Università degli Studi di Parma

Dipartimento di Ingegneria dell'Informazione

Anno Accademico 2013-2014

Relazione su Algoritmi di Network Coding in sistemi P2P

Esame di Sistemi di Elaborazione del Prof. Michele Amoretti.

Autore - Leonardo Rossi

Contents

1 Introduzione

Il seguente progetto ha lo scopo di implementare una rete p2p per la memorizzazione distribuita di risorse. A tal riguardo sono state sfruttate le caratteristiche degli algoritmi di Network Coding per garantire all'interno di questa rete le seguenti caratteristiche:

High Availability - Alta disponibilità della risorsa. La risorsa deve poter essere acquisibile dalla rete con il livello di servizio richiesto.

Long-Term Durability - Definendo una tempo di vita della risorsa, la rete deve garantire una durata minima della disponibilità nel lungo termine maggiore del suo tempo di vita.

Fault Tollerance - A discapito di possibili disconnessioni o guasti di una parte dei peer adibiti alla memorizzazione dei frammenti della risorsa codificata, la rete deve risultare comunque affidabile.

Per prima cosa è stata creata la libreria GFJama [2] per manipolare matrici ed eseguire operazioni nella matematica dei campi finiti.

Successivamente, è stato implementato l'algoritmo di Network Coding [4] per generare i vari frammenti da memorizzare sui peer della rete. A tal riguardo, si è formalizzato le classi per approcciarsi alle risorse [3] e si è definito come deve essere composto un frammento. Di conseguenza, si è definito i metodi per custodire i frammenti stessi.

Infine, si è costruito la rete p2p [5] attraverso l'uso della libreria Sip2Peer, atta a gestire le funzionalità richieste. Si è definito il Peer generico che partecipa alla rete e tutti i comportamenti, chiamati Behavior, che possono avere. I comportamenti definiscono il tipo di Peer a cui si fa riferimento. Sono stati identificati quattro tipi diversi, ognuno con compiti distinti: Bootstrap Peer, Client Peer, Storage Server Peer e NetworkAdmin Peer.

Per acquisire le funzionalità associate a uno dei quattro tipi di Peer è sufficiente l'attivazione delle Behavior necessarie. In ogni momento è possibile attivarle e disattivarle. Non presentando nessun comportamento auto-escludente, è possibile che un Peer appartenga a due o più tipi diversi nello stesso momento.

2 Libreria GFJama

Gli algoritmi di Network Coding richiedono di eseguire pesanti operazioni matriciali su campi finiti. Per questo motivo si è deciso di prendere spunto dalla libreria JAMA [2], che implementa le operazioni richieste su numeri a virgola mobile, e adattare le varie classi in modo da renderle compatibili con l'aritmetica dei campi finiti.

La libreria risultante si compone delle seguenti classi:

GaloisField Classe che realizza le operazioni di aritmetica base sui campi finiti.

GFMatrix Classe che implementa una matrice di valori e offre i metodi per svolgere le operazioni necessarie sulla matrice stessa.

RandomGFMatrix Classe che estende GFMatrix per generare una matrice riempita di valori casuali.

GFLUDecomposition Questa classe implementa la decomposizione LU per la risoluzione di sistemi di equazioni lineari, rappresentati da matrici quadrate $N \times N$.

Per via della sua natura, si è deciso di confinare queste classi in un progetto separato. Questo potrebbe risultare utile per utilizzare la seguente libreria in altri progetti.

2.1 GaloisField Class

Questa classe si occupa di implementare tutte le operazioni base tra due numeri in un Campo Finito, detto anche *Galois Field*.

I campi finiti svolgono un importante ruolo nella teoria dei numeri, geometria algebrica, teoria di Galois e in diversi algoritmi tra cui l'AES, la crittografia ellittica e nella teoria dei codici.

L'attenzione si concentra per lo più su campi $GF(2^n)$, poichè presentano alcuni vantaggi che ne garantiscono una ottima efficienza implementativa:

- Permettono di rappresentare univocamente ogni polinomio del campo in n bit.
- La somma può essere eseguita efficientemente come semplice XOR bit-a-bit.
- La moltiplicazione per piccoli coefficienti richiedono al massimo uno shift a sinistra e uno XOR.

Nel nostro caso, si è scelta una strada differente: accelerare l'aritmetica attraverso una tabella del logaritmo di Zech per quel campo specifico.

La classe implementa i seguenti campi: $GF(2^3)$, $GF(2^8)$, $GF(2^{16})$.

Ogni operazione viene eseguita su *char*. In java il *char* è rappresentato come un dato a 16 bit, che può rappresentare valori compresi tra 0 e 2^{16}. Quindi, nel campo $GF(2^{16})$ possono venire eseguite operazioni al massimo tra due valori a 16 bit. Nel nostro caso, è più che sufficiente, in quanto il nostro intento è di leggere 8 bit alla volta dal file di origine e successivamente eseguire le operazioni nel campo di Galois con valori casuali di 8 bit.

Eseguendo operazioni su numeri a 8 bit sarebbe stato sufficiente lavorare nel campo $GF(2^8)$.

Per estendere ad altri campi, sarebbe sufficiente scrivere dentro la directory GaloisField il corrispettivo file contenente la tabella. Ad es: per implementare la $GF(2^4)$, generare e salvare dentro GaloisField il file ZechLogGF16.txt

In caso si volesse usare un campo più grande di $GF(2^{16})$, sarebbe necessario abbandonare l'uso di *char* e usare il tipo di dato *BigInteger* messo a disposizione da Java all'interno del *package java.math*.

