Distributed File Systems

di

Pironti Antonio, Russo Vincenzo

{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Sommario

- Introduzione ai File System Distribuiti
- Tipologie di servizio
- Requisiti
- Semantiche della consistenza
- Problematiche
- Overview
  - Andrew File System
  - Coda
  - InterMezzo
- Approfondimenti
  - Rimozione di Bottlenecks nei DFS
    - Esempi: Coda e Intermezzo
  - Coda Internals
- Appendice
  - Mosix File System

Introduzione ai DFS (1/3)

- I file system distribuiti (DFS) nascono con lo scopo di permettere ad utenti di macchine fisicamente distribuite di condividere dati e risorse di memorizzazione usando un file system comune.
- Una tipica configurazione per un DFS è un insieme di workstation e server interconnessi da una LAN.
- Un DFS è implementato come parte del sistema operativo di ogni computer connesso alla rete.

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Introduzione ai DFS (2/3)

- Per descrivere meglio la struttura di un DFS occorre definire i termini *servizio*, *server* e *client*
- 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*

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Introduzione ai DFS (3/3)

- Un DFS è 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 banalmente svariati e indipendenti.
- La configurazione e l'implementazione concrete di un DFS possono essere di vario tipo: configurazioni con server su macchine dedicate oppure configurazioni in cui una macchina può essere sia server che client
- Un DFS può essere implementato, tra gli altri modi, come parte di un sistema operativo distribuito o come uno strato software che si occupa della gestione della comunicazione tra i sistemi operativi convenzionali e i file system

Tipologie di servizio (1/3)

- Servizio con informazione di stato (*statefull*)
  - ✳ All'apertura di un file si inizializza una sessione di comunicazione tra client e server. La sessione termina alla chiusura del file o per mezzo di un meccanismo di *garbage collection*.
  - ✳ Il server fornisce al client un identificatore di connessione, usato per i successivi riferimenti al file.
  - ✳ Migliori prestazioni
    - ✤ l'informazione sul file è sottoposta a caching in memoria centrale ed è possibile accedervi tramite l'identificatore di connessione, risparmiando accessi al disco.
    - ✤ il server conosce lo scopo delle richieste del client e può agire di conseguenza
      - ✓ *esempio* - pre-fetching dei blocchi di un file aperto in lettura sequenziale

Tipologie di servizio (2/3)

- Servizio senza informazione di stato (*stateless*)
  - Nessuna informazione di stato.
  - Le richieste sono autocontenute: ogni richiesta identifica completamente la posizione il file e la posizione al suo interno, per gli accessi di lettura/ scrittura.
  - Le operazioni di apertura e chiusura file non causano trasmissioni di messaggio remoto, al contrario, ovviamente, delle operazioni di lettura/ scrittura.
  - Prestazione inferiori al servizio *statefull*
    - 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

Tipologie di servizio (3/3)

- Fault tollerance nel servizio statefull
  - In caso di guasti al server, c'è bisogno 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
  - In caso di guasti ai client, questi devono essere notificati al server, per dar via alla fase conosciuta come *rilevamento ed eliminazione degli orfani*
  - Conclusioni: fault tollerance tedioso
- Fault tollerance nel servizio stateless
  - Un server senza informazione di stato non presenta gli stessi problemi di cui sopra e subito dopo il ripristino può rispondere senza problemi alle richieste, essendo esse autocontenute
  - Dal punto di vista dei client, non esiste differenza tra un server lento e uno in via di ripristino

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Requisiti (1/4)

- I primi requisiti di un DFS sono quelli classici dei file system per sistemi multiutente e multiprogrammati:
  - *struttura dei nomi consistente*
  - *interfaccia per le applicazioni*
  - *mapping dei nomi*
  - *garanzia di integrità dati*
  - *sicurezza*
  - *controllo di concorrenza*
- Vanno aggiunti, come 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*

Requisiti (2/4)

- Altro requisito fondamentale per un DFS è la *trasparenza*.
- In linea del tutto generale, ciò significa che un DFS deve apparire ai client come un file system centralizzato convenzionale
- La molteplicità e la dispersione dei server e dei dispositivi di memoria 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)
- E' compito del DFS localizzare i file e predisporre il trasporto dei dati.

Requisiti (3/4)

- 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
    - L'URI di un file non deve essere modificata se cambia la posizione fisica dello stesso

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Requisiti (4/4)

- *Trasparenza di mobilità*
  - I programmi devono funzionare anche se vengono spostati da un server all'altro
- *Trasparenza delle prestazioni*
  - Le prestazioni di un DFS, dal punto di vista del client, devono essere paragonabili a quelle di un file system locale
  - Le prestazioni del file system devono poter essere aumentate per far fronte ad aumenti di carico di lavoro (*scalabilità*)

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Semantiche della consistenza (1/3)

- Semantica UNIX
  - Vige la politica *last write wins*
  - Tutti i cambiamenti sono immediatamente visibili a tutti i processi che hanno aperto un determinato file
  - Il file è associato ad una singola immagine fisica, vista quindi come una risorsa esclusiva. La contesa causa ovvi ritardi ai processi utente.
  - *Svantaggio*: impossibile rafforzare la semantica per un miglior adattamento a un DFS (l'unica soluzione sarebbe che tutti gli accessi avvengano al server; ciò è palesemente inaccettabile)
  - Utilizzata dai seguenti DFS: Sprite, DCE/DFS

Semantiche della consistenza (2/3)

- Semantica delle Sessioni
  - ✳ 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, le modifiche apportate saranno visibili solo nelle sessioni che inizieranno successivamente
  - ✳ Questa semantica fa sì che un file sia associato a più immagini fisiche, probabilmente diverse, nello stesso momento. Ciò favorisce gli accessi concorrenti di lettura/scrittura, riducendo ritardi
  - ✳ Utilizzata da moderni DFS: AFS, AFS2, Coda, Intermezzo

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Semantiche della consistenza (3/3)

- Semantica dei file condivisi immutabili
  - ✳ Semantica di semplice implementazione: un file condiviso diviene immutabile nel nome e nel contenuto.
  - ✳ Consentiti accessi in sola lettura ai file condivisi.
- Semantiche deboli
  - ✳ Criteri: aging, timeout, etc.
  - ✳ Utilizzate in: NFS, SMB/CIFS

Problematiche (1/4)

- Problematiche di primo livello
  - ✳ Massimizzare le prestazioni generali
    - ✤ Utilizzo di caching
    - ✤ Utilizzo di servizi con informazione di stato
  - ✳ Massimizzare la disponibilità delle informazioni sul mapping [nome file, locazione di memoria]
    - ✤ Replicazione dei file
  - ✳ Mantenere gestibile il mapping [nome file, locazione di memoria] necessario all'implementazione del *trasparent naming*
    - ✤ Aggregare insieme di file in unità componenti 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)

Problematiche (2/4)

- Le problematiche citate inducono a problemi secondari
  - Locazione della cache
    - *Cache su disco* - 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

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Problematiche (3/4)

- 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.
  - *Approccio iniziato dal client*: il client contatta il server per un controllo di coerenza. Il fulcro di questo approccio è la frequenza di questo controllo, che può caricare sia la rete che il server
  - *Approccio iniziato dal server*: il server tiene traccia delle porzioni di file che sottopone a caching per i client. Quando rileva una potenziale incoerenza (ad es: uno stesso file sottoposto a caching per due client, in modalità conflittuali), deve reagire.

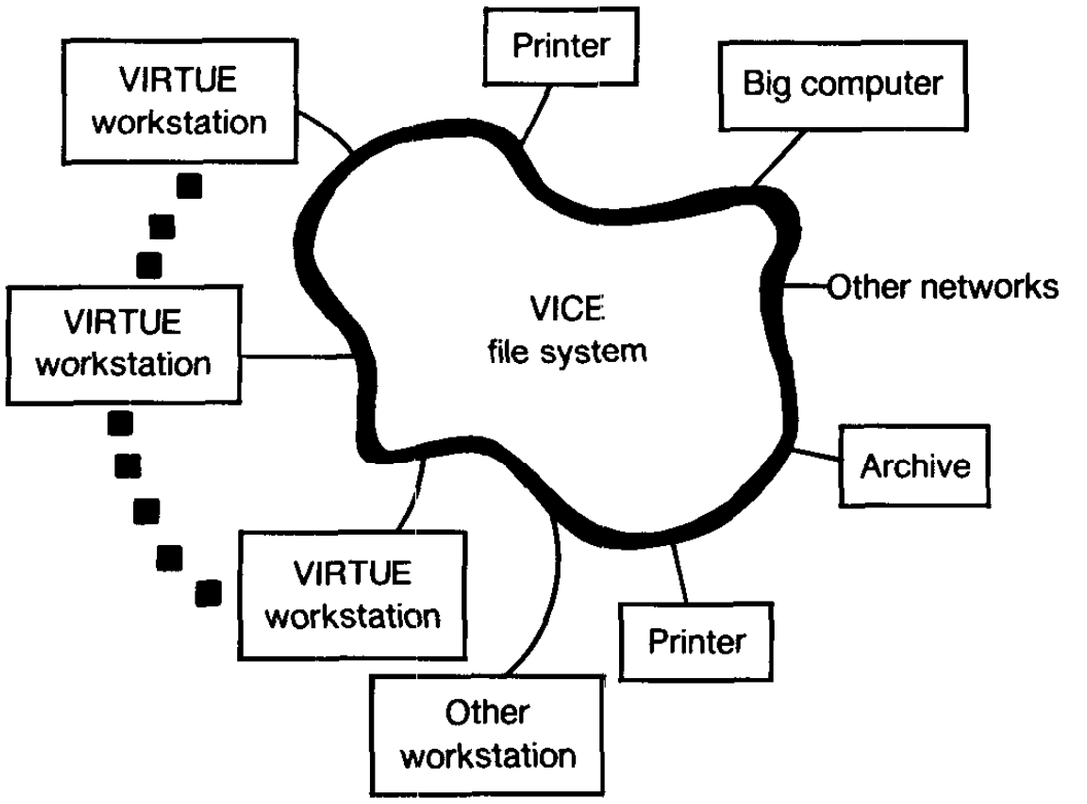
Problematiche (4/4)

