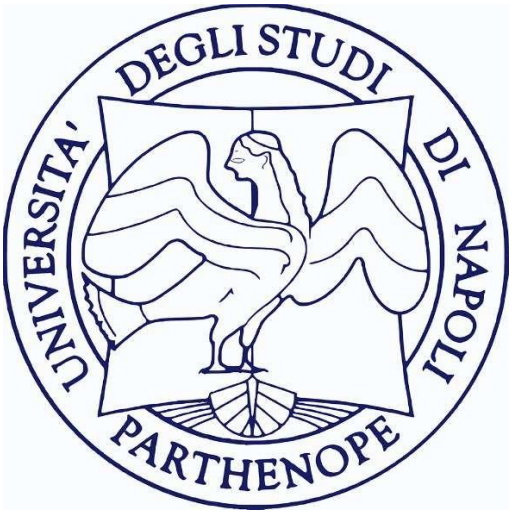


Università degli Studi di Napoli "Parthenope" www.uniparthenope.it
Dipartimento di Scienze e Tecnologie
Centro Direzionale Isola C4
80143 Napoli - Italy

Reti di Calcolatori e Laboratorio di Reti di Calcolatori

A.A. 2018-2019

Progetto BlockExplorer

Studenti:

COGNOME	NOME	MATRICOLA
Bevilacqua	Vincenzo	0124001490
Di Marino	Antonio	0124001344

Docente:

Alessio Ferone

INDICE

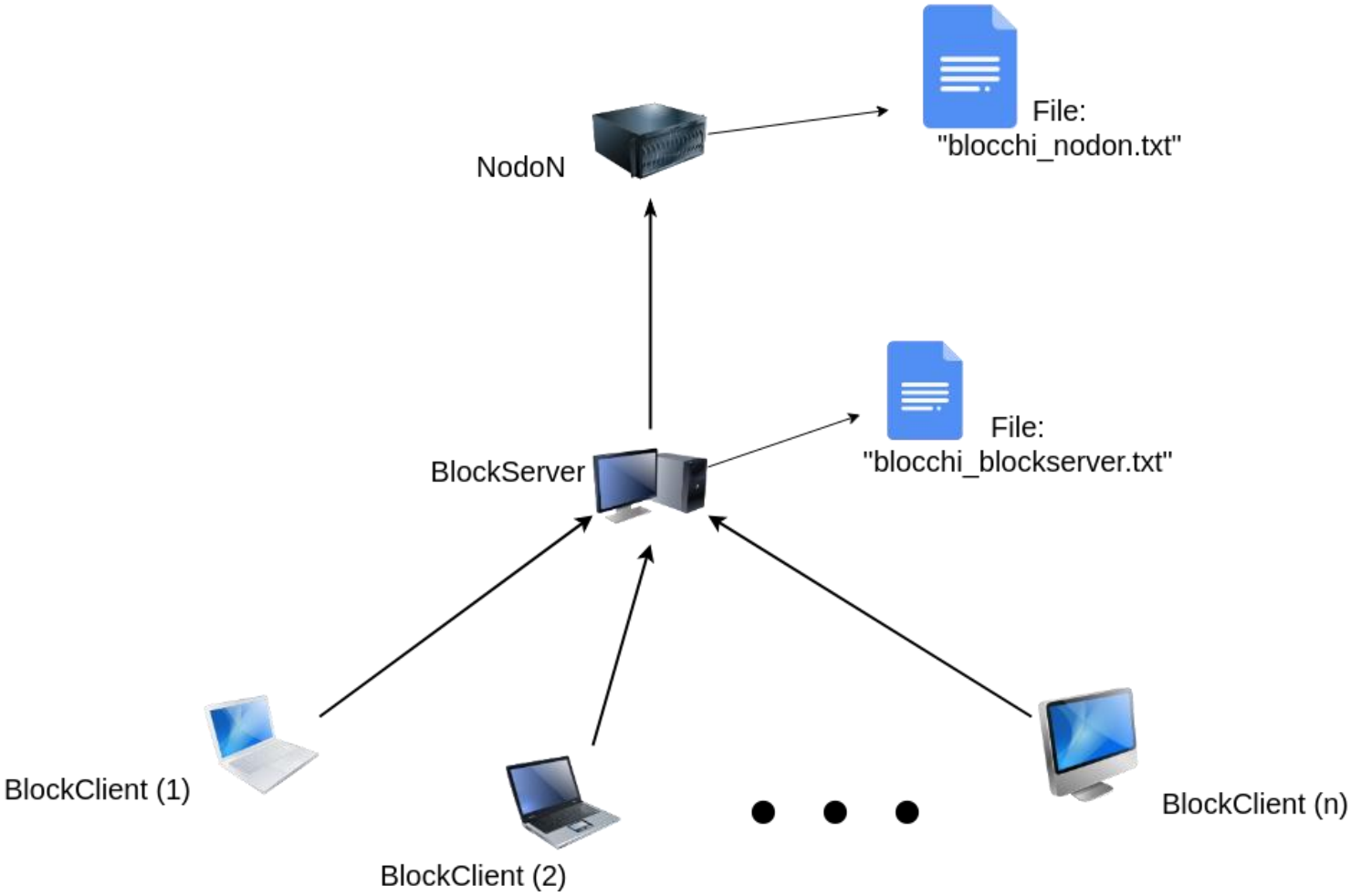
Descrizione del progetto

Il progetto si pone l'obiettivo di analizzare una *blockchain*, essa è una sequenza di blocchi in cui ogni blocco contiene una transazione. La rete si compone di tre entità fondamentali, un *NodoN*, un *Block-Server* e un *BlockClient*.

