JOLLAR
LABORATORIO SISTEMI OPERATIVI A.A.2017/2018

#DEADLOCK
INDIRIZZO DI RIFERIMENTO: federica.vespi@studio.unibo.it
COMPONENTI:
Franceschini Giada, 758288
Gibertini Virginia, 758368
Passerini Serena, 765715
Polini Elena, 767654
Vespi Federica, 767820

Jollar implementa un sistema di scambio elettronico decentralizzato, dove gli utenti effettuano transazioni certificate all'interno della rete, senza il bisogno di un organo centrale garante; ciò si realizza tramite la costruzione di una rete Peer-To-Peer (P2P) che utilizza un algoritmo che produce valori che vengono utilizzati per verificare che sia stata eseguita una notevole "quantità di lavoro", chiamato Proof-of-Work (POW).

Si obbligano quindi i nodi che vogliono scrivere un blocco a cercare un valore che sia difficile da trovare e di cui sia facile controllarne la correttezza. Le transazioni certificate vengono registrate in un archivio pubblico, detto **Blockchain**.

L'architettura di Jollar è composta da una serie di **Nodi**, attivi in un determinato momento che compongono la rete P2P, il server **Timestamp**, che offre un servizio che permette di conoscere l'ora esatta all'interno della rete e che garantisce l'esistenza dei dati al momento della richiesta, un servizio **Broadcast** e il **Network Visualiser** , che consente di monitorare lo stato dei nodi, visualizzando le loro Blockchain e le transazioni che hanno effettuato.

Struttura del report:

Il report inizia con una descrizione generale del progetto e prosegue con le istruzioni per eseguire una demo; in seguito alla quale sono descritte e discusse le scelte implementative riguardanti Blockchain.ol, Broadcast.ol, Nodo.ol, nw.ol, ProofOfWork.ol, Timestamp.ol e MainInterface.iol

Istruzioni per la demo:

1. Avviare server Timestamp.ol
2. Avviare server NetworkV.ol
3. Avviare server Broadcast.ol
4. Avviare file Nodo.ol: da terminale, dopo essersi spostati nel path corretto del file inserire il comando

```
jolie -C "location=\"socket://localhost:8001\"" Nodo.ol
```

5. Avviare file Nodo.ol: da terminale, dopo essersi spostati nel path corretto del file inserire il comando

   jolie -C "location=\"socket://localhost:8002\"" Nodo.ol

6. Avviare file Nodo.ol: da terminale, dopo essersi spostati nel path corretto del file inserire il comando

   jolie -C "location=\"socket://localhost:8003\"" Nodo.ol

7. Avviare file Nodo.ol: da terminale, dopo essersi spostati nel path corretto del file inserire il comando

   jolie -C "location=\"socket://localhost:8004\"" Nodo.ol

8. Nel terminale del nodo 8001, inserire in input da tastiera la stringa:

   8002/1

   E premere invio, per inviare 1 Jollar dal primo al secondo nodo

9. Nel terminale del nodo 8001, inserire in input da tastiera la stringa:

   8003/2

   E premere invio, per inviare 2 Jollar dal primo al terzo nodo

10. Nel terminale del nodo 8001, inserire in input da tastiera la stringa:

    8004/2

    E premere invio, per inviare 4 Jollar dal primo al quarto nodo

<u>Le transazioni e la catena di blocchi</u>:

La **Blockchain** è una struttura dati ordinata gestita dal file Blockchain.ol, formata da una serie di blocchi contenenti una transazione. Ogni nodo ha a disposizione la propria Blockchain salvata in modo persistente su file fileBlockchain(global.Node.publicKey).txt. Ogni blocco è un contenitore, una struttura dati a sua volta, che aggrega le informazioni riguardanti una unica transazione.
Le informazioni che vengono mandate dal nodo interessato al file Blockchain.ol tramite l'operation updateBlCh sono:
- hashPrevious
- requestBlCh.block.difficulty
- requestBlCh.block.transaction.nodeSeller.publicKey
- requestBlCh.block.transaction.nodeBuyer.publicKey
- requestBlCh.block.transaction.jollar
- requestBlCh.block.transaction.timestamp
- requestBlCh.block.timestamp

Se il file fileBlockchain(global.Node.publicKey).txt è gia presente nella cartella con path dichiarato [exists@File(filename)(esiste)], il blocco e le informazioni sopra elencate relative alla transazione vengono aggiunte in coda al file.
Nel caso in cui il file non esista, si tratta della prima transazione presa in carico dal nodo e di conseguenza la Blockchain crea il suddetto file inserendo in testa le informazioni relative al GenesisBlock, che è appunto il primo blocco creato nel sistema; il quale viene trascritto, insieme alla prima transazione effettuata, sulla Blockchain di ogni blocco.
L'assegnazione dei 6 jollar da parte del GenesisBlock al primo nodo che si registra sulla rete, è invece di competenza dell'operation sonoNato()() implementata nel file Broadcast.ol.
sonoNato()() effettua un controllo simile a quello sopraindicato, ovvero se il file delle location nodesLocations.txt non è ancora presente nella cartella, il servizio crea il file e assegna tramite la variabile res.jollarGenesis i jollar al nodo che per primo ha effettuato la RequestResponse.

