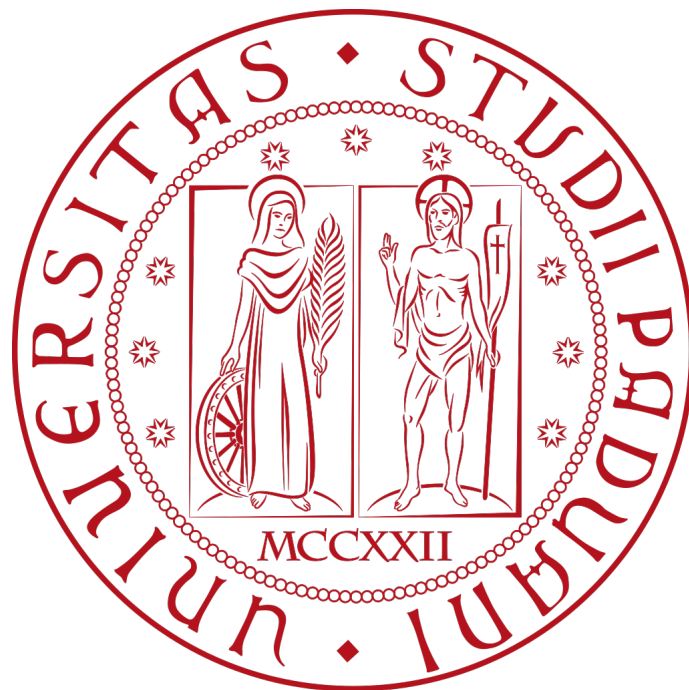


UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”

SCUOLA DI SCIENZE
CORSO DI LAUREA IN INFORMATICA



SERVIZI DI SUPPORTO ALLE TRANSAZIONI SU BLOCKCHAIN
ETHEREUM

Informazioni sul documento

Titolo	Servizi di supporto alle transazioni su Blockchain Ethereum
Creazione	10 Novembre 2018
Redazione	Aaron Cesaro
Email di riferimento	aaron.cesaro@studenti.unipd.it

Sommario

Questo documento si presenta come Tesi di Laurea triennale per il corso di laurea in informatica dello studente Aaron Cesaro. In esso è contenuta una relazione dettagliata dell'attività di stage svolta presso l'azienda Sgame SA, situata a Lugano (Svizzera). Durante il tirocinio, della durata complessiva di 300 ore, ogni obiettivo prefissato è stato raggiunto con successo, permettendo allo studente di acquisire dettagliate conoscenze sulle tecnologie e le metodologie di sviluppo utilizzate dalla società. Particolare enfasi è stata data alla comprensione ed all'utilizzo della blockchain *Ethereum* e di tutti i servizi di supporto necessari alla corretta integrazione dello stesso all'interno della piattaforma *Sgame Pro*.

Indice

1	Introduzione	5
1.1	Contenuto del documento	5
1.2	Norme tipografiche	5
2	Blockchain network	6
2.1	Cos'è una Blockchain	6
2.1.1	Definizione	6
2.1.2	Server based e P2P	6
2.2	Blockchain e Bitcoin	7
2.2.1	Ledger	7
2.2.2	Transazioni	9
2.2.3	Blocchi	10
2.2.4	Mining	11
2.3	Ethereum	12
2.3.1	Cos'è Ethereum	12
2.3.2	Smart Contract	12
3	La piattaforma Sgame Pro	13
3.1	Overview	13
3.2	Funzionalità principali	14
3.2.1	Challenges	14
3.2.2	Leaderboard	15
3.2.3	User Profile	15
3.2.4	Marketplace	16
3.2.5	Wallet	16
3.3	Sgame Pro ed Ethereum	17
3.3.1	Sgame Token	17
4	Token Value Service	19
4.1	Overview	19
4.2	Pianificazione	20
4.3	Analisi dei requisiti	21
4.3.1	Notazioni	21
4.3.2	Specifica dei requisiti	21
4.4	Architettura	22
4.5	Progettazione	23
4.5.1	Tecnologie	23
4.6	Sviluppo	24
4.6.1	Metodologia di sviluppo	24
4.6.1.1	Scrum	24
4.6.2	Processo di implementazione e rilascio	25

4.6.2.1	Processo di implementazione delle features	26
4.7	Integrazione	27
5	Ethereum Testnet	28
5.1	Overview	28
5.2	Pianificazione	29
5.3	Analisi dei requisiti	30
5.3.1	Specifica dei requisiti	30
5.4	TX Service	31
5.4.1	Overview	31
5.4.2	Specifiche	32
5.5	Private Tesnet	33
5.5.1	Strumenti utilizzati	33
5.5.2	Creazione	34
5.5.3	Integrazione	35
5.6	Public Tesnet	37
5.6.1	Integrazione	37

1 Introduzione

1.1 Contenuto del documento

All'interno del documento verranno trattati i seguenti argomenti:

- **Capitolo 2: Blockchain network:** verrà descritta la rete *blockchain*, le tecnologie che ne permettono il funzionamento, le logiche di base su cui si fondano le transazioni e le principali differenze tra le piattaforme *Bitcoin* ed *Ethereum*.
- **Capitolo 3: La piattaforma Sgame Pro:** in questo capitolo verrà presentata l'applicazione *Sgame Pro* e l'utilizzo di *Ethereum* all'interno della stessa. Verrà inoltre data una visione d'insieme sulla piattaforma oltre a giustificare le motivazioni alla base dello sviluppo del servizio *Token Value*.
- **Capitolo 4: Token Value service:**
- **Capitolo 5: Considerazioni finali:**

1.2 Norme tipografiche

2 Blockchain network

2.1 Cos'è una Blockchain

2.1.1 Definizione

Blockchain è una tecnologia che permette la creazione ed amministrazione di un grande database distribuito tramite la gestione di transazioni condivisibili tra più nodi di una rete peer-to-peer. Si tratta quindi di un database strutturato in blocchi (*block*) che sono tra loro collegati (*chain*) in modo che ogni transazione avviata sulla rete debba essere validata dalla rete stessa. In estrema sintesi la *blockchain* è rappresentata da una catena di blocchi che contengono e gestiscono più transazioni facendo uso della crittografia per rendere sicuro l'immagazzinamento di dati ed il trasferimento di strumenti di valuta.

Pur essendo quest'ultima la definizione "formale" di *blockchain*, ritengo che essa non evidenzia con chiarezza e semplicità cosa effettivamente una *blockchain* sia.

Proverò quindi a scomporre ed analizzare la prima parte della definizione per rendere più fruibile il concetto.

2.1.2 Server based e P2P

La rete che normalmente viene utilizzata tutti i giorni per navigare in internet è quasi sempre *server based* [Figura 1a].

La peculiarità di questo sistema è che tutte le informazioni sono contenute in un solo posto, il *Server* (da qui il nome *server based*), il quale spesso gestisce un database che esercita il ruolo di archivio per l'immagazzinamento di dati ed effettua su di essi ricerche qualora vengano richiesti.



(a) Server based network

(b) P2P based network

Figura 1: Tipologie di network

A differenza di una rete *server based*, in una rete *peer-to-peer* [Figura 1b] (anche detta *P2P*) non esiste la presenza di un server centrale che invia informazioni e tutti i

dati vengono scambiati direttamente tra i nodi collegati alla rete. Ogni utente è quindi un *client* ed un *server* contemporaneamente. Proprio per questa duplice funzione ogni dispositivo connesso viene detto *nodo* della rete.

La sostanziale differenza tra le due tipologie di rete risiede nel fatto che mentre nel primo caso il *server* contiene tutte le informazioni, nel secondo sono tutti e soli i *client* a contenere i dati.

Ciò significa che nel primo caso il proprietario del *server* può aggiungere, modificare o eliminare i dati che sono contenuti in esso, mentre nel secondo, anche se un nodo cancella o modifica i propri dati, gli altri nodi conterranno comunque tutte le informazioni originali. Da qui il termine distribuito.

È ora possibile riprendere in mano la definizione originale: *blockchain* è una tecnologia che permette di creare e gestire un grande archivio di informazioni, non contenute in un unico posto, ma del quale esiste una copia in ogni nodo connesso alla rete.

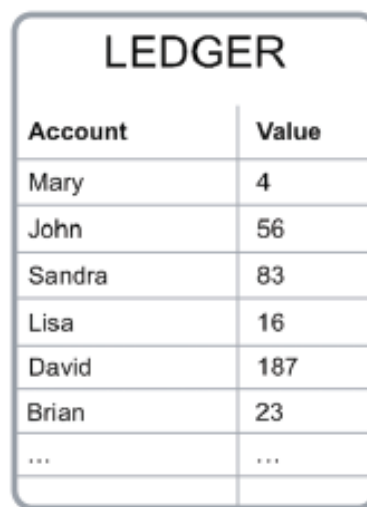
2.2 Blockchain e Bitcoin

2.2.1 Ledger

In letteratura è ritenuto più intuitivo usare *Bitcoin* (*BTC*) per esporre, tramite esempi semplificati, il funzionamento di una *blockchain*.

Un *Bitcoin* è una singola unità di valuta digitale che, proprio come l'*Euro* non ha valore intrinseco, se non quello intenzionalmente attribuitogli grazie al consenso di scambio per l'acquisizione di beni o servizi.

Per tenere traccia della quantità di *Bitcoin* che ogni utente possiede si utilizza ciò che viene definito un *Ledger* (libro mastro), che altro non è che un file in cui viene tenuta traccia di tutte le transazioni. Il *ledger* non è contenuto in un server centrale come ad esempio quello di una banca, ma ne esiste una copia in ogni nodo partecipante alla rete. In [Figura 2] è riportato, anche se in modo estremamente semplificato, un esempio di *ledger*. Anche se nella realtà un ledger è molto diverso da quello in figura, nella pratica il disegno rappresenta fedelmente la funzione principale ricoperta da ogni copia del *ledger* posseduta dai singoli nodi. Nella colonna **Account** viene riportato il nome del proprietario dei *Bitcoin*, mentre nella colonna **Value** è indicata la quantità posseduta da ognuno dei partecipanti.