Vediamo ora un elenco dei metodi più importanti messi a disposizione:

public GaloisField(byte n) Costruttore della classe. Prende in ingresso il valore esponente n che definisce il campo $GF(2^n)$ (nel nostro caso: 3, 8 o 16).

public char sum(char a, char b) Restituisce la somma nel campo dei numeri finiti: $a + b$.

public char minus(char a, char b) Restituisce la sottrazione nel campo dei numeri finiti: a - b.

public char product(char a, char b) Restituisce il prodotto nel campo dei numeri finiti: $a \times b$.

public char divide(char a, char b) Restituisce la divisione nel campo dei numeri finiti: $a \setminus b$.

public static char getZero() Restituisce il valore zero nel campo dei numeri finiti.

public static char getOne() Restituisce il valore uno nel campo dei numeri finiti.

public static boolean isZero(char a) Restituisce vero se $a == Zero$ nel campo dei numeri finiti.

public static byte getN() Restituisce l'esponente n del campo $GF(2^n)$

2.2 GFMatrix Class

Questa classe implementa una matrice bidimensionale $m \times n$ contenente valori appartenenti a un campo di numeri finiti. Di conseguenza, tutte le operazioni svolte sopra questi valori sono affidate a un oggetto della classe *GaloisField*.

In particolare, le operazioni più importanti tra matrici sono: la trasposizione, la somma, sottrazione, moltiplicazione e divisione tra matrici, moltiplicazione con uno scalare e soluzione di una matrice $m \times m$, il calcolo dell'inversa il calcolo del determinante, la generazione di una matrice identità $m \times n$.

Vediamo ora nel dettaglio l'elenco delle operazioni:

public GFMatrix(int m, int n, GaloisField gf) Inizializza una matrice $m \times n$ e imposta il campo di Galois.

public GFMatrix(char[][] A, GaloisField gf) Inizializza la matrice con valori già esistenti e imposta il campo di Galois.

public GFMatrix copy() Copia una matrice.

public char[][] getArray() Ottiene la matrice di valori originale.

public char[][] getArrayCopy() Come la precedente, ma in questo caso restituisce una copia della matrice.

public int getRowDimension() Ottieni il numero di righe della matrice.

public int getColumnDimension() Ottieni il numero di colonne della matrice.

public GFMatrix transpose() Restituisce la matrice trasposta.

public GFMatrix uminus() Restituisce $-A$.

public GFMatrix plus(GFMatrix B) Restituisce $C = A + B$

public GFMatrix plusEquals(GFMatrix B) Restituisce $A = A + B$

public GFMatrix minus(GFMatrix B) Restituisce $C = A - B$

public GFMatrix minusEquals(GFMatrix B) Restituisce $A = A-$

public GFMatrix arrayTimes(GFMatrix B) Restituisce $C = A. \times B$

public GFMatrix arrayTimesEquals(GFMatrix B) Restituisce $A = A. \times B$

public GFMatrix arrayRightDivide(GFMatrix B) Restituisce $C = A. \setminus B$

public GFMatrix arrayRightDivideEquals(GFMatrix B) Restituisce $A = A. \setminus B$

public GFMatrix arrayLeftDivide(GFMatrix B) Restituisce $C = A.\ B$

public GFMatrix arrayLeftDivideEquals(GFMatrix B) Restituisce $A = A. \setminus B$

public GFMatrix times(char s) Restituisce $C = s \times A$

public GFMatrix timesEquals(char s) Restituisce $A = s \times A$

public GFMatrix times(GFMatrix B) Restituisce $A \times B$

public GFMatrix solve(GFMatrix B) Risolve $A \times X = B$

public GFMatrix solveTranspose(GFMatrix B) Risolve $X \times A = B$, che è anche $A' \times X' = B'$

public GFMatrix inverse() Calcola la matrice inversa.

public char det() Calcola il determinante.

public char trace() Esegue la somma degli elementi sulla diagonale.

public static GFMatrix identity(int m, int n, GaloisField gf) Genera una matrice identità $m \times n$.

public char[] getColumnCopy(int index) Ottieni una copia della colonna i-esima.

2.3 GFLUDecomposition Class

La seguente classe serve per risolvere il sistema di equazioni lineari, utile per calcolare l'inversa di una matrice $m \times m$.

Vediamo l'elenco dei metodi messi a disposizione:

public boolean isNonsingular() Restituisce vero se la matrice è non singolare.

public char det() Calcola il determinante della matrice.

public GFMatrix solve(GFMatrix B) Risolve il sistema di equazioni lineari $A \times X = B$

2.4 GFMatrixException

Questa classe estende la classe Exception di Java ed è utilizzata nei casi in cui si cercasse di eseguire un'azione errata sulla matrice.

2.5 UnitTest per GFJama

Per la seguente libreria sono stati sviluppati delle classi di test con JUnit. I test coprono la classe GaloisField, GFMatrix e RandomGFMatrix.

3 Gestione delle risorse

Siccome i Peer lavorano sulle risorse, manipolandole in diversi modi, sono state costruite diverse classi per gestirle nel modo opportuno, a seconda della funzionalità richiesta. Di seguito vengono presentate le varie classi e interfacce definite.

3.1 MediaResource Class

Questa classe incapsula una singola risorsa su cui verranno eseguite operazioni di codifica e decodifica tramite un Coding Engine [4]. Tramite il costruttore viene inoltre impostato la grandezza di ogni singolo frammento e il campo di Galois su cui eseguire tutte le operazioni matematiche.

Sono stati implementati anche altri metodi. Vediamo di seguito un breve elenco:

- **public int getNumberOfFragments()** - Per ricavare il numero di frammenti in cui verrà divisa la risorsa. Questo valore dipende dalla grandezza dei frammenti e coincide con il numero di frammenti codificati minimo da recuperare per poter ricostruire la risorsa.
- **public byte[] getResourceKey()** - Ritorna un vettore identificativo univoco della risorsa. La classe calcola lo *SHA-256* sulla risorsa. Potrebbe essere utilizzato per controllare l'integrità della risorsa.
- **public static void save(File file, char[][] data, long totalLength)** - Funzione utile per salvare una risorsa che è appena stata ricostruita.
- **public GFMatrix loadTransposedPiece(int index)** - Ritorna una matrice contenente, per ogni frammento $a_{i-esimo}$, la colonna index-esima di dati, inserendoli in una matrice $1 \times m$. Per intenderci, se prendiamo l'esempio in Figura [1] e imponiamo index uguale a 1, il metodo ritorna una matrice 1×2 contenente i valori: [140, 44].

