Relazione progetto d’esame Ingegneria Internet e Web
Professore: Francesco Lo Presti

EXODUS

Server Web con adattamento dinamico di contenuti statici

A cura di:

-Ezio Emanuele Ditella

-Giovanni Maria Cipriano

Sommario

Architettura del Web Server

Per la gestione della concorrenza si è adottata una strategia tale da incorporare I vantaggi offerti da un'architettura multi-processo ed una basata su eventi.

Exodus effettua preforking dinamico, in cui ogni processo generato è una 'diramazione' del Web Server ed è basata su eventi.

L'obbiettivo è quello di inglobare la velocità offerta da un approccio basato su eventi ed eliminarne gli svantaggi.

Con l'ausilio di più 'diramazioni' il server è più robusto e non impone limiti sul numero dei client connessi: la failure di una di esse non fa fermare l'intero server, ed inoltre il numero massimo di file descriptor (e quindi di client) gestiti da una singola diramazione non comporta un limite per l'intero Server Web.

Componenti Exodus

Il Web Server è costituito da due processi principali

- **ServerBranchesHandler**

  Gestisce la creazione e l'unione (in seguito riferita come operazione di merge) di diramazioni, in modo che ci sia sempre una certa percentuale di 'posti disponibili' per la gestione di nuovi client oppure un numero minimo di diramazioni nel caso in cui il numero dei client connessi sia inferiore ad una certa percentuale.

- **ServerBranch**

  Precedentemente riferita come 'diramazione', gestisce le richieste di un numero massimo di connessioni secondo un approccio basato su eventi e monitora la variazione del numero dei client ad essa connessi rendendola nota al ServerBranchesHandler

ServerBranchesHandler

La prima operazione effettuata dalla ServerBranchesHandler (o **handler**) al momento della sua creazione è l'inizializzazione delle seguenti zone di memoria condivisa

- Memoria dedicata alla cache e che comprende un array di strutture dati, ciascuna contenente
    1. 2 semafori per la sincronizzazione degli accessi in cache
    2. Un array di strutture di tipo 'images', ciascuna delle quali conterrà le informazioni su una determinata immagine salvata in cache
- Memoria in cui verrà salvata una struttura dati con le informazioni dell'handler utili alle singole ServerBranch, e sono
    1. Il suo PID
    2. File desciptor della socket di ascolto
    3. Semaforo per sincronizzare le varie ServerBranch per l'acquisizione della socket di ascolto
    4. Semaforo per sincronizzare il ServerBranch 'ricevitore' e 'inviatore' di client nell'operazione di merge
    5. Semaforo per sincronizzare 'ricevitore', 'inviatore' e lo stesso ServerBranchHandler durante l'operazione di merge
    6. Altri due semafori per sincronizzare i componenti 'logger' e 'loggerManager'
- Memoria in cui verrà salvato un array di strutture dati, ciascuna con le informazioni di una ServerBranch utili alla ServerBranchesHandler, contenenti
    1. PID della ServerBranch
    2. Numero di connessioni attualmente gestite dalla ServerBranch
    3. Semaforo per sincronizzare l'accesso alla variabile precedente
    4. Variabile usata durante l'operazione di merge in cui la singola ServerBranch capisce se ricevere oppure inviare I propri client
    5. Un array di strutture dati di tipo 'log' (discussa in seguito)

Dopo l'inizializzazione delle zone di memoria l'handler procede alla creazione di un dato numero iniziale di ServerBranch, passando loro un valore intero. Tale valore, sommato all'indirizzo di memoria (condivisa) iniziale, riferisce la struttura dati che la ServerBranch appena creata riempirà con le sue informazioni. Il ServerBranchesHandler organizzerà tali strutture dati in una lista doppiamente

collegata, che gli consentirà di poter manipolare le informazioni di ogni singola ServerBranch.

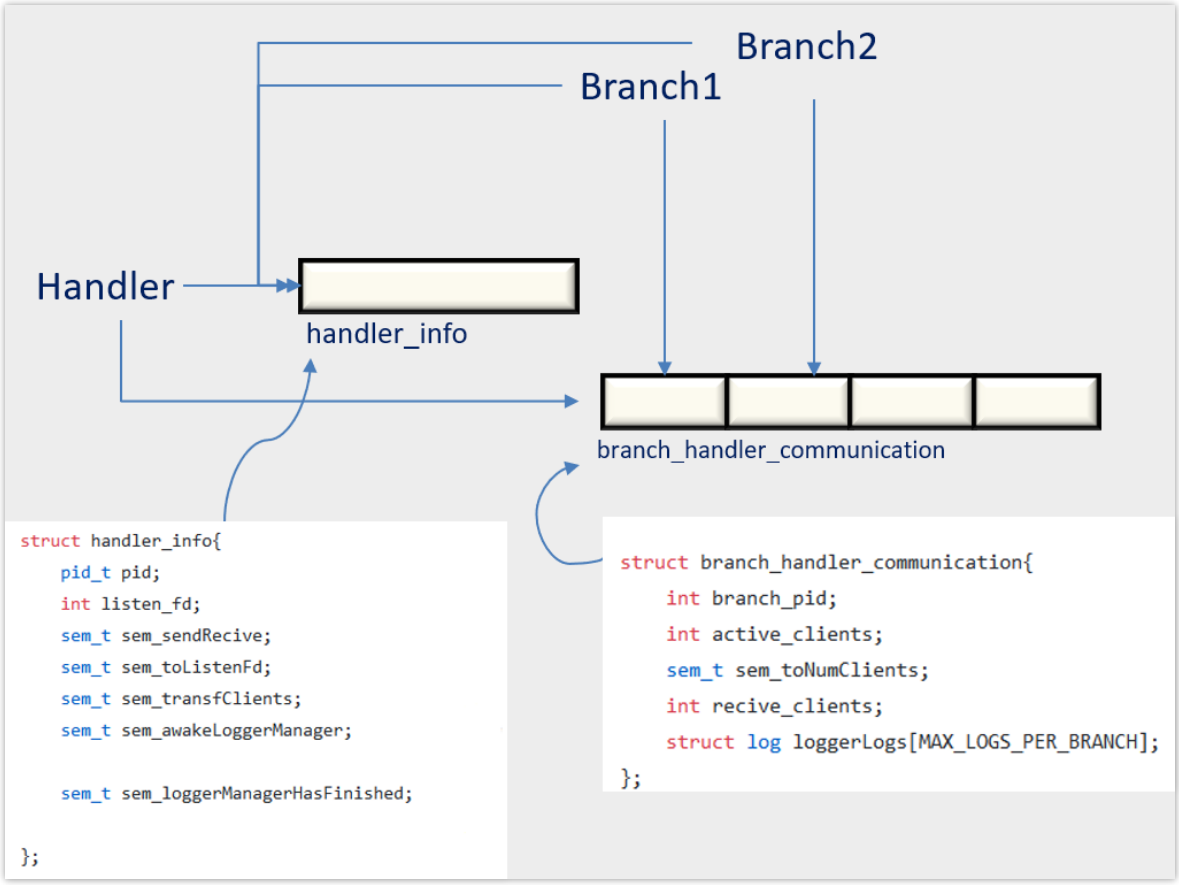


FIGURA 1 SCHEMA DELLA MEMORIA CONDIVISA

L'ultima operazione effettuata è la creazione di un thread, il 'loggerManager' il cui compito sarà quella di scrivere i log dei client raccolti dai 'logger' (thread creati dalle ServerBranch).

