Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Application Specific Blockchain

Come creare sistemi decentralizzati su reti distribuite

Antonio Francesco Gentile, Emilio Greco

RT- ICAR-CS-21-08 **Ottobre 2021**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 8-9C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.icar.cnr.it*
– Sezione di Palermo, Via Ugo La Malfa, 153, 90146 Palermo, URL: *www.icar.cnr.it*

Sommario

Premessa

La prima blockchain fu introdotta nel 2008 ad opera di Satoshi Nakamoto. Ha raggiunto una certa notorietà a livello globale a partire dal 2014 dove la dimensione della sua blockchain "**Bitcoin**" raggiunse i 20 gigabyte. Solo nell'aprile del 2019 è stato presentato il primo manufatto artigianale made in Italy, nel quale sono stati tracciati interamente i passaggi produttivi tramite tecnologia blockchain. L'implicazione della **blockchain nell'Industria 4.0** ha generato una grande quantità di innovazioni che hanno consentito la realizzazione di nuovi modelli di business ottimizzati, flessibili e più efficienti, basati sulla fiducia e la sicurezza di tutte le parti interessate. In tale contesto la tecnologia è di grande aiuto ove i consumatori finali sono sempre più interessati a scoprire l'esatta tracciabilità dei prodotti acquistati ma anche per le istituzioni sempre più severe nei controlli di filiera e dei processi di produzione. Oggi non può che suscitare particolare interesse da parte della comunità scientifica che la annovera tra le tecnologie emergenti più promettenti. Con la blockchain viene per la prima volta definito un ecosistema del tutto decentralizzato, privo di autorità centrale che controlli gli scambi informativi e che possa quindi teoricamente modificarne il contenuto o nascondere parte di esse. L'aspetto straordinario di Bitcoin è, che non è di nessuno: nessuna società, nessuno stato la possiede è semplicemente un programma "**Open Source**" operante su Internet. Contemporaneamente allo sviluppo della tecnologia blockchain, l'IoT **(Internet of Things)** ha trovato campo di applicazione in una quantità enorme di applicazioni industriali, al fine di effettuare il monitoraggio e il controllo da remoto o automatico di sistemi elettronici. La peculiarità dell'IoT sta nell'intuizione di collegare ad internet direttamente i dispositivi che raccolgono dati o che effettuano operazioni di controllo. La commistione delle due tecnologie sopracitate ha preso il nome di **Blockchain of Things (BCoT)**, e rappresenta l'ultima evoluzione dello scambio informativo tra dispositivi IoT, che sono quindi in grado, oltre che di effettuare operazioni sull'ambiente, anche di certificare i dati raccolti in maniera automatica tramite la blockchain e fornirli all'utente finale all'interno di un ledger decentralizzato. Oltre a fornire servizi evoluti di "comunicazione", oggi si indaga sulla possibilità di integrazione tra blockchain ed altri sistemi distribuiti, come ad esempio i sistemi robotici a sciame, per fornire a quest'ultimi le capacità necessarie per realizzare sistemi di controllo decentralizzati di **swarm robotics**, più sicuri, autonomi e flessibili. In tale ambito i dispositivi non comunicano con l'essere umano per richiedere comandi o istruzioni ma sono in grado di comunicare tra loro in maniera autonoma ed effettuare operazioni, anche complesse, prendendo decisioni in autonomia, a volte supportati da algoritmi di Machine Learning o Intelligenza Artificiale. In questo lavoro vengono analizzati alcuni esempi applicativi della tecnologia blockchain applicato al contesto delle Dapp (Decentralized applications), ed in particolar modo sul framework di sviluppo per blockchain Tendermint.

Introduzione

La tecnologia blockchain può fornire soluzioni innovative a quattro problemi noti nel campo di ricerca della swarm robotics come la gestione della sicurezza, la realizzazione di modelli decisionali, la differenziazione del comportamento e lo sviluppo di modelli di business validi.

Uno dei principali ostacoli allo sviluppo di applicazioni commerciali su larga scala per la swarm robotics è **la sicurezza**. La ricerca nel settore ha evidenziato come vi sia la necessità di sviluppare sistemi in cui i membri di uno sciame debbano potersi fidare delle loro controparti per poter raggiungere l'obiettivo. Questo è particolarmente importante, dal momento che è stato dimostrato che l'inclusione di membri "difettosi" nello sciame o elementi che hanno volutamente intenzioni malevoli potrebbero essere un potenziale rischio anche per gli obiettivi che deve raggiungere l'ntero sciame. La sicurezza deve essere quindi garantita in qualsiasi ambiente, sciame compreso, e riguarda fondamentalmente la fornitura di servizi che rispettino: la riservatezza, l'integrità e l'origine dei dati, nonché l'autenticazione dell'entità che li genera. A differenza di altri campi in cui la ricerca in materia di sicurezza viene condotta attivamente, i sistemi robotici a sciame soffrono della mancanza di soluzioni a causa delle caratteristiche complesse ed eterogenee dei sistemi come: l'autonomia del robot, il controllo decentralizzato, un numero elevato degli attori, il comportamento collettivo emergente, ecc. La tecnologia blockchain può quindi fornire non solo un canale di comunicazione peer-to-peer affidabile tra gli agenti dello swarm, ma è anche un modo per superare potenziali minacce, vulnerabilità e attacchi.

Gli **algoritmi decisionali distribuiti** hanno svolto un ruolo cruciale nello sviluppo di sistemi di swarm robotics. Uno degli esempi più importanti è stata la realizzazione della rete ad hoc MANET sviluppata per testare applicazioni di rilevamento distribuito. Questi sistemi hanno la capacità di rilevare le stesse informazioni da più punti e, quindi, di aumentare la qualità dei dati ottenuti. Tuttavia, i robot nello sciame hanno bisogno di raggiungere un accordo globale sull'oggetto di interesse, ad esempio, sui percorsi da attraversare, sulla forma di un oggetto da rilevare o sugli ostacoli da evitare. Pertanto, è necessario sviluppare protocolli decisionali distribuiti che garantiscono la convergenza verso un risultato comune. Gli algoritmi decisionali distribuiti sono stati adottati in molte applicazioni robotiche, tra cui l'allocazione dinamica delle attività, la costruzione di mappe collettive e l'elusione degli ostacoli. Tuttavia la dislocazione di grandi quantità di agenti che fanno uso di un processo decisionale distribuito rappresenta ancora un problema aperto in quando per risolverlo, allo stato attuale è necessario adottare il noto compromesso nel bilanciare velocità di elaborazione ed accuratezza. In questo contesto la blockchain è una tecnologia eccezionale per garantire che tutti i partecipanti di una rete decentralizzata possano condividere una visione identica del mondo. Ogni

volta che un membro dello sciame si trova in una situazione che richiede un accordo, può emettere una transazione speciale, creando un indirizzo associato a ciascuno delle possibili opzioni che lo sciame robotico deve votare. Dopo essere state incluse in un blocco, le informazioni sono disponibili pubblicamente, quindi gli altri membri dello sciame, possono votare in base alla loro situazione, ad esempio, trasferendo un token a l'indirizzo corrispondente all'opzione scelta. Il raggiungimento di un accordo come ad esempio attraverso la regola della maggioranza, può essere ottenuta rapidamente e in un modo sicuro e verificabile poiché tutti i robot possono monitorare gli indirizzi coinvolti nel processo di voto.

Anche se gli algoritmi allo stato dell'arte hanno consentito a team di robot specializzati di **gestire comportamenti collettivi specifici** come aggregazione, raggruppamento, foraggiamento, ecc. ancora non siamo in grado di gestire applicazioni nel mondo reale. Si possono verificare alcuni scenari dove lo sciame deve gestire comportamenti diversi in funzione dell'ambiente, ad esempio, commutando da un algoritmo di controllo all'altro per raggiungere un determinato obiettivo. La combinazione di diversi comportamenti in uno sciame è ancora oggetto di studio in letteratura. In questo caso, la tecnologia blockchain offre la possibilità di collegare diverse blockchain in modo gerarchico, note anche come catene laterali ancorate, che consentirebbero agli agenti di agire in modo diverso a seconda della particolare blockchain in uso, con diversi parametri, con diversità dei miners, permessi, ecc., personalizzando per l'appunto i diversi comportamenti dello sciame.

Il termine blockchain viene associato generalmente ad applicazione per gestire una valuta, o per essere più precisi una "cripto valuta", ma è grazie al lavoro di Vitalik Buterin, fondatore di Ethereum, che nel 2014 si dà il via ad una blockchain di seconda generazione introducendo gli **smart contracts.** Ethereum ha dato il via a numerosi progetti, oggi si parla della quinta generazione della tecnologia. La più recente implementazione della Blockchain riguarda il così detto **Web 3.0** noto anche come **cryptointernet**. Il Web 3.0 nasce come contraltare allo strapotere GAFA (Google, Apple, Facebook, Amazon) i cui modelli di business si basano sulla centralizzazione dei dati e del conseguente potere decisionale. In questa eccezione, la tecnologia blockchain, può essere vista come un'**Interfaccia di programmazione per una applicazione (API)**, ideale sicuramente per applicazioni economiche, ma anche come framework per consentire a sciami di robot di accedere direttamente e partecipare ad un'economia. Per questo motivo la tecnologia blockchain ha il potenziale per stimolare l'uso della robotica a sciame in applicazioni nell'ambito industriale e di mercato. Una delle implementazioni prototipiche più ovvie per quanto riguarda l'uso di sciami robotici in applicazioni economiche è il processo di scambio dati in cambio di valuta tra un robot ed un richiedente. Questo nuovo modello di business emergente in campo dell'Internet delle cose (IoT) prende il nome di **Sensing-as-a-Service**.

SaaS aiuta a creare mercati multiformi per i dati prodotti dai sensori in cui uno o più clienti, il lato acquirente dei mercati, si sottoscrivere per pagare i dati forniti da uno o più sensori, lato vendita.

Anche se la combinazione della tecnologia blockchain e la robotica a sciame può fornire soluzioni utili per affrontare in maniera efficace diversi problemi, occorre ancora lavorare per risolvere diverse sfide tecniche legate alla blockchain al fine di aumentarne l'efficienza.

La latenza è il principale problema su cui si sta lavorando, attualmente con la versione più utilizzata di blockchain, Bitcoin, un blocco impiega circa 10 minuti per essere elaborato. Ciò significa anche che ogni singola transazione richiede circa 10 minuti per essere confermata. Anche se questo problema può essere notevolmente ridotto attraverso l'uso di blockchain private e/o tramite l'utilizzo di diverse politiche di mining, come il proof-of-stake, questo consentirebbe di ottenere prestazioni accettabili **solo per applicazioni peer-to-peer che richiedono una bassa iterazione** con gli utenti finali. Il problema della latenza diventa notevolmente rilevante quando interessa invece applicazioni di swarm robotics, in questo caso, sono necessarie informazioni rapide e affidabili per orchestrare i movimenti dello sciame. Potrebbero sorgere collisioni o altri inconvenienti in situazioni in cui c'è una discrepanza tra lo stato attuale dell'ambiente e quello invece che è stato rilevato o attuato da una transazione. Una possibile soluzione per mitigare questo problema potrebbe essere la creazione basata sull'affiliazione di sistemi robotici appartenenti alla stessa organizzazione in modo che gli elementi appartenenti ad un gruppo non sono tenuti ad aspettare lunghi periodi di tempo per accettare o elaborare transazioni che riguardano l'intero sistema. Potrebbe essere costruito un sistema di reputazione basato su elenchi di precedenti transazioni accettate all'interno del gruppo per ridurre questi tempi di attesa. Altre soluzioni fanno uso della crittografia per stabilire collegamenti off-chain tra i due nodi peer, utilizzando la chiave pubblica del richiedente ed incapsulamento nel campo dati di una transazione. La comunicazione off-chain in questo caso previene la congestione della blockchain e garantisce che solo il richiedente può leggere il messaggio previsto. La blockchain verrà usata solo per finalizzare un accordo e non per lo scambio di dati tra i due contraenti.

Un secondo problema da affrontare nell'uso della tecnologia per la swarm robotics è legato alle **dimensioni, throughput e larghezza di banda**. Se grandi quantità di robot dovessero essere impiegate per un lungo periodi di tempo, potrebbero espandere la blockchain al punto tale da diventare troppo grande per poterne conservare una copia sui singoli nodi. Questo problema, che la comunità Bitcoin chiama "**bloat**", è di particolare rilevanza nella robotica a sciame dove i singoli robot hanno capacità hardware limitate.

In questo primo lavoro cercheremo di indagare se la tecnologia disponibile allo stato dell'arte riesce a rispondere alle sfide che sono state descritte, mettendo in luce i lavori che si stanno conducendo in tale senso.

Nel primo capitolo descriveremo in modo più dettagliato una blockchain, le piattaforme attualmente più utilizzare e le tecnologie abilitanti. Nella seconda parte analizzeremo alcuni esempi di Blockchain e di Dapp realizzate attraverso il framework Tendermint.

Blockchain

La Blockchain costituisce uno degli sviluppi tecnologici in ambito Internet of Things (IoT). L'Internet of Things, anche chiamato Internet of Everything, prevede la costituzione di una rete globale di macchine e dispositivi capaci di interagire autonomamente; questa rete rappresenta un sistema interconnesso che permette lo scambio di informazioni tra nodi. È definita come un'applicazione decentralizzata dell'IoT, dall'inglese Decentralized applications (DApp). La tecnologia Blockchain permette la creazione, il coordinamento e la sincronizzazione di un complesso database distribuito, costituito da blocchi contenenti le transazioni avvenute tra i nodi di una rete. Questa è rappresentabile idealmente come una catena formata da blocchi contenenti gli eventi della rete. Le transazioni presenti nei blocchi devono essere validate dai nodi che compongono la rete, ciò dà luogo ad una rete di fiducia distribuita.

In altri termini, la Blockchain è un sistema di archiviazione dati sicuro, distribuito, ed immutabile condiviso tra una rete di attori. I dati vengono immagazzinati in "blocchi" (block), connessi l'uno all'altro in una catena (chain) tramite un hash, ovvero una funzione che converte caratteri alfanumerici in una nuova sequenza criptata e di lunghezza predeterminata.

Questi blocchi possiedono una "testa", che include metadati, e un corpo, che invece riguarda i dettagli dei dati veri e propri. Dato che ogni blocco è connesso al precedente e al successivo e distribuito tra tutti i partecipanti, al crescere del numero di attori nella rete diventa esponenzialmente più complesso modificare qualsiasi informazione. Esistono diversi tipi di Blockchain categorizzate a seconda del differente permesso di accesso. In altre parole, alcune Blockchain possono essere rese liberamente accessibili ("public" vs "private") e la capacità di scrivere sul registro illimitata o controllata ("permissionless" vs "permissioned"), ma esistono anche modelli più ibridi (come la Blockchain consortile).

Partita come un semplice modello per certificare i dati di un file digitale, nel tempo la Blockchain ha avuto numerose implementazioni. La prima e più aderente al modello originario riferisce al puro sistema di archiviazione dati. Garantendo l'immodificabilità dei dati registrati, la Blockchain viene utilizzata come strumento di verifica delle informazioni e impiegata nei più disparati sistemi di certificazione dei dati: a puro titolo esemplificativo come strumento di certificazione alimentare, registrazioni contratti e registro di proprietà di beni.

Tuttavia, il più noto utilizzo della tecnologia dopo il white paper di Satoshi Nakamoto del 2008 riferisce al suo impiego come tecnologia alla base della criptovaluta Bitcoin. Sfruttando la sicurezza dei dati la Blockchain consente di sviluppare un sistema di pagamento che funziona in assenza di una autorità centrale. Mentre infatti nei sistemi di e-money tradizionali quali le transazioni bancarie, il presupposto della correttezza dei dati deriva dalla fiducia nei sistemi centrali, nelle criptovalute la fiducia viene riposta nel sistema di archiviazione. Questo approccio noto come "zero knowledge proof space" fa in modo che soggetti che non hanno conoscenza reciproca possano tranquillamente scambiarsi criptovalute.

Il terzo impiego della Blockchain risale a innovazioni avvenute verso la metà degli anni '90 con l'introduzione degli Smart Contract. In sostanza uno Smart Contract altro non è che un programma per computer che registra e finalizza un accordo. Sebbene i suoi utilizzi siano tutt'altro che recenti (si pensi ai banali antivirus che rinnovano automaticamente la licenza allo scadere dei termini), la loro diffusione diviene massiccia grazie al lavoro di Vitalik Buterin, fondatore di Ethereum. Inseriti in un sistema di Blockchain, gli Smart Contract consentono la certificazione dei termini del contratto e la loro esecuzione automatica senza che via sia una terza parte coinvolta.

I problemi che le blockchain di terza generazione stanno cercando di risolvere sono legati alla scalabilità, in particolare attraverso la creazione di molteplici layer (tendenza che ha portato anche alla nascita di Lightning Network, layer di secondo livello per transazioni istantanee su Bitcoin), all'interoperabilità tra blockchain diverse e **allo sviluppo di tecnologie ad hoc per la realizzazione di applicazioni blockchain M2M (machine to machine) in ottica Internet of Things**.

La quarta applicazione della Blockchain si collega ai cosiddetti "Initial Coin Offering", innovativa modalità di crowdfunding basata sulla possibilità di creare nuovi modelli di business basati sulla tokenizzazione dei diritti. Il mercato complessivo delle ICO ha superato a settembre 2019 i 15 miliardi di euro.

La quinta e più recente implementazione della Blockchain riguarda il così detto **Web 3.0** noto anche come cryptointernet. Il Web 3.0 nasce come contraltare allo strapotere GAFA (Google, Apple,

Facebook, Amazon) i cui modelli di business si basano sulla centralizzazione dei dati e del conseguente potere decisionale. Lavorando su modelli distribuiti, il Web 3.0 consente lo sviluppo di nuovi modelli di business capaci di sfruttare le risorse inutilizzate. Gli esempi non mancano. La startup Golem, che l'anno scorso ha sfiorato una capitalizzazione sul mercato delle ICO di quasi un miliardo di dollari, offre un modello distribuito alla capacità di calcolo che si contrappone ai modelli centralizzati di Amazon AWS. Riconoscendone le possibili implementazioni, in molti concordano che la Blockchain scatenerà la digital disruption di tutti i settori, facendo diventare distribuiti i modelli di business dominanti, creando valore da nuovi asset, riducendo i costi delle transazioni e incrementando la fiducia degli stakeholders. I primi settori a essere trasformati saranno quelli della finanza, dell'agroalimentare, della sanità, della moda, dello sport e intrattenimento, dei servizi professionali, della distribuzione e manifattura. La Blockchain pone anche molte sfide normative, in primis in merito alla tutela dei risparmiatori, e ambientali, in merito al consumo di energia richiesta dalla necessità di duplicare l'archiviazione dei dati.

Golem si appoggia all'algoritmo di consenso PoS (Proof of Stake), per cui i GNT non possono essere minati. **Il progetto si basa sul distributed computing, ovvero sul concetto di calcolo distribuito**. Esso non è altro che un sistema in cui tanti computer comunicano fra loro, pur essendo indipendenti. Chiunque può utilizzare Golem per far girare diversi tipi di software, in diversi settori tra i quali computer grafica, business, machine learning, crittografia, ecc… Cos'è che si condivide tramite questo network decentralizzato su blockchain Ethereum? La potenza di calcolo del proprio pc. Essa viene ceduta dagli utenti al network stesso. In poche parole si va a prestare una parte della potenza fornita dal PC per essere pagati con la cripto valuta Golem. Si può usare Golem per fare numerose attività, tra cui le predizioni di mercato, inoltre è consentito sviluppare e vendere propri software sul network di Golem. Il network si basa sulla rete peer-to-peer ed usa un linguaggio open source.