3.1.1 EncodedFragment Class

Questa classe rappresenta un frammento codificato tramite il Coding Engine [4]. La classe si compone di un vettore di char contenente i dati codificati e un EncodedFragmentHeader, contenente tutti le informazioni aggiuntive necessarie.

3.1.2 EncodedFragmentHeader Class

Questa classe rappresenta l'intestazione di un frammento codificato. Contiene le seguenti informazioni:

- **resourceKey** - Identificativo univoco della risorsa. Utile per risalire a quale risorsa appartiene il frammento.
- **codingVector** - Colonna della matrice G usata per la codifica dei dati. Sono dati necessari per poter successivamente ricostruire la risorsa. Vedi sezione sui Coding Engine [4].

- **fragmentSize** - Grandezza del frammento.
- **galoisField** - Campo di Galois utilizzato.
- **fragmentKey** - Identificativo univoco del frammento. É stato calcolato come *SHA-256*. Sarebbe possibile usare questa chiave per potersi assicurare dell'integrità del frammento.

3.2 StorageFragments Interface

L'interfaccia rappresenta un gestore di frammenti. Ogni implementazione si dovrebbe occupare di memorizzare e restituire frammenti. L'interfaccia viene ad esempio usata all'interno della Behavior *StorageFragmentsServerBehavior* [5.2.8].

3.2.1 MapUniqueFragmentStorageFragments Class

Questa classe rappresenta una delle possibili implementazioni dell'interfaccia *StorageFragments*. Definisce un oggetto di tipo *Map* per conservare tutte i frammenti.

Possiede una limitazione: può memorizzare un singolo frammento per ogni risorsa. Difficilmente utilizzabile in un caso generale, ma sufficiente per una versione semplificata di *Storage Server Peer*, in cui a ogni nodo può venir affidato al massimo un frammento della stessa risorsa.

3.3 UnitTest su MediaResource

Sono stati creati alcuni test per verificare la correttezza dell'implementazione della classe *MediaResource* [3.1].

4 Coding Engine

Il motivo per cui si fa riferimento ad algoritmi di codifica di una risorsa nell'ambito della memorizzazione distribuita è per mettere in atto una strategia efficiente ed efficace per la distribuzione di risorse tra i peer presenti sulla rete. Le caratteristiche che si vogliono garantire sono le seguenti:

High Availability - Alta disponibilità della risorsa. La risorsa deve poter essere acquisibile dalla rete con il livello di servizio richiesto.

Long-Term Durability - Definendo una tempo di vita della risorsa, la rete deve garantire una durata minima della disponibilità nel lungo termine maggiore del suo tempo di vita.

Fault Tollerance - A discapito di possibili disconnessioni o guasti di una parte dei peer adibiti alla memorizzazione dei frammenti della risorsa codificata, la rete deve risultare comunque affidabile.

Siccome esistono molteplici tipi di codifica e decodifica di risorse, è stata creata una interfaccia generica CodingEngine che definisse le operazioni che il sistema mette a disposizione e su che oggetti deve eseguire le operazioni. Per far riferimento a una risorsa generica è stata utilizzata la classe MediaResource [3.1] che la ingloba, fornendone accesso e aggiungendo alcune informazioni specifiche.

L'interfaccia presenta due metodi distinti:

public List<EncodedFragment> encode(MediaResource ms) Questo metodo esegue le operazioni di codifica della risorsa e restituisce una lista di frammenti pronti per essere pubblicati.

public char[][] decode(List<EncodedFragment> fragments) Questo metodo esegue le operazioni di decodifica della risorsa attraverso la lettura di un sottoinsieme di frammenti totali e restituisce una matrice con i valori risultanti dalla decodifica. La classe MediaResource mette a disposizione un metodo per salvare la risorsa attraverso questa matrice.

4.1 Network Coding Engine

I codici di rete possiedono una interessante caratteristica: l'indipendenza dei frammenti generati. Per ricostruire la risorsa sono necessari un sotto-insieme minimo qualsiasi di frammenti codificati.

Questo permette ipoteticamente di distribuire in una rete le richieste nel modo più uniforme possibile su tutti i possessori di frammenti.

Per contro, si inserisce un *overhead* di potenza di calcolo e memoria necessari per la codifica e decodifica.

Negli articoli [5] e [4] viene comparato l'Erasure Coding con la strategia di replicazione dell'intera risorsa su più peer, mettendo in evidenza vari punti di forza e debolezza.

Nell'articolo [1] vengono elencati diversi sistemi che ne fanno uso, analizzando e simulando la disponibilità della risorsa. Proponendo, in alternativa al classico *Erasure Code*, un *Randomized Network Coding*.

La classe NetworkCodingEngine rappresenta l'implementazione dell'algoritmo di Network Coding che genera in modo casuale la matrice di coefficienti usati per la codifica.

La classe prende in input il *redundancy rate*, ossia il fattore di espansione della risorsa.

$$\text{redundancy rate} = \frac{n}{m} \geq 1$$

Calcolato come rapporto tra il numero di frammenti generati dalla codifica (n) e il numero di frammenti in cui la risorsa originale è stata divisa (m). Ovviamente, il valore deve essere maggiore o uguale a uno. Altrimenti ne risulterebbe un sistema non risolvibile.

