FunGo Studios Srl
Via Sile, 41 - 31056 Roncade
Treviso - ITALY
P. IVA: 04463380263

mail: team@fungostudios.com
tweet: @fungostudios

Introduzione

Un tipico *social game* ha esigenze molto diverse dalla maggior parte delle applicazioni web. Queste ultime, infatti, sono costruite su architetture di tipo *stateless,* basate su un *load balancer che ripartisce le richieste in ingresso*, un *pool* di server di frontend che trattano le richieste, e un database SQL per la persistenza dei dati. In un'architettura di questo tipo, il contesto completo di una sessione utente è recuperabile unicamente interrogando il database alla base dello *stack* architetturale.

Una sessione di gioco di un singolo utente può generare centinaia di chiamate HTTP verso i server. In questo caso, un'architettura *stateless* risulta inadeguata. Ad ogni chiamata, infatti, il *load balancer* può selezionare un server diverso nel pool di server di frontend (si veda la Figura 1). Per questo motivo il database viene stressato con l'esecuzione di molte operazioni *statement*, sia di lettura (*query*), che di scrittura. Inoltre a ogni richiesta i server di frontend devono deserializzare e serializzare molti dati per recuperare le informazioni necessarie per aggiornare l'ambiente di gioco in base alle azioni dell'utente, incrementando in questo modo il carico di lavoro già cospicuo.

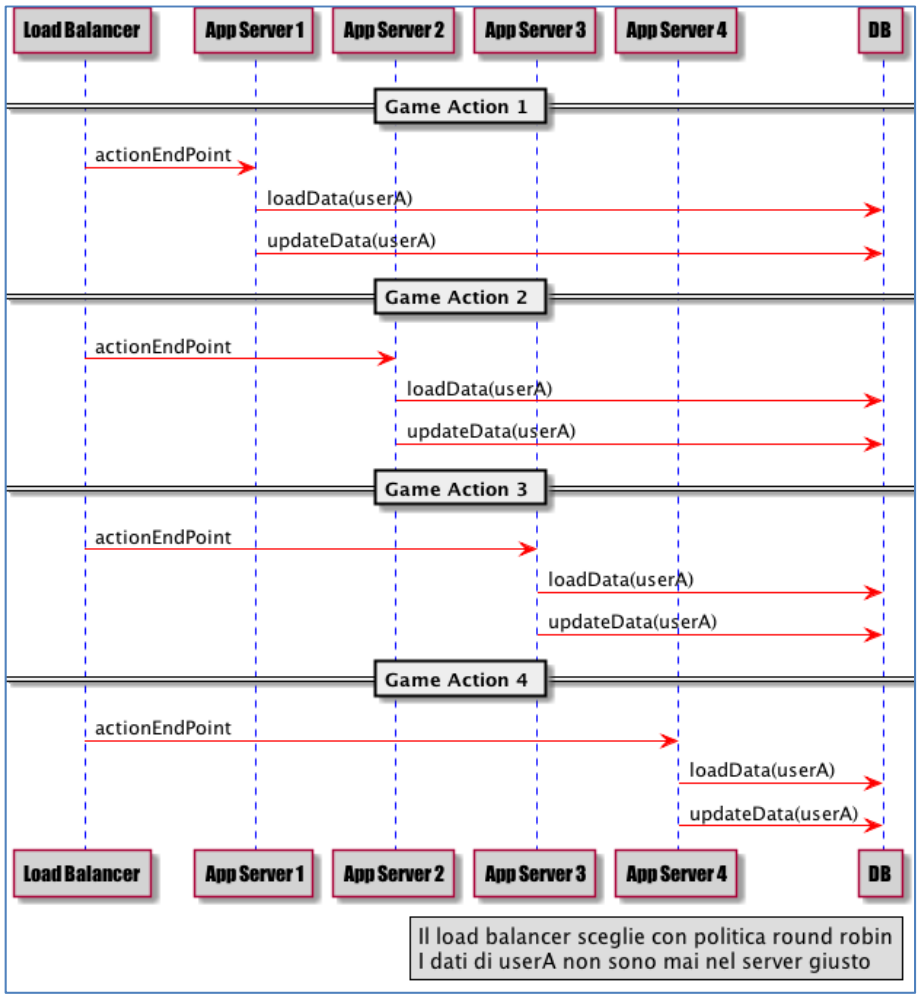

Figura 1: Sequenza di chiamate necessarie a soddisfare una richiesta in un'architettura *stateless*.

