Parallel Asynchronous Particle Swarm Optimization over Blockchain-based framework

Emilio Greco

RT- ICAR-CS-21-09 **Novembre 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 ... 3
Introduzione ... 4
Blockchain ... 7
Tendermint ... 15
  Avvio di Tendermint Core ... 21
  Resettare la chain ... 26
  Debug ... 26
  Local Testnet ... 27
  Distributed Testnet ... 29
  Configuration of a Distributed Testnet ... 31
  Example Distributed Testnet : kvstore ... 31
Parallel Asynchronous Particle Swarm Optimization ... 37
  Basic PSO ... 37
  Java Code ... 45
  Parallel Particle Swarm Optimization ... 49
Protocollo ABCI ... 52
Architettura del sistema ... 53
Implementazione dell'ABCI in Java ... 55
  Protocollo ABCI: convalida di una transizione ... 56
  Protocollo ABCI: validazione di un blocco ... 57
  Protocollo ABCI: gestione delle query ... 59
Implementazione della Dapp in Java ... 59
  Example Distributed Testnet : JAVA PSO ... 66

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 analizzeremo e svilupperemo una soluzione al problema del PSO utilizzando la tecnologia emergente delle Dapp (Decentralized applications) su tecnologia 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'intero 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 algoritmi di swarm intelligence come l'algoritmo Particle Swarm Optimization nelle sue diverse implementazioni e ne mostreremo una implementazione utilizzando l'infrastruttura Blockchain offerta da 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 contracts. 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 criptovaluta 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.

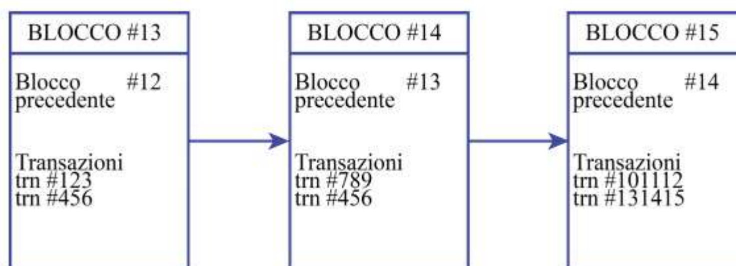


Figura 1- esempio di catena a blocchi

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);
- i 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 Contract. 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 ed 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.

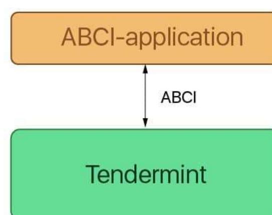


Figura 2 - Struttura di una blockchain basata su Tendermint

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:

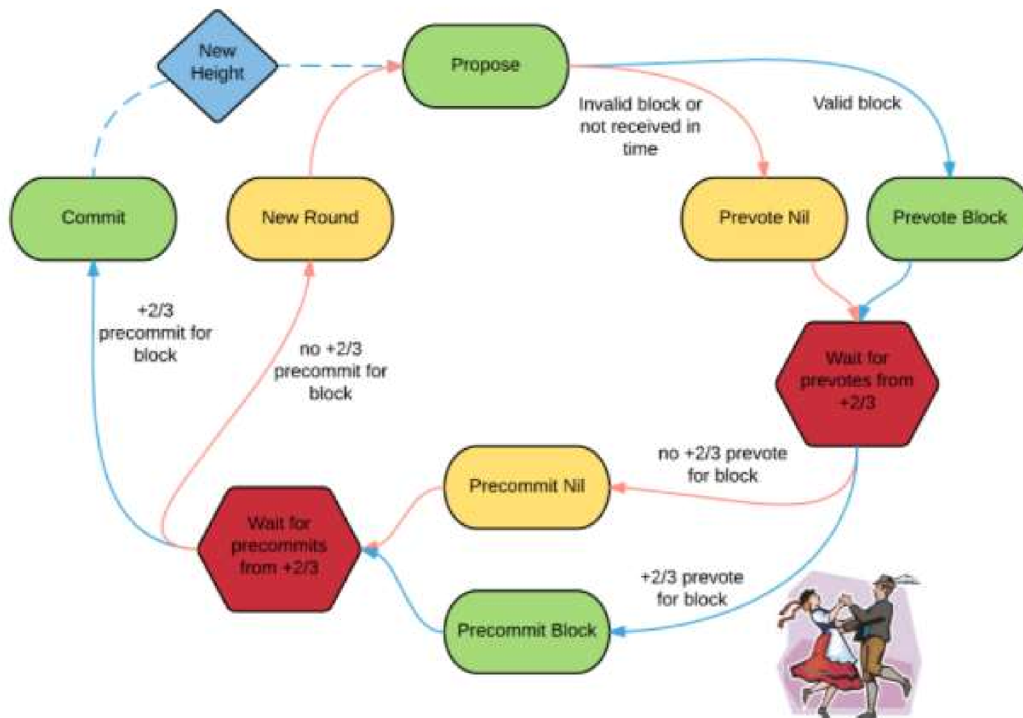


Figura 3- Algoritmo approvazione di un blocco

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.

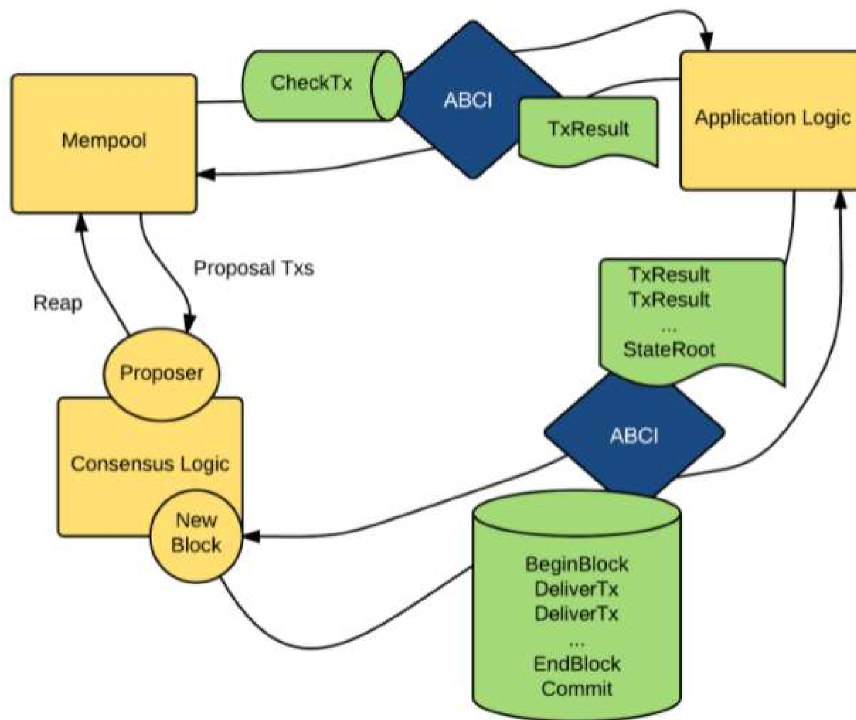


Figura 4 - Flusso dei messaggi dei tre layer applicativi

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:

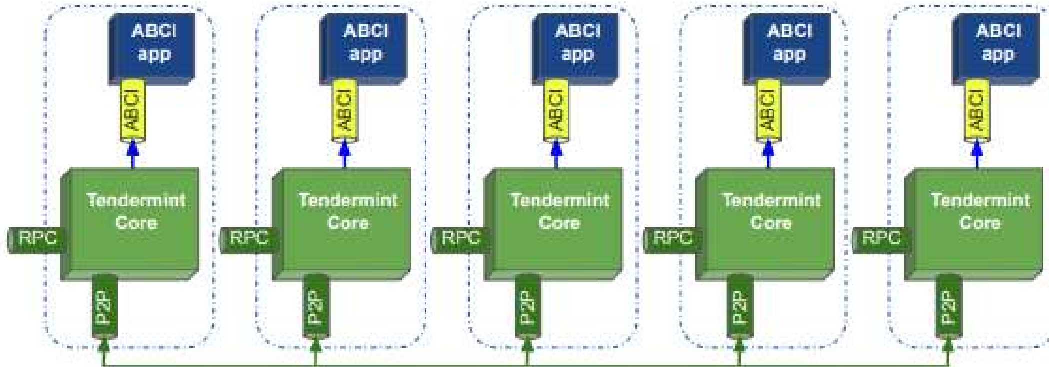


Figura 5 - struttura di una rete blockchain Tendermint

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.

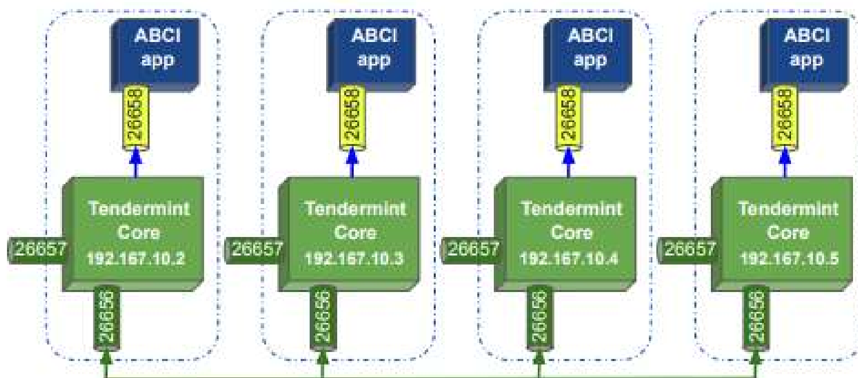
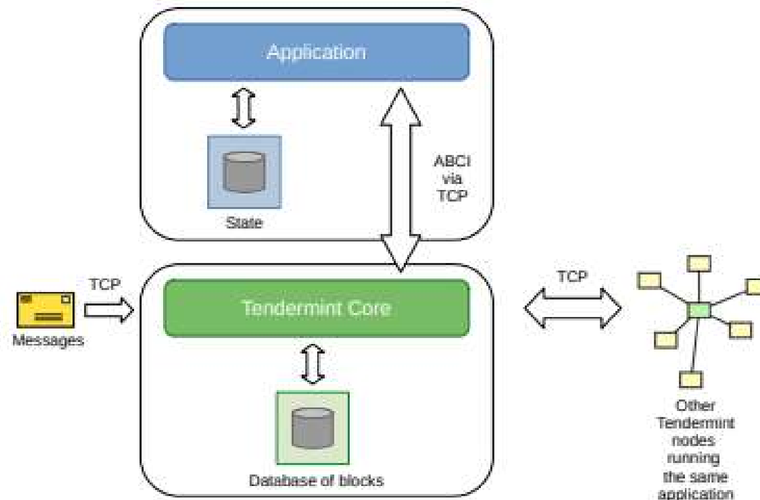


Figura 6 - Schema porte di comunicazione



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.

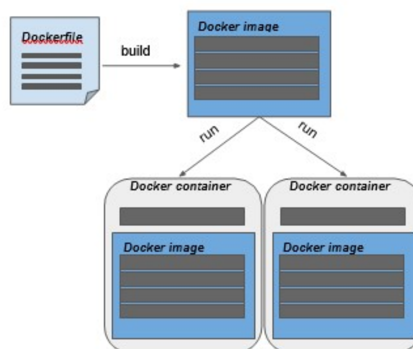


Figura 7 - Deployment dell'applicazione

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 o 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 relaitivi tool 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 rest ache 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

