Hunting the Bismarck

S. Barone M63/610 - A. Di Martino M63/654 - P. Liguori M63/556

3 dicembre 2017

Indice

1 Introduzione

Hunting the Bismark è un gioco online multiplayer di battaglia navale.

Il giocatore, una volta collegato al server attende in una *waiting room,* nel caso in cui non ci siano almeno due utenti collegati, prima di poter dar inizio ad una sessione di gioco.

Per poter giocare, l'utente deve essere registrato e quindi l'accesso viene garantito solo con l'autenticazione dello stesso. I dati degli utenti vengono salvati in un database, creato e gestito attraverso MySQL.

Il gioco rispecchia le regole della tradizionale battaglia navale. Ogni giocatore visualizza due griglie di gioco, come illustrato nell'immagine 1:

- sulla sinistra è presente la griglia del giocatore, identificata dal proprio username (inserito in fase di registrazione / login), e dallo score del math corrente che viene aggiornato volta per volta. In questa griglia è possibile vedere la disposizione delle proprie navi (disposte in maniera randomica).

- sulla destra è presente una seconda griglia, quella dell'avversario. Anche qui è visibile l'username dell'avversario con lo score che egli sta realizzando nel match corrente. Su questa griglia il giocatore, cliccando su una delle caselle, prova a colpire una delle navi dell'avversario la cui posizione, ovviamente, non è visibile.

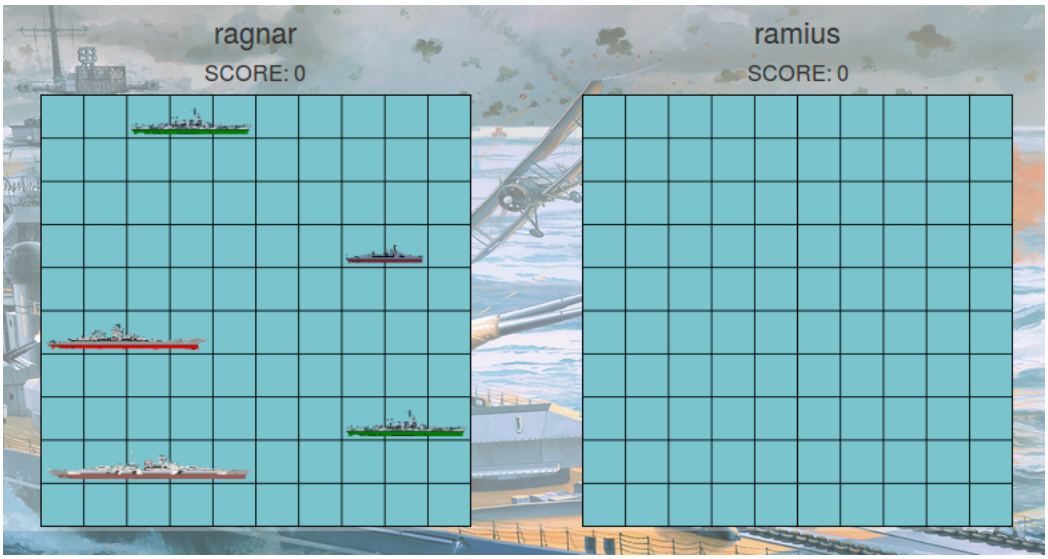

Figura 1: Griglie giocatore

Le navi disposte in modo casuale nelle griglie degli utenti sono in ugual numero e condividono le medesime dimensioni (ma non la posizione).

Il giocatore, quando è il proprio turno, clicca su una delle caselle della griglia dell'avversario. Al momento del click viene emesso un suono di un cannone nel tentativo di voler emulare una battaglia navale quanto più "reale" possibile. In seguito allo sparo, ci sono due possibili scenari:

- una nave avversaria era posizionata sulla casella cliccata: in questo caso sulla casella appare un simbolo di una palla di fuoco, accompagnato da un suono che emula un'esplosione.

- nessuna nave avversaria era posizionata sulla casella cliccata: in questo caso sulla casella appare un simbolo che fa riferimento a uno "splash", accompagnato da un suono che emula una situazione in cui il colpo sparato è caduto in acqua.

Nel caso in cui l'utente faccia centro, allora sarà ancora il suo turno fino a quando non sbaglierà colpo: in questo caso sarà il turno dell'avversario e l'utente deve aspettare il fuoco nemico. Ogni qualvolta si fa centro, l'utente incrementa il proprio score. Colpi multipli andati a buon segno incrementano

linearmente lo score corrente. L'utente non può, inoltre, sparare su una casella che aveva già colpito in precedenza.

