Testo del problema

Una azienda specializzata nella vendita all'ingrosso di componenti elettronici intende attivare un servizio in rete da affiancare ai loro punti vendita sparsi sul territorio nazionale .
I clienti, preventivamente registrati, potranno accedere all'acquisto dei prodotti desiderati tramite internet, quindi riceveranno i prodotti richiesti tramite posta .
Nuovi clienti, per motivi legislativi, dovranno registrarsi presso uno dei punti vendita sul territorio.
Data la natura specializzata dell'utenza il sistema di vendita potrà essere implementato sia con un apposito software, sia con strumenti di uso comune come i normali browser web .
Siccome gli articoli sono acquistati dalle ditte produttrici in lotti per ogni lotto si avrà associato un prezzo e delle caratteristiche quindi il sistema dovrà controllare che non vengano venduti più pezzi di quanti siano realmente disponibili .
In ottemperanza alle leggi vigenti un cliente può annullare un ordine, quindi i prodotti richiesti devono essere resi nuovamente disponibili per la vendita .
Il sistema di vendita deve essere affidabile garantendo la continuità del servizio anche a fronte di guasti singoli.
A tal fine si deve poter utilizzare un insieme dinamico di server che possa variare nel tempo.
I clienti devono poter richiedere la lista dei prodotti disponibili, ordinare dei prodotti, disdire un ordine.

Analisi dei requisiti

Dall'analisi delle specifiche appare evidente che esistono due tipologie di utenza a cui corrispondono due tipi di applicativi.
Il primo preposto alla gestione degli ordini ricevuti, accessibile solo dalla azienda; il secondo rivolto agli utenti che intendono acquistare dei prodotti.
Risulta naturale la suddivisione in :

- **Server** : a cui si richiede la gestione dei cataloghi dei prodotti, la validazione degli ordini ricevuti e la gestione dei protocolli necessari per garantire affidabilità.
- **Client** : a cui si richiede la possibilità di inoltrare un ordine, rimuovere un ordine precedentemente effettuato, mostrare la lista dei prodotti disponibili.

Come già detto il sistema deve essere affidabile, ossia i client devono poter accedere al servizio in ogni momento ed inoltre i dati presenti non devono essere persi .
Per garantire ciò l'unica strada percorribile è quella di introdurre la replicazione dei server .
Il numero di server attivi deve poter variare, cioè un operatore dell'azienda deve poter inserire o rimuovere un server in qualsiasi momento, inoltre il sistema deve essere in grado di accorgersi se un server è caduto e quindi estrometterlo fintantoché non venga ripristinato, il tutto senza interrompere il servizio.
Il numero dei client contemporaneamente connessi può variare in modo dinamico e soprattutto non è noto a priori.
Inoltre ogni client deve essere sequenziale mentre i server devono poter gestire più richieste in contemporanea (ovviamente le richieste proverranno da client diversi).
Perché ogni client possa sempre usufruire del servizio è necessario che conosca tutti i server, aggiornandone dinamicamente la lista (locale) qualora avvengano delle variazioni.
Infatti se ogni client conoscesse solo un server, qualora questo cadesse, tutti i client ad esso referenti non sarebbero più in grado di effettuare delle operazioni.
Tutti i server sono attivi, a fronte di un nuovo ordine (o cancellazione) l'operazione effettuata sul database dovrà propagarsi a tutti i server e quindi essere registrata.
Occorre che i server si coordinino per mantenere uno stato consistente.
I guasti che si ritengono possibili (e che dovranno essere tollerati dal sistema) sono :

• **Caduta di un client che sta effettuando una operazione di scrittura** : il sistema dovrà rigettare la richiesta se non viene completato il protocollo previsto; qualora il client non avesse carichi pendenti il sistema ignora l'avvenimento .
• **Caduta di un server inattivo** : Il sistema dovrà accorgersene e provvedere ad estromettere tale server
• **Caduta di un server che sta effettuando una operazione di scrittura** : Il sistema dovrà decidere se completare o meno la transazione quindi dovrà estromettere il server caduto ed informare il client dell'esito della richiesta agganciandolo ad un altro server (in modo trasparente all'utente) .

Funzionalità richieste

- **Server**
  - Memorizzare un nuovo ordine e ridurre la disponibilità dei prodotti richiesti in modo coerente (operazione che necessita di accurati controlli per garantire la consistenza dei dati)

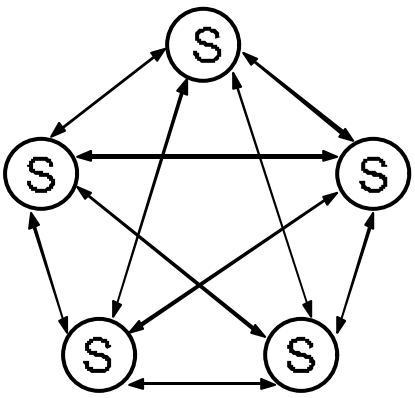
- Fornire ai clienti le informazioni sui prodotti disponibili
- Consentire l'inserimento di nuovi prodotti nel catalogo
- Consentire la rimozione dei prodotti non più disponibili
- Offrire un supporto affidabile per la gestione della concorrenza fra client
- Aggiungersi ai server attivi (nuovo server)
- Disconnettersi dal sistema (chiusura di un server)
- Riconoscere la caduta di un altro server e quindi informare gli altri.
- Fornire ai client una lista dei server disponibili

- **Client**
  - Fornire all'utente la lista dei prodotti disponibili
  - Consentire l'inoltro di un ordine
  - Consentire la rimozione di un ordine
  - Richiedere una lista dei server disponibili (operazione invisibile all'utente)

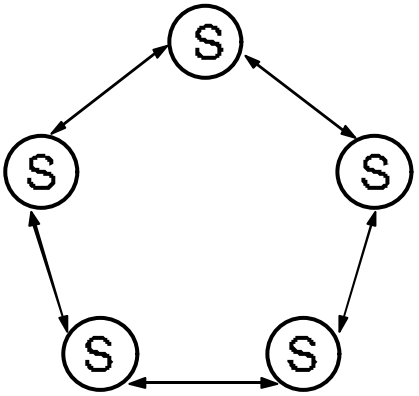
Occorre inoltre stabilire una politica di gestione per la concorrenza.
Qualora due utenti richiedano contemporaneamente l'ultimo prodotto di un lotto si distinguono le seguenti strategie.
Se entrambe le richieste pervengono allo stesso server non ci sono problemi perché l'archivio dati è acceduto in mutua esclusione, così solo una richiesta sarà accettata mentre l'altra sarà rifiutata.
Se gli ordini arrivano su server differenti, poiché l'archivio dei rispettivi server dà il prodotto come disponibile entrambe le richieste verranno registrate localmente e quindi comunicate agli altri server.
A questo punto un meccanismo di controllo distribuito dovrà portare alla registrazione di una sola delle prenotazioni provocando l'undo della registrazione abortita (sui server interessati).

Struttura dei server

L'interazione potrebbe essere di tipo "tutti con tutti", dato il basso numero di server presenti (assumibile intorno alle 6 unità) , consentendo un aggiornamento rapido dell' archivio.
Purtroppo tale soluzione causa un forte sovraccarico di comunicazione per ogni nodo che deve effettuare solo dei broadcast .
Sembra allora più idonea una struttura ad anello in cui ogni server deve conoscere solamente il suo predecessore ed il suo successore nella catena.
Tale soluzione è accettabile anche per quanto riguarda i tempi di latenza nell'aggiornamento degli archivi dato lo scarso numero di server.



Sistema tutti con tutti



Sistema ad anello

Per realizzare la struttura preposta ogni nodo della rete dovrà conoscere chi lo precede e chi lo segue nella catena.

La conoscenza del predecessore tornerà utile nella gestione dei conflitti e dei guasti.
I server scambieranno messaggi in ambo i sensi:

- **Avanti** : per comunicare variazioni all'archivio dei prodotti e degli ordini e per messaggi di controllo
- **Indietro** : per messaggi di controllo e recovery

Il funzionamento del sistema è esprimibile come segue :
Un client effettua una richiesta dei prodotti disponibili quindi inoltra un ordine al server a cui è collegato .
Tale server controlla l'ordine effettuato, se vi sono inconsistenze abortisce l'operazione di registrazione dell' ordine ed informa il client .
Se l'ordine è corretto, esegue la registrazione dell'ordine nel suo archivio locale, quindi invia un messaggio al server che lo segue lungo la catena per imporgli di aggiornare il suo archivio, questo registrerà l'ordine quindi passerà il messaggio al successivo server, l'operazione si ripete finché il messaggio non giunge al nodo che l'ha inviato; quest'ultimo concluderà la procedura informando il client dell'avvenuta registrazione.

Semplificazioni introdotte

Nella realizzazione del progetto verranno introdotte le seguenti semplificazioni :

- **Archivio** : il database preposto alla gestione dei prodotti sarà semplificato con tabelle dinamiche in memoria e con file di testo opportunamente formattati, questa scelta, inaccettabile in termini di efficienza e sicurezza, per un progetto reale, viene introdotta in quanto non strettamente legata agli obiettivi del corso ed inoltre potrà essere sostituita con una soluzione performante in un secondo momento, senza richiedere eccessive variazioni alla parte inerente le tematiche di rete.
- **Sicurezza** : viene trascurata in quanto non legata alle tematiche che si intende trattare, inoltre sarà possibile aggiungerla in un secondo momento senza intaccare la struttura dei protocolli di comunicazione adottati.

Progetto del sistema

Scelta del linguaggio

Dovendo realizzare un sistema basato sull'uso di internet ed in cui i client posso essere eseguiti su sistemi eterogenei appare conveniente l'utilizzo del linguaggio Java in quanto più adatto a soddisfare questi requisiti.

Supporto di comunicazione

Volendo garantire che tutti i messaggi inviati, sia fra client e server, che fra server e server siano consegnati correttamente, senza appesantire eccessivamente il codice dei server, la scelta migliore sembra essere quella di utilizzare delle connessioni di tipo Socket Stream.
Le connessioni fra client e server resteranno aperte durante tutta la sessione di utilizzo da parte di un client, tale scelta, molto onerosa per i server, è motivata dalla volontà di privilegiare i client riducendone i tempi di latenza fra una operazione e la successiva (i client hanno un comportamento sequenziale), questa scelta è dovuta dalla volontà di massimizzare la customer satisfactions, parametro fondamentale per ogni sistema commerciale.
Anche la comunicazione tra server avverrà su Socket Stream, queste saranno mantenute sempre aperte.
Tali socket, utilizzate per trasmettere le informazioni che devono essere propagate fra i server, saranno due per ogni nodo, inoltre si prevede che il flusso dei dati abbia dei tempi morti estremamente ridotti.
La scelta fatta avvantaggia i client che riceveranno una risposta più sollecita.

Ogni server avrà due tipi di connessioni:

- **Client** : in cui una server socket attenderà le richieste degli utenti generando un thread servitore per ogni cliente.
- **Server** : in cui una server socket ed una socket consentiranno il passaggio delle informazioni fra server e server, in modo sequenziale, quindi con un unico thread

Il server sarà composto di più thread, ognuno dei quali si pone in ascolto su una differente socket, restando bloccato fino all'arrivo di un messaggio.

Conoscenza pregressa

Il sistema si presenta all'operatore come una catena di server, ogni server dovrà conoscere all'avvio l'indirizzo, inteso come indirizzo IP e porta, del server che lo segue, detto nextServer, e che lo precede, detto prevServer.
Dinamicamente acquisisce il nome logico dei server presenti, durante i protocolli di inserzione di un nuovo server o durante il protocollo di recovery in caso di crash.
I client devono, allo stato attuale del progetto, conoscere tutti gli indirizzi dei server, questa condizione apparentemente stringente viene motivata con la scelta di comunicare ai vari utenti, per scopi propagandistici, le evoluzioni del sistema.