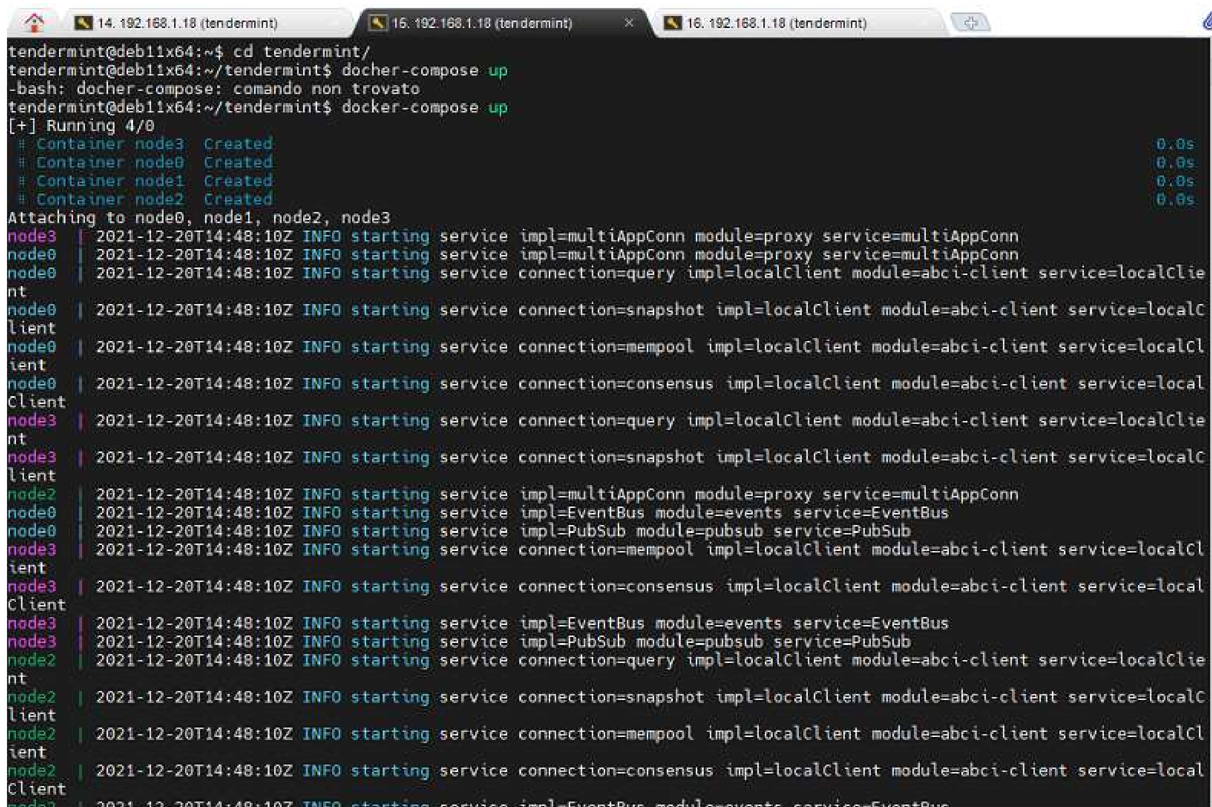
make localnet-stop

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

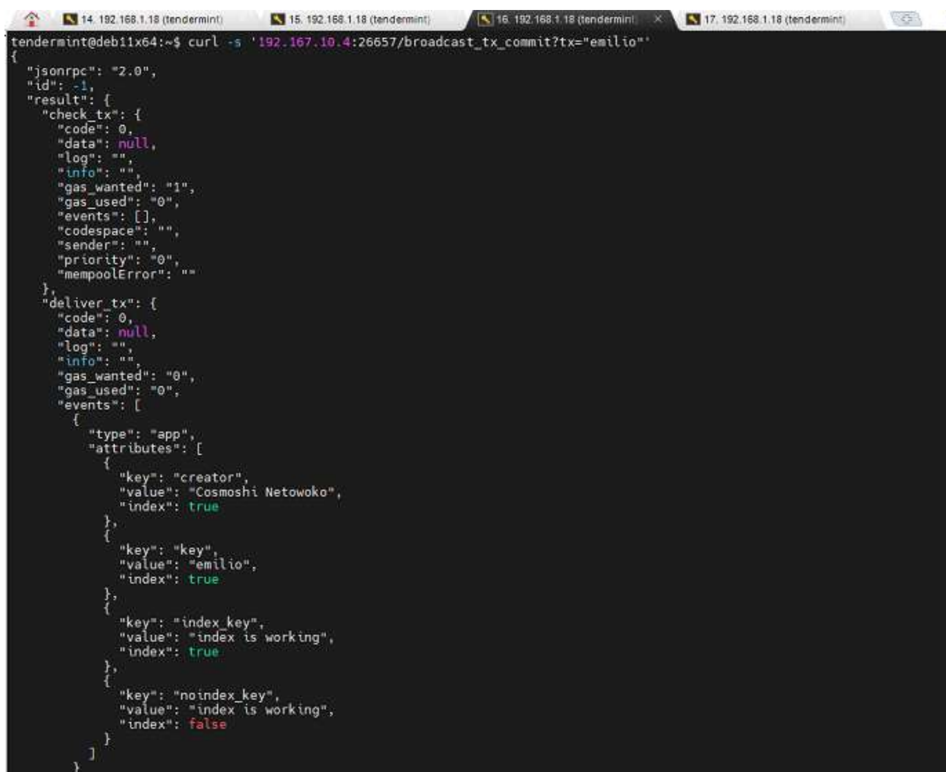
docker run –it node0

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:

14. 192.168.1.18 (tendermint) | 15. 192.168.1.18 (tendermint) | 16. 192.168.1.18 (tendermint) | 17. 192.168.1.18 (tendermint)

```
node2  | 2021-12-20T14:54:44Z INFO executed block height=188 module=state num_invalid_txs=0 num_valid_txs=1
node1  | 2021-12-20T14:54:44Z INFO committed state app_hash=0400000000000000 height=188 module=state num_txs=1
node2  | 2021-12-20T14:54:44Z INFO committed state app_hash=0400000000000000 height=188 module=state num_txs=1
node0  | 2021-12-20T14:54:44Z INFO committed state app_hash=0400000000000000 height=188 module=state num_txs=1
node3  | 2021-12-20T14:54:44Z INFO committed state app_hash=0400000000000000 height=188 module=state num_txs=1
node1  | 2021-12-20T14:54:45Z INFO Timed out dur=978.316171 height=189 module=consensus round=0 step=1
node3  | 2021-12-20T14:54:45Z INFO Timed out dur=972.520995 height=189 module=consensus round=0 step=1
node0  | 2021-12-20T14:54:45Z INFO Timed out dur=977.915134 height=189 module=consensus round=0 step=1
node2  | 2021-12-20T14:54:45Z INFO Timed out dur=983.332703 height=189 module=consensus round=0 step=1
node0  | 2021-12-20T14:54:45Z INFO received proposal module=consensus proposal={"Type":32,"block_id":{"hash":"7675040D9C4C
4B04A63DCF6BCCD04382DD36724374C894B55125439CFBB1188C","parts":{"hash":"E2BF6B0F10B8637158B6D841199C9E346918079D420EBADB098
FAB0505E53584","total":1}},"height":189,"pol_round":-1,"round":0,"signature":"T5FpS6usKiAR5RgtY8ls0mNHZQU5yv+TxEVBUokuoiu+
yBC22PpRFFWrbawg8G6IlFNd8P880rjM2GWm2rczBQ==","timestamp":"2021-12-20T14:54:45.721534557Z"}
node0  | 2021-12-20T14:54:45Z INFO received complete proposal block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55
125439CFBB1188C height=189 module=consensus
node3  | 2021-12-20T14:54:45Z INFO received proposal module=consensus proposal={"Type":32,"block_id":{"hash":"7675040D9C4C
4B04A63DCF6BCCD04382DD36724374C894B55125439CFBB1188C","parts":{"hash":"E2BF6B0F10B8637158B6D841199C9E346918079D420EBADB098
FAB0505E53584","total":1}},"height":189,"pol_round":-1,"round":0,"signature":"T5FpS6usKiAR5RgtY8ls0mNHZQU5yv+TxEVBUokuoiu+
yBC22PpRFFWrbawg8G6IlFNd8P880rjM2GWm2rczBQ==","timestamp":"2021-12-20T14:54:45.721534557Z"}
node3  | 2021-12-20T14:54:45Z INFO received complete proposal block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55
125439CFBB1188C height=189 module=consensus
node1  | 2021-12-20T14:54:45Z INFO received proposal module=consensus proposal={"Type":32,"block_id":{"hash":"7675040D9C4C
4B04A63DCF6BCCD04382DD36724374C894B55125439CFBB1188C","parts":{"hash":"E2BF6B0F10B8637158B6D841199C9E346918079D420EBADB098
FAB0505E53584","total":1}},"height":189,"pol_round":-1,"round":0,"signature":"T5FpS6usKiAR5RgtY8ls0mNHZQU5yv+TxEVBUokuoiu+
yBC22PpRFFWrbawg8G6IlFNd8P880rjM2GWm2rczBQ==","timestamp":"2021-12-20T14:54:45.721534557Z"}
node1  | 2021-12-20T14:54:45Z INFO received complete proposal block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55
125439CFBB1188C height=189 module=consensus
node2  | 2021-12-20T14:54:45Z INFO received proposal module=consensus proposal={"Type":32,"block_id":{"hash":"7675040D9C4C
4B04A63DCF6BCCD04382DD36724374C894B55125439CFBB1188C","parts":{"hash":"E2BF6B0F10B8637158B6D841199C9E346918079D420EBADB098
FAB0505E53584","total":1}},"height":189,"pol_round":-1,"round":0,"signature":"T5FpS6usKiAR5RgtY8ls0mNHZQU5yv+TxEVBUokuoiu+
yBC22PpRFFWrbawg8G6IlFNd8P880rjM2GWm2rczBQ==","timestamp":"2021-12-20T14:54:45.721534557Z"}
node2  | 2021-12-20T14:54:45Z INFO received complete proposal block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55
125439CFBB1188C height=189 module=consensus
node3  | 2021-12-20T14:54:46Z INFO finalizing commit of block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55125439
CFBB1188C height=189 module=consensus num_txs=0 root=0400000000000000
node2  | 2021-12-20T14:54:46Z INFO finalizing commit of block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55125439
CFBB1188C height=189 module=consensus num_txs=0 root=0400000000000000
node3  | 2021-12-20T14:54:46Z INFO executed block height=189 module=state num_invalid_txs=0 num_valid_txs=0
node2  | 2021-12-20T14:54:46Z INFO executed block height=189 module=state num_invalid_txs=0 num_valid_txs=0
node3  | 2021-12-20T14:54:46Z INFO committed state app_hash=0400000000000000 height=189 module=state num_txs=0
node2  | 2021-12-20T14:54:46Z INFO committed state app_hash=0400000000000000 height=189 module=state num_txs=0
node0  | 2021-12-20T14:54:46Z INFO finalizing commit of block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55125439
CFBB1188C height=189 module=consensus num_txs=0 root=0400000000000000
node1  | 2021-12-20T14:54:46Z INFO finalizing commit of block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55125439
CFBB1188C height=189 module=consensus num_txs=0 root=0400000000000000
node1  | 2021-12-20T14:54:46Z INFO executed block height=189 module=state num_invalid_txs=0 num_valid_txs=0
node0  | 2021-12-20T14:54:46Z INFO executed block height=189 module=state num_invalid_txs=0 num_valid_txs=0
node1  | 2021-12-20T14:54:46Z INFO committed state app_hash=0400000000000000 height=189 module=state num_txs=0
node0  | 2021-12-20T14:54:46Z INFO committed state app_hash=0400000000000000 height=189 module=state num_txs=0
node3  | 2021-12-20T14:54:47Z INFO Timed out dur=984.472001 height=190 module=consensus round=0 step=1
node2  | 2021-12-20T14:54:47Z INFO Timed out dur=983.973088 height=190 module=consensus round=0 step=1
```

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.

14. 192.168.1.18 (tendermint) | 15. 192.168.1.18 (tendermint) | 16. 192.168.1.18 (tendermint) | 17. 192.168.1.18 (tendermint)

```
tendermint@deb11x64:~$ curl -s '192.167.10.3:26657/abci_query?data="emilio"'
{
  "jsonrpc": "2.0",
  "id": -1,
  "result": {
    "response": {
      "code": 0,
      "log": "exists",
      "info": "",
      "index": "0",
      "key": "ZW1pbGlv",
      "value": "ZW1pbGlv",
      "proofOps": null,
      "height": "189",
      "codespace": ""
    }
  }
}tendermint@deb11x64:~$
```

Possiamo quindi avere varie informazioni interagendo con il server RCP :

14. 192.168.1.18 (tendermint) | 15. 192.168.1.18 (tendermint) | 16. 192.168.1.18 (tendermint) | 17. 192.168.1.18 (tendermint)

```
tendermint@deb11x64:~$ curl -s 192.167.10.4:26657/status
{
  "jsonrpc": "2.0",
  "id": -1,
  "result": {
    "node_info": {
      "protocol_version": {
        "p2p": "8",
        "block": "11",
        "app": "1"
      },
      "id": "e25e5aa079042827a052844e73fcc622b60c5a28",
      "listen_addr": "tcp://0.0.0.0:26656",
      "network": "chain-J9MP5t",
      "version": "unreleased-master-23be048294bad33667cb7f52fe7df22109b36f0f",
      "channels": "402021222330386061626300",
      "moniker": "node2",
      "other": {
        "tx_index": "on",
        "rpc_address": "tcp://0.0.0.0:26657"
      }
    },
    "sync_info": {
      "latest_block_hash": "926ECD3F4ECFC0917B24E960A9E5C4064A8AA708DC7917CB463203B1F52C2BF9",
      "latest_app_hash": "0200000000000000",
      "latest_block_height": "187",
      "latest_block_time": "2021-12-14T14:29:18.611750601Z",
      "earliest_block_hash": "A704F56FC5ECA40315F63D943F51005555D63BEE1957E0FCE82F90C9040103DF",
      "earliest_app_hash": "",
      "earliest_block_height": "1",
      "earliest_block_time": "2021-10-25T15:57:56.177199544Z",
      "max_peer_block_height": "187",
      "catching_up": false,
      "total_synced_time": "0",
      "remaining_time": "0",
      "total_snapshots": "0",
      "chunk_process_avg_time": "0",
      "snapshot_height": "0",
      "snapshot_chunks_count": "0",
      "snapshot_chunks_total": "0",
      "backfilled_blocks": "0",
      "backfill_blocks_total": "0"
    },
    "validator_info": {
      "address": "7895B7102CEE25F825BD3B65C55B9D385B67F1A0",
      "pub_key": {
        "type": "tendermint/PubKeyEd25519",
        "value": "96wocpX6TWTcUHcJXFMX253pQot6Y6CIoG0g/YvMOJI="
      },
      "voting_power": "1"
    }
```

Oppure analizzando la rete Docker, come segue:

14. 192.168.1.18 (tendermint) | 15. 192.168.1.18 (tendermint) | 16. 192.168.1.18 (tendermint) | 17. 192.168.1.18 (tendermint)

```
tendermint@deb11x64:~$ docker network inspect tendermint_localnet
[
    {
        "Name": "tendermint_localnet",
        "Id": "8bbe7dfd1e74a0e162752a54bef35a07f78eeee035f91ca273b867d42bfeb390",
        "Created": "2021-12-14T14:39:50.275479195+01:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "192.167.10.0/16",
                    "Gateway": "192.167.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "05f6e71e18806b9160e774d4dac62a031f12864c6a23c7f300b9e278f026a2e6": {
                "Name": "node2",
                "EndpointID": "756a0073e50f4ea36bdd26e9518290d7f570f488ed3621578965fdbc669ba186",
                "MacAddress": "02:42:c0:a7:0a:04",
                "IPv4Address": "192.167.10.4/16",
                "IPv6Address": ""
            },
            "30edb628d5493a25525c6392743298786e017b2f93fcd0446954b3d77ded80be": {
                "Name": "node3",
                "EndpointID": "a1aabf2f6577749131932c135bec2cb8e6b6fc59f665962e52511c1eb59f4022",
                "MacAddress": "02:42:c0:a7:0a:05",
                "IPv4Address": "192.167.10.5/16",
                "IPv6Address": ""
            },
            "ccd6e5218eb26f6db6fbed3760745ddbc618bb77e8b8494a3891b1b8558a0fc5": {
                "Name": "node1",
                "EndpointID": "3af2e75ad72d134070709b5d79a867338c51c9df0a9b56066e48f6dbb54f4292",
                "MacAddress": "02:42:c0:a7:0a:03",
                "IPv4Address": "192.167.10.3/16",
                "IPv6Address": ""
            },
            "d6fc9c246ddd9ac4d691281684db0e91a1a1cc61ea12e190bb5a1a05e59b85da": {
                "Name": "node0",
                "EndpointID": "3382b81b27bb91b9ee88500901196b3c98a3c0e90befed1104f6413d80435433",
```

Parallel Asynchronous Particle Swarm Optimization

L'intento iniziale del concetto di sciame di particelle era quello di simulare graficamente la coreografia imprevedibile di uno stormo di uccelli con l'obiettivo di scoprire i modelli che regolano la capacità degli uccelli di volare in sincronia e di cambiare improvvisamente direzione e nel raggrupparsi per formare una formazione ottimale. Da questo obiettivo iniziale, il concetto si è evoluto in un algoritmo di ottimizzazione semplice ed efficiente. In PSO gli individui sono indicati come particelle (boids). Le modifiche alla posizione delle particelle all'interno dello spazio di ricerca si basano sulla tendenza socio-psicologica degli individui ad emulare il successo di altri individui. Le modifiche a una particella all'interno dello sciame sono quindi influenzate dall'esperienza, o conoscenza, dei suoi vicini. Il comportamento nella ricerca di una posizione ottimale di una particella è quindi influenzato dalla sua conoscenza e delle altre particelle all'interno dello sciame.

