Università di Pisa
Laboratorio di Reti di calcolatori (274AA)

Progetto TURING
Relazione finale

Alessandro Antonelli
(matricola 507264, corso A)

Appello straordinario aprile 2019
A.A. 2018/2019

Indice

1 Come compilare ed eseguire il codice

Il codice va compilato eseguendo il comando `javac` su tutti i file sorgente `.java`, sia quelli contenuti nella cartella `Server` che quelli nella cartella `Client`. Il programma è stato sviluppato tramite l'IDE Eclipse e testato su Ubuntu 16.04 LTS 64 bit.

Dopo la compilazione, per mettere in funzione TURING è necessario avviare il server e il client (nella maggior parte dei casi senza necessità di passargli alcun argomento), nel seguente modo:

```
java /Server/MainClass
java /Client/MainClass
```

Nel raro caso in cui le porte di default 41318, 41319 e 45687 fossero già assegnate a un altro servizio (cosa che verrebbe segnalata da un messaggio di errore), è possibile indicare manualmente delle porte libere da utilizzare, passandole come argomento nel seguente modo:

```
java /Server/MainClass PortaPrincipale PortaNotifiche
java /Client/MainClass PortaPrincipale PortaNotifiche PortaMessaggi
```

...dove tutti gli argomenti sono interi > 1024 e le porte principale e notifiche devono essere le stesse sia per il server che per il client. Non ci sono altri argomenti passabili dalla shell; tutta l'interazione con i due programmi avviene dopo l'avvio, tramite la lettura dello standard input (rimangono sempre attivi in attesa di input da tastiera).

Dopo la comparsa dello *splash screen* di benvenuto, sarà possibile utilizzare l'interfaccia a riga di comando del client, che permette l'inserimento dei comandi veri e propri. La lista dei comandi disponibili può essere visualizzata digitando `help` (al quale rimandiamo per la loro precisa sintassi); si tratta comunque dei comandi elencati nell'appendice A delle specifiche, ai quali sono stati aggiunti i comandi `help`, `stats` (che visualizza statistiche e parametri di configurazione) e `quit` (per chiudere il client). Similmente, anche il server ha una piccola interfaccia grafica che supporta i comandi `help`, `stats` (stampa di statistiche), `quit` e `quitnow` (rispettivamente, terminazione del server "morbida" e immediata).

2 Architettura complessiva e scelte

I/O su rete

La principale scelta di progetto da compiere è stata quella relativa alla comunicazione sulla rete, tra l'uso di primitive di I/O bloccanti (e quindi il ricorso al multithreading per gestire più connessioni) e il ricorso al multiplexing (e quindi l'uso dei channels e selectors forniti dal New I/O).

Nell'implementazione presentata si è scelto di adottare un'architettura del primo tipo, ricorrendo alle chiamate del pacchetto `java.io` e a un pool di thread di dimensione regolata dinamicamente, ciascuno dei quali si occupa di eseguire solo le richieste provenienti da una particolare connessione.

La ragione di questa scelta sta nella maggiore semplicità d'uso di tali chiamate di funzione, nella minore possibilità di commettere errori di programmazione e nella minore complessità del codice risultante, qualità che sono state ritenute più importanti della scalabilità e dell'efficienza nell'uso delle risorse di memoria e di calcolo (che – indubbiamente – sarebbero state molto migliori se si fosse adottato il multiplexing).

La connessione principale server $\leftrightarrow$ client è di tipo TCP e bidirezionale. Viene aperta quando il client invoca il comando `login` ed è persistente, ossia rimane aperta fino all'invocazione del comando `logout`.

I/O su disco

Per la lettura e la scrittura dei file su disco, invece, sono stati adottati i FileChannel di NIO, e in particolare si è utilizzato il *Direct Channel Transfer* per riversare efficientemente il contenuto dei file dalla/alla rete internet, cosa importante visto che la dimensione delle sezioni può essere potenzialmente molto elevata.

