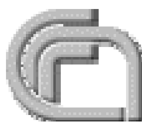
Federated Learning Using PSO with Blockchain-based framework

Emilio Greco, Sabrina Celia, Antonio Francesco Gentile

RT- ICAR-CS-23-01 **Gennaio 2023**



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

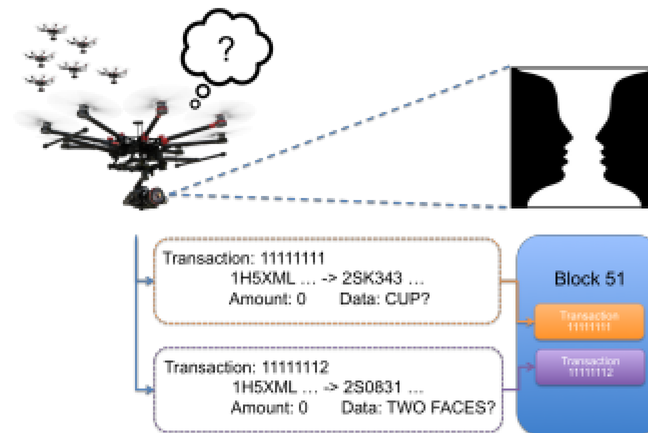
Premessa 3
Introduzione 5
Lavori correlati 8
Blockchain 12
Tendermint 20
  Avvio di Tendermint Core 27
  Resettare la chain 31
  Debug 31
  Local Testnet 32
  Distributed Testnet 34
  Configuration of a Distributed Testnet 36
  Example Distributed Testnet : kvstore 37
Basic PSO 42
Parallel Particle Swarm Optimization 50
Federated Learning PSO 53
Protocollo ABCI 66
Architettura del sistema 66
Implementazione dell'ABCI 69
  Protocollo ABCI : convalida di una transizione 73
  Protocollo ABCI : validazione di un blocco 74
  Protocollo ABCI : gestione delle query 75
Implementazione della Dapp 75
  Example Distributed Testnet 76

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.

Questo lavoro aggiunge un contributo alla ricerca proposta da ***Eduardo Castelló Ferrer*** nel suo articolo: ***"The blockchain: a new framework for robotic swarm systems"*** che mette in luce alcuni

aspetti interessanti sull'utilizzo delle blockchain come framework per applicazioni decentralizzate e calcolo distribuito per algoritmi di swarm intelligence.



Analizzeremo e svilupperemo una soluzione al problema del Federate Learning utilizzando l'algoritmo PSO al fine di ottimizzare lo scambio di informazioni sulla rete. Basandoci sulla tecnologia blockchain, vedremo i passi necessari per realizzare una Dapp (Decentralized application) per sistemi IoT.

L'apprendimento federato è una metodologia che, diversamente dalle tecniche tradizionali che usufruiscono di dati centralizzati su server i quali ospitano il modello previsionale, permette l'addestramento di modelli di machine learning o deep learning come le reti neurali, adoperando dati locali depositati su una moltitudine di nodi (nel nostro caso dispositivi IoT appartenenti ad uno swarm) senza che vi sia uno scambio verso un servizio cloud. L'idea alla base di questa metodologia ***è spostare la fase di training del modello all'origine dei dati*** (cloud vs edge) movimentando delle copie del modello di learning verso le sorgenti ***e non i dati verso una struttura*** centralizzata.

Questa tecnica ***si differenzia dal più comune apprendimento distribuito*** perché con il primo è possibile dividere il carico computazionale derivante dall'addestramento di una rete che adopera un dataset unico ed omogeneo, su più nodi, mentre il federated learning ha lo scopo di ***effettuare il training di più modelli identici***, copie del modello centrale, ***su un set di dati eterogenei provenienti da fonti diverse*** .

Il PSO è una tecnica di ottimizzazione stocastica, che 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.

L'idea che sta alla base dell'integrazione tra l'apprendimento federato ed il PSO, oltre che all'ottimizzazione degli scambi di informazioni sulla rete, è che quest'ultimo può essere adoperato per "calibrare" l'apprendimento dei vari modelli distribuiti sui nodi verso le specificità locali dettate dai dati raccolti, spingendolo verso ottimi locali piuttosto che ottimi globali a seconda delle esigenze. Questo meccanismo potrebbe ad esempio essere utile per gestire set-poit locali su reti per la gestione di condizioni di confort che caratterizzano diverse aree di un edificio. Pur partendo da un modello globale con funzioni di fitness preconfigurate quali ad esempio il risparmio energetico o il tasso di umidità, attraverso il FedPSO si cerca di realizzare un modello globale di gestione di un sistema mantenendo allo stesso tempo funzionalità e tipicità che rispondono a esigenze locali.

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 Contract.** 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 **cripto internet**. 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.

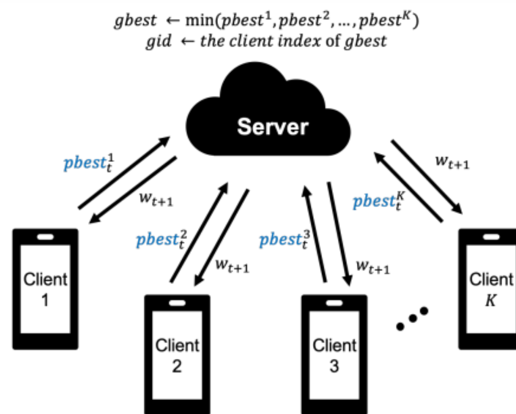
Nel primo capitolo descriveremo in modo più dettagliato una blockchain, le piattaforme attualmente più utilizzare e le tecnologie abilitanti.

Nella seconda parte utilizzeremo un algoritmo di swarm intelligence, il PSO in particolare, per realizzare una rete di scambio di conoscenze tra modelli di previsione (Federated Learning) utilizzando l'infrastruttura di comunicazione e certificazione di una blockchain realizzata ad oh sfruttando il framework Tendermint.

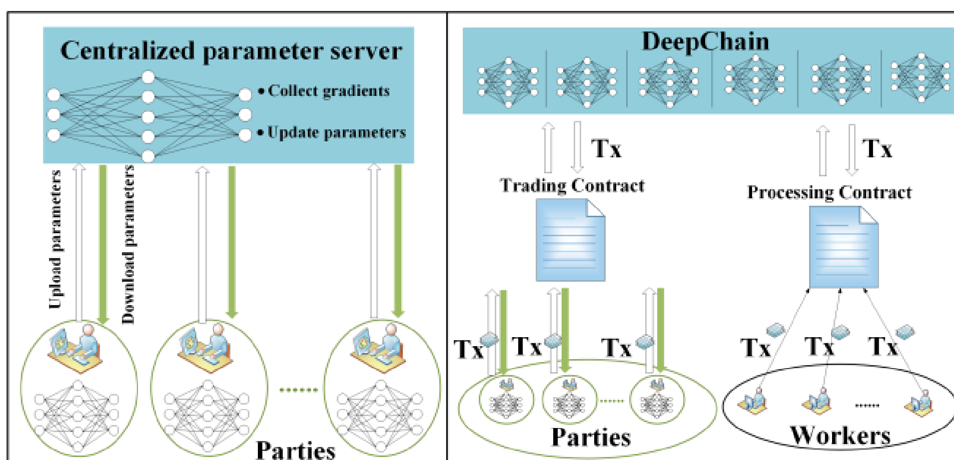
Lavori correlati

Come già anticipato in premessa, questo lavoro è un contributo alla ricerca condotta da ***Eduardo Castelló Ferrer***, riportato nel suo articolo*:* ***"The blockchain: a new framework for robotic swarm systems".*** Nel suo elaborato si mettono in luce alcuni aspetti interessanti sull'utilizzo delle blockchain come framework per applicazioni distribuite e di swarm intelligence. In particolar modo, in questo lavoro analizzeremo e svilupperemo una soluzione al problema del Federate Learning utilizzando una metodologia che fa uso dell'algoritmo PSO. Quest'ultimo lavoro di ricerca è stato condotto da ***S. Park*** in ***"FedPSO: Federated Learning Using Particle Swarm Optimization to Reduce Communication Costs".*** Il team di ricerca presenta un algoritmo interessante che utilizza l'algoritmo

PSO al posto del classico FedAvg per aggiornare il modello globale del sistema e preservando allo stesso tempo il modello locale ricavato da ogni client. Tuttavia l'algoritmo presenta ancora dei margini di miglioramento. Miglioramenti che si possono ottenere eliminando il collo di bottiglia rappresentato dal nodo centrale delegato a raccoglie e ridistribuisce i pesi della rete in fase di addestramento. In ogni caso i risultati presentati dimostrano un incremento delle prestazioni di rete in termini di sulla robustezza e privacy ed anche migliori prestazioni di velocità di esecuzione rispetto all'algoritmo FedAvg.



In ***"[DeepChain: Auditable and Privacy-Preserving Deep Learning with Blockchain-based Incentive](#)"*** gli autori propongono un prototipo di DeepChain, realizzando un framework sicuro e decentralizzato basato sul meccanismo di incentivazione e sulle primitive crittografiche delle Blockchain per preservare la privacy. Sviluppano le basi per un deep learning distribuito, che può fornire riservatezza dei dati, verificabilità dei calcoli e incentivi per le parti a partecipare alla formazione collaborativa.



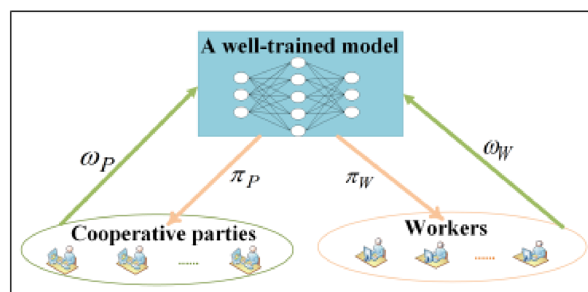
L'applicazione è stata realizza utilizzando la blockchain Corda. Corda è una tecnologia blockchain permissioned molto popolare nel campo finanziario. Ogni nodo conosce soltanto la porzione dei fatti che lo riguarda direttamente, nessun nodo conosce il ledger nella sua interezza. L'oggetto

fondamentale del concetto di Corda è lo state object. Uno stato corrisponde a un oggetto immutabile che rappresenta un fatto conosciuto da uno o più nodi Corda in un dato momento di tempo.

Gli Smart Contract in Corda possono essere considerati come contratti veri e propri. Corda aggiunge un nuovo tipo di validità per quanto riguarda le transazioni, ossia che ogni transazione per essere considerata valida, oltre ad avere tutte le firme necessarie, deve essere contrattualmente valida: ogni stato deve essere collegato a un contratto, il contratto prende in input delle transazioni e controlla se sono considerate valide in base alle regole del contratto stesso. A differenza delle altre tecnologie blockchain, Corda non invia messaggi a tutta la rete, ma usa una messaggistica Point-to-Point. Questo implica che, a ogni proposta di aggiornamento del ledger, tutti i partecipanti debbano sapere esattamente quali informazioni mandare, a chi mandarle e in quale ordine.

Per la fase di addestramento di una rete neurale si utilizza l'algoritmo SGD (Stochastic Gradient Descent). Ogni nodo di calcolo mantiene una copia del modello di addestramento ed un sottoinsieme dell'intero set di dati di input del modello. Quindi, i nodi caricano i loro gradienti di addestramento ricavati localmente su un nodo master, in base ai quali aggiorna il modello globale utilizzando l'algoritmo SGD. Successivamente scaricano i parametri aggiornati dal server e continuano ad eseguire il training del modello locale. Questo processo si ripete finché i nodi non ottengono un modello finale abbastanza attendibile.

Per realizzare l'applicazione si fa uso di due tipi di contratti, il contratto di compravendita ed il contratto di elaborazione che è responsabile dell'aggiornamento dei parametri:

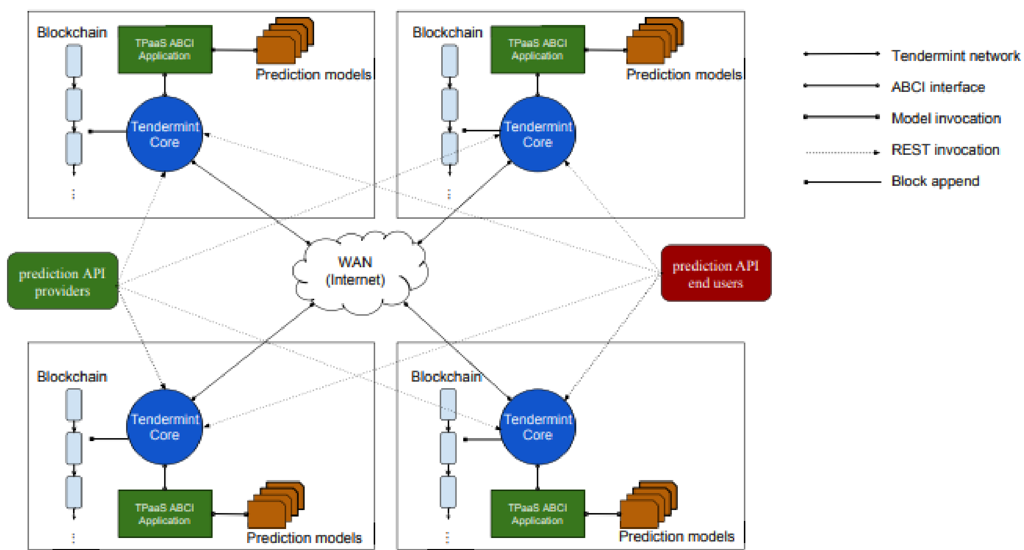


I workers elaborano le transazioni sommando i gradienti e inviano i risultati dei calcoli al contratto di elaborazione. Il contratto di elaborazione verifica che i risultati del calcolo siano corretti e aggiorna i parametri del modello per il gruppo. Una volta definito, il contratto intelligente, può essere eseguito automaticamente in risposta a qualche evento trigger. Questi due contratti sono invocati iterativamente, in modo da portare a termine l'intero processo di addestramento.