- Aggiornamento delle repliche dei file
  - ✳ Dal punto di vista utente, le repliche indicano la stessa unità logica, quindi l'aggiornamento apportato a una replica deve rilfettersi anche su tutte le altre repliche
  - ✳ Bisogna scegliere tra conservare la coerenza ad ogni costo (ciò implica potenziali blocchi indefiniti in caso di guasti) e il sacrificio della coerenza nei casi di guasto, a vantaggio di una garanzia del procedere degli eventi

Andrew File System

- Andrew è un ambiente di calcolo distribuito sviluppato originariamente presso la Carnegie Mellon University. AFS è il suo file system.
- Caratteristica fondamentale di AFS è la scalabilità: Andrew è stato progettato per comprendere oltre 5000 workstation
- AFS ha avuto svariate implementazioni, alcune commerciali come quella di IBM e altre Open Source, come OpenAFS (http://www.openafs.org).
- Sistemi operativi supportati (da OpenAFS): svariati Unix, Linux, OSX/Darwin, Windows 2000/XP/2003

Andrew File System



- Andrew opera una distinzione tra client e server dedicati.
- Server e client sono interconnessi da una LAN.
- I client prendono il nome di *Virtue* dal protocollo che usano per dialogare con i server
- I server sono identificati con *Vice*, che è il nome del software che eseguono

Andrew File System - Spazio dei nomi

- I client sono presentati con uno spazio di nomi frazionato
  - *Spazio di nomi locale*: fornito dal file system locale del client (tutti i client devono necessariamente avere un disco). Nello spazio di nomi locale si trovano programmi di sistema fondamentali per un funzionamento autonomo, file temporanei e file che il proprietario della stazione per ragioni private ha voluto non condividere.
  - *Spazio di nomi condiviso*: presentato ai client dai server Vice, sotto forma di gerarchia di file omogenea con caratteristiche di trasparenza di locazione. Ogni client accede allo spazio di nomi condiviso attraverso il mountpoint `/afs` (a partire dalla root del file system locale).

Andrew File System - Scalabilità

- Osservando la struttura più attentamente, si nota che client e server sono organizzati in cluster interconnessi tramite una WAN.
- Ogni cluster è formato da un insieme di client e un Vice, chiamato *cluster server*
- Tale struttura è nata per affrontare il problema di scala: per prestazioni ottimali, le stazioni di lavoro devono usare il proprio *cluster server* per la maggior parte del tempo, riducendo le operazioni *intercluster*

Andrew File System - Operazioni su file

- Andrew è basato sul caching di interi file: i client interagiscono con i Vice solo durante apertura e chiusura di un file.
- Le system call di open/close di file vengono intercettate dal SO della stazione e inviate a *Venus*, un processo sulla stessa stazione
- Le letture/scritture avvengono in cache, eseguite direttamente dal kernel, senza l'intervento di Venus

Andrew File System - Operazioni su file

- Venus si occupa del caching all'apertura di un file da un server Vice e dell'aggiornamento delle copie master sui Vice al momento della chiusura

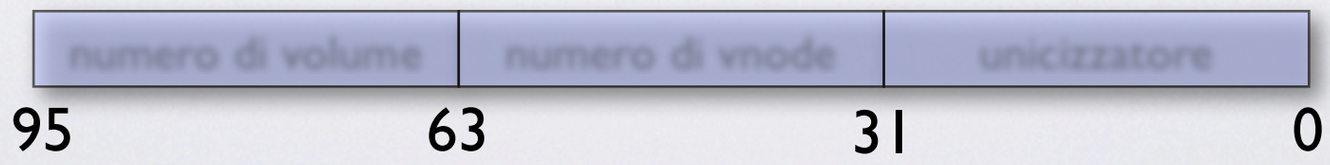
- Venus presuppone che i dati in cache siano validi se non è specificato diversamente: meccanismi che supportano tale politica, prendono il nome di *callback* e riducono il numero di richieste di validazione

- I callback per un client vengono rimossi all'invalidazione dei dati in cache, così, per un client che riapre un file non più coerente con la copia master, viene rieffettuato il caching dei dati per fornire dati coerenti

- AndrewFS adotta la semantica delle sessioni

Andrew File System - Dettagli implemenativi

- Spazio dei nomi condivisi
  - Lo spazio dei nomi condivisi è formato da unità componenti dette *Volumi*;
  - Un file o una directory Vice sono identificati da un identificatore di basso livello: il *fid*.
  - Ogni elemento di directory mappa un path name su un fid. Un fid è lungo 96 bit ed è diviso in tre parti di uguale dimensione *numero di volume*, *numero di vnode* e *unicizzatore*
  - Il *numero di volume* identifica, banalmente, un volume; il *numero di vnode* funge da indice nell'array di inode di un volume; l'*unicizzatore* permette di riutilizzare i numeri di vnode in più volumi



Andrew File System - Dettagli implemenativi

- I processi client sono interfacciati ad un kernel Unix 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

- 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 systemcall

- Venus gestisce due cache diverse: una per i dati e l'altra per lo stato. E' utilizzato un semplice algoritmo LRU, per contenere la dimensione delle cache.

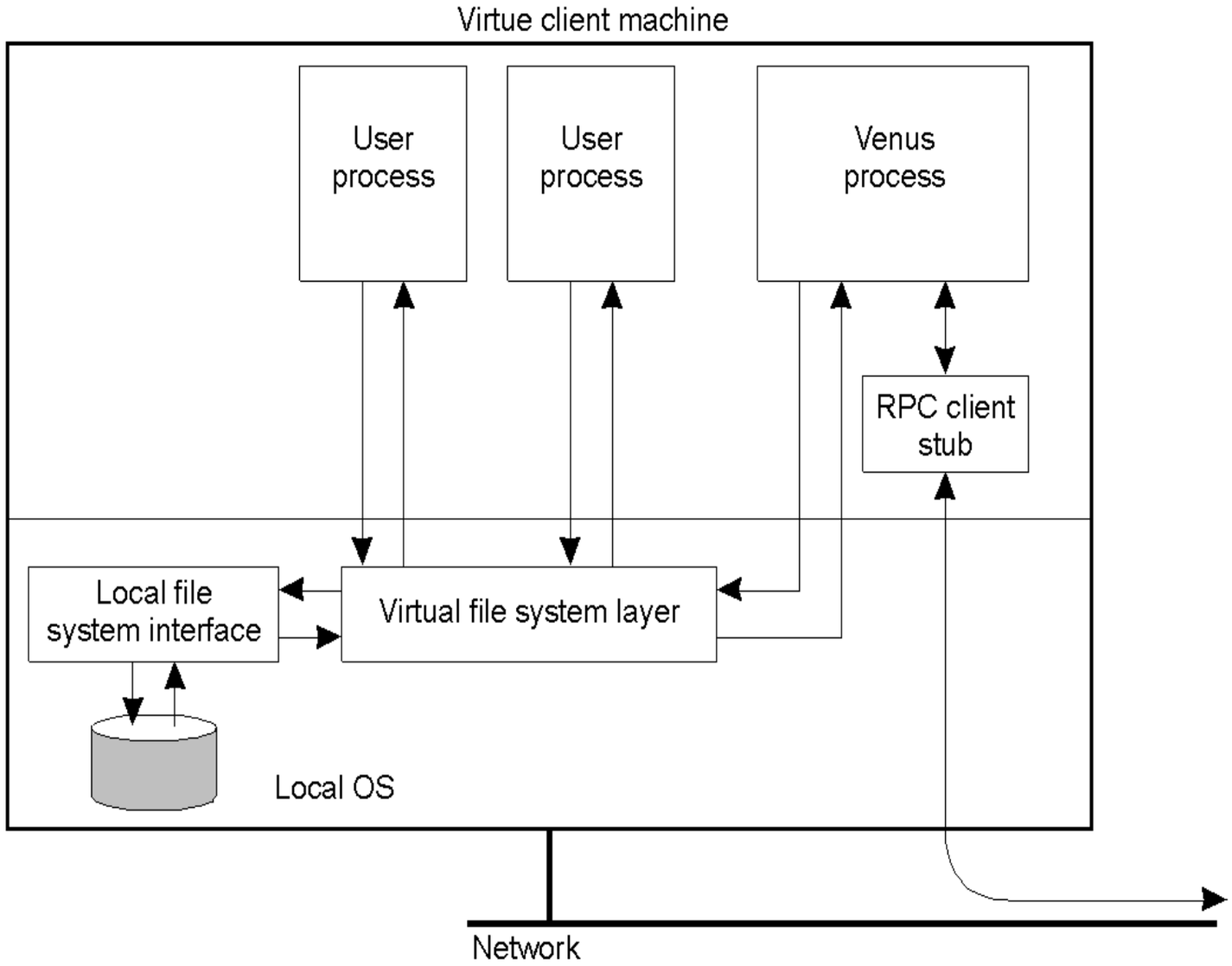
- Ogni server ha un singolo processo multithreaded per servire concorrentemente le molte richieste dei client

- L'utilizzo di un singolo processo multithreaded permette il caching delle strutture dati necessarie per soddisfare le richieste di servizio; come contro abbiamo che se muore il processo, il server è bloccato.

