Distributed File Systems

Pironti Antonio (antonio.pironti@gmail.com),
Russo Vincenzo (vincenzo.russo@neminis.org)

14 Settembre 2005

Sommario

L'intento di questo articolo è quello di fornire una panoramica sul mondo dei File System Distribuiti, mostrandone le caratteristiche peculiari, i requisiti e al contempo le problematiche da affrontare. L'introduzione generale sarà arricchita successivamente con tre casi di studio principali: Andrew File System (AFS), Coda e InterMezzo, tre file system distribuiti che, senza troppi giri di parole, possiamo definire imparentati (Coda nasce da AFS2 e InterMezzo è invece una reingegnerizzazione di Coda ed è il più giovane dei tre DFS). Proprio sulla scia che porta da Coda a InterMezzo viene più avanti trattato un argomento delicato per i file system distribuiti: la risoluzione dei *bottlenecks* (colli di bottiglia).

Per concludere verranno anche trattati alcuni internals sia della parte client che della parte server del file system Coda.

In appendice, sarà invece brevemente trattato il Mosix File System, caso di studio per la classi di file system DFSA (Direct File Sytem Access).

Indice

Elenco delle figure

1 I File System Distribuiti

Con la nascita delle reti, l'evoluzione dei sistemi, il volgere verso la dislocazione fisica e logica delle unità di elaborazione e l'evolversi dei sistemi distribuiti, nasce l'enorme esigenza di poter condividere dati e risorse di memorizzazione usando un file system comune.

Un *file system distribuito* (DFS) è un'implementazione distribuita del modello classico di un file system in un sistema time-sharing, dove più utenti condividono file e risorse di memoria. Tale tipo di file system consente di soddisfare l'esigenza sopra citata, dando una nuova marcia ai sistemi distribuiti.

Una tipica configurazione per un DFS è quella di un insieme di workstation e server interconnessi da una LAN; il DFS può essere implementato come parte del sistema operativo di ogni nodo connesso a tale rete.

Per descrivere e comprendere al meglio la struttura di un DFS occorre definire i termini *macchina*, *servizio*, *server* e *client*.

Il termine *macchina* viene utilizzato per indicare sia un server che una worstation; una determinata macchina di un sistema distribuito considera remote le restanti macchine e le rispettive risorse, mentre considera locali le proprie risorse.

Il *servizio* è un'entità software in esecuzione su una o più macchine e fornisce un tipo particolare di funzione a client sconosciuti a priori.

Il *server* è il software di servizio su una singola macchina.

Il *client* è un processo che può richiedere un servizio, attraverso la cosiddetta *interfaccia del client.*

Un DFS è, pertanto, un file system i cui client, server e dispositivi di memoria sono sparsi tra le macchine di un sistema distribuito; di conseguenza, l'attività di servizio deve essere eseguita attraverso la rete e i dispositivi di memorizzazione sono svariati e indipendenti. La configurazione concreta di un DFS può essere di vario tipo, come ad esempio, una configurazione con server su macchine dedicate oppure configurazioni in cui una macchina può essere sia server che client. Anche per quanto riguarda l'implementazione abbiamo svariate scelte: implementare il DFS come parte di un sistema operativo distribuito oppure come uno strato software che si occupa della gestione della comunicazione tra i sistemi operativi convenzionali e i file system.

1.1 Tipologie di servizio

Esistono due approcci sul come porsi rispetto alle informazioni del server:

- il server tiene traccia di ogni file al quale ha accesso ogni client
- il server si limita a fornire i blocchi secondo quanto richiesto dal client senza sapere che utilizzo ne verrà fatto.

Nel primo caso parleremo di *servizio con informazione di stato (statefull)*, mentre nel secondo diremo che si tratta di un servizio *senza informazione di stato (stateless)*.

1.1.1 Servizio con informazione di stato

Prima di accedere a un file, un client deve eseguire un'operazione di apertura su quel file. Il server preleva dal suo disco alcune informazioni sul file, le registra nella propria memoria e fornisce al client un identificatore di connessione, unico per quel client e quel file aperto. Questo identificatore viene usato per gli accessi successivi, fino al termine della sessione. Un servizio con informazione di stato viene descritto come una connessione tra il client e il server durante una sessione. Alla chiusura del file, oppure tramite un meccanismo di garbage collection, il server può reclamare lo spazio di memoria principale utilizzato da client che non sono più attivi.

Il vantaggio offerto dal servizio con informazione di stato è dato dalle migliori prestazioni. L'informazione su di un file è sottoposta a caching in memoria centrale ed è possibile accedervi tramite l'identificatore di connessione, risparmiando così accessi al disco. Inoltre un server con informazione di stato è in grado di sapere se un file è stato aperto per accesso sequenziale e può quindi effettuare un *pre-fetching* dei blocchi successivi.

1.1.2 Servizio senza informazione di stato

Un server stateless evita l'informazione di stato, rendendo ogni richiesta autocontenuta. Ciò significa che ogni richiesta identifica completamente il file e la posizione nel file (per accessi di lettura e scrittura). Il server non deve tenere in memoria principale una tabella di file aperti, anche se ciò in effetti avviene per motivi di efficienza. Inoltre non è necessario stabilire e terminare una connessione per mezzo delle operazioni di apertura e chiusura. Queste operazioni risultano del tutto superflue, in quanto ogni operazione su file è indipendente e non viene considerata parte di una sessione. Un processo client apre un file, ma tale apertura non causa la trasmissione di un messaggio remoto. Le letture e le scritture hanno luogo naturalmente come messaggi remoti, oppure come ricerche nella cache. La chiusura finale da parte del client è (come la `open()`) soltanto un'operazione locale. Per ciò che concerne invece le prestazioni, è facile immaginare come queste possano essere inferiori a quelle di un server statefull; le motivazioni sono semplici:

- i messaggi di richiesta sono più lunghi e quindi si ha più overhead di rete
- L'elaborazione delle richieste è più lenta, in quanto non esiste alcuna informazione in memoria centrale utile al fine di accelerare l'elaborazione

1.1.3 Fault tollerance

Come si può facilmente immaginare, la tolleranza ai guasti dei due tipi di servizio è alquanto diversa.

- Servizio *statefull*: in caso di guasti al server, si necessita di un *protocollo di ripristino dello stato precedente*, basato su un dialogo con i client o su un meno elegante *abort* di tutte le operazioni che erano in corso durante il verificarsi del guasto al server. Nel caso di guasti ai client, questi devono essere notificati al server, per dar via alla fase conosciuta come *rilevamento ed eliminazione degli orfani*. La conclusione è che il fault tollerance risulta essere un processo alquanto tedioso.
- Servizio *stateless*: un server senza informazioni di stato non presenta gli stessi problemi di cui sopra ed è in grado, subito dopo il ripristino, di rispondere senza problemi alle richieste, essendo esse stesse autocontenute. Una peculiarità è che dal punto di vista dei client, non esiste alcuna differenza tra un server lento e uno in via di ripristino.

1.1.4 Conclusioni

Un server statefull risulta essere più performante, ma a suo discapito giocano le procedure di fault tollerance troppo onerose in termini di tempo; d'altro canto un server stateless pur non avendo particolari problemi con il fault tollerance, risulta essere prestazionalmente inferiore alla prima tipologia. Come in ogni caso, la scelta è dettata dalla particolare esigenza del caso, anche se possiamo affermare che in gran parte si preferisce una migliore tolleranza ai guasti piuttosto che prestazioni estreme.

1.2 Requisiti di un File System Distribuito

I primi requisiti di un file system distribuito sono esattamente i medesimi di un file system per sistemi multiutente e multiprogrammati:

- *struttura dei nomi consistente*
- *interfaccia per le applicazioni*
- *mapping dei nomi*
- *garanzia di integrità dei dati*
- *sicurezza*
- *controllo di concorrenza*

Ai requisiti classici sopra descritti, vanno aggiunti requisiti intrinseci dei DFS:

- *interfaccia per l'allocazione dei file sui nodi della rete*
- *disponibilità del servizio di File Sharing anche in presenza di reti non perfettamente affidabili*

Ai requisiti appena elencati, se ne aggiunge uno particolarmente importante per un file system distribuito: la *trasparenza*.

La crucialità di questo requisito è dovuta al fatto che un DFS deve apparire ai client come un convenzionale file system centralizzato:

- la molteplicità e la dispersione dei server e dei dispositivi di memorizzazione devono essere rese trasparenti
- l'interfaccia dei client di un DFS non deve distinguere tra i file locali e file remoti. Ciò nei moderni sistemi operativi è reso possibile dalla presenza del VFS (Virtual File System)
- è compito del DFS localizzare i file e predisporre il trasporto dei dati

La trasparenza è una caratteristica poliedrica:

- *trasparenza di piattaforma:* il DFS deve permettere la condivisione a prescindere da sistema operativo e hardware delle singole unità di rete
- *trasparenza di accesso:* gli utenti non devono curarsi del tipo di accesso (locale o remoto) ad un file system
- *trasparenza di locazione:* il nome di un file non deve rivelare alcuna informazione sull'effettiva locazione fisica del file e l'URI non deve essere modificata se cambia la posizione fisica dello stesso
- *trasparenza di mobilità:* i programmi devono funzionare anche se vengono spostati da un server all'altro (previa compatibilità di architettura e supporto per il formato eseguibile)
- *trasparenza delle prestazioni:* le prestazioni di un DFS, dal punto di vista del client, devono essere paragonabili a quelle di un file system locale; inoltre le prestazioni devono poter essere aumentate per far fronte ad aumenti di carico di lavoro (*scalabilità*)

1.3 Semantica della consistenza

La semantica della consistenza è un importante criterio per la valutazione di qualsiasi file system che supporti la condivisione file e risorse. Si tratta di una caratterizzazione del sistema che specifica la semantica di operazioni in cui più utenti accedono contemporaneamente a una risorsa condivisa. In particolare, questa semantica deve specificare quando le modifiche ai dati apportate da un utente possano essere osservate da altri utenti. Di seguito illustreremo brevemente alcuni esempi di semantica della consistenza, adottati sia nei file system convenzionali che in quelli distribuiti.

1.3.1 Semantica UNIX

Nella semantica UNIX vige la politica *last write wins:* l'ultima scrittura sul file è quella che prevale su tutte le altre e pertanto ogni lettura successiva rifletterà sempre l'ultima operazione di scrittura avvenuta.

Punti cardine di tale semantica sono i seguenti:

- le scritture su un file aperto da parte di un processo sono immediatamente visibili ad altri processi che hanno aperto contemporaneamente lo stesso file
- il file è associato ad una singola immagine fisica, vista quindi come una risorsa esclusiva. Il contendersi delle risorse causa ovvi ritardi ai processi utente.

Lo svantaggio di questa semantica rapportata ai DFS è che risulta impossibile rafforzare la semantica per un miglior adattamento al caso distribuito (l'unica soluzione consisterebbe nel fare in modo che tutti gli accessi avvengano al server; ciò è palesemente inaccettabile).

I DFS che hanno adottato questa semantica sono: Sprite, DCE/DFS.

1.3.2 Semantica delle sessioni

Questa semantica è utilizzata da moderni file system distribuiti, quali AFS, AFS2, Coda e InterMezzo.

