TuringFX

Turing meets JavaFX

Marco Costa
<mcsx97@gmail.com>
matricola 545144

27 gennaio 2019

Indice

Parte I
Architettura

NOTA: I vincoli per l'implementazione sono stati rispettati (ovviamente secondo la personale interpretazione delle richieste di progetto) e ciò che poteva essere scelto o aggiunto è descritto nel seguito; perciò, per evitare ridondanze con quanto specificato nella descrizione del progetto fornita, la seguente relazione evita di ritrattare quanto già stato detto.

1 GUI

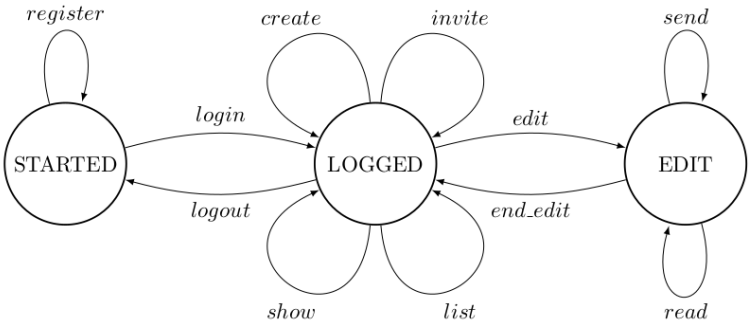

Figura 1: l'automa fornito nel documento *Info addizionali progetto.pdf*

La caratteristica principale di quanto prodotto è, ovviamente, l'interfaccia grafica per l'applicativo client realizzata mediante JavaFX, l'attuale standard per lo sviluppo di applicativi grafici con Java.

È stata realizzata un'interfaccia minimale composta da tre scene che rappresentano i tre stati dell'automa fornito (Figura 1), realizzate mediante *SceneBuilder,* un tool grafico per la costruzione di interfacce mediante *drag-and-drop.*

Il risultato prodotto per ogni scena è un file con estensione *.fxml* che fornisce la struttura della finestra, alla quale è una classe Controllore (realizzata in Java) contenente la logica applicativa. A queste è stato aggiunto un modulo per la gestione delle transizioni tra le diverse scene.

Per la realizzazione dei controlli grafici, quali notifiche ed interfacce di dialogo, è stato utilizzato *ControlsFX*[1]*,* un progetto OpenSource per JavaFX che fornisce innumerevoli controlli grafici «*out of the box»;* i quali, altrimenti, sarebbero risultati troppo *costosi* da implementare.

2 Protocollo di comunicazione

Mentre su UDP per realizzare la chat, lo scambio di stringhe, rappresentanti i singoli messaggi e trasformate in vettori di byte secondo una codifica prefissata,

[1] `https://github.com/controlsfx/controlsfx`

era una scelta quasi obbligatoria viste le specifiche del progetto, lo scambio di messaggi su TCP richiedeva una scelta implementativa: questa è ricaduta sulla realizzazione di un insieme di classi formanti una *gerarchia*[2] *(*formulata mediante interfacce ed ereditarietà) di possibili messaggi TCP da serializzare in vettori di byte, trasmettere sul canale mediante NIO e deserializzare alla ricezione. Una volta eseguita la deserializzazione del messaggio esso viene comparato mediante l'operatore *instanceof* di Java per verificarne la posizione specifica nella gerarchia ed essere elaborato di conseguenza.

Inoltre, poiché *instanceof* è un operatore binario che verifica che un oggetto sia tipo o sottotipo di una determinata classe o interfaccia, è stato possibile implementare un meccanismo di controlli *«a cascata»* partendo dalla radice fino alla classe effetivamente istanziata. Un esempio di funzionamento:

```
1  il client invia un oggetto serializzato di tipo InvitationRequest
2  il server riceve un vettore di bytes e lo "casta" ad oggetto TCPMessage
3  poiché il server accetta solo messaggi di tipo richiesta e non risposta,
       verifica che l'oggetto sia istanceof TCPRequest
4     altrimenti restituisce un errore al client
5  poiché ogni oggetto TCPRequest contiene il nome utente di chi ha
       effettuato la richiesta, verifica che (a meno che questa non sia
       una richiesta di Login) l'utente sia registrato al sistema
6     altrimenti restituisce un errore al client
7  switch su tutti i possibili oggetti che rappresentano messaggi di
       richiesta
8     poiché l'oggetto è instanceof InvitationRequest, gestiscilo come tale
```

3 Sicurezza

Anche se il focus principale del progetto non è quello della sicurezza vi è un minimo di attenzione a riguardo.