I due avversari hanno possibilità di interagire tra loro anche tramite una chat che è disposta nella parte basse della pagina:

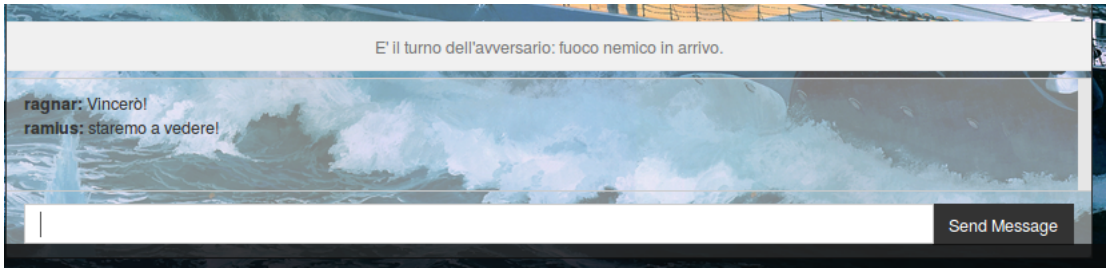

Figura 2: Chat giocatori

Vince la partita chi affonda per primo tutte le navi dell'avversario.

Il numero di partite vinte, lo score totale realizzato e il record personale dell'utente vengono tutti salvati nel database ed è possibile visualizzarli nell'apposita pagina di rank.

2 Realizzazione

In questa sezione e nelle successive sarà spiegato come realizzare il gioco sopra illustrato.

Il gioco si basa su **nodeJS**, fa utilizzo di **socket.io** per la comunicazione client-server e di **canvas** per il rendering dinamico delle immagini. Il database in cui vengono salvate le informazioni degli utenti è stato creato e gestito con **MySQL.**

Per prima cosa diamo un occhio alla disposizione dei file e delle cartelle che hanno costituiscono nel loro insieme il gioco:

- nella cartella principale *nodejs* troviamo:
  - *server.js*: vengono richiesti tutti i moduli, e viene messo il server in ascolto su un porto specifico;
  - *config.js*: file di configurazione del database;
  - *huntingthebismark.mwb*: il database creato;
  - *package.json*: il pacchetto json;
- nella sottocartella *node/module* troviamo tutti i moduli installati;
- nella sottocartella *app/public* troviamo:
  - le pagine html richieste;
  - *client.js* e *game.js* che definiscono il comportamento e la grafica lato client;
  - *login.js* e *register.js* per poter effettuare particolari particolari funzioni in fase di registrazione e login, come ad esempio verificare la disponibilità dello username scelto;
  - i file *.css*;
- nella sottocartella *app/server* troviamo:
  - la cartella *game*, all'interno della quale si trovano tutti file js che contengono le funzioni necessarie ad implementare tutta la logica del gioco;
  - la cartella *controllers*, contenente i file js necessari alla registrazione ed autenticazione dell'utente. I moduli implementati in questi file vengono utilizzati dal file *router.js*;
  - il file *gameServer.js*, che implementa la logica di gioco lato server.

Per quanto riguarda il database, esso contiene i seguenti campi:

- *id*: identificativo unico dell'utente;

- *nome, cognome, country*: sono informazioni relative all'utente che vengono inserite all'atto della registrazione;

- *username* e *password*: sono le credenziali di accesso dell'utente. Vengono create in fase di registrazione ed utilizzate per effettuare il login;

- *score*, *max_score*, *vinte*: informazioni legate ai risultati che l'utente ottiene durante le varie sessioni di gioco. *Score* indica il punteggio totale realizzato dall'utente in tutte le sue partite disputate; *max_score* rappresenta il record personale dell'utente ottenuto in una singola partita; *vinte* indica quante match disputati sono stati vinti dall'utente.

Prima di passare alla descrizione dei singoli file *js*, viene mostrato il caso d'uso che specifica le azioni che un utente, registrato o meno, può effettuare interagendo col sistema.

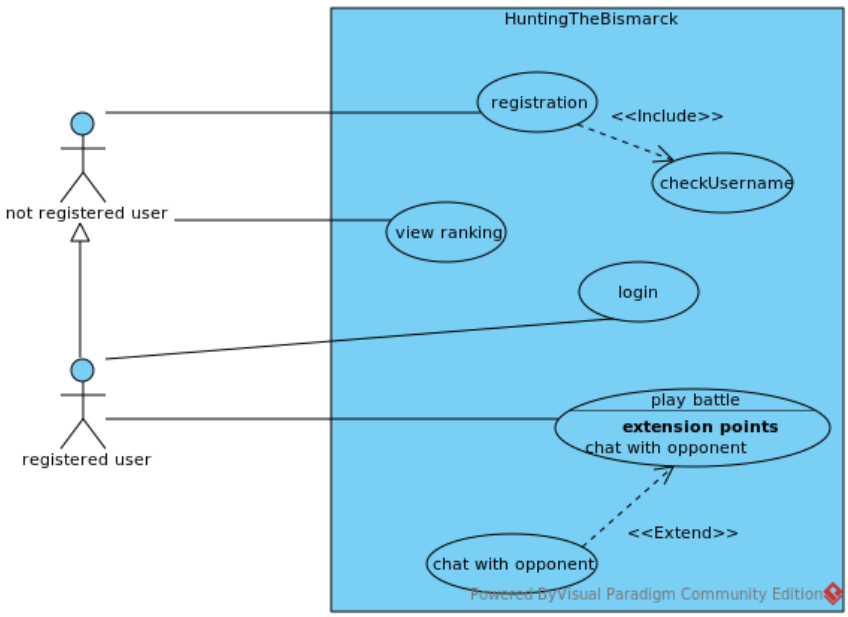

Figura 3: Use-case diagram

3 Registrazione

La form di registrazione è inclusa nella pagina html *register.html*. Essa presenta diversi campi di input, tra cui *username* e *password* che verranno usati per identificare l'utente al login.