Basic PSO

Il particle swarm optimization è una tecnica di ottimizzazione stocastica, basata su una popolazione di individui, inventata da Russel Eberhart e James Kennedy nel 1995. Fa parte della famiglia degli algoritmi evolutivi e può essere utilizzato per risolvere diversi problemi che vanno dall'allenamento di reti neurali alla minimizzazione di funzioni non convesse. La versione originale fu inspirata dal comportamento sociale di stormi di uccelli in movimento, con l'obiettivo di trovare il modello che permette a questi di volare in sincrono e cambiare improvvisamente direzione per poi raggrupparsi in una nuova formazione ottimale. In particolare, l'idea che sta alla base del PSO è che ogni individuo in un gruppo, di fronte ad un particolare problema, tende ad interagire con gli altri per risolverlo e man mano che le interazioni si susseguono, si modificano le credenze, le attitudini e i comportamenti di ciascuno di questi. Nel PSO, gli individui, a cui ci si riferisce con il termine particelle, vengono letteralmente fatti "volare" in uno spazio di ricerca multidimensionale. I movimenti delle particelle all'interno di questo sono influenzati dalla tendenza socio-psicologica che porta ogni individuo ad emulare il successo degli altri. La ricerca di una singola particella è pertanto collegata a quella delle altre particelle all'interno del gruppo.

Il PSO cerca di simulare la cooperazione sociale appena descritta. Allo sciame viene assegnato un problema per il quale esiste un modo per valutare l'ottimalità della soluzione trovata attraverso una funzione obiettivo chiamata funzione di fitness. E inoltre definita una struttura di comunicazione che assegna ad ogni individuo un insieme di vicini con cui interagire. L'algoritmo mantiene poi una

popolazione detta anche sciame di particelle ognuna delle quali ha una posizione che corrisponde ad una possibile soluzione del problema. Ogni particella è in grado di valutare, tramite la funzione obiettivo, la bontà della propria posizione e di ricordare la migliore visitata. Questa informazione è condivisa tra tutti i vicini cosicché ogni particella conosce anche la migliore posizione di tutti i vicini con cui interagisce. Gli individui all'interno dello sciame hanno un comportamento molto semplice: modificano la propria posizione in funzione della loro esperienza e di quella dei loro vicini.

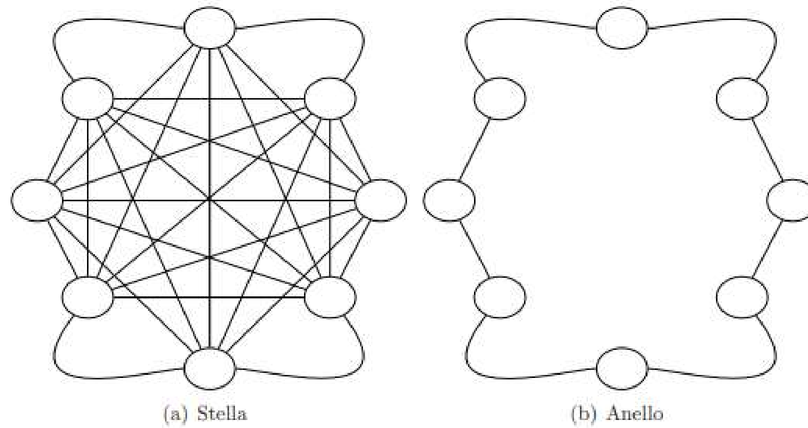
La posizione di ogni particella, calcolata per ogni step temporale ha la seguente funzione:

$$x_{k+1}^{i} = x_{k}^{i} + v_{k+1}^{i} \quad (1)$$

È quindi il vettore velocità quello che guida il processo di ottimizzazione, e riflette sia l'esperienza personale sia quella acquisita dalle interazioni sociali. Questi due fattori vengono rispettivamente chiamati componente cognitiva e componente sociale. Esistono due versioni di PSO chiamate gbest e lbest PSO. La differenza tra le due è data dall'insieme dei vicini con cui una data particella interagisce direttamente. Nel **gbest PSO** ogni particella scambia informazioni con tutte le altre. La struttura di comunicazione è definita dalla tipologia a stella e la componente sociale nel calcolo della velocità riflette l'informazione ottenuta da tutte le particelle.

Require: creazione ed inizializzazione di uno sciame n-dimensionale
repeat
  for all $i \in s$ **do**
    if $f(\vec{x}_i) < f(\vec{y}_i)$ **then**
      $\vec{y}_i \leftarrow \vec{x}_i$
    end if
    if $f(\vec{y}_i) < f(\hat{y})$ **then**
      $\hat{y} \leftarrow \vec{y}_i$
    end if
  end for
  for all $i \in s$ **do**
    aggiorna la velocità di i con la (3.2)
    aggiorna la posizione di i con la (3.1)
  end for
until non è verificata una condizione di uscita

Il local best PSO utilizza la struttura di comunicazione ad anello dove ogni particella comunica con un sottoinsieme di vicini. La componente sociale riflette l'informazione scambiata tra questi, evidenziando la natura locale del sistema



(a) Stella (b) Anello

Nonostante esistano strategie basate sulla similarità spaziale, i vicini di ogni particella vengono tipicamente scelti in base ai loro indici.

Le ragioni che spingono a far questo sono essenzialmente due:

- Si riduce la complessità computazionale, poiché non è necessario calcolare la distanza Euclidea tra tutte le coppie di particelle.
- Si favorisce la diffusione dell'informazione a tutte le particelle, indipendentemente dalla loro posizione nello spazio di ricerca.

Bisogna anche notare che i sottoinsiemi di comunicazione si sovrappongono. Una particella può essere membro di diversi sottoinsiemi favorendo l'interconnessione e lo scambio d'informazione tra questi. Questo consente allo sciame di convergere verso un unico punto che corrisponde all'ottimo globale del problema. Il global best PSO può essere visto come un caso speciale del local best PSO in cui s è uguale alla cardinalità dello sciame.

Require: creazione ed inizializzazione di uno sciame n-dimensionale
  repeat
    for all $i \in s$ **do**
      if $f(\vec{\mathbf{x}}_i) < f(\vec{\mathbf{y}}_i)$ **then**
        $\vec{\mathbf{y}}_i \leftarrow \vec{\mathbf{x}}_i$
      end if
      if $f(\vec{\mathbf{y}}_i) < f(\hat{\mathbf{y}}_i)$ **then**
        $\hat{\mathbf{y}}_i \leftarrow \vec{\mathbf{y}}_i$
      end if
    end for
    for all $i \in s$ **do**
      aggiorna la velocità di i con la (3.3)
      aggiorna la posizione di i con la (3.1)
    end for
  until non è verificata una condizione di uscita

Grazie all'interconnessione tra tutte le particelle il global best PSO converge più velocemente. Tuttavia questa maggiore velocità di convergenza si paga in termini di una minore esplorazione. Mentre come conseguenza della sua maggiore diversità (che si manifesta nella copertura di più zone nello spazio di ricerca), il local best PSO è meno soggetto a rimanere intrappolato in un minimo locale.

Il calcolo della velocità consiste nella somma di tre termini:

- La velocità precedente, che serve alla particella per ricordare la direzione del suo movimento. In particolare questo termine può essere visto come un momento che impedisce di cambiare drasticamente direzione. Questa componente è anche chiamata inerzia.
- La componente cognitiva, che può essere vista come il ricordo della miglior posizione raggiunta. L'effetto di questo termine è l'attrazione di ogni particella i verso l'ottimo trovato da ciascuna, in accordo alla tendenza degli individui a tornare nei luoghi e nelle situazioni che più li hanno soddisfatti.
- La componente sociale, che quantifica il successo della particella i in relazione all'intero gruppo o all'insieme dei vicini con cui comunica. Concettualmente la componente sociale rappresenta una norma o uno standard a cui gli individui tendono ad attenersi. L'effetto di questo termine è l'attrazione delle particelle verso la miglior posizione trovata dall'intero gruppo o dal gruppo di vicini.

Anche se il PSO ha dimostrato di essere un algoritmo efficiente e con buoni risultati, la sua struttura non garantisce che la soluzione migliore venga trovata, dato che si basa sull'esplorazione dello spazio, ma questa d'altronde è una caratteristica intrinseca delle tecniche euristiche, sebbene esse offrano una buona certezza sul fatto che ci si avvicini all'ottimo globale senza fermarsi ad uno locale.

Trattandosi di problemi multi-dimensionali, genericamente il vettore velocità viene definito da:

$$v_i = (v_{i,0}, v_{i,1}, \ldots, v_{i,D})$$

ed il vettore posizione

$$x_i = (x_{i,0}, x_{i,1}, \ldots, x_{i,D}).$$

Essi, nel caso della gBest PSO sono calcolati come segue:

$$\mathbf{x_{i,d}(G+1) = x_{i,d}(G) + v_{i,d}(G+1)} \qquad \mathbf{(3.1)}$$

e

$$\mathbf{v_{i,d}(G+1) = v_{i,d}(G) + C1 \cdot Rnd(0,1) \cdot [pb_{i,d}(G) - x_{i,d}(G)] + C2 \cdot Rnd(0,1) \cdot [gb_d(G) - x_{i,d}(G)]} \qquad \mathbf{(3.2)}$$

in cui abbiamo:

i: indice delle particelle;

d: dimensione considerata, ogni particella ha una posizione ed una velocità per ogni dimensione;

G: generazione, cioè il time stamp di elaborazione dell'algoritmo;

xi,d: posizione della particella i nella dimensione d;

vi,d: velocità della particella i nella dimensione d;

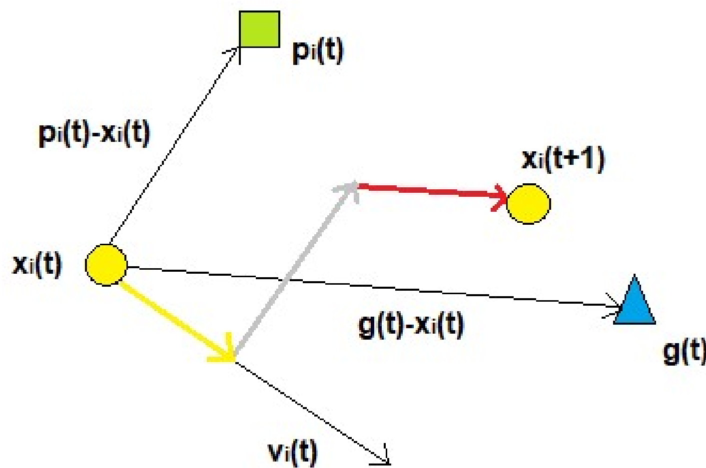
C1: costante di accelerazione per la componente cognitiva;

pbi,d: la locazione nella dimensione d con il miglior valore della fitness tra tutte quelle visitate in questa dimensione dalla particella i;

C2: costante di accelerazione per la componente sociale;

gbd: la locazione nella dimensione d con il miglior valore di fitness tra tutte le posizioni visitate in quella dimensione da tutte le particelle.

Se per semplicità consideriamo un problema bidimensionale dove lo spazio delle soluzioni del problema è rappresentato dal piano cartesiano di figura



Le leggi che governano il moto di una particella sono di conseguenza:

$$\mathbf{v}(t+1) = (w * \mathbf{v}(t)) + (c1 * r1 * (\mathbf{p}(t) - \mathbf{x}(t)) + (c2 * r2 * (\mathbf{g}(t) - \mathbf{x}(t))$$

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \mathbf{v}(t+1)$$

Ogni particella occupa, in istanti discreti di tempo, una posizione del piano X(t), la quale rappresenta una possibile soluzione del problema. Per determinare la nuova posizione della particella si deve calcolare la risultante dei vettori come somma pesata dei tre vettori mostrati.

Dall'equazione 3.1 abbiamo che la posizione alla nuova generazione per ogni particella è quindi data dalla somma tra la posizione attuale e la velocità. La velocità a sua volta si calcola con la 3.2.

Uno dei punti di forza del PSO è la capacità degli individui di condividere le informazioni, ed il fatto che conseguentemente il comportamento di ognuno di essi venga influenzato da quanto esplorato dagli altri componenti della popolazione. Le equazioni 3.1 e 3.2 sono auto-aggiornanti essendo ricorsive, e permettono un miglioramento progressivo delle posizioni di tutte le particelle: proprio a causa della ricorsività, un altro aspetto importante è l'inizializzazione delle particelle della prima iterazione, dato che da esse dipende inevitabilmente ogni generazione successiva ed hanno un grande impatto sulla buona riuscita della ricerca. Normalmente le posizioni iniziali vengono estratte da una variabile uniforme, in modo che esse coprano al meglio lo spazio parametrico.

Le velocità vengono inizializzate casualmente, con la cura di settarle ad un valore basso perché sono generalmente più adeguate per evitare grandi spostamenti iniziali. Di seguito vediamo l'algoritmo con il suo diagramma di flusso.

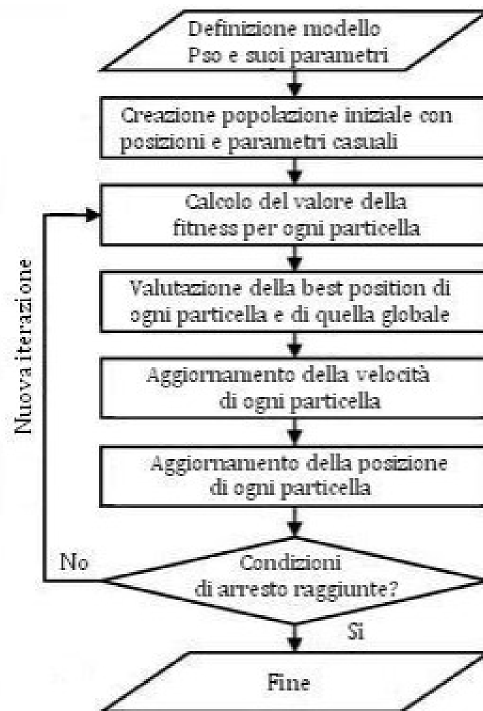


Figura 8 - Algoritmo Basic PSO

Inizialmente oltre alle velocità ed alle posizioni vanno inizializzate anche le costanti per la parte sociale e cognitiva, che non variano mai durante l'esecuzione della ricerca, ed hanno anch'esse un grosso peso specifico, controllando quanto pesino negli spostamenti le componenti cognitiva e sociale. Per la corretta implementazione dell'algoritmo è basilare aggiornare prima le velocità delle particelle, e successivamente le loro posizioni.