Accesso concorrente ai file

Per assicurare che sul lato server non si verfichino sovrapposizioni tra la scrittura del contenuto un file (`end-edit`) e altre scritture o letture (`show`), il descrittore del documento (classe `DocumentoLatoServer`) è stato dotato di un oggetto di pura sincronizzazione chiamato `lockTuttiFile`, che viene utilizzato per regolare l'accesso su disco a tutti i file che memorizzano tutte le sezioni di cui è composto il documento, per mezzo della sua lock implicita. La lock viene acquisita sia in caso di lettura che in caso di scrittura. Questo approccio è molto restrittivo e avrebbe ampi margini di miglioramento

(consente un'unica lettura o scrittura alla volta tra tutte le sezioni del documento), ma si è preferito non usare soluzioni più complesse per maggiore semplicità di progetto e per non rischiare di compiere errori che compromettessero la correttezza dell'implementazione. Si sarebbe potuta aumentare la granularità delle lock usandone una per ogni sezione, ma poi sarebbero stati necessari accorgimenti per evitare la deadlock (due comandi "visualizza intero documento" avrebbero potuto andare in deadlock dopo aver acquisito ciascuno una porzione incompleta delle lock). Inoltre l'uso delle ReadWriteLock avrebbe potuto migliorare il grado di parallelismo, consentendo un comportamento di tipo "multiple readers – single writer".

Modifica di sezione bloccata

Se al server perviene una richiesta di modifica su una sezione già in corso di modifica da parte di un altro utente, si è scelto di non consentire l'operazione, segnalando al client fallimento (tramite una `OtherUserEditingException`). L'alternativa di registrare l'interesse dell'utente è stata scartata perché avrebbe complicato non poco il progetto, richiedendo la creazione di una coda e di meccanismi di sospensione e risveglio dei client (nonché di *keep-alive* per prevenzione delle deadlock).

Interfaccia utente

Per l'interazione con l'utente si è scelto di adottare un'interfaccia testuale, con immissione dei comandi e visualizzazione dei risultati a riga di comando (*Command Line Interface*). Un'interfaccia di tipo grafico sarebbe stata certamente più intuitiva e *user-friendly*, ma la maggiore complessità di realizzazione e la mancanza di familiarità con le librerie coinvolte (poco trattate durante il corso), hanno portato a preferirgli la più semplice CLI.

Conseguentemente, il client realizzato è di tipo "sempre attivo" (avvio → richiesta1 → ... → richiestan → chiusura), e non di tipo "volatile" (avvio → richiesta → chiusura) come sembra suggerire l'appendice A delle specifiche.

Notifiche

Per la consegna delle notifiche di inviti, si è utilizzata una seconda connessione TCP monodirezionale (server → client). Questa viene aperta al momento dell'accesso all'account, subito dopo aver stabilito la connessione principale, e viene chiusa al logout. La connessione avviene con un handshake opposto rispetto a quella principale (è il client ad aprire un listening socket verso il server e ad attendere che quest'ultimo si connetta).

Se l'utente è online, l'esecuzione della `share` provoca l'immediato invio di un oggetto di tipo `Documento` (che contiene la coppia `<nome documento, autore>`), tramite la connessione dedicata alle notifiche. Quando l'utente è offline, gli inviti sono accumulati in un campo del descrittore relativo all'utente (la coda `condivisioniPendenti`); alla connessione gli elementi al suo interno vengono inviati al client tramite la connessione apposita.

Chat

Il servizio di chat è stato realizzato tutto sul lato client, senza coinvolgere in nessun modo il server (quindi senza un vero e proprio stato). Le chatroom sono costituite semplicemente dagli stessi gruppi multicast; ogni documento in fase di modifica ha una sua chatroom appositamente riservata (con un proprio indirizzo IP). La cronologia dei messaggi non viene salvata da nessuna parte, e se nessuno è in ascolto il messaggio viene perso.

