Università degli Studi di Modena e Reggio Emilia

Dipartimento di Ingegneria “Enzo Ferrari”
Corso di Laurea in Ingegneria Informatica

Blockchain e Privacy: sistema di consenso al trattamento dati

Relatore:
Prof.ssa Sonia Bergamaschi

Candidato:
Michele Di Stefano

Anno Accademico 2020-2021

Keywords:

Bitcoin
Blockchain
Smart Contract
Consesus
Hyperledger
Distributed Ledger Technology
Chaincode

1 Introduzione

Nel seguente elaborato sarà illustrata la modellazione di un sistema di gestione del consenso del trattamento dei dati delle persone, con il fine di mostrare come sia possibile utilizzare la tecnologia blockchain per risolvere problemi reali e quali vantaggi ne derivino da essa.

Lo scopo del sistema è quello di mediare lo scambio di informazioni personali, contenute in documenti, tra un utente e una clinica tramite blockchain, rendendone il più sicuro e trasparente possibile il trasferimento che, se gestito da una normale terza parte, presenterebbe molti problemi di privacy e sicurezza.

Per realizzarlo, sarà utilizzato il framework blockchain Hyperledger Fabric, un progetto open source della Linux Foundation che si pone l'obiettivo di aumentare l'adozione della tecnologia blockchain nel settore aziendale e industriale.

La blockchain, un registro distribuito e immutabile, tiene traccia di tutte le transazioni che avvengono al suo interno senza il bisogno di un'autorità centrale.

Il tipico compito di questa autorità di verificare le transazioni viene invece distribuito a tutti i partecipanti della rete, rendendola completamente decentralizzata.

Questa decentralizzazione denota l'elemento più innovativo del sistema, in quanto ne rende la sua manomissione pressoché impossibile.

Inoltre, sarà anche trattato Bitcoin, il quale ha introdotto questa tecnologia rivoluzionaria, creando un sistema di pagamento internazionale sopra di essa.

2 Bitcoin: Una Moneta Virtuale

2.1 Introduzione

Figura 1 - Logo Bitcoin

Il concetto di Blockchain nasce nel 2008 quando Satoshi Nakamoto (un “alias” dietro il quale c’è una persona o un gruppo di persone) pubblica il whitepaper "Bitcoin: A Peer-to-Peer Electronic Cash System"[1]; un manifesto in cui Satoshi ha spiegato i dettagli di implementazione di una valuta digitale che può essere trasferita su una rete peer-to-peer senza la necessità di una banca che la controlli.

Nakamoto all'interno del whitepaper espone una rete peer-to-peer, in cui ogni nodo condivide un registro distribuito contenente tutte le transazioni eseguite. Ogni nodo della rete contiene una chiave pubblica, ovvero l'indirizzo pubblico, e una chiave privata utilizzata per firmare le transazioni.

Nel 2009 rilascia il software open-source e Bitcoin diventa realtà.

È possibile per qualsiasi individuo entrare a far parte della rete Bitcoin installando un client sul proprio computer o in alternativa registrandosi su determinati siti che gestiscono questi clients per i propri utenti.

L'asset scambiato sulla rete è il BTC, una criptovaluta trasferita tra gli utenti della rete.

2.2 Come si usa

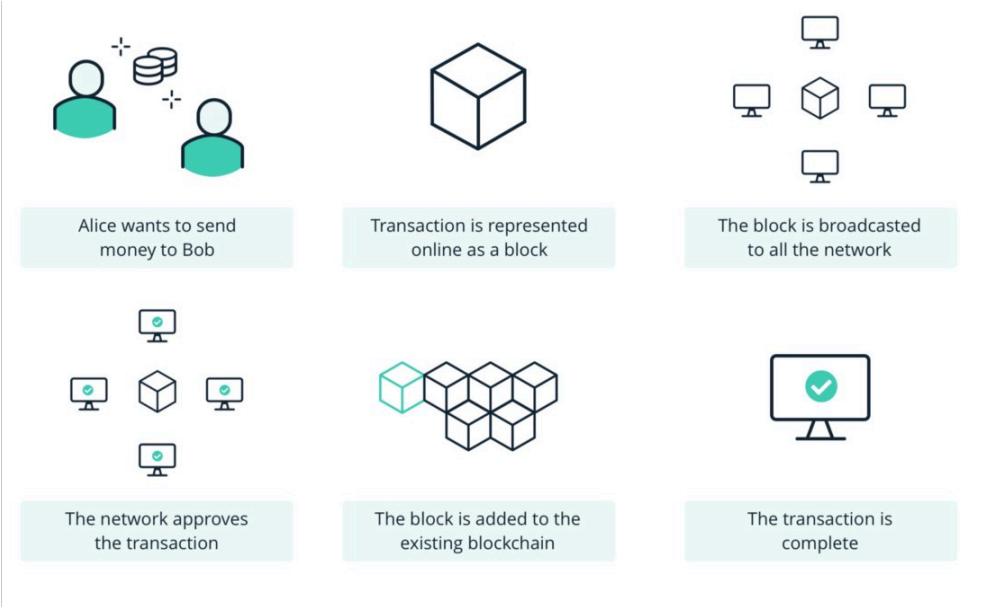


Figura 2 - Processi di transazione tra due utenti

Proprio come con i contanti, quando si ottengono unità di Bitcoin, si possono acquistare servizi o merci in luoghi diversi o su internet attraverso il proprio portafoglio, senza il bisogno di un intermediario.

Per fare ciò si deve installare un "portafoglio bitcoin"[2], un'applicazione che si può trovare su pc o cellulare. Una volta installata e creato il portafoglio, una frase mnemonica (una serie di 12 parole) sarà generata. Questa sarà la "seed phrase" (frase seme), infatti come si può intuire dal nome sarà utilizzata per generare le chiavi crittografiche per inviare e ricevere i bitcoin.

Questa frase è unica per ogni portafoglio ed è la parte che bisogna tenere più al sicuro perché, se una persona ne venisse a conoscenza, potrebbe di conseguenza importare il portafoglio su qualsiasi dispositivo e utilizzare i bitcoin al suo interno. Una volta generata e conservata la "seed phrase" in un posto sicuro, il processo di creazione del portafoglio sarà terminato e si potrà cominciare a inviare e ricevere bitcoin.

Per ricevere fondi si dovrà condividere uno degli indirizzi pubblici disponibili, facilmente accessibili dall'applicazione, che saranno utilizzati dagli altri utenti per inviare il pagamento. Allo stesso modo, per inviare pagamenti ad altre persone, l'utente dovrà conoscere i loro indirizzi.

Immaginiamo l'invio di bitcoin tra due utenti, Alice e Bob. Bob comunica il proprio indirizzo ad Alice che lo utilizza per inviare 1 bitcoin. Una volta inviato il pagamento, la trasmissione sarà comunicata via broadcast ai nodi della rete che, una volta approvata, sarà memorizzata nel prossimo blocco della blockchain.

Di solito dopo sei conferme, cioè la creazione di ulteriori sei blocchi dopo quello in cui la transazione è inclusa, il pagamento si può dire valido. Si aspettano sei conferme per offrire una sicurezza maggiore, siccome la probabilità che un malintenzionato alteri la transazione cala drasticamente per ogni blocco aggiunto.

2.3 Come funziona

La valuta è stata progettata su base crittografica, il che le consente di essere completamente decentralizzata.

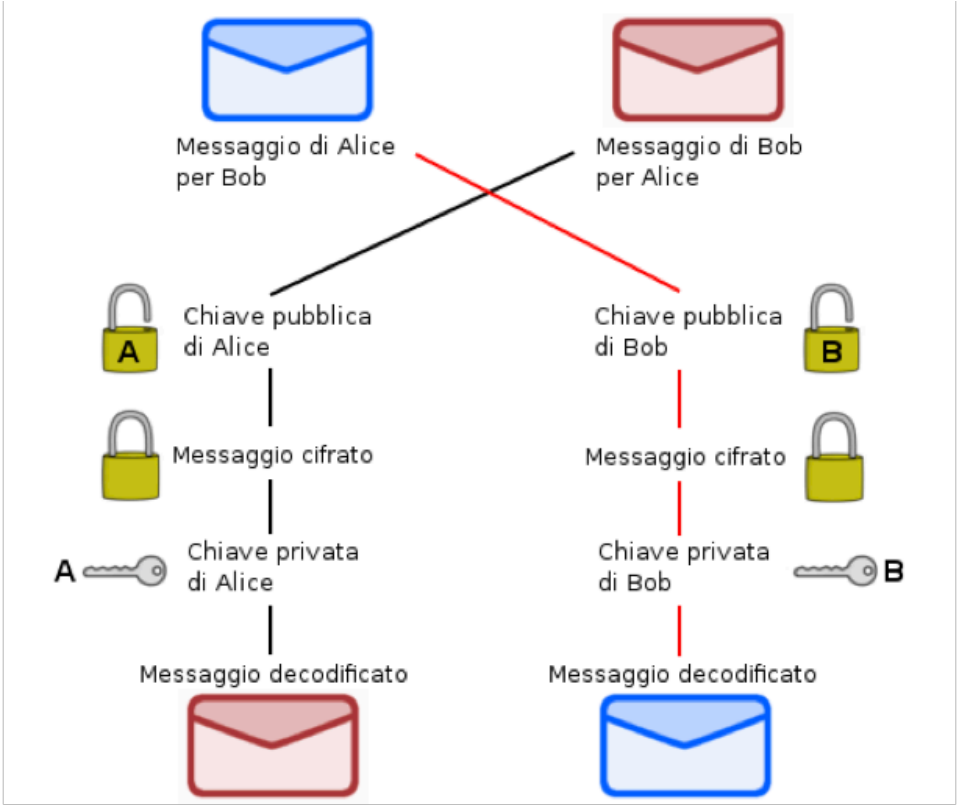
Sono impiegati vari tipi di crittografie, tra cui lo SHA256[3] per la creazione dei blocchi (processo di mining) e di indirizzi pubblici, e crittografia a chiave pubblica per lo scambio della valuta.

2.3.1 Scambio di bitcoin: Public Key Infrastructure (PKI)

L'indirizzo bitcoin che viene scambiato per inviare e ricevere pagamenti è la chiave pubblica del sistema d*i crittografia asimmetrica*[4].

In questo tipo di crittografia ogni utente possiede una coppia di chiavi:

- La *chiave pubblica*, usata per codificare le informazioni inviate al destinatario.
- La *chiave privata*, usata dal destinatario per decodificare le informazioni ricevute.



<

Figura 3 - Scambio di messaggi con crittografia a chiave pubblica

Per esempio, se Alice vuole inviare un messaggio a Bob, Bob manda a Alice la sua chiave pubblica. Alice utilizzerà la chiave per cifrare il messaggio per poi inviarlo a Bob. Bob utilizzerà la propria chiave privata per decifrare e leggere il contenuto.

Questa è la crittografia con cui ogni utente che utilizza bitcoin interagisce, a volte anche inconsapevolmente. Si può infatti notare una sottile somiglianza allo scambio di bitcoin precedente tra Alice e Bob.

Nel caso di bitcoin l'indirizzo pubblico sarà derivato, tramite la funzione SHA256, dalla chiave pubblica dell'utente, e la chiave privata sarà usata per utilizzare i bitcoin inviati all'indirizzo corrispondente.

Le due chiavi saranno derivate dalla frase seed.

La chiave privata, anche se di solito un utente non la utilizza direttamente, è molto importante che non sia condivisa con nessuno, dato che chiunque ne venga in possesso può spenderne i bitcoin all'interno.

2.3.2 Processo di mining

Il *processo di mining*[5] è un sistema di consenso distribuito, che serve per la creazione di nuovi blocchi dove saranno validate le transazioni appena avvenute.