In breve possiamo dire che l'infrastruttura elementare di una blockchain è costituita da:

- Database distribuito
- Meccanismo di consenso
- Token come premio di convalida.

Il funzionamento della blockchain include inoltre ulteriori componenti come:

- Nodo
- Transazione
- Blocco

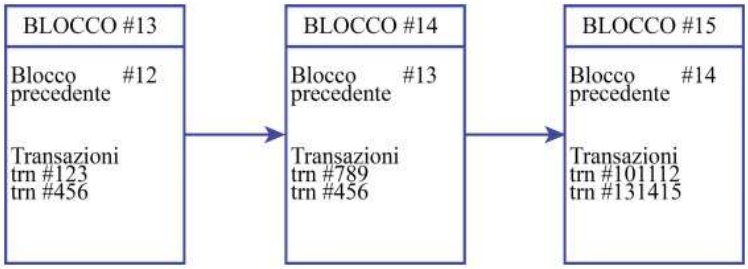
• Ledger (registro)

• Hash

• Miners

Riepilogando, ogni blocco della catena può contenere un certo numero di transazioni. Le transazioni riguardano lo scambio di risorse digitali, ed utilizzano una rete peer-to-peer che memorizza queste transazioni in maniera distribuita attraverso la rete.

Gli attori proprietari dei beni digitali e le transazioni che comportano un cambio di proprietà sono registrati all'interno del blocco mediante l'utilizzo della crittografia a chiave pubblica/ privata e delle firme digitali che garantiscono sicurezza e autenticità allo scambio. Ogni blocco possiede un valore di **hash** identificativo. L'hash è in grado di mappare una stringa numerica o di testo, in una stringa unica ed univoca di lunghezza determinata. In questa maniera grazie all'hash, si è in grado di identificare in maniera univoca e sicura ciascun blocco. L'hash è strutturato in modo da impedire la rievocazione del testo o la stringa numerica da cui esso è stato generato. Inoltre ogni blocco oltre ad avere il proprio hash identificativo contiene anche l'hash del blocco che lo precede. In questa maniera, quando un nuovo blocco viene aggiunto alla catena di blocchi, questo mantiene una visione condivisa e concordata dello stato attuale della blockchain. Un esempio è presentato nella figura successiva.



Il **ledger** (registro) contiene lo stato condiviso e concordato della catena di blocchi e l'elenco di tutte le transazioni che sono state elaborate. In tale maniera, tutti i nodi che partecipano alla rete di questo sistema decentralizzato avranno una copia dell'intera catena di blocchi che viene continuamente aggiornata e sincronizzata tra tutti i nodi della rete. Questo aspetto è fondamentale per la tecnologia blockchain, perché in questa maniera non esiste un punto centrale di vulnerabilità che gli hacker possono sfruttare, come invece può accadere per i database centralizzati.

Nel caso in cui qualcuno fosse intenzionato a modificare qualche transazione all'interno di un blocco, questo modificherebbe il valore hash identificativo e quindi affinché l'attacco possa andare a buon fine, la modifica deve essere replicata a sua volta su tutti i nodi della rete. Questa operazione

richiederebbe una potenza di calcolo enorme che, con le tecnologie attualmente esistenti, risulterebbe impossibile.

L'architettura peer-to-peer contribuisce alla sicurezza e all'immutabilità delle transazioni e dei blocchi registrati nella blockchain. Colui che convalida le transazioni all'interno di un blocco e aggiunge il blocco alla catena prende il nome di **miner**. Il miner convalida il blocco attraverso un meccanismo di consenso, che corrisponde alla risoluzione di un complesso problema matematico. Nel caso specifico della blockchain Bitcoin questo sforzo computazionale che comporta un importante consumo di energia elettrica, prende il nome di "**proof of work**". L'intera sicurezza e validità della catena è garantita proprio dal lavoro dei miners. Il ruolo del miner è fondamentale per il corretto funzionamento della blockchain. Il miner è un utente volontario che partecipa liberamente all'interno della rete mettendo a disposizione la propria CPU (Central Processing Unit) per risolvere questi problemi matematici che permettono di convalidare i blocchi. In breve, il compito del miner è quello di raggruppare e verificare le transazioni che ancora non sono state inserite all'interno di un blocco e dopo averle verificate, provare a risolvere il problema computazionale svolgendo tale proof of work.

Quando è stata trovata la soluzione a tale problema, il blocco viene trasmesso alla rete e dopodiché avviene l'aggiornamento della catena per tutti i nodi della rete. Un aspetto molto importante è che il miner che risolve e quindi conseguentemente convalida il blocco, deve aggiungere il nuovo blocco alla catena più lunga esistente, questo è un passaggio fondamentale per la salvaguardia dell'intera piattaforma. Nella blockchain Bitcoin, un nuovo blocco viene aggiunto alla catena dopo circa 10 minuti. È importante sottolineare come oltre alla proof of work esistono anche altri meccanismi di consenso come ad esempio la **proof of stake**. La proof of stake semplifica il processo relativo al mining descritto precedentemente. Per quanto riguarda la proof of stake, il lavoro richiesto per eseguire il processo di verifica viene ripartito tra i singoli membri in base alla loro percentuale di partecipazione. Ad esempio, se un utente possiede il 20% del totale delle attività di blockchain in circolazione, l'utente dovrà eseguire il 20% dell'attività di mining richiesta. In questa maniera si riduce la complessità del processo di verifica decentralizzata e si possono quindi generare anche dei risparmi relativi ai costi energetici e operativi (Hasse et al., 2016).

Siccome il compito del miner è di assoluta importanza per mantenere la sicurezza dell'intera catena di blocchi, il suo sforzo viene remunerato attraverso un **token**. L'incentivo che viene dato ai miners attraverso il token per la risoluzione del problema è la chiave principale affinché l'intero sistema sia completamente affidabile. Nella blockchain Bitcoin, il miner per il suo sforzo ottiene appunto dei Bitcoin. Nel 2009 per ogni blocco validato il miner riceveva in cambio 50 Bitcoin. Tale valore, per

come è stato strutturato l'algoritmo, viene dimezzato ogni quattro anni; ad oggi per ogni blocco validato il sistema riconosce come compenso 12,5 Bitcoin. Ovviamente questo vale esclusivamente per la blockchain Bitcoin. Infatti ogni blockchain si struttura intorno ad un algoritmo che avrà regole differenti che dipendono dalle logiche di programmazione e degli obiettivi e funzionalità offerte dal sistema stesso.

Il token in una blockchain pubblica ricopre un ruolo molto importante. **Esso può essere identificato come un insieme di informazioni digitali capace di attribuire il diritto di proprietà ad un determinato soggetto**. Il token consiste in un insieme di informazioni registrate sulla blockchain che attraverso un protocollo possono essere trasferite. Il token più "famoso" è appunto il Bitcoin ma, dopo di esso, ne sono comparsi molti altri. A tal proposito un esempio è l'Ether che è il token appartenente alla blockchain pubblica Ethereum. Al momento possiamo distinguere tre tipologie di token:

- i token di classe 1 che rappresentano una vera e propria moneta e che tramite la blockchain possono essere trasferiti (es: Bitcoin);
- token di classe 2 che permettono di esercitare alcuni diritti verso una controparte;
- i token di classe 3 che hanno un ruolo misto, ovvero che raffigurano diritti di comproprietà ed alla stessa maniera attribuiscono diritti diversi come per esempio il diritto di voto.

Inizialmente si è parlato di **chiavi crittografiche**: vediamone ora il funzionamento in relazione alla blockchain Bitcoin. Nello specifico il sistema Bitcoin si basa su due tecnologie crittografiche: crittografia a chiave pubblico-privata e la crittografia per le transazioni di rete. Come spiegato precedentemente ad ogni transazione è correlata una firma digitale che è diversa per ogni transazione. La tecnologia che permette tutto questo è la crittografia a chiave pubblico-privata, che permette con la chiave privata di creare una "firma" associata ad una chiave pubblica. La chiave pubblica è condivisa nella rete, mentre la chiave privata è personale ed è utilizzata per de-crittografare i dati. Inoltre di fondamentale importanza è la "**crittografia ellittica**" che praticamente permette di calcolare la chiave pubblica data la chiave privata ma non permette il contrario. In questa maniera tutti gli utenti che partecipano alla rete sono identificabili attraverso la loro chiave pubblica. Per tale motivo nessun ulteriore dato personale è disponibile all'interno della rete. Tutto questo permette l'anonimato degli utenti o meglio, come sostengono alcuni autori (Swan, 2015) che le transazioni non siano realmente anonime ma "pseudo anonime".

In generale si può riassumere che ci sono tre tipologie di blockchain:

• **Blockchain pubblica**. La blockchain pubblica è una blockchain nella quale chiunque può diventare un nodo della rete, chiunque può leggere e inviare transazioni che poi saranno

successivamente incluse e validate in un blocco, chiunque può essere un miner e per tale motivo partecipare al meccanismo di consenso offrendo volontariamente la propria potenza di calcolo. Come già definito in precedenza, la blockchain pubblica è sicura grazie alla presenza di un incentivo economico che ripaga lo sforzo compiuto dai miner, grazie alla crittografia e dal principio secondo il quale il grado di influenza di un singolo attore all'interno della rete nel processo di consenso, è proporzionale alla quantità di risorse economiche che può apportare. Questa tipologia di blockchain è definita "completamente decentralizzata".

• **Blockchain privata**. La blockchain privata è una blockchain in cui le autorizzazioni di scrittura all'interno dei blocchi sono mantenute completamente centralizzate. Per quanto riguarda invece le autorizzazioni di lettura della blockchain, queste possono essere pubbliche o anch'esse limitate ad un numero finito di utenti. In questa maniera una blockchain privata è sicuramente più vicina ai modelli di business più tradizionali, nonostante questo non debba necessariamente essere visto come un aspetto negativo. Il fatto che questa tipologia di infrastruttura, almeno a prima vista, non abbia lo stesso impatto rivoluzionario della blockchain pubblica, non significa che non possa comunque svolgere un ruolo preponderante nel processo di efficientamento di un'attività di business.

• **Permissioned Blockchain (Consortium)**. La permissioned blockchain a differenza delle precedenti è una blockchain in cui il meccanismo di consenso è controllato da un insieme di nodi preselezionati. Si pensi ad un "consorzio" di 10 istituti finanziari, ognuno dei quali gestisce un nodo. In questo caso è sufficiente che 8 di loro firmino un blocco affinché il blocco sia valido. A tal proposito il diritto di leggere la blockchain può essere sia pubblico che limitato ad alcuni partecipanti. Questa tipologia di blockchain è definita "parzialmente decentrata".

A prima vista potrebbe non essere molto chiara la differenza tra una blockchain privata ed una permissioned, per quanto riguarda la blockchain, essa è fondamentalmente un "ibrido" tra la "bassa fiducia" (minor controllo) che fornirebbe la blockchain pubblica e "la singola entità altamente affidabile" che invece contraddistingue la blockchain privata. La blockchain privata può essere definita come un sistema centralizzato tradizionale con l'aggiunta di un grado di verificabilità crittografica.

Per quanto riguarda le blockchain private si possono individuare tali vantaggi:

• Sia che si tratti di un consorzio o di una blockchain completamente privata, nel caso in cui fosse necessario, sarebbe possibile in maniera più semplice modificare le regole della piattaforma blockchain, o per esempio ripristinare delle transazioni.

• Chi svolge il ruolo del miner è noto a priori e gode di una fiducia pregressa.

• Le transazioni sono convalidate in maniera più veloce, perché solo alcuni nodi hanno questo ruolo.

• I possibili errori possono essere risolti in breve tempo attraverso un intervento manuale.

• Se i permessi di lettura all'interno di una blockchain privata sono limitati, si ottiene una privacy maggiore sui dati.

Alla luce di queste considerazioni, può sembrare che in realtà le blockchain private siano la scelta migliore per una istituzione, come può essere la pubblica amministrazione. In realtà, anche in un contesto come quello del settore pubblico, la blockchain pubblica ha sempre il suo grande valore e questo valore risiede soprattutto nelle virtù filosofiche dei suoi principali sostenitori che promuovono la libertà, la neutralità e l'apertura.

A tal proposito i vantaggi di una blockchain pubblica possono essere suddivisi in due grandi categorie:

• La blockchain pubblica fornisce un modello di protezione riguardante gli utenti di una certa applicazione dagli stessi sviluppatori di quell'applicazione. In questa maniera ci sono alcune cose che neanche gli stessi sviluppatori possono essere in grado di fare. Questo meccanismo permette di rendere molto difficile se non impossibile la possibilità di effettuare dei cambiamenti alla catena, garantendo una maggiore fiducia degli utenti nei confronti del sistema.

• La blockchain pubblica è aperta, immutabile e può essere utilizzata da tutti e letta da tutti ma allo stesso tempo garantisce agli utenti l'anonimato (o lo pseudo-anonimato) all'interno della rete. Questo crea un effetto rete all'interno della piattaforma che rende la blockchain sempre più sicura e protetta. Di contro a questa estrema sicurezza vi è la lentezza nella validazione nei blocchi (in Bitcoin ogni 10 minuti) e il dispendioso spreco energetico dovuto al lavoro compiuto dai miners.

Alla luce di questa analisi è facile capire che la situazione ottimale, in termini di implementazione della tecnologia, varia da settore a settore. In alcuni casi l'implementazione di una blockchain pubblica può essere chiaramente la scelta ottimale, in altri casi invece, il maggior controllo dato dalla blockchain privata la rende necessaria per un certo sistema. Si pensi per esempio al settore pubblico,

dove la fiducia nel sistema può essere pregressa. Per tale motivo una blockchain privata (o permissioned) potrebbe essere preferita rispetto ad una pubblica. Per tutte queste ragioni la risposta riguardante la scelta migliore tra le due è ovviamente: dipende.

Tendermint

Tendermint fornisce una infrastruttura software che consente agli sviluppatori di realizzare nuove soluzioni blockchain. Fondamentalmente Tendermint è un motore di consenso ad alte prestazioni e scalabile, dove la logica dell'applicazione è tenuta separata dalla logica del consenso. La separazione netta tra questi tue elementi del sistema, consente di iniettare una logica personalizzata nelle applicazioni blockchain, **andando ben oltre il concetto tradizionale degli Smart Contract**. Esse possono usare **qualsiasi tipo di software aziendale** per gestire scenari applicativi complessi, ed ha dato il via ad innumerevoli progetti, come quelli descritti in precedenza, consentendo lo sviluppo delle application-specific blockchain.

Le **application-specific blockchain** sono blockchain personalizzate realizzate ad oh per far funzionare una singola applicazione, invece di creare un'applicazione decentralizzata su una blockchain sottostante comune come ad esempio Ethereum basata su macchine virtuali in grado di interpretare programmi completi chiamati Smart Contracts. Questi **Smart Contract sono molto utili per casi d'uso come eventi una tantum** (ad es. ICO), ma possono non essere all'altezza della creazione di piattaforme decentralizzate complesse.

Gli Smart Contract sono generalmente sviluppati con linguaggi di programmazione specifici che possono essere interpretati dalla macchina virtuale sottostante. Questi linguaggi di programmazione sono spesso immaturi e intrinsecamente limitati dai vincoli della macchina virtuale stessa. Ad esempio**, la macchina virtuale Ethereum non consente agli sviluppatori di implementare l'esecuzione automatica del codice**. Gli sviluppatori sono anche limitati al sistema basato su account dell'EVM e possono scegliere solo da un insieme limitato di funzioni per le loro operazioni crittografiche. Questi sono esempi, ma suggeriscono la mancanza di flessibilità che spesso comporta un ambiente basato su Smart Contract. Gli Smart Contract sono tutti gestiti dalla stessa macchina virtuale, ciò significa che competono per le stesse risorse, il che può limitare gravemente le prestazioni. Anche se la macchina a stati dovesse essere suddivisa in più sottoinsiemi, gli Smart Contract dovrebbero comunque essere interpretati da una macchina virtuale, il che limiterebbe le prestazioni rispetto a un'applicazione nativa implementata a livello di macchina a stati. Un altro problema, derivante dalla condivisione dello stesso ambiente sottostante, è la conseguente limitazione

sul controllo. Un'applicazione decentralizzata è un ecosistema che coinvolge più attori. Se l'applicazione è costruita su una blockchain di macchine virtuali di uso generale, le parti interessate hanno un controllo molto limitato sulla loro applicazione, fortemente influenzata dalla governance della blockchain sottostante. Se c'è un bug nell'applicazione, si può fare ben poco.

Una blockchain basata su Tendermint permette la replica coerente e sicura e di un'applicazione su macchine diverse. Sicura perché l'applicazione continua a funzionare anche se 1/3 delle macchine si guasta arbitrariamente (capacità nota come tolleranza d'errore bizantina-BTF) e coerente perché ogni macchina vede le stesse transazioni e si trovano nello stesso stato.

L'algoritmo per il consenso ad alte prestazioni su cui è realizzata l'infrastruttura è noto come PBFT, ovvero Practical Byzantine Fault Tolerance.

L'algoritmo di consenso BFT è uno dei più vecchi algoritmi di consenso. Apparso per la prima volta nel 1999, prende il nome dal Problema dei Generali Bizantini in cui alcuni Generali non sapevano se attaccare o meno a causa di informazioni discordanti ricevute dal comandante e dagli altri Generali. Il problema sottendente che alcuni Generali erano traditori e quindi le informazioni che trasmettevano erano falsate. La soluzione del problema è stata affidata ad un alto numero di messaggi tra i partecipanti. L'ordine corretto è quello alla cui versione aderisce la maggioranza.

Applicato alle reti distribuite come la blockchain il Practical Byzantine Fault Tolerance consocia i nodi in reti minori. All'interno di ciascun gruppo, la maggioranza dei nodi vota il nodo leader che sulla base delle informazioni ottenute dai nodi consociati verifica i blocchi. L'obiettivo del sistema è di impedire a nodi malevoli di partecipare alla creazione dei blocchi, rendendo la blockchain meno esposta ad attacchi.

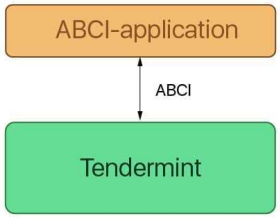
Tendermint offre prestazioni eccezionali, il consenso di Tendermint può elaborare migliaia di transazioni al secondo, con latenze di commit dell'ordine di uno o due secondi. In particolare, le prestazioni di oltre un migliaio di transazioni al secondo vengono mantenute anche in condizioni conflittuali difficili, con i validatori che si bloccano o trasmettono voti fraudolenti.

I principali vantaggi sono:

- Può gestire il volume delle transazioni alla velocità di 10.000 transazioni al secondo per transazioni fino a 250 byte.
- Sicurezza migliore e più semplice per il client che diventa più leggero rendendolo ideale per dispositivi mobili e IoT. Al contrario, i client leggeri Bitcoin richiedono molto più lavoro e hanno molte richieste che lo rendono poco pratico per determinati casi d'uso.