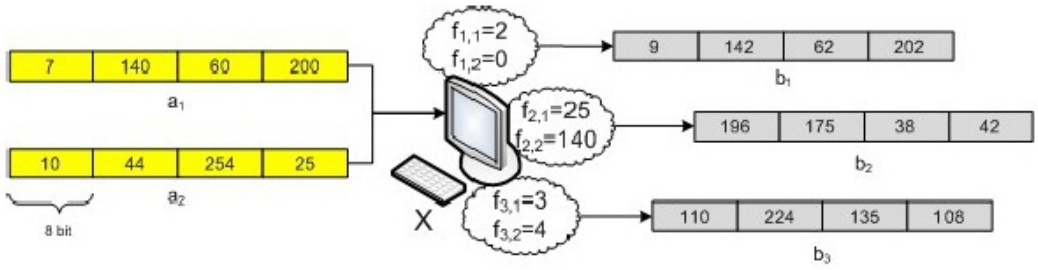

Figure 1: Un semplice esempio di Linear Coding su $GF(2^8)$ in cui $m = 2$ e $n = 3$. Gli elementi del campo sono rappresentati con la notazione esponenziale. Per esempio: $\alpha^7 \times \alpha^{25} + \alpha^{10} \times \alpha^{140} = \alpha^{32} + \alpha^{150} = \alpha^{Z(118)+32} = \alpha^{164+32} = \alpha^{196}$.

Supponiamo si consideri il campo finito $GF(2^{16})$ e che la risorsa venga divisa in m frammenti, ciascuno grande $j \times u$ bit. Nel nostro caso, u è stato scelto uguale alla grandezza del byte, cioè 8.

Ogni byte corrisponde ad un elemento $a_i(j)$ in $GF(2^{16})$. Il valore di j può essere scelto arbitrariamente e determina il numero totale di frammenti in cui viene divisa la risorsa originale.

$$\text{m} = n^\circ \text{ frammenti} = \frac{grandezzarisorsa}{j \times u}$$

La codifica avviene eseguendo la seguente moltiplicazione tra matrici per ogni j:

$$s(j) = l(j) \times G \tag{1}$$

Dove:

- **s** = matrice $1 \times n$ contenente il risultante vettore codificato
- **G** = matrice $m \times n$ contenente i coefficienti generati casualmente
- **l** = matrice $1 \times m$ contenente un vettore dei dati originari

Nell'esempio mostrato in figura [1], la matrice $G_{2,3}$ è la seguente:

$$G_{m,n} = \begin{pmatrix} 2 & 25 & 3 \\ 0 & 140 & 4 \end{pmatrix}$$

Generando come risultato il sistema di equazioni lineari:

$$b(j) = \begin{cases} b_1(j) = 2a_1(j) + 0a_2(j) \\ b_2(j) = 25a_1(j) + 140a_2(j) \\ b_3(j) = 3a_1(j) + 4a_2(j) \end{cases} , j = 1, 2, 3, ...$$

Una volta costruiti i frammenti risultanti dalla codifica, avremo che la grandezza di ogni frammento non cambia rispetto alla grandezza dei frammenti originali, ma il numero risultante ora è pari ad n.

In ogni blocco degli n generati dalla codifica è distribuita uniformemente l'informazione. La proprietà fondamentale di questo tipo di codifica è che, sotto le dovute condizioni, sono sufficienti m qualsiasi degli n blocchi codificati per riottenere i dati originali.

Per ricostruire l è necessario invertire una matrice G' $m \times m$, sotto-matrice di G, ed eseguire la seguente operazione a livello matriciale:

$$l(j) = s'(j) \times G'^{-1} \tag{2}$$

Dove s' è una matrice $m \times m$, sotto-matrice di s.

Per fare un esempio, assumiamo di avere i blocchi b_2 e b_3. La matrice G' richiesta corrisponde alla 2° e 3° colonna della matrice originale G e s' è formata dalla 2° e 3° riga di s, cioè b_2 e b_3:

$$G'_{m,m} = \begin{pmatrix} 25 & 3 \\ 140 & 4 \end{pmatrix}$$

$$s'_{1,m}(0) = \begin{pmatrix} 196 & 110 \end{pmatrix}$$

$$s'_{1,m}(1) = \begin{pmatrix} 175 & 224 \end{pmatrix}$$

$$s'_{1,m}(2) = \begin{pmatrix} 38 & 135 \end{pmatrix}$$

$$s'_{1,m}(3) = \begin{pmatrix} 42 & 108 \end{pmatrix}$$

La condizione necessaria perchè sia possibile decodificare correttamente i blocchi è che la matrice G' sia di rango massimo. Quando ciò accade, la decodifica corrisponde alla soluzione di un sistema di equazioni lineari su campi finiti.

Una matrice G in cui tutte le sotto-matrici quadrate G' sono a rango massimo corrisponde ad un codice a Massima Distanza Separabile (MDS) e tali costruzioni combinatorie sono piuttosto difficili da ottenere.

I codici Reed-Solomon [3] realizzano tali matrici sfruttando le proprietà dei polinomi su campo finito. L'idea chiave è che qualsiasi set di m punti interpolatori è sufficiente per ricostruire i coefficienti di un polinomio di grado $m - 1$.

Nell'implementazione viene sfruttato un rilassamento di tale requisito (nearly-MDS). Ossia che, se vengono scelti in modo casuale i valori della matrice G, la probabilità può essere portata arbitrariamente vicino ad uno scegliendo una dimensione del campo finito sufficientemente grande.

I tempi richiesti all'algoritmo di codifica e di decodifica sono influenzati ovviamente dalla grandezza della risorsa da codificare; in quando cresce in modo lineare

il numero di operazioni da svolgere per codificare e decodificare ogni singolo frammento.

L'efficienza è altresì condizionata dal fattore di ridondanza e, di conseguenza, anche dal numero di frammenti in cui viene diviso il file originale. Più aumenta il numero di frammenti, a parità di fattore di ridondanza, e più la matrice G si ingrandisce. Su questa matrice sono applicate le operazioni computazionalmente più complesse, tra cui l'inversione su una sua sotto-matrice $m \times m$.