La soluzione che proponiamo nel nostro lavoro riguarda la formazione di modelli globali all'interno di una rete di dispositivi IoT quindi limitata ad una comunicazione M2M efficiente, dove la gestione

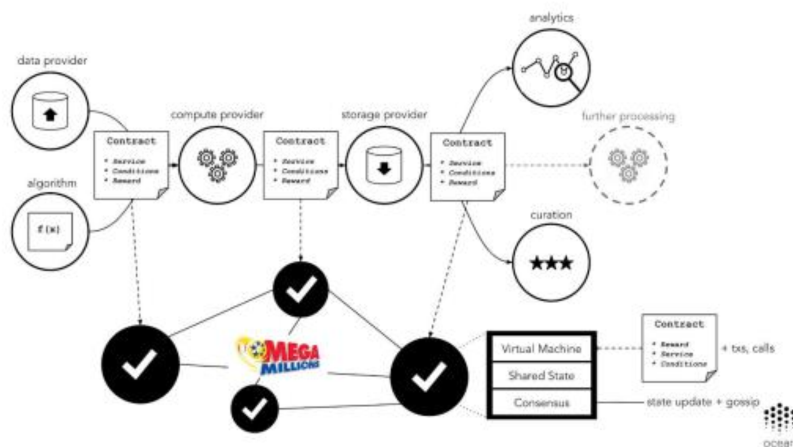
di Oracoli o trigger e Smart Contract limiterebbero le prestazioni, oltre che a complicare la programmazione nella gestione della temporizzazione.

In "***Towards Trustless Prediction-as-a-Service***" viene presentato un framework decentralizzato, per l'appunto, Trustless (basato sulla fiducia) che offre servizi di Prediction. Anch'esso basato su tecnologia blockchain.



Il framework consente l'interazione di fornitori di API di previsione, fornitori di servizi cloud ed utenti creando un nuovo paradigma di business, che va nella direzione del lavoro "***Ocean Protocol: A Decentralized Substrate for AI Data & Services***" migliorandone alcuni aspetti.

Ocean Protocol collega i fornitori dati con i suoi consumatori in modo che i primi possano essere premiati per sbloccare e fornire i propri dati a quest'ultimi. L'infrastruttura consente la connessione, la monetizzazione e la gestione di servizi di dati arbitrari. A questo, Ocean aggiunge ricompense di rete per incentivare la condivisione dei dati, tutelando la privacy.



Skychain è un altro esempio di sforzo nella ricerca verso infrastrutture blockchain che mira ad ospitare, addestrare e utilizzare reti neurali artificiali (ANN). L'elemento centrale dell'ecosistema Skychain è la rete blockchain che fornisce interazione tra Produttori di dati e fornitori di servizi. Un blocco, la cui dimensione è limitata a 10 MB, include la conferma dell'esecuzione delle seguenti operazioni (transazioni):

- Transazioni Skychain Global Coin
- Richieste di inferenza utilizzando una rete neurale
- Richieste di addestramento sulla rete neurale
- Pubblicazione dei risultati dell'inferenza
- Pubblicazione di una nuova rete neurale nel registro
- Modifica del proprietario di una rete neurale
- Aggiornamento di una rete neurale

Il servizio addebita una commissione per ogni transazione. La commissione è fissata dagli autori delle transazioni. Il funzionamento della rete è garantito dal concetto di proof-of-stake.

Il framework adopera il seguente algoritmo per il distribuited Learning:

1. repeat
   a) Send a unique part of the data set to each server
   b) Get updated neural network parameters received through training from each trained server on the forwarded data set $\nabla\omega_\iota$
   c) Calculate updated neural network parameters ω based on all $\nabla\omega_\iota$
   d) Send updated neural network parameters to all trained servers ω
   e) If the data set is completed, break
2. until forever

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 cripto valuta 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 cripto valute 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 cripto valute.

Il terzo impiego della Blockchain risale a innovazioni avvenute verso la metà degli anni '90 con l'introduzione degli Smart Contract. In sostanza uno Smart Contract altro non è che un programma per computer che registra e finalizza un accordo. Sebbene i suoi utilizzi siano tutt'altro che recenti (si pensi ai banali antivirus che rinnovano automaticamente la licenza allo scadere dei termini), la loro

diffusione diviene massiccia grazie al lavoro di Vitalik Buterin, fondatore di Ethereum. Inseriti in un sistema di Blockchain, gli Smart Contract consentono la certificazione dei termini del contratto e la loro esecuzione automatica senza che via sia una terza parte coinvolta.

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

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

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

Golem si appoggia all'algoritmo di consenso PoS (Proof of Stake), per cui i GNT non possono essere minati. **Il progetto si basa sul distributed computing, ovvero sul concetto di calcolo distribuito**. Esso non è altro che un sistema in cui tanti computer comunicano fra loro, pur essendo indipendenti. Chiunque può utilizzare Golem per far girare diversi tipi di software, in diversi settori tra i quali computer grafica, business, machine learning, crittografia, ecc… Cos'è che si condivide tramite questo network decentralizzato su blockchain Ethereum? La potenza di calcolo del proprio pc. Essa

viene ceduta dagli utenti al network stesso. In poche parole si va a prestare una parte della potenza fornita dal PC per essere pagati con la cripto valuta Golem. Si può usare Golem per fare numerose attività, tra cui le predizioni di mercato, inoltre è consentito sviluppare e vendere propri software sul network di Golem. Il network si basa sulla rete peer-to-peer ed usa un linguaggio open source.

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

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

Il funzionamento della blockchain include inoltre ulteriori componenti come:

- Nodo
- Transazione
- Blocco
- Ledger (registro)
- Hash
- Miners

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

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

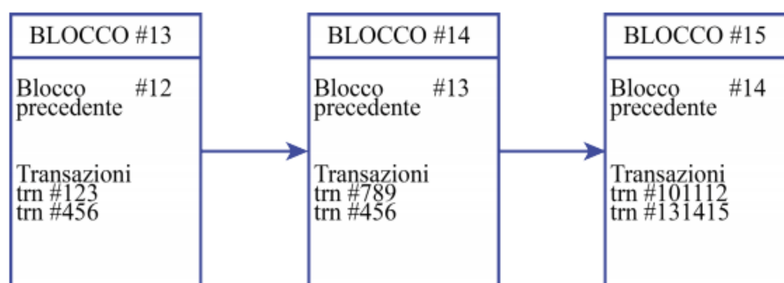


Figura 1- esempio di catena a blocchi

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);
- 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, consentendo lo sviluppo delle application-specific blockchain.

Le **application-specific blockchain** sono blockchain personalizzate realizzate ad oh per una singola applicazione. Si contraddistingue così da una blockchain come ad esempio Ethereum basata su macchine virtuali in grado di interpretare più programmi completi chiamati Smart Contract. Gli 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.

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. Smart Contract che sono gestiti tutti 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 replicata in una blockchain 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.

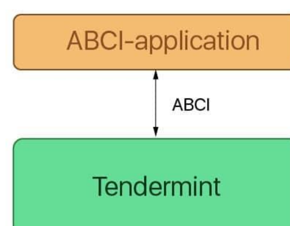


Figura 2 - Struttura di una blockchain basata su Tendermint

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 descritta nel seguente schema:

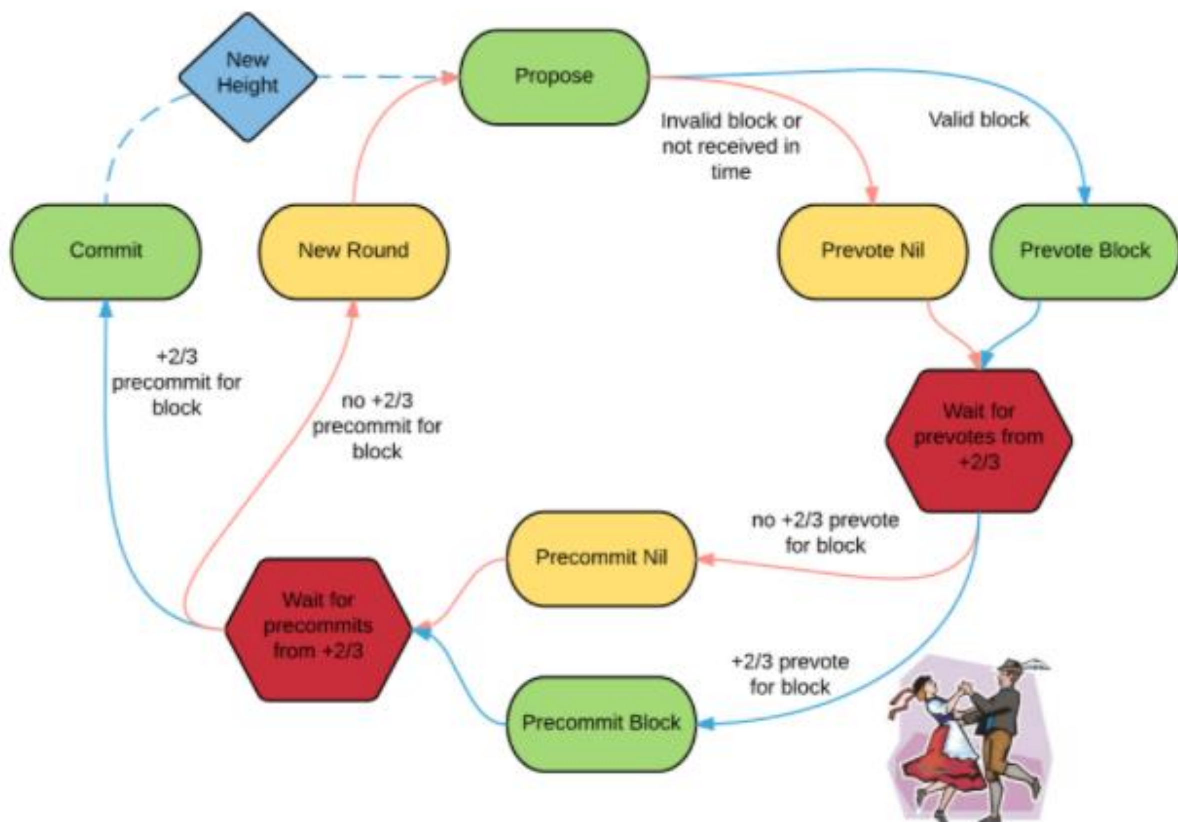


Figura 3- Algoritmo approvazione di un blocco

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.

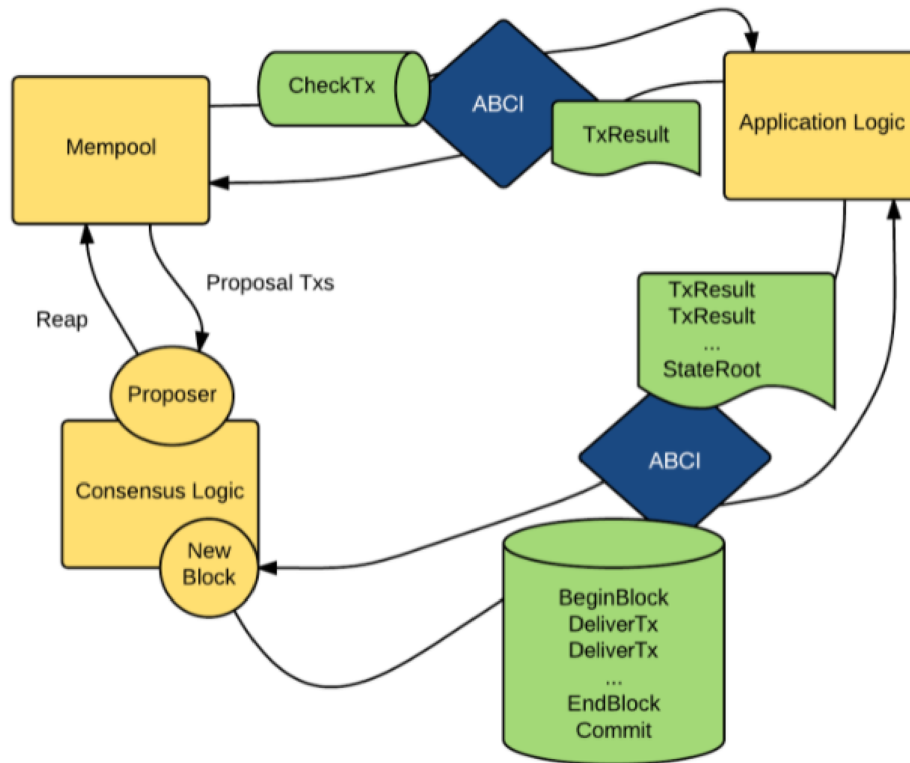


Figura 4 - Flusso dei messaggi dei tre layer applicativi

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:

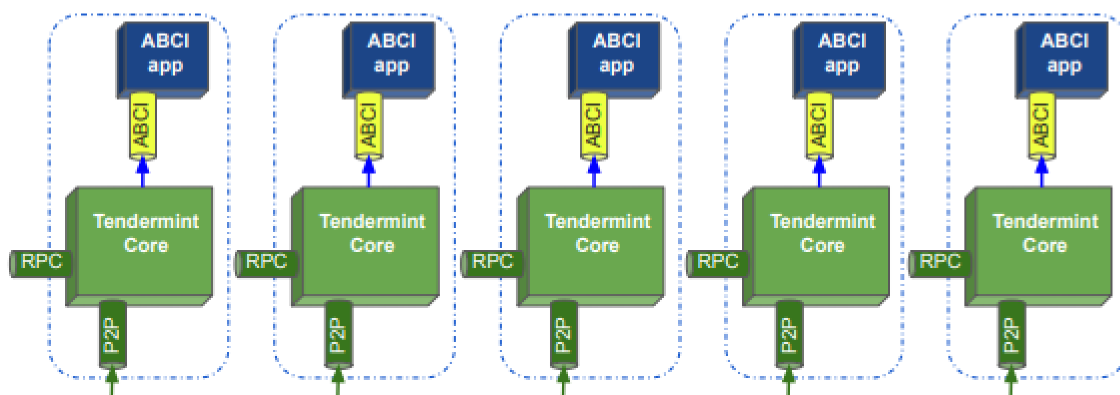


Figura 5 - struttura di una rete blockchain Tendermint

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 nel file genesis.json comune a tutti i nodi che partecipano alla chain, questo per facilitare l'identificazione dei 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 (valore 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). Queste informazioni bisogna reperirli dai file **priv_validator.json e pub_validator.json** che

contendono l'ID del nodo, la chiave pubblica ed il suo indirizzo. Questa configurazione può essere eseguita manualmente, in fase di test, o a run time inserendo queste info nei messaggi ResponseInitChain.

L'ID del nodo può essere inserito all'interno del file config.toml e più precisamente 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 RPC)

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 kvstore 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, verificare lo stato del server o altro, possiamo usare la porta socket utilizzata per le chiamate RPC, 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 visualizzare 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 RPC.

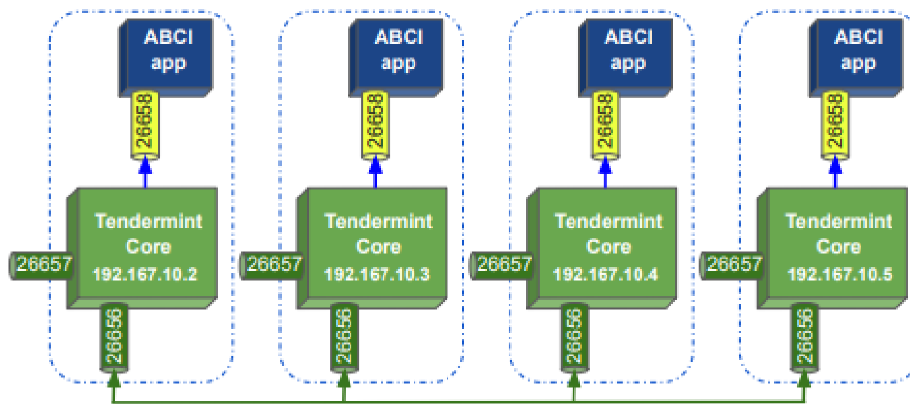


Figura 6 - Schema porte di comunicazione

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 RPC 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 root "home/pi/.tendermint/". La directory data contiene il database delle transizioni, dei blocchi ed il mempool, che sono stati elaborati durante l'esecuzione.

Se queste transizioni non sono 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 comando **tendermint start --log-level "info" o** sul file config.toml

- **Info:** Viene utilizzato per mostrare che tutti i servizi 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 RPC 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: come ad esempio il numero di volte che il nodo si sincronizza, 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 iscrivere 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

Nella libreria di comandi disponibili sulla piattaforma, ne esiste uno che genera ed avvia una rete di nodi, che per impostazione predefinita, definisce quattro nodi validatori che possono essere lanciati sulla stessa macchina. Anche in questo caso, attraverso linea di comando è possibile impostare una diversa configurazione, e definire ad esempio più di 4 nodi validatori o definire altri tipi di nodi client. Se si vuole rendere permanete questa impostazione, si dovrà invece modificando un apposito file di configurazione (Makefile contenuto nella root principale).

Il seguente comando, ad esempio, esegue l'inizializzazione per una testnet composta da cinque nodi, dove il parametro v indica 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, all'interno della quale vi si trovano le chiavi, il file genesis.json comune a tutti i nodi

ed il file config.toml. Questo comando semplifica la fase di test di una applicazione, generando in automatico tutte le configurazioni corrette dei nodi validatori, ovvero settando nei vari file config.toml, contenute nelle cartelle di ogni nodo, gli indirizzi corrispondenti ai peer.

Per evitare conflitti, dato che tutti i nodi, eseguendo questa procedura di test, sono lanciati sulla stessa macchina, in fase di creazione della rete verranno configurate porte di comunicazione differenti. Naturalmente se i nodi sono collocati su macchine diverse o in container, come vedremo successivamente, le porte P2P e RPC saranno identiche e cambieranno in tal caso gli indirizzi IP. 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 ad esempio del 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 replicato variando i parametri corrispondenti. Il lancio del comando avvia la piattaforma consentendo la connessione socket tra i peer e l'esecuzione dell'applicazione di test di default kvstore. Successivamente sarà possibile inoltrare una transazione ad uno dei nodi attraverso il seguente comando:

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

In questo esempio abbiamo sottoposto la nostra richiesta al nodo0, se volessimo inoltrarla al nodo1 bisognerebbe indirizzarla alla porta 26660.

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. Sovrascrivendo i file di configurazione in essi contenute nelle rispettive cartelle dei nodi fisici, avremmo configurato una rete in modo semplice e veloce.

Distributed Testnet

Un altro modo, che la piattaforma ci mette a disposizione, per testare la nostra applicazione è attraverso una rete di container Docker. Docker è una piattaforma che facilita la virtualizzazione a livello di sistema operativo. Questa meccanismo consente la simulazione di un sistema distribuito in esecuzione su una singola macchina, che diversamente dall'approccio visto nel paragrafo precedente, genera ambienti virtuali con propri indirizzi di rete e sui quali si può accedere, interagire, sospendere l'esecuzione, e così via simulando di fatto una rete reale. Per realizzare un'ambiente virtuale dove poter eseguire un'applicazione, innanzitutto è necessario definire un Dockerfile.

Un Dockerfile è un file di script, contenente le istruzioni di configurazione di un ambiente di esecuzione per una specifica applicazione. Creato il file di script è possibile generare una immagine dell'ambiente. Infine, attraverso l'immagine generata è possibile creare diversi containers che rendono eseguibile l'immagine generata in più istanze, consentendone, in modo del tutto indipendente, l'aggiunta, la modifica o l'eliminazione di dati, file, applicazioni. L'immagine Docker può essere quindi utilizzata per creare più contenitori, tutti in esecuzione con la stessa applicazione, ma tutti con il proprio livello di personalizzazione. 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.

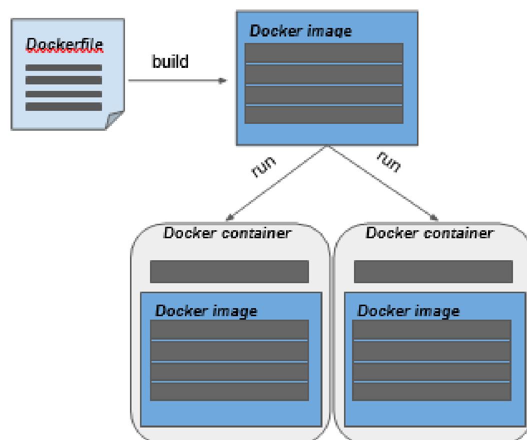


Figura 7 - Deployment dell'applicazione

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 la creazione e l'avvio di tutti i servizi che necessita 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 pre-configurato contenente le configurazioni per una rete di container in cui vengono eseguiti quattro nodi validatori Tendermint con l'applicazione kvstore, replicando di fatto 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 per l'esecuzione di un nodo Tendermint.
2. Genera i file di configurazione di tutti i nodi della rete che inserisce all'interno della root : "tendermint/networks/local/localnode/". Questa è una fase preparatoria allo step successivo, in cui i file generati vengono copiati in ogni container che verrà creato successivamente. Questo automatizza la configurazione dei vari ambienti, per cui i nodi saranno già pronti all'esecuzione, senza la necessità di effettuare ulteriori configurazioni o personalizzazioni.
3. Esegue docker-compose up.

Le configurazioni che si inseriscono nel file docker-compose.yml specificano ad esempio l'indirizzo IP associato ad ogni nodo e le rispettive porte, sia per la comunicazione peer-to-peer che per l'RPC, ed eventuali altre porte che si intendono lasciare aperte per esempio per i tool di monitoraggio e

metriche di rete. Possiamo osservare come necessariamente deve essere mantenuta aperta la posta 26656 che rappresenta la porta predefinita per la comunicazione P2P, mentre rimane chiusa la porta 26658 utilizzata esclusivamente per la comunicazione all'applicazione ABCI in-process (kvstore nel nostro caso). Oltre alla porta 26656 deve rimanere aperta anche la porta 26657 del servizio RPC indispensabile come nel nostro caso per le comunicazioni di Dapp non in-process, ma anche per fare debug e monitoraggio dei singoli nodi. In questo caso la porta del servizio RPC viene mappata sulle porte: 26657, 26660, 26662 e 26664, quindi, le transazioni o le richieste possono essere inviate dall'esterno del container ad una di queste porte. Più precisamente, il servizio RPC del 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 '192.167.10.4:26662/broadcast_tx_commit?tx="name=emilio"'

La transazione verrà inoltrata in questo caso sulla porta RPC di node2 (192.167.10.4:26657).

Configuration of a Distributed Testnet

Per configurare una rete Tendermint con una applicazione client, si procede come di seguito:

1. Si definisce l'ambiente Docker in cui eseguire l'applicazione attraverso un Dockerfile.

2. Si aggiunge un container contenete l'applicazione nel docker-compose.yml, e si associata ad ogni nodo della rete. Ad esempio, se si desidera avviare una rete di cinque nodi Tendermint, in totale devono essere elencati dieci container, cinque per i nodi Tendermint Core e cinque per l'applicazione.

3. Infine occorre configurare, all'interno di ogni container Tendermint Core, il percorso della proxy-app da richiamare per esegue il protocollo ABCI. Questo può essere realizzato, come già discusso nei paragrafi precedenti, a ttraverso linea di comando all'avvio di Tendermint. Ad esempio, se vogliamo che Tendermint Core in esecuzione nel container node0 utilizzi l'app ABCI in esecuzione nel container abci0, il comando da lanciare all'avvio di Tendermint su node0 sarà il seguente:

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 ed i relativi tool a supporto (go, Docker, Docker Composer, etc.). Procediamo con i test sull'applicazione kvstore che è un'applicazione scritta in go, linguaggio nativo di Tendermint, per cui l'applicazione nativa viene definita in-process e comunica con Tendermint Core attraverso la porta 26658, diversamente da applicazioni esterne, scritte in altri linguaggi che invece possono comunicare soltanto attraverso il servizio RPC. La porta su cui è in ascolto il server RPC 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

per lanciare l'esecuzione del nodo oppure il comando:

abci-cli kvstore

quest'ultimo avvia Tendermint Core, l'ABCI ed una app che gestisce le richieste da linea di comando. Nel primo caso, dopo aver lanciato ***tendermint start,*** possiamo interagire con l'applicazione kvstore (lanciata di default dal comando start) attraverso una l'app ***abci-cli*** da una seconda finestra (promt dei comandi) eseguendo 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 RPC, 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. Successivamente eseguiamo il comando:

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

Di contro il comando:

make localnet-stop

termina 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

rispettivamente per avviare e terminare la rete Docker. Il risultato dell'esecuzione sarà 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
```

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 uno spazio di indirizzamento predefinito (192.167.10.0/15), dove ad esempio il nodo0 sarà raggiungibile all'indirizzo 192.167.10.2.

A questo punto possiamo decidere di aprire un'istanza Docker sul nodo0, con il comando:

docker run –it node0

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

abci-cli console

oppure utilizzando il servizio RPC 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.

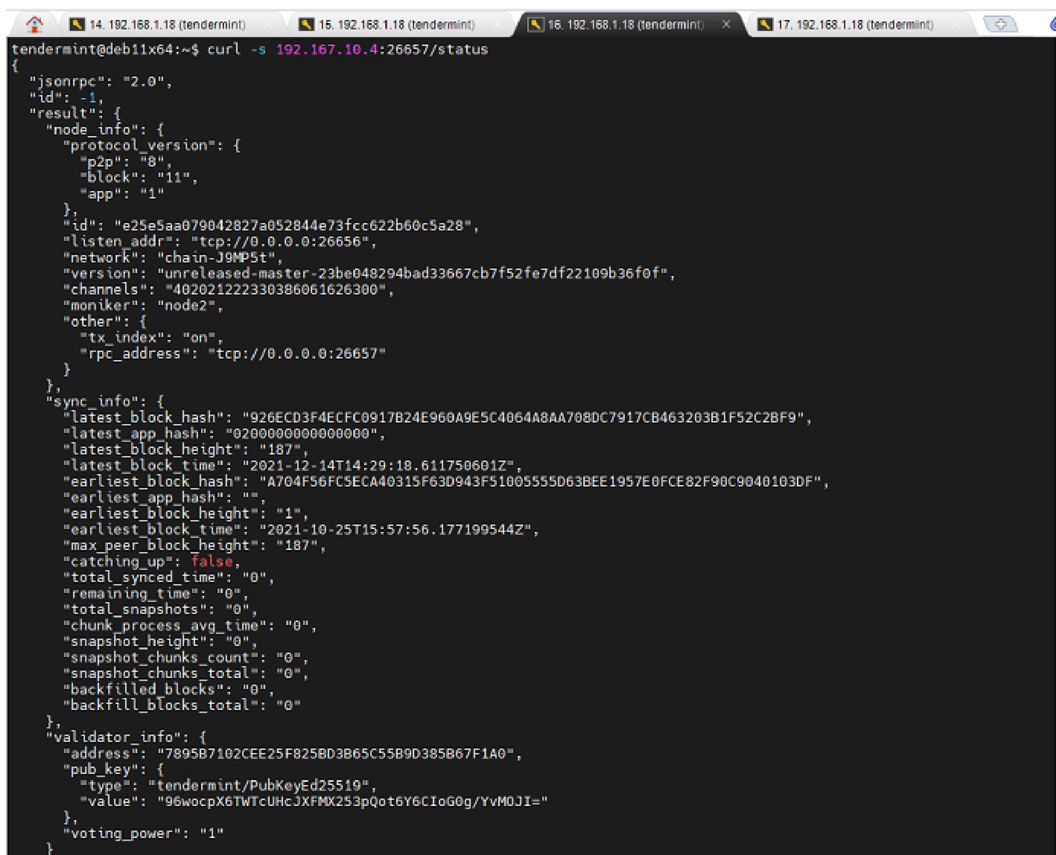
```
14. 192.168.1.18 (tendermint)   15. 192.168.1.18 (tendermint)   16. 192.168.1.18 (tendermint)   17. 192.168.1.18 (tendermint)
tendermint@deb11x64:~$ curl -s '192.167.10.4:26657/broadcast_tx_commit?tx="emilio"'
{
  "jsonrpc": "2.0",
  "id": -1,
  "result": {
    "check_tx": {
      "code": 0,
      "data": null,
      "log": "",
      "info": "",
      "gas_wanted": "1",
      "gas_used": "0",
      "events": [],
      "codespace": "",
      "sender": "",
      "priority": "0",
      "mempoolError": ""
    },
    "deliver_tx": {
      "code": 0,
      "data": null,
      "log": "",
      "info": "",
      "gas_wanted": "0",
      "gas_used": "0",
      "events": [
        {
          "type": "app",
          "attributes": [
            {
              "key": "creator",
              "value": "Cosmoshi Netowoko",
              "index": true
            },
            {
              "key": "key",
              "value": "emilio",
              "index": true
            },
            {
              "key": "index_key",
              "value": "index is working",
              "index": true
            },
            {
              "key": "noindex_key",
              "value": "index is working",
              "index": false
            }
          ]
        }
```

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
```

La verifica che una transizione è stata replicata su tutti i nodi, può essere effettuata attraverso una query. Di fatti la query non coinvolge tutti i nodi della rete ma è il singolo nodo interrogato a produrre i dati richiesti. Pertanto se inoltriamo la richiesta ad un nodo diverso da quello di inserimento della transazione, possiamo verificare la correttezza della duplicazione delle informazioni. In questo caso stiamo chiedendo al node1, l'esistenza della chiave "emilio". La risposta è positiva e la transizione è inserita all'altezza 189 della chain.

Possiamo quindi accedere a tutte le informazioni di un nodo interagendo con il servizio RPC :



Oppure analizzare le informazioni di rete Docker, attraverso il seguente comando:

```
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",
```

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.

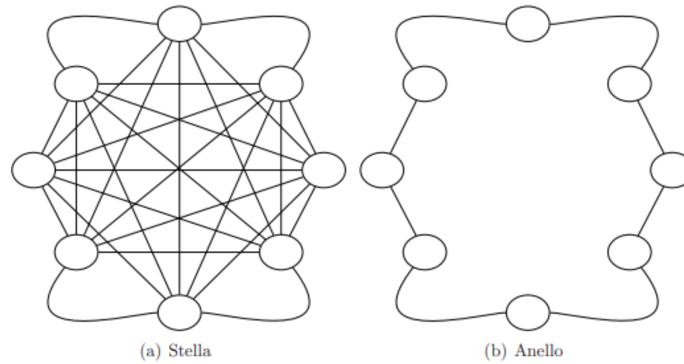
La posizione di ogni particella, calcolata per ogni step temporale ha la seguente funzione:

$$\mathbf{x}_{k+1}^{i} = \mathbf{x}_{k}^{i} + v_{k+1}^{i} \quad (1)$$

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 $f(\vec{\mathbf{x}}_i) < f(\vec{\mathbf{y}}_i)$ **then**
      $\vec{\mathbf{y}}_i \leftarrow \vec{\mathbf{x}}_i$
    end if
    if $f(\vec{\mathbf{y}}_i) < f(\hat{\mathbf{y}})$ **then**
      $\hat{\mathbf{y}} \leftarrow \vec{\mathbf{y}}_i$
    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:

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

ed il vettore posizione

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

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

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

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;

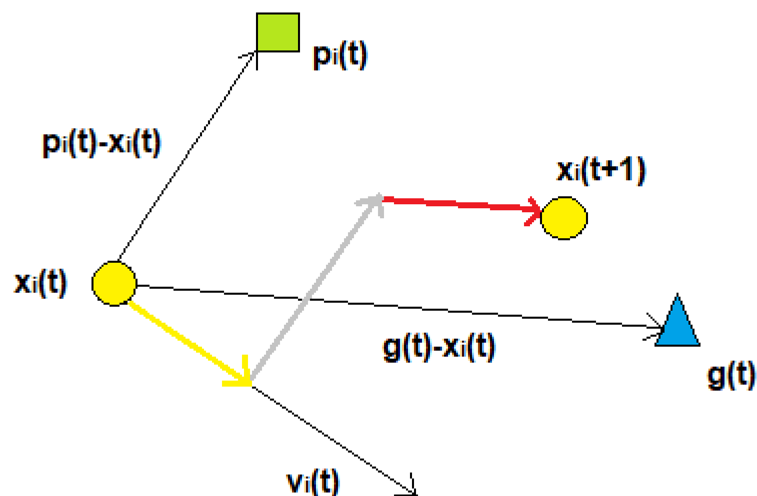
C1: costante di accelerazione per la componente cognitiva;

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.

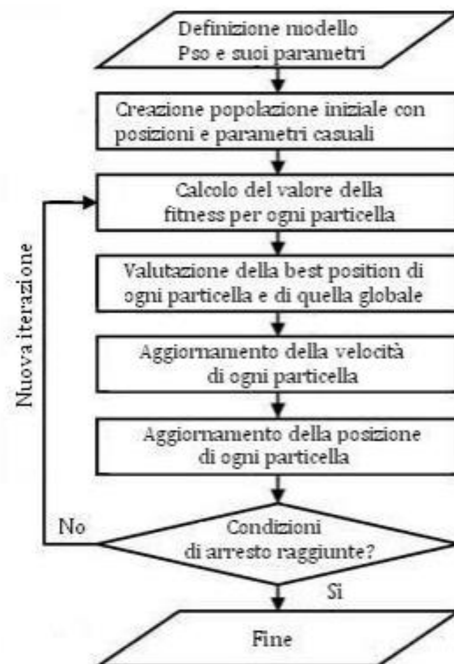


Figura 8 - Algoritmo Basic PSO

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, poiché 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 $v_{i,d}(G)$ 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 sub ottime.

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 $f(xi) \leq |f(x^*) + \varrho$, dove x^* è l’ottimo della funzione obiettivo f ed ϱ è l’errore consentito. Il valore di soglia, ϱ, deve essere scelto con cura. Se ϱ è troppo grande, il processo potrebbe terminare in una soluzione subottima. Se invece ϱ 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.

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

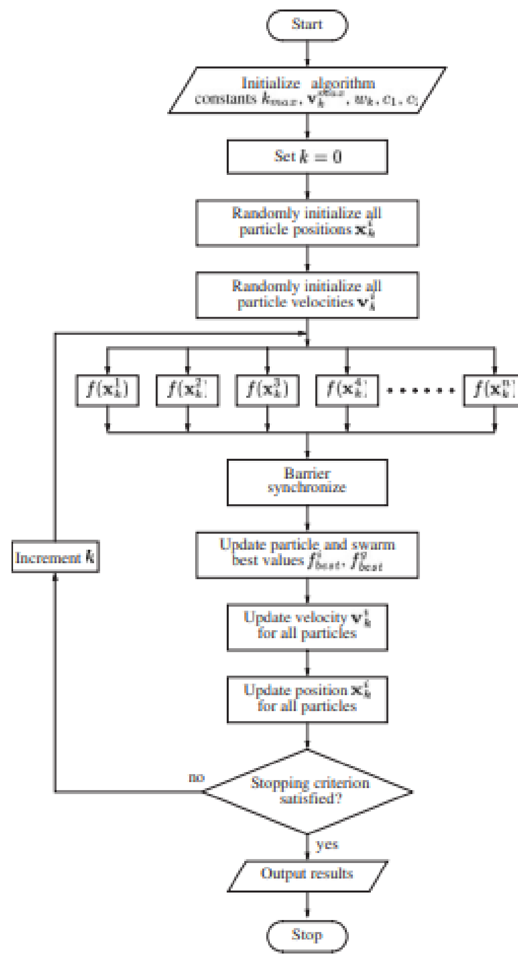
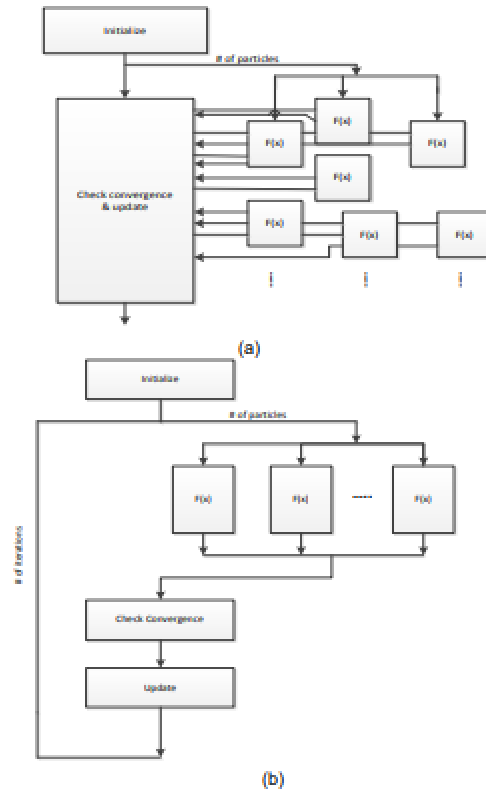


Figura 9 - Algoritmo parallel synchronous PSO



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

Federated Learning PSO

L’apprendimento federato (o meglio conosciuto come Federated Learning, termine americano coniato da Google AI che ha introdotto la tecnica nel 2017) è una metodologia che, diversamente dalle tecniche tradizionali che usufruiscono di dati centralizzati su server i quali ospitano il modello previsionale, permette l’addestramento di modelli di machine learning o deep learning come le reti neurali, adoperando dati locali depositati su una moltitudine di nodi senza lo scambio degli stessi tra un nodo e l’altro o verso un modello centrale. L’idea alla base di questa metodologia ***è spostare la fase di training del modello all’origine dei dati*** movimentando delle copie di questo verso le sorgenti ***e non i dati verso una struttura*** centralizzata. In altre parole, il federated learning connette diversi nodi computazionali (sorgenti dati) in un sistema decentralizzato di apprendimento, consentendo il training di modelli locali, ospitati su questi nodi, con i dati ivi generati e, grazie all’interscambio dei parametri (e.g. variabili sinaptiche come i pesi e il bias di una rete neurale) con il server centrale, consente di costruire un modello globale.

Questa tecnica ***si differenzia dal più comune apprendimento distribuito*** perché con il primo è possibile dividere il carico computazionale derivante dall’addestramento di una rete che adopera un dataset unico, su più nodi, mentre il federated learning ha lo scopo di ***effettuare il training di più modelli identici***, copie del modello centrale, ***su un set di dati eterogenei provenienti da fonti diverse*** (i dati utilizzati nell’approccio distribuito sono omogenei e di stesso volume su ogni nodo, cosa che non accade per la tecnica federata in cui i dataset possono essere di dimensioni molto diverse).