- Tendermint utilizza una fork-accountability che blocca gli attacchi come long-range-nothing-at-stake double spends e gli censorship.
- Tendermint è implementato tramite un "motore di consenso indipendente dall'applicazione". Fondamentalmente può trasformare qualsiasi applicazione blackbox deterministica in una blockchain replicata in modo distribuito.

Tendermint è divisibile in due parti principali: Tendermint Core, ossia la parte che gestisce il "motore" della blockchain, e l'Application Blockchain Interface (ABCI), che permette alle transazioni di essere gestite da una logica applicativa scritta in qualunque linguaggio di programmazione. La parte core di Tendermint garantisce, invece, che le transazioni vengano memorizzate su ogni macchina nello stesso ordine.



Progetti diversi hanno esigenze diverse. Alcuni progetti devono avere un sistema aperto in cui chiunque può partecipare e contribuire, come Ethereum. D'altra parte, abbiamo organizzazioni come l'industria medica, che non possono esporre i propri dati a tutti, allora come può Tendermint aiutare a soddisfare entrambe queste esigenze?

Tendermint rappresenta soltanto il primo strato dell'applicazione, cioè quella delegata a gestire solo il networking ed il meccanismo di consenso per la blockchain.

Quindi, si occupa di:

- Propagazione della transazione tra i nodi tramite il protocollo gossip
- Aiuta i validatori a concordare l'insieme di transazioni che vengono aggiunte alla blockchain.

Ciò significa che il livello dell'applicazione il programmatore è libero di definire come viene gestito il set di validatori all'interno dell'ecosistema. Gli sviluppatori possono consentire all'applicazione di avere un sistema elettorale che elegge i validatori in base a ai loro token nativi creando quella che viene definita **Proof-of-stake** per una blockchain pubblica. Inoltre, gli sviluppatori possono creare un'applicazione che definisce un insieme ristretto di validatori pre-approvati che si occupano del consenso per i nuovi nodi che entrano nell'ecosistema. Questa è chiamato **proof-of-authority** ed è il meccanismo di consenso distintivo di una blockchain autorizzata o privata.

L'implementazione del proof-of-stake di Tendermint è molto più scalabile di un tradizionale algoritmo di consenso proof-of-work. Il motivo principale è che i sistemi basati su POW non possono eseguire lo sharding. Lo sharding fondamentalmente partiziona orizzontalmente un database e crea database o frammenti più piccoli che vengono quindi eseguiti in parallelo dai nodi.

Tendermint mette a disposizione dei nodi validatori (validators), identificati dalla loro chiave pubblica, e ogni nodo è responsabile del mantenimento di una copia integrale dello stato del sistema, della proposta di nuovi blocchi e del voto per validare i suddetti blocchi. A ogni blocco viene assegnato un indice incrementale (height), così facendo si avrà un blocco valido per ogni height. Ogni blocco viene proposto da un nodo diverso ogni volta (il nodo viene detto proposer), dividendo il processo di consenso in veri e propri round. Il processo di consenso può essere diviso in 3 fasi:

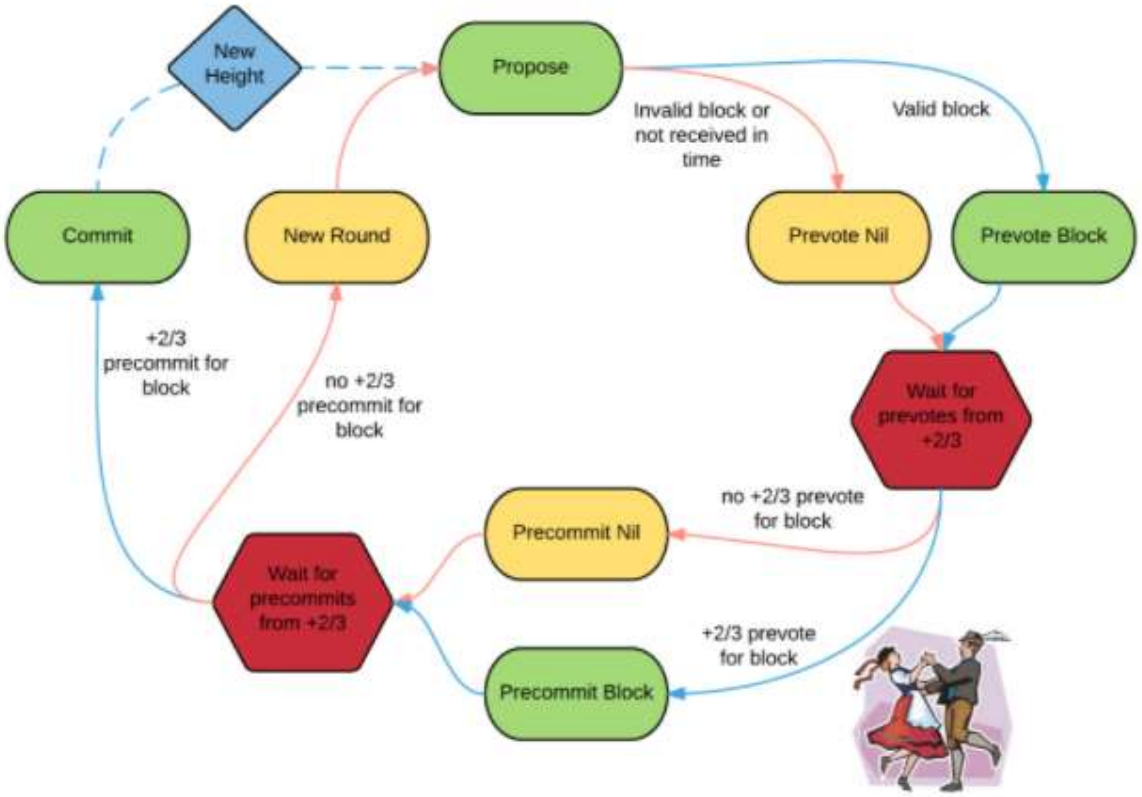
• Proposta (proposal): il proposer di turno propone un nuovo blocco e gli altri validatori lo ricevono. Se non lo ricevono entro un determinato periodo di tempo si passa al proposer successivo;

• Votazione (votes): la fase di votazione si suddivide anch'essa in due sotto parti, ossia pre-vote e pre-commit.

• Lock: Tendermint si assicura che nessun validatore inserisca più di un blocco a un dato indice (height).

Ogni round inizia con una nuova proposta. Il nodo che effettua la proposta prende le transazioni presenti all'interno della sua cache, chiamata Mempool, assembla il blocco e lo spedisce sulla rete tramite un messaggio firmato (ProposalMsg). Una volta che la proposta viene ricevuta da un nodo validatore, quest'ultimo firma un messaggio per il pre-vote di quella proposta e lo invia a tutta la rete. Se un validatore non riceve una proposta entro un determinato periodo di tempo (ProposalTimeout), il suo voto verrà considerato come nullo. Se almeno i 2/3 della rete hanno votato a favore del blocco, si passa a un'altra votazione, ossia quella per la pre-commit. In sintesi, la votazione di pre-vote prepara la rete a ricevere un nuovo blocco da inserire alla blockchain. Se la rete è pronta a ricevere questo nuovo blocco, ossia è stato votato dai 2/3 della rete, allora si passa alla votazione per il pre-commit e se un validatore riceve voti da almeno i 2/3 dei nodi, il blocco viene aggiunto alla blockchain e viene computato il nuovo stato del sistema.

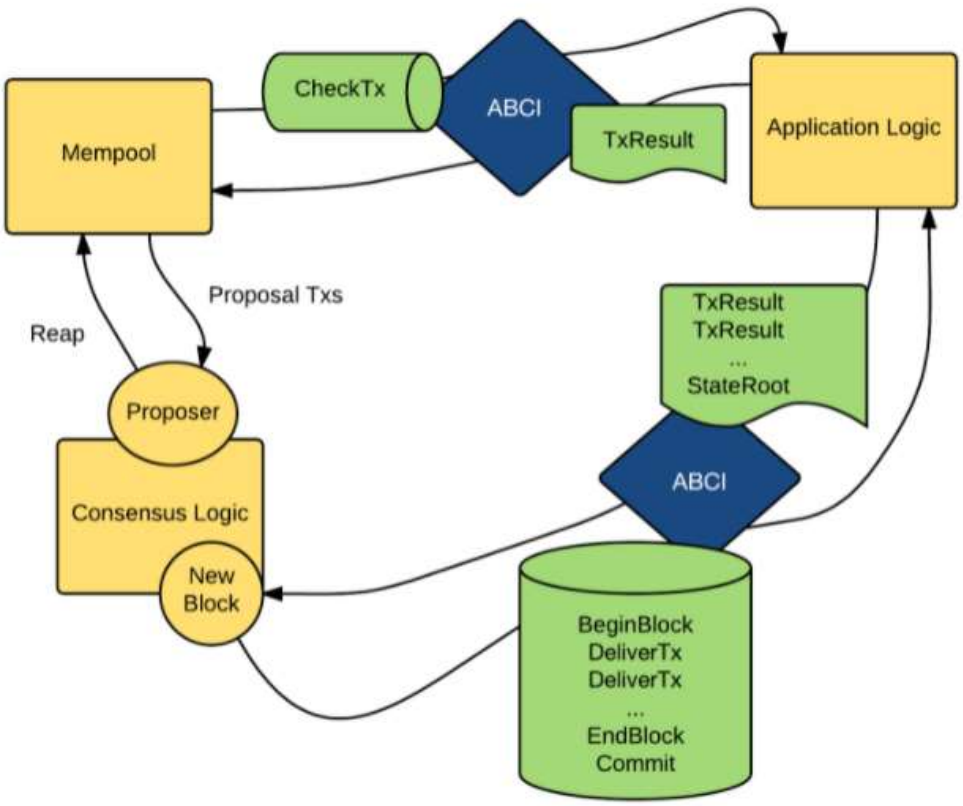
Il protocollo segue una semplice macchina a stati che assomiglia a questa:



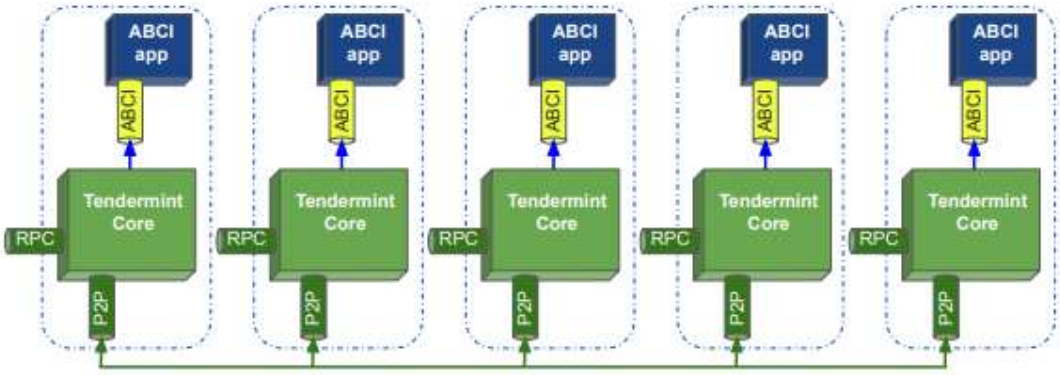
L'altra componente di Tendermint, l'Application BlockChain Interface, funge da interfaccia tra il processo applicativo e il processo di consenso. L'ABCI comunica con il Core di Tendermint principalmente attraverso 3 tipi di messaggi: **DeliverTx, CheckTx e Commit**. Il messaggio **DeliverTx** viene usato ogni qual volta viene inviata una transazione sulla blockchain. L'applicazione poi dovrà verificare la validità della transazione rispetto allo stato attuale. Se la transazione risulterà valida, allora l'applicazione dovrà aggiornare il proprio stato con le nuove informazioni ottenute dalla transazione. Quando un'applicazione presente sulla rete Tendermint vuole inviare una transazione, i dati immessi dall'utente vengono pre-processati e temporaneamente salvati all'interno di una memoria cache, la Mempool Cache. Prima di essere immessa nella Mempool vera e propria, ossia la memoria da cui poi un nodo può recuperare le transazioni da includere nel blocco che proporrà, il nodo verifica la validità della transazione tramite CheckTx: il contenuto della transazione viene confrontato con l'attuale stato del sistema e, se viene ritenuto coerente con esso, la transazione verrà accettata dal sistema, altrimenti verrà rifiutata. Di primo acchito può sembrare che CheckTx sia una DeliverTx semplificata, ma in realtà questi due messaggi vengono inviati in momenti diversi. Occorre analizzare le connessioni che l'interfaccia ABCI mantiene per capire la differenza tra i due.

L’applicazione ABCI presente su ogni nodo mantiene tre connessioni con il Tendermint Core: la Mempool Connection, usata per validare le transazioni presenti sulla Mempool tramite l’utilizzo di CheckTX, la Consensus Connection, usata soltanto quando viene effettuata la commit di un nuovo blocco, e la Query Connection, usata per effettuare query all’applicazione senza passare dal consenso. Lo stato dell’applicazione fornisce le informazioni necessarie, solo in lettura, alla Mempool Connection e alla Query Connection, mentre la scrittura viene presa in carico dalla Consensus Connection ed è proprio il blocco ricevuto da questa connessione a contenere tanti messaggi DeliverTX quante transazioni sono presenti nel blocco. Riassumendo, il messaggio CheckTx viene utilizzato quando una transazione deve essere inserita all’interno della Mempool, quindi prima del processo di consenso, mentre il messaggio DeliverTx viene usato dal consenso quando va ad assemblare il blocco, ordinando le transazioni. Infine, il messaggio Commit viene utilizzato per computare l’hash del Merkle tree corrispondente allo stato dell’applicazione.

Il diagramma seguente illustra il flusso dei messaggi tramite ABCI.



Per ricevere transazioni da Tendermint Core tramite ABCI, un'applicazione client deve implementare un **wrapper**, chiamato **applicazione ABCI**. Ad eccezione di Tendermint Core, nient'altro dovrebbe comunicare con l'applicazione ABCI, per garantire risultati deterministici. Tendermint Core e l'applicazione ABCI insieme formano un nodo e più nodi formano una rete peer-to-peer, come mostrato in figura:



Un client può inviare transazioni che devono essere elaborate da Tendermint Core a qualsiasi nodo della rete tramite il protocollo Remote Procedure Call (RPC), come illustrato nella figura precedente. Questa interfaccia REST può essere utilizzata anche per formulare query. I nodi comunicano tra loro su una rete peer-to-peer e con la loro applicazione ABCI tramite un protocollo socket descritto all'interno dell'ABCI.

Tendermint Core crea tre connessioni ABCI verso il relativo wrapper: uno per la validazione delle transazioni, prima di inoltrarle agli altri peer, uno per la proposta dei blocchi per la procedura di consenso, e uno per l'interrogazione dello stato dell'applicazione.

Avvio di Tendermint Core

Per gestire una rete Tendermint, ogni nodo che partecipa al consenso richiede una chiave pubblica e una privata. Ogni nodo mantiene una cartella di configurazione, contenente un file dove sono memorizzate le informazioni sulle chiavi: priv_validator_key.json. Inoltre, le chiavi pubbliche devono essere elencate in un file genesis.json comune a tutti i nodi che partecipano alla chain, per facilitare l'identificazione tra i nodi e della chain utilizzata. Il file config.toml contiene l'indirizzo per la comunicazione peer-to-peer tra i nodi e l'indirizzo per il listener RPC.

Per realizzare una chain composta da un solo nodo validatore (utile nella fase di implementazione degli altri stack applicativi) occorre lanciare il comando:

tendermint init validator

questo comando genera i file **genesis.json, config.toml, etc**. all'interno della cartella (home/pi/.tendermint/config, o in windows C:/.tendermint/config) inizializzando un nodo che in questo caso ha la funzione di "validator". Si possono inizializzare nodi anche con altre funzionalità.

Il file genesis.json serve per inizializzare il primo blocco della chain e contiene le seguenti informazioni:


```
{
  "genesis_time": "2020-04-21T11:17:42.341227868Z",
  "chain_id": "test-chain-ROp9KF",
  "initial_height": "0",
  "consensus_params": {
    "block": {
      "max_bytes": "22020096",
      "max_gas": "-1",
      "time_iota_ms": "1000"
    },
    "evidence": {
      "max_age_num_blocks": "100000",
      "max_age_duration": "172800000000000",
      "max_num": 50,
    },
    "validator": {
      "pub_key_types": [
        "ed25519"
      ]
    }
  },
  "validators": [
    {
      "address": "B547AB87E79F75A4A3198C57A8C2FDAF8628CB47",
      "pub_key": {
        "type": "tendermint/PubKeyEd25519",
        "value": "P/V6GHuZrb8rs/k1oBorxc6vyXMlnzhJmv7LmjELDys="
      },
      "power": "10",
      "name": ""
    }
  ],
  "app_hash": ""
}
```


Il file configura una blockchain locale composta da un solo nodo validatore. I campi descrivono quanto segue:

- **genesis_time**: è il tempo ufficiale di creazione della blockchain.

- **chain_id**: ID della blockchain. Questo deve essere unico per ogni nodo della chain**.** Nel caso di una testnet quindi se l'identificativo non è unico non funzionerà nulla.
- **initial_height**: rappresenta il punto iniziale da cui parte la blockchain. In caso di una nuova sarà 0, ma potrebbe trattarsi anche di un fork o un aggiornamento della rete a partire dall'ultimo nodo ritenuto valido.
- **consensus_params :** definisce i parametri per l'algoritmo di consenso.
- **validators**: Elenco dei validatori iniziali. Può essere lasciato vuoto per rendere esplicito che l'applicazione inizializzerà il set di validatori attraverso la procedura ResponseInitChain, ovvero eseguendo una ricerca dei nodi sulla rete.
- **app_hash**: L'hash dell'applicazione previsto (come restituito dal messaggio ABCI ResponseInfo) al momento della genesi. Se l'hash dell'app non corrisponde, Tendermint si blocca.
- **app_state**: lo stato dell'applicazione (ad es. distribuzione iniziale dei token).

Il primo passo quindi è aggiungere al file genesis.json una lista dei nodi validatori iniziali(almeno 4). I dati bisogna reperirli dai file **priv_validator.json e pub_validator.json** che contendono l'ID del nodo, la chiave pubblica ed il suo indirizzo.

L'ID del nodo può essere inserito all'interno del file config.toml all'interno del campo: persistent-peers = "" nella sezione : P2P Configuration Options, oppure in alternativa si possono esplicitare i nodi peer a linea di comando all'avvio di Tendermint Core, come nella seguente istruzione:

tendermint start --p2p.persistent-peers
"429fcf25974313b95673f58d77eacdd434402665@10.11.12.13:26656,
96663a3dd0d7b9d17d4c8211b191af259621c693@10.11.12.14:26656"

Il file config.toml è usato per configurare il server locale ed è molto simile al file di configurazione di apache o postgres.

Tra i vari settaggi che si possono compiere, ad esempio è utile settare il campo create-empty-blocks a false in modo che il server non inserisca nella chain blocchi vuoti :

 # EmptyBlocks mode and possible interval between empty blocks
create-empty-blocks = true
create-empty-blocks-interval = "0s"

Se ad esempio viene settato create-empty-blocks = false e create-empty-blocks-interval = "30s", Tendermint creerà blocchi solo se ci sono transazioni valide oppure dopo aver atteso 30 secondi senza ricevere transazioni.