3.1 Controllo della disponibilità di uno username

È necessario che ogni username sia unico: non possono, ovviamente, esserci due utenti con lo stesso username; per questo motivo è stato previsto un controllo, implementato usando una richiesta asincrona spedita verso il server, al fine di evitare che l'utente possa tentare di registrarsi usando uno username non disponibile, perché già utilizzato da qualche altro giocatore.

In figura 4 viene riportato uno schema, non troppo formale, delle interazioni esistenti tra gli oggetti coinvolti nel controllo. Ogni volta che un utente inserisce un nuovo carattere nella textbox associata allo username, quando l'utente rilascia il tasto premuto, viene generato un evento "on key up", al quale è associata la la callback *checkUsername()*, la cui implementazione è visibile nel file *register.js* ed è riportata nel box 1.

Algoritmo 1 checkUsename()

```
1  function checkUsername() {
2      var usern = document.getElementById("uname").value;
3      var req = newXMLHttpRequest();
4      req.onreadystatechange = function() {
5          if (req.readyState == 4)
6           if (req.status == 200)
7              manageResponse(req.responseText);
8      }
9      req.open("POST", "/username", true);
10     req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
11     var par = "username=" + escape(usern);
12     console.log(par);
13     req.send(par);
14 }
```

Tale funzione crea un nuovo oggetto *XMLHttpRequest* usando la funzione *newXMLHttpRequest()*, riportata nel box 2, seguendo un approccio "canonico", il quale prevede la creazione di suddetto oggetto, la registrazione di una callback da richiamare al sopraggiungere della risposta dal server, e, in questo caso usando il metodo *POST*, inviare il contenuto della textbox al server. Tale funzione ha lo scopo di garantire portabilità dei metodi previsti dalla tecnica AJAX.

Algoritmo 2 newXMLHttpRequest()

```
1  function newXMLHttpRequest() {
2      var request = null;
3      var browser = navigator.userAgent.toUpperCase();
4      if(typeof(XMLHttpRequest) == "function" ||
5          typeof(XMLHttpRequest) == "object") {
6          request = new XMLHttpRequest();
7      } else {
8          if(window.ActiveXObject && browser.indexOf("MSIE 4") < 0) {
9              if(browser.indexOf("MSIE 5") < 0) {
10                 request = new ActiveXObject("Msxml2.XMLHTTP");
11             } else {
12                 request = new ActiveXObject("Microsoft.XMLHTTP");
13             }
14         }
15     }
16     return request;
17 }
```

La funzione *checkUsername()* associa all'evento "*on ready state change*" della risposta una callback, la quale, se la risposta è un messaggio "*200: OK*" e se lo stato della stessa è "*4: ready*", richiama la funzione *manageResponse()*, passandogli come parametro la risposta giunta dal server. La risposta viene generata, lato server, dalla funzione riportata nel box 3, richiamata al sopraggiungere di una richiesta *POST* all'url "*/username*".

Algoritmo 3 generazione della risposta per checkUsername()

```
app.post('/username', function(req, res) {
    var uname = req.body.username;
    connection.query('select count(*) as namescount from users where username = ?', [uname],
        function(err, result) {
            if (err) {
                console.log('error');
            } else {
                var response = result[0].namescount;
                res.send({'response': response});
            }
        });
});
```

Tale risposta contiene il numero di occorrenze di "username" nel database che mantiene le informazioni relative ai giocatori registrati. Se la risposta è diversa da zero la callback *manageResponse()* farà si che di fianco alla textbox dello username compaia un messaggio "Username non disponibile", così come riportato in figura 5, altrimenti il messaggio sarà "Username disponibile".

Figura 5: form di registrazione

3.2 Registrazione di un utente

Il processo di registrazione, in se, è molto semplice: lo schema in figura 6 mostra le interazioni esistenti tra gli oggetti coinvolti nel processo di registrazione.

Quando l'utente clicca sul pulsante "*Registrati*" della form di registrazione, viene richiamata la funzione *formValidation()* associata a tale evento. La prima operazione effettuata da tale funzione è controllare che i campi username e password non siano vuoti; in questo caso viene mostrato un messaggio che invita l'utente a riempire suddetti campi (fig. 7). Se entrambi i campi sopracitati non sono nulli, viene inviata una richiesta POST al server, contente i dati che l'utente ha inserito nella form, affinché vengano registrate nel database.

Lato server viene eseguita la funzione riportata nel box 4, la quale non fa altro che inserire i dati inviatigli dal client all'interno del database. Dopo che la registrazione si è conclusa con successo, l'utente viene automaticamente rediretto alla pagina del gioco, affinché attenda l'inizio di una nuova partita.

