

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA



Relazione Progetto del Corso **Sistemi Peer-to-peer** A.A. 2017/2018

Antonio Faienza

0000798822

antonio.faienza@studio.unibo.it

Introduzione

I sistemi **peer-to-peer** rappresentano un'architettura logica di rete in cui i nodi non sono gerarchizzati unicamente sotto forma di client o server fissi, ma anche sotto forma di nodi equivalenti o 'paritari' (peer), potendo fungere al contempo da client e server verso altri nodi. Sono molteplici le applicazioni che sfruttano questo modello come *Spotify*, *Skype* oltre ad applicazioni di file sharing: *eMule*, *Torrent* e così via.

Lo scopo di questo progetto è quello di dare una chiara esemplificazione di come questi sistemi funzionano. Per la sua realizzazione si è scelto di implementare un gioco. Ci si poteva concentrare su tante altre idee di progetto come ad esempio i Bitcoin e le Blockchain che negli ultimi tempi vanno per la maggiore, ma la scelta si è soffermata sulla possibilità di implementare un gioco in modalità Multiplayer, perché come le guide in rete suggeriscono¹, la loro realizzazione può essere di non facile riuscita in considerazione del fatto che occorre un server (più o meno potente a seconda dei casi) che gestisca gli stati di gioco. Per questo, ad esempio, qualsiasi operazione eseguita da un giocatore deve essere trasmessa ad un server, il quale provvederà a ritrasmetterla al giocatore avversario. Ciò potrebbe portare ad un eccessiva *latenza* e congestione della rete. Al contrario, con un sistema peer to peer, si cerca di non impattare eccessivamente sulle performance, essendo la comunicazione diretta, cercando di ottenere una *user experience* soddisfacente.

L'idea di gioco è stata quella di realizzare un labirinto. L'utente dovrà cercare di raggiungere l'obiettivo finale: un tesoro², cercando di superare gli ostacoli che si interpongono. Per farlo, ha a disposizione tre vite, dopo delle quali si decreterà la fine della partita o la vittoria, a seconda dei casi.

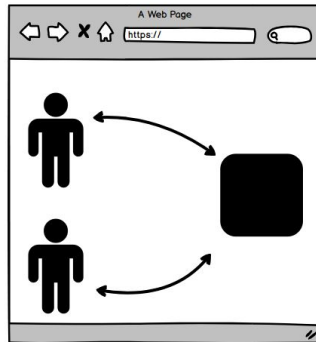
Definita l'idea, si è dovuto decidere come implementare il multiplayer. Tra le varie opzioni disponibili la scelta si è soffermata tra due principali:

1. il giocatore A gioca insieme al giocatore B e cercano di raggiungere lo stesso obiettivo finale;
2. il giocatore A gioca contro il giocatore B e gareggiano per arrivare prima dell'avversario alla fine del gioco.

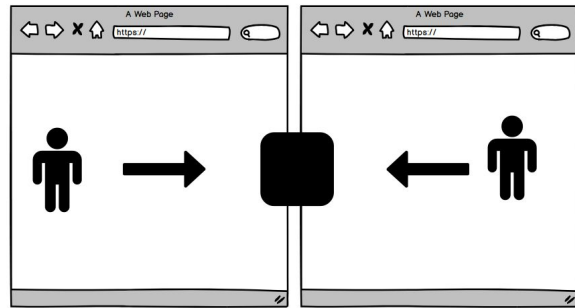
¹ <http://www.html5gamedevs.com/topic/19158-multiplayer-tips-and-resources/>

² muovendosi con le frecce

Soluzione 1



Soluzione 2



Tra le due la più facile da realizzare è sicuramente la Soluzione 2, perché implica un minor numero di casi da gestire. Al contrario la Soluzione 1 è sì molto più complessa da realizzare, perché come già asserito, qualsiasi operazione, che va dal semplice movimento, alla perdita di una vita deve essere trasmessa al giocatore avversario, ma allo stesso tempo è più affascinante perché risalta quelle che sono le potenzialità dei sistemi Peer-to-peer mettendo alla prova la tecnologia che ne è alla base. Per questa serie di ragioni si è optato per la Soluzione 1.

Nella prima parte della relazione, verrà descritta la struttura del progetto, quali tecnologie sono state adoperate per realizzare il gioco. Nella seconda parte invece ci si concentrerà esclusivamente sulla realizzazione dell'architettura Peer-to-peer fino ad arrivare alle Conclusioni dove si capirà quali sono i vantaggi e gli svantaggi del loro utilizzo.

Tecnologie usate

Per la realizzazione del progetto si è usato:

- **Xampp**³⁴⁵: è una piattaforma software per l'esecuzione del codice in locale;
- **Phaser**⁶: un framework open source veloce, gratuito e divertente realizzato per i giochi online;
- **Tiled**⁷: un editor di mappe gratuito e flessibile;
- **PeerJS**⁸: libreria che sfrutta il WebRTC ed offre una connessione peer-to-peer completa configurabile e di facile utilizzo;

³ <https://wiki.ubuntu-it.org/Server/Xampp>

Per avviare il server digitare: `sudo /opt/lampp/lampp start`

I file sono contenuti nella directory `/opt/lampp/htdocs`

⁴ Se si volesse installare il Control manager:

<https://askubuntu.com/questions/444623/xampp-control-panel-can-not-launch>

⁵ per lanciare il file dal browser: `http://localhost/nomeProgetto/`

⁶ <https://phaser.io/>

⁷ <https://www.mapeditor.org/>

⁸ <https://peerjs.com/>

- **Bootstrap**⁹: un toolkit open source per lo sviluppo con HTML, CSS e JS;
- **Heroku**¹⁰: piattaforma Cloud dove è possibile fare il deploy del proprio server.

Struttura del progetto

Affinché il progetto possa essere compreso, si è creata una repo su Github¹¹. La struttura complessiva è la seguente:

```
| -- SP2P
    |-- asset/
    |-- chat/
    |-- css/
    |-- game/
    |-- lib/
    |-- P2P/
    |-- util/
    |-- index.html
    |-- README.md
```

All'interno sono presenti oltre ai file e alle directory citate, anche quelli generati dal server, realizzato in Expressjs, come package.json e la cartella node_modules.

Tiled¹²

Tiled è un *map editor* creato da Thorbjørn Lindeijer che consente di creare livelli basati su *tile*¹³ : piccole immagini quadrate (o, molto meno spesso, rettangoli, parallelogrammi o esagoni) disposte in una griglia.