La struttura dati del server contenente gli identificativi degli utenti non è composta dalle associazioni *«utente - password»* in chiaro bensì utilizza una struttura più complessa (simile alle soluzioni adottate dai moderni sistemi operativi) nella quale ogni record è così strutturato: *«utente - nonce - hash(nonce:password)»*. Dove la funzione di hash è crittograficamente sicura, il nonce generato mediante TRNG (*True Random Number Generator*) e ':' l'operazione di concatenazione tra stringhe.

Questo permette diversi funzionamenti interessanti:

1. Il server non ha conoscenza della password dell'utente ma è *sempre* capace di effettuarne l'autenticazione.

2. Le proprietà di hash garantiscono di identificare univocamente messaggi diversi e ne rendono *difficile* l'invertibilità.

3. Un'eventuale serializzazione della struttura su disco non espone alcuna informazione sensibile.

4. La concatenazione tra nonce e password garantisce che, ad utenti diversi con la stessa password, sia associato un hash *differente* che non permetta di inferire, da parte di un attaccante, alcuna informazione circa l'uguaglianza delle password esaminando la tabella.

[2] consultabile nel Javadoc

Per la funzione di hash è stato scelto l'algoritmo *SHA-256* (la cui implementazione è garantita su tutte le versioni di Java[3]) fornito dalla classe *MessageDigest;* invece per la generazione del nonce è stata utilizzata la classe Java *SecureRandom*.

Tuttavia la richiesta di Login su TCP implementata richiede come parametro la coppia «username-password» in chiaro, contraddicendo a quanto di buono scritto prima. La motivazione risiede nel fatto che QUALSIASI protocollo di identificazione sicuro verso un server richiede che questo sia provvisto *almeno* di una coppia «chiave pubblica-privata». Infatti, inviare la coppia «username-hash» (dopo aver ricevuto dal server il proprio nonce associato e calcolato l'hash) è *tanto sicuro quanto* inviare la coppia «username-password» poiché si è banalmente esposti ad un attacco di tipo *replay-attack.*

4 Libreria

Per poter suddividere al meglio il codice e le classi di uso comune tra applicativo Client e Server è stato realizzata una libreria (*«TuringFXLibrary»)* contenente tutte le classi di uso comune ai due applicativi. Una descrizione dettagliata delle classi e dei *package* è contenuta all'interno del Javadoc.

5 Client

5.1 Architettura del Client

Il client è suddivisibile in due sezioni logiche: la GUI e la Logica di Comunicazione; la prima è, ovviamente, relativa ai controlli grafici e alla gestione delle finestre, mentre la seconda fornisce la logica di comunicazione con il Server mediante la definizione di interfacce (indipendenti dall'interfaccia grafica) e il sistema di gestione della Chat. Schematizzando:

- La **GUI** è formata dai vari Thread grafici di default (non sotto il controllo dell'utente) e l'Application Thread: il Thread principale con il quale l'utente interagisce; questo ha la responsabilità di interconnettere l'interfaccia e la logica applicativa, occupandosi delle risorse e della transizione tra le varie scene di modo che venga fornito all'utente un risultato trasparente.
- La **Logica di Comunicazione** è composta da:
  - Il **Driver** (*TuringFXClient/src/*client/ClientDriver.java): il quale si occupa di fornire un insieme di interfacce corrispondenti ai possibili messaggi di richiesta TCP (es. *requestLogin, requestDocumentCreation, ecc.)* e un sistema per la ricezione di notifiche *«in real-time»;* in particolare, poiché la ricezione della notifica deve essere *asincrona* rispetto all'utilizzo dell'utente, il Driver realizza un Thread per la comunicazione con il Server in modo *sequenziale non bloccante* (utilizzando una *SocketChannel*), al quale è stato delegato anche l'invio e la ricezione di richieste e risposte avanzate in modo *sincrono* dal Thread Grafico. Si occupa quindi di:

[3] vedi `https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html`

1. Ricevere le richieste TCP da parte del Thread Grafico, inviarle al Server, ricevere una risposta e comunicarla al Thread Grafico in attesa. La comunicazione tra i due Thread avviene mediante l'utilizzo di variabili sincronizzate.
2. Attendere dal Server una possibile notifica di invito, riceverla e aggiungerla alla coda degli eventi grafici, in modo che questa possa essere gestita dal Thread Grafico quando possibile.

Questo Thread viene avviato automaticamente una volta effettuato il Login[4] e richiesta la terminazione una volta effettuata la disconnessione.

– Il **ChatService** (*TuringFXClient/src/*client/ChatService.java): il quale fornisce una ricezione *asincron*a dei messaggi della chat senza bloccare il Thread Grafico, occupandosi di inviare sulla chat i nuovi messaggi provenienti dall'utente e richiedere l'aggiornamento della vista a seguito di un nuovo messaggio ricevuto.
Poiché il problema è molto simile al precedente, la soluzione adottata è affine a quella descritta poc'anzi (utilizzo di un Thread di comunicazione sequenziale e non bloccante, ma su UDP invece che TCP, quindi realizzata mediante *DatagramChannel* e non *SocketChannel*) con queste uniche differenze:
  * Il Thread Grafico «passa» la richiesta al Thread per la comunicazione UDP inserendo il messaggio alla coda di una *LinkedBlockingQueue.*
  * L'avvio del Thread avviene a seguito della conferma del Server relativa ad una richiesta di *Edit(S, D)* e chiuso successivamente ad una avvenuta *End_ Edit(S, D).*

5.2 Manuale d'uso

Poiché l'utente non interagisce digitando i comandi descritti nelle specifiche di progetto (*Create, Edit, ecc.)*, è riportata la corrispondenza tra i controlli grafici ed i comandi.

[4]la registrazione viene effettuata contattando il Server RMI e, prima di aver effettuato il Login, non è necessaria una gestione *asincrona* delle richieste

5.2.1 Login (o Started nell'automa)

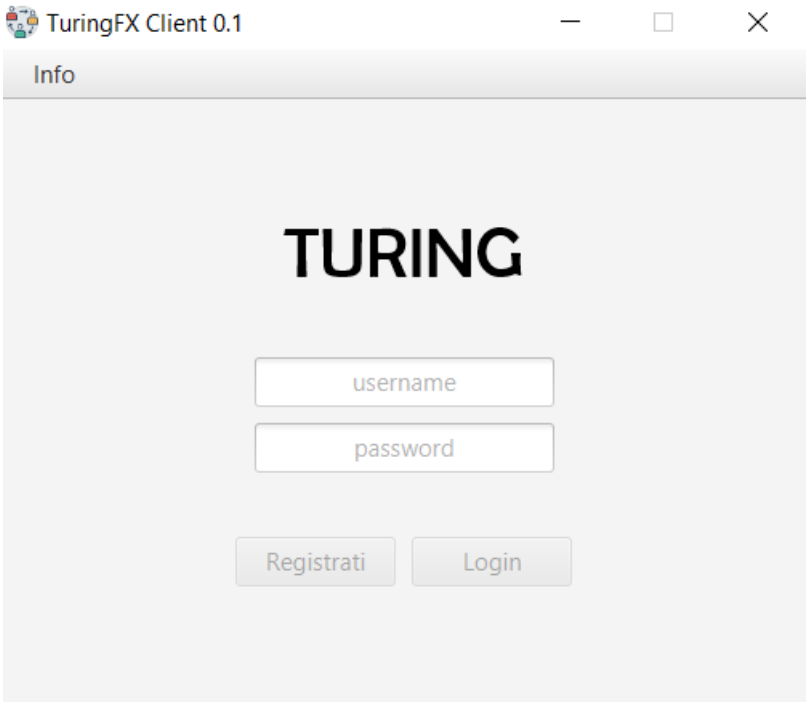

Figura 2: la schermata di Login, realizzata da LoginFXML.fxml e LoginFXMLController.java

La schermata è abbastanza intuitiva, sono presenti i due campi *username* e *password* nei quali inserire le informazioni dell'utente; cliccando su *Registrati* si procederà alla richiesta di registrazione con le credenziali inserite (sarà necessario confermare la password) o altrimenti su *Login* si procederà ad effettuare una richiesta omonima.

