Nome e Cognome: Andrea Zorzi
Matricola: 561818

Programmazione Concorrente e Distribuita
Progetto Data Sharing “Musharilla”

Sommario

Di seguito è presentata la relazione del progetto di Programmazione Concorrente e Distribuita riguardante la realizzazione di un sistema di sharing per la condivisione e lo scambio di un certo numero di risorse.

Indice

1 Avvio dell'applicazione (solo per piattaforma Linux) 3

2 Scelte progettuali ed organizzazione classi 3
2.1 Risorsa . . . 4
2.2 Client . . . 4
2.3 Server . . . 5
2.4 InterfacciaGrafica . . . 5
2.4.1 GUIServer . . . 6
2.4.2 GUIMusharilla . . . 6

3 Utilizzo interfaccia grafica 7
3.1 Lato GUIServer . . . 7
3.2 Lato GUIMusharilla . . . 7

4 Concorrenza 8

5 Robustezza 10

Nome e Cognome: Andrea Zorzi
Matricola: 561818

1 Avvio dell'applicazione (solo per piattaforma Linux)

Nella shell, posizionarsi nella cartella del sorgente (src) dove è presente il file *Makefile* ed avviare l'RMI Registry con il comando **rmiregistry &**.
Successivamente, dare il comando **make server**.
Verrano avviate e visualizzate tre finestre grafiche (oggetti GUIServer) che permettono di avviare i tre server. Come da specifica di progetto, si richiede di avviare prima i server quindi per ognuno di essi, si raccomanda di inserire il nome scelto nella relativa cella e cliccare sul bottone avvia.
Successivamente dare il comando **make client**.
Verrano avviate e visualizzate tre finestre grafiche (oggetti GUIMusharilla) che permettono di avviare tre client.
Ora si può iniziare ad utilizzare l'applicativo.
È disponibile il comando **make clean** per la pulizia del codice oggetto.
Nella cartella "src" si trovano i sorgenti del codice.

2 Scelte progettuali ed organizzazione classi

Il programma è stato separato in cinque package:

risorsa : contiene la definizione e l'implementazione dell'oggetto risorsa.
Sorgenti: Risorsa.java

client : contiene l'interfaccia remota del client e la sua implementazione.
Sorgenti: Client.java, ClientImpl.java

server : contiene l'interfaccia remota del server e la sua implementazione.
Sorgenti: Server.java, ServerImpl.java

interfacciaGrafica : contiene le due classi che realizzano la grafica del server e del client.
Sorgenti: GUIServer.java, GUIMusharilla.java

main : contiene i due main che costruiscono l'istanza di GUIServer e l'istanza di GUIMusharilla.
Sorgenti: MainServer.java, MainClient.java

Le scelte implementative più importanti sono state quelle di avviare tutti i server con un proprio nome, e ciò è fatto prima dell'avvio dei client, anche questi identificati da un nome inserito nella schermata iniziale.
Inoltre, è stato evitato l'utilizzo di bottoni di refresh e thread che facevano polling, relativamente all'aggiornamento delle liste dei server e dei client presenti in GUIServer, utilizzando il pattern observer. Ovvero l'oggetto GUIServer (Observer) contiene come campo dati il riferimento all' oggetto ServerImpl (Subject), che dualmente, contiene come campo dati il rifemento all'oggetto GUIServer. Pertanto, questo collegamento biunivoco, permette l'aggiornamento automatico delle liste di ogni server quando si registra/chiude un server remoto e quando si connette/disconnette un client. Vediamo meglio ognuna di questa due operazioni:

- **Aggiornamento lista server remoti** : quando si avvia un server dalla GUI, dopo la creazione dell'oggetto ServerImpl, viene invocato il metodo di ServerImpl aggiorna() che carica nella relativa lista grafica del server avviato la lista dei nomi dei server registrati in RMI; ed inoltre aggiorna le relative liste grafiche delle GUI dei server remoti.
  Il metodo che effettua quest'ultimo aggiornamento è aggiornaListaServerGUIServerRemota, che è implementato in ServerImpl e contiene la singola istruzione **serverGui.aggiornaListaServer (listaServerRMI)** . Il metodo aggiornaListaServer descritto in GUIServer effettua l'aggiornamento della lista grafica e dei riferimenti del server ai server che si sono registrati dal registro RMI.
  Anche la chiusura del server, funzionalità aggiuntiva implementata, opera con la stessa

logica ed è eseguita dal metodo di ServerImpl chiudiServer() invocato in GUIServer alla chiusura della finestra.

