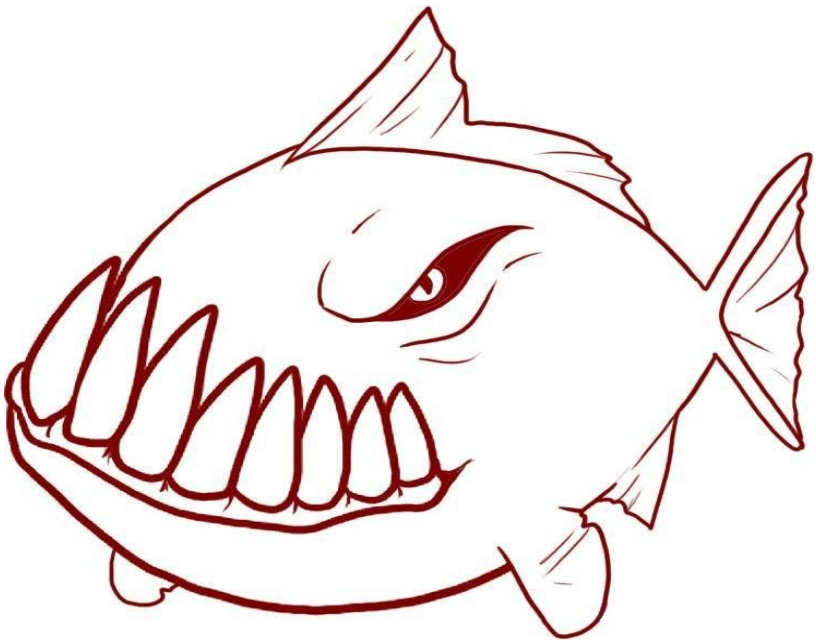
Sistemi Distribuiti - A.A. 2009/10

PACU File System



Progetto 1 per modalità B di esame:

File system distribuito transazionale con replicazione

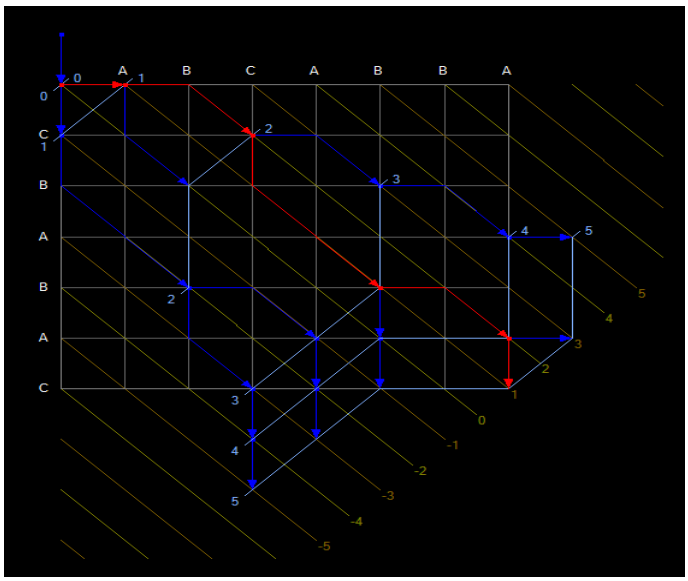
Progetto realizzato da Gabriele Cuppone e Antonio Papa

Introduzione

Il progetto PACU File System rispecchia le caratteristiche elencate nei requisiti di progetto.
La relazione descrive sinteticamente le modalità di interazione client–primary, primary-replica e replica-replica soprattutto con la presenza di guasti. Il sistema è stato simulato e testato su una rete virtuale realizzata tramite Netkit, che consente di simulare l'interconnessione di differenti dispositivi di rete su un unico pc.
La difficoltà maggiore riscontrata durante lo sviluppo del progetto è stata la gestione delle transazioni concorrenti su uno stesso file di testo. In particolare, effettuando il locking parziale del testo e non potendo prevedere l'ordine di terminazione delle transazioni, non siamo riusciti a trovare una soluzione per la gestione della variazione degli offset. Correndo ai ripari, abbiamo introdotto un file di tipo testuale formattato opportunamente e con estensione *.ga (vedi sezione Lock logico capitoli).

