JOLLAR

LABORATORIO SISTEMI OPERATIVI A.A.2017/2018

TEAM SO

INDIRIZZO DI RIFERIMENTO: andrea.chiellino@studio.unibo.it

COMPONENTI:

Emanuela Calvanese 765662
Andrea Chiellino 771788
Marco Viceconte 794250

Struttura del report

Il report inizia con una descrizione delle features e prosegue con le istruzioni per eseguire una demo; in seguito è descritto il funzionamento generale del sistema , e per finire abbiamo descritto l'implementazione dei Servizi .

Elenco features

Il progetto Jollar consiste nella realizzazione di una sistema di pagamento elettronico .E' stato scritto in Jolie , in particolare sono stati scritti servizi che interagiscono tra loro al fine di permettere la creazione/trasferimento di valuta digitale tra diversi nodi connessi alla rete. I terminali connessi alla rete utilizzano come sistema di pagamento la criptovaluta Jollar, una moneta digitale che sfrutta una particolare Proof-of-Work basata sui numeri primi.

Moneta digitale basata su Proof-of-work

Una critica alle criptovalute attualmente in circolazione è rivolta allo spreco di corrente elettrica prodotto dal cosiddetto "Mining" . Per garantire la sicurezza del sistema , le criptovalute che si basano su Proof-of-Work "costringono" gli utilizzatori della rete a mettere a disposizione la propria potenza computazionale al fine di validare transazioni altrui e permettere il regolare fluire dei pagamenti. L'incentivo per l'utente che decide di partecipare a questo "lavoro" , è un premio in moneta elettronica.

Proof-of-work sostenibile

Per risolvere questo problema, nella criptomoneta Jollar la Proof-of-Work smette "solo" sforzo computazionale diventando lavoro per la creazione di "prove" che abbiano duplice obiettivo: da una parte testimoniare che uno sforzo è stato fatto per validare le transazioni della rete. Dall'altro generare un valore utilizzabile in altri "settori" scientifici. Nel caso specifico , seguendo la soluzione adottata da PrimeCoin, la prova è una catena di numeri primi detta catena di Cunningham. La generazione di queste catene è utile in altri campi scientifici.

Rete p2p decentralizzata

La rete composta tra i nodi che avviano il software Jollar sulle loro macchine è una rete p2p , in quanto ogni nodo svolge funzioni sia di client e sia di server, inviando richieste quando ha bisogno e servendo invece le richieste altrui quando viene contattato da un client che necessita di un'elaborazione. Tutte le richieste vengono allora gestite dagli stessi nodi dalla rete, non essendo presente alcuna unità centralizzata.

Blockchain, un archivio decentralizzato

Alla decentralizzazione dei compiti , è legata la decentralizzazione dei dati: anche questi infatti, sono obbligatoriamente conservati dai nodi stessi. I nodi devono quindi occuparsi della corretta replicazione dei dati e del mantenimento dello stato di aggiornamento degli stessi.
I dati da conservare saranno quindi relativi agli utenti, alle loro transazioni, alla loro ricchezza attuale e altro.
Queste informazioni sono conservate in un archivio detto blockchain, ed è compito del sistema da noi realizzato la gestione della corretta replicazione di questi dati

Sincronizzazione automatica tra i nodi

Se un nodo "perde traccia" dell'ultima versione di questi dati condivisi, ad esempio per un ritardo di rete , deve preoccuparsi di ripristinare il proprio stato aggiornando la copia locale della blockchain all'ultima versione collettivamente accettata. Il sistema è realizzato in modo da permettere questa re-sincronizzazione , i dettagli si trovano nella parte finale di questa documentazione "Implementazione delle features".

Wallet con sistema chiave pubblica - chiave privata

Per poter ricevere e spendere i Jollar , è stato ricreato il meccanismo crittografico alla base delle monete digitali cioè la firma digitale con chiavi asimetriche. Per la generazione delle chiavi è stato utilizzato l'algoritmo Digital Signature Algorithm (DSA) creato per gestire situazioni di firma digitale. Il sistema a chiave pubblica e chiave privata è utilizzato per identificare l'utente. In questo modo è possibile ricevere pagamenti comunicando la propria chiave pubblica e pagare utilizzando la propria chiave privata.

ISTRUZIONI PER AVVIARE UNA DEMO

Assicurarsi di avere Jolie installato sul sistema e di aver scaricato tutta la directory contenente il progetto. Inserire i file.jar presenti nella cartella JavaServices del progetto, nella directory JavaServices della propria installazione di Jolie.
La demo simula l'interazione tra 3 utenti.
Nella directory devono essere presenti oltre ai tre lanciatori e il proxy, anche i file per la demo cioè le tre coppie di chiavi e le tre copie della blockchain.
Utente1privk,Utente1publick,blockchainUtente1
Utente2privk,Utente2publick,blockchainUtente2
Utente3privk,Utente3publick,blockchainUtente3

I file **lanciatore1.ol**, **lanciatore2.ol**, **lanciatore3.ol** servono a simulare i **3** utenti del sistema.

