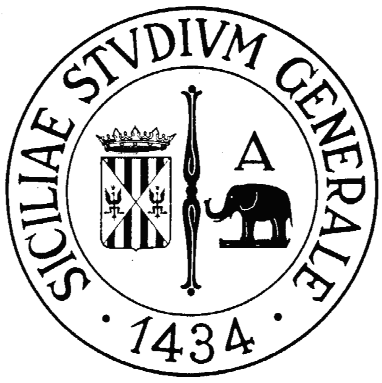
UNIVERSITÀ DEGLI STUDI DI CATANIA



DIPARTIMENTO DI MATEMATICA E INFORMATICA

CORSO DI LAUREA IN INFORMATICA, PRIMO LIVELLO

Oreste Delitala Matr. M01/000041

APPLICAZIONE DELL'ALGORITMO REED-SOLOMON PER LA DISTRIBUZIONE DI FILE IN INTERNET

Relatore:

Chiar.mo Prof. Salvatore Riccobene

Anno Accademico 2012/2013


SICILIAE STVDIVM GENERALE
1434

RINGRAZIAMENTI

Desidero innanzitutto ringraziare il Professor Salvatore Riccobene per i preziosi insegnamenti e le per le numerose ore dedicate alla mia tesi.

Un ringraziamento particolare va ai colleghi, mia sorella e agli amici che mi hanno incoraggiato e sostenuto nelle attività universitarie in questi tre anni, in particolare Salvo, Fabrizio, Roberto, Andrea I., Bebo, Andrea DB. e Gabriele.

Il ringraziamento più grande va a mia madre che mi ha sempre supportato in tutta la carriera universitaria ed è grazie ai suoi sacrifici se ho potuto raggiungere questo obiettivo.

Infine ringrazio Ilaria, la mia fidanzata, che mi ha dato la forza di completare gli ultimi esami della mia carriera con la sua frequente presenza durante le ore di studio.

INDICE

Introduzione

Lo scopo di quest'applicazione è la distribuzione di file su Internet tra un insieme di n host, con la manipolazione del file attraverso l'algoritmo di codifica Reed-Solomon che ne aggiungerà informazioni aggiuntive, e in seguito dividere il file in n frammenti e inviare il proprio frammento ad ogni host.

Prima di inviare il frammento di file va verificato se l'host sia connesso, in caso contrario sarà sospeso l'invio e in seguito sarà l'host stesso a controllare se è aggiornato con tutto il sistema e ricavarsi il proprio frammento di file.

Nel caso della richiesta del file dal sistema invece il procedimento è diverso, grazie all'algoritmo di decodifica Reed-Solomon non avremo la necessità di avere tutti gli host online al momento della richiesta ma è sufficiente avere un numero di host che permetta, componendo i singoli frammenti, di avere un file con un numero di byte noti uguale alla grandezza in byte del file iniziale prima della codifica Reed-Solomon, i restanti byte sconosciuti vengono sostituiti con un carattere alternativo e conservate le loro posizioni. Se per esempio il nostro file iniziale è formato da 300 byte e dopo l'algoritmo di codifica sono aggiunti ulteriori 600 byte e si hanno 9 host, il file finale avrà la grandezza di 900 byte che sarà suddiviso per gli host con frammenti di grandezza 100 byte ciascuno, per ricostruire il file di partenza avremo bisogno di 3 frammenti di file, quindi soltanto 3 host connessi. Composti i vari frammenti di file, il risultato finale è mandato in

decodifica assieme alle posizioni dei frammenti mancanti salvate e come output si avrà il file iniziale.

Gli host da selezionare per ricevere i frammenti di file sono selezionati tramite una funzione hash, è verificato se l'host restituito sia connesso e si procede, altrimenti è ricalcolata la funzione hash per avere un altro host. Questo è ripetuto finché ho raggiunto il numero di host connessi necessario per ricostruire il file.

Capitolo 1

Reed-Solomon

1.1 Generalità

I codici Reed-Solomon sono dei codici non binari ciclici, orientato cioè ad un alfabeto di q simboli. I simboli generalmente vengono considerati come un insieme di m bit, con m intero positivo maggiore di 1, e cardinalità potenza di due.

Definiamo n e k, come n numero totale dei simboli che compongono la parole di codice e k come il numero di simboli dell'informazione da preservare, data la dimensione dei simboli si ha il codice di Reed-Solomon RS(n,k) per ogni n e k tali che:

$$0 < k < n < 2^m + 2$$

Anche se il limite imposto da n è pari a $2^m + 2$, i Reed-Solomon prevedono maggiormente una struttura con una codeword di lunghezza pari a $2^m - 1$.

