richieste arrivano tutte sullo stesso server, non c'è problema poiché l'accesso in mutua esclusione al data base garantisce che solo la prima arrivata venga registrata mentre l'altra (tentando di sovrascrivere la precedente) fallirà.
Quando le prenotazioni arrivano su macchine differenti, un server non ha la possibilità di sapere se un altro sta facendo la stessa prenotazione oppure no.
Scriverà la prenotazione nel suo data base (il posto è "localmente" libero) e la invierà al suo successore.
Ma anche l'altro server sta facendo le stesse operazioni.
Mettiamo che due client (C1 e C2) richiedano a due server differenti (S1 e S3) di prenotare rispettivamente Tizio e Caio per uno stesso spettacolo e per lo stesso posto.



Ognuno dei server troverà "localmente" che il posto è libero, vi registrerà il proprio nuovo spettatore e invierà in avanti un messaggio di prenotazione contenente la prenotazione e le informazioni su se stesso (sulla macchina server).
Ognuno dei due messaggi sta circolando in senso orario (in avanti) e in breve tempo accadrà che uno dei due arriverà al server opposto.
Mettiamo che S3 riceva il messaggio di S1 che dice di prenotare "Tizio", ma localmente è già prenotato "Caio". La prenotazione fallirà!
Occorre **disfare** la prenotazione relativa al messaggio che è arrivato (non quella nella copia locale, per ora). Ovviamente, si dovrà andare in senso opposto!
Il server S3 bloccherà la circolazione del messaggio arrivatogli (lo "dimentica") e invierà in senso opposto un nuovo messaggio che indicherà la **cancellazione** della prenotazione ricevuta. Tale messaggio tornerà indietro, fino a S1 (che lo riconoscerà come proprio) e capirà che la prenotazione non è stata fatta (e ne informerà il client C1)!
Analogamente accadrà per la prenotazione di C2.

Nota: quando un server intermedio riceve un messaggio di cancellazione, cancellerà la prenotazione relativa dal proprio data base… ma solo se in esso è contenuta effettivamente quella prenotazione.
Se il posto è prenotato da un altro, non lo cancellerà (e non lo considererà un errore).



Ma non è simpatico che tutte e due le prenotazioni falliscano!
Così facendo entrambi i client vengono avvisati che il posto è già stato occupato da un altro, mentre in realtà è ancora libero!

> "Ascoltate dunque", disse Trasimaco, "io affermo che la giustizia o il diritto è semplicemente ciò che è nell'interesse della parte più forte.
> (da "la repubblica" di Platone)

3.14.2) Cosa si vuole

Si vuole che non tutte le prenotazioni falliscano, ma che una e una soltanto venga registrata da tutti i server.

3.14.3) Come Fare? (discussione sull'uso della priorità)

L'aspetto delicato è che bisogna garantire che tutti i server si comportino nello stesso modo e che la decisione di ognuno sia presa il più localmente possibile.
Non è proponibile che per ogni "collisione" tutti i server (in numero non predefinito) si scambino messaggi per stabilire chi sia "arrivato prima".
Così com'è il protocollo di prenotazione causa un fallimento di tutte le prenotazioni che collidono. Con piccole modifiche si può ovviare a ciò.
Ho pensato a una soluzione facile facile: assegnare una **priorità a ogni server** dell'anello.
L'idea è alquanto banale:

1. a ogni macchina è assegnato (al momento del suo inserimento nell'anello) un indicatore di "priorità";
2. a ogni messaggio di prenotazione aggiungo il campo "priorità", in cui viene inserita la priorità del server che ha generato quella prenotazione;
3. quando registro la prenotazione nel data base, le associo anche il valore della priorità del server che l'ha generata (che è contenuta nel messaggio);
4. quando due prenotazioni collidono, ogni server deciderà (**localmente**) quale tenere basandosi su un semplice confronto tra i valori di priorità della nuova prenotazione e di quella che è già stata memorizzata nel data base.

3.14.4) Le Modifiche da Fare

Bisogna rivedere un po' le operazioni sul data base, per contemplare la priorità della nuova prenotazione da inserire e di quella che è già presente nel data base (vedere parte la finale del paragrafo "data base").
Dopo aver modificato il data base aggiungendo l'operazione di inserimento con priorità occorre modificare "GestSer" aggiungendo un "if GenOra" per distinguere il tipo di inserimento da fare; al posto di:

```
inserisci Prenotaz nel DataBase locale
```

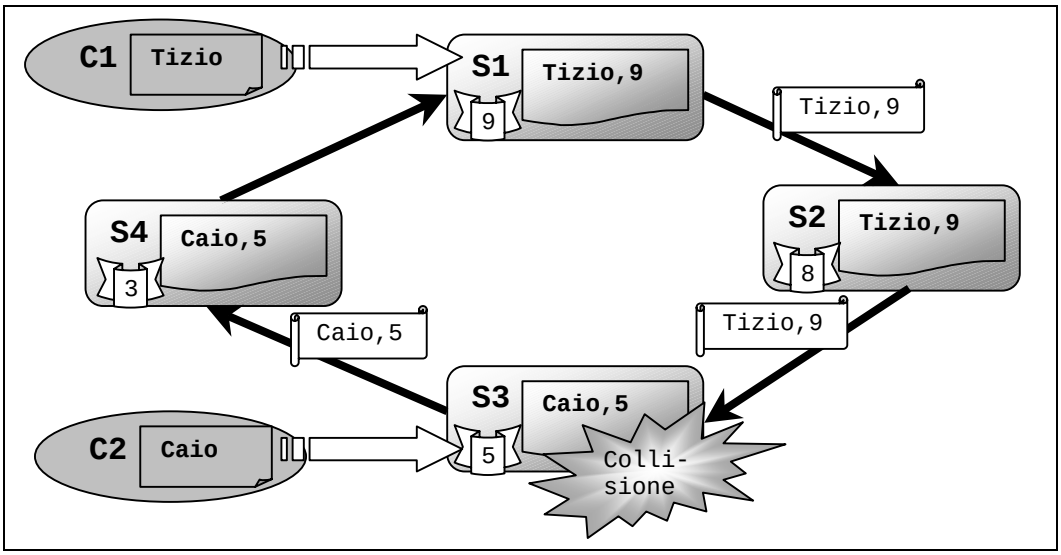
si dovrà scrivere:

```
se GenOra  inserisci Prenotaz nel DataBase locale
altrimenti inserisci Prenotaz con Priorità nel DataBase locale
```

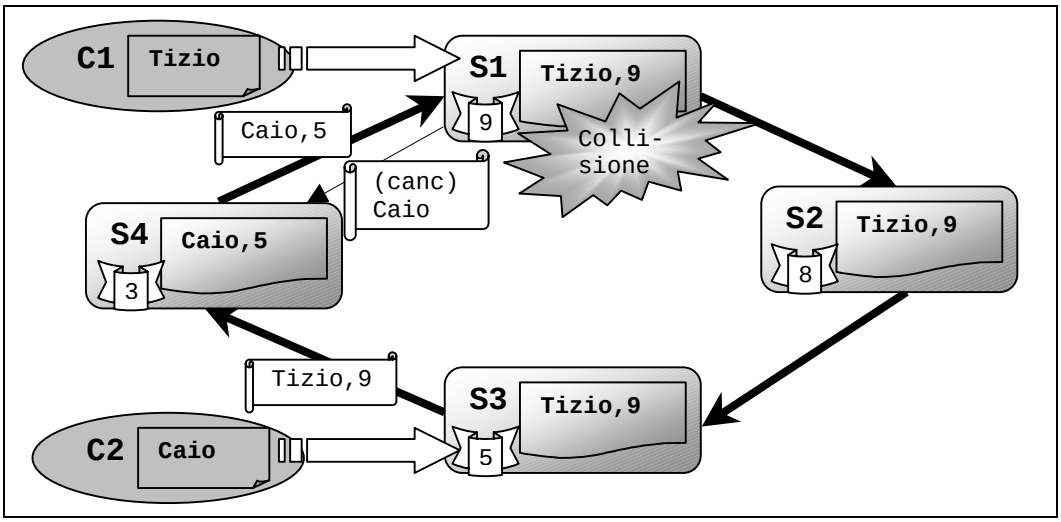
Questa modifica è necessaria perché se un server inserisse sempre usando la priorità, ogni prenotazione richiesta da un client al server più prioritario finirebbe con sovrascrivere qualsiasi altra prenotazione (anche quelle già terminate).

3.14.5) Il Nuovo Comportamento

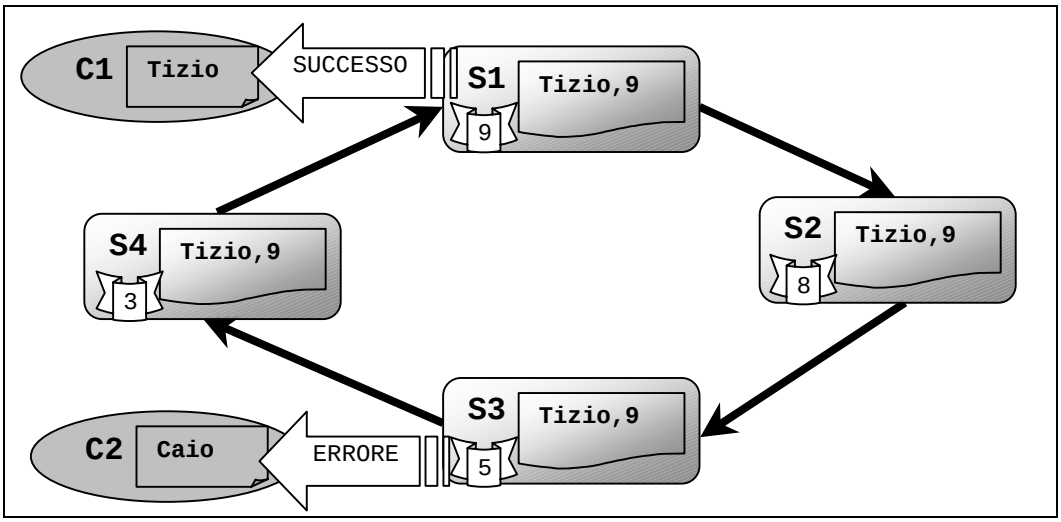
Quando un server inserisce una prenotazione che gli è appena stata richiesta da un client non deve guardare la priorità; o il posto è libero (o occupato dallo stesso spettatore), e quindi lo occupa, oppure è occupato, e allora non può farci niente (la prenotazione fallirà).



In questo nuovo schema, a ogni server è associato un valore di priorità e in ogni messaggio e in ogni data base allo spettatore è associata la priorità del server che ha ricevuto la prenotazione da parte del client.
Quando si ha la collisione su S3, la prenotazione di Tizio sovrascriverà quella di Caio poiché è più prioritaria (9 è maggiore di 5) e il messaggio di Tizio continuerà a circolare.
Intanto il messaggio di prenotazione di Caio continuerà a circolare... finché non arriverà a S1, quando collliderà con Tizio. Confrontando 5 con 9, si vede che è maggiore il secondo, cioè la priorità dello spettatore che era già nel data base.



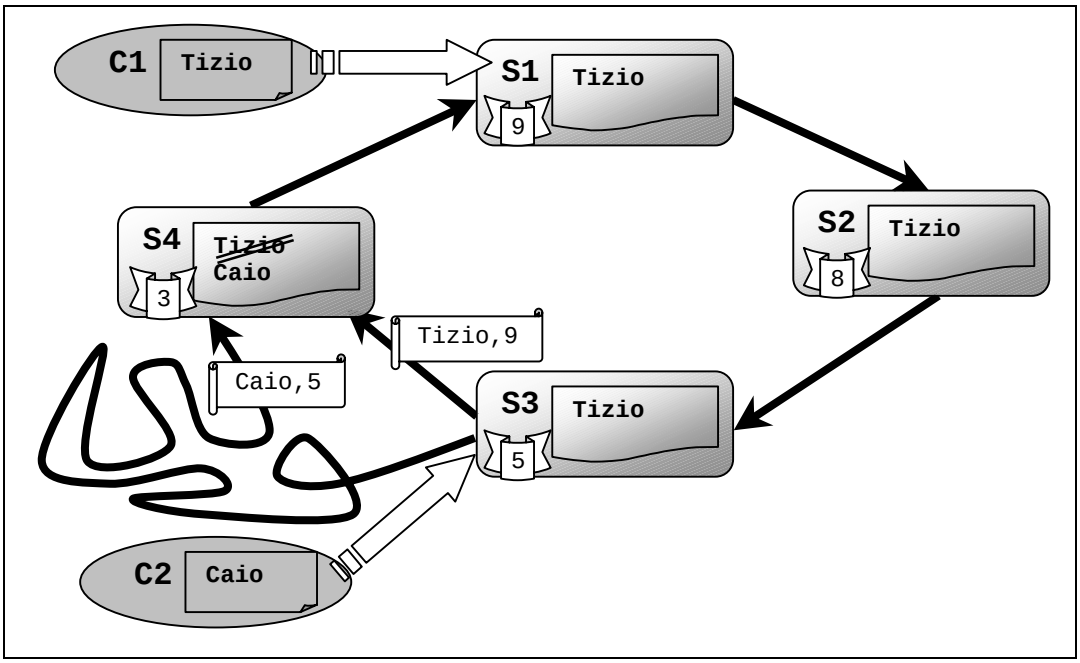
La prenotazione di Caio va disfatta; si genererà un messaggio di cancellazione in senso opposto. Questo, farà cancellare la prenotazione di Caio da S4 e poi da S3.
Il server S3 riconoscerà la cancellazione come una sua prenotazione fallita e informerà il suo cliente che non è stato possibile prenotarlo.
Guardiamo S4, che per il momento ha registrato “Caio,5”. Vediamo cosa accade se arriva prima il messaggio di cancellazione (per Caio) o quello si prenotazione (per Tizio).
Se gli arriva prima quello di cancellazione, procederà a cancellare Caio dal suo data base; successivamente (all’arrivo dell’altro messaggio) provvederà a prenotare il posto (ora libero a nome di Tizio.
Se arriva prima la prenotazione di Tizio, questa sovrascriverà quella di Caio (ha priorità maggiore) e quindi procederà in avanti; la cancellazione che arriverà poco dopo non andrà a cancellare la prenotazione di “Tizio”, perché è una cancellazione “speciale” (cancellerà “Caio” solo se il posto è prenotato a lui; se è stato dato a un altro, non farà nulla).
Intanto la prenotazione di Tizio continua; non appena arriverà a S1, verrà bloccata e il cliente sarà informato dell’avvenuta prenotazione.