Per contro, avere un fattore di ridondanza alto consente di distribuire su più nodi i frammenti, con conseguente aumento dell'affidabilità della rete a discapito dell'aumento della banda e dello spazio totale utilizzato.

Di conseguenza, questo valore va scelto con cura come *trade-off* tra i vari aspetti.

4.2 UnitTest per NetworkCodingEngine

La classe di test *TestNetworkCodingEngine* contiene diversi metodi che provano la codifica e la successiva decodifica di una risorsa, confrontando il risultanto. La risorsa risultante dalla decodifica viene salvata in un nuovo file.

I file presi come riferimento sono sia testuali che binari (archivi tar.gz) e vengono provate diverse grandezze di frammento.

5 Rete p2p

Lo scopo finale della tesina è l'implementazione di una rete p2p in cui i peer partecipanti siano in grado di richiedere la memorizzazione distribuita e il recupero di una risorsa. La memorizzazione fa uso degli algoritmi di Network Coding: il peer che richiede di fare lo *store* distribuito, codifica la risorsa e genera una serie di frammenti. Il peer si costruisce una propria lista di peer disponibili a conservare un frammento, in numero uguale al numero di frammenti. A questo punto viene spedito il frammento affidandolo a un peer della lista. I nodi che custodiranno i frammenti affidati, rimarranno in ascolto per eventuali richieste di recupero di questi.

La rete qui descritta mette in luce l'esistenza di peer che possiedono diversi comportamenti (Behaviors) [5.2]). Viene presentato di seguito un elenco dei tipi di peer individuati attraverso le competenze richieste:

- **Bootstrap Peer**. Si occupare di realizzare un server di Boostrap per tutti gli altri peer della rete. Su questo peer è attivato il Behavior *BootstrapServerBehavior* [5.2.4]. I peer che richiedono di partecipare alla rete aggiungono il Behavior *BootstrapClientBehavior* [5.2.3].

- **Client Peer**. Salva e recupera una risorsa in modo distribuito all'interno della rete. Questo peer deve possedere tre diversi Behavior: *FillStorePeerListClientBehavior* [5.2.5] per ottenere da un NetworkAdmin Peer una lista di *Storage Server Peer* disponibili a immagazzinare i frammenti codificati. *PublishEncodedFileClientBehavior* [5.2.6] per richiedere a un Storage Server Peer di mantenere un frammento. *RetrieveEncodedFileClientBehavior* [5.2.7] per richiedere indietro i frammenti custoditi collegati a una determinata risorsa.

- **Storage Server Peer**. Mette a disposizione una parte del proprio spazio per lo *storage* e il recupero di frammenti di risorse dei Client Peer. Su questo peer è attivato il Behavior *StorageFragmentsServerBehavior* [5.2.8] che si occupa di rispondere a richieste di Client Peer di custodire frammenti e di restituirli.

- **NetworkAdmin Peer**. Si occupano di fornire ai Client Peer liste di Storage Server Peer disponibili a custodire frammenti. Su questo peer è attivato il Behavior *FillPeerListServerBehavior* [5.2.2] per mettersi in ascolto di Client Peer che richiedano liste di Storage Server Peer.

L'attivazione di Behavior sul peer permette a quest'ultimo di aggiungere un comportamento nuovo a quelli già attivati in precedenza. Ossia, gli permette di compiere diversi tipi di azioni, come inviare e/o mettersi in attesa di determinati tipi di messaggi.

Ogni peer può possedere uno o più comportamenti, a meno che ovviamente un Behavior non entri in conflitto con un altro. Questo significa che un peer può assumere uno o più ruoli all'interno della rete. In questo modo si può promuovere (o degradare) ogni peer a uno *status* diverso in base alle regole stabilite dalla rete stessa in cui questo agisce.

Mettiamo, ad esempio, di voler realizzare una rete simile a Wuala [1], in cui esistono tre tipi di nodi: supernodi, nodi di storage e client. Un nodo client può venire promosso a nodo di storage, in cambio di un impegno a garantire una certa

disponibilità online. Per promuoverlo (o degradarlo) a Storage Server Peer sarebbe sufficiente attivare (o disattivare) su di esso i Behaviors associati a questo profilo. Nel nostro caso il Behavior StorageFragmentsServerBehavior [Sezione 5.2.8].

5.1 Peer della rete

5.1.1 SimplePeer Class

Questa classe estende la classe base Peer. Definisce un Peer generico della rete in grado di acquisire funzionalità tramite la gestione delle sue Behavior [5.2].

Una volta inizializzato, esegue una semplice inizializzazione tramite la lettura da file di configurazione. Il formato del file di configurazione è il medesimo richiesto dalla classe Peer. Per cui è possibile inserire all'interno di un unico file tutte le impostazioni. Le variabili che è possibile configurare tramite file sono le seguenti:

- *req_npeer* = numero minimo richiesto di vicini di cui il peer è a conoscenza. All'atto della *Join* alla rete, risulta essere il numero di Peer richiesti al server di Bootstrap.
- *bootstrap_peer* = ip e porta del server di bootstrap.

Questi due valori non sono obbligatori e, ad esempio, non sono richiesti a un Peer il cui ruolo è di server di Bootstrap.

Di seguito vengono elencati i metodi pubblici più importanti:

- **public void joinBootstrapPeer(FullFillPeerListener listener)** - Esegue il join alla rete tramite il bootstrap server configurato. Una volta che la lista di peer vicini si è riempita, il *listener* viene avvertito che l'azione è stata portata a termine.
- **protected void onReceivedJSONMsg(JSONObject jsonMsg, Address sender)** - Scandisce tutti i Behavior configurati ed avverte ognuno di essi che è sato ricevuto un nuovo messaggio. Sarà compito dei Behavior agire di conseguenza alla ricezione di specifici messaggi.

5.1.2 BootstrapPeer Class

Siccome è molto probabile che esista almeno un server di bootstrap dedicato a questo ruolo, è stata creata una classe che estende SimplePeer che attiva in modo automatico il Behavior BootstrapServerBehavior [5.2.4]. In questo modo, il peer si mette in ascolto di eventuali richieste da parte di peer.