Le scritture di un utente su un file aperto non sono immediatamente visibili agli altri utenti che hanno aperto tale file in contemporanea; alla chiusura del file stesso, le modifiche apportate saranno visibili solo nelle sessioni che inizieranno successivamente quella in chiusura.

Questa semantica fa sì che un file sia associato a più immagini fisiche, probabilmente diverse, nello stesso momento. Ciò favorisce gli accessi concorrenti in lettura/scrittura, riducendo così i ritardi.

1.3.3 Semantica dei file condivisi immutabili

Questa è una semantica di semplice implementazione: un file condiviso diviene immutabile nel nome e nel contenuto, ma resta possibile accedere contemporaneamente in sola lettura ai file condivisi.

1.3.4 Semantica debole

Si tratta di una semantica poco precisa e poco restrittiva, basata su criteri non troppo raffinati, come l'aging, il timeout, etc.

Questo tipo di semantica è utilizzata da file system di rete come NFS e SMB/CIFS.

1.4 Problematiche

Le problematiche inerenti alla progettazione e relativa implementazione di un file system distribuito si possono distinguere in due categorie: problematiche di primo livello e problematiche di secondo livello.

Nella prima categoria rientrano tutte quelle problematiche che si incontrano nel voler implementare al meglio il file system, mentre nella seconda vi cadono le problematiche che sorgono dalla risoluzione dei problemi di primo livello (ad esempio, in seguito ad introduzioni di particolari features all'interno del DFS). Vediamo più dettagliatamente cosa si intende.

1.4.1 Problematiche di primo livello

- Massimizzazione delle prestazioni generali: è possibile venire in contro a questa problematica facendo uso di *caching* e di *servizi con informazione di stato*.
- Massimizzare la disponibilità delle informazioni sul mapping *(nome file, locazione di memoria)*: tale mapping è necessario per l'implementazione della nominazione trasparente. Per soddisfare tale esigenza è possibile utilizzare metodi come la *replicazione dei file* oppure il *caching locale*.
- Mantenere gestibile il mapping *(nome file, locazione di memoria):* per fare ciò è possibile aggregare insieme di file in unità componenti più grandi e fornire il mapping sulla base di tali componenti. In Unix si utilizza l'albero gerarchico delle directory per fornire il mapping: i file sono "aggregati" in directory in modo ricorsivo.

1.4.2 Problematiche di secondo livello

L'utilizzo di alcuni metodi per affrontare problematiche di primo livello (come il caching, la replicazione, etc.) inducono a problemi secondari, comunque non del tutto irrisolvibili.

Il *caching* porta ad incappare nel problema di *locazione della cache*: quale è il luogo migliore dove allocare una cache? Le possibilità sono due:

- *cache su disco*: questa locazione è molto affidabile. Le modifiche a dati sottoposti a caching non vanno perse in caso di guasti. Inoltre, dopo un ripristino non è necessario ripopolare la cache, essendo essa non volatile.
- *cache in memoria centrale*: tale soluzione permette l'utilizzo di stazioni diskless, un accesso più veloce ai dati in cache e l'utilizzo di un meccanismo unico per la gestione delle cache dei dati client e quelle dei dischi del server, il tutto a discapito dell'affidabilità precedente.

L'altra problematica è ancora una volta inerente al caching; nella fattispecie, si tratta dei problemi legati alla *coerenza della cache*.

Un client deve decidere se una copia nella cache sia o meno coerente con la copia master. Qualora il client invalidasse i dati nella propria cache, i dati master dovranno essere sottoposti a un nuovo caching. Per verificare la validità dei dati sottoposti a caching è possibile seguire due approcci:

- *approccio iniziato dal client*: il client inizia un controllo di validità nel quale contatta il server e controlla se i dati locali sono coerenti con la copia master. La frequenza del controllo di validità è il fulcro di questo approccio e determina la conseguente semantica della consistenza. E' possibile effettuare un controllo prima di ogni accesso, fino a scendere a un solo controllo durante il primo accesso a un file (di solito all'apertura del file). Ogni accesso che sia accoppiato a un controllo di validità risulta essere ritardato rispetto a un accesso servito immediatamente dalla memoria cache. In alternativa è possibile iniziare un controllo a intervalli di tempo prefissati. A seconda della frequenza di esecuzione il controllo di validità può caricare sia la rete che il server.

- *approccio iniziato dal server*: il server registra le parti di file che sottopone a caching per ogni client. Quando il server individua una potenziale incoerenza, deve reagire. Tale potenziale incoerenza si verifica quando un file viene sottoposto a caching da parte di due (o più) client diversi con modalità conflittuali. Se viene implementata la semantica delle sessioni (Paragrafo 1.4.2), ogni volta che un server riceve una richiesta per la chiusura di un file che è stato modificato, deve reagire informando i client che i dati nella cache sono incoerenti e vanno scartati. I client che in quel momento hanno il file aperto scartano la loro copia non appena termina la sessione corrente. Altri client scartano instantaneamente la loro copia. Nel caso della semantica delle sessioni il server non deve essere informato delle aperture dei file già sottoposti a caching. La reazione del server viene attivata solo dalla chiusura di una sessione di scrittura, e quindi viene ritardato solo questo tipo di sessione. In Andrew, come vedremo, essendo implementata proprio la semantica delle sessioni, viene utilizzato un approccio iniziato dal server, conosciuto col nome di *callback*.

Infine, l'ultimo problema (da noi preso in esame) che cade in questa categoria di problematiche è quello che nasce dall'introduzione della feature di *file replication*. Il problema dell'aggiornamento delle repliche dei file, infatti, non può essere minimamente tralasciato, poichè dal punto di vista dell'utente le repliche di un file indicano la stessa entità logica, e quindi l'aggiornamento apportato a una replica deve riflettersi anche su tutte le altre. Più precisamente, la relativa semantica della consistenza deve essere conservata quando gli accessi alle repliche sono considerati accessi virtuali al file logico originario. Se la coerenza non costituisce un fattore di importanza primaria, può essere sacrificata a vantaggio della disponibilità e

delle prestazioni. Si tratta della concretizzazione di un fondamentale compromesso nell'ambito della tolleranza ai guasti. Dev'essere fatta una scelta tra la conservazione della coerenza ad ogni costo, che implica la creazione di un potenziale blocco indefinito, e il sacrificio della coerenza nei casi di guasti catastrofici a vantaggio di un procedere garantito.

2 Andrew File System

Andrew è un ambiente di calcolo distribuito sviluppato originariamente presso la Carnegie Mellon University. Andrew File System (AFS) altri non è che il file system nato come cuore per la condivisione delle informazioni tra i client di tale ambiente. AFS ha avuto svariate implementazioni, alcune commerciali come quella di IBM e altre Open Source, come OpenAFS (http://www.openafs.org). Il file system è ormai disponibile per una ricca varietà di Sistemi Operativi: svariati Unix, Linux, *BSD, OSX/Darwin, Windows (2000+).

Uno degli attributi più importanti di Andrew è la scalabilità: il sistema è progettato per comprendere oltre 5000 workstation.

2.1 Generalità

Andrew opera una distinzione tra *macchine client* (talvolta indicate col nome di *workstation*) e le *macchine server* dedicate. I server e i client utilizzano uno dei sistemi operativi supportati e sono collegati attraverso una LAN.

I client sono prensentati con uno spazio di nomi di file partizionato: uno *spazio di nomi locali* e uno *spazio di nomi condiviso.* Server dedicati, chiamati nel loro insieme *Vice* (dal nome del software che eseguono), presentano ai client lo spazio dei nomi condivisi in forma di gerarchia di file: tale gerarchia è omogenea e presenta caratteristiche di trasparenza di locazione. Lo spazio dei nomi locali è il file system root di una stazione di lavoro, dal quale discende anche lo spazio dei nomi condiviso (a partire dal mountpoint **/afs**). Le *workstation* eseguono il protocollo *Virtue* per comunicare con i server *Vice* e devono possedere dischi locali in cui memorizzare il loro spazio di nomi locale. L'insieme dei server è responsabile della memorizzazione e della gestione dello spazio dei nomi condiviso. Lo spazio dei nomi locale è piccolo; ne esiste uno per ogni stazione di lavoro e ciascuno contiene programmi di sistema fondamentali per un funzionamento autonomo. Sono locali anche file temporanei e i file che il proprietario della stazione non vuole condividere.

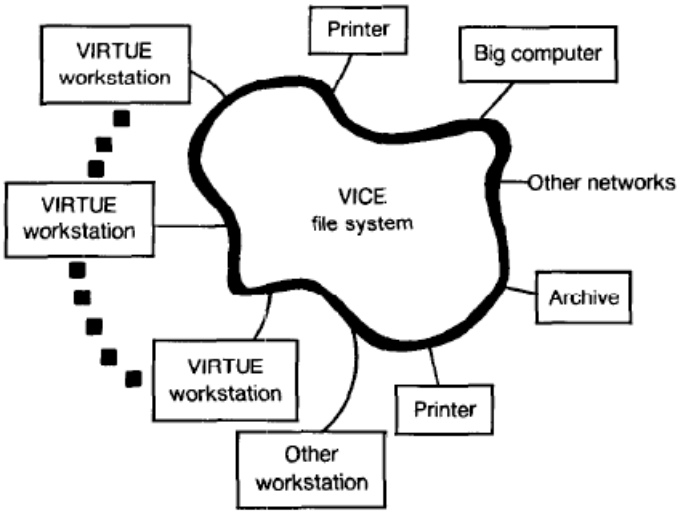

Figura 1: Schema client/server di AFS

Attraverso un esame più ravvicinato, è possibile notare che client e server sono strutturati in cluster interconnessi tramite una WAN. Ogni cluster è formato da un insieme di stazioni di lavoro su una LAN e un rappresentante *Vice*, chiamato *cluster server* che è collegato alla WAN. La decomposizione in cluster viene effettuata soprattutto in riferimento al problema di scala. Per ottenere delle prestazioni ottimali, le workstation devono utilizzare il server sul loro cluster per la maggior parte del tempo, così da ridurre i riferimenti ai file intercluster.

2.2 Operazione su file e semantica della consistenza

Il principio fondamentale su cui è basata l'architettura di Andrew è il caching per intero dei file, effettuato dai server. Di conseguenza, una stazione di lavoro client interagisce con i server *Vice* soltanto all'apertura e alla chiusura di un file; la lettura e la scrittura, invece, non causano alcuna interazione remota. Questa distinzione fondamentale influenza notevolemente le prestazioni e la semantica delle operazioni su file.

Il sistema operativo di ogni stazione intercetta le system call relative all'apertura/chiusura di file e le invia a un processo user-level su quella stessa stazione di lavoro. Questo processo, chiamato *Venus*, sottopone a caching i file provenienti da Vice nel momento della loro apertura e al momento della chiusura rimemorizza le copie modificate dei file sui server dai quali provenivano. Le operazioni di lettura e scrittura, invece, vengono eseguite direttamente dal kernel sulla copia nella cache e aggirano Venus. Il risultato di questa operazione è che le scritture effettuate su un sito non sono immediatamente visibili sugli altri siti.

