

# GRANDMA'S TOMBOLA

*Politecnico di Milano – Progetto software*

Leonardo Arcari

Anno accademico 2015/2016



# INTRODUZIONE

---

## Specifiche

Si desidera realizzare un videogioco che simuli una partita a Tombola, noto gioco familiare italiano, in un'implementazione distribuita al fine di consentire a più giocatori di giocare la stessa partita, ognuno sul proprio calcolatore.

Pertanto è richiesta la presenza di un programma **server** che si occupi di sincronizzare le azioni dei giocatori che prendono parte allo stesso match al fine di simulare un'interazione reale. Il server inoltre consente di giocare più partite contemporaneamente, composte da un numero di giocatori arbitrario.

Inoltre è stato sviluppato un programma **client** dotato d'interfaccia grafica per consentire ad ogni giocatore di prendere parte ad una partita e giocarla nella sua completezza, interagendo con i suoi avversari.

## Linguaggio e librerie utilizzati

Il linguaggio di programmazione scelto per lo sviluppo è C. La scelta di esso è stata favorita da due motivazioni cardine. In primo luogo è l'unico linguaggio ufficialmente appreso finora durante il corso di studi ed è stato interessante realizzare un software *nella sua interezza* affrontando problematiche quali la modularizzazione del codice. In secondo luogo, con una vena più romantica, la mia passione per il mondo dell'informatica è nata grazie all'avvicinamento col sistema operativo Linux e ambiente grafico GNOME, che è scritto interamente in C ed è stato gratificante proseguire su questa linea.

Oltre alla libreria standard, sono state utilizzate altre tre librerie del progetto GNOME: **Glib**, **GIO** e il toolkit grafico **Gtk+3.0**.

Pregio di queste librerie è quello di aver dotato il C di una struttura *object-oriented-like*, attraverso strutture dati in luogo degli oggetti, funzioni particolari in luogo dei metodi e controllo di tipi per emulare il concetto di polimorfismo ed ereditarietà. Inoltre permetterebbero in linea teorica di poter scrivere un programma *cross-platform* quasi senza rimettere mano al codice.

# DOCUMENTAZIONE DEL CODICE

Allegata a questo documento, è stata prodotta una documentazione consultabile online a partire da commenti strutturati all'interno del codice sorgente. Mediante lo strumento gratuito **Doxxygen** è stato possibile generare una documentazione *javadoc-like* in formato HTML al fine di fornire una descrizione di ogni struttura dati, funzione, enum e costante originali scritte per questo progetto.

Motivo di questa scelta è stato fornire uno strumento di navigazione all'interno del codice in grado di garantire una maggiore leggibilità e al tempo stesso per permettere a questa relazione di non entrare troppo in dettagli implementativi. Qualora ci fossero curiosità in merito alla vera e propria implementazione, associato ai sorgenti sarebbe presente anche la documentazione.

Enumeration Type Documentation

Enum Name	String representation	Description
PLAIN	0000	Chat message
CREATE_GAME	0001	Create a new game room. Message content must have username and game name separated by a comma
JOIN_GAME	0010	Join a game room. Message content must have username and game name separated by a comma
START_GAME	0011	Start a game. Will be sent to all players in the same room
END_GAME	0100	Quit a game. Usually sent after a tombola has been granted
CLIENT_READY	0101	Placeholder. Not used
AMBO	0110	Ambo: 2 numbers on the same row
TERNA	0111	Terna: 3 numbers on the same row
QUATERNA	1000	Quaterna: 4 numbers on the same row
CINQUINA	1001	Cinquina: 5 numbers on the same row
TOMBOLA	1010	Tombola: all numbers in a table
EXTRACTED	1011	New number extracted

msg\_types\_t is an enum of types of message sent in the socket.  
These can be identified by their string message, so it's possible to interpret the message's content.  
Communication between server-client is made possible by sending a string in the following format:  
`OP_CODE MESSAGE_SEPARATOR MESSAGE_CONTENT`  
As string representation of OP\_CODE, natural binary coding has been chosen. Here's a short description of them:

```
gboolean number_clicked_handler ( GtkWidget * widget,
                                  GdkEvent * event,
                                  gpointer user_data
                                )
```

Callback function.