La versione appena descritta è quella, definita del **Global Best**, ma esistono molte varianti che cercano di migliorarla. Le varianti si possono basare su come vengono inizializzate velocità e posizioni (ad esempio le velocità iniziali possono essere poste uguali a zero anziché essere inizializzate casualmente) oppure su quanto venga influenzata ogni particella dalla miglior posizione globale rispetto a quella delle particelle vicine. Le due varianti principali sono il **Local Best** e **l'Inertia Weight.**

Local Best è una variante nella quale, a differenza della versione base, si riduce ad una zona più ristretta lo scambio di informazioni. Le particelle infatti vengono divise in sotto-gruppi, e la condivisione di quale sia la miglior posizione avviene solo tra loro. Gli spostamenti sono quindi maggiormente influenzati dal "vicinato" rispetto a quanto avviene nel Global Best. Ciò comporta che la convergenza all'ottimo globale è più lenta, ma il lato positivo è che la copertura dello spazio di ricerca avviene con maggiore efficacia, riducendo la probabilità di restare in un ottimo locale, ed aumentando quella di arrivare all'ottimo globale. La scelta tra le due varianti dipende quindi dal tempo a disposizione, poichè in termini di costi si ha una perdita solo da questo punto di vista a fronte però di una ricerca più affidabile. La suddivisione avviene perlopiù in due modi: statisticamente, guardando all'indice della particella, o sulla base delle distanze.

Inertia Weigth è una variante che mira a bilanciare due possibili tendenze del PSO, quella di sfruttare l'intorno di soluzioni note e quella di esplorare nuove aree dello spazio di ricerca. A tale scopo essa si focalizza sulla prima componente $v_{i,d}(G)$ della (3.2), che tiene memoria della velocità precedente, moltiplicandola per una costante w. Questa è la versione del PSO che verrà utilizzata nei calcoli. L'equazione (3.2) diventa quindi

$$
\begin{aligned}
v_{i,d}(G+1) = {} & w \cdot v_{i,d}(G) \\
& + C1 \cdot Rnd(0,1) \cdot [pb_{i,d}(G) - x_{i,d}(G)] \\
& + C2 \cdot Rnd(0,1) \cdot [gb_d(G) - x_{i,d}(G)]
\end{aligned}
\tag{3.3}
$$

Si noti che togliendo questa componente il movimento sarebbe costante attorno ad un punto, unico candidato come soluzione. L'idea dell' Inertia Weight si traduce nel dare un peso bilanciato all'inerzia del movimento, moltiplicando la componente per una costante w, cercando di dare più importanza

all'esplorazione di nuove aree di quanto non accada con la versione base del PSO. Per fare questo c'è la necessità di contrastare il movimento precedente, spingendo le particelle in nuove direzioni, con la moltiplicazione di vi,d(G) per la costante.

L'altro aspetto che va considerato riguarda la condizione di uscita e quindi il criterio utilizzato per terminare il processo di ricerca. Per scegliere questo vanno considerati due fattori:

1. La condizione di uscita non dovrebbe causare una convergenza prematura, o si otterrebbero soluzioni subottime.

2. La condizione di uscita dovrebbe evitare valutazioni inutili della funzione obiettivo. Se il criterio di arresto richiede frequenti valutazioni della funzione di fitness, la complessità del processo di ricerca potrebbe aumentare notevolmente.

Le condizioni di uscita più comuni sono le seguenti:

• Termina dopo un numero massimo di passi o di valutazioni della funzione obiettivo. E chiaro che se il numero massimo di iterazioni è troppo piccolo si potrebbe uscire prima di aver trovato una buona soluzione. Questo criterio è generalmente usato insieme a un test di convergenza per forzare l'uscita se l'algoritmo non converge entro un limite di passi. Usato da solo questo metodo è utile in quei casi in cui l'obiettivo è valutare la migliore soluzione trovata in un certo periodo di tempo.

• Termina quando una soluzione accettabile è stata trovata. Questo criterio termina il processo di ricerca non appena viene trovata una posizione xi tale che f(xi) ≤ |f(x^*) + ϙ, dove x^* è l'ottimo della funzione obiettivo f ed ϙ è l'errore consentito. Il valore di soglia, ϙ, deve essere scelto con cura. Se ϙ è troppo grande, il processo potrebbe terminare in una soluzione sub-ottima. Se invece ϙ è troppo piccolo la ricerca potrebbe non terminare mai. Inoltre questo metodo richiede una conoscenza a priori di quale sia l'ottimo e questo è possibile solo in alcuni problemi.

• Termina quando non ci sono miglioramenti dopo un numero fissato di passi. Ci sono vari modi per valutare un miglioramento. Per esempio se il cambiamento medio delle posizioni è piccolo, si può assumere che lo sciame sia arrivato a convergenza. Oppure se la media delle velocità è prossima allo zero, verranno fatti solo piccoli passi, e quindi la ricerca può essere terminata.

Comunque questi metodi introducono altri due parametri per cui bisogna trovare un valore corretto:
1) il numero di iterazioni entro cui valutare se ci sono miglioramenti

2) una soglia che definisce quando non ci sono progressi da un passo ad un altro.

Java Code

Di seguito viene mostrata una semplice implementazione scritta in linguaggio java dell'algoritmo PSO per la ricerca dell'ottimo di una funzione pre-assegnata.

$$f(x, y) = (2.8125 - x + xy^4)^2 + (2.25 - x + xy^2)^2 + (1.5 - x + xy)^2$$

Con i seguenti vincoli:

$$1 <= x <= 4; -1 <= y <= 1$$

Per risolvere questo problema utilizzando PSO, avremo bisogno di queste classi:

- **Location**: per rappresentare la parte Posizione della particella
- **Velocity**: per rappresentare la parte di Velocità della particella
- **Particle**: la particella stessa
- **SimplePSO**: il controllo principale del programma
- **PSOContants**: definisce i parametri utilizzati nell'algoritmo

Poiché andremo a risolvere l'ottimizzazione di funzioni a due variabili, dovremo fornire posizione e velocità bidimensionali.

Per la posizione abbiamo:

```
1    public class Location {
2          private double[] loc;
3          public Location(double[] loc) {
4                super();
5                this.loc = loc;
6          }
7          public double[] getLoc() {
8                return loc;
9          }
10         public void setLoc(double[] loc) {
11               this.loc = loc;
12         }
     }
```

Per la velocità abbiamo:

```
1    public class Velocity {
2          private double[] vel;
3
4          public Velocity(double[] vel) {
5                super();
6                this.vel = vel;
7          }
8          public double[] getPos() {
9                return vel;
10         }
11         public void setPos(double[] vel) {
12               this.vel = vel;
13         }
     }
```

La particella avrà la seguente definizione:

```
1   public class Particle {
2         private double fitnessValue;
3         private Velocity velocity;
4         private Location location;
5         public Particle() {
6               super();
7         }
8         public Particle(double fitnessValue, Velocity velocity, Location
          location) {
9               super();
10              this.fitnessValue = fitnessValue;
11              this.velocity = velocity;
12              this.location = location;
13        }
14        public Velocity getVelocity() {
15              return velocity;
16        }
17        public void setVelocity(Velocity velocity) {
18              this.velocity = velocity;
19        }
20        public Location getLocation() {
21              return location;
22        }
23        public void setLocation(Location location) {
24              this.location = location;
25        }
26        public double getFitnessValue() {
27              fitnessValue = ProblemSet.evaluate(location);
28              return fitnessValue;
          }
    }
```

Definizione della funzione di fitness del problema con i relativi vincoli:

```
1   public class ProblemSet {
2         public static final double LOC_X_LOW = 1;
3         public static final double LOC_X_HIGH = 4;
4         public static final double LOC_Y_LOW = -1;
5         public static final double LOC_Y_HIGH = 1;
6         public static final double VEL_LOW = -1;
7         public static final double VEL_HIGH = 1;
8
9         public static final double ERR_TOLERANCE = 1E-20;
10        public static double evaluate(Location location) {
11              double result = 0;
12              double x = location.getLoc()[0]; // the "x" part
13              double y = location.getLoc()[1]; // the "y" part
14
15              result = Math.pow(2.8125 - x + x * Math.pow(y, 4), 2) +
16                          Math.pow(2.25 - x + x * Math.pow(y, 2), 2) +
17                          Math.pow(1.5 - x + x * y, 2);
18
19              return result;
20        }
    }
```

Vediamo ora come parametrizzare l'algoritmo di PSO.

Inseriamo i parametri in nella classe PSOConstants:

```
1   public interface PSOConstants {
2         int SWARM_SIZE = 30;
3         int MAX_ITERATION = 100;
4         int PROBLEM_DIMENSION = 2;
5         double C1 = 2.0;
6         double C2 = 2.0;
7         double W_UPPERBOUND = 1.0;
8         double W_LOWERBOUND = 0.0;
9   }
```

L'algoritmo Serial PSO sarà quindi:

```
1   private void initializeSwarm() {
2      Particle p;
3      Random generator = new Random();
4      for (int i = 0; i < SWARM_SIZE; i++) {
5          p = new Particle();
6          double posX = generator.nextDouble() * 3.0 + 1.0;
7          double posY = generator.nextDouble() * 2.0 - 1.0;
8          p.setLocation(new Position(posX, posY));
9          double velX = generator.nextDouble() * 2.0 - 1.0;
10         double velY = generator.nextDouble() * 2.0 - 1.0;
11         p.setVelocity(new Velocity(velX, velY));
12         swarm.add(p);
13      }
14   }
15  public void execute() {
16     Random generator = new Random();
17     initializeSwarm();
18     evolutionaryStateEstimation();
19     int t = 0;
20     double w;

21     while (t < MAX_ITERATION) {
22       // calculate corresponding f(i,t) corresponding to location x(i,t)
23       calculateAllFitness();

24       // update pBest
25       if (t == 0) {
26         for (int i = 0; i < SWARM_SIZE; i++) {
27              pBest[i] = fitnessList[i];
28              pBestLoc.add(swarm.get(i).getLocation());
29         }
30       } else {
31         for (int i = 0; i < SWARM_SIZE; i++) {
32           if (fitnessList[i] < pBest[i]) {
33              pBest[i] = fitnessList[i];
34              pBestLoc.set(i, swarm.get(i).getLocation());
35           }
36          }
37       }

38       int bestIndex = getBestParticle();
39       // update gBest
40       if (t == 0 || fitnessList[bestIndex] < gBest) {
41           gBest = fitnessList[bestIndex];
42           gBestLoc = swarm.get(bestIndex).getLocation();
```

```
43        }
44
45        w = W_UP - (((double) t) / MAX_ITERATION) * (W_UP - W_LO);
46
47        for (int i = 0; i < SWARM_SIZE; i++) {
48        // update particle Velocity
49        double r1 = generator.nextDouble();
50        double r2 = generator.nextDouble();
51        double lx = swarm.get(i).getLocation().getX();
52        double ly = swarm.get(i).getLocation().getY();
53        double vx = swarm.get(i).getVelocity().getX();
54        double vy = swarm.get(i).getVelocity().getY();
55        double pBestX = pBestLoc.get(i).getX();
56        double pBestY = pBestLoc.get(i).getY();
57        double gBestX = gBestLoc.getX();
58        double gBestY = gBestLoc.getY();
59
60        double newVelX = (w * vx) + (r1 * C1) * (pBestX - lx) + (r2 * C2) * (gBestX -
           lx);
61        double newVelY = (w * vy) + (r1 * C1) * (pBestY - ly) + (r2 * C2) * (gBestY -
62        ly);
63        swarm.get(i).setVelocity(new Velocity(newVelX, newVelY));
64        // update particle Location
65        double newPosX = lx + newVelX;
66        double newPosY = ly + newVelY;
67        swarm.get(i).setLocation(new Position(newPosX, newPosY));
68      }
69      t++;
70    }
```

E’ possible richiedere i parametri e la funzione di fitness all’avvio del programma, come segue:

```
private static void printMenu () {
    System.out.println("----------------------------MENU----------------------------");
    System.out.println("Select a function:");
    System.out.println("1. (x^4)-2(x^3)");
    System.out.println("2. Ackley's Function");
    System.out.println("3. Booth's Function");
    System.out.println("4. Three Hump Camel Function");
    System.out.print("Function:  ");
}

private static Particle.FunctionType getFunction (int input) {
    if (input == 1)         return Particle.FunctionType.FunctionA;
    else if (input == 2)    return Particle.FunctionType.Ackleys;
    else if (input == 3)    return Particle.FunctionType.Booths;
    else if (input == 4)    return Particle.FunctionType.ThreeHumpCamel;
    System.out.println("Invalid Input.");
    return null;
}
```

L'otuput che verrà generato sarà il seguente:

```
tendermint@Host-008:~/Particle-Swarm-Optimization$ java PSO.Main
Use the parameter '-p' to change the inertia, cognitive and social components.
Otherwise the default values will be:
Inertia:             0.729844
Cognitive Component: 1.49618
Social Component:    1.49618
---------------------------MENU---------------------------
Select a function:
1. (x^4)-2(x^3)
2. Ackley's Function
3. Booth's Function
4. Three Hump Camel Function
Function:  3
Particles: 30
Epochs:    300
-------------------------EXECUTING-------------------------
Global Best Evaluation (Epoch 0):       545.0
Global Best Evaluation (Epoch 3):       15.833895234995364
Global Best Evaluation (Epoch 5):       4.082682947656216
Global Best Evaluation (Epoch 7):       3.0189240605300505
Global Best Evaluation (Epoch 8):       0.43949969814004425
Global Best Evaluation (Epoch 24):      0.34790276350243554
Global Best Evaluation (Epoch 28):      0.12104995479109455
Global Best Evaluation (Epoch 30):      0.07699442478359528
Global Best Evaluation (Epoch 32):      0.062115654014238623
```

Parallel Particle Swarm Optimization

Le prime implementazioni dell'algoritmo PSO parallelo sono stati sviluppate per macchine multiprocessore utilizzando la strategia master-slave. In questa strategia, il processore master coordina tutti i passaggi dell'algoritmo e gestisce l'esecuzione delle singole funzioni obiettivo sui processori slaves. L'algoritmo sincrono parallelo di PSO, può essere visto come una semplice estensione dell'algoritmo seriale. È sostanzialmente identico all'algoritmo seriale, con la differenza che le valutazioni della funzione obiettivo vengono eseguite in parallelo su più processori slave.