I client si aggiungono sul gruppo all'inizio di una modifica (occasione in cui viene inviato automaticamente un messaggio che avvisa dell'ingresso in chat dell'utente); inviano un messaggio ad ogni invocazione della `send`; infine al termine della modifica escono dal gruppo (occasione in cui viene inviato automaticamente un messaggio che avverte dell'abbandono dell'utente). I dati inviati sono la serializzazione di un tipo di dato apposito, che comprende il corpo del messaggio, la data d'invio e il nome dell'utente mittente.

L'indirizzo del gruppo è generato casualmente dal server alla ricezione di una richiesta di modifica a un documento dove nessun altro utente stava editando, estraendo un indirizzo IPv4 a caso dal range riservato al multicast (224.0.0.0 - 239.255.255.255), controllando però che l'indirizzo generato non coincida con uno di quelli già in uso su altre chatroom (se ne tiene conto con una HashMap di `InetAddress`), e che non sia uno di quelli riservati da IANA. Non è stato necessario preoccuparsi dell'uso di questi indirizzi da parte di altre applicazioni, perché il client imposta il TTL di tutti i pacchetti a 1, in modo tale che questi non lasciano mai la rete locale. Se i client non fossero tutti sullo stesso host, diventerebbe necessario evitare collisioni, utilizzando protocolli come MADCAP o MASC. Alla disconnessione dell'ultimo editor, l'indirizzo della chatroom viene rimosso dalla HashMap di quelli in uso.

3 Strutturazione del codice e delle classi

3.1 Pacchetto `turing.server`

MainClass Contiene il `main` del server.

ConfigurazioneServer Luogo centrale per la memorizzazione di tutte le costanti necessarie per l'esecuzione del server (fissate all'avvio e non modificate successivamente).

CLI Contiene il codice che genera la Command Line Interface (in particolare la procedura `CicloFetchDecode`), interpretando l'input da tastiera.

TuringServer Definisce l'oggetto "server del servizio TURING", con il suo apparato di servizi, thread di supporto, di cartelle del file system e strutture dati di book-keeping. Contiene il codice per inizializzare il server e i metodi getter e setter per operarvi e per terminare la sua esecuzione.

TaskEsecuzione Definisce il concetto di sessione utente (intesa come il servizio di uno stesso utente nell'intervallo compreso tra login e logout). Costituisce il task che viene sottoposto ed eseguito dal thread pool alla ricezione di una nuova connessione. Contiene pertanto il codice dei thread appartenenti al pool (in particolare, il ciclo in cui si attendono e si interpretano le richieste, e i vari metodi che si interfacciano col server per applicare gli effetti dei diversi comandi).

TaskAccettaConnessioni Definisce il task eseguito dal thread di accettazione delle nuove connessioni, con il codice per rimanere in attesa, eseguire l'handshake, e sottomettere il task della sessione al thread pool.

CreatoreAccount Interfaccia remota del servizio RMI che gestisce la registrazione di utenze nel server, implementata dai due stub.

GestoreRegistrazioniUtenti Classe implementazione (stub server) del servizio RMI che gestisce la registrazione di utenze.

3.2 Pacchetto `turing.client`

MainClass Contiene il `main` del client.

ConfigurazioneClient Luogo centrale per la memorizzazione di tutte le costanti necessarie per l'esecuzione del client (fissate all'avvio e non modificate successivamente).

CLI Contiene il codice che genera la Command Line Interface, in particolare quello per decodificare l'input da tastiera (procedura `CicloFetchDecode`) e quello per isolare e controllare gli eventuali parametri (procedure `Decode`*`Comando`*) prima di inoltrare la richiesta.

InoltratoreRichieste È il cuore del client. Definisce un oggetto "inoltratore" che si occupa di inviare (inoltrare) richieste sintatticamente valide al server e di ricevere la risposta. Contiene i metodi per inizializzare la connessione al server e per inoltrare i vari tipi di comandi. Si occupa inoltre di creare e gestire i thread di supporto, la struttura delle cartelle sul file system e di controllare la terminazione del client.

GestoreChat Definisce la struttura utilizzata per gestire la partecipazione del client alla chat, con il codice per la ricezione di messaggi eseguito dal thread dedicato e la coda dei messaggi pendenti acceduta dal thread principale.