Il caching viene ulteriormente sfruttato per aperture successive del file nella cache. Venus presuppone che gli elementi nella cache siano validi fintantoché non è specificato diversamente; perciò Venus, all'apertura di un file, non deve contattare Vice per validare la copia della cache. Il meccanismo che supporta questa politica, chiamato *callback*, riduce drasticamente il numero delle richieste di validazione della cache ricevute dai server.

Tale meccanismo funziona come segue: quando un client A sottopone a caching un elemento del file system, il server aggiorna la sua informazione di stato registrando l'avvenuto caching; a questo punto il client A possiede un callback su quel file. Prima di permettere ad un altro client B di modificare il precedente file, il server informa il client A. Così facendo, si dice che il server ha rimosso il callback per il client A. Un client può effettuare una *open()* su un file nella cache solo se detiene un callback per lo stesso. Se un client chiude un file dopo averlo modificato, tutti gli altri client che stanno effettuando caching su questo file perdono i loro callback: quando questi client riapriranno il file dovranno ricevere dal server la versione aggiornata.

Detto ciò, è palese che Venus contatta i server Vice solo all'apertura di file non presenti nella cache (o i cui callback sono stati revocati) e alla chiusura di file modificati localmente.

Sostanzialmente, AFS implementa la semantica delle sessioni.

2.3 Dettagli implementativi

Al fine di comprendere al meglio quanto successivamente verrà detto a proposito dei file system Coda e InterMezzo (che ricordiamo discendere dall'idea originale di AFS), è necessario spendere qualche parola per alcuni dettagli implementativi riguardanti taluni aspetti del file system di Andrew.

2.3.1 Spazio dei nomi condiviso

Lo spazio di nomi condiviso di Andrew è formato da unità componenti chiamate *volumi*. Generalmente i volumi di Andrew non sono piccoli, anzi, di norma sono associati ai file di un singolo client. All'interno di una singola partizione del disco risiedono pochi volumi, che possono crescere e restringersi di dimensione. Dal punto di vista concettuale, i volumi vengono uniti tra loro per mezzo di un meccanismo simile al meccanismo di mount di UNIX, ma la differenza di granularità è significativa: in UNIX è possibile montare soltanto un'intera partizione.

Un file o una directory Vice vengono identificati da un handle di basso livello, chiamatio *fid*. Ogni elemento di directory mappa un path name su di un *fid*. Un fid è una struttura dati lunga 96 bit, costitituita da tre elementi di uguale dimensione (32 bit): un *numero di volume*, un *numero di vnode* e un *unicizzatore*.

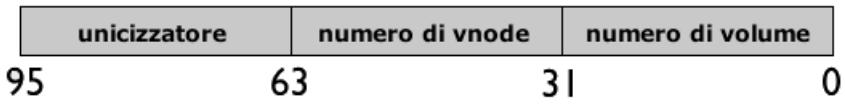

Figura 2: Schema della struttura di un *fid*

Il primo identifica univocamente un volume, il secondo serve come puntatore alla tabella di inode di un volume, mentre il terzo elemento, l'unicizzatore, permette di riutilizzare i numeri di vnode su tutti i volumi, mantenendo così le strutture più compatte. Cosa molto importante: i fid sono trasparenti alla locazione, quindi lo spostamento di file da server a server non invalida il contenuto delle cache.

2.3.2 Kernel, server e cache manager

I processi client sono interfacciati ad un kernel con il solito set di system call. I kernel sono leggermenti modificati per individuare i riferimenti ai file Vice e per inviare richieste al processo Venus, il cache manager di Andrew. Venus e i processi server accedono ai file direttamente tramite inode, per evitare il costo della risoluzione del pathname. Poichè l'interfaccia inode non è normalmente visibile ai processi, sono state aggiunte nuove system call. Venus è incaricato in realtà di gestire non una, ma due cache differenti: la prima è utlizzata per i dati mentre la seconda per tenere traccia dello stato del sistema. Per contenere la dimensione di dette cache, viene utilizzato un semplice algortimo LRU (*Least Recently Used*).

Per ciò che concerne il server, esso consiste di un singolo processo multithreaded, capace quindi di servire concorrentemente le molteplici richieste inviate dalle macchine client. La scelta di un singolo processo multithreaded piuttosto di una soluzione a multiprocesso permette il caching delle strutture dati necessarie per soddisfare le richieste di servizio. Questo è sì un vantaggio in termini prestazionali, ma che purtroppo sacrifica la tolleranza ai guasti: se il processo muore, il server si blocca.

2.4 Conclusioni

Giunti a questo punto, chiuderemo questa panoramica su AFS con un elenco dei vantaggi introdotti rispetto a file system come NFS, etc.

- Ogni client ha la stessa visione dei file, in qualsiasi punto si trovi, grazie alla radice comune di AFS. Ne segue la proprietà di *mobilità dei client*: i client possono accedere da qualsiasi stazione di lavoro a qualsiasi file che si trovi nello spazio di nomi condiviso.
- Maggiori performance grazie alla cache locale responsabile della riduzione del carico di rete

- Maggiore scalabilità
- Maggiore sicurezza (è utilizzato un sistema di autenticazione Kerberos, per brevità non trattato in quest'articolo)
- Ammette replicazione dei volumi, garatendo maggiore disponibilità e sicurezza

3 Coda File System

Presso la medesima università dove è stato sviluppato Andrew, la Carnegie Mellon University, nasce anche il progetto Coda, basato su AFS 2.

La struttura principale di Coda è stata ereditata totalmente da AFS; infatti anche in Coda abbiamo i Vice server e i Virtue client che eseguono il cache manager Venus.

In questa sezione (e più avanti nella sezione degli internals e dei bottleneck) si noterà quali migliorie e modifiche sono state apportate a Coda rispetto al progenitore.

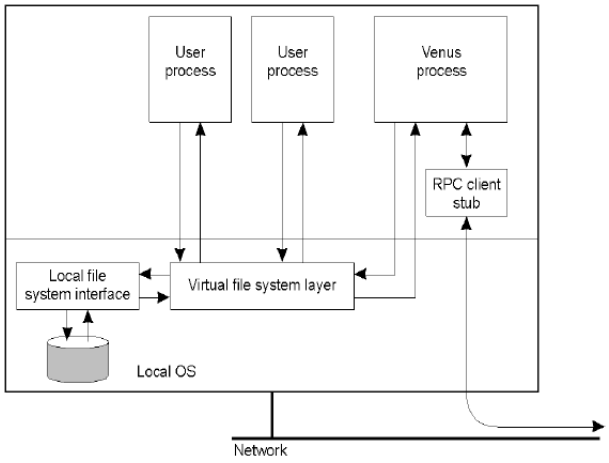

Figura 3: Schema di un client Virtue di Coda

3.1 Interazione dei componenti

In questo paragrafo osserveremo come un'applicazione interagisce col kernel e con i moduli user-level del Coda File System, durante l'apertura di un file; il discorso di seguito è valido anche per AFS.

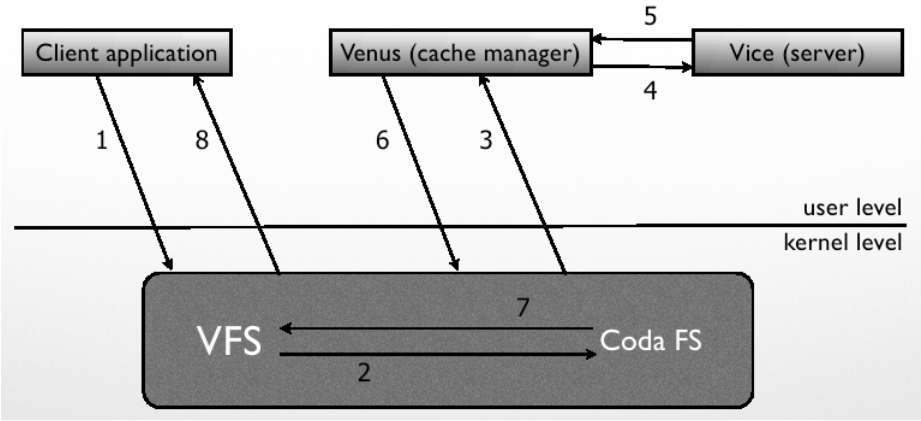

Figura 4: Schema dell'invocazione di una system call open/close

Osservando la figura, vediamo che, come ci aspettavamo, l'interfaccia immediatamente disponibile all'applicazione è quella del Virtual File System, ormai componenente standard di tutti i sistemi operativi. Una volta invocata la open() del VFS (1) su di un file facente parte di un file system Coda, all'interno del kernel viene eseguita l'implementazione Coda di tale metodo (2); Coda contatta il cache manager Venus (3) per verificare se si sta aprendo un file già sottoposto a caching, verificare quindi lo stato dei callback relativi e in ogni caso verificare la consistenza; nell'esempio della figura, il file non è in cache e Venus contatta un server Vice (4) per il prelievo dello stesso; il server invia il file al cache manager (5) che a sua volta ritorna alla routine di open di Coda; Coda, attraverso il VFS, ritorna il risultato della open() all'applicazione lato client.

Da questo semplice esempio è facile notare che la open() (e anche la close()) sono dei colli di bottiglia per il file system Coda; infatti, oltre ai due cambi di contesto classici per l'invocazione di una system call (user->kernel e kernel->user), ad ogni invocazione di una open/close si deve effettuare un nuovo context switch per contattare il processo Venus e ancora un altro cambio di contesto per ritornare a livello kernel. Vedremo successivamente, nel capitolo dedicato ai Bottleneck, come questo overhead sia tutt'altro che trascurabile.

3.2 Caratteristiche

Coda è stato progettato al fine di realizzare un file system che avesse delle ben determinate caratteristiche, in parte ereditate da AFS e in parte modificate e/o aggiunte al solo scopo di avere un file system distribuito più moderno, flessibile e scalabile dei suoi predecessori.

3.2.1 Scalabilità e performance

La scalabilità e le performance erano già un obiettivo elegantemente raggiunto da AFS (progenitore di Coda). Come sempre, però, è possibile fare di meglio. Oltre alla già ottima scalabilità di AFS (ottenuta grazie al sistema di suddivisione della rete in cluster già visto nel capitolo 2) Coda introduce delle nuove feature molto interessanti: le *Disconnected Operations* che, come vedremo più avanti, permettono un migliore supporto alle reti con poca larghezza di banda e un supporto trasparente al lavoro off-line.

3.2.2 Robustezza

Coda raggiunge l'obiettivo di aumentare la robustezza del sistema allo stesso modo di AFS, adottando una politica di *replicazione dei server,* ma fornendo una risoluzione migliore dei conflitti server/server. Come vedremo nei paragrafi successivi, Coda utilizza come unità di replicazione base il *volume*, allo stesso modo del suo antenato.

3.2.3 Consistenza e caching

Il modello di consistenza è l'ennesima caratteristica simile ad AFS; in ogni caso la semantica della consistenza adottata è ancora la *semantica delle sessioni*.