Il consenso è la base della blockchain, infatti la rete si deve mettere d'accordo su quali transazioni includere nel blocco successivo, mantenendo intatta la sicurezza e l'integrità della rete.

2.3.3 Proof of Work

Il meccanismo di consenso utilizzato dalla rete Bitcoin è la *Proof of Work*[6].

I miners, ossia chi si prende in carico il lavoro di creare nuovi blocchi, competono per risolvere un problema matematico estremamente complicato che, una volta risolto, permetterà al miner vincitore di creare un nuovo blocco e lo ricompenserà con una somma in bitcoin.

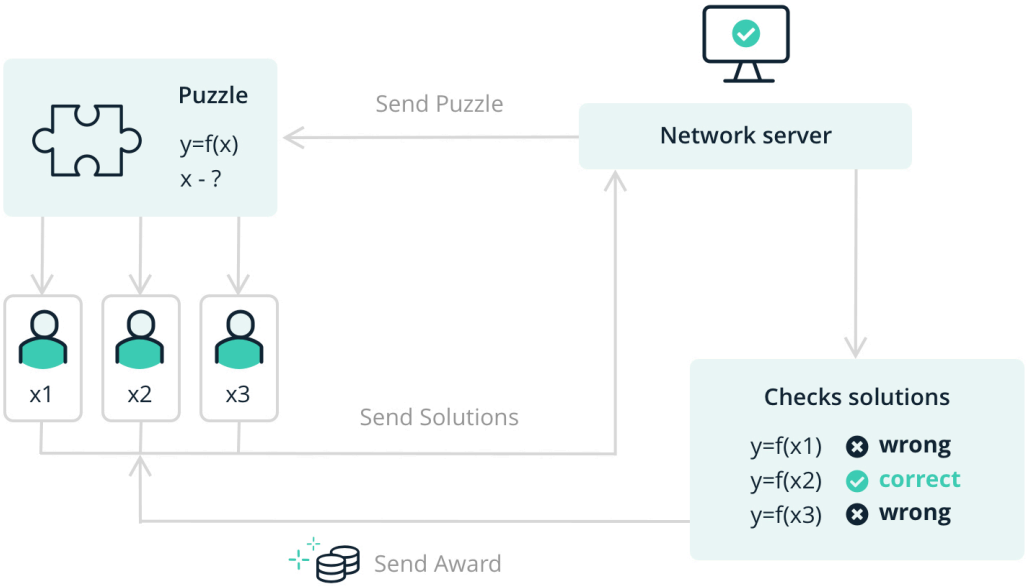


Figura 4 - Rappresentazione schematica della Proof Of Work

Questo è anche il processo col quale nuovi bitcoin vengono creati.

L'offerta massima di bitcoin è di 21 milioni; di conseguenza le ricompense per i miners vengono dimezzate ogni 4 anni (evento definito come "halving"). L'ultimo bitcoin sarà "minato" nel 2140, anno dopo il quale i miners non riceveranno più ricompense.

Il problema matematico consiste nel calcolo del codice hash dell'intestazione di un blocco. L'hash[7] è una funzione crittografica non invertibile che prende in ingresso una stringa binaria di dimensione fissa detta valore di hash e un nonce[8] (number used once) che sarà incrementato ogni volta che la funzione di hash sarà chiamata. Nel caso di bitcoin l'algoritmo crittografico utilizzato è lo SHA-256.

La funzione di hash quindi sarà eseguita fino a che non genererà un valore che inizia con un certo numero di zeri. Il numero di zero iniziali richiesti è identificato dalla difficoltà che varia, con l'obiettivo di generare come media un blocco ogni 10 minuti. Se i blocchi sono generati troppo velocemente la difficoltà aumenterà e viceversa.

Quando un miner trova il corretto nonce, gli altri miners potranno facilmente verificare se la soluzione è esatta e, quando il 50% + 1 della rete ne ha verificato la validità, il blocco sarà reso valido e il miner riceverà la ricompensa in bitcoin.

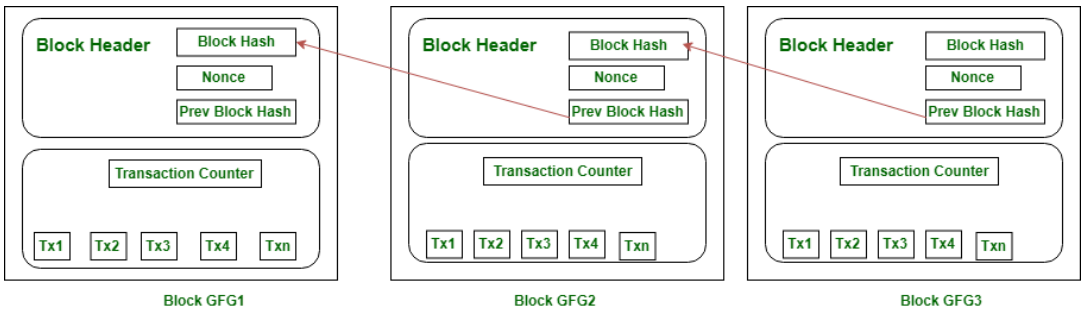


Figura 5 - Contenuto dei blocchi creati

Il blocco creato conterrà oltre alle informazioni, il codice hash del blocco, il “nonce” usato per risolvere il problema e il codice hash del blocco precedente.

Questo processo impedisce ad un malintenzionato di creare un blocco con una transazione illecita, dato che il risultato della sua funzione hash sarà diverso da quello di tutti gli altri nodi della rete. Per compromettere la rete si dovrebbe riuscire a prendere il controllo del 50% +1 dei nodi presenti.

2.4 Trasparenza

A differenza del modello bancario che garantisce un livello di privacy limitando l'accesso alle informazioni soltanto alle parti coinvolte, Bitcoin è completamente trasparente. Chiunque può navigare la blockchain e vedere tutte le transazioni avvenute dalla creazione nel 2009, divise per blocchi.

La privacy viene mantenuta dato che ogni indirizzo è completamente anonimo. In questo modo il pubblico può vedere che qualcuno sta mandando un certo importo a qualcun altro, ma non può sapere l'identità delle parti coinvolte. Una protezione aggiuntiva sarebbe quella di utilizzare una coppia di chiavi differenti per ogni transazione; in questo modo risalire al proprietario sarebbe ancora più difficile. La trasparenza del sistema inoltre implica che chiunque può verificare l'integrità della rete tracciando i bitcoin tra vari periodi.

3 Blockchain

Figura 6 - Network Blockchain

3.1 Che cos'è la Blockchain

La Blockchain[9], anche definita come Distributed Ledger Technology (Tecnologia del Libro Mastro Distribuito) è un elenco di record, chiamati blocchi, che sono collegati. Ogni blocco contiene un hash crittografico del blocco precedente, un timestamp e i dati di transazione.

Funziona su una rete peer to peer e ogni nodo della rete conserva una copia del registro distribuito. Pertanto, sfruttando l'architettura di rete peer to peer e la scienza crittografica, si ottiene uno storage di dati *distribuito*, *immutabile*, *sicuro* e continuamente *sincronizzato* tra i nodi della rete.

La parte fondamentale della Blockchain è il *meccanismo del consenso*, è l'algoritmo che gestisce il processo di consenso incaricato di convalidare le transazioni e aggiungerle alla catena di blocchi, eseguendo l'aggiornamento del ledger.

Oltre al contesto della rete Bitcoin, alla quale molti la associano, questa tecnologia ha infinite implementazioni e potrebbe rivoluzionare molti settori dato che risolve il problema della fiducia e della trasparenza tra le parti coinvolte.

Implementazioni possibili potrebbero essere il tracciamento della filiera alimentare di un prodotto per garantire che sia sano e proveniente dal luogo dichiarato, garantire il corretto scambio di beni (automobili, case, azioni ecc.) e dei dati personali.

In breve, questa tecnologia ha il potenziale di cambiare la società e il commercio in moltissime aree se implementata correttamente.

3.2 Meccanismi di consenso

Per mantenere l'integrità dei dati e la sincronizzazione tra i diversi nodi, la Blockchain ha bisogno di un algoritmo di gestione del consenso.

Gli algoritmi più conosciuti[11] sono:

- **Kafka:** Usato in Hyperledger Fabric. Sono presenti dei servizi di ordinamento chiamati orderers che distribuiscono ai nodi i blocchi ordinati.

- **Byzantine fault tolerance - BFT:** Raggiunge un accordo o consenso sui blocchi anche quando alcuni nodi non rispondono o emettono valori dannosi per fuorviare la rete. Il consenso si raggiunge se più di 2/3 dei nodi approvano il nuovo blocco.

- **Proof of Work - PoW:** Attualmente usato da Bitcoin, Ethereum e molte altre criptovalute, PoW risolve il problema del consenso in quanto raggiunge un accordo di maggioranza senza alcuna autorità centrale, nonostante la presenza di parti sconosciute/potenzialmente inaffidabili e nonostante la rete non sia istantanea, attraverso i miners che risolvono il problema crittografico spiegato precedentemente. Il potere decisionale è sparso tra i miner uniformemente ed è direttamente proporzionale all'hashrate (potenza di calcolo) che il miner possiede. Usa un alto quantitativo di energia per alimentare le schede grafiche che risolvono i problemi crittografici.

- **Proof of Stake - PoS:** Prossimamente sarà usato da Ethereum con il nuovo aggiornamento. Richiede la presenza di una criptovaluta o un token. Gli utenti depositano (stake) l'ammontare di criptovaluta che desiderano, diventando così validatori (validators). I validatori vengono quindi selezionati casualmente per "minare" o convalidare il blocco e premiati con la criptovaluta della rete. La probabilità che si sia selezionati cresce proporzionalmente all'ammontare di criptovaluta depositata (in staking). Per gli utenti diventa una sorta di entrata passiva che varia in base alla percentuale di criptovaluta in staking rispetto al totale. Questo sistema randomizza il convalidatore piuttosto che utilizzare un meccanismo basato sulla concorrenza come il proof-of-work. Questo algoritmo è al centro di molte polemiche perché risolve il problema energetico, ma distribuisce il potere decisionale meno uniformemente rispetto a PoW, in quanto alcune persone, di solito le whales, cioè coloro che dispongono di un enorme quantitativo di criptovaluta, hanno più influenza sulla rete.

3.3 Tipi di Blockchain

Negli ultimi anni, molte aziende si sono interessate alla blockchain. In molti casi, però, i dati che esse condividono sono informazioni sensibili che non vogliono mantenere pubbliche. Queste esigenze delle aziende hanno portato recentemente alle prime soluzioni blockchain *Permissioned*, mantenute da un sistema di nodi consortile.

Il mondo blockchain al giorno d'oggi è diviso tra blockchain *Permissionless* e *Permissioned*[13]:

- **Permissionless Blockchain**: è una rete completamente decentralizzata in cui chiunque può leggere o inviare transazioni, nonché partecipare al processo di consenso come Bitcoin, Ethereum e altre cripto.

- **Permissioned Blockchain**: la soluzione *Permissioned* è una blockchain in cui il processo di consenso è controllato da un insieme preselezionato di nodi, che potrebbe essere definita come una rete "parzialmente decentralizzata". Per capire come i nodi siano coinvolti nel processo di consenso, si potrebbe immaginare un consorzio di 15 istituzioni finanziarie, dove ognuna gestisce un nodo e, tra le quali, solo 10 preselezionate possono partecipare al processo di validazione.