Il formato file usato è TMX. Per utilizzare questi livelli in Phaser, è necessario esportarli come JSON.

Per la realizzazione di questo gioco è stata creata una mappa di tiles di dimensione 70 x 40. Ogni tile ha a sua volta dimensione 16 x 16. Per cui la grandezza finale della mappa sarà di 1120 x 640. Una volta settata la dimensione, apparirà una finestra vuota

⁹ <https://getbootstrap.com/>

¹⁰ <https://dashboard.heroku.com/apps>

¹¹ <https://github.com/antoniofaienza93/SP2P>

¹² <https://www.mapeditor.org/>

¹³ https://en.wikipedia.org/wiki/Tile-based_video_game

in cui è possibile aggiungere vari tipi di livelli¹⁴. Quelli usati nel progetto sono essenzialmente due:

- **Tile layer**: layer fatto di tiles
- **Objects layer**: layer in cui si creano oggetti vettoriali che possono contenere metadati.

I livelli creati nel progetto in esame sono:

- **objectLayer**: rappresenta il layer di tipo Object che permette l'aggiunta degli items. Questi ultimi sono immagini da dover caricare nella mappa della stesse dimensioni delle tile.
- **blockedLayer**: sono gli elementi mobili sui quali è abilitata la fisica del gioco e con i quali il giocatore può urtare.
- **backgroundLayer**: rappresenta il livello di sfondo.

Una volta definiti i Layer è necessario iniziare a popolarli caricando le immagini di cui si necessita o alternativamente, caricando una *TileMap*. Essa rappresenta una immagine al cui interno contiene svariati disegni, che sono utilizzabili nel gioco. Serve per non caricare singolarmente le immagini da utilizzare. Generalmente si usano per dare all'utente che le adopera un tema di disegno su cui basarsi. Ogni immagine che viene selezionata dalla Tilemaps o ogni singola immagine caricata, deve possedere necessariamente la stessa dimensione di una singola tile. In questo caso quindi sono state usate immagini 16x16 scaricabili da [OpenGameArt.org](https://opengameart.org/)¹⁵¹⁶ o da qualsiasi altro repository di immagini¹⁷.

Scelta e caricata la *TileMap* che comparirà nella parte destra dello schermo, selezionando la tile desiderata, è possibile riportarla nella mappa ancora vuota andandola a posizionare dal livello di tipo **Tile** più basso (*backgroundLayer*) fino a quello più alto (*blockedLayer*).

Discorso a parte per l'*objectLayer*. In questo specifico caso, per inserire nuovi oggetti, si deve:

- ❖ selezionare il layer *objectsLayer*;
- ❖ fare clic sul pulsante *Insert Tile*¹⁸;
- ❖ cliccare sulla tile che si desidera usare;
- ❖ posizionare l'oggetto all'interno della mappa.

In che modo Phaser saprà cosa rappresentano questi elementi? Per farlo, bisogna aggiungere le proprietà agli oggetti.

¹⁴ L'ordine di aggiunta del livello è fondamentale e può essere usato riordinato usando i comandi appositi.

¹⁵ <https://opengameart.org/>

¹⁶ <https://en.wikipedia.org/wiki/OpenGameArt.org>

¹⁷ Per questo progetto è stato usato in alternativa a OpenGameArt: <https://www.iconfinder.com/> o <https://www.flaticon.com/>

¹⁸ quello che mostra l'immagine di un tramonto

Dopo essersi posizionati sull'*objectLayer* e selezionato un oggetto già posizionato sulla mappa, nella parte bassa a sinistra è possibile aggiungere le proprietà da conferire all'oggetto. Per il giocatore, è stato aggiunto il campo `type` di tipo `String` e conferito il valore di `playerStart`. Per tutti gli altri oggetti invece, sono state definite tre proprietà: `type`, `sprite`, `order`. Il primo con valore `item` per ognuno, indicava che l'oggetto era uno degli ostacoli del labirinto. Il secondo indicava il nome assegnato, mentre il terzo indicava l'ordine di raccolta.

Una volta ultimata la propria mappa, la si può esportare in formato JSON per poter essere utilizzata in Phaser¹⁹.

Phaser

Phaser è un framework HTML5 che ha lo scopo di aiutare gli sviluppatori a realizzare giochi potenti e multibrowser in maniera veloce. È stato creato appositamente per sfruttare i vantaggi dei browser moderni, sia desktop che mobili. L'unico requisito del browser è il supporto del tag `canvas`.

Esistono varie versioni. Per la realizzazione di questo progetto è stata usata la **Community Edition**²⁰ ossia la versione Open Source. La CE rappresenta un proseguimento della seconda versione che si slega completamente dalla terza. Questa scelta è dettata da un'esigenza di cambiamento da parte degli sviluppatori, ma allo stesso tempo è data la possibilità di continuità di utilizzo e aggiornamento ad una versione ancora ampiamente usata.

Di qui in avanti verranno descritti brevemente i file necessari per comprendere la struttura portante del gioco.

Main.js

Il file `main.js` funge da snodo e contiene le configurazioni di cui il gioco necessita e ha il compito di avviare il primo file: il *Boot State*.

La prima operazione effettuata è stata la creazione di un *namespace* unico in modo da evitare conflitti con altre librerie che si sarebbero potute utilizzare. In questo progetto lo spazio dei nomi è `P2PMaze`. Il nome è a scelta dello sviluppatore a patto che sia univoco e improbabile da trovare altrove. In generale significa che se l'oggetto esiste già, lo si userà. Altrimenti verrà creato un nuovo oggetto.