Come abbiamo già visto in precedenza, la configurazione può essere data anche da promt dei comandi:

tendermint start --consensus.create_empty_blocks=false
--consensus.create_empty_blocks_interval="5s"

Eseguito l'init del nodo, per eseguire Tendermint Core occorre lanciare quindi il comando:

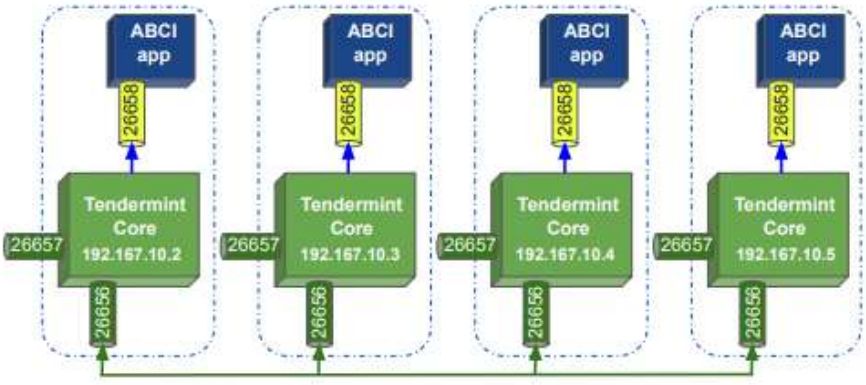
1. tendermint start (nel caso di applicazione in-process) / tendermint node (in caso di RCP)
2. tendermint start --proxy-app=kvstore

Il primo avvia un server e si mette in ascolto verso l'applicazione ABCI sulla porta di default 127.0.0.1:**26658**, mentre il secondo comando avvia il server ABCI su una porta specificata. Nel caso di esempio, in cui l'app è in-process (è in forma binaria o scritta in go) i due comandi sono equivalenti.

Abbiamo quindi definito una prima porta di comunicazione con Tendermint Core: la porta 26658. Su questa porta possono comunicare soltanto l'applicazione Tendermint Core con l'ABCI scritta in go. Se vogliamo comunque interagire con la blockchain, per fare test, vedere lo stato del server o altro possiamo usare la porta socket utilizzata per le chiamate RCP, ovvero la porta 127.0.0.1:**26657**. Pertanto digitando dal browser:

http://localhost:26657

verranno visualizzati tutti i servizi esposti e sarà possibile cliccare su alcuni di essi per vedere lo stato della chain. Nel caso in cui l'applicazione ABCI non è scritta in go, e quindi il processo non può essere lanciato in-process, occorrerà che l'applicazione ABCI realizzata comunichi utilizzando il servizio RCP.



Per concludere la terza porta di comunicazione utilizzata è la **26656**, che identifica la porta di comunicazione del protocollo di gossip P2P e quindi utilizzata esclusivamente da Tendermint Core per gestire lo scambio di dati con il resto della rete.

Nel caso dell'avvio dell'applicazione di test "tendermint start --proxy-app=kvstore", aprendo un'altra finestra e digitando il comando:

curl http://localhost:26657/broadcast_tx_commit?tx=\"chiave = abcd\"

comunicheremo attraverso il servizio RCP alla blockchain kvstore. In questo caso inviando una transizione costituita da una coppia chiave valore.

Se vogliamo conoscere lo stato della chain possiamo digitare:

curl http://localhost:26657/status | json_pp

oppure:

curl http://localhost:26657/status | json_pp | grep latest_app_hash

per visualizzare l'app_hash, e così via.

Resettare la chain

In fase di sviluppo e test dell'applicazione è necessario effettuare il reset dei dati nella chain e ripartire la zero. Per eseguire questa operazione, occorre anzitutto sospendere il servizio attraverso il comando:

tendermint stop

dopo di che occorre digitare il comando:

tendermint unsafe_reset_all

questo eliminerà la directory data presente nella directory "home/pi/.tendermint/". La directory data contiene il database delle transizioni, dei blocchi, il mempool, che sono stati elaborati durante l'esecuzione.

Se queste transizioni non cono coerenti con le modifiche apportate al codice, l'applicazione non funzionerà.

Nel caso in cui è necessario anche rimuovere i file di configurazione e rigenerare tutto da capo occorre eseguire i comandi:

rm -rf ~/.tendermint

tendermint init validator

Debug

Ci sono tre livelli di debug: info, debug and error. Questi possono essere configurati da command line attraverso il commando **tendermint start --log-level "info"** o sul file config.toml

- **Info:** Viene utilizzato per mostrare che i moduli sono stati avviati, arrestati e se stanno funzionando.
- **Debug:** Il debug viene utilizzato per tenere traccia di varie chiamate o problemi.
- **Error:** L'errore rappresenta qualcosa che è andato storto. Un log degli errori può rappresentare un potenziale problema che può portare all'arresto del nodo.

Verranno stampati sul promp i vari log di esecuzione. In maniera analoga, il nodo può essere interrogato attraverso il servizio RCP attraverso le chiamate:

curl http(s)://{ip}:{rpcPort}/status

curl http(s)://{ip}:{rpcPort}/dump_consensus_state

Attraverso il servizio **status** vengono fornite informazioni di massima: quante volte che il nodo si sincronizza o meno, a che altezza si trova della chain, ecc. mentre il servizio **dump_consensus_state** fornisce una panoramica più dettagliata sullo stato di consenso: proponente, ultimi validatori, stati dei pari, etc. Da questo, ad esempio è possibile capire perché, ad esempio, la rete si è interrotta.

Tendermint emette diversi eventi, a cui ad esempio si ci può iscriverte tramite Websocket. Questo può essere utile per applicazioni di terze parti (per l'analisi) o per l'ispezione dello stato. In questo caso si può utilizzare il **tool wsat** che visualizza su CLI (Command Line Interface) la sequenza dei log relativo ad un particolare evento:

wscat ws://127.0.0.1:26657/websocket

> { "jsonrpc": "2.0", "method": "subscribe", "params": ["tm.event = 'NewBlock'"],"id": 1 }

In questo caso ci siamo sottoscritti all'evento generato dall'approvazione di un nuovo blocco.

Un altro tool analogo **è ws**:

ws ws://localhost:26657/websocket > { "jsonrpc": "2.0", "method": "subscribe", "params": ["tm.event='NewBlock'"], "id": 1 }

Local Testnet

Tendermint mette a disposizione un comando per avviare una rete di nodi, che per impostazione predefinita sono quattro nodi validatori. Ma, come già visto per altri comandi, attraverso linea di comando è possibile impostare una diversa configurazione, o se si vuole rendere permanete questa impostazione, modificando un apposito file di configurazione.

Il seguente comando, ad esempio, esegue l'inizializzazione per una testnet composta da cinque nodi, dove il parametro v sta per il numero di validatori:

tendermint testnet --v 5

Questo comando crea nella root principale una directory chiamata **mytestnet**, contenente una cartella per ogni nodo (nodo0, nodo1, nodo2, nodo3, nodo4). Ogni nodo, a sua volta contiene una cartella di configurazione per ogni nodo, quindi vi si trovano le chiavi, il file genesis.json comune a tutti i nodi ed il file config.toml. Per impostazione predefinita, tutti i nodi hanno la stessa configurazione dell'indirizzo specificata nel file config.toml, altrimenti la net non funzionerebbe.

Per evitare conflitti, dato che tutti i nodi in fase di test possono essere lanciati sulla stessa macchina, questi avranno configurate tutte porte differenti, diversamente da quando detto nei paragrafi precedenti. Di seguito viene riportato uno schema di configurazione:

node	P2P address	RPC address
0	26656	26657
1	26659	26660
2	26661	26662
3	26663	26664
4	26665	26666

Affinché i nodi possano stabilire una comunicazione tra di loro, all'avvio di ogni nodo occorre specificare l'elenco dei peer con le relative porte P2P. Quindi, ogni nodo viene avviato fornendogli un elenco di tutti i peer nella rete. Questo elenco contiene gli ID e gli indirizzi di tutti i nodi e il seguente comando mostra come si ottiene l'ID di node0:

tendermint show-node-id --home ./tendermint/mytestnet/node0

Dopo aver eseguito questo comando per ogni nodo, è possibile costruire il comando di avvio. Per esempio, per avviare node0, il comando sarà del tipo:

tendermint node --home ./mytestnet/node0 --proxy_app=kvstore --p2p.persistent_peers="7793b14e436a37e0d18bb3820546a3aca98e1694@localhost:26656,36796da0e43680b711c580c032eb10199ad58a4a@localhost:26659, aa5cc9c62324744f55e80eb2257690cbdd5b5544@localhost:26661, e957be554edbe0acb36f3888a2f8255ace7614d0@localhost:26663, f3888a2f8255ace7614d09e57be554edbe0acb36@localhost:26665,"

Per eseguire tutti gli altri nodi, lo stesso comando deve essere eseguito con i parametri corrispondenti. Dopo l'avvio, i nodi stabiliranno connessioni tra loro e nell'esempio dato eseguiranno l'applicazione kvstore, che è un'applicazione di esempio fornita da Tendermint per dimostrare la funzionalità. Infine, i nodi Tendermint saranno pronti a ricevere le transazioni da elaborare.

curl http://localhost:26657/broadcast_tx_commit?tx=\"chiave = abcd\"

se ci riferiamo al nodo0 o localhost:26660 se indirizziamo la nostra richiesta al nodo1.

Se si vogliono creare delle configurazioni di nodi da distribuire su nodi fisici (macchine) differenti, occorrerà lanciare il comando:

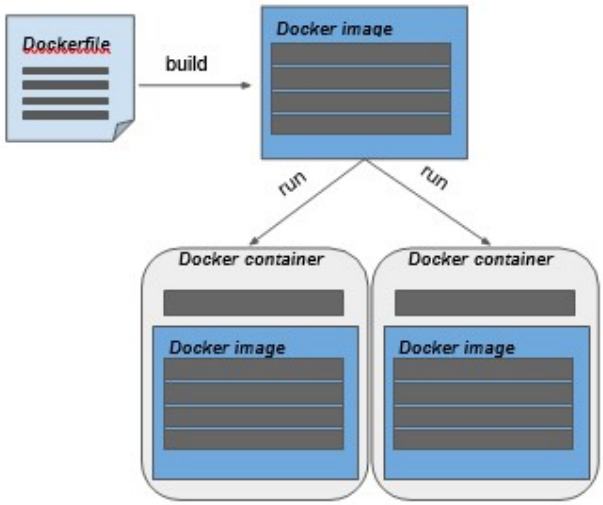
tendermint testnet --v 4 --o ./output --populate-persistent-peers --starting-ip-address 192.168.10.111

Questo commando genera la cartella "**output**" contenente i file di configurazione di quattro nodi validatori a cui viene assegnato un indirizzo IP a partire dall'indirizzo 192.168.10.111.

Distributed Testnet

Tendermint fornisce l'infrastruttura per eseguire una rete in un ambiente distribuito virtualizzato. Questo viene realizzato con l'ausilio di Docker, una piattaforma che facilita la virtualizzazione a livello di sistema operativo. Questa virtualizzazione consente la simulazione di un sistema distribuito, più precisamente, l'utilizzo di uno spazio di indirizzi virtuale per i nodi Tendermint e l'applicazione ABCI. Per virtualizzare un'applicazione con Docker, innanzitutto è necessario un Dockerfile specifico per l'applicazione. Un Dockerfile è un file di script, contenente le istruzioni su come configurare l'ambiente per l'applicazione. In secondo luogo, viene creata l'immagine Docker, che consiste in layers di sola lettura, uno per ogni istruzione specificata nel Dockerfile. Infine, viene creato un contenitore Docker aggiungendo un livello scrivibile sopra l'immagine precedentemente creata, dove tutto quello che viene eseguito nel contenitore, come l'aggiunta, la modifica o l'eliminazione di

dati, vengono confinati e memorizzati nel container. L'immagine Docker può essere utilizzata per creare più contenitori, tutti in esecuzione con la stessa applicazione, ma tutti con il proprio livello modificabile. I container Docker sono isolati l'uno dall'altro e comunicano attraverso canali ben definiti. Il processo di creazione di un'immagine Docker da un Dockerfile e l'esecuzione di conteiners, basato su una immagine Docker, è illustrato in figura. Questo design semplifica il deployment e l'esecuzione delle applicazioni.



Per facilitare la gestione dei contenitori Docker, Docker Compose fornisce un file di configurazione di facile comprensione, il docker-compose.yml. In questo file, si possono definite, abilitare la creazione e l'avvio di tutti i servizi di un'applicazione multi-contenitore con il solo comando:

docker-compose up

Questo comando creerà un contenitore Docker per ogni servizio elencato nel file di configurazione. Tendermint fornisce un file docker-compose.yml con le configurazioni per una rete composta da quattro nodi validatori, che replica la testnet vista nel paragrafo precedente.

Tutti i passaggi descritti per la creazione di una testnet su Docker sono inseriti in un file di script "Makefile" che può essere lanciato col comando:

make localnet-start

questo innesca l'esecuzione dei seguenti passaggi:

1. Crea l'immagine Docker in base dal Dockerfile fornito per un nodo Tendermint.
2. Creare le cartelle per ogni nodo, contenenti le informazioni sulle chiavi, il genesis.json e il config.toml, all'interno dalla cartella "tendermint/networks/local/localnode/"
3. Esegui docker-compose up.

Le configurazioni nel file docker-compose.yml specificano un indirizzo IP per ogni nodo e le rispettive porte, la porta 26656 per la comunicazione peer-to-peer e la porta 26657 per l'RPC. Ogni nodo stabilisce una connessione all'applicazione ABCI sulla porta 26658, nel caso dell'applicazione kvstore on-process.

Sulla macchina host, i nodi legano i propri server RPC alle porte 26657, 26660, 26662 e 26664, quindi, le transazioni possono essere inviate a uno di questi indirizzi. Più precisamente, node0 è accessibile tramite porta 26657, node1 sulla porta 26660 e così via. Considerando l'applicazione di esempio kvstore, che memorizza una chiave e un valore, una transazione valida, indirizzata a node2, sarebbe:

curl -s 'localhost:26662/broadcast_tx_commit?tx="name=emilio"'

All'interno dell'ambiente Docker, la transazione verrà inoltrata all'indirizzo di node2 (192.167.10.4:26657) e node2 avvierà il consenso ed eventualmente risponderà all'applicazione.

Configuration of a Distributed Testnet

Per configurare una rete Tendermint con una applicazione client, devono essere eseguiti i seguenti step:

1. Definire l'ambiente l'immagine Docker in cui eseguire l'applicazione con un Dockerfile. Nota, che una porta 26658 deve essere esposta per consentire a Tendermint Core di stabilire la comunicazione con l'applicazione ABCI (se non si tratta di una App scritta in go).

2. Aggiungere l'applicazione nel docker-compose.yml, in modo che sia enumerata un'applicazione per ogni nodo. Ad esempio, se si desidera avviare una rete di cinque nodi, in totale devono essere elencati dieci servizi, cinque per i nodi e cinque per l'applicazione. Docker Compose creerà quindi un contenitore per ogni servizio.

3. Infine, per consentire ai container di nodi di stabilire una connessione con i corrispondenti container applicativo all'avvio è richiesto un comando idoneo all'utilizzo del servizio. Ad esempio, se vogliamo che il container node0 si connetti col container abci0 all'avvio, quindi aggiungiamo il cogente comando al file docker-compose.yml:

comand: node --proxy_app=tcp://abci0:26658

L'esecuzione di una testnet Tendermint con Docker è un approccio all'avanguardia per impacchettare ed eseguire software, e la combinazione con Docker Compose consente una gestione efficiente di applicazioni Docker multi-contenitore.

Example Distributed Testnet : kvstore

Iniziamo col mostrare i passi da seguire per la verifica della corretta installazione della piattaforma con i relativi tools a supporto (go, Docker, Docker Composer, etc.). Noi abbiamo utilizzato l'ultima versione di Tendermint disponibile su Github (V0.34.15).

Ricordiamo che l'applicazione kvstore è un'applicazione scritta in go (in-process) e comunica con Tendermint core attraverso la porta 26658 e non attraverso il server RCP. La porta su cui è in ascolto il server RCP invece è la 26657 che utilizzeremo per inviare transizioni da linea di comando.

Possiamo inizialmente verificare che l'applicazione funzioni correttamente lanciando una singola istanza di Tendermint (ovvero una chain composta da un solo nodo validatore) attraverso il comando:

tendermint init validator

Che come già detto crea i file di configurazione della chain nella cartella _ ./tendermint, dove se vogliamo possiamo intervenire nel modificare il file config.toml, ad esempio per settare:

create-empty-blocks = false

che disabilita l'approvazione di blocchi vuoti. Questo ci evita di digitare il comando ogni volta che lanciamo l'applicazione.

A questo punto possiamo utilizzare il comando:

tendermint start

oppure il comando:

abci-cli kvstore

quest'ultimo avvia Tendermint Core, l'ABCI che gestisce le richieste di comando via riga di comando (CLI).

L'app kvstore quindi può essere lanciata su una seconda finestra (promt dei comandi) attraverso il comando:

abci-cli console

A questo punto si possono inviare diverse richieste, come ad esempio:

- deliver_tx "abc" ----> inoltra una transizione di contenuto abc
- info ----> verifica l'approvazione da parte di ABCI
- commit ----> richiesta di inserimento della transazione nella chain
- query "abc" ----> verifica inserimento nella chain
- deliver_tx "def=xyz" ----> inoltra una transizione chiave valore

Per terminare l'esecuzione dell'app è sufficiente digitare CTRL-C.

Un altro test che si può compiere è la verifica del funzionamento del servizio RCP, quindi oltre ad andare sul browser e digitare l'indirizzo:

http://localhost:26657

possiamo inoltrare richieste attraverso il comando curl di linux, ad esempio:

```
curl -s 'localhost:26657/broadcast_tx_commit?tx="abcd"'

curl -s 'localhost:26657/abci_query?data="abcd"'

curl -s 'localhost:26657/broadcast_tx_commit?tx="name=emilio"'

curl -s 'localhost:26657/abci_query?data="name"'
```

Bene, ora siamo pronti per lanciare una blockchain composta da più nodi (minimo 4 validatori per testare il protocollo di gossip) avvalendoci dei container Docker.