It handles the `clicked` signal emitted by the GTKEventBox widget that contains tables' numbers. In first place, the function collects number info

- The number already exists in its table's `Interface_references::picked_numbers` list. It's removed and its label's background set to transparent.
- The number doesn't exist so it's added to the list and its background is set to yellow.

Parameters

- `widget`: Pointer to GTKEventBox widget
- `event`: GdkEvent emitted
- `user_data`: Pointer to a `_Interface_references` struct

Returns

- Always TRUE

Bug:

Two fast clicks on the same widget doesn't change number's background two times, so an un-picked numbers has still yellow background, be executed when no other "more important" signals have to be handled.

La pagina è raggiungibile al seguente indirizzo:

[easyPOLI.it/grandma\\_s\\_tombola\\_doc/index.html](http://easyPOLI.it/grandma_s_tombola_doc/index.html)

# COMUNICAZIONE CLIENT-SERVER

La prima scelta di progetto richiedeva di fissare uno standard nel metodo di dialogo client-server. Si è scelto di adottare una comunicazione via **socket** inviando stringhe di caratteri opportunamente formattate.

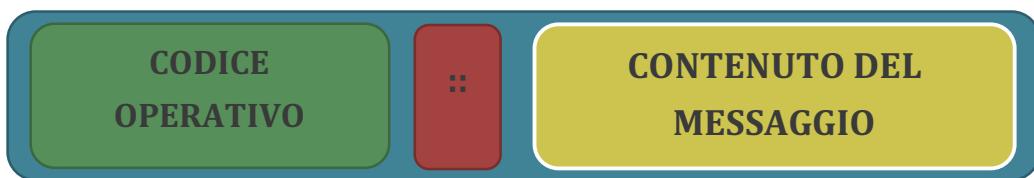
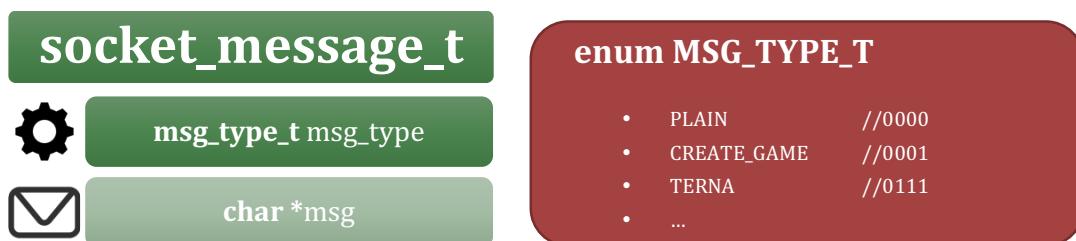


Figura 1 - Formato messaggio via socket

Il codice operativo è la rappresentazione testuale di un numero in codifica binaria naturale, che informa client e server su come interpretare la stringa successiva al separatore. Ad ogni codice è associata un'azione legata o alla logica di gioco vera e propria (come un ambo o un'estrazione di un numero) oppure ad azioni relative alla partita (come un messaggio in chat o che il match è finito).

Dalla stringa letta dal *socket* viene poi creata una struttura dati che rispecchia le proprietà del messaggio, ma maggiormente gestibile:

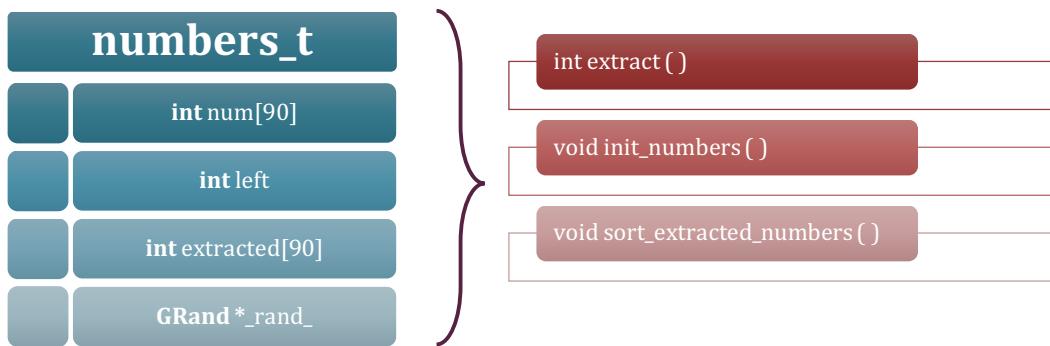


Di norma un messaggio viene inviato nel socket senza attendere che l'interlocutore l'abbia effettivamente letto, questo per evitare di bloccare il server sull'attesa di una notifica dal/i client oppure per non avere un pool di thread che attende risposte. Tuttavia per messaggi sufficientemente importanti, come un numero estratto o la fine della partita è stato implementato un sistema di comunicazione mutuato dall'architettura dei calcolatori: **Interrupt-Acknowledgement**. Il server invia il messaggio nel socket e attende che il client risponda affermativamente, segno che il messaggio è stato gestito.