GestoreNotifiche Definisce la struttura utilizzata per gestire la ricezione di notifiche dal server, con il codice eseguito dal thread "ascoltatore" e la coda delle notifiche pendenti acceduta dal thread principale.

CreatoreAccount Interfaccia remota del servizio RMI che gestisce la registrazione di utenze nel server, necessaria per ricevere lo stub client tramite il meccanismo della *reflection*.

3.3 Pacchetto `turing.tipidato`

Documento Rappresenta un documento memorizzato in TURING come semplice coppia `<nome documento, utente creatore>`.

DocumentoConDettagli Estende `Documento` aggiungendo alla rappresentazione l'insieme dei collaboratori e il numero di sezioni.

DocumentoLatoServer Estende `DocumentoConDettagli` aggiungendo una serie di strutture di book-keeping necessarie per la memorizzazione dei documenti sul lato server.

UtenteLatoServer Rappresenta il descrittore di un utente registrato al servizio TURING, comprensivo di tutte le strutture di book-keeping necessarie sul lato server per tenere traccia della sua attività.

MessaggioChat Rappresenta i messaggi scambiati sulla chat come tripla `<orario invio, utente mittente, testo messaggio>`.

3.4 Pacchetto `turing.exceptions`

Contiene classi che definiscono dei tipi di eccezioni personalizzati, che sono utili a identificare situazioni di errore comuni nel progetto e motivi per cui le

richieste del client possono non essere accettate (in tali casi, infatti, il server serializza e invia al client un'istanza dell'opportuna eccezione). Sono tutte di tipo *checked* e discendono dal supertipo `EccezioneTuring`, che viene esteso dalle seguenti classi specifiche (il cui nome è autoesplicativo dello scopo): `AlreadyLoggedException`, `EditingException`, `NonPositiveNumberException`, `NoSuchDocumentException`, `NoSuchSectionException`, `NoSuchUserException`, `NotEditingException`, `NotLoggedException`, `OtherUserEditingException`, `UnavailableNameException`, `UserRightsException`, `WrongPasswordException`.

4 Protocollo client-server

4.1 Handshake

La connessione del client al server avviene al momento del login, tramite un "handshake" che segue il seguente ordine:

- il client si connette al listening socket del server e quest'ultimo accetta, creando la connessione principale
- il client si crea un listening socket per ricevere la connessione delle notifiche e invia al server un byte di dati tramite la connessione principale, per comunicare di essere pronto
- il server riceve il byte e si connette, creando la connessione delle notifiche

4.2 Comandi successivi

Terminato l'handshake, l'esecuzione di tutte le richieste successive segue il seguente protocollo:

- il client invia (sulla connessione principale) una stringa che contiene il nome del comando oggetto della richiesta, che può essere: `logout`, `create`, `share`, `showsect`, `showentire`, `list`, `edit` ed `end-edit`. Questo passo viene saltato per il comando login, in quanto sottinteso (è sempre eseguito subito dopo l'handshake iniziale).
- il client invia gli eventuali argomenti del comando (stringhe o interi) sotto forma di dati strutturati.
- il server legge tali argomenti e tenta di eseguire la richiesta:

- se l'esecuzione va a buon fine, invia al client il valore booleano `true` seguito dagli eventuali oggetti e dati strutturati richiesti come risposta;
- se la richiesta non può essere eseguita, invia al client il valore booleano `false` seguito da un oggetto appartenente a un sottotipo di `EccezioneTuring`, che rappresenta la ragione per cui l'esecuzione non è stata possibile e contiene il testo da mostrare all'utente.

I dati strutturati di tipo primitivo si intendono convertiti in binario e viceversa con la classe `DataInput/OutputStream`; gli oggetti si intendono serializzati tramite la serializzazione standard di java (`ObjectInput/OutputStream`). Tutte le stringhe si intendono codificate in formato UTF-8 modificato.

