

Analisi della tecnologia Blockchain e implementazione di Linda sui framework Hyperledger Fabric e Corda

Report di progetto
Sistemi distribuiti

Bosello Michael

1 Introduzione

Lo scopo di questo progetto è quello di introdurre lo studente alla Blockchain Technology e permettergli di affrontare un problema semplice ma ricco di significato per i concetti relativi alla coordinazione di sistemi distribuiti. A questo proposito si desidera implementare uno spazio di tuple distribuito sfruttando due piattaforme blockchain molto promettenti: Hyperledger Fabric e R3 Corda. In questo contesto l'utilizzo della tecnologia blockchain ha l'obiettivo di studiare le potenzialità intrinseche in essa per risolvere problemi tipici dei sistemi distribuiti e portare alla luce criticità rimaste o create. Le primitive definite dal modello di coordinazione Linda sono semplici ma richiedono l'utilizzo di diversi pattern di interazione. Valutare quindi le capacità di interazione delle API fornite dai framework e confrontare queste piattaforme come naturale punto di sintesi delle attività svolte. Di seguito viene definito il percorso affrontato, dopodiché vengono fornite alcune nozioni base su Fabric e Corda che serviranno per affrontare l'analisi che le seguirà, per brevità si evita di fornire una introduzione a BCT e Linda.

Svolgimento Inizialmente ho esplorato la BCT grazie a diverse letture a riguardo e ho studiato le piattaforme in questione. Entrando nel merito dello sviluppo, con Fabric ho prima di tutto sperimentato diverse opzioni relative alla sua architettura di rete per comprenderne al meglio la composizione, quindi testato il sistema con più organizzazioni e nodi. In seguito però ho mantenuto una rete minimale adatta allo sviluppo software. Dopo semplici prove per testare le API fornite e il meccanismo degli eventi, sono passato all'implementazione dello spazio di tuple. Con Corda l'avvio è stato più naturale grazie alla sua struttura più limitata, passando quasi subito alla stesura dello spazio di tuple che diversamente è risultato più arduo. Durante lo sviluppo le capacità e limitazioni di interazione sono emerse e i loro frutti sono sintetizzati in questo report che ha anche l'intenzione di consolidare quanto appreso.

2 Hyperledger Fabric, concetti base

Hyperledger Fabric presenta uno scheletro più articolato rispetto alle altre piattaforme per via degli importanti traguardi che si pone. La sua evoluzione è guidata dalla visione del gruppo che desidera sviluppare una distributed ledger technology opportuna per i casi d'uso business. Le differenze chiave rispetto ad altre BCT sono: modularità, performance e scalabilità, identity service, uso di linguaggi standard per lo sviluppo di Smart Contract (chiamati qui Chaincode), mezzi per ottenere anonimato e confidenzialità. Tra i componenti modulari troviamo il servizio di ordinamento, in questo modo si può scegliere l'algoritmo di consenso più opportuno al caso d'uso regolando il trade-off tra fiducia e prestazioni, per esempio in una rete con elevata fiducia dei peer si può utilizzare un algoritmo meno oneroso che sia solamente crash fault-tolerant, nel caso di una rete dove i partecipanti sono più diffidenti si può adottare un protocollo che sia byzantine fault tolerant. Nell'infrastruttura di Fabric più organizzazioni si aggregano per formare un consorzio che definisce, quando viene creato, le policy e i componenti della rete secondo gli accordi presi dalle suddette organizzazioni. Dopodichè variazioni e aggiornamenti alle policy e alla composizione di rete richiederanno opportune firme secondo quanto definito dalle policy iniziali.

Le organizzazioni della rete sono rappresentate da un insieme di nodi e forniscono un Membership Service Provider a cui far riferimento per autenticazione e identità dei propri nodi. I nodi possono essere Client, Peer e Orderer. L'ordering service è il punto di amministrazione della rete poichè mantiene il Genesis Block in cui sono delineate le configurazioni di rete e le Channel Configuration Transaction che definiscono la configurazione dei canali. I canali sono "subnet" private dove due o più membri possono comunicare in modo confidenziale ed ogni canale ha il proprio ledger privato. In un canale possono essere installati più chaincode ovvero il codice eseguito dai peer per accedere e aggiornare il ledger.