Panoramica sull'algoritmo di Myer

Ogni file GA è suddiviso in capitoli. Le modifiche su un singolo capitolo vengono gestite tramite gli algoritmi diff-match-patch di Google. Per poter utilizzare tali algoritmi è stato necessario integrare codice python nel progetto. Le librerie citate si basano sull'algoritmo di Myer per il calcolo della differenza tra due testi. In linea generale, l'obiettivo è di individuare lo Shortest Edit Script che converte un file A in un file B. Lo SES contiene solo due tipi di comandi: cancellazioni dal file A e inserimenti nel file B. Cercare l' SES equivale a trovare la sequenza di caratteri più lunga che può essere generata da entrambi i file rimuovendo alcuni caratteri (LCS – longest common subsequence). Ad es; dato A = (A, B, C) e dato B = (A, C, B), possiamo avere due LCS: (A,C) rimuovendo B e (A,B) rimuovendo C.
Consideriamo ora l'esempio in cui il file A contenga la sequenza “ABCABBA” di lunghezza N = 7 e il file B contenga “CBABAC” di lunghezza M = 6. Il problema di trovare l'SES si riduce nel cercare il percorso dal punto (0,0) al punto (N,M) con il minor numero di archi verticali e orizzontali.



Nella figura le diagonali vengono chiamate k-linee e sono molto utili: la linea che comincia da (0,0) è definita da k = 0. k cresce per le linee sulla destra della diagonale principale e decresce per quelle sulla sinistra, quindi tutte le linee sono rappresentate dall'equazione y = x – k. Si consideri il D-path come un percorso che parte da (0,0) e include D archi non diagonali. Ne consegue che un D-path è costituito da un (D – 1) path seguito da un arco non diagonale e una sequenza possibilmente vuota di archi diagonali chiamata "snake". A questo punto citiamo il lemma che afferma che un D-path deve finire sulla linea k appartenente a {-D, -D+2, …, D-2, D}. L'algoritmo greedy sfrutta il lemma e la proprietà dei D-path per ricercare il percorso più lontano da raggiungere su ciascuna k-linea per D successivi. L'algoritmo termina quando si raggiunge l'angolo in basso a destra della matrice. N + M è il massimo valore di D indica che non vi è alcuna corrispondenza tra le stringhe analizzate.

```
V[ 1 ] = 0;
for ( int d = 0 ; d <= N + M ; d++ )
{
  for ( int k = -d ; k <= d ; k += 2 )
  {
    // down or right?
    bool down = ( k == -d || ( k != d && V[ k - 1 ] < V[ k + 1 ] ) );

    int kPrev = down ? k + 1 : k - 1;

    // start point
    int xStart = V[ kPrev ];
    int yStart = xStart - kPrev;

    // mid point
    int xMid = down ? xStart : xStart + 1;
    int yMid = xMid - k;

    // end point
    int xEnd = xMid;
    int yEnd = yMid;

    // follow diagonal
    int snake = 0;
    while ( xEnd < N && yEnd < M && A[ xEnd ] == B[ yEnd ] ) { xEnd++; yEnd++; snake++; }

    // save end point
```

```
    V[ k ] = xEnd;


    // check for solution
    if ( xEnd >= N && yEnd >= M ) /* solution has been found */
  }
}
```

L'algoritmo presenta due cicli annidati. Il primo scorre i valori di D fino al valore massimo N+M. Il secondo scorre le k-linee secondo il lemma citato in precedenza. L'array V è indicizzato dai valori di k e contiene il valore x dell'end point associato a k.
La decisione di andare in basso o a destra per raggiungere la linea k data è demandata alla riga

```
bool down = ( k == -d || ( k != d && V[ k - 1 ] < V[ k + 1 ] ) );
```

Se k = -d dobbiamo andare in basso, se k = d dobbiamo necessariamente muoverci sulla destra. Per le situazioni intermedie scegliamo di partire dal quella linea vicina che ha il valore della x più grande. Questo algoritmo impiega O((M+N)D) per trovare la soluzione. Introducendo ulteriori ottimizzazioni, Myer arriva ad ottenere O(ND)

Il testo dei file non deve contenere caratteri non ASCII perchè non supportati dal python.

