Università di Pisa

Dipartimento di Informatica

Relazione del progetto `TURING`

RCL a.a. 18-19

Autore: Michele Zoncheddu

Maggio 2019

Indice

1 Scelte progettuali

Di seguito verranno illustrate le principali scelte progettuali effettuate durante la realizzazione del progetto.

1.1 Server

1.1.1 Architettura

L'architettura del server è stata scelta tra due soluzioni, che sono state trattate più in dettaglio durante il Laboratorio di Reti di Calcolatori:

1. multithread sincrona con I/O bloccante (`Sockets` di `Java IO`);
2. monothread sincrona con I/O non bloccante (`Selectors` di `Java NIO`).

Le due soluzioni hanno pregi e difetti complementari, ma il trade-off principale è tra **velocità** e **scalabilità**. Ai fini di questo progetto didattico è stata ritenuta più importante la reattività, garantita (sotto carichi non eccessivi) dalla prima soluzione.

Possiamo dividere il server in due livelli: **interfaccia** e **core**.

1. A livello di interfaccia si trovano:
   (a) Il socket TCP del server
   (b) L'API del servizio di registrazione utente
   (c) Il servizio RMI di notifiche push.
2. A livello core si trovano:
   (a) Il thread pool di client handlers
   (b) I manager di utenti, documenti e indirizzi IP multicast.

Per ogni client connesso, il server riserva un thread, che si occuperà di gestire la connessione TCP per l'intero tempo di vita del client, controllando la validità dei messaggi ricevuti e soddisfando le richieste.

L'**automa a stati** del client di `TURING` è stato implementato con tre stati interni ad ogni client handler, per garantire che le operazioni richieste siano coerenti con lo stato del client.

L'invio di messaggi **multicast** viene effettuato tramite un `datagramChannel` con il **TTL** settato a 1, che viene aperto quando c'è almeno un utente che sta modificando il documento, e chiuso quando nessuno lo sta più modificando.