Un *NodoN* è un ` entità che simula una blockchain e ne salva la copia su di un file, il *BlockServer* ha il compito di connettersi con il *NodoN* per ricevere una copia della blockchain e fornire informazioni su di essa ad uno o più *BlockClient* , il *BlockClient* a sua volta si connette al *BlockServer* per poter analizzare le transazioni contenute nella blockchain.

Descrizione e schemi dell'architettura

Schema della rete



Descrizione della rete

La rete si compone di un *NodoN* che genera blocchi casuali in una blockchain per poi salvarli in un suo file locale "blocchi_*NodoN*.txt". In una visione più ampia della rete il *NodoN* si collega ad una rete di peer dove la funzione principale è quella di scambiarsi i blocchi di una blockchain, in questo progetto invece si presuppone che tutto ciò già avvenga, per cui il trasferimento dei blocchi viene simulato generando i blocchi casualmente, quindi il *NodoN* verrà utilizzato solo

come **server**. La finalità di questo progetto consiste nell'analisi dei blocchi e per tal fine si necessita di un solo *NodoN*.

Il *BlockServer* è l'entità che ha una doppia funzionalità, esso si occupa di reperire i blocchi connettendosi al *NodoN* e di salvarli in un proprio file locale "blocchi_*BlockServer*.txt", inoltre si occupa di fornire ad n *BlockClient* dei servizi di analisi della blockchain.

Il *BlockClient* rappresenta un generico client che vuole reperire informazioni riguardanti i blocchi della blockchain, esso si connette al *BlockServer* per interrogare la blockchain. In particolare, i *BlockClient* possono richiedere di:

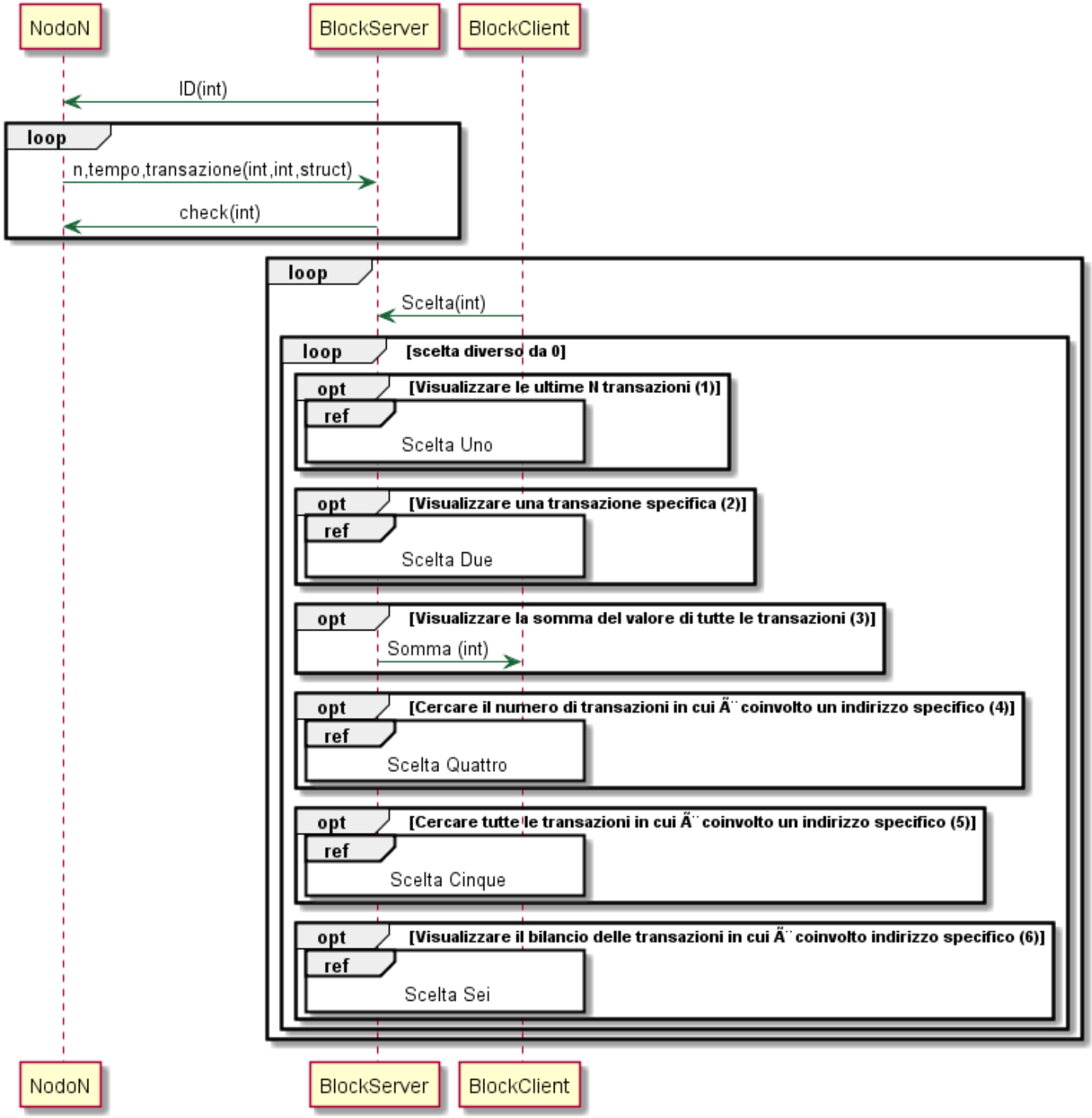
- Visualizzare le ultime n transazioni.
- Visualizzare i dati di una specifica transazione
- Visualizzare la somma dei valori di tutta la blockchain
- Visualizzare il numero di transazioni in cui è coinvolto un indirizzo specifico.
- Visualizzare i dati di tutte le transazioni in cui è coinvolto un indirizzo specifico.
- Visualizzare il bilancio delle transazioni in cui è coinvolto un indirizzo specifico.