- **Aggiornamento lista client remoti** : Quando un client si connette/disconnette dal server, all'interno del metodo connetti di ClientImpl, descritto in seguito, viene eseguita l'istruzione **server.aggiornaListaClientGUIServerRemota([true se si connette, false se si disconnette])**. Il metodo remoto aggiornaListaClientGUIServerRemota implementato in ServerImpl contiene la singola istruzione **serverGui.aggiornaListaClient(conn)** (conn di tipo booleano). Il metodo aggiornaListaClient descritto in GUIServer effettua l'aggiornamento della lista grafica dei client connessi al server.

Da come si noterà nell'utilizzo, ho scelto di implementare la notifica di connessione di un client nel server e la notifica della connessione al server nel client.
Per quanto riguarda, l'aggiornamento della lista risorse locali che possiede un client è gestito da un thread (VisualizzaAggiornaRisorse, classe interna a GUIMusharilla) che viene risvegliato ogni qualvolta viene aggiunta o scaricata una risorsa. In questa situazione nella quale viene risvegliato, il thread effettua l'aggiornamento della lista grafica contenente le risorse, aggiorna il numero di risorse ed impostando un booleano a vero si sospende andando in wait.

2.1 Risorsa

La classe Risorsa estende *Serializable* perchè per l'operazione di scaricamento di una risorsa, questa è un oggetto posseduto da un riferimento remoto e quindi non avrebbe senso spedire il suo riferimento. Pertanto l'oggetto risorsa viene serializzato e spedito al chiamante. Tra i campi che identificano una risorsa, sono presenti un contatore di scaricamenti e un vector di String che contiene i nomi dei client che l'hanno scaricata.

2.2 Client

Il package client contiene in Client.java l'interfaccia pubblica Client che estende *Remote*, la quale definisce i metodi relativi al client invocabili remotamente. E in ClientImpl.java, la classe ClientImpl che estende UnicastRemoteObject per supportare la comunicazione point-to-point usando TCP ed implementa Client. ClientImpl contiene come campi dati un vector di Risorsa **listaRisorse** e un riferimento remoto **server** di tipo Server che identifica il server alla quale il client è connesso. Tra i metodi invocabili localmente i principali sono:

- **public void aggiungiRisorsaInLocale(String r, Integer dim,String c)** : aggiunge in coda a **listaRisorse** una nuova risorsa con i parametri passati dall'interfaccia grafica, rispettivamente il nome, la dimensione e il contenuto.
- **public void aggiungiRisorsaScaricata(Risorsa r)** : aggiunge in coda a **listaRisorse** la risorsa scaricata che è passata come parametro.

Tra i metodi invocabili remotamente i principali sono:

- **public Risorsa getRisorsaRemota(String id) throws RemoteException** : ritorna la copia serializzata della risorsa remota. Viene passato come parametro la stringa che identifica il nome della risorsa da scaricare.
- **public String connetti(Object obj) throws RemoteException, MalformedURLException, NotBoundException** : riceve come parametro il nome del server che il client ha selezionato per connettersi ed effettua il lookup.
  Il riferimento remoto ritornato dal lookup di tipo Server viene assegnato al campo dati **server** e vengono eseguite nell'ordine le seguenti operazioni:
  - **server.registraClient(this)** che aggiunge il client alla lista dei client del server.
  - viene invocato il metodo notificaRegistrazione che notifica al client la registrazione sul server.

  - **server.aggiornaListaClientGUIServerRemota(true)** che ha la funzione di segnalare all'interfaccia grafica del server di aggiornare la lista dei client connessi al server.

