

Università di Pisa

***DIPARTIMENTO DI INFORMATICA***

***RETI DI ELEBORATORI E LABORATORIO***

***Anno Accademico:*** *2018/2019*  ***DOCENTE:*** *Federica Paganelli*

*Davide Neri*

***DATA DI CONSEGNA:*** *5/02/19*

***AUTORE***

*Alessio Russo*

**RELAZIONE**

*Version 1.4*

Contenuti

[1. Struttura Progetto 3](#_Toc195163)

[2. Scelte di Progettazione 3](#_Toc195164)

[2.1. Lato Server 3](#_Toc195165)

[2.1.1. Configurazione 3](#_Toc195166)

[2.1.2. ThreadPool 3](#_Toc195167)

[2.1.3. ConcurrentHashMap 4](#_Toc195168)

[2.1.4. Pool Multicast 5](#_Toc195169)

[2.1.5. Struttura dati Message 5](#_Toc195170)

[2.2. Lato Client 6](#_Toc195171)

[3. Struttura Moduli 6](#_Toc195172)

[4. Struttura processi client/server e protocolli di comunicazione 7](#_Toc195173)

[4.1. Comunicazione client e server 8](#_Toc195174)

[4.2. Protocollo di terminazione 8](#_Toc195175)

[5. Struttura logica del codice 8](#_Toc195176)

[6. Difficoltà incontrate 9](#_Toc195177)

[7. Interpretazione personale 9](#_Toc195178)

[8. Manuale 9](#_Toc195179)

# Struttura Progetto

L’obiettivo del progetto è la realizzazione di TURING. Il progetto è strutturato in 2 parti: un lato client e un lato server. Il lato client presenta una interfaccia grafica elementare realizzata tramite libreria java swing.

# Scelte di Progettazione

In seguito, la descrizione della configurazione e delle strutture dati usate dal server e dai client.

## Lato Server

Il server ha le seguenti caratteristiche:

### Configurazione

Il server è identificato dalla coppia <indirizzo di loopback, porta locale> con porta=4022-23, in modo tale da provare l’applicazione sulla stessa macchina.

Il server configura e offre un servizio RMI “TURING-SERVER” esportando un oggetto remoto, tramite registry, che espone il metodo per l’operazione di registrazione.

Per la gestione degli indirizzi multicast, ho assunto che al server venga assegnato un range di indirizzi “239.1.1.1-254” con possibilità di riuso.

In caso di esaurimento di indirizzi, è possibile proseguire con l’operazione di edit, senza avere la chat momentaneamente a disposizione, in attesa che si liberi un indirizzo.

Per ogni utente, al momento della creazione del suo primo documento, verrà creata una cartella in locale nel server relativo all’utente che memorizzerà le sezioni dei documenti. Ogni documento è suddiviso in n sezioni che corrispondono a n file nella forma “nome\_n.sezione”.

La cartella “server” conterrà quindi directories associati agli utenti con il seguente path “C:\\server\...” e verrà creata al momento della esecuzione di Server.java.

### ThreadPool

Per gestire l’interazione con più client in contemporanea, ho deciso implementare un server multithreading, tramite un threadpool.

Ad ogni client, quando effettua l’operazione di login, viene assegnato un thread worker appartenente al threadpool.

Tra le varie possibili politiche, ho scelto di usare un CachedThreadpool, dato che non è stato stabilito un limite massimo per il numero di client connessi, in modo tale da creare, senza limiti, un thread quando è necessario, e che eventualmente può essere riusato una volta terminato il suo lavoro. E se rimane inattivo per un determinato periodo di tempo, esso verrà terminato.

Quando un client si connette, viene creato un task worker che viene sottomesso al threadpool, per essere eseguito da un thread che soddisferà le richieste del client.

### ConcurrentHashMap

Il server utilizza delle strutture dati per mantenere traccia degli utenti registrati, utenti online e tutti i documenti creati.