FunGo Studios Srl
Via Sile, 41 - 31056 Roncade
Treviso - ITALY
P. IVA: 04463380263

mail: team@fungostudios.com
tweet: @fungostudios

Dimensionamento del database

Considerato su base mensile, un tipico gioco "social" di successo può generare nell'ordine di 14 miliardi di *request*, da un milione di utenti attivi giornalmente , 5.000 HTTP *request* per secondo, di cui circa il 90% rappresentate da operazioni di scrittura o aggiornamento (POST e PUT).

Utilizzando un'architettura *stateless*, per soddisfare una richiesta HTTP possono essere necessarie oltre 10 query verso il database, per un totale di oltre 50.000 *query* al secondo.

La gestione di 50.000 *query* constringe a partizionare un database SQL su più macchine utilizzando tecniche di partizionamento orizzontale (*sharding,* http://en.wikipedia.org/wiki/Shard_(database_architecture)). Ne risulta un'architettura poco elastica e complessa con grosse problematiche di *scaling* sia nella fase di aumento del numero di server che in quella di riduzione. I tempi per riconfigurare l'intera struttura dopo l'aggiunta o la rimozione di un server possono essere molto lunghi, soprattutto all'aumentare della dimensione del database.

Per questa ragione si tende a utilizzare delle configurazioni sovradimensionate nel timore di non poter reggere i picchi di carico. La conseguenza più importante è il non beneficiare dei vantaggi dell'elasticità fornita da cloud provider come Amazon o Joyent (http://aws.amazon.com/ec2/ http://www.joyent.com/products/compute-service). Infatti questi provider consentono di aggiungere o rimuovere uno o più server in base all'andamento delle metriche provenienti dai tool di monitoraggio e di pagare solo le ore di effettivo utilizzo.

Caching inefficiente

Ulteriore problematica tipica dei *social game* è il ridotto beneficio degli strati di *caching* utilizzati comunemente nelle architetture web. Per aumentare le *performance* e ridurre il carico di lavoro del database, spesso si introduce fra i server che rispondono alle richieste utente e il database stesso uno strato di *caching* realizzato a software. La *cache* verifica se la richiesta proveniente da un server sia già stata effettuata in precedenza e in caso affermativo fornisce la precedente risposta, senza interrogare nuovamente la base dati.

Nei *social game* il carico è molto sbilanciato in scrittura. Questo contrasta con la ragione d'essere dell'utilizzo di *cache*, che punta a velocizzare le letture dal *database*. Paradossalmente quindi, in alcuni casi si potrebbe arrivare a ottenere un peggioramento delle *performance* a causa delle continue invalidazioni dei contenuti in *cache*.

Fortunatamente nel caso dei *social game*, i dati che interessano all'utente durante una sessione di gioco sono spesso *self contained*. Il capitolato qui presentato si propone di utilizzare questa caratteristica per realizzare un'architettura efficente per il supporto di *social game*.

Social Game con Architettura Distribuita (*actor pattern*)

La soluzione individuata per ovviare ai problemi illustrati nell'introduzione è l'utilizzo di un'architettura *stateful*. In un'architettura di questo tipo esiste un processo per ogni sessione di gioco, contraddistinto dalla seguenti caratteristiche:

- può risiedere su una macchina qualunque del *cluster*.

- ha lo stato del giocatore ed è l'unico che lo può modificare.
- gestisce le chiamate alle API per quell'utente tramite l'*actor model*.
- carica lo stato del giocatore all'inizio di una sessione e lo scrive sul DB alla fine di una sessione (*timeout*) oppure periodicamente e alla fine della sessione.

L'interazione fra le componenti di una architettura di tipo *stateful* può essere rappresentata nel diagramma riportato in Figura 2.

