Reti e Laboratorio di reti

9 CFU

BATTLESHIP

Progetto d'esame: Battaglia navale

Giuseppe Cianci Pio

Anno accademico 2016/2017

Docenti:

Teoria: prof. Umberto Scafuri

Lab.: prof. Alessio Ferone

Università degli Studi di Napoli "Parthenope"

RETI DI CALCOLATORI E LABORATORIO DI RETI

Studente: Giuseppe Cianci Pio Matr.: 0124001064 a.a 2016/2017

Progetto Reti di calcolatori

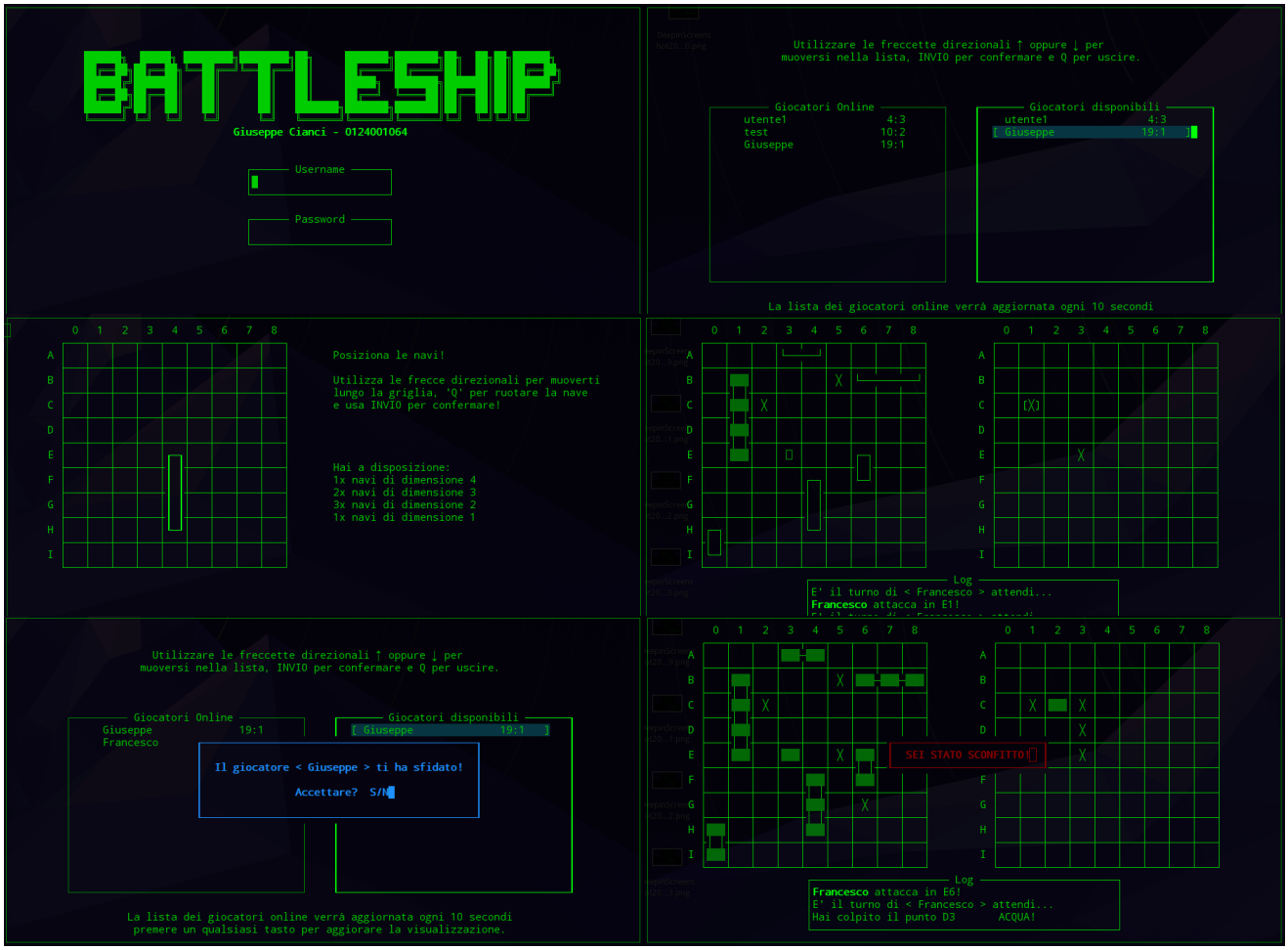
Giuseppe Cianci Pio - 0124001064

Indice

1 Traccia

Per il progetto è stata scelta la traccia #1:
"Si vuole realizzare una piattaforma di gioco on line per il gioco della battaglia navale. La piattaforma prevede la presenza di un server che consente di mettere in comunicazione gli utenti. Ogni utente è identificato da un ID univoco. Un utente, attraverso un client, si connette al server dal quale riceve la lista degli utenti attualmente connessi e la lista degli utenti non impegnati in una partita tra cui può sceglierne uno da sfidare. Una volta scelto lo sfidante il server mette in comunicazione i due utenti che giocano la partita. Al termine della partita l'esito viene comunicato al server che memorizza la statistica (partite vinte/giocate) di ogni utente. Il server consente lo svolgimento di più partite contemporaneamente, ma ogni ID può essere impegnato in una sola partita alla volta. La piattaforma prevede inoltre la possibilità per un admin di richiedere il resoconto delle partite giocate e la lista delle partite in corso."

2 Descrizione del progetto



2.1 Implementazione

L'intero progetto è stato sviluppando utilizzando il C++11 in modo tale da avere una gestione semplificata delle stringhe, delle strutture dati grazie alla STL e sfruttare i vantaggi dell OOP. Tuttavia la parte network del progetto è interamente realizzata in C seguendo le specifiche della traccia.

L'applicazione è stata dotata di un'interfaccia grafica intuitiva e più coinvolgente sia per la scelta dello sfidante che per il gioco stesso, ad esempio è possibile posizionare graficamente le navi sulla griglia o la visualizzare delle stesse durante una partita ecc...

2.2 Il Gioco

La battaglia navale è un gioco due giocatori, estremamente popolare e diffuso in tutto il mondo. Per giocare sono necessarie quattro tabelle (due per giocatore), tutte di uguali dimensioni. I quadretti della tabella sono identificate da coppie di coordinate, lettere e numeri, corrispondenti a riga e colonna. All'inizio, i giocatori devono "posizionare le proprie navi" segnandole su una delle loro due griglie. Una "nave" occupa un certo numero di quadretti adiacenti in linea retta (orizzontale o verticale) sulla tabella. Una volta posizionate le navi, il gioco procede a turni. Il giocatore di turno "spara un colpo" dichiarando un quadretto (per esempio, "B-5"). L'avversario controlla sulla propria griglia se quella cella è occupata da una sua nave. In caso affermativo risponde "colpito!", e marca quel quadretto sulla propria tabella; in caso negativo risponde "acqua" o "mancato". Sulla seconda tabella in dotazione i giocatori prendono nota dei colpi che hanno sparato e del loro esito. Quando un colpo centra l'ultimo quadretto di una nave non ancora affondata, il giocatore che subisce il colpo dovrà dichiarare "colpito e affondato!", e la nave si considera persa. Vince il giocatore che per primo affonda tutte le navi dell'avversario. Il turno del giocatore continua fin quando, non mancherà il bersaglio in seguito ad una mossa.

Il gioco è stato totalmente automatizzato, verificando automaticamente l'esito dei colpi, le navi colpite o quelle affondate comunicandone l'esito all'avversario, in modo da lasciare all'utente solamente il compito di "Giocare".

2.2.1 Gli utenti

Un utente non è altro che un giocatore registrato sulla piattaforma. Ogni utente è identificato tramite un username, e per effettuare il login al'interno del gioco necessita della corrispettiva password.

Una volta effettuato l'accesso, si passerà alla schermata di lobby dove potrà visualizzare gli utenti online e quelli disponibili a giocare. In modo tale da cercare uno sfidante o essere contattato da qualcuno per una partita che potrà essere accettata o meno.

La procedura di registrazione di un utente non è stata implementata, è possibile trovare una lista di quelli presenti nella sezione 6.3.2 .

2.2.2 L'admin

L'admin è stato implementato come un utente a tutti gli effetti, dispone dunque di un username e password ed è in grado di giocare, è al momento del login che il server verificherà i privilegi di quell'utente, comunicandoli al client in modo da diversificare la GUI.

Esso oltre a tutte le operazioni consentite agli utenti, ha i permessi aggiuntivi per effettuare richieste sugli utenti registrati, le partite in corso e lo storico delle partite giocate.

2.3 Entità principali

L'intero progetto si basa essenzialmente su due componenti principali:

- `MainServer` - costituisce il server principale di gioco il quale gestisce gli utenti, le loro statistiche, le partite in corso e permette ai client di iniziare una partita.
- `MainClient` - rappresenta il client tramite il quale l'utente si connette con il `MainServer`, esso contiene a sua volta due elementi:
  - `SubServer` - un server che parte in background, non gestito direttamente dall'utente che rimane in ascolto di connessioni in modo tale da permettere all'utente di accettare o meno richieste di sfida da parte di altri utenti.
  - `SubClient` - un client che, una volta ottenute le informazioni dal `MainServer`, si mette in contatto con il `SubServer` del giocatore sfidato al fine di giocare una partita.

3 Descrizione e schemi dell'architettura

Qui di seguito viene mostrato uno schema dove si illustra il funzionamento della piattaforma ed in particolare le relazioni fra le varie parti, client e e server.