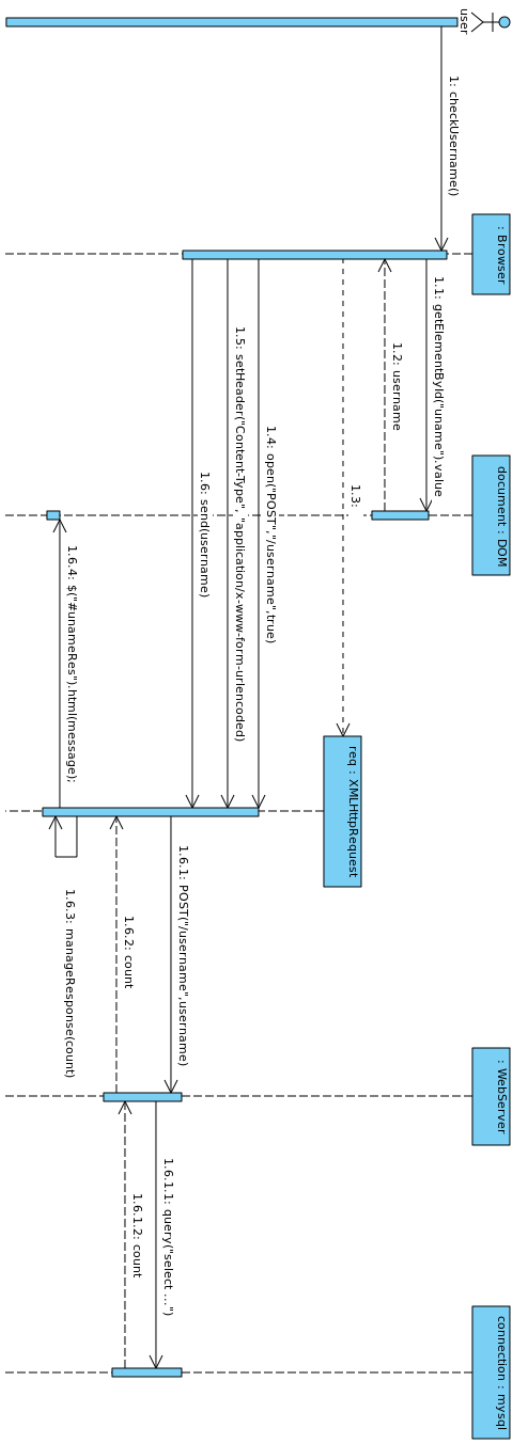

Figura 4: controllo disponibilità di uno username

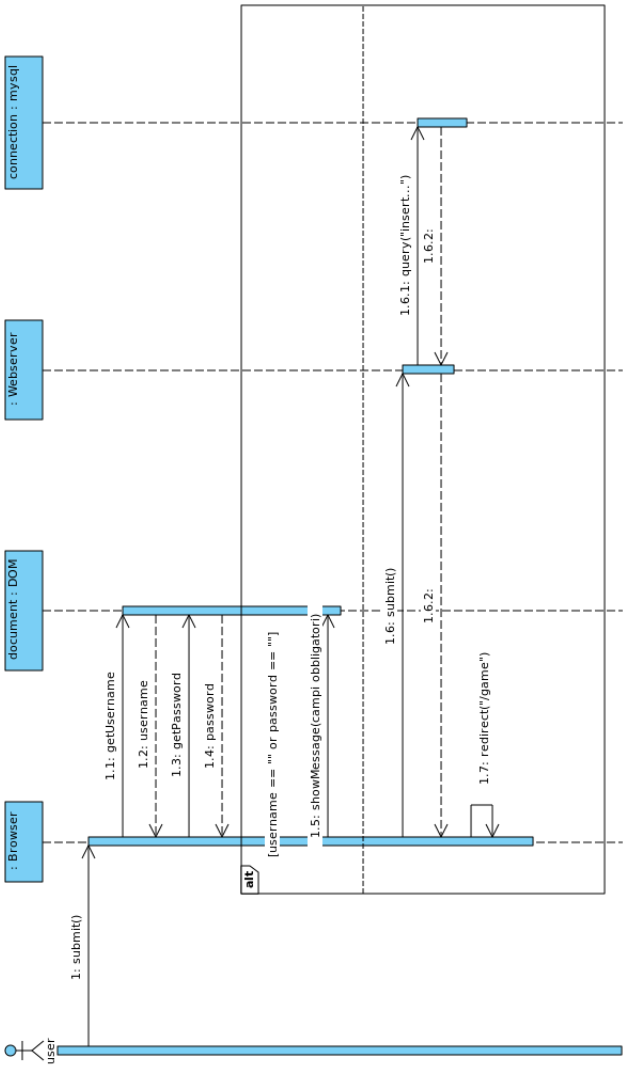

Figura 6: processo di registrazione

Figura 7: campi obbligatori

Algoritmo 4 codice server-side per la registrazione di un utente

```
app.post('/register', function(req, res) {
    var users = {
        "nome": req.body.nome,
        "cognome": req.body.cognome,
        "country": req.body.country,
        "username": req.body.username,
        "password": req.body.password
    }
    //Check user's data and insert it into database if username does not already exist
    R.register(req.body.nome, req.body.cognome, req.body.country, req.body.username, req.body.password,
        function(e) {
            if (e) {
                res.sendFile(path.join(__dirname, '../public', 'error.html'));
            } else {
                //Set user session and redirect to game page
                req.session.user = users.username;
                res.redirect('/game');
            }
        });
});
```

4 Login

La form di login è estremamente semplice: presenta due soli campi, ossia username e password, in cui l'utente deve inserire le proprie credenziali per poter accedere al gioco, ed un pulsante mediante il quale l'utente può confermare le credenziali inserite.