Successivamente si sono impostate le dimensioni dell'intera finestra. `Phaser.AUTO` indica che il rendering del gioco poggia su un elemento automaticamente individuato

¹⁹ Per evitare problemi con la lettura del file JSON in phaser si consiglia di incorporare tutte le immagini nella tile. Questa operazione è possibile effettuarla selezionando l'apposito pulsante nella parte in basso a destra

²⁰ <https://phaser.io/download/phaserce>. Rappresenta la versione Open Source di Phaser tenuta in costante aggiornamento.

tra tre scelte: CANVAS, WebGL o HEADLESS²¹. Se WebGL è disponibile, è utilizzato come prima opzione poiché fornisce prestazioni migliori.

```
var config = {
  type: Phaser.AUTO,
  width: SCREEN_SIZE.WIDTH,
  height: SCREEN_SIZE.HEIGHT,
  parent: 'P2PMaze'
};

var P2PMaze = P2PMaze || {};

P2PMaze.game = new Phaser.Game(config);

P2PMaze.game.state.add('Boot', P2PMaze.Boot);
P2PMaze.game.state.add('Preload', P2PMaze.Preload);
... other states ...

P2PMaze.game.state.start('Boot');
```

States methods

Nell'implementazione di un gioco con Phaser, esistono metodi riservati che servono a scopi specifici. Questi sono denominati **States methods**. Ce ne sono di svariati²², ma i più importanti sono:

- **Init**: richiama l'inizializzazione dello stato. Può essere usato per inviare parametri allo stato successivo/precedente.
- **Preload**: vengono caricate le risorse.
- **Create**: chiamato alla fine del completamento degli asset.
- **Update**: chiamato ad ogni ciclo di gioco. E' qui che si desidera includere elementi che devono essere costantemente testati come il rilevamento delle collisioni.

Stati

Phaser fino alla versione 2 offre l'organizzazione del gioco in Stati. Ce ne possono essere di svariati e li si può pensare come a fasi di gioco. Per esempio durante un concerto c'è la fase di apertura dei cancelli, la fase di controllo dei ticket, lo spettacolo e così via. Non è obbligatorio il loro uso, e a tal proposito nella terza versione il loro utilizzo non è più ammesso²³. Tuttavia, anche in virtù del fatto che si è utilizzata la

²¹ <https://phaser.io/docs/2.4.4/Phaser.Game.html>

²² <http://www.html5gamedevs.com/topic/1372-phaser-function-order-reserved-names-and-special-uses/>

²³ <http://www.html5gamedevs.com/topic/37926-tutorials-on-phaser-3-basicsbest-practices/>

seconda versione del framework, rappresentano un ottimo modo per organizzare il codice.

Boot State

Nel **Boot State** vengono definite le impostazioni generali del gioco e caricate le risorse dello stato successivo (ad esempio una *loading bar*). In questo frangente, nulla viene mostrato all'utente.

Preload State

Nel **Preload State** le risorse di gioco (immagini, sprite sheets, audio, trame, ecc.) sono caricate in memoria. Una schermata di caricamento, precedentemente caricata, può mostrare lo stato di avanzamento di tale operazione.

Main Menu

Il **Main Menu State** rappresenta la schermata di benvenuto al gioco. Dopo il preload, tutte le immagini di gioco sono già caricate nella memoria, in modo che possano essere rapidamente accessibili.

Game State

Il **Game State** è la parte vera e propria del gioco, dove è implementata la logica, la struttura e il divertimento prende piede. Qui di seguito ci si andrà a analizzare i metodi salienti.

Nel metodo *create* si abilita la fisica del gioco, utilizzati tutti gli asset precedentemente caricati nel Preload, tra i quali la mappa, e creati i *Layer*. Inoltre vengono caricate le *lives* e gli *items*²⁴ da dover collezionare per completare il gioco. In particolare in quest'ultimo caso, per definire la logica del gioco, si aggiunge l'item ad un *HashMap* precedentemente istanziata. Tale struttura dati avrà come chiave il nome del *items* e come valore l'ordine di “raccolta” secondo quelle che sono le proprietà stabilite in Tiled.

```
P2PMaze.Game.prototype = {
  create: function () {

    this.game.physics.startSystem(Phaser.Physics.ARCADE);
    map = this.game.add.tilemap('maze');
    map.addTilesetImage('tiles', 'mazeImage');
    backgroundLayer = map.createLayer('backgroundLayer');
```

²⁴ All'interno del metodo vi è un ciclo che legge tutti gli elementi di un determinato livello della Mappa creata e caricata in Tiled, con una specifica proprietà e vi crea l'oggetto, denominato *Sprite*, a cui vengono conferite caratteristiche necessarie al gioco.

```

        blockedLayer = map.createLayer('blockedLayer');
        this.createItems();
        this.createLives();
        result = this.findObjectsByType('playerStart', map, 'objectLayer');
        // create the player
        player = this.game.add.sprite(result[0].x, result[0].y, 'player');
        ...
        ...

        player.animations.add('left', [0, 1, 2, 3], 10, true);
        player.animations.add('right', [5, 6, 7, 8], 10, true);

    },

```

Inoltre si aggiunge il personaggio al gioco a cui vengono date caratteristiche di movimento²⁵.

Nella funzione `update`, che esegue innumerevoli cicli al secondo, vengono rilevati le collisioni e viene implementa l'animazione che conferisce movimento al personaggio oltre ad essere implementata e utilizzata la logica del gioco.

Logica del Gioco

Come appena accennato ad ogni ciclo si verifica che tutto ciò che compie il personaggio rispetti la logica del gioco.

```

update: function () {
    ...
    ...
    this.game.physics.arcade.overlap(player, items, this.collect,
                                     this.choiceItems, this);
},

```

Questa riga di codice verifica quando e quali *items* effettuano l'overlapping con il *player*. Sulla base di questa sovrapposizione viene richiamata la funzione/callback `choiceItems`, la quale solo e soltanto se ritorna `true`, richiamerà l'altra *callback* `collect`.

La funzione `choiceItems` dato il giocatore e uno degli *items* verifica se tale oggetto sia presente nell'Hasmap, e se il suo valore sia uguale al `playerOrder` (variabile globale che tiene conto dell'ordine di acquisizione). In tal caso viene aumentato il `playerOrder`, eliminato l'item dall'hashmap e ritornato `true`. In caso contrario viene scalata una vita e ritornato `false`.

²⁵ E' da sottolineare come il personaggio, nel Preload, è stato caricato non come un'immagine, ma come una *spritesheet* (`this.load.spritesheet(...)`). Questo perché contiene frame di animazione. Aprendo l'immagine si nota che ci sono 9 fotogrammi in totale, 4 per correre a sinistra, 1 fronte alla telecamera e 4 per correre a destra. - <https://phaser.io/examples/v2/tilemaps/tileset-from-bitmapdata>

```

choiceItems: function (player, item) {

    if ((item.key in logicalOrder) && logicalOrder[item.key] ===
                                                playerOrder) {

        itemCorrect.play();
        playerOrder++;                // icrease the order of player
        delete logicalOrder[item.key]; // delete key-value
        return true;
    } else {
        this.game.physics.arcade.collide(player, items);
        this.decreaseLive();
        return false;
    }
},
...
...

