WORTH - Progetto Reti

Lorenzo Leonardini - matricola 598608

Introduzione

Data la natura complessa del progetto, per poterne affrontare una chiara analisi in questa relazione, si è scelto di procedere partendo da descrizioni generali della struttura del codice, per poi muoversi gradualmente nello specifico delle implementazioni. In particolare, dopo aver fornito le necessarie indicazioni per la compilazione ed esecuzione, si procederà con la struttura generale dei file sorgente e dell'implementazione, presentando brevemente la gestione dei Thread e della concorrenza. Dopodiché si affronterà la comunicazione tra client e server, introducendo una descrizione ad alto livello del protocollo applicativo; per poi analizzare alcune parti dell'implementazione. Seguirà un'analisi per le scelte di modalità di I/O e si concluderà con un'analisi dettagliata dell'implementazione della chat, piuttosto ampliata rispetto a quella richiesta nel testo del progetto. L'ultimo paragrafo racconta alcuni deadlock remoti che sono stati risolti.

Compilazione ed esecuzione

Il progetto è stato realizzato in Eclipse, i file nell'archivio consegnato sono il contenuto della cartella `src`. Client e Server sono entrambi nello stesso progetto. La versione di Java utilizzata è Java 8. Si può compilare ed eseguire il codice sia con un'IDE, sia da terminale con `make`.

Se si sceglie di importare il progetto in un'IDE, i file contenenti il metodo main sono `dev.leonardini.worth.client.ClientMain` e `dev.leonardini.worth.server.ServerMain`. Notare come il server ha bisogno della definizione di alcuni argomenti (su Eclipse *Run* → *Run Configurations...*).

Si consiglia tuttavia di compilare ed eseguire il progetto da terminale. Per la compilazione basta invocare il comando `make`, o con il target di default `make all`. Make si occupa di compilare i file class e di creare due jar eseguibili, `ClientWORTH.jar` e `ServerWORTH.jar`. Per l'esecuzione del client si può utilizzare il comando `java -jar ClientWORTH.jar` oppure, poiché non vi è bisogno di un terminale, si può eseguire direttamente da file manager. Il server invece necessita di essere eseguito con il comando `java -jar ServerWORTH.jar RMIhost`, specificando l'argomento `RMIHost`, che coincide con l'IP del server utilizzato dai client per la connessione. Il server utilizzerà questo valore per impostare la proprietà `java.rmi.server.hostname`. Per chiudere il server utilizzare il comando exit, altrimenti non vi è garanzia che progetti e utenti vengano salvati su file.
Maggiori discussioni su RMI e hostname sono presenti nel paragrafo sull'implementazione della chat.

Il progetto è stato testato con successo su Arch Linux e Windows 10.

Struttura generale

Come anticipato nel paragrafo precedente, server e client sono stati entrambi sviluppati nello stesso progetto Eclipse. La ragione è puramente una questione di comodità, poiché le due "entità" condividono alcuni oggetti (in particolare per esempio la definizione degli oggetti remoti), quest'approccio è più rapido di creare una libreria apposita. Ciò nonostante, il Makefile allegato

si occupa di rimuovere il codice specifico del client dal jar del server, e viceversa, in modo da alleggerire gli eseguibili finali.

Si è scelto di implementare il client con un'interfaccia grafica, utilizzando la libreria Java Swing. Il server, invece, offre input e output da linea di comando per semplificare il debugging e i test del software.

Il Client

La maggior parte del codice del client implementa l'interfaccia grafica. Questi file non verranno presi in esame, in quanto l'UI esula dagli scopi del laboratorio di reti, ma si è cercato di scrivere codice il più chiaro possibile, con opportuni commenti, in modo da semplificarne la lettura nel qual caso si fosse interessati ad alcune implementazioni.

Dopo aver effettuato il login da una finestra apposita, l'utente interagisce con il programma tramite una schermata principale, che, come Trello, implementa il metodo Kanban. In questa finestra si trovano la lista degli utenti, la chat, e la visualizzazione del progetto corrente (o della lista dei progetti disponibili). Alcune interazioni specifiche e minori sono invece gestite con degli appositi popup.