Il flusso delle transazioni inizia da un client (applicazione esterna al network ma comunque autenticata) che invia una Transaction Proposal, una richiesta di invocare una funzione del chaincode per leggere o scrivere sul ledger, agli Endorsing Peer specificati dalle Endorsement Policy. Le endorsement policy impongono l'origine (da quale organizzazione) e il numero di Endorsement necessari per accettare la transazione. Quando un peer riceve una transaction proposal verifica che sia ben formata, che la richiesta non sia già stata presentata in passato (replay-attack protection), che la firma sia autentica e il mittente sia autorizzato ad eseguire l'operazione. Se tutti i controlli hanno successo allora il peer simulerà l'esecuzione del chaincode e risponderà con una Proposal Response, è importante notare che in questa fase il ledger non viene modificato. È necessaria l'approvazione e l'esecuzione solo di una parte della rete¹, questo permette di ottenere scalabilità orizzontale poichè più transazioni possono essere eseguite contemporaneamente da diversi nodi. A questo punto il client colleziona le risposte e verifica di aver soddisfatto le endorsement policy e che i risultati della funzione siano gli stessi per tutti i peer contattati, è qui che viene eliminato il non-determinismo presente nel linguaggio utilizzato. Se si tratta di una query il flusso termina qui. Se si tratta di un aggiornamento le approvazioni dei peer vengono assemblate in una transazione e spedite all'ordering

¹A meno di policy molto restrittive

service. L'ordering service decreta l'ordine delle transazioni all'interno di un blocco senza verificarne il contenuto, un blocco viene formato dopo la ricezione di un certo numero di transazioni o dopo la scadenza di un tempo massimo. I blocchi vengono spediti a tutto il canale ed ogni peer eseguirà nuovamente le verifiche che avrebbe dovuto fare il client e si controlla che non ci siano transazioni in conflitto, le transazioni non idonee vengono marcate come invalide e mantenute allo scopo di audit senza però apportare le proprie modifiche. In conclusione il World State (ledger) viene aggiornato.

Possibilità offerte dalle API Fabric fornisce un SDK per i client e API per i chaincode. Per il progetto sono stati utilizzati Node.js per il client e Go per i chaincode poichè sono stati i primi adottati da Fabric e gli unici disponibili quando questo progetto è iniziato. Le operazioni di nostro interesse offerte dal modulo fabric-client sono:

- Sottoporre una transazione tramite due passaggi, propose e transact.
- Esecuzione di una richiesta a un chaincode per ottenere lo stato dell'applicazione.
- Monitoraggio di eventi quali block event generati quando viene pubblicato un nuovo blocco, transactions event per scoprire se una transazione ha avuto successo o è stata marcata come invalida, custom event generati dai chaincode.

Le API per i chaincode permettono, tra le altre cose, di:

- Rispondere alle chiamate con un valore, esprimendo successo o errore.
- Ottenere dal ledger il valore associato a una chiave o i valori per un range di chiavi.
- Ottenere l'id della transazione.
- Aggiornare o creare un valore per una chiave.
- Impostare un evento custom con un payload.
- Invocare un altro chaincode.

3 Corda, concetti base

Corda è orientato ai servizi finanziari e su questo basa i propri casi d'uso, sebbene per questo possa essere avvertita una sensazione di inadeguatezza quando si realizzano applicazioni per altri scopi, questa piattaforma può essere comunque utilizzata in altri campi. Come per Fabric tra i requisiti cardine ci sono privacy, identità dei partecipanti e abilità di scalare. Tra le prerogative si aggiungono produttività e integrazione aziendale ottenuti tramite l'utilizzo di tecnologie consolidate come JVM e database relazionali.