```

La funzione `collect` elimina l'*item* individuato, lo aggiunge al `div` contenente gli elementi presi e verifica che non sia l'ultimo item da prendere. In tal caso verrà lanciata la schermata di vittoria.

Peer to Peer

Dopo aver descritto come è stato costruito il gioco, ci si può concentrare sulla parte focale del progetto: l'implementazione del Multiplayer, realizzato utilizzando `PeerJS`, una libreria che sfrutta WebRTC.

WebRTC è una tecnologia che consente la comunicazione in tempo reale tra i browser web. È relativamente nuovo e la definizione delle API è ancora considerata come una bozza. In aggiunta, WebRTC non è ancora supportato da tutti i principali browser Web²⁶ ²⁷ (e tra quelli che lo fanno, alcuni di essi non supportano tutte le funzionalità di questa tecnologia), il che lo rende relativamente difficile da utilizzare per qualsiasi tipo di applicazione.

²⁶ <https://caniuse.com/#feat=rtcpeerconnection>

²⁷ <http://iswebrtcreadyyet.com/>

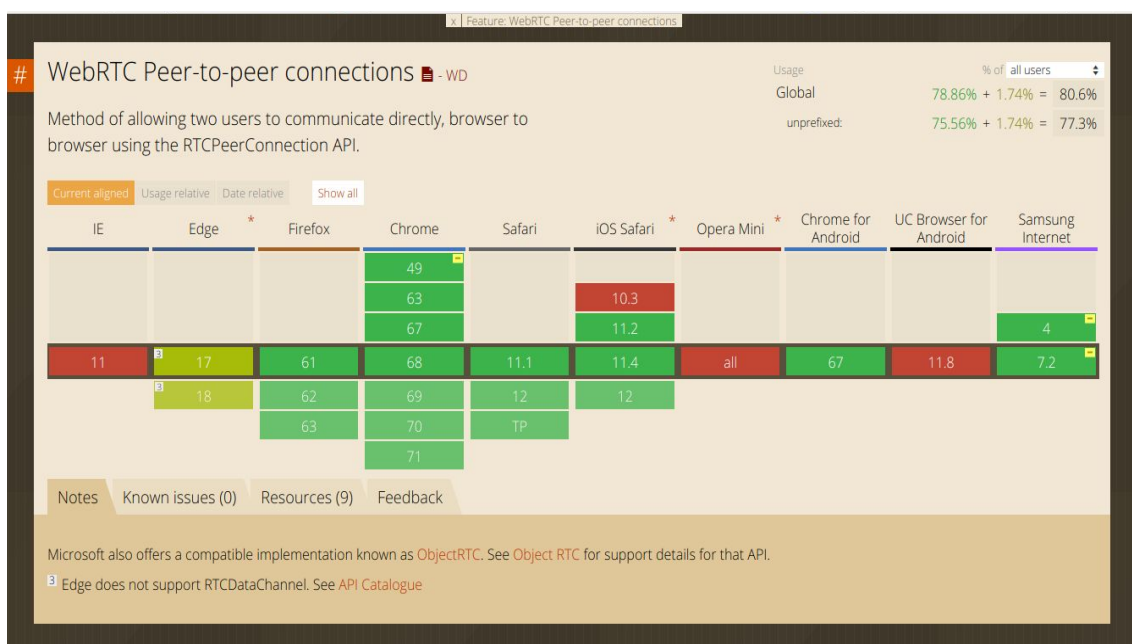
Is WebRTC ready yet?



There are lots of issues and bugs remaining of course. [Report bugs](#) when that is not the case or use a shim like [adapter.js](#) until implementations match the specification.

If you have any questions, use the [discuss-webrtc mailing list](#).

If you are looking for the scorecard that used to be on this site, you can find it [here](#).



Da quando è stato introdotto per la prima volta da Google nel maggio 2011, WebRTC è stato usato in molte moderne applicazioni web. Essendo una caratteristica fondamentale di molti browser Web moderni, le applicazioni Web possono trarre vantaggio da questa tecnologia per offrire un'esperienza utente migliorata in molti modi. Le applicazioni di streaming video o di conferenza possono sfruttare le reti peer-to-peer (P2P) trasmettendo direttamente al loro interlocutore (senza usufruire di un server), ma questo è solo un esempio di quello che può essere raggiunto con WebRTC. E' importante capire che questa tecnologia non riguarda esclusivamente la

trasmissione di flussi audio e video. Fornisce inoltre supporto per i canali²⁸ dei dati P2P. Questi canali sono disponibili in due varianti: *affidabili*²⁹ e *inaffidabili*³⁰.

Ciò apre la porta a molti altri casi di utilizzo, inclusi, a titolo esemplificativo, giochi multiplayer, distribuzione di contenuti e condivisione di file. Di nuovo, tutto senza la necessità di alcun server intermedio e quindi con latenze più basse.

PeerJS utilizza l'implementazione di WebRTC del proprio browser facilmente in maniera configurabile. Permette di far fronte ai bug implementativi dei browser precedenti³¹. Ad esempio, in Chrome 30 o versioni precedenti, con WebRTC erano disponibili solo canali dati inaffidabili. Con PeerJS, l'identificazione degli altri peer è molto semplice. Con le ultime versioni ciò avviene nient'altro che con un ID: una stringa che il peer può scegliere autonomamente o che un server gli può generare. Sebbene WebRTC prometta comunicazioni peer-to-peer, è comunque necessario un server che agisca da broker di connessione e gestisca i segnali. `PeerServer` rappresenta un'implementazione open source di questo tipo di server (scritto in Node.js)³².

Oltre `PeerJS` esistono altre alternative in rete^{33 34}, ma la scelta finale si è incentrata sul suo utilizzo per il suo largo uso e per la possibilità che offre di mettere a disposizione una Github issues page³⁵ estremamente efficiente.

I file di interesse in questa sezione sono:

- ❖ `P2P/PeerServerExpress.js`: rappresenta il server. Implementato con `PeerServer` e deployato con Heroku³⁶;
- ❖ `P2P/PeerClient.js`: classe che definisce un Peer;
- ❖ `P2P/managePeerCommunication.js`: file che gestisce la comunicazione tra due client;
- ❖ `game/GameMultiplayer.js`: file che implementa il Multiplayer State.

Qui di seguito si andrà a descrivere nel dettaglio cosa fanno i vari file mettendo in luce quelli che sono gli aspetti peculiari.