Caratteristiche architetturali e scelte progettuali

L'architettura è di tipo Client-Server. Abbiamo adottato lo schema leader-follower con prethreading e mutex sull'accept. Il thread principale (dispatcher) si mette in ascolto di eventuali connessioni e genera un pool di thread (helper) di dimensione statica. Ogni helper cerca di ottenere il lock sul mutex per lanciare l'accept sul socket di ascolto. Al più un thread helper (detto leader) si può trovare in ascolto, mentre gli altri (detti follower) sono in attesa di poter accedere alla sezione critica.
Il mutex garantisce il funzionamento corretto dello schema anche nei sistemi UNIX dove l'accept è una funzione di libreria (kernel UNIX derivati da SYSTEM V).

Il thread dispatcher:

- Crea il socket listensd adibito a trasmissioni tipo TCP/IP.
- Effettua la bind( ) su listensd per avvisare il Sistema Operativo di inviargli i dati provenienti dalla rete, relativi all'ip addr.sinaddr.saddr e alla porta addr.sinport.
- Mette il socket listensd in ascolto di eventuali connessioni. BLACKLOG identifica il numero di connessioni che possono essere accettate e messe in attesa di essere servite nella coda di richieste di connessione.
- Infine crea n thread helper a cui passa il socket listensd.

Il thread helper:

- Chiede il lock sul semafono sem_accept, appena lo ottiene prende dalla coda di backlog la prima connessione completa sul socket listends con la chiamata accept( ). Se la coda di backlog è vuota rimane bloccato sulla chiamata accept( ), finché non viene accettata una nuova connessione. Infine rilascia il lock.

- Ritorna a competere con altri thread helper per l'utilizzo dell'accept( ).

Il protocollo di replicazione è il primary-backup. Abbiamo suddiviso lo spazio del filesystem in domini. Ogni server del sistema è responsabile (primary) di uno o più domini ed è replica di altri. Il FSD usa UDP per i messaggi di ping/pong utili per conoscere lo stato di un server, per tutte le altre operazioni viene usato TCP (soprattutto per lo scambio di messaggi durante la transazione, in modo da tutelarsi da eventuali perdite dovute all'inaffidabilità della rete).

Assunzioni:

- Qualora un server rimanga down per più di un tempo d'attesa stabilito nella configurazione, il programma provvede ad escluderlo dalla lista server ed è compito dell'amministratore ripristinare lo stato del sistema e la consistenza del server in questione.
- Non può accadere che cadano tutte le repliche.

Le operazioni che un client può compiere sono:

- ld: mostra l'elenco dei domini del filesystem;
- mount <nome dominio>: accede al dominio specificato;
- ls: mostra il contenuto della directory corrente;
- cd <nome directory>: cambia la directory corrente;
- dwn <nome file>: scarica un file;
- open <nome file>: legge un file da remoto;
- search <filtro>: ricerca un file o una directory;
- readga <path locale>: legge un file di tipo GA;
- upload <path locale> <path remoto>: carica un file nel filesystem;
- rm <path remoto>: cancella un file dal filesystem;
- mkdir <nome directory> <path remoto>: crea una cartella nel filesystem;
- newga <nome file GA> <path remoto>: crea un file in formato GA;
- newcap <path remoto> <posizione capitolo>: crea un capitolo nel file GA;
- mvcap <lpath remoto> <path remoto> <posizione attuale capitolo> <nuova posizione capitolo>: sposta un capitolo del file GA;
- modcap <path locale> <path remoto> <numero massimo capitoli da modificare> : modifica un capitolo del file GA;
- rmcap: elimina un capitolo del file GA;
- rmdir: elimina una directory solo se è vuota;

Abbiamo predisposto un BootServer per consentire ai client, all'avvio, di richiedere la lista aggiornata dei primary da contattare. Inoltre, tramite bootserver è possibile configurare i parametri del sistema da remoto. Questa operazione può essere effettuata soltanto dall'amministratore di sistema, che può autenticarsi tramite l'applicazione client per mezzo di username e password criptate nel database remoto. Abbiamo pensato anche di predisporre la console client per l'inserimento di comandi SQL direttamente nel database. Il BootServer non rappresenta un single point of failure perché il sistema può benissimo funzionare senza: i client, infatti, conservano una copia locale della lista dei server del sistema ed una copia della lista di tutti i domini con i relativi primary. Se un primary non è disponibile, il client contatta un server qualsiasi e aggiorna la propria lista Domini.