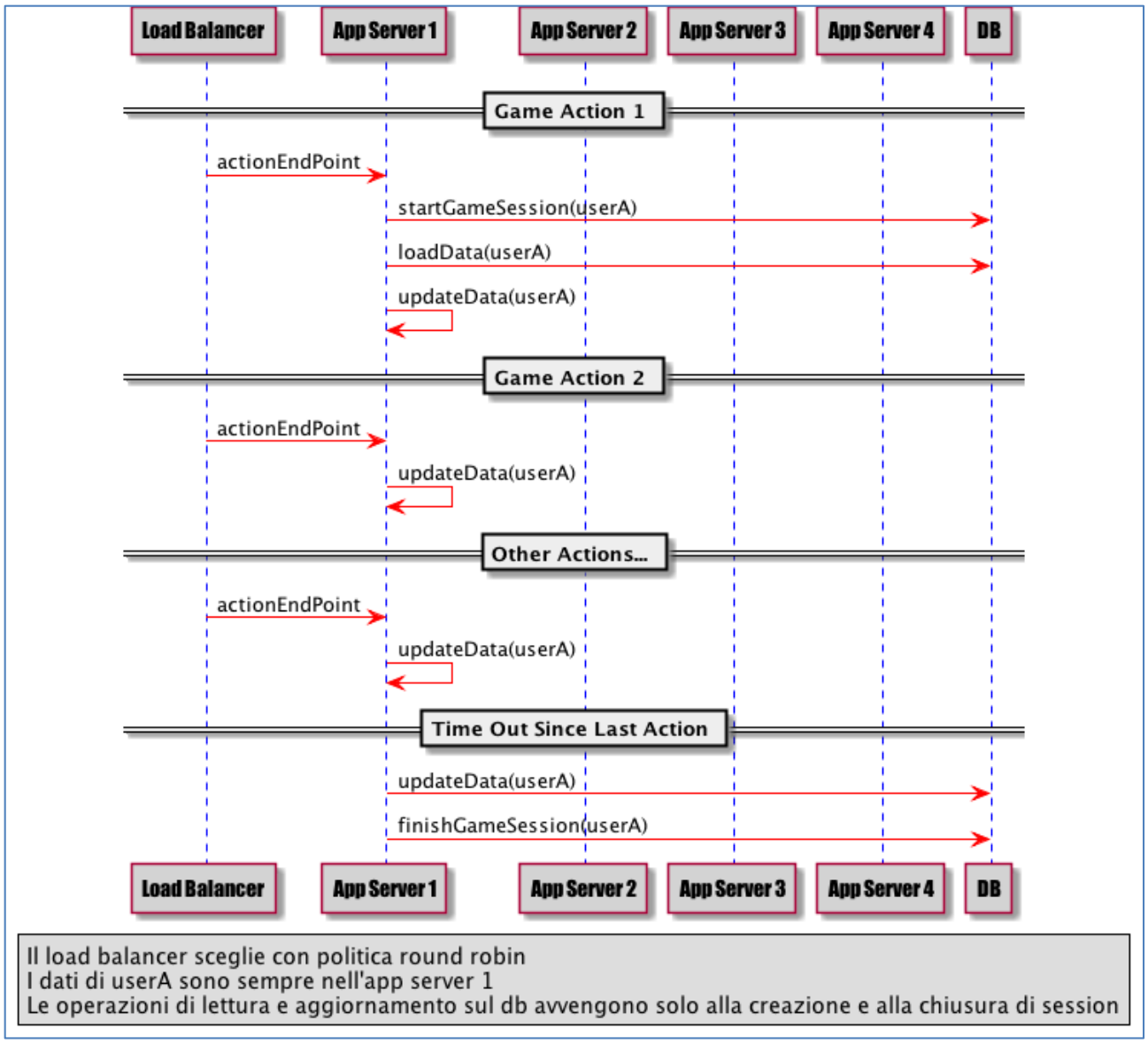

Figura 2: Sequenza di chiamate necessarie a soddisfare una richiesta in un'architettura *stateful*.

Modello ad attori (*actor model*)

Un'architettura di tipo *stateful* può essere realizzata utilizzando un modello ad attori. Il modello ad attori è un particolare modello di programmazione concorrente. Gli attori sono alla base di tale teoria. Ogni attore può ricevere messaggi dall'esterno e sulla base di questi può compiere determinate azioni, quali ad

FunGo Studios Srl
Via Sile, 41 - 31056 Roncade
Treviso - ITALY
P. IVA: 04463380263

mail: team@fungostudios.com
tweet: @fungostudios

esempio creare altri attori, inviare altri messaggi e determinare in che modo rispondere al prossimo messaggio (equivalente al cambiare stato di una macchina a stati).