LEDGER	
Account	Value
Mary	4
John	56
Sandra	83
Lisa	16
David	187
Brian	23
...	...

Figura 2: Ledger

Mettiamo caso che David voglia inviare cinque *Bitcoin* a Sandra. Per farlo è necessario che David mandi un messaggio sulla rete, il quale contiene la richiesta di transazione ed il numero di Bitcoin da lui posseduti. Come informazione aggiuntiva viene inoltre trasmessa la quantità di *Bitcoin* che possederà Sandra nel caso in cui la transazione avesse luogo. Il messaggio viene raggiunto dai nodi vicini a David i quali aggiornano i propri ledger con il risultato della possibile transazione (cioè David -5 *BTC* e Sandra +5 *BTC*) e rinviando il messaggio ai nodi a loro adiacenti. In questo modo il messaggio si espande per tutta la rete.

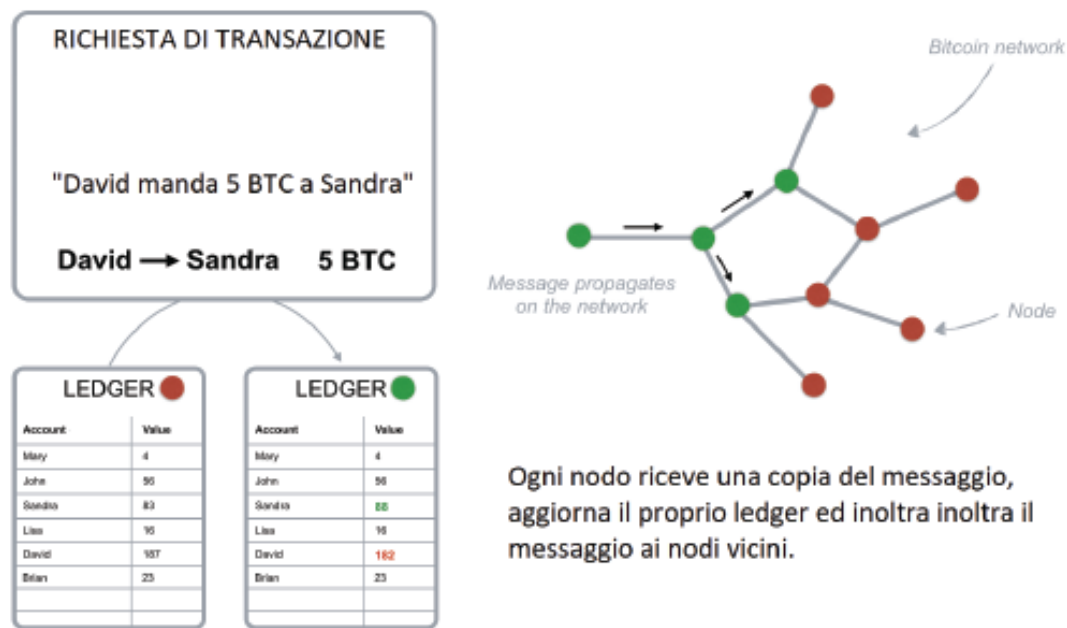


Figura 3: Richiesta di transazione tra due nodi

Il fatto che il ledger sia mantenuto da tutti i nodi implica tre cose fondamentali, che stanno alla base del concetto della blockchain:

- tutti sono a conoscenza di tutte le transazioni che avvengono sulla rete;
- se la transazione non va a buon fine nessuno se ne prende la responsabilità in quanto non esiste un'entità centrale che si prenda carico dell'esito delle transazioni;
- non esiste il bisogno di garanzie o fiducia in quanto la sicurezza è ottenuta tramite particolari funzioni matematiche estremamente sicure.

2.2.2 Transazioni

Perchè una transazione possa avere luogo è necessario ciò che viene definito un *Wallet* (portafogli), ossia un software che permetta di depositare e scambiare scriptovaluta, tra cui *Bitcoin*. Poichè deve essere possibile solo ed esclusivamente al proprietario di un determinato *Wallet* inviare i propri *Bitcoin*, ogni *Wallet* è protetto tramite una tecnica crittografica che usa una coppia di chiavi tra loro connesse. Esse prendono il nome di chiave privata (*private key*) e chiave pubblica (*public key*).

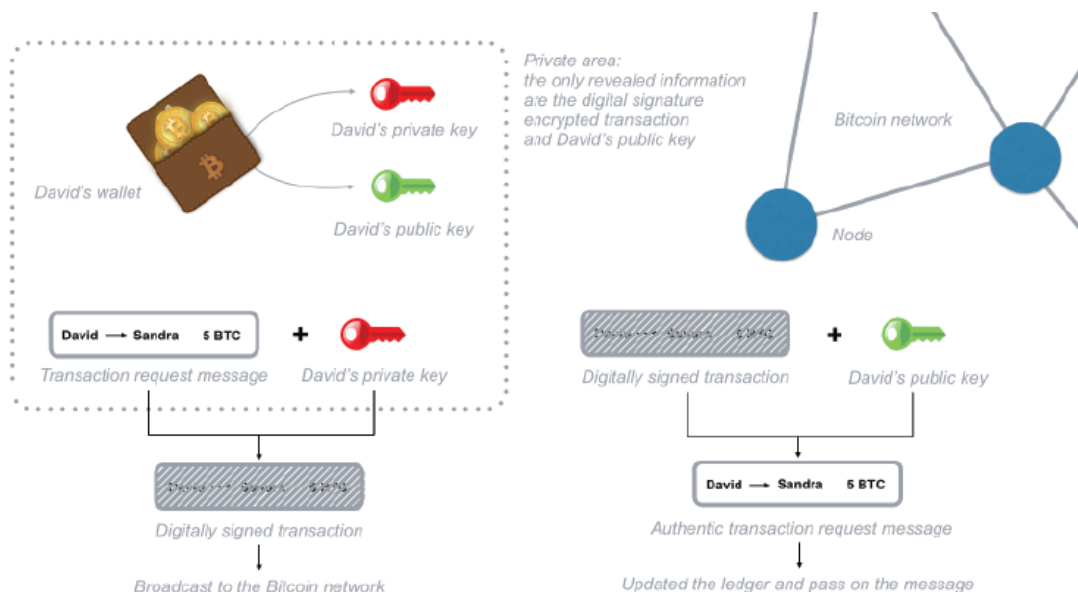


Figura 4: Verifica della transazione tramite chiavi

Ogni messaggio in uscita da un singolo indirizzo viene criptato con una chiave privata il quale, una volta derivata la corrispondente chiave pubblica con cui è possibile identificare univocamente l'indirizzo di partenza (*from*), deve poi essere validato dal nodo locale. La validazione è necessaria per accertarsi che la transazione sia stata realmente messa sulla rete dal proprietario dell'account. A questo punto solo i possessori della chiave pubblica associata potranno decifrare il messaggio.

Quando David vuole mandare 5 *Bitcoin* a Sandra, deve inviare sulla rete il messaggio criptato con la sua chiave privata in modo che venga identificato come il possessore di un certo numero di *Bitcoin* e sia di conseguenza l'unico a poter sbloccare il proprio *Wallet*. Tutti gli altri nodi validano la transazione, verificando tramite la chiave pubblica di David che la richiesta di inviare valuta sia effettivamente partita da lui. In questo modo si ottiene la validazione della transazione. In altre parole per poter inviare un *Bitcoin* è necessario provare alla rete di essere i possessori dell'indirizzo da cui partono i *Bitcoin*. Nella rete inoltre non viene tenuto conto del bilancio dei singoli utenti, ma vengono semplicemente registrate le transazioni che avvengono. La verifica di una transazione in

questo modo si riduce semplicemente al controllo di tutte le transazioni passate, effettuate dall'utente che vuole inviare una certa somma di *Bitcoin*.

2.2.3 Blocchi

Su una *blockchain* le transazioni vengono ordinate tramite accorpamento con altre transazioni avvenute in un lasso di tempo definito. In altre parole più transazioni vengono raggruppate insieme ed inserite dentro a quello che viene chiamato *Block* (Blocco). Ogni *block* contiene quindi un definito numero di transazioni ed un collegamento al nodo precedente. In questo modo si viene a creare una catena di blocchi molto simile ad una *linked list*. Da qui il nome *blockchain* (catena di blocchi).

