Università degli Studi di Verona

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Informatica

Zettabyte File System

Una breve presentazione

Ricerca aperta

Referente:
Chiar.mo Prof.
Graziano Pravadelli

Presentata da:
Patrick Trentin
id084071

Indice

1 Introduzione **3**

2 Livelli *ZFS* **4**
2.1 Livello Interfaccia 6
2.2 Livello Oggetti Transazionali 6
2.3 Livello Storage Pool 9

3 Astrazione della Memoria **12**
3.1 Capacità di Memorizzazione 12
3.2 Storage Pool 13

4 Gestione dello Spazio **15**
4.1 Dimensione Variabile dei Blocchi 15
4.2 Rappresentazione spazio libero ed occupato 15
4.3 Allocazione blocchi 16
4.4 File Sparsi 17

5 Integrità end-to-end dei dati **17**
5.1 Scrittura transazionale *copy-on-write* 18
5.2 *Checksumming* 19
5.3 *Data Scrubbing* 20
5.4 *Resilvering* 21

6 Ridondanza Dati **22**
6.1 *Ditto Blocks* 22
6.2 *Raid-Z* 24
6.3 *Snapshot* 26
6.4 *Clone* 27
6.5 Deduplicazione 28

7 Cenni conclusivi **28**

1 Introduzione

Il *file system* pone le sue radici storiche agli albori dell'informatica, quando si è reso necessario adattare la manipolazione e conservazione dell'informazione binaria alla limitata disponibilità di risorse ed ai vincoli fisici dei primi dispositivi di memorizzazione di massa. Esso fornisce al sistema operativo un'interfaccia che astrae i metodi di accesso e manipolazione ai dati memorizzati su disco, riducendone la complessità.

Nel corso degli anni si è reagito allo sviluppo in campo tecnologico ed alle nuove esigenze del mercato sviluppando *file system* sempre più raffinati ed affidabili, talvolta anche specializzati per particolari applicazioni. Questa evoluzione ha comportato lo stratificarsi di una serie di assunzioni implementative nei *file system*, molte delle quali al giorno d'oggi non vedono altro presupposto per la loro conservazione se non nella necessità di retro compatibilità del software e nel consolidamento di paradigmi già utilizzati.

È da questa consapevolezza, unita alla necessità di dare nuove risposte a proprietà quali affidabilità, sicurezza, astrazione e semplicità, che è stato sviluppato presso l'allora *Sun Microsystem* il *file system* di nuova generazione *Zettabyte File System*, che da questo momento in poi sarà anche chiamato più semplicemente *ZFS*. La filosofia con la quale è stato concepito si può riassumere in queste poche ma significative parole del capo progettuale Jeff Bonwick: "Abbiamo gettato via 20 anni di tecnologie ormai superate che erano basate su assunzioni al giorno d'oggi non più vere", come riportato in [11].

Dopo solo un anno di lavori, ad ottobre del 2001, è stato ultimato il primo prototipo utente funzionate di *ZFS*, ed un altro ancora è dovuto trascorrere prima che fosse possibile caricarlo dallo spazio utente del kernel. Nel 2005 *ZFS* è stato integrato definitivamente nel codice di Solaris, sistema operativo della SUN. Non si è trattato di un semplice riadattamento del vecchio al nuovo, bensì di una radicale riscrittura che in meno di 80.000 righe di codice ha ridefinito ruoli e contenuti di interi livelli di astrazione, fondendo problematiche di *volume managing* a quelle di più tradizionale competenza del *file system*.

Dalla prima versione rilasciata ne sono susseguite altre le quali, a poco a poco, hanno reso sempre più maturo lo *ZFS*. Alcune delle versioni che hanno introdotto caratteristiche maggiormente significative sono:

- 14 – Supporto per *OpenSolaris* 2009.06, FreeBSD 8.1;
- 15 – Supporto per *Solaris* 10 10/09;
- 17 – Parità tripla per *Raid-Z*;
- 19 – Supporto per *Solaris* 10 09/10;
- 21 – *Deduplication*.

Una sintesi delle caratteristiche di maggiore spicco del *ZFS* segue:

- Capacità di memorizzazione pressoché infinita;
- Scrittura dati transazionale *copy on write* (5.1);
- Gestione dei dischi in *pool*;
- Dimensione variabile degli stripe su disco;
- Garanzia di integrità *end-to-end* dei dati con funzionalità di *self-healing* e *checksumming* (5.2);
- Funzionalità avanzate *backup* e *cloning* dei dati;
- Amministrazione semplice ed intuitiva;
- Portabilità;

L'elenco di peculiarità innovative potrebbe in realtà essere maggiormente esteso, tuttavia molte di esse riguardano dettagli specifici dell'implementazione stessa di *ZFS*, e saranno per questo introdotti solo nei successivi paragrafi di approfondimento. È inoltre da segnalare che il progetto *ZFS* è tuttora aperto, ed alcune funzionalità interessanti come il supporto per gli ambienti cluster attraverso *Lustre* e la possibilità di criptare *on-the-fly* i dati sono tuttora in fase di sviluppo.

2 Livelli *ZFS*

Pur non trattando mai nel dettaglio il codice sorgente di *ZFS*, ritengo possa essere d'aiuto alla comprensione delle restanti sezioni il familiarizzare con l'organizzazione e la stratificazione delle responsabilità del *file system*.

L'individuazione dei componenti fondamentali del sistema, i ruoli ad essi associati e come interagiscono può anche fornire punti di riferimento precisi per coloro che siano intenzionati ad approfondire lo studio dello *Zettabyte File System*. A tale scopo nelle successive pagine saranno, ove possibile, conservati espliciti riferimenti alle componenti o alle funzioni di maggior interesse. Saranno inoltre approfonditi gli aspetti centrali dello *ZFS*, chiarendo ed anticipando così molti dei concetti che ritorneranno nel seguito del documento. Ulteriori approfondimenti delle strutture dati qui citate sono disponibili in [17], [1] e [13].

Come si vede in Figura 1, la struttura organizzativa del codice sorgente è piuttosto semplice, essendo costituita da tre livelli appoggiati uno sull'altro.

I *file system consumers* rappresentano le normali applicazioni, che interagiscono con *ZFS* attraverso l'interfaccia *POSIX*. I *device consumers* non sono nient'altro che *device* virtuali montate come usuali periferiche a blocchi in "*/dev*", supportate tuttavia con lo spazio libero presente nello *storage*

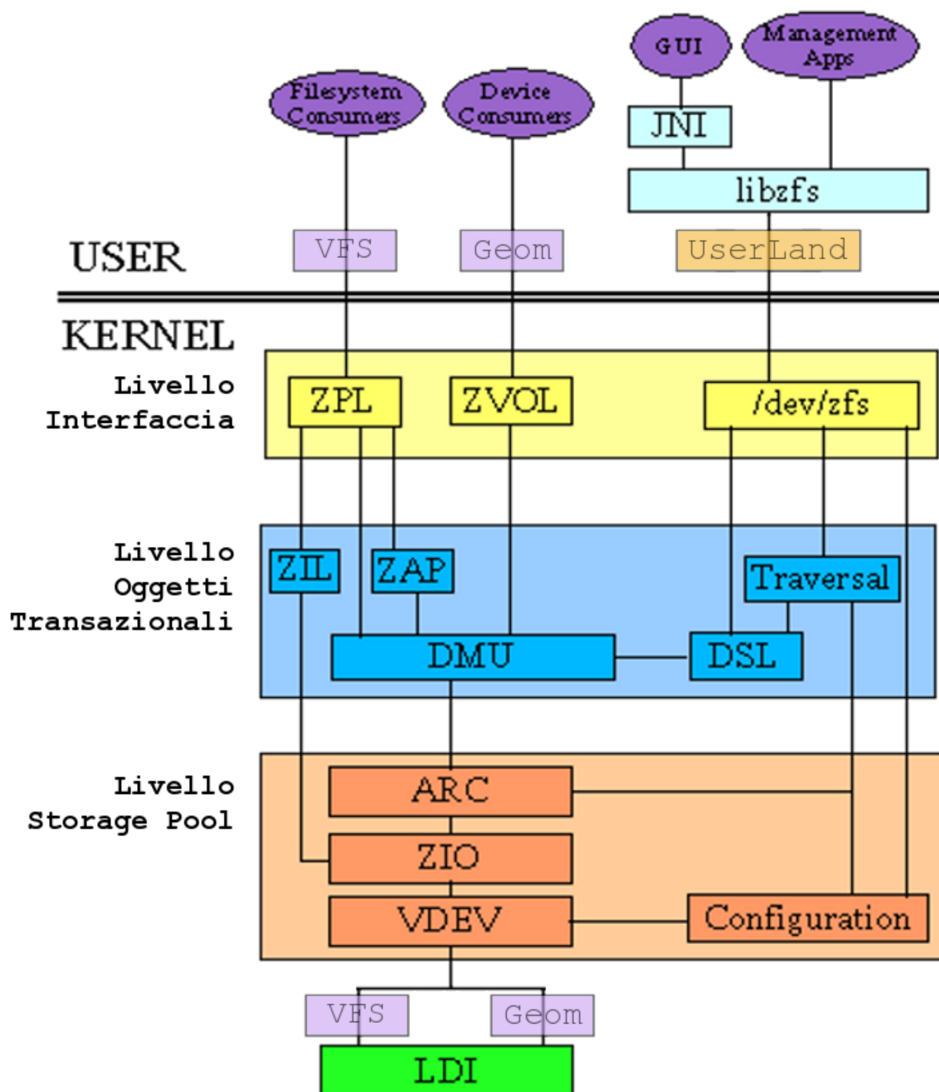

Figura 1: Struttura, livelli e componenti fondamentali dello Zettabyte File System

pool (3.2). Nello *userland* fanno invece parte applicazioni che interagiscono con il *file system* fornite dalla *Sun*. Due importanti comandi per manipolare il *file system* sono "*zpool(8)*" e "*zfs(8)*", di questi se ne può trovare un'estesa documentazione nel manuale online (*man*) di Unix all'interno della sezione indicata nelle parentesi. Il livello *LDI*, o *Layered Driver Interface*, consente l'accesso ai device fisici astraendone le funzioni.

2.1 Livello Interfaccia

Come ne suggerisce il nome, è quello strato applicativo che fornisce tutti i servizi orientati all'utente ed al sistema operativo. Lo *Zettabyte File System* supporta in modo trasparente le usuali applicazioni, attraverso l'implementazione dell'interfaccia *POSIX*.

