DOCUMENTAZIONE PROGETTO JOLLAR

- Jollar – Laboratorio Sistemi Operativi A.A. 2017-2018
- Nome Gruppo: **G.I.Jollar**
- Indirizzo mail di riferimento: stupa ecc..
- Componenti:
    - Elatfi Hamza, Matr. 766364
    - Filippini Daniele, Matr. 768278
    - Stupazzoni Leonardo, Matr. 765585
    - Lodi Thomas, Matr. 800963

CONTENUTO DELLA DOCUMENTAZIONE

La seguente documentazione contiene una breve descrizione del progetto, la descrizione in generale delle features implementate, i passi da seguire per eseguire una demo e una descrizione dettagliata delle alternative di implementazione, dei problemi riscontrati e delle funzioni presenti nel progetto.

DESCRIZIONE DEL PROGETTO

Il progetto ha lo scopo di simulare una rete peer to peer in cui i nodi della rete possano scambiarsi transazioni aventi come oggetto la moneta virtuale creata, in questo caso chiamata Jollar, e registrare queste transazioni in una blockchain. I nodi della rete fungono da client/server in quanto possono inviare e ricevere risposte/transazioni. Oltre ad i client la rete presenta ulteriori servizi quali il servizio di broadcaster, il servizio di Timestamp e il Network Visualizer. Il servizio broadcaster è svolto da un client/server chiamato Broadcast il quale ha il compito di fare da servizio intermediario per alcune operazioni, per esempio l'invio di chiavi pubbliche alla rete, l'invio di transazioni, l'invio di blocchi, ecc. Il servizio di Timestamp, invece, è un server sempre in ascolto che ha il compito di fornire tramite apposite richieste l'ora esatta all'interno della rete. Il servizio chiamato Network Visualizer ha lo scopo di richiedere alla rete lo stato attuale, facendo richiesta ai vari client di fornirgli alcune informazioni da stampare a video, come le transazioni effettuate da ogni client e la sua relativa Blockchain; inoltre il Network Visualizer richiede al broadcast di fornirgli la blockchain ufficiale della rete.

DESCRIZIONE DELLE FEATURES IMPLEMENTATE

Ogni client al suo lancio crea una chiave pubblica che viene inviata al Broadcast che la salva per poi inviarla a tutti gli altri nodi connessi alla rete in quel momento, che a loro volta memorizzano in una variabile interna la chiave ricevuta così da utilizzarla per inviare le transazioni al client in questione. Quando viene creata una transazione questa deve essere convalidata dalla maggior parte dei nodi della rete prima di poterla inviare agli altri nodi tramite il servizio di Broadcast. La convalidazione consiste nel verificare che le chiavi pubbliche utilizzate del nodo venditore e del nodo compratore esistano nel registro di chiavi che ogni nodo ha memorizzato al suo interno. Oltre a questa verifica ogni nodo controlla che all'interno della transazione vi sia una chiave privata, nel caso questa non sia vuota viene considerata sempre veritiera. Una volta convalidata la transazione questa viene inviata a tutti i nodi, quest'ultimi quando ricevono la transazione fanno partire la proof of work. La proof of work consiste nel compiere del lavoro per trovare una catena di numeri che soddisfi determinati requisiti. Una volta trovata questa catena, il nodo crea un

blocco da poter scrivere nella blockchain, inserendo all'interno la transazione, la catena di numeri trovata, la difficoltà della catena e altre informazioni utili. Prima di poter scrivere il blocco, però, il nodo deve far convalidare il proprio blocco creato agli altri nodi della rete. La convalida del blocco consiste in: -verifica che la transazione del blocco ricevuto non sia già stata spesa; -verifica che i numeri della catena siano primi attraverso il principio del teorema di Fermat; -verifica che la difficoltà della catena inserita nel blocco sia maggiore della difficoltà presente nell'ultimo blocco della blockchain. Una volta controllati tutti questi parametri viene inviata una risposta al nodo che aveva richiesto la convalidazione del blocco. Si configurano tre possibili casi:

- Nel caso in cui la maggioranza delle risposte siano positive, il nodo procede alla scrittura, solo dopo aver controllato che nella propria blockchain il blocco che si vuole scrivere non sia già stato scritto da un altro nodo della rete. Nel caso in cui il blocco risulti già scritto si controlla se la difficoltà della catena di numeri trovata è maggiore della difficoltà inserita nell'ultimo blocco della blockchain. In questo caso la catena viene salvata così da poter essere riutilizzata per la scrittura di un prossimo blocco e non perdere il lavoro effettuato precedentemente. Ogni volta che viene scritto un nuovo blocco il file contenente la blockchain viene aggiornato così da non perdere i dati.
- Nel caso in cui la maggioranza delle risposte contengano un errore relativo alla catena dei numeri, il nodo aumenta di 1 l'origine della catena e controlla che la transazione non sia già stata spesa. Nel caso la transazione sia ancora valida cerca di trovare una nuova catena per poter scrivere un nuovo blocco.
- Nel caso in cui la maggioranza delle risposte contengano un errore relativo alla difficoltà della catena, viene aumentata la lunghezza della catena trovata precedentemente di 1 e si controlla che la transazione non sia già stata spesa. Nel caso in cui la transazione sia ancora valida si cerca di trovare una nuova catena per poter scrivere un nuovo blocco.

Nel caso in cui due o più nodi inviano versioni differenti del nuovo blocco contemporaneamente, alcuni nodi potrebbero riceverne una versione, altri un'altra. Il nodo che ha lavorato su un blocco non sincronizzato se ne rende conto quando una nuova validazione viene richiesta. In questo caso il nodo in questione richiede al broadcast le blockchain degli altri nodi. Una volta ricevute le blockchain esegue dei controlli per cercare quella più lunga. Nel caso ci siano due o più blockchain con la stessa lunghezza, il nodo controlla quale di esse è più presente sulla rete. Se anche in questo caso vi sono più blockchain, verifica quale ha il timestamp dell'ultimo blocco più piccolo. Una volta trovata la blockchain ufficiale il nodo procede alla sincronizzazione.
Nel caso in cui il nodo non sincronizzato si accorge di aver ricevuto un reward per la scrittura di un blocco non presente nella blockchain ufficiale, i Jollar ingiustamente ricevuti vengono scalati.

Il Broadcast quando viene lanciato resta in attesa di ricevere richieste dai nodi della rete. Le richieste che può ricevere sono le seguenti:

- Richiesta di invio della chiave di un nodo agli altri nodi della rete;
- Richiesta di invio di una transazione creata da un nodo agli altri nodi della rete;
- Richiesta di invio di un blocco creato da un nodo agli altri nodi della rete;
- Richiesta da parte dei nodi della rete di inviare le risposte di convalidazione al nodo scrittore;

- Richiesta da parte di un nodo non sincronizzato di ricevere le blockchain presenti negli altri nodi della rete;
- Richiesta da parte di un nodo scrittore di un blocco di inviare la notifica di avvenuta scrittura agli altri nodi della rete.

Il Network Visualizer quando richiesto interroga i nodi per ricevere informazioni sullo stato della rete e il Broadcast per ricevere la versione ufficiale della blockchain. Una volta completate le richieste stampa a video le seguenti informazioni:

- La data e l'ora del server Timestamp al momento della richiesta;
- Per ogni nodo: la sua chiave pubblica, le transazioni che ha effettuato divise per entrate ed uscite, la versione della blockchain che utilizza ed i Jollar che possiede;
- La versione ufficiale della blockchain con tutta la struttura dati che contiene.