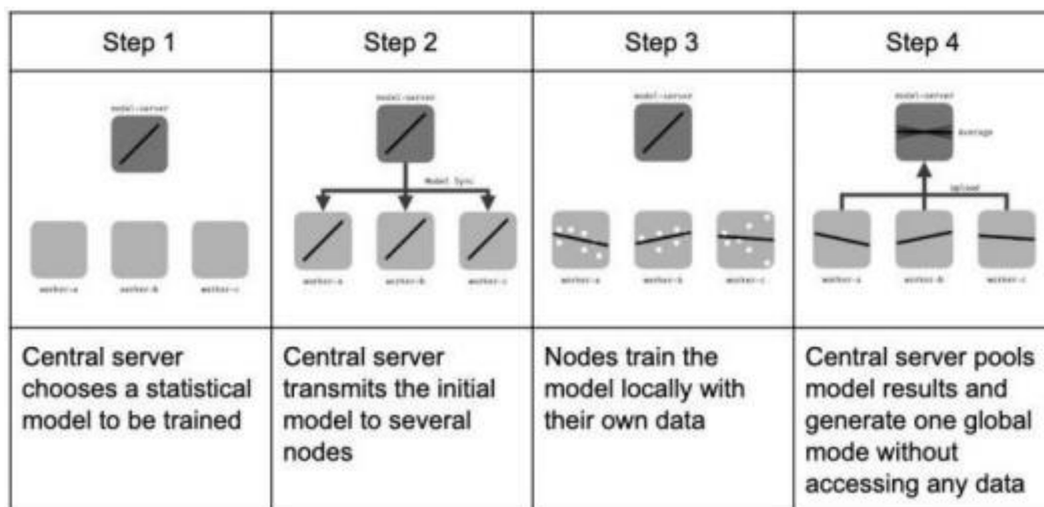
Nella sezione successiva di questo capitolo, andremo più nel dettaglio nel processo di funzionamento di questa nuova metodologia approfondendo i concetti appena introdotti. Processo generale e tipologie di apprendimento federato.

In un sistema di apprendimento federato, i diversi nodi collegati possiedono ciascuno una copia locale del modello centrale. I nodi di computazione eseguono la tradizionale fase di training direttamente sui loro dati locali. Alla conclusione di un’epoca di training, i diversi valori sinaptici/pesi generati dai modelli sono inviati al nodo master, il quale aggrega i parametri e aggiorna le metriche del modello globale.

Il processo appena descritto è un’operazione iterativa che si ripete un certo numero di epoche (federated round) fino al raggiungimento dell’accuratezza target che si desidera. L’apprendimento federato è suddiviso in due tipologie che si differenziano per lo più per la modalità di gestione e supervisione del processo di training e interscambio dei parametri. Il primo approccio, chiamato centralizzato, è caratterizzato da un solo nodo master (un server centrale) a cui è affidato il

coordinamento degli step procedurali del training federato e la gestione dei nodi appartenenti al sistema di apprendimento. Questa entità centrale è incaricata di selezionare le sorgenti dati che parteciperanno al federated learning e di aggregare i parametri generati dal training locale, delle copie del modello globale. Il federated learning centralizzato, nonostante i limiti derivanti dal carico computazione che il server centrale deve sopportare (tutti i nodi invieranno i parametri ad una sola entità che dovrà gestirli), è il più utilizzato nell'ambito del learning collaborativo di più nodi con dati eterogenei.

Questo approccio è spesso suddiviso in quattro diversi passaggi, il primo step è caratterizzato dalla scelta di un modello di apprendimento automatico da parte dell'entità centrale, che inizialmente addestra sulla base dei dati in suo possesso. Al passo successivo, questo modello è trasmesso a tutti i nodi della rete che eseguono l'aggiornamento sui dati locali. Nello step finale tutte le entità partecipanti al federated learning inoltrano i risultati ottenuti (parametri generati) al server centrale che li immagazzina, li elabora (esegue un tipo di aggregazione derivante dall'algoritmo utilizzato per svolgere il federated learning) e aggiorna il modello globale con i nuovi parametri, prima di ritrasmetterli nuovamente ai nodi per ripetere il processo. L'iterazione termina quando si raggiunge l'accuratezza desiderata o il numero massimo di federated round.



L'altra tipologia di federated learning, spesso adoperata, è quella dell'apprendimento decentralizzato in cui i nodi che partecipano alla procedura sono in grado di autogestirsi e coordinarsi autonomamente (queste entità hanno la capacità di procurarsi il modello centrale e di scambiarsi i parametri in modo autonomo, per il resto il processo resta identico).

Questa tecnica risolve la questione di rallentamento del processo dovuto al collo di bottiglia costituito dal nodo centrale e dal carico computazionale che esso si ritrova a gestire perché le entità partecipanti al federated learning hanno la capacità di scambiarsi i risultati e aggiornare il modello globale senza un nodo centrale che faccia da supervisor. Anche questo approccio possiede dei limiti derivanti dalla

configurazione della rete che può condizionare le performance di apprendimento. Alla base del processo di apprendimento Federated troviamo due algoritmi di ottimizzazione principali introdotti entrambi da Google nel 2017.

Il primo è il Federated Stochastic Gradient Descent (FedSGD), evoluzione del più noto algoritmo di discesa stocastica del gradiente (calcolo del gradiente negativo su un subset di dati casuali al fine di trovare la direzione giusta per raggiungere il minimo della funzione di costo generata dal training), utilizzatissimo dagli addetti ai lavori nell'ambito dell'apprendimento delle reti neurali artificiali.

L'algoritmo più famoso in ambito di apprendimento federato, cuore di questa metodologia, è il Federated Averaging (FedAVG). Di seguito possiamo vedere lo pseudo-codice che lo descrive in modo dettagliato.

Algorithm 1 FederatedAveraging (FedAvg) algorithm (simplified from [5]); K = number of clients; E = client total epochs; Select client by the C ratio.

1: **function** SERVEREXECUTES
2: initialize w_0
3: **for** each round $t = 1, 2, \ldots$ **do**
4: $S_t \leftarrow$ (random set of $\max(C \cdot K, 1)$ clients)
5: **for** each client $k \in S_t$ **in parallel do**
6: $w_{t+1}^k \leftarrow$ **ClientUpdate**(k, w_t)
7: $w_{t+1} \leftarrow$ (averaging of the collected weights w_{t+1}^k of S_t clients)

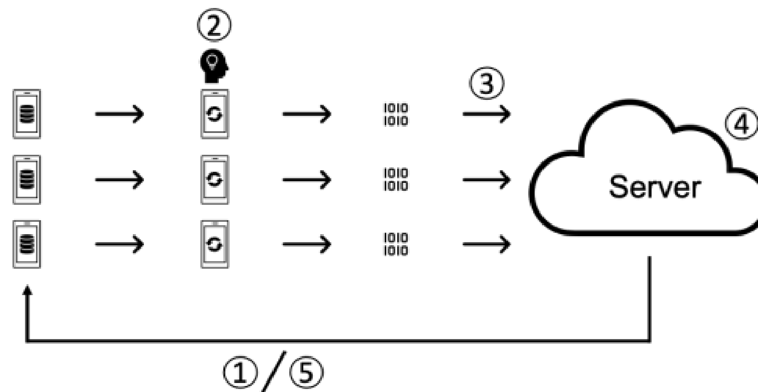
8: **function** CLIENT UPDATE(k, w)
9: Perform learning process on client k with weight w until the client reaches E epoch
10: $w \leftarrow$ updated weight after learning
11: **return** w to server

Il Federated Averaging è un'espansione dell'algoritmo di gradiente stocastico in versione federata, appena descritto. Questa tecnica permette alle entità partecipanti all'apprendimento federato di svolgere molteplici aggiornamenti e di trasmettere i pesi derivanti dalla fase di training dei vari modelli locali. Il federated averaging si caratterizza per il fatto che, in caso le entità della rete di apprendimento federato vengano inizializzate in modo identico, la media dei gradienti calcolati equivale alla media dei valori sinaptici.

I vantaggi primari derivanti dall'approccio federato per il training di un modello sono diversi: in primo luogo esso rende possibile l'utilizzo di diversi dataset eterogenei per addestrare uno stesso modello a svolgere una determinata task in modo sempre più accurato e quindi di acquisire esperienza da una grande varietà di set di dati. Un altro vantaggio sostanziale deriva dal fatto di disporre di copie dei modelli sui vari nodi, il che permette di non dover effettuare uno scambio dei dati computazionalmente pesante e, in conseguenza, di ridurre la latenza del processo.

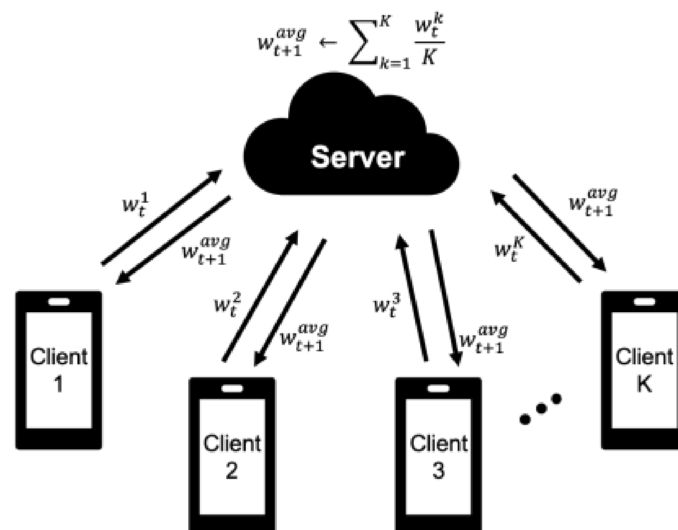
Il processo di apprendimento federato, si può quindi suddividere nei seguenti passi:

1. Il server invia il modello di apprendimento a ciascun client
2. I modelli ricevuti vengono addestrati sui dati del client
3. Ciascun client invia il proprio modello addestrato al server
4. Il server elabora i modelli raccolti e li aggrega in un unico modello globale
5. Il server invia il modello globale a ciascun client e vengono ripetuti i passaggi da 1 a 5

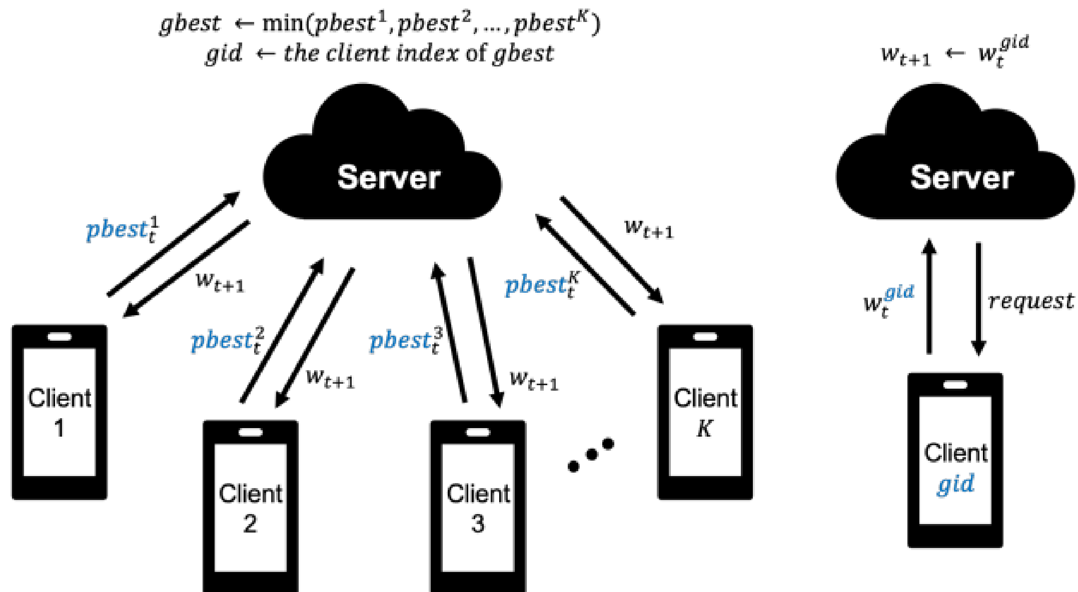


Nel Federated Averaging quello che succede è quando segue:

- Il client che partecipa al round viene selezionato tramite la riga 4.
- Il processo di ricezione dei valori dei pesi appresi dal client è completato attraverso le righe 5 e 6.
- Quando la raccolta dei pesi è completata, viene calcolata la media dei pesi raccolti tramite la riga 7, quindi vengono calcolati i pesi globali.
- Il client riceve i pesi globali dal server e apprende i dati tramite le righe 8–10.



Di seguito invece mostriamo il modello basato su PSO (FedPSO) che riceve i pesi del modello solo per il cliente che ha fornito il punteggio migliore in modo che i pesi del modello non debbano essere trasmessi da tutti clienti. Il processo è mostrato nella figura seguente. Il punteggio migliore viene determinato in base al valore di perdita più basso (valore di loss più piccolo), calcolato dopo l'addestramento di un client. Questo valore di perdita è di soli 4 byte.



FedPSO identifica il miglior modello tramite le variabili pbest e gbest ed aggiorna il valore dei pesi della ANN secondo l'equazione riportata di seguito:

$$\begin{aligned} V_i^t &= \alpha \cdot V_i^{t-1} + c_1 \cdot rand_1 \cdot (pbest - V_i^{t-1}) + c_2 \cdot rand_2 \cdot (gbest - V_i^{t-1}) \\ w_i^t &= w_i^{t-1} + V^t \end{aligned} \quad (2)$$

I passi che compie l'algoritmo quindi sono i seguenti:

1. I client aggiornano i pesi del modello utilizzando l'equazione (2). Eseguono un ciclo di addestrano tenendo in conto sia dati locali che l'informazione globale. Determinano il valore di loss e lo trasmettono al server.
2. Il server calcola il valore minimo di loss tra quelli ricevuti ed il relativo client che lo ha generato.
3. Il server esegue una richiesta di inoltro dei pesi al client che ha prodotto il loss minore.
4. Il server trasmette i pesi alla rete.
5. In ciclo continua al punto 1.