Infine, il ServerBranchesHandler attende l'arrivo di segnali (SIGUSR1) provenienti dalle ServerBranch.

Alla ricezione di tale segnale verrà attivato il meccanismo di controllo del numero delle connessioni totali attualmente gestite. Tale meccanismo consentirà di effettuare le seguenti operazioni

1. **Creazione** di una nuova ServerBranch

   Nel caso in cui il numero deli client totali superi una certa percentuale, calcolata sul numero totale dei client gestibili dalle ServerBranch attualmente attive

2. **Merge**, o unione di due ServerBranch

   Effettuata nel caso in cui il numero di client attivi scenda sotto una certa percentuale e il numero di ServerBranch sia superiore al numero di quelle create in partenza. Il merge consiste nel trasferimento (su socket unix) di file descriptor tra la ServerBranch con meno client, l' 'inviatore', e la seconda ad avere meno client, il 'ricevitore'

Schema del merge e creazione

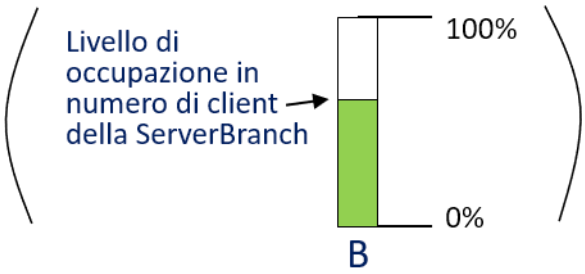


FIGURA 2 LEGENDA SERVERBRANCH

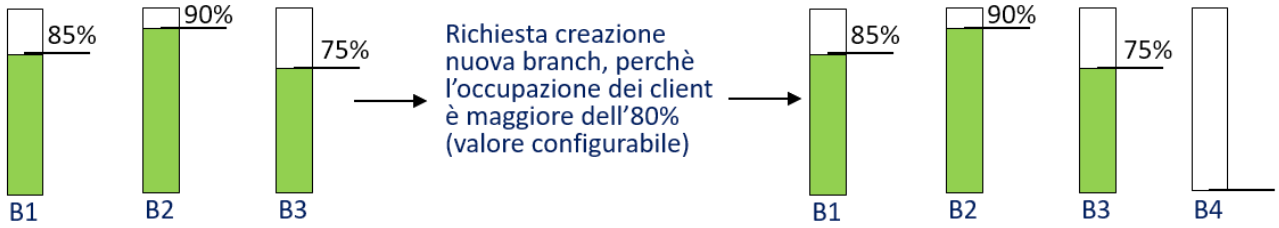


FIGURA 3 CREAZIONE NUOVA BRANCH

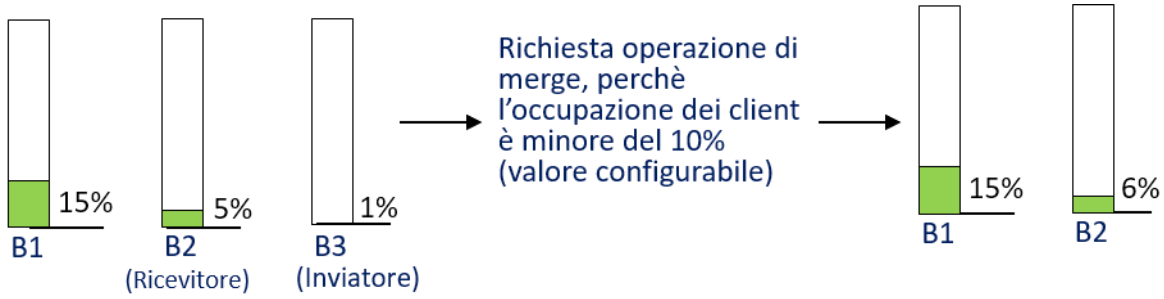


FIGURA 4 MERGE

ServerBranch

Appena viene creata, la ServerBranch (o **branch**) si 'collega' alla memoria condivisa inizializzata dal ServerBranchesHandler, ovvero quella in cui andrà ad inserire le sue informazioni e quella in cui saranno presenti i dati del suo creatore. In tal modo potrà avvenire la comunicazione bidirezionale tra le due componenti.

Procede poi alla creazione di un thread, il 'logger', il cui compito sarà quello di leggere periodicamente i log accumulati dalla branch per poi consegnare tali log al 'loggerManager', il quale li scriverà sul disco.

Prima di entrare nel corpo principale, aggiunge la socket di ascolto all'insieme di file descriptor che verranno usati nella 'select'.

Gestione eventi

Il corpo principale della ServerBranch consiste nell'esecuzione di un loop nel quale aspetta che ritorni la primitiva 'select'. In particolare

- Se la select ritorna perché un client sta tentando di connettersi al server, la branch esegue i seguenti passi
    1. Prova ad acquisire la socket di ascolto (mediante una trywait sul semaforo condiviso da tutte le branch) per stabilire una connessione con il client
    2. Se acquisisce il semaforo, inserisce i dati del client in una struttura dati (e la memorizza in una lista doppiamente collegata) con le seguenti informazioni
        - Indirizzo del client (di tipo sockaddr_in)
        - File descriptor della socket ad esso associate
        - Tempo relativo all'ultima interazione con il server
    3. Se non acquisisce il semaforo, verifica se vi siano altri client ad aver inviato delle richieste
- Se la select ritorna perché un client ha inviato una richiesta (compresa la chiusura di connessione), viene scandita la lista doppiamente collegata in cui la branch salva i dati dei client finché non viene gestita la richiesta del client che l'ha inviata

Livello di occupazione

Ogni qualvolta che una branch stabilisce una connessione con un client o ne rimuove uno, verifica la variazione della percentuale del numero delle connessioni.