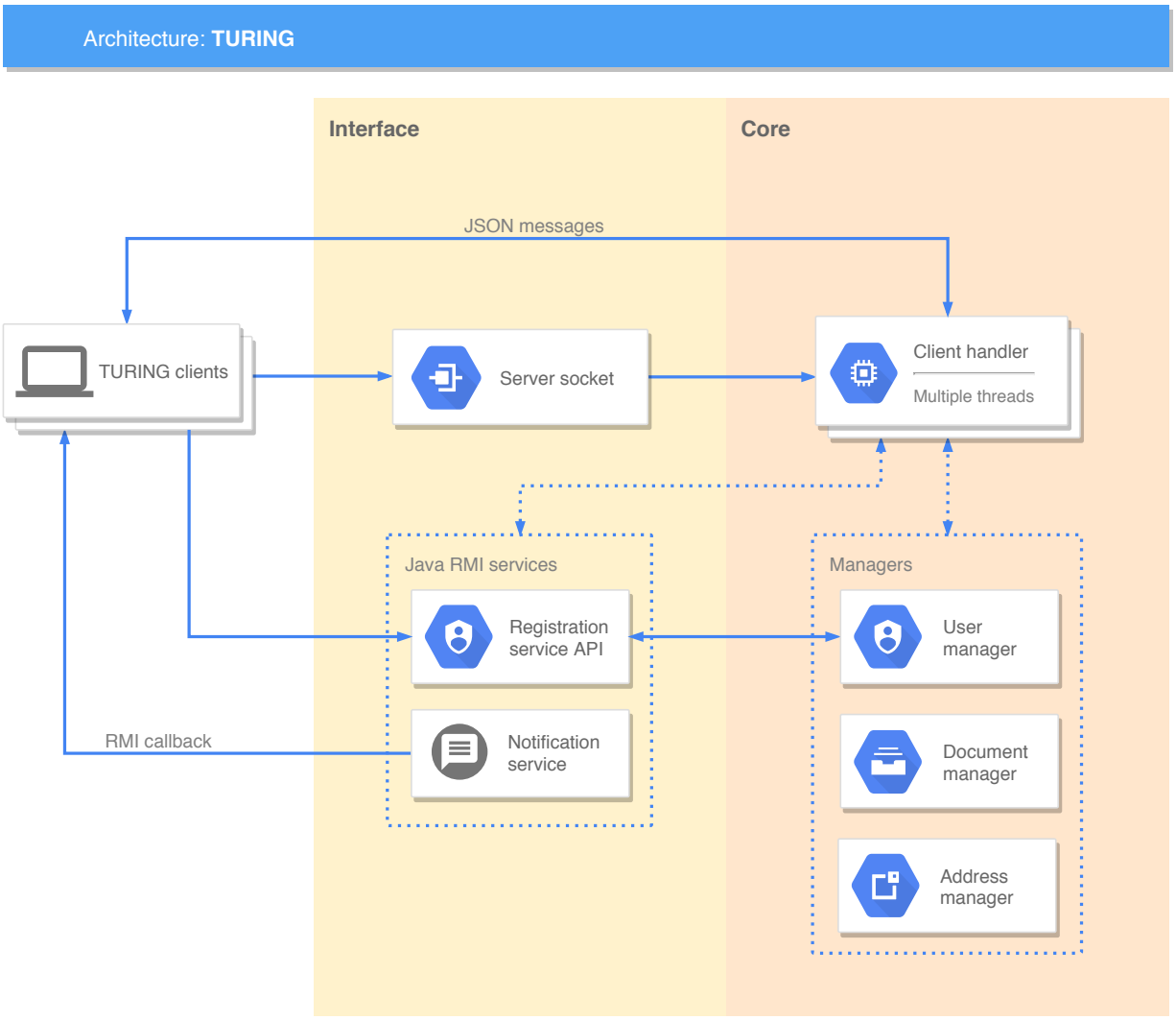

Figura 1: Architettura di `TURING` e interazioni tra le varie componenti.

1.1.2 Strutture dati

Le principali strutture dati utilizzate sono **tabelle hash** (per gli utenti e per i documenti) e **alberi** (per gli indirizzi). Verranno approfondite nella sezione di gestione della concorrenza.

1.1.3 Protocollo di terminazione

Quando il server viene terminato (a parte il caso eccezionale del segnale `SIGKILL`), viene avviata una routine di cleanup in un nuovo thread, nella quale:

- viene chiuso il socket del server;
- viene attesa la terminazione dei thread del pool per un tempo massimo pari al doppio del tempo di risveglio dei socket verso i client; se dopo questo tempo qualche thread è ancora in esecuzione, viene terminato forzatamente;
- vengono cancellati tutti i file di testo salvati dagli utenti, in quanto il server non mantiene uno stato da caricare al riavvio.

1.1.4 Organizzazione dei documenti

I documenti vengono salvati come file di testo codificati nello standard `UTF-8`, rispettando la seguente gerarchia (utente → documento → sezione):

```
data/
├── Michele/
│   └── relazione_turing/
│       ├── 1
│       ├── 2
│       └── 3
└── Dante Alighieri/
    ├── Comedia/
    │   ├── 1
    │   ├── 2
    │   ├── ...
    │   └── 100
    └── De vulgari eloquentia/
        ├── 1
        └── 2
```

1.2 Client

1.2.1 Architettura

L'architettura del client è più semplice: effettua richieste in formato JSON e resta in attesa di una risposta. Il thread grafico è aggiornato all'occorrenza dal metodo `invokeLater`. Durante la normale esecuzione, il client è single threaded; viene generato al bisogno un thread per la ricezione di messaggi multicast, durante la modifica di un documento.

L'interfaccia grafica si adatta automaticamente alle dimensioni dello schermo, ed è stato cercato di renderla il più coerente e intuitiva possibile.

Dalla schemata di registrazione e login si può accedere all'area personale di `TURING`, dove ogni utente può creare nuovi documenti, invitare altri utenti alla modifica, e in una tabella dinamica vede i documenti che può modificare, con la lista delle sezioni e informazioni aggiuntive, come il nome del creatore del documento (indispensabile nel caso di documenti omonimi) e un'icona che

indica se il documento è stato condiviso o no. La tabella si aggiorna in tempo reale quando si viene invitati alla modifica di un nuovo documento (con apposito messaggio di notifica), ma è anche possibile forzare un aggiornamento manuale tramite il tasto "refresh".

Dopo che una sezione di un documento è stata selezionata per la modifica, viene visualizzata la schermata di lavoro, nella quale è possibile modificare il testo precedentemente salvato, chattare con gli utenti che stanno modificando lo stesso documento, salvare le modifiche o ignorarle.