Il fattore di correzione t è dato dal rapporto $\frac{n-k}{2}$, tale rapporto appartiene alla classe dei codici MSD (Maximum Distance Separable), a parità di n e k conseguono la massima distanza minima ottenibile rispetto ad altri codici lineari a blocchi.

Il fattore di correzione della cancellazione è dato da ($n - k$). E per quanto riguarda la correzione simultanea di errori e cancellazioni la capacità del codice è data da:

$$2err + canc < n - k$$

L'errore del simbolo è considerato quando il simbolo non è corretto, anche se si riferisce a un singolo bit errato. Un codice Reed-Solomon può correggere fino ad un massimo di *t* bit errati a condizione che non ci siano più di un bit per simbolo errato nel peggiore dei casi, invece nel migliore dei casi fino a *tm* bit errati a condizione che tutti i simboli siano totalmente errati.

Gli algoritmi di decodifica correggono errori fino ad un massimo di *t* e cancellazioni fino ad un massimo di 2*t*.

Se in fase di decodifica non vengono soddisfatte le proprietà di cardinalità degli errori e cancellazioni il decoder rileverà la presenza di errori oppure commetterà un errore in fase di decodifica nel caso in cui il pattern di errori abbia trasformato una codeword trasmessa in un'altra codeword valida.

1.2 I campi di Galois

I principi base delle operazioni di codifica e decodifica dei codici ciclici, come quello di Reed-Solomon, fanno riferimento alla teoria dei campi a elementi finiti, detti anche campi di Galois.

Per ogni numero primo q, esiste un campo di Galois GF(q) composto da q elementi, che può anche essere esteso nel campo GF(q^m) con m intero maggiore di 1. Nel contesto dei codici Reed-Solomon si fa riferimento al campo GF(2^m), estensione del campo GF(2).

Un campo di Galois è un insieme finito di elementi dove sono definite le operazioni di somma e prodotto e che gode anche di altri proprietà:

a) Chiusura rispetto alla somma e prodotto
b) Associatività, commutatività, distributività
c) Esistenza dell'elemento neutro rispetto alla somma (0) e rispetto al prodotto(1)
d) Esistenza dell'opposto e del reciproco

Un altro elemento nel campo GF(2^m) è l'elemento primitivo α tale ogni elemento non nullo del campo può rappresentarsi come potenza di α.

Con queste definizioni, a partire dalla terna 0, 1, α, è possibile costruire un insieme infinito di elementi; se invece si vuole creare un insieme finito con 2^m elementi si impone un'altra condizione:

$$\alpha^{2^m-1} = 1 = \alpha^0$$

Per una potenza di α maggiore di 2^m - 1 può ridursi ad un altro elemento che risulti da una potenza di α minore:

$$\alpha^{2^m+n} = \alpha^{(2^m-1)} \cdot \alpha^{n+1} = \alpha^{n+1}$$

L'elemento primitivo α è una delle radici del cosiddetto polinomio primitivo *p(x)* che definisce un campo di Galois. Un polinomio di grado *m* è primitivo se è irriducibile se il più piccolo intero *n* per cui risulta divisore di $x^n + 1$ è $2^m - 1$.

I codici Reed-Solomon e i campi di Galois si legano se poniamo in corrispondenza biunivoca i *q* simboli da *m* bit dell'alfabeto del codice, con gli elementi di un campo GF(2^m). Quindi si può far riferimento indifferentemente agli elementi del campo o ai simboli del codice.

Si hanno due diverse modalità per rappresentare gli elementi del campo di Galois. Una viene detta ***index form*** e i termini

dell'esponente α sono rappresentati in modo del tutto simile alla rappresentazione in decibel:

$$\beta_{(index\ form)} = \log_{\alpha} \beta$$

Per 0 si considera 0 =α^{-1}

L'altra viene detta ***polynomial form*** rappresenta gli elementi per mezzo di m-ple binarie del tipo (00…0) per 0, (00…1) per 1, (100…0) per α^{m-1}. Gli elementi successivi l'm-pla si ottengono dalla divisione in modulo per il polinomio primitivo, considerando i coefficienti del resto in termini di $\alpha^{0} \cdots \alpha^{m-1}$.

Le operazioni aritmetiche su GF(2^m) sono definite nel secondo modo:

- Addizione tramite l'operatore logico *or esclusivo* (*xor*) tra le m-ple coinvolte.
- La sottrazione anch'essa tramite *xor* tra le m-ple coinvolte.
- Per il prodotto conviene usare la forma *index form* in quanto il risultato è dato dalla somma degli elementi.

Le fasi di codifica e decodifica usano le operazioni descritte sopra sui simboli delle parole, visti come elementi di un opportuno campo di Galois.