Se dopo l'arrivo di un certo numero di client, la percentuale di occupazione della ServerBranch varia (spostandosi in uno di questi intervalli configurabili: [0%, 10%], [10%, 50%], [50%, 80%], [80%, 100%]), allora

- Invia al ServerBranchHandler il segnale 'SIGUSR1', che innescherà il meccanismo che deciderà di creare nuove branch o effettuarne il merge
- Cambia la sua priorità in base alla percentuale di occupazione della branch. Il livello di priorità assegnato sarà proporzionale al numero di client gestiti dalla branch

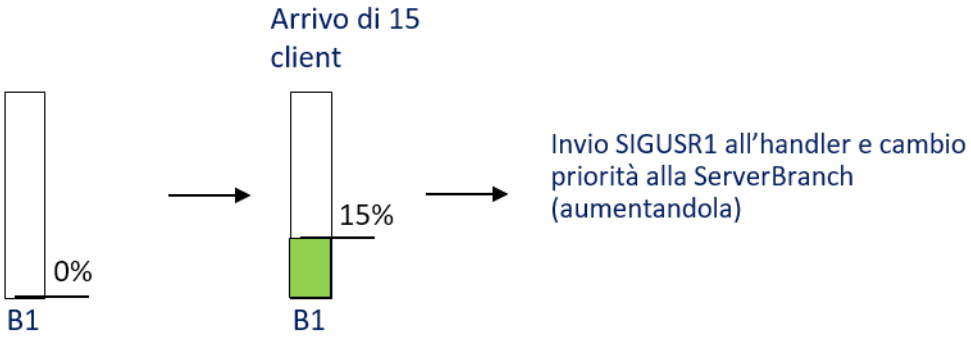


FIGURA 5 ESEMPIO CONTROLLO LIVELLO DI OCCUPAZIONE ALL'AUMENTARE DELLE RICHIESTE

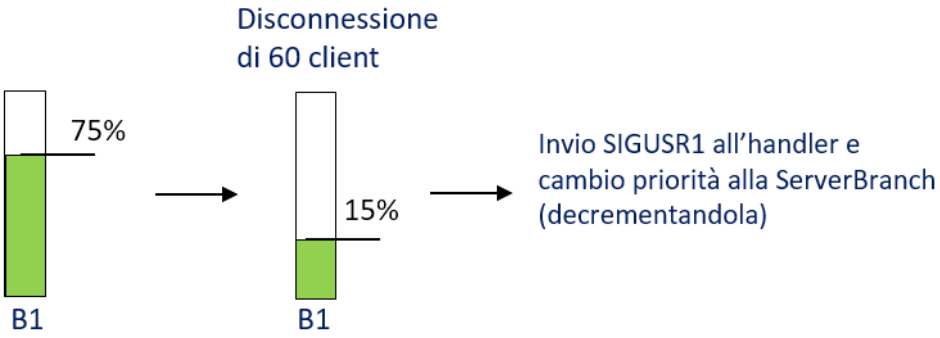


FIGURA 6 ESEMPIO CONTROLLO LIVELLO DI OCCUPAZIONE AL DIMINUIRE DELLE RICHIESTE

Cleaner

Per ottimizzare l'utilizzo delle risorse della ServerBranch, viene fatto partire un timer che fa partire un **cleaner** (un handler del segnale 'SIGALARM'). Il suo scopo è scandire la lista collegata in cui sono conservate le informazioni dei client connessi e chiudere la connessione con tutti quei client che hanno un tempo di ultima interazione con il server 'vecchio', ovvero che non interagiscono con il server per più di un certo periodo di tempo.

Trasferimento client

La ServerBranch è configurata per rispondere al segnale 'SIGUSR2', che le verrà inviato dal suo creatore per comunicarle di effettuare l'operazione di merge. All'interno della memoria condivisa l'handler 'comunicherà' alla branch se avrà il compito di 'inviatore' o 'ricevitore' di client. In particolare

- Se sarà l'**inviatore**, aprirà una socket unix sulla quale verranno inviati I file descriptor delle socket tra I due processi, comunicherà al ricevitore dell'apertura della socket e al termine del trasferimento comunicherà all'handler che l'operazione è stata completata, per poi terminare
- Se sarà il **ricevitore**, aspetterà che la branch con il compito di inviare I client, apra la socket unix. Tale sincronizzazione avviene sfruttando I semafori nella memoria condivisa in cui sono presenti le informazioni dell'handler. Dopodiché procede a ricevere I client, ovvero I file descriptor delle socket, uno per volta e al termine di tale operazione comunicherà all'handler del completamento dell'operazione. Tale comunicazione, assieme a quella dell'inviatore, sbloccherà l'handler, che rimarrà in attesa durante l'operazione di merge

Gestione richieste

La gestione della richiesta del client consiste in

1. Analisi "formale" della richiesta
2. Riconoscimento del contenuto statico richiesto
3. Riconoscimento del dispositivo da cui è generate la richiesta
4. Riadattamento (se non presente in cache) del contenuto statico in base alle caratteristiche del dispositivo
5. Salvataggio del contenuto statico in cache (se non era già presente) ed invio della risposta
6. Aggiornamento ultimo tempo di interazione del client con il server

Analisi "formale" della richiesta

La richiesta ricevuta viene passata alla funzione 'parsingManager' che innanzitutto la analizza controllando che sia conforme alle direttive del protocollo http, salvando eventuali errori trovati, richieste non supportate (come nel caso di richieste con metodi differenti dal GET o HEAD) o informazioni necessarie per il corretto adempimento della richiesta all'interno della struttura "paramResponse".

Riconoscimento contenuto statico

Tramite analisi del path presente nella richiesta, il parsingManager comprende se l'utente desidera la Homepage oppure un'immagine. Nel secondo caso avviene un'ulteriore analisi delle header lines alla ricerca dell'Accept, da cui prelevare i formati accettati o preferiti dall'utente con il relativo *'quality factor',* 'q', e dell'User Agent, contenente informazioni riguardanti il suo browser e il suo dispositivo

Riconoscimento del dispositivo

Il riconoscimento del dispositivo avviene tramite la libreria libwurfl, insieme ad un file .Xml contenente molte informazioni relative alla maggior parte dei dispositivi, fra cui la risoluzione dello schermo. Le ServerBranch appena inizializzate si occupano di mantenere aperto il file .Xml al quale si accede con una funzione di lookup sull'User Agent per individuare il dispositivo

Riadattamento contenuto statico

Una volta interpretata correttamente la richiesta ed individuato il dispositivo viene effettuata una ricerca in Cache dell'immagine.

Nel caso in cui non viene trovata allora viene effettuato il riadattamento attraverso la libreria di funzioni software di ImageMagick.

Il riadattamento dell'immagine prevede, in base alla richiesta del client, resizing delle immagini ,conversione del formato, e compressione di un fattore percentuale pari a 'q'.