Nei comandi che prevedono l'invio di file (`show` ed `end-edit`), viene inviato prima un `long` che esprime la dimensione in byte e solo successivamente il contenuto del file. L'`end-edit` costituisce una piccola eccezione all'ordine tradizionale, in quanto il long `fileSize` e il contenuto del file (pur costituendo un "argomento" della richiesta), vengono inviati dal client solo dopo la ricezione del booleano `true` da parte del server.

5 Implementazione del server

5.1 Schema dei thread

Il server è costituito da due thread più un pool di dimensione variabile:

Server-Interfaccia utente È il thread principale (entry point: metodo `main` di `MainClass.java`); si occupa di avviare il server e di gestire l'interfaccia utente, interpretando i comandi da tastiera (`CicloFetchDecode` della classe `CLI`), e infine di portare a termine la sua terminazione sicura.

Server-Accettazione connessioni Demone che rimane in attesa di richieste di connessione, occupandosi di accettarle e di sottomettere il task al pool, che si occuperà di eseguire i comandi. Viene avviato dal thread principale all'inizializzazione del server, in sede di costruzione dell'oggetto `TuringServer`; rimane in esecuzione fino allo spegnimento del server. Entry point: `run` di `TaskAccettaConnessioni.java`.

pool-1-thread-*i* Pool di thread che esegue i comandi richiesti dal client; ciascuno di essi serve sempre lo stesso, unico client fino alla disconnessione di quest'ultimo. I task (di tipo `TaskEsecuzione`) sono sottomessi

dal thread di accettazione, all'arrivo di una nuova connessione; la terminazione del pool è innescata dal thread principale all'inserimento del comando `quit` o `quitnow`. Entry point: `run` di `TaskEsecuzione.java`. La dimensione del pool è dinamica: i thread vengono creati ad ogni nuova richiesta di connessione, senza un limite superiore, nel caso in cui il pool sia già tutto occupato; vengono terminati dopo un minuto di inattività, senza eccezioni (non esistono thread "core"). La dimensione attuale del pool e la dimensione massima raggiunta durante l'esecuzione sono reperibili con il comando `stats`.

5.2 Schema delle principali strutture dati

Le tre principali strutture dati sono memorizzate come variabili di istanza della classe `TuringServer.java`:

utentiEsistenti HashSet di oggetti di tipo `UtenteLatoServer`, che ad ogni istante contiene i descrittori di tutti gli utenti registrati al servizio Turing.

documentiEsistenti HashSet di oggetti di tipo `DocumentoLatoServer`, che ad ogni istante contiene i descrittori di tutti i documenti memorizzati dal servizio Turing.

indirizziUDPinUso HashSet di oggetti di tipo `InetAddress`, che ad ogni istante contiene gli indirizzi IP di tutte e sole le chatroom (i gruppi multicast) che sono in uso, ossia con almeno un partecipante alla chat.

Queste tre strutture sono accedute, tramite i metodi getter e setter della classe, dai vari thread del pool e sporadicamente (a scopo di stampa delle statistiche) anche dal thread principale gestore dell'interfaccia grafica. La sincronizzazione di tutte e tre le strutture è ottenuta inserendo in tutti i metodi getter/setter dei blocchi synchronized(struttura) che agiscono sulla lock implicita della struttura stessa.

5.3 Schema delle cartelle nel file system

Per la memorizzazione del contenuto dei documenti salvati su TURING, il server utilizza la cartella `DatabaseDocumenti`, posta all'interno della "user working directory" (indicata dalla proprietà di sistema `user.dir`). Il percorso di tale cartella viene stampato chiamando il comando `stats`.