1.3 Codifica

La forma RS($2^m - 1, 2^m - 1 - 2t$) risulta la più utilizzata per i codici Reed-Solomon, dove $2t$ è il numero di simboli di ridondanza e t la capacità di correzione del codice.

Il polinomio generatore per un RS è definito come:

$$g(x) = (x - \alpha^{j0}) \cdot (x - \alpha^{j0+1}) \cdot (x - \alpha^{j0+2}) \cdot \ldots \cdot (x - \alpha^{j0+2t-1}) = g_0 + g_1 x + g_2 x^2 + \ldots + g_{2t} x^{2t}$$

in cui $j0 \in [0, 1]$.

Il grado del polinomio generatore è uguale al numero di simboli di parità.

Consideriamo ($d_1, d_2, \ldots, d_k$) come messaggio da codificare, in cui ogni d_i è un numero composto da m bit, che può essere rappresentato da un elemento del campo GF(2^m). Poniamo questi simboli come i coefficienti di un polinomio:

$$d(x) = d_0 + d_1 x + d_2 x^2 + \ldots + d_{k-1} x^{k-1}$$

Nell'algoritmo di codifica si cerca di ottenere una codeword, rappresentata da un polinomio di grado $n - 1$, che sia divisibile per $g(x)$. Per ottenere la parola di codice sfruttiamo il prodotto tra la data word e il polinomio generatore:

$$c(x) = d(x)\, g(x)$$

Purtroppo questa implementazione porta ad un codice non sistematico. Per un codice sistematico si considera come codeword:

$$c(x) = x^{2t} d(x) p(x)$$

con:

$$p(x) = x^{2t}d(x) \bmod g(x)$$

In questo modo il polinomio $c(x)$ presenta come coefficienti dei termini di grado $0 \ldots 2t$ i simboli di parità, e come termini di grado successivo i simboli di informazione.

Il decoder per individuare gli errori basterà eseguire la divisione per $g(x)$ che se presenterà resto allora la codeword sarà errata.

1.4 Decodifica

Poniamo che nella codeword ottenuta ci siano degli errori, rappresentiamo il pattern di errori sotto forma polinomiale:

$$e(x) = r(x) - c(x)$$

con $r(x)$ parola ricevuta.

Si può vedere $e(x)$ come:

$$e(x) = e_0 + e_1 x + e_2 x^2 + \ldots + e_{n-1} x^{n-1}$$

Dove gli e_i appartengono a GF(2^m) e indicano il valore dell'errore, mentre il grado va a definire la posizione dello stesso.

Esiste una differenza con i codici binari, l'unico problema della correzione degli errori era rilevare la posizione del simbolo errato in quanto bastava negare il valore per correggerlo. Nel caso in cui il valore non sia binario invece non basta conoscere la posizione errata ma anche il suo valore originario. In questo caso abbiamo il doppio delle incognite e di conseguenza avremmo bisogno del doppio delle relazioni per arrivare ad una soluzione.

Per prima cosa nei codici a blocchi nella fase di decodifica va calcolata la sindrome associata alla parola ricevuta. Con questo passo riusciamo a stabilire se sono presenti degli errori in quando solo le codeword corrette presentono una sindrome nulla. La sindrome diversa da zero fa presente che ci sono degli errori nella parola ricevuta.

La sindrome S sarà composta da $n - k$ simboli $S = \{S_i\}$ con $i = 1, 2, \ldots, n - k$.

Il calcolo della sindrome è molto facilitato dalla struttura del codice. Ogni codeword valida sarà proporzionale al polinomio generatore. Partendo dalle radici di $g(x)$, ($\alpha, \alpha^2, \ldots, \alpha^{2t}$) dovranno essere radici anche del polinomio rappresentante una tra le parole codice valide. Gli elementi della sindrome S associata alla parola $r(x)$ saranno quindi calcolati con la seguente formula:

$$S_i = r(\alpha^i) \qquad i = 1, 2, \ldots, 2t$$

Partendo dalla precedente definizione:

$$r(x) = c(x) + e(x)$$

e da:

$$c(\alpha^i) = 0 \qquad i = 1, 2, \ldots, 2t$$

possiamo calcolare la sindrome che è data dal polinomio $e(x)$ rappresentante il pattern di errori calcolato sulle radici del polinomio generatore.

Poniamo di avere ricevuto una $r(x)$ con v errori nelle posizioni x^{j1}, x^{j2}, …, x^{jv}, cosicché:

$$e(x) = e_{j1} x^{j1} + e_{j2} x^{j2} + \ldots + e_{jv} x^{jv}$$

dove l'indice $i = 1, 2, \ldots, v$ si riferisce al primo, secondo, …, v-mo errore, mentre j si riferisce alla posizione dell'errore. In conclusione si ricava l'espressione per il calcolo dei simboli della sindrome:

$$s_i = r(\alpha^i) = e(\alpha^i) = e_{j1} \beta_1^i + e_{j2} \beta_2^i + \ldots + e_{jv} \beta_v^i \qquad i = 1, 2, \ldots, 2t$$

avendo posto $\alpha^{j1} = \beta_1$.