Un network Corda è formato da un insieme di nodi peer-to-peer autenticati in esecuzione sulla JVM. Tutti i nodi forniscono i servizi di base che vedremo in dettaglio successivamente, essi sono: un network map service, uno o più notary

service, eventualmente zero o più oracle service. La rete è semi-privata e ognuna ha un servizio doorman, un nodo per unirsi alla rete deve fornire al doorman le informazioni richieste ricevendo così un certificato firmato, l'identità viene poi aggiunta alla mappa di rete. In Corda non c'è broadcast, le comunicazioni sono point-to-point e i dati sono condivisi su necessità.

Il ledger in questa piattaforma non è globale, ogni nodo ha la propria visione di un sottoinsieme del ledger sulla base delle transazioni a cui partecipa, viene però garantito che gli stessi fatti condivisi da più nodi siano gli stessi, la sotto-parte del ledger è chiamata Vault. I fatti rappresentati sul ledger sono definiti da State composti da riferimento al contratto, partecipanti e proprietà. Gli stati sono immutabili e finali, per aggiornare uno stato se ne crea uno nuovo che si collega a quello precedente formando una sequenza di stati connessi, quando si aggiorna uno stato si marca quello precedente come historic. I contratti qui sono diversi da quelli delle altre piattaforme, un contratto viene utilizzato solo per verificare la validità contrattuale di una transazione in base alle regole stabilite al suo interno che hanno la possibilità di controllare diverse proprietà, i contratti nonostante possano essere scritti in qualsiasi linguaggio della JVM sono deterministici, essi devono sempre accettare o sempre rifiutare una transazione data, il determinismo è ottenuto grazie alla sandbox che permette di utilizzare solamente le librerie facenti parte di una whitelist, prevenendo l'uso di librerie che possano essere origine di non-determinismo. Corda utilizza il modello UTXO, il ledger si evolve nel tempo applicando transazioni le quali possono consumare zero o più stati e produrne zero o più. Le transazioni sono quindi link tra sequenze di stati. Inizialmente la transazione è solo una proposta con gli output generati dal transaction builder e gli input selezionati tramite riferimento. Una transazione è valida se soddisfa due proprietà: Transaction validity, la transazione ha tutte le firme richieste ed è contrattualmente valida. Transaction uniqueness, la transazione non ha input che sono già stati consumati. Per ottenere consenso non è sufficiente che la transazione sia valida, prima di firmare una transazione un nodo deve controllare la validità della transazione stessa ma anche quella di tutte le transazioni facenti parte della catena delle transazioni che ha portato alla generazione degli input della transazione proposta, il processo è conosciuto come walking the chain. Non è necessario che un nodo riveli tutte le informazioni delle transazioni passate, diversi livelli di confidenza sono ottenuti grazie all'utilizzo di Merkle Tree. La transaction uniqueness è ottenuta per mezzo dei notary service che definiscono il punto in cui una transazione, se valida, diviene finale. Un notary firmerà la transazione solo se i suoi input non sono già stati consumati prevenendo tentativi di double-spending. Opzionalmente un notary può verificare anche la validità della transazione per evitare attacchi di "denial of state", consumo non autorizzato degli stati. L'algoritmo di consenso all'interno di un notary è personalizzabile. I processi che delineano il comportamento di una CorDapp sono espressi per mezzo di Flow. Un flusso può essere chiamato tramite una RPC ed esegue una sequenza di passaggi per raggiungere l'obiettivo prefissato, anche la comunicazione tra nodi è espressa tramite flussi. Ad esempio un flusso che vuole aggiornare uno stato farà i seguenti passi: inizialmente ottiene i dati dal vault, costruisce e firma una transazione, la invia ai nodi che devono firmarla e attende la risposta prima di eseguire il commit. Dall'altro lato un altro flusso responder verifica la transazione, la firma e risponde alla richiesta. In conclusione una CorDapp, cioè una applicazione che viene eseguita sulla rete Corda, è composta da un insieme di stati, contratti e flussi.