ISTRUZIONI PER ESEGUIRE LA DEMO

1. Avvio del Server Timestamp, contenuto nel file timestamp.ol;
2. Avvio del Broadcaster, contenuto nel file broadcast.ol;
3. Avvio del Network Visualizer, contenuto nel file networkVisualizer.ol;
4. Avvio del primo nodo, generatore del primo blocco, contenuto nel file client1.ol;
5. Avvio dei client 2, 3 e 4 contenuti rispettivamente in client2.ol, client3.ol, client4.ol;
6. Per effettuare una transazione scrivere: numero_nodo/jollar_da_inviare;
7. Per chiudere il client digitare "exit".
8. Per interrogare il Network Visualizer premere invio.

DISCUSSIONE DELLE STRATEGIE DI IMPLEMENTAZIONE

STRUTTURA E DIVISIONE DEL PROGETTO

Nell'implementazione del progetto siamo partiti dalla creazione della rete peer-to-peer e quindi dalla creazione dei quattro client che ne fanno parte. Ad ognuno di questi client è stata assegnata una porta di comunicazione univoca in modo che questi potessero comunicare tra loro. Il secondo passo è stata la creazione di un servizio di broadcast che consentisse di instradare i messaggi provenienti da un nodo verso tutti gli altri nodi della rete. Quindi ogni nodo ha una outputPort verso il broadcast che a sua volta è a conoscenza di tutte le porte dei nodi sulla rete. La creazione del servizio di broadcast è stata fatta per evitare che i client si dovessero preoccupare personalmente di inviare i messaggi a tutti i nodi della rete. Il passo successivo è stato implementare un sistema di scambio di moneta. I singoli nodi quindi hanno un portafoglio contenente i jollar disponibili che possono essere scambiati. Le transazioni si compongono da: chiave del seller e del buyer, dei jollar da trasferire, del timestamp per verificare il momento in cui è avvenuta la transazione e di un identificativo id. Il nodo può effettuare la transazione solo se nel suo portafoglio dispone di una somma pari o maggiore ai jollar da inviare.
La transazione, dopo essere stata convalidata dagli altri nodi, viene inviata al broadcast che verifica che il nodo di destinazione sia online e successivamente procede all'invio della transazione a tutti i nodi.
Abbiamo implementato la validazione verificando che in tutti i nodi siano presenti sia la chiave del seller sia quella del buyer, in modo tale che la transazione sia effettivamente spedita ad uno dei

nodi presenti nella rete. Una volta finita la validazione della transazione ogni nodo cerca di scrivere il blocco per inserirlo all'interno della blockchain. A questo punto abbiamo implementato quindi il sistema di proof-of-work. Come prova di lavoro abbiamo scelto di creare solamente catene di Cunningham del primo tipo, in modo da gestire meglio il passaggio da una catena all'altra. Inizialmente abbiamo impostato l'origine delle catene create a 2, se queste verranno validate si aumenterà la lunghezza della catena, invece nel caso in cui i numeri trovati non siano primi abbiamo deciso di aumentare l'origine di 1. Quindi avremo [2,5], [2,5,11], [2,5,11,23], [2,5,11,23,47]. Arrivati a questo punto la catena successiva avrà come ultimo numero 95 che non soddisfa il requisito di primalità, quindi passeremo ad una catena con origine aumentata di 1, [3, 7] e così via. Dopo aver concluso l'implementazione della creazione delle catene, abbiamo introdotto il concetto di difficoltà. La formula per calcolare questa difficoltà è: $D = k + (pk – r)/pk$ dove k è la lunghezza della catena trovata, pk è il numero successivo della catena ed r è il resto prodotto dal Test di Fermat. Calcolata la difficoltà, il nodo procede alla creazione del blocco da far convalidare agli altri nodi della rete. Il blocco è formato dalle seguenti informazioni: - transazione; - catena creata; - difficoltà della catena creata; - chiave pubblica del nodo creatore del blocco; - timestamp al momento della creazione; - id per identificare il blocco; - id del blocco precedentemente scritto. Il passo successivo è stato implementare in ogni nodo un metodo per la convalidazione dei blocchi e un metodo per l'invio della risposta, descritti nel paragrafo delle features implementate. Infine, abbiamo sviluppato il metodo per la scrittura dei blocchi convalidati nella blockchain di ogni nodo, con la conseguente scrittura della blockchain nel file.