Avviare il Network Visualizer contenuto nel file NetworkVisualizer.ol inserendo in un prompt il comando:

jolie NetworkVisualizer.ol

Avviare il **Proxy Broadcast** contenuto nel file **myproxy.ol** aprendo un prompt ed inserendo il comando**:**

jolie myproxy.ol

Aprire 3 prompt diversi e inserire i seguenti comandi

Nel primo prompt per avviare il primo nodo:

jolie lanciatore1.ol

Nel secondo prompt per avviare il secondo nodo:

jolie lanciatore2.ol

Nel terzo prompt per avviare il terzo nodo:

jolie lanciatore3.ol

Attendere alcuni istanti , quando il servizio è stato caricato mostrerà un messaggio di benvenuto.
Dove è stato avviato ogni lanciatore, scegliere un nome utente diverso tramite
il comando **-setUser**. Tre nomi utenti (e le rispettive chiavi e copie della blockchain) sono gia' presenti nella cartella per semplificare l'esecuzione demo.Per assicurarsi che il sistema sia avviato e sia pronto a ricevere i comandi da tastiera, è possibile digitare il comando **-h** per ricevere una stampa a video dei comandi che si possono utilizzare.
I **comandi** e i **nomi** utente sono **case-sensitive**. (inoltre i comandi iniziano con un trattino)

Per impostare i nomi utenti inserire quindi nel primo terminale

-setUser

premere invio**.** Inserire: **Utente1** premere invio.

Nel secondo terminale:

-setUser

premere invio**.** Inserire**: Utente2.** premere invio**.**

Nel terzo terminale:

-setUser

premere invio**.** Inserire**: Utente3** premere invio**.**

In uno dei prompt avviati inserisci il comando:

-inviaJollar

premi invio**.**
Verranno mostrati gli output contenenti **jollar** spendibili, quindi scegliere un numero rappresentante la transazione che vuoi utilizzare/spendere
(relativo ad una delle transazioni che sono state mostrate a video**)** e premi invio**.**
Inserisci un numero da **1 a 3**, per utilizzare una chiave pubblica gia presente nella "Rubrica" che viene mostrata a video e premi invio.
La transazione verrà inviata alla rete.
Adesso gli altri nodi hanno salvato la transazione in una coda di transazioni locale.
Per far si che la transazione sia confermata è necessario che venga scritta in un blocco . Allora spostarsi in uno qualunque degli altri prompt e inserire il comando

-startMining

e alla fine del processo di **mining**, il nuovo blocco conterrà anche la transazione che volevamo inserire.
E' possibile verificare con -getMyUtxo, l'effettivo ricevimento dei Jollar da parte dell'utente destinatario ed il fatto che non siano piu' disponibili nelle Utxo del mittente.
Per notificare il networkVisualizer dello stato di tutti i lanciatori, lanciare da ognuno di loro il comando

-status

Spostarsi sul network visualizer per visualizzare lo stato ricevuto da ogni nodo.

Nota:Per avere un immediato riscontro di quali utenti sono presenti all'interno del sistema, è possibile, prima di usare -**setUser**, usare il comando -**list** che mostra a schermo gli username utilizzabili.

Un **alternativa** di esecuzione della demo prevede la creazione di 3 nuovi utenti con il comando -create .Inserire un nome a scelta, verranno create le chiavi pubblica e privata. Settare i tre utenti nei tre diversi lanciatori con il comando -setUser. Verrà creato un file vuoto per contenere la blockchain. Avviare myproxy.ol. Minare qualche blocco con -s da uno dei lanciatori. Gli altri lanciatori riceveranno il blocco in automatico. In un altro lanciatore procurarsi la chiave pubblica di un utente diverso da quello che ha minato i blocchi. Copiare la (lunga) stringa ed utilizzarla , nel terminale dell'utente che aveva minato i blocchi(e ricevuto i premi) , nel comando -inviaJollar come indirizzo di pagamento (incollare la lunga stringa nel momento in cui richiede un indirizzo di pagamento o un numero della rubrica) . In questo modo, al prossimo blocco minato ed aggiunto, la transazione sarà validata ed i Jollar si saranno spostati da un utente all'altro.
(se i lanciatori non sono sincronizzati si puo' usare -sync su tutti i lanciatori)

Anzichè avviare **terminale**.**ol** , avviamo invece i **3 "Lanciatori.ol"** perchè contengono gli indirizzi delle **location** (**socket**) della input port già impostati al loro interno.
Questi indirizzi sono socket://localhost alle porte 8001, 8056 e 8059
I file **lanciatore1.ol** , **lanciatore2.ol lanciatore3.ol.** contengono un include a **terminale.ol** , in questo modo avviando i vari lanciatori possiamo utilizzare terminale.ol sulla stessa macchina ma sulle diverse porte elencate prima .
Un alternativa era utilizzare, da linea di comando il flag -c seguito dal nome della costante=valore (nel nostro caso mylocat="socket://localhost:porta")

