Your Own Distributed System using Consensus Protocols

Bellagio Casinó roulette multiplayer

Stefano Agostini
agostinistefano1991@gmail.com

Salomé Paolo
paolosalome@gmail.com

Valenti Alessandro
alessandro.valenti1991@gmail.com

ABSTRACT

Lo scopo del progetto é la realizzazione di un'architettura distribuita per lo sviluppo di una piattaforma di gioco multiplayer per mezzo di tecnologie Cloud. Il metodo adottato per tale realizzazione é un processo di sviluppo che aderisce al manifesto Agile, sfruttando una metodologia di tipo Scrum, con sprint ripetuti e brevi. Per aderire al meglio a questo processo di sviluppo sono state utilizzate tecnologie che permettessero di focalizzarsi principalmente sul problema anziché sulle difficoltá di realizzazione. Al termine di questo lavoro sono stati prodotti risultati soddisfacenti e che rispecchiano totalmente i requisiti progettuali.

1. INTRODUZIONE

1.1 Premessa

Le tecnologie Cloud, esplose in questi ultimi anni, hanno permesso di trasferire il peso della computazione e dell'approvvigionamento delle risorse informatiche verso internet. Perció la rete come strumento di sviluppo oggi offre scenari ampi nella direzione della distribuzione delle risorse e della computazione: la distribuzione, come nel caso che poniamo in esame, deve rappresentare un'avanguardia per lo sviluppo di applicazioni con caratteristiche di scalabilitá e di alta disponibilitá per l'utente finale. Gli sviluppatori utilizzando i servizi Cloud possono gestire in maniera semplificata l'immagazzinamento di informazioni e l'elaborazione di molti dati, concentrandosi principalmente sullo sviluppo della loro applicazione piuttosto che nell'utilizzo di un sistema ortodosso e dispendioso come avveniva nel passato. Aziende come *Spotify*, *Netflix* ed altre sfruttano il servizio Cloud per rendere fruibile le loro applicazioni agli utenti di tutto il mondo, ottenendo molteplici consensi. Molti non sanno che queste applicazioni sono cosí efficienti proprio perché sfruttano una tecnologia Cloud, che per quanto possa agevolare gli utenti e gli sviluppatori, presenta sempre qualche difficoltá da gestire:

1. garantire l'accesso concorrente ad una vasta molteplicitá di utenti.
2. la possibilitá di connettersi da ogni angolo del mondo, senza ledere la qualitá del servizio.
3. la sincronizzazione degli eventi associati ad ogni applicativo.
4. la reperibilitá dei dati.

1.2 Obiettivi

L'obiettivo del progetto consiste nell'elaborare un gioco multiplayer che supporti molteplici entitá autonome che si contendono risorse condivise, l'aggiornamento in tempo reale di una qualche forma di stato condiviso, la scalabilitá e sia distribuito su molteplici nodi (eventualmente distribuiti geograficamente). E' proprio con l'obiettivo di rispettare tali punti cardine che abbiamo proceduto nello sviluppo della nostra applicazione distribuita sulla piattaforma Cloud: abbiamo operato nell'intenzione di venire incontro alle esigenze degli utenti, sempre piú abituati ad avere interazioni costanti e ripetute con le applicazioni di uso comune. Il *Bellagio Casinó* é l'applicazione (accessibile mediante interfaccia Web) da noi proposta che consente agli utenti di giocare ad una versione semplificata di una roulette francese condividendo un unico tavolo di gioco. Tale applicazione permette di effettuare particolari puntate in base alla propria disponibilitá di credito.

2. ARCHITETTURA

In questa sezione verrá illustrata a livello logico la nostra architettura descrivendo i servizi *Amazon* utilizzati e i moduli sviluppati (come illustrato in figure 1). I componenti architetturali sono:

- un Cluster composto da istanze *EC2 t2-micro*, con sistema operativo *Ubuntu 16.01* situato nella regione Francoforte.
- due Cluster composti da istanze *EC2 t2-micro*, con sistema operativo *Ubuntu 16.01* situati nella regione Eire.
- Client.

i servizi *AWS* sfruttati dai componenti sono:

- *SNS* per la comunicazione inter-cluster.
- *DynamoDB* per la persistenza dei dati, situato nella regione Eire.
- *Redis* per supportare la logica del gioco, situato sia nella regione Eire che Francoforte.
- *LoadBalancer* per distribuire il carico dei Client.

La motivazione che ci ha spinto nell'adottare piú cluster su piú aree geografiche europee é stata la possibilitá di realizzare una scalabilitá di tipo geografico. Ció non toglie che tale realizzazione si possa estendere anche all'interno di altre

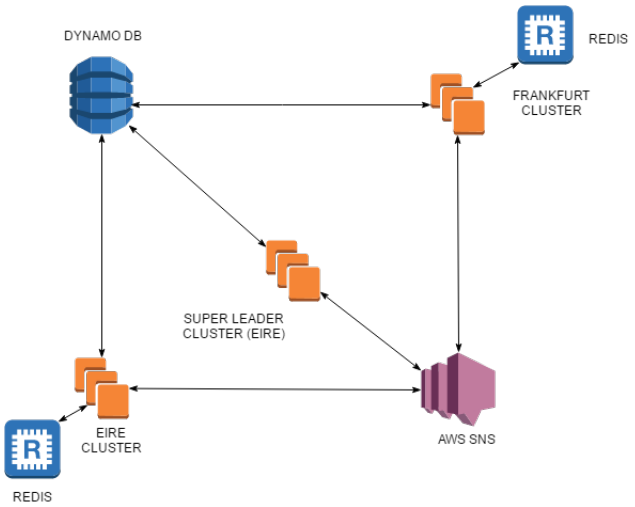

Figure 1: Architettura

regioni geografiche messe a disposizione da *AWS*. La causa che ci ha spinto a questa scelta é legata a fattori di latenza ma anche a fattori economici, oltre alla semplificazione dello sviluppo dell'applicazione e della fase di testing.