Spesso nelle blockchain *Permissioned* c'è anche l'interesse all'implementazione di un meccanismo di appartenenza, al fine di gestire l'accesso in lettura e scrittura sulla rete. Per questo motivo, c'è un'altra distinzione tra Blockchain pubbliche e private:

- **Blockchain Pubbliche:** sono reti peer to peer in cui chiunque ha diritto di lettura e scrittura sulla rete e può partecipare al processo di consenso. In una blockchain pubblica ogni nodo è potenzialmente inaffidabile, quindi il meccanismo del consenso viene sviluppato in modo da prevenire ogni nodo dannoso che potrebbe compromettere i dati e le transazioni eseguite sulla rete. L'intera architettura e il consenso sono distribuiti in modo da ridurre al minimo la responsabilità della manipolazione dei dati. Le risorse mirate dipendono dall'algoritmo di consenso utilizzato. Ad esempio, la prova di lavoro (PoW) coinvolge risorse computazionali nel meccanismo del consenso, mentre la prova di partecipazione (PoS) implica la quantità di token del nodo coinvolto nel consenso. Queste blockchain sono generalmente considerate "completamente decentralizzate".

- **Blockchain Private:** una blockchain privata è una blockchain consortile in cui i permessi di scrittura e lettura sono riservati a certi utenti. L'idea di base della blockchain privata è che la rete sia composta da un insieme di nodi fidati o semi-fidati, che compongono la governance della rete, quindi una blockchain *Permissioned*. Il meccanismo del consenso non è così complesso come quello delle blockchain pubbliche, perché l'ipotesi di partenza è diversa e solitamente necessita di buone prestazioni e bassa latenza delle transazioni.

I vari tipi di blockchain possono essere riassunti in breve nella tabella successiva:

	Pubblica (Tutti hanno permessi di lettura e scrittura)	**Privata** (Permessi di lettura e scrittura riservati a certi utenti)
Permissionless (Qualsiasi nodo può validare le transazioni)	Rete **pubblica e decentralizzata** Chiunque può essere validatore e interagire con la rete **Esempio**: Bitcoin e Ethereum	Rete **privata e decentralizzata** Chiunque può essere validatore ma solo certi utenti possono interagire con la rete **Esempio**: Istanze private di reti pubbliche
Permissioned (Solo certi nodi possono validare le transazioni)	Rete **pubblica e centralizzate** Il ruolo di validatore è riservato a certi utenti ma chiunque può interagire con la rete **Esempio**: Una rete Permissioned di aziende finanziare che vogliono tenere i dati pubblici per gli shareholders	Rete **privata e centralizzata** Il ruolo di validatore e l'interazione con la rete sono riservati a solo utenti autorizzati **Esempio:** Reti di debugging o reti private di aziende che non vogliono condividere i dati

3.4 Smart Contracts

Il concetto di *smart contract*[14] è stato introdotto dall'informatico Nick Szabo.

Si tratta di un programma per computer o di un protocollo di transazione destinato a eseguire, controllare o documentare automaticamente eventi e azioni in base ai termini di un contratto o di un accordo al verificarsi di specifiche condizioni.

Con l'integrazione di smart contract all'interno della Blockchain è possibile sfruttare la rete per trasferire ogni tipo di asset descritto dal contratto secondo particolari eventi, proprio come se fosse un contratto virtuale.

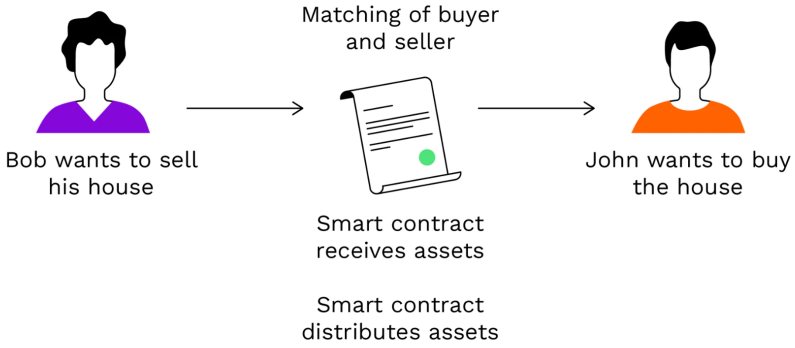


Figura 7 - Esempio di utilizzo di uno smart contract

Il vantaggio che uno smart contract possiede rispetto a un contratto reale è che assicura un "giudizio oggettivo" dato dal programma scritto. Il contratto eseguirà le determinate funzioni al verificarsi delle condizioni specificate, indipendentemente dal contesto o dai fattori esterni.

La prima criptovaluta che ha integrato gli smart contracts è stata Ethereum.

Questo ha permesso la nascita dei primi Dex (Decentralized Exchanges) come Uniswap, ossia exchange (siti che offrono servizi per scambiare criptovalute) completamente decentralizzati, quindi senza la presenza di un intermediario che gestisce gli scambi come avviene negli exchange classici. Lo scambio avviene totalmente tramite smart contracts e il processo è completamente automatizzato.

Questa funzionalità è quella che permette alle aziende e imprese di avvicinarsi alla tecnologia Blockchain, dato che permette di integrare quest'ultima a una moltitudine di sistemi.

In questo elaborato infatti, saranno utilizzati gli smart contract e la tecnologia blockchain per permettere alle persone di condividere i propri dati personali con organizzazioni (ospedali, test clinici, dottori ecc..) che ne fanno richiesta.

4 HyperLedger

Figura 8 - Logo Hyperledger

HyperLedger[16] è un progetto open source rilasciato nel 2015 con lo scopo di supportare lo sviluppo dei Distributed Ledger (Registri distribuiti) basati su tecnologia blockchain.

L'obiettivo è quello di avere un framework comune per lo sviluppo di blockchain in modo da aumentare la collaborazione tra i vari settori, e di conseguenza aumentare l'efficacia e l'affidabilità di questa tecnologia in modo da poter essere adottata da società tecnologiche, distributive e finanziarie.

Il progetto ha varie piattaforme ognuna progettata per usi specifici. Tra questi moduli ci sono blockchain con servizi per la gestione dell'identità, funzionalità per progetti IoT, storage e Smart Contracts.

4.1 HyperLedger Fabric

HyperLedger Fabric[17] è un framework del progetto *HyperLedger*, sviluppato inizialmente da IBM, e attualmente distribuito sotto la licenza della Linux Foundation.

Hyperledger Fabric fornisce le seguenti funzionalità di rete blockchain:

- **Gestione dell'identità -** Fabric è privato e *Permissioned*. A differenza di un sistema aperto e senza autorizzazione, Fabric registra nuovi membri della rete tramite un Membership Service Provider (MSP). Utilizza l'infrastruttura a chiave pubblica per produrre certificati crittografici legati a componenti di rete, organizzazioni, utenti finali o applicazioni client.

- **Elaborazione efficiente -** In Fabric ogni nodo ha uno o più ruoli. Per una migliore concorrenza e parallelismo sulla rete, l'impegno e l'esecuzione delle transazioni vengono mantenuti separati tra nodi peer e servizi ordinatori.

- **Funzionalità chaincode -** L'applicazione esterna che interagisce con il ledger può invocare gli Smart Contracts, chiamati anche “chaincode”. Fabric supporta chaincode scritti in Go, Node.js e Java.

- **Privacy e riservatezza -** Fabric offre anche l'opportunità di creare canali. Un ledger esiste nell'ambito di un canale. In questo modo un gruppo di partecipanti può creare un registro privato separato delle transazioni.

- **Design modulare -** L'architettura di Fabric offre diverse opzioni collegabili per quanto riguarda protocolli e formati di consenso.

4.2 Architettura di rete Fabric

Una rete *Hyperledger Fabric*[18] è composta da:

- Nodi peer
- Servizi ordinatori
- Canale/i
- Autorità di certificazione (MSP)
- Ledger (uno per canale)
- Smart contract (aka chaincode)

Gli utenti dei servizi di rete sono:

- Clients di amministratori di rete Blockchain
- Applicazioni client di proprietà di organizzazioni

Prendiamo in considerazione la seguente figura:

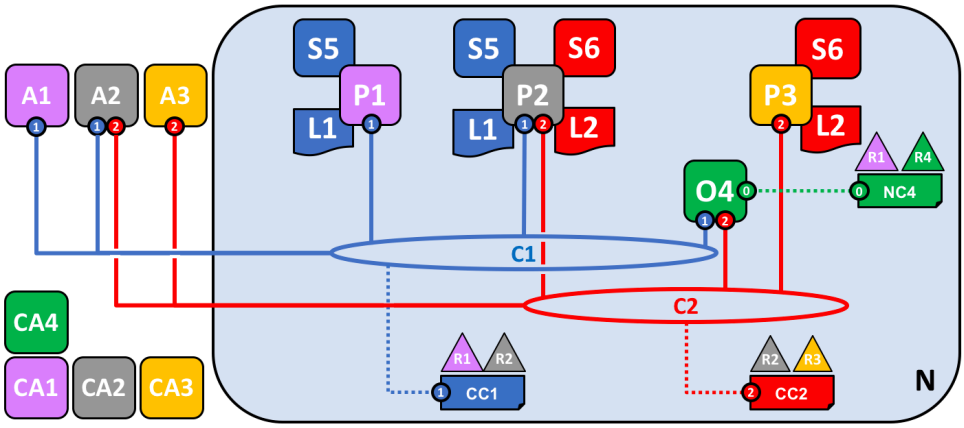


Figura 9 - Rete Fabric completa di esempio

Essa descrive una rete Fabric con quattro organizzazioni R1, R2, R3, R4:

R4 è stata scelta come organizzazione con il potere di configurare la prima versione della rete e non vuole effettuare transazioni sopra di essa;

R1 e R2 necessitano di un canale privato tra di loro, così come R2 e R3.

L'organizzazione R1 possiede un'applicazione per interagire con la rete ed effettuare transazioni all'interno del canale C1; R2 invece può eseguire transazioni sia nel canale C1 che C2, mentre R3 può interagire sul canale C2.

Il nodo peer P1 mantiene una copia del ledger L1 vincolata al canale C1 e il relativo chaincode S5, che può essere invocato da P1 in modo che l'applicazione client A1 possa leggere e aggiornare il ledger tramite esso.

Il nodo peer P2 ha sia una copia del ledger L1 che una copia del ledger L2, mentre P3 conserva solo una copia del ledger L2.

Il canale C1 segue le regole di policy definite nella configurazione del canale CC1 scelta da R1 e R2 e allo stesso modo il canale C2 segue quelle definite nella configurazione CC2 da R2 e R3.

La rete segue le regole di policy definite nella configurazione NC4, scelta da R1 e R4.

4.2.1 Peers / Nodi

In Fabric i peers possono avere due ruoli:

- **Committing peer** è un peer che ha il compito di convalidare blocchi di transazioni validate. Ogni peer che ha una copia del Ledger (blockchain) è un committing peer.

- **Endorsing peer** *(peer di approvazione)* è un peer che fornirà approverà la transazione che gli è proposta attraverso una firma digitale. E ’definito dalla policy di endorsement, e agisce attraverso uno smart contract che simula la transazione proposta, lasciando poi una risposta.
  La policy di endorsement definisce le organizzazioni i cui peer devono firmare digitalmente una transazione prima di includerla nel ledger.

4.2.2 Identità e Certificate Authorities

I partecipanti nel network possono essere orderers, peers, administrators o applicazioni client. Ogni partecipante deve avere un'identità digitale in un certificato digitale X.509. Le identità definiscono i permessi sulle risorse di ogni attore nella rete blockchain. Il processo di utilizzo di una CA per generare un'identità lato client è denominato come enrollment.

La Fabric Certificate Authority integrata, nota come Fabric CA, gestisce le identità digitali dei membri di Fabric.

4.2.3 Membership Service Provider (MSP)

Supponiamo che alla cassa in un negozio siano accettate solo alcune carte, ad es. Mastercard e Visa. Potresti avere un'altra carta, ad esempio American Express, che è valida e contiene denaro sufficiente, tuttavia quel tipo di carta non è accettato.