Si è scelto di implementare un'interfaccia il più intuitiva possibile, specialmente in rapporto alle interfacce di quelli che sono veri prodotti commerciali, come appunto Trello. Ci si sofferma in particolare sulla gestione delle card in un progetto: per avere maggiori informazioni riguardo una card, come la sua storia, basta cliccarci sopra; per spostare una card da una lista a un'altra, basta trascinarla nella colonna di destinazione[1]. Lo spostamento potrebbe apparire in ritardo, in quanto l'operazione viene prima confermata dal server.

Per "abbellire" l'interfaccia, si è scelto di aggiungere una funzionalità al progetto: la possibilità di impostare un'immagine profilo, visualizzabile nella lista utenti e in chat. L'immagine non viene caricata e gestita dal server, ma ogni utente ha la possibilità di associare una mail Gravatar, il cui hash verrà condiviso con gli altri client per poter scaricare l'immagine da Internet.

La ragione per cui è possibile sorvolare sull'implementazione della UI, è che il client è stato progettato in modo modulare: "backend" e "frontend" sono separati, e sarebbe possibile sostituire la GUI con una CLI, o addirittura permettere all'utente di scegliere tra le due, senza dover riadattare il codice attuale. Tutta l'interazione con il server avviene infatti mediante la classe `ClientAPI`, che implementa le funzioni richieste nel testo del progetto, fornendo al client un livello di astrazione: lo sviluppo della UI può procedere ignorando completamente l'idea di server, interfacciandosi con `ClientAPI` come se i dati fossero tutti disponibili localmente.

Sempre attraverso `ClientAPI`, l'interfaccia utente può registrare dei callback per la gestione di update come quelli riguardanti lo stato degli utenti online. Notare che non si tratta di RMI, come detto l'UI non ha alcuna relazione con il server, si tratta di callback interni che vengono richiamati quando `ClientAPI` riceve update tramite RMI.

`ClientAPI` è implementata con il pattern singleton, permettendo di accedere ai metodi in modalità "statica" e senza la necessità di passare un'istanza tra una classe e l'altra.

Raggruppare tutta l'interazione del server in un'unica classe singleton, permette anche di semplificare la gestione della concorrenza e l'accesso alle variabili condivise: è sufficiente infatti proteggere i metodi di `ClientAPI` con la keyword `synchronized`. Differentemente da quanto si potrebbe pensare, però, la gestione della concorrenza non è solo necessaria per via di RMI. Poiché il main thread è "occupato" dal rendering di Swing, infatti, per evitare che la UI si blocchi in attesa delle risposte del server, un Thread "usa e getta" viene creato ogni qual volta viene chiamata una funzione. Ad esempio:

[1] Come ho avuto modo di scoprire durante lo sviluppo, Swing non è la libreria più semplice con cui implementare il drag&drop. La maggiore conseguenza di ciò è che le card possono essere trascinate solamente sulle colonne, in uno spazio libero. Se si trascina una card su un'altra card, l'operazione fallisce.
Swing in generale non è una libreria molto "elastica": un altro piccolo "bug grafico" si verifica con l'invio di messaggi in chat: il calcolo della dimensione dei fumetti non è preciso, e la dimensione del pannello della chat è maggiore del necessario.

```
public void refresh() {
    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    new Thread(() -> {
        // ...
        projects.clear();
        List<String> prs = ClientAPI.get().listProjects();
        // ...
        updateUI();
    }).start();
}
```

Si potrebbe contestare il fatto che venga creato un nuovo Thread ogni qual volta si voglia interpellare il server, tuttavia in questo caso si è scelto di preferire la soluzione più semplice, rafforzata dal fatto che l'overhead di creazione e distruzione di un singolo thread sia irrilevante nel contesto di interazione con l'utente. L'alternativa di creare una situazione di thread produttore/consumatore, piuttosto che una thread pool, non avrebbe infatti portato a nessun tipo di vantaggio e avrebbe anzi solamente complicato l'implementazione.