L'algoritmo sincrono mostra un punto debole nella sua strategia: la necessità di sincronizzazione, rende l'algoritmo limitato dallo slave più lento. In altre parole, al punto di sincronizzazione il master deve attendere che tutti gli slave terminino il proprio lavoro per continuare l'algoritmo lasciando in attesa gli eventuali slave disponibili. Per superare questa limitazione è stata implementata una strategia asincrona di parallelizzazione, dove non esiste punto di sincronizzazione. Viene così sostituito il concetto di volo delle particelle con la definizione di pseudo-volo. Negli algoritmi sincroni, il numero di valutazioni della funzione fitness è uguale al numero di particelle, mentre nell'algoritmo asincrono questo numero non è costante perché non si attende che tutte le particelle abbiano concluso la loro esecuzione per eseguire una valutazione della convergenza. Pertanto indipendentemente da quali particelle hanno effettuato queste valutazioni, il master verifica la permanenza ottimale e i criteri di arresto senza interrompere il lavoro svolto dagli slave. In conseguenza, l'unica perdita di tempo nella parallelizzazione si verifica quando uno slave vuole

comunicare con il master durante l'esecuzione della fase di ottimizzazione. In questa situazione, lo slave diventa momentaneamente inattivo.

Per ottenere l'asincronia, l'esecuzione delle particelle sono poste in una coda FIFO (First In First Out). Poiché il tempo di calcolo della funzione obiettivo varia, e con esso varia anche l'ordine delle particelle in coda per essere schedulate su un processore, di conseguenza alcune particelle potrebbero non contribuire all'ottimizzazione in uno pseudo-volo. Inoltre, gli slave possono eseguire un numero diverso di valutazioni, il che rende l'algoritmo asincrono incapace di mantenere la consistenza. Un approccio sincrono mantiene la coerenza tra implementazioni sequenziali e parallele, evitando così l'alterazione delle caratteristiche di convergenza dell'algoritmo.

Poiché i vettori di velocità e posizione aggiornati della particella i sono calcolati utilizzando la posizione migliore PiK e la posizione migliore a livello globale PGK fino all'iterazione k , l'ordine delle valutazioni della funzione particella influenza l'esito dell'ottimizzazione. Di conseguenza, se permettiamo all'ordine delle particelle di cambiare continuamente a seconda della velocità con cui ogni processore li elabora, vengono alterate le caratteristiche di convergenza.

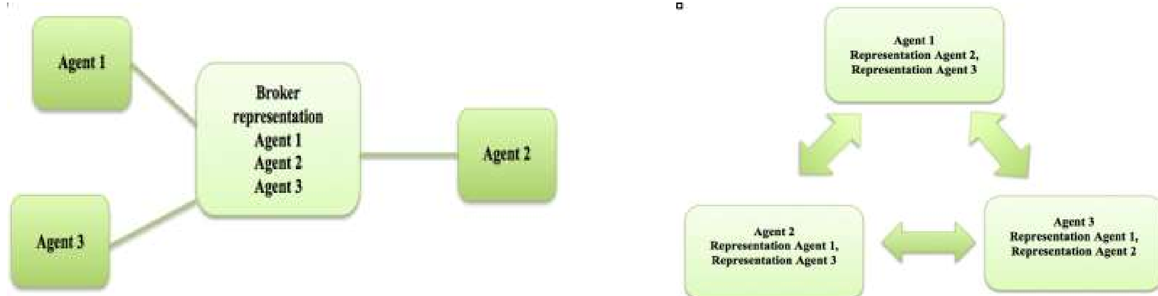
Un'implementazione del PSO asincrona parallela è presente in letteratura ed è stata sviluppata utilizzando l'approccio multi-agente. L'approccio multi-agente può generalmente gestire un problema suddividendolo in sotto problemi più semplici, in modo che gli agenti si devono dedicare solo a sotto-attività del problema generale. Nella programmazione, l'uso dell'approccio multi-agente è utilizzato per realizzare il parallelismo è l'esecuzione simultanea di calcoli (eventualmente collegati) al fine di accelerare l'elaborazione di problemi informatici intensivi e per eseguire un numero elevato di operazioni in un tempo limitato.

I sistemi multi-agente (MAS) possono essere caratterizzati dalla presenza di un'entità centrale denominata "broker", con cui tutti gli agenti del sistema possono comunicare e funge da "portalettere" verso gli altri agenti del gruppo. Ogni agente, prima di poter interagire col sistema, deve dichiararsi al Broker e deve fornirgli le sue caratteristiche.

Il vantaggio di architettura centralizzata di comunicazione consiste nella facilità di aggiunta e rimozione di un agente; perché un nuovo agente si integri, è sufficiente stabilire un protocollo di interazione con il Broker. Lo svantaggio è che la centralizzazione può rallentare gli scambi e la comunicazione nel sistema.

In un MAS che utilizza un approccio di comunicazione distribuito, ogni agente ha l'autonomia di interagire con gli altri agenti senza intermediario. Ogni elemento deve mantenere una base di conoscenza che descrive le caratteristiche e gli indirizzi degli agenti con cui collabora. Quando un

agente vuole fornire un nuovo servizio, deve informare tutti gli altri di questo nuovo servizio. Il vantaggio dell'approccio distribuito è che facilita gli scambi e migliora la comunicazione. Lo svantaggio è che la fornitura del nuovo servizio richiede un aggiornamento della base di conoscenza di ciascun agente.



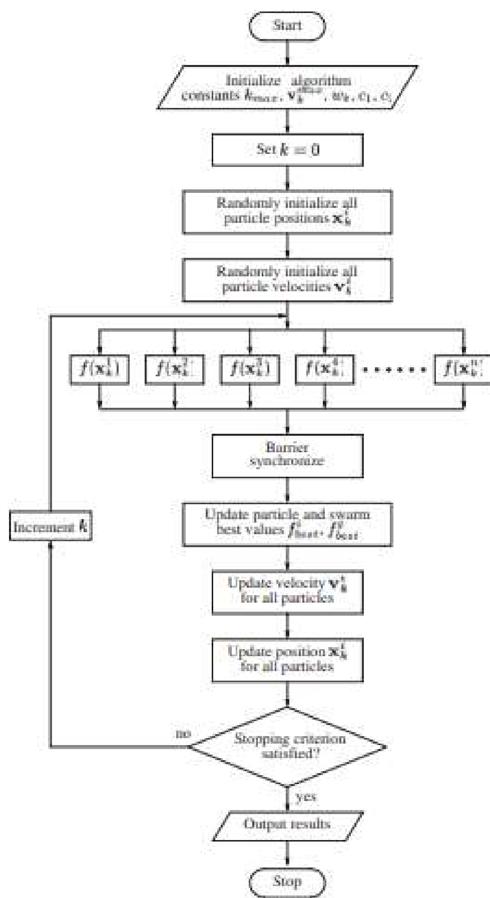
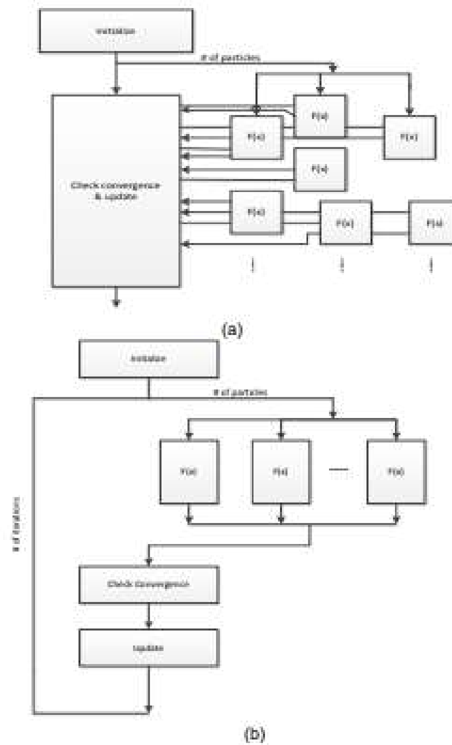


Figura 9 - Algoritmo parallel synchronous PSO



Block diagrams: (a) parallel asynchronous; (b) parallel synchronous PSO

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 le richieste 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.

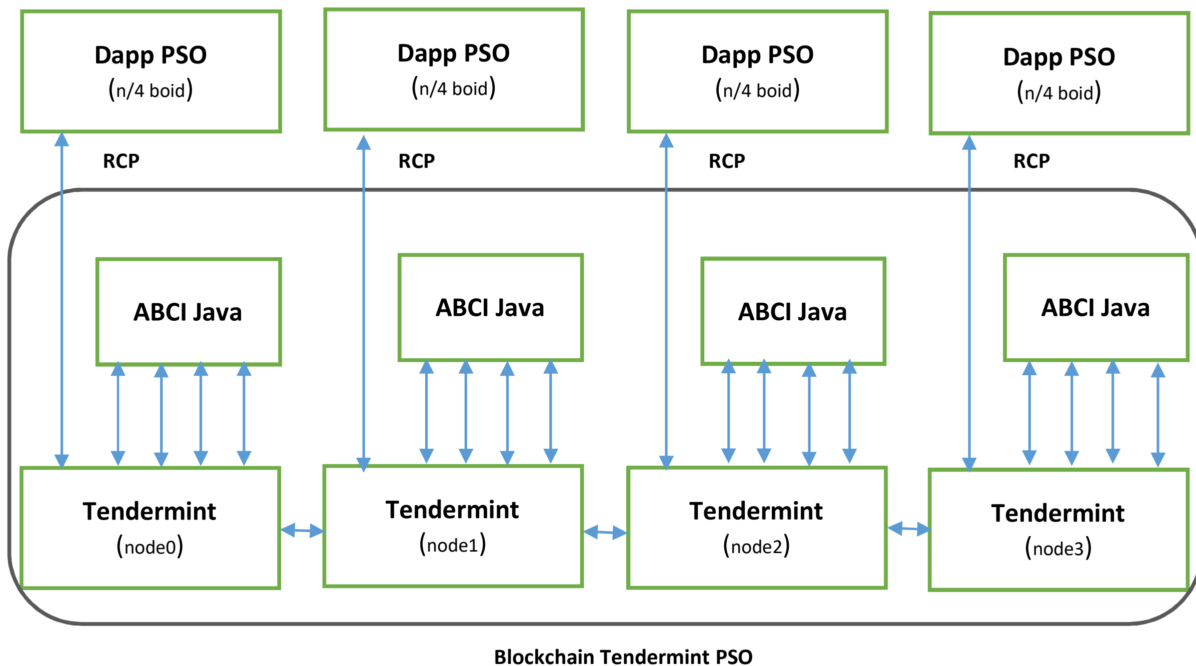
Architettura del sistema

L'implementazione dell'algoritmo su piattaforma blockchain è molto simile al sistema MAS basato su broker. La blockchain sostituisce il lavoro del broker del sistema MAS, di fatti un nuovo nodo che vuole aggiungersi alla chain, deve dichiararsi ed utilizzare i formalismi della chain a cui si sottoscrive; ogni nodo della chain non conosce il numero o la collocazione degli altri nodi ed interagisce semplicemente con la chain.

Rispetto all'algoritmo PSO parallelo sincrono, nel nostro caso non abbiamo il blocco Master che controlla il flusso dell'applicazione e sincronizza i processi. In uno scenario in cui abbiamo N processi identici che eseguono le stesse operazioni in parallelo, quindi occorre parallelizzare l'intero flusso del programma mostrato nel paragrafo precedente. Per l'implementazione dell'algoritmo si può decidere per entrambe le versioni: asincrona o sincrona. La versione asincrona è quella che viene più naturale con l'uso di una blockchain in quando l'unica informazione di scambio con lo swarm è il valore della Gbest finora calcolata. Al termine di ogni operazione di ottimizzazione, ogni elemento dello swarm aggiorna, se necessario, la propria local best ed eventualmente anche il global best, se ad esempio il valore del local best è migliore del global best finora calcolato dallo swarm. Successivamente aggiorna la sua velocità e posizione.

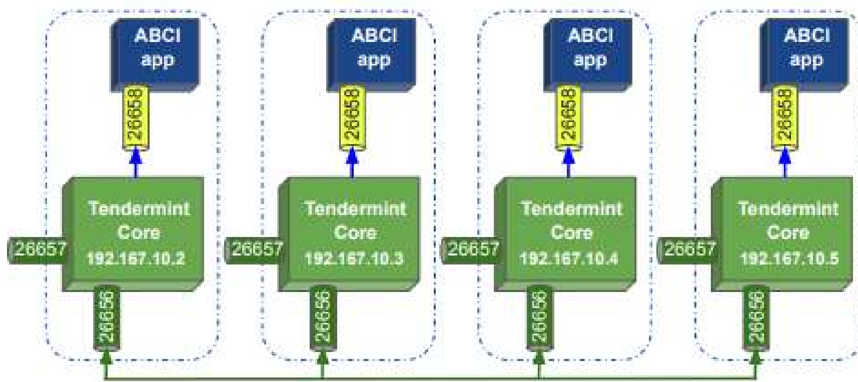
In questo caso ci potrebbero essere più di una particella a richiedere l'aggiornamento del valore della gBest. Questo aspetto però non ci preoccupa in quando tutte le transizioni vengono temporizzate dalla piattaforma, l'unica accortezza da rispettare nel caso ad esempio di utilizzo della piattaforma

Tendermint è quella di utilizzare transazioni con conferma come: "*broadcast_tx_commit*". Se vogliamo possiamo usare questa transazione come punto di sincronismo con gli altri partecipanti allo swarm. ovvero il nodo prima di aggiornare la propria posizione e la propria velocità deve attendere l'approvazione del nuovo global best. Qualora vogliamo mantenere un comportamento del tutto asincrono, ogni nodo comunica al chain il proprio gbest e continua nella sua elaborazione utilizzando però il gbest memorizzato nella chain (eseguendo una query request) per mantenere la consistenza dell'applicazione. Ogni nodo conterrà quindi una sequenza di richiesta di aggiornamento della global best ed il valore della gbest allo step precedente. L'aggiornamento della global best avverrà attraverso la chiamata della funzione "**deliverTx**" (messaggio usato per l'inserimento del mempool nella chain). DeliverTx analizza tutto l'insieme delle transizioni (richieste di aggiornamento della gbest) ed aggiorna il gBest prendendo l'ottimo tra tutte le richieste.