All'interno di questa, vengono create sottocartelle per ogni documento memorizzato, il cui nome è uguale al titolo del documento (che non crea problemi in quanto è assicurato essere unico), depurato da eventuali caratteri

illegali per il file system host tramite l'applicazione del meccanismo di escape usato per gli URL (classe `java.net.URLEncoder`). All'interno della cartella documento, sono presenti tanti file quante le sezioni del documento, ciascuno chiamato `sezi.txt`. La cartella-database viene eliminata alla chiusura regolare del server, oppure alla successiva esecuzione del server in caso di mancata eliminazione dovuta a crash o terminazione forzata.

```
User working directory
|- DatabaseDocumenti
|  |- NomeDocumento1
|  |  |- sez0.txt
   |  |- ...
   |  |- sezn.txt
   |
   |- NomeDocumento2
   |
   |- NomeDocumenton
   |  |- sez0.txt
   |  |- ...
   |  |- sezn.txt
```

6 Implementazione del client

6.1 Schema dei thread

Il client è costituito da tre thread:

Client-Interfaccia utente È il thread principale (entry point: `main` di `MainClass.java`); si occupa di avviare il client, di gestire l'interfaccia utente con un ciclo fetch/decodifica comando/esegui/stampa risultato (il `CicloFetchDecode` della classe `CLI`), inviare le richieste al server e ricevere le risposte (invocando i metodi della della classe `InoltratoreRichieste`).

Client-Notifiche Demone che rimane in attesa di ricevere notifiche dal server e le appende in una coda condivisa con il thread principale (sincronizzazione risolta tramite l'uso di blocchi `synchronized` sulla lock implicita della coda). Viene avviato dal thread principale quando il client esegue l'accesso a un account (comando `login`) e rimane in esecuzione finché non si esce dall'utenza (`logout`). Entry point: `run` di `GestoreNotifiche.java`.

Client-Chat Demone che rimane in attesa di ricevere messaggi dalla chat e li appende in una coda condivisa con il thread principale (sincronizzazione risolta tramite l'uso di blocchi `synchronized` sulla lock implicita della coda). Viene avviato dal thread principale solo quando il client entra in modalità modifica (comando `edit`) e viene chiuso non appena la modifica è conclusa (`end-edit`). Entry point: `run` di `GestoreChat.java`.

6.2 Schema delle cartelle nel file system

Per la memorizzazione nel file system dei documenti ricevuti dal server (durante la loro visualizzazione o modifica), il client utilizza la cartella `DocumentiScaricati`, posta all'interno della "user working directory" (indicata dalla proprietà di sistema `user.dir`). All'avvio del client, ogni istanza del programma crea al suo interno una sottocartella `Istanza`*i*, cercando il primo indice disponibile, in modo tale che i download di un client non interferiscano con quelli di un altro. Il percorso della cartella legata all'istanza viene stampato chiamando il comando `stats`.

Al suo interno, i file sono organizzati in sottocartelle che hanno per nome il titolo del documento (che non crea problemi in quanto è unico), depurato da eventuali caratteri illegali per il file system host tramite l'applicazione del meccanismo di escape usato per gli URL (classe `java.net.URLEncoder`); al suo interno i file scaricati con le `edit` e `show sezione` prendono il nome di `sez`*i*`.txt`, mentre quelli scaricati con la `show documento` prendono il nome di `DocumentoCompleto.txt`.

All'uscita, il client fa pulizia eliminando la propria cartella dei download. Questo non accade se il processo crasha o viene ucciso; per evitare la proliferazione di cartelle residue, è stato introdotto un file "segnadata" vuoto, denominato `.DataSessione`, di cui ogni client aggiorna periodicamente la data di ultima modifica. All'avvio, ogni client controlla tutte le sottocartelle di `DocumentiScaricati` e si occupa di cancellare quelle dove il file ha un'ultima modifica più vecchia di 12 ore.

```
User working directory
|- DocumentiScaricati
|  |- Istanza1
|  |  |- .DataSessione
   |  |- NomeDocumento
   |  |  |- DocumentoCompleto.txt
   |  |  |- sez0.txt
   |  |  |- ...
```

```
|  |  |- sezn.txt
|
|- Istanza2
|
|- Istanzan
|  |- .DataSessione
|  |- NomeDocumento
|  |  |- DocumentoCompleto.txt
|  |  |- sez0.txt
|  |  |- ...
|  |  |- sezn.txt
```