ATTENZIONE: come in altri controlli nell'applicazione, i pulsanti di Registrazione e Login sono disabilitati fino all'inserimento dei dati necessari a proseguire la richiesta.

5.2.2 Main Menu (o Logged)

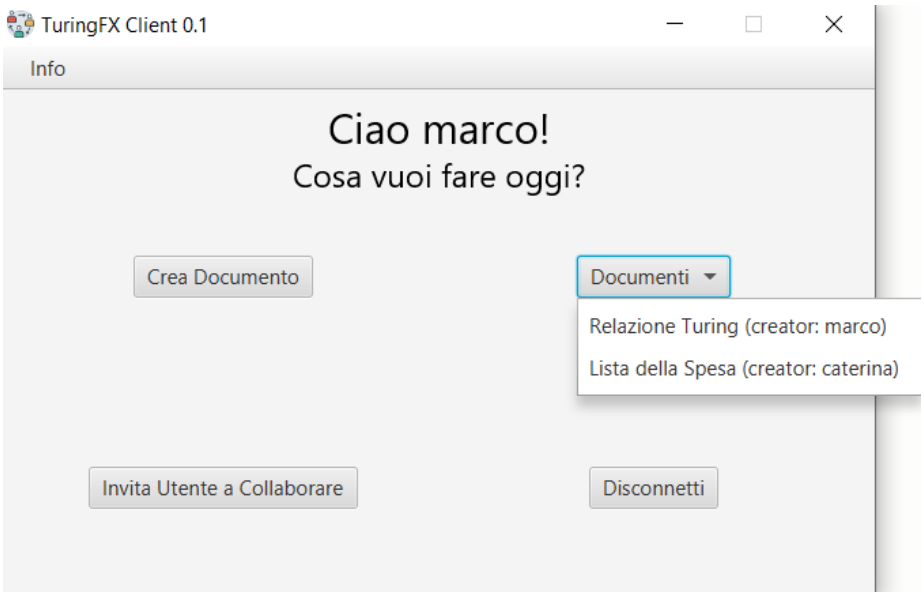

Figura 3: il menù principale, realizzato da MainMenuFXML.fxml e MainMenuFXMLController.java

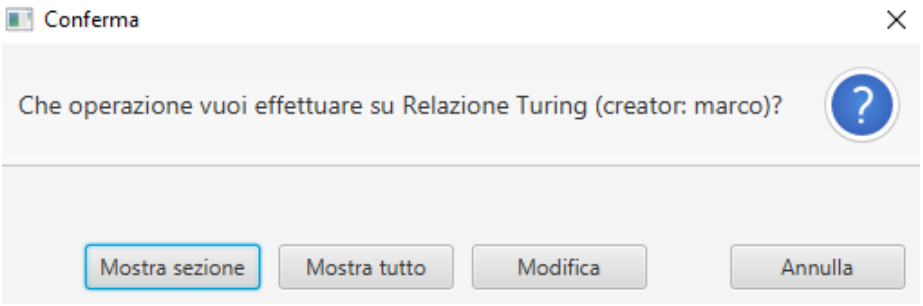

Figura 4: La schermata di scelta operazione su un documento

Dopo aver effettuato correttamente il Login (o terminando l'editing su un documento, vedi più avanti) si accede a questa schermata. I controlli corrispondono ad i seguenti comandi:

- **Crea Documento** ≡ (equivalente a) **Create(D, numeroSezioni):** una finestra di dialogo richiederà l'inserimento del nome del documento e del numero di sezioni
- **Documenti (menù):** un menù (disabilitato finché non si crea il primo documento o si riceve un invito) contenente la lista dei documenti ai quali

si ha accesso[5]. Selezionando una voce dal menù[6] apparirà un dialogo (Figura 4) contenente le seguenti scelte:

  - **Mostra sezione ≡ Show(S, D):** una finestra di dialogo richiederà l'inserimento del numero di sezione
  - **Mostra tutto ≡ Show(D)**
  - **Modifica ≡ Edit(S, D):** una finestra di dialogo richiederà l'inserimento del numero di sezione

- **Invita Utente a Collaborare:** invia ad un utente il permesso a collaborare: una finestra di dialogo richiederà il nome dell'utente da invitare e il documento creato sul quale collaborare. Il controllo è disabilitato finché un utente non crea il suo primo documento.
- **Disconnetti**: esecuzione della disconnessione e ritorno alla finestra di Login[7]

5.2.3 Edit Menu (o Edit)

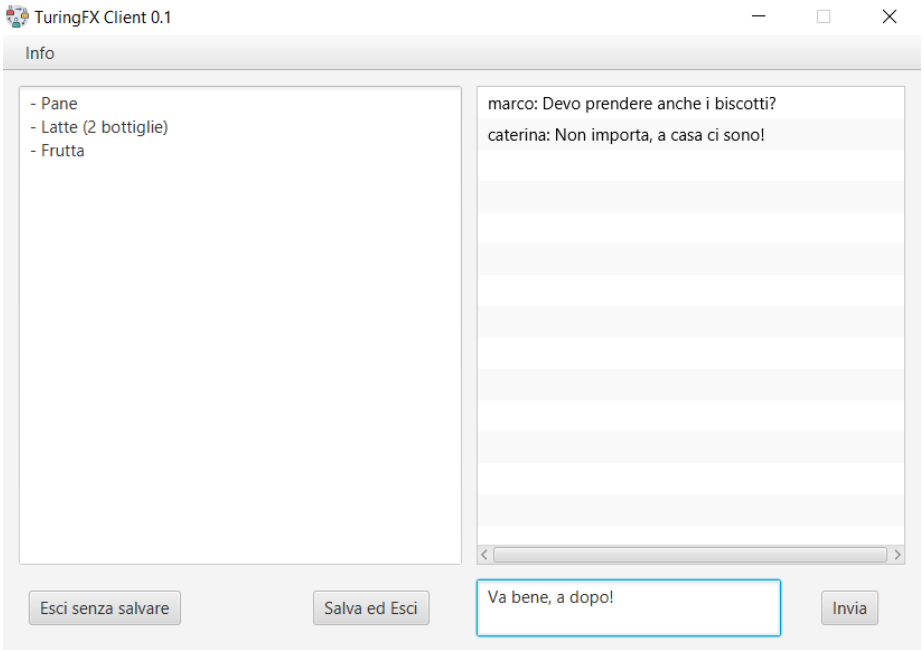

Figura 5: la finestra di Editing, realizzata da EditPageFXML.fxml e EditPageFXMLController.java

[5]per non creare omonimie è presente anche il nome del creatore

[6]**ATTENZIONE:** la GUI non utilizza il messaggio *DocumentListRequest* per ottenere la lista dei documenti accessibili dall'utente ma, la richiede al Login e la mantiene automaticamente aggiornata. In questo modo quando viene ricevuto un invito il documento corrispondente viene aggiunto *dinamicamente* al menù.

[7]**NOTA:** utilizzare il controllo o chiudere la finestra gestisce la disconnessione dal Server allo stesso modo

A questa finestra si accede dopo aver richiesto con successo una Edit o una Show, tuttavia nell'ultimo caso la finestra avrà un comportamento diverso: risulteranno disabilitati controlli e finestra relativi alla chat, disabilitato l'editing sulla TextArea del documento e disabilitato il pulsante di salvataggio, potendo solo visualizzare il documento.

In caso di Edit abbiamo i seguenti comandi:

- **Salva ed Esci ≡ End_Edit(S, D):** dove la nuova sezione è quella contenuta all'interno della TextArea sovrastante.
- **Esci senza Salvare ≡ End_Edit(S, D):** dove la nuova sezione è quella ottenuta dal Server prima di effettuare l'editing.
- **Invia:** invio del messaggio nella Chat del documento.