Per la gestione della configurazione, dei log e delle informazioni di sistema abbiamo scelto di utilizzare un database relazionale embedded (sqlite). Ciò che motiva tale scelta è il supporto alla gestione dei crash che garantisce la transazionalità delle operazioni sul database. Sui Server vi sono quattro database: Conf (contine tutti i parametri di configurazione del sistema), Server (contiene la lista dei server primary), Domini (contiene l'associazione domini- primary – replica) e Logfile (contiene il log delle transazioni). Vediamo in dettagli il contenuto di questi database e il significato dei campi delle tabelle:

Conf:

Tabella Parametri (visualizzazione dei parametri di un server)

1|port_client|5005|porta usata per la connessione server-client
2|port_server|5002|porta usata per la connessione primary-replica
3|backlog_client|100|max dimensione della coda di backlog server-client
4|backlog_server|150|max dimensione della coda di backlog server
5|thread_helper_client|40|numero helper del dispatcher per i client
6|thread_helper_server|75|numero helper del dispatcher per i server
7|maxline|1024|numero di byte del blocco lettura-scrittura
8|time_helper|60|il tempo di attessa del dispatcher prima di forzare la chiusura degli helper
9|timeout_init_msg|15|timeout per i messaggi di tipo InitMessage
10|timeout_domlist|25|timeout per la lista domini-primary
11|timeout_tr|300|timeout per le transazioni
12|port_cl_pp|5003|porta usata per lo scambio ping-pong fra client e server
13|port_serv_pp|5004|porta usata per lo scambio ping-pong fra i vari server
14|timeout_pp|5|tempo stimato di RTT
15|thread_ppCl|20|numero di thread che gestiscono i ping-pong dei client
16|thread_ppSrv|20|numero di thread che gestiscono i ping-pong dei server
17|max_cap|5|numero massimo di capitoli da modificare
18|wait_rep|150|tempo di attesa per il reup di una replica
19|wait_pri|200|tempo di attesa per il reup del primary
20|timeout_pr|20|timeout per alcune operazioni server-server
21|porta_pr|5006|porta usata per le comunicazioni urgenti server-server
22|thread_helper_pr|10|numero di thread che gestiscono i messaggi PR
23|crash|0|segnala un probabile crash
24|ciclo_bully_f|200|tempo fisso del ciclo di bully

25|ciclo_bully_r|100|tempo random del ciclo di bully
26|timeout|60|tempo di attesa prima di iniziare il bully

Tabella NameIdServer

0|Ace|1030 rispettivamente: id riga | nome del server | id del server

Server:

Tabella ListServer

Ace|116.116.50.14|5005 rispettivamente: nome server | ip server | porta

Domini:

Tabella Dom

18|CH|Ace|2|116.116.50.14|5005 rispettivamente: id riga | Nome Dominio | Nome server | tipo server(1 – Primary, 2 – Replica) | Ip server | porta

Logfile:

Tabella Log

1|1000|2|116.116.100.4|5005|45|FileSystem/IT/prova.ga|tmp/CS4Tnf|tmp/CA2Dnp|1|-1|3

rispettivamente:

id riga | id server | count transazione | ip client | porta | id operazione | posizione della risorsa | posizione del file temporaneo | posizione file di salvataggio | opzione 1 (dipende dalla transazione) | opzione 2 (dipende dalla transazione) | stato transazione (0 – abort, 1 – prepara, 2 – pronto, 3 – commit)

(id server | count transazione) costituiscono l'identificativo univoco della transazione.

Lock logico dei capitoli

Quanto segue mostra la struttura di un file GA.

##PACUFS__GA__FORMAT##21412341234213412 3##
§§§2
^^^1

Strange Deja Vu
Subconscious strange sensation
Unconscious relaxation
What a pleasant nightmare
And I can`t wait to get there again

^^^2

I get this feeling sometimes
Like theres nothing in the world that isn't mine
And I could have it all
Maybe I should have it all
I'll take you out of your life
Just the one sweet night
And you could take it all
Oh maybe I could rip it off

La prima riga è una stringa di formato per riconoscere che il file trattato è un GA. La seconda riga è un contatore del numero di capitoli presenti all'interno del testo. Le righe che iniziano con la sequenza "^^^" rappresentano l'identificativo del capitolo. Un capitolo è la più piccola porzione di testo oggetto di una transazione. Una transazione può riguardare più capitoli ma un capitolo non può essere oggetto di più transazioni concorrenti.
Ogni capitolo è rappresentato dalla seguente struttura:

```
struct list_el {
  int init;
  Two_rooms *cap_mutex;
  struct list_el *next;
};
```

Init è l'identificativo del capitolo, next è un puntatore al capitolo successivo. Cap_mutex è di tipo Two_rooms, una struttura di sincronizzazione che gestisce la mutua esclusione sul capitolo.

