Documentazione progetto LSO

• **Jollar Laboratorio Sistemi Operativi A.A. 2017-2018**

• **LosJollarHermanos**

• **Indirizzo mail di riferimento:** andrea.gurioli2@studio.unibo.it

• **Componenti**: Gurioli, Andrea, Matr. 0000738568
Mongardi, Michele, Matr. 0000801511
Mazzini, Federico, Matr. 0000800858
Pietrucci, Giovanni, Matr. 0000802834
LaMura, Francesco, Matr. 0000789522

DESCRIZIONE GENERALE DEL PROGETTO

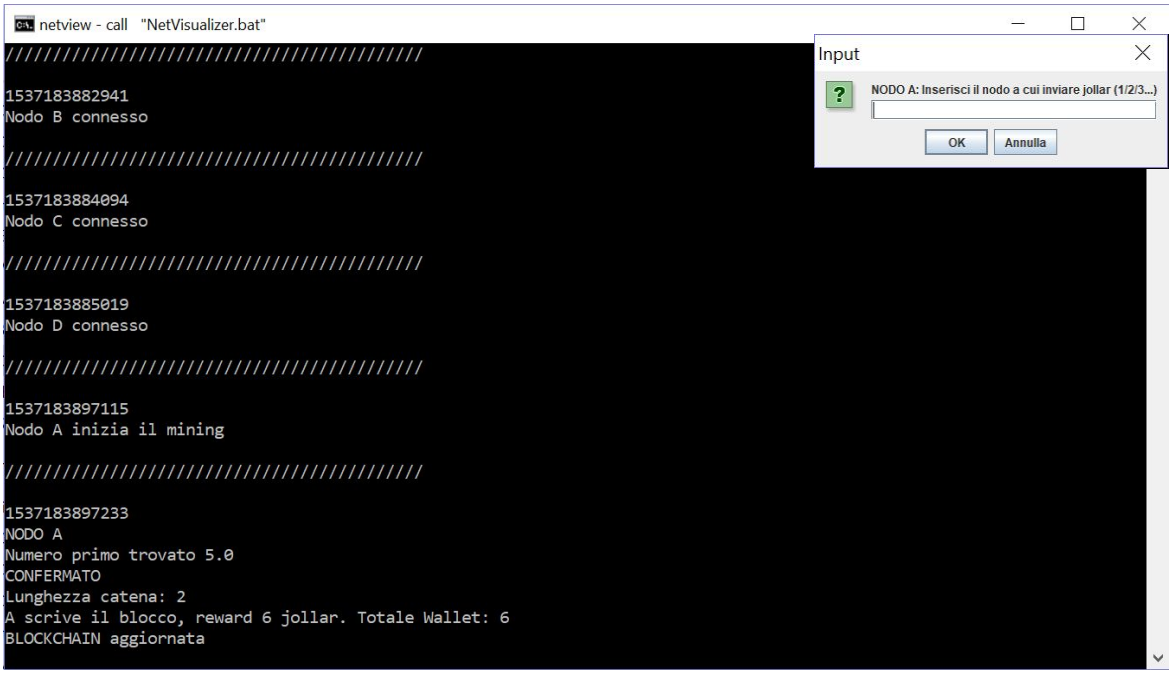
Il progetto consiste nella simulazione di una rete peer-to-peer adibita allo scambio di transazioni con le specifiche tipiche di una criptovaluta, con il linguaggio Jolie.
Le componenti principali di questo sono la Blockchain e la Proof of work.
Blockchain: una struttura dati che può essere vista come registro pubblico di tutte le transazioni avvenute fino a quel momento nella rete.
Proof of work: metodo con cui le transazioni possono essere scritte nella blockchain dai nodi, ovvero trovando numeri primi attraverso la catena di Cunningham del 1° tipo.
Peer o nodi: si è deciso di scomporre questi in diverse parti, server, client e un' ultima che si occupa delle transazioni tra i vari peer.
Inoltre è utilizzato un server **Timestamp** il quale offre un servizio che permette di conoscere l'ora esatta all'interno della rete ed un **Network Visualizer** che mostra lo stato in tempo reale della rete.
La documentazione contiene una descrizione dettagliata dei servizi implementati.
Partendo dal fatto che il linguaggio Jolie rende ogni variabile una struttura dati ad albero, si è deciso di salvare blockchain, wallet e storico transazioni di ogni nodo in file .json, data la facilità di scrittura di alberi all' interno di esso.
Infatti, per visualizzare correttamente la Blockchain.json basterà aprire il file Blockchain.
Per lo storico transazioni dei vari nodi guardare Transazioni[1,2,3,4].json e per il wallet Wallet[1/2/3/4].json