Descrizione generale progetto

Il sistema deve occuparsi delle richieste da parte dell'utente , della comunicazione sulla **rete p2p** ,di verificare e controllare le transazioni ,blocchi, catene di numeri. In fase progettuale abbiamo raggruppato queste diverse funzionalità in **Servizi**, secondo principi di coerenza dei compiti svolti ed cercando di in genere di aumentare la coesione . In questo modo ogni **servizio** si occuperà di un insieme limitato di compiti.
I servizi che saranno in funzione sulla singola macchina (utente della rete) sono:
Terminale.ol(È il vero e proprio nodo della rete),**Blockchain.ol** (Servizio che si occupa di leggere , scrivere e svolgere operazioni che utilizzano la blockchain,**Miner.ol** (Si occupa della ricerca di catene di numeri primi),**myproxy.ol** (Semplice servizio che realizza il messaggio in Broadcast da un nodo a tutti gli altri**) NetworkVisualizer.ol** (Servizio di controllo. Puo' interrogare i nodi per ottenere info dal sistema).**TimeStamp.ol** (Fornisce la data e l'ora ai nodi della rete).Inoltre utilizziamo due servizi scritti in Java ed embeddati e cioè **catene.jar** (Cerca e verifica catene di numeri primi) **wallet**.**jar** (Genera coppie di chiavi pubblica-privata)
L'utente è identificato da un username e possiede una coppia di chiavi, una pubblica ed una privata. Utilizzando il "Miner" trova catene di primi e le utilizza per costruire blocchi da agganciare alla rete. Il terminale.ol prende i comandi da tastiera e rimane in ascolto di eventuali transazioni in arrivo sulla rete. Se una o piu' transazioni vengono ricevute, vengono incluse nel blocco successivo.
Per realizzare il sistema di transazioni abbiamo scelto di utilizzare il modello UTXO anzichè quello balance-based. In questo modo teniamo traccia degli Output di Transazione Non spesi , che rappresentano le monete spendibili da un utente.
Il dettaglio del funzionamento della transazione è illustrato nella descrizione del comando **-inviaJollar** nella spiegazione del pannello comandi di terminale.ol, in questa documentazione.

Una volta trovata una catena, viene creato un blocco che la contiene come prova, e il blocco viene propagato sulla rete .Ogni nodo della rete in maniera indipendente effettua le verifiche sul blocco e sulla catena , se tutto rispetta le regole il blocco viene aggiunto alla blockchain .

Un esempio di funzionamento :

Operazione di mining (ricerca di numeri primi + costruzione blocco) .

l'utente inserisce il comando -startMining sul Terminale,

Terminale chiede la blockchain a Blockchain (RR *readBlockchain()()*) e la invia in input a Miner con l'operation One-Way *minaBlocco@Miner(global.blockchain)*

Miner cerca la catena utilizzando le operation specifiche delle catene di cunningham del servizio Java embeddato Catene.

Una volta trovata una catena con giusta difficoltà , Miner la inserisce in una varibile albero rappresentante il blocco , apponendo i figli/campi .origine, .difficolta, .altezza , .data e la manda al **Terminale** (Embedder) usando la one-way *pushBlock@MyTerm()* che prende in input un Blocco "quasi pronto".

Terminale.ol riceve il blocco e nella input-choice relativa aggiunge i campi mancanti prima di inoltrare il blocco in broadcast alla rete.

I campi già aggiunti dal Miner l'origine della catena, la difficulty e all'altezza del blocco. Adesso il terminale.ol aggiunge eventuali transazioni ricevute sulla rete, prima di inoltrarle il blocco in Broadcast sulla rete.

Se nessuna transazione era in attesa di essere inserita in un blocco, viene aggiunta solo la transazione 0, definita come transazione di Reward, che rilascia 6 jollar (cioè ha .output.amount=6) all'utente che sta minando (letta con la R-R readPublicKey tramite l'output port @Blockchain)

La transazione di premio contiene come hash della transazione precedente una stringa di default, siccome l'input non è un "effettivo spendere di monete" ma una generazione di monete nuove.In questo modo nel momento del controllo transazioni possiamo saltare il controllo del campo .previousTxId quando questo presenta la stringa "predefinita". (in terminale.ol input-choice [sendBlock])

Dettagli implementativi pannello comandi (Terminale.ol)

Il pannello comandi utente è realizzato in terminale.ol all'interno della define comandiUtente.

Una println@Console()() mostra a video i seguenti comandi digitabili da tastiera:

-create Serve a creare la coppia di chiavi pubblica-privata , identificabile come un portafoglio Utente.