Di soluzioni per effettuare il login ne esistono di più disparate; in questo caso si è adottata una soluzione più semplice possibile: quando l'utente clicca sul tasto "Login" viene creata una richiesta "POST", contenente le credenziali dell'utente, che viene spedita al server; quest'ultimo controlla le credenziali e agisce di conseguenza. Se le credenziali sono sbagliate l'utente viene rediretto su una pagina di errore, praticamente identica a quella di login, ma recante un messaggio di errore come quello riportato in figura 8, altrimenti viene rediretto alla pagina "/game".

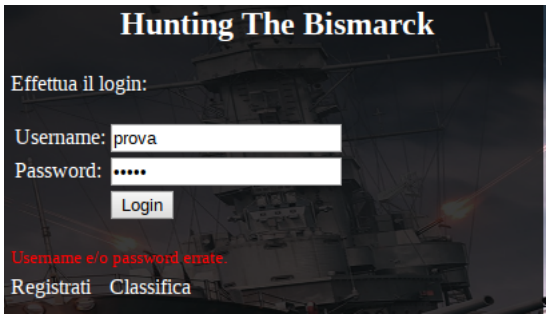

Figura 8: messaggio di errore visualizzato in fase di login

In figura 9 viene riportato un diagramma, non troppo formale, che mostra quali siano le interazioni esistenti tra gli oggetti coinvolti nelle operazioni di login. Il diagramma è estremamente semplice. In fig. 5 viene riportato il codice server-side eseguito all'atto del login.

Algoritmo 5 codice server-side eseguito al login

```
app.post('/login', function(req, res) {
    //Using authenticate method to check if username and passoword match
    A.authenticate(req.body.username, req.body.password, function(o) {
        if (!o) {  //if they not match, redirect to error page
            res.sendFile(path.join(__dirname, '../public', 'error.html'));
        } else {
            //If they match, set user session and redirect to game page
            req.session.user = o.user;
            console.log("Provo a salvare la sessione " + req.session.user);
            res.redirect('/game');
        }
    });
});
```

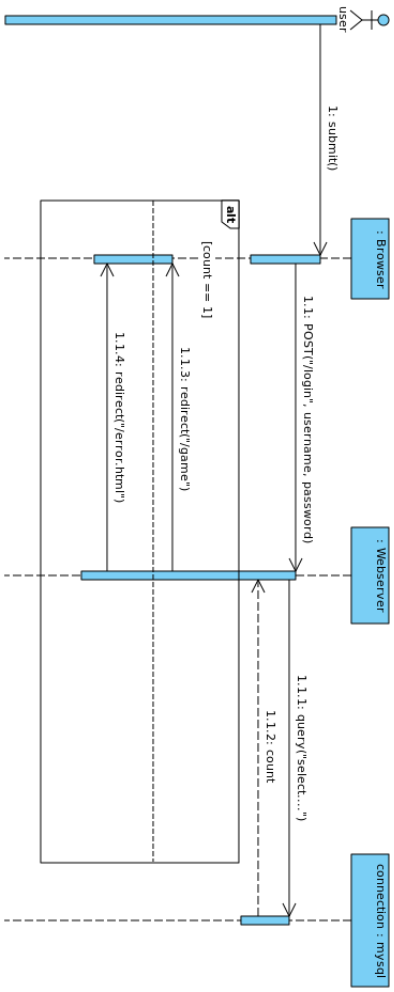

Figura 9: sequence-diagram per le operazioni di login

5 Server

Nel file *gameServer.js* vengono implementate tutte quelle funzionalità lato server che servono alla corretta realizzazione di una sessione di gioco.

E' stata utilizzata la libreria **socket.io** per implementare una comunicazione real-time bidirezionale basata sugli eventi tra il server e i client.

Sull'evento di connessione da parte di un utente, viene creata una struttura dati associata all'utente contenente informazioni quali nickname, score, hit, ecc. Tutti questi campi vengono inizializzati a zero o null.

Il server attende l'evento 'new player' per poter assegnare l'username alla struttura dati associata in precedenza. Nel caso in cui risulti che l'username sia null (per un qualsiasi motivo, come problemi di trasmissione, o di rete ad esempio), il server richiede di nuovo l'username fino a quando non sarà diverso da null. Ciò crea una sorta di loop dal quale non si può uscire fino a quando non è stato ricevuto correttamente l'username dell'utente, come mostrato nel codice 6. Tale azione si è resa necessaria per poter inserire tutti i dati legati alla sessione di gioco nel database.