Blockchain Tendermint PSO

La Dapp che eseguirà il PSO comunicherà quindi con la blockchain utilizzando il protocollo RCP (Remote Procedure Call) sulla porta 26657, mentre l'ABCI utilizzerà la libreria JTendermint e quindi comunicherà col Core attraverso la porta 26658 sui quattro socket predefiniti dal protocollo ABCI (Consensus connection, Mempool connection, Info connection, Snapshot connection).



Implementazione dell’ABCI in Java

In questo paragrafo vedremo nei dettagli la realizzazione di una blockchain realizzata ad oh (Tendermint Core + ABCI) per ricevere e gestire le richieste di una Dapp (ovvero un’applicazione decentralizzata) per l’esecuzione dell’algoritmo PSO parallelo asincrono. La Dapp dovrà gestire un certo numero di boid che eseguono il processo di ottimizzazione sull’intero spazio delle soluzioni. Ogni sottogruppo di boid esegue in parallelo la ricerca della soluzione ottimale. Ogni volta che un gruppo, che risiede su un nodo della rete blockchain, trova un valore di “gBest” migliore di quello finora trovato dagli altri gruppi, lo pubblica sulla chain rendendolo disponibile (pubblico) per gli altri gruppi, attraverso i meccanismi di sincronizzazione della blockchain sottostante.

Ad ogni iterazione di ottimizzazione, di ogni sottogruppo, emette una query sulla chain per verificare il valore della gBest. Quest’ultima operazione potrebbe sembrare superflua in quando, gBest rappresenta lo stato dell’app che viene aggiornata a seguito di ogni operazione di approvazione di un nuovo blocco nella chain, in realtà non dobbiamo dimenticare che stiamo interagendo con registro distribuito, pertanto la query (che è un’operazione molto veloce in quando non interessa altri nodi se non quello a cui viene inoltrata la richiesta) ci garantisce la consistenza dell’informazione, sospendendo la request se nel frattempo si sta eseguendo l’approvazione di un nuovo blocco. Utilizzeremo il processo DeliverTx per calcolare il gBest minimo, questo approccio ci sembra utile al fine di gestire più richieste di aggiornamento della gBest per lo stesso mempool.

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.

Modificheremo l'algoritmo visto nel paragrafo precedente con un flusso delle operazioni simile a quello mostrato in Figura 9. Non verrà implementato il blocco "barrier synchronizer", per cui lo stormo sarà parzialmente sincronizzato nel volo, la sincronizzazione avverrà a livello di gruppo e non per intero stormo.

Andiamo per ordine, definiamo quindi lo stato che deve essere memorizzato nella chain. Per la nostra Dapp è sufficiente memorizzare il valore e la locazione della gBest, pertanto definiamo dei valori temporanei (utili durante l'approvazione di un nuovo blocco) e valori correnti:

```
// stato dell'app : gbest
public static double gbest = 1000;
public static double xgbest = 0;
public static double ygbest = 0;

public static double gbest_cache = 1000;
public static double xgbest_cache = 0;
public static double ybest_cache = 0;
```

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 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. In questo caso vogliamo che la transizione sia una stringa composta da due parti separate da ":" es. "node0:3.5a3a7" dove node0 è la sorgente e 3.5a3a7 rappresenta rispettivamente valore e posizione della gbest proposta.

```java
@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 (ad esempio verifica la consistenza dei conti economici per verificare se la transizione è accettabile). 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 la 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. Nel nostro caso si riesegue il parsing dei messaggi

e si individua il nuovo gbest salvandolo nel db di cache. 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();
    // parsing dei messaggi
    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);

    // In the delivertx message handler, we will only update gbest in this block.
    // statement è nella forma 1.2a1.1a0.2
    String [] stparts = statement.split("a");
    Double Dgbest = Double.parseDouble(stparts[0]);
    if (Double.compare(Dgbest, gbest_cache) < 0) {
      gbest_cache = Dgbest; // statement è minore di gbest ----> aggiorniamo gbest
      xgbest_cache = Double.parseDouble(stparts[1]);
      ybest_cache = Double.parseDouble(stparts[2]);
    }

    System.out.println("The fact is validated by this node!");
    return ResponseDeliverTx.newBuilder().setCode(CodeType.OK).build();
}
```

Quando l'applicazione vede il messaggio salva il gbest temporaneo in archivio e restituisce l'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 ");
    gbest= gbest_cache;
    xgbest = xgbest_cache;
    ygbest = ybest_cache;
    String result = (long)(gbest*1000)+"a"+(long)(xgbest*1000)+"a"+(long)(ygbest*1000);
    return ResponseCommit.newBuilder().setData(ByteString.copyFromUtf8(String.valueOf(result.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. La blockchain conserva i dati convalidati inviati dalle applicazioni esterne mentre l'applicazione abci conserva il valore di gbest aggiornato ad ogni iterazione.

```
@Override
public ResponseQuery requestQuery (RequestQuery req) {
    String query = req.getData().toStringUtf8();
    System.out.println("Query : " + query);
    String result = gbest+"a"+xgbest+"a"+ygbest;
    if (query.equalsIgnoreCase("gbest")) {
        System.out.println(result);
        return ResponseQuery.newBuilder().setCode(CodeType.OK).setValue(
                ByteString.copyFromUtf8(result)
        ).setLog(result).build();
    }
    return ResponseQuery.newBuilder().setCode(CodeType.BadNonce).setLog("Invalid query").build();
}
```

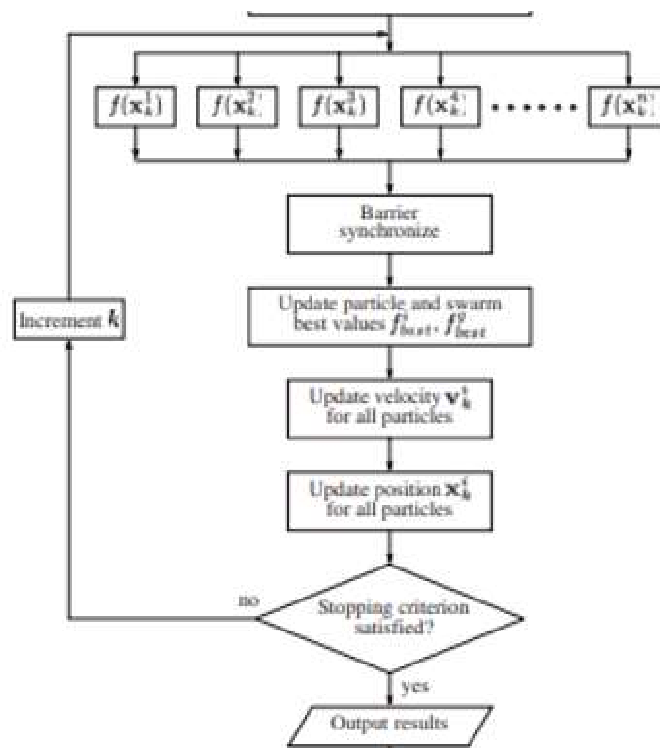
Implementazione della Dapp in Java

In questo paragrafo vedremo nei dettagli la realizzazione di una Dapp (ovvero un’applicazione decentralizzata) per l’esecuzione dell’algoritmo PSO parallelo asincrono. La Dapp dovrà gestire un certo numero di boid che eseguono il processo di ottimizzazione sull’intero spazio delle soluzioni. Ogni sottogruppo di boid esegue in parallelo la ricerca della soluzione ottimale. Ogni volta che un gruppo, che risiede su un nodo della rete blockchain, trova un valore di “gBest” migliore di quello finora trovato dagli altri gruppi, lo pubblica sulla chain rendendolo disponibile (pubblico) per gli altri gruppi, attraverso i meccanismi di sincronizzazione della blockchain sottostante.

Ad ogni iterazione di ottimizzazione, di ogni sottogruppo, emette una query sulla chain per verificare il valore della gBest.

Ripercorrendo tutti gli step dell’algoritmo, dobbiamo implementare i seguenti passaggi:

Step 1: update fitness particle
Step 2: update pBest
Step 3: update gBest
Step 4: update velocity
Step 5: update location
Step 6: stopping criterion satisfied



Gli step 1,2 e 6 sono identici a quelli descritti nel paragrafo precedente, riguardo al Basic PSO:

```
// while optimization
  while(t < MAX_ITERATION && err > ProblemSet.ERR_TOLERANCE) {

      // step 1 - update fitness particle
      updateFitnessList();

      // step 2 - update pBest
      for(int i=0; i<SWARM_SIZE; i++) {
          if(fitnessValueList[i] < pBest[i]) {
              pBest[i] = fitnessValueList[i];
              pBestLocation.set(i, swarm.get(i).getLocation());
          }
      }
```

Lo step tre richiede l'interazione con la blockchain, sia per aggiornare la gBest qualora il gruppo di boid ne abbia determinato una migliore, e sia per richiedere la gBest attuale utile per ricavare la componente sociale delle regole di aggiornamento. Da notare che anche se il sottogruppo produce un gBest migliore di tutti gli altri, deve comunque seguire lo step di aggiornamento e di query. Questo

per due motivi, il primo è che l’aggiornamento sulla blockchain può non andare a buon fine ed il secondo motivo è che se non si effettua una query il gruppo non conosce lo stato attuale della chain.

```
if(t == 0 || fitnessValueList[bestParticleIndex] < gBest) {
          gBest = fitnessValueList[bestParticleIndex];
          gBestLocation = swarm.get(bestParticleIndex).getLocation();
  // invio il mio gbest
  try {
     int ris = postRequest("http://localhost:26657/broadcast_tx_commit?tx=\"node:"
     +String.valueOf(gBest)+"a"+String.valueOf(gBestLocation.getLoc()[0])+"a"
     +String.valueOf(gBestLocation.getLoc()[1])+"\"", "");
     if (ris==1)
       System.out.println("put ok");
     else
       System.out.println("put not ok");
     //System.out.println("gbest:"+ String.valueOf(gb));
  }catch(IOException e) {
    e.printStackTrace();
  }
      }
// ad ogni step richiedo il gbest alla blockchain
try {
   double[] gb = getRequest("http://localhost:26657/abci_query?data=\"gbest\"", "");
   //System.out.println("gbest:"+ String.valueOf(gb));
   // se il gbest globae è minore del mio ---> aggiorno
   if(gb[0] < gBest){
     gBest= gb[0];
     double[] loc = {gb[1],gb[2]};
       gBestLocation.setLoc(loc);
   }
}catch(IOException e) {
  e.printStackTrace();
}
```

Nell’esempio, per semplicità, abbiamo accoppiato un client per ogni nodo Tendermint. In questo caso il client esegue una richiesta in localhost al proprio nodo, diversamente la richiesta andava inoltrata ad uno dei nodi peer reperibile dal file addBock presente nella cartella di configurazione del nodo.

Nella nostra applicazione abbiamo utilizzato due metodi post e get non bloccanti, pertanto in caso di problemi di comunicazione sulla chain, ogni sottogruppo di boid continuerà comunque il suo processo di ottimizzazione considerando il proprio gBest, allineandosi con quello della chain quando riotterrà la linea.

Infine la restante parte dell’algoritmo rispecchia l’algoritmo Basic PSO:

```java
    // update velocity and location
    w = W_UP - (((double) t) / MAX_ITERATION) * (W_UP - W_LO);

    for(int i=0; i<SWARM_SIZE; i++) {
        double r1 = generator.nextDouble();
        double r2 = generator.nextDouble();
        Particle p = swarm.get(i);
        // step 4 - update velocity
        double[] newVel = new double[DIMENSION];
        newVel[0] = (w * p.getVelocity().getPos()[0]) +
                    (r1 * C1) * (pBestLocation.get(i).getLoc()[0] - p.getLocation().getLoc()[0]) +
                    (r2 * C2) * (gBestLocation.getLoc()[0] - p.getLocation().getLoc()[0]);
        newVel[1] = (w * p.getVelocity().getPos()[1]) +
                    (r1 * C1) * (pBestLocation.get(i).getLoc()[1] - p.getLocation().getLoc()[1]) +
                    (r2 * C2) * (gBestLocation.getLoc()[1] - p.getLocation().getLoc()[1]);
        Velocity vel = new Velocity(newVel);
        p.setVelocity(vel);
        // step 5 - update location
        double[] newLoc = new double[DIMENSION];
        newLoc[0] = p.getLocation().getLoc()[0] + newVel[0];
        newLoc[1] = p.getLocation().getLoc()[1] + newVel[1];
        Location loc = new Location(newLoc);
        p.setLocation(loc);
    }

    err = ProblemSet.evaluate(gBestLocation) - 0; // minimizing the functions means it's getting closer to 0

    System.out.println("ITERATION " + t + ": ");
    System.out.println("     Best X: " + gBestLocation.getLoc()[0]);
    System.out.println("     Best Y: " + gBestLocation.getLoc()[1]);
    System.out.println("     Value: " + ProblemSet.evaluate(gBestLocation));

    t++;
}// while optimization

System.out.println("\nSolution found at iteration " + (t - 1) + ", the solutions is:");
System.out.println("     Best X: " + gBestLocation.getLoc()[0]);
System.out.println("     Best Y: " + gBestLocation.getLoc()[1]);
```

Per il primo test dell'applicazione, si utilizza come di consueto una blockchain composta da un solo nodo validatore. Le operazioni da eseguire per compilare il codice con le modifiche apportate ed avviarlo sono le seguenti:

Per la compilazione della sola app (PsoApp), occorre posizionarsi nella classe principale dove è contenuto il package, nel nostro caso: cd /home/tendermint/pso/src/ e lanciare la compilazione:

javac main/java/com/ringful/blockchain/pso/abci/PsoApp.java

Se l'app viene inserita nello stato package del modulo ABCI, basterà settare nel file pom.xml la classe main, cioè PsoAbci.java e lanciare la compilazione attraverso Maven : ***cd pso ---> mvn package -U***

In questo modo verranno compilate ed importate tutte le librerie necessarie.

Per l'avvio dell'applicazione, occorre seguire i soliti passaggi, ovvero aprire tre finestre dove lanciare le tre applicazioni: Tendermint Core, PsoAbci ed PsoApp.