- **public void disconnetti() throws RemoteException** : Controlla che il campo dati **server** non sia nullo, ovvero il client sia connesso ad un server, e vengono eseguite nell'ordine le seguenti operazioni:
  - **server.rimuoviClient(this)** che rimuove il client dalla lista dei client del server.
  - **server.aggiornaListaClientGUIServerRemota(false)**che ha la funzione di segnalare all'interfaccia grafica del server di aggiornare la lista dei client connessi al server.
  - **server=null** tolgo il riferimento.
- **public Vector <Client>cercaRisorsa(String id) throws RemoteException** : invoca solamente il metodo remoto del server con l'istruzione **server.ricerca(id,this.nomeClient)**. id identifica il nome della risorsa da cercare e this.nomeclient il nome del client che vuole fare la ricerca e che quindi non deve essere considerato. La chiamata ritorna un vector di riferimenti remoti ai client che possiedono la risorsa.
- **public boolean scaricaRisorsa(Vector <Client>possessoriRemRis, String nomeRisSelez, String nomeCliRemSelez) throws RemoteException** : scarica la risorsa del client selezionato tra i client che la possiedono. Se il client selezionato per lo scaricamento si disconnette, verrà lanciata una RemoteException e la risorsa sarà scaricata da uno degli altri client che la possiedono.
  Si nota che lo scaricamento della risorsa è implementato nel client perché a seguito della ricerca fatta nel server, il client è in possesso del vector di riferimenti remoti ai client che possiedono la risorsa cercata. Infatti nel caso il server cada dopo la ricerca, sarà lo stesso possibile scaricare le risorsa dai client che la possiedono (come una rete KAD). Mentre poter fare ricerche sarà necessario connettersi ad un altro server.

2.3 Server

Il package server contiene in Server.java l'interfaccia pubblica SERVER che estende *Remote*, la quale definisce i metodi relativi al server invocabili remotamente. E in ServerImpl.java, la classe SERVERIMPL che estende UnicastRemoteObject ed implementa SERVER. CLIENTIMPL contiene come campi dati un vector di SERVER **listaServer** dove ci sono i riferimenti ai server ad esso connessi, e un vector di CLIENT **listaClient** che contiene i riferimenti ai client ad esso connessi. È presente inoltre il campo dati **serverGUI** che è il riferimento all'interfaccia grafica GUISERVER.
Il costruttore di SERVERIMPL contiene l'istruzione di rebind che registra il server in RMI. I principali metodi implementati sono quelli definiti nell'interfaccia SERVER che realizzano la ricerca, l'aggiunta e la rimozione di un client. Sono descritti nella sezione Concorrenza perché sono operazioni che possono creare interferenza. **public String aggiorna() throws RemoteException, MalformedURLException, NotBoundException** e **public void chiudiServer() throws RemoteException, MalformedURLException, NotBoundException** descritti in precedenza, realizzano l'observer per la visualizzazione nella lista grafica dei nomi dei server remoti connessi.

2.4 InterfacciaGrafica

Il package interfacciaGrafica contiene le classi grafiche GUISERVER e GUIMUSHARILLA. Inoltre è presente la classe AGGIUNGICOMPONENTE di visibilità package che rende disponibile alle classi grafiche l'utilizzo del metodo statico **void aggiungi(Container container, Component c,int x, int y, int width, int height)** che consente di aggiungere un oggetto grafico generico (Component) in un contenitore generico (Container) con posizionamento assoluto.

2.4.1 GUIServer

GUIServer si compone di alcuni campi dati, tra cui i principali sono:

- **server** di tipo ServerImpl, che rappresenta l'oggetto "logico" server.
- **listaServerConnessi** di tipo JList che visualizza i nomi dei server a cui l'oggetto **server** è connesso.
- **listaClientConnessi** di tipo JList che visualizza i nomi dei client connessi a **server**.

GUIServer contiene delle classi interne relative ad oggetti cliccabili o a oggetti lista che implementano dei specifici listener:

- AvviaServer implementa ActionListener e gestisce il click del pulsante che avvia il server.
  Viene fatto il controllo sul nome inserito per il server dal metodo **controlloEsisteNomeUguale(String nomeDaControllare)**. Se passa, viene costruito il campo dati **server** ed è eseguita l'istruzione **server.aggiorna()**.
- VisualizzaInfoClient implementa ListSelectionListener e deve gestire l'operazione che alla selezione di un client della lista vengano stampate nell'area di testo sottostante i nomi delle risorse da lui possedute.
  Principalmente, effettua prima dei controlli sulla lista come la selezione (**evento.getValueIsAdjusting() == false**) e l'indice selezionato della lista (**listaClientConnessi.getSelectedIndex() != -1**). Passati i controlli, si ottiene il riferimento remoto del client selezionato e si scorre la sua lista risorse stampando i nomi delle risorse nell'area di testo.