Il meccanismo di caching è ancora il *callback* mentre, nelle ultime versioni di Coda, viene introdotto il *write-back* in luogo del *write-through* come metodo di copia dei dati dalla cache alla master copy. Un'altra innovazione di Coda sono gli *sticky files* in cache: è possibile, per un client, specificare quali file non devono essere mai rimossi dalla cache, anche quando il principio di località dei riferimenti suggerirebbe la rimozione del file.

3.3 Replicazione dei server

Come già accennato nel precedente paragrafo, Coda File System supporta la replicazione dei server. L'unità di replicazione è il *volume*. Esso è una collezione di file che sono memorizzati in un server a partire da un sottoalbero dello spazio dei nomi condiviso. Come avviene in AFS, anche Coda supporta la *read-only replication* dei volumi.

L'insieme dei server che contengono le repliche di un volume è definito *Volume Storage Group* (VSG). Di solito si sceglie di memorizzare i dati critici (ampia frequenza di accesso, dati sensibili, etc.) in volumi replicati, mentre si raccolgono i dati non critici in volumi non replicati. Per ogni volume che possiede dati in cache, Venus tiene traccia del sottinsieme del VSG che è raggiungibile al momento, denominato *Accessible Volume Storage Group* (AVSG). Un server viene eliminato dall'AVSG quando non riesce a servire un'operazione per temporaneo stato di off-line (le operazioni van-

no in time-out); verrà reintegrato quando Venus riuscirà a ristabilire la connessione.

La strategia di replicazione adottata è una variante dell'approccio *read-one, write-all*. Quando viene chiuso un file che ha subito modifiche, esso viene trasferito a tutti i membri dell'AVSG a cui si sta facendo capo. Questo tipo di approccio massimizza la probabiltà che ogni replica abbia sempre i dati aggiornati. Il rovescio della medaglia è la latenza nella propagazione della sincronizzazione, che viene limitata adoperando un meccanismo di chiamate parallele a procedure remote (parallel-RPC).

3.4 Disconnected Operations

Le già citate disconnected operations permettono ai client di lavorare quando un membro del VSG risulta essere irraggiugibile.

Durante il periodo di disconnessione, i dati richiesti in lettura vengono serviti sempre dalla cache; le scritture vengono anch'esse eseguite direttamente sulla copia nella cache e le modifiche vengono annotate nel *Client Modification Log* (CML). All'atto della riconnessione il CML viene inviato al server che lo analizzerà al fine di reintegrare i dati modificati durante il periodo di disconnessione.

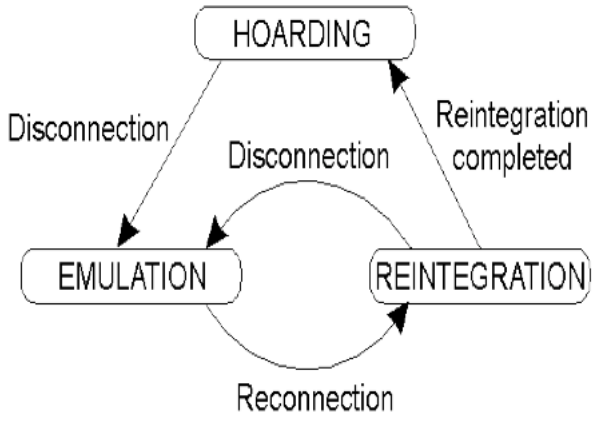

Figura 5: Diagramma degli stati delle fasi di disconnessione/riconnessione

Come è possibile vedere anche dalla figura, la disconnessione può avvenire in due punti: durante la fase di *reintegrazione* dei dati e durante la fase di *tesaurizzazione* (la fase di normale funzionamento connesso del file system). Ogniqualvolta si verifica la disconnessione, il sistema passa nella fase di *emulazione*, che rende solitamente la disconnessione trasparente all'utente. Non appena è possibile ristabilire la connessione si passa nella fase di reintegrazione, che una volta completa lascia il posto alla fase

di “comportamento normale”, quella di tesaurizzazione. C’è da aggiungere che la disconnessione non permette soltanto di rendere trasparenti e meno ostacolanti taluni guasti di rete (o ai server), ma lascia la possibilità agli utenti di notebook connessi al sistema di potersi disconnettere volutamente e poter continuare ugualmente a lavorare restando isolati dai server Coda anche per un lungo periodo di tempo. Infine, grazie alle disconnected operations, Coda risolve un problema che il suo progenitore aveva rimasto irrisolto: è possibile lavorare tranquillamente con reti a basse performance.

3.5 Conclusioni

Concludendo, Coda assolve molto bene ai suoi obiettivi primari, come l’alta scalabilità e l’alta disponibilità. La prima è garantita dalle caratteristiche ereditate da AFS alle quali si aggiungono le Disconnected Operations, la Reintegrazione e il WriteBack Caching; la seconda è garantita dal meccanismo di replicazione. E’ superfluo dire che Coda non è affatto perfetto nella sua implementazione. Uno degli svantaggi più evidenti di questo file system distribuito è stato già citato precedentemente: si tratta dell’overhead introdotto dall’invocazione delle system call di open() e di close(), che invocano sincronamente il cache manager ad ogni chiamata.

4 InterMezzo File System

Il file system InterMezzo rappresenta la reigegnerizzazione del file system Coda. Il progetto non consiste in un fork di quello precedente, ma in una completa riscrittura (pur essendo Coda un progetto Open Source); l’obiettivo principale è infatti quello di creare un file system “più piccolo” in termini di codice sorgente, mantenendo (e migliorando) ugualmente le caratteristiche di Coda.

4.1 Linee progettuali

Le principali linee progettuali di InterMezzo consistono nel:

- reimplementare in maniera più efficienti le caratteristiche di Coda
- sfruttare il file system locale come struttura dati per il caching; questo implica anche la possibilità di avere *cache persistenti*.
- implementare un meccanismo di *write-back caching* (all’inizio della progettazione di InterMezzo, Coda non aveva ancora introdotto tale meccanismo)
- inserire dei filtri nel driver che si interfaccia al file system locale dei client; ciò permette di contattare il cache manager solo nei casi necessari e non più ad ogni open/close come avveniva in Coda.

- introduzione di *metadati* atti a supportare meglio le operazioni off-line
- creare un meccaniscmo unico per le due differenti politiche di caching (lato client e lato server)

Più avanti chiariremo meglio questi aspetti.

Vediamo ora brevemente quali sono i servizi offerti da questo file system e quali sono le sue caratteristiche peculiari.

4.2 Panoramica

Innanzitutto InterMezzo, come già detto, non elimina nessuna delle caratteristiche del suo file system di riferimento: Coda.

Quello che invece tenta di fare è di rendere il tutto più performante con l'obiettivo primario di portare InterMezzo ad essere un file system distribuito dalle prestazioni paragonabili a quelle di un file system centralizzato.

In aggiunta alle già viste e riviste caratteristiche di Coda e AFS (multi-server, sicurezza, etc.), InterMezzo aggiunge un miglioramento delle performance ottenuto anche attraverso la minimizzazione delle interazioni di rete, grazie soprattutto alla riduzione di comunicazione con il cache manager. Inoltre, per rendere più efficienti le interazioni di rete che comunque restano d'obbligo, è stato anche ottimizzato il protocollo di messagistica.

Infine, come vedremo nel seguito, all'ereditata ottima semantica di condivisione viene aggiunto un buon supporto al versioning e viene leggermento esteso il concetto di *volume*.

4.3 File sets

Ogni file tree reso disponibile dai server di InterMezzo è costituito da *file sets* o *volumi,* che sono simili ai volumi di Coda e AFS. Ogni file set ha una root directory e può contenere mount point di InterMezzo o altri file set. Un mount point InterMezzo è concettualmente simile al classico mount-point Unix, ma è nella pratica differente e distinto da esso. Un client può avere qualsiasi file set come root di un filesystem InterMezzo.

Un file set è associato sempre a un *File Set Storage Group*, che descrive il server che detiene il file set. Il *File Set Location Database* (*FSLDB*) descrive invece i mount point dei file set e i loro Storage Group. Un cluster InterMezzo è un gruppo di server che ha in comune lo stesso *FSLDB*.

I File Set contengono file e directory e generalmente il software di livello utente non ha visione di essi.

Il File Set Location Database è implementato con un albero sparso di directory, gestito come un oggetto speciale nel file system. Un'operazione di aggiornamento di tale albero è associata ad ogni modifica dell'albero del file set e i numeri di versione sono associati ad ogni aggiornamento. Questo permette al FSLDB di essere sempre aggiornato senza dover scaricare l'ultima versione.

4.4 Server and client

InterMezzo effettua una distinzione tra client e server; tale distinzione, nella fattispecie, è basata sulla politica piuttosto che sul meccanismo. Infatti, il meccanismo utilizzato resta lo stesso, ma i server mantengono le strutture su disco come record di stato dei file e in più coordinano la coerenza degli stati tra i client InterMezzo. I server InterMezzo possono anche assumere il ruolo di client, anche se il layout del file system dei server è leggermente differente da quello di un comune client. Il mount point di un file set sui server non necessariamente è una directory del file system InterMezzo, come accade per i client. Se il server stesso immagazina il file set in questione sui propri dischi, allora invece di un mount point vero e proprio viene creato un link simbolico che punta alla directory contenente i dati: la *File Set Holding Location* (FSHL).

4.5 Journaling e filtering

Una cache di InterMezzo è semplicemente un file system locale incapsulato in un layer aggiuntivo chiamato *Presto* nei sistemi Linux e *Vivace* nei sistemi Windows. Questo layer aggiuntivo ha funzionalità di filtering e ha un compito doppio:

- filtrare gli accessi, al fine di contattare il cache manager soltanto nei casi necessari (coerenza di cache invalidata, etc.)
- effettuare il journaling delle modifiche effettuate sul file system

Tutte le richieste che poi vengono emanate da Presto (Vivace) in seguito all'esigenza di contattare il cache manager, vengono gestite da *Lento*, che agisce sia come cache manager per i client che come file server.

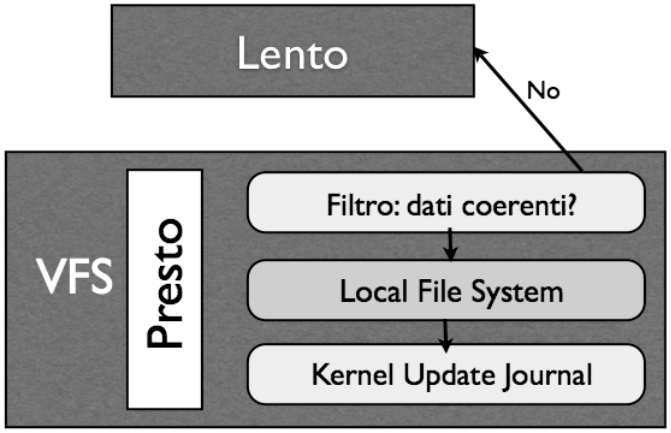

Figura 6: Interazione di base con il cache manager Lento

Quando intermezzo viene montato, Presto viene informato circa il tipo di file system che esso deve incapsulare, in modo tale da ”instanziare” la giusta implementazione dei metodi del Virtual File System. La realizzazione lato kernel di InterMezzo (e quindi quella di Presto), prevede infatti l'implementazione differenziata dei metodi del VFS (metodi delle strutture *dentry, inode, file object,* etc.) per ognuno dei file system centralizzati da incapsulare.