2.1 Cluster EC2

In questa sezione andiamo a trattare nel dettaglio quali sono le funzionalitá e le componenti dei Cluster *EC2*.

I Cluster istanziati sono soggetti ad una gerarchia dettata dalla realizzazione di uno stato condivisibile, che corrisponde alle fasi di gioco dell'applicazione. Pertanto possiamo suddividere tali cluster in base alla loro mansione :

- *SuperLeader* Cluster: identifica il Cluster per la gestione dello stato di gioco, e per le azioni di registrazione e di accesso sito in Irlanda.
- *Frankfurt/Eire* Cluster: identifica uno dei Cluster di gioco sito sia in Irlanda che a Francoforte.

All'interno di ogni Cluster per mantenere lo stato corrente del gioco, abbiamo adottato *Raft* come protocollo del consenso. Abbiamo utilizzato 3 istanze *EC2* per Cluster, in quanto é il numero minimo di nodi necessario per raggiungere il consenso mediante tale protocollo.

3. IMPLEMENTAZIONE

3.1 Metodologie di sviluppo

L'intero progetto é stato realizzato seguendo la metodologia di sviluppo *Scrum*, la quale prevede di dividere il progetto in blocchi rapidi di lavoro (Sprint), alla fine di ciascuno dei quali si deve creare un incremento del software. L'intera stesura del progetto, dalla progettazione all'implementazione, é stata quindi suddivisa in sprint con scadenza molto stretta assegnati ai singoli componenti del team. Per la suddivisione dei task sono stati effettuati incontri giornalieri, mentre per la sincronizzazione e per il controllo di versione é stato utilizzato il servizio di *Google Drive*. Il periodo di sviluppo, é stato caratterizzato da incontri giornalieri (Daily Scrum) incentrati sulla tematica del giorno, con la presenza del team al completo. A supporto dello sviluppo é stato previsto, fin dal primo istante, un ulteriore incontro prima della partenza di ogni nuovo sprint. La metodologia utilizzata si é dimostrata efficiente poiché ha permesso di progettare e realizzare un architettura ben congegnata in un arco temporale ridotto.

3.2 Tecnologie utilizzate

Il lato server del progetto é stato interamente realizzato utilizzando il framework *Node.js*, che consente di sfruttare il linguaggio *JavaScript*, tipicamente utilizzato nel client-side, anche per la scrittura di applicazioni server-side. La caratteristica principale di *Node.js* risiede nella possibilitá di accedere alle risorse del sistema operativo in modalitá event-driven evitando il modello basato su processi o thread concorrenti utilizzato dai classici web server. Per facilitare l'uso di *Node.js* é stato utilizzato il framework *Express.js*, per la realizzazione semplificata del middleware *HTTP*. L'applicazione client é costituita da una singola pagina *HTML* con contenuti *Javascript* ed elaborata attraverso l'uso della libreria *JQuery* rendendo l'interfaccia dinamica, consistente e di facile utilizzo.

3.3 Raft

Raft é un protocollo del consenso in alternativa a *Paxos* che offre la possibilitá di condividere uno stato comune tra i vari nodi del Cluster. Questi si suddividono in leader, candidate e follower. Viene raggiunto il consenso attraverso l'elezione di un leader, la cui mansione principale é quella di propagare il log dei cambiamenti di stato ai propri follower. Inoltre per ribadire la propria leadership deve informare periodicamente i propri follower attraverso dei brevi messaggi. Ogni follower deve ricevere questi messaggi all'interno di un arco temporale fissato. Se ció non dovesse avvenire verrá richiesta una nuova elezione di leader; colui che indice la nuova elezione cambia il proprio ruolo in candidate. Per far sí che *Raft* tolleri K guasti occorrono almeno $2K + 1$ nodi all'interno del Cluster.

Per l'implementazione di tale protocollo é stato adottata la libreria *Skiff*. Implementa la persistenza dello stato attraverso un database *in-memory* di tipo *MemDown* il quale utilizza un'interfaccia *LevelUp*. Per quanto concerne la comunicazione del log e dell'heartbeat utilizza server *TCP*, in ascolto su ogni nodo, attraverso chiamate di tipo *RPC*.

Le motivazioni che ci hanno condotto a questa scelta sono da addurre alla semplicitá di esecuzione e di condivisione del log tra i vari nodi del Cluster, la cui propagazione avviene in tempi irrisori. L'utilizzo di *Skiff* con l'architettura dei Cluster da noi ideata consente di raggiungere una certa tolleranza a guasti di vario genere sulle varie istanze *EC2*. In caso di un guasto ad una macchina, se questa risulta essere un leader quello che avviene é una nuova elezione, con conseguente riallineamento dei nodi del Cluster allo stato corrente del gioco. Nel caso di un guasto nei confronti di un follower bisogna monitorare il numero di nodi presenti nel Cluster, come precedentemente illustrato.

3.4 Comunicazione

Per quanto riguarda la comunicazione, vogliamo porre l'attenzione verso le due tipologie presenti all'interno del nostro sistema:

- Comunicazione *intra-Cluster*: sfruttiamo le chiamate *RPC*, come descritto nel paragrafo precedente.
- Comunicazione *inter-Cluster*: sfruttiamo il servizio *AWS*

di *SNS*.