La struttura Two Rooms

Questa struttura di sincronizzazione è usata per eliminare il problema della starvation tra i processi che attendono per accedere a una data risorsa. L'idea è di utilizzare due tornelli per creare due stanze di attesa, dette anche waiting rooms, prima della sezione critica (ossia prima dell'accesso al capitolo). La struttura di sincronizzazione lavora in due fasi. Durante la prima fase il primo tornello è aperto ed il secondo chiuso, per far accumulare i processi nella seconda stanza. Durante la seconda fase, il primo tornello è chiuso, in modo che nessun nuovo processo possa accedere alla seconda stanza, ed il secondo è aperta in modo da far accedere i processi della seconda stanza alla sezione critica. Anche se i processi che attendono nella seconda stanza possono essere di numero arbitrario, ad ognuno di essi è garantito l'accesso alla sezione critica prima dei processi presenti nella prima stanza.

La struttura che invece rappresenta l'intero file GA è la seguente:

```
typedef struct
{
  struct list_el *lista;
  int inode;
  int access;
```

}regions;

Contiene l'inode del file, il numero di thread che stanno accedendo a quel file e un puntatore alla testa della lista dei capitoli.

Quando i thread concorrono per accedere ad un file GA, si sincronizzano tramite mutex per l'accesso ad un array di regions che è globale. Ottenuto l'accesso alla risorsa, il thread verifica la presenza dell'inode del file nell'array e se manca lo aggiunge. Rilascia il mutex e si mette a concorrere per accaparrarsi i capitoli desiderati. Operazioni come upload e remove richiedono il lock di tutti i capitoli. Proprio per questo motivo si rischia lo stallo a causa dell'attesa circolare. La soluzione a questo problema richiede un ordinamento totale delle risorse (nel nostro caso i capitoli sono ordinati in base all'id). Ogni thread deve richiedere i capitoli in ordine crescente.

Tolleranza ai guasti

Failure di tipo omissione su client: il protocollo di comunicazione relativo alle transazioni prevede un campo nel messaggio adibito a contatore che viene incrementato ad ogni invio e ricezione di messaggi per la transazione specifica. Se il primary o il client si accorgono che il proprio contatore è diverso da quello previsto, abortiscono la transazione.

Failure di tipo failstop del client: se il client ha un crash prima dell'invio del commit della transazione al primary, la transazione è abortita. Se il primary ha ricevuto il commit prima del crash, la transazione viene portata a termine salvo eventuali altri fault. In ogni caso, quando risale, il client interroga il primary per aggiornare lo stato del proprio log e delle proprie risorse. Quest'ultima interrogazione coinvolgerà gli altri server qualora il primary fosse caduto.

Failure di tipo bizantino sul client: per ogni operazione si controlla il tipo di messaggio attraverso il campo paydescriptor, l'id della transazione (solo per le operazioni che lo richiedono), l'esistenza della risorsa, il tipo di risorsa, controllo dell'input.

Failure di tipo failstop su server: risolto con l'algoritmo Two Commit.

Two Commit

Client

Il client, dopo aver richiesto l'id della transazione, inserisce nel log PREPARE_TR. Il client elabora tutte le operazioni relative alla transazione (ad es. modifica di più capitoli), le invia al primary, inserisce PRONTO_TR e conclude la trasazione lato client inviando un commit. Il client, dopo aver ricevuto un commit dal primary, inserisce COMMIT_TR nel log.

Flussi alternativi:

Il client cade e nel log c'è PREPARE_TR: effettua l'abort della transazione cancellando i file temporanei ad essa inerenti.

Il client cade e nel log c'è PRONTO_TR ma non ha inviato il commit al primary: quando risale chiede al primary lo stato della transazione e ha come risposta ABORT.

Il client cade dopo l'invio del commit e nel log c'è PRONTO_TR: chiede al primary lo stato della transazione e ha come risposta commit o abort a seconda della riuscita o del fallimento della

transazione lato server. Se riceve commit aggiorna anche la risorsa locale.