- *ZPL – ZFS Posix Layer*
  Fornisce una interfaccia *POSIX* di iterazione con *ZFS*, ovvero un'astrazione della struttura di *directory* e *file*, nonché delle operazioni ad esse inerenti, come lettura, scrittura o modifica dei permessi.

- *ZVOL – ZFS Emulated Volumes*
  Forniscono una rappresentazione dei *device* fisici, detti "*zvols*", che stanno dietro ad uno *storage pool*.

- */dev/zfs*
  È il canale di comunicazione tra i tool utente, "*zfs(8)*" e "*zpool(8)*" con il *kernel*. Valida e traduce le richieste gestionali effettuate sul *file system*, demandandole agli appositi componenti.

2.2 Livello Oggetti Transazionali

Questo livello opera in modo indipendente dai dispositivi fisici, in uno spazio di memorizzazione virtuale ed omogeneo. Qui vengono descritti tutti gli oggetti che compongono il *file system*, e le operazioni che possono essere svolte sugli stessi.

- *ZIL – ZFS Intent Log*
  Poiché la maggior parte dei dati non sono scritti su disco immediatamente, per non rallentare le *performance*, è necessario fornire alle applicazioni che fanno uso di "*fsync()*" o "*o_dsync(fd)*", una adeguata garanzia di consistenza del *file system* quando le chiamate ritornano.

  Lo *ZIL* conserva, all'interno di un registro detto *Intent Log*, un elenco delle transazioni sincrone necessario a ripristinare la consistenza del *file system* in caso di *power failure*.

  I meccanismi ed il funzionamento dello *IntentLog*, anche in un contesto reale, possono essere approfonditi con [14].

- *DMU – Data Management Unit*
  È il cuore di *ZFS*, poiché gestisce tutte le transazioni che devono essere svolte su collezioni di oggetti od oggetti singoli. Un oggetto viene definito in una struttura di 512 bytes chiamata *dnode*, contenente campi aggiuntivi sull'oggetto, ed i puntatori a blocchi contenenti i dati veri e propri. Un dnode è in parte cocettualmente simile ai classici *inode*. Come si può tuttavia apprezzare dallo schema in Figura 2, un *dnode* non è specificamente progettato per ospitare soli file, e per questo motivo non presenta i campi tipici di un *inode*. In *ZFS* i blocchi dati possono avere una dimensione variabile compresa tra i 512 bytes ed i 128 Kbytes, per questo motivo un *dnode* dotato di soli tre puntatori a blocchi può gestire file con una dimensione massima di 384 Kbytes.

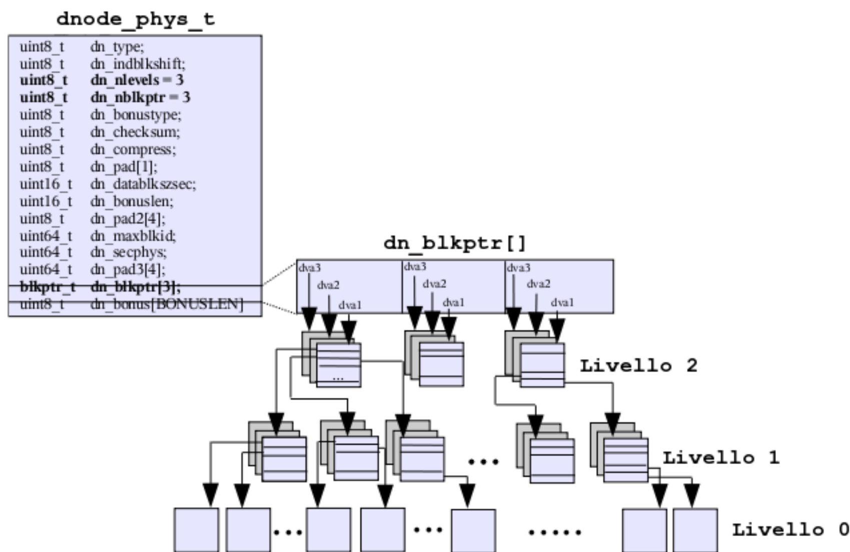

Figura 2: Struttura interna di un dnode rappresentante un file.

Per supportare file di dimensioni più grandi *ZFS* consente l'utilizzo di blocchi di intermediazione in grado di contenere fino a 1024 puntatori a blocchi. Con i 6 livelli di intermediazione consentiti, un file può raggiungere le dimensioni massime di 2^{64} bytes.

Ciascuno di questi puntatori a blocchi, può contenere fino a tre *data virtual address* o *DVA*. Ciascun *DVA* è una coppia offset–vdev che identifica locazioni di memoria con gli stessi dati. Quando lo spazio dello *storage pool* scarseggia e non è più possibile allocare blocchi contigui, *ZFS* fa uso dei *GANG block*, blocchi puntatori frammentabili in diverse porzioni minori.

A ciascun blocco appartenente all'oggetto viene assegnato un id numerico, con il quale è possibile risalire alla sua posizione esatta all'interno dell'albero di blocchi. Un *metadnode* raccoglie in una struttura di tipo array fino a 3072 *dnode*, ciascuno identificato da un proprio *object number* di 64 bit. La Figura 3 chiarisce la gerarchia dei rapporti tra gli oggetti fin qui menzionati, salvo per l'*uberblock* spiegato nella prossima sezione.

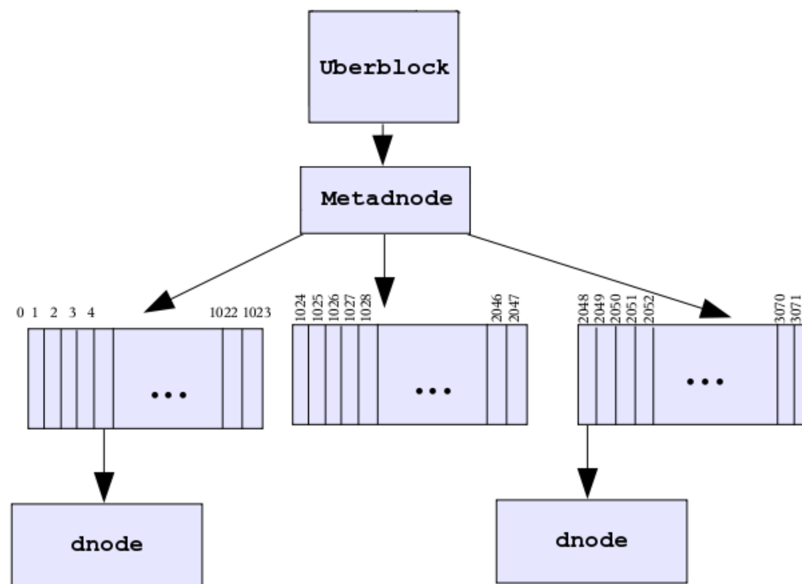

Figura 3: Semplice gerarchia delle strutture dati che compongono il *file system*

- *ZAP – ZFS Attribute Processor*
  Fornisce funzionalità necessarie per la memorizzazione di *file* e *directory*, è utilizzato anche dalla componente *DSL* e come strumento per l'archiviazione delle proprietà di *pool* molto ampi. Fa uso di algoritmi *hash* scalabili per la creazione di associazioni arbitrarie (nome, oggetto) all'interno di una collezione di oggetti. Esistono due algoritmi distinti per la gestione delle *directory*: il *microzap* utilizzato per *entry* di piccole dimensioni ed il *fatzap* utile nel caso di *file* con nomi estesi o *directory* di grandi dimensioni.

- *Traversal*
  Mette a disposizione metodi sicuri ed efficienti per l'attraversamento di un *live pool*. E' utilizzato per l'implementazione delle applicazioni di *scrubbing* (5.3) e *resilvering* (5.4) dello *Zettabyte File System*.

- *DSL – Dataset and Snapshot Layer*
  Aggrega oggetti *DMU* in un *namespace* gerarchico, dotato di proprietà ereditarie e della possibilità di abilitare *quota* o *reservation*. E' anche responsabile della gestione di *snapshots* (6.3), *dataset* e *clones* (6.4).

  Più nel dettaglio, fornisce un meccanismo per descrivere e gestire le relazioni e le proprietà tra gli *objsets*. Un *objset* è un raggruppamento di oggetti tra loro correlati che, nell'ambito *DSL*, viene anche chiamato *dataset*. Ne esistono quattro tipologie fondamentali:

  – *ZFS file system*: memorizza ed organizza oggetti affinché sia possibile accedervi secondo la semantica *POSIX*;
  – *ZFS clone*: è originato da uno snapshot, supporta le stesse operazioni di un *file system*;
  – *ZFS snapshot*: istantanea in sola lettura di un particolare *file system*, *clone* o *volume*;
  – *ZFS volume*: un volume logico esportato da *ZFS* come un device a blocchi.

  Alcune delle più importanti relazioni di interdipendenza tra questi *objsets* sono:

  – Un *clone* impedisce la rimozione dello *snapshot* da cui prende origine;
  – Uno *snapshot* impedisce la rimozione del *objset* da cui è stato creato;
  – *Objsets* genitori di altri *objsets* non possono essere rimossi prima di questi ultimi.

  DSL tiene traccia, oltre che delle interdipendenze tra i *dataset*, anche delle statistiche di consumo spazio e della loro locazione nello *storage pool*.

  Come si vede dalla Figura 4, i *dataset* sono ragruppati gerarchicamente nelle *dataset directory*. Uno soltanto di questi è il *dataset* attivo, ovvero il *live file system*. Gli altri possono essere una lista concatenata di *snapshot*, *cloni* o *dataset* figli/genitori. I dati effettivamente appartenenti ai *dataset* sono gestiti all'interno di una struttura di tipo *DMU*.

2.3 Livello Storage Pool

Il livello *Storage Pool* si occupa di gestire direttamente lo spazio messo a disposizione dai dispositivi fisici, traducendo gli indirizzi virtuali in fisici, a fronte delle richieste di scrittura e lettura provenienti dai livelli superiori.