Tutti i server ora hanno la prenotazione a nome di Tizio.
La prenotazione di Caio è fallita.

3.14.6) Nota sulla Priorità nel DataBase

Una cosa da notare è la necessità della priorità contenuta nel data base.
Può sembrare inutile, visto che si può pensare che il primo server su cui ci sarà una collisione sarà uno di quelli che ha messo in circolo uno dei messaggi di prenotazione.
Si può essere tentati a usare la priorità del server su cui si ha la collisione come la priorità della prenotazione contenuta nel data base.
Questo però non è vero! Usando socket a datagrammi, non esiste un percorso fisico prestabilito su cui viaggia il messaggio, perciò i vari messaggi nell'andare da un nodo al successivo possono prendere strade diverse e quindi arrivare in ordine differente rispetto a quello con cui sono stati trasmessi.
Può perciò accadere che un messaggio "sorpassi" un altro messaggio
Riprendiamo l'esempio precedente.
Dopo la collisione di Tizio su S3, Tizio "vince" e viene scritto nel data base di S3 e viene spedito a S4. Ipotizziamo che il messaggio contenente "Caio" abbia preso una strada molto lunga per andare da S3 a S4, mentre "Tizio" abbia avuto la fortuna di percorrerne una molto breve.
Su S4 arriverà il messaggio con "Tizio" che verrà registrato nel data base, e poi arriverà "Caio" che genererà una collisione.
Se usassi la priorità del server S4, andrei a confrontare 3 con 5 e perciò vincerebbe "Caio" che andrebbe a sovrascrivere "Tizio".



Ma una volta arrivato a S1, colliderebbe con Tizio e vincerebbe "Tizio" (9 contro 5) perciò la prenotazione di Caio sarebbe disfatta, lasciando in S4 il posto libero!
È perciò necessario associare a ogni spettatore la priorità del server che ha richiesto la prenotazione.

3.15) PROTOCOLLO DI AGGIUNTA DI UNO SPETTACOLO

> Dal momento che si aggiunge continuamente qualcosa che prima non c'era, la ragione è 'progressiva', cioè la conoscenza umana continua a espandersi e quindi ad 'andare avanti'.
> (da "il mondo di Sofia" di J. Gaarder)

L'aggiunta di uno spettacolo è un'operazione che può fare solo un amministratore (da uno dei server), perciò il messaggio (con l'informazione opportuna) viaggerà in un datagramma.
Il protocollo è in tutto e per tutto identico a quello di prenotazione, eccetto che, inserendo solo l'informazione sullo spettacolo, non può esserci alcuna collisione.
Rispetto al protocollo di prenotazione cambierà solo l'operazione da eseguire sul data base.
L'aggiunta può fallire se viene superato il time-out (di attesa per il ritorno del messaggio dall'anello) perciò l'utente (in questo caso, l'amministratore) va avvisato dell'avvenuta registrazione (o meno) dello spettacolo.
Il thread che attende il time-out non è un GestCli bensì lo stesso processo "Server".
Infatti, ogni server può aggiungere un solo spettacolo alla volta; non può farne uno nuovo fintantoché il nuovo spettacolo non è stato memorizzato da tutti i server.
Da notare che, mentre il thread server è in attesa, tutti i thread di "Input" e di "Gest" generati dal server non vengono sospesi! Il server continua a funzionare mente sta attendendo la conferma della registrazione.
Dunque è il "Server" che genera il "GestSer"; occorre perciò distinguere in "GestSer" se dovrà andare a risvegliare un "GestCli" o il "Server".
Poiché il server è unico, non mi serve una tabella (come ho fatto per "MiePrenotazioni") ma basta un oggetto di classe "Spettacolo".
Il messaggio inviato agli altri server conterrà lo spettacolo che memorizzerò lì dentro; all'arrivo guarderò se sono uguali.
Insomma, l'idea di base non cambia... cambiano solo le caratteristiche degli oggetti usati.

3.16) PROTOCOLLO DI CANCELLAZIONE DI UNO SPETTACOLO

La cancellazione di uno spettacolo è simile alla precedente, cambia solo l'operazione da eseguire sul data base.

3.17) NOTA SUI PROTOCOLLI BASATI SU DATAGRAMMI

Ai quattro protocolli precedentemente mostrati corrispondono le classi di messaggi discendenti da "MessSerDg".
Non c'è moltissima differenza tra le operazioni che devono svolgere, perciò ho inglobato tutta la loro semantica in un unico metodo all'interno di "GestSer", che ho chiamato "Servi_MessSerDg_PrenCanc_AggCanSc".
Devo anche far notare uno strano fenomeno che ho incontrato...
L'oggetto "InputSer" gestisce l'accettazione dei messaggi che arrivano su datagrammi.
Quando viene fatta la "receive" per attendere i dati occorre usare un oggetto di classe DatagramPacket in cui verranno memorizzati i dati arrivati.
In pratica per ricevere un datagramma occorre fare così:

```
Pacchetto = new DatagramPacket (new byte[DIM], DIM);
Sock      = new DatagramSocket (PORTA);
Sock.receive (Pacchetto);
```

Si può notare che la "receive" usa come ingresso un pacchetto e "Pacchetto" viene usato una sola volta e poi non serve più. È intuitivo pensare di creare un solo pacchetto e di darlo di volta in volta alla "receive", evitando così creazioni inutili di nuovi pacchetti.

Facendo così, la “receive” **non funziona** più come dovrebbe.
Infatti a tutti i pacchetti successivi al primo viene attribuita una stessa dimensione... quella del primo pacchetto.
O almeno questo è ciò che accade nel JDK v1.1 su piattaforma Windows.
Bisogna perciò creare un nuovo pacchetto ogni volta che si esegue la “receive”; esso può benissimo essere creato usando lo stesso array (che può essere creato una sola volta).

3.18) DEADLOCK?

> Se non ci fossero stati anche in natura fattori limitanti, una specie animale o vegetale si sarebbe diffusa in tutta la Terra. Solo perché esistono specie diverse, esse si tengono in scacco a vicenda. (Benjamin Franklin)

In ogni momento, un server ha una sola istanza dei thread di classe “Input” mentre può avere nessuna, una o più istanze dei thread di classe “Gest”.
Questi thread usano oggetti che sono presenti nel server in un’unica istanza, perciò l’accesso deve essere mutuamente esclusivo.
Difatti tutti i metodi di accesso all’oggetto “DataBase” sono dichiarati “synchronized”, ossia prima di poter eseguire un metodo ci si deve impossessare del “monitor” associato all’oggetto.
Java introduce automaticamente un monitor in ogni oggetto e genera automaticamente il codice per la sua acquisizione all’inizio dei metodi “synchronized” e il suo rilascio alla fine degli stessi.
In questo modo, quando un thread sta scrivendo nell’oggetto DataBase, ogni altro thread che provasse a accedervi verrebbe “sospeso”, ossia inserito nella coda associata al monitor dell’oggetto “DataBase”.
Quando il thread termina di usare il DataBase (esce dal metodo dell’oggetto), verrà controllata la coda dei thread in attesa e, se non è vuota, verrà risvegliato il primo thread.
Con i monitor si garantisce la mutua esclusione nell’accesso al DataBase.
Non può esserci deadlock perché nessun thread tenta di acquisire più di un oggetto (protetto da monitor) alla volta, e non si trova mai nella condizione di tentare il possesso di un oggetto senza avere prima rilasciato quello eventualmente in suo possesso.
Infatti, i metodi dell’oggetto DataBase non invocano metodi di altri oggetti che sono a loro volta protetti da monitor, e così tutti gli altri oggetti (e thread).
Gli oggetti “DataBase” e “MiePrenotazioni” sono oggetti distinti e mutuamente esclusivi tra di loro ma non si usano a vicenda.

3.19) SOSPENSIONE DI UN THREAD CON TIME-OUT

> Tempo per te e tempo per me,
> E tempo anche per cento indecisioni,
> E per cento visioni e revisioni,
> Prima di prendere un tè col pane abbrustolito.
> (Thomas Stearns Eliot)

Sia nel lato client sia in quello server è necessario poter sospendere un thread che è in attesa di dati finché questi non sono disponibili. Ovviamente non è garantito che prima o poi i dati arriveranno, perciò si dovrà fare una sospensione non definitiva, ossia una sospensione con "**time-out**"; si fissa a priori un "tempo massimo" che il thread è disposto a aspettare e al suo scadere il thread verrà risvegliato in ogni caso (anche se i dati non sono arrivati!).

3.19.1) Sospensione del Thread su un Monitor

A parte la sospensione temporanea di un thread che si ha quando questo invoca un metodo "synchronized" di un altro oggetto, ci interessa qui sospendere un thread su un oggetto. In particolare, ci interessa bloccare tutti i processi "GestSer" durante le operazioni di aggiunta di un altro server (vedere "blocco delle transazioni di un server" più avanti).
A tale scopo costruirò un nuovo oggetto (che chiamerò "GestTransazione") il cui scopo è quello di tener traccia dello stato delle "transazioni", ossia dei thread "GestSer" che stanno eseguendo in ogni momento.
Per sospendere un thread "GestSer" sul "monitor" associato all'oggetto "GestTransazioni" basterà eseguire la primitiva "**wait**" su tale oggetto; per risvegliare uno dei thred che sono in coda (su tale monitor) si usa la primitiva "notify" (o "**notifyAll**" per risvegliare tutti i thread in coda).
Un particolare importante, prima di poter invocare la "wait" occorre avere il possesso del monitor dell'oggetto. La si invocherà da metodi dell'oggetto stesso definiti come "synchronized" (ossia quando si è al loro interno si è i possessori del monitor).
Per attendere "con time-out" si usa una variante della "wait" che ha come parametro il numero di millisecondi da aspettare al massimo.

3.19.2) Sospensione del Thread in Attesa di Ricevere dei Dati

Sia il processo "ClientApplet" sia i thread che derivano da "Protocollo" (cioè "Server" e "GestSys") si trovano nella condizione di aspettare dei dati via socket-stream.
Per default la lettura è bloccante; questo vuol dire che se i dati non sono pronti, il thread è sospeso finché non arrivano. Nel caso non arrivassero, il thread non verrebbe mai più risvegliato!
Il modo migliore per aggirare il problema sta nell'invocare il metodo "**setSoTimeout**()" sulla socket da cui successivamente si andrà a leggere. Tale metodo permette di definire il tempo massimo (time-out) di attesa dei dati.
Nel momento in cui il thread esegue la lettura, se i dati non sono pronti verrà sospeso.
Se i dati arrivano entro il tempo massimo, tutto funziona regolarmente; ma se non arrivano entro tale tempo il thread è risvegliato e l'azione di lettura genererà un'eccezione di classe "InterruptedIOException".

3.20) BLOCCO DELLE TRANSAZIONI DI UN SERVER

> Gli ideali maggiormente appaganti sono quelli che non dispongono di incarnazioni concrete atte a bloccare la fantasia. (Aldous Huxley)

In ogni server deve poter essere necessario intraprendere delle operazioni che vanno a modificare i riferimenti ai server vicini oppure che, mentre si eseguono, non si vuole che il data base venga modificato (per es. l'azione di aggiunta di un nuovo server necessita di tutte e due le caratteristiche).
Il metodo più facile per fare ciò è di bloccare temporaneamente i thread chiamati in causa.
Qui gli unici thread che modificano il database sono i "GestSer", mentre quelli che accedono ai riferimenti ai server vicini sono i "GestSer", i "GestSys" e il "Server".
Voglio che tali thread vengano bloccati, senza che eseguano un'attesa attiva: li sospenderò su un particolare monitor.
Ho pensato di "isolare" la logica della sospensione all'interno di un oggetto che chiamerò "**GestTransazione**", che server per consentire o negare le "transazioni".
È in tale oggetto che memorizzerò i riferimenti (indirizzo e priorità) ai server vicini.
Innanzitutto servono due metodi che chiamerò "iniziaTransazione" e "terminaTransazione", che devono essere chiamati all'inizio e alla fine da ogni "GestSer".
Tali metodi dicono al gestore delle transazioni quando è in atto un'operazione "delicata".
All'interno della "iniziaTransazione" si controllerà se è possibile eseguire una transazione (ossia se la direzione interessata non è stata "bloccata") e in tal caso il metodo ritorna (dopo aver incrementato un contatore interno) e il processo chiamante continuerà a eseguire; se invece la direzione è "bloccata", il thread chiamante viene sospeso (con una "wait") sul monitor dell'oggetto "GestTransazione".
La "terminaTransazione" serve solo per aggiornare il contatore interno del numero di thread che al momento stanno facendo un'operazione "delicata".
Definirò quindi i due metodi "bloccaTransazione" e " abilitaTransazione" (parametrici rispetto a una direzione, avanti o indietro) che servono bloccare o sbloccare le "transazioni" in una direzione.
La "bloccaTransazione" dovrà mettere il "blocco" sulla direzione voluta ma non si potrà ritornare subito... occorre attendere la terminazione di tutti i thread che hanno cominciato un'azione "delicata" ma che non l'hanno ancora terminata.
La "abilitaTransazione" dovrà togliere il "blocco" alla direzione e risvegliare tutti i thread che sono stati sospesi sul monitor dalla "iniziaTransazione" (si userò una "notifyAll").
Tutti i metodi descritti devono essere definiti "synchronized" perché devono essere acceduti in mutua esclusione (e per fare la "wait" occorre possedere il monitor dell'oggetto).
Un altro metodo utile è "hoTransazioneBloccata" che serve per testare se una certa direzione è o meno "bloccata".
Inoltre il metodo "AttendiSeTransazBloccate" esegue la sospensione del thread sul monitor se la transazione è bloccata, ma (a differenza dell'"iniziaTransazione" non modifica il contatore dei thread). Questo è utile per sospendere un "GestCli" in modo che non si metta a fare ritrasmissioni inutili se le transazioni interne sono bloccate.