Andrew File System - Conclusioni

- Chiudiamo questa panoramica su AFS con un elenco dei vantaggi rispetto al diffuso e comune NFS
  - ✳ 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*
  - ✳ Maggiori performance grazie alla cache locale che riduce il carico di rete
  - ✳ Scalabilità
  - ✳ Maggiore sicurezza (utilizza un sistema di autenticazione Kerberos)
  - ✳ Ammette replicazione dei volumi, garatendo maggiore disponibilità e sicurezza

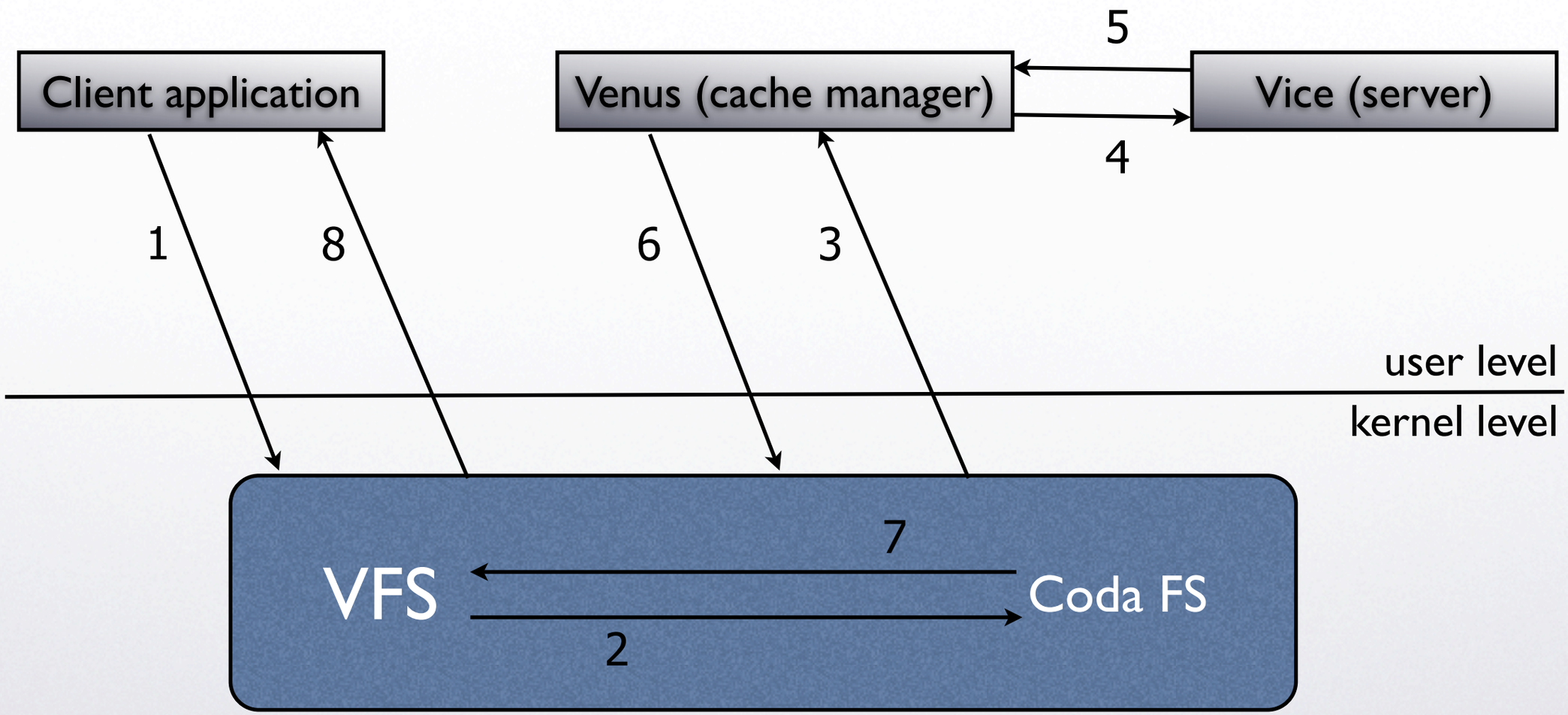
Coda File System - Intro



- Coda è basato su AFS2 ed è stato svilupatto presso Carnegie Mellon University, come lo stesso AFS
- La struttura è sempre quella di AFS: *Vice* servers e *Virtue* client che eseguono il processo Venus

Coda - Interazione dei moduli

- Di seguito un grafico che mostra l'interazione tra un'applicazione, il kernel e i moduli user-level di coda



Coda File System - Caratteristiche

- Scalabilità e performance
  - Addattamento alla larghezza di banda
  - Disconnected operations
  - Reintegrazione dei dati dopo le riconnessioni
- Robustezza
  - Replicazione server
  - Risoluzione dei conflitti server/server
- Consistenza
  - Il modello di consistenza è simile a quello di AFS; in ogni caso la semantica della consistenza adottata è ancora la *semantica delle sessioni*
- Caching
  - Lo schema di caching utilizza il *callback* come in AFS
  - La politica di caching è *write-back* (nelle ultime versioni)

Coda - Replicazione dei Sever

- L'unità di replicazione in Coda è un *volume*.
  Un volume è una collezione di file che sono memorizzati in un server da un parziale sotto albero condiviso nel file name space.
- L'insieme dei server che contengono le repliche di un volume è definito *VSG* (Volume Storage Group)
- I dati che non sono frequentemente richiesti possono essere immagazzinati in volumi non replicati.
- Coda supporta anche la read-only replication dei volumi (caratteristica eriditata da AFS)

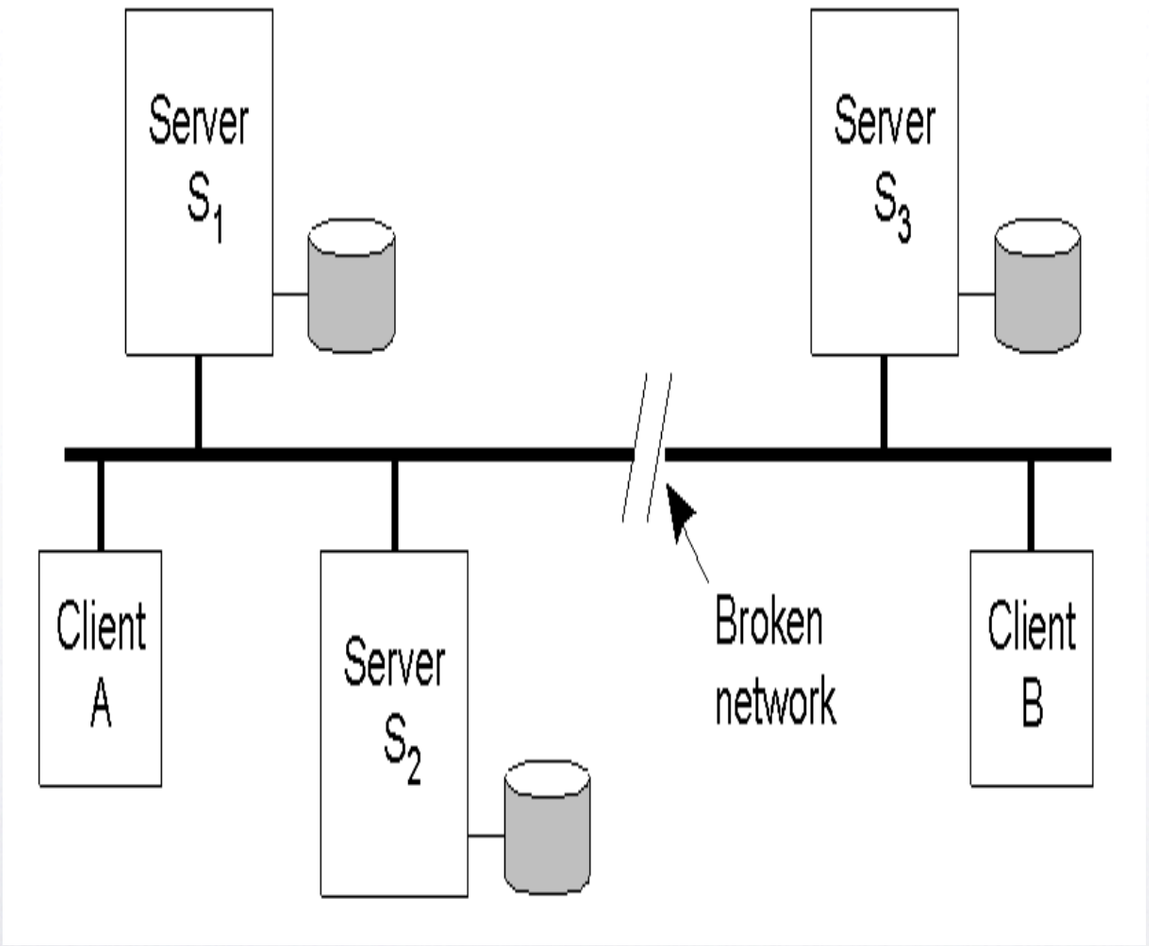
Coda - Replicazione dei Sever

- Per ogni volume che ha dati in cache, Venus tiene traccia del sotto insieme di VSG che è raggiungibile al momento. Questo sottoinsieme è chiamato *AVSG* (Accessible Volume Storage Group).
- Un server viene eliminato dall'AVSG quando un'operazione va in time out e verrà reinserito nell'insieme quando Venus riuscirà a ristabilire la connessione.
- Ogni client può avere differenti AVSGs per ogni volume.