Possibilità offerte dalle API Le API di Corda sono molto vaste, di seguito ne vengono elencate solo alcune, tra cui quelle utilizzate per la realizzazione del progetto. Uno stato è rappresentato dalla classe `ContractState`, da sviluppare come POJO con l'aggiunta di un metodo `getParticipants` che deve restituire i nodi (classe `Party`) che partecipano alla transazione. Un `TupleSpaceSchema` permette di mappare un `ContractState` ad una tabella hibernate (framework di persistenza, servizio di Object-relational mapping), necessario per poter effettuare query sugli stati. Un contratto è realizzato dalla classe `Contract` e deve implementare il metodo `verify` che accetta una transazione come parametro. Se durante l'esecuzione il metodo `verify` lancia un'eccezione allora la transazione non è contrattualmente valida, altrimenti lo è. Si può indicare l'intenzione di una transazione aggiungendo un componente `Commands` che è solamente una classe segnaposto, in questo modo un contratto può effettuare verifiche diverse a seconda del comando indicato. I flussi estendono la classe generica `FlowLogic`, quando una chiamata RPC innesca un flusso il suo metodo `call` viene eseguito, con il parametro di tipo si può scegliere cosa deve restituire il flusso alla RPC. All'interno di un flusso si possono costruire e firmare transazioni, richiamare altri flussi (ad esempio il flusso per richiedere le firme degli altri partecipanti e per finalizzare le transazioni), si può interrogare il vault per ottenere gli stati al suo interno e registrare handler da eseguire quando il vault viene aggiornato. Per i client esterni alla rete è possibile invocare flussi tramite RPC, interrogare direttamente il vault di un nodo e sottoscrivere ai suoi aggiornamenti. Per questo progetto il codice di tutti i componenti Corda è scritto in Java.

4 Implementazione di uno spazio di tuple su BCT

L'implementazione è suddivisa in tre parti, lo spazio di tuple che opera sulla rete blockchain, una interfaccia lato client che fornisce funzioni rappresentanti le primitive Linda e infine il client che sfrutta la suddetta interfaccia e interagisce con l'utente.

Fabric Lo spazio di tuple è implementato in un chaincode che gestisce anche la semantica delle primitive (vedremo che per Corda non è così), infatti esso presenta una funzione per ogni primitiva. Dato che non è possibile per un chaincode attendere che il ledger presenti lo stato richiesto, è necessario separare in due parti l'esecuzione delle primitive, si avrà una parte di invocazione e una parte di completamento. Per notificare il completamento di una operazione viene sfruttato il meccanismo degli eventi. Un evento è formato da un nome e un payload JSON, l'evento viene inoltrato per mezzo dello event hub a tutti i nodi del canale che hanno effettuato la sottoscrizione. Nel chaincode vengono utilizzate due strutture. La struttura `Tuple`, i cui campi sono:

Content Il valore della tupla.

Quantity Numero di occorrenze della tupla nel tuple space.

WaitingTransactions Lista di processi in attesa di ricevere la tupla.

La struttura `Event`, i cui campi sono:

Content Il valore della tupla.

Recipients Lista di destinatari dell'evento.