ALTERNATIVE DI IMPLEMENTAZIONE

- In ogni client è presente l'istruzione execution(concurrent), così che ognuno di essi possa eseguire più thread contemporaneamente. Un'alternativa poteva essere l'uso di execution(sequential) che non avrebbe consentito di soddisfare richieste arrivate contemporaneamente, ma solamente una di esse alla volta. Noi abbiamo optato per introdurre un esecuzione concorrente perché nel caso in cui arrivassero due richieste, per esempio di validazione, da parte di altri client contemporaneamente, con l'esecuzione concorrente si è in grado di soddisfare entrambe le richieste simultaneamente.
- Un'alternativa al servizio di broadcast che abbiamo implementato poteva essere quella di far inviare i messaggi da un nodo ad ogni altro nodo della rete senza passare per un intermediario. Abbiamo deciso di inserire questo servizio intermediario così da non gestire in ogni nodo i vari errori che si possono incontrare nell'invio di messaggi. Un esempio è l'invio di un messaggio ad un nodo non raggiungibile. Adottando questa soluzione, però, abbiamo riscontrato un inconveniente, ovvero, nel caso in cui il broadcast sia offline, i nodi della rete non sono in grado di comunicare tra di loro. Questo inconveniente l'abbiamo risolto stampando un messaggio in cui viene richiesto di lanciare il broadcast prima di effettuare una qualsiasi operazione.
- Per la validazione delle transazioni un'implementazione più efficace e corretta sarebbe stata quella di creare una firma digitale della transazione attraverso una coppia di chiavi pubblica e privata, con questa firma, creata con la chiave privata, sarebbe stato facile per gli altri client verificare l'identità di colui che ha effettuato la transazione attraverso l'utilizzo della sua chiave pubblica. Non siamo riusciti a implementare questa funzione

perché ciò avrebbe richiesto l'uso di un algoritmo a chiave pubblica, come RSA, per crittografare e decriptare i messaggi.

- Per la creazione delle catene abbiamo utilizzato il solo modello di Cunningham di primo tipo, un'implementazione alternativa poteva essere quella di introdurre tutti e tre i modelli ed effettuare una scelta casuale nel nodo tra uno di essi. Non abbiamo optato per questa soluzione perché ci risultava difficile controllare il cambio di origine di tutti i tipi di catene e volevamo avere una omogeneità nel lavoro svolto da tutti i nodi.
- Per il salvataggio della blockchain vi erano due alternative, salvarla in un file oppure in un database. Noi abbiamo optato per il salvataggio su file perché ci risultava più semplice e comodo in quanto vi era la possibilità di salvare tutta la struttura dati in formato json. Questo formato permette di salvare ed estrapolare i dati in maniera molto più semplice.
- Per l'assegnazione del reward di 6 jollar al client che ha scritto il blocco, inizialmente avevamo pensato di assegnare i jollar solamente al momento della validazione del blocco successivo, così da essere sicuri che questo fosse stato inserito nella blockchain ufficiale. Abbiamo poi optato per assegnare il reward subito dopo la scrittura del blocco e successivamente (alla richiesta di validazione del blocco seguente) abbiamo verificato se questo era effettivamente inserito nella blockchain ufficiale. In caso negativo procediamo alla rimozione dei jollar precedentemente assegnati ingiustamente.