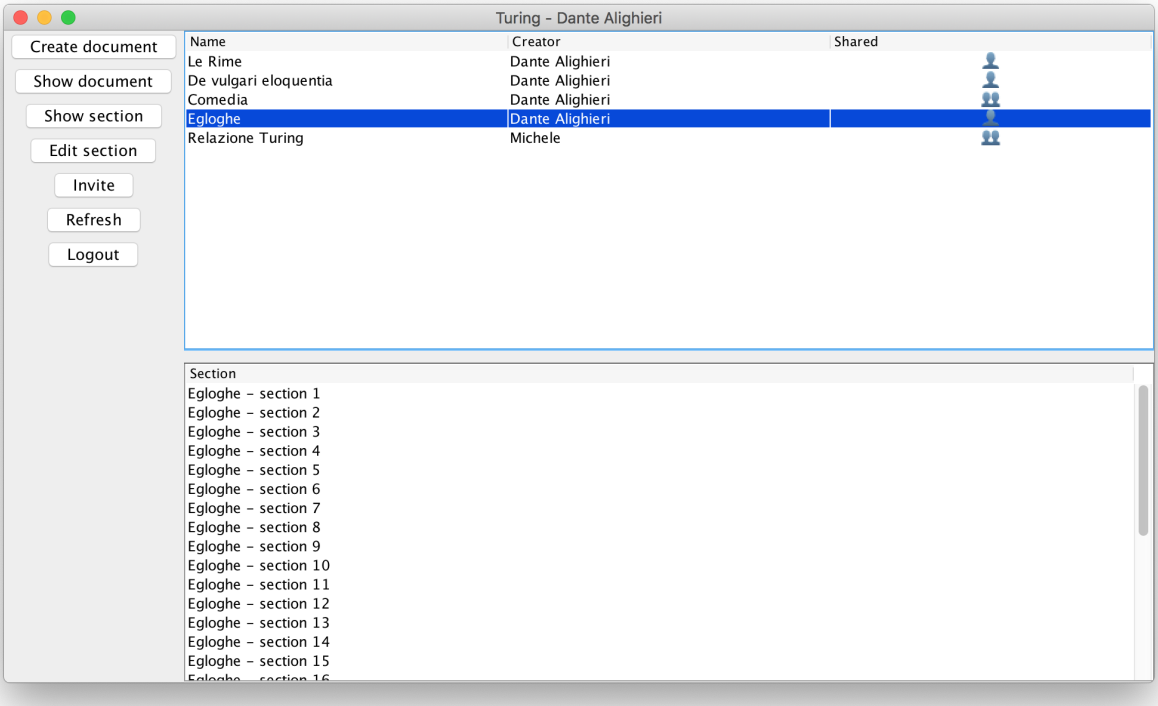

Figura 2: Elenco dei documenti di Dante Alighieri. Notare le icone per indicare lo stato di condivisione di un documento.

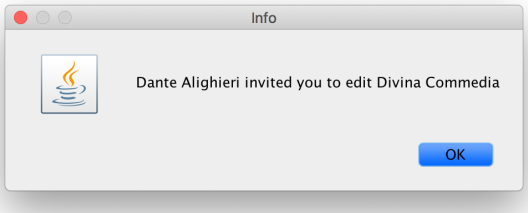

Figura 3: Notifica che indica che l'utente corrente è stato invitato alla modifica della Divina Commedia da parte di Dante Alighieri.