La prima volta che viene lanciata una testnet occorre generare le immagini Docker da caricare nei container, quindi i comandi da lanciare sono i seguenti:

make build-linux

Il comando genera un eseguibile di tendermint che inserisce nella cartella ./build, ed :

make build-docker-localnode

che crea l'immagine Docker: tendermint/localnode. A questo punto non ci resta che creare I file di configurazione per i Quattro nodi e mandarli in esecuzione attraverso il Docker compose. Tutto questo viene fatto lanciando lo script:

make localnet-start

ed

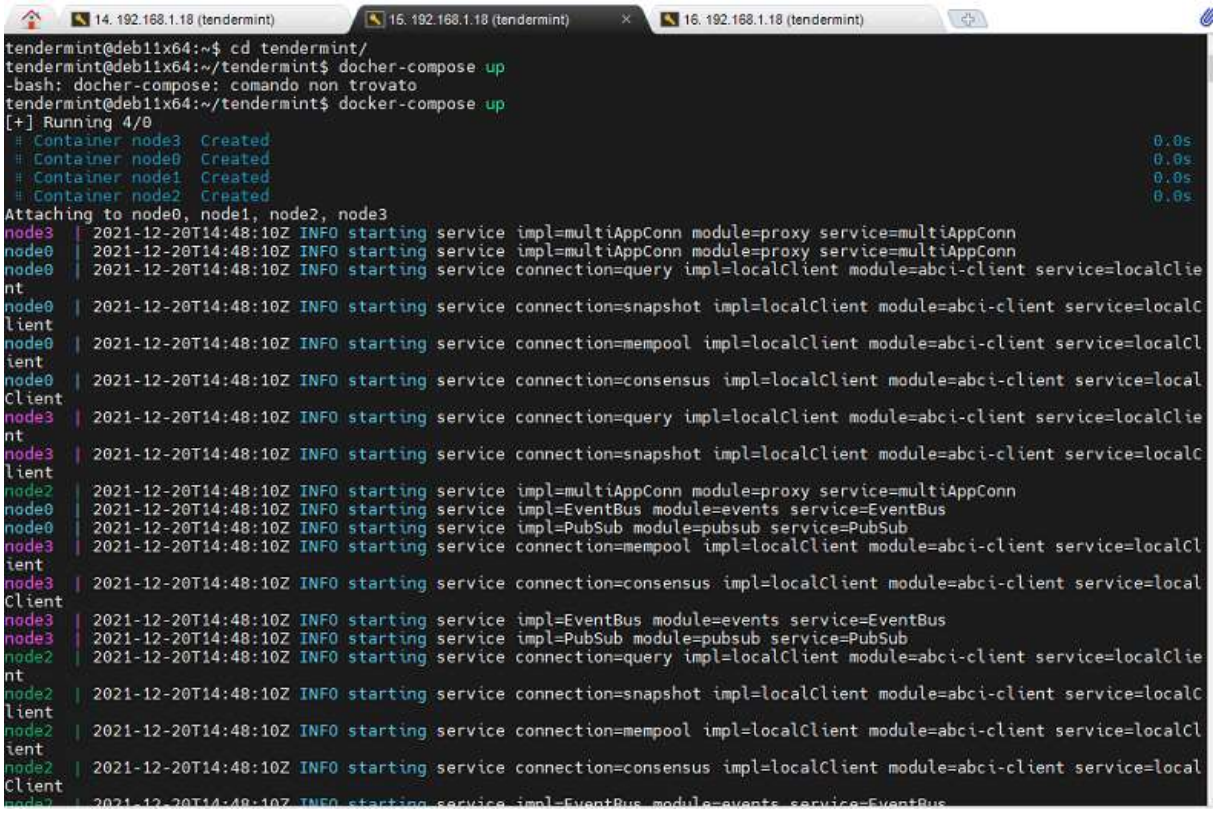
make localnet-stop

per terminare l'esecuzione e rimuovere i container precedentemente creati.

La seconda volta che si lancia l'esecuzione della rete è sufficiente lanciare i comandi:

docker-compose up ed docker-compose down

per costruire ed avviare la rete Docker e per terminarla. Il risultato dell'esecuzione è il seguente:



Siamo pronti per inviare le nostre transizioni. Quindi occorre aprire una seconda finestra ed attraverso il comando linux curl inviare i nostri dati.

Nel file docher-compose.yml è definita una rete virtuale Docker che utilizza gli indirizzi dal 192.167.10.0/15, dove il nodo0 sarà raggiungibile all'indirizzo 192.167.10.2.

A questo punto possiamo decidere di aprire un'istanza docker sul nodo0

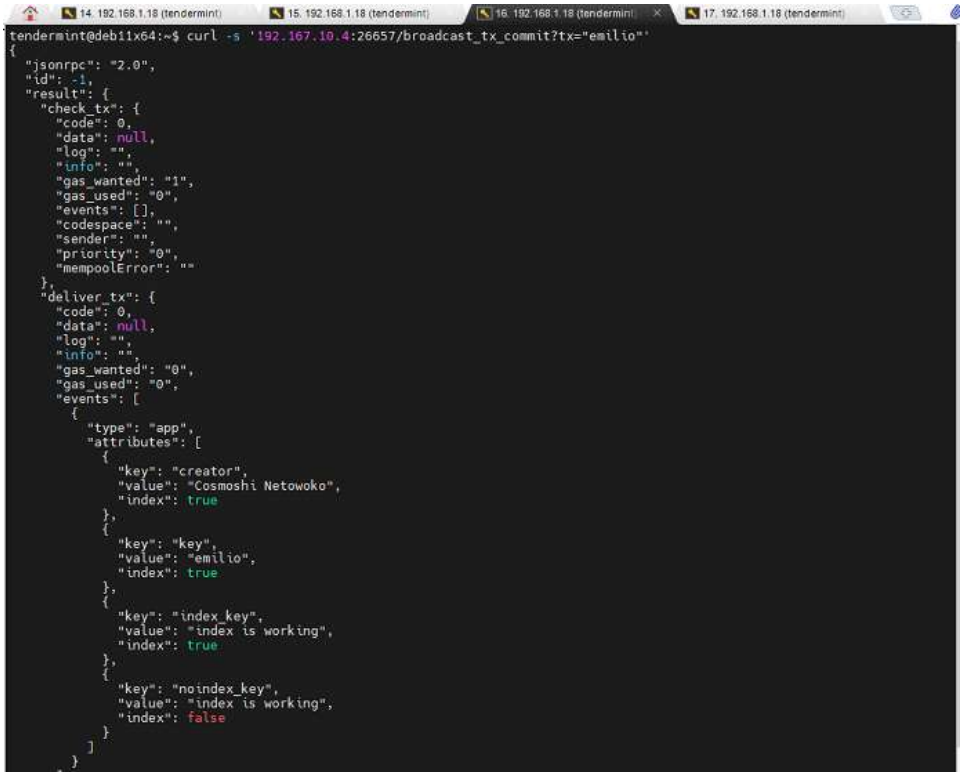
docker run –it node0

ed interagire con la propria app kvstore digitando l'interfaccia da riga di comando:

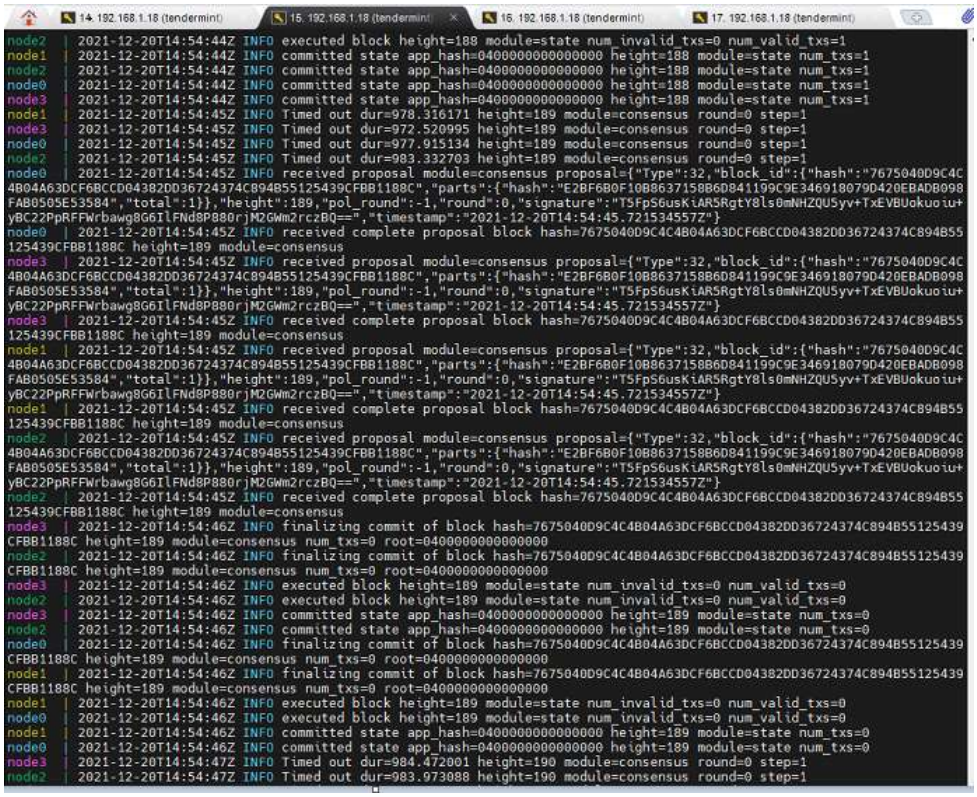
abci-cli console

oppure utilizzando l'interfaccia REST e quindi inviando i dati attraverso il comando curl.

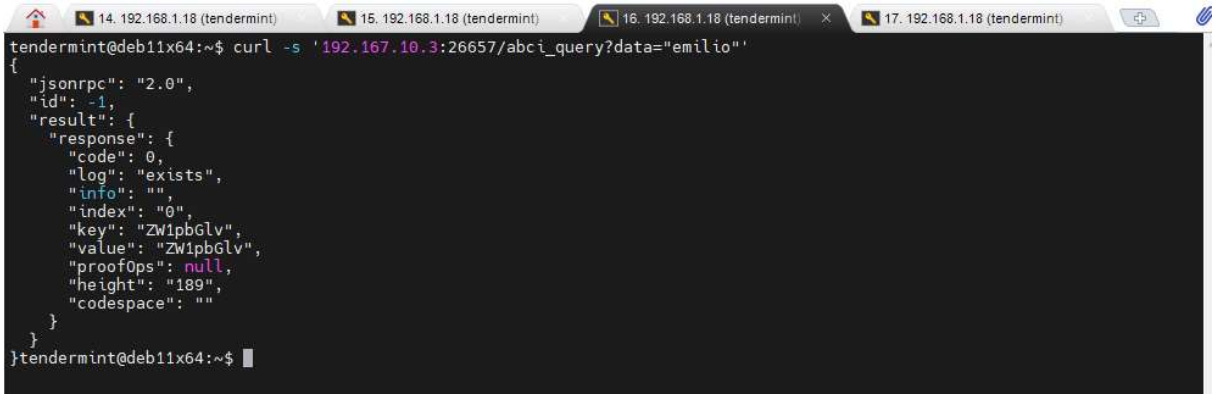
Nell'esempio seguente, dal container node0 viene inviata una transazione al node2 con richiesta di feedback all'inserimento in un blocco.



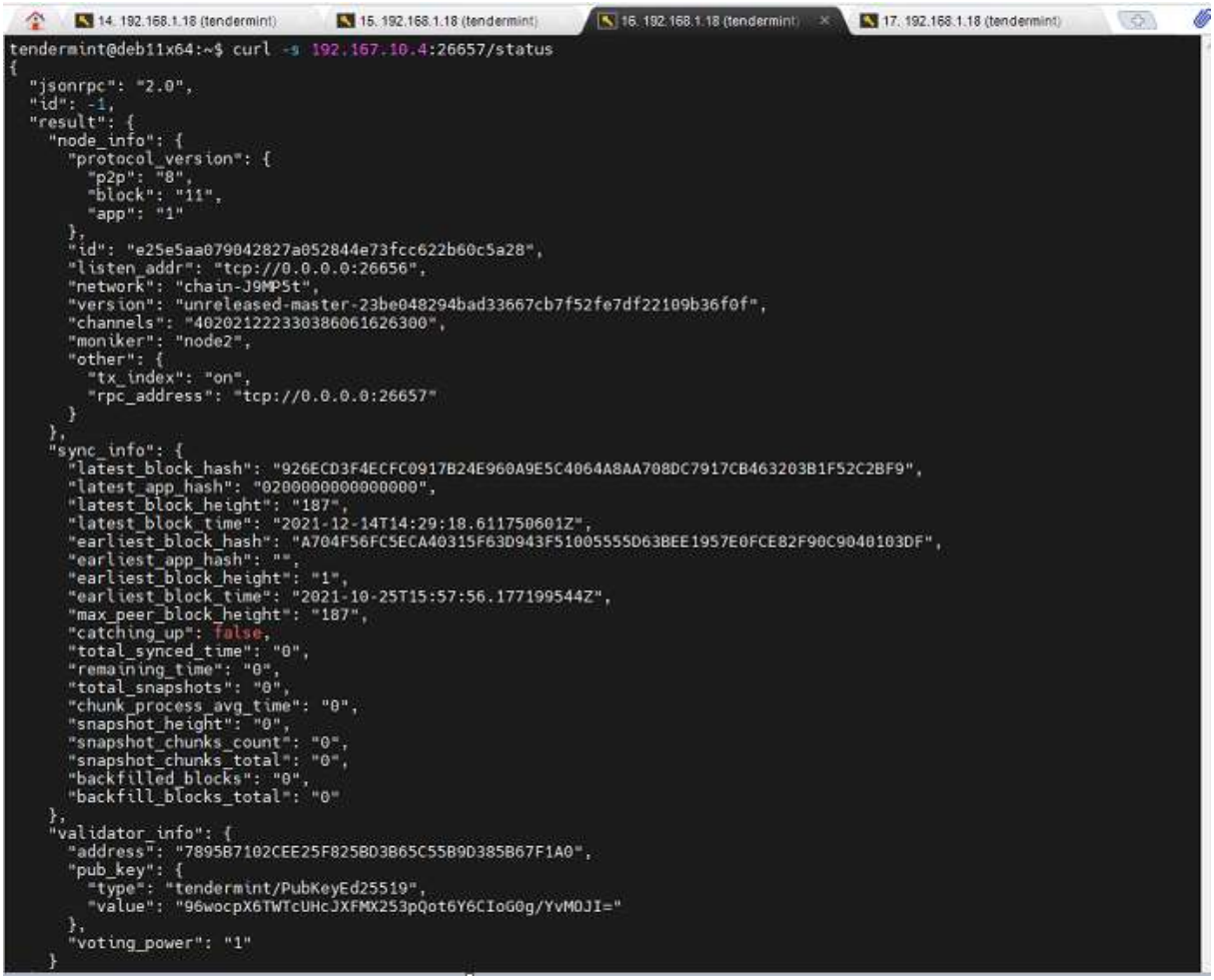
Tornando alla finestra precedente, di lancio del docker-compose, potremmo analizzare la sequenza di attività di approvazione della transazione:



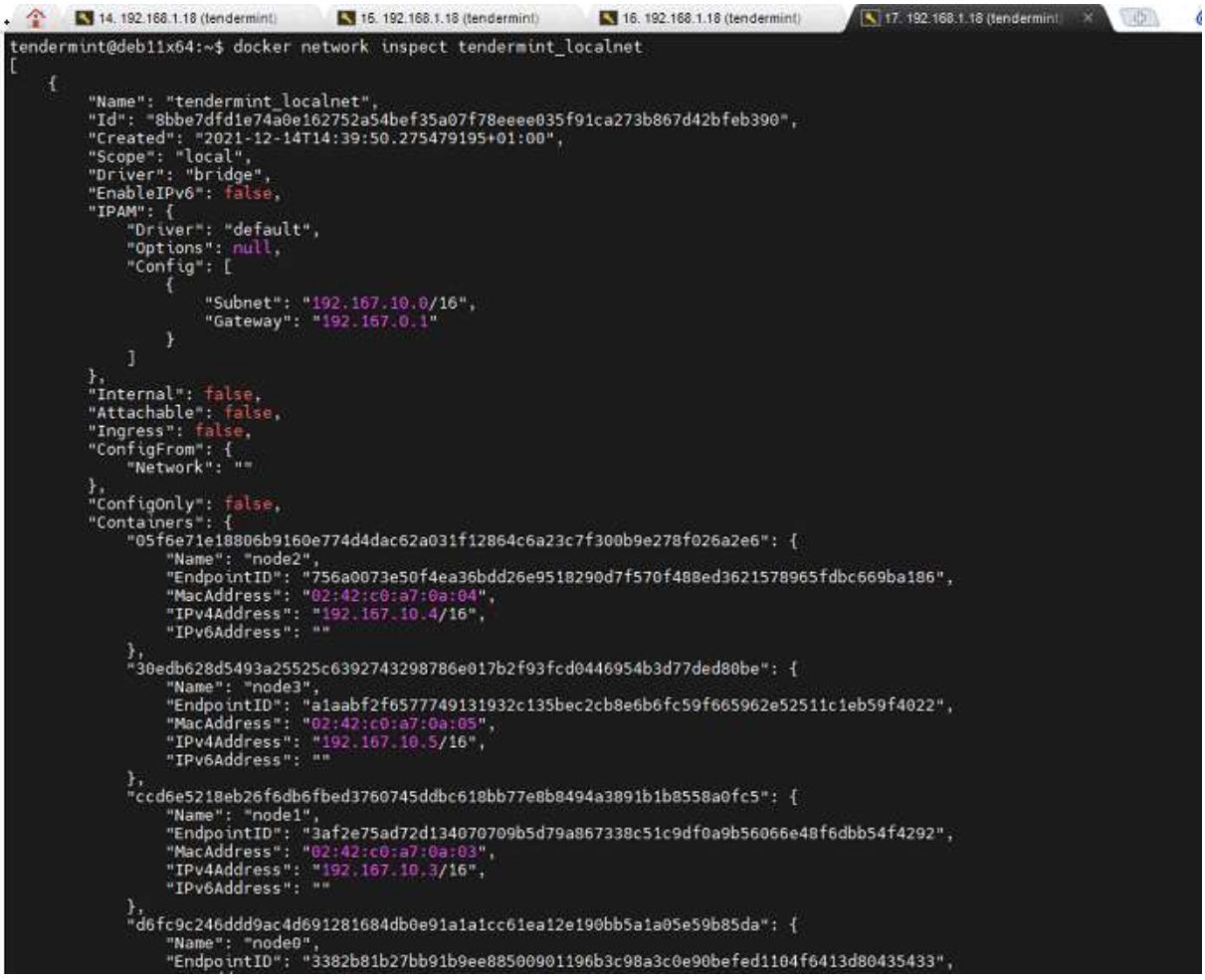
Per verificare che la transizione è stata replicata su tutti i nodi, possiamo effettuare attraverso una query. In questo caso stiamo chiedendo al node1, l'esistenza della chiave "emilio". La risposta è positiva e la transizione è inserita all'altezza 189 della chain.



Possiamo quindi avere varie informazioni interagendo con il server RCP:



Oppure analizzando la rete Docker, come segue:



Protocollo ABCI

ABCI sta per " **A**pplication **B**lock **C**hain **I**nterface". Ovvero rappresenta è l'interfaccia tra Tendermint Core e l'applicazione (la vera macchina a stati). Consiste in un insieme di *metodi* , ciascuno con una corrispondente coppia di messaggi di tipo Request e Response. Per eseguire la replica di una macchina a stati sui nodi, Tendermint chiama i metodi ABCI inviando messaggi di Request e ricevendo in cambio i messaggi di Response.

Questa separazione con la logica di controllo consente a Tendermint di funzionare con applicazioni scritte in molti linguaggi di programmazione.

Quando Tendermint e l'applicazione ABCI vengono eseguiti come processi separati, Tendermint apre quattro connessioni socket all'applicazione per i diversi metodi ABCI. Le connessioni gestiscono ciascuna un sottoinsieme delle chiamate.

Questi sottoinsiemi sono definiti come segue:

Consensus connection

- è responsabile della gestione dei blocchi
- gestisce la richiesta InitChain, BeginBlock, DeliverTx, EndBlock, e Commit.