Ho deciso di utilizzare questa struttura dati per i seguenti motivi:

* buona **scalabilità**
* **overlapping** di operazioni sugli elementi della struttura
* **lock** **striping**: hash table divisa in sezioni: possibili write simultanee se modificano sezioni diverse della tabella

1. **RegisteredUser:** contiene l’insieme degli utenti registrati tramite servizio RMI. L’username di ogni utente è univoco e viene quindi usato come chiave. Gli elementi sono quindi costituiti da coppie <username,UserData> dove UserData contiene la password e la lista delle notifiche. Poiché più thread worker possono andare a scrivere sulla stessa lista contemporaneamente, ho deciso di creare una versione sincronizzata tramite synchronizedlist della classe Collections. In questo modo tutti i metodi sono thread-safe.
2. **LoggedUser:** contiene l’insieme degli utenti online che hanno effettato con successo il login. Come nel caso degli utenti registrati l’username è univoco e viene usato come chiave per accedere al valore. Si tratta di coppie <username,LoginUserData> dove il valore è una struttura dati che contiene informazioni sul socket per la comunicazione con il thread worker e lo stream che viene usato per inviare a quel client la notificadi un invito da parte di un altro client, se esso è online o per notificare se una sezione richiesta, in passato occupata, viene rilasciata. L’utente viene rimosso da questa struttura nei seguenti casi:

* Logout esplicito
* Scade timer in caso di inattività persistente del client. Dopo il timeout se il client si riprende, quando proverà ad inviare una richiesta la connessione è ormai scaduta ed esso dovrà effettuare il login
* Scade il timer per la modifica della sezione, la connessione è scaduta e il client deve effettuare il login

1. **Document:** contiene la lista dei documenti creati dagli utenti. Poiché utenti diversi possono chiamare il documento con lo stesso nome, ho deciso di usare come chiave per memorizzare i documenti la stringa costituita da “username\_nome doc”, in questo modo più utenti possono usare lo stesso nome per i documenti. L’insieme contiene quindi coppie <username\_nome,DocumentData>. DocumentData è una struttura dati che contiene informazioni sul singolo documento come il creatore, i collaboratori, il numero delle sezioni, lista delle sezioni in editing. Per garantire l’accesso in mutua esclusione alle sezioni, ogni sezione è protetta da lock dedicata. Usando il meccanismo delle tryLock, se una sezione è già occupata , il client non si sospende, riceve la notifica che la sezione è occupata e può continuare a fare altro, in attesa che la sezioni si liberi. Per gestire la struttura dei collaboratori, ho deciso di usare delle read/write lock. Tale struttura può essere scritta da un solo thread, quello del creatore e può essere letta da molti thread. In questo modo possono esistere più lettori contemporaneamente e un solo scrittore alla volta senza nessun lettore.

### Pool Multicast

Si tratta di un List di Gruppi, dove Gruppi è una struttura dati che contiene informazioni su un determinato gruppo, in particolare: l’indirizzo multicast, la porta, assegnati in fase di inizializzazione del server, il nome del documento a cui è assegnato e il numero utenti.

Dopo una operazione di edit , il server, se non esiste nessun gruppo già attivo su quel documento, assegna il primo indirizzo libero e restituisce <indirizzo multicast, porta>. Se già esiste, allora restituisce <multicast, porta> ed incrementa il numero di utenti. Dopo una operazione di end-edit, se il gruppo non ha nessun utente attivo allora viene liberato per un possibile riuso futuro.

Tramite l’uso di lock si protegge tale struttura da accessi in contemporanea da parte di più thread.

### Struttura dati Message

Client e server si scambiano un oggetto di tipo Message:

* **Size**: contiene la dimensione del campo messaggio o valori speciali per alcune operazioni quando il campo messaggio è vuoto
* **Message**: contiene il messaggio da inviare che possono essere richieste o risposte
* **Operation:** indica il tipo di operazione da effettuare in fase di richiesta oppure codici sullo stato dell’operazione in fase di risposta

La classe Message implementa l’interfaccia Serializable. Ho fatto questa scelta per vari motivi:

* fornire un meccanismo di persistenza al programma.
* fornire un meccanismo di interoperabilità mediante oggetti condivisi.
* rappresentare l'oggetto in maniera indipendente dalla JVM che ha generato l'oggetto stesso.

## Lato Client

Il lato client presenta una interfaccia grafica costituita essenzialmente da textbox e bottoni. Ogni client per poter usufruire dei servizi offerti da TURING deve effettuare registrazione e login. La registrazione avviene tramite RMI , quindi il client deve ottenere il riferimento all’oggetto remoto messo a disposizione dal Server e invocare i suoi metodi.

Il Client è a conoscenza del nome del servizio remoto offerto dal Server e usa anch’esso l’indirizzo di loopback.