5.2.4 Notifiche

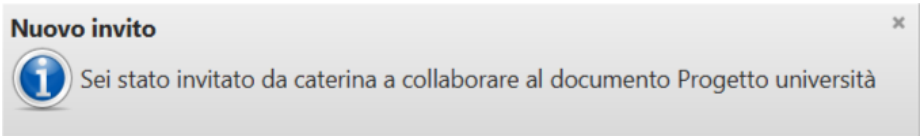

Figura 6: un esempio di notifica

Le notifiche possono essere ricevute solo una volta eseguito il Login: al momento del Login se l'utente ha delle «pending notification» o durante l'utilizzo dell'applicazione.

6 Server

Il Server è composto da *tre* Thread: uno realizza il Server RMI per effettuare la registrazione al sistema, il secondo è il Server TCP sequenziale e non bloccante e l'ultimo gestisce la creazione delle strutture dati, l'avvio degli altri due Thread, la gestione della terminazione e il rilascio delle risorse con conseguente chiusura.

Le principali strutture dati utilizzate sono due:

1. **SHA256UserManager**[8] (una istanza): una tabella Hash contenente tutti gli utenti (e relative informazioni e dati, si veda la classe *User*) registrati al sistema. La rappresentazione interna è stata realizzata mediante una *ConcurrentHashMap* in quanto l'accesso ad essa avviene sia dal Thread Server TCP che dal Thread Server RMI. La chiave di accesso alla tabella è il nome utente di registrazione.
2. **DocumentListHashImpl**[9] (una istanza): una tabella Hash contenente tutti i documenti creati nel sistema. La rappresentazione interna è stata realizzata mediante *HashMap*, infatti l'accesso ad essa avviene solo dal Thread Server TCP. La chiave di accesso è la coppia «nome documento - nome utente creatore»

[8] TuringFXLibrary/src/core/SHA256UserManager.java
[9] TuringFXLibrary/src/files/DocumentListHashImpl.java

6.1 Sincronizzazione sui File

Oltre alla *ConcurrentHashMap* non è stato necessario utilizzare nessun altro meccanismo di sincronizzazione sui dati, infatti, i metodi per lettura e la scrittura sui File rappresentanti le sezioni dei documenti sono state implementate in modo che ogni file rimanga in uno stato consistente al termine della funzione. Poiché l'accesso ad essi è sequenziale da parte unicamente del Thread Server TCP non è necessaria alcuna sincronizzazione.

7 Impostazioni di Turing

Sono stati impostati i seguenti valori predefiniti (modificabili in *TuringFXLibrary/src/core/TuringParameters.java)* :

- **SERVER_ADDRESS = "localhost"**: indirizzo IP del Server
- **TCP_SERVER_PORT = 1111**: porta di ascolto del Server TCP
- **RMI_SERVER_PORT = 1112**: porta di ascolto del Server RMI
- **CHAT_PORT = 1113**: porta utilizzata per la Chat Multicast UDP
- **FIRST_MULTICAST_ADDRESS = "230.0.0.0"**: primo indirizzo di multicast assegnabile (gli indirizzi vengono assegnati sequenzialmente a partire dall'indirizzo successivo)
- **PASSWORD_HASH_ALGORITHM = "SHA-256"**: funzione hash crittograficamente sicura
- **MIN_USERNAME_LENGTH = 4, MAX_USERNAME_LENGTH = 15**: vincoli sulla dimensione del nome utente
- **MIN_PASSWORD_LENGTH = 4, MAX_PASSWORD_LENGTH = 15**: vincoli sulla dimensione della password
- **MAX_CHAT_MESSAGE_LENGTH = 100**: massima dimensione di un messaggio di chat in *bytes*
- **DEFAULT_CHARSET = StandardCharsets.UTF_8**: charset utilizzato dal sistema per la codifica dei documenti e dei messaggi
- **DOCUMENT_MAX_SECTIONS = 10**: numero massimo di sezioni di un documento
- **DOCUMENT_SECTION_ENDS_WITH_NEWLINE = false**: indica se le sezioni sono separate da un'interruzione di linea *automaticamente* o a discrezione dell'utente. Il comportamento di default scelto è quello di non separare automaticamente le sezioni ma di lasciare all'utente l'opzione di terminare la propria sezione con un *newline* o meno.
- **FILE_PARENT_DIRECTORY = System.getProperty("java.io.tmpdir")**: nodo genitore per la directory di salvataggio dei File. È stata scelta la cartella temporanea della JVM.
- **FILE_DIRECTORY_NAME = "Turing"**: nome della directory dell'applicazione