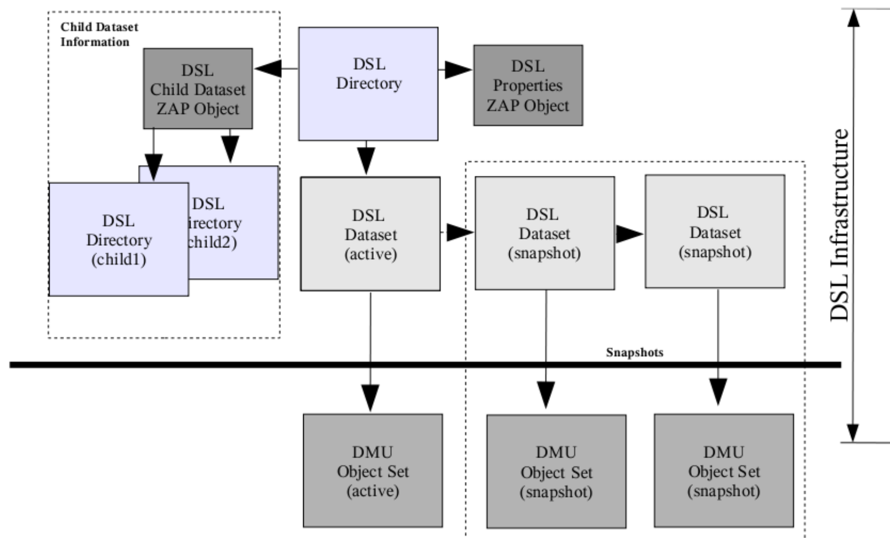

Figura 4: Organizzazione dataset ZFS

- *ARC – Adjustable Replacement Cache*
  Ha il compito di mantenere in *cache file* e blocchi di recente o frequente utilizzo per ridurre i tempi di lettura dati.

- *ZIO – ZFS I/O pipeline*
  È la *pipeline* che raccoglie tutte le richieste di *I/O* sui *device* fisici, traducendo i *device virtual addresses* negli indirizzi logici del disco. Si occupa anche della compressione *on-the-fly* dei dati, della verifica dei *checksum*, e delle operazioni di criptazione e decriptazione dati. Cerca di dare maggiore priorità alle richieste regolari piuttosto che a quelle di sicronizzazione tra i *device*, come nel caso di un *mirror*, per non peggiorare le prestazioni.

- *VDEV – Virtual Devices*
  Fornisce tutti i metodi per accedere ai *vdev* fisici, nonché le funzionalità di *mirroring*, *Raid-Z* (6.2), e di *caching* addizionale sui *device*.

  All'interno di uno *storage pool* sono presenti due tipi di *device* virtuali:

  – *leaf vdev* o *device* virtuali fisici, rappresentano un *device* a blocchi effettivamente scrivibile;
  – *interior vdevs* o *logical virtual devices*, raggruppamenti logici di *leaf vdev*.

  Come si può vedere nella Figura 5, tutti i *vdev* sono organizzati all'interno di un albero la cui radice ha il nome speciale di *root vdev*.

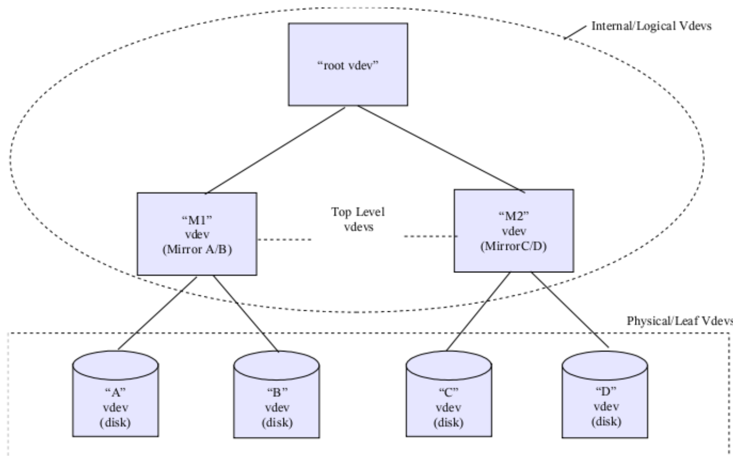

Figura 5: Organizzazione logica dei device di uno storage pool

È importante aprire una piccola finestra che illustri quali sono le strutture di appoggio che consentono a *ZFS* di accedere ai dischi ed al *file system.*

In ogni *vdev* fisico all'interno dello *storage pool* viene memorizzata una struttura di 256 KB chiamata *vdev label.* L'etichetta descrive questo particolare tipo di *device*, e tutte le altre *vdev* che condividono lo stesso *top/level vdev* come antenato. Essa fornisce l'accesso ai contenuti del *pool* e viene utilizzata per verificare l'integrità del *pool* e la disponibilità dei dati in esso salvati.

Esistono 4 copie della *vdev label* su ciascun disco, distribuite metà al principio e metà in fondo al disco, per garantire con buona probabilità che sia sempre disponibile almeno una copia integra dei dati. Questa etichetta si trova in una zona fissata del disco, per questo il suo aggiornamentone richiede la sovrascrittura transazionale in due stadi.

Come riportato in Figura 6, la *vdev label* ospita un array di *uberblock*, quelle strutture dati contenenti tutte le informazioni necessarie per l'accesso ai contenuti del *pool.* Vi è un solo *uberblock* valido in ogni momento, ed è quello con il gruppo di transazione più alto con valore *sha-256* valido. Data l'importanza di questa struttura, il *uberblock* non viene mai sovrascritto, bensì modificato ed inserito in una nuova posizione dell'array secondo una politica *round robin.*

Per consentire la retrocompatibilità con i vecchi metodi di descrizione del *layout* del disco come il *Volume Table Of Contents* (*VTOC*) e le

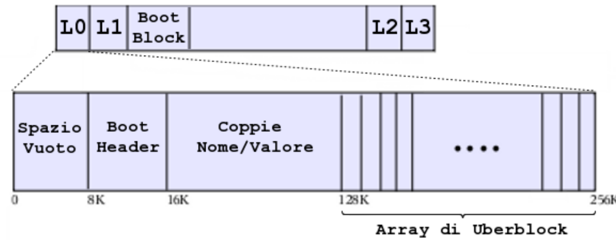

Figura 6: Vdev Label su disco

etichette *Extensible Firmware Interface* (*EFI*), la *vdev label* prevede anche 8 KB di spazio bianco nella parte iniziale.

- *Pool Configuration – Storage Pool Allocator*
  È responsabile per la gestione di tutti i *pool*, delle operazione di aggiunta e sostituzione di *device* fisici e di tutte le attività correlate. Tiene traccia di tutti i file affetti da errori all'interno dei *pool*. Il comando "*zpool(8)*" comunica direttamente con questo livello.

3 Astrazione della Memoria

ZFS utilizza lo spazio di memorizzazione in modo leggermente diverso rispetto agli usuali *file system*, in questa sezione sarà dunque introdotto il concetto di *pool* di dischi, spiegando brevemente come collaborano tra loro i livelli descritti nella precedente sezione e quali proprietà scaturiscono naturalmente da queste scelte implementative. La maggior parte delle strutture dati che permettono l'accesso di dati e metadati del *file system* sono già state approfondite, e non saranno quindi riprese per ulteriori approfondimenti. Lo scopo di questa sezione è infatti quello di dare un assaggio di quelle che sono le caratteristiche che di primo impatto un utente può apprezzare di *ZFS*, senza dover conoscere necessariamente le strutture sottostanti o la teoria.

3.1 Capacità di Memorizzazione

Uno degli aspetti meglio conosciuti di *ZFS* è la sua straordinaria capacità di memorizzazione teorica, peculiarità che ne ha condizionato il nome "*Zettabyte*", ovvero l'equivalente di 10^{21} byte. Sebbene possa sembrare uno degli aspetti esotici maggiormente identificativi ed interessanti di questo *file system*, è una caratteristica che passerà ben presto in secondo piano ad un'analisi più approfondita delle altre virtù di *ZFS*.

Il *ZFS* è il *file system* dei grandi numeri: lavorando a 128 bit consente una capacità di memorizzazione pressoché infinita. Si valuta che, per un

utente che riesca a creare mille file al secondo, occorrerebbero 9 mila anni per saturare le capacità del sistema.

Un *file system* o un singolo file possono raggiungere le dimensioni di 16 exabyte mentre è possibile raccogliere fino a 2^{48} *file* in una *directory* e 2^{64} *file system* in un *zpool*. Uno *zpool* può essere a suo volta composto da ben 2^{64} device di memorizzazione, raggiungendo la capacità massima di 2^{78} bytes, e ne possono essere gestiti contemporaneamente ben 2^{64}.

Occorre però anche evidenziare che le capacità di memorizzazione dello *ZFS* sono sovradimensionate rispetto alle necessità del mercato attuale ed addirittura alla disponibilità di memorie fisiche fino ad oggi prodotte.

3.2 Storage Pool

In passato, quando si fu costretti ad affrontare la crescita delle necessità di spazio rispetto alle capacità dei singoli dischi, si trovò più semplice inserire un livello intermedio tra *file system* e dispositivi fisici che ne astraesse ulteriormente le proprietà. Il *Logical Volume Management* (*LVM*) ha così permesso di superare la limitazione dei *file system* tradizionali, che potevano essere associati soltanto ad un singolo device, estendendone artificiosamente la capacità di memoria totale. È stato anche possibile introdurre tecniche di *mirroring*, *striping* e *raid* per incrementare le prestazioni e l'affidabilità dei dati. Questa scelta ha però portato anche conseguenze negative. Il *LVM*, trasparente al *file system*, è diventato infatti una possibile causa di problemi di sincronizzazione e inconsistenza su disco in caso di *power failure*. Inoltre la tecnica utilizzata non ha consentito di sfruttare al massimo le capacità di *throughput* e di memorizzazione di una batteria di dischi, poiché ogni File System può avere un numero limitato di device fisici dedicati.

Con *ZFS* viene rivoluzionata questa gerarchia, riscrivendola come viene mostrato in Figura 7. Partendo dal più alto livello di astrazione troviamo lo *ZFS Posix Layer*, che rende disponibile l'usuale semantica *POSIX* dei *file system*, il *Data Management Unit*, che definisce le transazioni di accesso agli oggetti e lo *Storage Pool Allocator*, che si occupa dell'allocazione dei blocchi virtuali e funzionalità aggiuntive quali compressione, criptazione o ridondanza dei dati.