Per indicare in maniera univoca un destinatario si è scelto di utilizzare l'id della transazione che ha invocato la primitiva. L'intenzione iniziale era quella di produrre un evento per ogni processo da risvegliare, dove il nome sarebbe stato il transaction id e il payload avrebbe contenuto la tupla. Purtroppo in Fabric si può creare al più un solo evento per transazione, visto che la out deve raggiungere più processi con una sola transazione (si può fare una sola transazione per chiamata a chaincode), il metodo ideato prevede che i processi siano in attesa su un evento il cui nome è indicato dal tuple template richiesto e che essi possano controllare se la tupla è destinata a loro grazie al campo Recipients, se in Recipients non è presente il suo transaction id allora il processo tornerà in attesa. Per le operazioni in e rd si procede come segue: si cerca la tupla nel ledger, se è presente e la sua quantità è maggiore di zero si diminuisce la quantità nel caso di in, poi in entrambi i casi si crea un evento il cui nome è dato dalla tupla, si aggiunge nel campo Recipients l'id della transazione corrente. Se la tupla non è presente o la sua quantità è zero si aggiunge l'id della transazione corrente alla lista di WaitingTransactions indicando il tipo di operazione. Per inp e rdp si imposta direttamente un evento in tutte le situazioni, se la tupla richiesta è presente con quantità maggiore di zero allora l'evento conterrà la tupla, altrimenti esso conterrà una stringa vuota. Per inp, se la tupla è presente, si esegue anche una transazione per diminuirne la quantità. La primitiva out si occupa di sbloccare i processi in attesa, essa produce un evento con la tupla indicata, inizialmente nel campo Recipients è presente solo l'id della transazione corrente per sbloccare il processo che ha effettuato la out. Se nel ledger non esiste un record per la tupla allora nessun processo è in attesa di quella tupla quindi la si crea e si imposta la quantità a uno. Se invece la tupla è presente si incrementa la sua quantità e si esegue un ciclo che continua finché ci sono processi in attesa o finché la quantità della tupla non scende a zero. Nel ciclo si rimuove il primo transaction id presente nella lista di attesa e lo si aggiunge ai destinatari dell'evento diminuendo la quantità nel caso in cui l'operazione in attesa sia una in. Il client viene eseguito tramite Node.js la cui architettura event-driven implica l'utilizzo di funzioni asincrone, in questo caso ho utilizzato il meccanismo di promise chaining. L'interfaccia lato client esegue lo stesso procedimento per tutte le primitive poiché esse sono gestite lato chaincode. L'interfaccia fornisce due funzioni: init e performOperation, entrambe restituiscono una promise. Init si occupa di ottenere l'utente per poter comunicare con la rete. performOperation prende come parametri la primitiva che si desidera eseguire e il tuple template con i quali forma la transazione da proporre alla rete. Da qui il metodo consiste nell'esecuzione di funzioni asincrone concatenate necessarie per progredire con il protocollo standard di Fabric e quindi eseguire una transazione, si aggiunge alla fine della sequenza un passo extra. Si registra un handler sullo event hub che risponde a tutti gli eventi il cui nome combacia con il tuple template specificato, la promise restituita dal metodo viene completata con il campo Content dell'evento solo quando se ne trova uno il cui payload presenta, nel campo Recipients, l'id della transazione appena effettuata.

Corda Nella soluzione proposta per Corda la semantica delle primitive viene gestita lato client. Nella rete Corda troviamo la definizione di uno State

rappresentante le tuple, sono presenti anche un contratto e due flussi. Il `TupleSpaceState` mantiene il contenuto della tupla, il nodo che l'ha prodotta e la lista dei membri del tuple space. I due flussi offerti sono `PutFlow` e `TakeFlow`. `PutFlow` crea un nuovo stato e lo aggiunge a una transazione che firma e inoltra al notary, non è richiesta la firma di altri partecipanti. Ottenuta l'approvazione diffonde la nuova tupla ai partecipanti dello spazio di tuple. `TakeFlow` invece per prima cosa interroga il vault alla ricerca della tupla richiesta, se non viene trovato nessuno stato contenente la tupla che non sia già stato consumato allora il flusso lancia un'eccezione, Corda gestisce automaticamente le eccezioni incontrate durante i flussi che si rivelano nel fallimento della transazione con conseguente fallimento delle eventuali future in attesa. Se si incontra almeno uno stato idoneo lo si aggiunge come input al `TransactionBuilder` e come da prassi si procede per finalizzare la transazione, in questo caso si richiede la firma di chi ha prodotto la tupla. Il contratto verifica che una operazione `Put` non consumi nessuno stato e produca esattamente uno stato, viceversa impone che una operazione `Take` consumi uno stato e non ne produca nessuno, inoltre controlla che la firma del produttore della tupla sia presente in entrambi i casi. In Corda non è presente un meccanismo di eventi, in compenso le API lato client permettono non solo di invocare flussi ma anche di interrogare direttamente il vault e ottenere un `Observable` per sottoscrivere agli update futuri. L'interfaccia creata per il client presenta una funzione per ogni primitiva poichè si comportano tutte in modo diverso visto che la semantica viene gestita lato client e non dai flussi Corda costruiti. Le invocazioni dei flussi Corda restituiscono una future su cui attendere il risultato, l'attesa della future può eventualmente scatenare un errore se viene incontrato un fallimento durante il flusso, in questo modo il client può gestire gli errori di esecuzione. Le query e gli update sono forniti tramite flussi reattivi (Rx). La out consiste nell'invocazione del flusso `PutFlow` e attesa di successo o rifiuto. Anche la primitiva `inp` può essere costruita solo utilizzando un flusso, in particolare essa invoca `TakeFlow` e restituisce il valore della tupla ottenuto o null in caso il flusso lanci un'eccezione. Per `rdp` si controlla semplicemente se nel vault è presente la tupla. In `e rd` invece, per ottenere la sospensione, utilizzano un semaforo evento. `Rd` sottoscrive tutti gli stati correnti e gli aggiornamenti futuri, quando si incontra la tupla desiderata si esegue la signal. Anche l'operazione `in fa` fa la stessa sottoscrizione poi però entra in un ciclo, all'interno del ciclo si attende sul semaforo, quando viene rilasciato si prova a invocare un `TakeFlow`. Si continua finchè un flusso non ha successo, il ciclo è necessario per affrontare il caso in cui un altro processo in attesa sia più veloce a consumare la tupla.