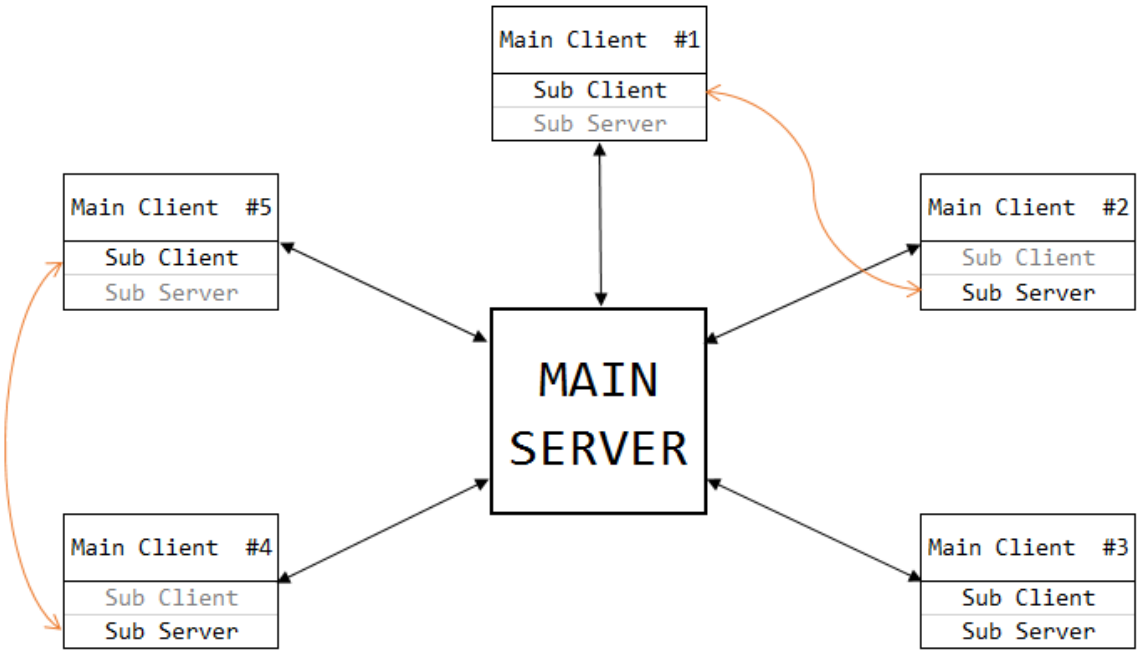


Figura 1: Architettura della piattaforma

In nero sono mostrate le connessioni fra i *MainClient* e il *MainServer*, mentre in arancione è raffigurata la connessione fra due *MainClient* che si instaura al momento della partita, il server principale è totalmente all'oscuro di questa connessione e pertanto non può intervenire su di essa.

3.1 Main Server - Server principale

Come già detto si tratta del server principale del gioco, esso è in grado di gestire più connessioni contemporaneamente e concorrentemente in modo da soddisfare le richieste di più giocatori insieme.

Egli comunica solamente con i *MainClient* e ne gestisce le loro richieste. Al momento della connessione con uno di essi il server principale resta in attesa che il client gli richieda un servizio tra quelli disponibili come ad esempio:

- Una richiesta di login;
- La lista degli utenti online;
- La lista degli utenti disponibili;
- La richiesta di sfidare un altro giocatore;

- etc...

Questi ed altri servizi sono eseguiti solamente quando richiesto dai client.

Dal momento in cui un utente effettua il login il server lo aggiunge alla lista degli utenti online e alla lista di quelli disponibili in questo modo potrà essere visualizzato e contattato da altri giocatori. Quando il *MainClient* si disconnette o la connessione cade l'utente viene automaticamente rimosso dalle suddette liste.

3.2 Main Client - Client principale

Rappresenta il client di gioco per l'utente tramite il quale può connettersi sia al server che ad altri giocatori per iniziare una partita. E' composto da un Client principale che comunica direttamente con il *MainServer* e da un *SubClient* e *SubServer* per permettere le partite fra giocatori.

Una volta avviato il *MainClient* verrà creata una connessione con il *MainServer* e mostrata la schermata di login, senza il quale non è possibile usufruire della piattaforma, in seguito viene mostrata una schermata con le liste dei giocatori online e disponibili dalla quale è possibile sceglier un giocatore da sfidare.

3.2.1 Sub Client - Client secondario

Si tratta di quella componente del *MainClient* che si occupa di connettersi agli altri utenti al fine di cominciare una nuova partita, le informazioni per la connessione sono ottenute attraverso una richiesta al *MainServer* che fornisce IP e porta.

Dal momento che l'utente sceglie di sfidare un giocatore si dichiara come host nella partita e pertanto il suo *SubServer* in ascolto viene temporaneamente bloccato.

3.2.2 Sub Server - Server secondario

Si tratta di quella componente del *MainClient* che eseguita in background all'avvio, ha il compito di rimanere in ascolto di eventuali connessioni da parte di giocatori che vogliono intraprendere una partita. Esso è costruito in modo da accettare una sola connessione per volta mostrando all'arrivo di una sfida una schermata di conferma all'utente sfidato.

4 Descrizione e schemi del protocollo applicazione

In questa sezione mostreremo i dettagli implementativi del protocollo applicazione ovvero come vengono svolte e gestite le varie operazioni e funzionalità delle varie entità all'interno della piattaforma.

4.1 Protocollo di connessione utilizzato

Il protocollo di comunicazione utilizzato per la piattaforma è il TCP tale scelta deriva dal fatto che si tratta di un protocollo affidabile, difatti i pacchetti inviati sono tracciati in modo che nessun dato venga perso o danneggiato durante il transito. Nel protocollo TPC abbiamo un flusso di byte continui raggruppati in segmenti e ricombinati dal destinatario.

E poiché l'intero funzionamento della piattaforma è basato sul paradigma richiesta-risposta ciò risulta molto importante.

In oltre, non trattandosi di un'applicazione puramente realtime non c'è la necessità di avere la massima efficienza possibile dunque l'overhead generato dal protocollo TCP rispetto al UDP risulta comunque accettabile.

4.2 Protocollo di comunicazione utilizzato

Al fine di mettere in comunicazione i vari server e client si è optato per l'utilizzo di invio e recezione pacchetti. In questo modo l'implementazione delle varie funzionalità è risultata più semplice e veloce.

In generale ogni richiesta è associata ad una coppia di pacchetti: il pacchetto della richiesta e quello della risposta, in questo modo sia il client che il server sanno con esattezza il numero di byte da leggere e inviare.

4.2.1 Struttura di un pacchetto

La struttura di ogni pacchetto è molto simile, ognuno di essi è composto da due parti, l' *header* che ne indica il tipo di servizio richiesto o il tipo di informazione che sta trasmettendo e una parte *data* contenente le informazioni e i dati necessari al completamento della richiesta oppure richiesti.
Nella figura 2 è presente un esempio di come è strutturato un pacchetto, in questo caso quello utilizzato per effettuare una richiesta di login al *MainServer*.

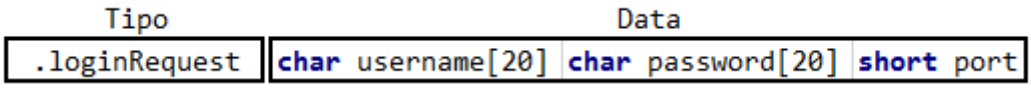


Figura 2: Esempio di pacchetto

Ogni pacchetto è stato implementato come una classe derivata da quella più generale `Package`, poichè il C++ non mette direttamente a disposizione funzioni per l'invio e recezione di oggetti tramite socket, per essere trasmesso risulta necessario di implementare i metodi:

- `const string serialize();` tale metodo permette di serializzare l'oggetto trasformandolo in una stringa in modo tale da poterlo inviare tramite una socket.
- `T deserialize(const char*);` tale metodo permette di deserializzare una stringa ricevuta tramite socket in modo da poterla trasformare di nuovo in pacchetto e poterne leggere le informazioni contenute. `T` in questo caso rappresenta il tipo del pacchetto di ritorno.
- `static size_t pkgSize()const;` tale metodo statico ritorna la dimensione in byte del paccetto, esso consente in qualsiasi momento di conoscere le dimensioni di un pacchetto di quella classe.

Il tipo del pacchetto è definito da un semplice enumeratore all'interno superclasse `Package` in cui ogni elemento identifica un particolare pacchetto.

In questo modo lo scambio di informazioni si riduce al riempimento dell'oggetto e al suo invio oppure alla sua ricreazione partendo da una stringa ricevuta.

4.2.2 Invio di una richiesta

Utilizzando questo tipo di protocollo a pacchetti è possibile effettuare delle richieste in maniera molto semplice: Il client invia inizialmente solo il tipo, ovvero l'enumeratore che identifica univocamente la richiesta, poi se necessario ai fini della richiesta in questione invia il pacchetto con le relative informazioni.

In fine si mette in attesa di una risposta del server.

> Esempio: Richiesta di login
> Il client per mandare una richiesta di login al server prepara prima il pacchetto (`LoginRequestPkg`) riempiendolo con le informazioni necessarie. Fatto ciò invia al server il tipo di richiesta (`.loginRequest`) e subito dopo il pacchetto in se. in questo modo il server sa che è in arrivo una richiesta di login con il relativo pacchetto.

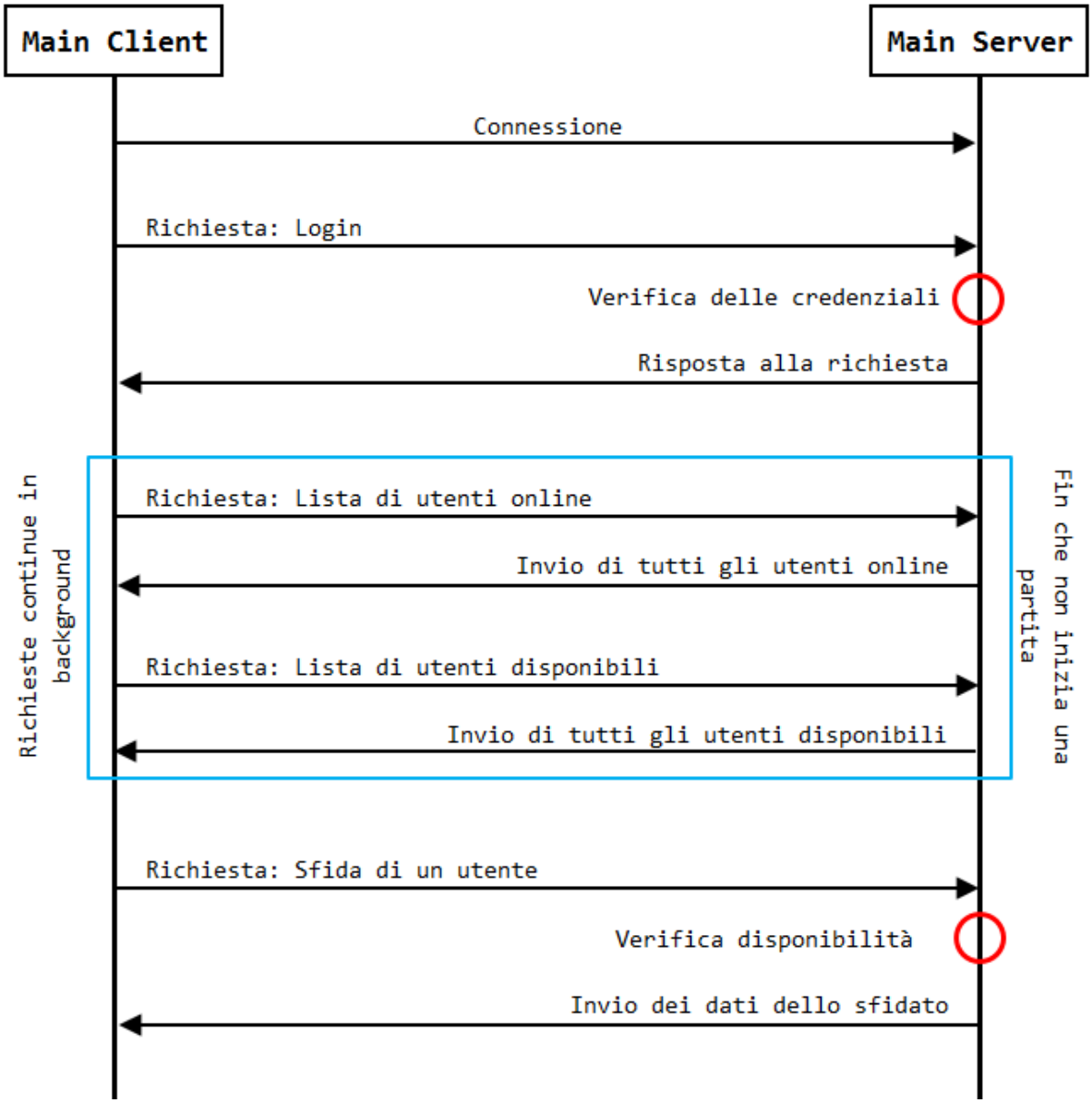
4.2.3 Attesa di una risposta

Una volta inviata la richiesta e l'eventuale pacchetto di informazioni il client si mette in attesa di una risposta del server. Quest'ultimo invia subito un codice che corrisponde allo status della richiesta. Anch'esso è definito tramite un enumeratore e può assumere i seguenti valori `success, error, unauthorized` tale status riguarda esclusivamente la richiesta e non il risultato della stessa!