Sarà comunque possibile passare ad una modalità completamente trasparente semplicemente espandendo i messaggi delegati all'aggiornamento dello stato, e discussi in seguito, sia lato client che lato server.
I client mantengono una lista dei server disponibili, l'ordine con cui sono registrati i nodi della catena corrisponde all'ordine con cui i client tentano l'accesso ai vari server.
Tale lista viene ordinata in base alle informazioni che i client ricevono connettendosi al server.
Il sistema mantiene un proprio clock logico, incrementato ogni volta che la catena dei server subisce una variazione.
Ogni server contattato da un client verifica se questo ha lo stato aggiornato, in caso contrario nell'acknowledge alla connessione gli comunica anche lo stato attuale del sistema.
Al successivo accesso il client utilizzerà le indicazioni ricevute.
Il client dopo aver ricevuto l'acknowledge verifica se il suo clock logico corrisponde a quello del server e se necessario aggiorna le informazioni sullo stato.
Le informazioni sullo stato sono rappresentate dal clock logico e dalla lista dei server attivi.
La conoscenza da parte dei client di tutti i potenziali server è necessaria per garantire che il client possa tentare l'accesso a tutti i server prima di dichiarare al cliente l'impossibilità di effettuare un ordine.
Per bilanciare il carico dei client sui server si adotta una politica non ottimale ma a basso overhead.
Dato che i client utilizzano l'ordine della propria lista dei server per determinare il nodo di riferimento, per ogni client che necessita di un aggiornamento dello stato la lista dei server attivi presente localmente sul nodo viene fatta scorrere di una posizione.
In tal modo, a fronte di una variazione della catena, ogni server ridistribuisce i suoi clienti su tutti i nodi della catena.
Per un numero elevato di client autorizzati tale soluzione consente una ridistribuzione del carico soddisfacente senza richiedere pesanti protocolli globali.

Archiviazione Dati

Per memorizzare le informazioni ogni server dispone di un oggetto che fornisce le astrazioni necessarie a rendere invisibili le operazioni di accesso ai dati.
Tale classe, detta database, contiene quattro tabelle dinamiche estratte in fase di avvio dai file di supporto.
Le tabelle sono le seguenti :

- **orderTable** : Tabella degli ordini, contiene tutti gli ordini registrati.
- **PendingTable** : Tabella degli ordini pendenti non ancora registrati (il concetto di ordine pendente sarà chiarito in seguito).
- **ProdTable** : Tabella dei prodotti disponibili, memorizza il numero di pezzi disponibili per ogni prodotto.
- **ProdPendingTable** : Tabella dei prodotti già richiesti ma non ancora registrati.

Su tali tabelle vengono effettuate le operazioni di aggiunta di una nuova entry e di cancellazione di una entry.
La classe database ha un'unica istanza per ogni server ed ogni thread può invocarne le API, attraverso il riferimento alla classe Server di cui dispone.
Le api fornite dalla classe database sono :

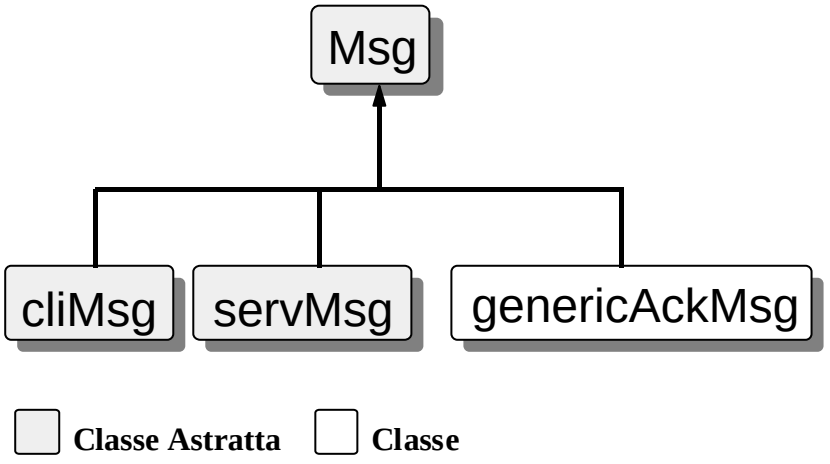
- **addOrder**: Inserisce un ordine nella coda dei pendenti in attesa di poterlo registrare.
- **checkOrder**: Verifica la possibilità di registrare un ordine con le risorse disponibili.

- **checkPendingOrder**: verifica la possibilità di registrare un ordine considerando anche l'eventualità di rimuovere un ordine pendente.
- **delOrder**: Rimuove un ordine dalla coda dei pendenti, l'ordine rimosso potrà essere inserito nella coda degli ordini registrati oppure potrà essere eliminato a causa di un conflitto.
- **flushDb**: Scarica le tabelle dinamiche aggiornando il contenuto del Database su disco.
- **initDb**: Inizializza le strutture dinamiche del database leggendole dall'archivio fisico.
- **readProdDB**: Restituisce un Array contenente la lista di tutti i prodotti disponibili per la vendita.
- **registerOrder**: Registra un ordine nella lista degli ordini.
- **unRegisterOrder**: Rimuove un ordine dalla orderTable, la rimozione può essere dovuta ad una richiesta del cliente che l'ha effettuato oppure potrà essere causata da un conflitto.
- **updateDb**: Permette di aggiornare il proprio database ricevendone il contenuto da remoto.
- **uploadDb**: Trasferisce il contenuto del proprio database ad un server che ne fa richiesta

Per garantire la consistenza dei dati gli accessi sono in mutua esclusione.

Messaggi

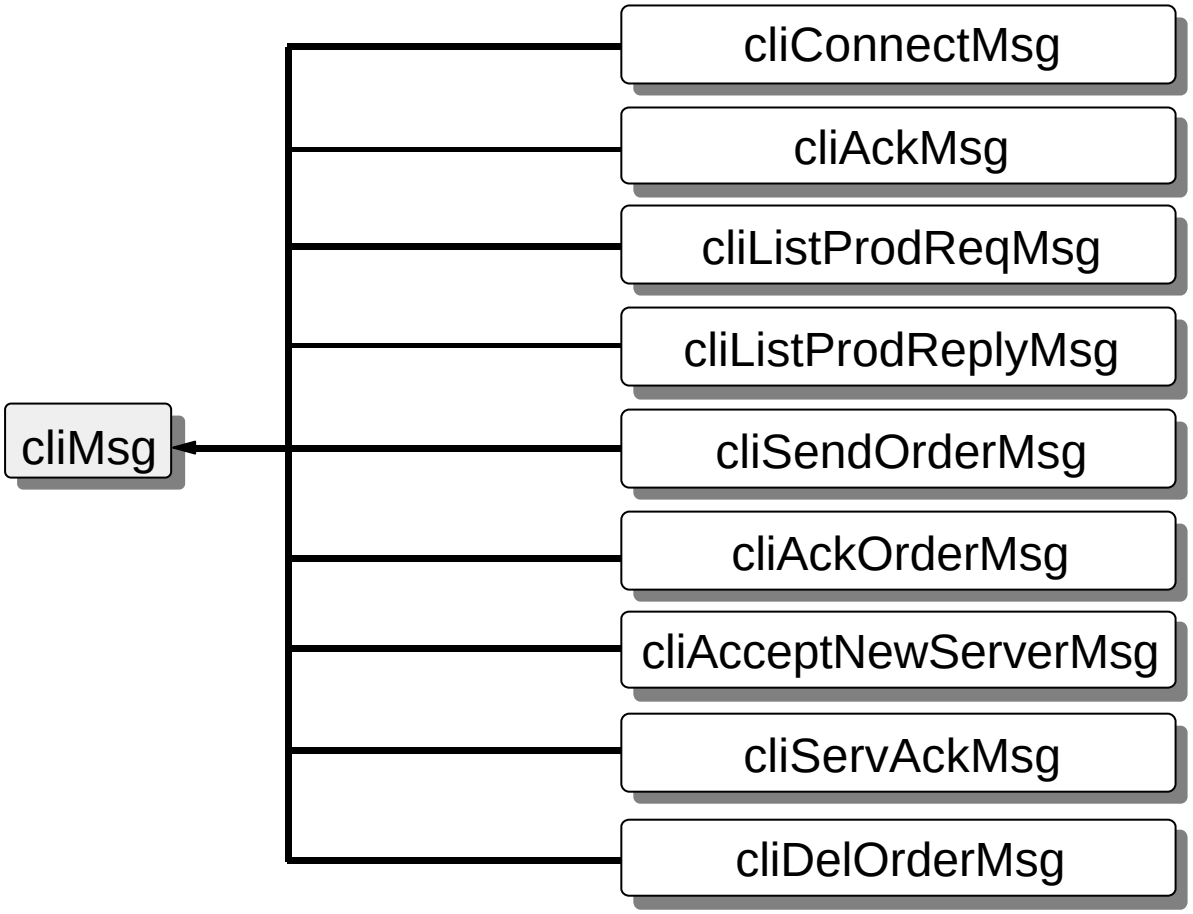
Le informazioni che girano nella rete sono distinte fra comunicazioni client-server e server-server, tali messaggi saranno incapsulati in oggetti per sfruttare le caratteristiche del linguaggio e semplificare la gestione delle interazioni .
Nasce spontanea l'idea di strutturare in modo gerarchico le tipologie di messaggi, definendo in primis una classe astratta da cui verranno specializzate le varie tipologie .
Le figure seguenti illustrano l'insieme dei messaggi e la loro gerarchia.



In questa figura viene rappresentata la struttura base della gerarchia delle classi Msg .

- **Msg** : Classe base da cui si specializzano tutte le classi che rappresentano un messaggio. L'unica informazione che introduce è il codice del messaggio, comune a tutte le sue sottoclassi. E' una classe astratta.
- **CliMsg** : E' la classe da cui si specializzano tutti i messaggi utilizzati nelle comunicazioni fra i client e i server. Non introduce nessuna informazione. E' una classe astratta.
- **servMsg** : Funge da classe radice per tutti i pacchetti usati nei dialoghi server con server. Introduce come informazione il codice del server che inoltra il pacchetto. E' una classe astratta.

- **GenericAckMsg** : Acknowledge generico, non fornisce nessuna informazione. Viene utilizzato per creare dei punti di sincronizzazione nelle comunicazioni.