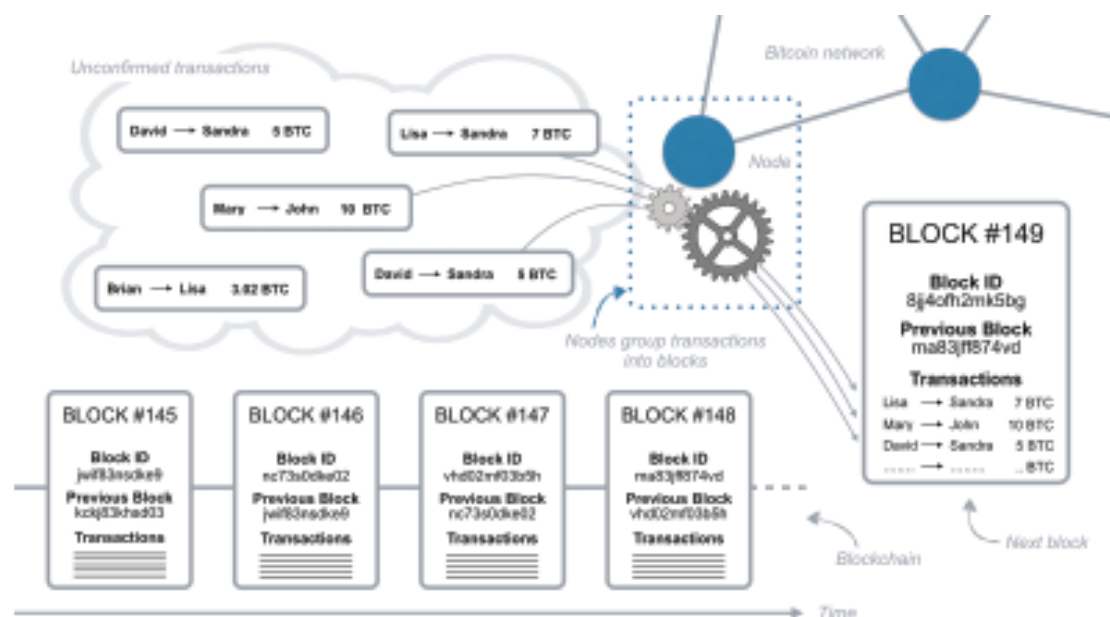


Figura 5: Rappresentazione di una blockchain

Le transazioni contenute nello stesso blocco sono considerate come avvenute nello stesso lasso temporale, mentre le transazioni che ancora non sono state raggruppate in un blocco sono considerate *Unconfirmed*, cioè non ancora validate.

Ogni nodo della rete può raggruppare più transazioni e creare un blocco, suggerendo alla rete di inserirlo come prossimo blocco della catena.

Per essere effettivamente inserito nella rete un blocco deve contenere la soluzione ad un complesso problema matematico che, per essere risolto richiede una grossa potenza di calcolo, ed un pò di fortuna. La risposta altro non è che un numero e l'unico modo per sapere quale sia il numero corretto da inserire consiste nel provarli tutti. Il nodo che per primo risolve il problema acquisisce il diritto di inserire il prossimo blocco sulla catena e lo invia a tutti i nodi adiacenti.

2.2.4 Mining

A questo punto sorge spontaneo una domanda, ossia: "Se i Bitcoin posseduti da un account sono il risultato della somma di tutte le transazioni inviate e ricevute da quell'account, come è possibile ottenere altri *Bitcoin*?"

La risposta a questa domanda é: "tramite il *mining*".

Il *mining* è l'attività svolta dai nodi definiti *miners*, i quali, tramite la risoluzione di un complesso problema matematico, validano i blocchi, permettendo così a tutti i partecipanti alla rete di inviare e ricevere transazioni.

La validazione di un blocco nella pratica è una attività molto dispendiosa, sia in termini di energia elettrica che di consumo di banda.

Perchè la catena possa proseguire (cioè perchè possano essere effettuate nuove transazioni) è necessario che i blocchi siano inseriti nella catena e per farlo è necessario risolvere questo problema matematico.

Il modo escogitato per ripagare chi indovina il numero che valida il blocco (cioè svolge il lavoro di *miner*) è una ricompensa in *Bitcoin* da parte della rete. Questa ricompensa è ciò che incentiva le persone a provvedere al necessario lavoro computazionale per far continuare la catena e mantenere la rete utilizzabile. Senza i *miners* i blocchi non potrebbero essere validati, la catena si fermerebbe e le transazioni non potrebbero più avere luogo.

2.3 Ethereum

2.3.1 Cos'è Ethereum

Come *Bitcoin*, *Ethereum* è una *public blockchain*.

Sebbene ci siano alcune significative differenze tecniche tra le due, la distinzione più importante da notare è che *Bitcoin* ed *Ethereum* differiscono sostanzialmente per scopo e capacità.

Il *Bitcoin* è stato lanciato come valuta alternativa, o moneta digitale, ed offre una particolare applicazione della tecnologia *blockchain*, ossia un sistema di pagamento elettronico.

Ethereum invece viene principalmente utilizzato per applicazioni decentralizzate tramite l'utilizzo degli *Smart Contract*.

A differenza di *Bitcoin*, *Ethereum* utilizza due concetti di *token*: il primo prende il nome di *Ether* e corrisponde alla "moneta" effettivamente scambiata tra gli utenti della rete, il secondo viene invece utilizzato per pagare i *miners*, i quali includono le transazioni nei blocchi, e prende il nome di *gas*.

2.3.2 Smart Contract

Uno *Smart Contract* è un programma che contiene un insieme di regole a cui le parti interessate accettano di aderire.

Nel caso in cui le regole definite all'interno di uno smart contract siano soddisfatte l'accordo tra le parti viene automaticamente applicato.

Il codice di uno *Smart Contract* facilita, verifica e impone la negoziazione o l'esecuzione di un accordo o di una transazione e corrisponde alla forma più semplice di automazione decentralizzata.

Il successo di *Ethereum* (e la più grande differenza con *Bitcoin*) dipende proprio dal concetto di *Smart Contract*. grazie a cui è possibile programmare una serie definita di azioni che vengono attuate se e solo se le condizioni in esso contenute vengono soddisfatte.

3 La piattaforma Sgame Pro



3.1 Overview

Sgame Pro è un aggregatore di *mobile games* di proprietà di *Sgame SA*, con sede a Lugano, Svizzera.

In sviluppo dal 2016, *Sgame Pro* ha lanciato con successo la sua *Alpha version* nel 2017, raggiungendo oltre 50.000 download senza alcuna spesa di marketing.

Sgame Pro è interamente focalizzato sull'industria dei *mobile games* e ha sviluppato due importanti innovazioni tecniche:

- Consentire ai giocatori di essere remunerati con un nuovo utility token (*SGM*) semplicemente giocando con il proprio cellulare ai titoli proposti;
- Aggregare il frammentato settore dei *publisher*, indipendenti e non, in un'unica piattaforma di gioco *one-stop-shop*.

L'*SGM* (token di tipo *utility* basato sullo standard *Ethereum ERC-20*) sarà l'unico modo per accedere ai servizi e beneficiare di tutte le funzionalità messe a disposizione degli utenti, oltre ad essere il metodo di pagamento unico per tutte le transazioni all'interno dell'ecosistema della piattaforma.

All'interno dell'applicazione i giocatori non solo avranno l'opportunità di trovare tutti gli ultimi titoli mobile resi disponibili, ma avranno anche l'opportunità di sfidare gli altri utenti sia in privato che tramite la funzionalità di *Public Challenge*.

Quest'ultima innovazione è davvero dirompente dato che il 78% del mercato dei giochi mobile è single player, mentre la maggior parte delle entrate derivano da giochi multi-player.

Gli *Influencers* di *Sgame Pro* includono *Pewdiepie*, *Tweakbox* e molti altri, con oltre 80 milioni di seguaci altamente coinvolti e sparsi in tutto il mondo.

3.2 Funzionalità principali

3.2.1 Challenges

Le *Challenges* sono una delle caratteristiche principali di *Sgame Pro*.

Esse consentono infatti di sfidare gli altri giocatori all'interno della piattaforma e di guadagnare gli *SGM* messi in palio per la sfida.

Le *challenges* permettono inoltre di trasformare qualunque gioco da semplice *single play* a *multiplayer*, dando agli utenti la possibilità di sfidarsi in modalità asincrona.

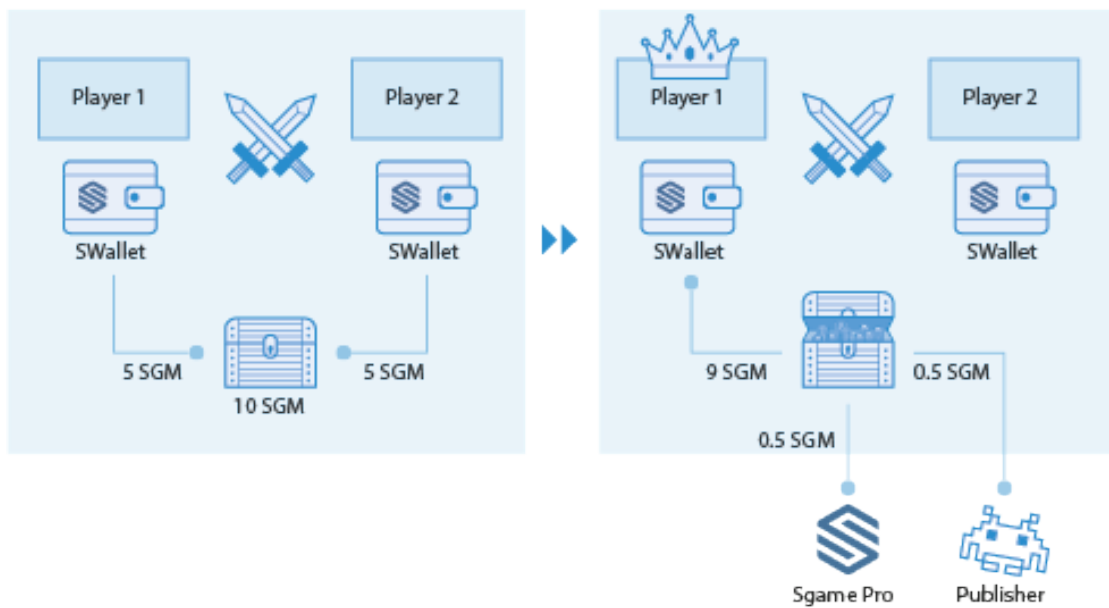


Figura 6: Funzionalità Challenges

All'interno dell'applicazione esistono due diverse tipologie di sfide:

- **Private Challenges:** le quali consentono agli utenti di sfidarsi tra loro, scegliendo tutti i partecipanti alla competizione direttamente tra i propri *Friends*, e mettendo in palio l'importo in *SGM* desiderato;
- **Public Challenges:** le quali invece danno la possibilità di creare una sfida *ad-hoc* a cui chiunque può partecipare, specificando il numero minimo/massimo di utenti necessario affinché la sfida possa avere luogo.