Il Server

Anche il server è costituito da diversi moduli che operano assieme: `ServerTCP` si occupa di gestire le comunicazioni sulla connessione TCP, `ServerCLI` si occupa di gestire l'I/O con l'utente tramite linea di comando, `ServerChat` si occupa di gestire le comunicazioni via chat e il fallback via RMI (di questo si discuterà ampiamente nell'ultima sezione). Ognuno di questi tre moduli ha un Thread associato.

Vi sono poi tre singleton aggiuntivi e una classe astratta per la gestione di alcuni moduli aggiuntivi e/o database: `UserManager` per la gestione degli utenti, `ProjectDB` per la gestione dei progetti, `Logger` per la stampa e il salvataggio di alcune informazioni di debug, `RMIServer` per l'inizializzazione di RMI.

È indubbio che la gestione della concorrenza sia più complessa per il server che per il client. Per semplificarne una parte, la comunicazione TCP viene eseguita su un singolo thread, sfruttando il multiplexing di canali tramite NIO. In generale, anche in questo caso i dati vengono protetti tramite l'utilizzo di blocchi `synchronized`, con l'accortezza che, nel tentativo di migliorare le prestazioni, in questo caso non vengono bloccati oggetti interi (usando la keyword nella firma del metodo), ma solo i dati su cui si eseguono le operazioni, per il periodo di tempo più breve possibile.

Protocollo TCP

La connessione TCP con cui client e server comunicano è una connessione persistente, alla quale il server associa una sessione che identifica l'utente finché questi non si disconnette. Ogni singola comunicazione viene iniziata dal client, il server elabora la richiesta e trasmette la risposta al client, che riporta il risultato all'utente.

Descrizione

Nella colonna di sinistra si mostra il formato delle richieste, mentre nella colonna di destra si mostrano i due formati possibili per le risposte:

op	[data...]

op	false	"error message"

op	true	[data...]

`op` è un codice che identifica l'operazione richiesta, definito come `enum Operation` nella classe `NetworkUtils`. I dati aggiuntivi (identificati da `[data...]` nello schema sopra) non hanno un vero e proprio formato, e di conseguenza client e server devono accordarsi per scriverli e leggerli esattamente nello stesso modo.

Seguono alcuni esempi di comunicazione tra client e server:

LOGIN	"username"	"password"

LOGIN	false	"Username o password errati"

LIST_PROJECTS

LIST_PROJECTS	true	2	"Progetto prova"	"Progetto 2"

Implementazione client

`ClientAPI` offre un metodo, `estabilish(host)`, utilizzato prima delle chiamate a `register(username, password)` e `login(username, password)` per predisporre la connessione TCP e inizializzare la registry RMI.

`register` effettua la registrazione dell'utente tramite RMI, riceve dal server gli errori sotto forma di eccezione (da cui estrae il messaggio per offrire un responso all'utente) e in caso di successo presenta all'utente un messaggio proveniente dal server.

`login` è uno dei metodi più complessi: si deve infatti occupare di inizalizzare la connessione TCP con il server, tentare l'accesso (secondo il protocollo introdotto poco più su), registrare i callback RMI e chiamare oppurtunamente alcuni dei metodi di `ClientChatAPI`, per inizializzare il Multicast UDP.
In particolare, i callback RMI registrati con il server sono di due tipi: quello per ricevere update sullo stato degli utenti (online/offline e cambio "propic") e quello per ricevere messaggi in chat dal server. Nel primo caso a implementare la callback è lo stesso `ClientAPI` con cui, come accennato in precedenza, si può registrare una callback locale per aggiornare la UI; la seconda callback RMI è invece implementata da `ClientChatAPI` che, come si vedrà più avanti, si occupa di gestire il sistema di chat per conto di `ClientAPI`.

Tutti gli altri metodi che interagiscono con il server si avvalgono delle funzionalità di `serverCommunicationBoolean` e `serverCommunicationObject` che, sfruttando la programmazione funzionale di Java 8, permettono di semplificare e accorciare la lunghezza dei vari metodi, evitando la ripetizione di codice. A loro volta, questi due metodi fanno uso della classe helper `ServerCommunication` che si occupa di predisporre la richiesta secondo il protocollo applicativo e di controllare il responso del server.