# ESTRAZIONE

---

Un'ulteriore scelta di rilievo, visto il gioco in questione, era su come implementare l'estrazione di numeri. Sarebbe stata necessaria sia per l'estrazione propria della partita sia per la generazione dei numeri delle cartelle dei giocatori. Si è deciso di optare per la seguente struttura dati:



**Num[]** è un array inizializzato con interi da 1 a 90, mentre **left** indica quanti numeri rimangono da estrarre. La funzione **extract()** genera una posizione  $x$  *pseudo-casuale* tra 0 e *left*, l'elemento in posizione  $x$  viene settato a -1 e spinto in fondo all'array. In **extracted** viene salvato il numero estratto e il campo *left* viene aggiornato.

Il motivo di questo sistema di estrazione è per evitare attese più lunghe procedendo con estrazioni successive. Supponendo infatti estrazioni distribuite uniformemente il valore atteso dell'estrazione è  $\frac{\theta}{2}$  con  $\theta = 90$ . A lungo andare dunque la posizione generata si concentrerebbe in  $x = 45$ . Per non dovere effettuare controlli sul numero estratto, una volta scelto lo si elimina dalle possibili determinazioni restringendo l'intervallo delle posizioni estraibili.

In fase di progetto si è valutato anche l'utilizzo di un solo array, randomizzato, in cui l'estrazione corrispondesse a far avanzare un cursore lungo di esso. Tuttavia il campione generato risultava poco casuale su estrazioni di pochi numeri, il che era un problema per creazione delle cartelle di gioco.

# SERVER

Il server è pensato come un demone, pertanto non è dotato di interfaccia grafica. Si occupa di gestire nuovi client appena connessi, organizzarli in **stanze** (o *room*) e gestire la logica della partita.

## Main loop

Il programma pone le sue fondamenta su un **loop event-driven** messo a disposizione dalla libreria **GLib**. Il motivo di questa implementazione è avere il processo principale che non occupi la CPU tutto il tempo rimanendo in polling su ogni singola *sorgente*. Infatti, essendo il server pensato per gestire *teoricamente* un qualsiasi numero di partite di un qualsiasi numero di giocatori un sistema di polling sarebbe impensabile ed introdurrebbe

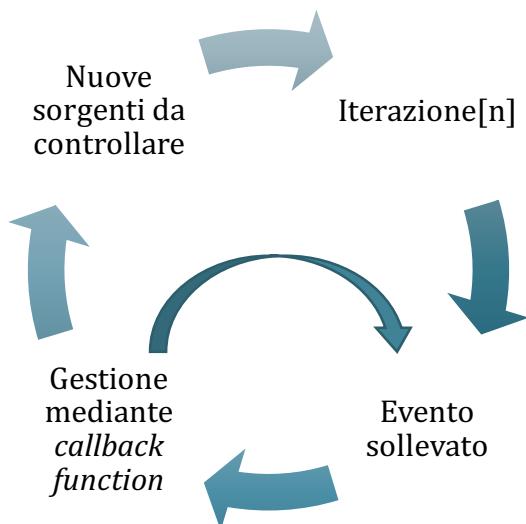


Figura 2 - Ciclo GMainLoop

ritardi frustranti crescenti col numero di giocatori. L'idea dunque è stata di avviare un *loop* (si veda **GMainLoop** dalla documentazione *GLib*) su cui fosse possibile registrare delle *sorgenti* (**GSource**) che avrebbero lanciato un evento qualora una particolare condizione si fosse verificata.