Amazon SNS é un servizio di notifiche push rapido, flessibile e completamente gestito che consente di inviare messaggi individuali o collettivi a un numero elevato di destinatari. Grazie a questa soluzione, inviare notifiche push a utenti di dispositivi mobili, destinatari di posta o addirittura altri servizi distribuiti é semplice e conveniente. Per comunicare lo stato del gioco a tutti nodi dei vari Cluster sono stati creati 5 *Topic* per permettere lo scambio di informazioni e di cambiamento di stato; ció verrá chiarificato successivamente.

La motivazione per cui é stato scelto questo servizio sono le seguenti:

- *SNS* permette la realizzazione di un servizio di tipo *Publish/Subscribe*.
- Sfrutta endpoint di tipo *http* per la ricezione di notifiche.
- Permette la comunicazione tra piú aree geografiche.
- Garantisce un certo grado di riservatezza.

Generalmente *SNS* viene utilizzato in abbinamento con *SQS*, peró nel nostro caso non avevamo una necessitá di permanenza e di ordinamento dei messaggi.

3.5 Persistenza

Poiché la nostra applicazione é un gioco multiplayer e avendo necessitá di gestire la registrazione e l'accesso di piú giocatori, ci occorre un servizio di persistenza dati. A tal proposito la persistenza é stata realizzata sfruttando due tipi di servizio:

- *Redis*: é un servizio di cache distribuito di *Amazon AWS*.
- *DynamoDB*: é un database *NoSQL* distribuito di *Amazon AWS*.

Nel dettaglio *Redis* viene sfruttato dai Cluster di gioco per poter immagazzinare le ultime informazioni relative alle puntate degli utenti, rendendo tali dati accessibili per l'elaborazione delle vincite da parte del sistema. Alla fine di questa operazione il sistema provvederá all'eliminazione di tali dati per permettere l'aggiornamento degli stessi coerentemente con la mano del gioco.

Per quanto concerne *DynamoDB* sono state create due tabelle

- una per conservare i dati relativi agli utenti (credenziali di accesso e credito).
- una per conservare le puntate elaborate dal sistema, precedentemente contrassegnate come vincenti o perdenti.

Quando un'applicazione effettua delle scritture su una tabella in *DynamoDB* i dati impiegano del tempo per propagarsi in ogni locazione di memoria della regione *AWS* corrente. I dati saranno consistenti in tutte le locazioni di memoria in un periodo lungo $\sim$ 1 sec o anche meno. Dato che la nostra applicazione non effettua chiamate al Database frequentemente, la scelta dell'*eventual consistency* ci é sembrata la piú opportuna anche in termini di costi.

Abbiamo sfruttato la *chiave di partizionamento* che offre *DynamoDB* per collocare tramite funzione hash le puntate

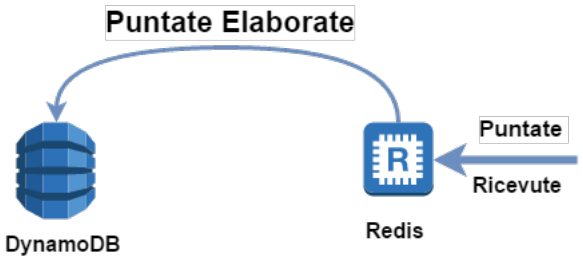

Figure 2: Processo scrittura delle puntate con interazione DynamoDB e Redis

appartenenti alla stessa mano nella stessa partizione; cosí come la *chiave di ordinamento* per catalogare nella stessa partizione le puntate appartenenti ad utenti differenti. Su consiglio delle linee guida fornite da *DynamoDB* abbiamo scelto di utilizzare come chiave di partizionamento la *mano* del gioco poiché offre un range di valori piú ampio rispetto a quello offerto dalla chiave *username*.

3.6 Implementazione Clusters

Come illustrato precedentemente i Cluster da noi sviluppati possono essere suddivisi funzionalmente in due gruppi distinti: *SuperLeader* e *Frankfurt/Eire* Cluster. Tutti i Cluster definiti sono configurati, come precedentemente indicato nella sezione Architettura, con un numero fissato pari a 3 di istanze *EC2*. La scelta di tale numero permette la gestione di un solo guasto (come descritto in *Raft*) e aumentando il numero di istanze il risultato sarebbe invariato con un grado di tolleranza ai guasti maggiore. Un'altra motivazione che ci ha spinto a tale scelta é di natura economica. In ogni caso l'aggiunta o meno di altre istanze all'interno dei nostri Cluster puó essere effettuata senza alcun problema, a patto che vengano identificate all'interno di una *VPC* attraverso un tag univoco. Il sistema é configurato in modo che ogni istanza all'avvio sappia identificare esattamente quali sono le macchine appartenenti al proprio gruppo, permettendole di sfruttare il protocollo *Raft* per comunicare con tutte le entitá dello stesso. Tutto ció sinteticamente rappresenta un servizio di *Discovery Service* atto a configurare dinamicamente i Cluster *Raft*.

3.6.1 Stati del Sistema

Poniamo ora l'attenzione verso gli Stati del Sistema. I due tipi di stato che si alternano sono:

- *Play*: identifica le fasi di gioco dell'applicazione in cui l'utente puó effettuare la puntata desiderata.
- *Compute*: identifica la fase di estrazione del numero e il calcolo delle puntate vincenti.

Nella definizione di questi stati viene allegato l'istante di tempo in cui vengono generati. Si ha un cambiamento di stato quando entrambi i Cluster di gioco comunicano la terminazione dello stato corrente, come illustrato nella figura 3.

3.6.2 SuperLeader

Il *SuperLeader* é il Cluster adibito alla definizione dello stato di gioco e all'accesso e alla registrazione degli utenti.