Concludendo, il compito di Presto è quello di filtrare tutti gli accessi e di effettuare il journaling di tutte le modifiche svolte in cache, eccezion fatta per quelle fatte dal cache manager Lento (esse restano immuni dal filtering e dal journaling).

4.6 Dettagli sul caching

InterMezzo si comporta, nell'adempimento di alcuni compiti, come l'antenato AFS:

- se l'oggetto non è presente in cache, ne si effettua il fetch dal server
- i file vengono prelevati nella loro interezza
- viene effettuato il caching dell'intero albero delle directory (non i dati dei file, ma solo le entry delle directory) sul client

L'ultimo punto può sembrare un introduzione di overhead apparentemente ingiustificata. In realtà, si tratta di una mossa abbastanza intelligente, che permette, nei successivi accessi, di eliminare la latenza nella risoluzione dei path name e nel reperimento degli attributi dei file, effettuando tutto localmente ai client.

Le modifiche vengono eseguite come segue. Innanzitutto è necessario acquisire un *permit* per il controllo di concorrenza.

Se chi sta effettuando una modifica è un client, questa viene effettuata in cache e ne viene effettuato il journaling sul *Client Modification Log* (già incontrato nel capitolo su Coda, ma è chiaro l'utilizzo sottilmente differente che ne fa InterMezzo). Successivamente verrà effettuata la *reintegrazione* sui server: i file modificati vengono coinvolti in un processo di *back-fetching*, che vede il server impegnato nel prelievo dei dati aggiornati dai client; se un server rileva che in una client-cache è presente una directory non esistente sui proprio dispositivi di memorizzazione di massa, allora ne effettua ricorsivamente il back-fetch.

Se invece ad effettuare la modifica di cui sopra è una macchina server, si effettua il journaling della modifica e la si propaga ai client registrati per la replicazione; gli altri client invalideranno la propria cache nel momento in cui rileveranno la modifica e si preoccuperanno di eseguire il fetch dei dati aggiornati.

4.6.1 Lento

Le operazioni effettuate da client e server non sono del tutto simmetriche, pertanto Lento, in esecuzione sia sui client che sui server, deve essere in grado di offrire:

- *File Service,* ovvero fetching (nel caso di esecuzione su client) e back-fetching (nel caso di esecuzione su server) di file e directory
- *Reintegration Service*, ovvero ricezione dei Client Modification Log (se in esecuzione su server) e notifica delle modifiche per la reintegrazione (se in esecuzione su client)

Le richieste che Lento riceve sia da remoto che dall'host su cui è in esecuzione, non vengono gestite secondo il modello di processo unico multi-threaded, ma mediante l'utilizzo delle sessioni. Quando giunge una richiesta, viene istanziata una sessione per gestirla; una sessione non è un thread, ma una struttura dati contenente *stati* e *gestori di eventi.* Le operazioni di I/O vengono eseguite in maniera asincrona e il kernel segnala il completamento di un'operazione tramite il dispatch di eventi. Il kernel, inoltre, attiva tutti i gestori di eventi ed è anche responsabile del *garbage collection* delle sessioni che non possono essere raggiunte da alcun evento (a causa di anomalie presentatesi nella gestione delle stesse, per guasti o quan'altro).

4.7 Conclusioni

Come fatto finora, concluderemo riassumendo brevemente i vantaggi e gli svantaggi del file system. Per quanto concerne i primi, InterMezzo eredita tutti i vantaggi del file system distribuito a cui si ispira: Coda. A tali vantaggi è d'uopo aggiungere l'aumento di performance dovuto all'introduzione di un nuovo livello[1] di callback tra il kernel e il cache manager. Ciò riduce notevolemente l'overhead presente prima in Coda durante le chiamate di servizio open() e close().

Per quanto riguarda gli svantaggi, essi sono per lo più da attribuire a Lento, il cache manager. Esso è implementato in Perl, che è un linguaggio interpretato. In più la sua configurazione è abbastanza difficoltosa e le operazione di rete leggermente lente.

4.8 Curiosità

Una curiosità che potrebbe nascere nel leggere i nomi delle componenti di InterMezzo, è capire qual è l'origine di questi termini. La risposta non è così difficile, soprattutto per chi si intende di musica. Infatti, tutti i nomi, compresi quello del file system stesso, sono termini utilizzati nel gergo musicale. L'*intermezzo* può essere una breve composizione pianistica in forma

[1] Questo nuovo layer vede la sua implementazione nel filtro Presto (Vivace, ricordiamo, per i sistemi Windows).

libera o comunque un interludio, mentre nomi come *lento*, *presto* o *vivace* indicano la velocità di esecuzione di una musica; i server portano il nome di *Maestro*.

5 Rimozione dei Bottleneck

Quando si parla di rimozione dei colli di bottiglia all'interno di un file system distribuito, la domanda che ci si deve porre è: *"è possibile ottenere, per un DFS, prestazioni paragonabili a quelle di un file system locale?"* Per la gioia di molti e lo stupore di altrettanti, la risposta a questo fatidico quesito è affermativa.

I principi più ovvi per portare le performance di un file system distribuito ad essere paragonabili a quelle di un file system centralizzato[2] sono:

- ridurre al minimo il numero di RPC invocate, come già accennato
- non avere protocolli di comunicazione sincroni col cache manager e introdurre un modello di caching *write-back*
- usare i file system locali come strutture dati per il caching persistente

La strada verso l'ottimizzazione delle performance dei DFS è di seguito percorsa partendo da Coda per terminare poi con la nascita del progetto InterMezzo; sarà nel seguito mostrato come quest'ultimo file system sia nato appositamente al fine di eliminare i problemi residui di Coda, che persistevano anche dopo un'accurata ottimizzazione. Lo scopo, come sappiamo, è stato raggiunto grazie a una reigegnerizzazione accurata e totale riscrittura del codice.

5.1 Panoramica

Quando si mira ad un utilizzo read-only di un file system, il problema cruciale è quello di ottenere delle performance ottime durante la fase di *warming della cache*, durante la quale avvengono tutti i cache miss iniziali[3].

Purtroppo, si è notato che Coda, anche con cache *warmed*, non riesce a raggiungere prestazioni paragonabili a quelle del file system ext2.

Nel seguito mostreremo come una seconda relazione di callback tra kernel e cache manager (presente anche, come già visto, nell'implementazione di InterMezzo) possa migliorare il comportamento di Coda visibilmente.

Per quanto riguarda, invece, un utilizzo in lettura/scrittura del file system, il numero di *bottleneck* nei quali ci si imbatte è sensibilmente maggiore.

Questa volta sia il client caching che il server caching necessitano di opportune migliorie.

[2] Il file system locale preso di riferimento nel seguito è *ext2* del sistema operativo Linux
[3] Primo accesso ai dati, prime risoluzioni di path name, etc.

Inoltre, la rete è spesso coinvolta sincronamente e questo incide troppo sulle prestazioni generali; purtroppo anche l'introduzione del modello *write-back* in Coda non permette di raggiungere le performance ambite, nonostante il traffico RPC (e di conseguenza l'overhead di rete) venga ridotto di molto. Ciò è dovuto ad altri fattori di rallentamento, come lo stesso Venus, il cache manager di Coda, che utilizza delle strutture proprietarie per il caching, invece di sfruttare il file system locale.

La soluzione che porterà all'obiettivo ambito, come vedremo, consisterà nell'utilizzare il file system locale come struttura dati per il caching e nel creare un modulo kernel che incapsuli il file system locale, permettendo quindi di creare il nuovo livello di callback tra kernel e cache manager. Tutto questo sfocerà nella creazione del file system InterMezzo.

5.2 Consistenza dei dati

L'eterno problema nell'utilizzo di un file system distribuito è che un file potrebbe essere modificato sul server in qualsiasi momento. Accessi ripetuti agli stessi dati sono all'ordine del giorno e più vie possono essere esplorate per far sì che il tutto si svolga efficientemente e che la consistenza dei dati regni sovrana.

Per iniziare, uno dei metodi più semplici, nonchè abbastanza grezzo, è quello di congetturare: se l'accesso a un dato ricade in un arco di tempo abbastanza breve trascorso dal precedente accesso, il client assume che i dati sono ancora validi. Questo schema è utilizzato, ad esempio, da NFS e produce il 50% di buoni risultati.

Successivamente incontriamo lo schema introdotto per la prima volta in AFS: i *callback*. Questo sistema riduce l'attività sincrona ed evita i fetch ripetuti. Lo schema a callback prevede che i dati nella cache di un client vengano ritenuti validi finchè il server non revoca al client i callback relativi a quei dati. La rimozione dei callback può avvenire, per esempio, in seguito alla modifica dei dati da parte di un altro client che condivideva in scrittura gli stessi file.

AFS e derivati, comunque, non sono gli unici DFS ad aver scelto lo schema a callback; anche SMB/CIFS e Sprite utilizzano tale schema, con l'unica differenza che nella famiglia AFS viene anche effettuato il caching dell'intero albero delle directory per eliminare l'utilizzo delle RPC di *pathname resolution*.

Inoltre, in AFS e derivati, lo schema a callback è stato successivamente arricchito con l'implementazione di un meccanismo che permettesse di disporre di *cache persistenti* (anche dopo disconnessioni o reboot di sistema); tale meccanismo prende il nome di *validazione*. I dati vengono ora memorizzati sui client con un preciso e univoco *version stamp*, che diventa l'unico parametro necessario a verificare la consistenza dei dati. I *version stamp* vengono aggiornati alla chiusura di un file, qualora questo sia stato modificato. In Coda (e successivamente in InterMezzo) il confronto attraverso *version stamp* viene effettuato prima a livello di volume e successivamente,

nel caso il controllo non termini con successo, il confronto viene ripetuto a livello di singolo file. Nella modalità con server replicati di Coda, inoltre, non basta memorizzare un solo version stamp, ma ogni client deve tenere un vettore di version stamp per ogni server del gruppo di replicazione.

Il prezzo da pagare per l'utilizzo di schemi a callback è che il server deve mantenere in memoria virtuale lo stato circa quali client detengono quali callback.

5.3 Callback a livello kernel

In questo paragrafo mostreremo come poter ottenere prestazioni simili a un file system locale in una configurazione a sola lettura, mettendo ancora una volta a confronto Coda ed Ext2.

Il test usato per il confronto è stato effettuato con l'esecuzione di un *ls -lR* di una directory contenente 1500 file e 300 sottodirectory.