All'evento è associata una *funzione di callback* che lo

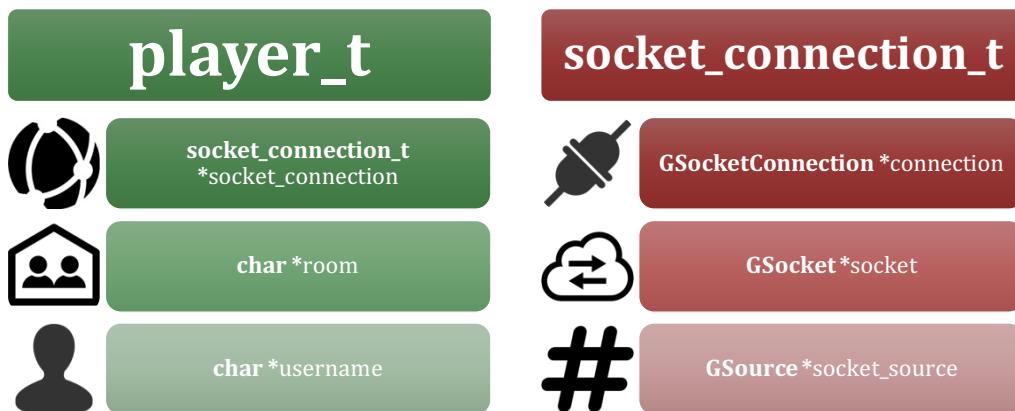
gestisce, cercando per quanto possibile di evitare l'utilizzo di funzioni bloccanti, piuttosto delegandole ad un thread apposito. Così facendo si massimizza il *throughput* di richieste al server gestite.

## Flusso operativo

Introdotto il ciclo principale, è possibile analizzare il flusso operativo che segue il server per condurre i client dalla connessione all'inizio della partita fino alla sua conclusione. Vale la pena ricordare che tutte le operazioni sono svolte in seguito ad una richiesta del client che avviene in modo asincrono ed è compito del server far sì che i client giochino una partita virtualmente in contemporanea.

## **Connessione di un client**

Qualora un client si connettesse al server in ascolto, viene sollevato un evento ***incoming connection***. La gestione dell'evento si occupa di creare una struttura dati che descriva le proprietà del nuovo giocatore connesso:



I campi di **socket\_connection\_t** sono immediatamente riempiti con i riferimenti agli oggetti che identificano il giocatore in qualità di client, al fine di poter dialogare facilmente attraverso il socket.

Il sistema di stanze è ispirato a quello del noto videogame **Worms** che fa del multiplayer tra amici una sua *core feature*. Un giocatore, **game master**, crea, dandole un nome, una nuova partita (o stanza) a cui i suoi amici possono unirsi inserendo il medesimo nome.

Attraverso il client, all'utente sarà possibile specificare sia il *nome utente* che il *nome della partita* da creare o a cui unirsi. Il server mantiene attraverso una lista concatenata tutte le stanze presenti così come una lista di *player* attualmente connessi e in attesa di iniziare a giocare. Sempre attraverso il client sarà possibile per il solo *game master* avviare la partita.

## **Avvio di una partita**

Al fine di permettere un qualsiasi numero di partite contemporanee, ogni partita è associata ad un *proprio thread* che si occuperà di gestire tutta la logica del gioco, mentre il processo padre continua a servire le connessioni entranti e la creazione di nuovi match. Questa fase è particolarmente delicata in quanto richiede di creare un'**istanza di gioco** che racchiude tutti i dati di una singola partita e di associarvi una lista di giocatori che appartengono a quella stanza.



La parte di pulizia è fondamentale in quanto d'ora in poi tutti i messaggi inviati nel socket saranno gestiti dal nuovo thread, ma ricordando della presenza di un loop ad eventi, la non rimozione della sorgente legata al socket genererebbe un doppio evento, uno nel processo padre e uno nel thread della partita. Ciò non è ammissibile in quanto si cadrebbe in una **race condition**: se il padre dovesse gestire il nuovo messaggio nel socket prima del thread figlio, quest'ultimo non troverebbe più alcun messaggio, e nessuna risposta verrebbe fornita.