Nel caso in cui le posizioni degli errori non siano note questa forma non può essere usata per calcolare la sindrome, che sarà calcolata a partire da $r(x)$ e gli s_i ottenuti saranno utilizzati come termini noti del sistema descritto sopra, che presenta $2t$ relazioni. Da ciò detto sopra si evince la limitazione alla correzione di soli t errori, infatti il sistema resta compatibile solo se v non supera il valore di t, dove il sistema presenta appunto $2t$ incognite.

Un limite del sistema è che non risulta lineare e quindi non è risolvibile con le tecniche usate per i sistemi lineari.

Per ricavare le posizioni degli errori occorsi si introduce il cosiddetto ***error locator polynomial***, le cui radici indicano le posizioni degli errori nel pattern $e(x)$:

$$\sigma(x) = x^t + \sigma_1 x^{t-1} + \ldots + \sigma_t$$

Il problema sta nel ricavare i valori dei coefficienti di σ_i a partire dal valore degli elementi della sindrome.

Calcolando il polinomio $\sigma(x)$ in una posizione di errore X_k, in modo che si annulli, e moltiplicandolo per X_k^j si ha:

$$X_k^j \sigma(X_k) = X_k^{t+j} + \sigma_1 X_k^{t+j-1} + \ldots + \sigma_t X_k^j = 0$$

Sommando questa espressione per $k = 1, 2, \ldots, t$ e considerato che:

$$s_i = r(\alpha^i) = c(\alpha^i) + e(\alpha^i) = \sum_{k=1}^{t} e_k X_k^i$$

si ha:

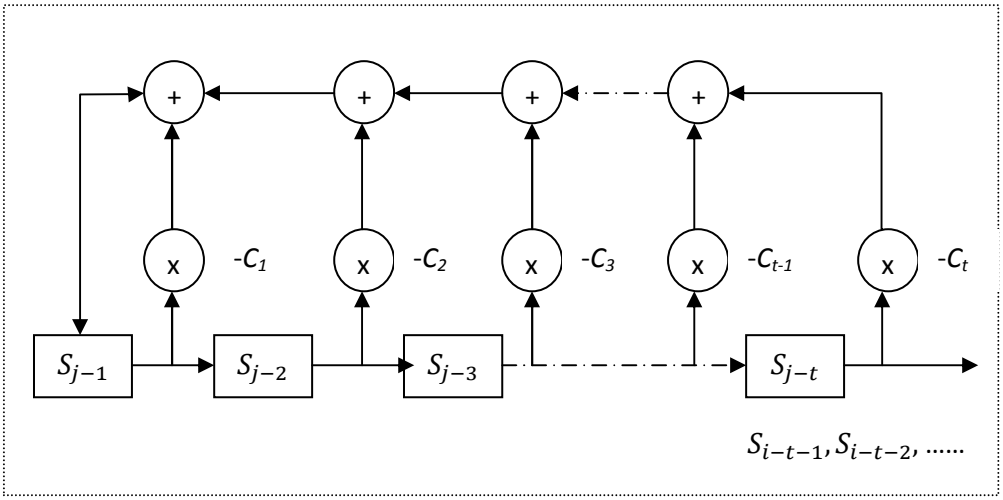
$$s_{t+j} + \sigma_1 s_{t+j-1} + \ldots + \sigma_t s_j = 0 \quad j = 1, 2, \ldots, t$$

Questo sistema di equazioni, dette ***identità di Newton***, ha come soluzione i coefficienti dell'*error locator polynomial*. Se i codici sono di piccole dimensioni questo sistema può risolversi mediante il metodo di Cramer, ma per codici più complessi bisogna usare un altro approccio. Questo problema venne risolto nel 1967, quando *E. R. Berlekamp* propose un algoritmo efficiente per il calcolo dei coefficienti σ_i, l'anno seguente inoltre *J. L. Massey* dimostrò che il problema è equivalente a quello di sintetizzare il più breve registro a scorrimento a retroazione lineare capace di generare una data sequenza, e che questo è equivalente all'algoritmo di *Berlekamp*. Questa soluzione viene correttamente impiegata per la codifica di tali codici, e prende il nome di algoritmo di *Berlekamp-Massey*.