Salvataggio in cache e invio della risposta

Se l'immagine è stata riadattata, viene richiesto al CacheManager (che vedremo in seguito) di salvare in Cache l'immagine. In parallelo la parsingManager prosegue e, se non sono stati riscontrati errori, costruisce l'Header basandosi sui dati salvati nella struct "paramResponse" e poi, se richiesto il metodo GET, gli concatena un array contenente i byte dell'immagine.

In caso di errore viene invece interrotta la normale esecuzione di parsingManager e viene ritornata una risposta per l'utente contenente un Header compilato con lo status Code adeguato ed una semplice pagina HTML che permette al client di visualizzare a schermo lo status Code di errore.

Logging

L'operazione di logging in Exodus è messa in atto da due tipologie di thread in modo tale per cui la scrittura su disco dei log non vada a bloccare o rallentare le singole ServerBranch. Tali thread sono

- **logger**, thread presente per ogni singola ServerBranch
- **loggerManager**, creato dal ServerBranchesHandler

Un 'log' è salvato come una struttura dati con le seguenti informazioni

1. Tempo in cui è stato registrato il log: espresso in secondi dal 1° gennaio 1970
2. Tipo del log generato: un valore intero usato poi dal loggerManager per tradurlo nella stringa opportuna
3. Indirizzo del client (del tipo struct sockaddr_in)
4. Stringa contenente ulteriori informazioni, come il messaggio d'errore se il tipo del log è un 'INTERNAL_SERVER_LOG' oppure la pagina richiesta del client se il log è del tipo 'CLIENT_SERVED'

Ogni ServerBranch ha un array di log, locale alla branch stessa (salvata nel suo stack).

Nella struttura dati di tipo 'branch_handler_communication' è invece salvato un array di log, condiviso tra branch ed handler (si ricorda che ogni struttura dati di questo tipo è riempita da ogni server branch nella memoria condivisa inizializzata dall'handler). La singola ServerBranch andrà a popolare l'array locale di logs, e il logger, dopo un certo periodo di tempo, andrà a copiare l'array locale nell'array condiviso.

La **distinzione** tra questi array consente alla ServerBranch di scrivere nuovi log, senza aspettare che logger e loggerManager finiscano l'operazione di scrittura su disco.

In questo modo ogni ServerBranch dovrà aspettare solamente che il 'proprio' logger copi l'array locale di log in quello condiviso.

In particolare, le ServerBranch potranno inserire nuovi log mediante la funzione '**LOG'**, passando ad essa i parametri: tipo del log e indirizzo del client ed eventuale stringa contenente altre informazioni.

Logger

Una volta avviato dalla ServerBranch, il compito del logger consiste nell'eseguire il seguente loop

1. Aspetta un certo numero di secondi
2. Acquisisce il semaforo sull'array locale di log (quello popolato dalle ServerBranch)

3. Copia tale array nell'array presente in memoria condivisa e azzera l'array locale, rilasciando il semaforo acquisito precedentemente (da questo momento le branch possono continuare la loro esecuzione senza aspettare il logger e loggerManager)
4. Incrementa il valore di un semaforo presente nella memoria condivisa in cui sono presenti le informazioni del ServerBranchesHandler. Tale operazione serve per notificare il loggerManager che il logger ha finito di prelevare i log della branch
5. Effettua una wait su un semaforo simile a quello precedente, che verrà incrementato dal loggerManager quando quest'ultimo avrà finito di manipolare i log posseduti dai logger per poi rilasciarli

LoggerManager

Il punto di forza del loggerManager è che è un thread creato dal ServerBranchesHandler dunque ha accesso a tutti i suoi dati. Questo consente al loggerManager di avere accesso alle informazioni di ciascuna ServerBranch, e dunque dei suoi logger.

Per garantire il corretto funzionamento dell'operazione di logging in un server il cui numero di ServerBranch, e dunque di logger, è dinamico, è necessario

- un loggerManager che sappia quanti logger 'aspettare' per l'acquisizione dei log
- un ServerBranchesHandler che non deve effettuare merge dopo che il loggerManager ha deciso quanti logger aspettare. Un merge in questa fase potrebbe causare la perdita dei log della branch eliminata o deadlock (N.B.: la creazione di nuove branch non è un problema, perché una volta noto il numero di logger da aspettare, verranno 'scandite' le informazioni dello stesso numero di branch)

In particolare, il loggerManager esegue il seguente loop

1. Acquisisce un semaforo che impedirà all'handler di effettuare eventuali operazioni di merge
2. Acquisisce dall'handler il numero di ServerBranch, e dunque di logger, attualmente attive
3. Effettua tante operazioni di wait quante sono le ServerBranch attive, su un semaforo in memoria condivisa, che verrà incrementato dai logger quando avranno prelevato i log dalle ServerBranch. In tal modo il loggerManager procederà con i passi successivi solo quando tutti i logger avranno finito

4. Preleva i log dei logger e li ordina cronologicamente e li converte in stringhe testuali in base al 'log_type'
5. Rilascia il semaforo acquisito al punto 1.
6. Scrive i log ordinati su disco
7. Rilascia i logger incrementando un semaforo in memoria condivisa, sul quale i logger si erano messi in attesa

Scrittura dei log su disco

Contando di mantenere il server costantemente aperto, se il loggerManager andasse a scrivere sempre nello stesso file di logger si andrebbe a creare un file di grandi dimensioni.

Per risolvere tale problema si è scelto di non superare un certo numero di byte per ogni file di log, così nel momento in cui si supererà tale valore verrà creato un uovo file di log.

A questo punto per evitare che il numero dei file di log cresca in modo incontrollato, si è fissato un numero massimo di file che il loggerManager potrà creare.

Quindi nel caso in cui si arrivi a questo numero, i file verranno sovrascritti partendo dal più obsoleto.

Ordinamento dei log

Per l'ordinamento dei log, il loggerManager scorre la lista doppiamente collegata (posseduta dall'handler) in cui sono presenti le informazioni delle ServerBranch (tra cui l'array di log popolati dai logger). È bene notare che tali array sono ordinati cronologicamente, poiché ogni ServerBranch elabora la richiesta dei client uno alla volta.

L'algoritmo d'ordinamento utilizzato consiste quindi nell'ordinare il primo elemento (il log più vecchio) dell'array di log di tutti i logger.

È mantenuto dunque un vettore ('branchIndex') di valori interi, di dimensione pari al numero di logger, che verrà utilizzato per mantenere la posizione dell'array di log del giusto logger. Se ad esempio branchIndex[4] = 6, al quinto logger sono stati prelevati 5 log.

E inoltre mantenuto un altro vettore di booleani ('branchCompleted') che indica se ci sono ancora log nell'array di log dei logger. Se, ad esempio, branchCompleted[5] = 1, significa che non ci sono più log da ordinare per il sesto logger.