Protocollo di livello trasporto

Nel livello trasporto del TCP/IP si è scelto di utilizzare il protocollo **TCP** invece dell'UDP, dato che bisogna assicurare che i blocchi vengano correttamente trasferiti dal *NodoN* al *BlockServer* con un preciso ordine, bisogna considerare anche l'importanza delle transazioni che nella realtà dei fatti contengono denaro ed esse non possono essere perse, e infine dato che il TCP mantiene lo stato della connessione si possono monitorare e gestire ipotetiche cadute di connessione.

Descrizione e schemi del protocollo applicazione

Schema generale



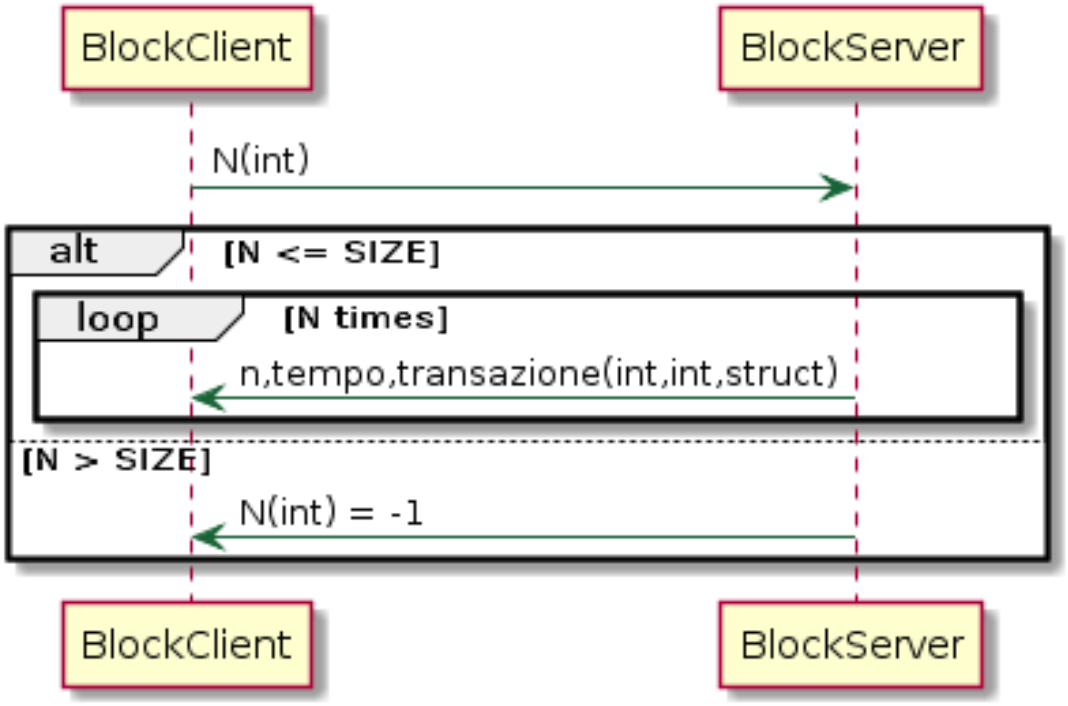
Descrizione schema generale

Questo schema raffigura come a livello applicazione si svolgono le comunicazioni tra le entità in gioco. Il *BlockServer* invierà un intero al *NodoN* per comunicargli l'ultimo indice della blockchain che lui stesso possiede in modo tale che il *NodoN* gli invierà i restanti blocchi della blockchain (in loop), i blocchi vengono inviati sottoforma di struttura che contiene un identificativo numerico del blocco **n**, un **tempo** randomico e una sottostruttura **transazione** che a suo interno contiene come campi **IP**:**PORTA** Mittente, **IP**:**Porta** Destinatario , il **credito** da trasferire e **l'identificativo** della transazione. Il *BlockServer* invierà un intero (1) al *NodoN* ogni qualvolta riceve un blocco, questo intero avrà la funzione di check, cioè verrà usato per valutare la stabilità della connessione.

Per quanto riguarda la comunicazione client-server tra *BlockClient* e *BlockServer*, le richieste e le risposte avvengono diversamente in base al servizio richiesto dal *BlockClient*.

Il *BlockClient* invierà in tutti i casi un intero al *BlockServer* che corrisponderà al servizio scelto, di seguito poi verranno mostrati i protocolli di comunicazione tra *BlockClient* e *BlockServer* in base alla scelta effettuata.