Tutto questo permette, per fare un esempio, di implementare `showCard(projectName, cardName)` nel seguente modo, che altrimenti sarebbe molto lungo e verboso:

```
public CardInfo showCard(String projectName, String cardName) {
    return (CardInfo) serverCommunicationObject(Operation.SHOW_CARD,
    (buffer) -> { // send
        buffer.putString(projectName);
        buffer.putString(cardName);
    }, (buffer) -> { // receive
        return new CardInfo(cardName, buffer.getString(), ...);
    });
}
```

Per finire, quando `ClientAPI` viene inizializzato, registra uno snippet di codice da eseguire prima della terminazione del programma, utilizzando

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> { ... }));
```

Questo snippet si occupa di effettuare il logout, terminare la connessione con il server, terminare il thread della chat e rimuovere i callback RMI.

Implementazione server

Il thread di `ServerTCP`, dopo aver inizializzato un `ServerSocket` e un `Selector`, si mette in attesa finché una o più delle `key` registrate non siano disponibili per l'azione corrente. Quando un nuovo client si connette, si associa alla key un'istanza della classe Session, che memorizza lo stato dell'utente, il suo username, e il buffer contentente la richiesta; poi il selector si mette in attesa di un messaggio da leggere. Quando il client è pronto per ricevere la risposta, la richiesta viene gestita da un `ServerHandlerManager`.

Si è notato che quando un client crashava, il server si bloccava nel tentativo di leggere pacchetti vuoti. Per evitare questo problema, e per permettere al server di riconoscere un client crashato, si è aggiunto alla classe `Session` un valore `ttl`, inizializzato a 100, decrementato a ogni pacchetto vuoto e resettato per ogni pacchetto pieno. Quando il valore di `ttl` raggiunge 0 il client viene considerato "timed out" e il server procede al logout.

Per gestire le richieste il server, così come il client, si avvale di alcuni helper. In questo caso non si tratta di metodi, ma di una classe specifica, `ServerHandlerManager`, con cui si possono registrare `ServerHandler` per ogni op code. Il manager si preoccupa di estrapolare l'operazione dal buffer in entrata e chiama l'handler più adatto, non dopo aver messo in atto una serie di controlli di permessi.
Quasi tutte le richieste possono infatti essere compiute solo da utenti loggati, e molte richiedono che l'utente in questione sia membro del progetto su cui intende operare. Per questo `ServerHandlerManager` fornisce la possibilità di definire una maschera di *requirements* per ogni op code. Al momento esistono solo due requirement: *LOGGED* e *PROJECT_MEMBER*. Notare come, specialmente per il secondo, si faccia affidamento sul protocollo di comunicazione, che prevede che il nome del progetto su cui si intende lavorare sia il primo argomento della richiesta.

IO

`WorthBuffer`

Poiché tutta la comunicazione di rete avviene tramite canali NIO, il "vettore" che trasporta i dati è il `ByteBuffer` di NIO. Tuttavia, il normale `ByteBuffer` non era sufficiente per i bisogni dell'applicazione. La prima problematica riguarda l'impossibilità di strutturare in maniera vera e propria i dati: sebbene con i metodi `asCharBuffer`, `asFloatBuffer`, etc., si possa avere un po' più di controllo, oggetti come stringhe, liste e array sono più complessi da trasmettere. La seconda problematica è legata alla dimensione del buffer, che una volta istanziata non può essere modificata: diventa necessario un modo per o frammentare le comunicazioni troppo lunghe o ridimensionare i buffer dinamicamente.

Per ovviare a tutte queste limitazioni, è stata scritta una classe, `WorthBuffer`, che si comporta da wrapper di `ByteBuffer`, ma aggiunge anche alcune peculiari funzionalità. Metodi come `putString` e `getString` permettono di mandare dati di tipo più "avanzato", e il metodo `grow`, in accoppiata coi vari metodi di put e i metodi `read`, permette di avere un buffer di dimensione dinamica:

```
public void putString(String s) {
    int size = Integer.BYTES + Character.BYTES * s.length();
    if(buffer.remaining() < size) grow(size);
    buffer.putInt(s.length());
    for(char c : s.toCharArray())
```