Così facendo è possibile sapere se la richiesta è andata a buon fine e dunque attendere una risposa con il risultato della stessa oppure bloccarsi in quanto c'è stato un problema.

4.3 Funzionalità del Main Client

In questa sezione verranno mostrate le principali funzionalità del *MainClient* in maniera molto generale attraverso una sua tipica esecuzione descritta nello schema sottostante. Di seguito ciascuna operazione verrà approfondita e vista più nel dettaglio.



1. La prima operazione effettuata dal *MainClient* è la connessione con il *MainServer* tramite la quale si instaura il canale di comunicazione.
2. Viene mostrata dunque la schermata di login da cui è possibile inserire le proprie credenziali ed infine inviarle al server per effettuare l'accesso. Fatto ciò è possibile procedere.

3. Vengono richieste in background e a intervalli di tempo regolari (10 sec.) le liste degli utenti online e utenti disponibili. Si accede dunque alla schermata del lobby, dove è possibile visualizzare le liste e scegliere un giocatore da sfidare fra quelli disponibili.
4. Scelto il giocatore, viene effettuata una richiesta al *MainServer* che, verificata la sua disponibilità, ci risponderà con i dati di quest'ultimo in modo da poterlo contattare.

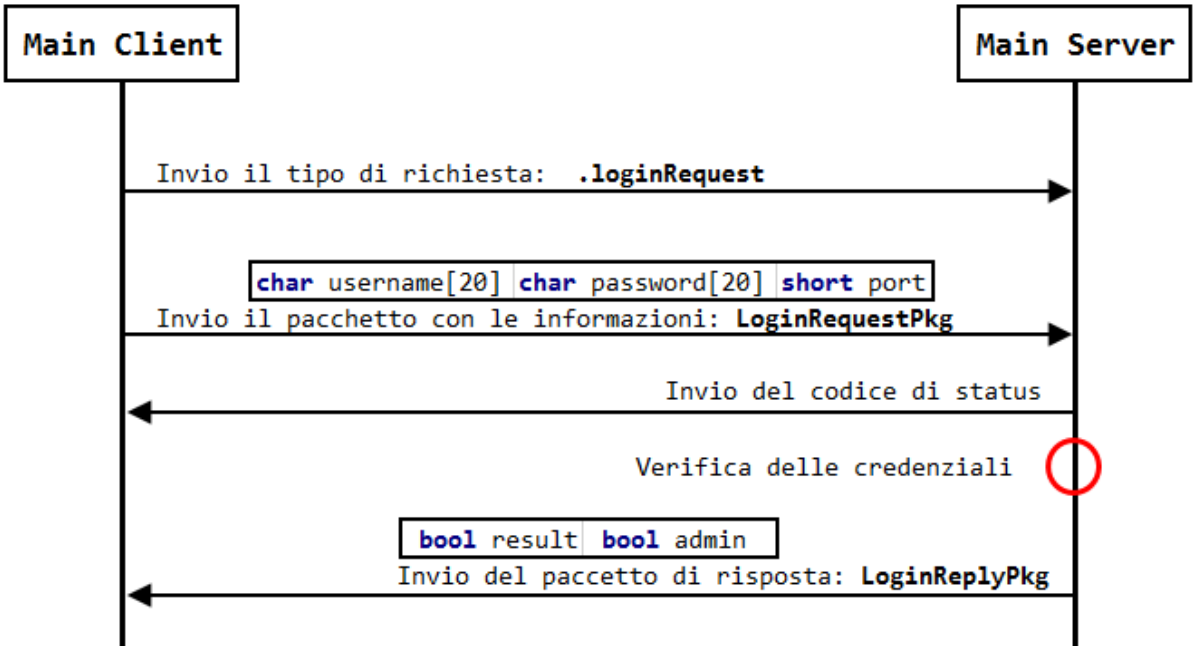
   Da qui in poi il *MainClient* resta in attesa e "passa il controllo" al *SubClient* che effettuerà la connessione con l'utente sfidato.

4.3.1 Richiesta di login

La richiesta di login, dopo la connessione, è la prima richiesta effettuata dal *MainClient* al *MainServer*, essa è fondamentale per eseguire tutte le altre operazioni.

```
int loginRequest(std::string username, std::string password, short port);
```

- `Scopo`: Invia una richiesta al *MainServer* con le credenziali inserite dall'utente per effettuare il login all'interno dell piattaforma di gioco.
- `Argomenti`: *username, password* dell'utente, necessari per l'autenticazione e la *porta* tramite la quale il *SubServer* dell'utente si metterà in ascolto per essere contattato da altri giocatori.
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.
- `Conseguenze`: Aggiorna gli attributi `logged` e `admin` del *MainClient* in modo tale da sapere quando l'utente ha effettuato l'accesso e se dispone o meno dell'autorità di amministratore.



1. Il *MainClient* crea un nuovo pacchetto di tipo `LoginRequestPkg` utilizzando le credenziali fornite al momento del login da parte dell'utente e la porta usata dal suo *SubServer*.
2. Viene poi inviata la richiesta `.loginRequest` insieme al pacchetto precedentemente creato.
3. Si resta in attesa del codice di status. Se ques'ultimo è positivo si attende l'arrivo del pacchetto di risposta altrimenti la funzione si interrompe.
4. Il pacchetto di risposta è di tipo `LoginReplyPkg` e contiene le infrazioni necessarie a far sapere al *MainClient* se l'autenticazione è andata a buon fine e se quell'utente è un amministratore.

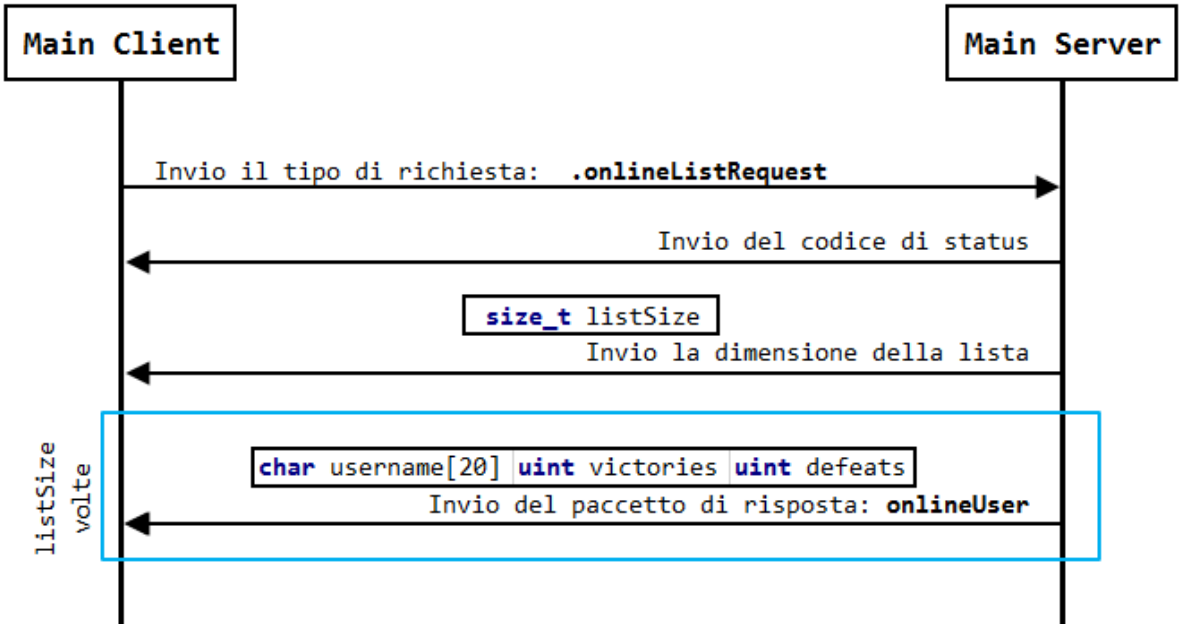
4.3.2 Richiesta delle liste degli utenti online/disponibili

Si tratta di una richiesta al *MainServer* da parte del *MainClient* della lista di utenti attualmente online o della lista degli utenti attualmente disponibili a giocare una partita. Essendo le du funzioni molto simili, sono state trattate insieme.

```
int onlineListRequest(); int freeListRequest();
```

- `Scopo`: Invia una richiesta al *MainServer* richiedendo la lista degli utenti online/-disponibili;
- `Argomenti`: -
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.

- `Conseguenze`: Aggiorna l'attributo `onlineList/freeList` del MainClient.