Il Client crea in locale una cartella nella locazione “C:\\client\...” che conterrà le sezioni da editare richieste, scaricate o interi documenti.

Il Client è organizzato in più thread, descritti in seguito.

# Struttura Moduli

**SERVER**

Multicast

Registrazione

RMI

Registered

User

1 INIZIO

2 OK

3 CONNECT

Thread Main

Pool

5 REQUEST

4 ACCEPT

6 REPLY

Logged

User

Thread worker

Listener

**COMUNICAZIONE**

**Client -> Servizio Registrazione RMI:**  richiesta di registrazione e risposta.

**Servizio Registrazione RMI -> Registered User** : registrazione.

**Client -> Thread Main**: instaurazione della connessione.

**Thread Main -> Worker:** accettazione connessione + creazione thread worker.

**Client -> Worker**: invio richieste.

**Worker -> Client**: invio risposte.

**Worker -> Registered User**: controllo utente registrato.

**Worker -> Pool**: assegnazione, rimozione e controllo indirizzi Ip multicast.

**Worker -> Listener Thread:**  invio di notifiche: inviti e sezioni libere.

**Client -> Multicast Thread:**  invio messaggi al gruppo.

# Struttura processi client/server e protocolli di comunicazione

Il server è costituito da 2 tipologie di thread:

* **1 thread Main:** si occupa essenzialmente di aprire due socket passivi per rimanere in ascolto di eventuali richieste di connessioni da parte dei client e per le notifiche.

Una volta che accetta la connessione, crea un task e lo sottomette ad un thread worker, generato eventualmente dal thread pool.

* **N thread Worker:** essi vengono generati dal thread pool ed eseguono il task, ovvero soddisfare le richieste di un determinato client. Un thread worker è dedicato ad un solo client.

Il client è costituto da 3 tipologie di thread nella comunicazione:

* **1 thread Main :** thread principale che si occupa di inviare richieste e ricevere risposte
* **1 thread Listener:** questo thread rimane in ascolto di notifiche per gli inviti e per le sezioni che il client ha richiesto ma che erano occupate e che si sono liberate. Questo thread viene lanciato se il login ha successo. Poiché più thread possono scrivere sullo stesso stream, ho deciso di proteggerlo mediante una lock, in modo tale che possa ricevere una notifica alla volta ed evitare conflitti. Ogni volta che si ha la necessità di terminare il client, in seguito ad un timeout o richiesta esplicita, si applica un protocollo per la terminazione del thread.
* **1 Thread Multicast:** questo thread viene creato durante la fase di edit. Esso rimane in ascolto di messaggi inviati dagli altri utenti che fanno parte dello stesso gruppo per la realizzazione della chat. Si gestisce la terminazione in caso di end-edit, inviando un messaggio speciale “username\_exit” che identifica il thread da terminare.

## Comunicazione client e server

Il client ed il server comunicano tra di loro mediante richieste/risposte.

Il client invia messaggi di richieste settando il campo operation in base all’operazione che vuole eseguire e settando eventuali parametri in input. Ho stabilito dei codici che permettono al server di capire quale operazione vuole eseguire:

* + - 0-> Login 1-> CreateDoc 2-> Invita 3-> Invia Lista
    - 4-> Edit 5-> End Edit 6-> ShowSez 7-> ShowDoc
    - 8 -> Logout

Il server risponde tramite un messaggio di risposta inviando un codice che indica lo stato che varia a seconda della operazione con un messaggio testuale corrispondente che l’utente deve visualizzare sia in caso di successo che di errore.

## Protocollo di terminazione

Vi sono due timer attivi che monitorano lo stato della connessione tra client e server. Un timer viene attivato alla creazione del thread worker. Se dopo un certo periodo di tempo il client non invia nessuna richiesta, scade la sessione, si inviano messaggi speciali ai thread del client per la loro terminazione. Se il client successivamente proverà ad inviare una richiesta esso verrà informato che la sessione è scaduta e dovrà rifare il login.

Un secondo timer viene attivato con l’operazione di edit. In caso di crash o di un possesso della sessione troppo prolungato, allo scadere del timer viene forzato il rilascio della sessione e la connessione scade e vengono inviati messaggi speciali di terminazione ai thread del client.

Il client deve effettuare esplicitamente la richiesta di logout per uscire e per spostarsi dalle finestre seguire i bottoni.

Nel caso di crash del server, il client visualizzerà un messaggio di connessione scaduta e tornerà alla finestra principale.