Figura 10 - PKI e MSP hanno un ruolo complementare: PKI fornisce identità, mentre MSP definisce quali di queste identità sono membri della rete Fabric

Come descritto in figura, PKI (Public Key Infrastructure) e MSP lavorano insieme in modo complementare. Una PKI emette molti diversi tipi di identità verificabili, mentre l'MSP definisce quali di queste identità sono i partecipanti fidati della rete.

Gli MSP, quindi, trasformano identità verificabili nei membri di una rete blockchain e definiscono quali CA sono attendibili per definire i partecipanti di una determinata rete.

Esistono due punti in cui gli MSP appaiono all'interno di una rete blockchain: nella configurazione del canale (MSP del canale) e localmente sull'attore stesso (MSP locale).

Gli MSP locali (presenti in utenti e peer) descrivono i diritti per quel nodo. Gli MSP di canale, invece, descrivono i diritti amministrativi e partecipativi a livello di canale.

4.2.5 Gateway

Un gateway si occupa di coordinare l'elaborazione delle proposte di transazione, degli ordini e delle comunicazioni tra le diverse componenti della rete, consentendo ad un'applicazione lato client di concentrarsi principalmente sulla generazione, invio e risposta delle transazioni.

Ogni gateway si occupa quindi di conoscere la topologia di rete con i relativi peer e orderers e di coordinare le comunicazioni lasciando che le applicazioni client si concentrino sulla logica del programma.

La configurazione del gateway è definita nel profilo di connessione e può essere:

- **Statica** - La configurazione del gateway è completamente definita nel profilo di connessione. Tutti i peers, orderers e CA disponibili per una applicazione sono definiti staticamente nel profilo di connessione utilizzato per configurare il gateway.

- **Dinamica** - La configurazione del gateway è minimamente definita nel profilo di connessione, in cui vengono specificati uno o due peers dall'organizzazione dell'applicazione. Per il resto della rete si utilizza un service discovery per rilevarne la topologia che include peers, orderers, canali, contratti e endorsement policies.

4.2.6 Ledger

Il ledger è composto da due parti:

- **Il world state** (stato globale) - Di solito definito da coppie chiave-valore, contiene i valori aggiornati del database.
- **La blockchain** - Un registro delle transazioni read-append-only (si può solo leggere e aggiungere transazioni, non modificare le già presenti)

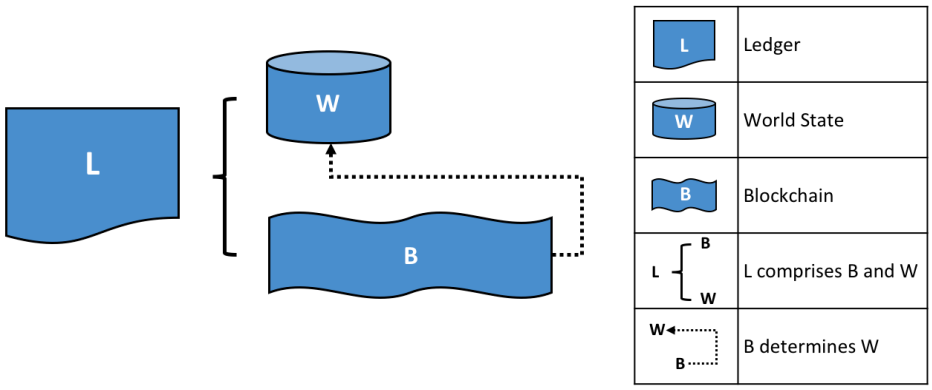


Figura 11 - Ledger L è composto da blockchain B e World State W. Blockchain B definisce implicitamente il world state W

World state - Il world state contiene i valori aggiornati di ogni record del database.

Attualmente può essere implementato come LevelDB e CouchDB.

LevelDB è l'impostazione predefinita ed è particolarmente appropriato per coppie chiave-valore, mentre CouchDB è una scelta consona quando i dati sono strutturati come documenti JSON perché supporta query avanzate e tipi di dati più ricchi (per esempio in transazioni commerciali).

Come descritto nella figura sottostante, gli stati contengono anche un numero di versione che viene incrementato ogni volta che essi cambiano, quindi ogni volta che vengono aggiornati. Questo è importante perché consente di controllare se la versione corrisponde a quella di quando la transazione è stata creata, garantendo che lo stato sia aggiornato come previsto.

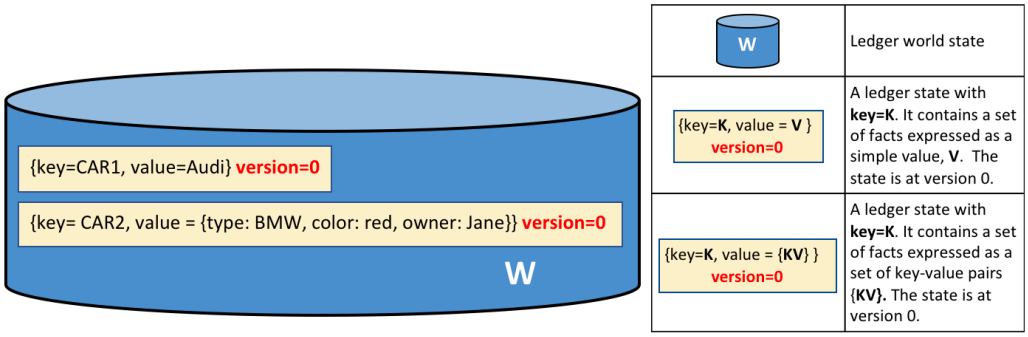


Figura 12 - Due esempi di un ledger state

Blockchain - La blockchain è un registro delle transazioni, composto da blocchi collegati. I blocchi sono costituiti da batch di transazioni, ciascuna delle quali può essere una query o un aggiornamento del ledger.

L'intestazione di ogni blocco contiene l'hash del blocco insieme all'hash del blocco precedente, in modo che ogni blocco sia collegato al precedente.

Viene solitamente implementata come file, a differenza del world state che utilizza un database. Questa è una scelta di progettazione abbastanza ovvia poichè la blockchain è fortemente orientata verso operazioni semplici e veloci.

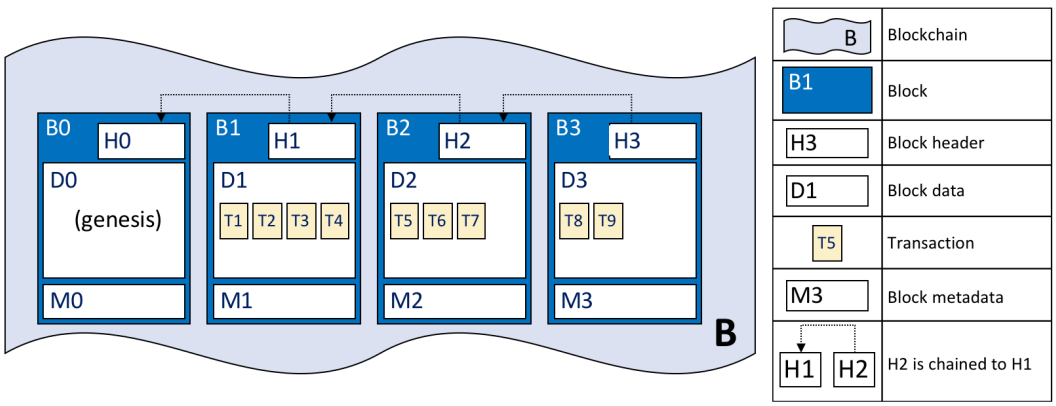


Figura 13 - La blockchain B è composta da B0, B1, B2 e B3 dove B0 è il blocco genesi

Transazioni - I campi principali nelle transazioni sono:

- *Header* - Contiene metadati come il nome e la versione del chaincode.

- *Signature* - E ’generata usando la chiave privata dell’applicazione, e serve come prova che la transazione non è stata manipolata.

- *Proposal* - Contiene i parametri di input dati dall’applicazione per eseguire una determinata transazione nel chaincode.

- *Response* - Contiene l’output del chaincode.

- *Endorsements* - È una lista di response ai proposal fornita da peer inclusi nell’endorsement policy.

4.3 Basic Transaction Flow

Simuliamo una semplice transazione tra due peers. L'endorsement policy prevede che entrambi i peers debbano approvare qualsiasi transazione.

Fase 1: Proposta - La fase 1 coinvolge solo i peers e non gli orderers. I peers di approvazione verificano che la proposta di transazione abbia una struttura corretta, che non sia stata già presentata in passato (protezione da replay-attack), che la firma sia valida (utilizzando l'MSP) e che il mittente sia autorizzato a eseguire l'operazione proposta su quel canale. Gli endorsing peers simulano (non vengono apportati aggiornamenti al ledger) quindi la transazione per calcolare il set di lettura e scrittura. L'insieme di questi valori, in aggiunta alla firma del peer di approvazione, viene passato come "risposta alla proposta" (proposal response). La fase 1 termina quindi con l'applicazione che dispone di tutte le risposte alla proposta di transazione richieste dall'endorsing policy. Nel caso in cui i peers restituiscano all'applicazione risposte di proposta di transazione diverse e incoerenti, l'applicazione richiede una risposta più aggiornata.

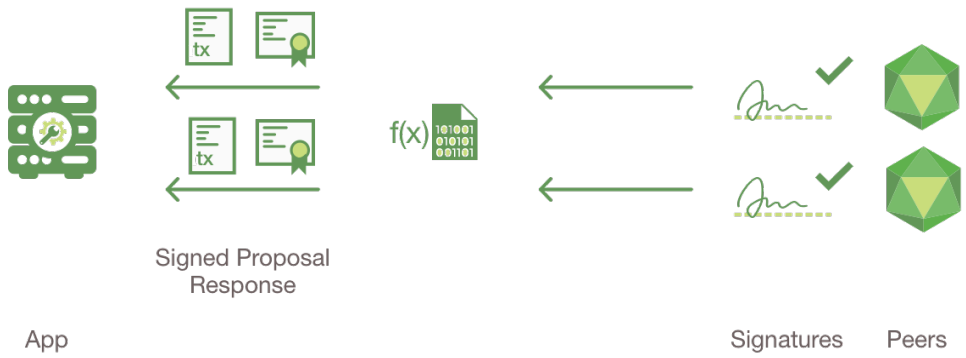


Figura 14 - Transaction Flow: Fase 1

Fase 2: Impacchettamento - La fase 2 coinvolge principalmente l'orderer. Se il chaincode esegue solo query sul ledger, l'applicazione ispezionerà solo la risposta alla query e in genere non invierà la transazione al servizio di ordinazione. Nel nostro caso invece, le transaction proposal con tutti i relativi endorsements sono inviate all'orderer (ordinatore). Una volta che l'orderer riceve molte transazioni, le ordina e crea un nuovo blocco pronto per essere inviato a tutti i committing peers. L'ordine in cui l'orderer ordina le transazioni può essere diverso dall'ordine di arrivo. La cosa importante è che ci sia un ordine, e che questo sia uguale per ogni committing peer.

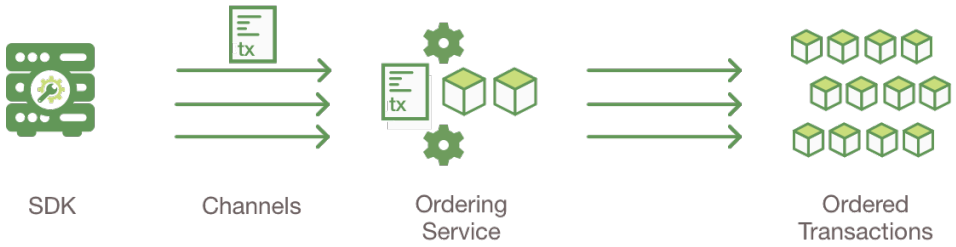


Figura 15 - Transaction Flow: Fase 2

Fase 3: Validazione - Nella fase 3, l'orderer invia il blocco preparato ad ogni committing peer della rete. Ogni peer, controlla indipendentemente che ogni transazione nel blocco abbia i relativi endorsements e dopo di che applica le modifiche al ledger.