Schema Scelta Uno



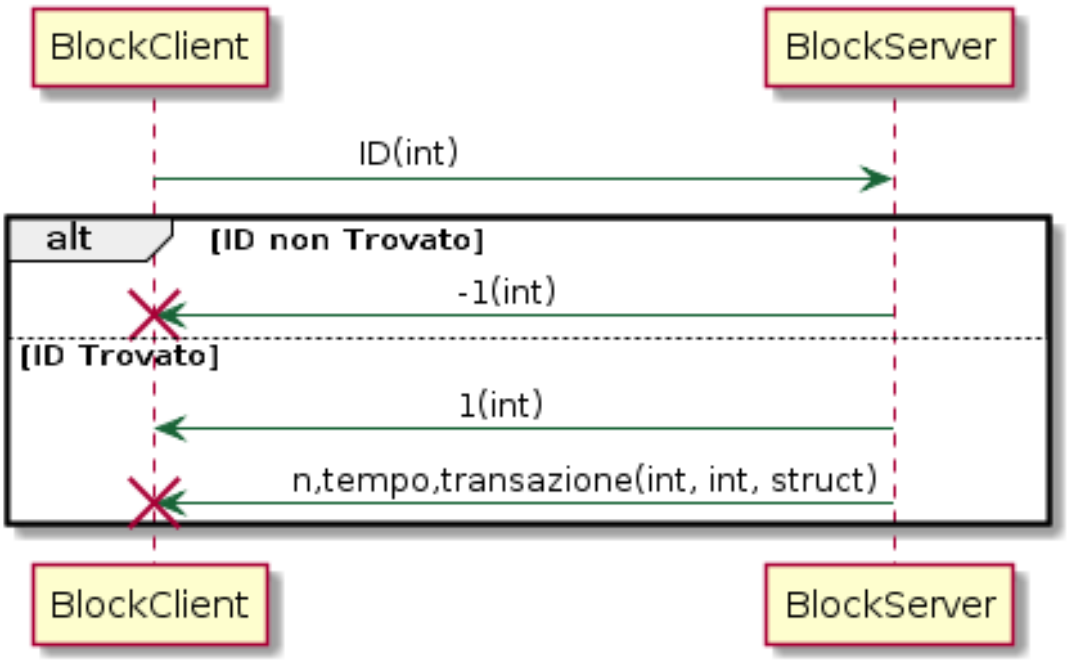
Descrizione scelta uno

Per quanto riguarda il protocollo applicazione per l'analisi della blockchain , nel caso in cui il *BlockClient* richieda di visualizzare le ultime N transazioni (Scelta = 1 nello schema [pagina|6]), allora le comunicazioni tra le due controparti avverranno come indicato nello schema sovrastante.

Il *BlockClient* richiede al *BlockServer* di inviargli gli ultimi N blocchi.

Se il *BlockServer* in quell'istante è in possesso dei blocchi richiesti, allora per N volte invierà i blocchi al *BlockClient*, altrimenti invierà al *BlockClient* un interno negativo (-1) che sta ad indicare che i blocchi richiesti in quell'istante non sono presenti nella blockchain.

Schema Scelta Due

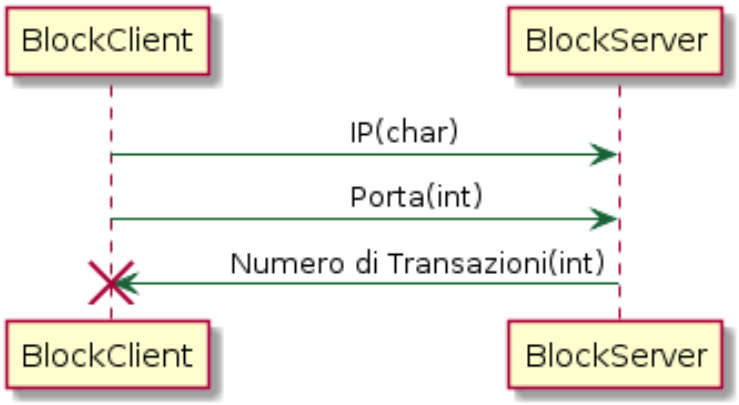


Descrizione scelta due

Il *BlockClient* quando vuole visualizzare una specifica transazione, richiederà al *BlockServer* il blocco con uno specifico ID.

A sua volta il *BlockServer* andrà a controllare se il blocco con l'ID richiesto del *BlockClient* esiste nella sua blockchain in quel determinato istante, se non è presente viene inviato -1 , altrimenti viene inviato 1 per indicare che il blocco esiste, dopodichè viene inviato il blocco richiesto.