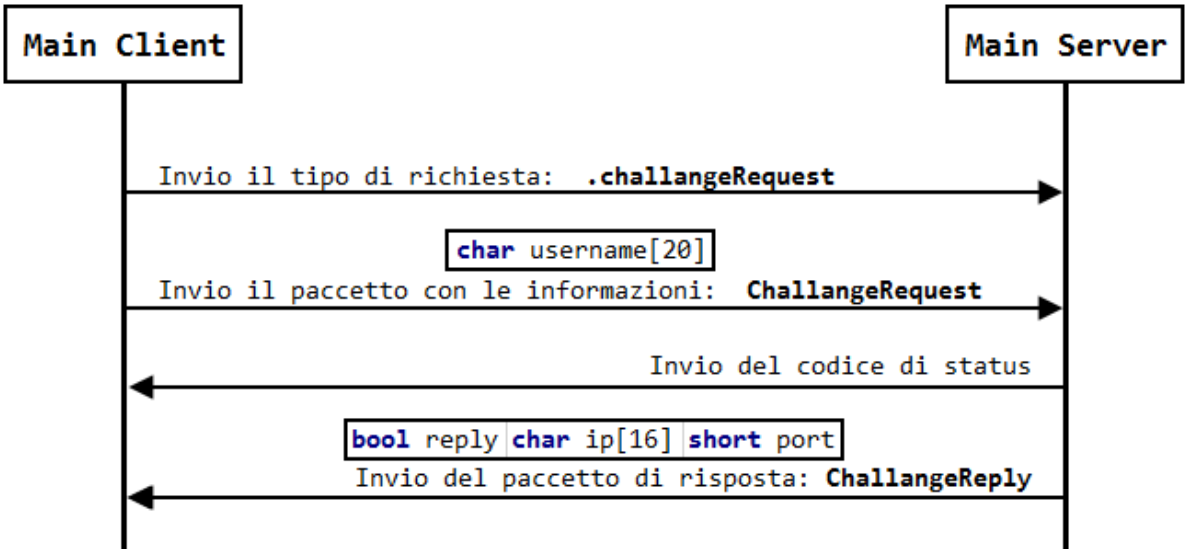
1. Viene inviata la richiesta `.onlineListRequest`/`.freeListRequest`.
2. Si resta in attesa del codice di status. Se ques'ultimo è negativo la funzione si interrompe.
3. Si resta in attesa del `listSize` ovvero il numero di utenti presenti nella suddetta lista.
4. In fine il *MainClient* attande l'arrivo di `listSize` pacchetti `onlineUser` ognuno dei quali è scomposto ottenendo le informazioni necessarie e inserito nella lista.

4.3.3 Richiesta delle informazioni di uno sfidante

Si tratta di una richiesta al *MainServer* da parte del *MainClient* circa le informazione di uno specifico utente presente nella lista di utenti disponibili al fine di poterlo contattare proponendogli una sfida.

```
int challangerRequest(string username, struct sockaddr_in *servaddr);
```

- `Scopo`: Invia una richiesta al *MainServer* richiedendo IP e porta del *SubServer* di un utente disponibile in modo da poterlo contattare.
- `Argomenti`: *username* dell'utente sfidato, *servaddr* dove verranno salvati IP e porta se la funzione va a buon fine.
- `Errori`: Restituisce `1` in caso di successo e ricevuta dei dati del giocatore, `0` in caso di successo ma senza ricevuta dei dati del giocatore e un numero negativo `<0` in caso di fallimento.



1. Viene inviata la richiesta `.challangeRequest` e in seguito il relativo pacchetto di tipo `ChallangeRequest`.
2. Si resta in attesa del codice di status. Se ques'ultimo è negativo la funzione si interrompe.
3. Dunque il *MainClient* attande l'arrivo di un pacchetto di tipo `ChallangeReply` dal quale è possibile sapere se la richiesta è andata a buon fine e le informazioni richieste.

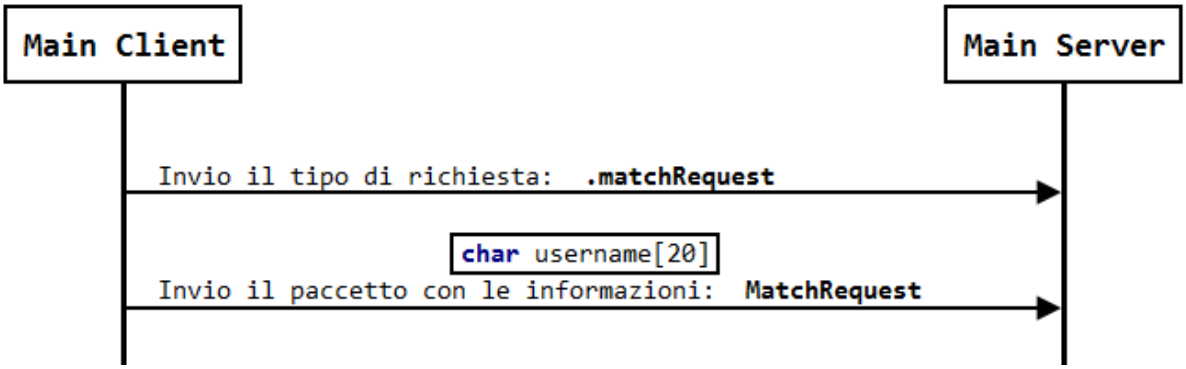
4.3.4 Richiesta di impegno in una partita

Si tratta di una richiesta al *MainServer* la quale notifica che la sfida richiesta dall'utente è stata accettata dallo sfidato, in modo che il *MainServer* possa rimuoverli dalla lista dei giocatori disponibili.

Tale operazione è effettuata solo dal *MainClient* dello sfidante.

```
int matchRequest(string foeUsername);
```

- `Scopo`: Invia una richiesta al *MainServer* informandolo di essere impegnato in una partita.
- `Argomenti`: *foeUsername* username dell'utente con il quale si è impegnati a giocare.
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.



1. Viene inviata la richiesta `.matchRequest`.
2. Viene in fine inviato il pacchetto di tipo `MatchRequest` contenenti le informazioni necessarie.

4.3.5 Notifica di Vittoria/Sconfitta

Si tratta di una richiesta al *MainServer* che notifica l'avvenuta vittoria o sconfitta dopo una partita in modo che le statistiche siano aggiornate.

```
int MainClient::sendNotify(Package::Type notify);
```

- `Scopo`: Invia una richiesta al *MainServer* informandolo dell'esito della partita.
- `Argomenti`: *notify* indicante l'avvenuta vittoria o sconfitta.
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.

4.3.6 Richiesta dello storico dei match terminati/in corso

Il *MainClient* invia una richiesta circa la lista dei match terminati/in corso. Tale richiesta è effettuabile solamente dall'amministratore.

```
int terminatedMatchesRequest();
int inProgressMatchesRequest();
```

- `Scopo`: Richiede al *MainServer* la lista dei match terminati/in corso
- `Argomenti`: -
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.

Tali funzioni sono molto simili a quelle per richiedere la lista degli utenti online e disponibili, con la differenza che il tipo di pacchetto utilizzato è `matchPkg`, pertanto non verranno mostrate graficamente.

4.3.7 Richiesta della lista di tutti gli utenti registrati

Il *MainClient* invia una richiesta circa la lista degli utenti registrati. Tale richiesta è effettuabile solamente dall'amministratore.

```
int registeredUsersRequest();
```

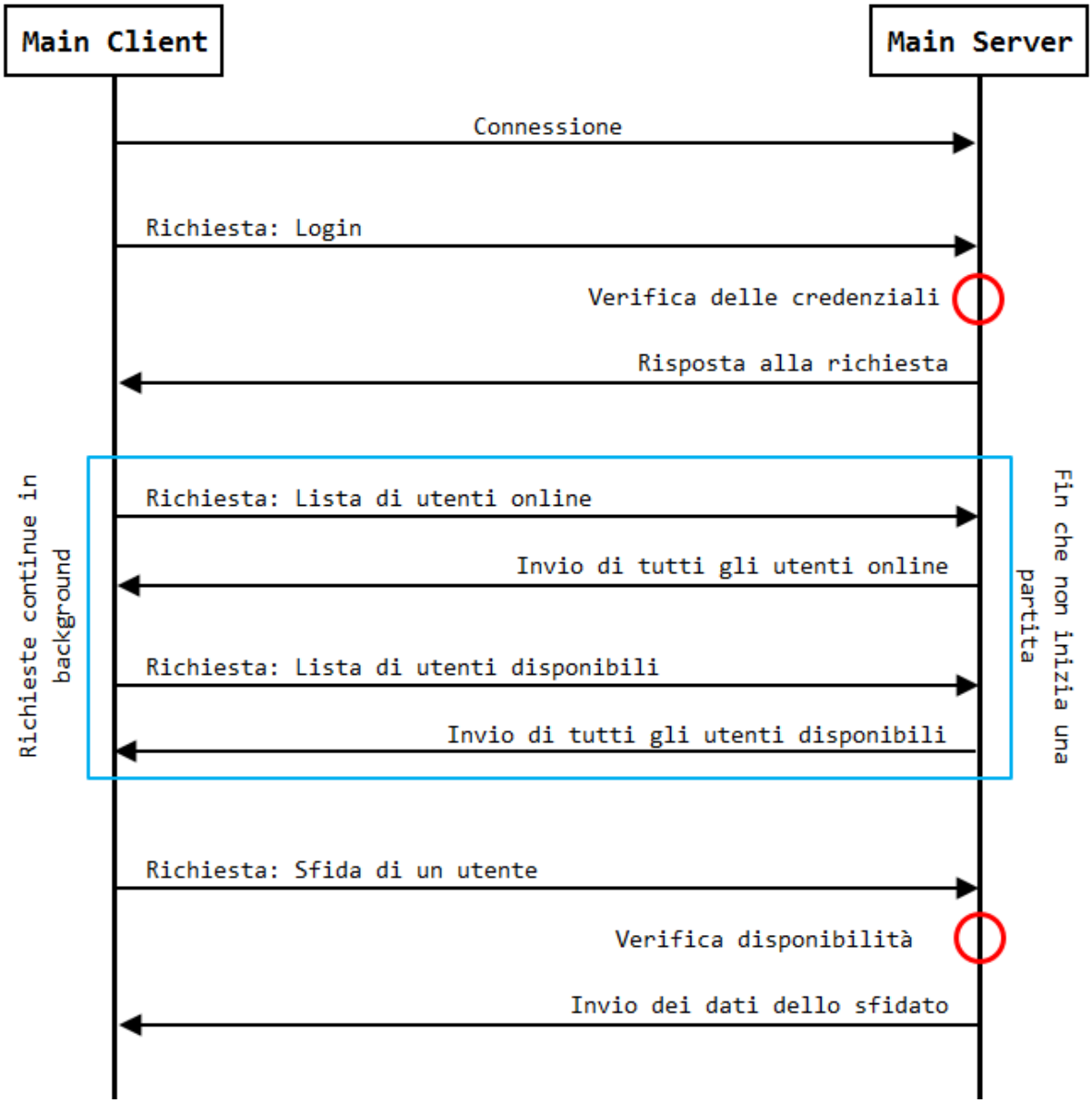
- `Scopo`: Richiede al *MainServer* la lista degli utenti registrati.
- `Argomenti`: -
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.

Questa funzione è molto simile a quelle per richiedere la lista degli utenti online e disponibili, pertanto non verrà mostrata graficamente.