Mempool connection

- è responsabile della convalida di nuove transazioni, prima che queste vengano condivise o incluse in un blocco.
- gestisce le chiamate CheckTx

Info connection

- si occupa dell'inizializzazione e delle query utente.
- gestisce le chiamate Info e Query.

Snapshot connection

- serve a supportare la sincronizzazione dello stato.
- gestisce le chiamate ListSnapshots, LoadSnapshotChunk, OfferSnapshot, e ApplySnapshotChunk.

ABCI Facts in java

In questo paragrafo vedremo nei dettagli la realizzazione di un'applicazione ABCI. L'applicazione tiene traccia di una serie di fatti in base alla loro fonte e conserva i conteggi in un proprio database. I fatti vengono inviati da applicazioni esterne a ogni nodo della blockchain. Se un fatto viene accettato dall'ABCI, verrà registrato nella blockchain come una transizione. Una volta che la blockchain, costituita da Tendermint Core e dell'ABCI sono in esecuzione, possono essere inviati una serie di fatti come transizioni per essere approvate. Ogni fatto contiene una fonte (source) e un'affermazione (statement).

L'applicazione java sfrutta la libreria jTendermint semplificandoci lo sviluppo dell'applicazione. Purtroppo non essendoci recenti aggiornamenti, dobbiamo attenerci alla compatibilità delle versioni richieste per Tendermint Core e JDK.

Pertanto utilizzeremo jtendermint0.32.3, openjdk-17-jdk e tendermint-0.32.3

Di seguito si riporta la sequenza di installazione:

- Assicurarsi che in pom.xml la versione di jtendermint sia corretta (0.32.3)
- sudo apt update

- sudo apt install openjdk-17-jdk
- mvn package -U
- mkdir -p $GOPATH/src/github.com/tendermint
- cd $GOPATH/src/github.com/tendermint
- git clone https://github.com/zlyzol/tendermint-0.32.3.git
- cd tendermint
- make get_tools
- make install
- make build

L'applicazione che vogliamo realizzare è molto semplice, quindi ci serve implementare soltanto I Quattro metodi principali del protocollo ABCI : IDeliverTx, ICheckTx, ICommit e IQuery.

Per semplicità non impiegheremo un database relazionale esterno per memorizzare lo stato dell'applicazione. Al contrario istanziamo nell'applicazione, come archivio per i dati, una tabella di codici hash globale. La chiave della tabella è la fonte univoca dei fatti e il valore è il numero di fatti associati a quella fonte. All'avvio dell'applicazione viene lanciato un socket server che rimane sempre in ascolto dei messaggi in arrivo dalla blockchain:

```
public FactsApp () throws InterruptedException {
    System.out.println("Starting Facts App");
    socket = new TSocket();
    socket.registerListener(this);

    // Init the database
    db = new Hashtable <String, Integer> ();
    cache = new Hashtable <String, Integer> ();

    // Do NOT subclass FactsApp
    Thread t = new Thread(socket::start);
    t.setName("Facts App Thread");
    t.start();
    while (true) {
        Thread.sleep(1000L);
    }
}
```

Protocollo ABCI : convalida di una transizione

Nel paragrafo precedente, abbiamo visto gli effetti del comando *broadcast_tx_commit*:

curl -s 'localhost:26657/broadcast_tx_commit?tx="abcd"'

che ci consente di inviare una transazione e di ricevere la risposta dell'avvenuto inserimento nella chain. Quando questa richiesta viene inviata a un nodo, Tendermint Core la gestisce trasmettendola all'applicazione ABCI per la convalida. Il metodo che verrà invocato a questo scopo è **CheckTx**. Se

una transizione viene convalidata dal metodo CheckTx, verrà inclusa nel mempool (archivio temporaneo delle transizioni approvate) e trasmesso ad altri peer.

Il metodo ResponseCheckTx gestisce i messaggi CheckTx, in questo caso dovremmo solo preoccuparci di eseguire il parsing del fatto suddividendolo in un elemento source ed un elemento statement. Se il parsing avrà successo, il metodo deve restituire OK e la transizione verrà inviata in broadcast e sincronizzata con tutti i nodi della rete.

```
@Override
public ResponseCheckTx requestCheckTx (RequestCheckTx req) {
    ByteString tx = req.getTx();
    String payload = tx.toStringUtf8();
    System.out.println("Check tx : " + payload);
    if (payload == null || payload.isEmpty()) {
        return ResponseCheckTx.newBuilder().setCode(CodeType.BAD).setLog("payload is empty").build();
    }
    String [] parts = payload.split(":", 2);
    String source = "";
    String statement = "";
    try {
        source = parts[0].trim();
        statement = parts[1].trim();
        if (source.isEmpty() || statement.isEmpty()) {
            throw new Exception("Payload parsing error");
        }
    } catch (Exception e) {
        return ResponseCheckTx.newBuilder().setCode(CodeType.BAD).setLog(e.getMessage()).build();
    }
    System.out.println("The source is : " + source);
    System.out.println("The statement is : " + statement);
    System.out.println("The fact is in the right format!");
    return ResponseCheckTx.newBuilder().setCode(CodeType.OK).build();
}
```

In questo esempio il metodo è semplicissimo ma nella maggior parte dei casi il CheckTx utilizzerà lo stato corrente dell'applicazione, tratto dal suo database, per controllare la transazione. Quello che non può assolutamente fare questo metodo è modificare lo stato dell'applicazione in quando si tratta solo del primo step di approvazione.

Protocollo ABCI: validazione di un blocco

Abbiamo appena descritto e realizzato all'interno della nostra App una funzione per la convalida di una transizione con conseguente inserimento nella mempool del nodo. Le fasi che invece intervengono per la convalida di un blocco sono tre: **BeginBlock -> DeliverTx -> Commit.**

Quando la rete raggiunge il consenso sul prossimo blocco da inserire in chain, ogni blocco invierà le transizioni in questo blocco come una serie di messaggi DeliverTx all'applicazione ABCI. Il metodo ResponseDeliverTx gestisce i messaggi DeliverTx. Esegue nuovamente il parsing dei fatti e li salva, suddivisi per fonte, in una cache temporanea. Poiché tutti i nodi vedranno esattamente lo stesso

insieme di messaggi ed esattamente nello stesso ordine, essi aggiorneranno il database dell'applicazione in sequenza. Il DeliverTx aggiorna solo una copia temporanea del database, sarà poi il processo di Commit a rendere persistenti i dati trasferendoli nel database definitivo ed aggiornando così lo stato dell'applicazione. Ciò garantisce che lo stato del database dell'applicazione sia sempre sincronizzato con lo stato del commit dell'ultimo blocco.

```
@Override
public ResponseDeliverTx receivedDeliverTx (RequestDeliverTx req) {
    ByteString tx = req.getTx();
    String payload = tx.toStringUtf8();
    System.out.println("Deliver tx : " + payload);
    if (payload == null || payload.isEmpty()) {
        return ResponseDeliverTx.newBuilder().setCode(CodeType.BAD).setLog("payload is empty").build();
    }
    String [] parts = payload.split(":", 2);
    String source = "";
    String statement = "";
    try {
        source = parts[0].trim();
        statement = parts[1].trim();
        if (source.isEmpty() || statement.isEmpty()) {
            throw new Exception("Payload parsing error");
        }
    } catch (Exception e) {
        return ResponseDeliverTx.newBuilder().setCode(CodeType.BAD).setLog(e.getMessage()).build();
    }
    System.out.println("The source is : " + source);
    System.out.println("The statement is : " + statement);
    if (cache.containsKey(source)) {
        int count = cache.get(source);
        cache.put(source, count++);
        System.out.println("The count in this block is : " + count);
    } else {
        cache.put(source, 1);
        System.out.println("The count in this block is : " + 1);
    }
    System.out.println("The fact is validated by this node!");
    return ResponseDeliverTx.newBuilder().setCode(CodeType.OK).build();
}
```

Quando l'applicazione ABCI vede il messaggio Commit, salva tutti i conteggi temporanei nell'archivio permanente Hashtable e restituisce il codice hash dello stesso come app hash. Tutti i nodi devono concordare su questo app hash dopo il commit del blocco. Se un nodo restituisce un hash app differente da quello degli altri nodi, è considerato corrotto e non potrà più partecipare alle successive votazioni per il consenso.

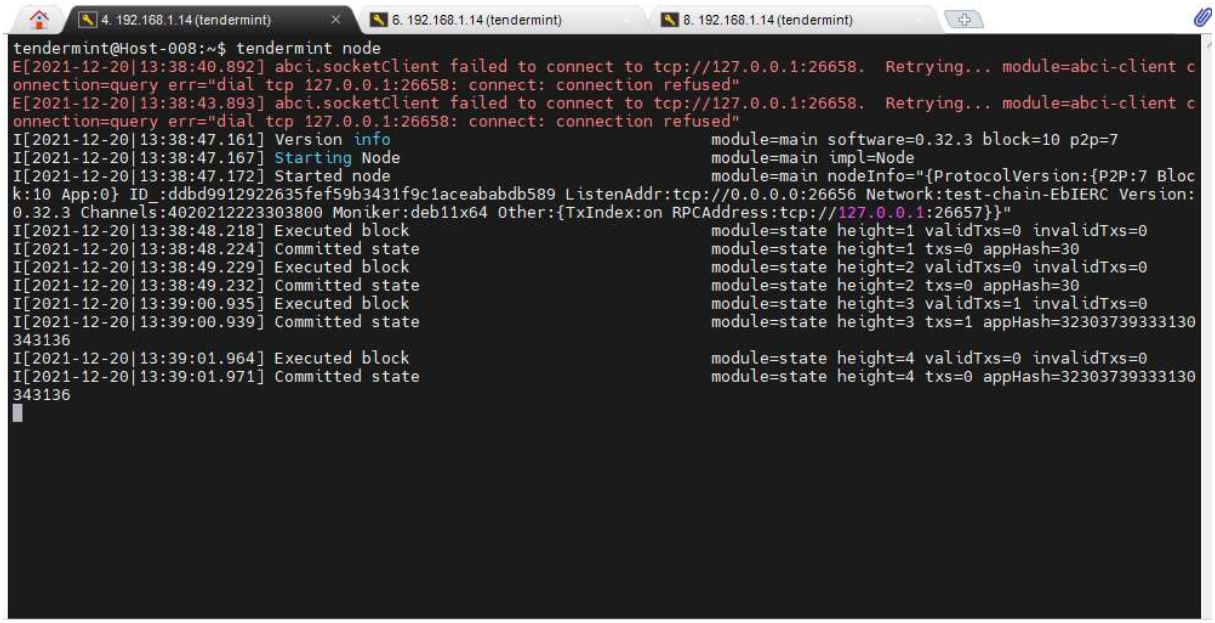
```
@Override
public ResponseCommit requestCommit (RequestCommit requestCommit) {
    System.out.println("Commit " + cache.keySet().size() + " items");
    Set<String> keys = cache.keySet();
    for (String source: keys) {
        System.out.println("Source : " + source);
        if (db.containsKey(source)) {
            db.put(source, cache.get(source) + db.get(source));
        } else {
            db.put(source, cache.get(source));
        }
    }
    cache.clear();
    return ResponseCommit.newBuilder().setData(ByteString.copyFromUtf8(String.valueOf(db.hashCode()))).build();
}
```

Protocollo ABCI: gestione delle query

Un client a questo punto non può semplicemente effettuare una query sul suo database ma deve usare Tendermint Core. In questo caso Tendermint passerà all'applicazione un messaggio Query. Il metodo ResponseQuery gestisce questo messaggio e restituisce i conteggi per tutte le fonti dell'archivio dati.

```
@Override
public ResponseQuery requestQuery (RequestQuery req) {
    String query = req.getData().toStringUtf8();
    System.out.println("Query : " + query);
    if (query.equalsIgnoreCase("all")) {
        StringBuffer buf = new StringBuffer ();
        String prefix = "";
        Set<String> keys = db.keySet();
        for (String source: keys) {
            buf.append(prefix);
            prefix = ",";
            buf.append(source).append(":").append(db.get(source));
        }
        System.out.println(buf.toString());
        return ResponseQuery.newBuilder().setCode(CodeType.OK).setValue(
                ByteString.copyFromUtf8((buf.toString()))
        ).setLog(buf.toString()).build();
    }
    if (query.startsWith("Source")) {
        String keyword = query.substring(6).trim();
        if (db.containsKey(keyword)) {
            System.out.println(keyword + " : " + db.get(keyword) + " | " + ByteString.copyFromUtf8(db.get(keyword).toString()).toString());
            return ResponseQuery.newBuilder().setCode(CodeType.OK).setValue(
                    ByteString.copyFromUtf8(db.get(keyword).toString())
            ).setLog(db.get(keyword).toString()).build();
        }
    }
    return ResponseQuery.newBuilder().setCode(CodeType.BadNonce).setLog("Invalid query").build();
}
```

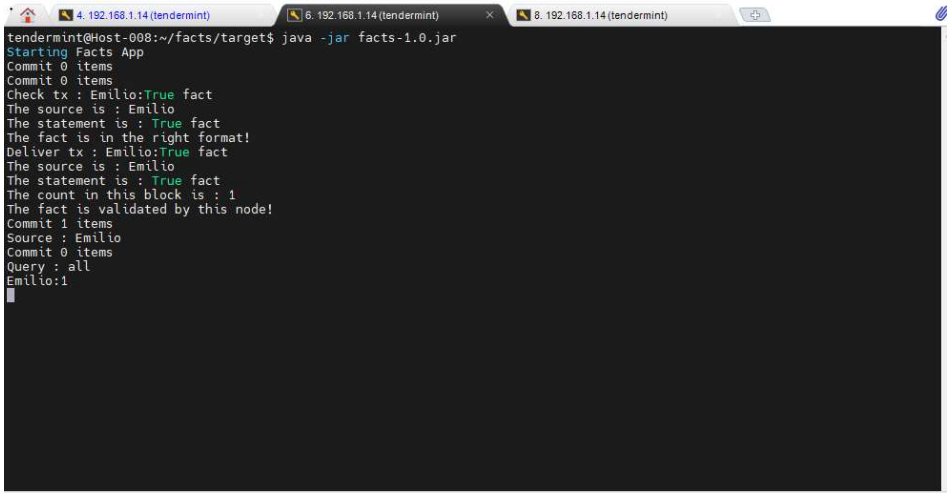
Per testare l'applicazione ABCI con un singolo nodo validatore, potremmo procedere come segue: si aprono tre console, su una si lancia il comando Tendermint node (ricordiamo che con Tendermint start lanciamo l'app pre configurata kvstore, mentre se utilizziamo app di terze parti comunichiamo attraverso RCP); su un'altra finestre si avvia l'applicazione ABCI ed infine nell'ultima finestra di lanciano le transizioni e le query.



Nell'immagine possiamo notare come Tendermint cerca inizialmente la connessione verso la socket 26658, ma una volta lanciata l'applicazione ABCI, si aggancia alla socket 26657 e non produce errori.

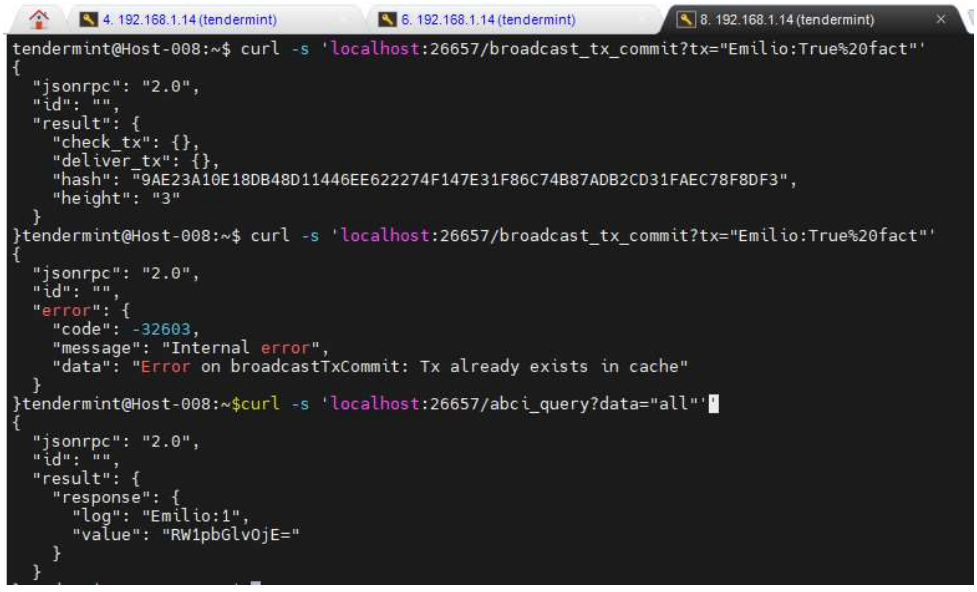
Ricordiamo che la compilazione dell'app attraverso Maven, viene generato l'eseguibile nella cartella targhet, per cui per lanciare l'applicazione ABCI occorre lanciare:

java –jar /facts/targhet/facts-1.0.jar



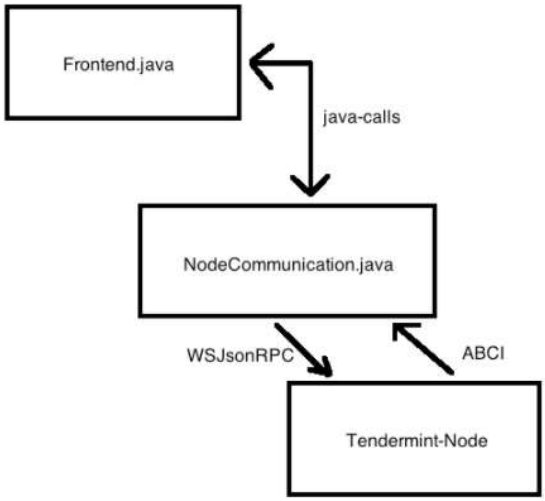
Su questa finestra verranno visualizzati gli output dell'applicazione ABCI.