5.2 Behaviors

Un Behavior rappresenta un comportamento che un Peer possiede. Per spingere verso una gestione ottimale, si è deciso di far implementare la classe Runnable per poter eseguire azioni in un thread separato. Siccome non descrive un Behavior preciso, la classe è stata definita come astratta. All'atto della creazione di un oggetto di una classe che estende Behavior, viene eseguita l'azione di registrazione della Behavior all'interno del Peer per cui è stata creata.

5.2.1 FillPeerListClientBehavior

Questa Behavior permette al Peer di fare richiesta a uno o più Peer di una lista di PeerDescriptor in loro possesso.

public FillPeerListClientBehavior(SimplePeer peer, PeerListManager peerList, int peerRequired)

public FillPeerListClientBehavior(SimplePeer peer, NeighborPeerDescriptor desc, int peerRequired)

Esistono due costruttori, in cui a cambiare è solo il secondo parametro, quello che identifica i Peer a cui fare le richieste. Di seguito vengono descritti i parametri:

- *peer* - il peer associato alla Behaviors.
- *peerList / desc* - Lista di PeerDescriptor, o singolo PeerDescriptor a cui fare le richieste.
- *peerRequired* - numero di PeerDescriptor richiesti in totale.

La classe è costruita in modo da permettere, facendo l'overloading di alcuni metodi, di differenziare la lista di PeerDescriptor a cui fare le richieste, dalla lista di PeerDescriptor da riempire e restituire al termine delle operazioni. Un esempio che sfrutta questa possibilità è la Classe BootstrapClientBehavior [5.2.3].

Le richieste vengono spalmate in modo uniforme su tutti i Peer di cui sia ha conoscenza.

5.2.2 FillPeerListServerBehavior

Il seguente Behavior mette in ascolto il Peer di eventuali richieste di liste di PeerDescriptor. Ad ogni richiesta, viene estratto un sotto insieme di quelli a disposizione e inviato un messaggio PeerListMessage [5.3.2]. Dal messaggio di richiesta viene letta la quantità richiesta.

5.2.3 BootstrapClientBehavior

Questo Behavior è utilizzato dai peer per richiedere al Bootstrap Server una lista di Peer aggiornata che andranno a rappresentare il suo vicinato. Questo tipo di comportamento è utilizzato da parte di SinglePeer per eseguire la connessione alla rete tramite il bootstrap.

Possiamo parlare di comportamento "*One Shot*", ossia utile sono una volta. Una volta eseguito il suo compito, è possibile eliminarlo. Di conseguenza, una volta che si sarà ottenuta una lista di peer grande a sufficienza dal server di Bootstrap, l'oggetto esegue automaticamente l'azione di rimozione di se stesso dalla lista di behavior del Peer.

La classe estende i metodi protetti *getPeerListToAsk()* e *getPeerListToRefill()* per distinguere la lista di peer a cui chiedere, in questo caso coincide con il Boostrap Server, e la lista di peer da riempire, ossia la lista interna in cui sono memorizzati i vicini.

5.2.4 BootstrapServerBehavior

Questa Behavior aggiunge la funzionalità di Server di Bootstrap ai Peer su cui viene attivata. Estende la classe FillPeerListServerBehavior. Quindi, come la classe madre, risponderà alle richieste di liste di peer dei client.

Inoltre, estende il metodo protetto *havePeerDescriptor(NeighborPeerDescriptor neighborPeerDescriptor)* il quale viene richiamato tutte le volte che un peer fa richiesta di *Join* alla rete. In questo modo può tenere traccia dei Peer che hanno cercato di accedere alla rete e aggiungerli alla lista dei peer attualmente connessi. Questo è appunto lo scopo di questa classe. In questo modo viene alimentata la lista di peer da cui attingere alla successiva richiesta.

5.2.5 FillStorePeerListClientBehavior

Questa funzionalità viene attivata sui peer che necessitano di ottenere una lista di Storage Server Peer disponibili a conservare nuovi frammenti.

public FillStorePeerListClientBehavior(SimplePeer peer, PeerListManager peerList, int peerRequired)

Nel costruttore della classe vengono passati tre parametri:

1. *SimplePeer* - il peer associato alla Behavior
2. *PeerListManager* - elenco di peer a cui fare richieste di liste di peer (ossia, hanno attivato una Behavior FillPeerListServerBehavior [5.2.2]).
3. *integer* - numero di peer richiesti.

Una volta attivata questa funzionalità, finchè non si sono ottenuti il numero di peer richiesti, la classe continua ad inoltrare richieste.

5.2.6 PublishEncodedFileClientBehavior

Questa è una delle funzionalità richieste alla rete p2p, ossia la possibilità di pubblicare una risorsa.

L'esecuzione di questa funzionalità porta a codificare la risorsa, con conseguente generazione dei frammenti. Ogni frammento viene spedito a un Storage Server Peer diverso che si occuperà di mantenerlo nel tempo.

public PublishEncodedFileClientBehavior(SimplePeer peer, PeerListManager storePeerListManager, File file);

Analizziamo il costruttore:

1. *peer* - il peer associato alla Behavior
2. *storePeerListManager* - elenco di Storage Server Peer a cui affidare i frammenti (ossia, peer che hanno attivato una Behavior StorageFragmentsServerBehavior [5.2.8]).

3. *File* - risorsa da pubblicare sulla rete.

Il costruttore inizializza le variabili di configurazioni necessarie al motore di codifica. Dal file di configurazione del Peer vengono letti, se presenti, i seguenti parametri:

- **GF_n** - Esponente n, necessario per definire il campo di Galois (di default viene usato $GF(2^{16})$).
- **redundancyRate** - Redundacy Rate, ossia il rapporto tra il numero di frammenti generati dopo la codifica e il numero di frammenti in cui viene diviso il file originale (di default *2.5*).
- **fragmentSize** - Grandezza di ogni frammento del file originale (di default 10kb).