Coda - Replicazione dei Sever

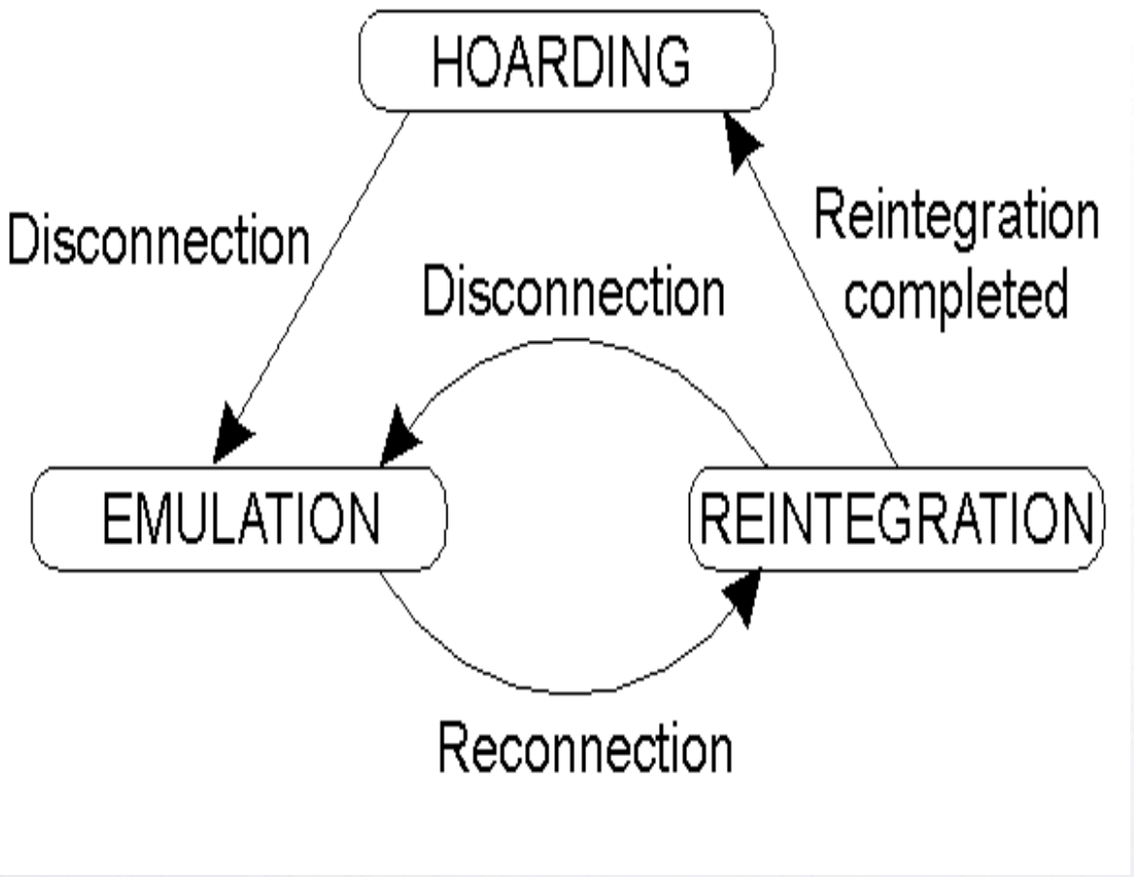
- La strategia di replicazione adottata è una variante dell'approccio *read-one, write-all*
- Quando un file è chiuso dopo la modifica viene trasferito a tutti i membri dell' *AVSG*.
- Questo approccio massimizza la probabilità che ogni replica abbia il dato aggiornato ogni volta.
- Uno svantaggio di questo approccio è la latenza nella propagazione della sincronizzazione che viene minimizzato adoperando un meccanismo di chiamate parallele a procedure remote.

Coda - Disconnected Operations



- Le disconnected operations permettono ai client di lavorare quando un membro del VSG è irraggiungibile
- Durante il periodo offline, i dati vengono serviti dalla cache, i cambiamenti vengono fatti in cache e scritti in un log (*CML*, Client Modification Log).
- Alla riconnessione, il log viene dato al sever per la reintegrazione dei dati.

Coda - Disconnected Operations



- La disconnessione è solitamente trasparente all'utente
- La disconnessione può essere anche voluta, permettendo ad un laptop di poter continuar ad operare isolato dai Coda servers per un lungo periodo
- Grazie alle DO, è possibile usare senza problemi reti a basse performance (wi-fi), al contrario di quanto accadeva con AFS

Coda - Conclusioni

- Vantaggi
  - Disconneted operations
  - Reintegration
  - Write-back caching
- Ne seguono gli obiettivi primari di Coda
  - Alta scalabilità
  - Alta disponibilità
- Svantaggi
  - Alcune system call (open, close) per operazioni su file contattano sincronamente il cache manager ad ogni chiamata
    - Vedremo successivamente come diminuire i danni (in termini di performance) causati da questo approccio

Intermezzo File System - Intro

- Rappresenta una reingegnerizzazione del file system Coda
- Il progetto ha l'obiettivo di creare un file system "più piccolo" - in termini di codice - rispetto a Coda, mantenendo comunque le sue principali caratteristiche
- Le principali linee progettuali sono:
  - Sfruttare il file system locale come server di memorizzazione e come una client cache
  - Inserire dei filtri nel driver che si interfaccia al file system locale, al fine di contattare il cache manager solo nei casi necessari

Intermezzo File System - Panoramica

- Le componenti di intermezzo e cosa offre
  - Più client che prelevano dati ed inviano modifiche ad uno o più server
  - Procedure di gestione di richieste attraverso la rete (RPC)
  - Buona semantica di condivisione, fornita attraverso protocolli dedicati e informazioni sul versioning
  - Buone performance, ottenibili attraverso la minimizzazione delle interazioni di rete che devono essere rese, al contempo, quanto più efficienti possibile
  - Sicurezza, gestione e backup
  - Implementazione di un file system stratificato attraverso layer che si poggiano su altri file system e ne estendono le funzionalità

Intermezzo - Linee Progettuali

- La memorizzazione dei file da parte del server risiede in un *file system nativo*
- I *client livello kernel* di Intermezzo sfruttano un file system già esistente e hanno una *cache persistente*
- Gli oggetti del file system devono avere dei *metadati* adatti alle operazioni off-line
- Implementazione di un meccanismo di *write-back caching*
- La gestione delle cache client e del server file system differiscono nella politica ma non nei meccanismi

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Intermezzo - Volumi

- Ogni file tree reso disponibile da un cluster di server di Intermezzo è costituito da file sets o *volumi*
- Ogni file set ha una *root directory* e può contenere mount point di altri file set
- Un client può avere un qualsiasi file set come root di un file system Intermezzo
- Ad ogni file set è associato un *file set storage group* che descrive come il server tiene traccia del file set
- Il *file set location data base* (FSLDB) descrive i mount point dei file set e il loro storage group
- Un *cluster Intermezzo* è un gruppo di server che condividono un singolo FSLDB

Intermezzo - Journaling e filtering

- Una *cache* Intermezzo è semplicemente un file system locale incapsulato in un layer (di filtro) aggiuntivo chiamato *Presto* (nei sistemi Linux) e *Vivace* (nei sistemi Windows)
- Tale filtro si occupa di:
  - *Filtrare gli accessi*, per contattare il cache manager solo quando è necessario (dati invalidati, etc.)
  - Effettuare il *journaling* delle modifiche effettuate sul file system
- Tutte le richieste sottoposte a *Presto* sono gestite da un altro layer, denominato *Lento*. Tale layer può agire:
  - Sia come un gestore di cache del sistema user-level
  - Sia come un file server

Intermezzo - Journaling e filtering

- Quando Intermezzo viene montato, Presto viene informato circa il tipo di filesystem che deve incapsulare
  - *Presto* replica tutte le implementazioni dei metodi VFS associati alle *dentry*, agli *i-node* e ai *file object*, relative al file system incapsulato, nelle sue strutture *bottom_ops* (struttura del FS ext2)
- *Presto* filtra tutti gli accessi ed effettua il journaling di tutte le modifiche, ad eccezione di quelle relative a *Lento*, il processo incaricato di gestire la cache

Intermezzo - Dettagli del caching

- Intermezzo si comporta in alcune funzionalità allo stesso modo di AFS
  - ✳ Se l'oggetto non è presente nella cache, ne si effettua il fetch
  - ✳ I file vengono prelevati nella loro interezza
  - ✳ Viene effettuato il caching dell'intero albero delle directory sul client
    - ✤ Ciò evita nei successivi accessi il presentarsi di latenza nella risoluzione di pathname

Intermezzo - Dettagli del caching

- Sia client che server sono in grado di effettuare modifiche e fornire dati. La distinzione chiave sta nel fatto che
  - Un client può effettuare cambiamenti nella sua cache soltanto per liberare spazio. Ovviamente tali modifiche non verranno propagate ai server
- Per modificare un oggetto del FS è necessario innanzitutto acquisire un *Permit* (controllo sulla concorrenza)
- Se il file è modificato su un client, le modifiche sono effettuate sulla cache e journaled in un un opportuno log di modifiche per la *reintegrazione* sui server
  - I file modificati sono coinvolti in un processo di back-fetch da parte del server
  - Se un server si accorge di non avere una directory creata da un client (ossia se una directory è presente in una cache client ma non sul server), allora effettua ricorsivamente il back-fetch della directory e del suo contenuto

Intermezzo - Dettagli del caching

- Se un file è modificato su un server:
  - Si effettua il journaling della modifica e la si propaga ai client registrati per la replicazione
  - Gli altri client invalideranno le loro cache nel momento in cui si accorgeranno della modifica ed effettueranno il *re-fetch* dei dati
- Le operazioni che effettuano i client e i server non sono del tutto simmetriche. Pertanto *Lento*, in esecuzione sui client e sui server, deve essere in grado di offrire:
  - *File Service*: fetching (se in esecuzione su un client) back-fetching (se in esecuzione su un server) di file e directory
  - *Reintegration service*: ricezione dei log delle modifiche (se in esecuzione su un server) e notifica delle modifiche per la reintegrazione (se in esecuzione su un client)

Intermezzo - Lento

- È responsabile di gestire le richieste relative al *file service* da parte della rete o del kernel
- Le richieste non vengono gestite secondo un modello multi-threaded ma mediante l'utilizzo di *sessioni*
  - Quando giunge una nuova 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 attraverso il dispatch di eventi
- Il kernel attiva tutti i gestori di eventi ed è anche responsabile del *garbage collecting* delle sessioni che non possono essere raggiunte da alcun evento

Intermezzo - Conclusioni

- Vantaggi
  - Tutti i vantaggi di Coda
  - Inserimento di un nuovo strato di callback tra kernel e cache manager: riduzione di context switch durante alcune system call
- Svantaggi
  - Il file system attualmente è ancora in via sperimentale e non è stato possibile reperire svantaggi di sorta

Rimozione dei Bottleneck - Intro

- E' possibile ottenere per un DFS prestazioni simili a quelle di un file system locale (il FS locale di riferimento nel seguito è ext2)? La risposta è sì.