```
        buffer.putChar(c);
}

public String getString() {
    int length = buffer.getInt();
    char data[] = new char[length];
    for(int i = 0; i < length; i++)
        data[i] = buffer.getChar();
    return new String(data);
}

public int read(SocketChannel socket) throws IOException {
    int n = socket.read(buffer);
    while(buffer.position() == buffer.limit()) {
        grow(0);
        n += socket.read(buffer);
    }
    buffer.flip();
    return n;
}
```

Serializzazione progetti

La serializzazione interna di Java non era adatta alle comunicazioni di rete, sia per NIO, sia per il suo overhead, sia perché gli oggetti `Project` e `Card` del server sono molto più complessi dei semplici dati che il client deve ricevere.

Tuttavia, la serializzazione è perfetta per il salvataggio del database di progetti. Una mappa che associa il project name alle istanze dell'oggetto `Project` viene salvata sul file *projectdb*. L'oggetto `Project`, però, ha bisogno di un processo custom di serializzazione: molti dei suoi dati infatti non vanno salvati su file o, nel caso delle card, bisogna salvarli su file separati. Per questo si definiscono i metodi `writeObject` e `readObject`, che serializzano solo i dati strettamente necessari, e chiamano la serializzazione della classe `Card`, che viene salvata su file separati, secondo la struttura richiesta dal testo del progetto.

Serializzazione utenti

Sebbene l'approccio sopra descritto fosse tranquillamente utilizzabile anche per gli utenti, in questo caso si è deciso che username e password sono dati così sensibili da dover essere salvati immediatamente, e non aspettando un timer o la chiusura del programma[2].

Ciò si traduce nel desiderio di poter fare un semplice append di dati in fondo a un file, senza doverlo riscrivere nella sua interezza. Gli utenti non possono quindi essere salvati con una normale serializzazione, ma devono essere rappresentati da uno stream di byte, al quale possa essere aggiunto un nuovo utente senza dover modificare i dati precedenti. Anche in questo caso, per avere strutturati i vari dati in un array di byte, si è utilizzata la classe `WorthBuffer`.

Chat, Multicast, RMI

Quando ragionavo sull'implentazione del sistema di chat, mi sono fermato a pensare su quanto fosse un'idea spiacevole, al di là dello scopo didattico, implementare un servizio di chat tramite

[2]Ogni minuto vi è un autosalvataggio del database dei progetti. Il database degli utenti viene aggiornato in diretta solamente quando qualcuno si registra (e quindi non salva aggiornamenti alle immagini profilo). Entrambi i database vengono salvati con il "gracefully exit" compiuto con il comando *exit* sulla console del terminale.

Multicast UDP. Ignorando il problema di sicurezza, per cui, in assenza di chiavi di cifratura e messaggi firmati, chiunque può fingersi chiunque, la problematica ovviamente maggiore è il fatto che si possa chattare solamente in rete locale, in quanto non si ha possibilità di registrare IP multicast su Internet. Non solo, come ho avuto modo di sperimentare durante una spiacevole sessione di debug, molte configurazioni di firewall bloccano gli IP multicast locali. Effettuando test tra un client su Windows 10 e uno su Arch Linux sembrerebbe anzi che i messaggi multicast non escano dall'host (non è chiaro se vengano bloccati dal router, da linux o da windows).

Mi è quindi velocemente nata l'idea di implementare una sorta di "fallback", un sistema che, riconosciuta l'impossibilità di comunicare con uno specifico utente tramite multicast, inoltrasse il messaggio anche al server, il quale avrebbe avuto la responsabilità di inviare il messaggio all'utente (o agli utenti) finali.

I paragrafi successivi descrivono la mia "avventura" nell'implementazione di questo sistema, e i miei esperimenti di esecuzione del progetto su Internet. Come accennato in precedenza, `ClientAPI` delega la gestione della chat a un'istanza di `ClientChatAPI`, la quale crea un secondo thread in ascolto per pacchetti UDP.

Multicast discovery