²⁸ https://developer.mozilla.org/en-US/docs/Games/Techniques/WebRTC_data_channels

²⁹ I canali affidabili garantiscono che i messaggi inviati arrivino all'altro peer e nello stesso ordine in cui vengono inviati. Questo è analogo a un socket TCP.

³⁰ I canali inaffidabili non offrono garanzie; non è garantito un esatto ordine di arrivo dei messaggi e, di fatto, non è garantito che arrivino a tutti. Questo è analogo a un socket UDP.

³¹ <https://peerjs.com/status/>

³² Alternativamente `PeerServer Cloud` <https://peerjs.com/peerserver>

³³ <https://github.com/feross/simple-peer>

³⁴ https://dev.to/rynobax_7/creating-a-multiplayer-game-with-webrtc

³⁵ <https://github.com/peers/peerjs/issues>

³⁶ Si veda l'Appendice per maggiori informazioni.

PeerServerExpress.js

Come il sito ufficiale riporta, PeerServer fa da intermediario tra le connessioni dei vari client. Ciò significa che, sebbene due client comunichino direttamente tra di loro, necessitano di un server che generi una chiave al Peer (se quest'ultimo non ne dispone di una) e supervisioni le successive comunicazioni fra di essi.

Il nostro server, è integrato con Express.js³⁷. Tale scelta è frutto della volontà di voler implementare un servizio di Peer Discovery³⁸. Essa consente di individuare i peer disponibili per il trasferimento di dati in una rete P2P. Questi ultimi possono essere all'interno di una rete locale o su una rete remota. Dopo aver installato³⁹ e settato il server basta lanciarlo con il comando `node PeerServerExpress.js`. Da questo momento il server sarà attivo. I seguenti metodi sono perennemente in ascolto:

```
function connectPeer(callback) {
  peerserver.on('connection', function (id) {
    callback(id);
  });
}

function disconnectPeer(callback) {
  peerserver.on('disconnect', function (id) {
    console.log("Disconnected user " + id);
    callback(id);
  });
}
```

Il primo individua quelli che sono i peer che si connettono mentre il secondo, quelli che si disconnettono. Entrambi i metodi dispongono di una *callback*⁴⁰ utile per aggiungere o eliminare uno specifico peer da un array globale.

Le altre funzioni del server sono delle chiamate `get` utili al client (come il peer discovery o come la generazione di Random String da assegnare al peer che lo richiede). Qui di seguito, uno snippet di codice esemplificativo:

```
// the server send to client the peer available (even himself)
app.get('/available-peer', function (req, res, next) {

  //Website you wish to allow to connect
  res.setHeader('Access-Control-Allow-Origin', '*');
```

³⁷ <https://github.com/peers/peerjs-server#combining-with-existing-express-app>

³⁸ <https://www.techopedia.com/definition/25772/peer-discovery>

³⁹ `npm install` // installazione di npn
`npm install express --save` // installazione del server Express
`npm install peer --save` // installazione di PeerServer
`npm install randomstring --save` // installazione di Random String

⁴⁰ <https://github.com/maxogden/art-of-node#callbacks>

```

// Request methods you wish to allow
res.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH,
          DELETE');

// Request headers you wish to allow
res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');

// Set to true if you need the website to include cookies in the requests sent
// to the API (e.g. in case you use sessions)
res.setHeader('Access-Control-Allow-Credentials', true);

res.setHeader('Content-Type', 'application/json');

res.send(JSON.stringify(peer_available));
// Pass to next layer of middleware
next();

});

```

In questo frammento di codice, vengono definiti gli Header⁴¹ e nel momento in cui il peer lo richiede, il server manda quelli che sono i peer connessi (compreso quello richiedente⁴²).

PeerClient.js

Per l'implementazione del PeerClient si è optato per la realizzazione di una classe. Quest'ultima definisce un costruttore:

```

class PeerClient {

  constructor(id, h, p, pth) {
    this._id = id;
    this._host = h;
    this._port = p;
    this._path = pth;
    this._conn = undefined;

    this._peer = new Peer(this._id, {
      host: this._host, // 'localhost',
      port: this._port, // 9000,
      path: this._path // '/peerjs'
    });

    this._peer.on('open', function (id_peer) {
      this._id = id_peer;
    });
  }
}

```

⁴¹ <https://stackoverflow.com/questions/18310394/no-access-control-allow-origin-node-apache-port-issue>

⁴² Sarà managePeerCommunication.js ad escludere il peer requestor.

Come si può osservare prende come parametro:

- ❖ l'id generato dal server;
- ❖ l'indirizzo del server
- ❖ la porta⁴³
- ❖ il percorso in cui è in esecuzione il Server.

Inoltre dopo aver creato un nuovo Peer, è necessario aprire la connessione⁴⁴ prima di potersi connettere con altri.

Il restante codice della classe serve per definire i metodi necessari per la comunicazione. Tra questi sicuramente vi è `conn(id_another_peer):`

```
conn(id_another_peer) {  
    this._conn = this._peer.connect(id_another_peer);  
    return this._conn;  
}
```

il quale, dato come parametro un id, permette di stabilire una connessione con un altro peer; `openConnection(data):`

```
openConnection(data) {  
    this._conn.on('open', function () {  
        this.send(data);  
    });  
}
```

che si premura di aprire la connessione con il peer con cui è stata stabilita la connessione; e `send(data):`

```
send(data) {  
    this._conn.send(data);  
}
```

che permette di inviare qualsiasi tipo di dato sulla base della connessione stabilita.

Infine per abilitare la ricezione si usa il metodo `enableReceptionData(callback)` Esso è in ascolto dei messaggi che riceve e li comunica al client con l'ausilio di una callback:

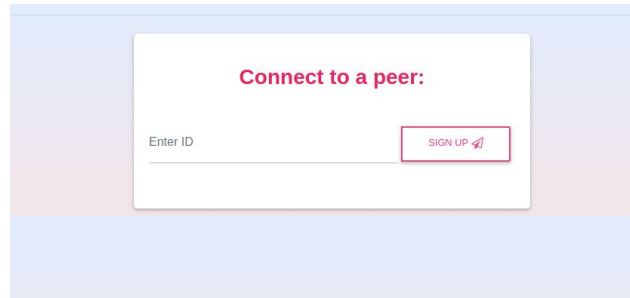
```
enableReceptionData(callback) {  
    this._peer.on('connection', function (conn) {  
        conn.on('data', function (data) {  
            console.log("-----");  
            console.log("MESSAGE RECEIVED : \n");  
            console.log(data);  
            console.log("-----");  
            callback(data);  
        });  
    });  
}
```

⁴³ Se si sta utilizzando il cloud PeerServer assicurarsi che la porta 9000 (porta di default) non sia bloccato.