3.2.2 Leaderboard

La *Leaderboard* è la classifica globale di punteggi ottenuti dagli utenti in uno specifico titolo.

Questa funzionalità consente ai giocatori di confrontare le proprie prestazioni con quelle degli avversari.

L'innovativo approccio di Sgame Pro a questa "classica" funzionalità consiste nell'esporre:

- **Classifiche comuni tra iOS ed Android:** i giocatori possono finalmente competere tra loro indipendentemente dal sistema operativo utilizzato;
- **Classifiche premiate:** i giocatori che battono particolari record od obiettivi vengono premiati in *SGM*.

3.2.3 User Profile

Ogni giocatore o *influencer* avrà il proprio profilo utente dedicato contenente una panoramica di punteggi, tempi di gioco ed altre statistiche utili.

Il profilo utente contiene inoltre le seguenti informazioni:

- livello dell'utente;
- numero di amici e *followers*;
- statistiche delle *challenges*;
- titoli giocati di recente;
- dati sugli avversari;
- dattagli sui guadagni.



Figura 7: Profilo

3.2.4 Marketplace

La sezione *Marketplace* permette agli utenti della piattaforma *Sgame Pro* di acquistare *digital goods* (coupons, free subscriptions etc..) direttamente all'interno dell'applicazione. È inoltre possibile fissare dei "target", ossia scegliere un particolare oggetto all'interno del *marketplace* e porsi come obiettivo il raggiungimento della cifra necessaria al suo acquisto.

3.2.5 Wallet

All'interno della funzionalità *Wallet* l'utente può sempre tenere sotto controllo il suo patrimonio ed avere una panoramica dettagliata del flusso di cassa (in *SGM*) del suo account.

Da questa sezione è inoltre possibile trasformare gli *SGM* guadagnati all'interno della piattaforma in criptovaluta fruibile all'interno della rete *Ethereum*.

È infatti tramite questa funzionalità che entrano in gioco la *blockchain* e tutti i servizi di supporto ad essa associati.

3.3 Sgame Pro ed Ethereum

L'idea iniziale alla base dell'integrazione di *Ethereum* all'interno della piattaforma *Sgame Pro* prevedeva un'applicazione completamente decentralizzata (*full decentralized*) a cui ogni movimento di *SGM* corrispondeva una transazione su *Ethereum*.

Questo approccio avrebbe portato numerosi benefici a livello di *trust*, ma sarebbe stato altamente insostenibile per il *business model* della società.

Infatti, come già detto, ogni transazione su *Ethereum* richiede una determinata quantità di *gas*, necessaria per l'esecuzione dello *smart contract*, che deve essere aggiunta all'ammontare che si desidera trasferire. In altre parole il costo di ogni transazione sarebbe stato maggiore dell'ammontare scambiato durante la transazione stessa.

Si è deciso quindi di utilizzare un sistema di remunerazione interno "classico", ossia tramite assegnazione a livello di database, dando però la possibilità agli utenti di trasferire i propri guadagni sul proprio *wallet* personale qualora lo desiderassero.

3.3.1 Sgame Token

Ethereum è stato creato per essere un vero e proprio ambiente di sviluppo, infatti esso può essere utilizzato, tramite la creazione di appositi *Smart Contract*, per creare criptovalute che si appoggino sulla sua *currency* base, ossia l'*Ether*.

Le criptovalute derivate da *Ether*, dette anche *token*, sono principalmente state utilizzate per l'attuazione delle *ICO* (*Initial Coin Offer*), ossia *crowdfundig* attuati tramite la vendita di particolari criptovalute, utilizzate come finanziamenti al posto della moneta tradizionale (*FIAT*).

Come già citato l'*SGM* è il token utilizzato dalla piattaforma *Sgame Pro*.

Esso è stato creato allo scopo di uniformare quelli che vengono definiti *In-App purchase*, permettendo così agli utenti di poter acquistare qualunque tipo di beneficio, all'interno dei titoli presenti nella piattaforma, con una unica valuta.

L'*SGM* si appoggia su quello che formalmente *Ethereum* definisce lo *Standard ERC-20*, ossia un'interfaccia che gli *Smart Contract* possono implementare nel caso in cui si intenda utilizzare alcune tipologie di servizi offerti da *Ethereum*.

```
contract ERC20Interface {
    function totalSupply() public view returns (uint);
    function balanceOf(address tokenOwner) public view returns (uint balance);
    function allowance(address tokenOwner, address spender) public view returns (uint
        remaining);
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool success);
    function transferFrom(address from, address to, uint tokens) public returns (bool
        success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
}
```

L'interfaccia *ERC-20* funge da contratto per il set minimo di funzionalità che si intende mettere a disposizione dopo la creazione (*issuing*) di un token derivato da *Ethereum*. La scelta di aderire a questo standard è stata guidata dall'altissima compatibilità con i vari *Exchange* presenti oggi sul mercato.

4 Token Value Service

4.1 Overview

Token Value è un servizio di acquisizione e memorizzazione dello storico di valori di criptovalute sulle diverse piattaforme di scambio (*Exchange*).

Il servizio espone delle *API* pubbliche, utilizzate direttamente dalla piattaforma *Sgame Pro*, per il calcolo delle vincite dei giocatori ed il prezzo dei beni acquistabili nella sezione *Marketplace*.

Grazie all'implementazione di questo servizio il sistema può applicare le logiche di business basate sul calcolo del prezzo presentato all'interno dell'applicazione, in modo da assicurare equità nell'attribuzione di *SGM* e coerenza del valore riportato sugli oggetti acquistabili all'interno della stessa.

Le funzionalità principali del servizio comprendono:

- ottenimento del valore del token *SGM* tramite l'utilizzo delle API messe a disposizione di ciascun exchange;
- memorizzazione persistente dei valori ottenuti su database PostgreSQL;
- caching dei valori per la diminuzione dei tempi di risposta;
- utilizzo e trasmissione di valori ottimali secondo le logiche di selezione approvate dalla società.

Il servizio rappresenta attualmente un tassello fondamentale per la società *Sgame SA* in quanto il valore del token riportato e propagato all'intera applicazione ha un impatto estremamente concreto sui guadagni dell'intera società.

Una logica scorretta all'interno del servizio porterebbe irreparabilmente ad una erronea attribuzione delle vincite agli utenti e ad una possibile perdita di denaro per la società.

Proprio per queste ragioni il servizio utilizza logiche estremamente semplici e facilmente testabili.

4.2 Pianificazione

- **Prima Settimana**(42,5 ore) – Incontro con il team di sviluppo, formazione sul funzionamento di Ethereum ed introduzione ai linguaggi C# e Solidity;
- **Seconda Settimana**(42,5 ore) – Acquisizione delle informazioni necessarie alla realizzazione del servizio TokenValue ed ai servizi interni che utilizzerà ;
- **Terza Settimana**(42,5 ore) – Sviluppo della logica di acquisizione e memorizzazione persistente dei valori necessari all'utilizzo del servizio TokenValue.
- **Quarta Settimana**(42,5 ore) – Debugging e sviluppo della documentazione del servizio TokenValue. Entro il termine della settimana il tirocinante dovrà aver terminato e documentato il servizio sviluppato.
- **Quinta Settimana**(42,5 ore) – Creazione dell'interfaccia grafica necessaria agli analisti di Sgame per la visualizzazione dei valori di mercato della criptovaluta target.

4.3 Analisi dei requisiti

4.3.1 Notazioni

Viene di seguito riportata la notazione che verrà utilizzata per l'identificazione dei requisiti classificati per utilità strategica:

- **Ob** – requisiti obbligatori, irrinunciabili per qualsiasi Stakeholders.
- **De** – requisiti desiderabili, aggiungono valore al prodotto.

Viene di seguito riportata la notazione che verrà utilizzata per l'identificazione dei requisiti classificati per verificabilità:

- **Vi** – requisiti vincolo, imposti dal cliente o dal sistema in cui lavora il software.

4.3.2 Specifica dei requisiti

Obbligatori:

- **Ob001** – il prodotto *Token Value* deve raccogliere i dati sui valori delle criptovalute tramite chiamata *REST API* al servizio cryptocompare.
- **Ob002** – il prodotto *Token Value* deve salvare dati persistenti sia sul valore attuale del token che sullo storico dei valori richiesti.
- **Ob003** – il prodotto *Token Value* deve essere corredato da documentazione.

Desiderabili:

- **De001** – il prodotto *Token Value* deve essere documentato tramite *XML notation*.
- **De002** – il prodotto *Token Value* deve essere facilmente configurabile sia in ambiente *Development* che in ambiente *Staging* tramite l'utilizzo di specifiche variabili d'ambiente.

Vincolo:

- **Vi001** – il prodotto *Token Value* deve essere realizzato nel linguaggio *C#* ed utilizzare *PostgreSQL* per il salvataggio persistente dei dati.

4.4 Architettura

L'architettura del servizio riprende fedelmente il modello utilizzato dagli altri servizi relativi ad *Ethereum* sviluppati sulla piattaforma.

Ogni servizio è composto da due moduli separati:

- **Public API:** interfaccia pubblica che espone *API* di tipo *REST* all'interno dell'ambiente stesso;
- **Core:** *business logic* del servizio. Si occupa delle richieste verso i servizi esterni e dell'interfacciamento con il database. Nel modulo sono quindi compresi sia il *Business Layer* che il *Data Layer*.