Il logger manager scorre dunque la lista collegata finché branchCompleted ha tutti valori pari ad 1.

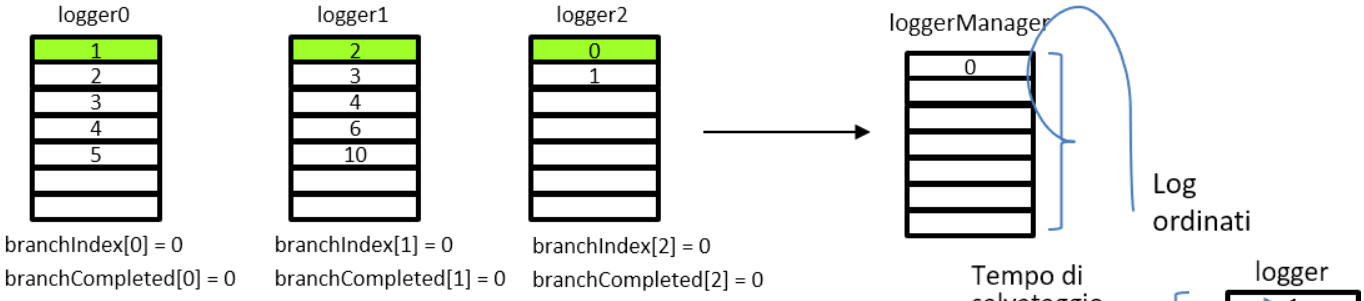


FIGURA 7 ESEMPIO ORDINAMENTO LOGGER PASSO 0

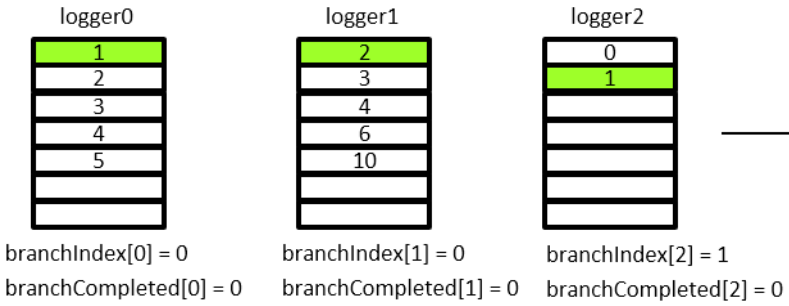


FIGURA 8 ESEMPIO ORDINAMENTO LOGGER PASSO 1

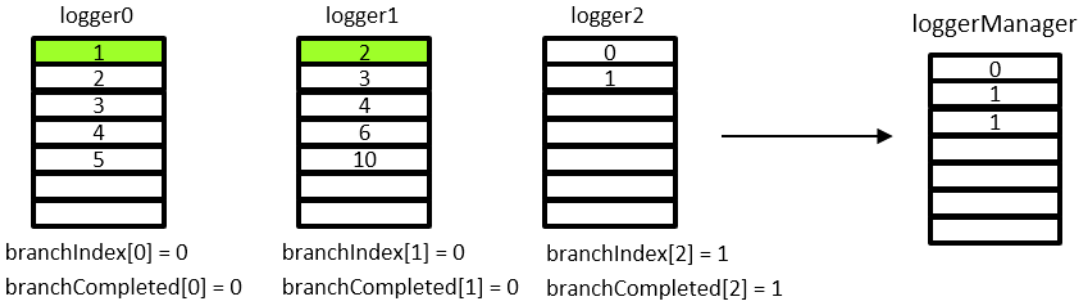


FIGURA 9 ESEMPIO ORDINAMENTO LOGGER PASSO 2

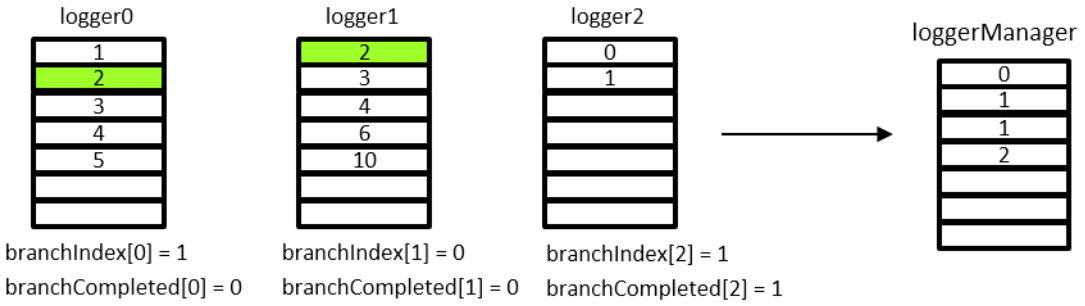


FIGURA 10 ESEMPIO ORDINAMENTO LOGGER PASSO 3

Cache

La cache implementata presenta come struttura di base un array, utilizzato come tabella hash, contenente un certo numero di elementi (strutture di tipo 'hash_element') ognuno dei quali contenente due semafori per sincronizzarne l'accesso e una lista di collisione, ovvero un array di strutture di tipo 'image'.

L'accesso ai singoli elementi della cache avviene utilizzando una funzione Hash calcolata sui dati della struttura 'image'.

L'idea è quella di sfruttare la velocità di una struttura hash ad accesso diretto combinata con la possibilità di mantenere le immagini con la stessa chiave in una coda con contatore sui riferimenti. Possiamo suddividere la cache in due azioni, la scrittura, gestita dal thread CacheManager, e la lettura, gestita invece dalla funzione 'getImageInCache'.

```
struct image{
    char name[MAX_IMAGENAME_DIM];
    char quality;
    unsigned short width;
    unsigned short height;
    char isPng;
    char imageBytes[MAX_IMAGE_DIM];
    int imageSize;

    unsigned int counter;
};
```

FIGURA 12 STRUCT IMAGE

```
struct hash_element{

    struct image conflictingImages[MAX_IMAGE_NUM_PER_KEY];
    sem_t semToHashBlock;
    sem_t dontRead;
};
```