Il seguente Circuito implementa l'equazione:

$$S_j = -C_1 S_{j-1} - C_2 S_{j-2} - C_t S_{j-t} \qquad j = t+1, t+2, \ldots$$

Ovvero le identità di Newton, avendo posto $\sigma_i = C_i$



Circuito LFSR

Detto ***connection polynomial*** il polinomio definito come:

$$C(x) = 1 + \sum_{i=1}^{t} C_i x^i$$

Si ha che il calcolo dei coefficienti dell'*error location polynomial* equivale a determinare il polinomio $C(x)$ per un LFSR che generi la sequenza $S_{t+1}, S_{t+2}, \ldots$ con stati iniziali $S_1, \ldots, S_t$.

La sindrome ha determinati una serie di valori che possono essere generati da un determinato numero di polinomi $C(x)$ di grado diverso. Questo vuol dire che diversi pattern di errori possono generare la stessa medesima sindrome. La decodifica va a scegliere il pattern che presenta il numero minore di errori.

L'algoritmo di *Berlekamp Massey* sintetizza il registro a scorrimento a retroazione lineare di lunghezza minima mediante una routine iterativa che inizia considerando il più piccolo registro possibile e facendogli generare la sindrome. Gli elementi così ricavati sono confrontati con quelli effettivi. Ogni qualvolta si ha una discrepanza il registro considerato viene modificato e il procedimento ricomincia fino a quando non si ottiene tutta la sindrome.

I passi dell'algoritmo sono i seguenti:

a) ***Inizializzazione delle variabili***:

   $C(x) = 1$ (*connection polynomial*)
   $D(x) = x$ (*termine di correzione*)
   $L = 0$ (*lunghezza del registro*)
   $n = 1$ (*contatore degli elementi della sindrome*)

b) ***Considera l'elemento successivo della sindrome e calcola la discrepanza***:

   $\delta = S_n + \sum_{i=1}^{L} C_i S_{n-i}$

c) ***Testa il valore della discrepanza***:

   se $\delta = 0$ vai al passo ***h)***
   altrimenti vai al passo ***d)***

d) ***Modifica il polinomio C(x)***:

$C^*(x) = C(x) - \delta D(x)$

e) ***Testa la lunghezza del registro***:

se $2L \geq n$ vai al passo ***g)***
altrimenti vai al passo ***f)***

f) ***Cambia la lunghezza del registro e modifica il termine di correzione***:

$L = n - L$
$D^*(x) = C(x) / \delta$

g) ***Aggiorna il polinomio C(x)***:

$C(x) = C^*(x)$

h) ***Aggiorna il termine di correzione***:

$D(x) = xD^*(x)$

i) ***Aggiorna il contatore degli elementi della sindrome***:

$n = n + 1$

j) ***Verifica il numero di elementi della sindrome***:

se $n < d$ ricomincia la ***b)***
altrimenti esci.

Il polinomio $C(x)$ così ottenuto è quello del registro di interesse.

Il *connection polynomial* è inizialmente posto a 1 ed è successivamente alterato in modo da generare correttamente gli elementi della sindrome in sequenza. Il polinomio viene modificato dal termine $D(x)$ quando viene riscontrata una discrepanza diversa da zero. Ad ogni iterazione tale discrepanza è la differenza tra il valore dell'elemento della sindrome e quello calcolato dal registro ottenuto in quella iterazione. È da notare che alla prima iterazione la discrepanza sia data da S_1 anche se non ci sono elementi precedenti da cui tale valore può essere stato calcolato.

Ogni volta che la discrepanza è non nulla $C(x)$ è modificato in funzione della discrepanza stessa e dal termine correttivo. Tale modifica è la parte più interessante dell'intero algoritmo in quando permette di ottenere un polinomio che annulli la discrepanza non solo al passo corrente, ma anche nei precedenti. In altre parole la modifica non è influente per quanto riguarda la generazione degli elementi della sindrome precedenti. Questo fatto rende molto efficiente l'algoritmo limitando notevolmente il numero di passi da svolgere.

Evidentemente, essendo t la capacità correttiva del codice, qualora la lunghezza finale L del registro (che coincide con il grado di $C(x)$ e quindi con il numero di errori) ecceda tale valore, non è assicurata un'esatta correzione della codeword.

Una volta che siano noti tali coefficienti è facile calcolare le radici dell'*error locator polynomial* ottenendo così le posizioni degli errori. Un algoritmo usato per questo scopo è *Chien Search*, che è in realtà un metodo di ricerca esaustivo. Le radici infatti saranno uno o più elementi del campo di Galois e l'algoritmo prevede semplicemente di

calcolare il polinomio in tutti gli elementi del campo GF(2^m). Ogni volta che il polinomio si annulla la radice appena trovata indica la posizione dell'errore.