Come primo passo definiamo la gerarchia del Cluster che si compone di un leader e di due peer (follower). Tale

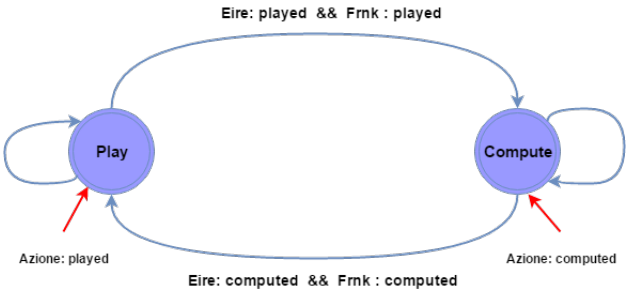

Figure 3: Rappresentazione degli Stati del sistema

definizione é derivata dai ruoli che impone *Raft*. Il leader, cosí come i due peer, si compone di due processi :

- processo Raft (padre) che si occupa del consenso per mezzo di server *TCP* e sfrutta *SNS* per pubblicare i cambi di stato.
- processo *HTTP* (figlio) che si occupa di ricevere le notifiche *SNS* e le richieste da parte degli utenti.

Il processo *Raft* sfrutta la libreria *Skiff* per mantenere il consenso su un determinato stato di gioco attraverso la propagazione del log e mediante chiamate *RPC*. Nella fase di inizializzazione il leader si sottoscrive ai topic di interesse:

- *Computed*: su questo topic attende i messaggi che indicano la conclusione della fase di gioco *Compute* da parte dei Cluster di gioco. Una volta ricevuti i messaggi da entrambi effettua la transizione di stato *Compute* $\rightarrow$ *Play* e lo invia ai Cluster di gioco.
- *Played*: su questo topic attende i messaggi che indicano la conclusione della fase di gioco *Play* da parte dei Cluster di gioco. Una volta ricevuti i messaggi da entrambi effettua la transizione di stato *Play* $\rightarrow$ *Compute*, genera il numero della Roulette inserendolo nello stato per poi inviare quest'ultimo ai Cluster di gioco.

Lo schema soprastante rappresenta il ciclo naturale di esecuzione del leader. In caso di guasto di quest'ultimo viene innescata una procedura di rientro che coinvolge il nuovo leader. Esso:

1. si sottoscrive al topic *LeaderState*.
2. richiede ad entrambi i Cluster di gioco il loro stato corrente, i quali a loro volta lo pubblicheranno sul topic *LeaderState*.
3. confronta lo stato ricevuto da ognuno dei Custer con il suo stato corrente e se il primo é piú recente rispetto al secondo lo sostituisce.
4. si sottoscrive ai topic *Computed* e *Played* e successivamente chiede ai Cluster di gioco di ritrasmettere l'ultima azione (*Computed* o *Played*) da loro completata.
5. riprende la normale esecuzione descritta sopra.

A differenza di questa funzione legata alla generazione degli stati del sistema, associata unicamente al leader, qualsiasi nodo del Cluster *SuperLeader* si occupa della registrazione e dell'accesso di nuovi utenti. Attraverso la pagina web messa a disposizione agli utenti, questi possono effettuare la propria sottoscrizione al gioco:

1. la richiesta *HTTP* viene catturata dal *middleware Express* del nodo server contattato.
2. vengono estratte le credenziali dell'utente, e rese persistenti sulla tabella *Accounts* per mezzo delle *DynamoApi*.
3. viene inviato al *client HTTP* la conferma di avvenuta registrazione.

Per quanto riguarda il login degli utenti, il nodo del Cluster contattato alla ricezione di una tale richiesta:

- cattura la richiesta *HTTP* e ne estrae le credenziali tramite *middleware Express*.
- controlla che le credenziali siano presenti nella tabella *Accounts* di *DynamoDB*.

3.6.3 Frankfurt/Eire Cluster

I Cluster di gioco si occupano dell'esecuzione delle operazioni relative allo stato corrente del sistema. Essi operano in due regioni differenti ma sono esattamente equivalenti, pertanto ne illustreremo soltanto uno.

Il Cluster di gioco sfrutta le funzionalitá offerte da *Skiff* per mantenere il consenso sullo stato corrente tra i suoi nodi mediante chiamate *RPC* e per crearne una gerarchia (*follower/leader*). Il leader, cosí come i due peer, si compone di due processi :

- processo Raft (padre) che si occupa del consenso per mezzo di server *TCP* e sfrutta *SNS* per pubblicare la terminazione della fase corrente di gioco (*Computed / Played*).
- processo *HTTP* (figlio) che si occupa di ricevere le notifiche *SNS* e le richieste da parte degli utenti.

Nella fase di inizializzazione il leader si sottoscrive al topic di interesse *RegionLeaderTopic*. Su questo topic attende notifiche da parte del Cluster *SuperLeader* che si differenziano in funzione del contenuto:

- se il contenuto é un *JSON* allora identifica un nuovo stato del sistema, al quale il Cluster di gioco dovrá allinearsi ed eseguire le relative operazioni.
- se il contenuto é la stringa *currentState* allora identifica la richiesta dello stato corrente del Cluster di gioco. Questa notifica viene inviata dal leader del Cluster *SuperLeader* quando é stata innescata la procedura di rientro di quest'ultimo.
- se il contenuto é la stringa *lastMessage* allora identifica la richiesta di ritrasmissione dell'ultimo messaggio di terminazione fase inviato (*Computed / Played*). Questa notifica viene inviata dal leader del Cluster *SuperLeader* quando viene innescata la procedura di rientro di quest'ultimo.