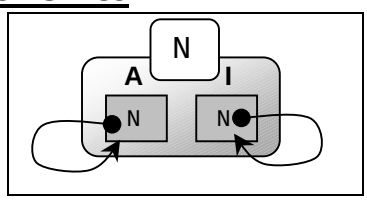
3.21) PROTOCOLLO DI AGGIUNTA DI UN SERVER

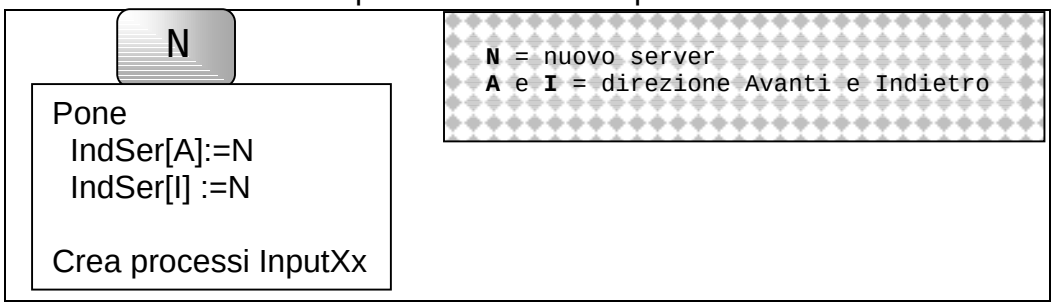
> Se raccogli un cane randagio e gli dai da mangiare non ti morderà: ecco la differenza fondamentale fra il cane e l'uomo. (Mark Twain)

Per aggiungere un nuovo server, si può pensare che questo conosca l'indirizzo di almeno uno dei server esistenti e che comunichi con questo per lo scambio dei dati.
Innanzitutto, il nuovo server non ha nessuno dei tre thred principali in esecuzione.
La sola cosa che può fare è di inviare messaggi (al server noto) tramite la socket-stream a cui fa capo il thread "InputSys" nel server attivo.
Il messaggio "MessSerAggMe" servirà a dire al server attivo che il nuovo server si vuole aggiungere.
È al momento dell'aggancio all'anello che viene stabilita la priorità del nuovo server.
Bisogna distinguere in tre casi, a seconda del numero di server che sono presenti nell'anello a cui ci si vuole agganciare: nessuno, uno, da due in su.

3.21.1) Protocollo di Aggiunta di un Server come un Server Unico

Questo è il caso iniziale del sistema; non c'è nessun server collegato in anello.
Il server da aggiungere sarà il primo server costituente l'anello.
Ho un anello di un solo server.
In pratica, i suoi riferimenti ai server vicini conterranno l'indirizzo di se stesso! Il protocollo sarà semplicissimo:

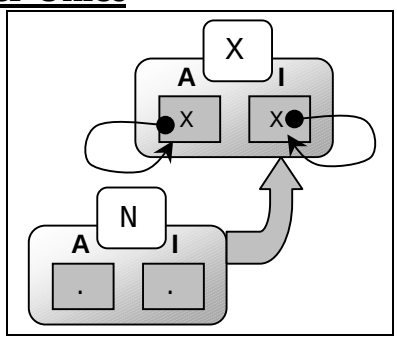




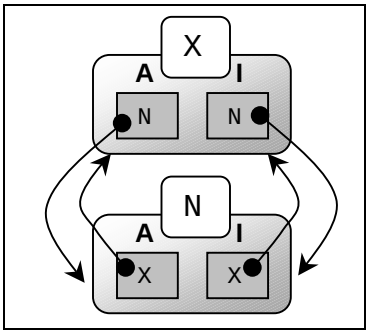
La priorità del server unico sarà la massima possibile.

3.21.2) Protocollo di Aggiunta a un Anello con un Server Unico

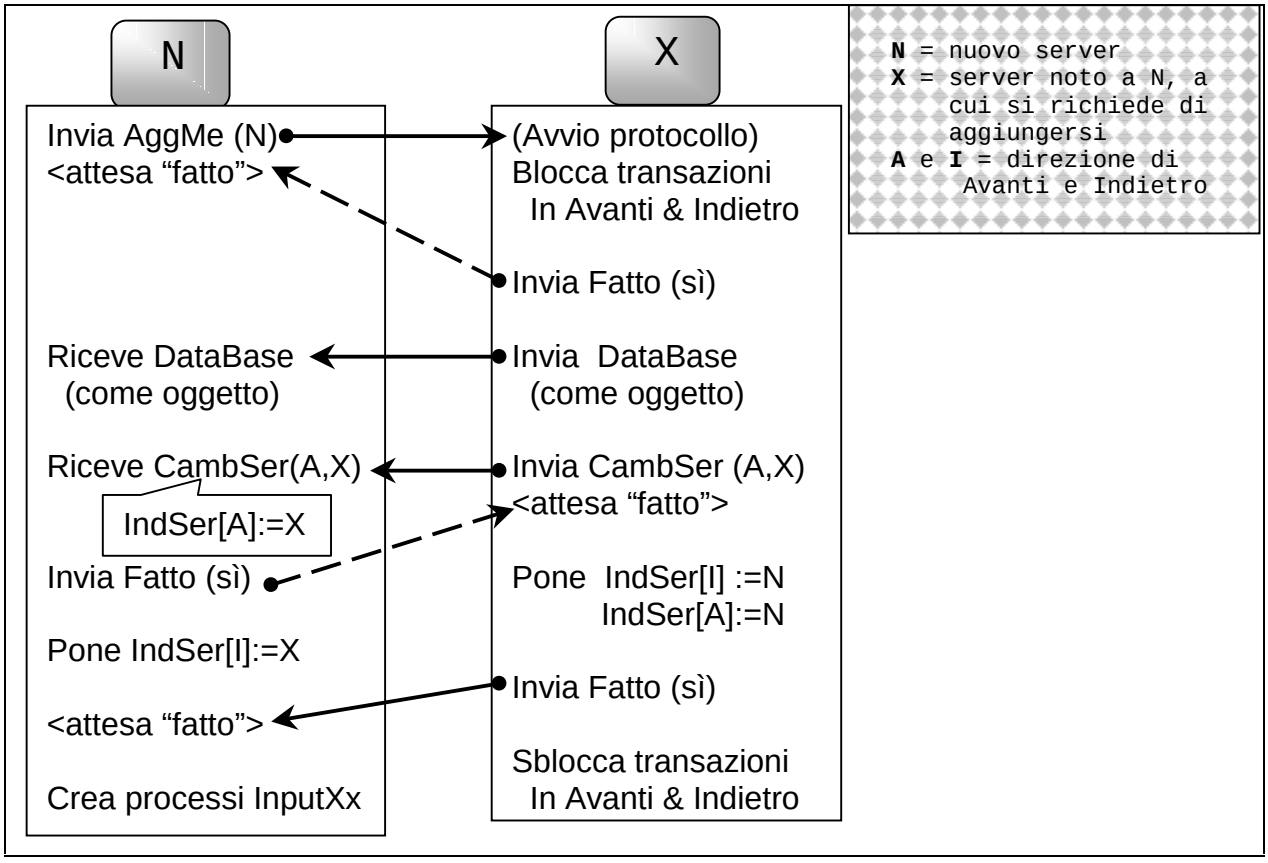
In questo caso il nuovo server deve aggiungersi a un anello formato da un solo server (quello creato nel caso precedente).
Il nuovo server conoscerà l'indirizzo dell'unico server esistente e comunicherà con esso.
Il problema principale sta nell'inibire qualsiasi modifica al data base locale del server X fintantoché il nuovo server N non si è aggiunto all'anello.



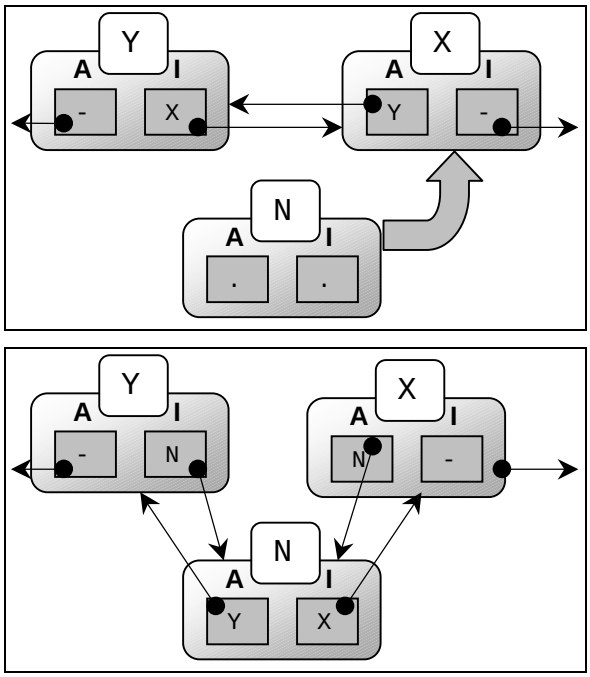
Per far ciò basta “sospendere” tutti i thread GestSer del server X.
Il Server X dovrà “bloccare le proprie transazioni” sia in avanti sia indietro, cioè sospendere ogni thread GestSer.
Quando N sarà stato aggiunto, si riabiliteranno le transazioni e tutti i thread GestSer potranno continuare la loro esecuzione e andare a scrivere nel data base e quindi mandare al nuovo server il messaggio di prenotazione relativo.
Da notare che, una volta attivato il blocco delle transazioni, si deve attendere che tutti i thread “GestSer” correntemente attivi (che stanno ancora eseguendo) terminino.



Se non lo facessi, potrebbe accadere che il data base venga modificato dopo che l’ho trasferito al nuovo server, o che quando GestSer passa in avanti il messaggio usi un riferimento al server “avanti” non valido (quello vecchio, e quindi il messaggio salterebbe il nuovo server portando il data base in uno stato inconsistente, oppure quello nuovo, venendo ignorato, perché non sono ancora stati creati i thread di “Input”).
A tal scopo ho creato la classe “GestTransazioni” (vedere “blocco delle transazioni”)
Ecco come ho pensato di realizzare il protocollo.

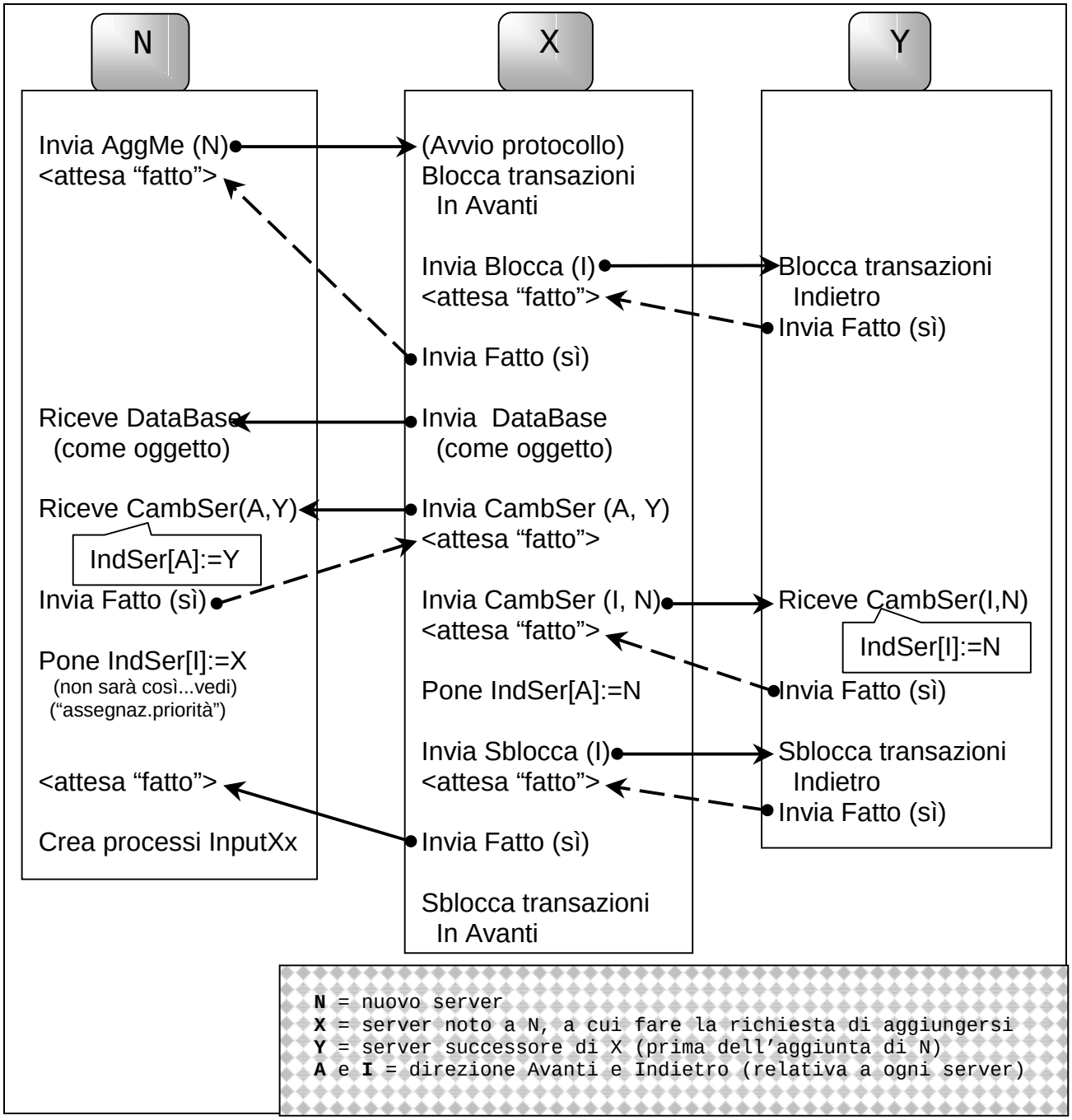


3.21.3) Protocollo di Aggiunta a un Anello di Almeno Due Server



Questo è il caso più generale, perché bastano due server aggiungerne uno nuovo.
Da notare che Y è il successore di X, prima dell'aggiunta di N (che diventerà il nuovo successore di X).
In questo caso bisogna bloccare tutti i messaggi che viaggiano su datagrammi (gli unici che possono scrivere sul data base), ossia sospendere tutti i nuovi thread "GestSer" (che vengono via via creati all'arrivo di messaggi dall'anello o da parte di thread GestCli).
È molto simile al caso precedente, ma non basta bloccare i GestSer locali.
Se facessi così, nella configurazione finale, quando andrò a risvegliare tutti i thread GestSer sospesi, mi accorgerei che i messaggi "in avanti" andranno al nuovo server Z ma quelli "indietro" non avrebbero modo di andare a modificare il data base in Z (andando "indietro" e trovandosi X, presuppongono di essere già passati per Z).
Occorre che tali messaggi vengano bloccati non in X ma nel suo corrente successore Y.
Riassumendo, devo **bloccare tutti i messaggi "in avanti" nel server X e tutti quelli "indietro" nel server Y**.
Da notare che dopo l'invio di un messaggio ci si aspetta l'arrivo di un messaggio "MessSerFatto" contenente l'indicazione se l'ordine precedentemente inviato è stato o meno eseguito.
Il server che attende un "MessSerFatto", attende con "**time-out**".
Allo scadere di tale tempo senza che sia arrivata la risposta, il server procede a **annullare tutte le operazioni fatte**!
Questo è molto importante, perché il protocollo deve inizialmente bloccare le "transazioni" e riabilitarle solo al termine di tutte le operazioni.
Se si restasse bloccati indefinitamente in attesa di un "fatto", il server non farebbe più circolare i messaggi nell'anello e quindi l'intero sistema risulterebbe non funzionante!
Nel momento in cui qualcosa va storto (un server risponde con un "fatto no" o non arriva la risposta entro il time-out) il server "X" deve annullare tutte le operazioni fatte su se stesso e sul server Y.