Successivamente, viene inizializzata una MediaResource [3.1] e il motore di coding NetworkCodingEngine [4.1].

Quando viene messa in esecuzione la Behavior, viene codificata la risorsa e generati i frammenti che verranno successivamente spediti a ogni peer presente nella lista di Storage Server Peer.

5.2.7 RetrieveEncodedFileClientBehavior

Un peer che avesse attivato questa Behavior è in grado di richiedere i frammenti associati a una risorsa ai Storage Server Peer presenti nella rete.

public RetrieveEncodedFileClientBehavior(SimplePeer peer, PeerListManager storePeerListManager, byte[] resourceKey, int numOfFragments, StorageFragments storage)

Analizziamo il costruttore:

1. *peer* - il peer associato alla Behavior
2. *storePeerListManager* - elenco di Storage Server Peer a cui richiedere i frammenti (ossia, peer che hanno attivato una Behavior StorageFragmentsServerBehavior [5.2.8]).
3. *resourceKey* - Chiave che serve per identificare in modo univoco una risorsa pubblicata sulla rete p2p.
4. *numOfFragments* - Numero di frammenti necessari per ricostruire la risorsa.
5. *storage* - Sistema di salvataggio dei frammenti. Una volta ricevuti vengono affidati a lui.

5.2.8 StorageFragmentsServerBehavior

Questa classe si occupa di attivare un nuovo comportamento: il peer si mette in ascolto di richieste da parte di Client Peer di memorizzare nuovi frammenti. In più, aspetta di ricevere richieste dei frammenti medesimi e li ritrasmette, incapsulandoli all'interno del messaggio di risposta.

public StorageFragmentsServerBehavior(SimplePeer peer, StorageFragments storage)

Analizziamo il costruttore:

1. *peer* - il peer associato alla Behavior
2. *storage* - Sistema di salvataggio e recupero dei frammenti. Si occupa di memorizzazione o estrarre i frammenti.

5.3 Messaggi

All'interno della rete p2p possono viaggiare diversi tipi di messaggi, ognuno con un significato e contenuto particolare. Di seguito vengono analizzati nel dettaglio.

5.3.1 MarkedMessage

Questa classe rappresenta un generico messaggio contrassegnato, in cui è possibile risalire al Peer mittente. Il messaggio incapsula il *PeerDescriptor* del Peer sorgente.

5.3.2 PeerListMessage

Questo messaggio rappresenta la risposta a una richiesta di una lista di Peer [5.3.3]. Il messaggio incapsula nel Payload la lista di Peer richiesta. In più è possibile specificare il tipo di lista se necessario.

5.3.3 RefillPeerListMessage

Questo messaggio rappresenta la richiesta di una lista di peer. La quantità richiesta viene specificata all'interno del messaggio. In caso consista in un numero uguale a zero, corrisponde alla richiesta di tutti i *PeerDescriptor* posseduti dal ricevente del messaggio.

5.3.4 FillStorePeerListMessage

Questo messaggio rappresenta la richiesta di una lista di Storage Server Peer disponibili a memorizzare nuovi frammenti. La classe estende RefillPeerListMessage [5.3.3], acquisendone le caratteristiche.

5.3.5 EncodedFragmentPublishMessage

Questo messaggio rappresenta la richiesta da parte di un Client Peer a un Storage Server Peer di pubblicare un frammento codificato di una risorsa. Il messaggio incapsula nel Payload il frammento da memorizzare.

5.3.6 EncodedFragmentRequestMessage

Questo messaggio rappresenta la richiesta da parte di un Client Peer a un Storage Server Peer di restituire il frammento di una risorsa che quest'ultimo custodisce. Il messaggio incapsula la chiave per identificare univocamente la risorsa.

5.3.7 EncodedFragmentResponseMessage

Questo messaggio rappresenta la risposta di un Storage Server Peer alla richiesta da parte di un Client Peer del frammento custodito relativo a una determinata risorsa. Possiamo vederlo come messaggio di risposta a una richiesta di un frammento [5.3.6]. Il messaggio incapsula nel Payload il frammento richiesto.

5.4 Utilità di sistema

5.4.1 EncapsulatedEncodedFragment

La ricezione di messaggi in una rete creata con *Sip2Peer* avviene tramite la lettura del messaggio in formato *String* o in formato *JSONObject*, dal quale vengono estratti i valori delle variabili e ricostruiti i messaggi originali.

All'interno di un frammento sono presenti dati in formato *char*, quali il vettore codificato e la colonna della matrice G usata per codificarli. Per ovviare a un problema di male interpretazione di questi valori *char*, prima di inviare un messaggio contenente un frammento, questo viene codificato in modo opportuno. Il ricevente, effettuerà da parte sua una decodifica per riportare all'interno del frammento i valori originali.

La classe mette a disposizione i metodi *encaplusate()* e *decapsulate()* per eseguire queste operazioni sia su un EncodedFragment [3.1.1] che su un EncodedFragment-Header [3.1.2].

I metodi *decodeJSONHeader()* e *decodeJSONFragment()* permettono l'estrapolazione da un oggetto *JSONObject* di un *EncodedFragmentHeader* o un *EncodedFragment*, applicando l'incapsulamento in modo trasparente.

5.4.2 JSONObject2Peer

Questa classe contiene due utilità per estrarre da un oggetto *JSONObject* un singolo *PeerDescriptor* oppure una lista di *PeerDescriptor*.

5.5 UnitTest per la rete p2p

Sono state create quattro classi di test per provare la correttezza dell'implementazione di diverse funzionalità all'interno di una rete p2p generata ad-hoc.

Per generare una rete ad-hoc sufficientemente grande e con delle caratteristiche predeterminate è stata create la classe InitNetwork. Questa classe inizializza una rete p2p di 100 peer, ognuno con una lista di vicini di almeno 30 elementi.