Implementazione:

Utilizziamo l'operation println del servizio Console, l'istruzione in() per l'input da tastiera, e l'operation CreateWall del servizio CreateWallet scritto in Java ed embeddato staticamente in Terminale

```
println@Console("Inserisci il nome utente.Sara' il nome dei file che conterranno le chiavi ")();
in(nomefile);
CreateWall@CreateWallet(nomefile)(res);
```

l'operation: **CreateWall(string)(string)** prende in input il nome utente (username) e genera una coppia di chiavi DSA con istanza SHA1 e le salva nella directory corrente in due file che hanno come nome la concatenazione tra "username"+pubkey e "username"+privkey. Se il nome è gia' utilizzato, concatena al nome del file "2", lo salva e restituisce l'username modificato all'utente che quindi viene informato del conflitto di nomi sui file,e del nome utente da utilizzare da ora in poi.

-setUser

Utilizzo: Dopo aver creato la coppia di chiavi serve "settare" il nome utente per poter creare/importare la blockchain e prepararci al mining.

Implementazione:

L'utente inserisce il suo username , che viene "catturato" dall'istruzione in(inputUtente);

aggiorniamo la variabile global.username , notifichiamo il servizio @Blockchain del settaggio/cambio di username e poi richiediamo al servizio @Blockchain la copia della blockchain salvata su file relativa al “giusto” utente .
-list il comando -list mostra su schermo i nomi validi di "utenti" da poter utilizzare sul proprio terminale
Implementazione: Un utente del sistema ha bisogno di una chiave pubblica ed una chiave privata . Cercando le coppie di chiavi nella directory corrente , otteniamo i "portafogli utente" già creati ed importabili .
Tramite la operation *getServiceDirectory@File(void)(dir)* del servizio **File** della standard library di jolie , otteniamo la directory di lavoro corrente.
Utilizziamo la risposta appena ottenuta per invocare , sempre verso il servizio File, la operation list@File()() .
L'espressione regolare nel figlio .regex della richiesta è stata impostata per ottenere, in risposta, solo i nomi dei file che terminano con la sottostringa "privk".
Dalla risposta, con un replaceAll@StringUtil(string)(string), del servizio StringUtil , eliminiamo le sottostringhe “privk” in modo da ritrovarci con la lista di username pronta per essere stampata.
Codice:

```
getServiceDirectory@File( void )( dir );
list@File({.directory=(dir),.regex=".*privk"})(res);//dynamic lookup
valueToPrettyString@StringUtils(res)(res);
with( res ){ .regex="privk";  .replacement="" };
replaceAll@StringUtils(res)(res);
with( res){ .regex=": java.lang.String"; .replacement="" } ;
replaceAll@StringUtils(res)(res);
println@Console( "Sono state trovate le seguenti chiavi private:" )();
println@Console(res)();
```

-startMining oppure -s oppure -mining : Avvia il Mining
Utilizzo: Serve ad avviare il "Mining" cioè la ricerca di catene di numeri primi , per ottenere ricompense in Jollar.E' necessario aver impostato l'utente con -setUser
Implementazione:
Un primo controllo su global.username lanciato un errore con il comando throw(UsernameError) per evitare che l'esecuzione prosegua, se l’username non è definito.
L'install che gestisce questo errore informa l'utente della necessità di usare prima -setUser

```
scope( userScope )
    { install( UsernameError => println@Console( "ERRORE:Usare prima il comando -SetUser" )() );
      if( !is_defined( global.username ) ) throw( UsernameError )
    }
```

Quando global.myusername è correttamente impostato, chiamiamo l’operation MinaBlocco(BlockchainType) offerta dal servizio Miner.ol .
La necessita di passare come parametro di input la blockchain, deriva dal fatto che il Miner ha bisogno di informazioni sull'attuale stato della blockchain per poter generare una catena di primi valida e con una difficoltà sufficiente.
Essendo un operation che puo' richiedere molto tempo, abbiamo realizzato una One-Way anzichè una Request-Response. Quando Miner troverà una catena notificherà il terminale con una One-Way "di ritorno" che viene gestita dalla Input-Choice pushBlock@(BlockType) nel main di terminale.ol .
-inviaJollar: Invia transazione
(1) **-InviaJollar** (Invio Transazione)
In sintesi:

1) Cerca tra le Utxo usando l'operation getMyUtxo@Blockchain()() di Blockchain
2) Costruisce la richiesta identificando l'input da spendere e l'output da generare
3) Invia in broadcast la transazione usando l'operation inoltraTx@ProxyBroadcast() di Proxy