Quando un utente effettua il login, viene chiamato nella classe `ClientChatAPI` il metodo `multicastDiscovery`. L'utente invia tramite multicast un messaggio di, appunto, discovery, annunciando il proprio username e la sua disponibilità per il multicast. Quando un utente riceve uno di questi messaggi, risponde con il proprio username, per annunciarsi a sua volta. Per evitare cicli infiniti, a ogni query di discovery, viene associato un valore randomico. Un utente invia il proprio username solamente se non ha ancora risposto alla query identificata da quell'id. Gli id vengono "dimenticati" dopo un certo tempo, per minimizzare il rischio di collisione di valori generati dalla classe `Random`.

Quando si desidera inviare un messaggio via chat, si può confrontare il numero di utenti online (conosciuto grazie callback RMI) con il numero di utenti che hanno risposto alla query di discovery. Se differiscono, oltre a inviare il messaggio via UDP, lo si inoltra anche al server, che ha la responsabilità di recapitarlo a tutti gli utenti.

Alla ricezione di un messaggio se ne calcola l'hash, per evitare che venga visualizzato due volte, una in seguito alla ricezione via UDP, una in seguito alla ricezione dal server[3]. L'invio al server viene effettuato tramite lo stesso socket TCP delle normali richieste viste fin'ora.

L'ultimo pezzo del puzzle è far si che il server inoltri il messaggio ai client. Tuttavia, non si può utilizzare la normale connessione TCP, in quanto il protocollo non prevede che il client possa ricevere un pacchetto dal server se non in risposta a una determinata richiesta; UDP non può essere utilizzato in quanto è la causa dei problemi che si sta cercando di risolvere. Tra aprire una seconda connessione TCP con il server (con conseguenti problemi di login/logout) e l'utilizzo di una nuova callback RMI, la seconda soluzione è indubbiamente la più veloce da implementare. `ClientChatAPI` implementa quindi l'interfaccia remota `ChatFallbackReceiver`, chiamata dal server all'invio di un messaggio e registrata al momento del login.

Test su Internet

Dopo aver verificato che il sistema funzionasse in locale, bloccando gli indirizzi IP di multicast tramite firewall, rimaneva da testare se il progetto funzionasse su Internet, in modo anche da poterlo testare con qualche amico alla ricerca di bug e deadlock.

Il primo problema si è presentato quando server e client hanno cominciato a sollevare eccezioni del tipo *Connection refused to host: 127.0.0.1.* Dopo qualche ricerca si è scoperto che RMI di

[3]Di fatto non c'è nessuna ragione per inviare il messaggio via UDP se già lo si inoltra al server, ma per mantenere il più possibile la compatibilità con il testo d'esame, e nel tentativo di velocizzare i messaggi inviati su rete locale, si è deciso di aggiungere l'invio tramite server come possibilità, non come alternativa.

default inizializza i registry su indirizzo di loopback, ignorando la possibilità di ricevere richieste esterne[4].

Il problema può essere risolto impostando la proprietà `java.rmi.server.hostname`; da qui nasce la necessità dell'argomento per l'eseguibile del server. Dopo questo veloce fix, la registrazione dell'utente finalmente aveva successo e non sollevava nessuna eccezione.

Ciò di cui non mi ero ancora reso conto, però, è che una callback RMI non è altro che una funzione remota come quelle registrate dal server, che per funzionare rimangono in ascolto su un socket dedicato sull'host remoto. Ciò significa che con le callback il client in un certo qual modo si comporta da "server"; e di conseguenza qualsiasi normale client, posizionato dietro NAT, risulta irraggiungibile dagli altri host. Il problema si presenta anche con alcune configurazioni firewall. Dopo ore di sviluppo, test e debug, la mia soluzione era sostanzialmente inutile.

Sebbene non sia riuscito a risolvere il mio cruccio di non poter eseguire un progetto di reti su Internet, ho deciso di mantenere il codice per un paio di ragioni. Prima di tutto perché non volevo il mio impegno andasse sprecato e non venisse presentato nel progetto. In secondo luogo perché si tratta di un fallimento parziale: sebbene non si possa utilizzare il progetto su qualsiasi rete, lo si può comunque utilizzare tra host diversi della stessa rete, anche quando (come nel mio esperimento con Windows e Linux) il multicast risulta bloccato.