PROBLEMI RISCONTRATI

- Uno dei problemi incontrati è stato gestire gli errori provenienti da client non connessi alla rete in quel momento. Per esempio, quando si cercava di inviare una richiesta ai nodi della rete, il broadcast controllava un nodo per volta che fosse online per inviargli la richiesta. Nel caso in cui trovasse un nodo offline catturava l'errore attraverso il costrutto install e lo stampava a video. Il problema vi era quando il broadcast trovava un nodo offline prima di averli controllati tutti, in quanto al primo nodo trovato offline stampava l'errore e si bloccava senza eseguire gli altri controlli. Abbiamo provato numerose soluzioni, quali introdurre un ciclo che controllasse i nodi della rete con all'interno un install per catturare gli errore incontrati, ma anche in questo caso si bloccava al primo errore incontrato non andando avanti con il ciclo. Un'altra soluzione provata è stata inserire all'interno del ciclo uno scope con al suo interno l'install per la cattura degli errori, in questo caso il ciclo andava avanti all'infinito ma stampando sempre lo stesso errore quindi non aumentando l'indice del ciclo.

  <esempio ciclo senza scope>

```
[broadcast( msg )]{
        users[0] = LOCATION_CLIENT1;
        users[1] = LOCATION_CLIENT2;
        users[2] = LOCATION_CLIENT3;
        users[3] = LOCATION_CLIENT4;
        tipo = msg instanceof Autenticazione;
        response = "ok";
        for ( i=0, i<#users, i++ ) {
            Client.location = users[i];
            install(IOException => {
```

```
                if( tipo == true ) {
                    if(i = 0){
                        println@Console("Client 1 offline, non gli ho inviato la chiave")()
                    }else if(i = 1){
                        println@Console("Client 2 offline, non gli ho inviato la chiave")()
                    }else if(i = 2){
                        println@Console("Client 3 offline, non gli ho inviato la chiave")()
                    }else{
                        println@Console("Client 4 offline, non gli ho inviato la chiave")()
                    }
                }
            });
            sendKey@Client(msg)
        }
    }
<esempio ciclo con scope>
[broadcast( msg )]{
   users[0] = LOCATION_CLIENT1;
   users[1] = LOCATION_CLIENT2;
   users[2] = LOCATION_CLIENT3;
   users[3] = LOCATION_CLIENT4;
   tipo = msg instanceof Autenticazione;
   response = "ok";
   for ( i=0, i<#users, i++ ) {
    Client.location = users[i];
    scope( scopeName )
    {
     install(IOException => {
      if( tipo == true ) {
       if(i = 0){
        println@Console("Client 1 offline, non gli ho inviato la chiave")()
       }else if(i = 1){
        println@Console("Client 2 offline, non gli ho inviato la chiave")()
       }else if(i = 2){
        println@Console("Client 3 offline, non gli ho inviato la chiave")()
       }else{
        println@Console("Client 4 offline, non gli ho inviato la chiave")()
       }
      }
     });
     sendKey@Client(msg)
    }
   }
  }
}
```

Non trovando altre soluzioni abbiamo optato per una soluzione che non fornisce scalabità al progetto, inserendo dentro ad ogni install i controlli mancanti. Per esempio, dentro all'install di controllo del client2, metto anche i controlli del client 3 e 4. Dentro all'install del client 3 metto il controllo del client 4. Uscito dal controllo del client 2, vi è il controllo del client 3, dentro al suo install vi è anche il controllo del client 4 e così via per ogni client. Con la nostra implementazione nel caso dovessimo aggiungere un ulteriore client alla rete bisognerebbe aggiornare gran parte del metodo di controllo, mentre con una soluzione scalabile basterebbe aggiungere poche righe di codice.

- Un problema inizialmente riscontrato con l'esecuzione concorrente è stato l'accesso alle risorse globali (condivise dai vari processi) del singolo client, alcuni di questi processi infatti hanno risorse in comune e il risultato finale dell'esecuzione dei processi dipendeva dall'ordine in cui questi erano eseguiti ed era del tutto impredicibile. Un esempio di questo errore lo abbiamo riscontrato nel conteggio delle risposte alla convalidazione dei blocchi. L'errore vi era quando i processi accedevano alla variabile di conteggio contemporaneamente producendo una somma che in alcuni casi risultava errata. La soluzione è stata inserire blocchi di sincronizzazione dove si rende necessario l'accesso a variabili globali, in modo tale che solo un processo alla volta possa entrare in un blocco synchronized con lo stesso id.

```
synchronized(id){
    global.numAck = global.numAck + 1 //variabile di conteggio
}
```