ISTRUZIONI PER L'UTILIZZO DELLA DEMO

Per avviare la demo basta semplicemente eseguire il file demo.bat. Questo è un file batch che al suo interno richiama a sua volta tutti i vari file scritti in jolie per l' implementazione della rete. (Tutti gli altri file .bat sono necessari per l' avvio, ma è automatico)

All' avvio i vari cmd riguardanti server, client e gestori transazioni, saranno eseguiti ma ridotti ad icona per una questione di chiarezza, infatti saranno visibili solamente la finestra del Network Visualizer e quella delle transazioni del peer A (in quanto nella demo è l' unico che manda denaro). Le altre finestre sono necessarie per l' utilizzo del programma, ma non fondamentali per l' utente.

Avvio. Il Network Visualizer mostrerà se sono conessi nodi e di cosa si stanno occupando, ad esempio se qualcuno trova un primo e aggiorna la blockchain. Inoltre sono visualizzate le transazioni avvenute.



Come effettuare transazioni.

La finestra si presenta in questo modo: in un primo momento inserire il numero del nodo a cui si vuole inviare Jollar e premere ok/invio. Inserire la quantità di essi e ripremere ok/invio. La numerazione dei nodi corrisponde alla successione delle lettere dell' alfabeto. (es A=1, B=2 ecc).

N.B. I nodi partiranno da un wallet vuoto, pertanto per eseguire transazioni dal nodo A è necessario aspettare che nel Network Visualizer arrivi l' avviso che A ha scritto il primo blocco[1], prima di questo avviso, il nodo A risulterà ovviamente con 0 jollar nel wallet.

Vi sono comunque controlli che non permettono l' invio di jollar negativi, mancanti o nulli, pertanto non è un problema provare la transazione anche prima del momento [1].

A volte l' interfaccia grafica non appare perchè rimane dietro il cmd del Network, consigliamo di chiudere tutto e riavviare per sicurezza, in ogni caso, è normale se non appare subito, perchè i bat sono lanciati sequenzialmente con ritardi di almeno 1 secondo e transaction è l' ultimo.

DISCUSSIONE SULLE STRATEGIE DI IMPLEMENTAZIONE

Non abbiamo diviso il progetto in parti, quindi non vi erano diversi compiti per i diversi componenti, in quanto ci sembrava il tutto molto collegato.

Pertanto questo è stato svolto prevalentemente via skype (per motivi di lontananza geografica), con tutti i componenti del gruppo presenti. A questo si sono aggiunte volte in cui ci siamo visti di persona per questioni di chiarezza.

PROBLEMI RISCONTRATI

Occuparsi sia delle transazioni sia del mining in contemporanea, abbiamo perciò pensato di dividere un peer in più parti, alcune eseguite in concurrent.

Gestione delle chiavi pubbliche e private, non abbiamo trovato un servizio ascii per implementarle, l' argomento è stato discusso più avanti al punto 4.

1. TIMESTAMP

Il timestamp è un server di supporto, eseguito in in modo execution{concurrent}, in modo tale da fornire le operation in qualsiasi momento. Al suo interno:

Operation getTime. Request Response che restituisce il tempo rappresentato come offset in millisecondi rispetto alla mezzanotte (UTC) del 1º gennaio 1970. E' stata implementata attraverso l' operation getCurrentTimeMillis dell' interfaccia time. Questo tempo è richiesto dai nodi ogni qual volta svolgano un operazione.

Operation getLocation. E' utilizzata per conoscere i nodi presenti e connessi nella rete, salvando tutte le location in un file chiamato LocationPorte. Leggibile da tutti, consente ai nodi di sapere l' indirizzo degli altri.

2. NODI

Ogni nodo deve sia svolgere operazioni, sia rimanere in ascolto di richieste, pertanto si è deciso di suddividere i compiti in quella che ci sembrava la maniera ottimale. Ottenendo quindi parte client, server, e gestore transazioni. I codici sono denominati rispettivamente Clientpeer_.ol, Serverpeer_.ol, Transaction_.ol.