### ***La partita***

Così come il processo padre, anche ciascuno dei thread figli che gestiscono una partita si basa su un loop ad eventi per la gestione della comunicazione client-server. Per analizzare il flusso d'esecuzione sono presentate le *feature* supportate dal programma server:

#### Estrazione

Grazie alla presenza di un loop ad eventi, è stato possibile creare una *sorgente* che sollevi un evento ogni x millisecondi in modo da lanciare con regolarità una *funzione di callback* che estragga un numero. Il numero è poi inviato ai **player**. La sorgente continua a scatenare un evento fino a quando la partita termina o fino a che tutti i numeri non sono stati estratti.

## Controllo di una vincita

Ad un certo istante al server arriva un messaggio da un client che comunica di aver ottenuto una sequenza vincente. Poiché nulla si può dire sull'onestà del giocatore, il programma server si occupa di controllare che i numeri comunicati dal client siano stati effettivamente estratti. Si supponga che si sia ricevuto un messaggio col seguente contenuto:

**1001::12,36,57,62,89**

Al codice **1001** è associato l'evento *cinquina* pertanto il server si aspetta che nel messaggio siano presenti 5 numeri. Viene dunque controllato che ciascuno di essi sia effettivamente già stato estratto. Se questo è il caso viene accreditata la vincita al client mediante un messaggio affermativo. In caso contrario la risposta è negativa.

## Fine partita

In accordo con il regolamento, una partita a Tombola termina nel momento in cui un giocatore effettua una tombola. Così come per un'altra vincita qualsiasi, viene accertata la validità della richiesta e, se affermativa, viene inviato un messaggio di fine gioco a tutti i partecipanti, con il nome del vincitore. Segue una fase di *clean-up* per ripulire la memoria dalle strutture dati utilizzate, arrestato il loop e ucciso il thread.

## Chat

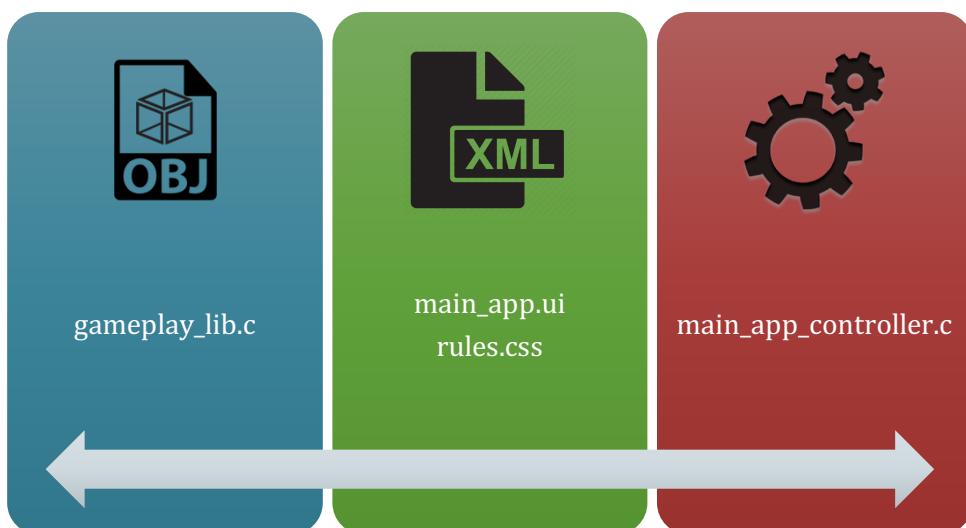
Un'ultima funzione di corredo è la possibilità di dialogare con gli avversari mediante una chat interna al client. Il server si occupa di mandare il messaggio scritto da un giocatore a tutti **gli altri**, comunicando il nome del mittente.

# CLIENT

---

## Design pattern

Per lo sviluppo del client con veste grafica, il toolkit adottato è **Gtk+3.0**. È stato scelto principalmente per due ragioni. In prima istanza per continuità con le librerie utilizzate lato server che fanno parte delle fondamenta su cui è stata sviluppata la libreria grafica. In seconda istanza perché consentivano uno sviluppo secondo il pattern **MVC: Model View Controller**.



Adottando questo *design pattern* è stato possibile separare la descrizione delle strutture dati legate alla logica del gioco e alla comunicazione client-server, dal disegno della UI e dalle risposte agli eventi relativi ad essa. Le prime sono già state analizzate in luogo del programma server, perciò qui si pone l'attenzione sulle altre due.