L'utente inserisce il comando -inviaJollar ,per trasferire i suoi Jollar ad un altro proprietario.
La transazione è formata da un input e da un output. L'input corrisponde al "bene" che l'utente A vuole "spendere", per poterlo trasferire ad un nuovo utente B . L'output sarà quindi il nuovo "bene" questa volta appartenente all'utente B e quindi spendibile dall'utente B.Ogni output, per essere a sua volta speso, diventa Input delle altre transazioni.
Per la rappresentazione in codice del concetto di transazione ci viene in favore la struttura ad albero delle variabili del linguaggio Jolie: Per memorizzare l'hash univoco con cui identifichiamo ogni transazione,usiamo la radice dell'albero. Due figli .input e .output identificano rispettivamente la vecchia "moneta" che viene spesa e la nuova "moneta che viene generata dalla transazione.
Ecco un esempio della struttura ad albero di una variabile Jolie contenente una ipotetica transazione:

```
.transaction[0] = 3a36f166a033f67e8aecfdb1f41bee24 : java.lang.String
        .output[0]
            .amount[0] = 6 : java.lang.Integer
            .payto[0] = PublicKeyExample34dd39id3 : java.lang.String
        .input[0]
            .amount[0] = 6 : java.lang.Integer
            .previousUtxoTxid[0] = 00000000000000000000 : java.lang.String
```

L'input si riferisce alla moneta che abbiamo deciso di trasferire. Se vogliamo spendere una moneta vol dire che quella moneta esiste ed è quindi presente sull'archivio distribuito ,la blockchain.Per trovarsi nella blockchain : o 1) E' stata ricevuta come premio per aver trovato una catena di numeri primi oppure 2) Qualcuno ci ha "pagato" trasferendo la proprietà di una sua vecchia moneta.La creazione del premio è realizzata da una speciale transazione che genera un output verso noi stessi , in questo modo sia i reward sia le monete che abbiamo ricevuto da altri utenti , hanno in output un pagamento verso la nostra chiave pubblica.
Questo è realizzato utilizzando un sottofiglio di **.output** denominato **.payto**, che verrà impostato , nel caso della transazione di premio, con la stringa rappresentante la chiave pubblica dell'utente che ha minato il blocco.
Nel caso invece della transazione tra un utente ed un altro, il campo **.output.payto** conterrà la chiave pubblica dell'utente destinatario della transazione.
Sara' quindi tra le varie transazioni che possiedono in .output.payto il nostro indirizzo di pagamento ,che dobbiamo cercare le monete che useremo questa volta in Input per generare una nuova transazione.Le operazioni come controllare se una transazione esiste, se è gia stata spesa, se il suo hash è corretto e quelle per trovare le transazioni non spese vengono realizzate dal servizio Blockchain e descritte nella parte di descrizione di Blockchain.ol.
-**getBlockchain** : Stampa la blockchain a video
Implementazione:Utilizziamo una chiamata alla Request-Response readBlockchain(username)(Blockchain) del servizio Blockchain raggiungibile tramite la output port @Blockchain. Questa operation prende in input una stringa relativa all'username e restituisce , dopo averla letta da file, la blockchain corrispondente. Il risultato è stampato a video con una println@Console()() dopo essere stato trasformato da albero in stringa "stampabile" grazie a valueToPrettyString(string)(string) offerto dal servizio StringUtil.
-**getMyUtxo** : Mostra i tuoi jollar spendibili
Implementazione:
Ottiene la lista degli hash delle transazioni spendibili, cioè quelle che hanno in output.payto la pro-

pria chiave Pubblica.Realizzato utilizzando la operation getUtxo(address)(UtxoRes) offerta da Blockchain.L'operation prende come input un indirizzo di chiave pubblica e ritorna una lista di hash rappresentanti le UTXO spendibili.
-sync: Controlla se la tua blockchain è sincronizzata con la rete
Utilizzo: Effettua la sincronizzazione manuale delle versioni della blockchain nel caso fossero non sincronizzate.
Implementazione: Abbiamo realizzato l'operazione di sync in maniera asincrona. L'utente inserisce -sync da terminale e viene allertato da un messaggio che avvisa che l'operazione sovrascriverà la blockchain attualmente presente, nel caso venisse trovata una blockchain piu' aggiornata. L'utente puo' confermare o rinunciare ad inviare la richiesta di -sync.
Tramite il proxy inviamo una oneWay getNextIndex() che verrà "girata" a tutti i nodi connessi alla rete.La richiesta mandata in input contiene la location del "richiedente". Quando i nodi della rete (terminale.ol) riceverranno la chiamata alla operation getNextIndex, creano un messaggio contenente la propria copia della blockchain e lo inviano tramite inoltraIndex()@Proxy al richiedente iniziale. In questo modo, la persona che voleva sincronizzarsi , riceverà da tutti i nodi la blockchain. Basterà a questo punto sovrascrivere la blockchain locale se risulta piu' corta rispetto a quella appena ricevuta.
L'operazione di -sync è necessaria solo se i nodi hanno blockchain con lunghezze diverse.
Se tutti i nodi sono invece sincronizzati ad un altezza blocco, ciascuno di loro puo iniziare a minare il blocco successivo avendo la garanzia che gli altri nodi agganceranno il blocco in maniera automatica, senza bisogno di sync manuali.