L'altra operation implementata nel file Blockchain.ol è contaBlocchiBlCh, la quale riceve in input fileDaLeggere da un Nodo, legge il contenuto del file con nome global.path + fileDaLeggere e invia come output il numero di blocchi presenti nella propria Blockchain. Ciò è stato realizzato con la funzione di split messa a disposizione dalla interfaccia string_utils.iol, che divide le informazioni blocco per blocco, utilizzando come separatore la stringa "//", e restituisce il numero di blocchi presenti nel file, assegnandolo alla variabile nBlocchi.

Consideriamo la blockchain con il numero maggiore di blocchi come la versione di blockchain più aggiornata all'interno della rete.
Ogni nodo, dopo aver aggiunto o meno alla Blockchain le informazioni del blocco a seconda del risultato della ProofofWork, invia al Broadcast una OneWay

```
nodoDaControllare.location =location;
checkBlChCorretta@Nodo_Broadcast( nodoDaControllare )
```

per richiedere la versione della blockchain di tutti gli altri nodi della rete.

Il Broadcast legge il file nodesLocations.txt, invia a tutti i nodi presenti sulla rete (tranne al Nodo da cui ha ricevuto la OneWay) la richiesta di inviargli la loro location e il numero di blocchi.

```
richiestaBlockchainNodi()(resBlockchainNodi)
resBlockchainNodi.location = location;
resBlockchainNodi.numeroBlocchi = nBlocchi
```

Una volta che il Broadcast ha ottenuto tutte le informazioni necessarie dai nodi ancora attivi le invia al nodo iniziale tramite inviaNumBlocchi(nodoOk).

A questo punto il nodo effettua il controllo nodoOk.numeroBlocchi > global.numeroBlocchiNodo.
Se la risposta risulta essere affermativa, viene stampata a video la stringa "blockchain da aggiornare" e viene copiata la Blockchain del nodo con versione più aggiornata

```
reqSplit = nodoOk.location;
reqSplit.regex = ":";
split@StringUtils(reqSplit)(resSplit);

copyRequest.from = global.path + "/fileBlockchain" + resSplit.result[2] + ".txt";
copyRequest.to = global.path + "/fileBlockchain" + global.Node.publicKey + ".txt";
copyDir@File(copyRequest)(copyResponse);
println@Console( "contenuto copiato" )()
```

Il Server Timestamp

Il **Timestamp** è un servizio, contenuto nel file Timestamp.ol, che riceve richieste per conoscere il tempo unix.

Le varie componenti del sistema si rivolgono al server Timestamp mediante delle RequestResponse che restituiscono il tempo in millisecondi o in formato data-ora.

Le operation implementate nel Timestamp sono due:

- TimeRequestResponse()(): che tramite la funzione getCurrentMillis()() dell'interfaccia Time.iol restituisce il tempo in millisecondi al richiedente.
- DateRequestResponse()(): che converte i millisecondi in formato data-ora, tramite la funzione getDateTime()() dell'interfaccia Time.iol.

Entrambe le operation vengono richiamate all'interno dei file mediante di due define: RichiestaTime e RichiestaData. Si è scelta questa soluzione perché le richieste al Timestamp vengono effettuate molteplici volte all'interno del main e quindi si evitano duplicazione di codice.

La Proof-of-Work

L'algoritmo Proof-of-Work è stato implementato come servizio embedded dentro il file ProofOfWork.ol. Tale servizio rimane sempre in ascolto, in attesa di ricevere dal nodo interessato i valori p[0] e n , tramite la RequestResponse givePoW@Pow(requestPow)(responsePow).