N.B. Al "_ " va sostituito A, B, C, D a seconda del peer.

2.2 Parte Server

I server si occupano di gestire la verifica dei numeri primi, attraverso il teorema di fermat. Ognuno di essi svolge questa verifica per ogni numero trovato nella rete, non soltanto quello trovato dal rispettivo client.

Esso gestisce anche tutto quello che riguarda la scrittura della blockchain nel relativo file, dello storico transazioni riguardanti il nodo e l' azzeramento delle transazioni temporanee già scritte nella blockchain.

Verifica numeri primi. La parte client del nodo mina con il relativo algoritmo e richiede verifica a tutti i nodi presenti nella rete con l' operation RR controllo()(), se il numero è validato da tutti allora si procede con la scrittura del blocco in blockchain, altrimenti il numero è rifiutato e non verrà scritto nulla.

```
//controllo del numero primo attraverso l'algoritmo di Fermat
[controllo(c1)(m){
    println@Console(c1)();
    powReq2.base=2;
    powReq2.exponent=c1-1;
    pow@Math(powReq2)(result2);
    m=result2%c1
}]
```

Ricezione di transazioni solo ricevute. Operation OW sender(). Aggiorna il wallet specifico aggiornando un file chiamato Wallet__.
es. Nodo A riceve 2 Jollar, al file WalletA sono aggiunti 2 Jollar.

```
// parte di ricezione di una transazione da un nodo x al nodo A
[sender( request )]{
    mon.filename="WalletA";
    mon.format="json";
    readFile@File(mon)(monres);
    //aggiornamento wallet
    jollar=int(monres)+int(request.jollar);
    println@Console("Nodo ricevente" + request.nodeSeller)();
    println@Console("Nodo inviante" + request.nodeBuyer)();
    mon.content<<jollar;
    writeFile@File(mon)()
}
```

Ricezione di transazioni riguardanti il nodo__. Operation OW saver(), il nodo(parte server) riceve tutte le transazioni che lo riguardano, sia emesse che ricevute. Scrive o aggiorna poi un file chiamato Transazioni__.

```
//ricezione e salvataggio delle transazioni riguardanti il nodo A
[saver(receive)]{
    undef(trans2);
    global.contatoreTran++;
    println@Console("CONTATORE:    "+global.contatoreTran)();
    global.block.transaction[global.contatoreTran]<<receive;
    trans2.filename="TransazioniA";
    trans2.format="json";
    trans2.append=1;
    trans2.content<<receive;
    writeFile@File(trans2)()
}
```

Ricezione di tutte le transazioni della rete. Tutti i nodi sono a conoscenza delle transazioni avvenute, in modo tale che quando un peer scrive un blocco, in questo possano essere messe tutte le transazioni fatte dopo la scrittura del blocco precedente. Pertanto in un blocco possono esserci più transazioni.

L’ operation saveForBlockchain() riceve un tipo definito transaction e salva questo in un albero.

```
//ricezione di tutte le transazioni avvenute dopo la scrittura dell'ultimo blocco,
// e quindi non ancora scritte in blockchain
[saveForBlockchain(transactions)]{
    global.contatoreTran2++;
    println@Console("CONTATORE:    "+global.contatoreTran2)();
    global.block2.transaction[global.contatoreTran2]<<transactions
}
```

L’ operation saveBlock() da il permesso di salvare quest’ albero in un nuovo blocco della blockchain (permesso che, come detto in precedenza, arriva dal client al quale è stato confermato il numero primo).

```
//Quando viene trovato un numero primo da parte del rispettivo client,
//avremo la scrittura di un nuovo blocco nella blockchain contenente global.block2
[saveBlock(blocco)]{
    blocco<<global.block2;
    println@Console(blocco.previousBlockHash)();
    sav.filename="Blockchain";
    sav.format="json";
    sav.append=1;
    sav.content<<blocco;
    writeFile@File(sav)();

    undef( req );
    req.filename="LocationPorte";
    req.format="json";
    readFile@File(req)(resLoc);

    //reset dinamico dei blocchi temporanei
    //(una volta che il blocco viene creato manda una richiesta agli altri server per cancellare il blocco
    for(i=1,i<=resLoc,i++){
        locationControl=(resLoc.("figlio"+i));
        if(locationControl!="socket://localhost:8001"){
            ResetServer.location=locationControl;
            resServ@ResetServer()
        }
    }
}
```