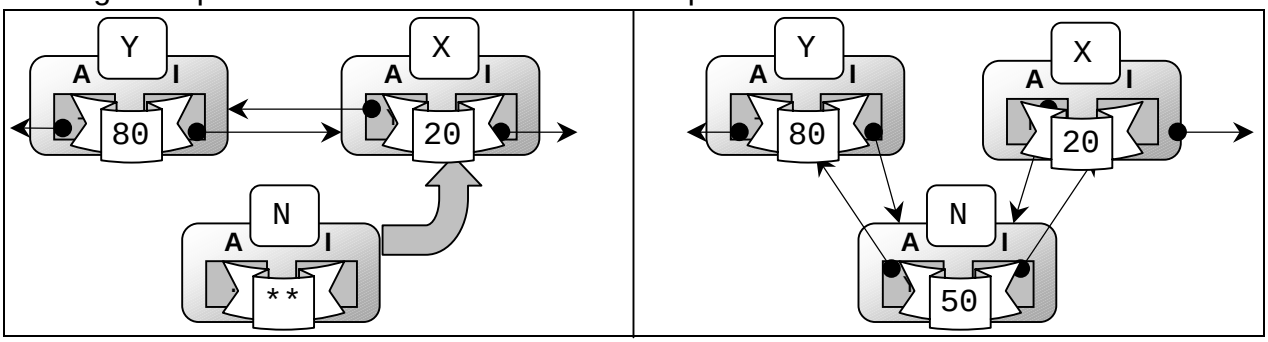
Ecco come ho pensato di realizzare il protocollo.



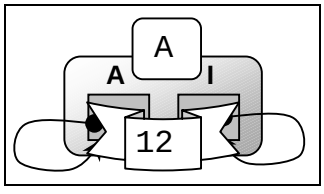
3.22) ASSEGNAZIONE DELLA PRIORITÀ AL NUOVO SERVER

> La matematica è la sola scienza esatta in cui non si sa mai di cosa di stia parlando né se quello che si dice è vero. (Bertrand Russell)

Per evitare le collisioni tra le prenotazioni è stato necessario introdurre il concetto di priorità tra i vari server. A ogni server è associato un valore di priorità che sarà assegnato a ogni prenotazione inviata agli altri server.
La caratteristica di dinamicità del sistema impone che la priorità ai server sia **attribuita dinamicamente**, mentre il protocollo di prenotazione impone che tutti i server abbiano un **valore** di priorità tra loro **differente**!
È proprio nel momento in cui viene aggiunto un nuovo server che bisogna attribuirgli una priorità.
Ma non può essere una priorità qualsiasi. Bisogna garantire che ogni server abbia un valore di priorità diverso da tutti gli altri server, altrimenti non sarebbe più possibile garantire la "consistenza" di tutti i data base (vedere la "Nota sulla Priorità nel DataBase" all'interno del paragrafo "collisione di prenotazioni").
Per non appesantire il protocollo bisogna fare in modo che la scelta sia il più possibile "locale".
Ho pensato a una soluzione molto semplice, che va a intaccare pochissimo quanto è stato stabilito finora.
Al termine del protocollo di aggiunta, il nuovo server verrà inserito tra i due server che sono stati tirati in ballo per aggiornare le informazioni sull'anello. Posso quindi pensare di attribuirgli una priorità il cui valore sta a metà tra quello sei suoi nuovi vicini.



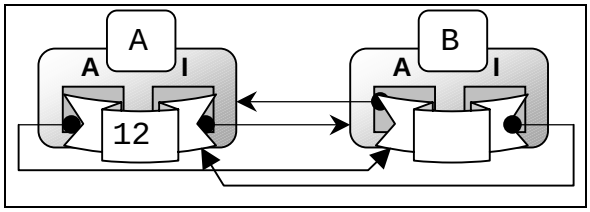
Ossia la priorità del nuovo server sarà la **media** tra quelle dei suoi vicini;`(80+20)/2 = 50`.
Ora bastano pochi accorgimenti per arrivare al caso generale.
Il primo server che viene lanciato si assegnerà priorità massima.



Stabiliamo fin d'ora che:

- la priorità massima ha un valore di `120`;
- la priorità minima ha un valore di `0`;
- tutte le altre priorità possono assumere valori tra quelli della minima e della massima.

Il successivo server inserito vedrà il primo server sia come predecessore sia come successore. Per il calcolo della priorità si ricorre a uno stratagemma; nel caso in cui il successore abbia priorità maggiore, lo si considera come se avesse priorità minima!



Difatti, dal protocollo di inserimento, ogni server ha "indietro" un server a priorità superiore alla propria mentre ha "in avanti" un server a priorità inferiore (a eccezione del server di priorità inferiore che ha "in avanti" un server a priorità superiore).
Il nuovo server calcolerà la media tra `120` e `0`, ottenendo `60`.

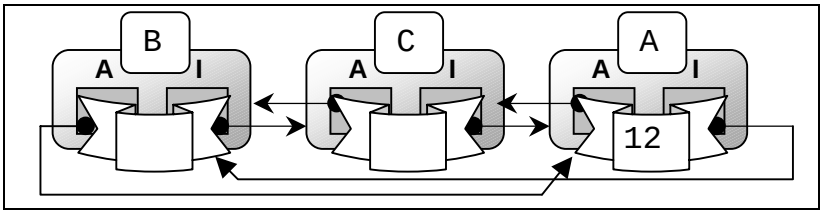
Nota: il secondo server ottiene un valore di priorità esattamente a metà nella scala delle possibili priorità.
Il terzo server C può essere inserito tra A e B o tra B e A; la sua priorità risentirà di questo!
Nel primo caso, C calcolerà:

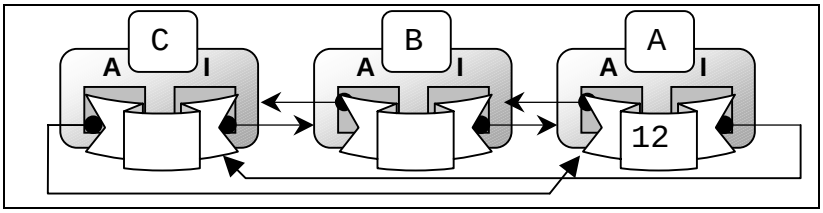
```
(120 + 60) / 2 = 90
```

Nell'altro caso:

```
( 60 + 0 ) / 2 = 30
```

(poiché il server C ha come successore un server a priorità maggiore).





Per poter fare il calcolo, il server C deve però conoscere le priorità dei due server tra cui si inserisce. Il server a cui ha richiesto l'aggancio deve dargliele.
Per non andare a modificare (e quindi a appesantire il protocollo) ho pensato di far in modo che il messaggio "MessSerCambSer" non contenga solo l'indirizzo del server con qui rimpiazzare quello attuale, ma anche il suo valore di priorità.
Ossia ogni server, oltre a mantenere il riferimento (l'indirizzo) al server vicino, ne mantiene anche la priorità. In questo modo non bisogna andare a chiedergliela.
L'unica modifica da fare al protocollo è l'invio verso il nuovo server dell'informazione della priorità della macchina a cui è stato richiesto di agganciarsi.
Nell'esempio, C conosce l'indirizzo di A e gli dice di agganciarsi; secondo il protocollo precedente, era inutile dire a C di porre A come suo predecessore, dal momento che ne conosceva l'indirizzo. Ora non è più così, perché C non ne conosce la priorità.
Dal momento che C conosce la priorità di A e B, può calcolarsi (localmente) la propria.
Analogamente, A può calcolarsi la priorità di C poiché sa la propria priorità e quella del suo corrente successore B.
Quando darò a B il "MessSerCambSer", gli darò anche la priorità di C appena calcolata.
Quindi, in "MessSerCambSer" sostituirò l'informazione sull'indirizzo con un informazione sulla "**Macchina**" (ossia la coppia indirizzo e priorità).
Se continuassimo a aggiungere server, chiedendo di volta in volta a ogni server presente, ecco la scala di tutti i possibili valori di priorità che ne verrebbe fuori:

- 120,
- 120 60,
- 120 90 60 30,
- 120 105 90 75 60 45 30 15,
- 120 112 105 97 90 82 75 67 60 52 45 37 30 22 15 7,
- 120 116 112 108 105 101 97 93 90 86 82 78 75 71 67 63 60 56 52 48 45 41 37 33 30 26 22 18 15 11 7 3,
- 120 118 116 114 112 110 108 106 105 103 101 99 97 95 93 91 90 88 86 84 82 80 78 76 75 73 71 69 67 65 63 61 60 58 56 54 52 50 48 46 45 43 41 39 37 35 33 31 30 28 26 24 22 20 18 16 15 13 11 9 7 5 3 1,
- 120 119 118 117 116 115 114 113 112 111 110 109 108 107 106 **105** 105 104 103 102 101 100 99 98 97 96 95 94 93 92 91 **90** 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 **75** 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 **60** 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 **45** 45 44 43 42 41

40 39 38 37 36 35 34 33 32 31 **30** 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 **15** 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0.

Nell'ultimo caso i numeri in grassetto indicano un conflitto di priorità: non possono essere assegnati tali valori.

La causa di tale conflitto deriva dal fatto che non è più possibile fare la media tra i due valori (si otterrebbe un numero frazionario che arrotondato coincide con uno degli estremi da cui è stato derivato).

Nel caso tutte le richieste venissero fatte sempre al primo server inserito, avrei:

- 120,
- 120 60,
- 120 90 60,
- 120 105 90 60,
- 120 112 105 90 60,
- 120 116 112 105 90 60,
- 120 118 116 112 105 90 60,
- 120 119 118 116 112 105 90 60,
- 120 **120** 119 118 116 112 105 90 60.

Nel caso peggiore si potrebbero inserire **al massimo otto server** nell'anello.

In ogni caso, ogni server sa se può inserire un nuovo server tra lui e il suo successore: basta che la priorità calcolata per il nuovo server non coincida con la sua o quella del suo successore. In tal caso non va permesso l'inserimento e perciò si rende subito al nuovo server un "fatto=no" (anziché il "fatto=sì" iniziale).

Nota: tengo 120 in modo che la priorità possa essere memorizzata in un singolo byte, così da risparmiare più memoria possibile (ci si deve ricordare che un valore di priorità viene associato a ogni spettatore inserito nel data base).

Può sembrare che tutto funzioni a dovere, ma rimane ancora un problema…

3.23) PROBLEMA DI CALO DELLE PRIORITÀ

Se rimuovo il server con priorità massima, nessun nuovo server inserito potrà mai raggiungere nuovamente tale priorità, visto che il calcolo basato sulla "media" fornisce solo numeri che sono sempre inferiori al maggiore tra i due valori in gioco.

Se il sistema è molto "dinamico" e si continua a inserire e rimuovere server accade inevitabilmente che ai nuovi server vengano attribuiti valori di priorità sempre più bassi.

Prima o poi l'anello avrà server con priorità basse e poco differenti. Questo è un problema, poiché tra due server posso inserirne un altro solo se la sua priorità (calcolata con la media) non è uguale a quella dei due server coinvolti nell'inserimento.

Devo quindi modificare il calcolo della priorità.

Ho pensato di estendere la formula introducendo qualche condizione in più, ossia:

```
Se not SonoServerUnico
  Se Pmia > Psucc  allora
    Pn :=( Pmia + Psucc) / 2
  Altrimenti
    Se Pmia < Psucc
      Se Psucc = PMAX  allora
        Pn := ( Pmia + 0 ) / 2
      Altrimenti
        Pn := PMAX
    Altrimenti
      !NonPossoInserirlo!
```

```
Pmia  - Priorità del server a
        cui si fa la richiesta
Psucc – priorità del successore
PMAX  - priorità massima
SonoServerUnico: è "true" se il
       server in anello è "unico"
```

```
Altrimenti (c'è un server unico)
  Se Pmia = PMAX  allora
    Pn := ( Pmia + 0 ) / 2
  Altrimenti
    Pn := PMAX
```

In questo modo si prevede che al nuovo inserito possa essere assegnata la priorità massima.

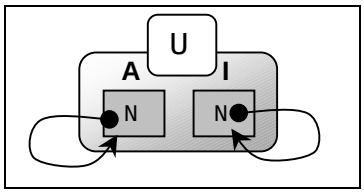
3.24) PROTOCOLLO DI RIMOZIONE DI UN SERVER

Come l'aggiunta, anche la rimozione ha bisogno di prevedere i tre casi (server unico, anello con due server e anello con più di due server).