Il passo successivo è realizzare un'app per interagire con la blockchain: una pagina HTML o una GUI per il dispositivo in uso. Nel nostro caso ci limitiamo ad inviare richieste attraverso il comando curl di linux



ABCI TMChat in java

L'applicazione TMChat è un altro esempio di applicazione ABCI scritta in java, in cui viene implementata anche una GUI, dato che si tratta di una chat. L'applicazione quindi può essere descritta dalla seguente struttura a blocchi :

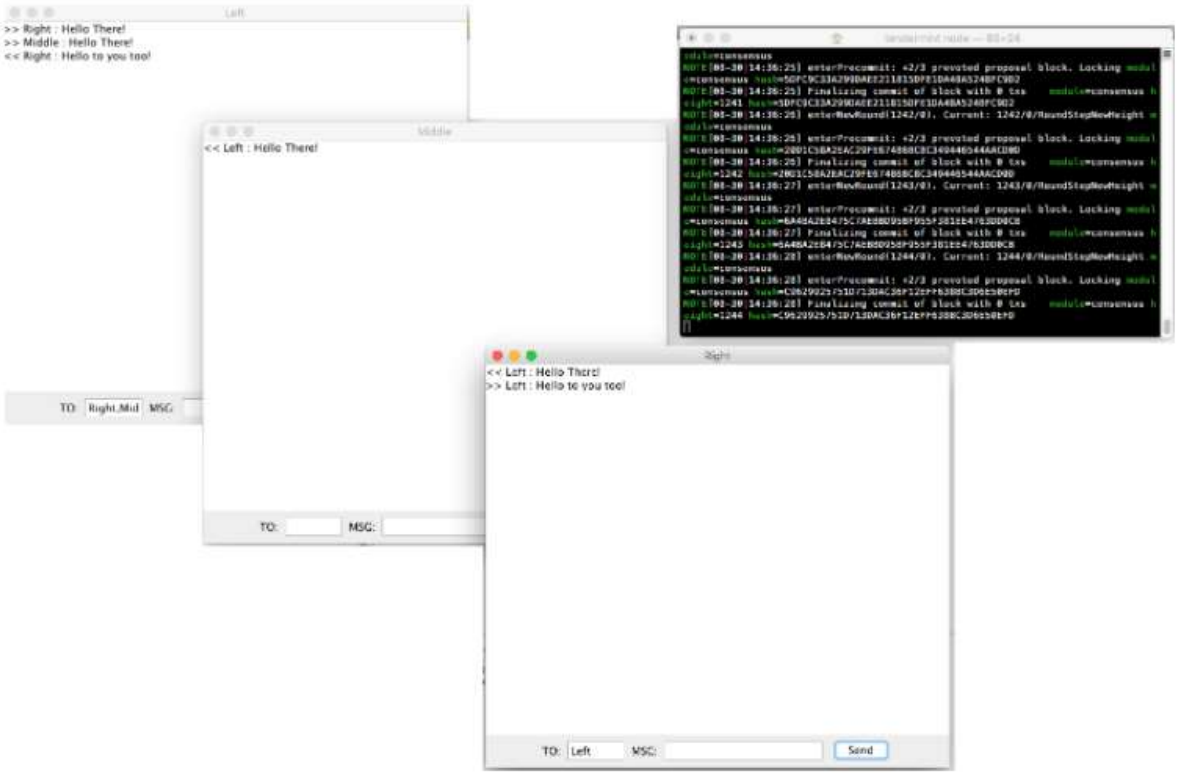


Questa applicazione è reperibile all'indirizzo:

https://github.com/wolfposd/TMChat.git

La parte di controllo è separata in due parti logiche, ciascuna al servizio di una diversa funzionalità. La classe NodeCommunication si connette al nodo Tendermint tramite il protocollo ABCI-Socket (usando jabci) per gestire tutti i messaggi Tendermint come AppendTX, CheckTx e Commit. Si connette anche tramite Websocket al nodo Tendermint per trasmettere i TX ricevuti dai suoi client connessi. NodeCommunication funge da proxy tra Tendermint e il Frontend. I blocchi NodeCommunication e Tendermint Core rappresentano nell'insieme una blockchain pubblica. TMChat, così come Facts sono blockchain pubbliche in quando hanno le seguenti caratteristiche:

- chiunque può diventare un nodo della rete, chiunque può leggere e inviare transazioni che poi saranno successivamente incluse e validate in un blocco, chiunque può essere un miner e per tale motivo partecipare al meccanismo di consenso offrendo volontariamente la propria potenza di calcolo
- Il codice utilizzato è open-source è verificabile ed utilizzabile da ogni elemento della rete
- è completamente decentralizzata



Per l'istallazione occorre seguire questi passaggi:

git clone https://github.com/wolfposd/TMChat.git
cd TMChat
mvn package -U

prima di eseguire la compilazione attraverso Maven, occorre verificare che nel file pom.xml sia definita l'ultima versione di jTendermint, ovvero la versione 0.32.3.

Come al solito, si consiglia di aprire due finestre (Promt dei comandi) per visualizzare il listato degli eventi dei due programmi. Nella prima finestra lanciare Tendermint Core attraverso il comando:

tendermint node

nella seconda finestra lanciare il comando:

java -cp bcprov-jdk15on-1.57.jar -jar tmchat-0.0.2-SNAPSHOT.jar Left Middle Righ

Rispetto all'esempio precedente, in questo caso il metodo ABCI ResponseCheckTx non esegue nessuna convalida, accetta tutte le transizioni che vengono inoltrate nella blockchain.

```
private Websocket wsClient;
private TSocket socket;

private Gson gson = new Gson();
private int hashCount = 0;

private Map<String, FrontendListener> frontends = new HashMap<>();

public NodeCommunication() {

    wsClient = new Websocket(this);
    socket = new TSocket();
    socket.registerListener(this);
    new Thread(socket::start).start();
    System.out.println("Started ABCI Socket Interface");

    System.out.println("Now waiting on ABCI-Sockets before connecting to Websocket...");

    // need atleast 3 socket connections: info,mempool,consensus
    while(socket.sizeOfConnectedABCISockets() < 3) {
        sleep(2000);
        System.out.println("sleeping 2 seconds, to wait for ABCI-connections");
    }
```

Possiamo subito notare che rispetto all'implementazione precedente, in questa oltre a Tsocket (connessione verso Tendermint Core) abbiamo una Websocket su cui apriamo delle connessioni sempre attive verso le applicazioni client.

Protocollo ABCI: convalida di una transizione

Quando una richiesta viene inviata a un nodo, Tendermint Core la gestisce trasmettendola all'applicazione ABCI per la convalida. Il metodo che verrà invocato a questo scopo è **CheckTx**. Se una transizione viene convalidata dal metodo CheckTx, verrà inclusa nel mempool (archivio temporaneo delle transizioni approvate) e trasmesso ad altri peer.

Il metodo ResponseCheckTx gestisce i messaggi CheckTx, in questo caso non viene eseguito il parsing, quindi ogni transazione avrà successo. Il metodo restituire OK e la transizione verrà inviata in broadcast e sincronizzata con tutti i nodi della rete.

```
@Override
public ResponseCheckTx requestCheckTx(RequestCheckTx req) {
    return ResponseCheckTx.newBuilder().setCode(CodeType.OK).build();
}
```

Protocollo ABCI: validazione di un blocco

Abbiamo appena descritto e realizzato all'interno della nostra App una funzione per la convalida di una transizione con conseguente inserimento nella mempool del nodo. Le fasi che invece intervengono per la convalida di un blocco sono tre: **BeginBlock -> DeliverTx -> Commit.**

Quando la rete raggiunge il consenso sul prossimo blocco da inserire in chain, ogni blocco invierà le transizioni in questo blocco come una serie di messaggi DeliverTx all'applicazione ABCI. Il metodo ResponseDeliverTx gestisce i messaggi DeliverTx. Poiché tutti i nodi vedranno esattamente lo stesso insieme di messaggi ed esattamente nello stesso ordine, essi aggiorneranno il database dell'applicazione in sequenza. Allo stesso modo del metodo ResponseCheckTx, il metodo ResponseDeliverTx in questo esempio non esegue nessun controllo sui messaggi del mempool e verifica soltanto se il messaggio è stato ricevuto dal destinatario.

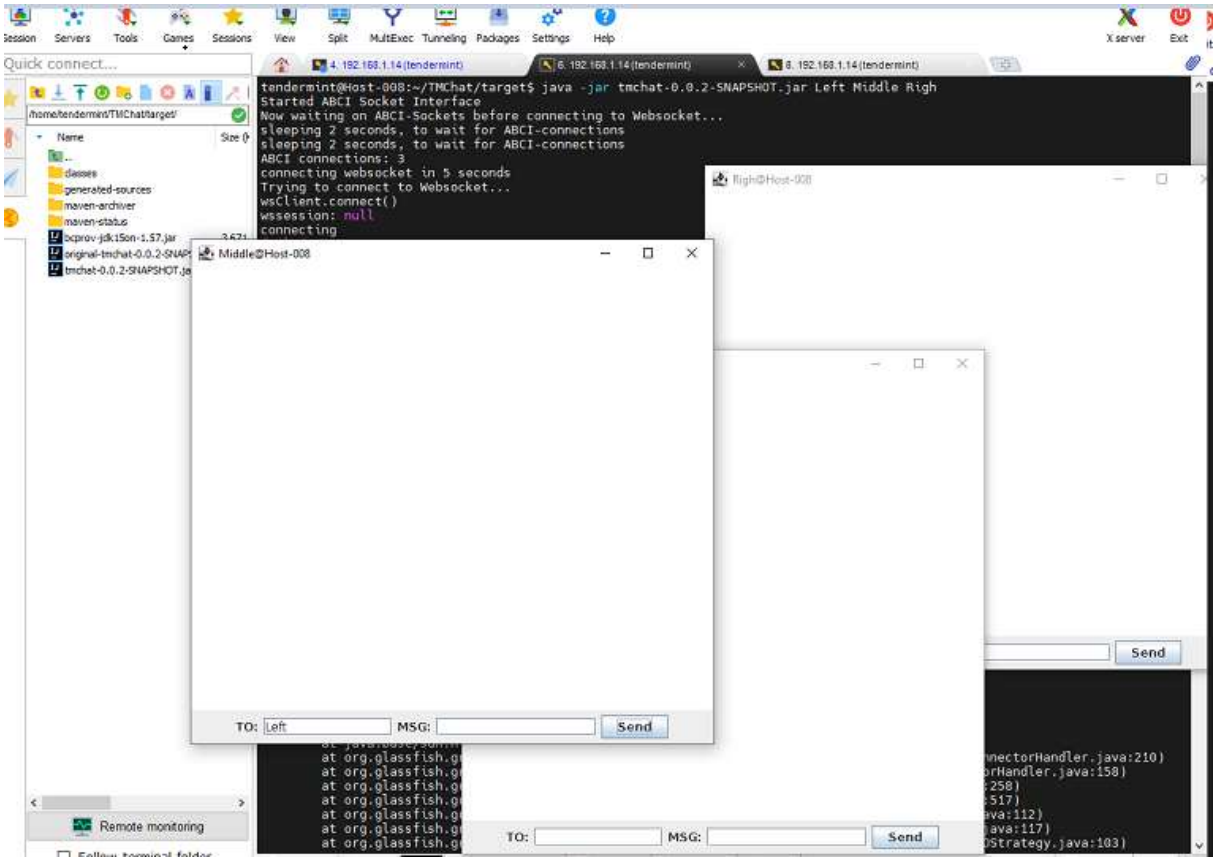
```
@Override
public ResponseDeliverTx receivedDeliverTx(RequestDeliverTx req) {

    byte[] byteArray = req.getTx().toByteArray();
    Message msg = gson.fromJson(new String(byteArray), Message.class);

    FrontendListener l = frontends.get(msg.receiver);
    if (l != null) {
        l.messageIncoming(msg);
    }

    return ResponseDeliverTx.newBuilder().setCode(CodeType.OK).build();
}
```

Quando l'applicazione ABCI vede il messaggio Commit, salva tutti i conteggi temporanei nell'archivio permanente e restituisce il codice hash dello stato corrente come app hash. Tutti i nodi devono concordare su questo app hash dopo il commit del blocco. Se un nodo restituisce un hash app differente da quello degli altri nodi, è considerato corrotto e non potrà più partecipare alle successive votazioni per il consenso.

```
@Override
public ResponseCommit requestCommit(RequestCommit requestCommit) {
    hashCount += 1;
    return ResponseCommit.newBuilder().setData(ByteString.copyFrom(ByteUtil.toBytes(hashCount))).build();
}
```

Protocollo ABCI: gestione delle query

Un client dell'applicazione potrebbe in linea teorica interrogare il proprio database o stato interno per conoscere le informazioni archiviate ma questo non è il modo corretto di procedete in quando lo stato potrebbe essere inconsistente a seguito di un processo di approvazione in corso. Quindi è consigliabile utilizzare sempre il metodo query messo a disposizione dalla piattaforma. In questo caso Tendermint passerà all'applicazione un messaggio Query. Il metodo ResponseQuery in questo caso non viene realizzato in quando i messaggi vengono gestiti diversamente.

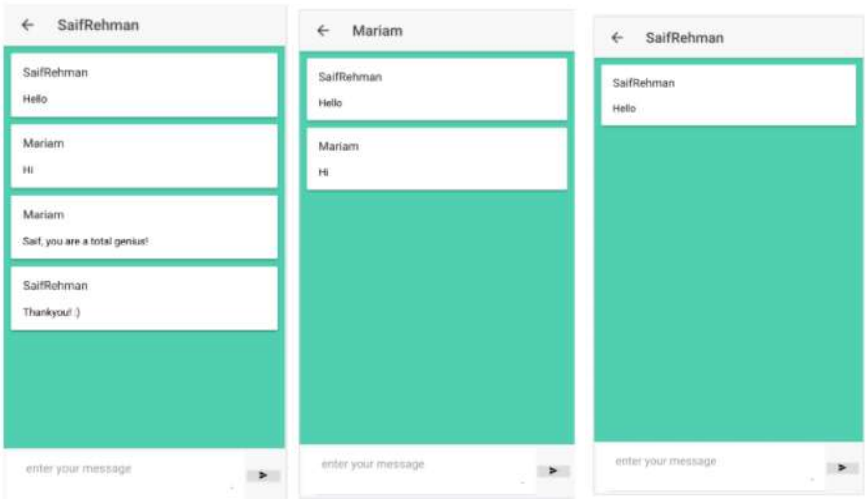


Una implementazione complete è reperibile al seguente link:

https://github.com/SaifRehman/tendermint-chat-app/tree/master/frontend

mentre al seguente indirizzo è disponibile una versione per android:

https://volontariat.vercel.app/post/tmchat/



ABCI Counter in Python

Nei paragrafi precedenti abbiamo visto due esempi di realizzazione del modulo ABCI utilizzando la libreria jTendermint e quindi sviluppando applicazioni specifiche per blockchain utilizzando il linguaggio Java.

In questo paragrafo vediamo come in maniera del tutto analoga è possibile sviluppare applicazioni distribuite utilizzando altri linguaggi, come ad esempio Python.

A questo scopo utilizzeremo la libreria py-abci raggiungibile sul sito:

https://github.com/davebryson/py-abci

Per funzionare è richiesto Tendermint *0.34.11* e Python >= 3.9.

Mentre per Tendermint, si può installare la versione master o in alternativa si può scaricare direttamente l'eseguibile ed inserendolo nella cartella usr/local/bin in modo che sia possibile lanciarlo da ogni cartella, per la versione Python richiesta occorre impostare il compilatore in modo corretto in quando di default Ubuntu ha settato Python 3.6.

In questo test lavoreremo direttamente su un ambiente virtuale utilizzando un container Docker. Per semplicità, elenchiamo l'insiemi dei comandi Docker che ci saranno utili per gestire un ambiente virtuale di esecuzione:

docker build --tag abci -f abci .	crea l'immagine abci a partire dal dockerfile abci
docker image ls	visualizza lista immagini
docker ps -a	visualizza lista container
docker run -it 81a41ea2afec	crea un container passandi l'id immagine

docker exec -it container command /bin/bash	esegue il container e lancia un comando all'avvio
docker rm 77c504449d1fa	rimuove un container
docker rmi 7d823fd330dd	rimuove una immagine
docker stop $(docker ps -aq)	stoppa tutti i container
docker rm $(docker ps -aq)	Remove all containers
docker rmi $(docker images -q)	Remove all images
docker start 3dcc14b4dbeb	avvia un container (ID container)
docker stop 3dcc14b4dbeb	termina un container
docker pull jaswanthv/python3.6-custom	scarica una immagine
docker rm abci{0,1,2,3}	cancella tutti i container abci
docker-compose up	esegue il composer
docker-compose down	stoppa il composer
docker-compose ps	lista dei composer run
docker network inspect tendermint_localnet	verifica rete docket

Quindi procediamo nel creare lo script che genererà l'immagine Docker contenente Tendermint Core e l'applicazione Counter.py contenuta nella libreria py-abci. Si suggerisce di scaricare una immagine ubuntu o altra versione linux, creare un container di prova ed effettuare le installazioni direttamente nel container in modo da verificare in corso d'opera i vari problemi di installazione. Una volta risolto i vari conflitti, dipendenze, etc. la sequenza di comandi adoperati per settare l'ambiente virtuale si va a replicare all'interno dello script Dockerfile. L'immagine Docker che vogliamo ottenere è la seguente:

1. s.o. ubuntu

2. tendermint_0.34.11_linux_amd64 nella cartella /usr/local/bin per essere visibile nella root

3. python3.9 ed python3-pip come compilatore di default

4. libreria abci Python

5. inizializziamo un nodo Tendermint come validatore ed eseguiamo l'avvio del servizio

Il file di script che chiameremo pythondocker sarà quindi il seguente:

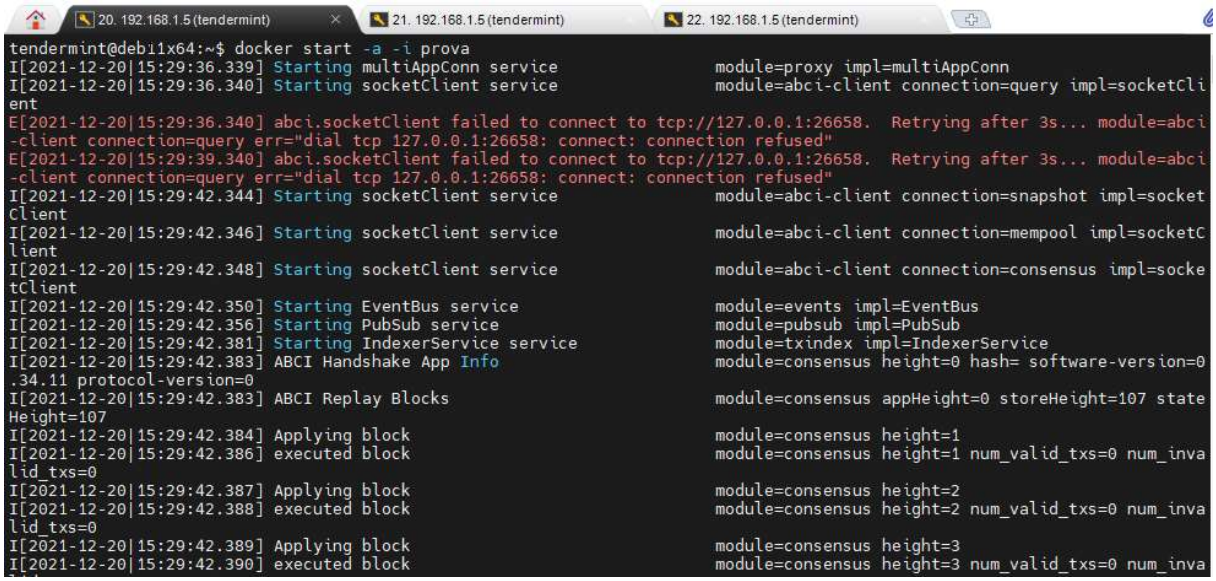
```
FROM ubuntu

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y apt-utils curl git tar gzip
RUN apt-get install -y wget && wget https://github.com/tendermint/tendermint/releases/download/v0.34.11/tendermint_0.34.11_linux_amd64.tar.gz
RUN tar xvzf tendermint_0.34.11_linux_amd64.tar.gz && rm tendermint_0.34.11_linux_amd64.tar.gz && mv tendermint /usr/local/bin
RUN apt-get -y install python3.9 python3-pip mlocate net-tools tmux screen
RUN unlink /usr/bin/python3 && ln -s /usr/bin/python3.9 /usr/bin/python3 && python3 --version
RUN tendermint init
RUN pip install abci

VOLUME [ /tendermint ]
WORKDIR /tendermint

# p2p, rpc and prometheus port
EXPOSE 26656 26657 26660

CMD ["tendermint", "node"]
```

Come consuetudine, creiamo l'immagine, il container e lo mandiamo in esecuzione:

docker build --tag tendermint/python -f pythondocker . ----> cera l'immagine tendermint/python
docker run --name prova -it tendermint/python ----------> crea il container prova

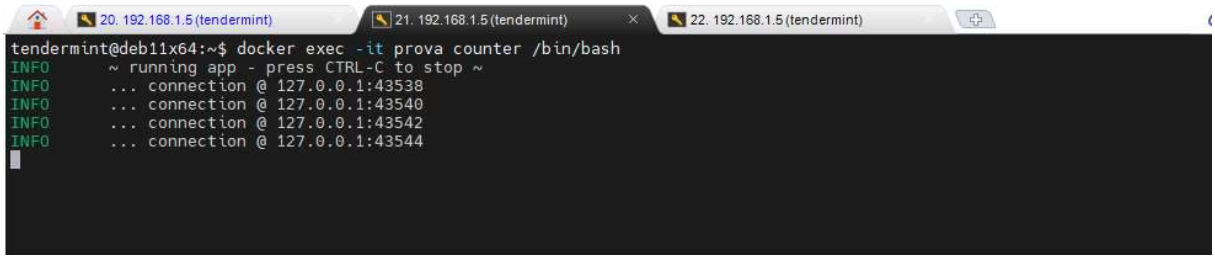
Entriamo quindi nel container col comando:

docker start -a -i prova



Troveremo già in esecuzione il server Tendermint che mostrerà errori di connessione con l'applicazione ABCI ancora non in esecuzione.
Apriamo una seconda finestra del container e lanciamo l'applicazione ABCI con il comando:

docker exec -it prova counter /bin/bash



L'app Counter è stata inserita nell'elenco delle librerie Python3.9 durante l'installazione per cui non è necessario specificare il compilatore.