Algoritmo 6 Codice verifica username

```
socket.on('new player', function(msg) {
      if (msg == "") {
        console.log("Errore Username, notifico ritrasmissione");
        io.to(socket.id).emit('errorUsername');
      } else {
        users[socket.id].nickname = msg;
        console.log("STAMPO IL NICKNAME :" + users[socket.id].nickname);
      }
    });
```

Ogni qualvolta si verifica un evento di *'shot'*, relativo allo sparo di un utente, verifica innanzitutto se il colpo è valido o meno (cioè se l'utente ha cliccato su una casella libera o su una non disponibile perché già colpita in precedenza). Nel caso di colpo valido, vengono effettuate le seguenti operazioni:

- invia un evento al giocatore che ha sparato che permette di sentire lo sparo di un cannone;
- verifica se il gioco è finito (se uno degli utenti ha affondato tutte le navi del suo avversario);
- nel caso in cui il gioco non è concluso, allora effettua un update dello stato degli utenti, quindi effettua il cambio turno se il colpo era andato a secco, altrimenti l'utente ha possibilità di sparare un altro colpo;
- update dello score e notifica dello stesso agli utenti;
- notifica agli utenti se hanno colpito o meno una nave, inviando eventi differenti che permettono loro di ascoltare un suono diverso a seconda dei due casi:

Algoritmo 7 Invio eventi hit / miss ship

```
if (game.getPlayerShot(opponent)) {
            io.to(socket.id).emit('hit_ship');
            io.to(game.getPlayerId(opponent)).emit('hit_ship');
          } else {
            io.to(socket.id).emit('miss_ship');
            io.to(game.getPlayerId(opponent)).emit('miss_ship');
          }
```

Nel caso in cui il colpo non sia valido, viene inviato solo un evento che permette al giocatore che ha sparato di udire un sono di warning, segnalando quindi un colpo non partito.

Di un certo rilievo sono le funzioni implementate in questo file:

- la funzione **joinWaitingPlayers()**: vede gli utenti collegati e crea una sessione di gioco non appena ci sono due utenti collegati. Ovviamente permette la creazioni di sessioni di gioco multiple nel caso ci siano più di due utenti collegati. Una volta creata la sessione, ai due utenti viene assegnato un playerID (id del giocatore all'interno della sessione di gioco, utile per effettuare il cambio di turno ed inviare eventi) e un gameID (id della sessione di gioco).

- la funzione **leaveGame(socket)**: controlla se uno dei giocatori ha abbandonato lo scontro (disconnessione, chiusura della pagina web, ricarica della pagina stessa). In questo caso, notifica all'altro utente la disconnessione del suo avversario e gli assegna la vittoria.

- la funzione **checkGameOver(game)**: verifica se uno dei giocatori ha affondato tutte le navi e in tal caso notifica la vittoria / sconfitta al vincitore / perdente. Dopodiché chiama la funzione updatePlayer, passando i parametri username e score realizzato del player e distinguendo il vincitore dal perdente assegnando rispettivamente il valore '1' o '0' al parametro winner.

- la funzione **updatePlayer(username, score, winner)**: aggiorna i dati contenuti nel database relativi all'utente identificato dall'username che la funzione riceve come parametro. In particolare aggiorna lo score totale dell'utente, verifica ed eventualmente aggiorna il suo max score se ha effettuato il suo miglior punteggio in una partita ed incrementa il numero di vittorie nel caso il parametro winner valga '1'.

Il file *gameServer.js* gestisce, inoltre, anche la chat. Quando riceve un evento di tipo *'chat'* da uno dei due player della sessione contenente un messaggio, prende tale messaggio e l'username del giocatore che lo ha scritto e li invia nella ai due giocatori della sessione di gioco, i quali visualizzeranno la coppia username mittente più messaggio nella finestra di chat.

6 Client

Nel file **client.js** vengono implementate tutte le funzionalità lato client.

Per prima cosa viene effettuata la registrazione al backend, dichiarando la variabile *socket = io()*. La connessione verrà notificata dall'evento *connect*.

Alla ricezione di un evento *'join'*, preleva la pagina il suo username dalla pagina html di gioco ed invia un evento contenente la sua username al server, che terrà traccia delle sue informazioni relative alla corrente sessione di gioco. Viene inoltre riprodotta una musica di sottofondo nella sessione stessa.

Algoritmo 8 Ricezione evento 'join'

```
socket.on('join', function(gameId) {
  var username = $('#myUsername').text();
  console.log('Username inviato ' + username);
  socket.emit('new player', username);
  Game.initGame();
      $('#messages').empty();
  $('#disconnected').hide();
  $('#waiting-room').hide();
  $('#game').show();
  $('#game-number').html(gameId);
  $('#myScore').html("0");
  $('#oppScore').html("0");
  socket.emit('oppNotification');    //Notify opponent's username
  myMusic.play();
      }
```

Spesso capita che il client invii un username nullo al server, probabilmente perché la pagina html del gioco, da cui egli prende il suo username, non si è ancora caricata del tutto. Allora il server riceve un evento dal server dove viene richiesto di rinviare la sua username: tale meccanismo può essere visto come una sorta di loop da cui si esce non appena il client sarà riuscito ad inviare un username diverso da null.