È importante notare l'arbitrarietà nell'associazione della posizione all'indicizzazione che nel polinomio $e(x)$ indica il numero dell'errore.

Una volta note le posizioni degli errori si possono ricavare il valore degli errori risolvendo il sistema di $2t$ equazioni dato dalle relazioni tra gli elementi della sindrome e il polinomio $e(x)$ calcolato nelle radici di $g(x)$.

Un metodo più efficace per l'individuazione del valore degli errori è dato dall'algoritmo di *Forney* che definisce un polinomio specifico che mette in relazione la posizione degli errori e il loro valore. Mediante tale relazione si possono ricavare i valori, essendo note le posizioni.

Come emerge in molti passi del processo di decodifica, qualora si abbia a che fare con il polinomio $e(x)$ di grado $v > t$, ovvero con pattern da più di t errori, il codice non è in grado di effettuare la correzione, infatti i sistemi introdotti non risultano compatibili.

Noti anche i valori $e_{j1}, \dots, e_{jv}$ è possibile ricostruire il polinomio $e'(x)$ che si suppone sia occorso e, di seguito, correggere la parola ricevuta e considerata errata in seguito al calcolo della sindrome:

$$c'(x) = r(x) + e'(x) = c(x) + e(x) + e'(x)$$

L'operazione di decodifica è verificata se il calcolo della sindrome dà risultato positivo ($S = 0$). In caso contrario si può comunque asserire la rivelazione dell'errore avvenuto.

Come nella maggior parte dei codici a blocchi, la mancata rivelazione di un errore è dovuta al fatto che il pattern di errore ha trasformato $r(x)$ in una nuova parola di codice valida e quindi, per definizione, caratterizzata da sindrome nulla.

Capitolo 2

Struttura del Sistema

2.1 Socket

La socket è un meccanismo che permette di mettere in comunicazione nella rete diversi dispositivi, indipendentemente dal dispositivo fisico che effettuiamo la connessione.

La struttura su cui si basa una socket è il ***Client-Server***, dove si ha un *Server* che offre dei servizi ai *Client*, e il *Client* richiede dei servizi al *Server*.

Una Socket è come una porta di comunicazione che frutta il protocollo standard ***TCP/IP***, dove ogni qual volta si vuole comunicare ad un dispositivo basta interpellare questa porta di comunicazione.

Quindi sue informazioni indispensabili per aprire una socket sono:

- Indirizzo IP
- Numero di porta

Una volta stabilita la connessione con la socket il *Client* ed il *Server* si scambiano ***stream*** contenenti un insieme di byte. Per mandare un *stream* si fa riferimento all'***OutputStream***, invece per ricevere *stream* si fa riferimento all'***InputStream***.

2.2 Struttura Generale

Il sistema in oggetto ha lo scopo di distribuire file in internet tra un insieme di host fissato. Tutti gli host sono comunicano fra di loro tramite delle socket, una per ogni host, che sono attivate ogni qual volta entriamo nel sistema.

Ogni host ha un file di testo che contiene tutti i file posseduti con la data dell'ultimo aggiornamento.

Il sistema ha un algoritmo interno che fa si che tutti gli host siano sempre aggiornati con tutti gli altri host del sistema, questo algoritmo fa si che ogni qual volta che un host disconnesso si connetta al sistema scarichi tramite la richiesta di un file, che vedremo nel capitolo 4, un file di testo condiviso che contiene all'interno i nomi dei file condivisi e la data di ultimo aggiornamento, e confronta le date ed i file condivisi con il proprio file di aggiornamento, e se vi trova presenti delle differenze richiede al sistema i file mancanti per effettuare la codifica e prelevare il proprio frammento di file codificato mancante.

Il sistema farà uso di librerie per i codici Reed-Solomon, dove sono implementate tutte le norme precedentemente descritte nel Capitolo 1, necessarie per la codifica e decodifica di un file, compreso il calcolo dei campi di Galois.

In queste librerie per rendere efficiente il codice si è impostato il numero di byte di ridondanza per la codifica pari a 200 byte, e il numero di byte da codificare per volta pari a 100 byte.

Con questa implementazione l'algoritmo fa si che ogni qual volta che si deve eseguire la decodifica, non si necessita del totale numero di frammenti di file codificati ma solamente di un terzo di essi.

Il sistema avrà un interfaccia con l'utente in modo da poter far scegliere una tra le due possibili operazioni da eseguire:

- ***Condivisione di un file***, che prenderà un file in input, e sarà codificato dall'algoritmo Reed-Solomon e frammentato per in seguito essere inviato agli altri host.
- ***Richiesta di un file***, che prenderà in input il nome del file richiesto e farà si di recuperare da internet i frammenti di file, comporli e eseguire la decodifica dell'algoritmo Reed-Solomon restituendo in output il file originale.

