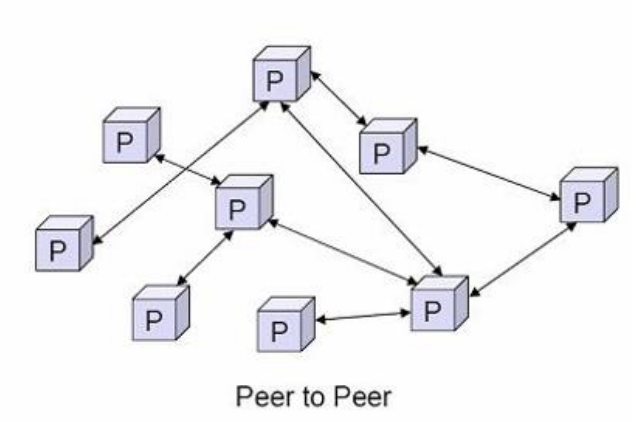
- Jollar – Laboratorio Sistemi Operativi A.A. 2017-2018
- Nome Gruppo: *Freeze all Jolie functions*
- Email di riferimento: andrea.lombardo7@studio.unibo.it
- Componenti:
    - Lombardo Andrea, 0000789952
    - De Giosa Matteo, 0000789982

Descrizione generale del progetto

Il progetto consiste nella realizzazione di una *pseudo-blockchain*, ovvero una rete *peer2peer* decentralizzata che gestisce delle transazioni senza l'intermediazione di un'autorità centrale garante.



La blockchain è stata realizzata come una struttura dati contenente blocchi: ogni blocco contiene una transazione. La nostra implementazione supporta quattro nodi: il primo nodo genera le transazioni (ovvero manda moneta – in questo caso Jollar – agli altri tre), mentre questi ultimi concorrono alla formazione dei blocchi.

È presente poi un *timestamp server* che, oltre a fornire il timestamp ai nodi (dato che poi servirà alla creazione di *hash* unici), gestisce anche il *discovery* di nuovi nodi.

Ad ogni nodo è assegnata una porta di input, con location *hardcoded*, e una porta di output per comunicare con gli altri nodi tramite *binding dinamico*. Una volta inizializzato ogni nodo notifica il timestamp server del suo collegamento e scarica da questo la lista degli altri nodi presenti online. Se non ci sono altri nodi online, crea egli stesso la blockchain e procede alla creazione di un blocco generatore. Se invece ci sono altri nodi online, richiede a questi la blockchain più aggiornata.

Il nodo uno attende che gli altri tre nodi si connettano, dopodiché inizia a generare delle transazioni. Ogni transazione contiene un hash univoco, per garantire che non venga "spesa" più volte. Una volta creata il nodo uno manda la transazione agli altri tre nodi. Questi la processano e la inseriscono in un blocco.

Per validare il blocco e garantire la sicurezza della rete è necessario che i nodi effettuino una *proof-of-work*: nella nostra implementazione, similmente a come fa *Primecoin*, la proof-of-work consiste nel generare catene di numeri *pseudo-primi* - nel nostro caso *catene di Cunningham di primo tipo.*

$(p_1, \ldots, p_n)$ tale che per ogni $1 \leq i < n$ allora $p_i + 1 = 2p_i + 1$

La lunghezza della catena da generare si calcola a partire dalla difficoltà, che nella nostra implementazione è fissa.

Una volta effettuata la proof-of-work ogni nodo invia il blocco creato agli altri tre nodi. Questi a loro volta prima verificano che il timestamp del blocco sia successivo a quello dell'ultimo blocco aggiunto, poi controllano che l'hash della transazione non sia già presente in blocchi precedenti, ed infine con il *teorema di Fermat* verificano che i numeri della proof-of-work siano primi (nonché che la lunghezza frazionaria della proof-of-work sia maggiore o uguale alla difficoltà richiesta).

Fermat's Little Theorem.

Let p be a prime which does not divide the integer a, then $a^{p-1} = 1 \pmod{p}$.

Verificatesi queste condizioni, il nodo aggiunge il blocco alla propria versione della blockchain.

Solo il primo che arriva viene verificato e aggiunto. Gli altri sono scartati. Al nodo che ha generato il blocco viene assegnato un reward fisso di 6 Jollar.

La stessa sequenza di operazioni viene ripetuta per altre due transazioni.

Completate queste operazioni, lo stato del sistema può essere monitorato per mezzo di un *Network Visualizer*. Esso calcola i *balance* di tutti i nodi e fornisce un elenco dettagliato delle transazioni avvenute.