> È giunta l'ora di andare. Ciascuno di noi va per a propria strada: io a morire, voi a vivere. Che cosa sia meglio, Iddio solo lo sa.
> (da "apologia di Socrate" di Platone)

3.24.1) Protocollo di Rimozione di un Server Unico

Per riconoscere questa configurazione, basta guardare se il riferimento a uno dei server vicini contiene l'indirizzo locale del server. Essendo un server unico, basta procedere alla terminazione dei thread di "Input".

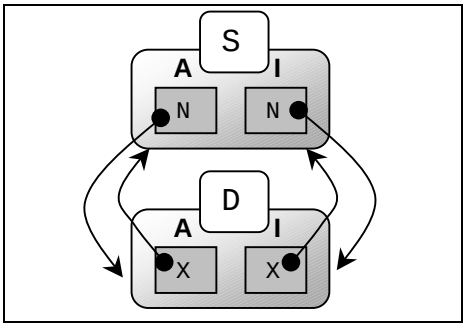


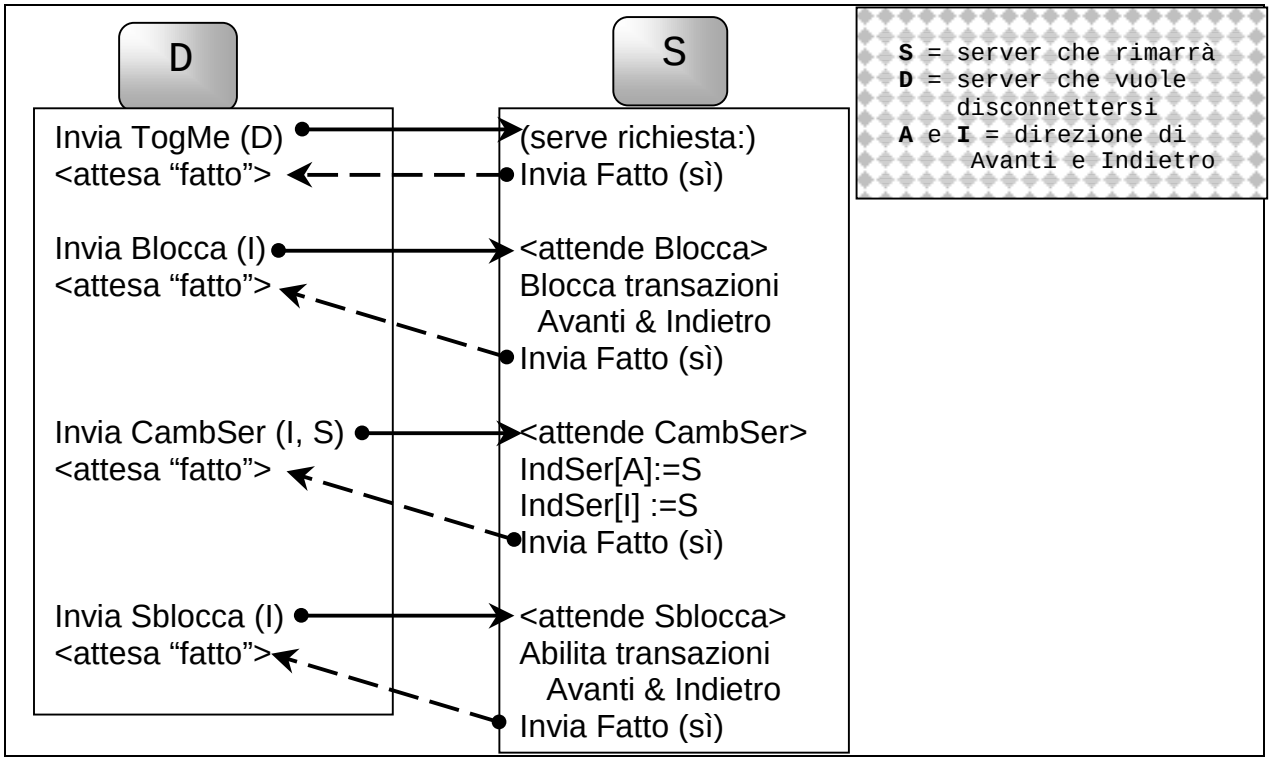
3.24.1.1) Nota sulla terminazione dei thread di "Input"

Da notare che non viene fatta una "stop()" dei thread (che tra l'altro, è uno dei metodi "invalidati" in Java 1.2) ma si ricorre all'invocazione di un metodo "terminaEsecuzione", contenuto in ognuno dei thread.
Tale metodo pone a "false" una variabile che serve al proprio metodo "run()" per continuare a eseguire dopo l'arrivo di ogni messaggio.
In pratica, il metodo "run" conterrà un ciclo "while" basato su questa variabile; all'interno del ciclo si fa una "accept" (per attendere su una socket) seguita dalle istruzioni per ka creazione e l'innesco del relativo thread di "Gest".
Il metodo "terminaEsecuzione" provvederà a chiudere la socket su cui sta attendendo il metodo "run" causando la generazione di un'eccezione da parte del metodo "accept" ("socket was closed"), si tornerà a testare la variabile e (essendo falsa) si uscirà dal ciclo e quindi il thread terminerà in modo naturale!

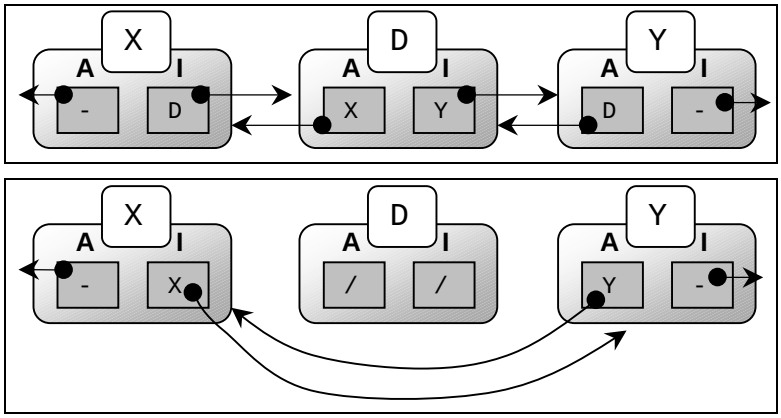
3.24.2) Protocollo di Rimozione da un Anello di Due Server

Quando i server sono due e uno vuole disconnettesi, le comunicazioni sono ristrette a un dialogo a due.
Chiamiamo D il server che vuole disconnettersi e S il server che rimarrà alla fine come un server singolo.
Si può scegliere una delle due direzioni per comunicare da D verso S… sceglierò la direzione "avanti".

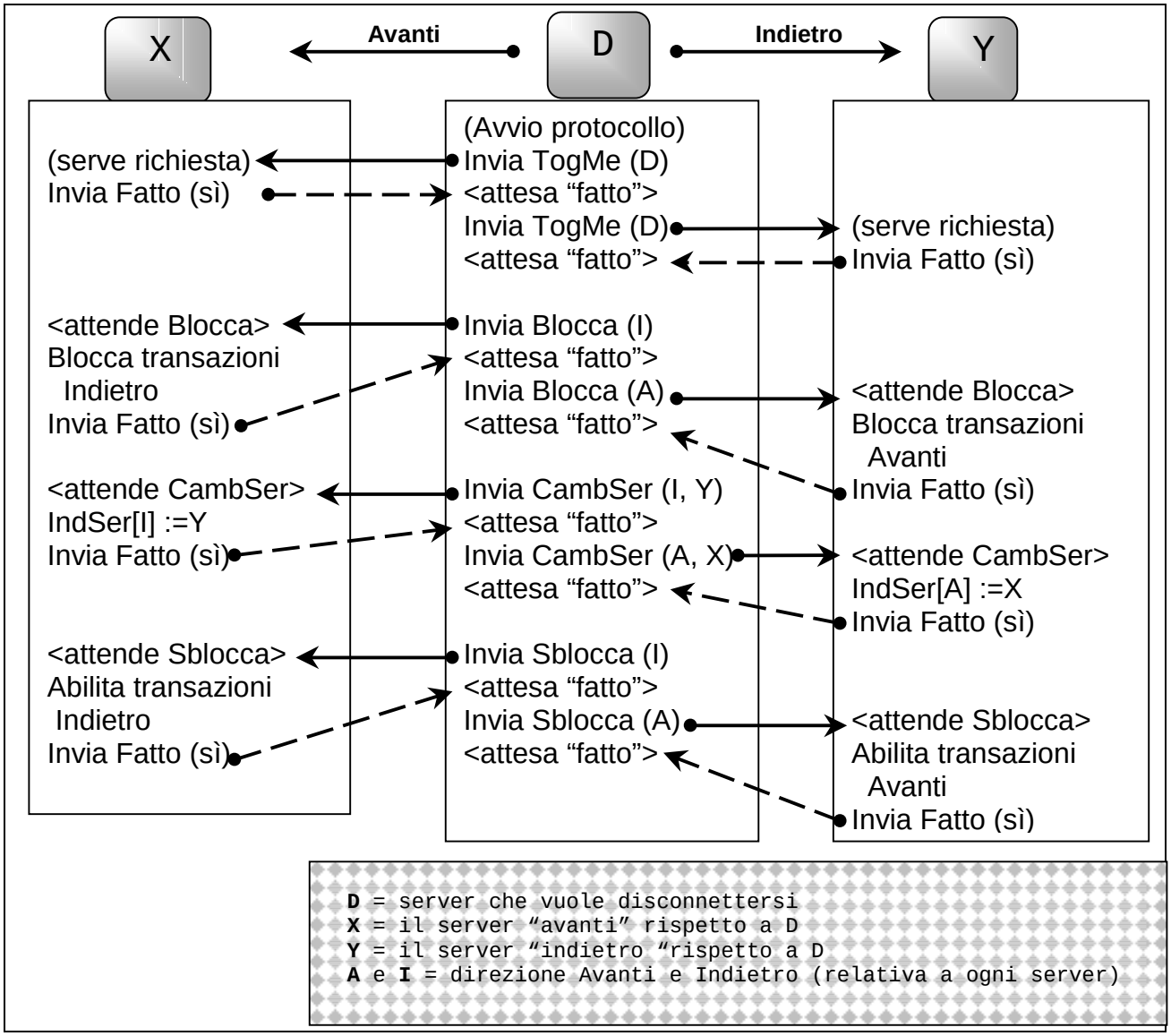




3.24.3) Protocollo di Rimozione da un Anello di Tre o Più Server



Quando i server nell'anello sono da tre in su, il protocollo non varia di molto.
Un server che vuole disconnettersi deve comunicarlo a due soli server: i suoi vicini.
Chiamiamo D il server che vuole disconnettersi mentere X e Y sono i suoi server vicini.
Le comunicazioni tra D e uno dei due sono duali a quelle fatte tra D e l'altro server.
In questo caso è bene precisare chi è X e Y: X è il server successore di D (in direzione "avanti") mentre Y è il predecessore (in direzione "indietro").



Aggiunte: per impedire che più protocolli di richiesta/rimozione siano inoltrati contemporaneamente allo stesso server, chi "avvia" un protocollo di rimozione deve innanzitutto bloccare le transazioni sia in avanti sia indietro.

3.25) RITRASMISSIONI

E' compito del client ritrasmettere la richiesta di prenotazione se entro un certo "time-out" non riceve risposta. Dopo tre o quattro tentativi, segnala l'errore all'utente.
Una volta arrivato al server, il messaggio è letto e viene creato un messaggio di prenotazione per i server. In esso è contenuta la stessa informazione di prenotazione che è arrivata dal client.
Tale messaggio circolerà tra i server; non appena è ricevuto, se ne registra la prenotazione nel data base e si passa avanti il messaggio.
Se viene perso (il datagramma che lo contiene non arriva a destinazione o cade il server che lo sta elaborando), solo il client che ha fatto la richiesta e il server che l'ha ricevuta inizialmente sono a conoscenza della prenotazione; perciò è compito loro ripristinare la transazione.
Ho già detto che il client ritrasmetterà la richiesta (dopo un time-out).
Il server che ha inviato agli altri il messaggio di prenotazione si aspetta che tale messaggio ritorni a lui dopo che ha fatto il giro (o che torni nel senso opposto come cancellazione).
Ma non è in attesa di quel particolare messaggio! Egli ha salvato in una tabella la prenotazione fatta e il thread che ha la comunicazione aperta col client.
Non farà nulla finché non gli torna la prenotazione!
Ma il thread associato (che aspetta che il messaggio ritorni) può "risvegliarsi" dopo un certo time-out e eseguire una nuova ritrasmissione (con un time-out inferiore a quello del client).
Però, occorre imporre che se a un server arriva una prenotazione che ha già nel suo data base (stesso spettacolo, posto e spettatore), non la riscriverà nuovamente e nemmeno lo considererà un errore; si limiterà a passare in avanti il messaggio.
In tal modo la prenotazione continuerà a propagarsi in avanti, fino a raggiungere il server che l'ha generata.
Ma se cade proprio quel server?
Il client provvederà a ritrasmettere il messaggio… ma a chi, se il suo server è caduto?
Posso pensare che il client conosca più di un server, per es. un server e il suo successore.
Sotto l'ipotesi di "guasto singolo" il client può continuare a fare ritrasmissioni.
Se al momento della nuova richiesta, la prenotazione (che contiene gli stessi dati) è già stata portata a termine, il nuovo messaggio di prenotazione circolerà, ma non avrà nessun effetto sui data base, e prima o poi tornerà al server iniziale che renderà "ok" all'utente.
Rimane però il problema che possono essere inviati più messaggi con la stessa prenotazione, mentre il server che li ha generati resta in attesa del primo che ritorna.
Nel messaggio ho inserito l'indicazione della macchina che ha fatto la richiesta (ho l'indirizzo della macchina e la sua priorità) perciò ogni macchina può sapere se un messaggio arrivato è stato in origine generato da lei oppure no.
Ma la macchina può cadere e quindi, dopo un "riaggancio" dell'anello, i suoi messaggi potrebbero continuare a circolare all'infinito, facendo calare le prestazioni del servizio!
Per evitare ciò occorre prevedere un "tempo di vita" per ogni messaggio; in pratica basta aggiungere un contatore dentro ogni messaggio.
Esso è posto a un particolare valore al momento della generazione del messaggio ed è decrementato di un'unità ogni volta che viene ricevuto da un server.
Quando arriva a zero il messaggio non viene più passato in avanti (viene perso).