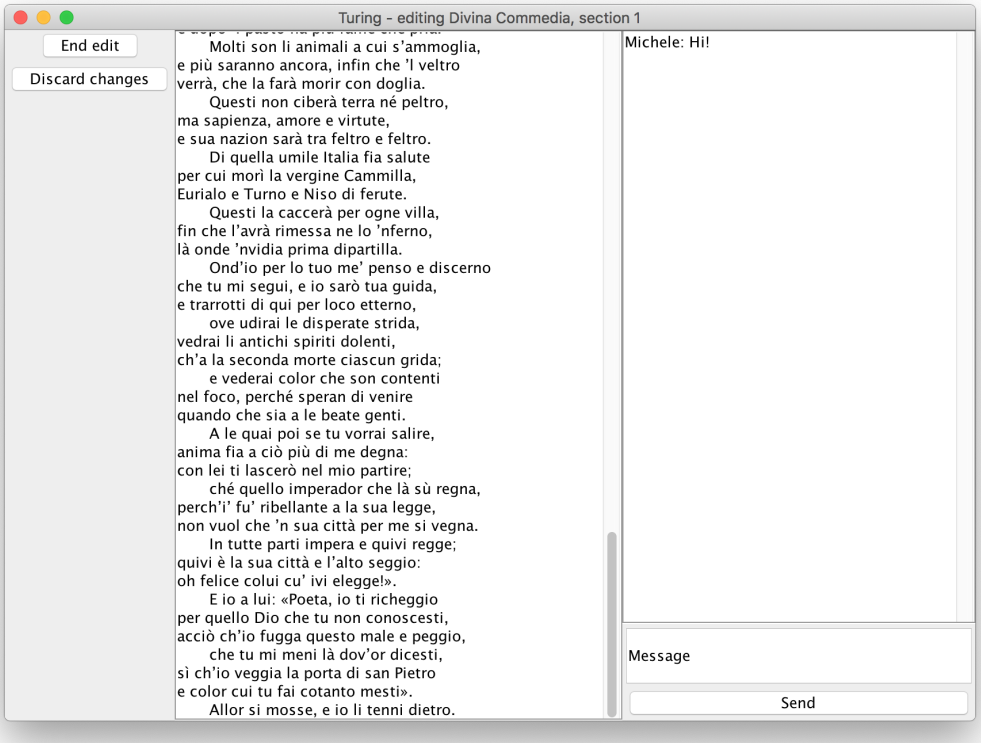

Figura 4: Dante Alighieri modifica la prima sezione della Divina Commedia, mentre gli arriva un messaggio in chat da Michele.

1.3 Comunicazione client-server

Per la comunicazione client-server è stata adottata una tecnica a scambio di messaggi sincrona. I messaggi sono in formato JSON, e vengono inviati attraverso una connessione TCP. La scelta del formato JSON al posto degli oggetti Java serializzati permette una **maggiore interoperabilità**, consentendo in un futuro momento di scrivere client per `TURING` in qualsiasi linguaggio di programmazione[1].

Il formato dei messaggi è indicato nel file `message_format.txt` all'interno della cartella `examples`, assieme ad un esempio di messaggio che richiede al server la creazione di un documento.

1.4 Librerie

Sia nel client che nel server è stata utilizzata una libreria per la creazione e il parsing di oggetti JSON, scaricabile dalla repository di Maven (**link**).

2 Gestione della concorrenza

I thread sono gestiti da un **cached thread pool**, scelto per la sua elasticità nella gestione delle risorse.
Nota: se si suppone che il server venga mandato in esecuzione su una macchina dedicata, il cached thread pool risulta comunque una soluzione migliore del fixed thread pool, infatti:

- per un numero di client connessi minore o uguale numero thread del fixed t.p., le prestazioni sono paragonabili - a meno di estremi cambi di contesto;
- per un numero di client maggiore, il fixed non riuscirebbe a gestirli, mentre il cached sì, seppur sempre più lentamente al crescere del carico.

I thread sono inoltre indipendenti ed isolati, e per ogni richiesta esterna devono utilizzare uno dei seguenti manager:

- `userManager`: gestisce concorrentemente gli utenti registrati;
- `documentManager`: gestisce concorrentemente i documenti salvati;
- `addressManager`: gestisce concorrentemente gli indirizzi IP multicast da assegnare alle chat dei gruppi di lavoro.

[1] A patto di modificare la parte di registrazione utente, che è stata richiesta esplicitamente in Java RMI.

2.1 `userManager`

È il nucleo del servizio di autenticazione di `TURING`, ed è implementato da una tabella hash concorrente (chiave: username).

Viene utilizzato dal server come manager locale per il login e per i controlli di sicurezza delle operazioni, e dai client come manager remoto per la registrazione.

Funzionamento Ogni nuovo utente registrato viene inserito all'interno della tabella tramite il metodo `signUp`, e recuperato col metodo `get`. I client interagiscono col manager invocando i metodi esposti da una API.

2.2 `documentManager`

Implementato da una tabella hash non concorrente[2] (chiave: username concatenato con il nome del documento), gestisce la creazione, la modifica e l'insieme degli utenti autorizzati alla modifica.

Funzionamento I documenti vengono aggiunti col metodo `put` e recuperati con due metodi get: uno per il creatore del documento (per permettere la condivisione), e uno per i collaboratori (limitati alla modifica delle sezioni).

2.3 `addressManager`

Per assegnare e rilasciare su richiesta gli indirizzi multicast per le chat dei documenti, è stato utilizzato un `TreeSet` con un comparatore apposito per indirizzi IP, in quanto la classe `InetAddress` non è nativamente comparabile in Java. Viene interrogato tramite metodi `synchronized`.

Funzionamento Al manager viene richiesto di riservare un indirizzo multicast (da 239.0.0.0 a 239.255.255.255[3]) non appena un utente inizia a modificare una sezione di un documento: l'indirizzo viene quindi inserito nell'albero, ad indicare che è correntemente in uso. Infine viene richiesta la liberazione dell'indizzo quando l'ultimo utente termina la modifica del documento.

[2] In questo caso la corretta gestione della concorrenza non è garantita da una struttura dati concorrente, ma dalle locks delle sezioni, in quanto hanno una granularità più fine.

[3] Organization-Local Scope. Fonte: IANA.