- Sistemi come AFS, Coda e altri hanno dimostrato come ridurre il traffico RPC porta ad enormi miglioramenti, mentre NFS mostra i vantaggi di una aggressiva ottimizzazione a livello kernel

- Analizzando Coda e NFS è chiaro che per ottenere prestazioni in lettura simile a un FS locale, c'è bisogno di maggior autonomia del kernel

- Per quanto riguarda le operazioni di scrittura, per eliminare molto overhead è necessario un protocollo di caching di tipo *write-back*, per eliminare la comunicazione sincrona con i *cache manager*

Rimozione dei Bottleneck - Intro

- Come massimizzare le performance al pari di un file system locale?
- I principi più ovvi sono
  - Limitare al minimo il numero di RPC
  - Non usare cache manager sincroni
    - Protocollo di caching *write-back*
  - Usare i file system locali per cache persistenti
- Cercando di arrivare al nostro obiettivo, si mostrerà come InterMezzo sia stato creato appositamente per eliminare i problemi residui in Coda anche dopo l'ottimizzazione

Rimozione dei Bottleneck - Panoramica

- Quando si mira ad un utilizzo read-only di un file system il problema cruciale è quello di ottenere delle ottime perfomance durante la fase di *warming della cache* (primo accesso ai dati, primo lookup di pathname)

- Ci sono svariati modi di effettuare il client caching

  - *Schema a callback*: i dati nella cache del client vengono ritenuti validi finchè il server non rimuove il callback per quel client; ciò accade quando qualche altro client che condivideva in scrittura i medesimi dati, modifica gli stessi

- Si è notato che anche con la cache warmed Coda ha prestazioni inferiori a quelle di ext2

  - Vedremo nel seguito come una seconda relazione di callback tra kernel e cache manager possa migliorare la situazione

Rimozione dei Bottleneck - Panoramica

- Quando si mira ad un utilizzo read/write del file system, il numero di bottlneck che si incontrano è sensibilmente maggiore; server e client caching necessitano di migliorie
- La rete è spesso coinvolta sincronamente e questo incide troppo sulle prestazioni
- L'impiego di un protocollo write-back per il caching riduce sensibilmente il numero di RPC impiegate e quindi l'overhead di rete; nonostante ciò ancora non si raggiungono le performance ambite
- Anche il cache manager di Coda ha prestazione inferiori rispetto a una soluzione che utilizza il filesystem come storage per caching
- Per bypassare questo problema è possibile creare un modulo kernel che faccia da filtro per il file system locale; il filtro interecetta i cache miss ed effettua il journaling delle modifiche, che saranno riportate sul server asincronamente quando necessario

Rimozione dei Bottleneck - Consistenza

- Molte vie possono essere intraprese
  - ✳ “Indovinare”, ovvero, tecniche grezze basate su aging e timeout (NFS, etc.)
  - ✳ Schemi a *oplock* o *callback*: sistemi che riducono l’attività sincrona e fetch ripetuti
  - ✳ Il prezzo degli schemi a callback è che il server deve mantenere in memoria virtuale lo stato circa quali client detengono quali callback
  - ✳ Per mantenere una cache persistente oltre le disconessioni/ reboot, viene usato il meccanismo della *validazione*

Rimozione dei Bottleneck - Consistenza

- Validazione della cache
  - I dati vengono tenuti sui client con un preciso e univoco *version stamp* che diventa l'unico parametro utilizzato per verificare la consistenza
  - In Coda il confronto attraverso *version stamp* viene effetuato prima a livello di volume e successivamente, nel caso di fallimento, a livello di singolo file
  - I *version stamp* sarebbero molto utili come metadati propri del file system, ma bisogna, per ora, memorizzarli come oggetti persistenti
  - Nel caso di configurazione Coda con server replicati, bisogna usare un array di *version stamp*

Rimozione dei Bottleneck

Callback a livello kernel

- Mosteremo ora come ottenere prestazioni simili a un file system locale in una configurazione a sola lettura, paragonando Coda ed Ext2.
- Il test è stato effettuato con un *ls -lR* di una directory di 1500 file e 300 sottodirectory
- Grazie ai callbacks, Coda risparmia molto traffico RPC; ciononostante, anche con cache *warmed*, Coda è veloce la metà di Ext2. A cosa è dovuto?
- La risposta è: il cache manager a livello utente usato da Coda. Vediamo il perchè.

Rimozione dei Bottleneck

Callback a livello kernel

- Anche se da un punto di vista ingegneristico la scelta di Coda è ottima, perchè garantisce un alto grado di portabilità, questo comporta 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), per avere informazioni sulla consistenza, prima di procedere

- Questo fa si che la durata di una open/close per Coda duri 50 volte di più di una open/close per Ext2 e considerando che nell'esecuzione del test 600 su 4500 system call sono del tipo incriminato, si spiega la lentezza di Coda rispetto a Ext2

- Coda è stato modificato facilmente introducendo un nuovo livello di callbacks a livello kernel, che permette di evitare di contattare il cache manager fintantoché il callback interessato non viene revocato

Rimozione dei Bottleneck

Write-back caching a livello utente: Coda

- Coda permette come già visto le *disconnected operations*: i dati vengono serviti dalla cache e i cambiamenti vengono fatti in cache e scritti in un log (*CML*, Client Modification Log). Alla riconnessione, il log viene dato al sever per la reintegrazione dei dati.
- Coda, nel caso di rete a basse prestazioni, usa un comportamento ibrido tra quello normale e quello disconnesso: serve i cache miss con dati coerenti dal server, ma le modifiche sono gestite come nelle disconnected operations.
- Tale modalità prende nome di *trickle reintegration*

Rimozione dei Bottleneck

Write-back caching a livello utente: Coda

- Il write-back caching di Coda poggia le sue basi sul *trickle reintegration*
- Quando un client **A** deve effettuare un'operazione di modifica, deve chiedere un *write-back permit* al server.
- Il server, prima di concedere tale permesso, revoca tutti i callback dei vari client sul file interessato
- Successivamente le modifiche possono esser fatte e scritte nel CML, che verrà trasmesso, asincronamente, al server
- Quando un client **B** chiede il permit, questo va revocato al client **A** e poi vanno reintegrati (trickle reintegration) i dati prima di concedere il permit al client **B**

Rimozione dei Bottleneck

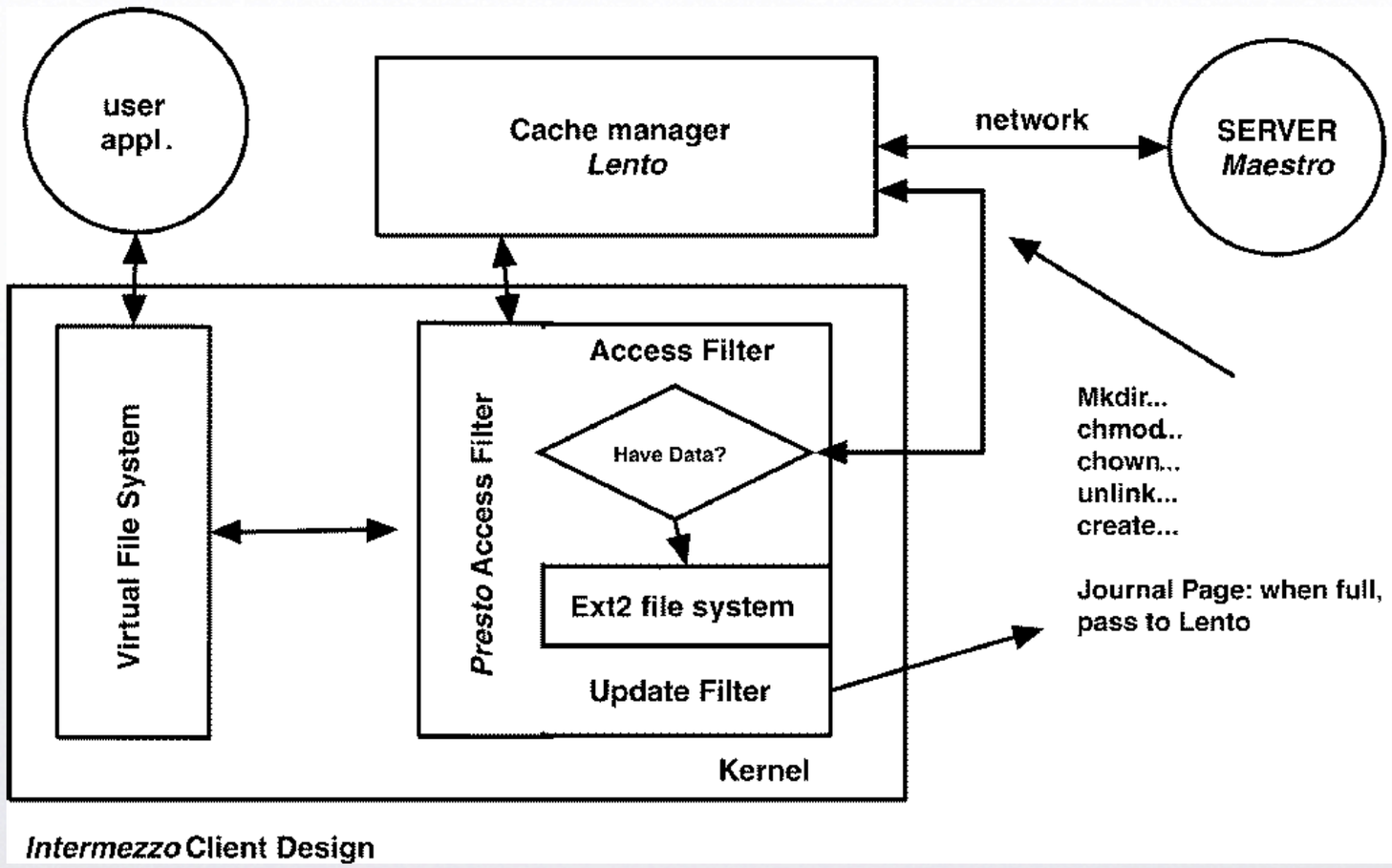
Write-back caching a livello kernel: Intermezzo