FIGURA 11 STRUCT HASH_ELEMENT

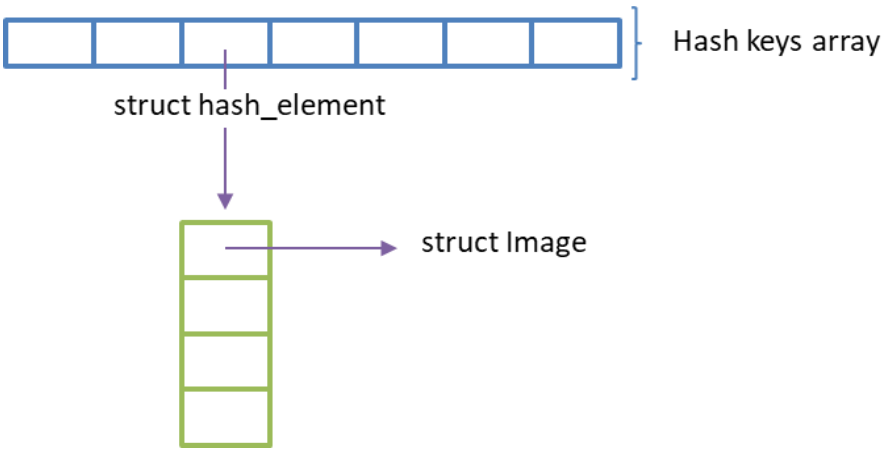


FIGURA 13 SEMPLICE RAPPRESENTAZIONE DELLA CACHE

CacheManager

Questa funzione, eseguita da un thread, presenta tre singoli elementi, due semafori necessari a sincronizzarsi con la ServerBranch mentre esegue la funzione 'parsingManager' e la funzione di Insert. Quest'ultima come prima cosa si calcola la chiave hash relativa all'immagine da salvare in memoria e poi effettua una wait sui due semafori dell'hash_element della posizione appena calcolata nell'array. Si scorre quindi l'array interno e se vi sono spazi liberi si posiziona lì l'immagine, in alternativa controlla i counter delle immagini e sostituisce quella che ha ricevuto meno accessi. Fatto ciò libera i semafori su cui aveva fatto una wait e ritorna il controllo alla cacheManager.

GetImageInCache

Come nella insert, la prima cosa che avviene è il calcolo della chiave hash da ricercare. Una volta fatto ciò viene fatto un controllo sui semafori di hash_element per controllare se c'è qualcosa in scrittura in quel particolare array interno. Dunque la funzione si scorre la lista alla ricerca dell'immagine ricercata. Se non la trova ritorna il controllo alla parsingManager, se invece la trova, incrementa di uno il contatore presente nella struct dell'immagine e poi ritorna.

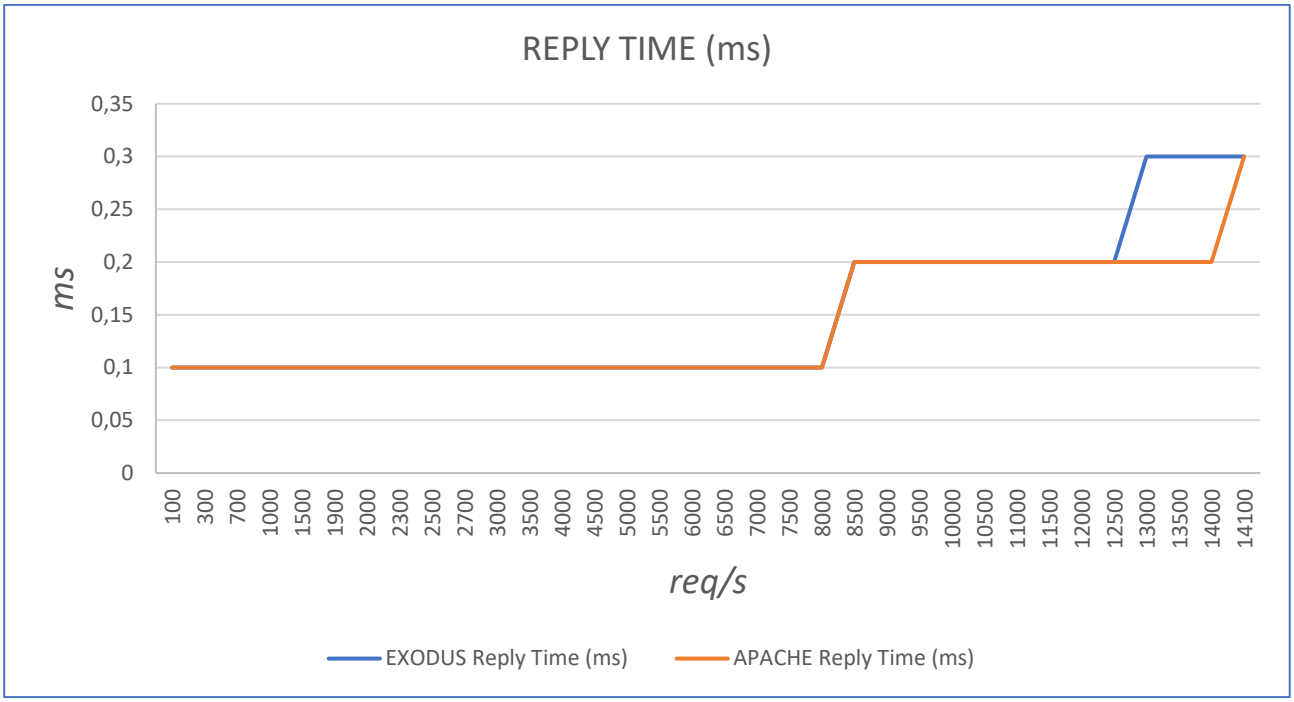
Struttura dei semafori della Cache

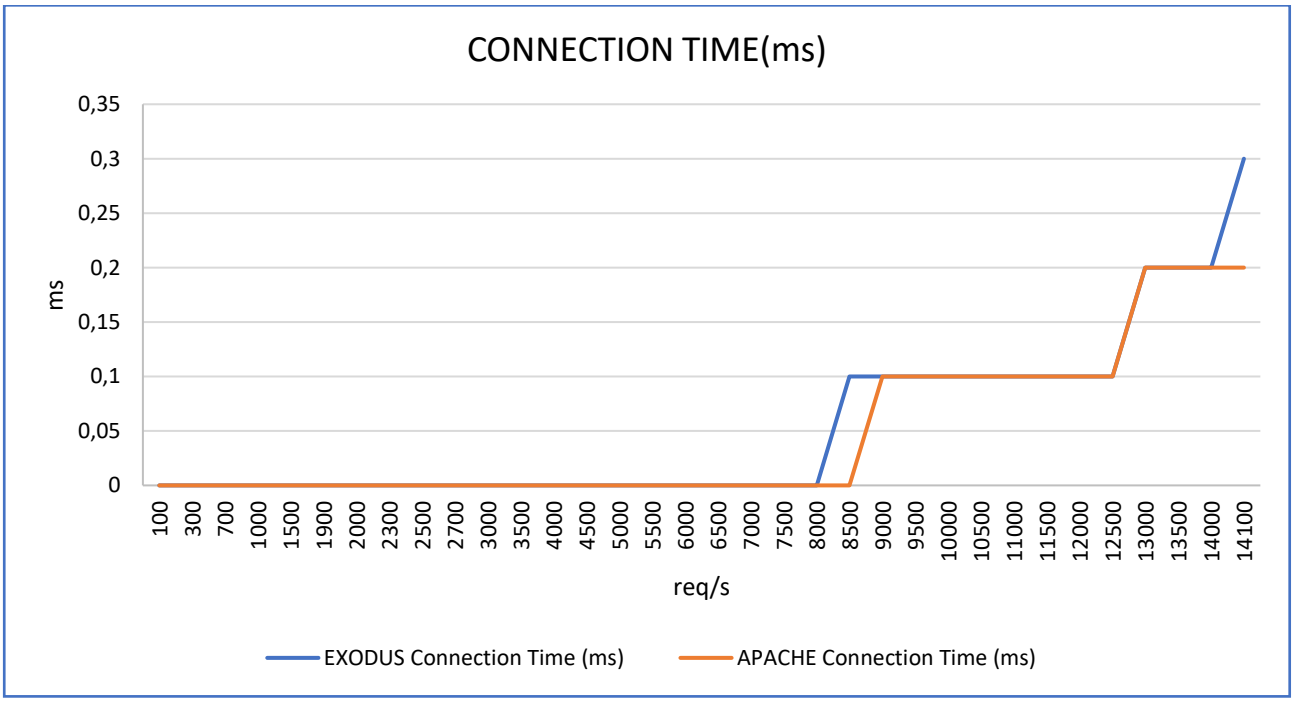
Legenda: ACM → activateCacheManager = 0
FCM → cacheManagerHasFinished = 1
NOR → dontRead = 1
HBK → semToHashBlock = 1