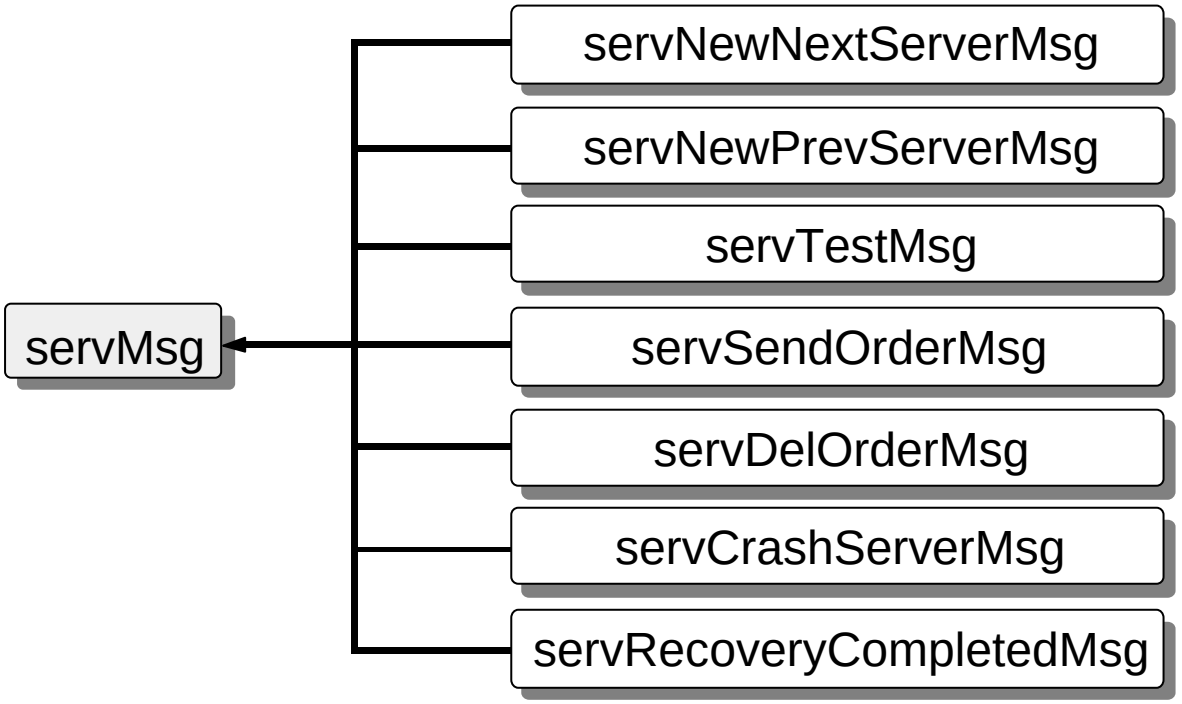
Messaggi fra client e server

- **cliConnectMsg** : Pacchetto inviato dal client subito dopo la connessione al server. Consente di informare il server dello stato del client. Include al suo interno il clock logico ed il codice del cliente
- **cliAckMsg** : Acknowledge alla connessione, è usata dal server per fornire al client l'accettazione e le informazioni sullo stato attuale del sistema. Include al suo interno il clock logico del server, il codice del cliente come ridondanza di controllo e le informazioni sui server attivi.
- **cliListProdReqMsg** : Permette al client di richiedere la lista dei prodotti disponibili.
- **cliListProdReplyMsg** : Permette al server di rispondere al cliListProdReqMsg inoltrato dal client. Contiene tutte le informazioni sui prodotti disponibili.
- **cliSendOrderMsg** : Inoltra al server la richiesta di registrazione di un ordine. Contiene il codice del cliente, la lista dei prodotti richiesti e le loro quantità
- **cliAckOrderMsg** : Permette al server di informare il client dell'esito di un ordine. Contiene il codice dell'ordine effettuato, il codice del server che conferma l'ordine, il codice del cliente che ha inviato l'ordine ed il risultato della transazione effettuata.

Tale risultato può essere :

- ORDER_ACCEPT : Ordine accettato
- END_PRODUCT : Prodotti esauriti
- SOCKET_BROKE : Interruzione della comunicazione
- ERROR_ON_CODE : Errore di esecuzione
- UNKNOW_ERROR : Errore sconosciuto

- **cliAcceptNewServerMsg** : Permette di attivare il protocollo di registrazione di un nuovo server nella catena. Contiene il codice del server che effettua la richiesta di connessione e l' indirizzo della sua previous Socket.
- **cliServAckMsg** : Utilizzato come risposta al cliAcceptNewServerMsg, con tale messaggio si accetta la richiesta di un nuovo server che vuole inserirsi nella catena e gli si passano le informazioni sullo stato del sistema. Contiene il codice del server, il clock logico del sistema, la conferma se si accetta la connessione e la lista dei server attivi.
- **cliDelOrderMsg** : Inoltra la richiesta di rimozione di un ordine precedentemente effettuato.Contiene il codice dell'ordine da rimuovere e il codice del cliente che ha effettuato l'ordine.



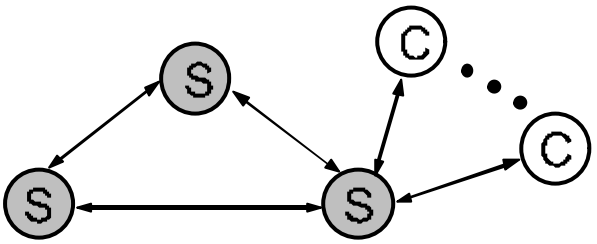
Messaggi fra server e server

- **servNewNextServerMsg** : è inviato dal nodo che ha ricevuto la richiesta di registrazione di un nuovo server . Consente di informare il previous che la connessione verrà chiusa e che dovrà riconnettersi al nuovo next server. Contiene l'Indirizzo del nuovo next server ed il clock logico del sistema.
- **servNewPrevServerMsg** : Informa il server che ha ricevuto servNewNextServerMsg in precedenza che può procedere all'attivazione del canale next. Contiene informazioni analoghe a quelle del servNewNextServerMsg.
- **servTestMsg** : Messaggio utilizzato per verificare lo stato dell'anello dei server. Contiene la lista dei server attraversati da questo pacchetto.

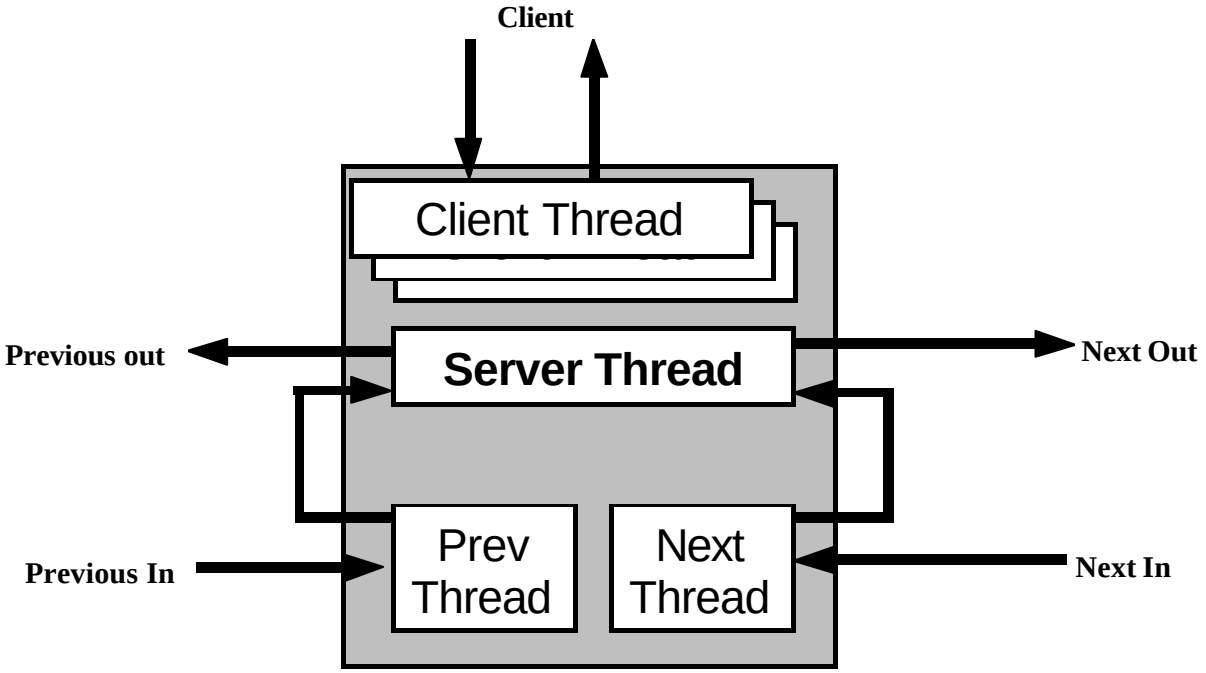
- **servSendOrderMsg** : Propaga la richiesta di registrazione di un nuovo ordine a tutti i nodi della catena. Contiene il codice dell' ordine che si sta inoltrando, il peso del ordine, necessario in caso di conflitto e il cliSendOrderMsg inviato dal cliente per effettuare l'ordine.
- **servDelOrderMsg** : Propaga la richiesta di cancellazione di un ordine a tutti i nodi della catena. Contiene il codice dell' ordine e del cliente che l'ha precedentemente inoltrato
- **ServCrashServerMsg** : Segnala la caduta di un server agli altri elementi della catena provocando un aggiornamento dello stato. Contiene l'indirizzo del nuovo next server, il clock logico del sistema e i codici dei server ancora attivi.
- **ServRecoveryCompletedMsg** : Informa tutti gli anelli della catena del completamento del protocollo di recovery. Contiene i codici dei server ancora attivi.

Struttura del Server

La catena dei server a regime si presenta come da figura.



Il dialogo fra client e server interessa tutti gli anelli della catena solo per le operazioni che influiscono sul database.
La struttura interna di un server è la seguente :

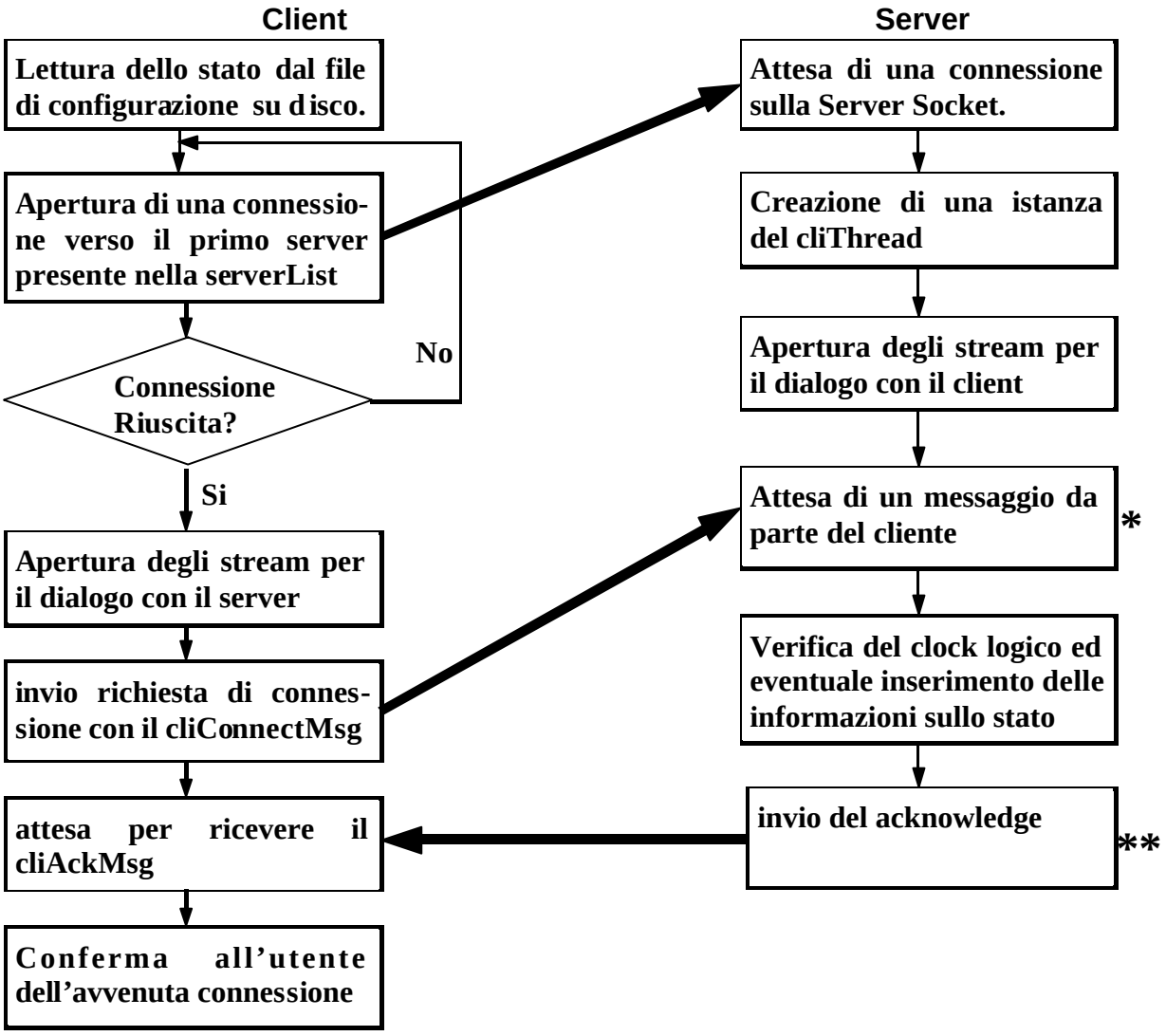


Come mostrato in figura i server sono composti dai seguenti thread:

- **prevThread** : a cui si demanda il compito di ricevere i messaggi dal Previous server, di eseguire le mansioni richieste da tali messaggi (operazioni di controllo e gestione database) e quindi inoltrarne comunicazione al nextServer, attraverso il canale di next out invocabile attraverso la funzione sendNextMsg(Msg mess) presente nella classe server.
- **nextThread** : a cui si demanda il compito di ricevere i messaggi dal Next server, di eseguire le mansioni richieste da tali messaggi (solo operazioni di controllo) e quindi inoltrarne comunicazione al previousServer, attraverso il canale di previous out invocabile attraverso la funzione sendPrevMsg(Msg mess) presente nella classe server.
- **cliThread** : Si incarica di gestire l'interazione con uno specifico cliente. Ne viene creata una istanza per ogni client che si connette al server. Le istanze permangono finché il client resta connesso al server.