Grazie ai callbacks e al write-back caching, Coda risparmia molto traffico RPC; ciononostante, anche con cache *warmed* Coda risulta essere ancora veloce la metà di Ext2. Analizzando approfonditamente i file system a confronto, si è scoperto che il colpevole di tale collo di bottiglia è il cache manager user-level di Coda. Anche se da un punto di vista ingegneristico la scelta dei progettisti di Coda è ottima, poichè garantisce un alto grado di portabilità, il rovescio della medaglia è che alcune system call sono servite dal kernel in maniera inusuale. In particolare, da Coda 5.0, la *open()* e la *close()* sono sempre servite contattando prima il cache manager (Venus) user-level, per avere informazioni sulla consistenza. Questo fa si che il tempo di servizio di una open/close per Coda sia cinquanta volte superiore a quello per Ext2, dato che aumenta considerevolemente il numero di context switch effettuati; considerando che, nell'esecuzione del test, 600 system call su 4500 sono del tipo incriminato, è facilmente spiegata la lentezza di Coda[4]. Ai fini di completezza del test, Coda è stato pertanto modificato introducendo un nuovo livello di callbacks a livello kernel, che permette di evitare di contattare il cache manager fintantochè il callback interessato non viene revocato. Le strutture dati necessarie per l'apertura e la chiusura dei file (la device e l'inode number della copia in cache del file) necessitano di rimanere sempre disponibili e saranno invalidate dal cache manager in due casi:

- il server revoca un callback in seguito alla modifica di un file (o directory) sui propri supporti di memorizzazione
- la cache client si riempie e risulta necessario effettuare un'operazione di *purge* per liberare spazio

Lo stesso meccanismo, ricordiamo, sarà introdotto nativamente nel file system InterMezzo.

[4]Nella fattispecie, Coda impiega il 50% del tempo di esecuzione del test nel servire le open/close.

5.4 Write back caching a livello utente

Abbiamo visto come il write back caching abbia incrementato le prestazioni di Coda drasticamente, pur essendo esso ancora inferiore prestazionalmente a Ext2. Coda, come abbiamo più volte detto, permette le *disconnected operations* (cfr. paragrafo 3.4). Durante la modalità disconnessa, Coda serve i dati dalla propria cache finchè ve ne è disponibilità. Le modifiche vengono anch'esse effettuate in cache e scritte nel *Client Modification Log* (CML). Alla riconnessione vengono verificati i version stamp e il log viene consegnato al server per la fase di *reintegrazione* (cfr. paragrafo 3.4). Coda possiede anche una terza modalità di funzionamento che consiste in un sistema ibrido tra la modalità connessa e quella disconnessa. Tale modalità prende il nome di *trickle reintegration* e viene attivata in presenza di reti a basse prestazioni: i cache miss sono serviti come nella modalità connessa, servendo dati aggiornati direttamente dai server, mentre le modifiche vengono effettuate secondo quanto previsto dalla modalità disconnessa, operando direttamente nelle client cache e appuntando i cambiamenti nel CML. Nella modalità trickle reintegration il CML viene spedito al server quando raggiunge una data dimensione o età (e non più alla riconnessione, trattandosi pur sempre di una particolare modalità connessa).

L'aspetto importante del CML è che questi può essere ottimizzato: molti file hanno vita breve e vengono presto rimossi, altri vengono salvati più e più volte. Questi ed altri casi particolari possono essere rilevati in seguito a una semplice scansione del CML, che può essere così snellito, evitando di inviare al server un CML più grande del necessario.

Il write back caching di Coda poggia le sue basi proprio sulla modalità *trickle reintegration*. L'unico problema, tra l'altro facilmente sormontabile, è quello che gli altri client non hanno più garanzie sul possesso di una visione consistente dei dati.

Quando un client *A* deve effettuare un'operazione di modifica, deve chiedere un *write-back permit* al server; questi, prima di concedere tale permesso, revoca tutti i callback dei vari client sui dati interessati. Successivamente le modifiche possono essere fatte e annotate nel CML, che verrà trasmesso asincronamente al server. Quando in un secondo momento un client *B* chiederà il permit, questo va revocato al client *A;* successivamente i dati vanno reintegrati (trickle reintegration) prima di concedere il permit al nuovo client.

Concludendo, il write-back caching di Coda dà sì una marcia in più alle sue performance, ma ancora non basta per paragonare Coda a Ext2, anche se le prestazioni migliorano a tal punto da renderlo comunque confrontabile col file system FFS del sistema operativo NetBSD.

5.5 Write back caching a livello kernel

Durante lo svolgimento di questi test e dopo le svariate ottimizzazioni apportate a Coda, si è giunti a un punto tale che non è stato più possibile

ottimizzare ulteriormente Coda, a causa dei limiti intrinseci dovuti al progetto iniziale. Pertanto Peter J. Braam, sempre fermo sulla propria idea che un DFS potesse aspirare a performance vicine a quelle di un file system locale, decise di dare vita al progetto InterMezzo, ricalcando le stesse orme del progetto Coda.

A vantaggio di InterMezzo ha giocato il fatto di essere nato come progetto ex-novo e non come semplice evoluzione di un precedente prodotto (come invece è accaduto col passaggio da AFS a Coda). Grazie a questa peculiarità, è stato possibile sin dalla progettazione iniziale tenere ben presenti le limitazioni di Coda, sia quelle già risolte sia quelle ancora tuttora presenti (e di fatto non eliminabili). In realtà le strutture dati che Coda utilizza per il caching frenano notevolmente le prestazioni[5].

InterMezzo preferisce invece sperimentare l'uso di un file system locale come struttura dati per il persistent caching. Di fatto le strutture dati per il caching passano da un livello utente (dove tutto era incapsulato nel cache manager) a un livello kernel, senza però mai abbandonare la figura del cache manager user-level (cfr. paragrafo 4.6 e 4.6.1), che assume ovviamente fattezze diverse.

Ricordiamo che InterMezzo non vuole assolutamente abbandonare ciò che di vantaggioso è presente in Coda: sono tuttora presenti in questo DFS il meccanismo dei write-back permit, lo schema a callback e i version stamp; questi ultimi, insieme alle informazioni di volume e alle *Access Control List* (ACL), vengono memorizzati nelle strutture dati del cache manager (*Lento*), non disponendo il file system locale di strutture atte a contenere tale informazioni sotto forma di metadati.

6 Coda Client internals

In questa sezione verrà presentata l'implentazione del metodo open() del VFS nei client di Coda.

La funzione che concretizza tale interfaccia è la `coda_open()`.

Per quanto già detto nella sezione relativa alla risoluzione dei bottleneck, la routine `open` contatta sempre *Venus*, il cache manager user-level[6]. Mostreremo questo aspetto con ulteriore dettaglio illustrando la routine `coda_open()` assieme ad altre routine ad essa innestate.

6.1 La funzione coda_open()

Di seguito è mostrato il codice della `coda_open()`.

[5] Da svariati benchmark risulta che tali strutture dati rallentano sensibilmente le operazioni di I/O

[6] Anche per la `close` avviene lo stesso, ma non la tratteremo: come vedremo di seguito, ciò che verrà detto riguardo la open può essere benissimo contestualizzato anche per la close

```
int coda_open(struct inode *coda_inode, struct file
*coda_file)
{
struct file *host_file = NULL;
struct inode *host_inode;
/* altre variabili */
...
...
lock_kernel();
error = venus_open(coda_inode->i_sb, coda_i2f(coda_inode),
coda_flags,
&host_file);
...
/* controllo errori */
host_file->f_flags |= coda_file->f_flags & (O_APPEND |
O_SYNC);
host_inode = host_file->f_dentry->d_inode;
...
unlock_kernel();
}
```

La funzione `coda_i2f(coda_inode)` è una funzione statica definita *inline* e viene mostrata di seguito:

```
static __inline__ struct ViceFid *coda_i2f(struct inode
*inode)
{
return &(ITOC(inode)->c_fid);
}
```

Essa non fa altro che convertire un inode in un *fid* (handle di basso livello per file e directory, introdotto già in AFS).

Come si evince dal codice appena riportato, il cuore della routine `coda_open()` è la routine `venus_open()`. Tuttavia, prima di illustrare quest'ultima funzione, è bene parlare del contenuto del modulo *upcall.c* (al quale appartiene sì detta funzione).

In questo file troviamo un set di funzioni la cui firma è della forma `venus_<fileoperation>()`; ne esiste una per ognuna delle `coda_<fileoperation>()`.

Tutte queste funzioni richiamano al loro interno la funzione `coda_upcall()` che è il cuore della connessione a Venus dal kernel e che comunica al cache manager l'operazione giusta da fare, grazie ad una strut-

tura di tipo `inputArgs` opportunamente inizializzata dalle funzioni `venus_<fileoperation>()` e poi passata come parametro alla `coda_upcall()`. E' possibile constatare quanto detto nella routine di seguito, la `venus_open()`.

```
int venus_open(struct super_block *sb,
struct ViceFid *fid, int flags, struct file **fh)
{
 union inputArgs *inp;
 union outputArgs *outp;
 int insize, outsize, error;
 insize = SIZE(open_by_fd);
 UPARG(CODA_OPEN_BY_FD);
 inp->coda_open.VFid = *fid;
 inp->coda_open.flags = flags;
 error = coda_upcall(coda_sbp(sb), insize, &outsize,
inp);
 *fh = outp->coda_open_by_fd.fh;
 CODA_FREE(inp, insize);
return error;
}
```

Come si nota, gli *inputArgs* vengono opportunamente impostati al fine da identificare un'operazione di apertura file e poi vengono passati come parametro alla routine `coda_upcall()`.

6.1.1 Strutture di supporto alle routine contenute in upcall.c

Prima di mostrare la routine `coda_upcall()` è bene mostrare alcune strutture utilizzate in molte delle routine contenute in *upcall.c*.

```
typedef u_long VolumeId;
typedef u_long VnodeId;
typedef u_long Unique_t;
typedef u_long FileVersion;
typedef struct ViceFid {
 VolumeId Volume;
 VnodeId Vnode;
 Unique_t Unique;
} ViceFid;
```

La struttura ViceFid rappresenta appunto il *fid*, già introdotto in AFS (cfr. paragrafo 2.3.1) ed ereditato da Coda senza alcuna modifica.

```
struct coda_sb_info
{
 struct venus_comm * sbi_vcomm;
struct super_block *sbi_sb;
};
```

La struttura `coda_sb_info` appena mostrata mantiene informazioni sul superblocco ed un puntatore alla Venus wait queue.

```
struct venus_comm {
 wait_queue_head_t vc_waitq;
 struct list_head vc_pending;
 struct list_head vc_processing;
 int vc_inuse;
 struct super_block *vc_sb;
};
```

La struttura `venus_comm` incapsula

- - la wait queue del cache manager
- - la communication processing queue
- - la communication pending queue
- - `vc_inuse` indica se Venus è ancora running

La struttura `upc_req` incapsula un messaggio che CodaFS (kernel) e Venus si scambiano per la comunicazione e la cui dimensione deve essere pari ad almeno 5000 bytes.