3 Descrizione delle classi

Di seguito una breve descrizione delle classi, suddivise per package. Per approfondire, vedere il Javadoc alla pagina `index.html` nella cartella `doc/javadoc/`.

3.1 Classi condivise

- `Fields`: per i valori dei campi dei messaggi JSON;
- `UserManagerAPI`: per la registrazione degli utenti;
- `ClientNotificationManagerAPI`, `ServerNotificationManagerAPI`: per l'invio delle notifiche.

3.2 Classi del server

Oltre ai tre manager e al confrontatore di indirizzi IP, troviamo le classi:

- `Server`: la classe principale che lo inizializza (caricando ed effettuando il parsing dell'eventuale file di configurazione) e gestisce il dispatch delle connessioni;
- `ClientHandler`: worker che gestisce le connessioni con gli utenti;
- `User`: per memorizzare i dati degli utenti, come password, documenti di proprietà, documenti che è possibile modificare e notifiche pendenti;
- `Document`: per memorizzare tutti i metadati dei documenti, come creatore, utenti ammessi alla modifica e indirizzo della chat di lavoro;
- `Section`: per interagire con la singola sezione del documento, permettendo la modifica da al più un utente alla volta, e leggendo e scrivendo sul disco.

3.2.1 Eccezioni

- `AlreadyLoggedException`: se l'utente è già loggato;
- `InexistentDocumentException`: se il documento cercato non esiste;
- `InexistentUserException`: se l'utente cercato non esiste;
- `PreExistentDocumentException`: se il documento che si vuole creare esiste già;
- `UserNotAllowedException`: se l'utente non è autorizzato ad interagire col documento.

3.3 Classi del client

- `ChatListener`: un thread dedicato per la ricezione dei messaggi della chat, creato e distrutto all'occorrenza;
- `Client`: la classe principale: carica ed effettua il parsing dell'eventuale file di configurazione e fa partire il thread per l'interfaccia grafica con il metodo `invokeLater`;
- `ClientGUI`: la classe che si occupa della creazione e della gestione dell'interfaccia grafica;
- `ClientNotificationManager`: per registrarsi al servizio di notifiche;
- `Connection`: per implementare il protocollo request/reply;
- `Document`: rappresenta una versione minimale del documento salvato sul server;
- `Operation`: per implementare la logica del client.

4 Testing

Il progetto è stato testato con successo sulle seguenti configurazioni:

- MacOS 10.14 su `Intel Core i5-5257U`;
- Ubuntu 16.04 e Windows 10 su `Intel Core i7-2670QM`;
- Ubuntu 18.04 su `Intel Core i5-7400`.

Durante i test sono state riprodotte le condizioni più critiche, per esempio:

- simulazione di errori sul disco, cancellando file di alcune sezioni e richiedendo la loro modifica;
- tentativi di modifica simultanea della stessa sezione;
- tentativi di condivisione non autorizzati;
- modifica di una sezione con crash improvviso del client e successiva modifica da parte di un altro utente;
- invio delle notifiche in real time e delle notifiche pendenti.

Il server è stato testato anche su un VPS cloud, con gli stessi risultati. L'unica funzione mancante è ovviamente la chat, per via del TTL impostato ad 1 e degli indirizzi multicast locali.

5 Manuale d'uso

Importante: per la compilazione è richiesto **Java 11** o superiore.

5.1 Linux e MacOS

Compilazione Eseguire lo script `build.sh`, che si occuperà di scaricare la libreria JSON e di compilare il client e il server.

Esecuzione Per eseguire il server e il client, eseguire gli script `run_server.sh` e `run_client.sh`. Se si vuole avviare il server o il client con una configurazione diversa da quella standard, passare il file di configurazione come parametro.
Esempio: `./run_server.sh config/server.conf`

5.2 Windows

Compilazione Per compilare il server eseguire il comando dalla cartella `Turing`:

```
javac
  -sourcepath src/
  -classpath lib/<nome della libreria>
  src/turing/server/*.java
  -d <cartella di output>
```

Per il client sostituire `server/*.java` con `client/*.java`.

Esecuzione Sempre dalla cartella `Turing`, eseguire:

```
java
  -classpath <cartella di output>;lib/<nome della libreria>
  -Djava.net.preferIPv4Stack=true
  turing.server.Server <eventuale file di configurazione>
```

Per il client sostituire `server.Server` con `client.Client`.

Importante: Per separare le cartelle argomento di `classpath`, su Windows è necessario il punto e virgola, mentre su Linux e su MacOS i due punti.

5.3 Storia delle versioni

Codice sorgente sotto licenza MIT.
L'intera storia delle versioni è consultabile all'indirizzo:
`github.com/michelezoncheddu/Turing`.