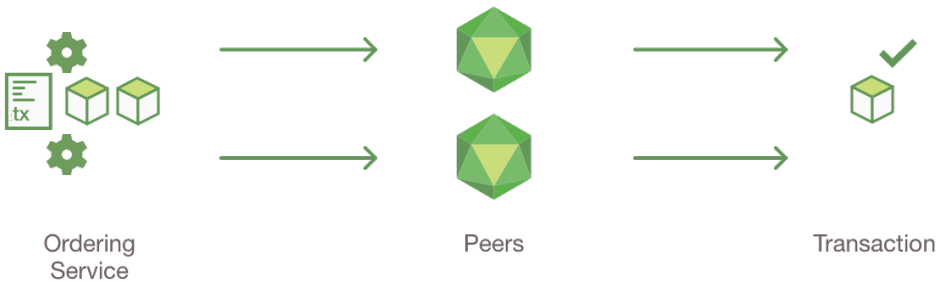


Figura 16 - Transaction Flow: Fase 3

N.B.Tutte le transazioni, valide o non valide, verranno registrate sulla blockchain, ma solo le transazioni valide aggiorneranno il world state.

5 Sistema di consenso al trattamento dati

Lo scopo di questo esempio è di utilizzare i concetti della blockchain, insieme alla piattaforma *HyperLedger*, mostrandone un potenziale utilizzo per risolvere un problema reale attualmente presente riguardante la gestione del consenso del trattamento dei dati personali.

5.1 Introduzione

L'obiettivo principale dell'esempio è quello di mediare, attraverso tecnologia blockchain, lo scambio di informazioni personali tra individui e organizzazioni.

Questo è un rischio poichè il trasferimento potrebbe causare problemi di privacy e riservatezza; senza tecnologia blockchain, infatti, questo scambio dovrebbe coinvolgere una terza parte non completamente affidabile che, spinta dal proprio interesse, potrebbe vendere le informazioni ad altri, omettere dati o commettere errori.

Nel nostro esempio, il tutto avviene tramite regole (Smart Contracts) definite al momento della creazione della rete.

Esso consiste in una persona e una clinica che si relazionano a vicenda tramite il chaincode caricato sul test network.

L'utente possiede un documento che potrebbe essere un record medico. La clinica chiede il trasferimento del documento per un determinato periodo, al termine del quale, dalla clinica o dall'utente stesso, può essere avviata la procedura di restituzione.

La rete dell'esempio è quindi composta da due organizzazioni: una per le persone e una per la clinica.

Come linguaggio di programmazione utilizzeremo Node.js.

Usando *Hyperledger Fabric* modelleremo questa rete e ne simuleremo il funzionamento.

5.2 Architettura della rete

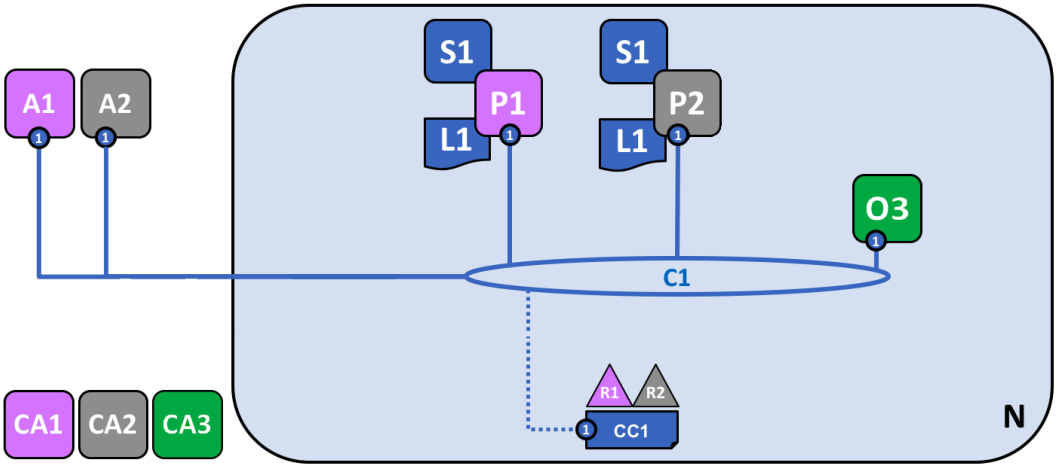


Figura 17 - Architettura di rete del nostro esempio

P1 e P2 sono i nodi peer delle organizzazioni. I due nodi hanno una copia del ledger L1 vincolato al canale C1 e un chaincode S1 che può essere invocato da entrambi. Le applicazioni client A1 e A2 possono leggere e aggiornare il ledger tramite i corrispettivi nodi peer. Il canale C1 segue le regole di policy definite nella configurazione del canale CC1 scelta da R1 e R2.

Nel nostro caso di esempio la policy che utilizziamo è la predefinita del canale, che richiede che la maggior parte delle organizzazioni sul canale approvi una transazione (quindi entrambe).

Nei prossimi capitoli quindi definiremo il chaincode S1 e le applicazioni client A1 e A2.

5.3 Organizzazioni

Le due organizzazioni della rete nel nostro esempio identificano una clinica e una generica organizzazione che gestisce i dati personali dei propri utenti, come un'organizzazione statale (eg spid, lepida ecc..).

Ciononostante, in un'implementazione reale l'organizzazione Clinica può rappresentare qualsiasi ente che ha l'interesse a chiedere accesso a documenti e dati di persone.

Dato che la rete di prova conterrà due organizzazioni (Clinica e Utente) e il chaincode, modifichiamo il nostro spazio di lavoro concettualmente in tre cartelle per simulare successivamente la rete.

La divisione delle organizzazioni serve principalmente in ambiente di testing per separare le applicazioni client che saranno differenti, ma che dovranno essere simulate dallo stesso computer.

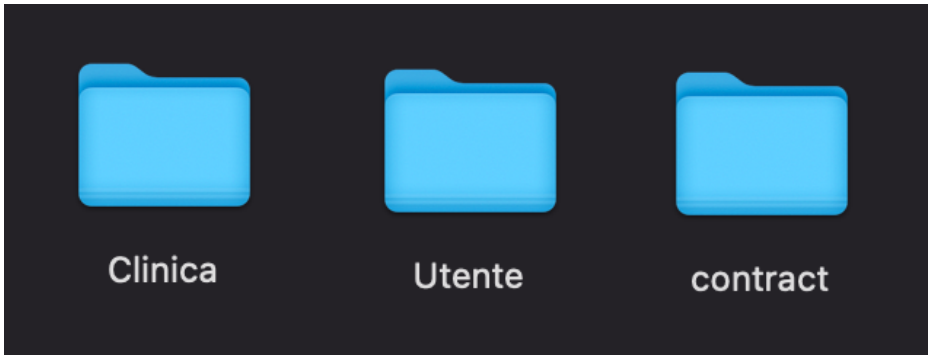


Figura 18 - File System del progetto

N.B. Nei seguenti capitoli i termini "Utente" e "Clinica" si riferiranno alle relative organizzazioni.

5.4 Transaction Flow

Il transaction flow sarà il seguente:

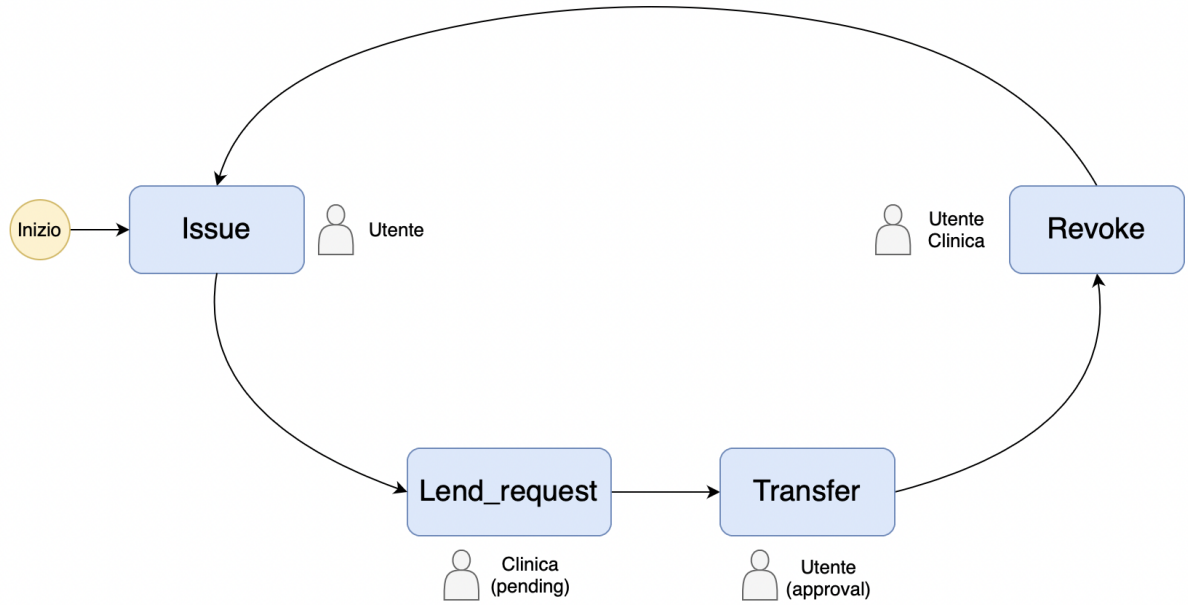


Figura 19 - Transaction flow della rete

La transazione *issue* rilascia (crea) il contratto nel network.

Lend_request chiede il trasferimento del documento per un determinato periodo e *transfer* approva la richiesta e trasferisce la proprietà.

Infine, la transazione *revoke,* dopo aver controllato che il periodo di prestito è scaduto, toglie l'accesso al documento a Clinica,

Issue e *Transfer* sono richiamabili solamente da Utente, *lend_request* da Clinica, mentre *revoke* da entrambe le organizzazioni.

5.5 Chaincode

All'interno della cartella contract definiamo il chaincode.

Il chaincode, come spiegato nei capitoli precedenti è lo smart contract caricato sul canale che contiene le funzioni con le quali i clients dovranno comunicare per apportare modifiche al ledger.

5.5.1 Classe Document e DocumentList

Cominciamo con il definire la classe **Document**.

Il documento ha quattro stati:

- **Issued**: Per indicare che è stato appena rilasciato
- **Pending**: Per indicare è in attesa di approvazione del trasferimento
- **Trading**: Per indicare è stato scambiato
- **Revoked**: Per indicare che è stato revocato

Inoltre disporrà delle funzioni *fromBuffer()* e *toBuffer()* per il convertimento tra formato Buffer (utilizzato da api hyperledger) e Json, la funzione *createInstance()* per istanziare la classe e *getClass()* che ritorna il namespace del chaincode in cui opera la classe.

Definiamo anche la classe **DocumentList** che rappresenta la lista di documenti presenti nella blockchain e che ci servirà per aggiungere, leggere e modificare i documenti nel ledger tramite le relative funzioni *addDocument()*, *getDocument()* e *updateDocument()*.

5.5.2 Classi State e StateList

Le classi Document e DocumentList sono estese rispettivamente da *State* e *StateList*. Questa divisione concettuale è di buona norma siccome ogni eventuale entità e relativa lista che vorremo aggiungere avrà bisogno degli stessi metodi e attributi.