DESCRIZIONE DETTAGLIATA DELLE FEATURES IMPLEMENTATE

- **Funzione caricamentoBlockchain**: questa funzione è leggermente diversa nei nodi presenti sulla rete. La funzione presente nel client1 ha il compito di controllare se esiste un file contenente la blockchain, in caso positivo lo carica, mentre in caso negativo crea un nuovo file scrivendo al suo interno il primo blocco della blockchain. In tutti gli altri nodi che non siano il primo la funzione è identica, esegue un controllo iniziale per verificare se esiste il file appartenente al client in questione, se questo esiste, carica la blockchain presente nel file, altrimenti carica l'unica blockchain presente nel sistema, ovvero quella del client1 che ha creato il primo blocco.
- **Funzione creationChain**: questa funzione è uguale in tutti i nodi della rete ed al suo interno contiene la formula di Cunningham di primo tipo per la creazione della catene e la formula per il calcolo della difficoltà.

```
<codice formula di Cunningham>
 pi = global.origine;
 catena[0] = int(pi);
 //println@Console("0:  "+catena[0])();
 i = 1;
 while(i < l+1){
  powrequest.base = 2;
  powrequest.exponent = i;
  pow@Math(powrequest)(num);
  p = num * pi + (num - 1);
```

```
  catena[i] = int(p);
  //println@Console(i+":  "+catena[i])();
  i = i+1
 };
<codice calcolo difficoltà>
 //calcolo del numero successivo della catena
 powrequest.base = 2;
 powrequest.exponent = l+1;
 pow@Math(powrequest)(num);
 pk = num * 2 + (num - 1);
 //test di fermat per determinare il resto r
 powrequest.base = 3;
 powrequest.exponent = pk-1;
 pow@Math(powrequest)(number);
 //formula per calcolare la difficulty
 r = number % pk;
 risultato = (pk-r)/pk;
 k = double(l+1);
 diff = k + risultato
```

- **Funzione scriviBlocco:** questa funzione è uguale per tutti i nodi presenti nella rete ed al suo interno contiene il codice per inserire un nuovo blocco nella blockchain del nodo. Ciò avviene solo dopo aver controllato che il blocco in questione non sia già stato scritto da un altro nodo. Successivamente scrive la blockchain in possesso in un file chiamato "FileClient-numero del nodo-.json". In caso il blocco sia già stato scritto esegue dei controlli che hanno lo scopo di non buttare il lavoro fatto dal nodo, quindi confronta la difficoltà della catena trovata con quella dell'ultimo blocco scritto, in caso questa sia maggiore la salva in una variabile così da utilizzarla per la scrittura del blocco successivo, mentre in caso sia minore non viene salvata e il nodo perde il lavoro fatto.
- **Funzione proceduraDiSincronizzazione:** questa funzione è uguale in tutti i nodi della rete. Il codice al suo interno esegue i confronti fra le blockchain ricevute dal broadcast in seguito ad una richiesta di sincronizzazione da parte del nodo. I confronti fra le blockchain hanno lo scopo di estrapolare la blockchain ufficiale, controllando se vi è una blockchain più lunga delle altre. Nel caso ci siano due o più blockchain con la stessa lunghezza, la funzione controlla quale di esse è più presente sulla rete. Se anche in questo caso vi sono più blockchain, verifica quale ha il timestamp dell'ultimo blocco più piccolo. Dopo aver trovato la versione ufficiale dalla blockchain la confronta con quella presente nel nodo. Nel caso fosse diversa, la funzione controlla se il client aveva scritto un blocco non sincronizzato, in caso affermativo vengono sottratti i jollar di reward avuti in precedenza.
- **Operazione [in(scelta)]:** questa operazione è uguale in tutti i nodi della rete e contiene all'interno tutte le operazioni principali che un nodo è in grado di poter eseguire, come l'invio dei jollar, conoscere il proprio saldo oppure uscire dal sistema. Questo metodo prende in input il comando inserito dall'utente e lo confronta con i possibili casi. Nel caso in cui l'utente digiti 'saldo' vengono mostrati a video i jollar disponibili. Nel caso in cui l'utente digiti 'exit' per uscire dal sistema viene inviato il saldo al broadcast che lo salva in un apposito file denominato cash.json, questo verrà poi caricato ad un successivo riavvio