Di conseguenza alla ricezione della notifica di cambio di stato del sistema nel nodo leader possono verificarsi due scenari differenti:

- il valore del campo *phase* dello stato ricevuto corrisponde a *Play*. Il leader del Cluster di gioco crea un processo che gestisce la nuova fase. Per un tempo pari a *20 secondi* quest'ultimo permane nello stato *Play*, terminato il quale pubblica sul topic *Played* il suo completamento specificando la sua zona di appartenenza (*Frankfurt o Eire*); infine aggiorna nel db *in-memory di Skiff* il valore identificato dalla chiave *lastMessage* al valore *Played*. In questa fase, inoltre, ogni nodo del Cluster di gioco (sincronizzato mediante *Skiff* allo stato *Play* corrente) riceve dai client connessi le puntate effettuate dagli utenti e le salva di conseguenza all'interno della cache *Redis* della regione in questione.

- il valore del campo *phase* dello stato ricevuto corrisponde a *Compute*. Il leader del Cluster di gioco crea un processo che gestisce la nuova fase. Per un tempo pari a *8 secondi* quest'ultimo permane nello stato *Compute*, terminato il quale pubblica sul topic *Computed* il suo completamento specificando la sua zona di appartenenza (*Frankfurt o Eire*); infine aggiorna nel db *in-memory di Skiff* il valore identificato dalla chiave *lastMessage* al valore *Computed*. Durante quest'arco temporale il leader del Cluster di gioco preleva dalla cache *Redis* le puntate raccolte durante la fase precedente, ne calcola l'esito e le salva sulla tabella *Bets* di *DynamoDB*. Dopodiché aggiorna il credito di ogni utente che ha effettuato la puntata all'interno della tabella *Accounts* ed infine libera la cache per la nuova fase.

Lo schema soprastante rappresenta il ciclo naturale di esecuzione del leader. In caso di guasto di quest'ultimo viene innescata una procedura di rientro che coinvolge il nuovo leader. Prima di effettuarla il nuovo leader provvederá ad effettuare le sottoscrizioni descritte prima. In base allo stato di *Skiff* che ha ereditato adotta due comportamenti differenti:

- se la sua fase corrisponde a *Play* allora il leader calcola il tempo di gioco restante rispetto all'istante di inizio della fase (lo stato ha come attributo un *timestamp* di creazione stato) al termine del quale comunica al *SuperLeader* la terminazione della fase; infine riprende la normale procedura di esecuzione precedentemente descritta.

- se la sua fase corrisponde a *Compute* allora il leader cerca eventuali puntate nella cache e, se presenti, le elabora calcolandone l'esito e le salva nella tabella *Bets*; di conseguenza comunica al *SuperLeader* la terminazione della fase e riprende la normale esecuzione.

3.7 Bilanciamento del carico

Per bilanciare il carico dei client su tutti i nodi presenti nei Cluster abbiamo deciso di utilizzare il servizio di Amazon *Application Load Balancer*. Il servizio garantisce anche un'analisi sempre aggiornata dello stato di salute dei nodi del Cluster iscritti, inoltrando le richieste Client solo alle istanze *healty*. Ci occorreva un servizio semplice e configurabile dinamicamente che permettesse di rendere il sistema bilanciato rispetto al numero di Client indipendentemente dal numero di istanze *EC2* del Cluster. L'utilizzo di tale servizio rallenta il processo di saturazione delle risorse poiché impedisce una concentrazione eccessiva di connessioni su uno stesso nodo. Tutto ció contribuisce a rendere il sistema piú robusto e meno esposto a guasti.

Abbiamo configurato tre *Load Balancer* di cui due per i Cluster della regione *Irlanda* e uno per il Cluster della regione *Francoforte*. Inoltre abbiamo sfruttato il *Cookie Stickiness* il quale permette di mantenere la connessione *Client Server* stabile su una sola macchina per un intervallo di tempo fissato (nel nostro caso *3 secondi*); questo risulta utile nei casi in cui un server target sta diventanto *unhealty* ma non risulta ancora tale per il *Load Balancer*.

3.8 Scalabilitá

Il termine scalabilitá, nelle telecomunicazioni, nell'ingegneria del software, in informatica e in altre discipline, si riferisce, in termini generali, alla capacitá di un sistema di *crescere* o diminuire di scala in funzione delle necessitá e delle disponibilitá. Un sistema che gode di questa proprietá viene detto scalabile. La tipologia di scalabilitá implementata nel nostro sistema é quella *geografica*. Un sistema geograficamente scalabile é quello che mantiene inalterata la sua usabilitá e utilitá indipendentemente dalla distanza fisica dei suoi utenti o delle sue risorse.

Poiché il nostro sistema viene installato all'interno di piú nodi situati in diverse aree geografiche, abbiamo analizzato varie *policy* che permettessero una gestione dinamica del carico degli utenti sui nodi della rete, mantenendo un servizio efficiente indipendentemente dalla propria distanza dal Cluster. A tal proposito abbiamo sfruttato le funzionalitá offerte dal servizio *AWS Load Balancer* per distribuire il carico sui vari nodi, come illustrato in precedenza, ed é stata adottata una politica di *Ping* per il reindirizzamento dell'utente al Cluster, permettendogli di accedere alle risorse senza avere una grossa latenza. Sfruttando tale *policy* implementiamo la scalabilitá geografica all'interno del protocollo di rete:

1. Si effettua il ping verso gli indirizzi dei due *Load Balancer* siti in due aree geografiche distinte.

2. Si analizza il *RTT* prodotto ricercando il minore tra i due.

3. Si reindirizza l'utente verso il *Load Balancer* che ha generato il *RTT* minore.

Non sfruttiamo *policy* come la localizzazione dell'indirizzo *IP* in quanto risulta essere sconveniente per effettuare una connessione ad una regione piuttosto che ad un'altra, poiché non sempre la ridotta distanza fisica (distanza tra la macchina usata dall'utente ed il server ospitante il *Load Balancer*) rispecchia una vicinanza in termini di rete.