Primary

Il primary crea l'id della transazione e la invia al client. Ricevuto il commit del client, inserisce nel log PREPARE_TR ed elabora la transazione, creando gli eventuali salvataggi per il redo. Inserisce PRONTO_TR e rende effettiva l'operazione. L'ordine è importante perchè se avviene un crash nel mezzo delle due operazioni o dopo di esse, il primary non deve preoccuparsi di un'eventuale inconsistenza in quanto procederà comunque al redo. La fase successiva è l'invio dell'operazione alle repliche, seguito dalla ricezione di un messaggio di pronto da quest'ultime. Appena ricevute tutte le risposte dalle repliche, inserisce nel log COMMIT_TR e manda la conferma alle repliche e al client.

Flussi alternativi:

Il primary cade e nel log c'è PREPARE_TR: risale e fa l'abort della transazione.

Il primary cade e nel log c'è PRONTO_TR: fa abort della transazione e lo comunica alle repliche.

Il primary riceve almeno un abort dalle repliche: fa abort su tutte le repliche e sul client.

Replica

La replica ricevono le operazioni della transazione dal primary ed inseriscono PREPARE_TR nel log, elaborano le operazioni, scrivono PRONTO_TR, rendono effettive le operazioni e rispondono con un messaggio di pronto. Quando riceve il commit dal primary scrive nel log COMMIT_TR.

Flussi alternativi:

Una replica cade e nel log c'è PREPARE_TR: risale e fa abort.

Una replica cade e nel log c'è PRONTO_TR: chiede al primary lo stato della transazione. Se non riesce a contattarlo, invia la richiesta alle repliche.

Meccanismo di elezione

Quando il primary cade e rimane down per più di un tempo specificato nella configurazione, le repliche avviano l'elezione per ciascun dominio gestito dal primary caduto tramite l'algoritmo bully. Abbiamo predisposto due modi per poter accorgerci della caduta di un primary: la caduta o il timeout della connessione durante una transazione oppure con thread adibiti all'invio di messaggi di ping/pong.

Protocollo applicativo

Per la comunicazione nel sistema utilizziamo varie classi di messaggi:

- *Boot Message:* sono usati nella comunicazione client – bootserver per permettere al client di aggiornare, nel caso ne abbia bisogno, la lista dei server.
- *Boot Administrator Message:* usati nella comunicazione amministratore – bootserver per modificare le impostazioni del server di boot tramite operazioni sul database.

- *UDP Message:* usati per il ping/pong tra le varie entità del sistema (escluso il bootserver). Richiedono un dispatcher dedicato in ascolto su una porta UDP prestabilita.
- *Initialization Message:*servono per le operazioni non transazionali e per preparare i server e i client all'invio/ricezione di un'operazione transazionale.
- *Transaction Message:*contengono le informazioni utili per la gestione della transazione.
- *Identification Message:*risolvono l'associazione primary – dominio quando un primary cade.
- *Primary – Replica Message*:usati per l'algoritmo di elezione, per la modifica degli stati dei log e per alcune operazioni sul DB.

BootMessage

1B	8B	8B
Pay Descriptor	Versione Dominio	Versione Server

Valori del Pay Descriptor:

- 0x01: boot_ping
- 0x02: boot_pong
- 0x03: boot_list
- 0x04: boot_error

Il client che accede al sistema avvia un boot_ping al server di boot contenente la propria versione di sistema (versione dominio + versione server). Il server invia il boot_pong contenente la propria versione di sistema. Il server confronta la versione di sistema del client con la sua: se sono diverse si prepara a ricevere un messaggio di boot_list dal client, altrimenti chiude la connessione. Il server, una volta ricevuto il messaggio di boot_list, invia al client la lista dei server del sistema. Se durante la comunicazione ci sono degli errori, come la mancanza di formattazione dei messaggi, il server invierà un messaggio di boot_error.

Boot Administrator Message

1B	30B	4B	30B	4B	1000B	4B
Pay Descriptor	Nick	Dim Nick	Password	Dim Password	Query	Dim query

Valori del Pay Descriptor:

- 0x05: boot_login_am
- 0x06: boot_exit_am
- 0x07: boot_use_db_am

- 0x08: boot_select_am
- 0x09: boot_ge_op_am
- 0x10: boot_error_am
- 0x11: boot_change_psw_am
- 0x12: boot_logout_am
- 0x13: boot_commit_am

L'amministratore manda il messaggio di boot_login_am al bootserver che risponde con boot_commit_am in caso di avvenuto login, altrimenti con boot_error_am. L'amministratore può utilizzare una piccola interfaccia per comunicare con il db del server. Ogni modifica richiede il riavvio della macchina del server, attraverso il messaggio boot_exit. Con boot_use_db_am si seleziona il db su cui operare, mentre con boot_ge_op_am si può effettuare qualsiasi comando sql che non richieda la restituzione di un risultato. Il significato del resto dei messaggi è intuitivo.