Tutti i dischi disponibili vengono raggruppati nello stesso *zpool*, un'entità virtuale all'interno della quale possono essere creati diversi *file system* logicamente indipendenti tra di loro. Ogni *file system*, salvo diversa configurazione dell'amministratore, ha capacità di memorizzazione (virtuale) massima pari a quella dello *zpool*, mentre i suoi contenuti sono automaticamente distribuiti tra tutti i dispositivi disponibili migliorando così affidabilità e *throughput*.

Non è più necessario partizionare i dischi: *ZFS* utilizza infatti l'intero device, supportando anche funzionalità avanzate quali quelle di *boot* e *swapping*. È possibile aggiungere in ogni momento un disco al pool, che verrà automaticamente utilizzato ottimizzando le performance, così come

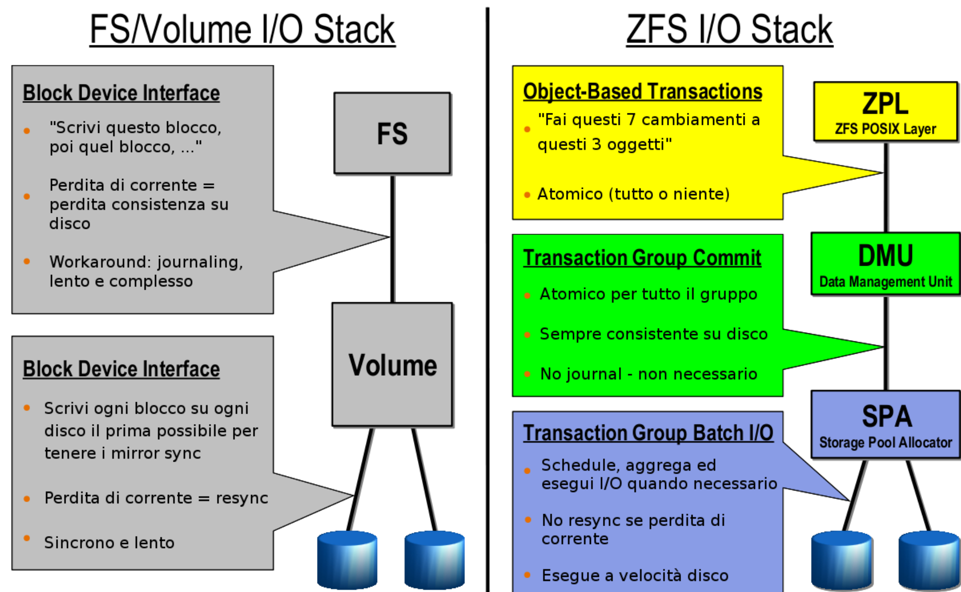

Figura 7: Confronto gestione I/O nel paradigma classico e quello ZFS

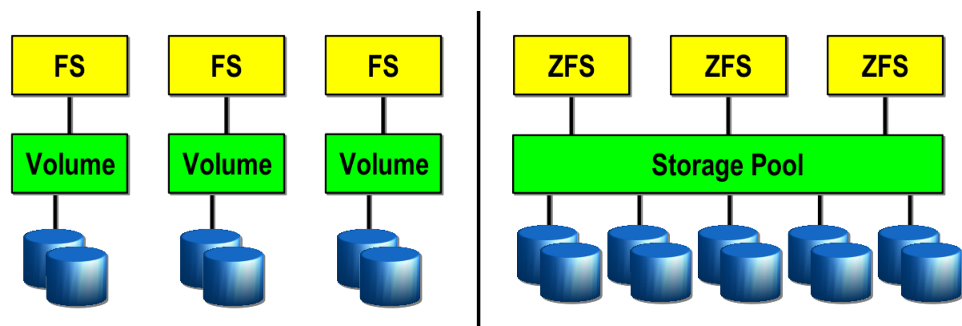

Figura 8: Confronto utilizzo dischi rigidi nel paradigma classico e quello ZFS

rimuoverne senza smontare l'intero *file system* o, in presenza di particolari forme di ridondanza, senza perdere l'accessibilità ai dati. I *pool* di dischi possono essere migrati da un sistema all'altro mantenendone inalterate le caratteristiche, permettendo una semplice gestione dei dati su più sistemi.

È possibile configurare un pool per ottimizzare le capacità di memorizzazione, le performance di *I/O*, la ridondanza, oppure per l'utilizzo di *striping*, *mirroring* o *RAID-Z*. Infine come specificato in [16], unità di memorizzazione *SSD* possono essere aggiunte allo *storage pool* in una soluzione ibrida , e configurate come unità di cache per i dati di accesso frequente al fine di migliorare ulteriormente le prestazioni del sistema.

4 Gestione dello Spazio

Per un *file system* in grado di memorizzare una tale mole di *file* e *directory*, è importante anche comprendere almeno in parte come è possibile organizzare in modo efficiente lo spazio libero e quello occupato, come si può allocare e liberare un blocco in pochi passi senza sovraccaricare la cpu. Ulteriori tecniche come la dimensione variabile dei blocchi e la gestione dei file sparsi saranno qui presentate di contorno, poiché orfane di una sezione propria ma certamente attinenti quando si parla di uso efficiente dello spazio di indirizzamento.

4.1 Dimensione Variabile dei Blocchi

Uno dei problemi affrontati dai *file system* è l'utilizzo ottimale dello spazio allocato ad un file. Quando questo viene sprecato poiché la dimensione dei dati effettivamente necessaria è inferiore a quella allocata, si parla di frammentazione interna.

ZFS affronta questo problema introducendo gli *stripe* di dati a dimensione variabile, che variano da un minimo di 512 bytes ad un massimo di 128 Kbytes. Il concetto di *stripe* in *ZFS* è del tutto indistinguibile da quello di blocco, ed è in effetti quest'ultimo ad essere scritto su disco. La dimensione effettiva del blocco dipende da quella del file da memorizzare e dalla disponibilità di dischi nello *storage pool*: essa viene adattata dinamicamente allo scopo di ottimizzare le operazioni di lettura e la quantità di metadati impiegata.

Per limitare lo spreco di spazio, *ZFS* fornisce anche uno strumento del tutto trasparente alle applicazioni di compressione *on-the-fly* dei dati, di prestazioni ragionevoli.

4.2 Rappresentazione spazio libero ed occupato

In *ZFS* lo spazio di indirizzamento di ciascun disco è suddiviso in qualche centinaio di regioni virtuali chiamate *Metaslab*. A ciascuna regione è asso-

ciato uno *Spacemap*, una struttura dati che permette di risalire allo spazio libero ed occupato del *Metaslab*. Lo *Spacemap* è organizzato come un array di interi che riassume le operazioni di allocazione e de-allocazione effettuate nel tempo.

L'utilizzo degli *Spacemap* comporta numerosi vantaggi:

- Non è necessaria alcuna inizializzazione delle strutture dati. Uno *Spacemap* senza entry indica che non ci sono state allocazioni o liberazioni, e tutto lo spazio è libero.
- La struttura dati è manipolata per *appending* e per questa ragione si rivela scalabile. È sufficiente tenere in memoria soltanto l'ultimo blocco dello *Spacemap* per assicurare buone performance.
- La tipologia di pattern di allocazione e liberazione dei blocchi non influisce sull'efficienza degli *Spacemap*.
- Le prestazioni non peggiorano se il *pool* è quasi interamente allocato.

Una motivazione approssimativa delle scelte che hanno portato all'adozione dello *Spacemap* si può trovare in [9].

4.3 Allocazione blocchi

ZFS gestisce lo spazio secondo una politica di *striping dinamico*, ovvero distribuisce uniformemente i dati tra i dischi rigidi disponibili. In questo modo le operazioni di *I/O* sono quanto più possibilmente parallelizzate, massimizzando l'utilizzo della banda.

Descritto in [8], il processo di allocazione dei blocchi avviene per raffinazioni successive, partendo dal *device* utilizzato, viene dapprima selezionata una regione *Metaslab* e solo in seguito il blocco effettivo. Allo scopo di ottimizzare l'utilizzo delle risorse di memoria, sono state definite *policy* precise per ciascuna fase decisionale:

- *Device*
  Il disco rigido viene selezionato con una politica *round robin*, alternando ogni 512K di dati. Questo valore è stato determinato empiricamente per sfruttare al meglio il *buffering reading* ed agevolare l'*I/O* sequenziale. Nella selezione, i dischi rigidi sottoutilizzati sono lievemente favoriti, in modo da garantire una distribuzione uniforme dei dati. A questa politica sfuggono soltanto i contenuti dello *Intent Log*, i quali debbono rimanere contigui per massimizzare le prestazioni di scrittura.
- *Metaslab*
  Si preferiscono le regioni più esterne del disco, ottimizzando quindi la banda di *I/O* utilizzata ed la *seek latency*.

- *Blocks*
  Lo spazio allocabile di uno *Metaslab* viene ricostruito da *ZFS* accedendo ai dati dello *Spacemap* associato. Per mezzo di un albero *AVL* ordinato per offset si riproducono in memoria le operazioni di allocazione e de-allocazione intervenute, verificando la disponibilità di spazio libero. Fino a quando lo spazio libero all'interno del *pool* non scende sotto una soglia prefissata, si seleziona il primo blocco sufficientemente grande per contenere i dati. In seguito, viene utilizzato un approccio di tipo *best-fit*.

4.4 File Sparsi

Un file sparso è caratterizzato da un numero esiguo di byte significativi rispetto allo spazio totale allocato, ovvero da un certo numero di varie regioni contigue di byte nulli. Jeff Bonwick in [5], afferma che essere in grado di conoscere l'ubicazione di queste aree vuote del file può essere utile per programmi di archiviazione, come *tar*, *cpio* ed *rsync*.

Con *ZFS* si introducono quindi due estensioni ad *lseek(2)*:

- *SEEK_HOLE*
  Restituisce l'offset del prossimo buco di grandezza maggiore o uguale al parametro fornito. L'algoritmo di identificazione delle regioni nulle non permette di trovare esattamente tutte queste zone del file, ma può essere coadiuvante per le applicazioni che effettuino questa ricerca.

- *SEEK_DATA*
  Imposta il puntatore al file alla prossima regione non-zero di grandezza maggiore o uguale al parametro fornito.

Data la struttura ad albero dello *Zettabyte File System*, è possibile ottimizzare le performance di questi algoritmi mantenendo in ogni blocco indiretto un contatore del numero di blocchi dati non-zero puntati. Diventa perciò possibile accedere in tempo logaritmico ad entrambi i tipi di blocchi.