Le socket di comunicazione fra server sono aperte e mantenute dal thread principale; nextThread e previousThread , all'atto della loro creazione, ricevono solo uno stream di input su cui si bloccano in attesa dei messaggi .
Le funzionalità di output sono fornite direttamente dal thread server attraverso le succitate funzioni.
La scelta di conglobare le operazioni sui dati e sullo stato dopo la ricezione delle informazioni rende inutile la presenza di thread delegati alla gestione del output, anzi, tale presenza renderebbe solo più complessi i meccanismi di comunicazione .
I thread operano in modo sequenziale, restando bloccati fino all'arrivo di un messaggio, eseguono le operazioni richieste e se necessario lo inoltrano al prossimo nodo della catena, quindi tornano ad attendere l'arrivo di un altro messaggio.
La presenza del codice del server in ogni pacchetto che gira sulla catena consente di verificare chi l'ha originato, in questo modo, terminato il giro, tale pacchetto verrà elaborato ed eliminato.
Il thread Server resta in attesa delle richieste di operazioni che possono provenire attraverso la serverSocket destinata agli utenti e che causerà l'attivazione di un cliThread oppure dalla interfaccia grafica che consente all'operatore di compiere azioni di verifica dello stato del server locale e della catena.

Protocollo di accesso e dialogo Client – Server



Per il client il ciclo di controllo sulla connessione viene ripetuto per tutti i server noti, qualora nessun tentativo vada a buon fine verrà segnalata all'utente l'impossibilità di instaurare una connessione.
Il server utilizza uno switch per distinguere il tipo di messaggio ricevuto e quindi effettuare le giuste operazioni.
Le operazioni fra * e ** sono incorporate in un ciclo che termina con la chiusura della socket.

Tutte le operazioni sono del tipo Domanda – Risposta, con il client che inoltra una richiesta ed il server che fornisce una risposta.
Alla chiusura della connessione da parte del client il cliThread termina e le risorse occupate vengono recuperate dal Garbage collector di Java.

Le operazioni che il client può richiedere sono :

- **Connessione** : illustrata in figura .

- **Richiesta della lista dei prodotti disponibili** : il client inoltra il messaggio cliListProdReqMsg al server, questo preleva dal database la lista dei prodotti e la impacchetta nel cliListProdReplyMsg che rinvia al cliente.

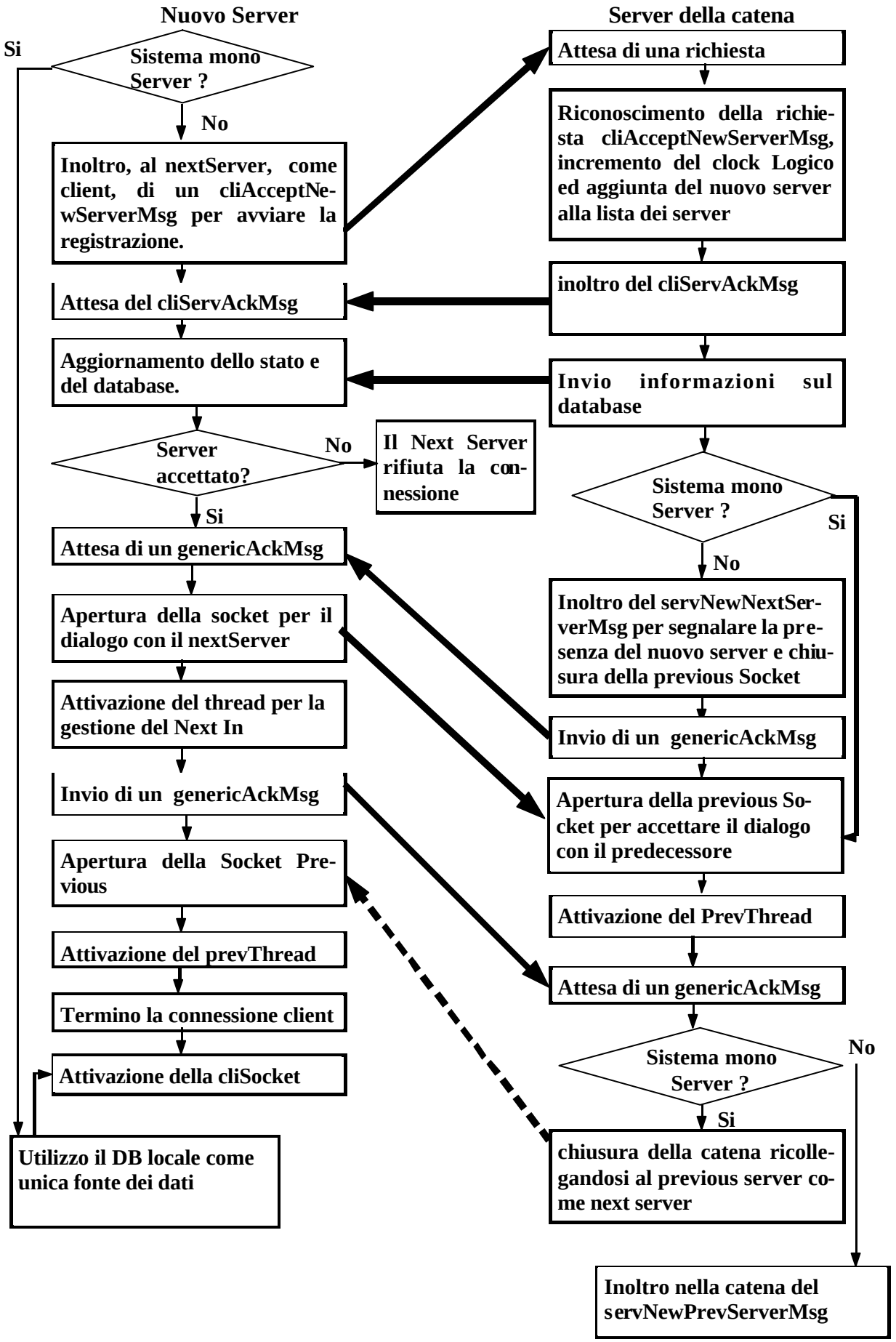
- **Inoltro di un ordine** : il cliente inoltra un cliSendOrderMsg, il server verifica la disponibilità dei prodotti quindi inoltra nella catena un servSendOrderMsg, se tale messaggio ritorna oppure se giunge una richiesta di rimozione dell'ordine corrente il server risponde al client con un cliAckOrderMsg in cui ci sarà la conferma o il fallimento dell'ordinativo .
  La risposta al client è separata dalla fase di registrazione dell'ordine.
  Precisamente il cliThread dopo aver inoltrato l'ordine nella catena si registra in una apposita coda del server e ritorna ad attendere le richieste del cliente.
  Il servSendOrderMsg quando ripassa sul nodo che l'ha generato innesca l'acknowledge al cliente prelevando dalla suddetta coda il riferimento al thread che ha innescato la richiesta (attraverso il codice dell'ordine che garantisce l'unicità) ed invocando la funzione cliAckOrder(orderCode) del cliThread che inoltra al cliente la conferma dell'ordine.
  Questo meccanismo permette di non incappare in spiacevoli "blocchi" del server ed inoltre si rivelerà utile nella gestione dei messaggi circolanti durante un crash.
  Il codice del ordine viene generato dal server aggiungendo al proprio codice un numero ricavato dalla data (anno, mese, giorno, ore, minuti, secondi).
  Questa tecnica garantisce l'unicità in quanto il nome del server assicura che non vi possano essere due ordini con lo stesso codice anche se gli orologi delle varie macchine non sono sincronizzati.

- **Rimozione di un ordine** : il cliente inoltra un cliDelOrderMsg, il server verifica il codice dell'ordine e del cliente quindi inoltra nella catena un servDelOrderMsg. Quanto tale messaggio ritorna, il server risponde al client con un cliAckOrderMsg in cui ci sarà la conferma o il fallimento della cancellazione. Anche in questo caso si utilizza il meccanismo di conferma esposto per l'inoltro di un ordine.

- **Disconnessione** : non si utilizzano messaggi, il client chiude la socket ed il server ne prende atto attraverso un eccezione. Il client thread termina ed il garbage collector ne recupera le risorse.

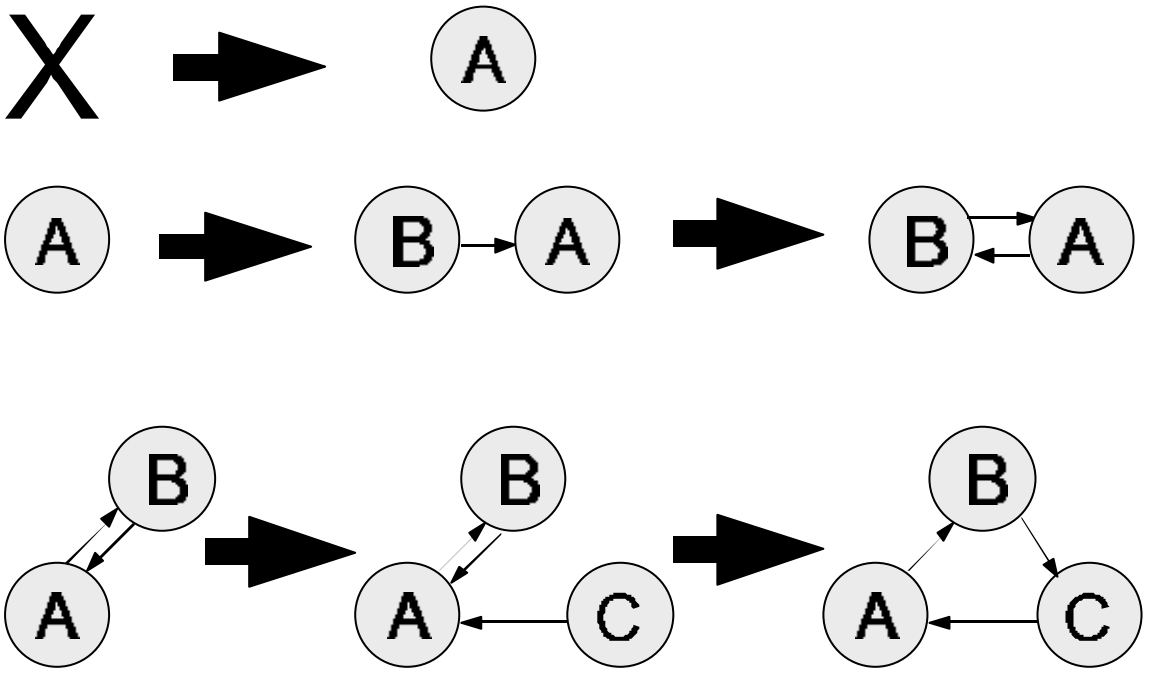
Protocollo di registrazione di un nuovo server

Il protocollo viene distinto dal caso di crash sulla base dei messaggi che vengono inoltrati nella catena.
Inoltre il next del nuovo server, chiude volontariamente la comunicazione on il precedente previous al fine di consentire l'inserimento del nuovo server .
Tale protocollo è descritto nella seguente figura .