Il file ProofOfWork.ol è composto da:

- i define CunPrimoTipo, CunSecondoTipo e BiTwin, che implementano rispettivamente le Catene di Cunningham del Primo Tipo, le Catene di Cunningham del Secondo Tipo e le Catene Bi-twin
- i define TestFermat, che implementa il piccolo Teorema di Fermat e CheckLunghDiff, che controlla che la lunghezza della catena sia maggiore o uguale alla difficolta, i quali servono a validare le catene di numeri primi
- il define PoW, il quale implementa l'algoritmo vero e proprio, ovvero inizia con la generazione random di una delle tre catene implementate nei define precedenti, verifica la loro validità tramite il richiamo di TestFermat, calcola la difficolta (d = k + (pk-r)/pk), controlla la lunghezza della catena richiamando CheckLunghDiff, e conclude controllando se effettivamente sia il teorema di Fermat che CheckLunghDiff diano esito positivo.

Tale controllo finale determina la validità o meno della Proof Of Work. L'esito viene salvato nella variabile responsePow.validita ed inviato al nodo che ha richiesto la "prova di lavoro".

In caso di esito positivo, il nodo che ha richiesto la Proof of work inserisce la transazione in un blocco, che procederà a trascrivere all'interno del proprio file contenente la blockchain.

La Rete Peer-To-Peer

Avvio dei nodi

La rete è costituita da nodi (utenti) che si scambiano transazioni e informazioni mediante l'intervento di un servizio di broadcast.

I nodi vengono generati tutti a partire dal file Nodo.ol, la distinzione avviene assegnando diverse location relative alla inputport, in modo dinamico. Perché questo sia possibile, la inputPort del nodo ConnNodo avrà come Location una variabile, chiamata location:

```
inputPort ConnNodo {
        Location: location //da assegnare
        Protocol: sodep
        Interfaces: BroadcastInterface
}
```

Ad essa verrà assegnato un valore corrispondente alla stringa indicante la socket ad ogni avvio del file. L'avvio deve essere eseguito da terminale specificando prima del nome del file, il valore da assegnare alla costante location:

```
jolie -C "location=\"socket://localhost:X\"" Nodo.ol
```

dove al posto di X l'utente può scegliere il numero di porta a cui collegarsi.
Questa scelta implementativa permette di non avere duplicazione di codice (quindi minore probabilità di commettere errori) e come conseguenza maggiore scalabilità (potenzialmente potremmo far operare più di 4 nodi).

Registrazione delle location

Subito dopo l'avvio del nodo viene invocata l'operation:

```
sonoNato@Nodo_Broadcast(location)(response);
```

essa invia al servizio Broadcast.ol la location del nodo assegnata da terminale, affinché essa venga registrata in un file chiamato nodesLocations.txt.

All'avvio del servizio Broadcast.ol, viene eseguito un controllo all'interno del path di riferimento riguardo l'esistenza o meno del file contenente le location: nel caso esso non sia stato cancellato in seguito ad una precedente esecuzione, si procede alla sua eliminazione. Per far sì che questo controllo avvenga ad ogni avvio del servizio di broadcast, esso è stato inserito nell'init del file:

Per la registrazione effettiva delle location, a lato broadcast verrà eseguita una scrittura su file mediante le operazioni messe a disposizione dall'interfaccia File.iol. Il codice sottostante riporta l'operazione di scrittura del file contemplando due casi possibili. Mediante il comando

```
exists@File( global.filename )( esiste );
```

si verificherà se il file esiste o meno (con riferimento all'esecuzione corrente del servizio). Il caso in cui il file non esiste coincide con la nascita del primo nodo e di conseguenza, con la prima chiamata della operation sonoNato()(); in questo caso viene effettuata una sola operazione di scrittura. Al contrario, nel caso in cui il file esista già verranno eseguite due operazioni, una di lettura e una di scrittura, per permetterne l'aggiornamento.

Questa operazione permetterà al broadcast di risalire alle location dei nodi registrati ogni qual volta ci sia necessità di comunicare con i nodi stessi. La outputPort ConnNodo nel file Broadcast.ol sarà definita nel seguente modo:

```
outputPort ConnNodo {
        Protocol: sodep
        Interfaces: BroadcastInterface, BlockchainInterface
}
```

Senza specificare la Location, che verrà assegnata dinamicamente prima di chiamare qualsiasi operation diretta ai nodi, mediante lettura del file nodesLocation.txt. La lettura scorrerà con un ciclo for le location presenti nel file assegnandole alla variabile ConnNodo.location. Il conteggio del numero totale di location registrate (necessario per scorrere il file) viene eseguito grazie alla funzione split@StringUtils()() fornita dall'interfaccia String_utils.iol, specificando come separatore il carattere "-" e in seguito contando il numero di stringhe separate.