Nella classe **State** contiene le funzioni *serialize()* e *deserialize()* che sono chiamate da *toBuffer()* e *fromBuffer()* per effettuare la conversione, insieme a *makeKey()* e *splitKey()* che servono per comporre e scomporre la chiave composta identificatrice (issuer + numero documento).

La classe **StateList** contiene le funzioni *addState()*, *getState()* e *updateState()* che utilizziamo per gestire gli stati (nel nostro caso rappresentati come documenti) nel ledger.

5.5.3 Classe QueryUtils

Aggiungiamo anche una classe **QueryUtils** che utilizzeremo per effettuare queries (interrogazioni) alla blockchain per poi mostrarne i dati.

Al suo interno definiremo la funzione *getAssetHistory()* per ritornare la cronologia delle transazioni del documento e le funzioni *getKeyByIssuer(), getKeyByOwner()* e *queryKeyByState() che* restituiranno rispettivamente la lista dei documenti dell'issuer, dell'owner e dello stato passato come parametro.

Infine definiremo la funzione *getAllResult()* per convertire il risultato delle query in formato Json leggibile.

5.5.4 Classe DocumentContract e DocumentContext

A questo punto non ci resta che definire il contratto vero e proprio che interagirà con le classi *Document*, *DocumentList* e *QueryUtils* definite precedentemente e al quale comunicheremo tramite le applicazioni client.

Definiamo due classi: **DocumentContext** e **DocumentContract** che estendono rispettivamente le classi predefinite di Fabric; Context e Contract.

Il Context è necessario per ogni Contract e serve per accedere alla lista degli stati sul ledger tramite l'oggetto DocumentList. Contract invece sarà il contratto con le relative funzioni al suo interno.

Procediamo col definire all'interno di DocumentContract le transazioni menzionate nel capitolo del Transaction Flow:

Transazione Issue() - Crea il documento sul ledger, lo pone nello stato "issued" e ne memorizza i valori tra cui issuer, informazioni e data di rilascio.

Transazione lend_request() - Pone il documento in stato "pending" e ne memorizza la data di termine del prestito richiesto.

Transazione *transfer()* - Trasferisce il documento cambiando il proprietario e ne pone lo stato in "trading". Memorizziamo anche l'mspid del nuovo proprietario per effettuare successivamente controlli di autenticazione.

Transazione *revoke()* - Dopo aver controllato che il periodo di prestito è effettivamente terminato, toglie lo stato di proprietario alla clinica e pone il documento in stato "revoked".

Infine, definiamo transazioni per leggere il documento ed effettuare queries:

Transazione *read()* - Può essere effettuata solo dal proprietario e dall'issuer del documento.

Transazioni *queryHistory()*, *queryOwner()*, *queryIssuer()* e *queryState()* - Utilizzano la classe QueryUtils definita precedentemente per effettuare queries al ledger e passare il risultato all'applicazione client.

5.6 Applicazioni client

Le applicazioni client servono alle organizzazioni per comunicare con il chaincode presente sul canale e le definiamo nelle rispettive cartelle delle organizzazioni.

Utilizziamo per i clients un gateway statico in cui sono specificati i peers, CA e mspid dell'organizzazione alla quale il client appartiene.

5.6.1 Applicazione enrollUser

Ogni client ha bisogno, per interagire con la rete, di un'identità rilasciata dalla CA della propria organizzazione. Nel nostro esempio abbiamo due identità: Alice per Utente e Andrea per Clinica.

Per ottenere l'identità Alice, eseguiamo il processo di enrollment attraverso il programma *enrollUser.js*. Dopo aver generato una coppia di chiavi privata e pubblica, il programma invia una Certificate Signing Request, ossia una richiesta di firma del certificato, alla CA.

Se il nome e il segreto corrispondono alle credenziali registrate presso la CA, sarà emesso un certificato codificato con la chiave pubblica di Alice, che stabilisce che Alice appartiene all'organizzazione Utente.

Le credenziali registrate, nel nostro caso, sono le credenziali predefinite della rete test di Fabric e saranno "user1" per il nome e "user1pw" per il segreto. Una volta che la signing request è terminata, il programma archivia la chiave privata e il certificato nel portafoglio di Alice. Il processo di enrollment per l'organizzazione Clinica è lo stesso e crea l'identità Andrea.

5.6.2 Struttura delle applicazioni

Dopo aver definito il programma di enrollment possiamo procedere con la definizione delle applicazioni che interagiranno con il chaincode.

Queste applicazioni in un caso di uso reale sarebbero chiamate da altri programmi e i parametri inviati sarebbero gestiti dinamicamente. Siccome, però, il nostro obiettivo è quello di simulare la rete localmente e accertarne il corretto funzionamento, chiameremo le transazioni con parametri statici.

Tutte le applicazioni si possono riassumere in sei passaggi:

1. Selezione dell'identità dal portafoglio

```
'use strict';

const fs = require('fs');
const yaml = require('js-yaml');
const { Wallets, Gateway } = require('fabric-network');
const Document = require('../../contract/lib/document.js');


async function main() {

    //carichiamo il wallet
    const wallet = await Wallets.newFileSystemWallet('../identity/user/alice/wallet');

  try {

    //dato che utilizziamo una rete di test definiamo il nome utente nel programma
    //in un caso uso reale questo parametro sarebbe passato esternamente
    const userName = 'alice';

    // Impostiamo le opzioni di connessione; in questo caso nome e wallet
    let connectionOptions = {
      identity: userName,
      wallet: wallet,
      discovery: { enabled:true, asLocalhost: true }
    };
```

2. Connessione al gateway

```
    // carichiamo la configurazione del gateway dell'org2
    let connectionProfile = yaml.safeLoad(fs.readFileSync('../gateway/connection-org2.yaml', 'utf8'));

    // Il gateway ci permette di connetterci ai peer che comunicano con la rete
    const gateway = new Gateway();

    await gateway.connect(connectionProfile, connectionOptions);

    console.log('Connesso al gateway.');
```

3. Connessione al network

```
    console.log('Network channel in uso: mychannel.');

    const network = await gateway.getNetwork('mychannel');
```

4. Invio della transazione

```

    //leggiamo il contratto dal network
    const contract = await network.getContract('documentcontract', 'org.docnet.documentcontract');

    // read document
    console.log('Inviando la transazione read del documento.');

    //chiamiamo la funzione read con i relativi parametri
    const readResponse = await contract.submitTransaction('read', 'alice', '00001', 'Org2MSP');

```

5. Elaborazione della risposta

```
50
51      console.log('Elaborazione risposta di read.');
52
53      //deserializziamo l'oggetto json ritornato attraverso una funzione della classe Document
54      let document = Document.fromBuffer(readResponse);
55
56      console.log(`Documento di ${document.issuer} : ${document.docNumber} è stato recuperato con successo.\ Il contenuto è: ${document.data}`);
57
58      console.log('Transazione eseguita correttamente.');
```

6. Gestione degli errori

```
59
60    } catch (error) {
61
62      console.log(`Errore nell'elaborazione della transazione. ${error}`);
63      console.log(error.stack);
64
65    } finally {
66      console.log('Disconnesso dal gateway')
67      gateway.disconnect();
68
69    }
70  }
71  main().then(() => {
72
73    console.log('Programma read completato.');
74
75  }).catch((e) => {
76
77    console.log('Eccezione nel programma read.');
78    console.log(e);
79    console.log(e.stack);
80    process.exit(-1);
81
82  });
```

Poiché i primi tre passaggi e l'ultimo sono gli stessi, saranno mostrati solo il quarto e quinto di ogni applicazione.

Le applicazioni enrollUser.js, revoke.js, read.js e queryapp.js sono definite per ciascuna organizzazione.

5.6.3 Applicazione Issue

L'applicazione *issue.js* esegue la transazione issue() passando i parametri richiesti definiti nel chaincode. Inviamo quindi il nome "Alice", il numero del documento "00001", il contenuto "Dati di prova" e la data di rilascio "2022-02-10".

In questo esempio il dato del documento è una semplice stringa, ma in un caso reale potrebbe essere un oggetto json, un file o un'immagine rappresentati da una stringa Base64.

```
43
44      console.log('Inviando la transazione issue del documento.');
45
46      //chiamiamo la funzione issue con i relativi parametri
47      const issueResponse = await contract.submitTransaction('issue', userName , '00001', 'Dati di prova', '2022-02-10');
48
49      console.log('Elaborazione risposta di issue.\n '+issueResponse);
50
51      //deserializziamo l'oggetto json ritornato attraverso una funzione della classe Document
52      let document = Document.fromBuffer(issueResponse);
53
54      console.log(`Documento di ${document.issuer} : ${document.docNumber} correttamente rilasciato.`);
55      console.log('Transazione eseguita correttamente.');
```

5.6.3 Applicazione Lend_Request

L'applicazione *lend_request.js* esegue la transazione lend_request() del chaincode passando come parametri gli identificativi del documento per il quale facciamo domanda di accesso. In questo caso passiamo come parametri gli identificativi del documento (nome e numero) che creiamo con l'applicazione issue appena definita aggiungendo la data fino alla quale vorremmo avere accesso al documento stesso.

```
50
51      console.log('Inviando la transazione lend_request del documento.');
52
53      //chiamiamo la funzione lend_request con i relativi parametri
54      const lendResponse = await contract.submitTransaction('lend_request', 'alice', '00001','2022-03-15');
55
56      console.log('Elaborazione risposta di lend_request.');
57
58      //deserializziamo l'oggetto json ritornato attraverso una funzione della classe Document
59      let document = Document.fromBuffer(lendResponse);
60
61      console.log(`Documento di ${document.issuer} : ${document.docNumber} è stato prestato provvisoriamente: il trasferimento deve ora essere completato
62      console.log('Transazione eseguita correttamente.');
```

5.6.3 Applicazione Transfer

L'applicazione *transfer.js* approva la richiesta effettuata eseguendo la transazione transfer() del chaincode. Qui, oltre a passare gli identificativi del documento, passiamo il nome di chi ha fatto la richiesta precedentemente, l'mspid della sua organizzazione e la data del trasferimento.

```
43
44      console.log('Inviando la transazione transfer del documento.');
45
46      //chiamiamo la funzione transfer con i relativi parametri
47      const transferResponse = await contract.submitTransaction('transfer', 'alice', '00001', 'andrea', 'Org1MSP', '2022-02-12');
48
49      console.log('Elaborazione risposta di transfer. \n'+ transferResponse);
50
51      //deserializziamo l'oggetto json ritornato attraverso una funzione della classe Document
52      let document = Document.fromBuffer(transferResponse);
53
54      console.log(`Documento di ${document.issuer}  : ${document.docNumber} è stato trasferito correttamente`);
55      console.log('Transazione eseguita correttamente.');
```

5.6.3 Applicazione Revoke

L'applicazione *revoke.js* toglie l'accesso al documento alla Clinica eseguendo la transazione revoke(). Passiamo come parametro gli identificativi del documento, l'mspid dell'issuer e la data di revoca.

```
44
45  // revoke document
46  console.log('Inviando la transazione revoke del documento.');
47
48  //chiamiamo la funzione revoke con i relativi parametri
49  const revokeResponse = await contract.submitTransaction('revoke', 'alice', '00001', 'Org2MSP', '2020-11-30');
50
51  console.log('Elaborazione risposta di revoke.');
52
53  //deserializziamo l'oggetto json ritornato attraverso una funzione della classe Document
54  let document = Document.fromBuffer(revokeResponse);
55
56  console.log(`Documento di ${document.issuer} : ${document.docNumber} è stato ritornato con successo da ${document.owner}`);
57
58  console.log('Transazione eseguita correttamente.');
```