Tale procedura é implementata nel Client del Cluster *SuperLeader* e viene innescata in corrispondenza del login di un utente alla piattaforma di gioco. In questo modo il Cluster distribuisce gli utenti sui diversi Cluster di gioco.

3.9 Client

Per l'elaborazione della piattaforma di gioco é stata implementata una singola pagina *HTML* che sfrutta i servizi offerti da *Bootstrap*, *CSS*, ed infine *JQuery* per la logica dietro la pagina. Prima di trattare la logica implementativa, poniamo l'attenzione sulle fasi del gioco presenti all'interno dell'applicazione. Le fasi di gioco sono 4 e servono per identificare le varie fasi dettate dal gioco della roulette.

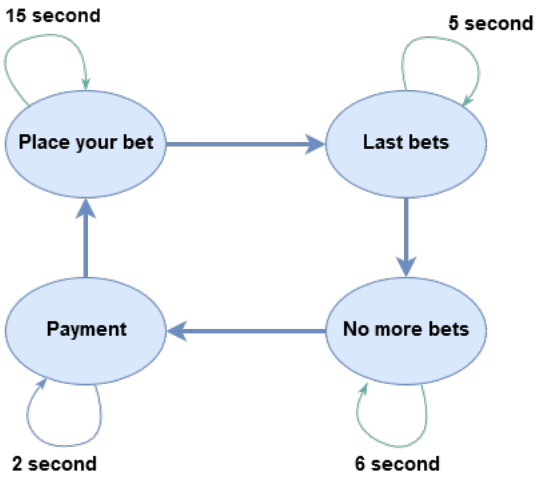

Figure 4: Rappresentazione degli Stati del Client

- *Place your bet*: in questa fase ogni utente che abbia effettuato l'accesso alla piattaforma ha a disposizione $\sim$ 15 secondi per puntare sulla roulette.
- *Last bets*: in questa fase ogni utente che abbia effettuato l'accesso alla piattaforma ha a disposizione gli ultimi $\sim$ 5 secondi per puntare sulla roulette.
- *No more bets*: in questa fase l'utente non puó piú effettuare puntate mentre attende l'uscita del numero del turno.
- *Payment*: in caso di vincita si effettua l'aggiornamento del credito a disposizione dell'utente.

Queste fasi di gioco del Client corrispondono alle fasi dettate dalle regole di gioco della roulette, ed inoltre vengono utilizzate per effettuare la sincronizzazione con le fasi del Server. A tal proposito le prime due fasi, *Place your bet* e *Last Bets* corrispondono alla fase di *Play* definita nel server, descritta nei paragrafi precedenti. La durata di queste due fasi é pensata in modo che ci sia una sincronia con le fasi del server, evitando cosí che le puntate degli utenti vengano effettuate durante la fase errata. Le restanti fasi, *No more bets* e *Payment*, corrispondono alla fase di *Compute* del server. Durante queste ultime fasi vengono effettuate diverse chiamate rest verso il *Load Balancer*, il quale contatterá a sua volta i nodi ad esso registrati, ricevendo le informazioni richieste, tra cui il credito dell'utente e lo stato del gioco. La fase di *Payment*, oltre alla funzione descritta in precedenza, svolge una funzione aggiuntiva di riallineamento con lo stato del server, quando necessaria, in caso di guasto di quest'ultimo.(vedi figure 4).

3.10 Esempio Flusso di Esecuzione

Qui di seguito illustriamo la tipica esecuzione di un utente che vuole entrare nella piattaforma per iniziare a giocare:

- Contatta l'*indirizzo DNS* del *Load Balancer* relativo al Cluster *SuperLeader* e questo risponde fornendo la pagina di login.
- Una volta richiesto l'accesso attraverso il form dedicato viene mandata un ulteriore richiesta al *Load Balancer* per confermare le credenziali.
- Il *Load Balancer* rimanda la richiesta ad uno dei server del Cluster *SuperLeader* il quale controlla le credenziali; una volta confermate si *pingano* gli altri *Load Balancer* e si seleziona quello con il *RTT* minore.
- Viene instradata la richiesta verso il *Load Balancer* della regione cosí selezionata il quale fornisce la pagina di gioco.
- Al primo accesso dell'utente alla piattaforma di gioco assegnata gli verrá richiesta la conferma di un codice di accesso (*daily code*). Se il codice é corretto si accede alla piattaforma altrimenti si ritorna alla pagina di login.

4. TEST

In data 25/11/2016 dalle ore 14:30 alle ore 17:30 abbiamo testato il nostro sistema coinvolgendo colleghi dell'ambiente universitario. Lo scopo principale del test é stato monitorare la capacitá di adattamento del sistema alle variazioni del carico e l'efficacia della *policy del ping* per il processo di *redirect*.

Il campione di utenti era costituito prevalentemente da utenti di nazionalitá italiana con l'eccezione di un utente residente a Londra ed un secondo residente a Siviglia. Entrambi hanno osservato che il server di gioco al quale si sono collegati era appartenente al Cluster irlandese; ció indica che il *RTT* del *ping* é anche influenzato dalla distanza geografica delle entitá comunicanti oltre che dalla distanza in termini di rete, non tradendo le nostre aspettative. Tutti gli altri utenti, situati nel territorio nazionale, hanno sperimentato un reindirizzameto al Cluster della regione geografica di Francoforte.