Il codice sottostante riporta l'esempio dell'invio di una transazione dal broadcast a tutti i nodi registrati:

```
scope( file ) {
        install( FileNotFound => println@Console( "File non trovato" )() );
        read.filename = global.path + "/nodesLocations.txt";
        readFile@File( read )( display );
        display.regex = "-";
        split@StringUtils(display)(riga);
        n = #riga.result;
        for (i=0, i<n, i++) {
                scope( i ) {
                        install( IOException => println@Console( "Nodo con location " +
                        riga.result[i] + "                down.")() );
                        println@Console( "indice: " + i + " / contenuto riga: " + riga.result[i] )();
                        ConnNodo.location = riga.result[i];
                        println@Console( "invio a location: " + ConnNodo.location )();
                        broadcastTrans@ConnNodo(transactionBr)
                }
         }
    };
```

Inizializzazione dei nodi

Dopo aver ricevuto la conferma di registrazione della location dal broadcast, all'interno dell'init del file Nodo.ol. si procede con l'assegnazione dei figli del tipo Node.

Chiave pubblica [?] per l'assegnazione della chiave pubblica è stato scelto di prelevare il numero di location associata al nodo, per facilitare la distinzione fra i nodi. Per fare ciò viene eseguito uno split della stringa indicante la location, usando come separatore il carattere "-" e prelevando il terzo risultato (indice 2) della funzione split@StringUtils()().

Chiave privata [?] per l'assegnazione della chiave privata viene utilizzata la funzione di Hash MD5 fornita dall'interfaccia message_digest.iol. Come digest della funzione è stato scelto di utilizzare una stringa generata con il comando new, in modo tale da evitare la possibilità che vengano utilizzati due digest uguali con seguente generazione di chiavi uguali.

Blockchain locale [?] all'avvio del nodo viene eseguito il controllo all'interno del path di riferimento riguardo l'esistenza del file contente la blockchain locale del nodo. Poiché il nome del file viene costruito a partire dalla chiave pubblica del nodo, tale controllo si rende necessario per evitare di sovrascrivere un file nel caso in cui in sessioni diverse venga assegnata la stessa porta, quindi la stessa chiave pubblica.

Per la gestione e scrittura del file contenente la blockchain si rimanda alla sezione della documentazione intitolata Blockchain.

Transazioni

Una volta che un nodo è avviato, registrato e gli sono state assegnate le chiavi pubblica e privata, è pronto ad inviare/ricevere transazioni. Il servizio resta in ascolto finché l'utente specifica da tastiera la chiave pubblica del nodo destinatario e la quantità di jollar da spedire, separati dal carattere "/".

Una volta ricevuto l'input da tastiera si procede con l'assegnazione dei figli del tipo Transaction. Chiave pubblica del destinatario e numero di jollar vengono assegnati mediante split della stringa ricevuta in input.

Prima di inviare la transazione al broadcast vengono fatti due controlli:

- Nodo destinatario e mittente non devono coincidere.
- Il conto dei jollar del nodo mittente deve essere maggiore o uguale alla quantità da inviare con la transazione.

L'operation invioTrans@Nodo_Broadcast(Transaction) invia la transazione da effettuare (con relativi dati) al broadcast, il quale si occuperà di inviarla a tutti i nodi della rete.

Il servizio di broadcast controlla che il nodo destinatario della transazione sia effettivamente presente nella rete: per fare ciò prova a creare una connessione con la location del nodo destinatario, costruita a partire dalla chiave pubblica di quest'ultimo. L'operation utilizzata per la connessione è una semplice OneWay senza parametri di input, chiamata checkNodoAttivo@ConnNodo().

Il controllo si serve di una variabile, chiamata down, inizialmente settata a false. Nel caso in cui la connessione venga rifiutata, tale variabile viene settata a true, la transazione viene annullata e nessun dato viene inviato ai nodi.

```
global.down = false;
scope( nodoAttivo ) {
    install( IOException => {println@Console( "Nodo con location socket://localhost:" +
    transactionBr.nodeBuyer.publicKey  + " down.")() | global.down = true } );
      ConnNodo.location = "socket://localhost:" + transactionBr.nodeBuyer.publicKey;
      checkNodoAttivo@ConnNodo()
};
 transactionBr.nodoDown = global.down;
if (global.down == true) {
      println@Console( "No transazione" )()
}
else { //codice invio transazione }
```

Superato il controllo, l'invio della transazione ai nodi avviene come specificato sopra, con la modalità di estrazione delle location da file e assegnazione dinamica di ConnNodo.location.