parsingManager	cacheManager (insert)	getImageInCache
…….	WAIT(ACM)	Calcola la chiave Hash
WAIT(FCM)	Calcola la chiave Hash	TRYWAIT(HBL)
Inizializzo la struct image	WAIT(HBL)	WAIT(NOR)
getImageInCache(struct* image)	WAIT(NOR)	POST(NOR)
…….	Inserisce l'immagine	Ricerco l'immagine
POST(ACM)	POST(NOR)	If (errno != EAGAIN):
…….	POST(HBK)	POST(HBL)
	POST(FCM)	

Test

Prestazioni

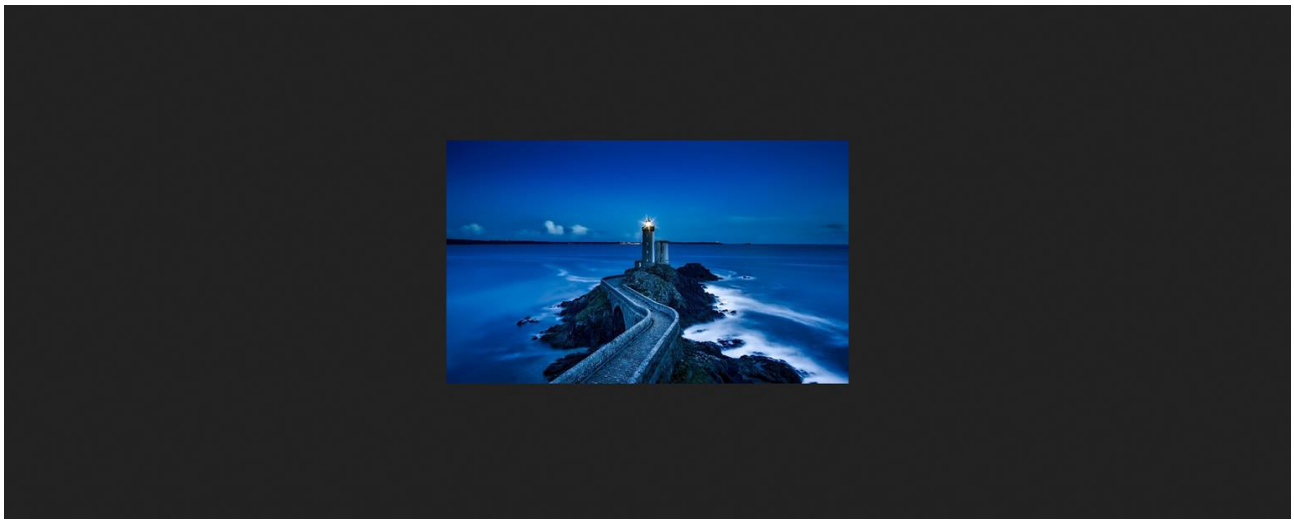




I test proposti sono stati eseguiti inizializzando 5 Branch, ognuna con un massimo di 1024 client gestibili

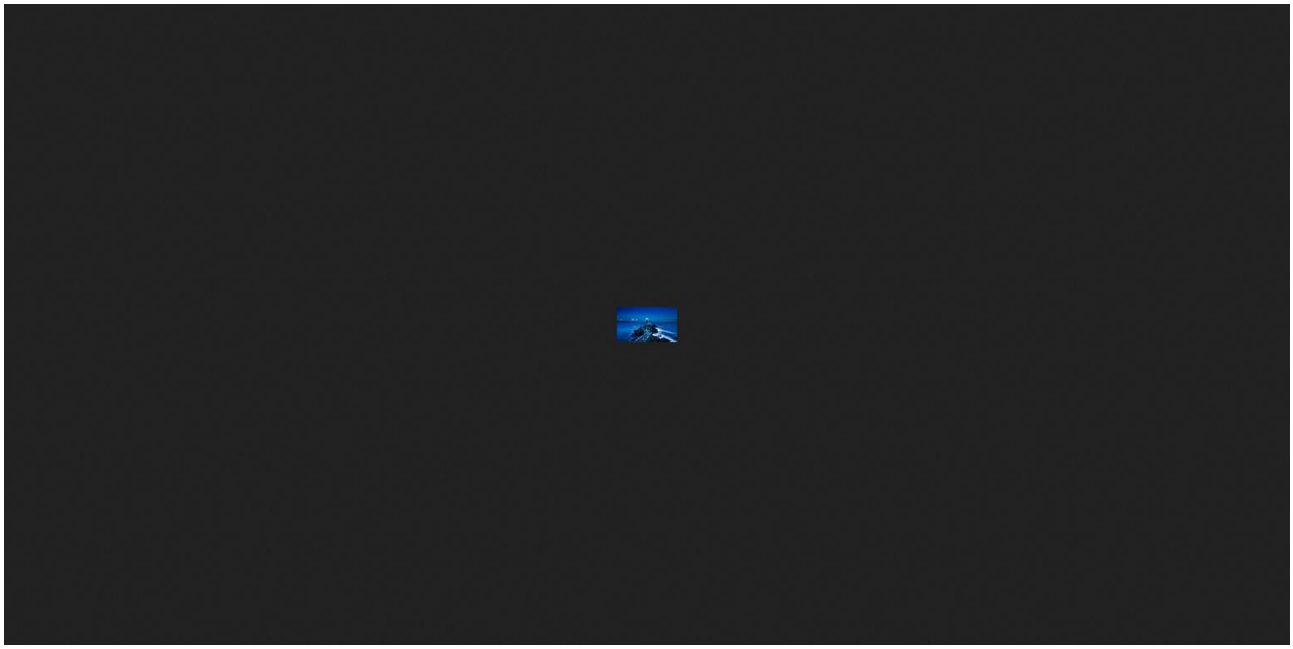
Riadattamento

Tramite l'ausilio dell'estensione di Firefox "User-Agent Switcher and Manager", abbiamo avuto modo di testare richieste con vari User-Agent relativi a dispositivi con diverse risoluzioni