- Nonostante quanto appena visto, Coda continua ad essere più lento di Ext2
- Questo ha spinto P. J. Braam a iniziare il progetto Intermezzo
- InterMezzo cerca di oltrepassare un altro limite di Coda: *la lentezza delle strutture dati per il persistent caching*
- InterMezzo preferisce sperimentare l'uso di un file system locale (ext2) come *storage place* per il persistent caching, affiancato sempre ad un cache manager
- Ricordiamo che InterMezzoFS possiede versione stamps, callback, writeback permit e un file protocol simile a coda

Rimozione dei Bottleneck

Write-back caching a livello kernel: Intermezzo

- Ricordiamo con una figura la struttura del client InterMezzo (fortemente diverso da quello Coda) già spiegata nella sezione apposita



Intermezzo Client Design

Coda Client Internals

- In questa sezione verrà presentata l'implentazione nel lato client di Coda, del metodo `open()` del VFS
- La funzione che concretizza tale interfaccia è la `coda_open()`
- L'illustrazione di tale routine (e altre innestate) ha lo scopo di mostrare quanto già detto nella sezione relativa alla risoluzione dei bottleneck: la *open* contatta sempre *Venus* (cache manager user-level)
- Anche per la *close* avviene lo stesso, ma non la tratteremo, poichè, come capiremo di seguito, il discorso è identico

Coda Client Internals - coda_open()

kernel 2.4.31 - fs/coda/file.c

```
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();
}
```

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

Coda Client Internals - upcall.c

- Prima di illustrare la funzione `venus_open()` è bene parlare del contenuto del file *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 struttura di tipo `inputArgs` opportunamente inizializzata dalle funzioni *venus_<fileoperation>()* e poi passata come parametro alla `coda_upcall()`

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Client Internals - venus_open()

kernel 2.4.31 - fs/coda/upcall.c

```
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;
}
```

```
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;
```

Coda Client Internals - coda_upcall()

kernel 2.4.31 - include/linux/coda_psdev.h

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

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;
};

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;
};
```

La struttura **coda_sb_info**
- mantiene informazioni sul superblocco
- puntatore alla Venus wait queue

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
La dimensione del messaggio deve essere almeno 5000 bytes

Coda Client Internals - Kernel & Venus

- Abbiamo visto dalle strutture elencate appena prima, 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 che il kernel invia tramite una **`read()`** sulla device e risponde con una **`write()`** sulla stessa
- Durante lo scambio di messaggi, il processo **P** che ha chiamato la system call viene messo in uno stato di TASK_INTERRUPTIBLE

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
static int coda_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;

     /* Venus non contattabile - Probabile crash o altri problemi */
     if ( !vcommp->vc_inuse ) {

          printk("No pseudo device in upcall comms at %p\n", vcommp);
               return -ENXIO;
     }

     /* Allocazione e formattazione del messaggio da inviare a Venus */
     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;
```

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
static int coda_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;

     /* Venus non contattabile - Probabile crash o altri problemi */
     if ( !vcommp->vc_inuse ) {

          printk("No pseudo device in upcall comms at %p\n", vcommp);
               return -ENXIO;
     }

     /* Allocazione e formattazione del messaggio da inviare a Venus */
     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;
```

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
static int coda_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;

     /* Venus non contattabile - Probabile crash o altri problemi */
     if ( !vcommp->vc_inuse ) {

          printk("No pseudo device in upcall comms at %p\n", vcommp);
               return -ENXIO;
     }

     /* Allocazione e formattazione del messaggio da inviare a Venus */
     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;
```

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
static int coda_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;

     /* Venus non contattabile - Probabile crash o altri problemi */
     if ( !vcommp->vc_inuse ) {

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

```
/* Allocazione e formattazione del messaggio da inviare a Venus */
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;
```

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
/* Aggiunge il messaggio alla coda dei messaggi pending di Venus e risveglia il cache manager */
list_add(&(req->uc_chain), vcommp->vc_pending.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
 * */
wake_up_interruptible(&vcommp->vc_waitq);

/* CodaFS si mette in stato di attesa di segnali, che avranno comunque effetto solo dopo un timeout */
runtime = coda_waitfor_upcall(req, vcommp);

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

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
/* Aggiunge il messaggio alla coda dei messaggi pending di Venus e risveglia il cache manager */
list_add(&(req->uc_chain), vcommp->vc_pending.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
 * */
wake_up_interruptible(&vcommp->vc_waitq);

/* CodaFS si mette in stato di attesa di segnali, che avranno comunque effetto solo dopo un timeout */
runtime = coda_waitfor_upcall(req, vcommp);

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

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
/* Aggiunge il messaggio alla coda dei messaggi pending di Venus e risveglia il cache manager */
list_add(&(req->uc_chain), vcommp->vc_pending.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
 * */
wake_up_interruptible(&vcommp->vc_waitq);

/* CodaFS si mette in stato di attesa di segnali, che avranno comunque effetto solo dopo un timeout */
runtime = coda_waitfor_upcall(req, vcommp);

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

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
/* Aggiunge il messaggio alla coda dei messaggi pending di Venus e risveglia il cache manager */
list_add(&(req->uc_chain), vcommp->vc_pending.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
 * */
wake_up_interruptible(&vcommp->vc_waitq);

/* CodaFS si mette in stato di attesa di segnali, che avranno comunque effetto solo dopo un timeout */
runtime = coda_waitfor_upcall(req, vcommp);

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

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
/* Aggiunge il messaggio alla coda dei messaggi pending di Venus e risveglia il cache manager */
list_add(&(req->uc_chain), vcommp->vc_pending.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
 * */
wake_up_interruptible(&vcommp->vc_waitq);

/* CodaFS si mette in stato di attesa di segnali, che avranno comunque effetto solo dopo un timeout */
runtime = coda_waitfor_upcall(req, vcommp);

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

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
/* Aggiunge il messaggio alla coda dei messaggi pending di Venus e risveglia il cache manager */
list_add(&(req->uc_chain), vcommp->vc_pending.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
 * */
wake_up_interruptible(&vcommp->vc_waitq);

/* CodaFS si mette in stato di attesa di segnali, che avranno comunque effetto solo dopo un timeout */
runtime = coda_waitfor_upcall(req, vcommp);

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

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
/* Aggiunge il messaggio alla coda dei messaggi pending di Venus e risveglia il cache manager */
list_add(&(req->uc_chain), vcommp->vc_pending.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
 * */
wake_up_interruptible(&vcommp->vc_waitq);

/* CodaFS si mette in stato di attesa di segnali, che avranno comunque effetto solo dopo un timeout */
runtime = coda_waitfor_upcall(req, vcommp);

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

Coda Client Internals - coda_upcall()

kernel 2.4.31 - fs/coda/upcall.c

```
/* Aggiunge il messaggio alla coda dei messaggi pending di Venus e risveglia il cache manager */
list_add(&(req->uc_chain), vcommp->vc_pending.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
 * */
wake_up_interruptible(&vcommp->vc_waitq);

/* CodaFS si mette in stato di attesa di segnali, che avranno comunque effetto solo dopo un timeout */
runtime = coda_waitfor_upcall(req, vcommp);

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

Coda Client Internals - Venus

- Dato che *Venus* non verrà trattato negli internals, è giusto spendere qualche parola in proposito
- E' stato già detto come avviene lo scambio di messaggi tra kernel e Venus
- Una volta che Venus ha identificato il contenuto del messaggio, agisce di conseguenza, chiamando 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

Coda Server Internals - Intro

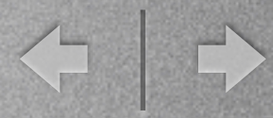
- In questa sezione si parlerà degli internals del server *Vice* di Coda
- A tale proposito, spiegheremo brevemente lo scopo dei seguenti moduli
  - *Salvager* (già presente in AFS)
  - *RVM* (Recoverable Virtual Memory)


Coda Server Internals - Intro

- 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)

Coda Server Internals - Intro

- Con *recoverable virtual memory* si intendono quelle regioni di spazio di memoria virtuale sulle quali esistono *garanzie transazionali*
- Questo significa che sono offerte operazioni atomiche, permanenza e serializzabilità
- RVM è un'implementazione 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 allo stesso tempo un potente tool di fault-tollerance e di transaction-facility

Coda Server Internals - Volume Structure

- Allo startup di un server, viene inizializzata una struttura di tipo **`camlib_recoverable_segment`**
- Tale struttura contiene un array (**`VolumeList`**) 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

Coda Sever Internals - Volume Structure

Coda 5.3.19 - coda_globals.h

```
#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;

        […]
};
```

Coda Sever Internals - Volume Structure

Coda 5.3.19 - coda_globals.h

```
#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;

        […]
};
```

Massimo numero di volumi in ogni partizione

Coda Sever Internals - Volume Structure

Coda 5.3.19 - coda_globals.h

```
#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;

        […]
};
```

Indica se è richiesta o meno l'inizializzazione

Coda Sever Internals - Volume Structure

Coda 5.3.19 - coda_globals.h

```
#define MAXVOLS                 1024

struct camlib_recoverable_segment {
        bool_t  already_initialized;

        struct VolHead VolumeList[MAXVOLS];

        VnodeDiskObject *SmallVnodeFreeList[SM
        VnodeDiskObject *LargeVnodeFreeList[LA

        short     SmallVnodeIndex;
        short     LargeVnodeIndex;

        VolumeId     MaxVolId;

        […]
};
```

Array di headers dei volumi

Coda Sever Internals - Volume Structure

Coda 5.3.19 - coda_globals.h

```
#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;

        […]
};
```

Liste di strutture *vnode* libere

Coda Sever Internals - Volume Structure

Coda 5.3.19 - coda_globals.h

```
#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;

        […]
};
```

Puntatore all'ultimo indice nelle liste di vnode liberi

Coda Sever Internals - Volume Structure

Coda 5.3.19 - coda_globals.h