del client. Nel caso in cui l'utente volesse inviare jollar ad un altro nodo, in questo metodo vi è il codice che crea la transazione.
<esempio di codice nel caso in cui volessimo inviare jollar al client2>

```
if(numeroNodo == 2){
      scope (scopeName)
      {
       if(global.pkClient2 == " "){
        println@Console("Il client a cui si vuole inviare i Jollar non e' disponibile!")();
        mostraOpzioni
       }else{
        //creazione della transazione
        nodeBuyer.publicKey = global.pkClient2;
        nodeSeller.publicKey = MyPublicKey;
        tr.nodeBuyerID.publicKey = nodeBuyer.publicKey;
        tr.nodeSellerID.publicKey = nodeSeller.publicKey;
        tr.jollar = global.jollarOut;
        requestTime@timestamp(request)(t);
        tr.time = t;
        tr.id = string(new);
        md5@MessageDigest(MyPrivateKey)(risposta);
        tr.nodeSellerID.privateKey = risposta;
        //prima di mandare la transazione al broadcast devo far validare la transazione agli
        //altri client
        install( IOException =>{
         println@Console( "Impossibile validare la transazione perche' non tutti i nodi
                                  sono online" )()
        } );
        numOk = 0;
        for(i = 0, i < #users, i++){
         Client.location = users[i];
         validationTr@Client(tr)(response);
         if(response == "ok"){
          numOk ++
         }
        };
        //dopo aver ricevuto le risposte controllo che il numero di ok sia maggiore della
        //metà dei client
        numMaggioranza = int(#users / 2)+1;
        //se il numero di ok eguaglio o supera la maggioranza dei client la transazione è
        //corretta e quindi posso inviarla al broadcast
        if(numOk >= numMaggioranza){
         sendTransaction@broadcast( tr )
        }
       }
      }
```

- **Operazione sendKey(autentication)**: questa funzione ci permette di inviare e ricevere le chiavi pubbliche che identificano i nodi della rete. La funzione presente nel client ha il compito di ricevere dal broadcast le chiavi dei nodi connessi e di salvarle in apposite variabili globali. La funzione, invece, presente nel broadcast ha il compito di ricevere le chiavi dai nodi della rete, salvarli in apposite variabili globali, ed re-inviare ai nodi della rete le chiavi in possesso.
- **Operation validationTr(request)(response):** questa operazione request-response viene richiesta dal nodo che crea una nuova transazione. I nodi che ricevono tale richiesta hanno il compito di svolgere le verifiche per la convalidazione delle transazioni. Il metodo riceve in input una transazione dopodiché controlla che le chiavi utilizzate esistano all'interno della rete e che vi sia la presenza di una chiave privata non vuota. A seconda del risultato della convalidazione restituisce una risposta affermativa oppure negativa.
- **Operazione sendTransaction(transaction):** è una operazione one-way che è presente sia nei client che nel broadcast. I client invocano l'operazione sendTransaction@broadcast nel momento in cui la transazione viene validata. Il broadcast eseguirà quindi un sendTransaction@client per ogni client online nella rete e manderà un ackSendTransaction@client al client che gli aveva inviato la richiesta per restituire l'esito della operazione. Quando un client riceve una transazione allora in quel momento comincia la proof of work per cercare di scrivere il blocco.
- **Operazione ackSendTransaction(risposta):** questa operazione one-way viene richiesta dal broadcast per inviare al nodo creatore della transazione l'esito di quest'ultima. Il client una volta ricevuta una risposta positiva scala i jollar della transazione dal suo saldo.
- **Operazione sendBlock(block):** è una operazione one-way che è presente sia nei client che nel broadcast. Una volta che il client ha finito di creare il blocco lo invia al broadcast tramite sendBlock@broadcast, questi lo invierà agli altri client tramite sendBlock@client. Quando il client riceve questa operazione compie la validazione del blocco e ritorna un ack di risposta con l'esito.