Durante il test si sono registrati e connessi ai server di gioco 30 utenti distribuiti durante l'arco temporale sopra indicato. Per mezzo del servizio *Cloud Watch* abbiamo prelevato dei dati riguardanti l'andamento del carico dei tre *Load Balancer*. Le metriche riportate sui grafici riguardano i valori medi in un intervallo di *5 minuti* delle connessioni attive e di quelle in ingresso. (vedi figure 5, 6, 7)

I risultati ottenuti attraverso questo test sono da ritenersi un successo in quanto i Cluster di gioco e quello di accesso non hanno subito alcun rallentamento. Questo test funzionale ci ha permesso di verificare che la sincronizzazione tra *client* e *server* non subisse alcuna alterazione rilevante dovuta al numero di utenti connessi al sistema.

Inoltre osservando le risorse delle singole istanze *EC2*, é stato riscontrato che la soglia di occupazione della memoria volatile fosse inferiore a $\sim$ 300Mb a fronte del 1Gb messo a disposizione dalle nostre istanze *EC2 t2-micro*.

É stata nostra premura gestire al meglio la memoria, poiché avevamo a disposizione istanze che non forniscono una grossa quantitá di RAM. Infatti come primo passo abbiamo cercato di ottimizzare il codice dei server cercando di liberare piú risorse possibili permettendo all'istanza di sfruttare a pieno le proprie limitate possibilitá. Ovviamente sfruttando istanze *EC2* con una memoria volatile piú elevate tale problema si sarebbe evidenziato di meno, ma a cause di fattori economici abbiamo cercato di ottimizzare quello che avevamo a nostra disposizione.