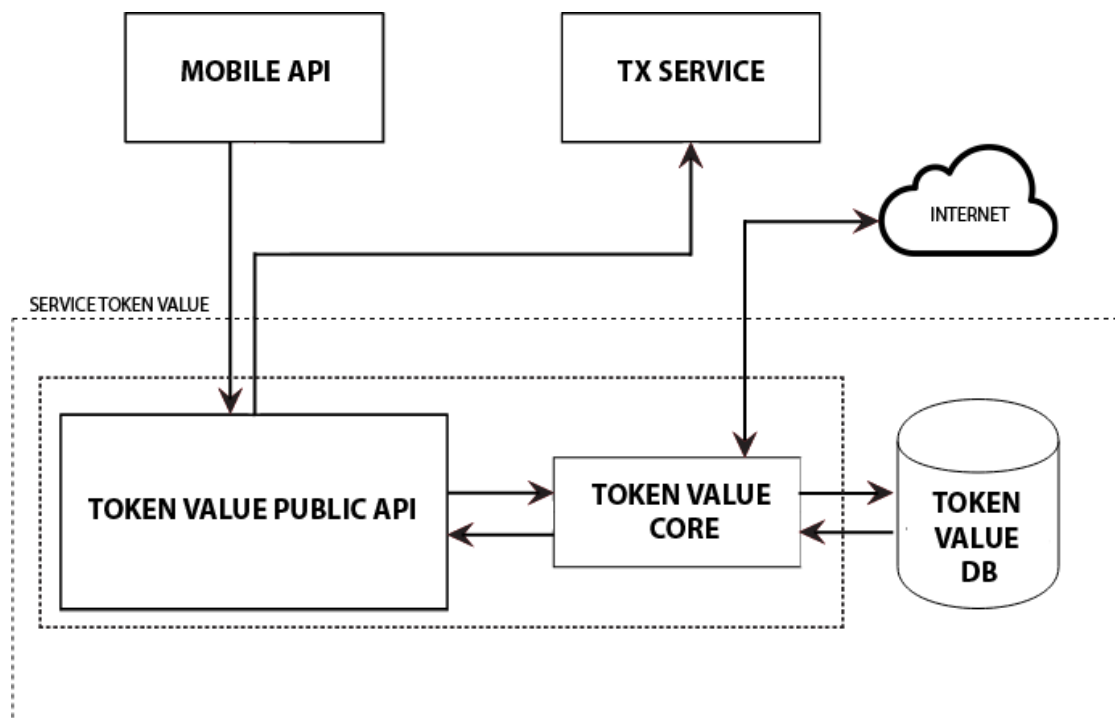


Figura 8: Architettura del servizio Token Value

Grazie a questa struttura a *layers* si garantisce l'impossibilità di comunicazione diretta con la logica del servizio ed una netta separazione delle responsabilità all'interno del servizio stesso.

Il servizio *Token Value* si interfaccia poi con il resto del backend dell'applicazione (quello che la compagnia ha deciso di chiamare *Mobile API*), e con *TX*, ossia il servizio che si occupa della comunicazione diretta con lo *Smart Contract* interfacciandosi alla *blockchain Ethereum*.

4.5 Progettazione

Viene di seguito esposto il diagramma delle classi (*UML 2.0*) del servizio *Token value*. All'interno nel servizio è possibile mappare i due moduli riportati nello schema dell'architettura nel seguente modo:

- **Public API** contiene la classe *ValueController*, la quale funge da interfaccia interna e viene richiamata da *Mobile API*;
- **Core** comprende tutto il resto del servizio, ossia i *Managers* ed i *Repository*, necessari all'orchestrazione ed alla comunicazione con il database rispettivamente.

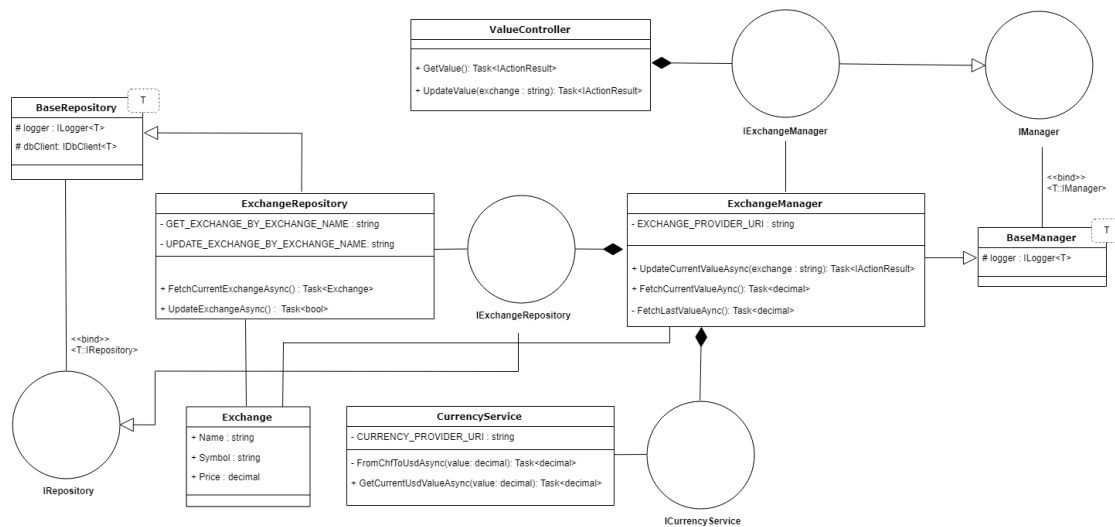


Figura 9: Diagramma delle classi del servizio Token Value

4.5.1 Tecnologie

Per il rispetto del requisito [Vi001] la scelta del linguaggio di programmazione da utilizzare è inevitabilmente ricaduta su *C#*.

Questo requisito è dovuto al fatto che l'intera piattaforma è stata sviluppata con la tecnologia *.NET Framework* di *Microsoft* e poggia sul *Cloud Azure*, anch'esso sviluppato e mantenuto dalla stessa *Microsoft*.

Questa scelta ha reso possibile una perfetta integrazione del servizio con tutte le estensioni (*Insight, Metrics etc..*) e funzionalità aggiuntive messe a disposizione dalla piattaforma.

Lo stesso requisito ha inoltre imposto la creazione del database tramite l'utilizzo di *PostgreSQL*. Questa è stata presa semplicemente per coerenza tecnologica con gli altri servizi che già utilizzavano la stessa tipologia di database.

4.6 Sviluppo

Per comprendere a pieno l'andamento dello sviluppo del servizio è necessario innanzitutto dare una panoramica sia delle metodologie e dei *tools* utilizzati dalla società che del *Flow di sviluppo* imposto a tutti gli sviluppatori.

4.6.1 Metodologia di sviluppo

Data l'elevata preparazione tecnica e professionalità degli sviluppatori presenti nel *team* di *Sgame Pro* e vista la necessità di continui e rapidi miglioramenti dell'applicazione la società ha deciso di utilizzare una metodologia di sviluppo *Agile* e, più in particolare, *Scrum*.

4.6.1.1 Scrum

Il team di Sgame Pro utilizza il framework *Scrum*, ossia un framework Agile, iterativo ed incrementale, per lo sviluppo di progetti complessi.

Scrum si basa su tre pilastri fondamentali:

- **Trasparenza:** I risultati dei processi devono essere compresi dagli Stakeholders, ossia dagli individui direttamente responsabili del risultato del prodotto;
- **Verifica:** Le verifiche devono essere effettuate con bassa frequenza e da persone con competenze tecniche;
- **Adattamento:** Nel caso in cui il risultato di una o più verifiche identifichi un livello di qualità insoddisfacente, il processo di sviluppo deve essere modificato il prima possibile, al fine di riallineare il processo alle aspettative.

In *Scrum* esiste ciò che viene formalmente definito lo *Scrum Team*, il quale è formato da tre figure fondamentali:

- **Product Owner:** responsabile della massimizzazione del valore del prodotto sviluppato dal *Development Team*;
- **Development Team:** formato dalle figure professionali che sviluppano effettivamente il prodotto;
- **Scrum Master:** coordina e supporta il *Development Team*.

Durante il tirocinio il mio ruolo si posizionava all'interno del *Development Team*, il quale veniva coordinato e supportato dallo *Scrum Master* ed i quali *output* venivano giudicati direttamente dal *Product Owner*.

Il rispetto e l'attuazione di questo modello si riflette perfettamente nel processo di sviluppo aziendale, trattato dettagliatamente nella sezione successiva.

4.6.2 Processo di implementazione e rilascio

La società *Sgame SA* ha richiesto che il *flow* di lavoro, durante l'implementazione di nuove *features* di uno stesso servizio debba sempre utilizzare la seguente *pipeline*:

1. **Development;**
2. **UAT;**
3. **Staging;**
4. **Production.**

La struttura sopra esposta segue fedelmente la struttura dei principali *branch* riportata sull'apposito *repository* presente in *GitHub*.

Ad ogni punto sopra elencato corrispondono ambienti di sviluppo differenti, ognuno con uno scopo ben definito.

L'ambiente di *Development* è dedicato esclusivamente all'implementazione di servizi, funzionalità e correzione degli errori (*bug fix*).

UAT, acronimo per *User Acceptance Testing*, è utilizzato per il testing interno delle funzionalità e per l'approvazione interna del *Product Manager*.

L'ambiente di *Staging* è dedicato invece al *Beta Testing*, ossia a test effettuati da personale esterno alla compagnia. Questo risulta estremamente efficace nello sviluppo di applicazioni mobile in quanto l'elevato numero di dispositivi presenti nel mercato richiede accertamenti (soprattutto lato *UI*) sulla corretta visualizzazione e funzionamento delle nuove *features*.

Production è invece l'ambiente "esterno", ossia dove l'applicazione viene utilizzata dagli utenti di tutto il mondo.