L'idea iniziale prevedeva un flusso per ogni primitiva dato che i flussi Corda supportano la sospensione. È stato effettuato un tentativo a questo scopo ma per qualche motivo i flussi dopo essere stati sospesi e risvegliati non riuscivano a finalizzare le transazioni. Per questioni di tempo l'indagine delle cause non è stata portata avanti dopo diversi fallimenti e quindi si è arrivati alla soluzione esposta sopra. Si può consultare il tentativo di realizzazione dei flussi bloccanti all'interno del package `'linda.flow.notworking'` del modulo `'cordapp'`.

5 Confronto e valutazione delle potenzialità e dei problemi relativi a coordinazione e consenso delle piattaforme

La prima generazione di blockchain ha permesso per la prima volta a diversi enti di eseguire transazioni direttamente senza terze parti grazie all'utilizzo di un ledger globale condiviso, meccanismi di consenso e immutabilità di ledger e transazioni. La seconda generazione ha portato la possibilità di redigere complesse logiche di business, aggiungendo però il problema del costo della computazione. Di questa seconda generazione fa parte Ethereum. Corda e Fabric fanno parte di una terza generazione di blockchain, anche se in realtà sono ledger distribuiti, dove si tentano di risolvere diversi problemi. Queste piattaforme aumentano decisamente lo throughput e la capacità di scalare. Esse risolvono anche il problema dell'aspetto economico della computazione che funzionava da incentivo per un comportamento legittimo e limitazione per gli attacchi DOS. Queste piattaforme però devono necessariamente reintrodurre un punto di fiducia, ovvero centralizzare nuovamente l'amministrazione del sistema. Qui si deve porre fiducia nell'organizzazione che amministra la rete e l'applicazione distribuita, eventualmente la rete potrà essere liberamente accessibile ma solo i nodi che rappresentano l'organizzazione o consorzio avranno l'autorizzazione di apportare modifiche al codice. Dal punto di vista inverso, si può dire che Fabric e Corda mancano di un *cryptoeconomic layer*.

Consenso Si raggiunge consenso quando l'ordine e i risultati delle transazioni sono decretati in modo univoco e finale. In Corda il consenso è stabilito dai notary per l'ordine e dai contract per la validità, non esistendo il concetto di blocco è sufficiente decretare l'ordine delle transazioni per non avere conflitti. In Fabric il consenso viene raggiunto collezionando i risultati di diversi peer, l'ordering service decreta poi l'ordine delle transazioni all'interno dei blocchi da esso creati al cui interno possono presentarsi dei conflitti. Il database sottostante utilizza il meccanismo MVCC (Multiversion concurrency control) per assicurarsi che non si creino inconsistenze nei dati, nel caso di un write-write conflict viene generato un MVCC error e le transazioni in conflitto successive alla prima vengono annullate.