Nuovo Server
Server della catena
Si
Sistema mono Server ?
No
Inoltro, al nextServer, come client, di un cliAcceptNewServerMsg per avviare la registrazione.
Attesa del cliServAckMsg
Aggiornamento dello stato e del database.
Server accettato?
No
Il Next Server rifiuta la connessione
Si
Attesa di un genericAckMsg
Apertura della socket per il dialogo con il nextServer
Attivazione del thread per la gestione del Next In
Invio di un genericAckMsg
Apertura della Socket Previous
Attivazione del prevThread
Termino la connessione client
Attivazione della cliSocket
Utilizzo il DB locale come unica fonte dei dati
Attesa di una richiesta
Riconoscimento della richiesta cliAcceptNewServerMsg, incremento del clock Logico ed aggiunta del nuovo server alla lista dei server
inoltro del cliServAckMsg
Invio informazioni sul database
Sistema mono Server ?
Si
No
Inoltro del servNewNextServerMsg per segnalare la presenza del nuovo server e chiusura della previous Socket
Invio di un genericAckMsg
Apertura della previous Socket per accettare il dialogo con il predecessore
Attivazione del PrevThread
Attesa di un genericAckMsg
Sistema mono Server ?
No
Si
chiusura della catena ricollegandosi al previous server come next server
Inoltro nella catena del servNewPrevServerMsg

Il nuovo server verifica inizialmente di non essere l'unico attivo, nel qual caso si predispone a ricevere i clienti evitando l'attivazione dei thread per la gestione della catena.
Se esiste già almeno un server (informazione nota dal file di configurazione) il nuovo server si connette al nodo che diventerà il suo next come client ed attiva il protocollo di registrazione.
Dopo aver aggiornato lo stato ed il database se la sua richiesta non viene accettata (eventualità possibile solo se va in crash il next) interrompe la procedura e si chiude altrimenti si predispone a riceve l'acknowledge alla creazione delle socket e dei thread necessari per le comunicazioni nel anello.
Il nodo della catena che ne riceve la richiesta se è l'unico server attivo crea i thread next e previous, precedentemente non attivi, per interfacciarsi al nuovo server creando così una catena di due macchine.
La condizione di monoServer viene modificata solo al termine della procedura di registrazione del nuovo nodo.
Se la catena ha più nodi attivi allora il next del nuovo server chiude e riapre la sua previousSocket dopo aver informato il vecchio previous, quindi inoltra nella catena un messaggio di riconnessione.
Il server che avrà la nextSocket caduta riconoscerà come suo il messaggio e si connetterà al nuovo nodo.

Nella figura seguente vengono esemplificati i tre casi possibili.

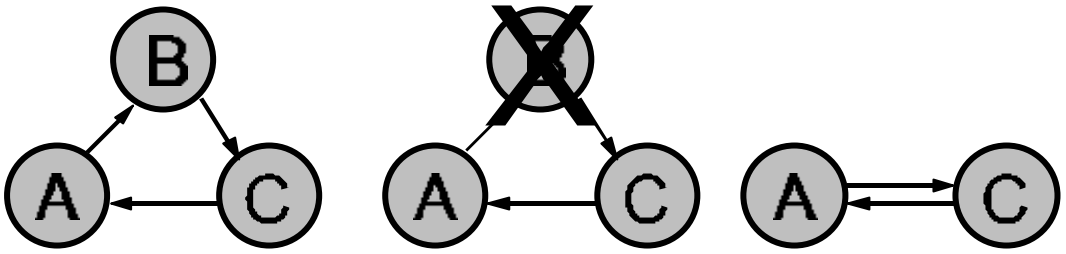


Nel primo caso si vede l'attivazione del primo server.
Nel secondo, il caso in cui sia già attivo un server e vi si voglia aggiungere il secondo.
Nel terzo vediamo la situazione di una rete potenzialmente già a regime con due o più server in cui il nuovo arrivato si inserisce fra il nodo contattato per la registrazione ed il suo predecessore.
La scelta di dove verrà collocato tale server è effettuabile tramite il file di configurazione specificando l'indirizzo del previous server desiderato.

Per questa procedura non vengono eseguiti forti controlli per garantire la consistenza dei messaggi in circolazione.
Questo è comprensibile in quanto, mentre un crash non può essere previsto, l'inserzione di un nuovo server è pilotata dal gestore del sistema che quindi può approfittare dei tempi morti in cui non si prevedono client connessi, come le ore notturne.

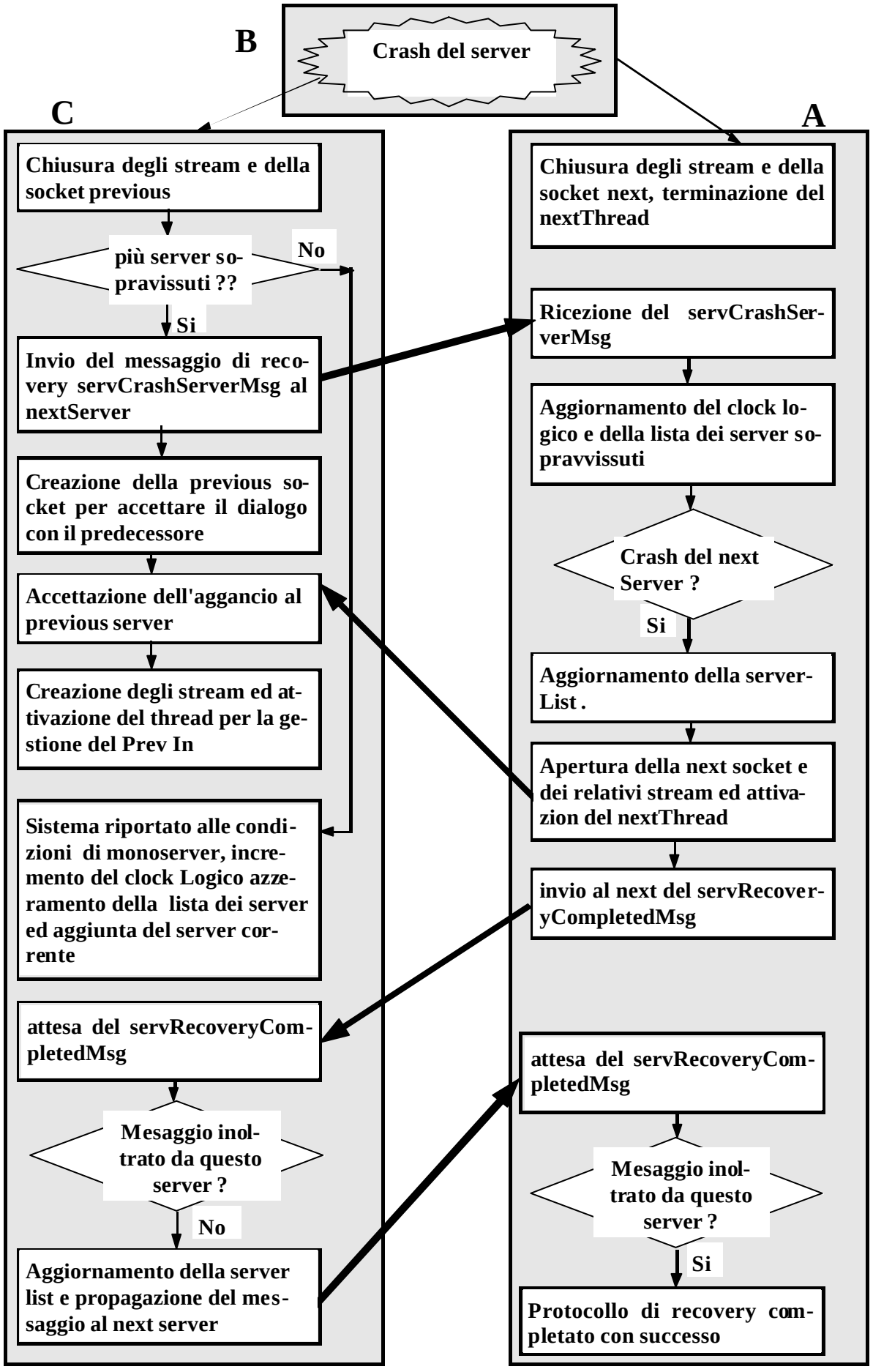
Protocollo di Recovery

Per descrivere il comportamento di tale protocollo è conveniente calarsi in un caso reale.
Assumendo di avere tre server attivi, denominati rispettivamente A, B e C e supponendo che sia il server B a cadere, le fasi di recupero della catena sono riportate in figura



FASE 1 **FASE 2** **FASE 3**

Durante il crash i messaggi generati o che stanno transitando sul server B e tutte le connessioni con i clienti saranno inevitabilmente perse.
I messaggi che hanno già superato B e che sono stati generati da un server ad esso precedente (A) termineranno correttamente il loro cammino.
I restanti messaggi verranno abortiti .
Questa politica porta ad una perdita statistica del 50 % dei messaggi presenti, tale scelta risulta un buon bilanciamento tra la pesantezza del protocollo di recovery, che risulta piuttosto snello, e la capacità di recupero del servizio.
Aggiungendo a ciò la constatazione che l'eventualità di crash non è elevata la soluzione risulta accettabile.
Per garantire il rigetto di tutte le operazioni pendenti legate a messaggi abortiti torna utile la tecnica di acnowledge utilizzata per confermare gli ordini.
Infatti il servRecoveryCompletedMsg, inviato al termine della fase di recovery, oltre alle succitate mansioni innesca anche la rimozione di tutti i cliThread in attesa di un acknowledge dalla coda del server e per ognuno provoca l'invio al client di un fallimento dell'operazione.
Inoltre elimina gli ordini pendenti che vengono rigettati ed invia nella catena il messaggio di cancellazione di tali ordini, queste ultime operazioni sono necessarie per garantire la consistenza del database.
Nella pagina seguente viene riportato in dettaglio il comportamento del protocollo di recovery per il caso illustrato nella precedente figura.

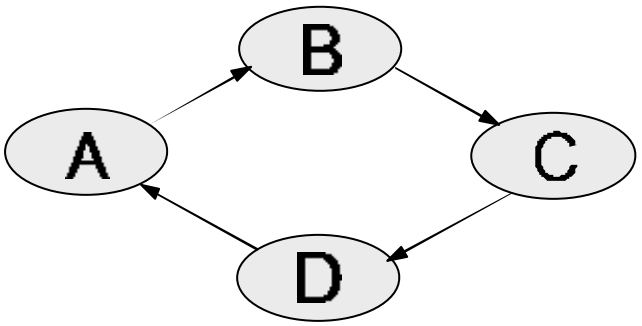

B
Crash del server
C
A
Chiusura degli stream e della socket previous
più server sopravissuti ??
No
Si
Invio del messaggio di recovery servCrashServerMsg al nextServer
Creazione della previous socket per accettare il dialogo con il predecessore
Accettazione dell'aggancio al previous server
Creazione degli stream ed attivazione del thread per la gestione del Prev In
Sistema riportato alle condizioni di monoserver, incremento del clock Logico azzeramento della lista dei server ed aggiunta del server corrente
attesa del servRecoveryCompletedMsg
Mesaggio inoltrato da questo server ?
No
Aggiornamento della server list e propagazione del messaggio al next server
Chiusura degli stream e della socket next, terminazione del nextThread
Ricezione del servCrashServerMsg
Aggiornamento del clock logico e della lista dei server sopravvissuti
Crash del next Server ?
Si
Aggiornamento della serverList .
Apertura della next socket e dei relativi stream ed attivazion del nextThread
invio al next del servRecoveryCompletedMsg
attesa del servRecoveryCompletedMsg
Mesaggio inoltrato da questo server ?
Si
Protocollo di recovery completato con successo

Dopo il crash del server B, il server C chiude la propria previous socket e se si ritrova come unico server, in quanto anche il proprio next thread è terminato, reimposta lo stato come unico server attivo, aggiorna il clock e resta in attesa dei client.
Se altri server sono sopravvissuti, in quanto la connessione next è ancora attiva invia nell'anello il messaggio servCrashServerMsg per avviare il protocollo di recovery; quindi si pone in attesa della connessione di un nuovo previous server.
Il messaggio servCrashServerMsg si propaga nell'anello finché non trova un nodo privo del nextThread, questo verificherà la presenza delle condizioni di crash, distinguibili dal caso di inserzione di un nuovo thread per i messaggi circolanti, (si fa riferimento all'interrogazione "Crash del next Server" presente in figura, la risposta no può verificarsi solo se si è attivato il protocollo di inserzione di un nuovo thread) quindi si connetterà al suo nuovo next Server , il cui indirizzo è contenuto nel messaggio servCrashServerMsg.
L'anello è così ripristinato inoltre, durante il suo percorso, il messaggio servCrashServerMsg ha raccolto i nomi dei server ancora attivi.
Questa lista viene inclusa nel messaggio servRecoveryCompleted e fatta circolare nell'anello.
Quando ritorna al server che l'ha generato il protocollo di recovery è completato e tutti i server hanno recuperato uno stato consistente.