Istruzioni per la demo

1. Avvio timestamp_server.ol
2. Avvio nodo1.ol
3. Avvio nodo2.ol
4. Avvio nodo3.ol
5. Avvio nodo4.ol
6. Il nodo 1 chiederà input da tastiera per procedere con seconda transazione
7. Input da tastiera per procedere con terza transazione
8. Avvio network_visualizer.ol
9. Input da tastiera per visualizzare riepilogo rete

Discussione sulle strategie di implementazione

Divisione del progetto

Abbiamo preferito andare controcorrente non suddividendo le pagine di codice da scrivere o assegnando compiti specifici per ciascuno di noi. Tale scelta, seppur possa sembrare inefficiente, nasconde un gran numero di vantaggi, come ad esempio l'*assenza di incomprensioni nel gruppo:* lavorando fianco a fianco ogni idea viene sufficientemente discussa, compresa, e di conseguenza implementata se giudicata corretta da parte di entrambi**.**

Inoltre riteniamo che il confronto *IRL* sia una pratica impopolare – vuoi per problemi logistici, vuoi per comodità dei membri del gruppo – ma che permette un ricco scambio di informazioni ed un miglioramento delle proprie abilità di team building.

Problemi principali:

1. **Assenza di *broadcast* in Jolie**

Le istruzioni per il progetto richiedevano che ogni nodo inviasse in *broadcast* la transazione agli altri nodi: dopo un'attenta analisi abbiamo scoperto che tale funzione non è disponibile primitivamente in Jolie.

Ciò ci ha costretto originariamente a ricorrere a delle coppie di porte di output e di input per ogni connessione nodo a nodo. Abbiamo poi parzialmente risolto il problema fissando una porta di input per ogni nodo e usando il *binding dinamico* sulla porta di output.

```
// Porta d'ascolto per il nodo1

inputPort In1 {
    Location: "socket://localhost:8000"
    Protocol: sodep
      Interfaces: Nodo_interface
}

// Porta di output per contattare gli altri nodi (cambio di location con binding dinamico)

outputPort Out {
    Protocol: sodep
    Interfaces: Nodo_interface
}
```

Un'altra soluzione al problema, sebbene molto più complessa, consisterebbe nel creare un protocollo personalizzato; un'altra ancora *embeddare* un servizio java (ma quest'ultima opzione negherebbe il vantaggio dell'usare la gestione semplificata delle socket di jolie).

2. ***Node discovery***

Il problema di "scoprire" gli altri nodi connessi alla rete.

Nell'implementazione finale abbiamo fatto in modo che il *timestamp server* tenesse una lista dei nodi connessi alla rete, e che ogni nodo nel suo *init* notificasse il server della sua connessione e scaricasse la lista.

```
init
{
    notifyServer@TimeOut()(response);

    // il server mi risponde con la mia location, che fungerà da chiave pubblica

    myInputPort = response;
    println@Console("Sono il nodo " + myInputPort)();
    getLocations@TimeOut()(listaPorte);

    synchronized( id1 ){
        global.listaPorte << listaPorte
    };
    println@Console("Ho preso la lista delle porte")();
    getBlockchain
}
```

```
// i nodi contattano questo servizio per aggiornare la lista dei nodi online e conoscere la propria location

[
    notifyServer()(b)
    {
        println@Console("Mi e' arrivata una richiesta di connessione")();
        synchronized(syncToken)
        {
            b = "socket://localhost:800" + global.nodiOnline;
            global.lista.porta[global.nodiOnline] = b;

            println@Console("Assegno al nodo la location " + b)();


            if (#global.lista.porta > 1) {
                println@Console("Avviso gli altri nodi di aggiornare le proprie liste")()
            };
            for (j = 0, j < #global.lista.porta - 1, j++)
            {
                NodoContact.location = global.lista.porta[j];
                aggiornaLocations@NodoContact();
                println@Console("Ho aggiornato il nodo " + global.lista.porta[j])()
            };

            global.nodiOnline++
        }
    }
]

// i nodi contattano questo servizio per avere la lista dei nodi online

[
    getLocations()(b)
    {
        b << global.lista
    }
]
```

Questa è una soluzione molto banale, ma non dissimile da quanto fatto dalle crypto-monete nei primi anni.

Una soluzione, sebbene molto più complessa, sarebbe stata quella di creare un algoritmo di *node discovery*, interno ad ogni nodo. A tal proposito, una soluzione più semplice di altre sarebbe quella di tenere una *seed list* di nodi sempre online (i *miners*, ad esempio, dovrebbero essere sempre online), e connettersi a uno o più di questi per ottenere la lista di tutti i nodi connessi alla rete. Soluzione tuttavia scartata per le difficoltà implementative.

3. **Binding dinamico sulle porte di input**

Inizialmente avremmo voluto che una volta contattato il *timestamp server* fornisse al nodo la location della sua porta di input (che funge anche da *chiave pubblica* per quel nodo).