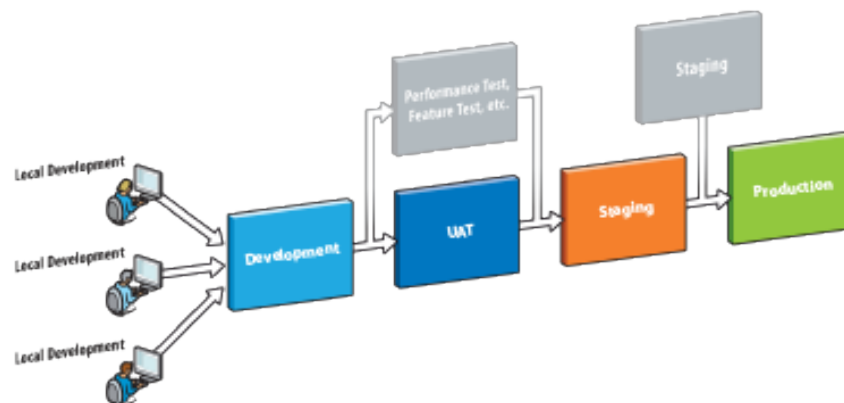


Figura 10: Processo di implementazione e rilascio in Sgame

4.6.2.1 Processo di implementazione delle features

Una volta iniziato il lavoro su una nuova *feature* del servizio è necessario innanzitutto creare un nuovo *branch* il cui nome deve tassativamente seguire la seguente struttura:

iniziali_dello_sviluppatore + nome_feature

Tutti i *commit* riguardanti la stessa *feature* andranno sempre ad utilizzare questo nuovo branch.

Una volta terminata, testata e validata la funzionalità o l'*improvement* è necessario fare lo *stash* di tutti i commit verso un unico commit che avrà (come *commit message*) il nome della *feature* stessa.

Questa andrà quindi inserita sul *branch* di *development* dopo aver effettuato il *rebase*.

Il procedimento è invece differente quando, una volta implementata e testata una nuova funzionalità sull'ambiente di *development*, questa vada inserita sull'ambiente di *UAT*, *Staging* e *Production*.

L'unico modo per modificare il codice presente in uno di questi tre ambienti dedicati è tramite *pull request*, il quale titolo deve essere tassativamente il nome della *feature* che si va ad inserire.

Il rispetto di questo processo si rivela assolutamente necessario per permettere altri membri del team di effettuare *Code review* sul codice che si intende inserire.

Questo permette un controllo efficace sul codice in ingresso nei vari ambienti tramite gli appositi sistemi di *CI/CD*.

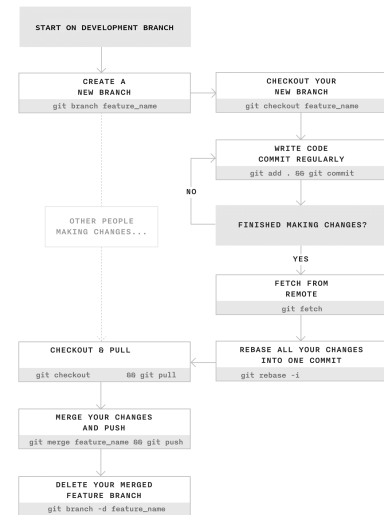


Figura 11: Implementazione di una feature

4.7 Integrazione

Una volta completata l'implementazione del servizio *Token Value* ed i relativi test funzionali e di unità, è stato necessario integrare il servizio con l'infrastruttura già esistente della piattaforma *Sgame Pro*.

L'intero *backend* dell'applicazione utilizza ciò che viene definito *Cloud Computing*, ossia un sistema per la distribuzione di servizi di calcolo (*server*), le risorse di archiviazione (*database*) e reti, erogata tramite internet. Il *Cloud Computing* ha il vantaggio di offrire risorse estremamente flessibili ed un livello di scalabilità estremamente performante.

Il processo di integrazione è stato intrapreso dal *DevOps* del team, seguendo le seguenti fasi:

- creazione dell'istanza;
- configurazione dell'istanza;
- *linking* con i *repository* presenti su *GitHub* per sfruttare la *CI/CD*.

Dei tre passi sopra elencati solo l'ultimo ha riguardato da vicino il mio lavoro.

Infatti, tramite la *CI/CD*, configurata con *Microsoft Azure* e collegata direttamente ai *repository* su *GitHub*, il codice commitato veniva compilato, testato (tramite l'uso di test automatizzati) e pubblicato in forma completamente automatica.

Questo processo, oltre a velocizzare notevolmente lo sviluppo, ha permesso a me ed agli altri membri del team di testare il servizio fin da subito e di raccogliere fin da subito i *feedback* dei *Beta Testers*, oltre a quelli del *Product Owner*.

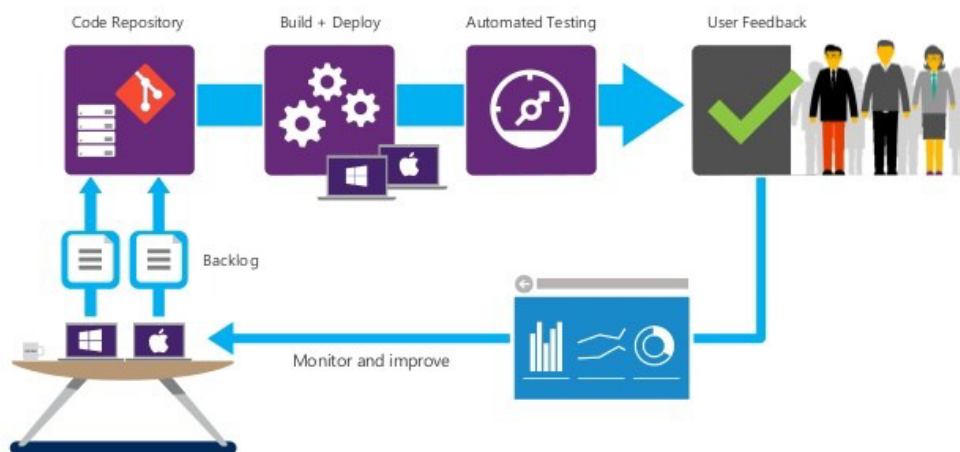


Figura 12: Processo di CI/CD

5 Ethereum Testnet

5.1 Overview

Oltre allo sviluppo del servizio *Token Value* al tirocinante è stato chiesto di testare integralmente le funzionalità dello *Smart Contract* utilizzato dalla piattaforma.

Per farlo è stato necessario creare innanzitutto quella che viene definita una *TestRPC*, ossia una rete locale che utilizza gli stessi protocolli della rete principale di *Ethereum*, ma in cui le transazioni non richiedono l'esborso di denaro, ossia il pagamento di *gas*.

In secondo luogo è stata invece configurata una *Testnet Ropsten*, ossia una rete remota che, oltre ad utilizzare gli stessi protocolli della *Mainnet (Ethereum)*, emula fedelmente le tempistiche ed i costi delle transazioni, ma la criptovaluta necessaria al pagamento delle *gas fee* ha valore zero.

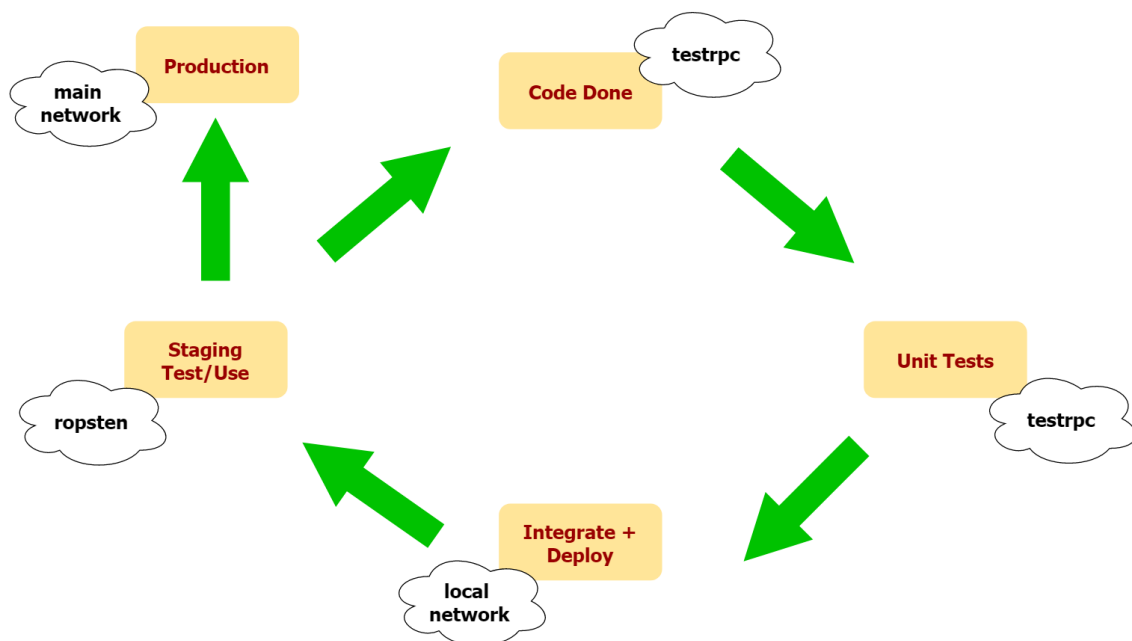


Figura 13: Processo di sviluppo, test e deploy in Ethereum

Grazie all'ausilio di queste reti è quindi possibile testare lo *Smart Contract*, verificandone ogni singola funzione per assicurarsi che il modello di business si rifletta sull'implementazione dello *Smart Contract* stesso.

5.2 Pianificazione

- **Sesta Settimana**(42,5 ore) – Studio del servizio *TX* e creazione dei diagrammi necessari a dimostrarne la comprensione;
- **Settima Settimana**(42,5 ore) – Creazione dell’ambiente di testing su *Microsoft Azure* tramite l’utilizzo del tool *Ganache* ed automazione del deploy dello *Smart Contract* utilizzato dalla piattaforma *Sgame*; Utilizzo della rete pubblica *Ropsten* per l’advanced testing dello *Smart Contract* necessario al funzionamento della piattaforma *Sgame Pro*;
- **Ottava Settimana**(2,5 ore) – Presentazione agli *Stakeholders* del lavoro svolto sul servizio *Token Value*, comprensivo di documentazione e giustificazione delle soluzioni adottate dal tirocinante.
Presentazione agli Stakeholders del lavoro svolto sulle *Testnet* ed esposizione dei risultati dei test.