5. CONCLUSIONI

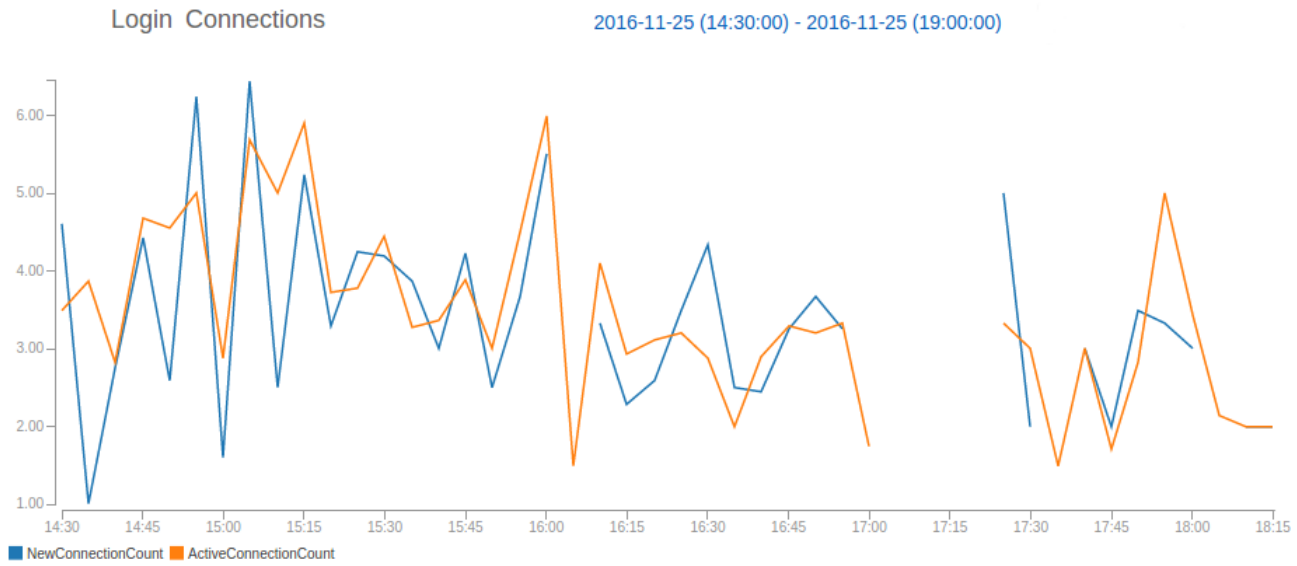

Figure 5: Andamento delle metriche del *Load Balancer del SuperLeader*

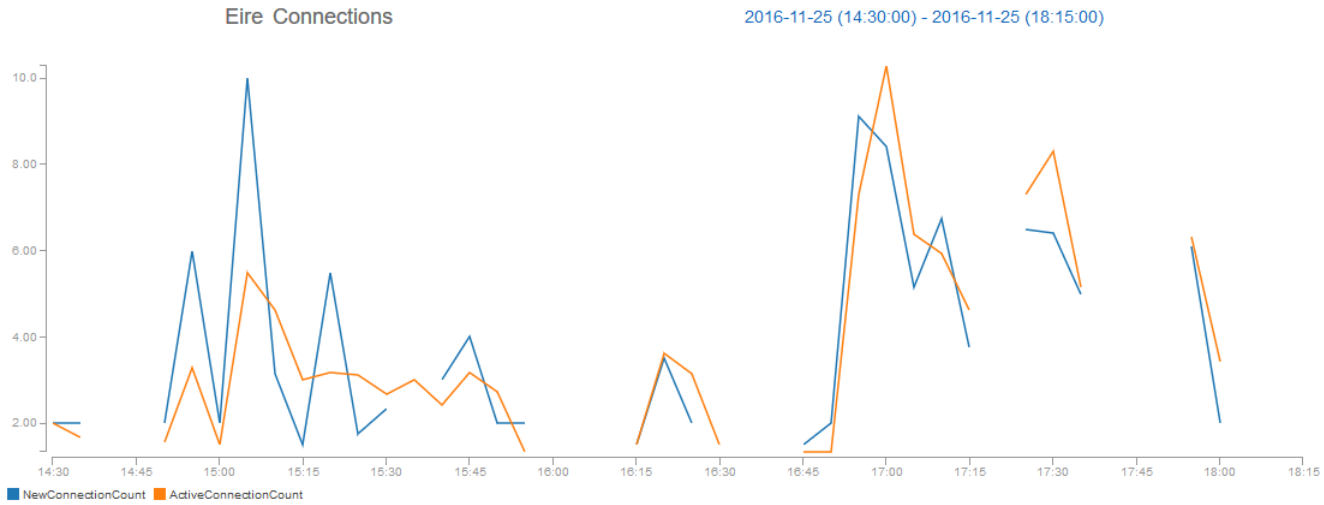

Figure 6: Andamento delle metriche del *Load Balancer del Cluster Eire*

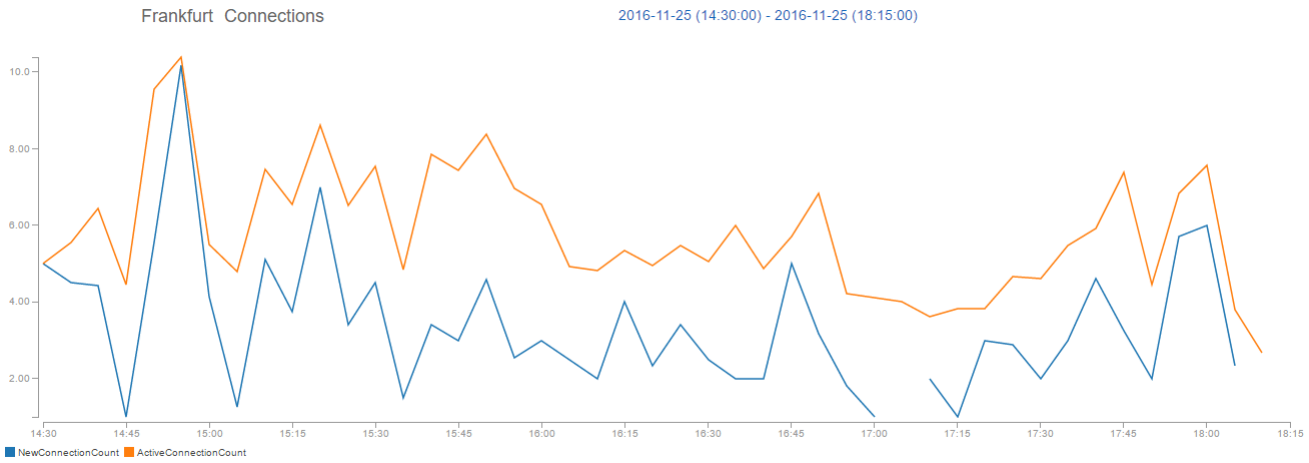

Figure 7: Andamento delle metriche del *Load Balancer del Cluster FrankFurt*

L'obiettivo del progetto era quello di realizzare una architettura distribuita applicata al caso di un gioco multiplayer. Nonostante i problemi riscontrati in fase iniziale di sviluppo, dovuti all'inesperienza nel campo, siamo riusciti a sviluppare un sistema che rispondesse ai requisiti preposti: distribuzione del carico, robustezza del sistema, distribuzione del sistema su scala geografica, tolleranza ai guasti, sincronizzazione degli eventi associati allo stato dell'applicazione ed infine la consistenza dei dati. Il soddisfacimento di tutti questi obiettivi é stato avvalorato dai test funzionali effettuati. Seppur soddisfatti gli obbietti essenziali per la realizzazione del sistema, sono stati identificati eventuali miglioramenti, alcuni dei quali sono stati ritenuti secondari per l'attuale bacino d'utenza.

6. SVILUPPI FUTURI

Per migliorare le prestazioni del sistema abbiamo pensato di introdurre un livello di scalabilitá orizzontale. Per implementare tale funzionalitá bisognerá usufruire del servizio di *Autoscaling* offerto da *AWS*. Questo servizio istanzierá una nuova macchina *EC2* configurata come le altre del Cluster in base ad alcune *policy* come la saturazione della CPU oppure un eccessivo numero di connessioni in ingresso. Per integrare tale servizio all'interno della nostra architettura abbiamo bisogno di alcune accortezze:

1. Generare un *Topic* per ogni regione per comunicare al leader del Cluster la nascita di un nuovo peer.
2. Il nuovo peer pubblica sul *Topic* il proprio ingresso al Cluster.
3. A livello di codice il leader del Cluster deve effettuare una operazione di *join Skiff* all'interno della rete *RAFT*.
4. Il leader iscrive il nuovo peer al *Load Balancer* del Cluster.
5. Si effettua un riallineamento dello stato corrente.

Un altro punto da migliorare sarebbe relativo alla sicurezza informatica. A tal proposito bisognerebbe gestire la sicurezza delle credenziali degli utenti contro possibili attacchi di tipo *Sniffing* attraverso una procedura di crittografia. Questo tipo di problema viene riscontrato anche nella propagazione dei *cookie* atti alla gestione delle credenziali d'accesso. Con l'implementazione di un servizio di sicurezza efficiente é possibile gestire l'inserimento e la visualizzazione sul Web del credito degli utenti.

Per quanto riguarda la metrica di scalabilitá geografica implementata sarebbe piú opportuno adottare una policy differente e piú robusta rispetto al *ping*. Ad esempio é possibile sfruttare algoritmi basati sui grafi per il calcolo delle distanze geografiche tra i nodi della rete.

Infine avendo ricevuto i feedback da parte degli utenti che hanno aderito al test dell'applicazione sono emersi delle possibile migliorie a livello grafico e logico da apportare all'applicazione per renderla piú *user-friendly*.

7. REFERENCES

[1] https://en.wikipedia.org
[2] https://aws.amazon.com
[3] https://expressjs.com
[4] https://nodejs.org
[5] https://www.npmjs.com/package/skiff