Le input choice che il terminale.ol presenta per ricevere le richieste dall'esterno sono:
[sendBlock(Block)] per ricevere un blocco dalla rete
Quando viene ricevuto un blocco dalla rete , in questa operation sono implementate la verifica della catena di primi e difficolta (operation di Blockchain), controllato l'hash di blocco, utilizzando la funzione md5 di MessageDigest .Inoltre viene effettuato il confonto tra il previousHash ,e l'hash del blocco precedente . Per finire viene controllata la correttezza degli hash della transazione di premio e di eventuali transazioni aggiuntive.
Dopo di che il blocco viene inoltrato a @Blockchain con l'operation addValidBlock per effettuare l'aggiunta del blocco alla blockchain memorizzata su file.
[sendTransaction(Transaction)] per ricevere una transazione dalla rete
Controlli degli hash come fatto per il blocco, e chiamata alla operation

```
checkIfUnspent@Blockchain(utxoreq)(response);
```

per verificare che la transazione ricevuta non sia già stata spesa.
Se il controllo è positivo avviene inserimento della transazione in una coda di transazioni. Questa coda di transazioni verrà agganciata al prossimo blocco minato dal terminale che mantiene le transazioni in coda.
[pushMeIndex()] la richiesta di sincronizzazione da parte degli altri nodi

Descrizione Servizio MINER
miner.ol è il servizio che si occupa della Proof of Work e cioè di ricerca di catene di numeri primi. viene embeddato da terminale.ol che puo' quindi utilizzare l'operation minaBlocco().Il suo compito principale è quindi controllare all'interno di un range di numeri, quali di questi siano origini valide per una catena di Cunningham. Questa operazione di ricerca di numeri primi in catena , è definita come "Mining". Ogni catena di numeri che verrà utilizzata come prova di lavoro , dovrà essere "nuova" rispetto alle catene di numeri già trovate. Questo impedisce di utilizzare catene già utilizzate e obbliga quindi ad effettuare ogni volta un nuovo lavoro .
Operation :

OneWay: minaBlocco(BlockchainType) è stata realizzata una operation OneWay minaBlocco che riceve in input (da terminale.ol) una blockchain e inizia a cercare una catena di numeri primi che abbia la giusta lunghezza, la giusta difficoltà e che non sia mai stata usata . Il servizio terminale.ol riceve da tastiera , dall'utente, il comando -startMining, il servizio Terminale effettua una request-response verso il servizio embeddato Blockchain raggiungibile tramite l'output port @Blockchain richiedendo l'ultima copia della blockchain come salvata su file. Una volta aggiornato lo stato della blockchain effettua la OneWay minaBlocco()@Miner passando come parametro la blockchain.
Esempio:
da terminale.ol parte **minaBlocco@Miner(req)** dove in req.message è stata salvata la blockchain con l'operatore deep copy req.message<<global.Blockchain.
Il servizio Miner riceve un messaggio che risveglia l'input choice [minaBlocco()] in "ascolto" nel main.
L'operation oneway minaBlocco( ) riceve il seguente stato della blockchain :
ESEMPIO

```
blockchain[0]                                   (sono mostrati solo alcuni campi)
        .block[0]
                .origin="29"
                .difficulty="2.489429"
        . block[1]
                .origin="53"
                .difficulty="2.673934"
```

Nella blockchain sono gia presenti due blocchi, quindi sono state già trovate due catene di numeri primi. La prima con origine 29 e difficulty 2.48 e la seconda con origine 53 e con difficulty 2.67 . Essendo la difficulty composta da : lunghezza della catena nella parte intera + difficulty decimale compresa tra 0 e 0.99999, allora sono entrambe catene di 2 numeri primi perchè la parte intera della difficoltà è 2.
La ricerca di catene di numeri primi impone una difficoltà crescente di blocco in blocco. Cio' significa che il Miner , che si appresta a cercare una catena valida per il blocco con indice 2, dovrà tenere traccia dell'ultima difficulty presente e cioè quella del blocco[1] ,in questo caso 2.67. La catena che Miner accetterà, dovrà avere sia lunghezza due, sia una difficulty decimale maggiore rispetto 0.67 . Inoltre, dovrà essere diversa da 53 e diversa da 29, siccome queste due sono già state utilizzate.
Questi sono i controlli effettuati in Miner.ol nell'operation minaBlocco. Per le operazioni di calcolo della lunghezza della catena (indicata dalla parte intera della difficoltà), controllo di primalità con il test di Fermat, e calcolo della parte decimale della difficulty, Miner sfrutta delle operation opportunamente create in un servizio embeddato Catene , che presenta tre operation scritte in Java:
boolean checkPrime(Integer) ---> controlla se un numero è primo
Per effettuare il controllo abbiamo implementato il test di primalità di Fermat come richiesto dalle specifiche del progetto .
Integer checkChain(Integer) ---> controlla se un numero è origine di una catena di cunningham di primo tipo e, se sì , ne restituisce la lunghezza .
double difficolta(Integer) ----> dato un numero primo calcola la parte decimale della difficulty e restituisce il corrispondente double compreso tra 0.0 e 1.0

Il calcolo della difficoltà decimale, cosi come i calcoli contenuti nel test di primalità di Fermat, utilizzavano numeri molto grandi che non potevano essere contenuti in una variabile Integer. E' stato quindi usato BigInteger. Questo è il motivo che ci ha spinti a realizzare queste operazioni in un servizio scritto in Java.