Inoltre altri metodi come:

- **private int setWindowActions()** gestisce la chiusura delle finestra e ritorna l'intero che è passato dentro al metodo setDefaultCloseOperation(int) che chiude l'oggetto grafico al click d'uscita.
  Aggiunge un listener che implementa la classe anonima WindowAdapter. Al suo interno vengono specificati due modi di chiusura del server: quella corretta, che con l'istruzione **server.chiudiServer()** toglie i riferimenti del server che si sta chiudendo dalle liste dei server remoti ed effettua l'unbind. L'altro modo non effettua la procedura descritta sopra.
- **public void aggiornaListaServer(String[] listaServerRMI) throws MalformedURLException, RemoteException, NotBoundException** Riceve come parametro l'array di String che contiene gli indirizzi dei server in RMI. Si scorre tale array: è usata la funzione split con lo scopo di scartare l'indirizzo di se stesso. Con ogni indirizzo dei server remoti, invece, si ottiene il suo riferimento (tipo Server), si controlla ed eventualmente si connette il server relativo alla lista grafica che stiamo aggiornando al server remoto di cui abbiamo il riferimento, ed infine si aggiunge il nome del server remoto nella lista grafica **listaServerConnessi**.
- **public void aggiornaListaClient(boolean) throws RemoteException** gestisce l'aggiornamento del campo dati **listaClientConnessi**, semplicemente scorrendo la lista client dell'oggetto ServerImpl campo dati **server**.

2.4.2 GUIMusharilla

GUIServer si compone di alcuni campi dati, tra cui:

- **client** di tipo ClientImpl, che rappresenta l'oggetto "logico" client.
- **var** di tipo VisualizzaAggiornaRisorse, classe che estende thread. Nella run dentro il ciclo infinito il thread si sincronizza su se stesso ed effettuato un controllo while sul booleano sospendi. Se vero, va in wait, altrimenti effettua l'aggiornamento della lista grafica delle risorse e reimposta sospendi a vero e come detto prima si sospende. Il thread è risvegliato,esternamente, solo al termine dello scaricamento di una risorsa remota o di una aggiunta di una risorsa locale nella lista; avviene mediante sincronizzazione sull'oggetto **var** nel quale si imposta sospendi a falso e viene fatto **var.notify()**.

Vi sono molti altri campi dati ma trattasi di oggetti grafici come liste, bottoni, tabelle ed oggetti di supporto alla grafica come vettori di stringhe. GUIMusharilla contiene delle classi interne relative ad oggetti cliccabili o a oggetti lista che implementano dei specifici listener: nei pulsanti connetti, disconnetti, cerca, vengono semplicemente richiamati i metodi propri dell'oggetto logico **client** che effettuano le specifiche operazioni e i ritornano gli opportuni valori, Stringa o array di stringhe che sono poi gestiti negli oggetti grafici. Solo la classe AvviaScaricamento, listener del pulsante scarica, ha un discreto numero di statement poiché contiene le istruzioni che prelevano i dati della risorsa visualizzati nella tabella della ricerca e che l'utente ha selezionato per lo scaricamento. Inoltre è stato implementato un timer di scarimento della risorsa la cui durata varia a seconda della dimensione della risorsa (dimensione=1 : 1 secondo).

3 Utilizzo interfaccia grafica

3.1 Lato GUIServer

All'avvio dell'interfaccia grafica viene visualizzata un cella dove inserire il nome del server, ed un bottone "Avvia" che avvia il server. Cliccato su "Avvia", sono visualizzate nella finestra due liste grafiche: quella di sinistra è la lista dei client connessi al server, mentre quella di destra è la lista dei server remoti connessi al server. Come da specifica, ogni server deve essere connesso con tutti gli altri server remoti. Sotto la liste è presente un'area di testo dove vengono visualizzate varie notifiche:

- ci sono server remoti registrati (connessi al server)
- si sono registrati server remoti (si sono connessi al server)
- nessun client connesso
- nessun server registrato (connesso al server)
- Notifica del client: "nomeclient" connesso
- Lista risorse del client "nomeclient":
  "nomerisorsa1"
  ...