I paragrafi che seguono prendono in esame l'implementazione della chat.

Protocollo

I messaggi che possono essere ricevuti tramite UDP possono essere quindi di tre tipi: messaggi dagli altri utenti, messaggi dal server e messaggi di discovery. Questo significa che è necessario un semplice livello di multiplexing, per recapitare i messaggi alle classi che ne hanno competenza. I pacchetti hanno uno di questi formati:

`MESSAGE`	`timestamp`	`"project"`	`"from"`	`"message"`		
`SERVER`	`timestamp`	`"project"`	`"card"`	`"user"`	`from`	`to`
`MULTICAST_DISCOVERY`	`id`	`2`	`"user"`			

Per i pacchetti di tipo `MESSAGE` e di tipo `SERVER` il campo *project* è fondamentale per scartare tutti i messaggi che non riguardano il progetto corrente.

I messaggi di tipo `SERVER` sono fondamentali per mantenere aggiornata la visualizzazione del progetto: vengono infatti trasmessi alla GUI che in autonomia sposta le card da una colonna all'altra. Quando il campo from è nullo, allora la card non è stata spostata ma è stata creata. All'atto pratico, essendo questi messaggi di fondamentale importanza per mantenere uno stato coerente e corretto, il server li invia principalmente tramite RMI invece che tramite UDP[5].

In `ClientAPI` il metodo `readChat` permette all'UI di registrare un callback da chiamare ogni qual volta si riceve un messaggio. La callback può, nel caso di un client da linea di comando, salvarsi i messaggi per uso futuro oppure, nel caso di GUI, mostrarli a schermo. Per interrompere la callback si utilizza il comando `exitChat`.

[4] "*The hostname and port number you see in the exception trace represent the address on which the looked-up server believes it is listening. [...] The hostname which you specified in Naming.lookup to locate the registry has no effect on the hostname which is already embedded in the remote reference to the server.*", da https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/faq.html#domain

[5] Di fatto si potrebbe implementare un thread sul server che "sniffi" i pacchetti di discovery multicast per tenere traccia, come fa il client, del numero di utenti raggiungibili via UDP. Tuttavia, poiché UDP non dà la certezza che i dati arrivino senza problemi, per una comunicazione del genere mantenere RMI sembra una scelta saggia

Remote deadlocks

La parte più delicata del progetto non è tanto la concorrenza locale, con l'accesso a risorse condivise da più thread, quanto il rischio di deadlock distribuiti in remoto. Un esempio si verificava quando si inviava un messaggio via chat inoltrandolo al server. `ClientChatAPI` otteneva la lock segnando il metodo come `synchronized`, mandava un messaggio al server, che a sua volta cercava di mandare il messaggio al client di partenza tramite RMI, ma rimaneva bloccato perché anche la callback utilizzava `synchronized`. Client e server si bloccavano aspettandosi a vicenda. A peggiorare la situazione, il fatto che il server non sia multithreaded ma utilizzi il multiplexing di NIO, faceva sì che qualunque altro utente rimanesse bloccato. Come se non bastasse gli utenti non potevano neanche chiudere l'applicazione, perché lo ShutdownHook rimaneva bloccato nel tentativo di effettuare il logout con il server. Il problema è stato risolto liberando la lock prima di inoltrare il messaggio su TCP.

Un altro deadlock distribuito si era verificato durante i test su Windows. In quest'occasione accadeva che (per una ragione tutt'ora sconosciuta, ma solo dopo che la finestra era stata chiusa) durante la "validation" della GUI il thread di Swing rimanesse bloccato, bloccando in questo caso le chiamate ai callback RMI per l'update dello stato degli utenti. Ancora una volta il main thread del server rimaneva bloccato su una chiamata remota. Questo bug è stato risolto sostituendo tutte le occorrenze di

```
invalidate();
validate();
repaint();
```

con il più appropriato

```
revalidate();
repaint();
```

Purtroppo, avendo un'esperienza limitata con Swing, non posso avere la certezza che altri deadlock del secondo tipo non si possano verificare; tuttavia deadlock derivanti dalle lock e da concorrenza dovrebbero essere tutti scongiurati.