⁴⁴ <https://github.com/peers/peerjs/issues/431#issuecomment-407459118>

ManagePeerCommunication.js

Durante la scrittura del codice si è reso necessario l'utilizzo di un gestore che coordinasse la creazione dei Peer e le operazioni che essi potevano essere in grado di eseguire. Il file `managePeerCommunication.js` assolve questa funzione. E' grazie a questo file che vengono effettuate le chiamate al server, (definendo il comportamento di Peer Discovery) e creati i Peer. Il tutto mediante un form.

A screenshot of a web form titled "Connect to a peer:". The form is white with a thin border and is centered on a light blue background. It contains a text input field with the placeholder text "Enter ID" and a button labeled "SIGN UP" with a right-pointing arrow icon.

Per gestire la comunicazione tra Peer si è reso obbligatorio la “registrazione” al server. Nel momento in cui un utente preme il pulsante *Sign Up*, viene creato un nuovo `PeerClient` usando come chiave la stringa generata dal server più il nome scelto:

```
function createPeerClient(data) {  
  
    var input = document.getElementById("inputid");  
    if (input.checkValidity()) {  
        ...  
        peerClient = new PeerClient(input + "-" + data, PEER_CLIENT.HOST,  
                                    PEER_CLIENT.PORT, PEER_CLIENT.PATH);  
  
        // enable reception of data  
        peerClient.enableReceptionData(dataReceived);  
  
        // we see the error of the server  
        peerClient.onError(handleServerError);  
        ...  
    }  
}
```

A questo punto, si è in grado di ricevere messaggi oltre ad essere in grado di visualizzare i peer disponibili.

Connect to a peer:

antonio

SIGN UP

USERNAME: antonio-0mswDSm

Join to a peer

SEE PEER AVAILABLE

☐ paola-NTJefjy

CHOICE PEER TO CONNECT

Connect to a peer:

paola

SIGN UP

USERNAME: paola-NTJefjy

Join to a peer

SEE PEER AVAILABLE

☐ antonio-0mswDSm

CHOICE PEER TO CONNECT

Nell'ipotesi in cui esista almeno un peer in più rispetto al nostro utente, si è in grado di poterlo selezionare da una checkbox e avviare la comunicazione:

Connect to a peer:

antonio

SIGN UP

USERNAME: antonio-lhyJRWv

Join to a peer

SEE PEER AVAILABLE

Il peer paola-nujSMfN want to connect with you

☐ YES

☐ NO

CHOICE PEER TO CONNECT

In questo frangente si effettuerà quindi una richiesta di connessione:

```
function requestConnection(peerSelected) {
  peerClient.conn(peerSelected);
  peerClient.openConnection(peerClient.getId());
}
```

Uno dei metodi cardini di tutto il file, è rappresentato dalla callback `dataReceived(data)`:

```
function dataReceived(data) {
  // For the first time, the connection is undefined. For this reason,
  // it's created a form for send the request of connection
  if (peerClient.getConnection() == undefined) {
    ...
    ...
    // Creation form
    // add item of peer available
    var dc;
    var decideToConnect = ["YES", "NO"];
    for (var i = 0; i < decideToConnect.length; i++) {
      ...
    }
  }
}
```

```

        ...
    }

    // add button to checkbox
    var buttonChoice = createButton("button", classItemForm,
FORM.CHOICE_PEER,

                                data, callbackConnectionChoicePeer)

    ...
    // in other cases, when the connection is establish, then:
    // - can be accepted
    // - can be refused
    // - it's a normal data. In this case:
    //     - can be the data of the multiplayer game
    //     - can be a chat message
    } else {

        if (data == PEER.CONNECTION_ACCEPTED) {
            enableGame();
        } else if (data == PEER.CONNECTION_REFUSED) {
            refuseConnection();
        }
        else {
            if (data[0].key == "CHAT_MESSAGE") {
                var m = data[1].message;
                chat.receiveMessage(peerClient.getId(), m);
            }
            else {
                P2PMaze.dataReceived = data;
            }
        }
    }
}
}

```

Questa callback ritorna al client il dato ricevuto e lo gestisce a seconda dei casi. Per comprenderla a fondo, si immagina che esistano due utenti: l'utente *P* e l'utente *A*. L'utente *P* avvia una connessione verso l'utente *A*. I possibili scenari che può gestire questo metodo sono i seguenti:

- ❖ l'utente *A* riceve una richiesta dall'utente *P*.
 - Se l'utente *A* non ha nessuna connessione in corso significa che può riceverla e quindi può visualizzare il form e decidere se accettarla o meno⁴⁵.
- ❖ l'utente *P* avendo avanzato una richiesta di connessione verso l'utente *A* deve quindi ricevere una risposta:

⁴⁵ può:

accettare la richiesta: in questo caso anche l'utente *A* stabilisce una connessione con l'utente *P*,

rifiutare la richiesta: anche in questo frangente, l'utente *A* stabilisce la connessione con l'utente *P* per potergli comunicare la sua scelta e poter consentire la chiusura della connessione da ambo le parti con il metodo `refuseConnection()`;

- `CONNECTION_ACCEPTED`: implica che la connessione è stata accettata e di conseguenza può essere avviato il gioco;
- `CONNECTION_REFUSED`: implica che la connessione è stata rifiutata e la connessione di conseguenza chiusa;

Se il dato ricevuto non è esattamente uguale alle precedenti due possibilità vuol dire che è:

- ❖ `CHAT_MESSAGE`: rappresenta la chiave di un hashmap, la quale indica che si tratta di un messaggio di chat;
- ❖ sono dati strettamente inerenti al gioco, come la posizione o la perdita di una vita.