4.4 Funzionalità del Main Server

In questa sezione verranno mostrate le principali funzionalità del *MainServer* in maniera generale attraverso una sua tipica esecuzione descritta nello schema sottostante. Di seguito ciascuna operazione verrà approfondita e vista più nel dettaglio.

Il *MainServer* è progettato per gestire tutte le connessioni in maniera concorrente pertanto ciò che viene illustrato di seguito e da ritenersi eseguito parallelamente da più client.



1. Una volta fatto partire, il *MainServer* resta in attesa di essere contattato da un *MainClient* e di instaurare dunque una connessione.
2. Da qui in poi il server è a completa disposizione del *MainClient*, resta in attesa di essere contattato, e può gestirne le richieste in qualsiasi ordine.
3. Per poter esaudire le richieste è comunque necessario per l'utente effettuare l'accesso pertanto la prima richiesta mandata ad ogni connessione è quella di login.
4. Effettuato il login verranno inviate le liste degli utenti online e utenti disponibili, il server risponde semplicemente alla richiesta, non è pertanto a conoscenza dell'intervallo di tempo fra una richiesta e l'altra della lista.
5. una volta che l'utente ha scelto lo sfidante invia la richesta al *MainServer* che

risponde con i rispettivi dati.

6. se l'altro giocatore accetta viene inviato al *MainServer* una richiesta di impegno alla partita, dunque i giocatori interessati vengono rimossi dalla lista dei giocatori disponibili.
7. Al termine della partita i giocatori inviano il risultato al server che aggiorna le liste e le statistiche dei giocatori.

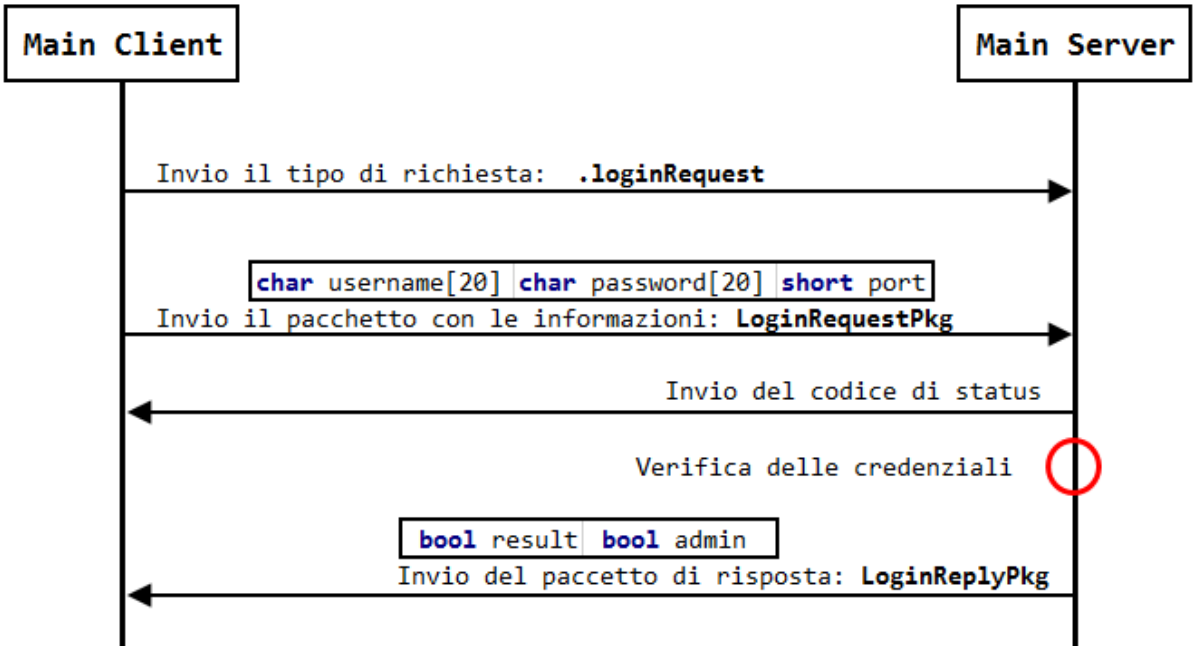
E' importante notare che il *MainServer* non ha un andamento sequenziale durante l'esecuzione delle varie operazioni, il loro ordine dipende esclusivamente dagli utenti.

4.4.1 Risposta al Login

La risposta al login consiste nel validare le credenziali inviate dal *MainClient*, e in caso affermativo effettuare il login dell'utente inserendolo opportunamente nelle varie liste. In questa fase si verifica in oltre se l'utente sia o meno un amministratore.

```
string loginReply(int _connfd, struct sockaddr_in clientaddr);
```

- `Scopo`: Invia una risposta al *MainClient* con il risultato del login e della verifica dei privilegi di amministratore.
- `Argomenti`: *_connfd* il descrittore della socket sulla quale è connesso il MainClient e, *clientaddr* che contiene alcune informazioni necessarie all'inserimento del giocatore loggato nelle varie liste.
- `Errori`: Restituisce l'username del giocatore in caso di successo e una stringa vuota `""` altrimenti.
- `Conseguenze`: In caso di successo, va ad aggiornare gli attributi `onlineList`/`freeList` del MainServer aggiungendo il giocatore.



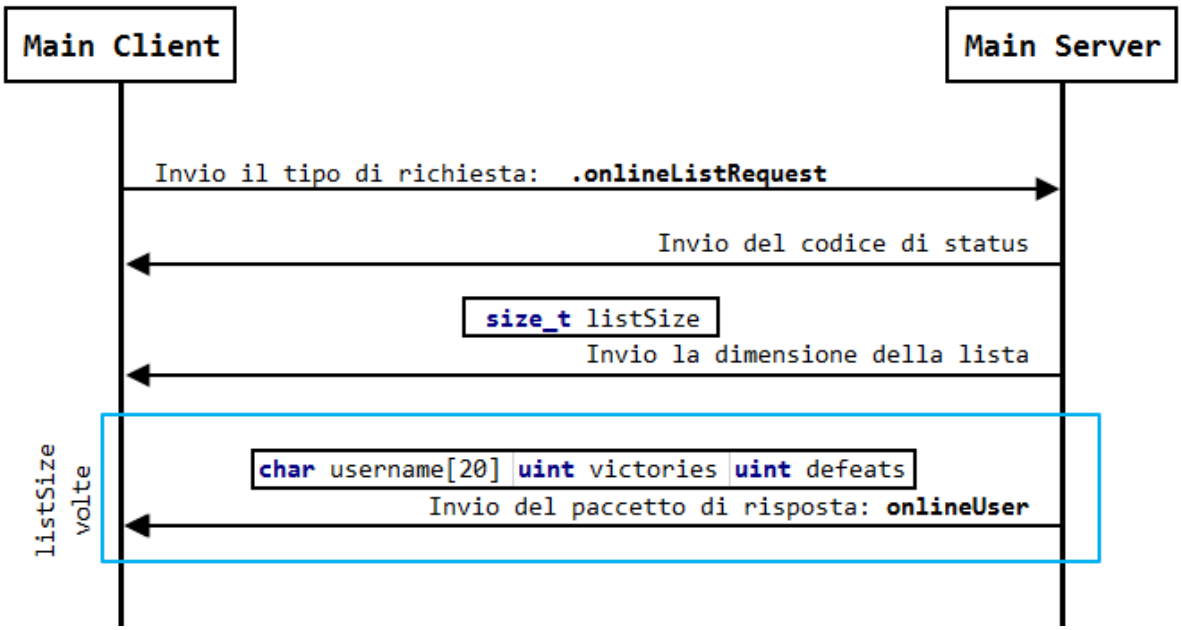
1. Ricevuta la richiesta e il pacchetto relativo dal *MainClient* il server invia il codice di status. Se quest'ultimo è negativo la funzione si interrompe generando un errore.
2. Vengono dunque verificate le credenziali dell'utente, che esista, non sia già online, che username e password corrispondano e se sia o meno un admin.
3. il risultato di tale verifica viene mandato al *MainClient* creando un pacchetto `LoginReplayPkg`

4.4.2 Invio delle liste degli utenti online/disponibili

Il *MainServer* invia la lista degli utenti online/disponibili inviandone un elemento per volta. Essendo le due funzioni molto simili, sono state trattate insieme.

```
int onlineListReply(int _connfd); int freeListReply(int _connfd);
```

- `Scopo`: Invia in risposta al *MainClient* la lista degli utenti online/disponibili.
- `Argomenti`: *_connfd* il descrittore della socket sulla quale è connesso il *MainClient*.
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.



1. Ricevuta la richiesta e il pacchetto relativo dal *MainClient* il server invia il codice di status. Se quest'ultimo è negativo la funzione si interrompe generando un errore.
2. Viene inviato `listSize` ovvero il numero di utenti presenti nella lista richiesta.
3. In fine vengono inviati `listSize` pacchetti `onlineUser` contenenti gli utenti e alcune loro informazioni.

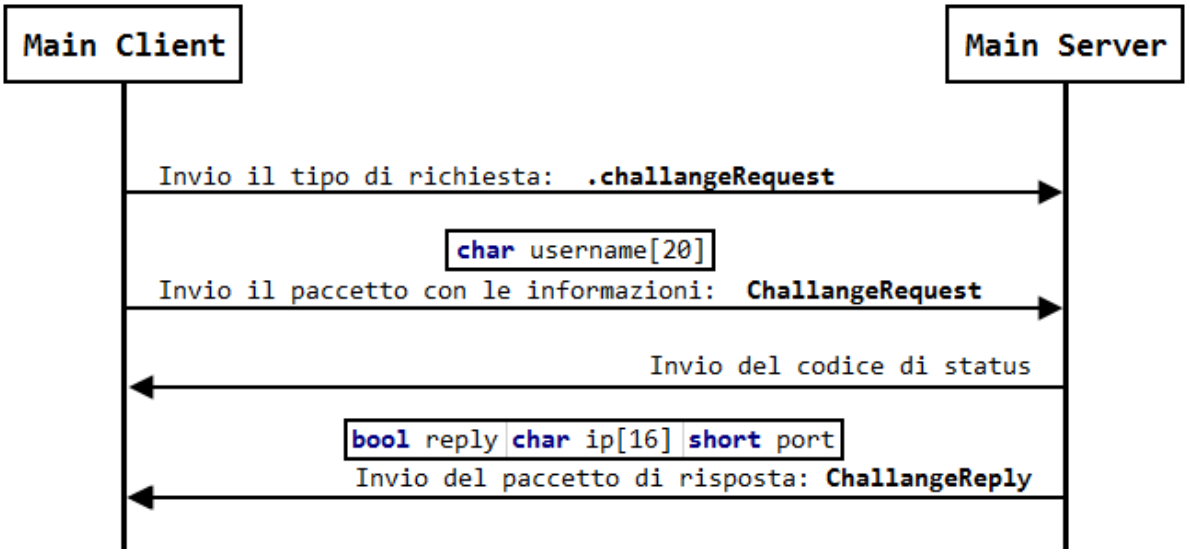
4.4.3 Invio delle informazioni di un giocatore

Il *MainServer* invia all'utente richiedente le informazioni necessarie per consentirgli di mettersi in contatto con il giocatore sfidato, ovvero l'IP e la porta usati dal suo *SubServer*.