La velocità V calcolata nell'equazione 2, è diversa per ogni layer del modello. Quindi possiamo descrivere l'algoritmo attraverso il seguente pseudocodice:

Algorithm 2 FedPSO

1: **function** ServerExecutes
2: initialize w_0, pbest, gbest, gid
3: **for** each round $t = 1, 2, \ldots$ **do**
4: **for** each client k **in parallel do**
5: $pbest \leftarrow$ **ClientUpdate**(k, w_t^{gid})
6: **if** $gbest > pbest$ **then**
7: $gbest \leftarrow pbest$
8: $gid \leftarrow k$

9: $w_{t+1} \leftarrow$ **GetBestModel**(gid)

10: **function** ClientUpdate(k, w_t^{gid})
11: initialize $V, w, w^{pbest}, \alpha, c_1, c_2$
12: $\beta \leftarrow$ (split ρ_k into batches of size B)
13: **for** each weight layer $l = 1, 2, \ldots$ **do**
14: $V_l \leftarrow \alpha \cdot V_l + c_1 \cdot rand \cdot (w^{pbest} - V_l) + c_2 \cdot rand \cdot (w_t^{gbest} - V_l)$
15: $w \leftarrow w + V$
16: **for** each client epoch i from 1 to E **do**
17: **for** batch $b \in B$ **do**
18: $w \leftarrow w - \eta \nabla l(w; b)$

19: **return** pbest to server

20: **function** GetBestModel(gid)
21: request to **Client**(gid)
22: receive w from **Client**
23: **return** w to server

In questo algoritmo abbiamo un punto di sincronizzazione, dovuto al fatto che il server deve ricevere tutti i pbest per calcolarne il gBest. Inoltre per ogni round è previsto l'aggiornamento del modello, indipendentemente dal fatto che, vi sia o non, un miglioramento della funzione di loss rispetto al round precedente.

Per rendere asincrono l'algoritmo, eliminiamo il punto di sincronizzazione ed aggiorniamo il modello solo se un client trova una combinazione di pesi che riduce il valore della loss globale. Il valore di loss (gBest) globale viene inserito nella blockchian, in modo che ogni nodo potrà consultarlo ed in autonomia decidere se il valore di loss raggiunto ad un round, migliora il valore del gBest. Se ciò avviene, pubblica sulla blockchian il valore di loss con i pesi della ANN corrispondenti. La Chain replicherà i dati su ogni nodo consentendo l'aggiornamento delle ANN su tutti i client.

Se il modello di apprendimento della rete è stato ben ideato, ad ogni epoca di training il loss diminuirà rispetto al gBest calcolato nel round precedente, pertanto possiamo ipotizzare che ad ogni round, vi sia una trasmissione dei pesi sulla chain di almeno un client.

FedPSO Descrizione dell’algoritmo Single-machine

In questo paragrafo illustreremo l’implementazione dell’algoritmo FedPSO Single-machine, ovvero eseguibile in locale su singola macchina e che ci servirà come punto di partenza, sia per confrontare i risultati che per illustrare i passaggi nel renderlo parallelo ed asincrono. Le modifiche che ci proponiamo consentiranno l’integrazione con i metodi RPC definiti dallo stack applicativo ABCI.

Per quando riguarda l’elaborazione dei modelli locali utilizzeremo per semplicità la libreria Keras. Keras è una libreria scritta in Python, un software open source per l'apprendimento automatico e le reti neurali. Offre moduli utili per organizzare differenti livelli. Il tipo principale di modello è quello sequenziale, ovvero una pila lineare di livelli.

È uno strumento che consente una prototipazione facile e veloce, supporta sia reti convoluzionali (CNN) che reti ricorrenti (RNN) o combinazioni di entrambi, supporta schemi di connettività come multi-input e multi-output e funziona sia su CPU che GPU. Per questo primo esempio utilizzeremo il popolarissimo **Boston Housing Dataset**, un dataset contenente diverse informazioni riguardo alcune abitazioni nei dintorni di Boston.

Un dataset strutturato si può presentare in diversi formati: CSV, TSV, XML, HTTP, JSON, EXCEL eccetera, in ogni caso questo ha una struttura tabellare, dove:

- In una delle colonne della tabella è contenuto il valore che vogliamo predire che prende il nome di **target**.
- Tutte le altre colonne sono proprietà che possiamo potenzialmente usare per creare il nostro modello, purché abbiano una relazione con il target, e vengono chiamate **features**.

L’approccio utilizzato del deep learning (apprendimento profondo) è un sotto campo specifico dell’apprendimento automatico: una nuova interpretazione delle rappresentazioni dell’apprendimento dai dati che pone l’accento sull’apprendimento di livelli successivi di rappresentazioni significative. Il “deep” rappresenta questa idea di strati successivi di rappresentazioni.

Queste rappresentazioni a strati vengono (quasi sempre) apprese tramite modelli chiamati reti neurali, strutturati in strati sovrapposti uno sull’altro.

```
from keras.datasets import boston_housing
from keras.callbacks import ModelCheckpoint
from keras.models import Sequential
from keras.layers import Dense
```

Il primo passo è quindi quello di definire i parametri di esecuzione dell'algoritmo. La parametrizzazione riguarda sia l'algoritmo di Learning che il PSO:

```
# client config
NUMOFCLIENTS = 10
EPOCHS = 30
CLIENT_EPOCHS = 5
BATCH_SIZE = 10
W = 0.3
C1 = 0.7
C2 = 1.4
```

Col primo parametro indichiamo il numero di particelle che concorrerà all'ottimizzazione del modello. Nello sviluppo di algoritmi paralleli, all'aumento dei nodi computazionali corrisponde un aumento della velocità di convergenza dell'algoritmo, nei limiti di saturazione della rete di comunicazione sottostante naturalmente. Come detto in premessa, l'obiettivo di questo lavoro non è la realizzazione di una infrastruttura di calcolo parallelo finalizzata a migliorare le prestazioni di convergenza di un algoritmo nella sua versione single-machine, ma bensì si pone l'obiettivo di aumentare la convergenza e l'accuratezza di modelli di previsione locali basati su dati eterogenei realizzati su una infrastruttura di edge computing come potrebbe essere ad esempio una rete di dispositivi IoT. Ad un aumento di client ci aspettiamo di conseguenza o un aumento dell'accuratezza delle previsioni o un aumento della velocità di convergenza del modello locale in fase di training o entrambi. Il termine "velocità" in questo contesto, si riferisce al numero di epoche necessarie a raggiungere i livelli di accuratezza richiesti dal modello.

Il secondo parametro definito in questo primo approccio di calcolo single-machine rappresenta il numero massimo di volte che può essere eseguita l'ottimizzazione del modello globale, ovvero il numero di cicli di esecuzione dell'apprendimento federato. Questo parametro rappresenta un upper bound sull'esecuzione della fase di training. In linea di principio la fase di training dovrebbe concludersi al raggiungimento del valore di accuratezza cercato da ogni singolo nodo, ma è necessario porre un limite per non saturare la rete.

Il terzo parametro definisce il numero di volte che deve essere eseguito il training sui dati locali da parte di ogni particella. In questo caso il valore è uguale per tutti i nodi. Come sopra, questo parametro rappresenta un upper-bound, il criterio di uscita è dato dal valore di accuratezza cercato dal modello. Nell'implementazione asincrona ci potrebbero essere nodi che raggiungono il valore di accuratezza imposto prima di altri e quindi potrebbero terminare ed uscire dal cluster iniziale.

Il parametro BATCH-SIZE definisce il numero di campioni che verranno propagati attraverso il modello di Learning durante l'addestramento. Questo parametro è particolarmente importante quando

si ha a disposizione un dataset per il training abbastanza grande da non poterlo caricare per intero nella memoria della macchina, di conseguenza si passano i dati all'unità di elaborazione suddividendoli per lotti. Questo parametro è particolarmente importante in una architettura IoT basata sull'edge computing dove la memoria dei dispositivi impiegati è molto ridotta. Dimensionare i dati in bach è tipicamente quando avviene nella realtà sui dispositivi che mantengono memoria solo degli ultimi dati raccolti e non hanno grosse capacità di archiviazione.

Gli ultimi tre parametri caratterizzano i parametri di calibrazione dell'algoritmo PSO nella declinazione **Inertia Weigth,** dove w rappresenta l'inerzia del movimento della particella ed è un parametro che cerca di bilanciare due possibili tendenze del PSO: quella di sfruttare l'intorno di soluzioni note e quella di esplorare nuove aree dello spazio di ricerca. I parametri C1 e C2 sono noti come costante di accelerazione per la componente cognitiva e costante di accelerazione per la componente sociale. Nel nostro caso la componente sociale è stata impostata ad un valore doppio rispetto alla componente cognitiva in modo da sfruttare al massimo le soluzioni prodotte dai nodi virtuosi.

Possiamo tranquillamente sorvolare sulle descrizioni dei successivi metodi che sono metodi classici utilizzati rispettivamente per la preparazione del dataset e del modello. Il primo metodo carica il dataset e prepara i dati per il training, mentre il secondo definisce il modello per il deep learning. Naturalmente in un contesto reale, se il training set viene fornito dalla raccolta di dati locali, il primo metodo dovrà essere integrato o sostituito con funzioni che rispecchiano l'eterogeneità dei dati raccolti.

```
def load_dataset():
    (train_data, train_targets), (test_data, test_targets) = boston_housing.load_data()
    # Preparing the data
    mean = train_data.mean(axis=0)
    train_data -= mean
    std = train_data.std(axis=0)
    train_data /= std
    test_data -= mean
    test_data /= std
    return (train_data, train_targets), (test_data, test_targets)

def init_model(train_data):
    model = Sequential(name='BostonHousingPrices')
    model.add(Dense(100, input_shape=(train_data.shape[1],), activation='relu', name='Hidden-1'))
    model.add(Dense(100, activation='relu', name='Hidden-2'))
    model.add(Dense(1, activation='linear', name='Output'))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

A questi due metodi aggiungiamo una terza funzione che suddivide il dataset in modo uniforme per i vari client.

È da notare che questa metodologia viene usta per il traing parallelo in cui ogni nodo che esegue la computazione ha a disposizione una partizione di uno stesso dataset di dati uniformi e dove tutti i nodi concorrono ad addestrare un unico modello. Ricordiamo invece che il Federated Learning nasce dall'esigenza di operare su dataset eterogenei dislocati in vari nodi e di addestrare modelli locali che siano in grado di eseguire inferenze anche su dati globali del sistema. Tuttavia in questa prima fase, faremo uso di questo approccio e successivamente vedremo come sbilanciare i dataset per i test.

```
def client_data_config(x_train, y_train):
    client_data = [() for _ in range(NUMOFCLIENTS)]
    num_of_each_dataset = int(x_train.shape[0] / NUMOFCLIENTS)
    for i in range(NUMOFCLIENTS):
        split_data_index = []
        while len(split_data_index) < num_of_each_dataset:
            item = random.choice(range(x_train.shape[0]))
            if item not in split_data_index:
                split_data_index.append(item)
        new_x_train = np.asarray([x_train[k] for k in split_data_index])
        new_y_train = np.asarray([y_train[k] for k in split_data_index])
        client_data[i] = (new_x_train, new_y_train)
    return client_data
```

Di seguito viene mostrato come realizzare una singola particella:

```
class particle():
    def __init__(self, particle_num, client, x_train, y_train):
        self.particle_id = particle_num
        self.x = x_train
        self.y = y_train
        self.particle_model = client
        self.local_best_weights = client.get_weights()
        self.local_best_score = 0.0
        self.global_best_weights = client.get_weights()
        self.global_best_score = 0.0
        self.parm = {'acc':W, 'local_acc':C1, 'global_acc':C2}
        self.velocities = [None] * len(client.get_weights())
        for i, layer in enumerate(client.get_weights()):
            self.velocities[i] = np.random.rand(*layer.shape)
```

Ogni particella sarà identificata da un id e conterrà una partizione del dataset, nonché una copia del modello che viene replicato su tutti i client e sul nodo server. A questi dati andranno aggiunte le informazioni necessarie per l'esecuzione dell'algoritmo PSO ed in particolare:

- I pesi del modello che ha raggiunto il migliore valore di loss all'epoca corrente, e naturalmente il relativo valore di loss. Questi valori vengono aggiornati dal client dopo l'esecuzione di ogni epoca di training sui dati locali, senza compiere interazioni con l'esterno. Definisce in altre parole il **local best calcolato dal nodo**;

- I pesi del modello che ha raggiunto il migliore valore di loss globalmente da tutto lo swarm, col relativo valore di loss. Caratterizza il **global best** e viene aggiornato al termine d’esecuzione del training di tutte le particelle;

Nella sequenza di test mostrata di seguito, si descrivono i passi eseguiti dall’algoritmo single-machine. Viene indicata con “**Epoch**” una iterazione di addestramento su una singola particella mentre con “**particle fitting**” l’inizio di una fase di training di una particella (nell’esempio vengono riportate cinque epoche per particella per un totale di tre particella). Quando tutte le particelle soso state processate viene eseguita una “**server evaluate**”. È proprio in tale fase che vi è il calcolo del gBest e lo scambio dei dati sulla rete.