5.6.3 Applicazione Read e QueryApp

L'applicazione *read.js* esegue la transazione read() e mostra il contenuto del documento. Passiamo come parametri gli identificativi del documento che vogliamo leggere.

```
44
45  // read document
46  console.log('Inviando la transazione read del documento.');
47
48  //chiamiamo la funzione read con i relativi parametri
49  const readResponse = await contract.submitTransaction('read', 'alice', '00001', 'Org2MSP');
50
51  console.log('Elaborazione risposta di read.');
52
53  //deserializziamo l'oggetto json ritornato attraverso una funzione della classe Document
54  let document = Document.fromBuffer(readResponse);
55
56  console.log(`Documento di ${document.issuer} : ${document.docNumber} è stato recuperato con successo.\ Il contenuto è: ${document.data}`);
57
58  console.log('Transazione eseguita correttamente.');
```

Infine l'applicazione *queryapp.js* esegue le transazioni di query definite nel chaincode e ne mostra i risultati.

```
44      //eseguiamo queries
45      console.log('-----------------------------------------------------------------------------------------');
46      console.log('****** Inviando query ****** \n\n ');
47
48      console.log('1. Cronologia transazioni....');
49      console.log('-----------------------------------------------------------------------------------------\n');
50      let queryResponse = await contract.evaluateTransaction('queryHistory', 'alice', '00001');
51
52      let json = JSON.parse(queryResponse.toString());
53      console.log(json);
54      console.log('\n\n');
55      console.log('-----------------------------------------------------------------------------------------\n\n');
56
57      console.log('2.Documenti rilasciati da alice');
58      console.log('-----------------------------------------------------------------------------------------\n');
59      let queryResponse2 = await contract.evaluateTransaction('queryIssuer', 'alice');
60      json = JSON.parse(queryResponse2.toString());
61      console.log(json);
62
63      console.log('\n\n');
64      console.log('-----------------------------------------------------------------------------------------\n\n');
65
66      console.log('3.Documenti a cui Andrea ha accesso');
67      console.log('-----------------------------------------------------------------------------------------\n');
68      let queryResponse3 = await contract.evaluateTransaction('queryOwner', 'andrea');
69      json = JSON.parse(queryResponse3.toString());
70      console.log(json);
71
72      console.log('\n\n');
73      console.log('-----------------------------------------------------------------------------------------\n\n');
74      console.log('4. Query per attributo -> tutti i documenti revoked' );
75      console.log('-----------------------------------------------------------------------------------------\n');
76      let queryResponse4 = await contract.evaluateTransaction('queryState', 'revoked');
77
78      json = JSON.parse(queryResponse4.toString());
79      console.log(json);
80      console.log('\n\n');
81      console.log('-----------------------------------------------------------------------------------------\n\n');
```

5.7 Creazione del test network enviroment

Per creare il network di test utilizzeremo script bash forniti da Fabric e l'applicazione Docker che ci permette di creare contenitori virtuali per simulare più dispositivi sullo stesso computer.

Apriamo il terminale e eseguiamo lo script bash network-starter.sh che crea il canale, i peers, l'orderer, i relativi certificate authorities e due couch database dove saranno memorizzate le copie del ledger per i due peers.

Possiamo visualizzare i docker aperti con il comando "docker ps":

```
michele@Air-di-Michele projtesi % docker ps
CONTAINER ID   IMAGE                               COMMAND                  CREATED          STATUS          PORTS                                                                                  NAMES
cb2ff8752174   hyperledger/fabric-tools:latest     "/bin/bash"              33 seconds ago   Up 30 seconds                                                                                          cli
dc7dedb56155   hyperledger/fabric-peer:latest      "peer node start"        33 seconds ago   Up 31 seconds   0.0.0.0:7051->7051/tcp, 0.0.0.0:9444->9444/tcp                                         peer0.org1.example.com
cb7f1c1c08ea   hyperledger/fabric-peer:latest      "peer node start"        33 seconds ago   Up 31 seconds   0.0.0.0:9051->9051/tcp, 7051/tcp, 0.0.0.0:9445->9445/tcp                               peer0.org2.example.com
295c1d4218f5   couchdb:3.1.1                       "tini -- /docker-ent…"   33 seconds ago   Up 32 seconds   4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp                                             couchdb0
70f31363e840   couchdb:3.1.1                       "tini -- /docker-ent…"   33 seconds ago   Up 32 seconds   4369/tcp, 9100/tcp, 0.0.0.0:7984->5984/tcp                                             couchdb1
eebfe72874c7   hyperledger/fabric-orderer:latest   "orderer"                33 seconds ago   Up 32 seconds   0.0.0.0:7050->7050/tcp, 0.0.0.0:7053->7053/tcp, 0.0.0.0:9443->9443/tcp                 orderer.example.com
9aa047066f18   hyperledger/fabric-ca:latest        "sh -c 'fabric-ca-se…"   44 seconds ago   Up 43 seconds   0.0.0.0:7054->7054/tcp, 0.0.0.0:17054->17054/tcp                                       ca_org1
7516e7d30f48   hyperledger/fabric-ca:latest        "sh -c 'fabric-ca-se…"   44 seconds ago   Up 43 seconds   0.0.0.0:9054->9054/tcp, 7054/tcp, 0.0.0.0:19054->19054/tcp                             ca_orderer
6cf493ca3d3d   hyperledger/fabric-ca:latest        "sh -c 'fabric-ca-se…"   44 seconds ago   Up 43 seconds   0.0.0.0:8054->8054/tcp, 7054/tcp, 0.0.0.0:18054->18054/tcp                             ca_org2
michele@Air-di-Michele projtesi %
```

Nel nostro esempio org1 sarà l'organizzazione Clinica, mentre org2 sarà l'organizzazione Utente.

5.8 Distribuzione del chaincode sul canale

Prima che il chaincode possa essere invocato dalle applicazioni, deve essere installato sui nodi peer della rete di test e quindi definito sul canale utilizzando il Fabric chaincode lifecycle, il quale consente a più organizzazioni di accettare la definizione del chaincode prima che venga distribuito su un canale. Di conseguenza, dobbiamo installare e approvare il contratto sia come organizzazione Utente che come organizzazione Clinica.

Il processo descritto di seguito dovrà quindi essere eseguito da entrambe le organizzazioni. In questo elaborato verrà mostrato soltanto per l'organizzazione Utente poichè il procedimento è lo stesso.

5.8.1 Installazione

Apriamo un'altra finestra di terminale e navighiamo alla cartella Utente.

Eseguiamo lo script bash utente.sh fornito da Fabric che configura le apposite variabili di ambiente in modo che il terminale si relazioni all'applicazione come se fosse l'org2.

Adesso possiamo procedere con l'impacchettamento del codice tramite l'apposito comando che comprime il chaincode in formato tar.gz pronto per essere poi installato sul peer. Specifichiamo il linguaggio, il percorso relativo e il nome del contratto con gli appositi flag.

```
peer lifecycle chaincode package doc.tar.gz --lang node --path ../contract --label doc_0
```

E installiamo il pacchetto sul peer.

```
peer lifecycle chaincode install doc.tar.gz
 21:59:10.416 CET 0001 INFO [cli.lifecycle.chaincode] submitInstallProposal -> Installed remotely
```

5.8.2 Approvazione

Dopo l'installazione approviamo la definizione del chaincode da parte del peer, specificando tramite flags l'indirizzo e il nome dell'orderer host, il nome del canale, il nome e la versione del chaincode e l'id del pacchetto installato.

Omettendo il flag --policy accettiamo di utilizzare l'endorsement policy predefinita del canale, che nel caso del canale di test richiede che la maggior parte delle organizzazioni approvi una transazione.

```
bash-3.2$ peer lifecycle chaincode approveformyorg --orderer localhost:7050 --ordererTLSHostnameOverride
 orderer.example.com --channelID mychannel --name documentcontract -v 0 --package-id $PACKAGE_ID --seque
nce 1 --tls --cafile $ORDERER_CA
```

5.8.3 Commit

Dopo aver eseguito l'installazione e l'approvazione anche per l'organizzazione Clinica possiamo procedere. Poiché entrambe le organizzazioni possono effettuare il commit del chaincode al canale, continueremo a operare come organizzazione Utente. Specifichiamo con gli appositi flag i dati di orderer, dei peers, del canale e del chaincode.

Dopo aver eseguito il commit sul canale, verrà creato un nuovo contenitore Docker per il chaincode che sarà utilizzato per eseguire il contratto su entrambi i peers.

```
bash-3.2$ peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride orderer.example
.com --peerAddresses localhost:7051 --tlsRootCertFiles ${PEER0_ORG1_CA} --peerAddresses localhost:9051 -
-tlsRootCertFiles ${PEER0_ORG2_CA} --channelID mychannel --name documentcontract -v 0 --sequence 1 --tls
 --cafile $ORDERER_CA –waitForEvent
```

5.9 Trasferimento e risultati

Dato che per il trasferimento simuliamo di essere entrambe le organizzazioni, utilizzeremo due terminali: uno per Utente e uno per Clinica.

Cominciamo con l'eseguire l'enrollment di entrambi gli utenti delle organizzazioni.

```
bash-3.2$ node enrollUser.js
Utente "alice" creato e importato nel wallet
```

```
bash-3.2$ node enrollUser.js
Utente "andrea" creato e importato nel wallet
```

Ora che abbiamo creato le identità possiamo rilasciare il primo documento di Alice tramite l'applicazione issue.

```
bash-3.2$ node issue.js
Connesso al gateway.
Network channel in uso: mychannel.
Inviando la transazione issue del documento.
Elaborazione risposta di issue.
 {"class":"org.docnet.documentcontract","issuer":"alice","docNumber":"00001","issueDateTime":"2022-02-10",
"currentState":1,"mspid":"Org2MSP","owner":"alice","data":"Dati di prova"}
Documento di alice : 00001 correttamente rilasciato.
Transazione eseguita correttamente.
Disconnesso dal gateway
Programma issue completato.
```

Il documento attualmente può essere letto da Alice, ma non da Andrea.

```
bash-3.2$ pwd
/Users/michele/test/fabric/projtesi/organization/Utente/application
bash-3.2$ node read.js
Connesso al gateway.
Network channel in uso: mychannel.
Inviando la transazione read del documento.
Elaborazione risposta di read.
Documento di alice : 00001 è stato recuperato con successo.
Il contenuto è: Dati di prova
Transazione eseguita correttamente.
Disconnesso dal gateway
Programma read completato.
```

```
bash-3.2$ pwd
/Users/michele/test/fabric/projtesi/organization/Clinica/application
bash-3.2$ node read.js
Connesso al gateway.
Network channel in uso: mychannel.
Inviando la transazione read del documento.
2022-03-14T09:23:24.723Z - error: [Transaction]: Error: No valid resp
    peer=peer0.org2.example.com:9051, status=500, message=error in si
th failure: Error:
Il documento alice00001 non può essere letto da Org1MSP, siccome non
 l'issuer
    peer=peer0.org1.example.com:7051, status=500, message=error in si
```

Per far sì che anche Andrea possa leggere il documento, dobbiamo effettuare il trasferimento. Iniziamo la procedura con la richiesta di prestito da parte di Andrea.

```
bash-3.2$ node lend_request.js
Connessione al gateway...
Network channel in uso: mychannel.
Inviando la transazione lend_request del documento.
Elaborazione risposta di lend_request.
Documento di alice : 00001 è stato prestato provvisoriamente
Il trasferimento deve ora essere completato dal titolare del documento
Transazione eseguita correttamente.
Disconnesso dal gateway
Programma lend_request completato.
```

E approviamo la richiesta con Alice.

```
bash-3.2$ node transfer.js
Connesso al gateway.
Network channel in uso: mychannel.
Inviando la transazione transfer del documento.
Elaborazione risposta di transfer.
{"class":"org.docnet.documentcontract","currentState":3,"data":"Dati di prova","docNu
mber":"00001","issueDateTime":"2022-02-10","issuer":"alice","mspid":"Org1MSP","owner"
:"andrea","maturityDateTime":"2022-03-15","transferDateTime":"2022-02-12"}
Documento di alice  : 00001 è stato trasferito correttamente
Transazione eseguita correttamente.
Disconnesso dal gateway
Programma transfer completato.
```