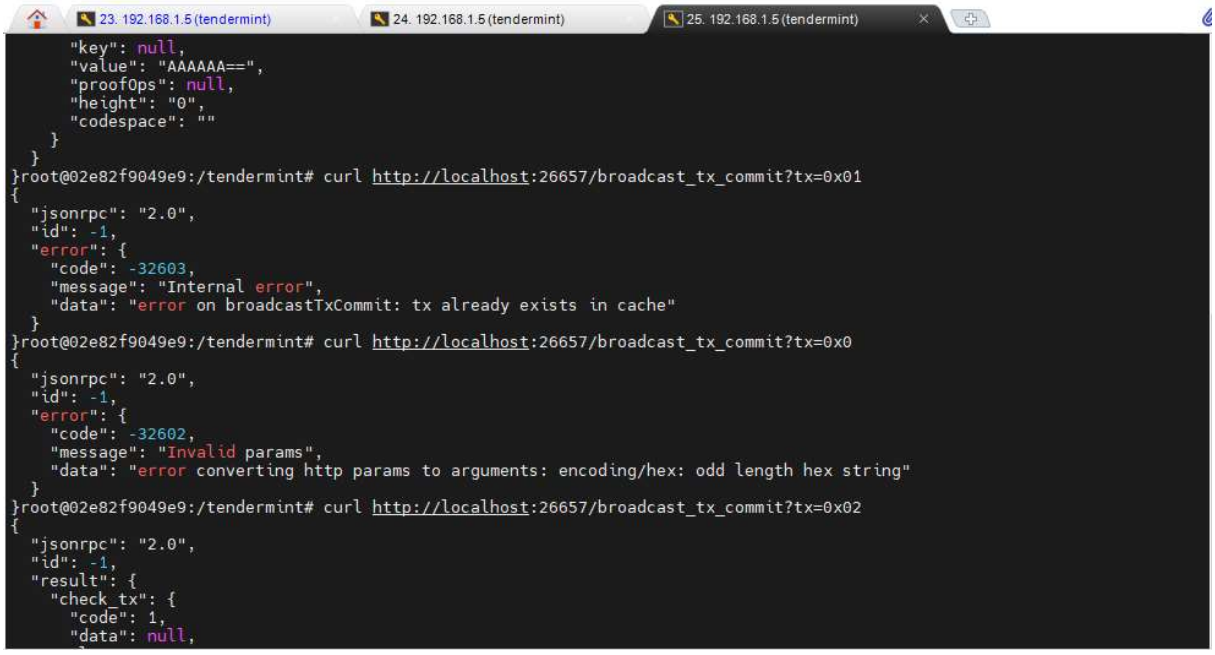
Dai log che vengono visualizzati dal prompt dei comandi è possibile constatare che sono state aperte quattro connessioni socket tra l'applicazione ABCI e Tendermint (ricordiamo che ogni una di queste è delegata a gestire i diversi messaggi del protocollo ABCI).

A questo punto possiamo procedere con l'inoltro delle transizioni. Per fare ciò apriamo un'altra istanza Docker:

docker exec -it prova /bin/bash

e procediamo con l'inoltro dei comandi attravrso il comando curl di linux.

curl http://192.168.172.6:26657/broadcast_tx_commit?tx=0x01
curl http://localhost:26657/broadcast_tx_commit?tx=0x05
curl http://localhost:26657/abci_query



L'applicazione counter.py è una semplice app di conteggio, accetta solo valori inviati nell'ordine corretto. Lo stato mantiene il conteggio corrente. Ad esempio, partendo dallo stato 0, inviando:

-> 0x01 = OK!

-> 0x03 = Fallirà! (si aspetta 2)

Riportiamo di seguito le quattro connessioni ed i relativi messaggi gestiti dal protocollo ABCI

Mempool connection

- è responsabile della convalida di nuove transazioni, prima che queste vengano condivise o incluse in un blocco.
- gestisce le chiamate CheckTx

Consensus connection

- è responsabile della gestione dei blocchi
- gestisce la richiesta InitChain, BeginBlock, DeliverTx, EndBlock, e Commit.

Info connection

- si occupa dell'inizializzazione e delle query utente.
- gestisce le chiamate Info e Query.

Snapshot connection

- serve a supportare la sincronizzazione dello stato.
- gestisce le chiamate ListSnapshots, LoadSnapshotChunk, OfferSnapshot, e ApplySnapshotChunk.

Mempool Connection: convalida di una transizione

Quando una richiesta viene inviata a un nodo, Tendermint Core la gestisce trasmettendola all'applicazione ABCI per la convalida. Il metodo che verrà invocato a questo scopo è **CheckTx**. Se una transizione viene convalidata dal metodo CheckTx, verrà inclusa nel mempool (archivio temporaneo delle transizioni approvate) e trasmesso ad altri peer.

Il metodo ResponseCheckTx gestisce i messaggi CheckTx, in questo caso viene eseguito il parsing, e verificato se viene inviato un numero. Il metodo restituire OK se il numero trasmesso corrisponde allo stato precedente, incrementato di una unità. La transizione verrà inviata in broadcast e sincronizzata con tutti i nodi della rete.
Nell'app Counter troveremo l'override di sei dei dodici metodi. Riguardo alla mempool connection verrà implementato il metodo ResponseCheckTx che gestisce i messaggi di tipo CheckTx per la convalida delle transizioni.

```
def check_tx(self, tx) -> ResponseCheckTx:
    """
    Validate the Tx before entry into the mempool
    Checks the txs are submitted in order 1,2,3...
    If not an order, a non-zero code is returned and the tx
    will be dropped.
    """
    value = decode_number(tx)
    if not value == (self.txCount + 1):
        return ResponseCheckTx(code=ErrorCode)
    return ResponseCheckTx(code=OkCode)
```

Consensus Connection: validazione di un blocco

Abbiamo appena descritto e realizzato all'interno della nostra App una funzione per la convalida di una transizione con conseguente inserimento nella mempool del nodo. Le fasi che invece intervengono per la convalida di un blocco sono tre: **BeginBlock -> DeliverTx -> Commit.**

Quando la rete raggiunge il consenso sul prossimo blocco da inserire in chain, ogni blocco invierà le transizioni in questo blocco come una serie di messaggi DeliverTx all'applicazione ABCI. Il metodo ResponseDeliverTx gestisce i messaggi DeliverTx. Poiché tutti i nodi vedranno esattamente lo stesso insieme di messaggi ed esattamente nello stesso ordine, essi aggiorneranno il database dell'applicazione in sequenza.

Quando l'applicazione ABCI vede il messaggio Commit, restituisce il codice hash dell'ultimo numero validato come app hash. Tutti i nodi devono concordare su questo app hash dopo il commit del blocco. Se un nodo restituisce un hash app differente da quello degli altri nodi, è considerato corrotto e non potrà più partecipare alle successive votazioni per il consenso.

Il metodo ResponseInitChain non fa altro che inizializzare lo stato interno dell'app, viene chiamato quando un nuovo nodo validatore si inserisce nella chain:

```
def init_chain(self, req) -> ResponseInitChain:
    """Set initial state on first run"""
    self.txCount = 0
    self.last_block_height = 0
    return ResponseInitChain()
```

Mentre ResponseDeliverTx si occupa di dare il consenso per l'inserimento di un nuovo blocco:

```
def deliver_tx(self, tx) -> ResponseDeliverTx:
    """
    We have a valid tx, increment the state.
    """
    self.txCount += 1
    return ResponseDeliverTx(code=OkCode)
```

In questo caso non esegue nessun ulteriore controllo sulle transizioni ricevute dal mempool e modifica lo stato dell'applicazione (incrementando il conteggio), cosa che come abbiamo detto in precedenza non è consigliabile fare, piuttosto questa operazione va delegata alla gestione del commit, che in questo caso si limita a calcolare solo l'hash app come vediamo di seguito.

```
def commit(self) -> ResponseCommit:
    """Return the current encode state value to tendermint"""
    hash = struct.pack(">Q", self.txCount)
    return ResponseCommit(data=hash)
```

Info Connection: gestione delle query

Riguardo alla Info connection vengono implementati entrambi i metodi ResponseInfo ed ResponseQuery.

```
def info(self, req) -> ResponseInfo:
    """
    Since this will always respond with height=0, Tendermint
    will resync this app from the begining
    """
    r = ResponseInfo()
    r.version = req.version
    r.last_block_height = 0
    r.last_block_app_hash = b""
    return r

def query(self, req) -> ResponseQuery:
    """Return the last tx count"""
    v = encode_number(self.txCount)
    return ResponseQuery(code=CodeTypeOk, value=v, height=self.last_block_height)
```

Per eseguire l'app counter su una rete Docker, occorrerà utilizzare il seguente file di script:

```
version: '3'
services:
  node0:
    container_name: node0
    image: "tendermint/localnode"
    ports:
      - "26656-26657:26656-26657"
      - "6060:6060"
      - "27000:26660"
    environment:
      - ID=0
      - LOG=${LOG:-tendermint.log}
    volumes:
      - ./build:/tendermint:Z
    command: node --proxy_app==tcp://abci0:26658 --consensus.create_empty_blocks=false
    #command: node --proxy_app=kvstore --consensus.create_empty_blocks=false
    networks:
      localnet:
        ipv4_address: 192.167.10.2


  abci0:
    container_name: abci0
    image: "abci"
    command: counter
    networks:
      localnet:
        ipv4_address: 192.167.10.6


networks:
  localnet:
    driver: bridge
    ipam:
      driver: default
      config:
      -
        subnet: 192.167.10.0/16
```

Dove riportiamo, per semplicità, la definizione dei primi due container, Tendermint e app, la definizione degli altri container è simile.

ABCI kvstore in Python

L'applicazione kvstore è una semplice applicazione che immagazzina transizioni all'interno di un merkle tree.

I merkle tree sono una struttura dati creata con l'obiettivo di facilitare la verifica di grandi quantità di dati organizzati mettendoli in relazione attraverso varie tecniche crittografiche e di gestione delle informazioni. È una struttura dati suddivisa in diversi livelli il cui scopo è metterli in relazione con un'unica radice associata ad essi. Per ottenere ciò, ogni nodo deve essere identificato con un

identificatore univoco. Questi nodi iniziali, chiamati nodi figli (foglie), sono quindi associati a un nodo non superiore chiamato nodo genitore (ramo). Il nodo padre avrà un identificatore univoco risultante dall'hash dei suoi nodi figlio.

Le transazioni della forma chiave=valore vengono archiviate come coppie chiave-valore nell'albero. Le transazioni senza segno = impostano sia la chiave che il valore sul valore dato. L'app non ha protezione per le repliche (a parte quella fornita dal mempool).

L'app è scaricabile all'indirizzo:

https://github.com/tendermint/tendermint/tree/e6a6c6523eae02ce62e08c107dbd27f33b79a107/abci/example/python3_kvstore

è compatibile con Python 3.6.7 ed abci==0.6.0 ma con qualche piccola modifica è possibile renderlo compatibile con la versione Tendermint *0.34.11,* Python >= 3.9 ed abci *0.17.0.*
Si consiglia di seguire l'installazione in un container, come è stato fatto nell'esempio precedente e testarlo con un singolo nodo, prima di testare il funzionamento su una reta di nodi.
Le modifiche da apportare al file sono riportate di seguito. L'import delle librerie abci naturalmente cambiano, quindi occorre sostituire le seguenti linee di codice:

```
from abci import (
    ABCIServer,
    BaseApplication,
    ResponseInfo,
    ResponseInitChain,
    ResponseCheckTx, ResponseDeliverTx,
    ResponseQuery,
    ResponseCommit,
    CodeTypeOk,
)
```

con:

```
from tendermint.abci.types_pb2 import (
    ResponseInfo,
    ResponseInitChain,
    ResponseCheckTx, ResponseDeliverTx,
    ResponseQuery,
    ResponseCommit
)

from abci.server import ABCIServer
from abci.application import BaseApplication, OkCode, ErrorCode
```

Ed aggiungere la direttiva:

```
logging.getLogger('asyncio').setLevel(logging.WARNING)
```

Sostituire le occorrenze di:

code=CodeTypeOk con code=OkCode

Utilizzando un container Docker occorre quindi:

1. Creare un'immagine: docker build --tag abci -f abcipy39 .
2. Lanciare il container: docker run --name prova2 -it abci
3. Eseguire l'applicazione: python3 py-abci/src/example/kvstore.py

Il Dockerfile per creare l'immagine per i test è la seguente:

```
FROM ubuntu
ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y apt-utils curl git tar gzip sed wget python3.9\
    python3-pip python3-dev build-essential libpython3.9-dev libhdf5-*  libssl-dev

# install dependencies from debian packages
RUN apt-get update -qq \
 && apt-get install --no-install-recommends -y \
    python3-matplotlib \
    python3-pillow \
    python3-pip\
    python3-dev \
    git

# install dependencies from python packages
RUN pip3 --no-cache-dir install \
    pandas \
    scikit-learn \
    statsmodels

# install tendermint
RUN wget https://github.com/tendermint/tendermint/releases/download/v0.34.11/tendermint_0.34.11_linux_amd64.tar.gz
RUN tar xvzf tendermint_0.34.11_linux_amd64.tar.gz && rm tendermint_0.34.11_linux_amd64.tar.gz && mv tendermint /usr/local/bin
RUN tendermint init

RUN unlink /usr/bin/python3 && ln -s /usr/bin/python3.9 /usr/bin/python3 && python3 --version
RUN pip install pysha3 gevent python-snappy cffi rlp==0.4.7 trie==0.2.4 protobuf
RUN yes | pip3 uninstall eth-utils
RUN pip3 install eth-utils

RUN git clone https://github.com/davebryson/py-abci.git
COPY kvstore.py py-abci/src/example/
RUN cd py-abci && pip install --editable '.[test]'

# ricompila l'applicazione
RUN cd py-abci && python3 setup.py install
```

Mempool Connection: convalida di una transizione

Quando una richiesta viene inviata a un nodo, Tendermint Core la gestisce trasmettendola all'applicazione ABCI per la convalida. Il metodo che verrà invocato a questo scopo è **CheckTx**. Se una transizione viene convalidata dal metodo CheckTx, verrà inclusa nel mempool (archivio temporaneo delle transizioni approvate) e trasmesso ad altri peer.

Il metodo ResponseCheckTx gestisce i messaggi CheckTx, in questo non viene eseguito il parsing. Il metodo restituire OK ad ogni transizione inviata. La transizione verrà inviata in broadcast e sincronizzata con tutti i nodi della rete.

```
def check_tx(self, tx):
    return ResponseCheckTx(code=OkCode)
    # return ResponseCheckTx(code=ErrorCode)
```

Consensus Connection: validazione di un blocco

Abbiamo appena descritto e realizzato all'interno della nostra App una funzione per la convalida di una transizione con conseguente inserimento nella mempool del nodo. Le fasi che invece intervengono per la convalida di un blocco sono tre: **BeginBlock -> DeliverTx -> Commit.**

Quando la rete raggiunge il consenso sul prossimo blocco da inserire in chain, ogni blocco invierà le transizioni in questo blocco come una serie di messaggi DeliverTx all'applicazione ABCI. Il metodo ResponseDeliverTx gestisce i messaggi DeliverTx. Esegue la scomposizione del messaggio estraendo chiave e valore. Poiché tutti i nodi vedranno esattamente lo stesso insieme di messaggi ed esattamente nello stesso ordine, essi aggiorneranno il database dell'applicazione in sequenza.

```
def deliver_tx(self, tx):
    """Validate the transaction before mutating the state.

    Args:
        raw_tx: a raw string (in bytes) transaction.
    """
    logger.info("Transaction recived %s", tx)

    parts = tx.split(b'=')
    if len(parts) == 2:
        key, value = parts[0], parts[1]
    else:
        key, value = tx
    self.state.db.set(prefix_key(key), value)
    self.state.size += 1

    logger.info("Transaction delivered succesfully")
    return ResponseDeliverTx(code=OkCode)
```

Quando l'applicazione ABCI vede il messaggio Commit, aggiorna lo stato e restituisce il codice hash corrispondente alla lunghezza dell'archivio come app hash. Tutti i nodi devono concordare su questo app hash dopo il commit del blocco. Se un nodo restituisce un hash app differente da quello degli altri nodi, è considerato corrotto e non potrà più partecipare alle successive votazioni per il consenso.

```
def commit(self):
    byte_length = max(ceil(self.state.size.bit_length() / 8), 1)
    app_hash = self.state.size.to_bytes(byte_length, byteorder='big')
    self.state.app_hash = app_hash
    self.state.height += 1
    self.state.save()
    return ResponseCommit(data=app_hash)
```

Info Connection: gestione delle query

Riguardo alla Info connection vengono implementati entrambi i metodi ResponseInfo ed ResponseQuery**.**

```
def info(self, req):
    """
    Since this will always respond with height=0, Tendermint
    will resync this app from the begining
    """
    r = ResponseInfo()
    r.version = "1.0"
    r.last_block_height = self.state.height
    r.last_block_app_hash = b''
    return r
```

```
def query(self, req):
    if req.prove:
        if self.state.db.exists(prefix_key(req.data))
            value = self.state.db.get(prefix_key(req.data))
            return ResponseQuery(code=OkCode, value=value)
    return ResponseQuery(code=ErrorCode)
```