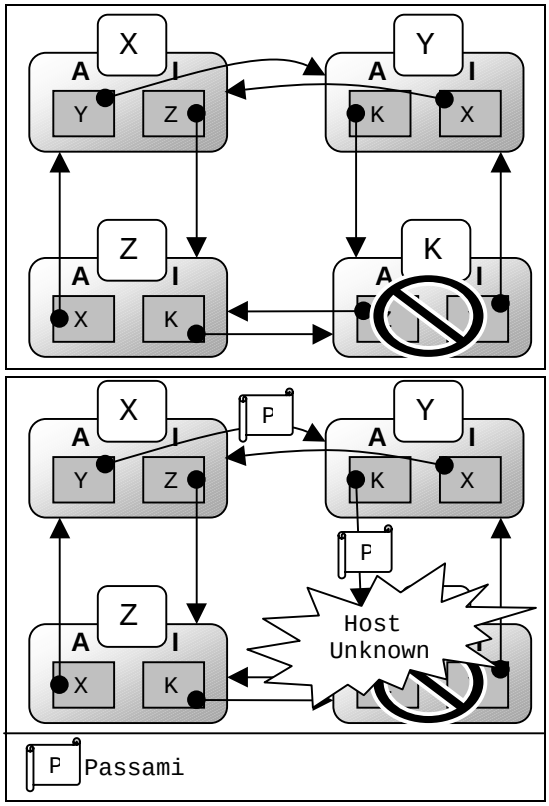
3.26) CHI CONTROLLA SE UN SERVER È CADUTO?

> La felicità più grande non sta nel non cadere mai,
> ma nel risollevarsi sempre dopo la caduta.
> (Confucio)

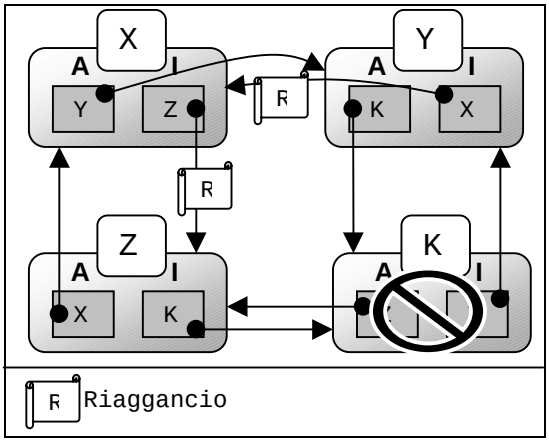
I server comunicano principalmente tra di loro attraverso messaggi in datagrammi.
Il thread "GestSer" invia in avanti ogni messaggio senza controllare che il destinatario esista ancora! Infatti nelle socket a datagrammi non c'è l'idea di "connessione".
Oltretutto, il messaggio può essere perso durante il suo cammino, senza che il destinatario sia caduto.
Il thread "**GestCli**" (sul server originario della prenotazione) innesca un "GestSer" e aspetta che il messaggio ritorni; se non ritorna entro un time-out, lo ritrasmette.
In questo modo aggiro la possibile perdita dei messaggi.
Ma non posso ritrasmettere all'infinito (il server successore o uno qualsiasi nella catena può essere caduto).
Se arrivo a fare quattro o cinque ritrasmissioni significa che probabilmente c'è qualcosa che non va nell'anello, perciò devo controllare che tutti i server funzionino.
Per fare questo controllo ho pensato di usare la connessione tra server a socket stream gestita dai "GestSys".
Sarà "GestCli" che dovrà iniziare il protocollo di "riaggancio".
Ma anche se un "GestSys" non riuscisse a connettersi a un server "vicino" si dovrà far iniziare il protocollo di "riaggancio".

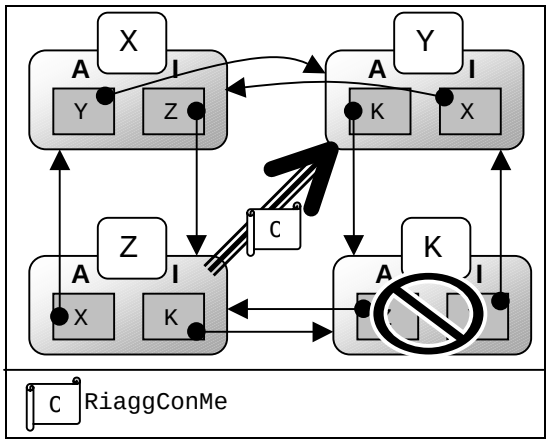
3.27) PROTOCOLLO DI RIAGGANCIO DOPO LA CADUTA

Innanzitutto, non è detto che il server caduto sia il proprio successore, perciò dovrò fare un protocollo che ricerchi il server caduto e, una volta riconosciuto che c'è e qual è, permetta di escluderlo dal sistema.
Consideriamo il seguente scema, in cui il server K è caduto!
Ma non è il server Y che sta inviando i datagrammi, bensì il server X.
A un certo punto il thread "GestCli" del server X si accorge che ha tentato cinque ritrasmissioni senza ricevere nessun messaggio di ritorno.
Procederà perciò a inviare al suo avanti server "avanti" un messaggio "MessSerPassami".
Ma non lo invierà su un datagramma, ma aprirà una connessione a stream.
Tentando una "connect" col server Y, se esiste in esso un "InputSys" in attesa su una "accept", verrà stabilita la connessione, altrimenti la "connect" genererà un'ecccezione (per esempio un "host unknown").
Già facendo la "connect" il server X sa se Y è vivo oppure no.

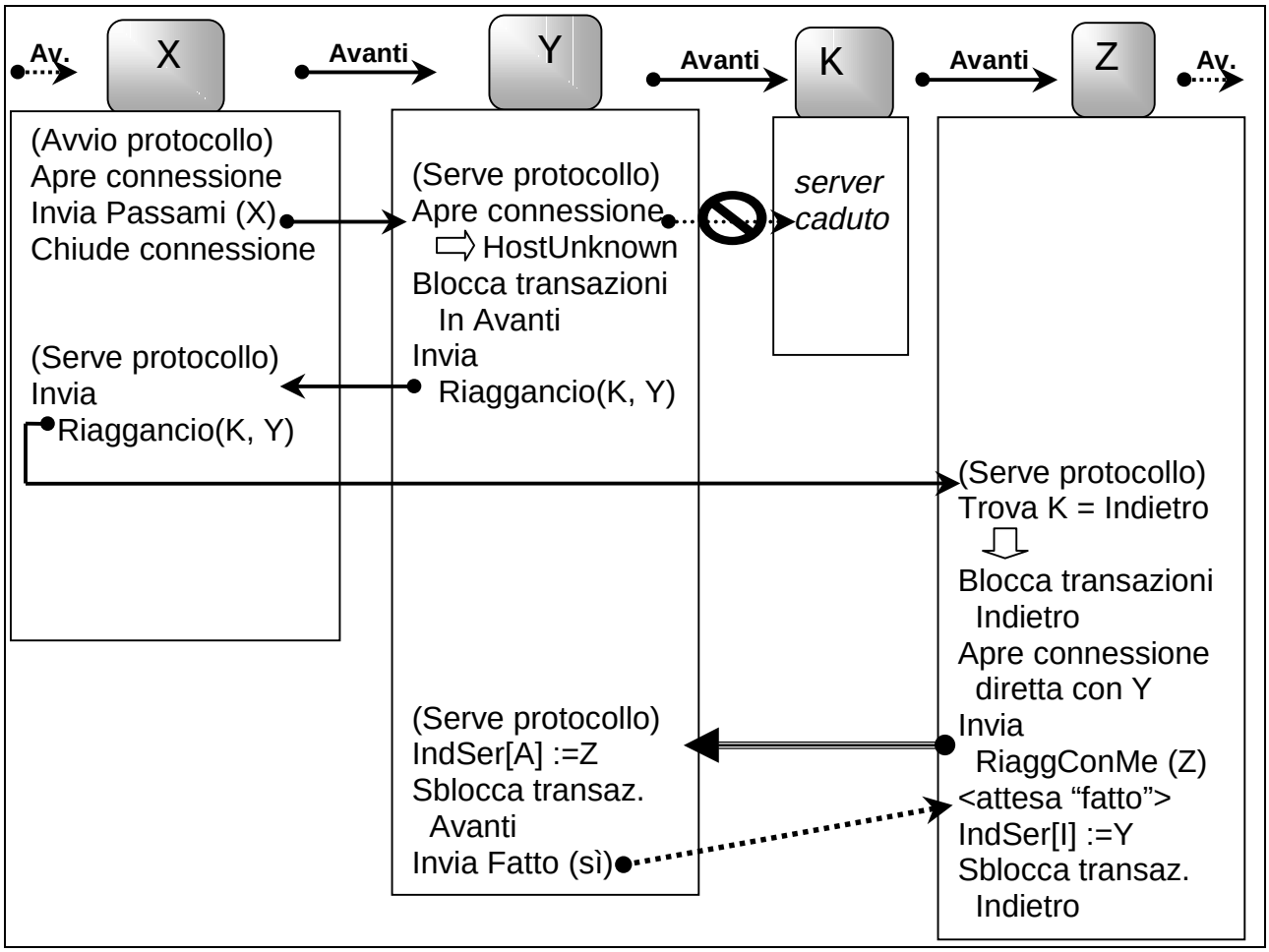


Però il "GestSys" creato si aspetta di leggere un messaggio da parte di X, perciò gli verrà inviato il "MessSerPassami".
Ora tocca a Y controllare che K (suo successore) sia vivo, mentre a X non resta nient'altro da fare!
Quando Y cercherà di connettersi a K, la "connect" genererà un'eccezione perché K non può accettare la connessione!
Il server Y procederà quindi a inviare indietro un nuovo messaggio di classe "MessSerRiaggancio" che conterrà l'indirizzo proprio (Y) e quello del suo successore, che è caduto (K).
Tale messaggio circolerà indietro nell'anello (ogni server lo farà circolare, ma su socket stream) finché un server non riconoscerà che il proprio server "indietro" ha proprio l'indirizzo del server caduto (nell'esempio, arriverà fino a Z).
Tale server genererà un nuovo messaggio "MessSerRiaggConMe" che conterrà il proprio l'indirizzo e lo invierà direttamente al server che ha generato il "MessSerRiaggancio" (è inutile farlo ripassare nell'anello, nel messaggio arrivato c'è l'indirizzo di chi avvisare).

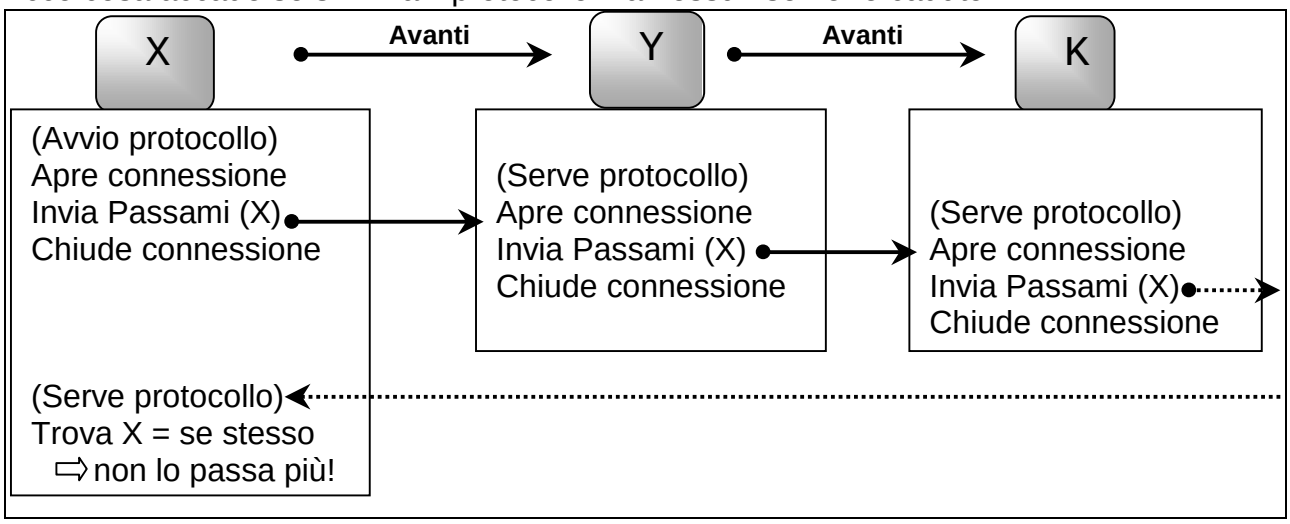




A questo punto sia X sia Z hanno la conoscenza reciproca e possono di conseguenza modificare i propri riferimenti in modo da ripristinare l'anello, escludendo il server caduto.
C'è da ricordare che le "transazioni" vanno bloccate; altrimenti perderei ulteriori messaggi.
Però può essere successo che non ci sia nessun server caduto.
Il messaggio "Passami" conterrà allora l'indirizzo del server che ha generato il primo messaggio, in modo che ogni server può controllarne la paternità.
Se a un server ritorna il "Passami", il messaggio va eliminato (significa che l'anello funziona perfettamente).
In questo modo il "Passami" può essere inviato in qualsiasi momento, per controllare se l'anello è funzionante.