Le classi usate per i test sono le seguenti, in ordine di complessità di azione simulata:

- *TestFillPeerListClientBehavior* - Controlla che i comportamenti FillPeerListServerBehavior e FillPeerListClientBehavior funzionino nel modo richiesto. Nel test viene preso il 70-esimo peer e richiesto di estendere fino ad almeno 50 elementi la sua lista di vicini. Da notare come questo test non si possa considerare del tutto deterministico, ma varia in base a come viene creata la rete. Nel nostro caso, è difficile pensare che questo test funzioni correttamente per ogni peer. In quanto i primi avranno una propria lista di peer molto meno variegata rispetto a gli ultimi creati. Sarà più difficile richiedere lo stesso quantitativo di peer al primo nella lista.

- *TestFillStorePeerListBehavior* - Il test è simile al precedente. Viene richiesto di riempire una lista di peer. In questo caso il numero richiesto varia a seconda del numero di frammenti codificati di una risorsa che si vogliono generare. Calcola il numero ed inoltra le richieste ai propri vicini.

- *TestPublishEncodeFileBehavior* - In questo test viene simulata la codifica e pubblicazione di una risorsa.

- *TestRetrieveEncodeFileBehavior* - In questo test viene simulato l'intero processo: codifica, pubblicazione, recupero dei frammenti, decodifica e controllo che la risorsa ricostruita sia identica alla risorsa pubblicata.

6 Sviluppi futuri

Il progetto in sè ha raggiunto i risultati sperati, producendo un prototipo di memorizzazione distribuita in una rete p2p. Per conseguire una sufficiente robustezza, la rete necessita di raffinare il suo comportamento per venire incontro a tutte le possibili situazioni impreviste che possono capitare in una situazione reale.

Per quanto riguarda la classe GaloisField, potrebbe essere preso in considerazione l'uso di *BigInteger* o *int* invece dell'uso di *char*. Richiede un sostanzioso refactoring del codice, sia di GFJama che del prototipo di rete p2p. La soluzione permetterebbe di approcciare campi di Galois ancora più ampi. Oltre a eliminare il problema dell'incapsulamento dei frammenti prima di poter essere trasmessi come messaggi JSON (vedi sezione 5.4.1). Eliminando un uso non necessario di risorse di calcolo.

Esistono diversi metodi per implementare in modo efficiente i campi di Galois. Per questo si potrebbe convertire la classe GaloisField facendola diventare una interfaccia, permettendo di aggiungere diverse implementazioni.

Le matrici sono il campo ideale per applicare meccanismi di parallelizzazione delle istruzioni. Il calcolo matriciale all'interno della libreria GFJama si potrebbe riscrivere per sfruttare la potenza di calcolo delle GPU, in modo da renderlo molto più performante.

Prendiamo in esame l'engine di coding, in futuro si potrebbe affinare le sue caratteristiche e implementando nuovi engine.

Inseriamo qualche osservazione per i Behavior.

Siccome il reperimento di liste di peer è molto importante, va sicuramente irrobustito il meccanismo, tenendo in considerazioni possibili timeout e risposte negative.

Sarebbe opportuno inserire un design pattern che permetta di impostare differenti strategie per distribuire le richieste all'interno di FillPeerListClientBehavior. Attualmente vengono distribuite in modo uniforme tra tutti i possibili peer. Dall'altro lato, anche nel momento in cui viene estratta una sotto lista di peer in FillPeerListServerBehavior, potrebbe essere opportuno inserire un meccanismo per scegliere la strategia più opportuna oltre a quella presente, cioè di estrazione casuale tramite il metodo *getRandomPeerList()*.

In riferimento alla Behavior di pubblicazione di una risorsa [5.2.6], sarebbe auspicabile creare una classe per raccogliere tutte le informazioni relative ai frammenti e ai Storage Server Peer che li conservano. Si potrebbe prendere in considerazione ad esempio una mappa per associare l'EncodedFragmentHeader con il PeerDescriptor del peer che conserva il frammento. A questo punto, il peer avrebbe a disposizione i dati necessari per poter controllare che la disponibilità dei frammenti nella rete non scenda sotto una determinata soglia, agendo di conseguenza per ripubblicare i frammenti andati persi perchè ad esempio il peer che lo custodiva è offline oppure per la corruzione dei dati.

Sarebbe auspicabile implementare una Factory per gestire le promozioni dei peer verso uno dei quattro tipi, con conseguente attivazione e disattivazione delle eventuali Behavior.

In futuro, si potrebbe definire meglio il tipo *NetworkAdmin Peer*, specificando meglio il suo ruolo all'interno della rete. Attualmente non sono stati definiti meccanismi per mantenere una lista di Storage Server Peer. Nè è stato definito un

meccanismo che permetta a quest'ultimo di specificare lo spazio disponibile. Di conseguenza, potrebbe essere raffinata la strategia di scelta dei Storage Server Peer in base a differenti parametri; ad esempio, privilegiando i Storage Server Peer più longevi, oppure quelli che hanno più spazio a disposizione.

References

[1] Marco Martal, Marco Picone, Michele Amoretti, Gianluigi Ferrari, and Riccardo Raheli. Randomized network coding in distributed storage systems with layered overlay. In *ITA*, pages 324–330. IEEE, 2011.

[2] MathWorks and NIST. Jama is a basic linear algebra package for java. http://math.nist.gov/javanumerics/jama/.

[3] Irving Reed and Golomb Solomon. Polynomial codes over certain finite fields. *Journal of the Society of Industrial and Applied Mathematics*, 8(2):300–304, 06/1960 1960.

[4] Rodrigo Rodrigues and Barbara Liskov. High availability in dhts: Erasure coding vs. replication. In *Peer-to-Peer Systems IV 4th International Workshop IPTPS 2005*, Ithaca, New York, February 2005.

[5] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *In Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002*, 2002.