Abbiamo visto dalle strutture appena elencate, che la comunicazione tra Venus e la parte kernel di CodaFS avviene tramite uno scambio di messaggi. L'implementazione Unix di questo meccanismo è stata fatta attraverso una pseudo-device a caratteri associata a Coda. Venus cattura i messaggi del kernel tramite una read() sulla device e risponde con una write() sulla stessa. Durante lo scambio di messaggi, il processo che ha chiamato la system call viene messo in uno stato di **`TASK_INTERRUPTIBLE`**.

```
struct upc_req {
 struct list_head uc_chain;
 caddr_t uc_data;
 u_short uc_flags;
 u_short uc_inSize;
 u_short uc_outSize;
 u_short uc_opcode;
 int uc_unique;
 wait_queue_head_t uc_sleep;
 unsigned long uc_posttime;
};
```

6.1.2 coda_upcall()

```
static int coda_upcall upcall(struct coda_sb_info
*sbi,int inSize, int *outSize, union inputArgs *buffer)
{
struct venus_comm *vcommp;
struct upc_req *req;
vcommp = sbi->sbi_vcomm;
```

In questo primo frammento del codice relativo a `coda_upcall()` c'è da notare la dichiarazione e l'inizializzazione dei riferimenti alle code venus ed al messaggio che Venus e CodaFS si scambieranno.

```
if ( !vcommp->vc_inuse ) {
printk("No pseudo device in upcall comms at %p\n",
vcommp);
return -ENXIO;
}
```

Se Venus non risulta contattabile a causa di un crash o di qualche altro problema, si ritorna un errore.

```
req = upc_alloc();
if (!req) {
printk("Failed to allocate upc_req structure\n");
return -ENOMEM;
}
req->uc_data = (void *)buffer;
req->uc_flags = 0;
req->uc_inSize = inSize;
req->uc_outSize = *outSize ?  *outSize :  inSize;
req->uc_opcode = ((union inputArgs *)buffer)->ih.opcode;
req->uc_unique = ++vcommp->vc_seq;
```

In questo frammento di codice viene mostrato come avviene l'allocazione e la formattazione del messaggio da inviare a Venus;

```
list_add(&(req->uc_chain), vcommp->vc_pending.prev);
```

la chiamada a `list_add()` permette di aggiungere il messaggio alla coda di messaggi pending di Venus e di risvegliare il cache manager.

```
wake_up_interruptible(&vcommp->vc_waitq);ing.prev);
```

Il processo può essere interrotto mentre si aspetta che Venus processi la sua richiesta:

- Se l'interruzione avviene prima che Venus abbia letto la richiesta, questa viene eliminata dalla coda e si ritorna.
- Se l'interruzione avviene dopo che Venus abbia preso atto della richiesta, viene ignorata

In nessun caso la system call viene riavviata.

Se Venus viene chiuso, si ritorna con **ENODEV,** coerentemente alla scelta che prevede l'utilizzo di una pseudo-device come mezzo di scambio di messaggi.

```
runtime = coda_waitfor_upcall(req, vcommp);
```

CodaFS si mette in stato di attesa di segnali, che avranno comunque effetto solo dopo un timeout.

La gestione degli errori nello specifico viene omessa per brevità. Basti dire che vengono gestiti tutti i casi di interruzione sopra citati.

6.2 Venus

Dato che Venus non verrà trattato negli internals, è giusto spendere qualche parola in proposito.

In relazione a quanto è stato già detto circa lo scambio di messaggi tra kernel e Venus c'è da aggiungere che una volta che Venus ha identificato il contenuto del messaggio, agisce di conseguenza, richiamando una routine correlata all'operazione incaricatagli. Tale routine effettua un controllo sulla coerenza della cache e, qualora fosse necessario, esegue una RPC del server Vice.

7 Coda Server Internals

In questa sezione si parlerà degli internals del server Vice di Coda. A tale proposito, spiegheremo brevemente lo scopo dei seguenti moduli:

- **Salvager** (già presente in AFS)
- **RVM** (*Recoverable Virtual Memory*)

Il compito del Salvager è principalmente di ripristinare la consistenza interna ai volumi read/write corrotti. Se vengono riscontrate delle corruzioni nei volumi read-only o di backup queste vengono eliminate (piuttosto che corrette).

Con *recoverable virtual memory* si intendono quelle regioni di spazio di memoria virtuale sulle quali esistono garanzie transazionali, ossia, regioni di memoria in cui è possibile eseguire operazioni atomiche e in cui è offerta permanenza e serializzabilità.

RVM è unimplementazione di recoverable virtual memory per Unix, in generale, portabile ed efficiente. Una caratteristica unica di RVM è che offre un controllo indipendente sulle proprietà transazionali di atomicità, permanenza, etc. RVM è, inoltre, allo stesso tempo un potente tool di **fault-tollerance** e di **transaction-facility**.

7.1 Volume Structure

Allo startup di un server, viene inizializzata una struttura di tipo `camlib_recoverable_segment`. Tale struttura contiene un array di strutture `VolHead`, ognuna delle quali rappresenta un particolare volume.

La struttura `VolHead` è una struttura di RVM. Tuttavia, ogni `VolHead` associata ad un particolare volume è copiata nella VM del server nel momento in cui si fa accesso al volume.

All'interno della VM è presente una *hashtable* (`VolumeHashTable`) che tiene traccia dei volumi tramite istanze di strutture `Volume`, memorizzate in chiave dell'id di volume.

```
#define MAXVOLS 1024
struct camlib_recoverable_segment {
 bool_t already_initialized;
 struct VolHead VolumeList[MAXVOLS];
 VnodeDiskObject *SmallVnodeFreeList[SMALLFREESIZE];
 VnodeDiskObject *LargeVnodeFreeList[LARGEFREESIZE];
 short SmallVnodeIndex;
 short LargeVnodeIndex;
 VolumeId MaxVolId;
 []
};
```

La costante `MAXVOLS` definisce il numero massimo di volumi in ogni partizione.

La variabile booleana `already_initialized` indica se è richiesta o meno l'inizializzazione del volume.

L'array `VolumeList` rappresenta un array di headers di volumi.

Le variabili `SmallVnodeFreeList` e `LargeVnodeFreeList` rappresentano delle liste di strutture *vnode* libere.

`MaxVolId` rappresenta il massimo id di volume allocato su questo server.

7.1.1 RVM structures

La figura seguente mostra in maniera schematica alcune delle più importanti strutture RVM:

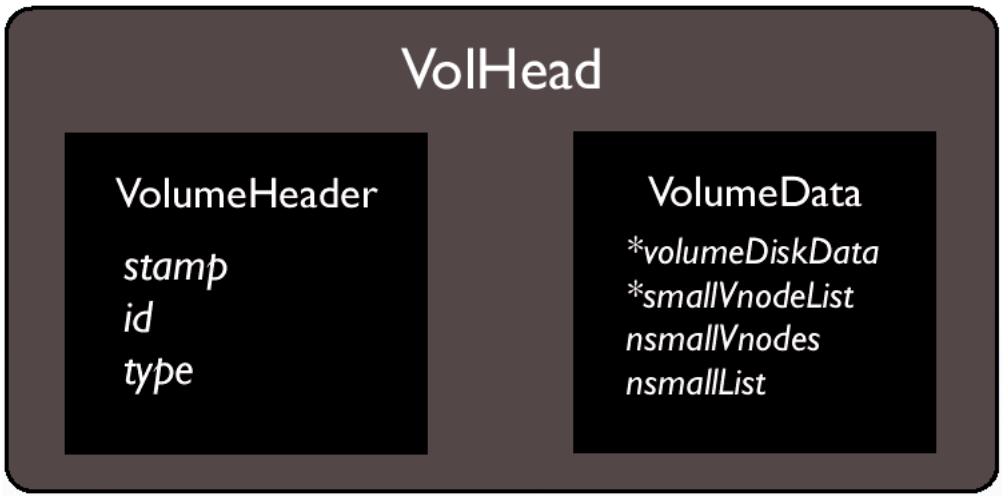

Figura 7: Alcune delle più importanti strutture RVM

Un volume viene identificato principalmente dal suo indice nell'array statico `VolumeList`. In alternativa si può fare accesso ai volumi attraverso il loro id.

La struttura `VolHeader` all'interno di `VolHead` permette di mappare un indice di `VolumeList` su un id di volume.

```
struct VolHead {
struct VolumeHeader header;
struct VolumeData data;
};
```

La struct `VolumeHeader` contiene delle informazioni che rappresentano metadati in RVM, mentre la struct `VolumeData` contiene informazioni utili a rappresentare il volume (ossia, informazioni sul versioning, id del volume, tipo del volume...).

```
    struct VolumeData
    {
        VolumeDiskData *volumeInfo;
        rec_smolist    *smallVnodeLists;
        bit32           nsmallvnodes;
        bit32           nsmallLists;
        [...]
    };
```

Fatta eccezione per il campo `volumeInfo`, che punta ad una struct `VolumeDiskData` (descritta di seguito), tutti gli altri campi della struttura `VolumeData` possono essere compresi in maniera abbastanza intuitiva:

- il campo `rec_smolist` è un puntatore ad un array di puntatori a liste di vnode
- il campo `nsmallvnodes` rappresenta il numero di vnode allocati sul volume
- il campo `nsmallList` rappresenta il numero di liste di vnode

```
typedef struct VolumeDiskData {
    struct versionStamp stamp;
    VolumeId    id;
    [...]
    byte        inUse;
    [...]
    byte        blessed;
    [...]
    int         type;
    VolumeId    groupId;
    VolumeId    cloneId;
    VolumeId    backupId;
    byte        needsCallback;
    [...]
}
```

La struttura `VolumeDiskData` contiene informazioni di natura amministrativa che vengono memorizzate in RVM. I suoi campi possono essere così descritti:

- `stamp` contiene informazioni utili al versioning
- `id` è un identificativo del volume, univoco per tutti i sistemi
- `inUse` indica se il volume è in uso (ossia on line) o se è avvenuto un crash del sistema mentre lo si stava usando
- `blessed` è una flag impostata da un utente con diritti amministrativi ed indica se il volume può o meno essere messo on line
- `type` è un intero i cui valori sono definiti dalle costanti `RWVOL` (volume read/write), `ROVOL` (volume read-only), `BACKVOL` (volume di backup)
- `groupId` rappresenta l'identificativo del gruppo di replicazione. Assume valore 0 se il volume non è replicato
- `cloneId` rappresenta l'identificativo dell'ultimo clone read-only. Assume valore 0 se il volume non è stato mai clonato
- `backupId` rappresenta l'identificativo dell'ultima copia di backup di questo volume read/write
- `needsCallback` è una flag settata dal salvager se non è cambiato niente sul volume

7.1.2 VM structures

In questo paragrafo focalizzeremo la nostra attenzione soltanto sulla struttura `Volume`, la cui descrizione risulta essere maggiormente propedeutica alla nostra analisi, rispetto a tutte le altre VM structures.

```
struct Volume {
    VolumeId    hashid;
    struct      volHeader *header;
    struct DiskPartition *partition;
    [...]
    bit16       cacheCheck;
    short       nUsers;
    [...]
    long        updateTime;
    struct      Lock lock;
    PROCESS     writer;
    struct  ResVolLock VolLock;
    [...]
  }
```

Ad ogni volume è associata una struct `Volume` che ne rappresenta il principale punto di accesso. Tali strutture vengono poi memorizzate tutte nell'hashtable `VolumeHashTable` (cfr. paragrafo 7.2). I suoi campi possono essere così descritti:

- `hashid` identificativo in VolumeHashTable
- `header` è un puntatore alla struttura volHeader che a sua volta contiene puntatori LRU e una struttura di tipo `VolumeDiskData`
- `partition` è un puntatore ad una struttura contenente informazioni circa la partizione Unix su cui risiede il volume
- `cacheCheck` è un sequence number utilizzato per invalidare la cache delle entry di vnode nel momento in cui il volume passa allo stato offline
- `nUsers` rappresenta il numero di utenti di questo volume
- `updateTime` è il timestamp relativo a quando questo volume è stato inserito nella lista dei volumi modificati
- `lock` rappresenta un lock a livello di volume utilizzato per il processo di *resolution*/*repair*
- `writer` è l'identificativo del processo che ha acquisito il lock di scrittura

- `VolLock` è semplicemente un lock interno al volume

7.2 Callbacks

Il server tiene traccia degli host che sono up attraverso la tabella `HostTable`. Ogni entry della tabella contiene:

- una lista di connessioni
- un callback connection id
- timestamp dell'ultima chiamata da parte dellhost

Il server tiene traccia delle callback concesse attraverso una hashtable di `FileEntry` e ogni qual volta viene concessa una callback ad un client, nella hashtable viene inserita una `FileEntry` che si riferisce al file relativo alla callback.