UDP Message

1B	4B	16B	4B
Pay Descriptor	ID Messaggio	IP Sorgente	Porta

Pay Descriptor:

- 0x25: ping_mess_udp
- 0x26: pong_mess_udp

Implementazione a livello di applicazione dei messaggi di ping e pong via UDP.

Initialization Message

1B	1024B	4B	4B	4B
Pay Descriptor	Risorsa	Lunghezza risorsa	Opzione 1	Opzione 2

Pay Descriptor:

- 0x30: init_update_dom
- 0x31: init_start_tr
- 0x32: init_ls
- 0x33: init_cd
- 0x34: init_search
- 0x35: init_read_resource

- 0x36: init_state_tr
- 0x37: init_whois_pr

Il messagio di init_update_dom è usato dal client per aggiornare la propria lista domini. Tale lista può essere aggiornata dinamicamente nel caso di crash di un primary attraverso il messaggio di init_whois_pr che risolve l'associazione primary-dominio del dominio orfano. Init_state_tr è utilizzato per riportare la coerenza nel record di un log della transazione. Inoltre, i messaggi init_ls, init_cd, init_search, init_read_resource sono utilizzati dal client per le operazioni da console non transazionali. Infine, init_start_tr è usato per preparare sia il client che il server all'inizio di una transazione.

Transaction Message

1B	4B	4B	4B	4B	4B
Pay Descriptor	ID Primary	Contatore	Contatore messaggi	Opzione 1	Opzione 2

Pay Descriptor:

- 0x41:TR_DO_ID
- 0x42:TR_COMMIT
- 0x43:TR_ABORT
- 0x44:TR_MKDIR
- 0x45:TR_UPLOAD
- 0x46:TR_NEW_GA
- 0x47:TR_RM
- 0x48:TR_ERROR
- 0x49:TR_NEW_CAP
- 0x50:TR_RM_CAP
- 0x51:TR_MV_CAP
- 0x52:TR_MOD_CAP
- 0x53:TR_RM_DIR

Il client riceve l'id della transazione (ID primary + ID count) con il messaggio di tr_do_id inviatogli dal server primary. I messaggi di tr_commit, tr_abort sono utilizzati per la comunicazione dello stato della transazine, mentre il tr_error trasporta le informazioni sullo specifico errore che può aver causato un abort. Gli altri messaggi contengono i dati necessari per eseguire le omonime transazioni.

Identification Message

1B	30B	4B	16B	4B
Pay Descriptor	Nome server	Lunghezza nome	IP server	Porta

Pay Descriptor:

- 0x71: idt_primary

Il messaggio viene usato da un server per rispondere al messaggio di init_whois_pr inviatogli dal client. Contiene le informazioni del primary che gestisce il dominio.

Primary-Replica Message

1B	30B	30B	4B	4B	30B
Pay Descriptor	Nome server	Nome dominio	Opzione 1	Opzione 2	Stringa opzione

Pay descriptor:

- 0x91: pr_bully_elect
- 0x92: pr_ack
- 0x93: pr_bully_primary
- 0x96: pr_error
- 0x97: pr_log_state
- 0x99: pr_set_log_state

Questi messaggi sono utilizzati nella comunicazione tra i server. I primi tre vengono usati nel meccanismo di elezione del primary, gli ultimi due per conoscere e aggiornare lo stato dei log. Pr_error informa sul tipo di errore avvenuto.

Conclusioni

I test sono stati eseguiti su alcune delle operazioni previste nell'implementazione del sistema. Il programma non brilla per prestazioni ma ciò è dovuto alla scelta di utilizzare molta comunicazione per avere garanzia dello stato dei server e delle transazioni. E' possibile aumentare in modo significativo le prestazioni eliminando i controlli “keep alive” interni alle procedure di gestione delle transazioni, in quanto esistono già dei controlli in background adibiti a tale scopo. Per avere informazioni sui test, consultare la cartella “Esempi di funzionamento” presente nel cd.