GameMultiplayer.js

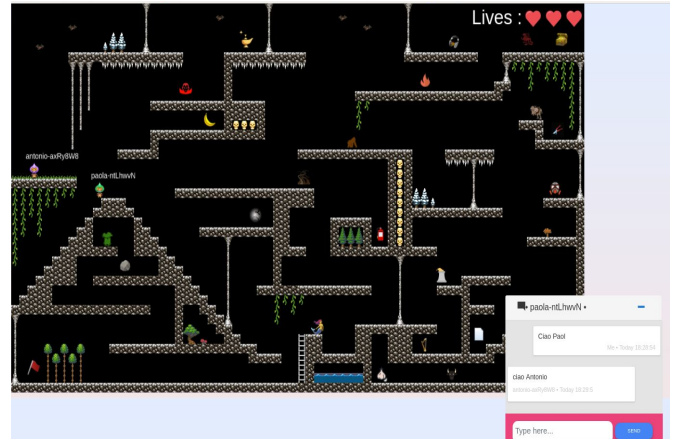
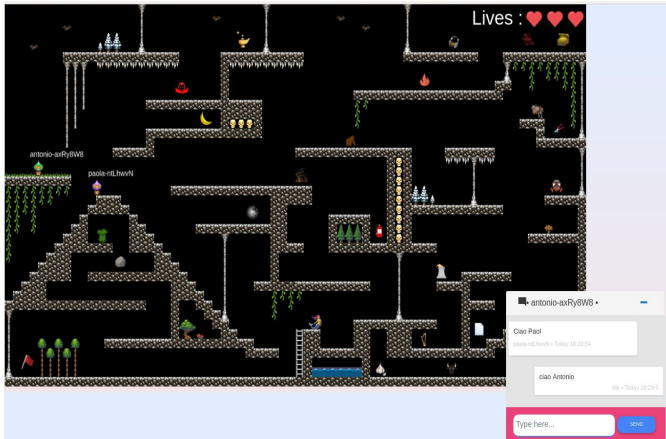
Una volta abilitata la connessione da entrambe le parti, viene chiamato il metodo `enableGame()`

```
function enableGame() {  
  
    var id_connected_peer = peerClient.getConnection().peer;  
  
    // Assignment  
    P2PMaze.peer = peerClient;  
  
    // create Chat  
    chat = new Chat();  
    chat.createChat(id_connected_peer);  
    chat.onclickSendChatButton(sendChatMessage);  
    chat.onclickKeyboard(sendChatMessage);  
  
    // start the multiplayer game  
    P2PMaze.game.state.start("GameMultiplayer");  
    ...  
    ...  
}
```

Tale metodo crea la chat, avvia il *Multiplayer State*⁴⁶, ma soprattutto assegna ad una variabile globale l'oggetto `peer` dell'utente.

⁴⁶ In questa sezione si descriveranno le parti che differiscono dal Game State. Per l'implementazione della parte Multiplayer si è partiti dal gioco Single Player e sono state aggiunte solo le parti che implicavano l'invio e la ricezione dei dati.

Il *Multiplayer State* è implementato nel file `game/GameMultiplayer.js`.



All'inizio del file si trova il metodo `send` il cui scopo è quello di inviare i dati:

```
P2PMaze.send = function (data) {  
  P2PMaze.peer.send(data);  
};
```

Nel *create state* invece è contenuto: la creazione del giocatore avversario oltre che le assegnazioni presenti in `Game.js`. Più importante di tutti è invece:

```
P2PMaze.peer.enableReceptionData((value) => dataReceived = value);
```

Con questa riga di codice si abilita il peer dell'utente a ricevere i dati. In particolare, tutto ciò che verrà ricevuto, sarà assegnato alla variabile `dataReceived`.

L'invio e la ricezione dei dati, può avvenire nei seguenti casi:

- ❖ il peer si muove (sinistra, destra e salto);
- ❖ il peer colleziona un item;
- ❖ il peer perde una vita;
- ❖ il peer perde la partita;
- ❖ il peer vince la partita;
- ❖ il peer invia un messaggio

L'invio dei dati, si effettua sfruttando un *template* che si ripete per ogni invio: viene creato un array, dove il primo oggetto è una *key*. Successivamente sono aggiunti i dati di interesse da dover spedire. Ad esempio:

```
var updatePos = [];  
var keyupdating = { "key": "left" };  
var updateX = { "updatePosX": player.x };  
var updateY = { "updatePosY": player.y };  
updatePos.push(keyupdating);  
updatePos.push(updateX);  
updatePos.push(updateY);  
P2PMaze.send(updatePos);
```

Nell'update, vengono gestiti i dati ricevuti, quindi sulla base dell'esempio precedente, si avrà:

```
if (P2PMaze.dataReceived != undefined) {
    if (P2PMaze.dataReceived[0].key == "right") {
        ...
    }
    else if (P2PMaze.dataReceived[0].key == "left") {
        var posx = P2PMaze.dataReceived[1].updatePosx;
        var posy = P2PMaze.dataReceived[2].updatePosy;

        if (Math.floor(opponentPlayer.x) === Math.floor(posx) ||
            Math.floor(opponentPlayer.x) === (Math.floor(posx) + 1) ||
            Math.floor(opponentPlayer.x) === (Math.floor(posx) - 1)) {
            opponentPlayer.animations.stop();
            opponentPlayer.frame = 4;
            // P2PMaze.dataReceived = undefined;
        }
        else {
            this.game.physics.arcade.moveToXY(opponentPlayer, Math.floor(posx),
                                                Math.floor(posy), 100);
            opponentPlayer.animations.play('left');
        }
    }
    ...
}
```

Il codice appena mostrato ci dice che se `dataReceived` contiene un dato e la sua `key` è esattamente `left`, allora tramite il metodo `moveToXY` viene spostato il giocatore avversario nella posizione determinata dai dati ricevuti. Tale movimento avviene fino a quando non si trova nell'esatta posizione del peer dell'utente⁴⁷.

Others part

Le altre parti del progetto, oltre alle librerie usate e ai file `css` per il layout, comprendono la directory `chat` e la directory `util`.

La prima, come il nome suggerisce riguarda la creazione di uno spazio per l'invio e la ricezione dei messaggi di testo. Anche in questo caso, così come per il *PeerClient*, è stata creata una classe. Nel costruttore vi è la definizione delle componenti della chat interamente scritto in Javascript⁴⁸. Gli altri metodi riguardano la ricezione dei messaggi e la gestione degli eventi.

La seconda parte invece contiene due file. `utilForm.js`: usato per creare le componenti HTML di cui il gioco necessita e `const.js`: contenitore di tutte le

⁴⁷ E' stato necessario approssimare per intero il valore della posizione sebbene il valore ricevuto fosse decimale per via del metodo `moveToXY` il quale non riusciva a collocare il giocatore avversario nell'esatta posizione indicata per via delle cifre decimali.