### *View*

Il toolkit Gtk+3.0 permette la descrizione della veste grafica mediante file **XML**. Ogni rapporto di parentela fra **GtkWidget** (la classe padre di tutti gli oggetti messi a disposizione per la realizzazione della UI) è sancito all'interno di una struttura ad albero in cui ogni nodo rappresenta un GtkWidget e di cui sono descritte le sue proprietà. Eccetto che per la radice (la finestra principale) ogni nodo ha un padre che *implementa* l'interfaccia

*GtkContainer*, ovvero funge da contenitore di altri *GtkWidget* riorganizzandoli secondo quanto specificato nelle proprietà.

Ciò che rende questo approccio pregevole è che una modifica all'interfaccia grafica non richiede una ricompilazione del codice. Infatti il programma si occupa di fare solamente una cosa: leggere il file XML che descrive l'interfaccia grafica e autonomamente vengono allocati in memoria gli oggetti corrispondenti con le proprietà desiderate, indipendentemente dal contenuto del file.

In ultima istanza, la libreria Gtk supporta l'applicazione di stili secondo il linguaggio **CSS**, anche in questo caso lo stile di ogni widget è caricato a runtime secondo quanto descritto nel file *rules.css*.

### **Controller**

Un'applicazione Gtk si fonda su un *loop ad eventi* analogo a quello utilizzato nel programma server. Ogni *GtkWidget* aggiunto viene registrato come sorgente ed è in grado di lanciare *segnali* quando una certa condizione si verifica. A titolo d'esempio si consideri un bottone. Qualora l'utente cliccasse su di esso un evento **clicked** verrebbe lanciato. Il *controller* si occupa proprio di gestire questi eventi, connettendo ad un segnale di un determinato widget una funzione di *callback* che serva l'evento.

### **Progetto dell'interfaccia grafica**

In fase di progetto della UI si è deciso di realizzare tutto il client in un'unica finestra, in linea con i trend del momento. Così come nelle applicazioni mobile e nei siti web l'utente si aspetta che il contenuto di interesse si mostri nella stessa finestra con cui sta interagendo piuttosto che esso sia presente in sottomenu o finestre alternative che vengono create per l'occasione. Pertanto si è optato per mostrare o nascondere delle *view* all'utente in funzione del punto nel flusso di esecuzione in cui si trova.

Seguendo una *best practice* appresa in un corso introduttivo sulla realizzazione dei videogame tutte le risorse grafiche (widget e immagini) sono caricate in memoria al momento dell'avvio rendendo così immediata qualunque transizione da una fase di gioco ad un'altra ed evitando attese *I/O bounded*.

Per realizzare questo progetto si è scelto un *container* speciale (GtkStack) che simulasse una pila di *view* e in cui fosse possibile scegliere quale *foglio*



Figura 3 - Stack delle view del client

della pila mostrare all'utente. In questo modo, con transizioni pulite e ridimensionamenti di finestra, il giocatore è guidato in tutte le fasi: dalla creazione della partita, al gioco, alla chiusura.

### Progetto logico

Il client è realizzato secondo un modello *multi-threaded* per poter gestire la componente grafica e del networking in modo autonomo e asincrono. Il motivo della scelta è dettato dalla volontà di mantenere l'interfaccia grafica *responsive* mentre altre azioni vengono svolte in background. Si consideri ad esempio l'estrazione di un nuovo numero da parte del server. Esso viene inviato al client che si deve occupare di mostrarlo all'utente. Tuttavia il messaggio deve prima essere interpretato per poter svolgere l'azione corretta. Questo processo è svolto dal thread dedicato al networking. Terminata la decodifica sarà aggiunta al thread della UI una routine da eseguire non appena possibile che aggiorni gli elementi grafici in accordo con l'evento atteso dal messaggio ricevuto dal server. Così facendo non si