Capitolo 3

Distribuzione di un File

3.1 Codifica RS e distribuzione del file

Dall'interfaccia utente del sistema quando viene selezionata la procedura di distribuzione di un file, prende in input il file e come effetto finale codificherà il file e lo frammenterà per il numero di host del sistema e si occuperà di mandare i corrispettivi frammenti.

Prima di effettuare la codifica viene chiamata un metodo della libreria Reed-Solomon ***inizialize_ecc()***, che inizializza la tabella con i polinomi generatori dei campi di Galois.

Adesso andiamo ad aprire le connessioni socket con gli altri host del sistema in modo da sapere chi in quel momento è connesso, e ogni qual volta la connessione si stabilisce setto ***true*** la posizione di un vettore di booleani, che chiamiamo ***host[]*** e inizialmente settato a *false* ogni sua posizione, corrispondente all'host.

Come stabilito nel capitolo precedente il numero di byte di ridondanza ***m*** viene settato a 200 e il numero di byte da codificare per volta ***n*** viene settato a 100.

Adesso prelevo i primi *n* byte dal file in input e si manda in esecuzione la procedura ***encode_data(buff[],nbyte,dst[])***, il parametro *buff[]* fa riferimento ai 100 byte estratti, il parametro *nbyte* il numero di mandati in codifica in questo caso 100 e il parametro *dst[]* fa riferimento al vettore dove verrà salvato in output il vettore di byte codificato.

I byte su *dst[]* contengono l'informazione codificata che saranno divisi in frammenti da $\frac{n+m}{h}$ e mandate tramite le corrispettive socket ad ogni host la sua parte (i primi $\frac{n}{h}$ byte riguarderanno informazione originale e i seguenti $\frac{m}{h}$ byte riguarderanno i byte di ridondanza):

- Host 0: i byte da 0 a $\frac{n}{h}-1$ e da n a $\frac{m}{h}+n-1$.
- Host 1: i byte da $\frac{n}{h}$ a $\frac{2n}{h}-1$ e da $\frac{m}{h}+n$ a $\frac{2m}{h}+n-1$.
- Host 2: i byte da $\frac{2n}{h}$ a $\frac{3n}{h}-1$ e da $\frac{2m}{h}+n$ a $\frac{3m}{h}+n-1$.
- Host 3: i byte da $\frac{3n}{h}$ a $\frac{4n}{h}-1$ e da $\frac{3m}{h}+n$ a $\frac{4m}{h}+n-1$.

.

.

.

- Host h-1: i byte da $\frac{(h-1)n}{h}$ a $n-1$ e da $\frac{(h-1)m}{h}+n$ a $m+n-1$.

Tramite le socket viene inviato quindi il comando all'host corrispondente per informarlo della ricezione di un frammento di file, questo comando conterrà il nome del file, e di seguito sarà inviato il frammento. Ovviamente prima di tale operazione va controllato il vettore booleano *host[]*, nella posizione dell'host selezionato, se è settato a *true*, e quindi l'host è connesso al sistema.

E quindi ripetiamo quest'algoritmo sopra elencato fin quando viene letto tutto il file in input e in seguito viene mandato un comando per ogni host di fine file, e i corrispettivi file frammentati saranno chiusi.

Adesso viene modificato il proprio file di aggiornamento e viene inviato con lo stesso algoritmo di distribuzione file agli altri host del sistema.

Capitolo 4

Richiesta di un File

4.1 Algoritmo di Hashing

L'algoritmo di Hashing è molto utilizzato quando si vuole estrarre degli elementi da una tabella, chiamata Tabella Hash, tramite una funzione hash che fa riferimento soltanto alla posizione del dato da estrarre e non al dato stesso:

$$h : U \rightarrow [\,0, 1, \ldots, m-1]$$

dove data una chiave $k \in U$ restituisce la posizione della tabella in cui l'elemento con chiave k viene memorizzato. È da notare bene che la dimensione m della tabella può non coincidere con la $|U|$, anzi in generale $m < |U|$.

Lo scopo di questo algoritmo è di definire una funzione d'accesso che restituisca la posizione di un elemento fornendogli la sua chiave in ingresso.

Con l'hashing, un elemento con chiave k viene memorizzato nella cella $h(k)$.