5.3 Analisi dei requisiti

Viene di seguito riportata la specifica dei requisiti richiesti per lo sviluppo e l'integrazione della rete di test per lo *Smart Contract* sviluppato per la piattaforma *Sgame Pro*.

5.3.1 Specifica dei requisiti

Obbligatori:

- **Ob004** – l'ambiente testnet di tipo *private* deve utilizzare uno script di *auto-deploy* per lo *Smart Contract*.

Vincolo:

- **Vi002** – l'ambiente *testnet* di tipo *private* deve utilizzare il tool *Ganache*.
- **Vi003** – l'ambiente *testnet* di tipo *public* deve utilizzare la rete *Ropsten*.

5.4 TX Service

5.4.1 Overview

Prima di poter configurare la *Testnet* è stato necessario studiare il codice del servizio che la società ha chiamato *TX*, necessario alla comunicazione con la rete *Ethereum* e quindi con lo *Smart Contract*.

Oltre alla comunicazione diretta con la *blockchain* ed alla creazione ed invio delle transazioni su *Ethereum*, *TX* utilizza al suo interno un modulo di *Anti-Fraud* basato sull'attribuzione dei punteggi ed un servizio ausiliario di *Back Office* chiamato *TX Admin*.

TX Admin, grazie alla sua interfaccia grafica, permette il controllo e la verifica delle richieste di transazioni *IN-OUT* da parte di un operatore.

Di seguito le funzionalità principali del servizio:

- esecuzione delle transazioni verso indirizzi *Ethereum* tramite l'utilizzo dello *Smart Contract* creato per *Sgame Pro*;
- stime sui consumi di *gas*;
- verificare la validit' degli indirizzi di destinazione forniti;
- effettuare *callback* al *backend* della piattaforma per comunicare i cambiamenti di stato delle transazioni.

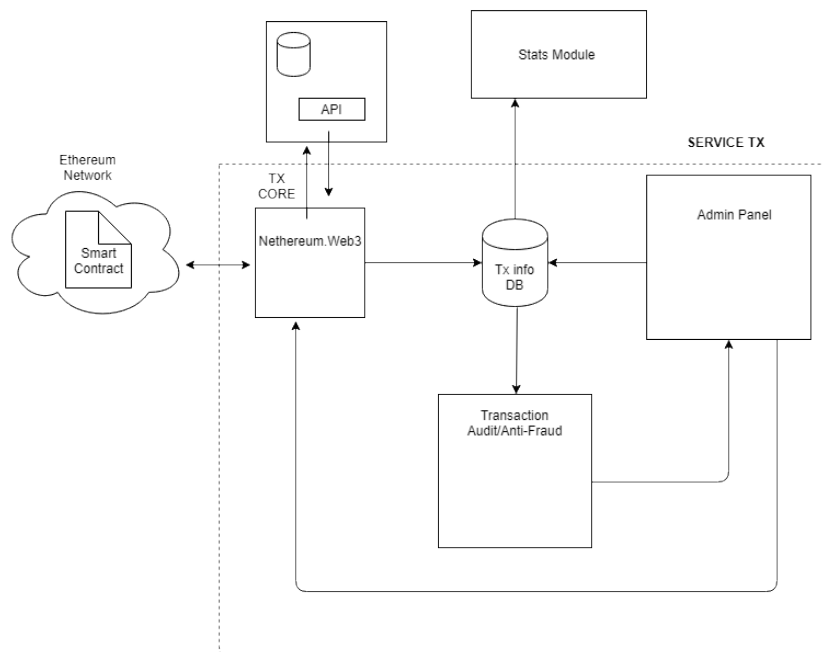


Figura 14: Architettura semplificata del servizio TX

5.4.2 Specifiche

La struttura di *TX* ha rappresentato la traccia per quella che poi è diventata la struttura di *Token Value*, e per questo ne condivide la separazione dei moduli in interfaccia e logica di business.

Anche in questo caso il servizio è composto da più moduli separati, ognuno con un compito specifico ed esplicitato in [Figura 15].

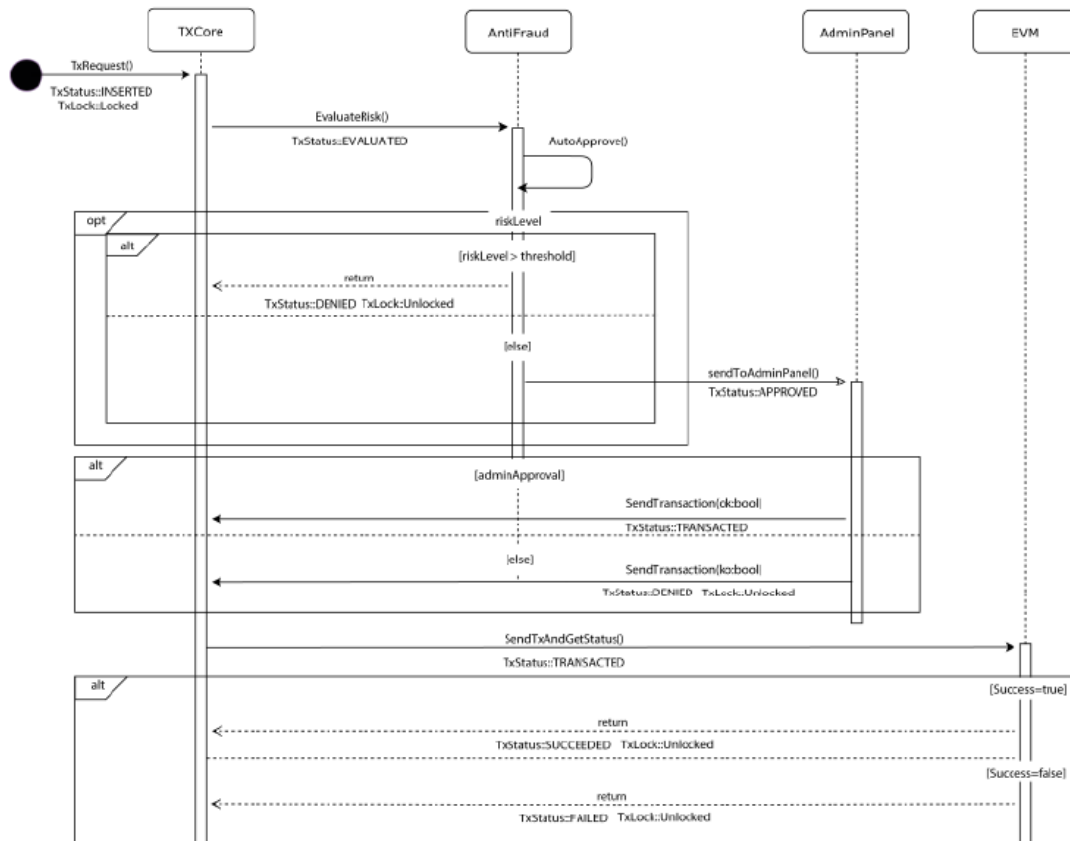


Figura 15: Diagramma di sequenza del servizio TX

Ogni transazione passa quindi prima da un controllo *Anti-Fraud*, per poi essere inviata al servizio *TX Admin* o direttamente su *Ethereum*.

Risulta quindi ora più chiaro come lo sviluppo della *Tesnet Ethereum* è stato obbligatoriamente preceduto dallo studio della comunicazione tra il servizio *TX* e la *blockchain Ethereum*.

5.5 Private Tesnet

Una *private testnet*, come già spiegato in precedenza, è una rete locale che utilizza gli stessi protocolli della rete principale *Ethereum*, ma in cui le transazioni vengono *minate* (ossia validate ed inserite in un nuovo blocco) istantaneamente e non richiedono il pagamento di *gas*.

Esistono più strumenti in grado di simulare una *blockchain*, ma, per il rispetto del requisito di vincolo **Vi002**, la scelta è caduta inevitabilmente su *testrpc*.

testrpc è un client *Ethereum*, basato su *Node.js*, utilizzato per il test e lo sviluppo di *dApps*. Esso utilizza *ethereumjs*, una implementazione della *EVM* in *Javascript*, per simulare il comportamento della rete e rendere lo sviluppo di applicazioni basata su *Ethereum* più semplice e veloce.

5.5.1 Strumenti utilizzati

Ganache è l'ultima versione disponibile sul mercato di *testrpc* e fa parte della *Truffle Suite*, una collezione di strumenti (*development environments, testing framework etc..*) estremamente utili per lo sviluppo di applicazioni decentralizzate.

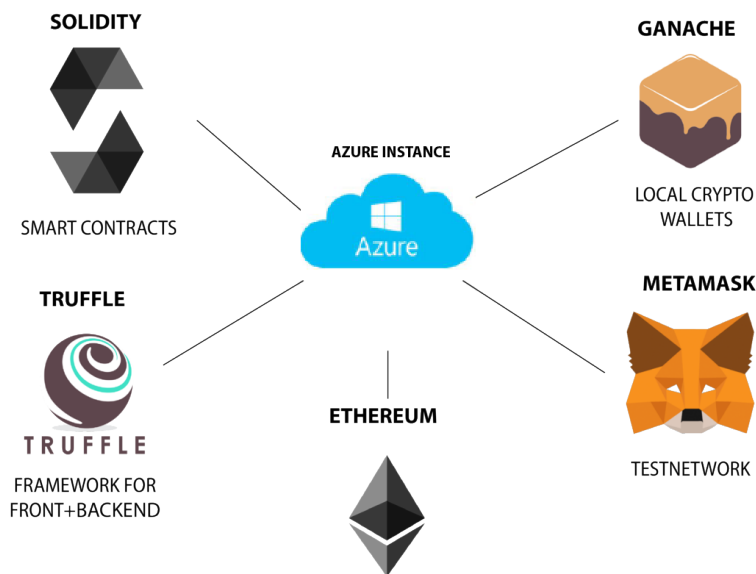


Figura 16: Tools di sviluppo utilizzati dalle private networks

Il *client* include tutte le funzioni *RPC* più comuni (come gli eventi) e può essere eseguito in modo deterministico. Ciò previene il verificarsi di comportamenti inattesi durante le transazioni e quindi un maggior controllo sulle stesse.