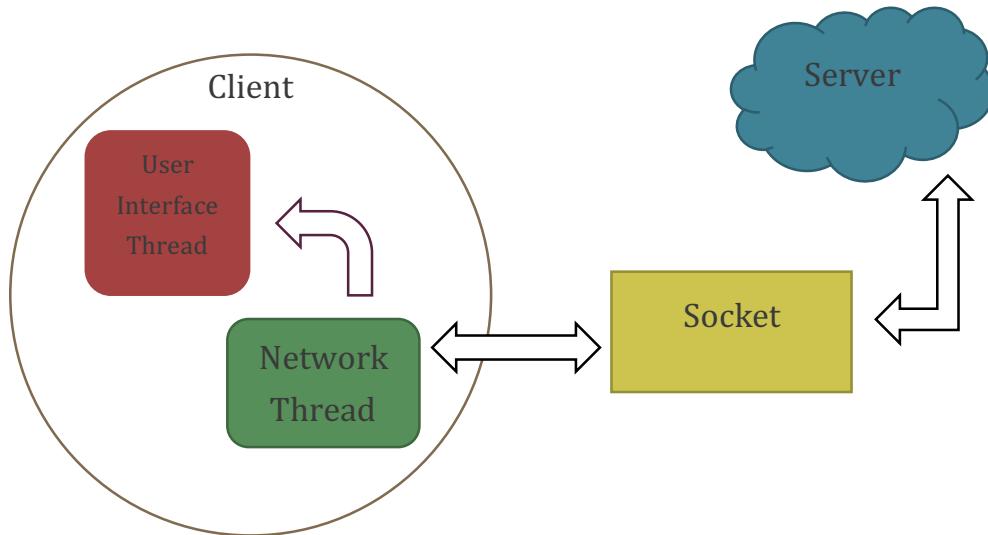


Figura 4 - Diagramma comunicazione client-server

genera alcuna *race condition* tra i due thread per il controllo dei widget. Tutto ciò che riguarda la UI è svolto dal thread relativo, il thread per il networking si occupa solo di informare l'altro sul da farsi.

### **Flusso operativo**

Anche per il client vengono di seguito descritte le azioni che si susseguono nel client.

#### ***Launcher***

All'avvio è presentata una sequenza di *view* che portino l'utente a poter avviare la partita in attraverso una fase di configurazione. Secondo il sistema a stanze descritto in precedenza, l'utente può scegliere se creare una nuova partita o unirsi ad una già presente. La disposizione della finestra è la medesima ma in funzione della scelta fatta il risultato sarà differente. Al giocatore è chiesto di inserire un nome utente che lo identifichi e il nome della partita (da creare o a cui unirsi). Seguirà una fase di attesa in cui il *game master* potrà decidere di avviare il match in qualsiasi momento scatenando la serie di eventi che sono stati descritti in luogo del server.

## Gameplay

La schermata di gioco si compone di due sezioni proprie della Tombola. Quella principale in cui l'utente ha a disposizione 3 cartelle di numeri con cui giocherà (generate casualmente), una serie di pulsanti che simulano le azioni possibili nel gioco, un riquadro che mostra gli eventi e un riquadro a scopo puramente goliardico che accompagna le azioni di gioco con immagini animate.



Figura 5 - Schermate di gioco

maggior chiarezza in questa sezione. Per saltare da una sezione all'altra è disponibile un selettore nella barra in testa alla finestra che con il meccanismo della pila descritto in precedenza mostra la view richiesta.

### Azioni di gioco

In qualsiasi momento della partita è possibile premere su uno dei pulsanti di azione per notificare una combinazione. Vista la presenza di righe e cartelle multiple, in seguito alla pressione di un bottone, è mostrato un dialogo che chiede al giocatore quale delle possibili sequenze inviare al server. Questa possibilità è pensata per consentire una strategia al *player* che potrebbe voler conservare una riga ed accedere a combinazioni maggiori.

La seconda sezione emula il tabellone delle estrazioni. Nonostante i numeri estratti vengano comunque comunicati nel riquadro degli eventi, la ricerca a partita inoltrata è pressoché impossibile.

Pertanto si è in grado di controllare le estrazioni con

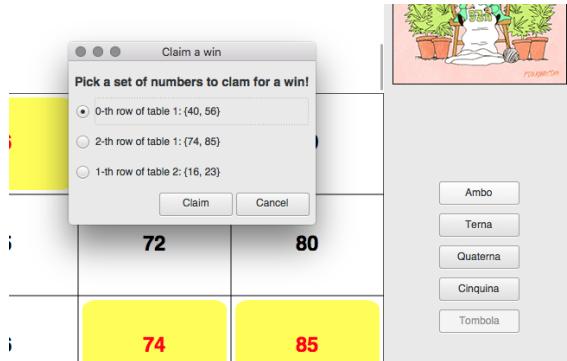


Figura 6 - Azione di gioco (Ambo)

inoltrato a tutti gli altri giocatori.