In modo analogo al modello a oggetti, nel quale tutto è considerato un oggetto, nel modello ad attori tutto è considerato un attore. I due modelli differiscono però per una proprietà fondamentale: mentre nel modello a oggetti le operazioni vengono eseguite in modo sequenziale, il modello ad attori è intrinsecamente concorrente. Non è possibile stabilire un ordine fra le operazioni eseguite dagli attori, che possono quindi essere portate a termine in modo parallelo.

L'utilizzo di un modello ad attori permette di disaccoppiare il mittente di un messaggio dalla sua lettura ed esecuzione, promuovendo quindi la comunicazione asincrona fra gli attori.

Oggetto dell'appalto

Per verificare la validità dell'architettura costruita sulle considerazioni precedenti è richiesta la realizzazione di un gioco con grafica semplificata. Il gioco è di tipo strategico in cui l'utente deve compiere 3 azioni principali:

1. Collezionare risorse
2. Costruire edifici e ampliare la tua squadra.
3. Combattere con altri utenti (non obbligatorio)

Esistono due tipi di risorse: oro e pozioni. L'oro è estratto dalle miniere, le pozioni sono create nelle scuole di magia. Il campo di gioco è un'area quadrata suddivisa in 40x40 sotto-aree quadrate dove il giocatore potrà collocare degli edifici o dei personaggi Sì, per altro ho visto che nelle ultime versioni Akka è diventata l'implementazione ufficiale degli actor:
http://www.scala-lang.org/download/changelog.html: "The original Scala actors are now deprecated. See the actors migration project for more information."
(si veda la Figura 3).

FunGo Studios Srl
Via Sile, 41 - 31056 Roncade
Treviso - ITALY
P. IVA: 04463380263

mail: team@fungostudios.com
tweet: @fungostudios

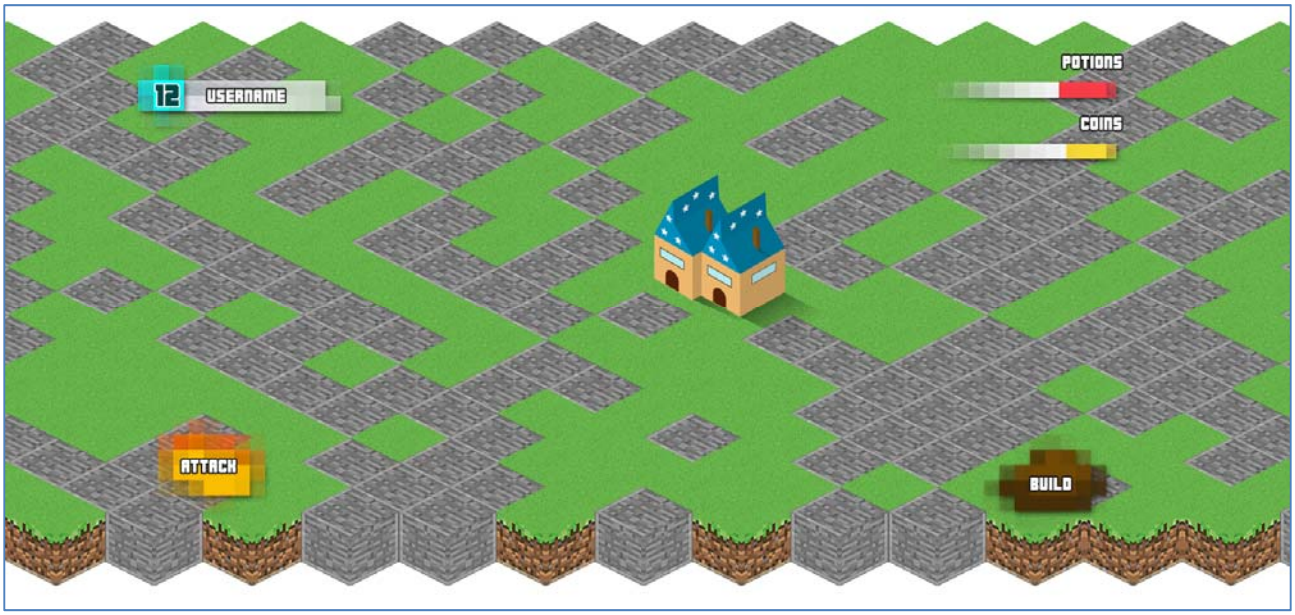

Figura 3: Esempio di area di gioco.

Quando l'utente inizia a giocare gli viene assegnata un'area che contiene solo una miniera, una scuola di magia e due lavoratori. Il giocatore ha bisogno dei lavoratori per costruire gli edifici e altre miniere. Per questa ragione il numero massimo di edifici in costruizione contemporaneamente è 2.