  <codice per svolgere il Test di Fermat>

```
//controllo della primalità dei numeri
    num << block.catena.numeriPrimi;
    catena = block.catena.lunghezza;
    i = 0;
    while( i < catena){
     if(num[i] == 2){
       powrequest.base = 3
     }else{
       powrequest.base = 2
     };
       powrequest.exponent = num[i]-1;
       pow@Math(powrequest)(number);
       if( (number % num[i]) != 1 ){
         i = i+1;
         isOk = "Errore_NumeriPrimi"
       }else{
         i = i+1
```

```
        }
    };
```

- **Operazione ackSendBlock(ack):** è una operazione one-way che è presente sia nei client che nel broadcast. I client invocano ackSendBlock@broadcast una volta validato un blocco ed inviano la risposta al broadcast. Quest'ultimo avrà il compito di inviare l'ack al nodo costruttore del blocco invocando ackSendBlock@client. Il client che riceve la risposta controlla il risultato della validazione. Gli ack vengono conteggiati e nel momento in cui sono pari alla metà+1 dei client si effettua una operazione. Se la maggior parte dei client ha dato risposta positiva si passa alla scrittura del blocco invocando la funzione scriviBlocco e mandando una notifica di scrittura agli altri client con write@broadcast.
  <codice per la verifica che il numero di risposte sia maggiore della metà +1 dei nodi>
  ```
  global.numAck = global.numAck + 1;
  numMaggioranza = int(#users/2)+1;
  global.isOk = ack.isOk;
  if( global.numAck == numMaggioranza){
    write@broadcast(ack.block) | scriviBlocco
  }
  ```
  Se nella risposta è contenuto un errore relativo alla catena di numeri significa che questi non sono primi allora si aumenta l'origine della catena e se il blocco non è ancora stato scritto si crea una nuova catena per provare di scriverlo.
  <codice per l'aumento di 1 dell'origine delle prossime catene>
  ```
  if(ack.block.catena.numeriPrimi[0] == global.origine){
  //println@Console( "Errore numeri primi: aumento il numero di origine" )();
    synchronized( syncOrigine  ){
      global.origine = global.origine + 1;
      global.lunghezza = 1
    }
  };
  ```
  Nell'aumentare l'origine del blocco abbiamo impostato un numero limite dopo il quale si arresta l'esecuzione per evitare numeri troppo grandi che jolie non riesce a calcolare.
  Se la risposta ha rilevato un errore nella difficoltà si aumenta la lunghezza di 1 e se il blocco non è ancora stato scritto si crea una nuova catena per provare di scriverlo.
- **Operation requestData(request)(response):** questa operazione request-response viene richiesta dal Network Visualizer e consiste nel richiedere ad ogni nodo connesso alla rete alcune informazioni quali la chiave pubblica, il saldo, le transazioni divise per entrate ed uscite e la versione della blockchain.
- **Operation getBlockchain()(resp):** questa semplice operazione request-response viene richiesta dal broadcast ai nodi della rete e consiste nel fornire appunto, al broadcast, la blockchain in uso dal nodo.
- **Operation write(block):** questa operazione one-way viene mandata dal nodo che ha scritto un nuovo blocco agli altri nodi. Si tratta di una notifica di scrittura di un blocco da parte di un altro nodo. Il nodo, quindi, scrive il blocco ricevuto nella propria blockchain controllando prima che non l'abbia già fatto. Dopo averlo inserito aggiorna il proprio file in cui è salvata la blockchain con il nuovo blocco scritto.