Algoritmo 9 Ricezione evento 'errorUsername'

```
socket.on('errorUsername',function(){
  socket.emit('new player',$('#myUsername').text());
});
```

E' stata creata una funzione **sound** che permette la creazione di un elemento audio a partire da path che specifica un file musicale, e i cui metodi *play()* e *stop()* permettono di riprodurre o stoppare la riproduzione del file audio specificato.

Algoritmo 10 Funzione sound()

```
function sound(src) {
  this.sound = document.createElement("audio");
  this.sound.src = src;
  this.sound.setAttribute("preload", "auto");
  this.sound.setAttribute("controls", "none");
  this.sound.style.display = "none";
  document.body.appendChild(this.sound);
  this.play = function() {
    this.sound.play();
  }
  this.stop = function() {
    this.sound.pause();
  }
}
```

Il client in prevalenza riceve eventi dal lato server e, in conseguenza di ciò, avviene qualcosa:

- alla ricezione dell'evento *'update'* viene aggiornato lo stato del gioco del client;
- alla ricezione degli eventi *'scoreMe'* e *'scoreOPP'* aggiorna rispettivamente lo score del giocatore e del suo avversario nella pagina html di gioco;

- alla ricezione degli eventi *'gameover_winner'* o *'gameover_loser'* viene decretata la fine del partite corrente ricevendo una notifica testuale ed audio diversa che dipende da se il giocatore ha vinto o perso la partita corrente;

- alla ricezione degli eventi di *'valid_shot'* e *'invalid_shot'*, che indicano un colpo valido o invalido (a seconda se il client ha precedentemente sparato su una casella libera o meno) vengono riprodotto rispettivamente il suono di un cannone da guerra o una notifica di warning;

- alla ricezione degli eventi di *'hit_ship'* e *'miss_ship'*, che vengono ricevuti a secondo del fatto che il colpo precedentemente inviato dal giocatore abbia colpito o meno una nave avversaria. Se una nave è stata colpita, allora viene riprodotto il suono di un esplisone, altrimenti viene riprodotto il tipico suono di *'splash'*;

Gli eventi che vengono inviati dal client verso il server sono:

- *'shot':* viene inviato quando viene chiamata la funzione *sendShot()*, cioè quando l'utente clicca su una casella della griglia avversaria durante il proprio turno di gioco. Come abbiamo visto nel file **gameServer.js**, alla ricezione di tale evento vengono effettuate varie verifiche (colpo valido / invalido, nave colpito / non colpita , gioco completato o meno, ecc.) a seconda delle quali vengono inviati vari eventi, alcuni dei quali illustrati precedentemente;

- *'chat'*: viene inviato quando viene premuto il button "*Send Message*" assieme al messaggio testuale presente nella casella di chat del giocatore. Alla ricezione di tale evento, il server rinvia ad entrambi gli utenti della sessione di gioco la coppia username mittente più messaggio ricevuto che verrà visualizzata nella chat dei giocatori.

Avendo descritto dettagliatamente il comportamento lato client e lato server, viene mostrato nell'immagine 10 un sequence diagram dell'interazione client-server durante una sessione di gioco.