Ci siamo però scontrati col fatto che in Jolie è impossibile fare binding dinamico sulle porte di input (non a torto: un cambio della porta di input a *runtime* potrebbe avere conseguenze disastrose).

Per questo motivo ci siamo dovuti accontentare di una soluzione molto banale: *hardcodare* le porte di input di ogni nodo, costringendoci a inizializzare i nodi in ordine (prima nodo1.ol, poi nodo2.ol, etc.).

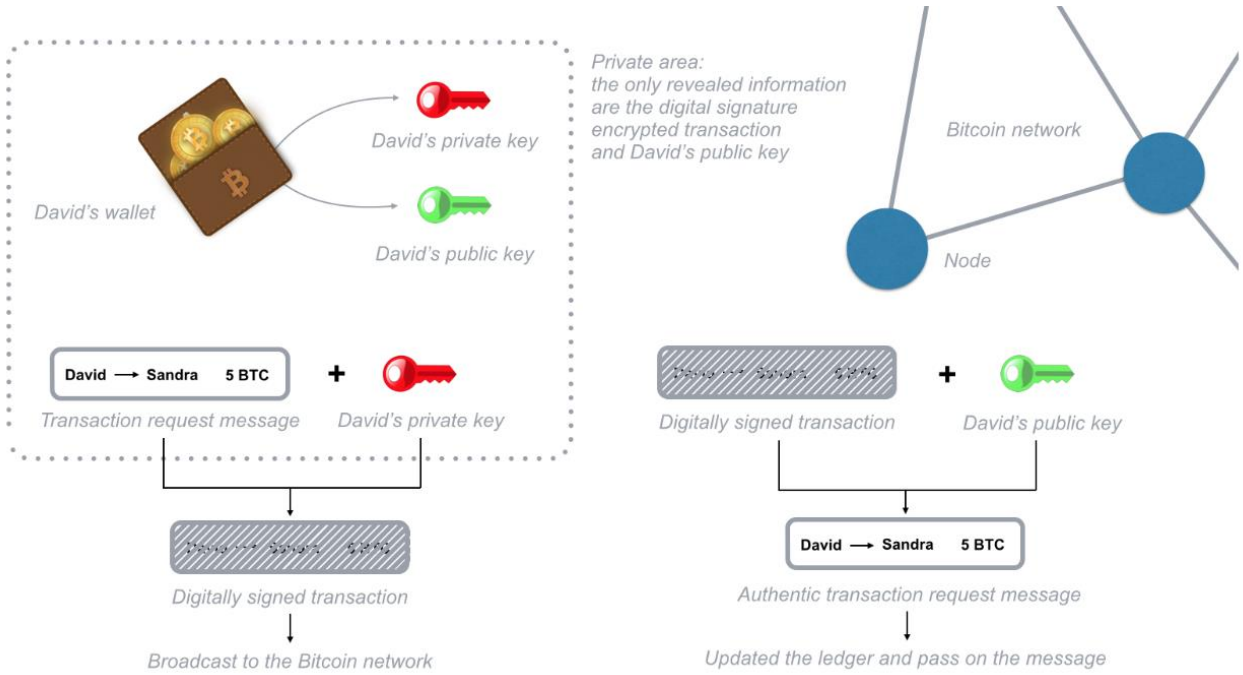
4. **Binding dinamico sulle porte di output e parallelismo**

Se un nodo vuole inviare una transazione o un blocco agli altri nodi con il binding dinamico sulle porte di output, non può farlo in parallelo (o per lo meno per quanto ne sappiamo). Ciò comporta che un nodo riceverà sempre la transazione e/o il blocco per primo, ottenendo un vantaggio nella *corsa per la proof-of-work* sugli altri nodi.

Non abbiamo trovato soluzioni a questo problema.

5. **Firma delle transazioni**

In Bitcoin ogni transazione viene firmata con la chiave privata del nodo che la crea. In questo modo l'integrità di questa viene verificata da chi la riceve per mezzo della chiave pubblica del nodo mandante. Questo funziona grazie alla crittografia a chiave asimmetrica.



Jolie non supporta nativamente la crittografia.

Soluzione a questo potrebbe essere l'*embeddare* un servizio Java che sfrutta le classi apposite, ma tale soluzione è stata scartata per via della dispendiosità a livello di tempo.

Ciò rende inutile la struttura dati nodo per come è stata presentata nelle istruzioni. La chiave pubblica coincide con l'indirizzo del nodo, la chiave privata è priva di significato in quanto per creare un hash univoco della transazione è sufficiente il timestamp.

6. Problematica sulla proof-of-work

Lavorando con il teorema di Fermat per verificare la primalità dei numeri si arriva molto velocemente a interi che sforano il double. Per questo motivo, lavorando con catene di Cunningham di primo tipo, l'unica catena gestibile da Jolie è quella di cinque elementi:

2-5-11-23-47.