La produzione di risorse è automatica, il giocatore non deve inizializzare la produzione di risorse ma semplicemente attendere che la miniera o la scuola abbiano prodotto la risorsa e raccoglierla. Ad esempio una miniera, che produce 10 chili d'oro ogni 2 ore e può accumulare al massimo 100 chili, si comporterà cosi:

- Tempo 0: 0 chili
- Tempo 5h: 50 chili
- Tempo 10h: 100 chili
- Tempo 20h: ancora 100 chili prodotti perchè il giocatore non li ha raccolti
- Tempo 30h: 0 chili perchè il giocatore li ha raccolti
- Tempo 35h: 50 chili
- e così via...

Gli edifici possono essere costruiti nelle sotto-aree e per semplicità ogni edificio occuperà una singola sotto-area. Ogni edificio ha un livello che va da 1 a 3 e aumenta le capacità di produzione. Gli edifici disponibili in questa prima iterazione del gioco sono:

- Miniera
- Scuole di magia
- Caserma
- Stalla
- Torre dello stregone

FunGo Studios Srl
Via Sile, 41 - 31056 Roncade
Treviso - ITALY
P. IVA: 04463380263

mail: team@fungostudios.com
tweet: @fungostudios

Per costruire un edificio o aumentarne il livello si devono spendere risorse e soddisfare dei vincoli come illustrato nella seguenti tabelle.

Miniera

Livello	Risorse (oro)	Tempo di costruzione (ore)	Altri edifici richiesti	Capacità produttiva
1	100	1	--	10 di oro ogni ora; capienza max 100
2	300	3	Torre dello stregone di livello 2	20 di oro ogni ora; capienza max 200
3	700	7	Torre dello stregone di livello 3	40 di oro ogni ora; capienza max 400

Tabella 1: Caratteristiche di un edificio di tipo Miniera.

Scuola di magia

Livello	Risorse	Tempo di costruzione (ore)	Altri edifici richiesti	Capacità produttiva
1	100 oro, 100 pozioni	1	--	10 di oro ogni ora; capienza max 100
2	300 oro, 200 pozioni	10	Torre dello stregone di livello 2	15 di oro ogni ora; capienza max 150
3	500 oro, 300 pozioni	20	Torre dello stregone di livello 3	25 di oro ogni ora; capienza max 200

Tabella 2: Caratteristiche di un edificio di tipo Scuola di magia.

Stalla

Livello	Risorse	Tempo di costruzione (ore)	Altri edifici richiesti	Capacità produttiva
1	100 oro, 100 pozioni	10	--	--
2	100 oro, 100 pozioni	20	Torre dello stregone di livello 2	--
3	100 oro, 100 pozioni	40	Torre dello stregone di livello 3	--

Tabella 3: Caratteristiche di un edificio di tipo Stalla.

Caserma

Livello	Risorse	Tempo di	Altri edifici	Capacità

FunGo Studios Srl
Via Sile, 41 - 31056 Roncade
Treviso - ITALY
P. IVA: 04463380263

mail: team@fungostudios.com
tweet: @fungostudios

		costruzione (ore)	richiesti	produttiva
1	100 oro, 100 pozioni	5	--	--
2	100 oro, 100 pozioni	10	Torre dello stregone di livello 2, stalla livello 2	Riduce del 10% il tempo di costruzione di una unità
3	100 oro, 100 pozioni	30	Torre dello stregone di livello 3, stalla livello 3	Riduce del 20% il tempo di costruzione di una unità

Tabella 4: Caratteristiche di un edificio di tipo Caserma.

Torre dello stregone

Livello	Risorse	Tempo di costruzione (ore)	Altri edifici richiesti	Capacità produttiva
1	È il solo edificio già presente all'inizio della partita	-	--	--
2	200 oro, 200 pozioni	10	--	--
3	300 oro, 300 pozioni	30	--	--

Tabella 5: Caratteristiche di un edificio di tipo Torre Dello Stregone.

I personaggi possono essere prodotti solo dalla caserma e ciascuno di essi ha un tempo base di produzione e delle risorse necessarie. Il tempo di produzione può essere ridotto se la caserma è di livello superiore a 1.