3.2 Lato GUIMusharilla

All'avvio dell'interfaccia grafica viene visualizzata un cella dove inserire il nome del client, ed un bottone "Start" che avvia l'applicativo Data Sharing per il client. Nella schermata che viene visualizzata, si notano tre tab: Connetti/Disconnetti, Cerca/Scarica risorse, Incoming risorse. Per default è attiva la scheda connetti/disconnetti. Qui per connettersi ad un server, basta selezionarne uno dalla lista e premere il pulsante connetti. Verrà visualizzata la notifica dal server nell'area di testo a fianco, se la connessione è avvenuta.
La scheda Cerca/Scarica risorse, si compone di una tabella inizialmente vuota dove verranno visualizzati i risultati sulla risorsa cercata, inserendo il nome nella cella e premendo il pulsante cerca risorsa. Per scaricarla basta selezionare tra i risultati la risorsa del client dal quale si vuole scaricarla, e premere scarica. Verrà attivato un timer di scaricamento e alla fine, se non è avvenuta disconnessione da parte del client da cui si stava scaricando, verrà visualizzato un messaggio di successo.
Per vedere le proprie risorse possedute e scaricate cliccare su Incoming risorse. Qui è possibile aggiungerne di nuove inserendo i dati che son proposti da inserire. Per visualizzare quante volte è stata scaricata la risorsa e da chi, selezionate una risorse e cliccare su dettagli risorsa.

4 Concorrenza

La gestione della concorrenza e delle possibili interferenze si trova solo su ServerImpl, per il motivo che sul client non aveva senso.
Le risorse sono inserite in un vector, ciò significa che se si scarica da un client una risorsa che viene inserita in coda (metodo add(risorsa)), un altro client può tranquillamente scaricare una delle altre risorse dello stesso vector senza interferenze. Questo perché la risorsa che il client richiede è stata ricercata prima della modifica al vector, e quindi si trova in una delle posizioni precedenti alla posizione alla quale sto inserendo.
Non è consentito al client scaricare risorse che ha già possiede (il discriminante è il nome della risorsa).

La classe ServerImpl ha la seguente gestione della concorrenza. l'oggetto critico è **listaClient**. Sono stati individuati i seguenti metodi, quelli che possono causare interferenza in esso.

- **public void registraClient(Client c) throws RemoteException** che permette la connessione/registrazione del client al server, inserendo il riferimento remoto del client in coda del **Vector<Client>listaClient**. Ha interferenza con:
  - i metodi ricerca(..) e rimuoviClient(..) il cui primo potrebbe vedersi modificare **listaClient** durante la ricerca e, il secondo potrebbe modificare **listaClient** con la sua esecuzione perché rimuove un client che può essere in una qualsiasi posizione.
    Ha interferenza anche con le chiamate degli altri client a questo metodo perché potrebbero modificare simultaneamente l'oggetto **listaClient** condiviso.

  Non ha interferenza con:
  - il metodo del client che scarica la risorsa, per lo stesso motivo descritto prima per l'aggiunta di una risorsa nella lista delle risorse, ovvero che l'inserimento di un client nella struttura è fatto in coda.
- **public void rimuoviClient(Client c) throws RemoteException** che permette la rimozione del client dal server, quando il client si disconnette. Ha interferenza con:
  - con i metodi di ricerca(..), registraClient(..) e le chiamate concorrenti di sè stesso dai vari client.

  Non ha interferenza con:
  - il metodo del client che scarica la risorsa. Questa è stata una mia scelta. Il motivo è il seguente: il metodo ricerca mi ritorna un vector di riferimenti di tipo Client, a client remoti che possiedono la risorsa cercata. Quando il client decide di scaricare la risorsa effettua lo scaricamento da uno dei riferimenti remoti di questa lista; quindi non si va a toccare **listaClient** da dove in questa situazione può essere rimosso senza problemi. Quindi se il client remoto si disconnette dal server, ciò non compromette lo scaricamento da parte del client che sta scaricando (ex. rete KAD). Lo scaricamento è interrotto se e solo se il client remoto chiude il suo applicativo, ovvero la sua GUI-Musharilla. Il comportamento adottato in questo caso, sarà spiegato nella sezione Robustezza.
- **public Vector<Client>ricerca(String id, String nomeClientInvocRic) throws RemoteException** che ricerca nel vector quali client hanno la risorsa che cerco. Ha interferenza con:
  - i metodi registraClient(..) e rimuoviClient(..), ma non con le chiamate concorrenti dai diversi client a questo metodo.
    Ha interferenza anche con il metodo del client che scarica la risorsa, ma in questo caso ho limitato la funzionalità che mentre il client sta scaricando, questi non può anche fare ricerca.
    Per quando riguarda gli altri client invece, questi possono ricercare mentre il client

sta scaricando perché come spiegato prima, nello scaricamento non si usa l'oggetto condiviso **listaClient** ed inoltre la risorsa scaricata viene aggiunta in coda alla lista delle risorse.