Ora, essendo Andrea nuovo proprietario del documento, anche lui potrà effettuare la sua lettura.

```
bash-3.2$ pwd
/Users/michele/test/fabric-samples/projtesi/organization/utente/application
bash-3.2$ node read.js
Connesso al gateway.
Network channel in uso: mychannel.
Inviando la transazione read del documento.
Elaborazione risposta di read.
Documento di alice : 00001 è stato recuperato con successo.
Il contenuto è: Dati di prova
Transazione eseguita correttamente.
Disconnesso dal gateway
Programma read completato.
```

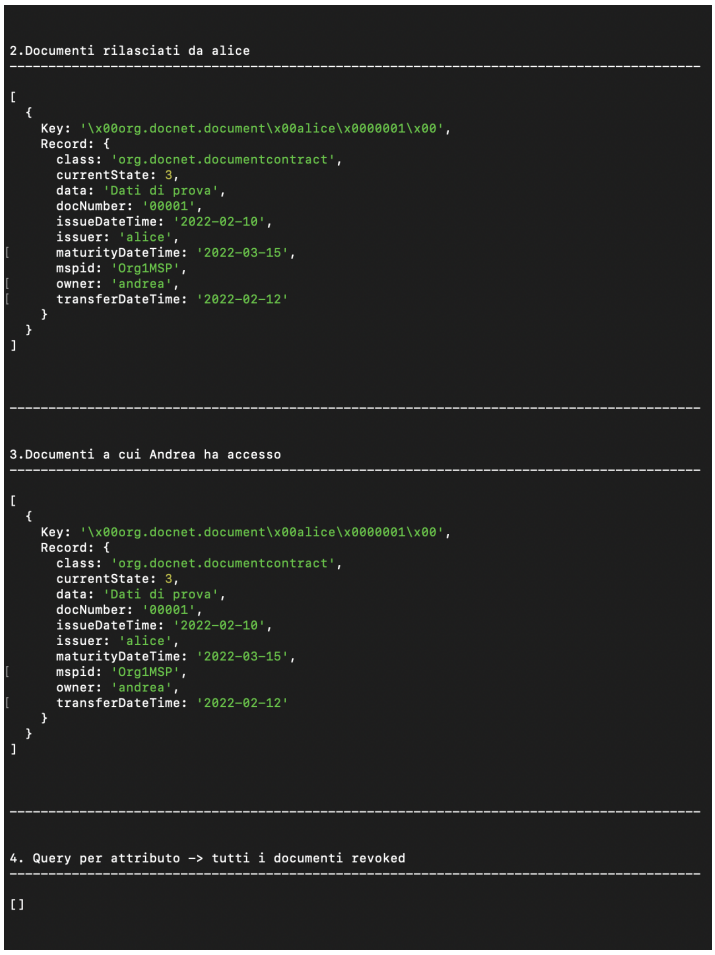
```
bash-3.2$ pwd
/Users/michele/test/fabric/projtesi/organization/Clinica/application
bash-3.2$ node read.js
Connesso al gateway.
Network channel in uso: mychannel.
Inviando la transazione read del documento.
Elaborazione risposta di read.
Documento di alice : 00001 è stato recuperato con successo.
Il contenuto è: Dati di prova
Transazione eseguita correttamente.
Disconnesso dal gateway
Programma read completato.
```

Proviamo a interrogare il ledger per vedere le transazioni registrate e lo stato del documento. La query cronologica mostra che le tre modifiche al ledger sono state registrate correttamente e il documento attualmente è in stato trading.

```
1. Cronologia transazioni....
--------------------------------------------------------------------------------

[
  {
    TxId: '3f9e9ed5eebc609041f4f0cbd64e50587670d6c3920037aa34da68e910ae863b',
    Timestamp: '2022-03-14T09:25:12.349Z',
    Value: {
      class: 'org.docnet.documentcontract',
      currentState: 'TRADING',
      data: 'Dati di prova',
      docNumber: '00001',
      issueDateTime: '2022-02-10',
      issuer: 'alice',
      mspid: 'Org1MSP',
      owner: 'andrea',
      maturityDateTime: '2022-03-15',
      transferDateTime: '2022-02-12'
    }
  },
  {
    TxId: 'ad909e87278e486e19228cb8f23e102b81009586df87365496dc1918e650d203',
    Timestamp: '2022-03-14T09:25:06.912Z',
    Value: {
      class: 'org.docnet.documentcontract',
      currentState: 'PENDING',
      data: 'Dati di prova',
      docNumber: '00001',
      issueDateTime: '2022-02-10',
      issuer: 'alice',
      mspid: 'Org2MSP',
      owner: 'alice',
      maturityDateTime: '2022-03-15'
    }
  },
  {
    TxId: 'c866b2de7686a269e55556ece2c74f2d3e82bff754e0b3c1b52bc47532df3560',
    Timestamp: '2022-03-14T09:20:41.664Z',
    Value: {
      class: 'org.docnet.documentcontract',
      issuer: 'alice',
      docNumber: '00001',
      issueDateTime: '2022-02-10',
      currentState: 'ISSUED',
      mspid: 'Org2MSP',
      owner: 'alice',
      data: 'Dati di prova'
    }
  }
]
```

Le altre query ci mostrano invece i documenti che Alice ha rilasciato, quelli a cui ha accesso Andrea e i documenti in stato revoked.



Terminiamo il transaction flow del documento con la revoca da parte di Alice. Qui si suppone che la data di scadenza del prestito sia già stata raggiunta.

```
bash-3.2$ node revoke.js
Connesso al gateway.
Network channel in uso: mychannel.
Inviando la transazione revoke del documento.
Elaborazione risposta di revoke.
Documento di alice : 00001 è stato ritornato con successo da alice
Transazione eseguita correttamente.
Disconnesso dal gateway
Programma revoke completato.
```

Infine eseguiamo un’altra volta il programma di query per vedere se tutti i cambiamenti sono stati registrati.

Chiaramente, in un caso reale il contenuto del documento non sarebbe mostrato nelle query, ma per lo scopo di questo esempio lo possiamo lasciare visibile.

```
1. Cronologia transazioni....
--------------------------------------------------------------------------------------------

[
  {
    TxId: 'f697721b42d0f33d4123c59a4d611a1a20ede5ea45574844a6228bc4d8f53052',
    Timestamp: '2022-03-14T09:33:11.741Z',
    Value: {
      class: 'org.docnet.documentcontract',
      issuer: 'alice',
      docNumber: '00001',
      issueDateTime: '2022-02-10',
      currentState: 'REVOKED',
      mspid: 'Org2MSP',
      owner: 'alice',
      data: 'Dati di prova',
      revokeDateTime: '2020-11-30'
    }
  },
2.Documenti rilasciati da alice
--------------------------------------------------------------------------------------------

[
  {
    Key: '\x00org.docnet.document\x00alice\x0000001\x00',
    Record: {
      class: 'org.docnet.documentcontract',
      currentState: 4,
      data: 'Dati di prova',
      docNumber: '00001',
      issueDateTime: '2022-02-10',
      issuer: 'alice',
      mspid: 'Org2MSP',
      owner: 'alice',
      revokeDateTime: '2020-11-30'
    }
  }
]



--------------------------------------------------------------------------------------------


3.Documenti a cui Andrea ha accesso
--------------------------------------------------------------------------------------------

[]



--------------------------------------------------------------------------------------------


4. Query per attributo -> tutti i documenti revoked
--------------------------------------------------------------------------------------------

[
  {
    Key: '\x00org.docnet.document\x00alice\x0000001\x00',
    Record: {
      class: 'org.docnet.documentcontract',
      currentState: 4,
      data: 'Dati di prova',
      docNumber: '00001',
      issueDateTime: '2022-02-10',
      issuer: 'alice',
      mspid: 'Org2MSP',
      owner: 'alice',
      revokeDateTime: '2020-11-30'
    }
  }
]
```

6 Commenti conclusivi

In questo elaborato è stato presentato un esempio di come si può adattare la tecnologia blockchain in ambiti reali e problematici dal punto di vista della fiducia, tramite lo strumento *Hyperledger Fabric.*

Lo scopo è stato anche quello di mostrare la trasparenza e l'affidabilità che la tecnologia offre, tramite lo scambio di una risorsa attraverso un programma e la possibilità in qualsiasi momento di vederne tutta la cronologia delle transazioni in cui è stata coinvolta fino a quell'istante.

HyperLedger permette la creazione di blockchain private e *Permissioned*, principalmente utili in un ambito aziendale.

Una blockchain privata può essere creata da una singola azienda per gestire transazioni interne o da un insieme consorziale per la gestione di qualsiasi bene materiale o digitale.

Essendo privata, i partecipanti vengono singolarmente invitati a partecipare e sono conosciuti all'interno della rete. Si possono gestire le autorizzazioni, assegnando particolari diritti di accesso in lettura e scrittura ai partecipanti. Tutte le transazioni avvengono rispettando il chaincode sul canale che, una volta accettato nel momento in cui si partecipa alla rete, garantisce l'imparzialità delle operazioni, aumentando di conseguenza la fiducia reciproca tra le parti.

Il consenso in una blockchain *Permissioned* non è ottenuto tramite il mining, ma attraverso il processo di "approvazione selettiva". La policy di endorsement, definita al momento della creazione del canale, affida ad alcuni membri il compito di verificare le transazioni. Questo è diverso dalle reti di tipo pubblico come Bitcoin, dove, per verificare le transazioni, l'intera rete deve lavorare, generando problemi di scalabilità e di tipo energetico.

La blockchain, pur essendosi ormai affermata, è ancora una tecnologia emergente, che richiede molto impegno per il suo sviluppo e sulla quale c'è ancora una forte ricerca attiva per cercare di migliorarne i suoi aspetti e ampliarne l'utilizzo.

7 Bibliografia / Sitografia

Capitolo 2:

1. https://bitcoin.org/files/bitcoin-paper/bitcoin_it.pdf
2. https://en.wikipedia.org/wiki/Cryptocurrency_wallet
3. https://en.wikipedia.org/wiki/SHA-2
4. https://it.wikipedia.org/wiki/Crittografia_asimmetrica
5. https://en.bitcoin.it/wiki/Mining
6. https://it.wikipedia.org/wiki/Proof-of-work
7. https://it.wikipedia.org/wiki/Funzione_crittografica_di_hash
8. https://it.wikipedia.org/wiki/Nonce

Capitolo 3:

9. https://it.wikipedia.org/wiki/Blockchain
10. https://www.ibm.com/it-it/topics/what-is-blockchain
11. https://medium.com/coinmonks/blockchain-consensus-algorithms-6c2459737a61
12. https://medium.com/blockchainspace/3-comparison-of-bitcoin-ethereum-and-hyperledger-fabric-cd48810e590c
13. https://hyperledger-fabric.readthedocs.io/en/release-2.2/whatis.html#permissioned-vs-permissionless-blockchains
14. https://it.wikipedia.org/wiki/Smart_contract
15. https://hyperledger-fabric.readthedocs.io/en/release/blockchain.html

Capitolo 4:

16. https://it.wikipedia.org/wiki/Hyperledger
17. https://www.ibm.com/it-it/topics/hyperledger
18. https://hyperledger-fabric.readthedocs.io/en/release-2.2/whatis.html
19. https://hyperledger-fabric.readthedocs.io/en/release-1.2/key_concepts.html

Capitolo 5:

20. https://hyperledger-fabric.readthedocs.io/en/release-1.2/tutorials.html