8 Documentazione del codice

Il codice è interamente commentato e documentato mediante *Javadoc.* All'interno della cartella *./javadoc* è possibile trovare il risultato della compilazione della documentazione consultabile a partire dal file *./javadoc/index.html.*

Parte II
Esecuzione

Il codice è stato realizzato e testato utilizzando Netbeans 8.2 su S.O. Windows 10 (64-bit) ed è, a causa delle differenze *platform-depending* delle librerie grafiche (*«write once, run everywhere»* ma pare fino a un certo punto) e del JDK, su questo Sistema Operativo che se ne consiglia l'esecuzione. Tuttavia l'applicazione è stata testata sia su Fedora 29 sia su macchina virtuale contenente *xubuntu 18.04* e sembra produrre lo stesso comportamento.

9 Requisiti di sistema

Per poter eseguire l'applicazione è necessario disporre del JRE[10] (almeno) versione 8 e delle librerie di JavaFX.

Su Windows le librerie di JavaFX sono incluse nel Runtime (utilizzando presumibilmente il Kit di Oracle). **Su Linux**, invece, si distinguono due casi:

1. Utilizzo del JDK di Oracle: come su Windows, le librerie sono incluse
2. Utilizzo di OpenJDK (pacchetto *openjdk-8-jre/jdk*): le librerie non sono incluse ed è necessario installare il pacchetto *openjfx*[11]

10 Compilare ed eseguire il codice

10.1 Compilazione

Il codice è già compilato e le librerie sono già inserite nella struttura del progetto, tuttavia, se si volesse rieseguire la compilazione è necessario eseguire i seguenti passaggi:

1. Se non fosse già presente sulla macchina, installare *ant* (su Linux: *sudo apt install ant*)
2. Estrarre la cartella compressa e posizionarsi nella directory principale del progetto
3. Aprire un terminale in quella posizione ed eseguire:

[10] *Java Runtime Environment*

[11] dalla versione 11 di OpenJDK è necessario installare un Runtime aggiuntivo per JavaFX, per questo se ne sconsiglia l'utilizzo

- ***$ ant -f TuringFXClient jfx-rebuild*** per compilare il client
  ATTENZIONE: su Linux l'output potrebbe produrre il seguente errore:

  ```
  BUILD FAILED
  ...
  Error: Bundler "WebStart JNLP Bundler" (jnlp) failed to
      produce a bundle.
  ```

  Questo errore in compilazione è relativo solo alla compilazione dell'eseguibile WebStart e non sulla compilazione dell'eseguibile Jar.
- ***$ ant -f TuringFXServer* -Dnb.internal.action.name=rebuild clean jar** per compilare il Server

10.2 Esecuzione

10.2.1 Server

Dalla directory principale del progetto (utilizzando il terminale):
java -jar TuringFXServer/dist/TuringFXServer.jar
Per terminare il Server digitare un input (es: premere invio) sulla linea di comando.

10.2.2 Client

È possibile fare «doppio clic» su *TuringFXClient/dist/TuringFXClient.jar*, oppure avviarlo da terminale digitando:
java -jar TuringFXClient/dist/TuringFXClient.jar

10.3 Bug noti

Spesso, da Linux (su Windows non avviene) eseguendo il Client da terminale viene prodotto il sollevamento della seguente eccezione a runtime:

```
1 Exception in thread "JavaFX Application Thread" java.lang.
      NullPointerException  at com.sun.javafx.scene.control.skin.
      MenuBarSkin.lambda$new$383(MenuBarSkin.java:304)
2         ...
```

la quale non è stata propagata da alcuna classe utente. Questo bug è noto ed è segnalato nel Bug System di OpenJDK al seguente link: `https://bugs.openjdk.java.net/browse/JDK-8183520`. Il bug è stato risolto nella versione 10 del JDK.