⁴⁸ Si è partiti dal seguente tema Bootstrap: <https://bootsnipp.com/snippets/Vg1EP>

costanti del progetto, come ad esempio i paths del server, la grandezza delle dimensioni delle finestre e così via.

Conclusioni

I sistemi peer-to-peer, come visto, rappresentano un'architettura molto potente per poter comunicare tra due o più nodi. In questa relazione, si è discusso come è stato realizzato un gioco. In particolare tale gioco, si pone come obiettivo quello di risolvere un labirinto, superando i vari ostacoli che si incontrano, fino al raggiungimento di un obiettivo finale. La parte centrale del progetto è rappresentata dall'implementazione del Multiplayer, realizzato con PeerJS: una libreria che sfrutta il WebRTC ed offre una connessione peer-to-peer completa configurabile e di facile utilizzo. Il gioco in sé è stato realizzato utilizzando Phaser: un framework open source veloce, gratuito e divertente realizzato per i giochi online.

La parte Single Player è risultata essere relativamente semplice. L'unica difficoltà è stata rappresentata dall'implementazione della logica del gioco.

Per quanto riguarda la comunicazione tra i nodi, PeerJS si è dimostrato uno strumento molto potente, ma ancora da migliorare. Infatti come si può leggere dalla sua documentazione molte delle feature che mette a disposizione sono in fase Beta e non disponibili né per tutti i Browser, né in tutte le versioni di quelli disponibili.

Dispone tuttavia di un forum estremamente efficiente che aiuta gli sviluppatori nella scrittura del codice, e a risolvere eventuali problemi che si possono incontrare.

La maggiore difficoltà avuta è stata la comunicazione tra due peer. In particolare dalla documentazione si evince che si debba aprire un nuovo canale ad ogni comunicazione:

Data connections

Connect

```
var conn = peer.connect('another-peers-id');  
// on open will be launch when you successfully connect to PeerServer  
conn.on('open', function(){  
  // here you have conn.id  
  conn.send('hi!');  
});
```

ma in una discussione avuta con uno degli autori di PeerJS⁴⁹, per avviare la comunicazione basterebbe esclusivamente stabilire la connessione:

```
/**  
 * This method is used for create a connection  
 * @param {object} id_another_peer is the id of peer that I want to connect
```

⁴⁹ <https://github.com/peers/peerjs/issues/431#issuecomment-407422647>

```

*/
conn(id_another_peer) {
  this._conn = this._peer.connect(id_another_peer);
  return this._conn;
}

```

e successivamente inviare esclusivamente i dati:

```

/**
 * This method is used for sharing data between browser
 * @param {*} data - data to exchange
 */
send(data) {
  this._conn.send(data);
}

```

Ma questa soluzione non risulta funzionare, per cui in accordo con l'autore si è optato per la soluzione utilizzata e spiegata nella sezione PeerClient.js che prevede la definizione di una connessione: `conn(id_another_peer)`; l'apertura di tale connessione: `openConnection(data)` e infine l'invio dei dati: `send(data)`.

Un altro problema che si è avuto riguardava l'impossibilità di inviare oggetti troppo complessi. L'invio di un oggetto *Sprite*, tipico nell'utilizzo di Phaser, avrebbe alleggerito di molto la gestione della ricezione dei dati.

Infine, spulciando la documentazione e da una discussione⁵⁰ è emerso come il server non offre nessuna callback che segnali al peer che il suo interlocutore non è più online. Allo stato attuale delle cose, se uno dei due giocatori non è più online semplicemente non comunica più con esso⁵¹.

⁵⁰ <https://github.com/peers/peerjs/issues/442#issuecomment-417270241>

⁵¹ Ai fini del gioco, questa mancanza voleva essere gestita col polling. Il client avrebbe dovuto chiedere ad intervalli regolari, se il suo avversario fosse ancora online. In caso di risposta negativa, lo stesso giocatore veniva eliminato e in aggiunta cancellata la chat. Tuttavia, avendo un server il cui timeout è di 55 secondi, allo scadere, veniva svuotato l'array Di conseguenza tutti i peer cancellati e come risultato finale, il peer avversario di ogni singolo giocatore veniva eliminato.

Riferimenti

- [1] <https://peerjs.com/>
- [2] <https://github.com/peers/peerjs-server>
- [3] <https://webrtc.org/>
- [4] <https://gamedevacademy.org/html5-phaser-tutorial-spacehipster-a-space-exploration-game/>
- [5] <http://perplexingtech.weebly.com/game-dev-blog/using-states-in-phaserjs-javascript-game-development>
- [6] <http://www.emanueleferonato.com/2014/08/28/phaser-tutorial-understanding-phaser-states/>
- [7] <https://phaser.io/tutorials/making-your-first-phaser-3-game>
- [8] <https://gamedevacademy.org/html5-phaser-tutorial-top-down-games-with-tiled/>
- [9] <https://www.toptal.com/webrtc/taming-webrtc-with-peerjs>
- [10] <https://medium.com/@yoobi55/setting-up-heroku-server-with-node-js-express-eng-b0c8cc61a2a9>
- [11] <https://expressjs.com/it/>
- [12] <https://devcenter.heroku.com/articles/getting-started-with-nodejs>
- [13] <https://help.heroku.com/PIAVPANS/why-is-my-node-js-app-crashing-with-an-r10-error>
- [14] <https://www.quora.com/What-ports-does-WebRTC-use>

Appendice

Per effettuare il deploy del server node è stato usato `Heroku: Cloud Application Platform`.

Partendo dal file e dai passi effettuati per la creazione del Server (si veda la nota 27), si modifichi il file `package.json`:

```
...  
...  
"main": "P2P/PeerServerExpress.js",  
"scripts": {  
  "start": "node P2P/PeerServerExpress.js",  
  ...  
},  
...
```

Successivamente:

```
// installare Heroku  
sudo snap install heroku --classic  
  
// login  
heroku login  
  
// verificare le versioni di Node.js, npm, git:  
node --version  
npm --version  
git --version  
  
// tornare sul progetto e dare il comando:  
heroku create  
  
// fare push del contenuto  
git push heroku master  
  
// Assicurati che almeno un'istanza dell'app sia in esecuzione:  
heroku ps:scale web=1  
  
// visita l'app all'URL generato dal suo nome app  
heroku open  
  
// per vedere il Log del server  
heroku logs --tail
```