5 Integrità end-to-end dei dati

Si può dire che l'integrità *end-to-end* dei dati sia il vero scopo per il quale lo *Zettabyte File System* è nato, e questa affermazione è senz'altro supportata dall'ampiezza di questa sezione. Una grande quantità di problemi, in parte analizzati in [6], possono affliggere i dati archiviati in un *file system*, e molti altri ne possono sorgere durante il trasferimento dati dalla memoria secondaria a quella primaria. Grazie a *ZFS* si è infatti scoperto che una buona parte delle inconsistenze sui dati che si accumulano nei dischi hanno origine in memoria e non da una corruzione della superficie del disco. Occorre dunque

combinare molteplici tecniche e conoscenze per garantire che i dati consegnati dal *file system* al sistema operativo siano effettivamente consistenti: a partire dalla strategia *copy-on-write*, si validano tutti i dati con *checksum* e si forniscono algoritmi in grado di rilevare e riparare automaticamente gli errori.

5.1 Scrittura transazionale *copy-on-write*

Nei *file system* tradizionali le operazioni di I/O sono gestite a livello di blocchi memorizzati nella cache e poi successivamente consolidate su disco fisico. Nello *ZFS* invece le operazioni si considerano a livello di transazione la quale si considera conclusa soltanto al riscontro dell'avvenuto completamento di ogni passo intermedio, secondo il principio "All or nothing". Le *system call* di scrittura su disco sono trattate con la strategia *copy-on-write*, la quale prevede le seguenti operazioni, riportate anche in Figura 9:

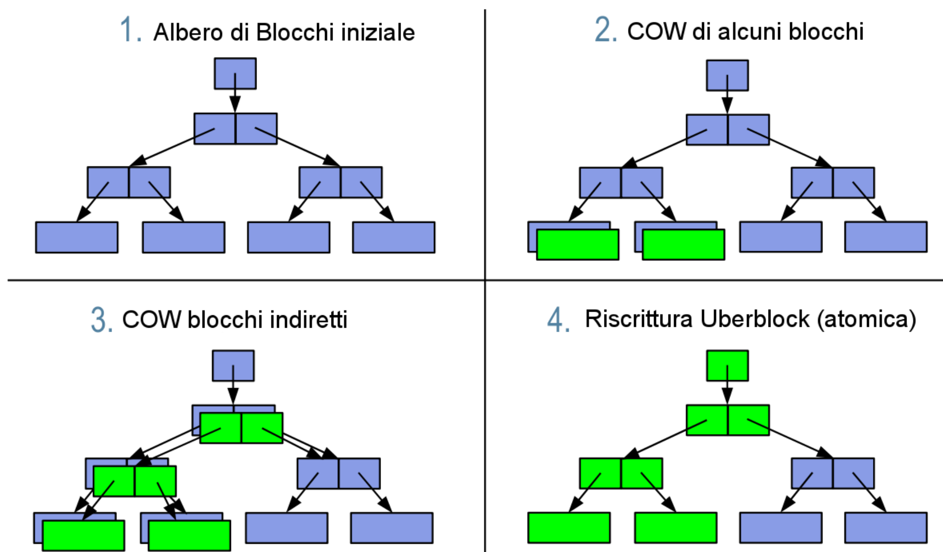

Figura 9: schema delle fasi di una procedura di scrittura copy on write

1. Assegnamento di un nuovo spazio di indirizzamento per i blocchi del *file* da aggiornare o aggiungere e ricalcolo del *checksum* a questi associato;

2. Scrittura su disco dei nuovi blocchi di dati e metadati secondo la configurazione prevista;

3. Aggiornamento dei puntatori del *file system* ai nuovi metadati risalendo l'albero del *file system*;

4. Una operazione atomica di sostituzione dell'*uberblock*, per aggiornare il *file system*.

La *copy-on-write* è una tecnica che in passato, con le ridotte capacità hardware a disposizione, era stata giudicata eccessivamente costosa nella sua implementazione ma che, oggi, si rivela straordinariamente utile al fine di preservare i dati in caso di *power failure* o *fault* nei dati. Essa inoltre consente una semplicissima implementazione di *features* aggiuntive quali gli *Snapshot*. Dopo un'operazione di scrittura infatti, i blocchi dati non aggiornati possono persistere sul *file system* se nel frattempo è stata creata una *Snapshot* del sistema, oppure tornare a fare parte della memoria disponibile.

Per evitare un *overhead* eccessivo del *file system* per operazioni di frequente aggiornamento delle stesse strutture dati, lo *Storage Pool Allocator* del *ZFS* mantiene in memoria una traccia delle transazioni programmate e cerca di aggregarle per ottimizzare performance e sicurezza. Questa struttura dati rimane in memoria fino a quando le operazioni non vengono effettivamente eseguite sul *file system* o, se richiesto da necessità di sincronizzazione, ne viene salvata una copia in un *intent log* stabile all'interno del *pool*. Il *backup* delle transazioni incomplete sul disco, effettuato ad intervalli temporali di 5 s., è utile in caso di *power failure* per consentire il ripristino del *file system* allo stato più recente, mentre non è necessario controllarne la consistenza poiché già garantita dal paradigma transazionale. In [10] viene riportata un'analisi più estesa dell'algoritmo di *scheduling* delle operazioni di scrittura.

5.2 *Checksumming*

Gli algoritmi di *checksum* sono particolari funzioni *hash* applicate a stream di dati binari che estrapolano un'impronta binaria degli stessi. Questa caratterizza in modo univoco i dati e permette, tramite confronti successivi, di rilevare con ottime probabilità l'alterazione o danneggiamento dei dati memorizzati sul disco fisso.

ZFS consente di scegliere tra algoritmi di checksumming a 256 bit semplici e veloci (*Fletcher-2* o *Fletcher-4*) ed algoritmi più lenti ma sicuri come *SHA-256*.

Nei *file system* tradizionali il *checksum* viene solitamente memorizzato assieme ai dati sui quali è stato calcolato, mentre nello *ZFS* esso viene memorizzato nelle strutture dei metadati. I blocchi dello *Storage Pool ZFS* formano un *Merkle Tree* autovalidante, con l'unica eccezione dell'*uberblock*, trattato differentemente. È stato dimostrato che questi alberi forniscono una autenticazione crittografica forte per ogni nodo dell'albero.

Quando si rileva una incongruenza tra dati e *checksum*, poiché il blocco dei metadati è già stato validato, *ZFS* è in grado di stabilire che i dati hanno subito una alterazione e li scarica. Se sono previste forme di ridondanza dei

dati può andare a prelevare la copia integra dei dati e ripristinare quella danneggiata. Questo meccanismo, combinato con gli adeguati metodi di ridondanza dei dati, permette di rilevare ed evitare problemi diffusi quali scritture fantasma e correzione silente dei dati.

5.3 *Data Scrubbing*

Lo *scrubbing*, traducibile in lingua italiana con il termine "spazzolamento", è una tecnica di revisione dei dati contenuti del *file system* svolta allo scopo di individuare e correggere errori presenti in memoria. *ZFS* è in grado di gestire autonomamente le operazioni di *scrubbing*, senza l'intervento dell'amministratore, eseguendo il processo in *background* con una priorità bassa mentre l'intero *file system* è online e montato. Affinché la utility di scrubbing possa svolgere il proprio lavoro è necessaria qualche forma di ridondanza nel *pool* di dischi, sia essa *mirroring*, *raid-z* o *ditto-blocks*.

Poiché il *checksum* degli oggetti del *file system* è salvato insieme ai puntatori dei blocchi, *ZFS* si vede costretto ad attraversare le strutture dei metadati e dati durante lo *scrubbing*. Questo vincolo porta a nuove difficoltà, perché nel *file system* esistono blocchi dati referenziati da più blocchi indiretti, come ad esempio nel caso in cui siano presenti *snapshots* o clones. Limitandosi a seguire la catena dei puntatori ai blocchi si potrebbe finire con il controllare diverse volte gli stessi dati, allungando considerevolmente i già lunghi tempi di lavoro.

L'algoritmo di attraversamento del *file system* è complesso da esaminare, ma se ne possono elencare facilmente alcune delle caratteristiche fondamentali, riportate in [3]:

1. *Sync Scrubbing*
   Lo *scrubber* lavora all'interno di un contesto di *sync*, per prevenire cambiamenti al *file system* durante l'attraversamento delle strutture dati. Ogni volta che la visita di un blocco termina, lo scrubber verifica il tempo impiegato e l'eventuale presenza di altre richieste di *sync* pendenti di maggior priorità. In caso sia necessario, lo scrubber viene messo in pausa (*scrub_pause( )*), e si salvano nello *zbookmark_t* sufficienti informazioni per il ripristino delle operazioni dal punto di interruzione come la locazione del dataset, l'oggetto, il livello ed il blockid.

2. *$ORIGIN*
   L'attraversamento delle strutture dati avviene a partire da *$ORIGIN*, il capostipite di tutti i *dataset* (*file system*, *snapshots* e *clones*), assicurando in questo modo una visita temporalmente ordinata per istante di creazione.

3. *Queue Dataset*
   Il software di *scrub* mantiene una struttura dati, la coda dei *dataset*,

in cui va ad inserire, via via che ne scopre l'esistenza, i *dataset* su cui deve ancora lavorare. Più precisamente, inserisce i puntatori *next snapshot* e *next clone* del *dataset* di cui ha appena terminato la visita. La coda dei *dataset* da visitare è sensibile ad alcune operazioni che possono intervenire durante lo stato di pausa dello scrubber, quali la rimozione di un *dataset* o lo *swapping* al termine di un'operazione di "*zfs recv*", e va per questo motivo contestualmente aggiornata.

4. Confronto Data
   Come primo raffinamento, lo *scrubber* visita soltanto i blocchi che hanno una data di nascita successiva al momento di creazione dello *snapshot* precedentemente visitato. Se il blocco scartato dall'attraversamento è un nodo intermedio dell'albero, è possibile evitare di attraversare i dati da esso puntati poiché anch'essi avranno una data di nascita minore o uguale a quella del capostipite.

5.4 *Resilvering*

Il *resilvering*, conosciuto anche come risincronizzazione o ricostruzione ed analizzato in [7], è un processo di riparazione di un disco danneggiato attraverso l'utilizzo dei contenuti appartenenti a device integri. Come il software di *scrubbing*, l'operazione di *resilvering* presuppone l'esistenza di una qualche forma di ridondanza dei dati memorizzati.