### Chat

Esterna alla logica del gioco, è disponibile anche una semplice sezione dell'interfaccia utente per interagire con gli avversari, mediante una chat testuale. L'utente può scrivere nel campo di testo in basso e il messaggio sarà

### **Fine della partita**

Qualora un giocatore dovesse effettuare una tombola, la partita termina e tutti i partecipanti vengono portati all'ultima schermata pensata per il client in cui viene mostrato il vincitore. Preso un utente, se è lui il vincitore gli sarà mostrata un'immagine animata di gioia, altrimenti una di delusione. Sarà per la prossima volta.

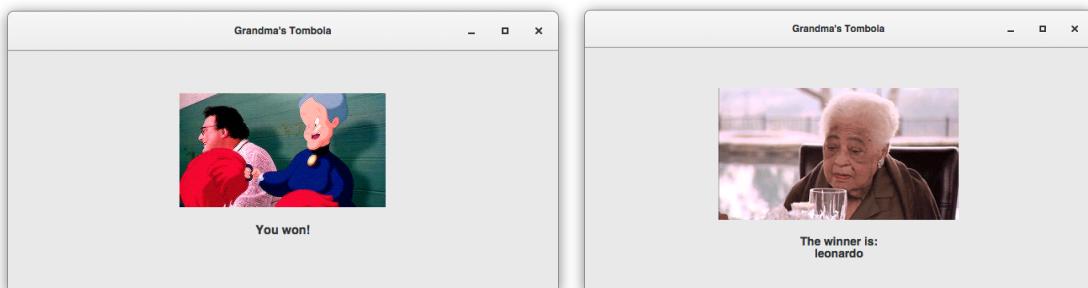


Figura 7 - Schermate di fine partita

# BUG NOTI E PROPOSTE DI MIGLIORAMENTO

---

Il progetto si è rivelato estremamente utile per entrare in contatto con problematiche *realistiche* che non erano note nelle prime fasi di sviluppo. Pertanto con l'esperienza accumulata nel corso del lavoro sono emersi alcuni bug o implementazioni troppo restrittive che si sarebbe potuto realizzare meglio.

## Saturazione del socket

Per quanto riguarda il networking, non è stato considerato il caso in cui il socket possa essere popolato da più messaggi consecutivi. Sarebbe la situazione in cui per qualche motivo (indisponibilità momentanea della rete o latenza tra i messaggi troppo bassa) l'evento “*nuovo messaggio presente nel socket*” non venga gestito immediatamente e il socket non venga svuotato prima dell'arrivo del nuovo messaggio.

La funzione di callback che gestisce l'evento si troverebbe a elaborare una stringa in un formato sconosciuto, mandando in crash l'applicazione.

Una possibile soluzione potrebbe essere estendere il formato della stringa introducendo un simbolo terminatore di stringa. Ottenuta dunque un insieme di messaggi dal socket, questi andrebbero aggiunti all'interno di una coda e serviti uno alla volta.

## Risoluzione su schermi diversi

Benché la risoluzione della finestra non sia forzata ad assumere un valore statico, permettendo così il resize pur mantenendo le proporzioni, a causa della dimensione delle risorse grafiche utilizzate la finestra potrebbe risultare troppo grande su schermi con risoluzioni minori di uno schermo HD senza possibilità di restringerla. Andrebbero caricate immagini di dimensioni variabili e scelte in funzione della risoluzione dello schermo.

## Rewarding / Banishment

Una possibile aggiunta al gioco sarebbe un sistema di crediti associati ad ogni giocatore. Di fatti al momento non vi sono reali motivazioni per effettuare un ambo piuttosto che una quaterna in quanto non portano a

nessun tipo di guadagno. Solo la tombola di fatto fa vincere la partita. Ciò che si potrebbe fare quindi è introdurre un costo di accesso ad ogni game che costituisce il montepremi. E l'utente si potrebbe aggiudicare una parte di esso comunicando una vincita. Allo stesso modo, al momento comunicare una sequenza di numeri errata non porta ad altro che un messaggio di avviso. Sarebbe interessante introdurre un sistema di banishment che disabilita ogni possibile azione del giocatore scorretto o distratto che invia una sequenza di numeri non ancora estratta.