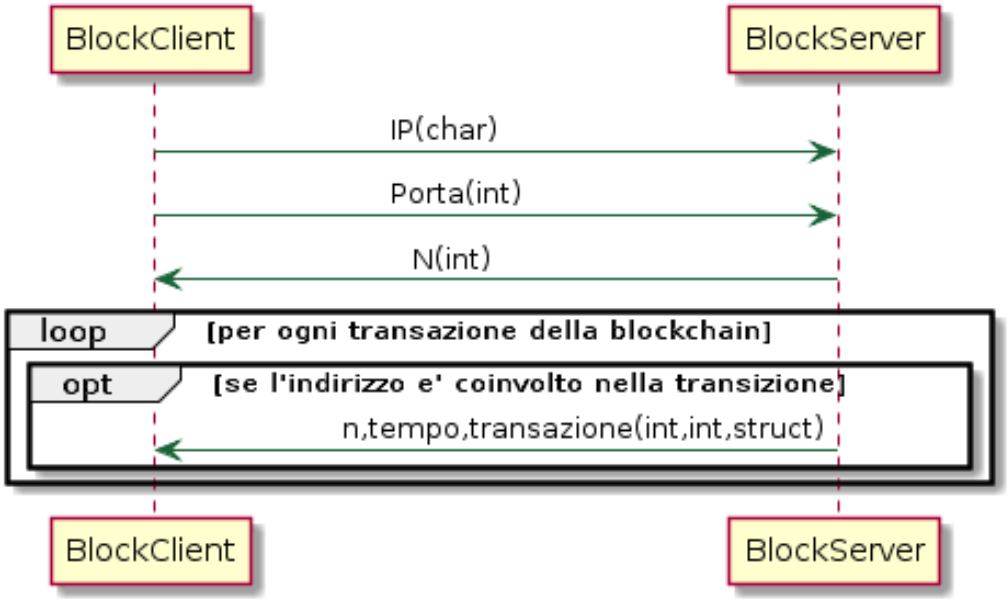
Schema Scelta quattro



Descrizione scelta quattro

Nel caso in cui il *BlockClient* richieda di conoscere il numero di transazioni in cui è coinvolto un determinato indirizzo (IP:Porta), il *BlockServer* gli invierà quante volte in quell'istante, l'indirizzo desiderato è presente nelle transazioni, nel caso in cui l'indirizzo non compare in nessun blocco, allora il Numero di Transazioni da inviare al *BlockClient* sarà pari a 0.

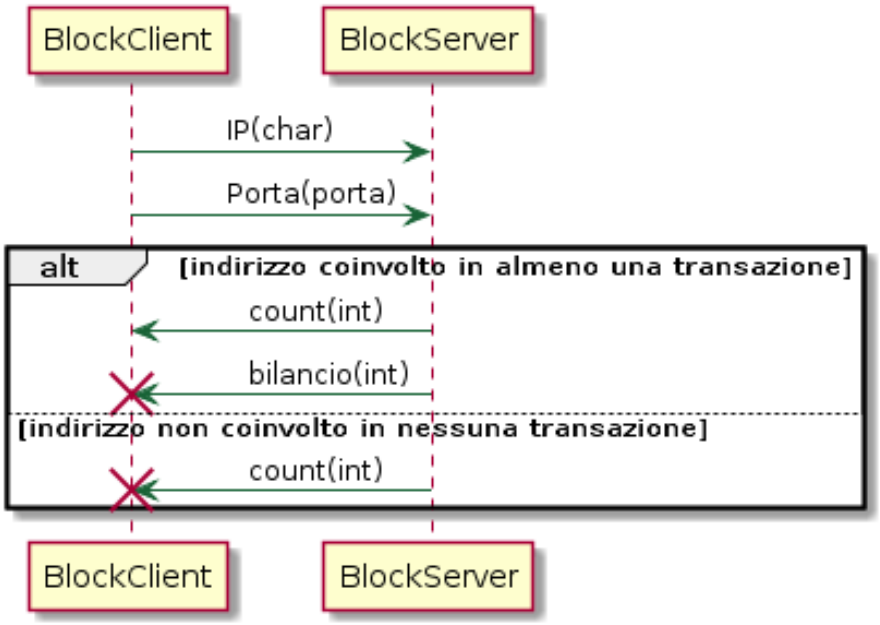
Schema Scelta Cinque



Descrizione scelta cinque

Nel caso in cui il *BlockClient* richieda di conoscere tutti i dati dei blocchi in cui è coinvolto un determinato indirizzo (IP:Porta), il *BlockServer* cercherà in quante transazioni è coinvolto quel determinato indirizzo in quell'istante, dopodiché invierà al *BlockClient* il numero di volte in cui l'indirizzo è presente, per far si che sia il *BlockClient* che il *BlockServer* comunichino lo stesso numero di volte per trasferirsi i blocchi, infine per ogni blocco il *BlockServer* invierà al *BlockClient* i loro dati.

Schema Scelta Sei



Descrizione scelta sei

Nel caso in cui il *BlockClient* richieda di conoscere il bilancio in cui è coinvolto un determinato indirizzo (IP:Porta), il *BlockServer* calcolerà il bilancio in base alla presenza dell'indirizzo nelle transazioni come mittente o destinatario, dopodiché se il *BlockServer* ha trovato delle transazioni, invierà al *BlockClient* quante volte ha trovato l'indirizzo e poi gli invia il bilancio, altrimenti invierà al *BlockClient* che l'indirizzo è stato trovato 0 volte.

Dettagli implementativi del client

Prima di analizzare le scelte implementative effettuate con il linguaggio di programmazione C, è di fondamentale importanza introdurre la struttura dei blocchi e il loro contenuto:

```
1.struct Blocco
2.{
3.    int n;
4.    int tempo;
5.    struct Transazione ts;
6.    struct Blocco *next;
7.} ;
```

n: Rappresenta l'identificativo progressivo dei blocchi, che in questo progetto parte da 0 con il blocco **genesi**.

tempo: Rappresenta un tempo randomico compreso tra 5 e 15 secondi che il *NodoN* deve attendere prima di inserire un nuovo blocco nella blockchain.

next: Rappresenta un puntatore al successivo blocco, considerando che la blockchain viene rappresentata come una lista.