Unità	Risorse	Tempo di costruzione (ore)
Fante	2 oro, 3 pozioni	0,1
Cavaliere	5 oro, 15 pozioni	0,2
Carro d'assalto	40 oro, 40 pozioni	1

Tabella 6: Caratteristiche dei personaggi.

Requisiti minimi

Il prodotto deve soddisfare i seguenti requisiti:

1. Realizzazione di un *client web* minimale che rappresenti l'area di gioco
2. API per la gestione del *game play*. In particolare:
    i. ogni utente è rappresentato da un attore nel sistema: 1 attore == 1 sessione di gioco
    ii. è possibile interagire con la sessione di un player con API via HTTP
    iii. quando una sessione termina (timeout a 180 secondi o logout esplicito dell'utente) i dati vengono salvati su database di tipo documentale (Mongo, http://www.mongodb.org/, o Couchbase, http://www.couchbase.com/).

FunGo Studios Srl
Via Sile, 41 - 31056 Roncade
Treviso - ITALY
P. IVA: 04463380263

mail: team@fungostudios.com
tweet: @fungostudios

3. L'architettura deve poter esser distribuita su più server
4. Test e report sulle performance dell'architettura sotto carico
5. Utilizzo di uno dei seguenti linguaggi per l'implementazione delle API e degli attori:
    i. Ruby, necessariamente insieme a un *framework* Dcell (https://github.com/celluloid/dcell) che si basa sull'actor model implementato da Celluloid (https://github.com/celluloid/celluloid)
    ii. Scala, necessariamente insieme al *framework* Akka (http://akka.io/)
6. Pubblicazione del progetto su GitHub e utilizzo delle *issue* di github per segnalare i bug.
7. Compatibilità con almeno uno di questi browser: Firefox >= 23.0, Chrome >= 29.0.1597.76

Requisiti opzionali

Il prodotto può soddisfare inoltre i seguenti requisiti:

1. Proporre ed implementare un meccanismo di interazione tra più giocatori (ad esempio delle battaglie tra villaggi)
2. Valutazione e implementazione di metodi per ottimizzare la distribuzione delle sessioni su più server
    a. quando aggiungere un nuovo server?
    b. come decidere su che macchina creare una nuova sessione per un giocatore?
3. Gestione dell'aggiornamento del codice mantenendo attive le sessioni di gioco correnti.
4. Sperimentare la supervisione degli attori per gestire eventuali fallimenti (crash)

Variazione dei requisiti

Non sono ammesse variazioni se non a evidente miglioramento di quanto richiesto dal committente. Non è esclusa la comunicazione, da parte del committente, di variazioni ai requisiti sia precedentemente alla consegna delle offerte che durante la realizzazione del sistema.

Documentazione

La consegna del sistema dovrà essere accompagnata dai necessari manuali d'uso e da ogni altra documentazione tecnica necessaria per l'utilizzo del prodotto da parte del personale operatore del committente. E' gradita la versione bilingue italiano e inglese.

Garanzia e manutenzione

Il fornitore dovrà garantire in sede di collaudo (Revisione di Accettazione) il funzionamento corretto del sistema. L'eliminazione dei difetti e delle non conformità eventualmente emersi in sede di collaudo sono a totale carico del fornitore.

Le modalità di collaudo saranno proposte dal fornitore e costituiranno titolo per la valutazione dell'offerta ai fini dell'aggiudicazione dell'appalto. I dati di collaudo costituiscono parte integrante delle modalità di collaudo. Le modalità di collaudo saranno considerate definitive e contrattualmente vincolanti solo a seguito di formale approvazione da parte del committente.

FunGo Studios Srl
Via Sile, 41 - 31056 Roncade
Treviso - ITALY
P. IVA: 04463380263

mail: team@fungostudios.com
tweet: @fungostudios

Rinvio

Per tutto quanto non previsto nel presente capitolato, sono applicabili le disposizioni contenute nelle leggi e nei collegati per la gestione degli appalti pubblici.

Proponente

FunGo Studios (http://www.fungostudios.com/) è un game studio che realizza giochi mobile e API per *gamification erogate* in modalità *Software as a Service* (SaaS). Dal 2011 ha realizzato due giochi di strategia ed è prossimo al lancio di un action game in 3D.

FunGo Studios è rappresentata da:

- Nicola Brisotto (nicola@playgowar.com): responsabile del progetto
- Matteo Centenaro (matteo@playgowar.com): responsabile dello sviluppo