```
#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;

        […]
};
```

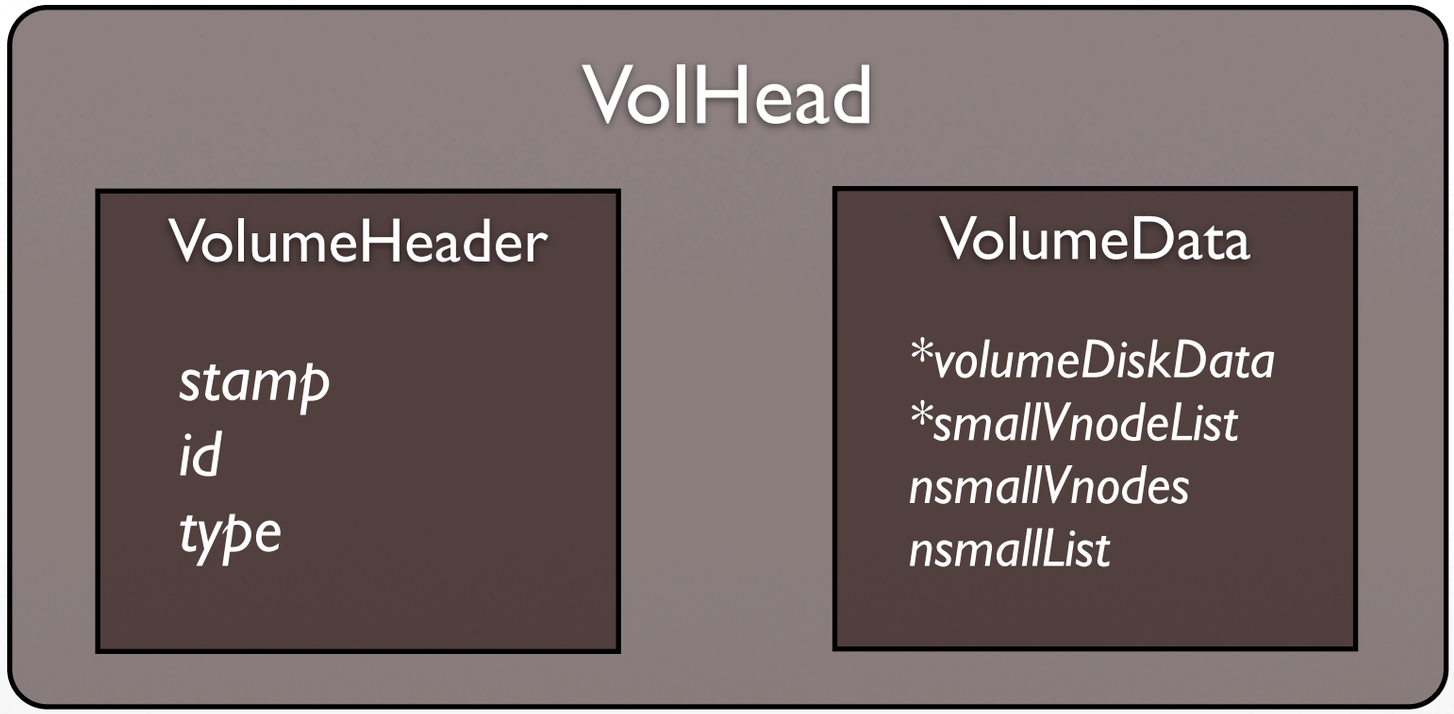
Massimo id di volume allocato su questo server

Coda Sever Internals - Volume Structure

Coda 5.3.19 - coda_globals.h

```
#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;

        […]
};
```

Coda Server Internals - RVM Structure



- 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

Coda Sever Internals - RVM Structure

Coda 5.3.19 - volume.h, camprivate.h

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

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

Coda Sever Internals - RVM Structure

Coda 5.3.19 - volume.h, camprivate.h

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

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

Contiene info utili a rappresentare il volume (info sul versioning, id del volume, tipo del volume, ...)

Coda Sever Internals - RVM Structure

Coda 5.3.19 - volume.h, camprivate.h

Meta dati del volume in RVM

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

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

Coda Sever Internals - RVM Structure

Coda 5.3.19 - volume.h, camprivate.h

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

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

Puntatore ad una struttura VolumeDiskData

Coda Sever Internals - RVM Structure

Coda 5.3.19 - volume.h, camprivate.h

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

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

Puntatore ad un array di puntatori a liste di vnode

Coda Sever Internals - RVM Structure

Coda 5.3.19 - volume.h, camprivate.h

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

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

Numero di vnode allocati

Coda Sever Internals - RVM Structure

Coda 5.3.19 - volume.h, camprivate.h

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

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

Numero di liste di vnode

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

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


Pironti Antonio, Russo Vincenzo

{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

Info utili al versioning

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


Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

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

Id del volume, univoco per tutti i sistemi

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

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

Indica se il volume è in uso (ossia se è on-line) o se è avvenuto un crash del sistema mentre lo si stava usando

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

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

Impostato da un utente con diritti amministrativi. Indica che il volume può andare on-line. Se zero, imposta il volume ad offline

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

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

RWVOL, ROVOL, BACKVOL

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

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

Id del gruppo di replicazione, 0 se il volume non è replicato

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

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

Id dell'ultimo clone read-only, 0 se il volume non è stato mai clonato

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

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

Id dell'ultima copia di backup di questo volume read-write

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - RVM Structure

Struttura contenente informazioni amministrative relative al volume, memorizzate in RVM

Coda 5.3.19 - volume.h, camprivate.h

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

Settato dal salvager se non è cambiato niente sul volume

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - VM Structure

Ad ogni volume è associata una struct **`volume`** che ne rappresenta il principale punto di accesso. Tali strutture vengono memorizzate nell'hashtable **`VolumeHashTable`**

Coda 5.3.19 - volume.h, camprivate.h

```
struct Volume {

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

Coda Sever Internals - VM Structure

Ad ogni volume è associata una struct **`volume`** che ne rappresenta il principale punto di accesso. Tali strutture vengono memorizzate nell'hashtable **`VolumeHashTable`**

Coda 5.3.19 - volume.h, camprivate.h

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

Struttura contenente:
- puntatori LRU
- struttura VolumeDiskData

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - VM Structure

Ad ogni volume è associata una struct **`volume`** che ne rappresenta il principale punto di accesso. Tali strutture vengono memorizzate nell'hashtable **`VolumeHashTable`**

Coda 5.3.19 - volume.h, camprivate.h

```
struct Volume {

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

Info circa la partizione unix su cui risiede il volume

Coda Sever Internals - VM Structure

Ad ogni volume è associata una struct **`volume`** che ne rappresenta il principale punto di accesso. Tali strutture vengono memorizzate nell'hashtable **`VolumeHashTable`**

Coda 5.3.19 - volume.h, camprivate.h

```
struct Volume {

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

Sequence number utilizzato per invalidare la cache delle entry di vnode nel momento in cui il volume passa allo stato offline

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - VM Structure

Ad ogni volume è associata una struct **`volume`** che ne rappresenta il principale punto di accesso. Tali strutture vengono memorizzate nell'hashtable **`VolumeHashTable`**

Coda 5.3.19 - volume.h, camprivate.h

```
struct Volume {

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

Numero di utenti di questo volume

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - VM Structure

Ad ogni volume è associata una struct **`volume`** che ne rappresenta il principale punto di accesso. Tali strutture vengono memorizzate nell'hashtable **`VolumeHashTable`**

Coda 5.3.19 - volume.h, camprivate.h

```
struct Volume {

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

Timestamp relativo a quando questo volume è stato inserito nella lista dei volumi modificati (ossia, la lista dei volumi che dovranno essere se il sistema dovesse andare in crash)

Coda Sever Internals - VM Structure

Ad ogni volume è associata una struct **`volume`** che ne rappresenta il principale punto di accesso. Tali strutture vengono memorizzate nell'hashtable **`VolumeHashTable`**

Coda 5.3.19 - volume.h, camprivate.h

```
struct Volume {

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

Lock interno

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - VM Structure

Ad ogni volume è associata una struct **`volume`** che ne rappresenta il principale punto di accesso. Tali strutture vengono memorizzate nell'hashtable **`VolumeHashTable`**

Coda 5.3.19 - volume.h, camprivate.h

```
struct Volume {

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

Id del processo che ha acquisito il lock di scrittura

Coda Sever Internals - VM Structure

Ad ogni volume è associata una struct **`volume`** che ne rappresenta il principale punto di accesso. Tali strutture vengono memorizzate nell'hashtable **`VolumeHashTable`**

Coda 5.3.19 - volume.h, camprivate.h

```
struct Volume {

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

Lock a livello di volume utilizzato per il processo di resolution/ repair

Coda Server Internals - 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 dell'host
- Il server tiene traccia delle callback concesse attraverso una hashtable di **`FileEntry`**
  - Ogni qual volta viene concessa una callback ad un client, viene inserita una *FileEntry* nella hashtable. Tale *FileEntry* si riferisce al file relativo alla callback

Coda Server Internals - Callbacks

- 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

Coda Sever Internals - Callbacks

Coda 5.3.19 - vicecb.cc

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

    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;
    }
```

Coda Sever Internals - Callbacks

Coda 5.3.19 - vicecb.cc

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

    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;
    }
```

- HostTable di client a cui bisogna aggiungere la callback
- vice file id del file cui la callback si riferisce

Pironti Antonio, Russo Vincenzo
{antonio.pironti@gmail.com, vincenzo.russo@neminis.org}

Coda Sever Internals - Callbacks

Coda 5.3.19 - vicecb.cc

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

    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 relaivo ad **afid**. Se c'è, **tf** ne conserva il puntatore altrimenti viene creata una nuova FileEntry a cui tf punterà

Coda Sever Internals - Callbacks

Coda 5.3.19 - vicecb.cc

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

    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;
}
```