Le operazioni numeriche di ricerca delle catene di numeri primi e controllo primalità , sono state implementate in Java ed embeddate in miner.ol .
Questa scelta ci è servita per utilizzare il tipo di dato BigInteger presente in Java. Le funzioni sono:
checkChain(int)(int) che prende in input il numero su cui controllare l'esistenza della catene e ritorna 0 se non è un numero primo, 1 se è un numero primo seguito da un non primo, 2 se a partire dall'origine esiste una catena di cunningam di lunghezza 2, e così via.
getDifficulty(int)(double) effettua un calcolo sulla catena identificata dall'origine passata come input, e ritorna il corrispondente double che indica la difficoltà della catena.
checkPrime(int)(boolean) restituisce true se il numero passato come input è primo, false altrimenti.

Blockchain.ol

Blockchain è il servizio che ci permette di comunicare con la blockchain salvata su file, effettuare operazioni di lettura su di essa , come la ricerca di transazioni spendibili, la lettura di hash di blocchi precedenti o il controllo delle difficoltà.
Ogni volta che un blocco ricevuto dalla rete viene validato, Blockchain riceve una OneWay addValidBlock( ) dal proprio terminale, ed aggiorna sia la propria variabile global.blockchain sia la copia su file della blockchain.
La copia su file della blockchain è memorizzata in json tramite l'operation writeFile@File del servizio File integrato in Jolie.
Questo assicura che tutte le volte che Blockchain riceve una richiesta, puo' utilizzare la versione della blockchain piu' aggiornata. Le operation del servizio Blockchain, sono sempre in ascolto grazie all'esecuzione concorrente impostata con {execution:concurrent}, essendo state scritte come input-choice nel main. Le operation (la cui definizione è nell interfaccia Blockchain.iol) sono:
OneWay: **setUser(string)** : L'username è usato nei nomi dei file relativi all'utente. E' questo il motivo per cui abbiamo scritto una operation che imposta la variabile global.myusername.
addValidBlock(undefined) : Riceve un blocco da terminale.ol (quindi già validato e pronto da agganciare al file json sul file system).
dopo aver agganciato il blocco con il comando deep copy solo nelle variabili global condivise tra le istanze di Blockchain.ol, effettuiamo il salvataggio su file. Per questo scopo utilizziamo la define saveBlockchain , che fa utilizzo della operation writeFile del servizio File di jolie.
Ecco il codice della input-choice

```
[addValidBlock(req)]{
    global.blockchain.block[#global.blockchain.block]<<req;
    saveBlockchain
  }
```

RequestResponse:
readPublicKey(string)(string):
Prende in input un nome utente e restituisce la chiave pubblica in formato base64.
readPrivateKey(string)(string):
Prende in input un nome utente e restituisce la chiave privata in formato base64
readBlockChain(string)(undefined);
prende in input un nome utente e restituisce la blockchain . E' realizzata leggendo prima la blockchain da file (il filename è costruito agganciando il nome utente alla parola "blockchain").
existOrigin(int)(bool):
E' utilizzata per controllare se una origine è gia stata utilizzata e quindi presente nella blockchain. E' realizzata con un ciclo che itera su tutti i blocchi, e confronta il campo .origine del blocco con il parametro ricevuto come input. Se sono uguali, allora l'origine data come input è gia stata utilizzata nella blockchain e quindi la risposta sara impostata su false;

```
existOrigin(originRequest)(res){
```

```
  res=true;
      v=0;
      while( res==true && v<#global.blockchain.block ) {
        if(global.blockchain.block[v].origin==originRequest){
          res=false
        };
        v++}
```

getDifficulty(int)(double)

Serve ad ottenere la difficulty double , cioè la parte decimale della difficulty, di un determinato blocco. Come input riceve l'altezza del blocco e come output risponde il double relativo alla difficoltà.

```
[getDifficulty(indexRequest)(res){
    if( is_defined( global.blockchain ) ) {
      res=global.blockchain.block[indexRequest].difficulty
      }
  }]
```