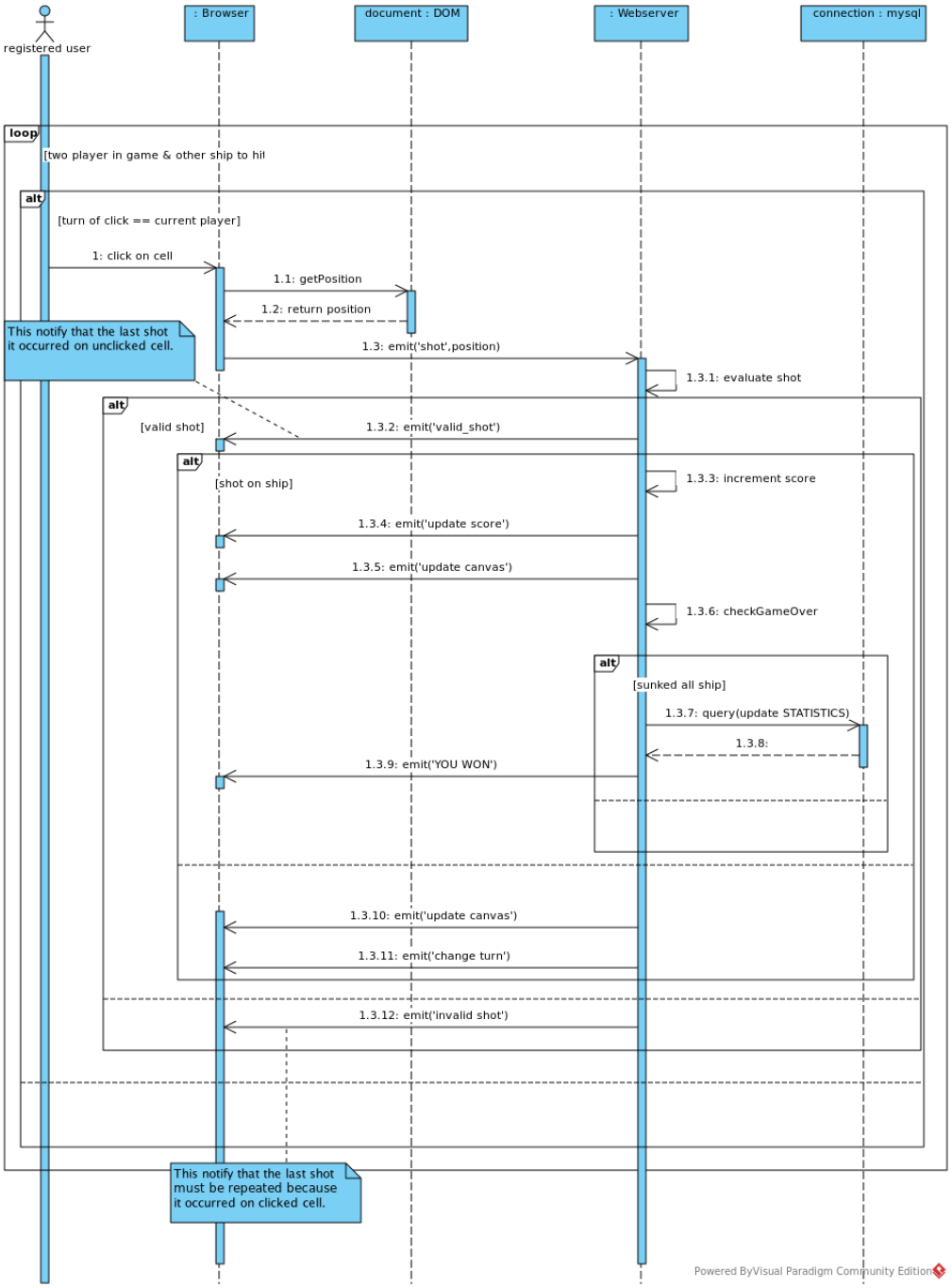

Figura 10: Sequence diagram interazione client-server

6.1 game.js

Il file **game.js** fa uso di **canvas**, un'estensione dell'HTML standard che permette il rendering dinamico di immagini bitmap gestibili attraverso un linguaggio di scripting.

Una delle funzionalità implementate più importanti è quella di realizzare la griglia dei giocatori attraverso le seguenti funzioni:

- *drawSquares()*: disegna le caselle della griglia;

- *drawShips()*: disegna le navi sulla griglia;

- *drawMarks()*: disegna sulla casella l'icona di una palla di fuoco in caso di nave colpita e l'icona di splash quando non viene colpita nessuna nave.

Quando il giocatore clicca una casella della griglia dell'avversario, vengono prese le coordinate del mouse al momento del click e viene richiamata la funzione *sendShot()*.

Algoritmo 11 Click event del giocatore su una casella avversaria

```
canvas[1].addEventListener('click', function(e) {
  if(turn) {
    var pos = getCanvasCoordinates(e, canvas[1]);
    var square = getSquare(pos.x, pos.y);
    sendShot(square);
  }
});
```

7 Game

In questa sezione vedremo in modo sommario i file che implementano la logica di base del gioco di battaglia navale.

Questi file .js si trovano all'interno della sottocartella game.

Il file **game.js**, da non confondere con quello contenuto nella sottocartella *public*, contiene tutte quelle funzioni legate alla realizzazione del gioco, in particolare:

- la funzione **BattleshipGame(id, idPlayer1, idPlayer2)**: crea una sessione di gioco, assegnandole un identificativo id, i cui partecipanti sono identificati da idPlayer1 ed idPlayer2;
- alcune funzioni che restituiscono informazioni legate al player, quali l'id, lo score, se l'utente ha colpito o meno una nave, ecc.;
- altre funzioni legate alla sessione di gioco, come ottenere la griglia di gioco con le navi, oppure aggiornare lo stato del gioco, ecc.

Il file **palyer.js** contiene tutte le funzioni che riguardano il giocatore durante una sessione di gioco, ad esempio:

- la funzione **Player(id)** è il costruttore di un giocatore identificato da id;
- tutte le funzioni che riguardano le navi, come la loro creazione, il loro posizionamento randomico in una griglia, la verifica dell'affondamento o meno di una nave quando viene colpita e la verifica del numero di navi rimanenti al giocatore che determinano poi la sconfitta quando questo valore è pari a zero.
- la funzione associata al calcolo del punteggio del giocatore quando quest'ultimo colpisce una nave, mette a segno più colpi consecutivi o non colpisce nessun obiettivo:

Algoritmo 12 Funzione per il calcolo del punteggio

```
if(this.shipGrid[gridIndex] >= 0) {
    // Hit!
    this.multipleHit ++;
    this.score += (this.multipleHit * 20);
    this.shot=1;
    this.ships[this.shipGrid[gridIndex]].hits++;
    this.shots[gridIndex] = 2;
    return true;
  } else {
    // Miss
    this.multipleHit = 0;
    this.shot=0;
    this.shots[gridIndex] = 1;
    return false;
  }
```

–

Il file **settings.js** contiene le impostazioni riguardanti la dimensione della griglia (10x10) e l'array di navi la cui lunghezza indica il numero di navi, mentre il valore di ogni elemento specifica la dimensione (in termine di cella della griglia) della nave.