La transazione ts è a sua volta una struttura così definita:

```
1.struct Transazione
2.{
3.    char ipMittente[16];
4.    int portaMittente;
5.    int credito;
6.    char ipDestinatario[16];
7.    int portaDestinatario;
8.    int numRandom;

};
```

Una nota particolare va fatta sulla scrittura nel file e sulle socket, poiché ci si avvale di una struttura temporanea per evitare di scrivere dati superflui come il campo “next” della struttura blocco:

```
struct temp
{
    int n;
    int tempo;
    struct Transazione ts;
};
```

L’interfaccia client del *BlockServer* ha lo scopo di richiedere e ricevere i blocchi dal *NodoN* il cui protocollo applicazione è stato introdotto a [pagina 6], la scelta implementativa che è stata adoperata in questo progetto si avvale di un thread “**ottieniNodi**” creato dal *BlockServer*.

Si è scelto di utilizzare i thread per garantire la concorrenza sulle varie azioni che può effettuare il *BlockServer*, un’alternativa potrebbe essere l’implementazione di un I/O Multiplexing con l’ausilio della select. La differenza sostanziale sta nel fatto che con il thread abbiamo piena autonomia delle azioni ed esse sono indipendenti l’una dall’altra, eccezion fatta per le sincronizzazioni tra i thread su alcune sezioni critiche, mentre con l’I/O Multiplexing si delega al kernel il monitoraggio dei canali di comunicazione pronti ad essere utilizzati, dopodiché se il kernel sblocca il processo dalla select, bisognerebbe controllare sequenzialmente quale di questi canali risulta essere disponibile.

La funzione di **ottieniNodi** è quella di connettersi al *NodoN*, inviargli l’identificativo dell’ultimo blocco della sua blockchain, poiché si prevede che il *NodoN* gli risponde sempre inviandogli il successivo blocco utile. In seguito, viene illustrata la porzione di codice che riguarda la comunicazione del thread *BlockServer* con il *BlockClient*.

```
1.  printf("BLOCKSERVER: Chiedo al NodoN i blocchi dall'indice: %d\n", numBl+1);
2.     FullWrite(socket, &numBl, sizeof(int));
3.
4.     while( FullRead(socket, &t, sizeof(struct temp)) != -1 )
5.     {
6.         if (t.n == size+1)
7.         {
8.             pthread_mutex_lock(&mutex);
9.             inserimentoCoda(t, genesi);
10.            size++;
11.            write(file, &t, sizeof(struct temp));
12.            printf("THREAD BLOCKSERVER: Blocco ricevuto ed inserito: %d.\n\n", t.n);
13.            pthread_mutex_unlock(&mutex);
14.
15.            printf("n = %d\ntempo = %d\nIp Destinata-
    rio: %s\t Porta: %d\n", t.n,t.tempo, t.ts.ipDestinatario, t.ts.portaDestinatario);
16.            printf("Ip Mittente: %s\t Porta Mittente: %d\nCredito: %d\nNumero Rando-
    mico: %d\n\n\n\n", t.ts.ipMittente, t.ts.portaMittente, t.ts.credito, t.ts.numRan-
    dom);
17.        }
18.
19.        FullWrite(socket, &check, sizeof(int));
20.    }
```

In questa porzione di codice si è scelto di creare una sezione critica, a cui si accede con un mutex, questa implementazione è stata utilizzata per soddisfare la legge di Bernstein sulla variabile globale size (che viene modificata da questo thread e letta da un altro thread), per garantire che il blocco venga considerato appartenente alla blockchain solo quando esso viene scritto anche in modo permanente sul file locale, e infine in modo tale da garantire che se in futuro altri thread (**gestoreClient**) effettuassero delle analisi sulla blockchain in mutua esclusione con questo mutex, allora nel frattempo questo thread **ottieniNodi** dovrà attendere che l'analisi venga completata prima di aggiornare la blockchain.

Una particolare nota sulla gestione di questo thread **ottieniNodi** riguarda l'utilizzo dei segnali e del gestore dei segnali, nel caso in cui il *NodoN* si disconnette da questa comunicazione. In particolar modo, quando il thread **ottieniNodi** si rende conto che la connessione è stata terminata dal *NodoN*, genererà un segnale e terminerà. Il segnale verrà catturato dal thread master, il quale manderà in esecuzione un opportuno gestore dei segnali che non dovrà far altro che effettuare una join per il thread appena

terminato, e provvederà a lanciare un nuovo thread **ottieniNodi** che proverà a stabilire una nuova connessione con il *NodoN*. Se il *NodoN* ancora non si è messo in ascolto, è stato previsto che il thread proverà sempre a riconnettersi ogni 10 secondi [per maggiori dettagli consultare *blockserver*.c].

Dettagli implementativi del server

Il server *NodoN* ha il compito di mettersi in ascolto di una connessione da parte di un *BlockServer*, nel frattempo il *NodoN* avrà un thread che genererà blocchi da aggiungere alla sua blockchain e al file.

Il *NodoN* riceve l'ultimo indice del blocco posseduto dal *BlockServer*, controlla per prima cosa se nella sua blockchain è presente questo blocco, altrimenti chiude la connessione.

Se il blocco è presente significa che il *NodoN* deve determinare:

1. Se l'indice è compreso negli indici dei blocchi presenti nel file.
2. Se l'indice è uguale all'indice dell'ultimo blocco posseduto dal *NodoN*
3. Nessuno dei casi precedenti

Caso 1.

Se l'indice corrisponde ad un blocco presente nel file, *NodoN* preleverà questo blocco e attenderà il tempo randomico del blocco, poiché considerando che era stato generato in una precedente esecuzione del *NodoN*, dato che in questa nuova esecuzione non ha atteso il tempo randomico, questo comportamento viene simulato prima di inviarlo al *BlockServer*.