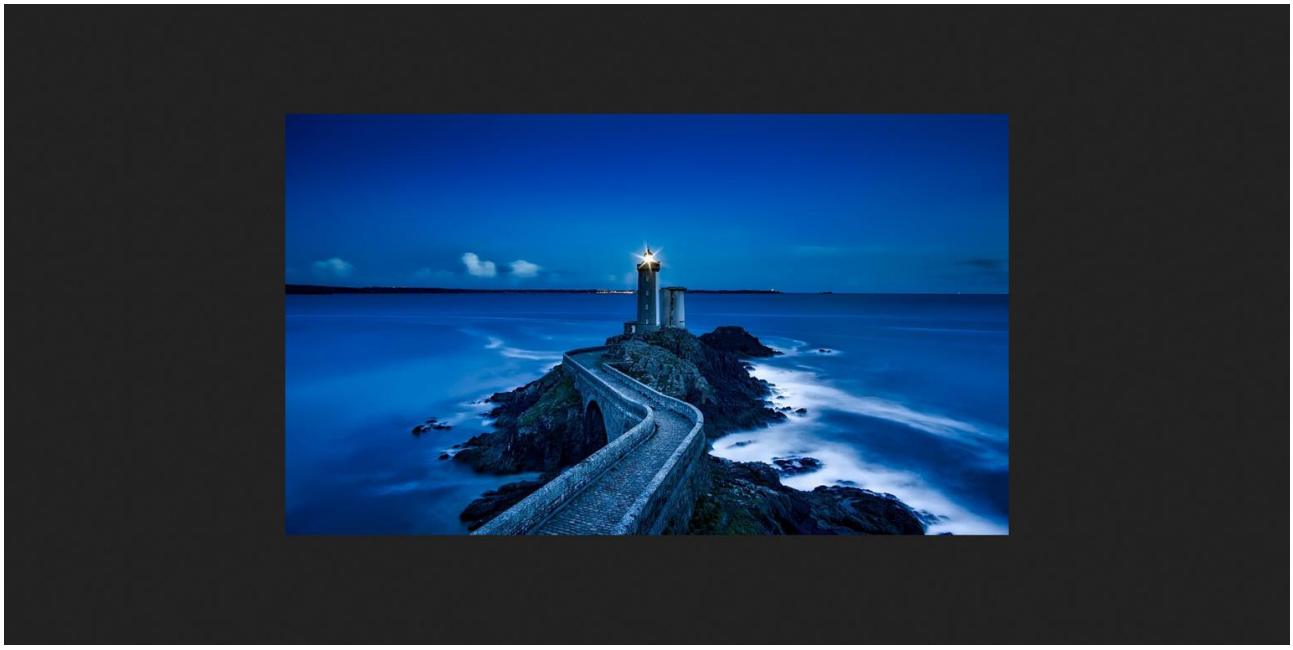
User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3790.0 Safari/537.36

Risoluzione Dispositivo 600 x 800(Risoluzione di Default in dispositivi non riconosciuti)



User Agent: CDM-8600/T10 UP.Browser/5.0.5 (GUI)

Risoluzione dispositivo: 120 x 128



User Agent: Mozilla/5.0 (Linux; Android 5.1; X6s Build/LMY47I) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/39.0.0.0 Mobile Safari/537.36

Resolution phone: 1080 x1920

User-Agent: Mozilla/5.0 (Linux; Android 4.4.4; ELITE TAB 9.7 RETINA WIFI Build/KTU84P) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/33.0.0.0 Safari/537.36

Resolution phone: 1536 x 2048

Limitazioni

ServerBranchesHandler

La principale limitazione a questo tipo di architettura si riscontra nel momento in cui la ServerBranchesHandler termina in modo inaspettato. Se dovesse accadere, le operazioni di Merge\Creazione delle ServerBranch e Logging, non verrebbero più eseguite.

Tuttavia, bisogna tenere comunque in considerazione che in questa eventualità le ServerBranch attive resterebbero continuerebbero a gestire le richieste dei client, lasciando così il server in grado di gestire, anche se in numero e funzionalità limitata, tutte le richieste in attesa di un riavvio del server

Logging

L'implementazione del logging fa fronte a due limitazioni

- La grandezza dell'array locale, presente in ogni ServerBranch, inizializzata dal ServerBranchesHandler. Nel caso in cui fosse molto piccola, c'è la possibilità che fra un'attivazione e l'altra del loggerManager, vengano scritti più log di quanto la ServerBranch possa mantenere perdendo quindi tutte quelle informazioni successive al riempimento dell'array
- Possibile perdita di qualche log in caso di merge consecutivi. Teoricamente possibile se dopo un merge vengono scritti dei dati in un'altra branch di cui andrà fatto un merge poiché non è stato fatto prima il controllo da parte del loggerManager. In pratica però questo caso risulta estremamente difficile da causare poiché il loggerManager dopo il primo merge è già in ascolto sul semaforo, quindi potrebbe accadere solo se per qualche motivo l'handler resta in scheduling per molto tempo senza dare il controllo al loggerManager

Cache

La limitazione della cache è data dalla grandezza fissa dei blocchi che compongono l'array interno alla hash_element, che porta quindi a poter lavorare con immagini che non siano più grandi di una dimensione decisa dall'host in fase di settaggio del server.

Questa limitazione viene fuori da una scelta fatta per far fronte al problema della frammentazione esterna. Comunque, la cache è stata creata in modo da poter settare tutti i parametri secondo le preferenze dell'utente potendo così gestire sia casi in cui si vuole dare priorità alla grandezza delle immagini, sia casi in cui si vuole dare priorità alla velocità di accesso nella cache o al mantenimento di un maggior numero di immagini in quest'ultima.

Note finali

Programmi utilizzati per l'implementazione di Exodus:

- CLion
- Wurfl
- ImageMagick

Per la realizzazione dello scambio di file descriptor su socket UNIX si è utilizzato il codice presente nella guida consultabile qui: https://keithp.com/blogs/fd-passing/