Una volta che un nodo scrive un blocco in Blockchain, con l’ operation resServ() tutti ne vengono a conoscenza ed eliminano le transazioni temporanee che dovevano essere scritte fino a quel momento.

```
//reset dei blocchi temporanei dopo il blocco viene scritto (vengono azzerate le transazioni precedenti)
[resServ()]{
    undef(global.block2);
    undef(blocco)
}
```

2.3 Parte Client

Il principale compito del client è lo svolgimento della Proof of Work. Inizialmente è stato impostato uno sleepTime, in modo tale che sia simulata una ricerca dei primi più complessa. Questo sleep è ripetuto ogni ciclo che il client svolge.

Il codice è suddiviso in cicli annidati tra loro. Inizialmente un while definisce il numero totale di valori che si andranno a calcolare, riducendo il costo computazionale. E' stato scelto come numero totale 8, perchè fino a questo, i numeri calcolati sono effettivamente primi, tranne uno che viene rifiutato, mentre dopo 8 l' algoritmo non riesce più a trovare altri primi, rifiutando tutti quelli trovati.

Annidato al while è presente un for che da il via al processo vero e proprio di POW. Da questo è ottenuto un numero che viene verificato due volte, la prima è un autoverifica del nodo in cui si sta lavorando, mentre la seconda è una verifica da parte di tutti gli altri.

Se il tutto viene verificato allora il numero è primo e iniziano le operazioni di scrittura del blocco in Blockchain.

```
//hash del blocco precedente per blockchain
md5@MessageDigest( "hashing"+ res2.primo )( response );
block.previousBlockHash=response;
block.difficulty=p[i]/1;
//invio alla parte server della restante parte della blockchain (hash e difficulty)
//insieme alle transazioni per quel blocco formeranno il nuovo blocco
saveBlock@Blocksender(block);
```

3. GESTORE TRANSAZIONI

Le transazioni sono gestite da una finestra di dialogo swingUI. Viene richiesto a quale nodo inviare Jollar e quanti, si effettuano controlli per verificare che i numeri inseriti siano positivi.

Dopodichè si procede all' invio della transazione al nodo ricevente con l' operation OW sender() in cui avviene l' effettiva ricezione del denaro con conseguente aggiornamento del wallet.

Poi con saver() si aggiorna lo storico delle transazioni dei nodi coinvolti, mentre con saveForBlockchain() tutti i nodi connessi alla rete vengono informati della transazione.

```
if(send.jollar<=monres && send.jollar>0){
    undef( req );
    req.filename="LocationPorte";
    req.format="json";
    readFile@File(req)(resLoc);
    sendString@Netview("TRANSAZIONE -> "+ send.jollar + " JOLLAR" +"\n"+"MITTENTE: " + send.nodeE
    //invio al ricevente la transazione
    sender@TransactionA( send );

    //invio a tutti la transazioni
    for(i=1,i<=resLoc,i++){
        locationControllo=(resLoc.("figlio"+i));
        TransactionForBlock.location=locationControllo;
        saveForBlockchain@TransactionForBlock(send);

        //invio solo ai nodi coinvolti la transazione
        if((locationControllo==send.nodeBuyer)||(send.nodeSeller==locationControllo)){
            TransactionSaver.location=resLoc.("figlio"+i);
            saver@TransactionSaver( send )
        }
    };
```

4. PUBLIC KEY E PRIVATE KEY

Non siamo riusciti ad implementare le chiavi pubbliche e private, tuttavia le abbiamo sviluppate concettualmente.

L'idea che è stata presa in considerazione è stata quella di generare per ogni blocco delle chiavi pubbliche e private. Al momento della transazione il nodo inviante avrebbe inviato la sua location ed i jollar criptate con la chiave pubblica del nodo ricevente, il quale avrebbe decriptato il tutto con la sua chiave privata ed avrebbe controllato la validità delle informazioni ricevute. Il problema è stato quello di non essere riusciti ad implementare il servizio Ascii che permette di assegnare ad ogni carattere un valore numerico o non, diverso da quello iniziale. In questo modo saremmo riusciti a criptare la Location ed i Jollar.

Tuttavia abbiamo inserito nel progetto il codice che permette di generare delle chiavi pubbliche e private attraverso la crittografia RSA.