Anche nei *volume manager* tradizionali sono previste tecniche di risincronizzazione, che diventano utili quando è necessario sostituire un disco danneggiato o qualora si subisca una temporanea interruzione dell'alimentazione elettrica. Ad esempio, in un *mirror* è sufficiente copiare byte per byte i contenuti di un disco nell'altro, mentre in un *raid-5* è necessario leggere tutti i dati distribuiti lungo lo stripe, comprese le informazioni di parità, ed effettuarne lo xor per ricostruire i dati mancanti.

Il problema principale dell'approccio tradizionale è quello di non verificare la consistenza dei dati oggetto di *resilvering* prima o durante la copia del disco. Ciò può causare una propagazione silente degli errori nei dati, anche in *device* appena installati e verosimilmente integri. Inoltre poiché i blocchi dati vengono copiati in ordine indipendente dalla struttura del *file system*, fino al termine dell'operazione di copia non ne è garantito il completamento. Cosa accade infatti se l'ultimo blocco ad essere copiato è la *directory* di *root*?

Per spiegare l'approccio utilizzato da *ZFS* è comodo coglierne la similitudine con il comando "*cp -r*" applicato ad una gerarchia dati di un *file system*, e definirne le peculiarità essenziali:

- *Top-Down Resilvering*
  Il *resilvering* segue l'albero di dati dello *storage pool*: parte dal blocco di *root*, e copia i blocchi in ordine discendente per importanza. Infatti

il danneggiamento di un blocco intermedio impedisce l'accesso a tutto il suo sotto-albero di blocchi dati.

- *End-to-End Integrity*
  Durante l'attraversamento dei metadati è possibile verificare e ripristinare la consistenza dei blocchi appartenenti al sotto-albero utilizzando *checksum* indipendenti.

- *Live-Blocks-Only Copy*
  Vengono copiati soltanto i blocchi che fanno parte della struttura del *file system*, questo permette di risparmiare grandi quantità di tempo quando un unità disco è relativamente vuota rispetto alle proprie capacità.

- *Transactional Pruning*
  È possibile effettuare il *resilvering* dei soli dati del *file system* che hanno subito una modifica nella finestra temporale in cui un disco è rimasto inacessibile, per esempio, a causa di una perdita di corrente. Ciò avviene confrontando il *transaction group* di nascita dei blocchi e permette anche di sfrondare facilmente l'analisi di intere sezioni del *file system* ricordando che il *transaction group* degli elementi di un sotto albero è sempre minore o uguale a quello del blocco radice.

6 Ridondanza Dati

L'affidabilità *end-to-end* di cui si è parlato nella sezione precedente, in molti casi, non può però avere luogo se non si introducono tecniche di ridondanza dei dati di qualche tipo. *ZFS* consente di utilizzare il classico *mirroring*, ma anche il *raid-Z* ed i *Ditto Blocks*, più sicuri e meno esigenti di memoria. *Snapshot* e *Clone* presentano invece la propria utilità per la possibilità di accedere a versioni precedenti o alternative del *file system* in uso, dalle quali recuperare *file* accidentalmente rimossi o modificati.

6.1 *Ditto Blocks*

Lo *Zettabyte File System* si presenta, in astratto, come un albero di blocchi nel quale solo i nodi foglia contengono dati, mentre i rimanenti costituiscono blocchi intermedi di puntatori e metadati. Come spiegato per esteso in [15], più del 98% dei blocchi in uno *storage pool* è composto da blocchi dati, perciò tipicamente da un errore nel disco provoca un *input/output error* (*EIO*) rendendo impossibile accedere ai dati danneggiati. Un errore in un blocco intermedio costituisce un caso di maggior gravità poiché nega anche l'accesso a tutto il sotto albero di dati integri, come si può vedere in Figura 10.

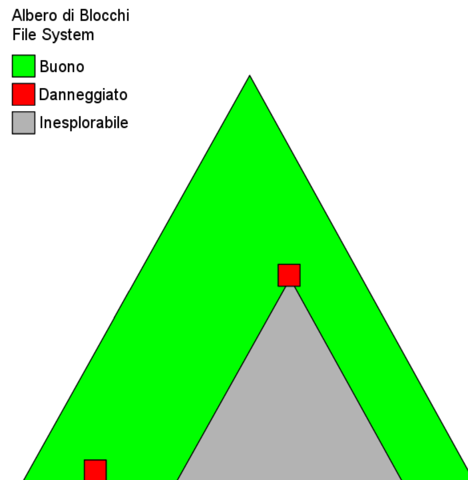

Figura 10: Effetti di un errore nel file system

Per salvaguardare i dati che fanno a capo dei puntatori ai blocchi, detti anche *Disk Virtual Addresses*, *ZFS* consente la memorizzazione di fino a tre copie del blocco intermedio, utilizzando i *ditto blocks*.

- Politica Ridondante
  La politica di ridondanza dei *ditto blocks* è definita in modo tale che i blocchi più vicini alla *directory* di *root* del *file system*, di maggiore importanza, possiedano più repliche. Sono dunque previsti per *default* tre *DVA* per i dati di accesso globale dello *storage pool*, due *DVA* per i metadati del *file system* ed un *DVA* per i dati utente. Questa strategia fa si che, se la probabilità di un errore in un blocco è P, la probabilità che l'intero *storage pool* venga compromesso sia di appena P^3. Inoltre, poiché meno del 2% dello spazio è occupato da *DVA*, la ridondanza dei dati ha un impatto minimale sulle prestazioni *I/O* e sulla disponibilità di spazio.

- Politica Diffusiva
  I *DVA* ridondanti vengono diffusi il più possibile lungo lo spazio di memorizzazione, per salvaguardarne l'integrità. Quando lo *storage pool* è composto da un singolo disco fisso si distanziano i *DVA* di almeno 1/8 del disco, mentre in caso siano presenti più dispositivi i *ditto blocks* vengono memorizzati in quelli adiacenti alla copia originale.

È possibile ridefinire le *policy* di ridondanza dei blocchi a livello di singolo *file system*, adottando così politiche personalizzate rispetto alla tipologia di file contenuti e alla disponibilità eventuale di ulteriori copie di *backup offline*.

In Figura 11, tratta da [12], si può apprezzare un esempio di *storage pool* in cui sono presenti due *file system* con diverse politiche di ridondanza, e come i dati sono stati distribuiti tra i due *device* disponibili.

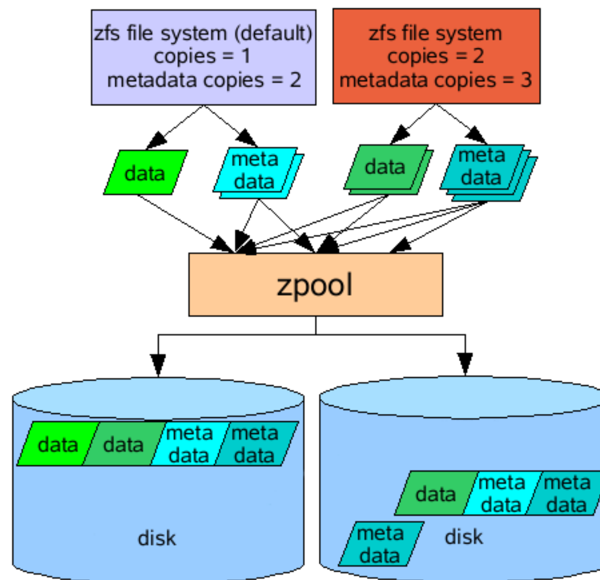

Figura 11: distribuzione di dati e metadati tra i dischi del live pool

6.2 *Raid-Z*

Come spiegato in [4], il *raid-Z* trae la sua fonte di ispirazione dal *raid-5*, da cui ha ereditato alcune caratteristiche importanti.

Il *raid-5* è una configurazione *raid* particolare che distribuisce le informazioni di parità, calcolate su stripe di dimensioni pari ad un blocco, tra tutti i dispositivi fisici disponibili. In questo modo *raid-5* è in grado di ottenere buone prestazioni in lettura e, tramite la ridondanza dei dati, di resistere alla perdita di un *device* del gruppo. Il *raid-5*, come altre configurazioni *raid* tradizionali, è affetto da una grave vulnerabilità definita in gergo *write-hole*.

Il *write-hole* rappresenta un problema di corruzione silente dei dati cui sono affette le configurazioni *raid* tradizionali che utilizzino la parità. Esso può avvenire se durante un'operazione di scrittura si verifica un guasto di sistema, o una perdita di corrente, prima che le informazioni di parità aggiornate siano a loro volta riportate su disco. Se questa inconsistenza non viene eliminata prima che l'utilizzo del *Cyclic Redundancy Check* (*CRC*) non riscontri un *data error*, il sistema cercherà di ricostruire i dati utilizzando le informazioni di parità inconsistenti.

Per eliminare il problema del *write-hole*, con il *raid-z* si è deciso di utilizzare una lunghezza variabile per gli *stripe*, cosicché ogni operazione di scrittura si concludesse con uno *stripe* completo. Questo, unito alla garanzia di scritture transazionali ed alla strategia *copy-on-write*, elimina di fatto ogni possibilità che si verifichi un *write-hole*. Infatti questi ultimi garantiscono, completando atomicamente tutte le modifiche alle strutture del *file system*, la consistenza dei dati e metadati prima e dopo l'operazione di scrittura. L'accorpamento dei dati e metadati in uno stripe completo, soggetto ad un'unica fase di scrittura, garantisce la consistenza impedendo che l'uno o l'altro non siano aggiornati contemporaneamente.