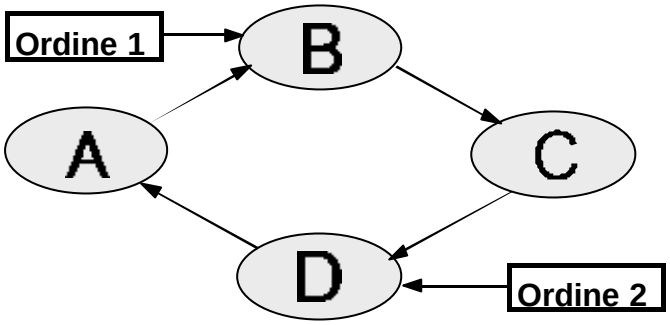
Concorrenza

Il problema della concorrenza, come accennato in precedenza, è legato alla disponibilità dei prodotti a fronte di un ordine.
Se due ordini in conflitto pervengono allo stesso server non ci sono problemi in quanto il database locale è sempre acceduto in mutua esclusione e quindi verrà processata prima una richiesta e poi l'altra (che sarà rifiutata) .
I problemi nascono per la replicazione del database su più nodi della rete e l'imprecisabile ritardo con cui gli altri server vengo informati del nuovo ordine.
Nel discutere le soluzioni impiegate per far fronte a tali problematiche assumiamo un caso concreto di riferimento onde semplificarne l'esposizione.

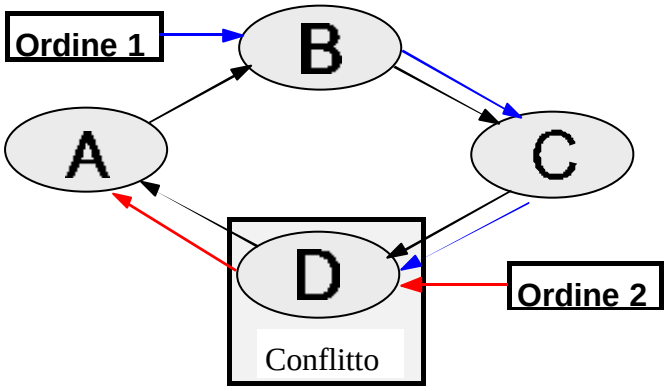
Nel caso scelto il sistema ha quattro server attivi denominati A, B, C e D connessi come da figura (le frecce nere indicano la connessione verso il next server).



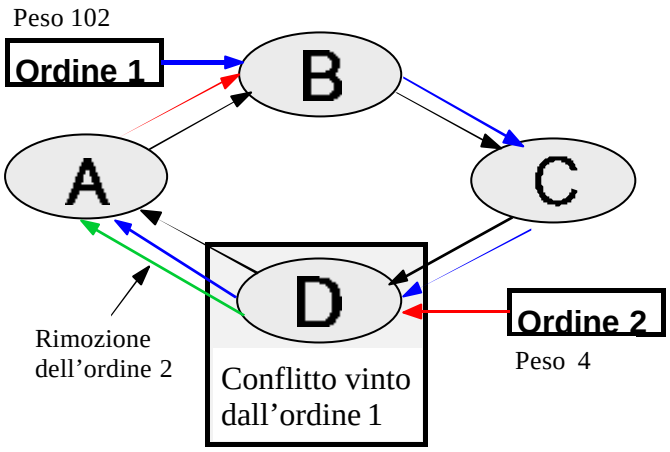
Supponiamo ora che sul server B e sul server D giungano due ordini in conflitto, cioè che siano soddisfabili con le scorte disponibili ma che non possano esserlo entrambi.



È presumibile che uno dei due ordini raggiunga per primo il server su cui viene registrato l'altro.
Anche se ciò non fosse, il protocollo adottato non ne risentirebbe riuscendo comunque a risolvere il conflitto .
L'esposizione del protocollo risulta però più chiara analizzandone il comportamento nella situazione sopraccitata.



Per risolvere il problema delle collisioni ogni ordine inoltrato nella catena non viene registrato subito dal server che lo riceve ma viene inserito in una apposita coda degli ordini pendenti, verrà registrato correttamente solo quando la sua richiesta di registrazione ritornerà a lui.
Durante l'attraversamento dei vari nodi, per un ordine, viene verificata la disponibilità delle risorse e quindi lo si registra (in modo stabile).
Se nei nodi della catena l'ordine non può essere registrato direttamente significa che si è verificato un conflitto come mostrato dalla figura precedente.
Le specifiche non forniscono una precisa politica di risoluzione dei conflitti per cui si è deciso di adottare un sistema di pesi.
Tali pesi, espressi come numeri, vengono generati come somma del numero contenuto nel nome dei server (Es. server s01 → peso 1 , server s05 → peso 5) più 100 qualora il server non abbia vinto l'ultimo conflitto a cui ha partecipato o 0 qualora l'abbia vinto.
Questa soluzione consente di ripartire in modo semplice, anche se non perfettamente bilanciato, la vittoria nei conflitti.
Tornando all'esempio considerato assumiamo che B abbia perso l'ultimo conflitto, C l'abbia vinto e che i numeri associati ai nomi dei server siano A = 1, B = 2, C = 3, D = 4 .



Come si vede in figura l'ordine 2 ha un peso minore del ordine 1 quindi il server D inoltra al next una richiesta di cancellazione del proprio ordine e lo rimuove dalla propria lista degli ordini pendenti rilasciandone i prodotti ed informando il client dell'impossibilità di completare l'operazione.
Quando riceverà il messaggio di cancellazione del ordine da lui generato ne riconoscerà la paternità ma non trovando l'ordine nella propria lista dei pendenti lo scarterà.
A questo punto il server D modifica il peso del ordine 1 eliminando il +100 (nel caso in esame il peso diventa 2) e lo inoltra normalmente .
Quando questo arriverà nuovamente a B il server constaterà che il peso è diverso da quello da lui impostato e provvederà a registrare la vittoria in un conflitto.

Il comportamento del server B a fronte della collisione è facilmente intuibile, confronterà i pesi dei propri ordini pendenti con quello del messaggio ricevuto.
Scoprendo che tale ordine ha un peso minore si limiterà a scartarlo bloccandone la propagazione.
Al ritorno della propria richiesta modificherà lo stato registrando la propria vittoria in un conflitto e procederà segnalando al client il buon esito della operazione.

La tecnica adottata garantisce la registrazione di almeno un ordine anche in presenza di più conflitti contemporanei però non assicura, in questo caso, l'alternanza delle vittorie fra i server .
Non garantisce, inoltre, che gli ordini vincitori siano quelli che massimizzano il profitto per l'azienda.

Piano dei Test

I test effettuati sono volti a verificare il corretto funzionamento del sistema ed il rispetto delle specifiche adottate.
I test sono stati condotti su un sistema locale (denominato in seguito *Locale*) in cui coesistevano più server e più client e su una rete LAN di tre macchine con un server su ogni macchina (denominato in seguito *Rete*).
Quest'ultima è utilizzata come insieme di server web per una azienda di servizi internet e per la durata del test tali servizi sono rimasti attivi onde verificare il comportamento del sistema in un caso il più possibile vicino alla realtà di utilizzo.

Le funzionalità verificate sono le seguenti :

1. Attivazione del primo server e di un client
2. Interazione fra client e singolo server
3. Attivazione di un secondo server senza client connessi
4. Attivazione di un secondo server con client connessi
5. Interazione fra client e catena con due server
6. Crash del secondo server in una catena di due server senza messaggi in circolo
7. Crash del primo server in una catena di due server con messaggi in circolo
8. Attivazione di un terzo server senza client connessi
9. Attivazione di un terzo server con client connessi
10. Interazione fra client e catena con tre server e di più client con catena di tre server
11. Crash del primo server in una catena di tre server senza messaggi in circolo
12. Crash del primo server in una catena di tre server con messaggi in circolo
13. Crash del secondo server in una catena di tre server senza messaggi in circolo
14. Crash del secondo server in una catena di tre server con messaggi in circolo
15. Crash del terzo server in una catena di tre server senza messaggi in circolo
16. Crash del terzo server in una catena di tre server con messaggi in circolo
17. Crash di due server in una catena di tre server senza messaggi in circolo
18. Crash di due server in una catena di tre server con messaggi in circolo
19. Conflitto sulla disponibilità dei prodotti

Considerazione

Durante una prima fase di test, è emerso il seguente problema.
L'inconsistenza della tabella degli ordini, adottata per velocizzare il protocollo, ed apparentemente innocua, nasconde una pericolosa insidia.
Un client che ha effettuato l'ordine su un server caduto e che era l'unico ad aver registrato tale ordine potrebbe non essere più in grado di rimuoverlo.
Per questa ragione occorrerà intervenire sulle specifiche imponendo che anche il database degli ordini ricevuti sia trasferito al momento dell'inserzione di un nuovo server nella catena.
Ovviamente tale soluzione inciderà sulle performance del sistema, soprattutto in un caso reale ove la mole dei dati è considerevole.
I Seguenti test sono stati effettuati dopo aver introdotto la succitata modifica.

Esito dei Test

Per ogni operazione di test eseguita, vengono riportati sia gli esiti visibili al cliente sia gli stati assunti dai server.
Il server fornisce due funzioni di test della catena, una in avanti (canale next) e l'altra all'indietro (canale previous) ed una funzione che consente di visualizzare le informazioni sullo stato.
Inoltre si può verificare lo stato del database visualizzando il contenuto delle tabelle dati.
Come ipotesi iniziali si assume, che i server abbiano un clock logico iniziale pari a 1 e che i client l'abbiano uguale a zero.
Inoltre si assume che i database contengano solo i seguenti prodotti .

Quantità	Codice	Descrizione	Prezzo
100	sv01	GOLD VideoMaster GP 4MB AGP	45.000
200	sv02	GOLD VideoWizard Pro 8MB PCI	67.000
300	mb01	GOLD Powerboard Socket A VIA KT133 ATA100	214.000
400	mb02	GOLD Powerboard VIA ApPro694X AGP4X 133Mhz	160.000
500	cpu01	INTEL Celeron II 633 128k (Socket 370 Fc-Pga)	152.000
600	cpu02	INTEL Pentium 4 1.4Ghz (Socket 423 pin Pga)	999.000

I test locali che non permettevano di rilevare il comportamento atteso sono stati ripetuti introducendo un ritardo di due secondi fra la ricezione di un messaggio e la sua propagazione al nodo successivo della catena.