```
particle 3/3 fitting
Epoch 1/5
11/11 [==============================] - 0s 9ms/step - loss: 410.9773 - mae: 17.9500 - val_loss: 215.4430 - val_mae: 13.0494
Epoch 2/5
11/11 [==============================] - 0s 5ms/step - loss: 285.4106 - mae: 14.2868 - val_loss: 137.7187 - val_mae: 9.7327
Epoch 3/5
11/11 [==============================] - 0s 6ms/step - loss: 183.3552 - mae: 10.4044 - val_loss: 90.0449 - val_mae: 6.8391
Epoch 4/5
11/11 [==============================] - 0s 5ms/step - loss: 119.5970 - mae: 7.8218 - val_loss: 69.3972 - val_mae: 6.0660
Epoch 5/5
11/11 [==============================] - 0s 5ms/step - loss: 86.7688 - mae: 6.7294 - val_loss: 57.4902 - val_mae: 5.7395
server 2/5 evaluate
11/11 [==============================] - 0s 1ms/step - loss: 34.6625 - mae: 4.7786
particle 1/3 fitting
Epoch 1/5
11/11 [==============================] - 0s 10ms/step - loss: 369.3125 - mae: 16.7932 - val_loss: 148.7206 - val_mae: 10.7608
Epoch 2/5
11/11 [==============================] - 0s 5ms/step - loss: 111.1672 - mae: 8.7822 - val_loss: 50.5556 - val_mae: 5.6083
Epoch 3/5
11/11 [==============================] - 0s 5ms/step - loss: 50.8133 - mae: 5.5194 - val_loss: 30.9644 - val_mae: 4.3606
```

Di seguito mostriamo come si realizza la fase di training inglobando l’algoritmo PSO

```python
def train_particle(self):
    print("particle {}/{} fitting".format(self.particle_id+1, NUMOFCLIENTS))
    # Step 4: update velocity
    step_weight = self.particle_model.get_weights()
    new_weight = [None] * len(step_weight)
    local_rand, global_rand = random.random(), random.random()
    for index, layer in enumerate(step_weight):
        new_v = self.parm['acc'] * self.velocities[index]
        new_v = new_v + self.parm['local_acc'] * local_rand * (self.local_best_weights[index] - layer)
        new_v = new_v + self.parm['global_acc'] * global_rand * (self.global_best_weights[index] - layer)
        self.velocities[index] = new_v
        # Step 5: update location
        new_weight[index] = step_weight[index] + self.velocities[index]
    self.particle_model.set_weights(new_weight)
    # Step 1: update fitness particle
    save_model_path = 'checkpoint/checkpoint_particle_{}'.format(self.particle_id)
    mc = ModelCheckpoint(filepath=save_model_path,
                         monitor='val_loss',
                         mode='min',
                         save_best_only=True,
                         save_weights_only=True,
                         )
    hist = self.particle_model.fit(x=self.x, y=self.y,
            epochs=CLIENT_EPOCHS,
            batch_size=BATCH_SIZE,
            verbose=1,
            validation_split=0.2,
            callbacks=[mc],
            )
    train_score_loss = hist.history['val_loss'][-1]
    self.particle_model.load_weights(save_model_path)
    # Step 2: update pBest
    if self.global_best_score >= train_score_loss:
        self.local_best_weights = self.particle_model.get_weights()
        self.local_best_score  =  train_score_loss
    return self.particle_id, train_score_loss
```

Secondo la legge di aggiornamento del PSO i nuovi pesi del modello saranno ricavati dai pesi al passo precedente a cui sommiamo la nuova velocità:

$x^i_{k+1} = x^i_k + v^i_{k+1}$ ➔ new_weight[index] = step_weight[index] + self.velocities[index]

il valore della velocità viene calcolato quindi per ogni layer del modello come descritto dal codice all'interno del ciclo for.

$$
\begin{aligned}
vi.d(G+1) = {} & w \cdot vi.d(G) \\
& + C1 \cdot Rnd(0, 1) \cdot [pbi.d(G) - xi.d(G)] \\
& + C2 \cdot Rnd(0, 1) \cdot [gbd(G) - xi.d(G)]
\end{aligned}
$$

```
new_v = W * self.velocities[index]
new_v = new_v + C1 * local_rand * (self.local_best_model.get_weights()[index] - layer)
new_v = new_v + C2 * global_rand * (self.global_best_model.get_weights()[index] - layer)
```

Riportiamo di seguito la sequenza degli step dell'algoritmo PSO:

Step 1: update fitness particle
Step 2: update pBest
Step 3: update gBest
Step 4: update velocity
Step 5: update location

È importante mantenere la sequenza delle operazioni nel giusto ordine, risulta irrilevante invece lo step iniziale del ciclo. Nel nostro esempio iniziamo dallo Step 4: update velocity mentre il valore del gBest viene calcolato in modo centralizzato dal nodo server. Quest'ultimo step, nella versione parallela dovrà essere inglobato nel nodo client.

Calcolati i nuovi pesi, si può eseguire l'operazione di addestramento sul dataset locale. A tal fine si utilizza la funzione di callback "ModelCheckpoint" che ci consente di migliorare le prestazioni poiché esegue il salvataggio del modello durante l'esecuzione nel file "save_model_path" ogni volta che la val_loss migliora. In questo modo, possiamo successivamente, scegliere di caricare i pesi che hanno prodotto la migliore val_loss ottenuta nel corso delle varie epoche di training.

Il parametro che viene monitorato durante il training è **val_loss**, ovvero il valore della funzione di costo calcolato su dati di convalida estratti dal trainingset, mentre il loss definisce il valore della funzione di costo ricavato sui dati di formazione. C'è una differenza tra i due dovuta al fatto che ai dati di addestramento si aggiunge un po' di rumore per migliorare l'apprendimento.

Notiamo che il modello locale definito in “**particle_model**” viene comunque aggiornato ad ogni epoca anche se il val_loss non decresce secondo quando definito dell’algoritmo PSO, dove si punta alla ricerca di ottimi globali piuttosto che ottimi locali. Diversamente, il modello contenuto nella variabile “**local_best_weights**” viene aggiornato ogni qualvolta si raggiunge un nuovo minimo nella fase di training, secondo i criteri del calcolo del pBest. Per quando riguarda l’aggiornamento del gBest, questo viene demandato al nodo server.

Il metodo main è composto da un ciclo che viene eseguito per un certo numero di epoche predeterminate, al cui interno troviamo due cicli annidati. Nel primo si esegue il training e si raccolgono i vari val_loss ottenuti. Se il minimo tra i valori raccolti è più piccolo del global best ottenuto all’epoca precedente allora si procede con l’aggiornamento del modello globale contenuto sul server e si esegue la replica sui nodi client nel secondo ciclo for.

```
if __name__ == "__main__":
    (x_train, y_train), (x_test, y_test) = load_dataset()
    server_model = init_model(x_train)
    print(server_model.summary())
    client_data = client_data_config(x_train, y_train)
    pso_model = []
    for i in range(NUMOFCLIENTS):
        pso_model.append(particle(particle_num=i, client=init_model(x_train),
                                  x_train=client_data[i][0], y_train=client_data[i][1]))
    server_evaluate_acc = []
    global_best_score = 0.0
    new_global_best = 0.0
    for epoch in range(EPOCHS):
        server_result = []
        start = time.time()
        for client in pso_model:
            pid, train_score = client.train_particle()
            server_result.append([pid, train_score])
        gid, new_global_best = get_best_score_by_loss(server_result)
        if (epoch == 0 or new_global_best <= global_best_score):
            global_best_score = new_global_best
            for client in pso_model:
                if client.particle_id == gid:
                    server_model.set_weights(client.particle_model.get_weights())
            for client in pso_model:
                if epoch != 0:
                    client.update_global_model(server_model.get_weights(), global_best_score)
```

Protocollo ABCI

ABCI sta per " **A**pplication **B**lock **C**hain **I**nterface". Ovvero rappresenta è l'interfaccia tra Tendermint Core e l'applicazione (la vera macchina a stati). Consiste in un insieme di *metodi* , ciascuno con una corrispondente coppia di messaggi di tipo Request e Response. Per eseguire la replica di una macchina a stati sui nodi, Tendermint chiama i metodi ABCI inviando messaggi di Request e ricevendo in cambio i messaggi di Response.
Questa separazione con la logica di controllo consente a Tendermint di funzionare con applicazioni scritte in molti linguaggi di programmazione.
Quando Tendermint e l'applicazione ABCI vengono eseguiti come processi separati, Tendermint apre quattro connessioni socket all'applicazione per i diversi metodi ABCI. Le connessioni gestiscono ciascuna un sottoinsieme delle chiamate.

Questi sottoinsiemi sono definiti come segue:

Consensus connection

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

Mempool connection

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

Info connection

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

Snapshot connection

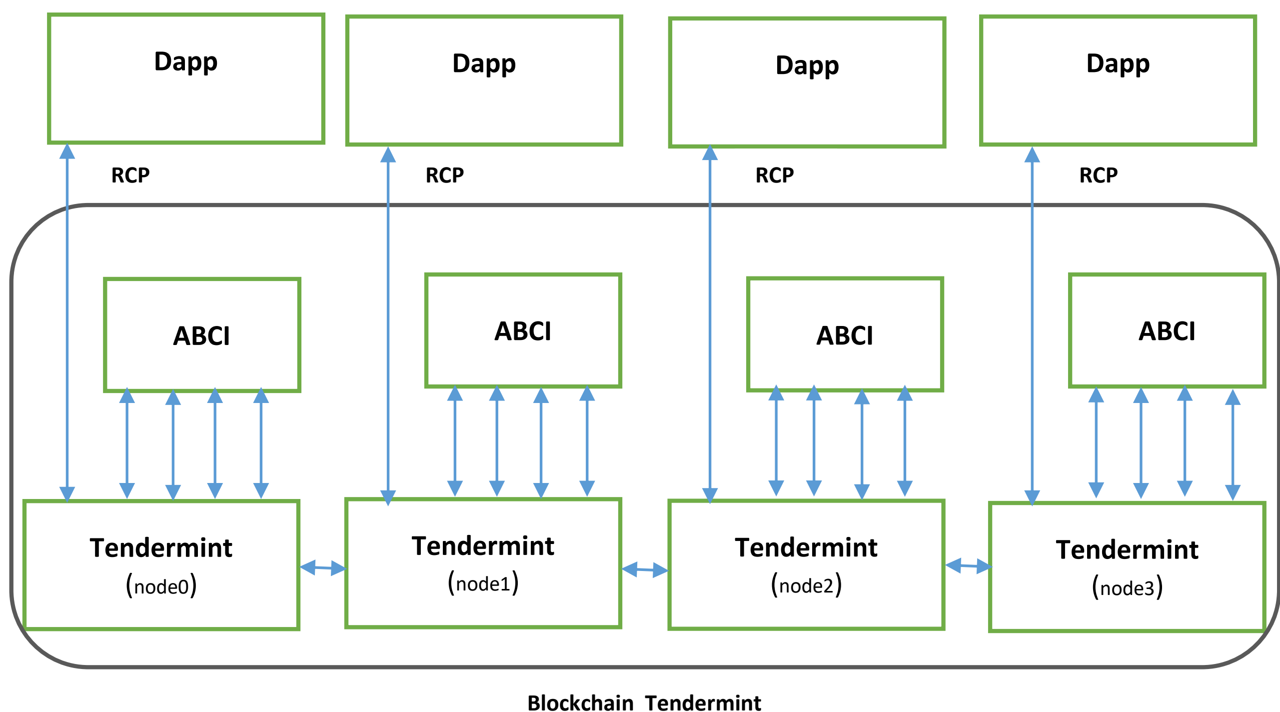
- serve a supportare la sincronizzazione dello stato.
- gestisce le chiamate ListSnapshots, LoadSnapshotChunk, OfferSnapshot, e ApplySnapshotChunk.

Architettura del sistema

L'architettura generale del sistema vede quindi una blockchain che si occupa della replica dello stato di una applicazione detta Dapp, ovvero un'applicazione decentralizzata che generalmente si occupa

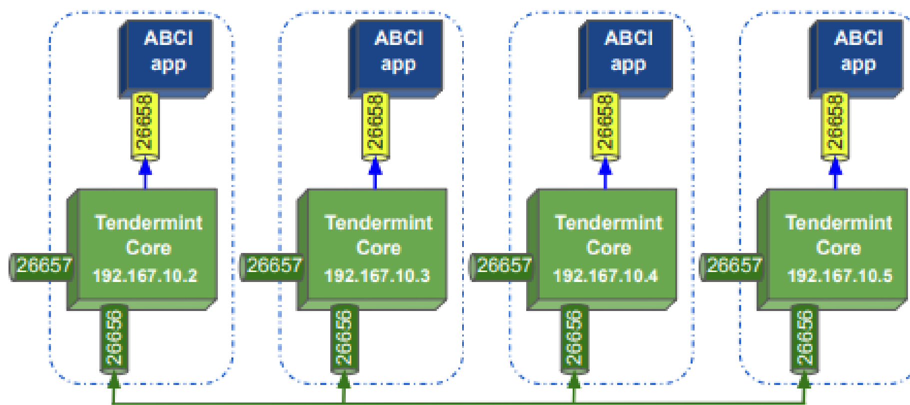
di presentare e fornire agli utenti un servizio. Nel nostro caso, trattando la comunicazione tra due o più macchine, ovvero una comunicazione detta M2M (Machine to Machine) in quando coinvolge l'erogazione e la fruizione di un servizio tra macchine, questo comporta una semplificazione dal punto di vista di presentazione del servizio, quello che generalmente va sotto il nome di GUI (graphical user interface), di contro pone altre sfide sotto diversi punti di vista. Occorre gestire la sincronizzazione tra processi in quando una macchina potrebbe essere più veloce di altre nel processare le richieste, e di conseguenza si pongono anche sfide in termini prestazionali. Ciò premesso, tornando allo stack applicativo, gli elementi che bisogna sviluppare sono essenzialmente due:

1. Il client della Blockchain, ovvero la Dapp centrata sugli aspetti prestazionali e non di presentazione dei dati/servizio
2. Un modulo che gestisce il consenso e la replica dello stato della Dapp , ovvero l'ABCI

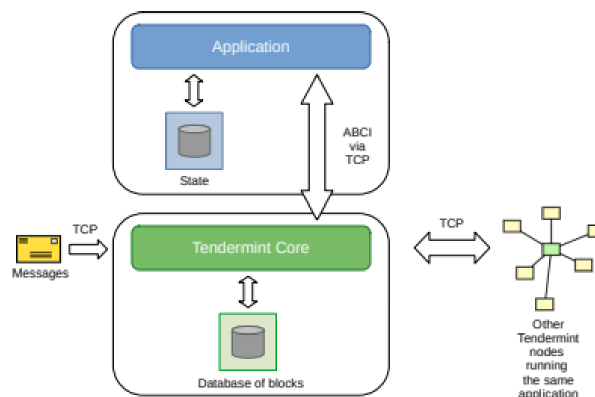


Blockchain Tendermint

La blockchain nel suo complesso è quindi definitiva dalla struttura mostrata di seguito:



Dove possiamo distinguere due tipi archivio, uno relativo alle transizioni (Database of blocks) ed uno relativo alle informazioni necessarie all'applicazione per svolgere le proprie operazioni (State database). Dato che utilizzeremo librerie Python per la gestione di reti neurali, il linguaggio che utilizzeremo per realizzare sia la Dapp che l'ABCI sarà quindi Python. Per fare ciò ci avvarremo della libreria (wrapper) PY-ABCI e di conseguenza realizzeremo una Dapp che comunica con la blockchain via TCP, ovvero che utilizza il servizio RPC. Diversamente utilizzando il linguaggio nativo di Tendermint, go, sarebbe possibile sviluppare l'ABCI utilizzando lo steso processo del Core.



Attraverso il docker-compose definiremo quattro nodi di tipo validator a cui assegneremo una copia della Dapp (ovvero una replica di una ANN) ed una partizione del set di dati di addestramento. Ad ogni ciclo di addestramento compiuta dalla Dapp sulla partizione, ogni nodo ricaverà un nuovo valore di loss ed accurance. Ogni nodo pubblicherà il proprio valore di loss sulla chain aggiornando così la gBest complessivamente ricavata da tutti i nodi. L'aggiornamento verrà eseguito direttamente della chain durante il processo di validazione di un blocco.

Se il valore di loss ricavato da un nodo risulterà minore di quello esistente nella chain, allora il nodo effettuerà una transizione di inserimento del nuovo valore con i relativi pesi del modello. Queste informazioni verranno quindi replicate su ogni nodo dalla piattaforma. Qualora ci fossero due nodi

che eseguono una transizione nello stesso momento/blocco, all'interno del modulo ABCI verrà selezionato quello che avrà il valore di loss minore.

Implementazione dell'ABCI

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, utilizzato per gestire lo scambio dei pesi per una Dapp che implementa l'algoritmo FedPSO. La Dapp dovrà gestire l'addestramento di una rete neurale. La gestione consiste nell'effettuare il training della rete su dati locali (una partizione di un trainingset nel nostro test) e nell'aggiornamento dei pesi utilizzando la funzione di aggiornamento del PSO sulla base del valore di gBest calcolato dalla chain. Ad ogni epoca di training, ogni client, emette una query sulla chain per reperire il valore della gBest. Questo valore verrà aggiornato ad ogni epoca se almeno un client riesce ad ottimizzare, altrimenti si procede con la seconda epoca di training considerando il gBest al passo precedente.

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.

Per realizzare il modulo ABCI utilizzeremo la libreria (wrapper) py-abci raggiungibile sul sito:

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

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

Per il framework Tendermint Core, si può installare la versione master o in alternativa si può scaricare direttamente l'eseguibile ed inserendolo nella cartella usr/local/bin in modo che sia possibile lanciarlo da ogni cartella. Per far funzionare la libreria py-abci correttamente occorre impostare il compilatore python, in quando di default Ubuntu utilizza python v. 3.6.

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

docker build --tag abci -f abci .	crea l'immagine abci a partire dal dockerfile abci
docker image ls	visualizza lista immagini
docker ps -a	visualizza lista container
docker run -it 81a41ea2afec	crea un container passandi l'id immagine
docker exec -it container command /bin/bash	esegue il container e lancia un comando all'avvio
docker rm 77c504449d1fa	rimuove un container

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

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

1. s.o. ubuntu
2. tendermint_0.34.11_linux_amd64 nella cartella /usr/local/bin per essere visibile nella root
3. python3.9 ed python3-pip come compilatore di default
4. libreria abci python
5. inizializziamo un nodo tendermint come validatore ed eseguiamo l'avvio del servizio

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