Prima di fare ciò viene verificato che l'username sfidato esista, che sia online e disponibile a giocare. L'esito di questo controllo viene inviato insieme agli eventuali dati.

```
int challangeReplay(int _connfd);
```

- `Scopo`: Invia in risposta al *MainClient* l'esito della richiesta di sfida di un giocatore.
- `Argomenti`: *_connfd* il descrittore della socket sulla quale è connesso il *MainClient*.
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.



1. Ricevuta la richiesta e il pacchetto relativo del *MainClient* il server invia il codice di status. Se quest'ultimo è negativo la funzione si interrompe generando un errore.
2. Si verifica che l'utente scelto per la sfida sia disponibile effettivamente e il risultato di tale verifica, insieme eventualmente le sue informazione, viene inviato in risposta attraverso il pacchetto `ChallangeReplay`

4.4.4 Invio dello storico dei match terminati/in corso

Il *MainServer* invia all'utente richiedente lo storico dei match terminati/in corso, poiché tale richiesta è effettuabile solamente dall'admin viene effettuato anche un controllo sui privilegi dell'utente richiedente.

Un utente non admin che effettua la richiesta riceverebbe in risposta alla richiesta il codice status `unauthorized` indicante che non ha permessi necessari per quella richiesta.

```
int sendTerminatedMatches(int _connfd, string connectedUsername);
int sendInProgressMatches(int _connfd, string connectedUsername);
```

- `Scopo`: Invia in risposta al MainClient lo storico dei match disputati/in corso.
- `Argomenti`: *_connfd* il descrittore della socket sulla quale è connesso il *MainClient* e *connectedUsername* l'username dell'utente che effettua la richiesta.
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.

Tali funzioni sono molto simili a quelle per inviare la lista degli utenti online e disponibili, con la differenza che il tipo di pacchetto utilizzato è `matchPkg`, pertanto non verranno mostrate graficamente.

4.4.5 Invio della lista di tutti gli utenti registrati

Il *MainServer* invia all'utente richiedente la lista di tutti gli utenti registrati sulla piattaforma, poiché tale richiesta è effettuabile solamente dall'admin viene effettuato anche un controllo sui privilegi dell'utente richiedente.

Un utente non admin che effettua la richiesta riceverebbe in risposta alla richiesta il codice status `unauthorized` indicante che non ha permessi necessari per quella richiesta.

```
int sendRegistredUsers(int _connfd, string connectedUsername);
```

- `Scopo`: Invia in risposta al *MainClient* la lista degli utenti registrati.
- `Argomenti`: *_connfd* il descrittore della socket sulla quale è connesso il *MainClient* e *connectedUsername* l'username dell'utente che effettua la richiesta.
- `Errori`: Restituisce `0` in caso di successo e un numero negativo $<$ `0` altrimenti.

Questa funzione è molto simile a quelle per richiedere la lista degli utenti online e disponibili, pertanto non verrà mostrata graficamente.

4.4.6 Gestione delle liste e degli utenti e loro statistiche

Il *MainServer* in oltre si occupa anche di gestire gli utenti della piattaforma, le loro partire e i loro match. In totale esso mantiene quattro liste:

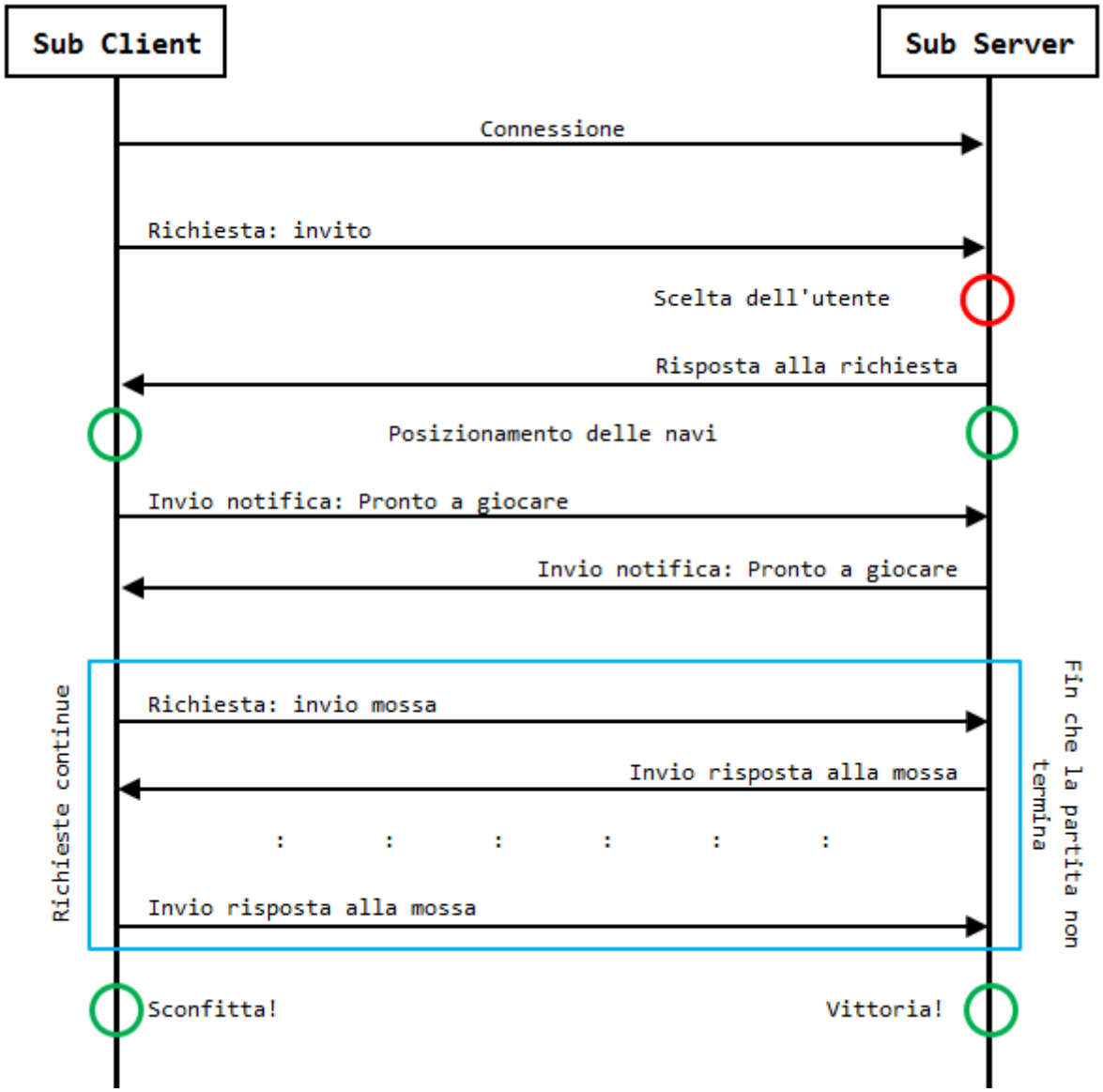
- La lista degli utenti registrati;
- La lista degli utenti online;
- La lista degli utenti disponibili;
- La lista dei match in corso.

Tutte queste liste sono costantemente aggiornate e si evolvono in base alle scelte e azioni dei vari utenti connessi al *MainServer*.

4.5 Funzionalità del Sub Client e del Sub Server

Il *SubClient* e il *SubServer* hanno essenzialmente le stesse funzionalità, l'unico elemento a cambiare è appunto il loro essere client o server. Nel caso in cui un utente inviti un'altro giocatore ad una partita il suo *SubClient* invierà una richiesta di connessione al *SubServer* del giocatore sfidato in modo tale da iniziare una partita. Il giocatore sfidato potrà a scegliere se giocare o rifiutare l'offerta portando dunque a termine la connessione.

Quindi per come è stato definito il protocollo lo sfidante giocherà sempre attraverso il *SubClient* mentre lo sfidato sempre attraverso il *SubServer*.



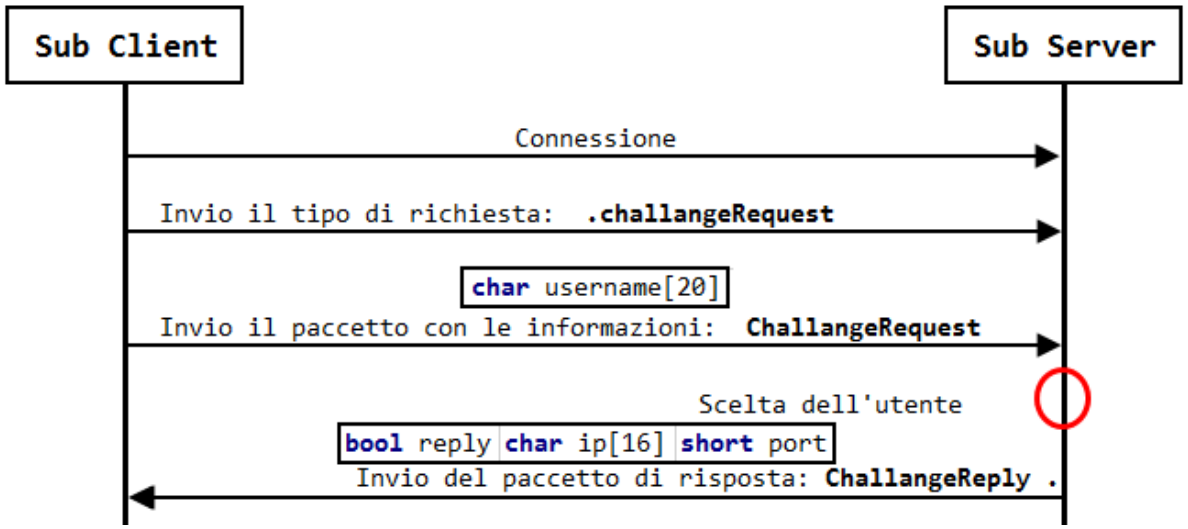
1. Una volta scelto il giocatore da sfidare e richiesti i suoi dati al *MainServer*, Il *SubClient* può effettuare la connect() con *SubServer* avversario.
2. Il *SubClient* invia dunque un invito a giocare e attende una risposta.
3. Se la risposta è affermativa, la partita comincia e dunque entrambi i giocatori iniziano a posizionare le navi. Fatto ciò, scambievolmente, notificano che sono pronti a giocare e attendono l'avversario.
4. Quando entrambi hanno posizionato le navi e sono pronti inizia la partita vera e propria in cui le due parti inviano le loro messe attendendone il risalutato.
5. Alla fine quando tutte le navi sono state abbattute la partita termina.