Ogni `File Entry` contiene:

- un vice file id (`ViceFid`)
- il numero di utenti del file
- una lista di callback (struct `CallbackEntry`)

Ogni `CallbackEntry` contiene una `HostTable`, attraverso la quale è possibile notificare tutti i client che detengono la callback nel momento in cui il file viene modificato.

Tutte le strutture relative alle callback sono gestite in VM.

Di seguito descriveremo la routine AddCallBack che, come suggerisce il nome, è richiamata nel momento in cui bisogna concedere ad un client una callback su un file.

```
CallBackStatus AddCallBack(HostTable *client, ViceFid
*afid)
{
 char aVCB = (afid->Vnode == 0 && afid->Unique == 0);
```

Il primo parametro della funzione rappresenta l'hostTable di un client a cui bisogna aggiungere la callback, mentre il secondo è il vice file id del file a cui la callback si riferisce.

```
        if (CheckLock(&tf->cblock)) {
              return(NoCallBack);
        }
```

```
    struct FileEntry *tf = FindEntry(afid);
        if (!tf) {
            /* Create a new file entry. */
            tf = GetFE();
            tf->theFid = *afid;
            tf->users = 0;
            tf->callBacks = 0;
            long bucket = VHash(afid);
            tf->next = hashTable[bucket];
            hashTable[bucket] = tf;
        }
```

Verifica se il server già possiede (nella sua hashtable) un fileEntry relativo ad `afid`. Se cè, `tf` ne conserva il puntatore altrimenti viene creata una nuova `FileEntry` a cui `tf` punterà che viene poi inserita nella hashtable.

Non è possibile invocare una `AddCallBack` annidata in una `breackCallBack`

```
        struct CallBackEntry *tc;
        for (tc = tf->callBacks; tc; tc = tc->next)
            if (tc->conn == client) return(CallBackSet);
        tf->users++;
        if (aVCB) VCBEs++;
        tc = GetCBE();
        tc->next = tf->callBacks;
        tf->callBacks = tc;
        tc->conn = client;
        return(CallBackSet);
    }
```

Scorre la lista di `CallBackEntry` relativa a `tf` per verificare se il client che si sta aggiungendo non sia già presente ed ovviamente lo aggiunge soltanto se la ricerca ha dato esito negativo.

7.3 RPC processing

Tutte le routine che fanno parte degli internals del Server Coda e che sono legate all'RPC processing hanno una strutturazione simile che può essere

schematizzata nelle seguenti fasi:

1. **Validazione dei parametri**: questa fase (svolta dalla Routine `ValidateParms()`) si occupa essenzialmente di verificare che i parametri passati via RPC siano compatibili con le strutture dati effettivamente utilizzate nella routine che mappa l'operazione server associata a detta RPC.
2. **GetObject**: verifica lintegrità del FID e recupero dell'oggetto corrispondete (routine `GetFsObj()`)
3. **Verifica della semantica**
4. **Controllo sulla concorrenza**
5. **Controllo dell'integrità**
6. **Controllo sui permessi**
7. **Esecuzione delle operazioni**: questa fase è l'unica contestuale all'operazione server
8. **Inserimento oggetti**: transazioni RVM, eliminazione i-node

Tutte le operazioni server sono implementate da funzioni la cui firma ha la forma `FS_Vice_<operation>()` e per ognuna di queste funzioni, l'implementazione dei punti 1, 2, 3, 5 è sempre uguale. Ciò che cambia, invece, è l'implementazione del punto 4 (*perform operation*).

Ogni funzione suddetta richiama una `Perform<operation>()` diversa che svolge le operazioni specifiche del caso. Tra le tante, si nota la particolarità della `FS_Vice_Fetch()` la cui perform operation consiste in uno stub vuoto, poichè il fetch è una conseguenza dell'implementazione del `GetFsObj()` (punto 2).

A Mosix File System

Mosix è un estensione che permette a un cluster Linux o un computer grid di eseguire come se fosse un singolo computer multiprocessore.

In questo capitolo, prenderemo come riferimento la versione Open Source di Mosix: OpenMosix. [7]

La principale peculiarità del Mosix File System risiede nel fatto che non vi è alcuna necessità di scrivere applicazioni ad hoc, poichè tutte le estensioni risiedono nel kernel; pertanto tutte le applicazioni, mediante l'intefaccia del VFS, godono automaticamente e trasparentemente dei benefici di OpenMosix.

Il maggior punto di forza di OpenMOSIX è rappresentato dall'efficienza con cui esegue in maniera distribuita processi CPU-bound. Tuttavia, per

[7] Tale versione risulta reperibile all'indirizzo openMosix `http://openmosix.sf.net`

gestire in maniera altrettanto efficiente anche i casi di processi I/O-bound, si possono utilizzare (sui singoli nodi) file system con supporto DFSA (*Direct File System Access*)

A.1 Direct File System Access

La caratteristica fondamentale di un supporto *DFSA* risiede nell'eseguire le operazioni di I/O (da parte di un processo in locale) sul nodo in cui è in esecuzione il processo e non via rete. Ciò consente di ridurre notevolmente gli overhead.

Affinchè un file system possa abilitare il supporto DFSA, è necessario che esso soddisfi dei requisiti fondamentali:

- Uno stesso mount point su tutti i nodi
- File consistency
- Time-stamp consistency

E' importante segnalare che attualmente pochissimi file system supportano DFSA.[8]

A.2 Caratteristiche di Mosix File System

OpenMosix è dotato di un proprio file system distribuito conforme alle specifiche DFSA. Tale file system prende il nome di **MFS** (*Mosix File System*).

La linea progettuale seguita dal creatore di Mosix File System consiste nello sviluppare un layer astratto che, poggiandosi sui file system residenti sui nodi, permetta di offrire agli utenti finali una visione unificata globale. Infatti Mosix File System non è un un file system che scrive dati su un proprio disco ma, per lo storage, si basa su qualsiasi file system supportato da Linux.

MFS permette un accesso parallelo ai file, mediante una distribuzione adeguata degli stessi. La differenza sostanziale tra MFS ed altri DFS (quali *NFS, AFS, etc.*) risiede nella migrazione dei processi e dei dati a cui essi fanno accesso: in MFS un processo migra verso i nodi che ospitano i file di cui necessita mentre con altri DFS sono i dati a seguire i processi.

Come descritto dalla figura, con MFS, tutte le direcotry e file regolari di tutti i nodi sono disponibili da ogni nodo del cluster OpenMosix attraverso il mountpoint `/mfs`.

[8] Tra i file system che supportano DFSA citiamo *GFS, xFS, Frangipani*

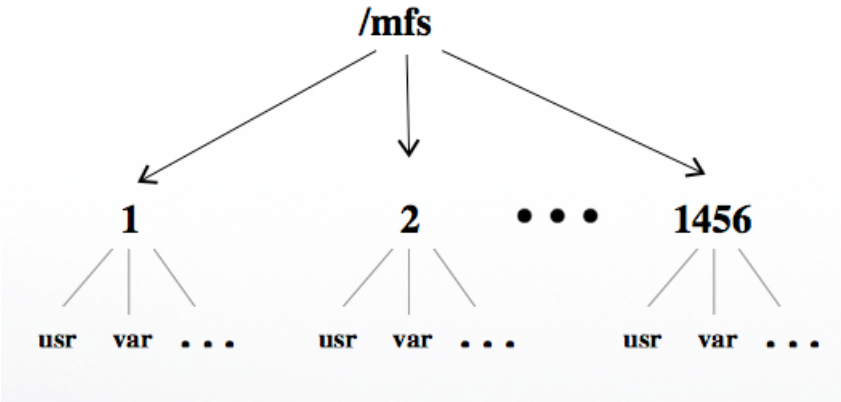

Figura 8: Struttura ad albero del file system di Mosix

E possibile quindi copiare file tra i vari nodi o accedere a file residenti su nodi diversi, direttamente attraverso il suddetto mountpoint.

Ciò permette ad un processo migrato su un nodo remoto di poter eseguire chiamate di sistema (open, close, read, etc.) senza dover tornare sul nodo di origine.

Al contrario di altri file system, MFS fornisce una consistenza per singolo nodo, mantenendo una sola cache sul (sui) server. Per implementare ciò, le cache disco standard di Linux sono usate solo sul server e vengono bypassate sui client.

A.3 Conclusioni

Concludiamo questa breve panoramica su Mosix File Sysem elencando alcuni dei vantaggi e degli svantaggi a noi pervenuti durante la nostra analisi.

- *Vantaggi:*
  - L' approccio di caching, descritto nella precedente sezione, fornisce un semplice ma scalabile schema per la consistenza
  - Le interazioni client-server avvengono a livello di system call; ciò rende altamente performanti le operazioni di I/O molto grandi
- *Svantaggi*
  - Nessun supporto per la alta disponibilità: il guasto di un nodo impedisce ogni accesso ai file del nodo stesso
  - MFS nasce (e resta tale) per l'utilizzo esclusivo in (Open)Mosix.

– Le interazioni client-server a livello di system call sono a discapito di operazioni di I/O piccole, non essendo presenti cache sui client

Riferimenti bibliografici

[1] *Distributed File Systems: concepts and examples*, E. Levy - A. Silberschatz, 1990

[2] *Sistemi Operativi (Quinta Edizione)*, Silberschatz - Galvin, 1998

[3] *Andrew: a distributed personal computing enviroment*, J.H. Morris - M. Satyanarayanan - M.H. Conner, 1986

[4] *The MOSIX Direct File System Access Method for Supporting Scalable Cluster File Systems*. Amar-Barak-Shiloh, 2003

[5] *Removing Bottlnecks in Distributed File Systems: Coda & Intermezzo as examples*, P. J. Braam - P. A. Nelson, 2000

[6] *Coda: A Highly Available File System for a Distributed Worksation Environment*, M. Satynarayanan, 1995

[7] *The Coda Distributed File System*, Linux Journal, June 1998

[8] *Disconnected Operation in the Coda File System*, James J. Kistler and M. Satyanarayanan, 1997

[9] *Coda File System Web Site* (http://www.coda.cs.cmu.edu)

[10] The MOSIX Scalable Cluster File Systems for LINUX, Lior Amar et al., 1999

[11] Intermezzo Web Site (http://www.inter-mezzo.org)