L'ultimo strumento necessario alla creazione della testnet prende il nome di *METAMA-SK*, un "ponte" che permette di utilizzare direttamente dal proprio *browser* un *Ethereum full node* virtuale.

5.5.2 Creazione

La creazione di una *testnet* di tipo *private* è diventato un processo relativamente semplice negli ultimi due anni, grazie soprattutto all'elevato numero di *tools* sviluppati (*vedi sezione precedente*) ed al continuo miglioramento e semplificazione delle funzionalità associate.

Le difficoltà riscontrate sono quindi ricadute principalmente sulla comprensione del funzionamento ed delle interdipendenze che intercorrono tra i vari strumenti.

Poichè la rete di test si deve trovare necessariamente in locale ed i servizi atti alla comunicazione con la *blockchain* si trovano in *cloud* è stato necessario in prima battuta creare un'apposita istanza su *Azure* dove "far girare" il server di *testrpc*.

Una volta terminato il procedimento è stata necessaria l'installazione delle dipendenze ed infine l'installazione degli strumenti stessi.

```
$ ganache-cli -p 7545 -u 0
Ganache CLI v6.0.3 (ganache-core: 2.0.2)

Available Accounts
=====
(0) 0x05a3cb62049ab7b5d4954361b05fc4a42001aebf
(1) 0x501e8ba67ed6986316c14a21337d42a4ff02b623
(2) 0x0e30a106903e8365970e91e7439e1093d1358bdd
(3) 0x828f543d890ceb1e3523cd6759f0c3850a5c120b
(4) 0x71c23cd768f75472f91a4a96c8db3ea1a6b601d0
(5) 0x3088d28fc25d9fe472070aed77da9616fe795983
(6) 0x19987997d6b67408c5e935cc4b6d57b8360c310f
(7) 0x682588814cb2640d0f92f48d726a857a5d94e87c
(8) 0x1fef3586ed684f7a9476eb898c8970bf9a242400
(9) 0x2ef2d423470567b669ff7e1ecaacaa26e3af438b
```

Figura 17: Creazione di una private network con Ganache-CLI

A questo punto, dopo essersi collegati all'istanza tramite *ssh* è semplicemente necessario avviare il server con il comando ***ganache-cli***, specificando la porta in ascolto (***-p 7545***) e l'account che si desidera sbloccare (***-u 0***).

In questo caso specifico il server starà in ascolto sulla porta *7545* e si potrà utilizzare l'account *0* (*0x05a...ebf*) per poter interagire con lo smart contract.

Durante le fasi iniziali di test è stato necessario eseguire più *deploy Smart Contract* e, per evitare che i server esegua il *reset* degli account a disposizione, è stata aggiunta l'opzione ***-d*** (deterministic).

5.5.3 Integrazione

Per l'integrazione della testnet con il servizio *TX* è stato necessario utilizzare una libreria *wrapper*, chiamata *Nethereum*, per scrivere il codice *C#* al fine di usare un'altra libreria (considerata uno *standard* nella creazione di *dApps*) che prende il nome di *web3.js*. *web3.js* è, più propriamente, una collezione di librerie che consente di inviare *Ether* da un account ad un altro, oltre a leggere e scrivere dati verso *Smart Contract*.

Grazie all'utilizzo di questa libreria è stato necessario seguire quattro semplici passi:

- prendersi nota dell'indirizzo (*public key*) e della chiave privata (*private key*) dell'account sbloccato con l'opzione *-u* precedentemente passata al tool *ganache-cli* e creare un apposito account.

In questo modo si stà esplicitando l'indirizzo dell'account (in gergo *owner*) autorizzato all'interazione con lo *Smart Contract* e quindi alla firma delle transazioni tramite *private key*.

```
// DEFINING ACCOUNT

/* a private key is needed to sign the transactions.
 * The ethereum address is calculated from this private key
 * so each transaction signed with this key can be related to our ethereum
 * address
 */
var privateKey =
    "dc925af9892df9f5f6f00dcbee5f6c59d53b631b32b475dfec49fabed8f57510";

/* Who can do transactions, signing them with his private key */
var senderAddress = "0x67af396FB49b054ad946a9E2cd67123750Fa1207";

/* now it is possible to create an instance of Account which
 * will be used to sign the transactions
 */
var account = new Nethereum.Web3.Accounts.Account(privateKey);

/* An instance of Web3 must be created to interact to the Ethereum client
 * via RPC (remote procedure call).
 * [constructor public Web3(IAccount account, IClient client)]
 * [client use ganache default port 8545]
 */
var web3 = new Web3(account, "HTTP://127.0.0.1:7545");
```

- passare al servizio *Tx* i dati sopra elencati, corredati dall'indirizzo dell'istanza creata e specificando la porta inserita come parametro sul tool *ganache-cli* (*-p*).

- utilizzare dell'indirizzo locale, il *bytecode* e l'*ABI* dello *Smart Contract* appena *deployato*;

```
// DEFINING CONTRACT

/* contract bytecode (for EVM) */
var byteCode = "608...029";

/* ABI (contract interface definition) */
var abi = @["{"constant":true,"inputs": ..... ,"type":"event"}"];
```

- effettuare il *deploy* dello *Smart Contract* di *Sgame Pro*;

```
// DEPLOYING CONTRACT

/* creating _totalSupply to initialize SGM contract using BigInteger
 * to pass the amount in wei (1 ETH = 10^18 wei)
 */
System.Numerics.BigInteger totalSupply =
    System.Numerics.BigInteger.Parse("5000000000000000000");

/* Now we deploy the smart contract using SGM's abi and bytecode.
 * The method send the initial transaction (SendRequest),
 * aka call the SGM constructor, waits for the trasaction to be mined
 * (AndAwait)
 * and finally returns the transaction receipt.
 * The transaction receipt of a newly deployed contract includes
 * the contract's address.
 * This address can be now used to interact with the contract itself.
 */
var receipt = await web3.Eth.DeployContract.SendRequestAndWaitForReceiptAsync(
    abi,                                     //contract's interface
    definition,                             //contract's bytecode
    byteCode,                               //contract's bytecode
    senderAddress,                          //deployer
    new Nethereum.Hex.HexTypes.HexBigInteger(900000), //gas
    null,
    totalSupply);                           //SGM constructor parameter
```

Alla fine della procedura il servizio *TX* ha potuto comunicare con successo con lo *Smart Contract* di *Sgame pro*.

Questo ha permesso agli sviluppatori del servizio di testare in prima persona tutte le funzionalità dello *Smart Contract* e di creare una *test suit ad-hoc* per accertarsi che la logica del servizio rispettasse la logica di business.

5.6 Public Tesnet

A differenza di una *testnet* privata come *testrpc*, una *testnet* pubblica ha la peculiarità di essere in tutto e per tutto una *blockchain*, con l'unica differenza che la criptovaluta utilizzata al suo interno non ha nessun valore.

Per il rispetto del requisito **Vi003** la scelta è ricaduta obbligatoriamente sulla rete *Ethereum Ropsten*.

Ropsten, come le altre *testnet* pubbliche, è utilizzata dagli sviluppatori di *Ethereum* di tutto il mondo per testare gli *Smart Contract* e tutti i servizi di interazione con la *blockchain Ethereum*.

Per poterla utilizzare è innanzitutto necessario procurarsi degli *rETH*, che altro non sono che la criptovaluta utilizzata per il pagamento delle transazioni all'interno della rete *Ropsten* stessa.

Ricevere *rETH* è un procedimento estremamente semplice. Basta infatti utilizzare quelli che vengono definiti *faucet* (rubinetti), ossia delle pagine web dove, una volta inserito l'indirizzo del proprio *wallet Ropsten*, è possibile ricevere gratuitamente uno o più *rETH*.



Figura 18: Ropsten Logo

Testare in una rete pubblica è estremamente differente che testare in una rete privata, anche se il risultato è lo stesso.

Essendo una rete pubblica un agglomerato di nodi sparsi in tutto il mondo e connessi via internet, per accedervi sono applicabili due strade:

- diventare un *full node*, ossia scaricare lo storico di tutte le transazioni avvenute nella rete *Ropsten*, partendo dal *Genesis Block* (80Gb);
- utilizzare un client che attui da interfaccia alla *blockchain* e che emuli un *full node*.

Ovviamente scelta è ricaduta sulla seconda opzione ed il tool utilizzato è stato *METAMASK*.

L'enorme comodità di *METAMASK* è che può essere utilizzato direttamente da *browser* (esiste un famoso *add on* per *Google Chrome*) ed è estremamente facile da utilizzare.

Prima di poter utilizzare la rete *Ropsten* con il servizio *TX* è stato necessario effettuare un nuovo *Deploy* dello *Smart Contract* sulla rete stessa.

Il procedimento è abbastanza semplice una volta capite le logiche di riascio, ma è richiesta comunque molta attenzione durante la procedura poichè, una volta pubblicato sulla rete, lo *Smart Contract* diventa pubblico e non è più possibile rimuoverlo (ovviamente).

5.6.1 Integrazione