Caso 2.

Se l'indice è uguale all'indice dell'ultimo blocco posseduto dal *NodoN* significa che il *BlockServer* e il *NodoN* hanno una stessa copia della blockchain. Quindi il *NodoN* deve attendere su un semaforo in attesa di essere risvegliato dal suo thread **produci,** che lo sbloccherà quando ci sarà un blocco prodotto che potrà essere inviato.

Il controllo dell'indice viene effettuato in mutua esclusione per garantire la correttezza delle leggi di Bernstein sulla variabile condivisa size.

Caso 3.

Se non ci troviamo in nessuno dei casi precedenti, vuol dire che nell'unico caso rimasto, cioè il caso in cui il *NodoN* in questa esecuzione ha già generato dei blocchi che il *BlockServer* non possiede ancora, allora in questo caso il *NodoN* potrà inviare direttamente i blocchi al *BlockServer* poiché il tempo randomico già sarà stato atteso dal suo thread **produci**.

In seguito, si mostrerà l'implementazione in C di ciò che è stato appena descritto sul *NodoN*:

```
1.  if(indice < numBlocchi)
2.              {
3.                  bl = bl->next;
4.
5.                  sleep(bl->tempo);
6.              }
7.              else
8.              {
9.                  pthread_mutex_lock(&mutex);
10.                 if(indice == size)
11.                 {
12.                     bloccato = 1;
13.                     pthread_mutex_unlock(&mutex);
14.                     sem_wait(disp);
15.                 }
16.                 else
17.                     pthread_mutex_unlock(&mutex);
18.
19.                 bl = bl->next;
20.             }
21.             t.n = bl->n;
22.             t.tempo = bl->tempo;
23.             t.ts = bl->ts;
24.
25.             FullWrite(conn_fd, &t, sizeof(struct temp));
26.             printf("NODON: blocco inviato numero: %d\ttempo: %d\n\n", t.n, t.tempo);
```

Per maggiori dettagli far riferimento al file [*nodon*.c]

Per quanto riguarda l'implementazione del lato server del *BlockServer* si vuole sottolineare che i thread **gestoreClient** verranno utilizzati sempre come thread distaccati, in modo da terminare definitivamente i thread dal momento in cui essi eseguono una pthread_exit, e non obbligare il thread master a doversi ricongiunge con i thread quando essi terminano, questo perché il thread master essendo in un ciclo infinito non può restare in attesa su una join per ottenere lo stato di terminazione di ogni thread **gestoreClient**, e per evitare che il thread **gestoreClient** resti in uno stato zombie indefinito si e scelto di distaccarlo.

Il resto delle funzionalità della parte server del *BlockServer* si attengono ai protocolli applicazione descritti in precedenza, cioè quando un *BlockClient* richiede una determinata scelta per l'analisi della blockchain, il *BlockServer* effettuerà la comunicazione nel modo opportuno come definito dai protocolli applicazione. Per maggiori dettagli fare riferimento al file [*BlockServer*.c] .

Dettagli implementativi per la generazione dei blocchi

Il *NodoN* appena va in esecuzione , lancia il thread **produci** che si occupa di aprire il file, per un tempo indefinito si occuperà di generare dei blocchi che verranno inseriti sia in coda alla blockchain che sul file, il seguente diagramma di sequenza ne descrive il funzionamento:

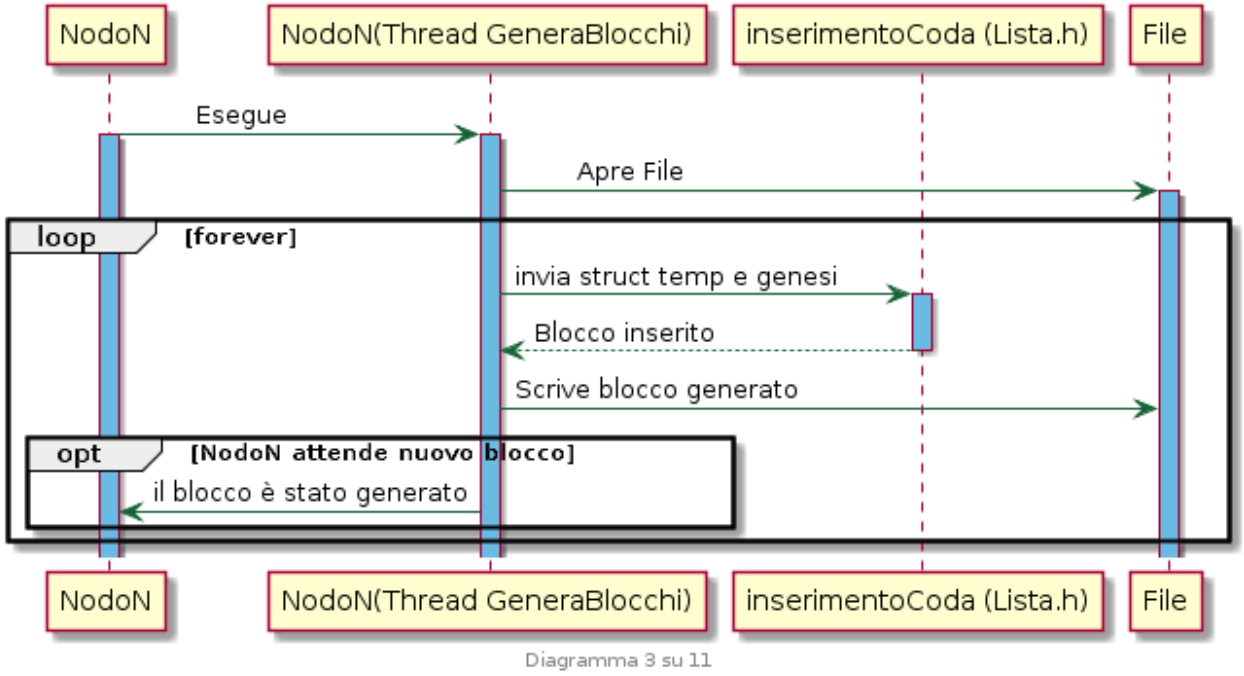


Diagramma 3 su 11

Si precisa che lo schema sovrastante non descrive un protocollo applicazione tra più entità della rete , ma bensì viene utilizzato per mostrare il funzionamento del thread che produce i blocchi.

A tal proposito, il thread produci garantisce che nella blockchain uno stesso indirizzo (IP:Porta) possa essere coinvolto in almeno 2 transazioni, questa scelta è stata fatta per rendere più complessa l'analisi dei blocchi, poiché ad esempio il servizio di visualizzare il bilancio delle transazioni in cui è coinvolto uno specifico indirizzo, si può apprezzare maggiormente se l'indirizzo desiderato venisse coinvolto in più transazioni.

La seguente porzione di codice illustra quanto appena descritto:

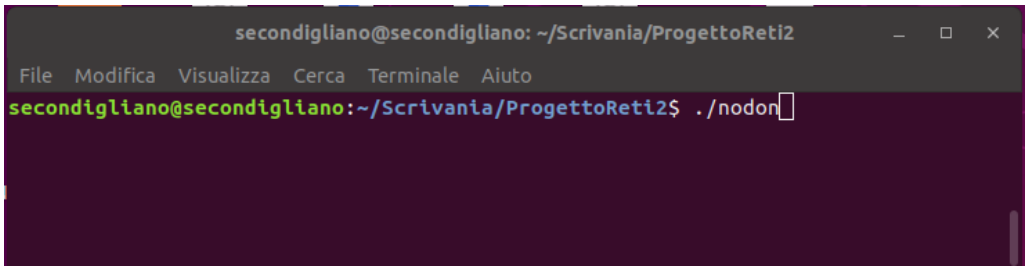
```
1.  while(1)
2.     {
3.         t.n = size+1;
4.         t.tempo = 5 + rand()%11;
5.
6.         if( (time(NULL)-temp) > 20 )
7.         {
8.             temp=time(NULL);
9.             if( (rand()%200) < 100)
10.            {
11.                snprintf(t.ts.ipMit-
    tente, 16, "%d.%d.%d.%d", rand()%256, rand()%256, rand()%256, rand()%256);
12.                t.ts.portaMittente = 1024 + rand()%64512;
13.            }
14.            else
15.            {
16.                snprintf(t.ts.ipDestinata-
    rio, 16, "%d.%d.%d.%d", rand()%256, rand()%256, rand()%256, rand()%256);
17.                t.ts.portaDestinatario = 1024 + rand()%64512;
18.            }
19.     }
20.
21.        t.ts.credito = 1 + rand()%1000000;
22.        t.ts.numRandom = 1 + rand()%999999;
23.  ...
```

Da come si evince dal controllo effettuato a riga 6, solo ogni 20 secondi i blocchi cambiano o l'indirizzo mittente o l'indirizzo destinatario, questa scelta è stata adoperata in modo da garantire che su almeno 2 blocchi vengono coinvolti gli stessi indirizzi (dato che dalla generazione di un blocco all'altro possono passare max 15 sec, di conseguenza in 20 sec vengono generati 2 blocchi con gli stessi indirizzi).

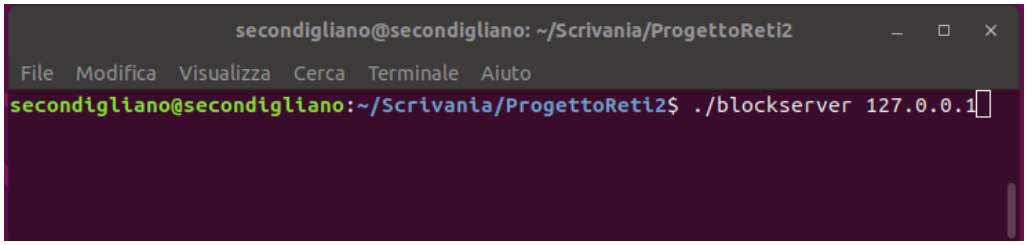
Manuale Utente

Per la compilazione dei sorgenti bisogna eseguire lo [script.sh] , che sulla shell viene effettuato digitando nella directory in cui è contenuto lo script, il comando: ./script.sh

Per una corretta esecuzione, viene consigliato di eseguire prima il *NodoN*



Poi eseguire il *BlockServer* specificando come argomento l'indirizzo IP di dove si sta eseguendo il *NodoN*:



E in fine il *BlockClient* specificando come argomento l'indirizzo IP di dove si sta eseguendo il *BlockServer*:

N.B. per come sono stati implementati il *NodoN* e il *BlockServer*, in realtà potrebbero anche essere eseguiti in un ordine diverso, garantendo comunque un corretto funzionamento, invece il *BlockClient* senza il *BlockServer* già attivo non potrà essere utile in alcun modo.