LBA \ Disk	A	B	C	D	E
0	P_0	D_0	D_2	D_4	D_6
1	P_1	D_1	D_3	D_5	D_7
2	P_0	D_0	D_1	D_2	P_0
3	D_0	D_1	D_2	P_0	D_0
4	P_0	D_0	D_4	D_8	D_{11}
5	P_1	D_1	D_5	D_9	D_{12}
6	P_2	D_2	D_6	D_{10}	D_{13}
7	P_3	D_3	D_7	P_0	D_0
8	D_1	D_2	D_3	X	P_0
9	D_0	D_1	X	P_0	D_0
10	D_3	D_6	D_9	P_1	D_1
11	D_4	D_7	D_{10}	P_2	D_2
12	D_5	D_8	•	•	•

Figura 12: Esempio di distribuzione degli stripe dei blocchi tra i dischi di un pool

ZFS raggiunge i dati attraversando completamente i metadati e ciò può diminuire le prestazioni quando lo *storage pool* è molto vicino ad essere pieno, mentre rappresenta un *trade-off* accettabile in situazioni normali. D'altro canto l'utilizzo di stripe di dimensione dinamica evita la lettura di altri dati su disco per il calcolo della parità, e consente anche di migliorare le prestazioni di ricostruzione dei dati rispetto alle configurazioni *raid* tradizionali. Il più grande vantaggio dello *ZFS* in *raid-z* risiede proprio nella distribuzione delle informazioni di *checksum* lungo i metadati. Il *file system* può accorgersi immediatamente della presenza di dati danneggiati e procedere al recupero dei dati ridondanti, ripristinando quelli originali. *Zettabyte File System* provvede anche ad avvisare l'Amministratore, attraverso il *software Solaris FMA*, del problema avvenuto.

Sono disponibili anche configurazioni *raid-z2* e *raid-z3* che resistono alla perdita di 2 e 3 *device*, rispetto alla tolleranza singola di un *raid-z* semplice.

6.3 *Snapshot*

Lo *snapshot* del *file system* non è altro che un'istantanea dello stato e dei contenuti di *file* e *directory* in un determinato momento, accessibile in sola lettura. Uno *snapshot* viene creato in *ZFS* con un'operazione atomica, cioè in tempo costante, e non occupa alcuno spazio aggiuntivo fino a successive modifiche del *file system* corrente. Dal punto di vista del *file system* uno *snapshot* è visto come un *dataset* di tipo *dls_dataset_phys_t*. Per supportarne le funzioni, *ZFS* mantiene due strutture dati correlate:

- Una data di nascita associata ad ogni blocco, che indica il valore del contatore incrementato ad ogni *sync* di un gruppo di transazione la quale si conclude con l'aggiornamento dell'*uberblock*.

- Una *dead list*, ovvero un array di puntatori ai blocchi referenziati nello *snapshot* precedente e che non sono più puntati dal *file system* o *snapshot* attuale.

È interessante esaminare anche solo in maniera superficiale come il team di sviluppo abbia affrontato le difficoltà legate alla creazione e rimozione all'interno dello *Zettabyte File System*, come affrontato in [2].

- Creazione di uno *snapshot* (*dsl_dataset_snapshot_sync( )*)
  Tutto ciò che è necessario fare per prendere uno *snapshot* è salvare l'*uberblock*, la radice dell'albero dei blocchi dell'intero *zpool*, prima che venga sovrascritto da una nuova transazione. I puntatori ai blocchi rimangono ancora validi, così come i dati, che non vengono perduti grazie la strategia *copy-on-write*. In realtà, per consentire l'indipendenza tra *snapshot* di diversi *file system*, ciò che viene salvato non è l'*uberblock* bensì la radice dei sotto alberi che rappresentano i *file system*. Gli *snapshot* sono conservati utilizzando una lista doppiamente concatenata ed ordinata per data di nascita. All'amministratore è data la facoltà di sceglierne il nome.

  Lo spazio occupato da uno *snapshot* è inizialmente nullo, e cresce man mano che vengono effettuate modifiche sul *file system* corrente. Prendere uno *snapshot* è un'operazione effettuata nell'arco di una transazione atomica, ed impiega quindi tempo costante. [$O(c)$]

- Rimozione di un blocco (*dsl_dataset_block_born( )*)
  Quando il *file system* viene modificato, in seguito ad un'operazione di scrittura o rimozione di alcuni dati, la *DMU* determina se è possibile liberarne lo spazio occupato reclamando lo spazio per successivi usi. Ciò è consentito soltanto se il blocco non è referenziato da precedenti *snapshots* (o *clones*), ed è possibile ricavare quest'informazione dal semplice confronto della data di nascita del blocco (ossia il numero del gruppo di transazione in cui è stato creato) con quella dello *snapshot*

più recente. Se il blocco è nato prima, allora è referenziato dallo *snapshot* e non può essere liberato, nel caso contrario non è più referenziato e lo si può disporre per futuri usi. Nel caso in cui il blocco non possa essere liberato, esso viene aggiunto alla *dead list* del *file system*.

- Rimozione di uno *snapshot* (*dsl_dataset_destroy_sync( )*)
  Il tempo impiegato per rimuovere uno *snapshot* è proporzionale al numero di blocchi puntati che devono essere inseriti nella *free list*.

  Un blocco, per essere libero, deve possedere contemporaneamente i seguenti requisiti:

  1. deve essere nato dopo il precedente *snapshot*;
  2. deve essere nato prima dello *snapshot* in rimozione;
  3. deve essere morto dopo la creazione dello *snapshot* in rimozione;
  4. deve essere morto prima della creazione dello *snapshot* successivo.

  La verifica di queste condizioni è semplificata dal fatto che tutti i blocchi appartenenti alla *dead list* dello *snapshot* successivo (o del *file system* corrente) possiedono implicitamente i requisiti 2, 3 e 4. Ciò che rimane da verificare è la condizione 1 la quale, similarmente alla rimozione dei blocchi, può essere sbrigata confrontando la data di nascita dei blocchi.

Con il sistema operativo *Solaris* è stato rilasciato anche un *tool* grafico, il *TimeSlider*, che permette di gestire in modo automatico le funzionalità offerte dagli *snapshots*. La utility fornisce una vista del *file system*, o di una cartella, correlata ad uno *slider* che può essere portato avanti o indietro lungo una linea temporale di *snapshot* precedenti, per visionarne i contenuti. Ripristinare un file che è stato cancellato o effettuare il ripristino ad uno stato precedente del *file system* diventa una semplice questione di trascinamento dello *slider* temporale.

Uno *snapshot* può essere inviato ad un dispositivo diverso, ad esempio uno *storage pool* esterno o ad un nastro magnetico, ottenendo in questo modo un backup dei dati. L'invio dei dati può anche essere di tipo incrementale, minimizzando lo spazio occupato.

6.4 *Clone*

Un *clone* è uno *snapshot* di un *file system* cui è stata consentita la modifica indipendente, ed è quindi capace di evoluzione propria. Inizialmente un *clone* non occupa spazio aggiuntivo rispetto alla struttura dati di cui è la replica, mentre cresce di dimensioni man mano che viene modificato.

Il *clone* ha una relazione di dipendenza con lo *snapshot* di origine e ne impedisce per questo motivo la rimozione. Da esso, tuttavia, non eredita tutte le proprietà caratteristiche del *dataset*.

Un *clone* supporta le normali operazioni di un *file system*: può essere creato, distrutto o promosso a sostituto del *file system* corrente.

6.5 Deduplicazione

La deduplicazione di un *file system* è utile per eliminare copie duplicate dei dati, riducendo il consumo di spazio e la quantità di accessi in scrittura effettuati sui dischi.

In *ZFS* la deduplicazione viene fatta sui blocchi, utilizzando una funzione crittograficamente forte come *SHA-256* che garantisce una probabilità di collisione di soli 2^{-256}, l'equivalente di 10^{-77}. I *checksum* vengono mappati all'interno di una tabella hash (*DDT*), la quale memorizza anche la locazione dei dati su disco per un accesso rapido ed il *reference count* associato ai blocchi.

La deduplicazione è integrata all'interno della pipeline di *I/O*, e viene effettuata *on-the-fly* da *ZFS* come parte di un *transaction group*. Quando si verifica uno *hit* nella *DDT*, lo *Zettabyte File System* non deve fare altro che incrementare il *reference count*, risparmiando tempo e spazio.

La deduplicazione di *ZFS* si intende *general purpose*, ed è fortemente legata alla tecnologia con cui opera. Assume infatti la presenza di un sistema operativo altamente *multithread*, come *Solaris*, ed un ambiente *hardware* in cui la *cpu* lavora molto più velocemente dell'*I/O*.

7 Cenni conclusivi

In questo breve documento ho cercato di presentare in modo compatto le soluzioni adottate dagli sviluppatori della *Sun Corporation* per affrontare le lacune principali dei *file system* della generazione corrente. È stato dato spazio ad una visione di insieme che senza scendere eccessivamente nei dettagli tecnici fosse in grado di chiarire ciascuna delle caratteristiche principali dello *Zettabyte File System*, senza pretesa di esaustività.

Il futuro di *ZFS* è incerto, la *Sun Microsystem* è infatti stata recentemente assorbita dalla concorrente *Oracle*, e non è ancora chiaro se questa sia intenzionata a portarne avanti lo sviluppo e la promozione. Questo perché la *Oracle* sta già da tempo lavorando su un *file system* con licenza GPL molto simile nei concetti e nelle funzionalità allo *ZFS*, il *BRTFS*.

Le soluzioni ed innovazioni concettuali apportate dallo *Zettabyte File System* hanno comunque aperto la strada per tutti gli altri progetti, dimostrando l'efficacia di questi nuovi approcci alla gestione dell'informazione, ed è probabile che per questo non saranno dimenticati ma ripresi e migliorati, per i nuovi mercati.

Glossario

albero AVL particolare tipo di albero bilanciato caratterizzato dalla presenza aggiuntiva su ciascun nodo di un coefficiente di bilanciamento. Supporta operazioni di ricerca ed eliminazione dei nodi in $log(n)$ ed un inserimento (ammortizzato) in $O(1)$.

appending dall'inglese, apporre, aggiungere.

background una modalità di esecuzione di un processo che non richiede l'interazione con l'utente, che può quindi essere soggetta ad apposite politiche di assegnazione delle risorse.

best-fit politica di gestione della memoria allocabile che seleziona la più piccola zona di spazio libero che soddisfa una richiesta.

buffering reading indica, in fase di lettura dati, l'utilizzo di una memoria tampone nella quale immagazzinare più dati di quelli immediatamente richiesti. In caso di operazioni di lettura sequenziali, questa tecnica permette di risparmiare tempo evitando numerosi accessi alla periferica di memorizzazione.

cache memoria ad alta velocità che consente un recupero molto veloce delle informazioni in essa contenute.

checksum indica una sequenza di bit che identifica, possibilmente in modo univoco, uno stream di dati. Algoritmi per la produzione di *checksum* possiedono capacità e costo computazionale anche molto diverso, e sono quindi adatti per molteplici scopi. Una descrizione di come viene utilizzato il *checksumming* in *ZFS* si può trovare in 5.2.

clone brevemente, *snapshot* di cui è consentita la modifica.

cloning dei dati generalmente, indica la copia *bit* a *bit* di un particolare *stream* dati od unità di memorizzazione.

copy-on-write indica una strategia di memorizzazione, per la quale ad ogni nuova versione di un file (o blocco) si sceglie di allocare un nuovo spazio di indirizzamento, strutture dati e puntatori anziché sovrascrivere la posizione di memoria originaria. Ciò evita, in caso di interruzione dell'alimentazione in fase di scrittura, di avere in memoria file (o blocchi) parzialmente aggiornati ed inconsistenti. Approfondimenti in 5.1.