Potenzialità Fabric supporta la semantica push con il modello publish subscribe, grazie allo event hub c'è la possibilità di risvegliare più flussi di controllo con un solo evento. Purtroppo c'è una limitazione per cui si può produrre un solo evento custom per transazione.

Corda supporta la concorrenza e la sospensione dei flussi, esso è progettato per eseguire più flussi in parallelo. Quando un flusso entra in uno stato bloccante² viene serializzato su disco in modo che il nodo possa continuare con gli altri flussi presenti nello scheduling per poi riprendere il flusso successivamente. In Corda, al contrario di quanto fatto fino adesso, possiamo spostare l'attesa dal client al mezzo.

²e.g. attesa I/O, chiamata di una funzione di networking, lock. Non funziona con costrutti di basso livello come semaphore.

È previsto per il futuro l'aggiunta della capacità di migrazione dei flussi tra le varie JVM. È apprezzabile anche l'attenzione di Corda alla gestione del tempo. Non ci sarà mai una sincronizzazione esatta tra un party che deve eseguire una transazione e un notary, non solo per i problemi di fisica e di latenza di rete ma anche perchè una transazione richiede diversi step che possono necessitare di più o meno tempo, nei sistemi distribuiti non si avrà mai un "tempo reale" ma solo una approssimazione di esso, impossibile dunque pensare di imporre vincoli che richiedano l'esecuzione in un esatto istante. Date queste premesse la soluzione adottata è il meccanismo delle Time-windows, un notary agisce anche come timestamping authority, si può imporre che una transazione sia valida solo se eseguita prima di un dato punto nel tempo, dopo un dato punto nel tempo o nel mezzo tra due diversi punti nel tempo.

Problemi In Fabric si deve tener conto delle minacce alla disponibilità indipendenti dalla propria organizzazione. Con le transaction policy ci si può proteggere da concorrenti non totalmente fidati, si deve tenere conto però che se un'origine viene a mancare allora nessuna transazione può avvenire. Si ipotizzi di avere un consorzio con molte organizzazioni che non ripongono fiducia l'una nelle altre, per questo impongono che ogni transazione debba ricevere la firma di almeno un nodo per organizzazione. Se anche solo un'organizzazione non fosse raggiungibile per via di una partizione di rete o per via di un guasto, allora nessuna transazione potrebbe aver luogo.

In Corda le comunicazioni sono point to point, quando si effettua una transazione si deve esplicitamente indicare tutte le parti che si vogliono rendere partecipi, inoltre si dovrà attendere la risposta di tutte le parti indicate anche se non si necessita della loro firma. Perdiamo dunque sia space uncoupling che time uncoupling, caratteristiche tipiche in Linda.

6 Espressività di Linda e Smart Contract

Il termine potere espressivo può essere usato per indicare due significati. Da un lato si trova l'espressività teorica delineata come la capacità espressiva indipendente dalla semplicità di utilizzo, dall'altro lato si ha l'espressività pratica definita come la capacità di esprimere concetti in maniera concisa e leggibile. Per il primo caso, l'espressività teorica, abbiamo che sia Smart Contract (Ethereum), Chaincode (Fabric), Flow e Contract (Corda) sia Linda³, come dimostrato in Busi, Gorrieri, and Zavattaro [1], sono Turing complete.

Per quanto riguarda l'espressività pratica possiamo utilizzare il problema dei dining philosophers, classico problema utilizzato per comparare vari formalismi riguardo la loro capacità di esprimere e risolvere problemi concorrenti, per ottenere un'idea di quanto Smart Contract e Linda siano espressivi riguardo problemi di coordinazione. Con Linda non è possibile ottenere una soluzione immediata, è necessario adottare soluzioni tradizionali come ad esempio imporre un ordinamento totale delle risorse o utilizzare una soluzione basata su ticket, anche sfruttando il costruito eval o processi daemon alcune accortezze per evitare corse critiche dovrebbero essere adottate. Il carico della coordinazione è