Una volta che un nodo riceve la transazione dal broadcast, controlla se la propria chiave pubblica coincide con quella di mittente/destinatario, procedendo eventualmente con la sottrazione/ addizione di jollar al proprio conto.

In seguito ogni nodo procede con il tentativo di racchiuderla in un blocco della sua blockchain, tramite proof of work e sua validazione (vedere sezioni relative a Blockchain e Proof Of Work).

Network Visualiser

Il **Network Visualiser** è un servizio in ascolto pensato per richiedere informazioni sullo stato della rete ogni quanto di tempo utilizzando un timeout che richiede lo stato della rete ogni volta che scatta.
Nell'init sono inizializzate tre variabili globali: global.path, global.maxNumBlocchi e global.maxLocation.
Global.path contiene il percorso in cui trovare i file che contengono le blockchain.
Al momento dell'avvio il nodo richiede al Network di entrare in funzione tramite la RequestResponse possoPartire()().

Una variabile globale global.nNodi viene utilizzata per tenere conto del numero di nodi attivi nella rete, salvando la loro chiave pubblica all'interno di un array denominato global.arrayTransazioni[], così da non ricevere più volte il permesso di poter partire dallo stesso nodo.

In seguito viene invocato il define sendRequestWithTimer che avvia il timer settando la durata, il nome, il messaggio.

Terminata la fase di attivazione, il Network Visualizer risulta essere operativo.
Come prima cosa riceve in entrata una OneWay dal server Time, in seguito inoltra al broadcast, tramite una OneWay denominata richiestaInfoBroadcast, una richiesta riguardante il numero di blocchi presenti nelle blockchain locali di ogni nodo.

Invoca RichiestaTime e RichiestaData(vedi Timestamp) per ottenere la data e l'ora al momento della richiesta alla rete e richiama sendRequestWithTimer per settare nuovamente il timeout.

Il broadcast una volta ricevuta la richiesta da parte del Network, inoltra a sua volta la richiesta a tutti i nodi della rete tramite una OneWay chiamata richiestaInfoNodo(). Anche essi rispondono direttamente al Network attraverso una OneWay denominata rispostaNumeroBlocchi(), specificando il numero di blocchi contenuti nella propria blockchain e la propria location.
Dopo aver ricevuto il numero di blocchi da parte di ogni nodo, il Network provvede a confrontare i numeri di blocchi ricevuti, trovando la blockchain più lunga della rete e stampandola a video.

stampaBlockchainNodo è un'operation grazie alla quale riusciamo a stampare la blockchain di ogni singolo nodo attivo nella rete; questo è possibile scorrendo l'array implementato nel network visualizer contenente tutte le chiavi pubbliche dei nodi e leggendo i rispettivi file blockchain e stampando i loro contenuti.

Nel momento in cui un nodo viene coinvolto in una transazione, invia, tramite l'operation invioInfoNodo@Nodo_Network(infoNodo), tutte le informazioni riguardanti tale transazione al Network.

Nel Network tale operation inizia con lo scorrimento dell'array global.arrayTransazioni[].
Una volta trovata la corrispondenza tra global.arrayTransazioni[i] e la chiave pubblica del nodo che ha chiamato l'operation, memorizza tutte le voci della transazione riguardante tale nodo :

```
global.arrayTransazioni[i].transaction[global.nTransazioni].nodeSeller.publicKey=infoNodo.transaction.nodeSeller.publicKey;

global.arrayTransazioni[i].transaction[global.nTransazioni].nodeBuyer.publicKey=infoNodo.transaction.nodeBuyer.publicKey;

global.arrayTransazioni[i].transaction[global.nTransazioni].jollar = infoNodo.transaction.jollar;

global.arrayTransazioni[i].transaction[global.nTransazioni].timestamp = infoNodo.transaction.timestamp;

 global.arrayTransazioni[i].transaction[global.nTransazioni].date  = infoNodo.transaction.date
```

In seguito stampa a video, solo per i due nodi coinvolti nell'ultima transazione eseguita nel sistema, tutte le transazioni che questi hanno effettuato divise per entrate/uscite.

Dopo aver verificato che sia stata eseguita una notevole quantità di lavoro, ogni nodo, indipendentemente dalla vincita o meno della ricompensa, si preoccupa di inviare al Network la quantità totale di Jollar che possiede, tramite l'operation invioInfoJollar(infoNodoJollar).

Tale valore viene memorizzato dal Network all'interno dell'array global.arrayTransazioni[i].jollarTot al fine di ottenere, al termine di ogni transazione, la quantità totale di Jollar all'interno della rete (jollarRete = jollarRete + global.arrayTransazioni[i].jollarTot).