I risultati sono divisi in una prima specifica di ciò che ci si attende e da due commenti detti *Locale* e *Rete* che esprimono le reali rilevazioni ottenute durante le prove.

- **Attivazione del primo server e di un client**

Il client non è ancora connesso al server quindi entrambi dovrebbero essere nelle condizioni iniziali riportate nelle seguenti tabelle .

Server s01

Server Attivi	**s01**
Clock	**1**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

Client

Clock	**0**

Server attivi	**S01 Attivo – s02 Attivo – s03 Attivo**
Ordini effettuati	**Nessuno**

Locale : l'analisi dello stato conferma le ipotesi
Rete : l'analisi dello stato conferma le ipotesi

- **Interazione fra client e singolo server**

Il client ora connesso al server ha aggiornato il proprio stato vista la differenza fra i clock, quindi si ipotizza l'inoltro di un ordine di una scheda grafica sv01.
Lo stato atteso è il seguente :

Server s01

Server Attivi	**s01**
Clock	**1**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**s01200114420395**

Client

Clock	**1**
Server	**S01 Attivo - s02 Inattivo – s03 Inattivo**
Ordini effettuati	**S01200114420395**

Locale : l'analisi dello stato conferma le ipotesi.
Rete : l'analisi dello stato conferma le ipotesi.

- **Attivazione di un secondo server senza client connessi**

Il client si sconnette dalla rete ed il server s02 viene attivato, parte il protocollo di registrazione del server, il clock logico viene aggiornato ed il database dei prodotti viene passato al nuovo server.
Lo stato atteso è :

Server s01

Server attivi	**S01 – s02**
Clock	**2**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**S01200114420395**

Server s02

Server attivi	**S01 –s02**
Clock	**2**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**S01200114420395**

Il client mantiene la configurazione precedente.

Locale : l'analisi dello stato conferma le ipotesi.
Rete : l'analisi dello stato conferma le ipotesi.

- **Attivazione di un secondo server con client connessi**

Considerando il sistema inizialmente con un server attivo che ha ricevuto un ordine di una sv01 dal primo client procediamo all'attivazione di un secondo server mantenendo il client connesso.
Quest'ultimo non si avvede del cambiamento dello stato mantenendo quello attuale, verrà informato delle nuove condizioni solo al prossimo accesso.

I server si adattano ed iniziano a dialogare.
Lo stato atteso è :

Server s01

Server attivi	**S01 – s02**
Clock	**2**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**S01200114595535**

Server s02

Server attivi	**S01 – s02**
Clock	**2**
Prodotti disponibili	**Cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**Nessuno**

Client

Clock	**1**
Server	**S01 Attivo - s02 Inattivo – s03 Inattivo**
Ordini effettuati	**S01200114595535**

La presenza del client non deve interferire nel protocollo di attivazione del nuovo server.

Locale : l'analisi dello stato conferma le ipotesi
Rete : l'analisi dello stato conferma le ipotesi.

- **Interazione fra client e catena con due server**

Riavviamo l'intero sistema e lo riportiamo nelle condizioni del punto precedente però senza ordini registrati per rendere più leggibili i risultati.
Disconnettiamo il primo client e riconnettiamolo quindi attiviamo, disconnettiamo e riattiviamo un secondo client. Questo dovrebbe portare il secondo client a ricevere come referente il server s02, in virtù della politica di ridistribuzione del carico adottata.
Concludendo per entrambi inoltriamo un ordine di 10 sv02 .
Lo stato atteso è :

Server s01

Server attivi	**S01 – s02**
Clock	**2**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 180 - sv01: 100**
Ordini ricevuti	**s01200114691356 - s02200114691357**

Server s02

Server attivi	**s01 – s02**
Clock	**2**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 180 - sv01: 100**
Ordini ricevuti	**s01200114691356 - s02200114691357**

Client 1

Clock	**2**
Server	**s01 Attivo - s02 Attivo – s03 Inattivo**
Ordini effettuati	**s01200114691356**

Client 2

Clock	**2**
Server	**s02 Attivo - s01 Attivo – s03 Inattivo**
Ordini effettuati	**s02200114691357**

Locale : l'analisi dello stato conferma le ipotesi

Rete : l'analisi dello stato conferma le ipotesi.

- **Crash del secondo server in una catena di due server senza messaggi in circolo**

Esaurite le comunicazioni mandiamo in crash il secondo server ottenendo un errore sulla connessione da parte del secondo client.
Tentiamo una riconnessione del secondo client che modificherà il suo stato e si connetterà al unico server sopravvissuto dopo aver tentato di connettersi al secondo server.
Per terminare inviamo un ordine di 10 sv01 dal secondo client.
Lo stato atteso è :

Server s01

Server attivi	**S01**
Clock	**3**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 180 - sv01: 90**
Ordini ricevuti	**s01200114691356 - s02200114691357 - s0120011469222**

Client 1

Clock	**2**
Server	**s01 Attivo - s02 Attivo – s03 Inattivo**
Ordini effettuati	**s01200114691356**

Client 2

Clock	**3**
Server	**s02 Attivo - s01 Attivo – s03 Inattivo**
Ordini effettuati	**s02200114691357 - s0120011469222**

Locale : l'analisi dello stato conferma le ipotesi
Rete : l'analisi dello stato conferma le ipotesi.

- **Crash del primo server in una catena di due server con messaggi in circolo**

Riattiviamo il secondo server e connettiamo, disconnettiamo e riconnettiamo prima il secondo client e poi il primo, in modo da riportare il sistema in una condizione analoga a quella del punto precedente ma con i client che si scambiano i server a cui accedono(cli1 a s02 e cli2 a s01).
Quindi propaghiamo una richiesta di cancellazione del primo ordine effettuato dal client 2 e contemporaneamente mandiamo in crash il server 2.
La richiesta di cancellazione dell'ordine sarà rigettata e lo stato del sistema sarà aggiornato come segue :

Server s01

Server attivi	**s01**
Clock	**5**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 180 - sv01: 90**
Ordini ricevuti	**s01200114691356 - s02200114691357 - s0120011469222**

Client 1

Clock	**4**
Server	**s02 Attivo - s01 Attivo – s03 Inattivo**
Ordini effettuati	**s01200114691356**

Client 2

Clock	**4**
Server	**s01 Attivo - s02 Attivo – s03 Inattivo**

Ordini effettuati	**s02200114691357 - s0120011469222**

Il client 2 riceve una segnalazione che l'ordine non è stato cancellato.

Locale : Non risulta possibile innestare il crash prima che la procedura di rimozione dell'ordine abbia terminato per cui l'operazione di cancellazione và a buon fine e solo dopo il server 2 va in crash. Lo stato che si ottiene è :

Server s01

Server attivi	**S01**
Clock	**5**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 190 - sv01: 90**
Ordini ricevuti	**s01200114691356 - s0120011469222**

Client 1

Clock	**4**
Server	**s02 Attivo - s01 Attivo – s03 Inattivo**
Ordini effettuati	**s01200114691356**

Client 2

Clock	**4**
Server	**s01 Attivo - s02 Attivo – s03 Inattivo**
Ordini effettuati	**s0120011469222**

Introducendo un ritardo di due secondi tra la ricezione del messaggio e la sua propagazione al server successivo, il comportamento locale risulta corrispondente a quello atteso.

Rete : l'analisi dello stato conferma le ipotesi.

- **Attivazione di un terzo server senza client connessi**

Azzeriamo lo stato del sistema terminando tutti i processi e ripartiamo attivando i tre server s01, s02 e s03.

Lo stato atteso è il seguente:

Server s01

Server Attivi	**S01 – s02 – s03**
Clock	**3**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

Server s02

Server Attivi	**s01 – s02 – s03**
Clock	**3**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

Server s03

Server Attivi	**s01 – s02 – s03**
Clock	**3**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

Locale : l'analisi dello stato conferma le ipotesi
Rete : l'analisi dello stato conferma le ipotesi.

- **Attivazione di un terzo server con client connessi**

Rispetto al punto precedente mandiamo in crash il terzo server quindi facciamo connettere il primo client e riattiviamo il terzo server.
Lo stato atteso è :

Server s01

Server Attivi	**s01 – s02 – s03**
Clock	**5**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

Server s02

Server Attivi	**s01 – s02 – s03**
Clock	**5**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

Server s03

Server Attivi	**s01 – s02 – s03**
Clock	**5**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 – mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

Client 1

Clock	**4**
Server	**s01 Attivo - s02 Attivo – s03 Inattivo**
Ordini effettuati	

Ovviamente il client non si avvede dell'aggiornamento dello stato che conoscerà solo al prossimo accesso.

Locale : l'analisi dello stato conferma le ipotesi
Rete : l'analisi dello stato conferma le ipotesi.

- **Interazione fra client e catena con tre server e di più client con catena di tre server**

Attiviamo un secondo client ed effettuiamo l'ordine di 10 sv01 10 cpu01 e 10 mb01 per il primo client e 1 sv02 1 cpu02 e 1 mb02 per il secondo entrambi saranno connessi al primo server in virtù dello stato iniziale noto a default.

NOTA : *Il comportamento erroneo dei client, che non memorizzano lo stato prima di terminare una connessione, è voluto.*
In questo modo è possibile verificare l'impatto di un nuovo cliente sul sistema .
La versione reale della applicazione (quella ipoteticamente utilizzata da una azienda) conterrà ovviamente la funzione di salvataggio su disco delle impostazioni prima della chiusura.
Così da rendere estremamente raro il bisogno di aggiornare la configurazione.

Lo stato atteso è :

Server s01

Server Attivi	**s01 – s02 – s03**
Clock	**5**
Prodotti disponibili	**cpu02: 599 - cpu01: 490 - mb02: 399 - sv02: 199 - mb01: 290 - sv01: 90**
Ordini ricevuti	**s01200114610177 - s012001146101739**

Server s02

Server Attivi	**s01 – s02 – s03**
Clock	**5**

Prodotti disponibili	**cpu02: 599 - cpu01: 490 - mb02: 399 - sv02: 199 - mb01: 290 - sv01: 90**
Ordini ricevuti	**s01200114610177 - s012001146101739**

Server s03

Server Attivi	**s01 – s02 – s03**
Clock	**5**
Prodotti disponibili	**cpu02: 599 - cpu01: 490 - mb02: 399 - sv02: 199 - mb01: 290 - sv01: 90**
Ordini ricevuti	**s01200114610177 - s012001146101739**

Client 1

Clock	**4**
Server	**s01 Attivo - s02 Attivo – s03 Inattivo**
Ordini effettuati	**s01200114610177**

Client 2

Clock	**5**
Server	**s01 Attivo - s02 Attivo – s03 Attivo**
Ordini effettuati	**s012001146101739**

Locale : l'analisi dello stato conferma le ipotesi
Rete : l'analisi dello stato conferma le ipotesi.

- **Crash del primo server in una catena di tre server senza messaggi in circolo**

Manteniamo attivi entrambi i client e mandiamo in crash il primo server.
I client cadono e ad una successiva connessione (prima cli2 poi cli1) vanno a collegarsi al s02 dopo aver tentato una connessione al primo. Entrambi aggiornano il proprio stato.

Server s02

Server Attivi	**s02 – s03**
Clock	**6**
Prodotti disponibili	**cpu02: 599 - cpu01: 490 - mb02: 399 - sv02: 199 - mb01: 290 - sv01: 90**
Ordini ricevuti	**s01200114610177 - s012001146101739**

Server s03

Server Attivi	**s02 – s03**
Clock	**6**
Prodotti disponibili	**cpu02: 599 - cpu01: 490 - mb02: 399 - sv02: 199 - mb01: 290 - sv01: 90**
Ordini ricevuti	**s01200114610177 - s012001146101739**

Client 1

Clock	**6**
Server	**s03 Attivo – s02 Inattivo - s01 Inattivo**
Ordini effettuati	**s01200114610177**

Client 2

Clock	**6**
Server	**s02 Attivo – s03 Attivo - s01 Inattivo**
Ordini effettuati	**s012001146101739**

Locale : l'analisi dello stato conferma le ipotesi
Rete : l'analisi dello stato conferma le ipotesi.