L'operazione per il calcolo del decimale è illustrata con un esempio :
La catena di primi che ha come origine il numero 2 ha una lunghezza 5.I numeri primi "quasi raddoppiati" che la compongono sono 2, 5, 11, 23, 47.Per calcolare la parte double della difficulty prendiamo il doppio dell'ultimo numero della catena e sommiamo 1 ottenendo così 95. La formula per ottenere questo numero di partenza è la stessa usata per le catene di cunningham, in modo abbiamo ottenuto il numero che , se fosse primo, sarebbe il 6 numero della catena.
La primalità di questo numero non ci interessa, lo utilizziamo per trovare il remainder del test di fermat cioè 2^94 modulo 95. Questa operazione ha come resto 54, quindi la difficulty della catena sarà (95-54) / 95 =0.43 .Una catena di lunghezza 5 con una difficulty decimale pari a 0.43 otterrebbe come difficoltà registrata nel blocco 5.43. Visto che il risultato delcimale della difficulty si avvicina ad uno senza mai raggiungerlo, è settata una soglia pari a 0.996 (piu' precisamente il risultato di 255/256). Quando la difficulty supera questa soglia, la lunghezza della catena è modificata ad un intero in piu' rispetto alla lunghezza attuale e la parte double è azzerata.In questo modo una catena di lunghezza 5 e difficulty 0.997 sarebbe memorizzata nel blocco con difficulty 6.0 , per far si che il minatori , dal prossimo blocco , inizino a cercare catene di lunghezza superiore (6 anzichè 5)
.

getUtxo(publicKey)(Utxo)

Utilizziamo due for con indice b da 0 a #blockchain.block e indice t da 0 a #blockchain.block[b].trasaction. In questo modo controlliamo tutti i blocchie per ognuno di essi tutte le transazioni. Confrontiamo la publicKey in input con tutti i campi block[b].transactions[t].output.payto. Ogni volta che questo confronto restituisce vero, vol dire che è presente un output che ci “paga”.
Se questo output generato non è mai stato speso, allora possiamo restituirlo nella lista di Utxo.
Salviamo tutti gl output che abbiano passato il primo controllo di esistenza, e con un altro for partente dal blocco in cui l’output è nato, rimuoviamo quelli che successivamente sono stati spesi.
Per controllare se sono stati spesi, guardiamo se l’hash della transazione contenente l’output che ci “paga” è stato utilizzato , in input , come previousUtxoId. Se non esiste un input che faccia riferimento alla transazione che vogliamo controllare, vol dire che non è mai stata spesa e possiamo aggiungerla alla variabile da restituire. Alla fine del procedimento abbiamo tutte le Utxo non spese e possiamo ritornarle al servizio chiamante.

getBlockHash(int)(string),

Restituisce l’hash del blocco[indice] con l’indice di blocco passato in input

trovaAltezza(string)(int),

Prende in input un hash di un blocco e restituisce l’altezza del blocco

getLastBlockIndex(void)(int),

Restituisce l'altezza dell'ultimo blocco presente

checkIfUnspent(undefined)(bool)

Prende in input un output di transazione e controlla che non sia mai stata spesa.
Meccanismo di funzionamento uguale a quello di getUtxo

myproxy.ol Servizio di broadcast

Il proxy è un servizio che effettua riceve richieste e le inoltra in **Broadcast** a tutti gli altri nodi collegati alla rete . Se consideriamo che il numero degli utenti non è noto a priori , non possiamo definire staticamente una serie di output port pari al numero di client da contattare. Sarà quindi compito di questo nuovo servizio tenere aggiornata la lista dei nodi connessi, e inoltrare a tutti loro le richieste fatte dal singolo nodo.
(Nota: quando il **proxy broadcast** invia la transazione/blocco agli altri nodi, la invia anche a noi stessi. in questo modo, anche "noi", inseriamo la transazione/blocco in una coda in modo da aggiungerla nel caso volessimo provare a minare il prossimo blocco)

NetworkVisualizer.ol

Il **Network Visualizer** è un tool amministrativo da terminale per il monitoraggio del sistema, che riceve messaggi dai nodi della rete contenenti lo stato della **blockchain** di tutti i **nodi**, la loro **data e ora** , le **transazioni** che contengono. L'esecuzione è impostata con execution:sequential per mostrare i log in modo sequenziale. Possiede una operation in input-choice [print(content)] che stampa il contenuto seguendo le specifiche del progetto .

```
[print(content)]{
        println@Console( "La data attuale del nodo e' : "+ content.content.data )();
        println@Console( "L'username e':"+content.content.user )();
        println@Console( "La chiave pubblica e': "+content.content.pubkey )();
        println@Console( "Le sue transazioni spendibili sono:" )();
        for ( i=0, i<#content.content.utxo.result, i++ ) {
                println@Console( "transazione n:"+i+" | Hash: "+content.content.utxo.result[i] )() }; }
```

Timestamp.ol

E' un servizio che restituisce il timestamp unix e la data e l'ora.

possiede due operation che utilizzano il servizio Time della standard library di Jolie e "girano" la risposta al nodo che ne ha fatto richiesta

```
[getTime(void)(res){
        getCurrentTimeMillis@Time()( res )
}]
[getDate(void)(res){
        getCurrentDateTime@Time()(res)
}]
```