Del pacchetto di risposta `ChallangeReply` interessa soltanto il campo `reply` con-

tenente la risposta alla sfida proposta. Gli altri campi non sono utilizzati e pertanto vuoti.

4.5.1 Invito a giocare / Risposta all'invito

Si tratta delle funzioni tramite le quali è possibile rispettivamente invitare a giocare una partita o rispondere all'invito.



Invito a giocare
Una volta stabilita la connessione con il *SubServer* tramite connect(), è possibile inviare la richiesta di sfida all'avversario tramite il seguente metodo:

```
int invite(string username);
```

- `Scopo`: Invia una richiesta di sfida al *SubServer* dell'avversario.
- `Argomenti`: *username* l'username dell'utente che effettua la richiesta di sfida in modo tale che lo sfidato possa sapere da chi parte l'invito.
- `Errori`: Restituisce `1` in caso di successo e risposta affermativa alla sfida proposta, `0` in caso di successo ma con una risposta negativa alla sfida proposta e un numero negativo $<$`0` in caso di fallimento.

Risposta all'invito
Una volta accettata la connessione con il *SubClient*, il *SubServer* resta in attesa dell'arrivo di un invito che viene mostrato all'utente il quale sceglierà se accettarlo o meno. L'esito di questa scelta verrà inviato come risposta al *SubClient*. Nel caso di più inviti da parte di giocatori distinti, questi saranno mostrati in ordine di arrivo, rifiutando il primo verrà mostrato il secondo etc...

```
int sendInviteReplay(bool b);
```

- `Scopo`: Invia una risposta alla sfida richiesta.
- `Argomenti`:b un booleano indicante l'esito della risposta.
- `Errori`:Restituisce un numero $>$`0` in caso di successo e `0` o un numero negativo$<$`0` altrimenti

4.5.2 Invio della mossa

Tramite questa funzione è possibile inviare da un *SubClient* a un *SubServer* o viceversa la mossa effettuata, ovvero inviare le coordinate della griglia su cui l'utente attacca.

La funzione, una volta inviate le coordinate, attende e poi ritorna l'esito dell'attacco, esso è definito dall'enumeratore `enum{ miss, hit, hitAndSunk};` ovvero i classici mancato, colpito e colpito e affondata.

```
int sendMove(int x, int y);
```

- `Scopo`: Invia la mossa effettuata.
- `Argomenti`:x, y le coordinate che si vogliono colpire sulla scacchiera avversaria.
- `Ritorno`:Restituisce un enumeratore intero contenente l'esito della mossa.
- `Errori`:Restituisce un numero $>$`0` in caso di successo e `0` o un numero negativo$<$`0` altrimenti

4.5.3 Attesa della mossa

Durante la partita quando non è il turno del giocatore, si attende la mossa avversaria, ovvero si attende che l'avversario invii la sua mossa con le relative coordinate.

La funzione non fa altro che attendere il pacchetto inviato dall'avversario, l'analisi della mossa e il suo esito e le sue conseguenze sono valutate esternamente.

```
int waitForMove(MoveRequest& m);
```

- `Scopo`: Attende l'arrivo della mossa avversaria;
- `Argomenti`:m il paccetto ricevuto, dal quale è possibile estrarre le coordinate;
- `Errori`:Restituisce un numero $>$`0` in caso di successo e `0` o un numero negativo$<$`0` altrimenti.

4.5.4 Invio del risultato della mossa

Una volta che la mossa attesa è arrivata, esternamente ne si verifica l'esito e quest'ultimo viene mandato come risposta.

```
int sendMoveReplay(MoveReplay::Result result);
```

- `Scopo`: Invia il risultato della mossa avversaria sulla propria scacchiera;
- `Argomenti`:*result* un enumeratore rappresentante il risultato;
- `Errori`:Restituisce un numero >`0` in caso di successo e `0` o un numero negativo<`0` altrimenti.

5 Dettagli implementativi

Qui di seguito mostreremo più nel dettaglio le scelte effettuate nella progettazione e nell'implementazione della piattaforma.

5.1 Implementazione dei client e server

Ogni client e ogni server all'interno della piattaforma è stato implementato attraverso una classe con i suoi metodi e attributi e funzionalità, pertanto avremo essenzialmente 4 classi, *MainClient, MainServer, SubClient, SubServer* all'interno di ciascuna vi sono i metodi necessari per il loro funzionamento e la comunicazione reciproca.

Dunque la connessione e la comunicazione fra client e server avviene nel `main()` attraverso istanze delle suddette classi e tramite i metodi ad esse associate, In questo modo la progettazione diventa più semplice e vengo definiti maggiormente i ruoli delle parti in gioco.

Anche se i client e i server all'interno dell'applicazione si basano su classi diverse le funzionalità base, ovvero quelle legate alla comunicazione e al network, sono le stesse e pertanto esse sono definite in due superclassi `Client` e `Server`.

5.1.1 I pacchetti

Come già accennato in precedenza nella sezione 4.2 , l'intera comunicazione far i vari client e server è basata sui pacchetti. Ogni pacchetto è stato rappresentato tramite una classe in C++, durante il normale funzionamento del programma i pacchetti vengono usati e manipolati come istanze della classe: vengono dunque creati, riempiti, vi si accede alle informazioni contenute ecc...

```
class LoginRequestPkg : public Package{
public:
    LoginRequestPkg() : Package(loginRequest) {}
    LoginRequestPkg(string username, string password, short port);

    static size_t pkgSize() {return sizeof(username) + sizeof(
       password) + sizeof(port);}
    size_t size() const {return pkgSize();}

    virtual const string serialize();
    LoginRequestPkg deserialize(const char* src);

    char username[20], password[20];
    short port;
};
```

Esempio di una classe pacchetto

Affinché quest'ultimi possano essere trasmessi su una socket è necessario però trasformare l'oggetto in una sequenza di byte e viceversa, ciò avviene tramite due processi detti rispettivamente serializzazione e deserializzazione. Tali funzioni non sono implementate direttamente nel linguaggio e pertanto è stato necessario definirle a parte come metodi della classe.

I pacchetti utilizzati sono in totale 10, ognuno legato strettamente ad una specifica richiesta o risposta:

`LoginRequestPkg`	`LoginReplyPkg`	`OnlineUserPkg`
`FreeUserPkg`	`ChallangeRequest`	`ChallangeReply`
`MatchRequest`	`MoveRequest`	`MoveReply`
`MatchPkg`		

Non tutte le richieste sono però legate ad uno specifico pacchetto e quindi inviate insieme ad esso. Esistono richieste, come ad esempio `.onlineUserRequest` e `.freeUserRequest` che non hanno bisogno di un pacchetto dati allegato.

5.1.2 Invio e recezione di richieste

L'invio e la recezione di una richiesta avviene attraverso i metodi:

```
int sendRequest(Package::Type type);
int waitRequest(Package::Type* type);
```

Essi consentono, ricevuto il tipo della richiesta di effettuare rispettivamente una `send/recv` per inviare o ricevere il tipo di richiesta alla socket con cui è connesso l'oggetto, che esso sia un client o un server.

Una volta che la richiesta è stata inviata è possibile attendere il codice status legato all'esito della richiesta tramite il metodo:

```
int waitReply(Package::Status* status);
```

5.1.3 Invio e recezione pacchetti

L'invio e la recezione di un pacchetto avviene attraverso i metodi:

```
int sendPkg(Package* pkg);
template<typename T> int waitPkg(T* pkg);
```

Si tratta di due metodi particolari, utilizzano il polimorfismo e template consentendo di inviare o ricevere un pacchetto qualsiasi il cui tipo è specificato esclusivamente dal puntatore `pkg` usato come argomento. In questo modo è possibile ad esempio creare il pacchetto da inviare e passarne il puntatore come argomento; tramite il polimorfismo verrà invocata la giusta funzione di serializzazione relativa a quel pacchetto e quindi inviata tramite socket.

5.2 Dettagli implementativi del Main Server

Il Main Server rappresenta il fulcro di tutta la piattaforma egli non solo permette la connessione fra gli utenti che voglio giocare un partita ma consente anche di tenere traccia del loro stato (offline, online, disponibile) e delle loro statistiche. Tutte queste informazioni sono state memorizzate in varie liste simulando un database di utenti.

5.2.1 Implementazione

Proprio poiché è necessario gestire le richieste di più utenti contemporaneamente è stato necessario implementare un server concorrente. La concorrenza viene ottenuta utilizzando i Thread anzichè i processi figli in questo modo è possibile condividere le informazioni e le varie liste mantenute dal Thread principale.

L'intero *MainServer* con tutte le sue funzionalità è implementato dall'omonima classe `MainServer`. Tale classe deriva dalla superclasse `Server` dalla quale eredita attributi e metodi necessari per la comunicazione, invio e recezione di richieste e pacchetti.

L'interfaccia pubblica del `MainServer` è semplicemente composta dal costruttore che consente di inizializzare la porta da utilizzare - di default la 1024 - e da una sola funzione `int start();` che permette di avviare il server e metterlo in ascolto.

5.2.2 Gestione dei client connessi

Come già accennato, il *MainServer* è stato implementato per gestire concorrentemente più client attraverso l'uso dei thread.

Una volta avviato il server esso resta in ascolto di connessioni da parte dei client, ogni qual volta un client si connette viene generato un nuovo thread a cui è demandata la gestione del client.

La funzione che il thread esegue è il metodo privato:

```
void MainServer::manageClient(int _connfd, struct sockaddr_in clientaddr);
```

A cui vengono passati il descrittore ricevuto dall'accept e la strutture clientaddr con le informazioni relative alla connessione. Fattò ciò la sessione è attiva e il thread si bloccherà attendendo l'arrivo di una richiesta. Una volta che la connessione sarà terminata il thread prima di ritornare effettuerà il logout dell'utente, libererà le eventuali risorse utilizzate e chiuderà il descrittore.

5.2.3 Data Management

la gestione di tutti i dati e informazione sui quali si basa il funzionamento del *MainServer* è effettuata essenzialmente mediante quattro liste implementate attraverso relative classi in modo da essere più semplici da usare:

- `registredList`
  Si tratta di un attributo del `MainServer`, istanza della classe `RegistredUsers`. Contiene dati riguardanti tutti gli utenti registrati sulla piattaforma, le loro statistiche, i loro privilegi e metodi che consentono di manipolare queste informazioni, come ad esempio vedere se un utente esiste o meno, aggiungerlo, rimuoverlo, validare il login, aggiornare le statistiche ecc... Internamente è implementata utilizzando `std::set<T>`.