Per risolvere questi problemi ho deciso di utilizzare come lock il vector **listaClient**, e i metodi descritti prima devono acquisire il lock prima di procedere.
D'altro canto, tranne per il metodo rimuoviClient(..), il lock non deve essere mantenuto durante le operazioni, per permettere possibili chiamate concorrenti allo stesso metodo, o per permettere chiamate a metodi che non hanno interferenza con essi. Per descrivere meglio il blocco istruzioni per i tre metodi sono questi:

public void registraClient(Client c) throws RemoteException

```
synchronized(listaClient){
    connessione=true;
    while(ricerca==true){
        listaClient.wait();
    }
    listaClient.add(c);
    connessione=false;
    listaClient.notifyAll();
}
```

Ho deciso di sequenzializzare le connessioni al server e dato e solo un'unica istruzione (textbflistaClient.add(c)) l'ho inserita dentro il blocco synchronized. Si imposta subito il booleano connessione a vero, perché poi può capitare che essendoci ricerca il metodo debba aspettare che la ricerca termini. In questo caso si fa wait su listaClient, in modo tale che si rilascia il lock e così si possono fare più operazioni di questo tipo contemporanee e permetto di eseguire a chi con me non ha interferenza.
Dopo l'add del client si imposta a false connessione e si risvegliano i thread dormienti con notifyAll.

public void rimuoviClient(Client c) throws RemoteException

```
synchronized(listaClient){
    while(ricerca==true || connessione==true){
        listaClient.wait();
    }
    listaClient.remove(c);
}
```

Deve aspettare sia la ricerca che la registrazione del client (vedi condizioni d'attesa, che mandano in wait). Non imposta alcun proprio booleano e non risveglia nessuno al termine delle sue operazioni perché nessuno (ricerca(..), registraClient(..)) è andato in wait mentre rimuoviClient manteneva il lock.

public Vector<Client>ricerca(String id, String nomeClientInvocRic) throws RemoteException

```
synchronized (listaClient) {
    ricerca=true;
    while(connessione==true){
        listaClient.wait();
    }
}
//OPERAZIONI METODO RICERCA
synchronized (listaClient) {
    ricerca=false;
    listaClient.notifyAll();
}
```

Si imposta subito il booleano ricerca a vero, perché poi può capitare che essendoci connessione da parte di un client (registraClient(..))) il metodo debba aspettare che la connessione termini. In questo caso si fa wait su listaClient, in modo tale che si rilascia il lock e così si possono fare più operazioni di questo tipo contemporanee e permetto di eseguire a chi con me non ha interferenza.
Le istruzioni di ricerca sono fuori dal blocco sincronized per permettere le ricerche multiple.Finita la ricerca si riprende il lock su listaClient, impostando a false ricerca e risvegliando i thread dormienti con notifyAll.

5 Robustezza

Le notifiche di errori remoti vengono visualizzate prevalentemente sull'interfaccia grafica del client. Riguardo alla connessione, se non è possibile connettersi ad un server l'eccezione lanciata dal metodo connetti del client è gestita in GUIMusharilla con la visualizzazione di un messaggio. Lo stesso vale per la disconnessione nel relativo metodo del client disconnetti.
Riguardo alla ricerca se un client remoto si disconnette o viene chiuso un server, l'eccezione è gestita all'interno della ricerca con la visualizzazione nella shell dell'errore, l'oggetto remoto irraggiungibile viene ignorato e la ricerca continua senza problemi sino alla sua fine. Se cade invece il server a cui il client che fa la ricerca è connesso, la ricerca fallisce e viene visualizzato nell'interfaccia grafica il messaggio d'errore.
Riguardo allo scaricamento della risorsa, se un client che possiede la risorsa chiude l'applicativo (approfondimento spiegato prima) l'eccezione lanciata nel metodo scaricaRisorsa del client viene gestita, impostando in un vettore di boolean **vClientDisconnessi** il valore true alla posizione del client possessore della risorsa che si è disconnesso. Da ciò viene azzerato il contatore usato per iterare **Vector<Client>possessoriRemRis** (il vettore dei riferimenti remoti ai client possessori della risorsa). E quindi, se possibile, si scarica la risorsa dal primo client che si incontra nello scorrimento del **Vector<Client>possessoriRemRis**.