Inserisce la FileEntry creata nella hashtable

Coda Sever Internals - Callbacks

Coda 5.3.19 - vicecb.cc

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

    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);
}
```

Coda Sever Internals - Callbacks

Coda 5.3.19 - vicecb.cc

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

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);
}
```

Coda Sever Internals - Callbacks

Coda 5.3.19 - vicecb.cc

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

    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 …

Coda Sever Internals - Callbacks

Coda 5.3.19 - vicecb.cc

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

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);
}
```

… ed ovviamente lo aggiunge soltanto se la ricerca ha dato esito negativo

Coda Server Internals - RPC Processing

- La struttura di ogni operazione server è la seguente:

  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 l'integrità del FID e recupero dell'oggetto corrispondete (routine GetFsObj)

  3. Verifica della semantica
     - Controllo sulla concorrenza
     - Controllo dell'integrità
     - Controllo sui permessi

  4. Esecuzione delle operazioni: questa fase è l'unica contestuale all'operazione server

  5. Inserimento oggetti: transazioni RVM, eliminazione i-node

Coda Server Internals - RPC Processing

- Le operazioni server sono implementate da funzioni la cui firma ha la forma **`FS_Vice_<operation>()`**
- Per ognuna di queste funzioni, l'implementazione dei punti 1, 2, 3, 5 è sempre uguale; cambia invece 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)

Mosix File System - Introduzione

- Mosix è un estensione che permette a un cluster Linux o un computer grid di eseguire come se fosse un singolo computer multiprocessore.
- Sarà preso come riferimento la versione Open Source: OpenMosix (http://openmosix.sf.net)
- Non si necessita di scrivere applicazioni ad hoc, poichè tutte le estensioni sono nel kernel e tutte le applicazioni "normali" automaticamente e trasparentemente godono dei benefici di OpenMosix

Mosix File System - Introduzione

- OpenMOSIX è molto efficiente nell'eseguire in maniera distribuita processi CPU-bound
- Per gestire in maniera efficiente anche i casi di processi I/O-bound si possono utilizzare, sui nodi, file system con supporto DFSA (Direct File System Access)

Mosix File System - DFSA

DFSA (Direct File System Access)

- Le operazioni di I/O vengono effettuate da un processo in locale, sul nodo in cui è in esecuzione e NON via rete, per evitare ulteriori overhead
- Il file system deve avere:
  - Uno stesso mount point su tutti i nodi
  - File consistency
  - Time-stamp consistency
- Attualmente pochissimi file system supportano DFSA (GFS, xFS, Frangipani, etc.)

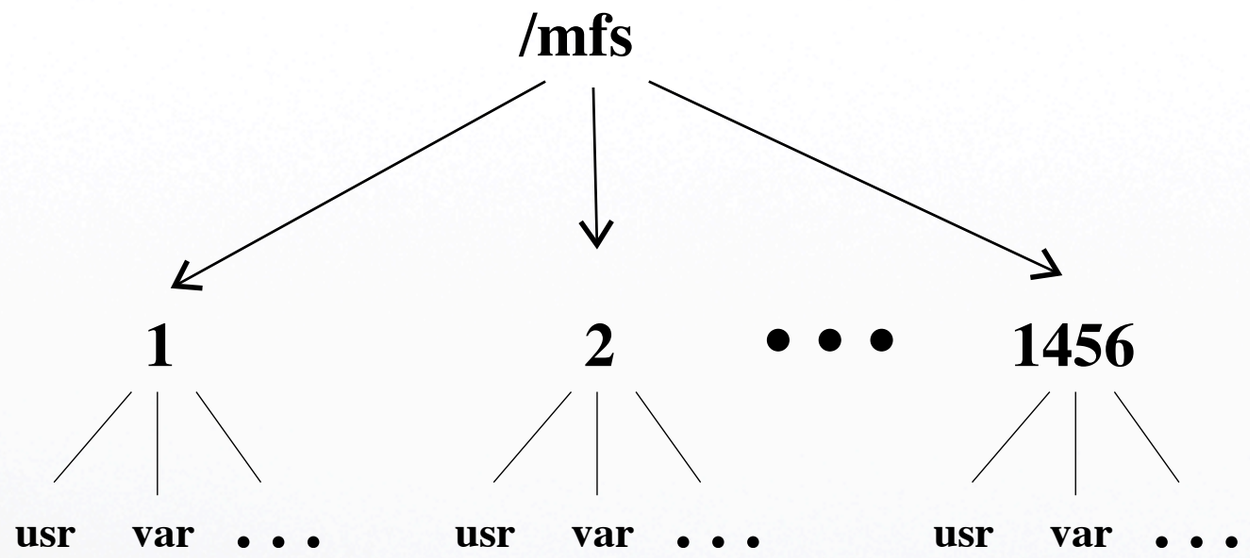
Mosix File System

- OpenMosix è dotato di un proprio file system distribuito, MFS (Mosix File Sytem), conforme alle specifiche DFSA
- Il creatore lo definisce: “uno strato astratto che permette una visione unificata globale”
- Non è un file system che scrive dati su un proprio disco.
- Per lo storage, MFS si basa su qualsiasi file system supportato da Linux

Mosix File System - Caratteristiche

- MFS permette un accesso parallelo ai file distribuendo adeguatamente gli stessi.
- Un processo migra verso i nodi che ospitano i file di cui necessita
- Con altri DFS (quali NFS, AFS, etc.) sono i dati a seguire i processi.
- Con MFS i processi migrano verso i dati

Mosix File System - Caratteristiche

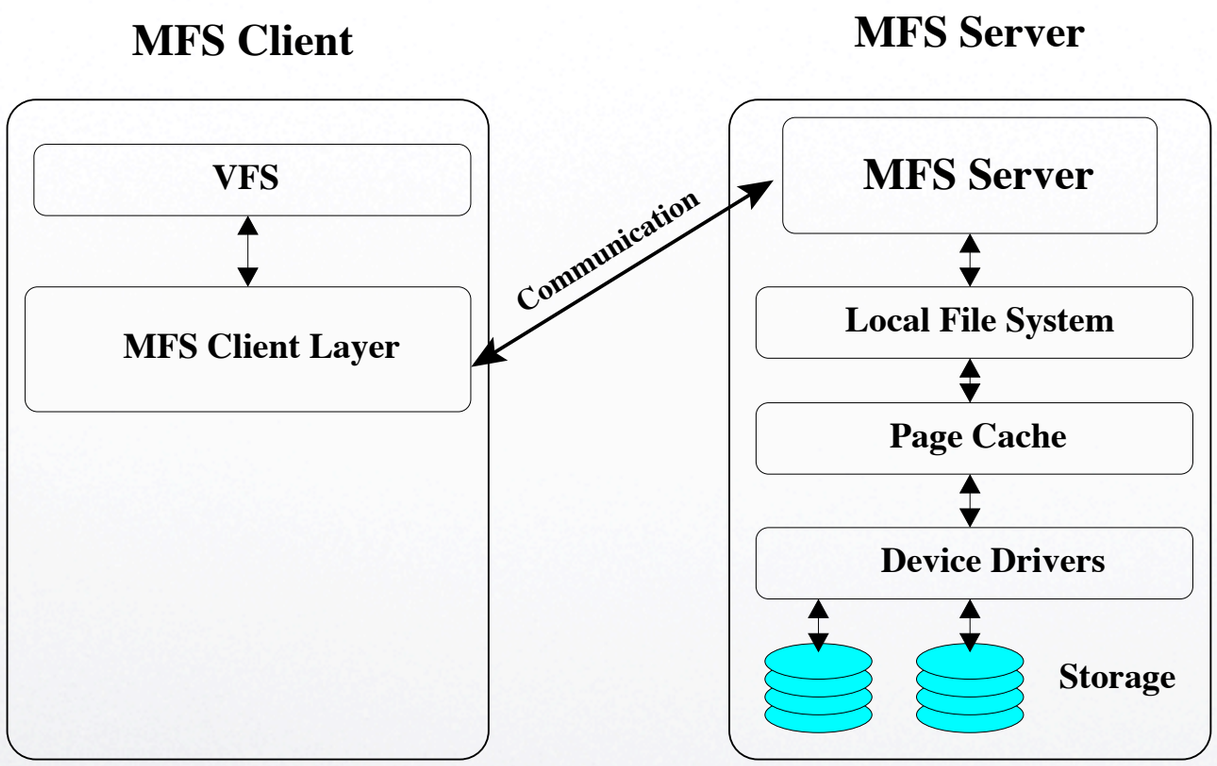


- Con MFS, tutte le direcotry e file regolari sono disponibili da ogni nodo del cluster OpenMosix, attraverso il mountpoint /mfs

- 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

Mosix File System - Caratteristiche



- Al contrario di altri FS, 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

Mosix File System - Conclusioni

- Vantaggi
  - ✳ L'approccio di caching appena descritto fornisce un semplice ma scalabile schema per la consistenza
  - ✳ Le interazioni client-server avvengono a livello di system call; ciò è ottimo per 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 per l'utilizzo con (Open)Mosix e tale è rimasto.
  - ✳ 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

- Distributed File Systems: concepts and examples, E. Levy - A. Silberschatz, 1990
- Sistemi Operativi (Quinta Edizione), Silberschatz - Galvin, 1998
- Andrew: a distributed personal computing enviroment, J.H. Morris - M. Satyanarayanan - M.H. Conner, 1986
- The MOSIX Direct File System Access Method for Supporting Scalable Cluster File Systems. Amar-Barak-Shiloh, 2003
- Removing Bottlnecks in Distributed File Systems: Coda & Intermezzo as examples, P. J. Braam - P.A. Nelson, 2000
- Coda: A Highly Available File System for a Distributed Worksation Environment, M. Satynarayanan, 1995
- The Coda Distributed File System, Linux Journal, June 1998
- Disconnected Operation in the Coda File System, James J. Kistler and M. Satyanarayanan, 1997
- Coda File System Web Site http://www.coda.cs.cmu.edu
- The MOSIX Scalable Cluster File Systems for LINUX, Lior Amar et al., 1999
- Intermezzo Web Site, http://www.inter-mezzo.org