Da notare che il server K potrebbe aver avuto dei problemi temporanei. Se viene riconosciuto “caduto” e perciò viene eliminato dall’anello, lui non può saperlo! Se “ritorna in sé” bisogna proibire che gli altri server accettino i suoi messaggi su datagrammi.
Quindi è importante che ogni server controlli che ogni messaggio arrivatogli sia stato inviato dal suo predecessore o successore!
Ogni altro messaggio non deve essere servito!
Ecco cosa accade se si inizia il protocollo ma nessun server è caduto:



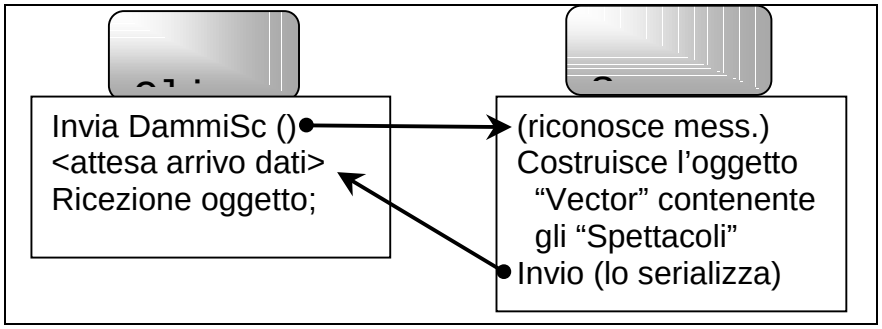
3.28) PROTOCOLLI DI RICHIESTA DATI

> E‘ difficilissimo parlare molto senza dire qualcosa di troppo.
> (Luigi XIV, re di Francia)

Ai client è necessario sapere alcune delle informazioni di cui sono in possesso i server.
Come può qualcuno prenotarsi a uno spettacolo se non sa quali spettacoli sono in programma? E come può decidere un posto se non conosce quelli che sono liberi?
In questa categoria cadono tutte le classi dei messaggi che discendono da “MessCliDammi”.

3.28.1) Richiesta dell’Elenco degli Spettacoli

Un client vuole sapere tutti gli spettacoli in programma; invierà a un server il messaggio “MessCliDammiSc”.
Il server gli renderà un oggetto di classe “Vector”, ossia una lista contenente tutti gli spettacoli.
La lista dovrà contenere oggetti di classe “Spettacolo”, poiché il client userà uno di essi per fare una successiva richiesta di prenotazione.
Per l’invio dell’oggetto si usa la serializzazione.
In pratica, il protocollo si riduce a:

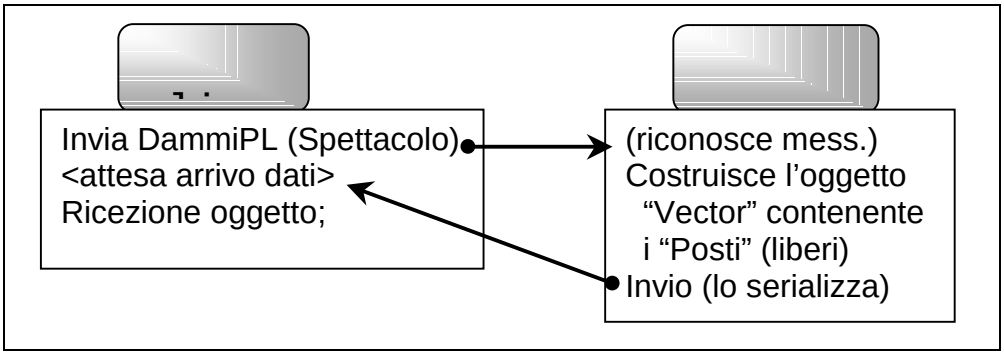


Da notare che non è il server che costruisce la lista, ma viene invocato un particolare metodo dell’oggetto “DataBase”.
Inoltre, l’oggetto “Vector” reso dall’oggetto DataBase conterrà tutti gli oggetti “Spettacolo” contenuti nel data base… gli oggetti, non una loro copia (c’è **aliasing**).
Se andassi a modificare i dati di uno spettacolo accedendo all’oggetto “Vector”, avrei un effetto collaterale di modifica anche dei dati nel data base.
Si potrebbe pensare gli mettere nella lista dei “cloni” degli “Spettacoli”, ma non lo farò perché nessuno accederà alla lista in scrittura; il server la invierà al client, perciò sul client ci sarà una copia della lista!

3.28.2) Richiesta dei Posti Liberi di uno Spettacolo

Questa richiesta è molto simile a quella precedente, soltanto che il client invierà un messaggio “MessCliDammiPL” (contenente il nome di uno spettacolo) e il server renderà un oggetto “Vector” contenente oggetti di classe “Posto”.

Perciò:

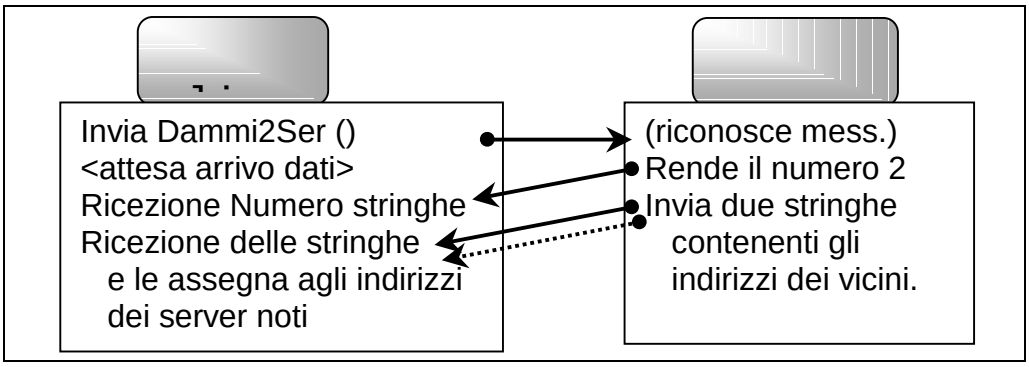


3.28.3) Richiesta dei Posti Occupati in uno Spettacolo da un certo Spettatore

È molto simile alla precedente, solo che il cliente indica uno spettacolo e uno spettatore e il server rende una lista con tutti i posti occupati per quello spettacolo dallo spettatore.

3.28.4) Richiesta degli Indirizzi di Altri Due Server

Se il client conosce solo l'indirizzo di un server, se questo cadesse il client non sarebbe più in grado di inviare le richieste di prenotazioni.
Ogni client deve conoscerne più di un server. Ho pensato di fargliene conoscere tre.
In realtà, all'inizio mi basta conoscerne uno, poiché posso sempre andargli a chiedere se mi dà altri due indirizzi.
Ho scelto di mantenere tre indirizzi (e non solo due), perché ogni server conosce già altri due indirizzi di altri server: quelli dei suoi "vicini"!
Il client invierà un "MessCliDammi2Ser" a un server il quale gli risponderà dandogli due stringhe contenenti gli indirizzi dei suoi vicini.



Però, se il server è unico gli farò rendere due stringhe nulle, in modo che il client sappia che non c'è nessun altro server.
Posso anche pensare a introdurre un "automatismo"; se al termine di una prenotazione il client conosce l'indirizzo di un solo server, si inizierà automaticamente questo protocollo.

3.29) SALVATAGGIO DELLA CONFIGURAZIONE

Ho previsto di salvare su disco la configurazione sia del client sia del server in modo che se si vuole temporaneamente uscire dal programma, quando si rientrerà non si dovranno reinserire tutti i dati.
Per un utente dal lato client questo è molto importante, perché vuol dire potersi disinteressare dell'indirizzo del server a cui si fanno le richieste.
L'operazione di salvataggio della configurazione va fatta:

- all'uscita dal programma;
- ogni volta che viene modificata la configurazione.

L'operazione di caricamento va fatta:

- all'avvio del programma.

Sul lato client le informazioni "da ricordarsi" sono:

- gli indirizzi dei server noti (a cui si inviano i messaggi);
- il time-out per le richieste inviate;
- il numero di ritrasmissioni delle richieste;
- l'uso (o meno) della finestra di "debug".

Sul lato server:

- l'indirizzo del server indietro (a cui si potrà richiedere di agganciarsi, all'avvio);
- il time-out per le richieste arrivate dai clienti;
- il numero di ritrasmissioni delle richieste arrivate dai clienti;
- il time-out per l'attesa dei "fatto" in messaggi server-server;
- l'uso (o meno) della finestra di "debug".

3.30) SALVATAGGIO DEL DATA BASE (LATO SERVER)

> Il problema di salvare l'ambiente fa tutt'uno col problema di salvare noi.
> (Vittorio Mathieu)

Anche il data base può dover essere salvato su disco.
Se il sistema dev'essere temporaneamente interrotto senza che le prenotazioni vengano perse l'unica maniera è ricorrere al salvataggio dell'intero data base su disco.
L'oggetto DataBase è già di per se serializzabile (ho usato la serializzazione per inviarlo via socket-stream a un nuovo server aggiunto), perciò è banale salvarlo su disco.
Basta aprire un "FileOutputStream" che verrà incapsulato in un "ObjectOutputStream" sul quale verrà scritto l'oggetto DataBase.
Si può anche introdurre un automatismo; se all'avvio del server nella directory c'è un file contenente il DataBase, si può chiedere all'utente se caricarlo o meno.

3.31) VISUALIZZAZIONE DEL DATA BASE (LATO SERVER)

Dal lato server può essere possibile vedere le prenotazioni fatte nel data-base.
Bisogna prevedere che l'utente (del server) possa vedere un elenco contenente tutti gli spettacoli in programma e, scegliendone uno, possa vedere tutti i posti in esso occupati (e da quali spettatori).

3.32) CONTROLLO AUTOMATICO DELL'ANELLO

La capacità di accorgersi che un server si è guastato e il successivo recupero dell'anello non serve assolutamente a nulla se il controllo non viene fatto periodicamente.

Il sistema non tollera il guasto contemporaneo di due server.

Se si guastano due server, l'anello "si spacca"! Ogni "pezzo" rimasto, a seguito del protocollo di riaggancio, si riunirebbe in un anello a sé stante.

Come risultato si otterrebbero due anelli distinti che non comunicano tra loro!

Il temine "guasti contemporanei" ha un significato relativo, poiché non è detto che debbano essere proprio "contemporanei".

Quello che ci rovina è il tempo che trascorre tra i due guasti relativamente a ogni quanto tempo viene eseguito il controllo dell'anello.

Se il controllo viene fatto ogni ora, quando capitano due guasti in un'ora siamo finiti.

È necessario che ogni server, automaticamente e di tanto in tanto, faccia partire il protocollo di controllo dell'anello.

Non ci sarà un solo server delegato a fare ciò, ma tutti i server si adoperano a farlo.

Ho pensato di fissare un periodo di cinque minuti. Un nuovo thread sarà delegato a "tenere il tempo". Non farà altro che fare una "sleep" per il tempo prestabilito dopodiché invocherà un particolare metodo del Server (il quale darà inizio al controllo dell'anello).

Il thread ciclerà, tornando quindi sulla "sleep".

Per l'esattezza, il thread in questione l'ho chiamato "TimeOutConta" e ho creato un interfaccia "TimeOutAvvisa" che dev'essere implementata da chiunque voglia essere "chiamato" allo scadere del tempo (in questo caso, il Server implementerà tale interfaccia).

Poiché la partenza del controllo è affidata a ogni server, il "caso peggiore" si ha quando tutti i cominciano nello stesso istante. Infatti si avrà un controllo ogni cinque minuti.

Ma è più probabile che i server siano "sfasati", perciò globalmente i controlli avverranno con maggiore frequenza.

Convalida e Verifica di Componenti Software

- Definizione delle proprietà che caratterizzano
  la qualità del prodotto
- Studio di metodi per il controllo e la verifica
  di ciascun passo di sviluppo
- Formalizzazione di tecniche per il collaudo
  del prodotto finale

Convalida: confronto con requisiti informali (utente)

Verifica: confronto con specifiche rigorose o formali (analista)

Convalida e Verifica: Definizioni IEEE

- ***Malfunzionamento*** o guasto (*failure*): comportamento non corretto di un programma
- ***Anomalia*** o difetto (*fault*): causa di un malfunzionamento; riguarda la struttura statica di un programma
- ***Errore*** (*error*): causa di un'anomalia

Approcci possibili per la verifica/convalida:

1. Metodi di ***prova formale***
2. Tecniche di ***test***
3. Tecniche di ***debugging***

Convalida e Verifica: le tecniche (1)

1. Metodi di ***prova formale***: verificare la correttezza del software provando l'assenza di anomalie nel prodotto
2. Tecniche di ***test***: tentare di rilevare eventuali malfunzionamenti
3. Tecniche di ***debugging***: tentare di localizzare le anomalie alla base dei malfunzionamenti rilevati.

Analisi **** statica*** (basata sull'analisi del codice sorgente)

- in compilazione (lessicale/sintattica/type checking)
- flusso dei dati (vincoli ed espressioni regolari)

**** dinamica*** (basata sull'esecuzione del programma)

- test strutturale (codice; *white box*)
- analisi mutazionale
- test funzionale (specifiche; *black box*)
- test "in grande"

Convalida e Verifica: le tecniche (2)

Esecuzione ☑ ***"numerica"***

☑ ***simbolica***

- utilizzando ***simboli*** come valori per le variabili
- prova di ***correttezza formale***
- ***trasformazioni***, invarianti, ***copertura***

Debugging

☑ tecnica naive

☑ dump di memoria

☑ debugging simbolico

☑ debugging per prova

5) PIANI DI TEST

> Perché i nostri processi cognitivi dovrebbero essersi indirizzati verso una ricerca che è un lusso, verso la comprensione dell'intero universo? Perché proprio noi?
> (John D.Barrow)

5.1) EVOLUZIONE & ESPLORAZIONE

Nel costruire il programma ho usato un approccio di tipo **evolutivo** incentrato su una "**programmazione esplorativa**".
Questo vuol dire che sono partito costruendo un prototipo del sistema che è stato man mano raffinato fino a diventare il sistema finale.
Ho scelto di usare un modello evolutivo perché i requisiti dell'applicazione non erano ben chiari all'inizio. Infatti sono state necessarie molte modifiche e cambiamenti al progetto durante la programmazione (per es. l'introduzione della priorità tra i server per gestire i casi di collisione delle prenotazioni).
All'inizio ho usato la metodologia **buttom-up,** poiché sono partito realizzando la classe DataBase, seguita dalla classe astratta Mess e da alcune delle sue sottoclassi (quelle di prenotazione e cancellazione sia del lato server sia di quello client) e dall'immancabile TraslaMess. Subito dopo sono passato a usare una metodologia **top-down**; infatti ho realizzato la classe Server e le due coppie di thread InputCli-GestCli e InputSer-GestSer.
Contemporaneamente ho realizzato le classi Server e Client.
Però non ho realizzato subito la parte di interfaccia grafica; nel client e nel server ho scritto i metodi per lo scambio dei messaggi e le azioni da intraprendere.
In questo modo potevo avere subito a disposizione un "prototipo" sia del server sia del client. Via via ho aggiunto nuove classi di messaggi con il codice che ne specifica la semantica. In questo modo ho potuto fare dei test già all'inizio (o quasi) dello sviluppo.