³Solo se la primitiva out presenta una semantica 'Ordered', ovvero l'emissione della tupla (invio al Tuple Space) e il suo rendering (presenza effettiva nel Tuple Space) sono un'azione atomica.

nelle entità coordinate. Utilizzando gli Smart Contract invece la soluzione è banale, è sufficiente prendere e rilasciare le due forchette. Questo è possibile perché la blockchain ci assicura che solo la prima transazione che consuma una risorsa sia valida garantendo la mutua esclusione, ci assicura che o entrambe le forchette o nessuna delle due vengano prese/rilasciate garantendo che non possa avvenire deadlock e, essendo il rilascio delle forchette contemporaneo, che ci sia fairness. La differenza risiede nella atomicità delle operazioni, in una blockchain ogni transazione è atomica rispetto alle altre⁴. Si può concludere che, mentre in Linda il carico della coordinazione ricade sui coordinabili, negli Smart Contract la complessità viene scaricata sulla struttura sottostante.

7 Conclusioni

Tutte le piattaforme analizzate, come normale che sia, presentano vantaggi e svantaggi. Dal punto di vista della coordinazione Corda presenta feature notevoli come sospensione dei flussi e gestione di time windows ma sembra anche che sia la più penalizzata quando si necessita di comunicazione indiretta e disaccoppiamento. Queste piattaforme forniscono una struttura per la coordinazione dei sistemi distribuiti già abbastanza matura per i casi d'uso di reti permissioned, per le piattaforme permissionless attualmente il peso dell'aspetto economico della computazione può essere problematico.

⁴In Fabric i Chaincode e in Corda i Flow possono essere eseguiti concorrentemente ma i conflitti sono risolti dopo aver raggiunto il consenso e prima che qualsiasi modifica sia finalizzata.

A Fabric network & Script

È già stata esposta la struttura logica di una rete Fabric, questa appendice dà un riferimento al contenuto del progetto dal punto di vista di un network operator. All'interno del progetto ci sono tre cartelle: `basic-network`, `chaincode`, `client`. `Chaincode` contiene solamente i chaincode da installare sui nodi. `Client` racchiude i file Nodejs per interagire con la rete e lo script che viene utilizzato per eseguire il deploy della rete. In `basic-network` troviamo diversi script dei quali viene utilizzato solo quello per avviare la rete, per completezza sono inclusi anche altri script di utilità. Sempre in `basic-network` abbiamo:

config Cartella che contiene il blocco di genesi e il transaction channel.

crypto-config Cartella che archivia i certificati di tutte le entità della nostra rete.

configtx.yaml Definisce la configurazione delle entità da aggiungere alla rete

crypto-config.yaml Definisce la configurazione dei certificati delle entità

docker-compose.yml Definisce le impostazioni delle immagini docker di cui eseguire il deploy.

Lo script `startFabric.sh` da me realizzato permette di eseguire tutti i passaggi necessari per il deploy della rete. Per i dettagli si può far riferimento ai commenti all'interno, in ogni caso i passi effettuati sono:

1. Se necessario, stop dei container attivi e delle reti docker, rimozione del key-store utilizzato a runtime.
2. Download delle dipendenze Nodejs del client.
3. Startup di quattro container: una Certificate Authority, un Orderer, un Peer, un database CouchDB.
4. Creazione del canale e aggiunta del peer tramite comando docker.
5. Avvio del CLI container, la CLI fornita da Fabric permette di interagire con gli altri nodi ed eseguire comandi di configurazione di rete.
6. Installazione e istanziazione del chaincode indicato come parametro
7. Se viene passato un secondo parametro, invocazione della funzione di inizializzazione indicata.
8. Enroll dell'Admin e registrazione di due utenti da utilizzare liberamente nel client.

Riferimenti bibliografici

- [1] Busi, Gorrieri, and Zavattaro. On the turing equivalence of linda coordination primitives. 1997. URL <http://www.informatica.unibo.it/it/ricerca/technical-report/1997/pdfs/97-05.ps.gz>.