Nella prima finestra di lanciano i comandi ***tendermint unsafe_reset_all*** seguito da ***tendermint node,***

3. 192.168.1.14 (tendermint) | 4. 192.168.1.14 (tendermint) | 5. 192.168.1.14 (tendermint) | 6. 192.168.1.14 (tendermint)

```
tendermint@Host-008:~$ tendermint  unsafe_reset_all
I[2021-12-20|12:47:55.128] Removed existing address book              module=main file=/home/tendermint/.tendermint/conf
ig/addrbook.json
I[2021-12-20|12:47:55.129] Removed all blockchain history             module=main dir=/home/tendermint/.tendermint/data
I[2021-12-20|12:47:55.136] Reset private validator file to genesis state module=main keyFile=/home/tendermint/.tendermint/
config/priv_validator_key.json stateFile=/home/tendermint/.tendermint/data/priv_validator_state.json
tendermint@Host-008:~$ tendermint node
E[2021-12-20|12:48:09.548] abci.socketClient failed to connect to tcp://127.0.0.1:26658.  Retrying... module=abci-client c
onnection=query err="dial tcp 127.0.0.1:26658: connect: connection refused"
E[2021-12-20|12:48:12.549] abci.socketClient failed to connect to tcp://127.0.0.1:26658.  Retrying... module=abci-client c
onnection=query err="dial tcp 127.0.0.1:26658: connect: connection refused"
E[2021-12-20|12:48:15.549] abci.socketClient failed to connect to tcp://127.0.0.1:26658.  Retrying... module=abci-client c
onnection=query err="dial tcp 127.0.0.1:26658: connect: connection refused"
I[2021-12-20|12:48:18.799] Version info                               module=main software=0.32.3 block=10 p2p=7
I[2021-12-20|12:48:18.809] Starting Node                              module=main impl=Node
I[2021-12-20|12:48:18.815] Started node                               module=main nodeInfo="{ProtocolVersion:{P2P:7 Bloc
k:10 App:0} ID_:ddbd9912922635fef59b3431f9c1aceababdb589 ListenAddr:tcp://0.0.0.0:26656 Network:test-chain-EbIERC Version:
0.32.3 Channels:4020212223303800 Moniker:deb11x64 Other:{TxIndex:on RPCAddress:tcp://127.0.0.1:26657}}"
I[2021-12-20|12:48:19.856] Executed block                             module=state height=1 validTxs=0 invalidTxs=0
I[2021-12-20|12:48:19.863] Committed state                            module=state height=1 txs=0 appHash=2D37323939363
4323733
I[2021-12-20|12:48:20.915] Executed block                             module=state height=2 validTxs=0 invalidTxs=0
I[2021-12-20|12:48:20.918] Committed state                            module=state height=2 txs=0 appHash=2D37323939363
4323733
```

mentre nella seconda finestra cd pso ---> ***java -jar target/pso-1.0.jar*** .

3. 192.168.1.14 (tendermint) | 4. 192.168.1.14 (tendermint) | 5. 192.168.1.14 (tendermint) | 6. 192.168.1.14 (tendermint)

```
tendermint@Host-008:~/pso$ java -jar target/pso-1.0.jar
Starting PSO Abci
Commit
Commit
```

Infine nella terza finestra si lancia l’app col comando: cd   /home/tendermint/pso/target/classes/

java main.java.com.ringful.blockchain.pso.abci.PsoApp

3. 192.168.1.14 (tendermint) | 4. 192.168.1.14 (tendermint) | 5. 192.168.1.14 (tendermint) | 6. 192.168.1.14 (tendermint)

```
tendermint@Host-008:~$ cd /home/tendermint/pso/target/classes/
tendermint@Host-008:~/pso/target/classes$ java main.java.com.ringful.blockchain.pso.abci.PsoApp
{
  "jsonrpc": "2.0",
  "id": "",
  "result": {
    "check_tx": {},
    "deliver_tx": {},
    "hash": "0B1FE92F1F785364F6310F6CAFF9348F1CA040E5B7B2EDF935923C58E9BECD14",
    "height": "3"
  }
}
put ok
{
  "jsonrpc": "2.0",
  "id": "",
  "result": {
    "response": {
      "log": "0.5143304745418636a1.5280832428714959a2.6272546969548376",
      "value": "MC41MTQzMzA0NzQ1NDE4NjM2YTEuNTI4MDgzMjQyODcxNDk1OWEyLjYyNzI1NDY5Njk1NDgzNzY="
    }
  }
}
ITERATION 0:
```

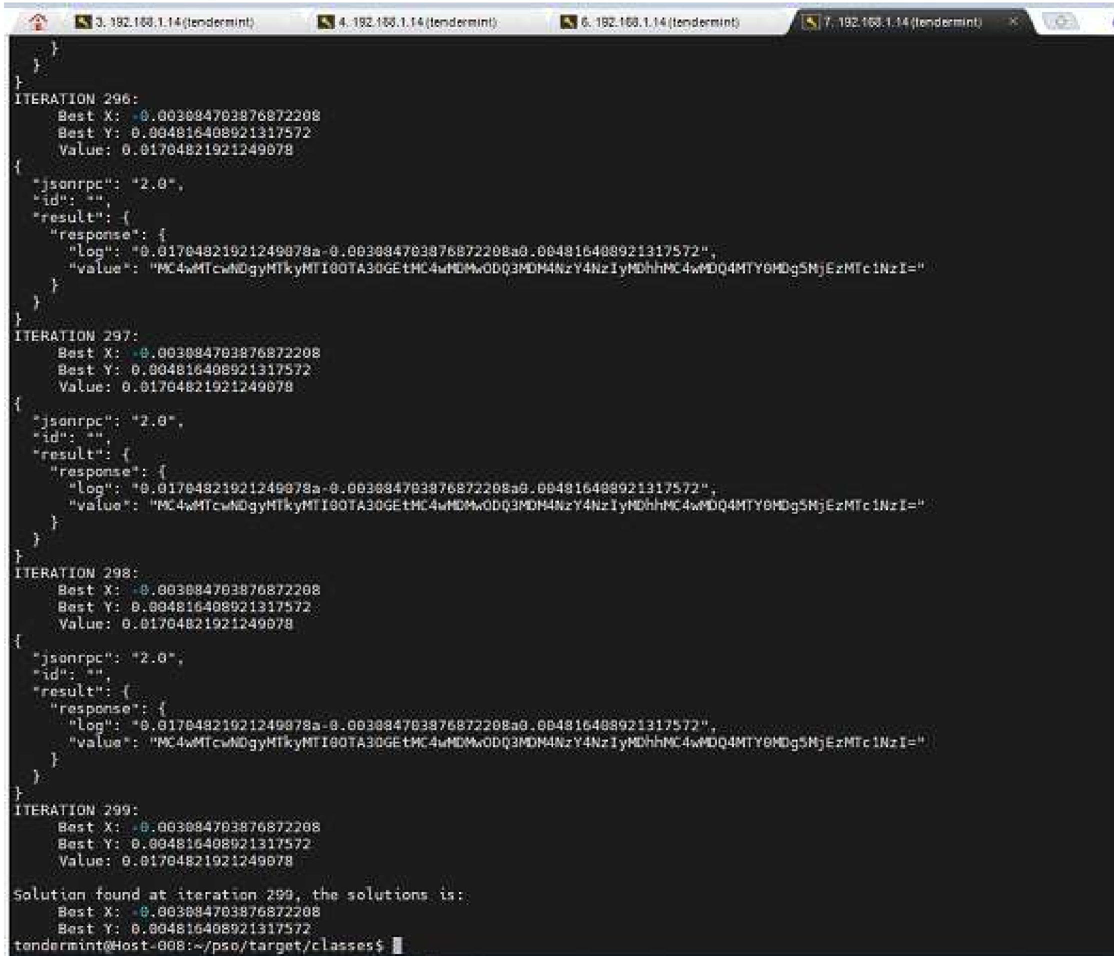
oppure possiamo interagire attraverso linea di commando, se vogliamo testare il funzionamento del modulo ABCI attraverso i comandi:

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

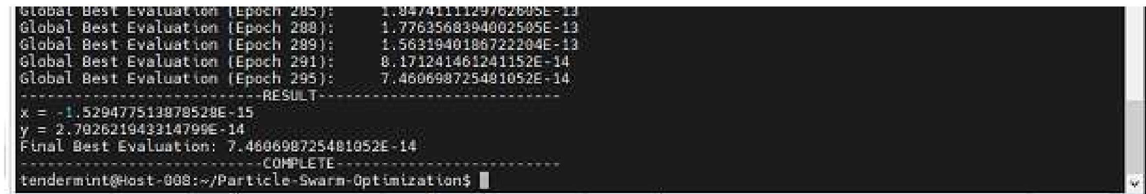
curl -s 'localhost:26657/broadcast_tx_commit?tx="node0:5.1a4.2a3.3"'

curl -X POST http://localhost:26657/broadcast_tx_commit --data tx=\"prova:6\"

L’app è stata testata con le funzioni di test canoniche (Ackley's function, Booth's function, Three-Hump Camel function), riportando in termini di step di computazione risultati identici all’applicazione Basic PSO.



In questa sequenza vediamo che si raggiunge l’ottimo della funzione di Ackley in circa 300 iterazioni e che ad ogni iterazione la Dapp esegue una query sulla gBest alla chain. Lo stesso risultato lo otteniamo lanciando l’applicazione seriale.



Aprendo le varie finestre, possiamo verificare la sequenza delle richieste inoltrate e l’interazione con la chain.

```
3. 192.168.1.14 (tendermint)   4. 192.168.1.14 (tendermint)   5. 192.168.1.14 (tendermint)   6. 192.168.1.14 (tendermint)
tendermint@Host-008:~$ tendermint  unsafe_reset_all
I[2021-12-20|12:47:55.128] Removed existing address book                 module=main file=/home/tendermint/.tendermint/conf
ig/addrbook.json
I[2021-12-20|12:47:55.129] Removed all blockchain history                module=main dir=/home/tendermint/.tendermint/data
I[2021-12-20|12:47:55.136] Reset private validator file to genesis state module=main keyFile=/home/tendermint/.tendermint/
config/priv_validator_key.json stateFile=/home/tendermint/.tendermint/data/priv_validator_state.json
tendermint@Host-008:~$ tendermint node
E[2021-12-20|12:48:09.548] abci.socketClient failed to connect to tcp://127.0.0.1:26658.  Retrying... module=abci-client c
onnection=query err="dial tcp 127.0.0.1:26658: connect: connection refused"
E[2021-12-20|12:48:12.549] abci.socketClient failed to connect to tcp://127.0.0.1:26658.  Retrying... module=abci-client c
onnection=query err="dial tcp 127.0.0.1:26658: connect: connection refused"
E[2021-12-20|12:48:15.549] abci.socketClient failed to connect to tcp://127.0.0.1:26658.  Retrying... module=abci-client c
onnection=query err="dial tcp 127.0.0.1:26658: connect: connection refused"
I[2021-12-20|12:48:18.799] Version info                                  module=main software=0.32.3 block=10 p2p=7
I[2021-12-20|12:48:18.809] Starting Node                                 module=main impl=Node
I[2021-12-20|12:48:18.815] Started node                                  module=main nodeInfo="{ProtocolVersion:{P2P:7 Bloc
k:10 App:0} ID_:ddbd9912922635fef59b3431f9c1aceababdb589 ListenAddr:tcp://0.0.0.0:26656 Network:test-chain-EbIERC Version:
0.32.3 Channels:4020212223303800 Moniker:deb11x64 Other:{TxIndex:on RPCAddress:tcp://127.0.0.1:26657}}"
I[2021-12-20|12:48:19.856] Executed block                                module=state height=1 validTxs=0 invalidTxs=0
I[2021-12-20|12:48:19.863] Committed state                               module=state height=1 txs=0 appHash=2D373239393634
323733
I[2021-12-20|12:48:20.915] Executed block                                module=state height=2 validTxs=0 invalidTxs=0
I[2021-12-20|12:48:20.918] Committed state                               module=state height=2 txs=0 appHash=2D373239393634
323733
I[2021-12-20|12:51:45.420] Executed block                                module=state height=3 validTxs=1 invalidTxs=0
I[2021-12-20|12:51:45.424] Committed state                               module=state height=3 txs=1 appHash=31313539303737
333939
I[2021-12-20|12:51:45.561] Executed block                                module=state height=4 validTxs=1 invalidTxs=0
I[2021-12-20|12:51:45.564] Committed state                               module=state height=4 txs=1 appHash=2D393135373838
323235
I[2021-12-20|12:51:45.711] Executed block                                module=state height=5 validTxs=1 invalidTxs=0
I[2021-12-20|12:51:45.716] Committed state                               module=state height=5 txs=1 appHash=31343534343938
383333
I[2021-12-20|12:51:45.932] Executed block                                module=state height=6 validTxs=1 invalidTxs=0
I[2021-12-20|12:51:45.936] Committed state                               module=state height=6 txs=1 appHash=2D333635323934
353631
I[2021-12-20|12:51:46.257] Executed block                                module=state height=7 validTxs=1 invalidTxs=0
I[2021-12-20|12:51:46.261] Committed state                               module=state height=7 txs=1 appHash=2D313637383535
35363031
I[2021-12-20|12:51:47.277] Executed block                                module=state height=8 validTxs=0 invalidTxs=0
I[2021-12-20|12:51:47.279] Committed state                               module=state height=8 txs=0 appHash=2D313637383535
35363031
I[2021-12-20|12:51:47.346] Executed block                                module=state height=9 validTxs=1 invalidTxs=0
I[2021-12-20|12:51:47.349] Committed state                               module=state height=9 txs=1 appHash=31303833313431
333737
I[2021-12-20|12:51:47.413] Executed block                                module=state height=10 validTxs=1 invalidTxs=0
I[2021-12-20|12:51:47.415] Committed state                               module=state height=10 txs=1 appHash=3132323632383
7313239
I[2021-12-20|12:51:48.435] Executed block                                module=state height=11 validTxs=0 invalidTxs=0
I[2021-12-20|12:51:48.439] Committed state                               module=state height=11 txs=0 appHash=3132323632383
7313239
```

```
3. 192.168.1.14 (tendermint)   4. 192.168.1.14 (tendermint)   5. 192.168.1.14 (tendermint)   6. 192.168.1.14 (tendermint)
1.8942724700658222E-4a1.0029115107323898a3.0035728052477877
Query : gbest
1.8942724700658222E-4a1.0029115107323898a3.0035728052477877
Query : gbest
1.8942724700658222E-4a1.0029115107323898a3.0035728052477877
Query : gbest
1.8942724700658222E-4a1.0029115107323898a3.0035728052477877
Query : gbest
1.8942724700658222E-4a1.0029115107323898a3.0035728052477877
Query : gbest
1.8942724700658222E-4a1.0029115107323898a3.0035728052477877
Commit
Query : gbest
1.8942724700658222E-4a1.0029115107323898a3.0035728052477877
Query : gbest
1.8942724700658222E-4a1.0029115107323898a3.0035728052477877
Check tx : node:7.569453369888853E-5a0.993590580901972a3.0057190347265017
The source is : node
The statement is : 7.569453369888853E-5a0.993590580901972a3.0057190347265017
The fact is in the right format!
Deliver tx : node:7.569453369888853E-5a0.993590580901972a3.0057190347265017
The source is : node
The statement is : 7.569453369888853E-5a0.993590580901972a3.0057190347265017
The fact is validated by this node!
Commit
Query : gbest
7.569453369888853E-5a0.993590580901972a3.0057190347265017
Check tx : node:7.86415512601804E-6a0.9980601686279404a3.0020189524069103
The source is : node
The statement is : 7.86415512601804E-6a0.9980601686279404a3.0020189524069103
The fact is in the right format!
Deliver tx : node:7.86415512601804E-6a0.9980601686279404a3.0020189524069103
The source is : node
The statement is : 7.86415512601804E-6a0.9980601686279404a3.0020189524069103
The fact is validated by this node!
Commit
Query : gbest
7.86415512601804E-6a0.9980601686279404a3.0020189524069103
Query : gbest
7.86415512601804E-6a0.9980601686279404a3.0020189524069103
Query : gbest
7.86415512601804E-6a0.9980601686279404a3.0020189524069103
Query : gbest
7.86415512601804E-6a0.9980601686279404a3.0020189524069103
Query : gbest
7.86415512601804E-6a0.9980601686279404a3.0020189524069103
Query : gbest
7.86415512601804E-6a0.9980601686279404a3.0020189524069103
Query : gbest
7.86415512601804E-6a0.9980601686279404a3.0020189524069103
Query : gbest
7.86415512601804E-6a0.9980601686279404a3.0020189524069103
Query : gbest
7.86415512601804E-6a0.9980601686279404a3.0020189524069103
```

Possiamo notare che sono state necessari 11 aggiornamenti della gBest sulla chain con la validazione di 11 blocchi.