# Struttura logica del codice

Le classi principali sono:

* Server.java: tale classe deve essere eseguita per prima ed attiva il server
* Client.java: deve essere eseguita ed attiva il client

Classi di supporto per Server:

* TuringServicesImpl.java: implementazione servizio RMI
* TuringServices.java: interfaccia servizio RMI
* Gruppi.java: struttura dati
* DocumentData.java: struttura dati
* LoginUserData.java: struttura dati
* Task Worker.java: classe che implementa i servizi offerti

Classi di supporto per Client:

* Graphics.java
* ChatWorker.java
* ListenNotify.java

# Difficoltà incontrate

Lo svolgimento del progetto è durato circa 1 mese. Le lezioni di laboratorio e il materiale fornito è stato sufficiente per la realizzazione del progetto. Le parte che mi ha creato più difficoltà è stata la gestione delle eccezioni con la corretta terminazione e rilascio lock. Inoltre, ho approfondito la parte dell’encoding dato che l’applicazione non supporta i caratteri accentati.

# Interpretazione personale

* Possono esistere documenti con lo stesso nome con la condizione che il proprietario sia diverso e perciò per determinate operazioni bisogna specificare il creatore del documento.
* Ogni volta che una connessione scade il client deve effettuare il login
* Se il pool di indirizzi multicast si esaurisce allora la chat non è disponibile
* Se un client richiede una sezione che è momentaneamente occupata, riceverà una modifica quando essa verrà rilasciata.
* Si possono annullare tutte le operazioni

# Manuale

Pe mandare in esecuzione TURING eseguire le due classi nel seguente ordine:

* Server.java
* Client.java

I due processi non prendono nessun input e sono già configurati per essere eseguiti sulla stessa macchina su determinate porte.

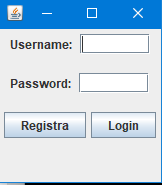
Se la porta sul pc su cui si esegue è già occupata, andare su questi due file modificare il parametro *serverPort*.

Seguire lo stesso risultato se la porta de registry è già occupata, ovvero modificare il parametro *servicesPort*

Lasciarsi guidare dall’interfaccia per muoversi tra le varie finestre.

L’applicazione client si divide essenzialmente in 3 finestre:

* Finestra di registrazione e login

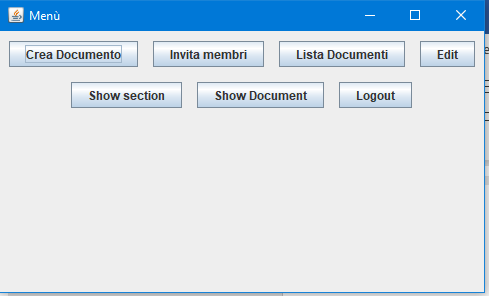


*Effettuare prima la registrazione e poi login al primo utilizzo.*

*Altrimenti effettuare direttamente il login.*

*Per chiudere l’app, chiudere la finestra.*

* Finestra Menù principale

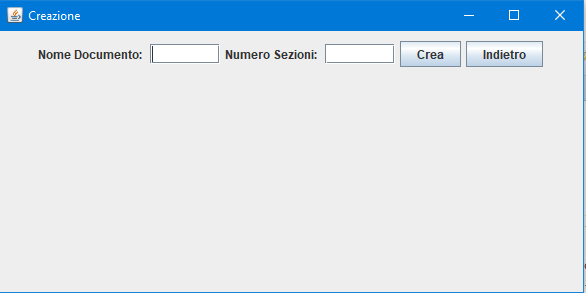


*Scegliere l’operazione desiderata.*

*Per uscire dall’app effettuare il logout.*

*La chiusura della finestra di default è disabilitata.*

* Finestre con in input variabili in seguito alla scelta di una operazione



*Esempio di creazione documento.*

*Compilare i campi.*

*La chiusura della finestra è disabilitata.*

*Per annullare l’operazione tornare indietro.*

NB:

* Per l’operazione di edit, l’utente può mantenere una sezione occupata per un massimo di 60 minuti.
* L’app client può rimanere attiva senza ricevere alcuna richiesta per un massimo di 100 minuti.
* L’utente in chat non può digitare “username\_exit”, in caso il messaggio non verrà inviato.

In tutte due i casi l’utente deve di nuovo effettuare il login.