- **Crash del primo server in una catena di tre server con messaggi in circolo**

Riavviamo l'intero sistema, connettendo i tre server, attiviamo sconnettiamo e riconnettiamo i client così da avere il primo su s01 ed il secondo su s02.

Facciamo inoltrare un ordine al secondo e contemporaneamente mandiamo in crash s01.
La richiesta viene persa e l'utente avvisato. Il sistema recupera il suo stato mentre il primo client perde la connessione.
Lo stato atteso è :

Server s02

Server Attivi	**s02 – s03**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

Server s03

Server Attivi	**s02 – s03**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

Client 2

Clock	**3**
Server	**s02 Attivo – s03 Attivo - s01 Attivo**
Ordini effettuati	**s02200114611542 Rifiutato a causa di un errore sulla linea**

Locale : Non risulta possibile innestare il crash prima che l'ordine sia stato propagato correttamente per cui l'operazione và a buon fine e solo dopo il server 1 va in crash. Lo stato che si ottiene è :

Server s02

Server Attivi	**s02 – s03**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**s02200114611542**

Server s03

Server Attivi	**s02 – s03**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**s02200114611542**

Client 2

Clock	**3**
Server	**s02 Attivo – s03 Attivo - s01 Attivo**
Ordini effettuati	**s02200114611542**

Introducendo un ritardo di due secondi tra la ricezione del messaggio e la sua propagazione al server successivo, il comportamento locale risulta corrispondente a quello atteso per l'utente mentre per s02 si verifica un problema di inconsistenza dovuto al meccanismo di ritardo (sleep del thread previous) che blocca la corretta chiusura del thread innescata dal recovery.

Rete : Il comportamento ottenuto è analogo al caso locale senza ritardo.

- **Crash del secondo server in una catena di tre server senza messaggi in circolo**

Riavviamo l'intero sistema, connettendo i tre server, attiviamo sconnettiamo e riconnettiamo i client così da avere il primo su s01 ed il secondo su s02.
Terminiamo il s02, il primo client non si accorgerà di nulla mentre il secondo perderà la connessione.
Lo stato atteso è :

Server s01

Server Attivi	**s01 – s03**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - sv02: 200 - mb01: 300 - sv01: 100**
Ordini ricevuti	**Nessuno**

Server s03

Server Attivi	**s01 – s03**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - sv02: 200 - mb01: 300 - sv01: 100**
Ordini ricevuti	**Nessuno**

Client 1

Clock	**3**
Server	**s01 Attivo – s02 Attivo - s03 Attivo**
Ordini effettuati	

Locale : l'analisi dello stato conferma le ipotesi .
Rete : l'analisi dello stato conferma le ipotesi

- **Crash del secondo server in una catena di tre server con messaggi in circolo**

Riavviamo l'intero sistema, connettendo i tre server, attiviamo sconnettiamo e riconnettiamo i client così da avere il primo su s01 ed il secondo su s02.
Facciamo inoltrare un ordine al primo e subito dopo mandiamo in crash s02 supponendo che l'ordine del client1 l'abbia già passato.
La richiesta viene completata. Il sistema recupera il suo stato mentre il secondo client perde la connessione.
Lo stato atteso è :

Server s02

Server Attivi	**s01 – s03**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**s012001146113551**

Server s03

Server Attivi	**s01 – s03**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**s012001146113551**

Client 1

Clock	**3**
Server	**s01 Attivo – s02 Attivo - s03 Attivo**
Ordini effettuati	**s012001146113551**

Locale : l'analisi dello stato conferma le ipotesi si noti però che anche in questo caso l'ordine ha già terminato il suo cammino prima che il server s02 vada in crash.
Con il ritardo risulta impossibile effettuare verifiche di interesse.
Rete : Il comportamento ottenuto è analogo al caso locale senza ritardo.

- **Crash del terzo server in una catena di tre server senza messaggi in circolo**

Riavviamo l'intero sistema, connettendo i tre server, attiviamo sconnettiamo e riconnettiamo i client così da avere il primo su s01 ed il secondo su s02.
Terminiamo il s03, i client non si accorgeranno di nulla.

Lo stato atteso è :

Server s01

Server Attivi	**s01 – s02**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - sv02: 200 - mb01: 300 - sv01: 100**
Ordini ricevuti	**Nessuno**

Server s03

Server Attivi	**s01 – s02**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - sv02: 200 - mb01: 300 - sv01: 100**
Ordini ricevuti	**Nessuno**

Client 1

Clock	**3**
Server	**s01 Attivo – s02 Attivo - s03 Attivo**
Ordini effettuati	

Client 2

Clock	**3**
Server	**s01 Attivo – s02 Attivo - s03 Attivo**
Ordini effettuati	

Locale : l'analisi dello stato conferma le ipotesi.
Rete : l'analisi dello stato conferma le ipotesi.

- **Crash del terzo server in una catena di tre server con messaggi in circolo**

Riavviamo l'intero sistema, connettendo i tre server, attiviamo sconnettiamo e riconnettiamo i client così da avere il primo su s01 ed il secondo su s02.
Facciamo inoltrare un ordine al primo e subito dopo mandiamo in crash s03; se supponessimo che l'ordine fosse già passato questo completerebbe correttamente.
Assumiamo invece che non abbia ancora passato s03.
La richiesta viene abortita per effetto del protocollo di recovery ed al termine il client 1 viene avvisato del fallimento.
Il secondo client non si accorge di nulla
Lo stato atteso è :

Server s01

Server Attivi	**s01 – s02**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**s012001146113635 abortito**

Server s02

Server Attivi	**s01 – s02**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**s012001146113635 abortito**

Client 1

Clock	**3**
Server	**s01 Attivo – s02 Attivo - s03 Attivo**
Ordini effettuati	**s012001146113635 abortito**

Client 2

Clock	**3**
Server	**s01 Attivo – s02 Attivo - s03 Attivo**

Ordini effettuati	**Nessuno**

Locale : Non risulta possibile innestare il crash prima che l'ordine sia stato propagato correttamente per cui l'operazione và a buon fine e solo dopo il server 3 va in crash. Lo stato che si ottiene è :

Server s01

Server Attivi	**s01 – s02**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**s012001146113551**

Server s02

Server Attivi	**s01 – s02**
Clock	**4**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**s012001146113551**

Client 1

Clock	**3**
Server	**s01 Attivo – s02 Attivo - s03 Attivo**
Ordini effettuati	**Nessuno**

Client 2

Clock	**3**
Server	**s02 Attivo – s03 Attivo - s01 Attivo**
Ordini effettuati	**s012001146113551**

Introducendo un ritardo di due secondi tra la ricezione del messaggio e la sua propagazione al server successivo, il comportamento locale risulta corrispondente a quello atteso per l'utente mentre per s03 si verifica un problema di inconsistenza dovuto al meccanismo di ritardo (sleep del thread previous) che blocca la corretta chiusura del thread innescata dal recovery.

Rete : Il comportamento ottenuto è analogo al caso locale senza ritardo.

- **Crash di due server in una catena di tre server senza messaggi in circolo**

Riavviamo l'intero sistema, connettendo i tre server, quindi mandiamo in crash prima s02 e poi s03.
Il recovery riporta prima il sistema al caso con 2 server e poi al caso con un solo server attivo.
Lo stato atteso è :

Server s03

Server Attivi	**s03**
Clock	**5**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**Nessuno**

L'interfaccia del server lo segnala come l'unico attivo.

Locale : l'analisi dello stato conferma le ipotesi .
Rete : l'analisi dello stato conferma le ipotesi.

- **Crash di due server in una catena di tre server con messaggi in circolo**

Riavviamo l'intero sistema, connettendo i tre server, attiviamo sconnettiamo e riconnettiamo i client così da avere il primo su s01 ed il secondo su s02.
Facciamo inoltrare un ordine al primo e subito dopo mandiamo in crash s02 e s03 contemporaneamente e supponiamo che l'ordine abbia superato s02 ma non s03.

La richiesta di ordine viene abortita in quanto non riesce a ritornare a s01 ed il protocollo di recovery ne provoca l'aborto quando s01 vede il crash di s03.
Lo stato atteso è :

Server s01

Server Attivi	**s01**
Clock	**5**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 100**
Ordini ricevuti	**s01200114614519 abortito**

Client 1

Clock	**3**
Server	**s01 Attivo – s02 Attivo - s03 Attivo**
Ordini effettuati	**s01200114614519 abortito**

Locale : Non risulta possibile innestare il crash prima che l'ordine sia stato propagato correttamente per cui l'operazione và a buon fine e solo dopo i server vanno in crash, uno alla volta. Lo stato che si ottiene è :

Server s01

Server Attivi	**s01**
Clock	**5**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 99**
Ordini ricevuti	**s01200114614519**

Client 1

Clock	**3**
Server	**s01 Attivo – s02 Attivo - s03 Attivo**
Ordini effettuati	**s01200114614519**

Con un ritardo di due secondi tra la ricezione del messaggio e la sua propagazione al server successivo si verifica un problema di inconsistenza dovuto al meccanismo di ritardo (sleep del thread previous).

Rete : Il comportamento ottenuto è analogo al caso locale senza ritardo.

- **Conflitto sulla disponibilità dei prodotti**

Assumiamo attivi e nelle condizioni iniziali due server e due client, il primo connesso a s01 ed il secondo a s02.
Entrambi i server non si hanno vittorie nel ultimo conflitto disputato ed i due client inoltrano contemporaneamente un ordine di 51 sv01.
Il peso del ordine di cli1 sarà 101 mentre quello di cli2 sarà 102 quindi il conflitto sarà vinto da cli2.
Lo stato atteso sarà :

Server s01

Server Attivi	**s01 – s02**
Clock	**2**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 49**
Ordini ricevuti	**s02200114614585 - s012001146145938 abortito causa esaurimento scorte**

Server s02

Server Attivi	**s01 – s02**
Clock	**2**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 49**
Ordini ricevuti	**s02200114614585 - s012001146145938 abortito causa esaurimento scorte**

Client 1

Clock	**2**

Server	**s01 Attivo – s02 Attivo - s03 Inattivo**
Ordini effettuati	**s012001146145938 abortito causa esaurimento scorte**

Client 2

Clock	**2**
Server	**s02 Attivo – s01 Attivo - s03 Inattivo**
Ordini effettuati	**s02200114614585**

Locale : Non risulta possibile verificare eventuali collisioni in quanto il primo ordine si propaga prima che sia possibile attivare il secondo. Quindi viene registrato l'ordine del cli1 e non quello del cli2. Lo stato che si ottiene è :

Server s01

Server Attivi	**s01 – s02**
Clock	**2**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 49**
Ordini ricevuti	**s012001146145938 - s02200114614585 abortito causa esaurimento scorte**

Server s02

Server Attivi	**s01 – s02**
Clock	**2**
Prodotti disponibili	**cpu02: 600 - cpu01: 500 - mb02: 400 - mb01: 300 - sv02: 200 - sv01: 49**
Ordini ricevuti	**s012001146145938 - s02200114614585 abortito causa esaurimento scorte**

Client 1

Clock	**2**
Server	**s01 Attivo – s02 Attivo - s03 Inattivo**
Ordini effettuati	**s012001146145938**

Client 2

Clock	**2**
Server	**s02 Attivo – s01 Attivo - s03 Inattivo**
Ordini effettuati	**s02200114614585 abortito causa esaurimento scorte**

Introducendo un ritardo di due secondi tra la ricezione del messaggio e la sua propagazione al server successivo, il comportamento locale risulta corrispondente a quello atteso.

Rete : Il comportamento ottenuto è analogo al caso locale senza ritardo.