```
FROM ubuntu

ENV DEBIAN_FRONTEND=noninteractive

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

VOLUME [ /tendermint ]
WORKDIR /tendermint

# p2p, rpc and prometheus port
EXPOSE 26656 26657 26660

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

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

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

Entriamo quindi nel container col comando:

docker start -a -i prova

```
20. 192.168.1.5 (tendermint)    21. 192.168.1.5 (tendermint)    22. 192.168.1.5 (tendermint)
tendermint@deb11x64:~$ docker start -a -i prova
I[2021-12-20|15:29:36.339] Starting multiAppConn service                module=proxy impl=multiAppConn
I[2021-12-20|15:29:36.340] Starting socketClient service                module=abci-client connection=query impl=socketCli
ent
E[2021-12-20|15:29:36.340] abci.socketClient failed to connect to tcp://127.0.0.1:26658.  Retrying after 3s... module=abci
-client connection=query err="dial tcp 127.0.0.1:26658: connect: connection refused"
E[2021-12-20|15:29:39.340] abci.socketClient failed to connect to tcp://127.0.0.1:26658.  Retrying after 3s... module=abci
-client connection=query err="dial tcp 127.0.0.1:26658: connect: connection refused"
I[2021-12-20|15:29:42.344] Starting socketClient service                module=abci-client connection=snapshot impl=socket
Client
I[2021-12-20|15:29:42.346] Starting socketClient service                module=abci-client connection=mempool impl=socketC
lient
I[2021-12-20|15:29:42.348] Starting socketClient service                module=abci-client connection=consensus impl=socke
tClient
I[2021-12-20|15:29:42.350] Starting EventBus service                    module=events impl=EventBus
I[2021-12-20|15:29:42.356] Starting PubSub service                      module=pubsub impl=PubSub
I[2021-12-20|15:29:42.381] Starting IndexerService service              module=txindex impl=IndexerService
I[2021-12-20|15:29:42.383] ABCI Handshake App Info                      module=consensus height=0 hash= software-version=0
.34.11 protocol-version=0
I[2021-12-20|15:29:42.383] ABCI Replay Blocks                           module=consensus appHeight=0 storeHeight=107 state
Height=107
I[2021-12-20|15:29:42.384] Applying block                               module=consensus height=1
I[2021-12-20|15:29:42.386] executed block                               module=consensus height=1 num_valid_txs=0 num_inva
lid_txs=0
I[2021-12-20|15:29:42.387] Applying block                               module=consensus height=2
I[2021-12-20|15:29:42.388] executed block                               module=consensus height=2 num_valid_txs=0 num_inva
lid_txs=0
I[2021-12-20|15:29:42.389] Applying block                               module=consensus height=3
I[2021-12-20|15:29:42.390] executed block                               module=consensus height=3 num_valid_txs=0 num_inva
```

Troveremo già in esecuzione il server Tendermint che mostrerà errori di connessione con l'applicazione ABCI ancora non in esecuzione.

Apriamo una seconda finestra del container e lanciamo l'applicazione ABCI con il comando:

docker exec -it prova counter /bin/bash

```
20. 192.168.1.5 (tendermint)    21. 192.168.1.5 (tendermint)    22. 192.168.1.5 (tendermint)
tendermint@deb11x64:~$ docker exec -it prova counter /bin/bash
INFO     ~ running app - press CTRL-C to stop ~
INFO     ... connection @ 127.0.0.1:43538
INFO     ... connection @ 127.0.0.1:43540
INFO     ... connection @ 127.0.0.1:43542
INFO     ... connection @ 127.0.0.1:43544
```

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

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

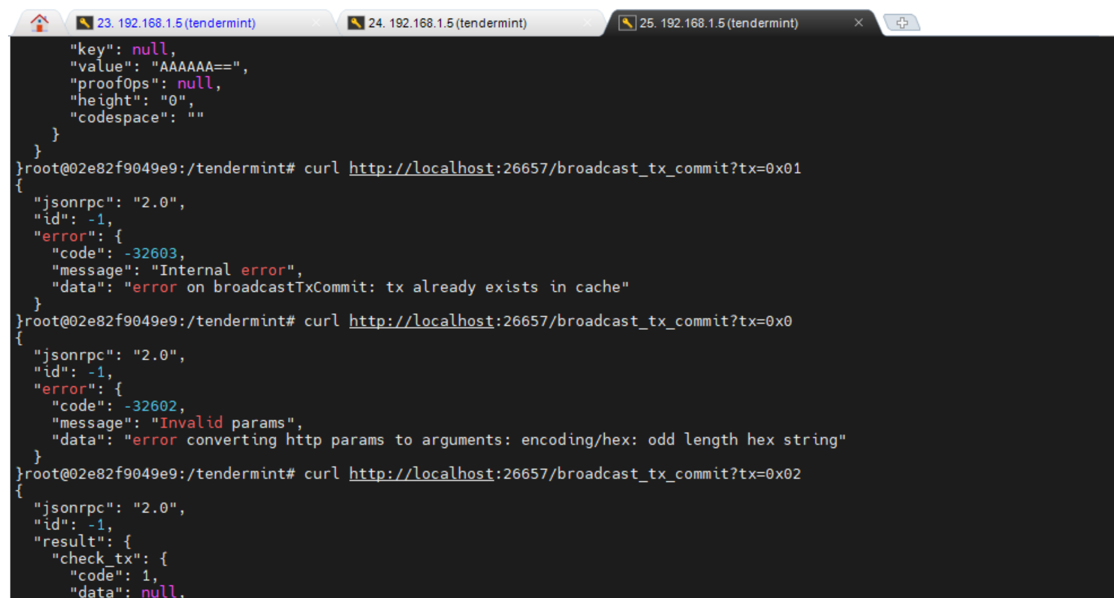
A questo punto possiamo procedere con l'inoltro delle transizioni. Per fare ciò apriamo un'altra istanza Docker:

docker exec -it prova /bin/bash

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

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

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



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

-> 0x01 = OK!

-> 0x03 = Fallirà! (si aspetta 2)

A questo punto siamo sicuri che l'ambiente di test è correttamente funzionante e possiamo procedere con lo sviluppo ed il test del modulo ABCI che convaliderà le nostre transizioni del FedPSO. Per sviluppare la nostra blockchain riportiamo di seguito le quattro connessioni ed i relativi messaggi gestiti dal protocollo ABCI

Mempool connection

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

Consensus connection

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

Info connection

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

Snapshot connection

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

Protocollo ABCI: convalida di una transizione

Quando una richiesta viene inviata ad 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 ai pari.

Il metodo ResponseCheckTx gestisce i messaggi CheckTx, in questo caso dovremmo solo preoccuparci di eseguire il parsing del messaggio 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. Di seguito viene proposta una implementazione in java:

```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. Di seguito una implementazione in java:

```java
@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 locale ma deve eseguire una richiesta a Tendermint Core. In questo caso Tendermint gestirà il messaggio Query attraverso l'ABCI. Ciò è necessario per assicurarci di non avere un valore di gbest obsoleto. Di seguito una implementazione in java:

```
@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 questo paragrafo vedremo nei dettagli la realizzazione di una Dapp (ovvero un'applicazione decentralizzata) per l'esecuzione dell'algoritmo FedPSO parallelo asincrono.

Example Distributed Testnet

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 FedPSO 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 fedpso.py /etc/tendermint/fedpso.py

Per copiare l'applicazione ABCI+FedPSO 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 fedpso.py  /etc/tendermint/fedpso.py

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

Naturalemente, nella direttiva COPY, o diamo il percorso complete del file fedpso.py, 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 && python3 /etc/tendermint/fedpso.py) &

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
```