Catene di lunghezza maggiore sforerebbero facilmente il double (ricordiamo che nel teorema di Fermat si eleva 2^(p-1), con *p* elemento della catena, mentre il limite del double è 1.79*10^308).

Questo problema ci ha costretto a fissare la difficoltà della proof-of-work a quattro, con l'ovvia conseguenza della sua facilità di generazione (la difficoltà attuale di Primecoin è **11.7248**).

Una soluzione a questo problema: passare per java usando la classe *BigIntegers*.

Ciononostante abbiamo simulato che Jolie sappia gestire numeri più grandi: se la prima catena non va bene, il nodo ne cerca un'altra a partire dall'elemento successivo a quello che non è primo.

```
    // se la verifica non va bene, rifaccio proof of work, partendo dall'elemento successivo della catena


    if (confermaInterna == false) {
        println@Console("Proof of work errata, non tutti i numeri sono primi")();
        esp.base = double(2);
        esp.exponent = double(index);
        pow@Math(esp)(potenza);

        powchain[0] = potenza*powchain[0] + potenza - 1;
        proofOfWork
    }
}
```

7. Esecuzione concorrente

Non siamo riusciti a rendere i nodi *concurrent*, cioè a rendere i servizi nuovamente disponibili una volta chiamati. Non sappiamo se questo sia un problema di sincronizzazione, che comunque abbiamo gestito per mezzo di semafori nel momento in cui un blocco deve essere inviato.

```
init
{
    global.semaphore.name = "Primo";
    global.semaphore.permits = 1;
    global.nodiOnline = 0;
    global.semaphore2.name = "Secondo";
    global.semaphore2.permits = 3;
    release@SemaphoreUtils(global.semaphore)(confermaSemaphore);
    release@SemaphoreUtils(global.semaphore2)(confermaSemaphore);
    release@SemaphoreUtils(global.semaphore2)(confermaSemaphore);
    release@SemaphoreUtils(global.semaphore2)(confermaSemaphore)
}
```

```
main {

    // metodo per prendere semaforo (il chiamante passa un int che indica quale semaforo)

    [
        prendiSemaforo(numeroSemaforo)()
        {
            if (numeroSemaforo == 1) {
                acquire@SemaphoreUtils(global.semaphore)(confermaSemaphore)
            } else {
                acquire@SemaphoreUtils(global.semaphore2)(confermaSemaphore)
            }
        }
    ]

    // metodo per rilasciare semaforo

    [
        lasciaSemaforo(numeroSemaforo)()
        {
            if (numeroSemaforo == 1) {
                release@SemaphoreUtils(global.semaphore)(confermaSemaphore)
            } else {
                release@SemaphoreUtils(global.semaphore2)(confermaSemaphore)
            }
        }
    ]
```

8. Difetti della documentazione di Jolie

Molti dei nostri problemi sono nati e si sono protratti a causa dell'incompletezza della documentazione di Jolie. Nella libreria delle API la maggior parte delle funzioni non sono spiegate, lasciando all'utente il dovere di "testarle" per capirne il funzionamento.

È stato il caso delle operazioni *ReadFile* e *WriteFile*: avremmo voluto che i nodi salvassero la propria versione della blockchain alla chiusura, ma l'assenza di qualsivoglia informazione su come questi lavorino ci ha dissuaso dal nostro intento.

writeFile

```
writeFile( WriteFileRequest )( void )
 throws


FileNotFound( FileNotFoundType )


IOException( IOExceptionType )
```

Writes a Jolie structure out to an external file

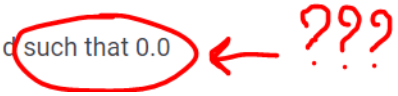
WriteFileRequest

```
type WriteFileRequest: void {
    .filename: string
    .format?: string {
        .schema*: string
        .indent?: bool
        .doctype_system?: string
        .encoding?: string
    }
    .content: undefined
    .append?: int
}
```

random

```
random( void )( double )
```

Returns a random number d such that 0.0



L'operazione *getTimeFromMilliSeconds* converte millisecondi in una struttura dati di tipo *TimeValuesType*, ma accetta soltanto *int*, quando invece una variabile che contiene millisecondi deve essere necessariamente *double*. In pratica è inutile.

getTimeFromMilliSeconds

```
getTimeFromMilliSeconds( int )( TimeValuesType )
```

In generale mancano nella documentazione molte delle informazioni più avanzate necessarie a realizzare un progetto quale una blockchain. Il fatto poi che sia un linguaggio relativamente nuovo e poco usato (ma non gliene diamo le colpe) fa sì che sul web escano più risultati su Angelina Jolie che sull'omonimo linguaggio.