Un esempio di funzione hash è quella calcolata con il metodo della divisione, che consiste nell'associare alla chiave k il valore hash:

$$h(k) = k \bmod m$$

4.2 Ricostruzione del File

Questa parte del codice si occupa di recuperare i frammenti dai vari host del sistema in modo da ricostruire il file originale inserito nel sistema.
L'utente del sistema tramite l'interfaccia seleziona la voce per la richiesta di un file e inserisce il nome del file. Il sistema adesso dovrà cominciare ad interrogare un host alla volta per farsi mandare il proprio frammento, e questo passaggio di informazioni avviene tramite una socket.
Il modo in cui vengono selezionati gli host a cui richiedere i frammenti di file è tramite una funzione hash, descritta nel paragrafo precedente, dandogli in input una *k* che sarà un numero generato casualmente e usando come *m* il numero di host del sistema, in modo da aver tornato in output ***i***, un numero compreso tra 0 e *m-1*, e sarà l'indice dove andare a prelevare in un vettore salvato l'IP e la porta dell'host a cui richiedere il frammento.
Prima di effettuare la richiesta viene controllato il vettore *host[i]* se è *true*, quindi l'host sarà disponibile al sistema, altrimenti sarà ricalcolata la funzione hash con un altro numero casuale in input.
Trovato l'host adesso andremo a conservare il suo contenuto in un buffer di byte chiamato *buff[]* di grandezza 100 come stabilito precedentemente nel capitolo 2.

Le posizioni in cui salvare i byte nel sistema saranno nel seguente modo:

- Host 0: i primi 10 byte da 0 a $\frac{n}{h} - 1$
  i seguenti 20 byte da n a $\frac{m}{h} + n - 1.$
- Host 1: i primi 10 byte da $\frac{n}{h}$ a $\frac{2n}{h} - 1$
  i seguenti 20 byte da $\frac{m}{h} + n$ a $\frac{2m}{h} + n - 1.$
- Host 2: i primi 10 byte da $\frac{2n}{h}$ a $\frac{3n}{h} - 1$
  i seguenti 20 byte da $\frac{2m}{h} + n$ a $\frac{3m}{h} + n - 1.$
- Host 3: i primi 10 byte da $\frac{3n}{h}$ a $\frac{4n}{h} - 1$
  i seguenti 20 byte da $\frac{3m}{h} + n$ a $\frac{4m}{h} + n - 1.$

.

.

- Host h-1: i primi 10 byte da $\frac{(h-1)n}{h}$ a $n - 1$
  i seguenti 20 byte da $\frac{(h-1)m}{h} + n$ a $m + n - 1.$

Questa procedura sarà ripetuta fin che non si sono stati prelevati un numero di frammenti pari a $\frac{nh}{n+m}$.

Nei byte restanti del buffer *buff[]* non inseriti, mettiamo un byte a scelta che farà riferimento alla posizione vuota, e andiamo a inserire la posizione del byte mancante all'interno di un vettore che chiamiamo ***erasure[ ]*** con indice ***nerasure++*** dove *nerasure* sarebbe una variabile inizializzata a 0 e che corrisponde al numero di caratteri non letti, salvando la loro posizioni su *erasure[]*.

Finito di caricare il buffer *buff[]* si richiama il metodo della libreria Reed-Solomon ***decode_data(data[ ],nbyte)***, dove in *data[]* mettiamo

buff[] e in *nbyte* mettiamo *n+m*, ed in seguito viene calcolata la sindrome ***check_syndrome( )*** per la rilevazione di errori, se il valore è diverso da zero vengono corretti gli errori in *buff[]* con il metodo della libreria Reed-Solomon:

correct_errors_erasures(codeword[ ], csize, nerasures, erasures[ ])

dove in *codewors[]* inseriamo *buff[]*, in *csize* inseriamo *n+m*, *nerasures* e *erasures[]* quelli calcolati precedentemente.
Come output della funzione di correzione errori avremo adesso *buff[]* con i primi *n* byte di informazione originale (si escludono i successivi *m* byte che sono solo informazioni aggiuntive non necessarie), ripetendo quest'algoritmo fin che i frammenti di file siano completamente letti avremo ricostruito interamente il file originale.

Conclusioni

Per sviluppare questa applicazione ho utilizzato il linguaggio di programmazione "***C***", il motivo di questa scelta è stata data dalla disponibilità delle librerie Reed-Solomon sviluppate in tale linguaggio. La simulazione di questo sistema per la distribuzione di file è stato svolto con un numero di dieci macchine collegate alla rete. Settati il numero di byte da codificare per volta a 100 byte, e il numero di byte di ridondanza a 200 byte, con un totale di 300 byte per sezione di file codificato. Questo ci ha permesso di richiedere soltanto quattro diverse frammentazioni di file per ricostruire l'originale, e quindi sono state interpellate soltanto quattro macchine su dieci con un risparmio considerevole delle risorse del sistema.

Bibliografia

[1] Michelson, A. H. Lovesque: “Error – Control Techiques for Digital Communications”, Wiley Interscience, 1985.

[2] B. Sklar: “Reed Solomon Codes”