- `onlineList` e `freeList`
  Si tratta di due attributi del `MainServer`, entrambi istanze della classe `UsersList`. Mantengono rispettivamente, i dati riguardanti gli utenti online e disponibili a giocare. E' possibile manipolare tali dati attraverso i vari metodi, come ad esempio vedere se un utente appartiene o meno alla lista, aggiungerlo, rimuoverlo, ecc... Internamente sono implementate utilizzando `std::list<T>`.

- `matches`
  Si tratta di un attributo del `MainServer`, istanza della classe `MatchList`. Contiene dati riguardanti tutte le partite giocate, sia quelle terminate che quelle in corso. tutto ciò non è rappresentato con un unica lista ma bensì ne vengono utilizzate due, quella dei match terminati e quella dei match in corso. Sono poi disponibile metodi per l'aggiunta di una nuova partita e la sua terminazione.

In questo modo è possibile aggiungere una partita solo ai match in corso, e la fine di quest'ultima provoca una rimozione dalla lista dei match in corso e l'aggiunta a quelli terminati. Internamente sono implementate utilizzando `std::list<T>`

5.3 Dettagli implementativi del client di gioco

Il client di gioco è l'applicazione lato client che consente all'utente di giocare e connettersi al *MainServer*, Proprio perché è utilizzata dagli utenti è stato dotato di un interfaccia grafica con finestre e box colorate ottenuta partendo da caratteri unicode.
Esso è composto essenzialmente di due parti:

- *Parte di gioco*: Ovvero tutti quegli elementi che fanno parte esclusivamente del gioco della battaglia navale, della GUI e che consentono appunto di giocare.
- *Parte network*: Ovvero quegli elementi che permettono una comunicazione sia con il *MainServer* che con gli altri utenti. In questa sezione rientrano il `MainClient`, il `SubClient` ed il `SubServer`.

5.3.1 Implementazione del Main Client

Il *MainClient* è implementato derivando la classe base `Client` dalla quale ne eredità tutti gli elementi, metodi e attributi, necessari alla connessione con un server.

Ogni metodo della classe equivale ad una precisa richiesta da effettuare al server con cui si connette. Pertanto il client di gioco utilizza questi metodi per effettuare le operazioni necessarie come ad esempio la richiesta di login, la richiesta dei giocatori online ecc...

5.3.2 Implementazione del Sub Client e del Sub Server

Anche il *SubServer* e il *SubClient* sono implementati attraverso due classi, e la loro struttura è essenzialmente uguale a quella del *MainClient* con la differenza che il tipo di richieste che possono effettuare è diverso e riguardano puramente il gioco della battaglia navale.

- *Sub Server*: Viene fatto partire in background all'avvio del client di gioco in modo che esso effettui il bind dinamicamente su una porta disponibile, tale porta sarà poi comunicata al *MainServer*.
  Quando si è contattati da un altro giocatore verrà mostrata una finestra di scelta che se accettata porterà all'inizio della partita, da questo momento in poi il controllo del client di gioco passa dal thread principale a quello che sta eseguendo il *SubServer*.
- *Sub Client*: Anch'esso viene creato all'inizio, tuttavia resta inattivo fin quando il giocatore non sceglie uno sfidante e riceve i suoi dati dal Main Server; a quel

punto effettua la connessione e invia la richiesta di gioco che se accettata porterà all'inizio della partita.

Entrambi dunque sono coinvolti solamente nella fase di gioco al termine della quale il *SubClient* viene disconnesso e il *SubServer* si blocca, di nuovo, in attesa di un nuovo sfidante.

5.3.3 GUI

L'interfaccia grafica utilizzata è stata realizzata ricorrendo alla combinazione di particolari caratteri ASCII e UNICODE e codici escape che permettono di cambiare colore del testo e coordinate del cursore all'interno del terminale.

Partendo da ciò sono state create delle classi `Box, TextBox, Grid, Ship, Mark, Board` che consentono di rappresentare gli elementi dell'interfaccia e del gioco mostrando le navi posizionate, i punti colpiti e la griglia dell'utente e dell'avversario.

Ogni oggetto delle suddette classi deriva dalla classe astratta `Drawable` dalla quale eredita una serie di attributi base come il colore, le coordinate x e y di rappresentazione e il metodo virtuale `void draw()= 0;` che deve essere implementato per poter disegnare correttamente l'oggetto sullo schermo.

5.4 Dettagli sul protocollo applicazione

Qui di seguito si illustrano alcuni dettagli sul funzionamento protocollo applicazione utilizzato e sul susseguirsi di operazioni svolte nel main():

1. All'avvio del client di gioco, viene avviato in background il *SubServer* che effettua il bind su una porta scelta dinamicamente, ed il *MainClient* che invia una richiesta di connessione al *MainServer*.
2. Viene mostrata la finestra di login tramite la quale posso effettuare l'accesso, quando viene inviata la richiesta oltre all'username e password si invia anche la porta sul quale il *SubServer* è in ascolto.

   In questo modo il MainServer è a conoscenza dell'IP e porta del SubServer di ogni giocatore che ha effettuato correnttamente il login, tali informazioni sono memorizzate nelle liste di utenti online e utenti disponibili.
3. Quando un utente ne vuole sfidare un'altro manda una richiesta al server con il suo username il quale, una volta effettuati i vari controlli, risponde con l'IP e Porta del *SubServer* dello sfidato.
4. Lo sfidante ora attraverso il suo *SubClient* può contattare direttamente l'utente sfidato (più precisamente il suo *SubServer*) per invitarlo a giocare...
5. Terminata la partita i due comunicano il risultato al *MainServer*.

6 Manuale Utente

In questo capitolo verrà illustrato come procedere alla compilazione, esecuzione e utilizzo del progetto.

6.1 Istruzioni per la compilazione

La compilazione avviene in maniera molto semplice utilizzando le utility `cmake` e `make`. La prima permette di generare automaticamente il *makefile* necessario alla compilazione, mentre la seconda effettua la compilazione vera e propria.

Nel caso in cui non si disponga delle suddette utility, è necessario scaricarle dalla repository, per distribuzioni basate su Ubuntu/Debian è sufficiente digitare da terminale:

```
$ sudo apt-get install make
$ sudo apt-get install cmake
```

Le sorgenti del progetto saranno divise essenzialmente in tre cartelle:

- `./BattleshipServer`: contenente il codice del server principale;
- `./BattleshipClient`: contenente il codice del client di gioco;
- `./library`: contenente alcuni file sorgente e librerie utilizzate sia dal client che dal server.

Per compilare il progetto sarà necessario ripetere l'operazione due volte; una per `./BattleshipServer` e l'altra per `./BattleshipClient`

6.1.1 Compilazione del Server principale

> Lo strumento `cmake` genera il *makefile* corredato da tutta una serie di file necessari alla compilazione pertanto è consigliabile creare una cartella ad hoc per contenere l'output oppure, come mostrato in seguito, eseguire il comando direttamente all'interno della cartella del codice sorgente.

1. Aprire il terminale e posizionarsi all'interno della cartella `./BattleshipServer`;
2. Generare il *makefile* attraverso il comando: "`$ cmake <path>`"; in questo caso poiché ci troviamo già all'interno della cartella target è sufficiente eseguire: "`$ cmake .`";
3. Compilare il codice sorgente attraverso il comando "`$ make`"

Seguiti questi tre semplici passi verrà generato, all'interno della cartella stessa, il file `BattleshipServer` ovvero l'eseguibile del server principale.

6.1.2 Compilazione del Client di gioco

La procedura compilazione del Client di gioco avviene nello stesso identico modo di quella del Server principale appena descritta, ovviamente con l'attenzione a posizionarsi con il terminale all'interno della cartella `./BattleshipClient`.

Seguita la procedura, verrà generato il file `BattleshipClient` ovvero l'eseguibile del client di gioco.

6.2 Istruzioni per l'esecuzione

In questa sezione verranno mostrate le istruzioni per l'esecuzione dei due eseguibili generati precedentemente, ovviamente per permettere il funzionamento dell'applicazione è necessario eseguire prima il Server principale e poi n istanze, una per ogni utente, del Client di gioco.

6.2.1 Esecuzione del Server principale

L'esecuzione del Server principale avviene semplicemente posizionandosi nella directory dell'eseguibile e usando il comando:

```
$ ./BattleshipServer -p [porta]
```

Dove il valore dell'opazione `-p` corrisponde alla porta sul quale si vuole far partire il server, nel caso quest'ultima non sia specificata, eseguendo quindi l'eseguibile senza alcuna opzione, la porta scelta di default sarà la 1024.

6.2.2 Esecuzione del Client di gioco

L'esecuzione del Client di gioco avviene in maniera analoga, posizionandosi nella directory dell'eseguibile e usando il comando:

```
$ ./BattleshipClient -a [ip] -p [porta]
```

Dove il valore delle opazioni `-a` e `-p` corrispondono rispettivamente all'ip del server e alla porta sul quale esso è in ascolto. Nel caso in cui, l'eseguibile sia eseguito senza alcuna opzione, l' ip scelto di default è 127.0.0.1 mentre la porta scelta sarà la 1024.

6.3 Istruzioni per l'utilizzo

6.3.1 Dimensioni del terminale

Affinché sia possibile giocare ed utilizzare il Client di gioco è necessario che le dimensioni della finestra del terminale siano appropriate. La dimensione necessarie sono:

- Almeno 100 colonne in larghezza;
- Almeno 30 linee in altezza;

Non è necessario rispettare con precisione tali valori il terminale può essere anche di dimensioni maggiori, la GUI si adatterà di conseguenza.

Tuttavia per evitare problemi o difficoltà all'avvio del Client di gioco nel caso in cui le dimensioni minime non siano rispettate verrà mostrata una schermata, mostrante le dimensioni correnti, invitando ad aumentarle per proseguire.

6.3.2 Credenziali per il login

Una volta avviato il client di gioco, se la connect() iniziale con il server va a buon fine, verrà visualizzata la schermata di login saranno dunque necessarie le credenziali dell'utente per proseguire:
Di seguito viene mostrata la lista degli utenti registrati presenti sulla piattaforma:

Username	Password	Vittorie	Sconfitte	admin
a	a	0	0	✗
b	b	0	0	✗
c	c	0	0	✗
utente1	utente1	10	5	✗
utente2	utente2	4	4	✗
utente3	utente3	6	3	✗
utente4	utente4	1	9	✗
utente5	utente5	12	14	✗
utente6	utente6	1	9	✗
giuseppe	12345678	3	1	✗
francesco	12345678	3	1	✗
admin	admin	0	0	✓