Se vogliamo che la Dapp venga lanciata nella stessa finestra di comando dell'applicazione ABCI, basterà modificare il metodo mail dell'applicazione Psoabci nel seguente modo:

```
public static void main(String[] args) throws Exception, InterruptedException {
    new Thread(){
    public void run() {
            try {
                Thread.sleep(7000);
                new PsoApp ();
                System.out.println("Starting PSOApp");
            } catch (Exception e) {e.printStackTrace();   }
        }
    }.start();
    new PsoAbci ();

}
```

Verranno creati due thread, uno per l'applicazione ABCI ed uno per l'app PSO che viene messo in sleep per un tempo sufficiente al modulo ABCI per stabilire la connessione con Tendermint Core.

Example Distributed Testnet : JAVA PSO

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.

I comandi messi a disposizione dalla piattaforma Tendermint, creano i file di configurazione con l'eseguibile di Tendermint che verranno inseriti nella cartella. build per poi essere copiati nei vari container della rete Docker.

In questa configurazione lanciamo Tendermint Core, ABCI e PSO nello stesso container sfruttando la rete Docker messa a disposizione dalla piattaforma.

Occorre eseguire quindi un minimo di configurazione. Per prima cosa modifichiamo il file config.toml in ogni nodo generato, settando la generazione dei blocchi vuoti a false.

Dopo di che, nel Dockerfile presente nella cartella "tendermint/networks/local/localnode", aggiungiamo la direttiva :

COPY pso-1.0.jar /etc/tendermint/pso-1.0.jar

Per copiare l'applicazione ABCI+PSO con le modifiche effettuate nel paragrafo precedente. Per semplicità utilizziamo l'immagine Docker openjdk:17-alpine:

```
FROM openjdk:17-alpine

RUN apk update && \
apk upgrade && \
apk --no-cache add curl jq file
VOLUME [ /tendermint ]
WORKDIR /tendermint

COPY wrapper.sh /usr/bin/wrapper.sh
COPY config-template.toml /etc/tendermint/config-template.toml
COPY pso-1.0.jar /etc/tendermint/pso-1.0.jar

EXPOSE 26656 26657
CMD ["node"]
ENTRYPOINT ["/usr/bin/wrapper.sh"]
STOPSIGNAL SIGTERM
```

Naturalmente, nella direttiva COPY, o diamo il percorso complete del file pso-1.0.jar, oppure lo copiamo nella stessa cartella del Dockerfile.

Mentre al lancio del container viene avviato Tendermint "node" come da direttiva CMD, l'avvio del modulo ABCI e dell'app vengono inserite all'interno del file wrapper.sh.

Più precisamente dopo il blocco "Assert linux binary " inseriamo il comando:

(sleep 5 && java -jar /etc/tendermint/pso-1.0.jar) &

Il risultato dell'esecuzione è il seguente:

```
14. 192.168.1.18 (tendermint)    15. 192.168.1.18 (tendermint)    16. 192.168.1.18 (tendermint)
tendermint@deb11x64:~$ cd tendermint/
tendermint@deb11x64:~/tendermint$ docher-compose up
-bash: docher-compose: comando non trovato
tendermint@deb11x64:~/tendermint$ docker-compose up
[+] Running 4/0
 # Container node3  Created                                                  0.0s
 # Container node0  Created                                                  0.0s
 # Container node1  Created                                                  0.0s
 # Container node2  Created                                                  0.0s
Attaching to node0, node1, node2, node3
node3  | 2021-12-20T14:48:10Z INFO starting service impl=multiAppConn module=proxy service=multiAppConn
node0  | 2021-12-20T14:48:10Z INFO starting service impl=multiAppConn module=proxy service=multiAppConn
node0  | 2021-12-20T14:48:10Z INFO starting service connection=query impl=localClient module=abci-client service=localClie
nt
node0  | 2021-12-20T14:48:10Z INFO starting service connection=snapshot impl=localClient module=abci-client service=localC
lient
node0  | 2021-12-20T14:48:10Z INFO starting service connection=mempool impl=localClient module=abci-client service=localCl
ient
node0  | 2021-12-20T14:48:10Z INFO starting service connection=consensus impl=localClient module=abci-client service=local
Client
node3  | 2021-12-20T14:48:10Z INFO starting service connection=query impl=localClient module=abci-client service=localClie
nt
node3  | 2021-12-20T14:48:10Z INFO starting service connection=snapshot impl=localClient module=abci-client service=localC
lient
node2  | 2021-12-20T14:48:10Z INFO starting service impl=multiAppConn module=proxy service=multiAppConn
node0  | 2021-12-20T14:48:10Z INFO starting service impl=EventBus module=events service=EventBus
node0  | 2021-12-20T14:48:10Z INFO starting service impl=PubSub module=pubsub service=PubSub
node3  | 2021-12-20T14:48:10Z INFO starting service connection=mempool impl=localClient module=abci-client service=localCl
ient
node3  | 2021-12-20T14:48:10Z INFO starting service connection=consensus impl=localClient module=abci-client service=local
Client
node3  | 2021-12-20T14:48:10Z INFO starting service impl=EventBus module=events service=EventBus
node3  | 2021-12-20T14:48:10Z INFO starting service impl=PubSub module=pubsub service=PubSub
node2  | 2021-12-20T14:48:10Z INFO starting service connection=query impl=localClient module=abci-client service=localClie
nt
node2  | 2021-12-20T14:48:10Z INFO starting service connection=snapshot impl=localClient module=abci-client service=localC
lient
node2  | 2021-12-20T14:48:10Z INFO starting service connection=mempool impl=localClient module=abci-client service=localCl
ient
node2  | 2021-12-20T14:48:10Z INFO starting service connection=consensus impl=localClient module=abci-client service=local
Client
node2  | 2021-12-20T14:48:10Z INFO starting service impl=EventBus module=events service=EventBus
```

Dopo sette secondi verrà avviata l’app PSO completando l’esecuzione:

```
14. 192.168.1.18 (tendermint)    15. 192.168.1.18 (tendermint)    16. 192.168.1.18 (tendermint)    17. 192.168.1.18 (tendermint)
node2  | 2021-12-20T14:54:44Z INFO executed block height=188 module=state num_invalid_txs=0 num_valid_txs=1
node1  | 2021-12-20T14:54:44Z INFO committed state app_hash=0400000000000000 height=188 module=state num_txs=1
node2  | 2021-12-20T14:54:44Z INFO committed state app_hash=0400000000000000 height=188 module=state num_txs=1
node0  | 2021-12-20T14:54:44Z INFO committed state app_hash=0400000000000000 height=188 module=state num_txs=1
node3  | 2021-12-20T14:54:44Z INFO committed state app_hash=0400000000000000 height=188 module=state num_txs=1
node1  | 2021-12-20T14:54:45Z INFO Timed out dur=978.316171 height=189 module=consensus round=0 step=1
node3  | 2021-12-20T14:54:45Z INFO Timed out dur=972.520995 height=189 module=consensus round=0 step=1
node0  | 2021-12-20T14:54:45Z INFO Timed out dur=977.915134 height=189 module=consensus round=0 step=1
node2  | 2021-12-20T14:54:45Z INFO Timed out dur=983.332703 height=189 module=consensus round=0 step=1
node0  | 2021-12-20T14:54:45Z INFO received proposal module=consensus proposal={"Type":32,"block_id":{"hash":"7675040D9C4C
4B04A63DCF6BCCD04382DD36724374C894B55125439CFBB1188C","parts":{"hash":"E2BF6B0F10B8637158B6D841199C9E346918079D420EBADB098
FAB0505E53584","total":1}},"height":189,"pol_round":-1,"round":0,"signature":"T5FpS6usKiAR5RgtY8ls0mNHZQU5yv+TxEVBUokuoiu+
yBC22PpRFFWrbawg8G6IlFNd8P880rjM2GWm2rczBQ==","timestamp":"2021-12-20T14:54:45.721534557Z"}
node0  | 2021-12-20T14:54:45Z INFO received complete proposal block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55
125439CFBB1188C height=189 module=consensus
node3  | 2021-12-20T14:54:45Z INFO received proposal module=consensus proposal={"Type":32,"block_id":{"hash":"7675040D9C4C
4B04A63DCF6BCCD04382DD36724374C894B55125439CFBB1188C","parts":{"hash":"E2BF6B0F10B8637158B6D841199C9E346918079D420EBADB098
FAB0505E53584","total":1}},"height":189,"pol_round":-1,"round":0,"signature":"T5FpS6usKiAR5RgtY8ls0mNHZQU5yv+TxEVBUokuoiu+
yBC22PpRFFWrbawg8G6IlFNd8P880rjM2GWm2rczBQ==","timestamp":"2021-12-20T14:54:45.721534557Z"}
node3  | 2021-12-20T14:54:45Z INFO received complete proposal block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55
125439CFBB1188C height=189 module=consensus
node1  | 2021-12-20T14:54:45Z INFO received proposal module=consensus proposal={"Type":32,"block_id":{"hash":"7675040D9C4C
4B04A63DCF6BCCD04382DD36724374C894B55125439CFBB1188C","parts":{"hash":"E2BF6B0F10B8637158B6D841199C9E346918079D420EBADB098
FAB0505E53584","total":1}},"height":189,"pol_round":-1,"round":0,"signature":"T5FpS6usKiAR5RgtY8ls0mNHZQU5yv+TxEVBUokuoiu+
yBC22PpRFFWrbawg8G6IlFNd8P880rjM2GWm2rczBQ==","timestamp":"2021-12-20T14:54:45.721534557Z"}
node1  | 2021-12-20T14:54:45Z INFO received complete proposal block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55
125439CFBB1188C height=189 module=consensus
node2  | 2021-12-20T14:54:45Z INFO received proposal module=consensus proposal={"Type":32,"block_id":{"hash":"7675040D9C4C
4B04A63DCF6BCCD04382DD36724374C894B55125439CFBB1188C","parts":{"hash":"E2BF6B0F10B8637158B6D841199C9E346918079D420EBADB098
FAB0505E53584","total":1}},"height":189,"pol_round":-1,"round":0,"signature":"T5FpS6usKiAR5RgtY8ls0mNHZQU5yv+TxEVBUokuoiu+
yBC22PpRFFWrbawg8G6IlFNd8P880rjM2GWm2rczBQ==","timestamp":"2021-12-20T14:54:45.721534557Z"}
node2  | 2021-12-20T14:54:45Z INFO received complete proposal block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55
125439CFBB1188C height=189 module=consensus
node3  | 2021-12-20T14:54:46Z INFO finalizing commit of block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55125439
CFBB1188C height=189 module=consensus num_txs=0 root=0400000000000000
node2  | 2021-12-20T14:54:46Z INFO finalizing commit of block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55125439
CFBB1188C height=189 module=consensus num_txs=0 root=0400000000000000
node3  | 2021-12-20T14:54:46Z INFO executed block height=189 module=state num_invalid_txs=0 num_valid_txs=0
node2  | 2021-12-20T14:54:46Z INFO executed block height=189 module=state num_invalid_txs=0 num_valid_txs=0
node3  | 2021-12-20T14:54:46Z INFO committed state app_hash=0400000000000000 height=189 module=state num_txs=0
node2  | 2021-12-20T14:54:46Z INFO committed state app_hash=0400000000000000 height=189 module=state num_txs=0
node0  | 2021-12-20T14:54:46Z INFO finalizing commit of block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55125439
CFBB1188C height=189 module=consensus num_txs=0 root=0400000000000000
node1  | 2021-12-20T14:54:46Z INFO finalizing commit of block hash=7675040D9C4C4B04A63DCF6BCCD04382DD36724374C894B55125439
CFBB1188C height=189 module=consensus num_txs=0 root=0400000000000000
node1  | 2021-12-20T14:54:46Z INFO executed block height=189 module=state num_invalid_txs=0 num_valid_txs=0
node0  | 2021-12-20T14:54:46Z INFO executed block height=189 module=state num_invalid_txs=0 num_valid_txs=0
node1  | 2021-12-20T14:54:46Z INFO committed state app_hash=0400000000000000 height=189 module=state num_txs=0
node0  | 2021-12-20T14:54:46Z INFO committed state app_hash=0400000000000000 height=189 module=state num_txs=0
node3  | 2021-12-20T14:54:47Z INFO Timed out dur=984.472001 height=190 module=consensus round=0 step=1
node2  | 2021-12-20T14:54:47Z INFO Timed out dur=983.973088 height=190 module=consensus round=0 step=1
```