5.2) FINESTRA DI DEBUG

Per controllare cosa effettivamente faceva il programma ho introdotto un oggetto che realizza una finestra su cui scrivere messaggi.
All'interno del codice ho introdotto delle istruzioni per stampare varie frasi di controllo e di informazioni sui messaggi inviati e ricevuti.
Questo mi ha permesso di verificare la correttezza dei messaggi scambiati e delle modifiche allo stato del sistema.
Inoltre, la scrittura su questa finestra provoca un certo rallentamento dell'esecuzione del programma e quindi è utile quando si va a verificare il comportamento di un server all'arrivo simultaneo di più messaggi di prenotazione (da parte di più client), infatti è molto più facile far "sovrapporre" due richieste

5.3) MODULI GUIDA

> Soltanto chi non ha bisogno né di comandare né di ubbidire è davvero grande.
> (Johann Wolfgang Goethe)

Ovviamente si è resa necessaria la costruzione di "**moduli guida**" per verificare che i moduli (o gli oggetti) di più basso livello funzionassero correttamente. A ben guardare, le classi Server e Client all'inizio erano dei moduli guida che man mano sono stati fatti evolvere nelle classi finali.

5.3.1) DataBase

Per verificare che il data base funzioni correttamente si è realizzato un modulo guida che:

- crea un oggetto DataBase vuoto;
- spettacoli:

    - inserisca due spettacoli, S1 e S2
    - reinserisca S2 (ci si aspetta un errore)
    - cancelli S2
    - cancelli nuovamente S2 (ci si aspetta un errore)
    - cancelli uno spettacolo non inserito, S3 (ci si aspetta un errore)
    - verifichi la presenza di S1 (ci si aspetta che esista)
    - verifichi la presenza di S2 (ci si aspetta che non esista)
- prenotazioni senza priorità
    - inserisca le due terne (S1, X1, P1) e (S1, X2, P2), ossia si stanno inserendo due spettatori X1 e X2 in due posti P1 e P2 diversi nello spettacolo S1;
    - reinserisca la terna (S1, X1, P1) (non ci si aspetta un errore)
    - inserisca la terna (S1, X3, P1), con X3 diverso da X1 (ci si aspetta un errore)
    - cancelli la terna (S1, X1, P1)
    - cancelli nuovamente la terna (S1, X1, P1) (non ci si aspetta un errore)
    - cancelli la terna non inserita (S1, X3, P1) (non ci si aspetta un errore)
    - verifichi la presenza di (S1, X2, P2) (ci si aspetta che esista)
    - verifichi la presenza di (S1, X1, P2) (ci si aspetta che non esista)
- prenotazioni con priorità
    - inserisca in sequenza le due terne (S1, X, P) e (S1, Y, P) rispettivamente con priorità Pr1 e Pr2. Si devono considerare i tre casi:
        - Pr1 > Pr2 (ci si aspetta che venga scritta la prima terna)
        - Pr1 < Pr2 (ci si aspetta che venga scritta la seconda terna)
        - Pr1 = Pr2 (ci si aspetta un errore all'inserimento della seconda terna)
    - poiché la priorità è solamente nell'inserimento, non si testa la cancellazione
- stampa
    - visualizzare l'elenco degli spettacoli, dei posti liberi nello spettacolo S1 e quelli occupati da P2 nello spettacolo S1

5.3.2) TraslaMess

Quest'oggetto esegue le trasformazioni degli oggetti messaggi in Data-I/O-Stream, nei due sensi. Per testare la corretta conversione basta:

- lato 1: costruire un messaggio (per es. MessCliRichiesi) e inviarlo (convertendolo) su una socket "vista" come DataOutputStream collegata al processo sull'altro lato;
- lato 2: ricevere i dati (convertendoli) da una socket "vista" come DataInputStream; una volta ricostruito l'oggetto, lo si rinvia dall'altro lato eseguendo una nuova conversione;
- lato 1: ricezione di un nuovo flusso di dati e conversione in un nuovo oggetto a cui segue il confronto con l'oggetto inviato inizialmente.

In aggiunta si può fare una stampa a video del contenuto del messaggio.

5.3.3) Client e Server

Come ho già detto, all'inizio questi oggetti sono nati come "moduli guida" che sono stati via via fatti evolvere (mediante raffinamenti successivi) fino alla versione finale.
L'interfaccia grafica non è stata inizialmente considerata, perciò questi moduli contenevano solo un "main" con l'invocazione di particolari metodi che svolgevano la comunicazione.
All'inizio il server era "unico" (non aveva altri server a cui passare i messaggi) perciò per prima cosa ho realizzato e testato le comunicazioni su socket-stream tra client e server.

Poi si è introdotto il parallelismo, facendo generare il thread “InputCli” dal Server.
In una seconda fase si è aggiunta e testata la connessione a datagrammi tra server.
Come prima cosa si è controllato che funzionasse (cosa che puntualmente non si è verificata) inviando dei messaggi di prova.
Una volta che il server era “pronto” si è creato e inserito in esso l’oggetto GestTransazioni.
Quindi è stato aggiunto il thread “InputSys” e tutti i messaggi che poteva gestire.

5.4) TEST DI INTEGRAZIONE

Procedendo con un modello “evolutivo”, non c’è stato un vero e proprio momento in cui sono stati uniti tutti i moduli in modo da ottenere il sistema finale.
Il programma si è evoluto col passare tempo, aggiungendo nuovi moduli o aggiungendo o modificando le funzionalità già esistenti.
Per testare pienamente il sistema ho pensato di eseguire le seguenti prove, la cui sostanza dipende dal numero di client e di server presenti.
Per una migliore comprensione si consiglia di attivare la “finestra di debug”.

5.4.1) Avvio di un Server Unico

Si lancia il programma Server su una macchina e si “connette” il server come “unico”, ossia si avrà un anello costituito da un solo server.
Si provvede quindi a riempire il data base con qualche spettacolo.
Per comodità, si esegua il salvataggio del data base su disco, in modo da non dover inserire nuovamente gli spettacoli.

5.4.2) Avvio di un Client

Una volta lanciato il programma Client, andare in “configurazione” e scrivere l’indirizzo fisico della macchina su cui risiede il server precedentemente lanciato.
Si facciano le seguenti azioni (inserendo a video gli opportuni dati nella “maschera”):

- fare due prenotazioni differenti
- rifare l’ultima prenotazione (verrà detto “il posto è occupato dallo stesso spettatore ”)
- rifare l’ultima prenotazione cambiando spettatore (verrà detto “il posto non è occupato dallo spettatore”)
- cancellare una di quelle fatte
- cancellarla nuovamente (verrà detto “il posto è già libero”)
- cancellare l’altra fatta, ma dando un diverso spettatore (verrà detto “ il posto è occupato da qualcun altro”)
- richiedere l’elenco degli spettacoli, dei posti liberi e dei posti occupati dallo spettatore.

5.4.3) Avvio di un Secondo Server

Su un’altra macchina si lanci un altro processo Server.
Quando si fa la “connessione”, si dia l’indirizzo della macchina su cui è stato lanciato il primo server; in questo modo si eseguirà il protocollo di connessione e si otterrà un anello di due server.
Sempre sul **client** precedente, rifare le stesse operazioni di prima.
La differenza è che ora i due server devono comunicare tra loro.
Ora provare a sconnettere il primo server e a fare una richiesta dal clinet; non essendoci più il primo server, andrà a fare la richiesta sul secondo!

Disconnettere quindi anche questo server; il client non troverà più nessun server e ne informerà l'utente tramite una finestra.

5.4.4) Connessione dei Precedenti Server e Aggiunta di un Terzo

Riconnettere in anello i due precedenti server e lanciare un terzo server su una nuova macchina e connetterlo all'anello dando l'indirizzo fisico di una delle due macchine su cui risiedono i server già in anello.
Per comodità, rilanciare per primo il server che aveva salvato il data base su disco, e connettere il secondo verso di lui.
Riprovare le prenotazioni/cancellazioni da parte del client, su uno qualsiasi di questi server.
In più, da uno dei server:

- eseguire il "controllo dell'anello" e provare a inviare un messaggio agli altri server.
- provare a inserire nuovi spettacoli o a cancellarne di esistenti.

Per testare l'anello si può fare nel seguente modo.
Fare in modo che il client invii le richieste a un server diverso da quello che si sta usando.
Sul server, "forzare" il blocco dei messaggi in avanti (usare la finestra di configurazione e bloccare la direzione "avanti") e fare una prenotazione da parte del client.
Accadrà che dopo dieci secondi scatterà il "time-out" all'interno del server a cui è arrivata la richiesta del client e si avrà una successiva ritrasmissione.
Dopo 3 ritrasmissioni (ossia trenta secondi) viene automaticamente avviato il "controllo dell'anello" e al client è reso uno stato di errore.
Provare ora a diminuire il time-out del client (a meno di 30 secondi); dopo una richiesta, se non gli arriva una risposta entro tale tempo informerà l'utente con una finestra.
Ora si può provare a simulare una caduta di un server; la maniera più brutale è quella di spegnere la macchina su cui risiede il server, ma può essere sufficiente premere in sequenza i tasti CTRL e C nella finestra di "prompt di Dos" da cui è stato lanciato il server.
Per testare il protocollo di riaggancio, si può far iniziare il "controllo" dal server che era il "successore" del server caduto.

5.4.5) Più Client

Ora lanciare e configurare un nuovo client su una nuova macchina, possibilmente vicina (fisicamente) a quella su cui risiede il precedente client.
Fare in modo che i client inviino le richieste allo stesso server.
Introdurre nei client i dati per due prenotazioni che siano differenti e **far partire le richieste esattamente nello stesso istante**. Saranno servite concorrentemente.
Da notare che il nuovo client potrebbe essere lanciato anche dalla macchina in cui è già presente il precedente client, ma avendo a disposizione un solo mouse è difficile far partire due prenotazioni contemporaneamente.
Ora si inserisca in ambo i client i dati di una prenotazione che abbia lo stesso spettacolo e lo stesso posto ma uno spettatore differente e si inviino le richieste (sempre contemporaneamente). Solo la prima che arriva al server sarà registrata; l'altra sarà rifiutata.
Ora si modifichi in ambo i client il solo campo riguardante il posto (tenendolo sempre uguale in entrambi). In uno dei due client cambiare la "configurazione" in modo che le sue richieste vadano a un altro dei tre server presenti nel sistema.
Inviare le richieste contemporaneamente. Solo la prenotazione che è stata richiesta al client più prioritario (tra i due coinvolti) verrà registrata; l'altra fallirà!

6) USO DI UNA METRICA

6.1) CALCOLO DEL LOC (LINES OF CODE)

Come metrica del software ho usato il “LOC”, ossia il **numero di linee di codice**.
Il calcolo può essere fatto solo a posteriori, ossia solamente quando il programma è terminato (e si hanno a disposizione i sorgenti).

So che il risultato può essere suscettibile di molte critiche, ma mi è sembrato il più “comodo” al mio caso. D’altronde, non avevo il problema prevedere a priori la “difficoltà” del programma poiché avevo solo quest’idea per un progetto da portare all’esame.
O questo o niente.
Quindi ecco il mio calcolo dei LOC; il codice è composto da:

9434 LOC considerando il codice come un testo (contando anche le linee vuote),
7396 LOC se non si contano le linee vuote,
6084 LOC se non si contano le linee vuote e le note
5092 LOC se non si contano le linee vuote, le note e i costrutti “begin-end”

6.2) COCOMO

Per avere degli indicatori, una volta ottenuto il LOC, ho pensato di usare la metrica del COCOMO (COnstructive COst MOdel).
So bene che non sarebbe corretto farlo, perché il modello si basa sull’ipotesi che il progetto e lo sviluppo del software sia avvenuto secondo il ciclo di vita “a cascata” mentre il mio software è stato sviluppato usando un modello “evolutivo”!

> Era un mercante di pillole perfezionate che calmavano la sete. Se ne inghiottiva una alla settimana e non si sentiva più il bisogno di bere. [..]
> "Gli esperti hanno fatto dei calcoli. Si risparmiano cinquantatré minuti alla settimana".
> "E che cosa se ne fa di questi cinquantatré minuti?"
> "Se ne fa quel che si vuole..."
> "Io", disse il piccolo principe, "se avessi cinquantatré minuti da spendere, camminerei adagio adagio verso una fontana..."
> (da "il piccolo principe" di Antoine De Saint-Exupéry)

Perciò i risultati che ne verranno fuori saranno, ben che vada, molto approssimativi.
Tuttavia ho pensato di applicare il COCOMO soltanto per ottenere un’indicazione (molto grossolana) dei parametri di “sforzo” e di “tempo”.
Visto che già l’ipotesi principale non è applicabile, mi limiterò a usare il BASIC COCOMO.
Il mio progetto è “**organico**”, ossia è un “progetto semplice e di limitate dimensioni che non impone requisiti particolarmente stringenti”.
Le equazioni sono (dove KLOC sta per migliaia di LOC):

Sforzo: $\mathbf{S} = 2{,}4 * KLOC^{1{,}05} = 2{,}4 * 5^{1{,}05} = \mathbf{13{,}0}$ mesi-uomo
Tempo: $\mathbf{T} = 2{,}5 * S^{0{,}38} = 2{,}5 * 13{,}0^{0{,}38} = \mathbf{6{,}6}$ mesi

Posso solo dire che non ho impiegato tutto questo tempo, ma “solo” due mesi (tenendo conto anche del tempo impiegato a scrivere questa documentazione).
Però bisogna tener conto che ho usato molte “utilità” presenti nel linguaggio Java (vettori, monitor, comunicazioni via socket). Se non avessi potuto disporne, il tempo impiegato sarebbe stato molto maggiore.
E poi si sa, il COCOMO è un metodo di stima che ci azzecca con un margine di errore inferiore al 20% nel 70% dei casi; in più nel mio caso non è verificata l’ipotesi di base.