Cyclic Redundancy Check un semplice algoritmo di *checksum*, particolarmente adatto in situazioni di trasmissione con rumore di fondo.

data error segnale d'errore generato da un controllo *CRC*, indica che i dati analizzati sono con buona probabilità corrotti.

dataset struttura generica di una collezione di oggetti, definita ed utilizzata da *DSL*. Sono esempi di *dataset* un *file system*, un *clone*, uno *snapshot* ed un *volume*.

dead list è un array di puntatori ai blocchi referenziati in snapshot precedenti ma non più presenti nel dataset di riferimento.

device dall'inglese, periferica o dispositivo.

directory contenitore virtuale di *file* e *directory*, pensata per dare una vista gerarchica ad albero dell'organizzazione dei contenuti memorizzati su disco.

end-to-end dall'inglese, traducibile con "da un capo all'altro", indica la fornitura di un servizio completo.

Extensible Firmware Interface è una specifica che definisce l'interfaccia tra sistema operativo e *firmware* sottostante. Stabilisce la compatibilità con la *GUID Partition Table* per la gestione delle partizioni dei dischi rigidi, solitamente posizionata nel *Master Boot Record* (*MBR*).

fault difetto, errore nel funzionamento.

file contenitore virtuale dell'informazione memorizzata su disco.

file system	software che astrae e gestisce l'organizzazione fisica dei dati sui dispositivi di memorizzazione, fornendo quindi al sistema operativo un'interfaccia di accesso e manipolazione.
free list	è un array di puntatori a blocchi liberi, disponibili per essere allocati ed ospitare dati.
GANG block	blocco contenente puntatori ad altri blocchi di dimensioni inferiori, utilizzato in situazioni di frammentazione nelle quali lo spazio disponibile residuo non è utilizzabile in blocchi contingui.
general purpose	dall'inglese, multiuso, di scopo generico.
hash	funzione non iniettiva in grado di mappare una stringa di *bit* qualsiasi in un'altra di lunghezza predeterminata.
hit	dall'inglese, un centro nel bersaglio, successo o risultato.
inode	struttura dati del *file system* che archivia le informazioni di un *file*, come ad esempio il proprietario, data ed ora di creazione ed i blocchi in cui si trovano i dati in esso contenuti.
Intent Log	letteralmente "registro degli intenti", un'apposita struttura dati che conserva una lista delle modifiche da apportare al *file system* che ancora devono essere completate. Presente sotto diversi nomi e forme in molti *file system* tradizionali, ha lo scopo di permettere il ripristino di uno stato consistente dei dati in caso di *power failure* del sistema.
live file system	in *ZFS* indica il *file system* principale o in uso, sul quale vengono svolte le normali operazioni di *input/output*.
live pool	in *ZFS* fa riferimento allo storage pool attivo ed in uso in un certo istante.

Logical Volume Management *software* che fornisce una astrazione dei dispositivi di memorizzazione fisici, riunendo questi ultimi in uno spazio di indirizzamento condiviso, sul quale è poi possibile stabilire partizioni logiche con un opportuno *file system*. Questo sistema fornisce diversi vantaggi, tra i quali la possibilità di ridimensionare le partizioni attive, cioè montate ed in uso.

Lustre *file system* scalabile utilizzato in ambienti cluster, cioè in presenza di reti telematiche di computer.

man il manuale in linea degli ambienti Unix, consultabile a linea di comando facendo seguire a "*man*" la parola chiave di interesse.

metaslab in *ZFS* indica una delle regioni virtuali nelle quali è scomposto lo spazio allocabile di un disco, viene utilizzato per semplificare la gestione dello spazio libero ed occupato su disco.

mirroring tecnica di ridondanza dati che consiste nel mantenere copie multiple di ciascun dato su dischi rigidi differenti.

multithread indica un sistema in cui è fornito supporto *hardware* all'esecuzione di più *thread* da parte di un processore.

namespace o spazio di nomi, indica un raggruppamento logico di identificatori unici (i nomi) all'interno di uno specifico ambiente virtuale, all'interno del quale ne è confinato l'utilizzo.

on-the-fly dall'inglese traducibile con "in volo", utilizzato per evidenziare l'esecuzione immediata di una particolare attività, in questo contesto essenzialmente prima che i dati vengano memorizzati su disco.

Oracle società di *software* storicamente diventata famosa in campo database, ha recentemente acquisito la *Sun Microsystems*.

overhead consumo aggiuntivo di risorse di calcolo, rispetto a quelle strettamente necessarie per svolgere una determinata attività.

policy dall'inglese, indica una politica di gestione.

power failure perdita del flusso di corrente che alimenta l'*hardware* di una macchina.

queue una struttura dati che realizza una coda.

quota in *ZFS* fa riferimento ad una limitazione del massimo spazio allocabile in un *file system*. Ulteriori riferimenti in 6.4.

raid riunisce in una singola unità logica due o più dischi rigidi, distribuendo uniformemente i dati mediante apposite soluzioni *hardware* o *software*. Ne sono state definite tipologie diverse, in generale si può dire che i vantaggi principali di un *raid* sono: il miglioramento nelle prestazioni di accesso e trasferimento dei dati, la possibilità di aumentare la capienza del sistema e la maggiore affidabilità del sistema di memorizzazione. Maggiori dettagli sulle strutture *raid* supportate in *ZFS* in 6.2.

reservation in *ZFS* indica lo spazio minimo di memoria allocabile garantito a ciascun *file system*.

resilvering processo di riparazione di un disco danneggiato attraverso l'utilizzo dei contenuti appartenenti a device integri. Approfondimenti in 5.4.

root nei sistemi *Unix/Linux* indica la *directory* radice dell'intero *file system* e l'account amministratore del sistema.

round robin distribuzione circolare ed omogenea di un certo numero di elementi.

scrubbing tecnica di revisione dei dati contenuti del *file system* svolta allo scopo di individuare e correggere errori presenti in memoria. Ulteriori riferimenti in 5.3.

self-healing in *ZFS* fa riferimento alla capacità del *file system* di riconoscere e porre rimedio ad eventuali alterazioni involontarie nei dati memorizzati in un disco. Questa capacità è normalmente subordinata alla presenza di meccanismi di ridondanza, *CRC* o *checksum*.

sha-256 algoritmo di *hash* crittografico che lavora con parole di 32 bit, produce un'impronta di 256 bit del file originale.

snapshot istantanea dello stato e dei contenuti di *file* e *directory* in un determinato momento, accessibile in sola lettura. Una spiegazione più completa si può trovare in 6.3.

Solaris sistema operativo Unix sviluppato dalla *Sun Microsystems*, utilizzato con piattaforme SPARC, x86 e x64.

Solaris FMA o *Fault Management Architecture*, è un software fornito nelle distribuzioni *Solaris* che fornisce servizi di analisi e ripristino in caso di errori di calcolo o nella memoria.

spacemap struttura dati che conserva le informazioni relative allo spazio libero ed a quello occupato all'interno di un *metaslab*.

storage pool in *ZFS* indica la composizione logica di più supporti di memorizzazione in un'unica entità virtuale. Per approfondimenti consultare 3.2.

stripe indica una frazione intera di un *file* composta a sua volta da un numero di *strip*, unità fondamentali di memorizzazione come settori, blocchi o tracce. Ciascuno *strip* di uno *stripe* appartiene ad un disco diverso, permettendo l'accesso in parallelo ai dati e quindi ad un *throughput* proporzionalmente migliore.

striping dinamico politica di gestione spazio che distribuisce uniformemente tra più dischi i dati di uno stesso file, in forma di *stripe*. In *ZFS* Con l'attributo dinamico si intende la caratteristica aggiuntiva di adattamento delle dimensioni dello stripe alla quantità di dati da scrivere su disco.

Sun Microsystems compagnia fondata nel 1982, occupata nella commercializzazione di sistemi *hardware* (es.: *SPARC Enterprise Server*) e *software* (es. s.o. *Solaris*) ed impegnata anche sul fronte dell'*open source* (es.: *Java*). Dal 2000, con l'esplosione della bolla finanziaria di *internet*, la compagnia ha visto un declino economico dalla quale non si è mai ripresa integralmente. Infine, il 27 febbraio 2010, la *Sun Microsystems* è stata acquisita dalla *Oracle*.

swapping supporto avanzato di gestione della memoria, consente di spostare dalla memoria principale a quella secondaria delle pagine di un processo a *runtime*.

sync indica la scrittura immediata (sincrona) dei dati sul disco, alternativa alla modalità *async* (asincrona) nella quale le operazioni di scrittura vengono prima memorizzate in un *buffer* tampone e poi eseguite ad intervalli regolari, ottimizzando cos l'uso dei dischi.

system call chiamata a sistema, permette ad un programma utente di recuperare un servizio fornito dal *kernel* del sistema operativo.

throughput indica la capacità di trasmissione di un canale di comunicazione effettivamente utilizzata.

TimeSlider applicativo fornito con *Solaris*, offre un sistema automatico di *backup* dei *file*. È in grado di effettuare automaticamente *snapshot* ad intervalli regolari nel tempo, e di fornire informazioni dettagliate sui consumi di spazio all'interno del *pool*.

trade-off comporta la scelta compromissoria tra più alternative reciprocamente dipendenti.

uberblock blocco capostipite del *file system* contenente tutte le informazioni necessarie per l'accesso ai dati del pool.

Volume Table Of Contents struttura dati posizionata all'inizio di un disco rigido che fornisce un modo per localizzare i *dataset* che risiedono all'interno del volume, utilizzata in ambienti *mainframe IBM*.

Riferimenti bibliografici

[1] Zfs source tour. http://hub.opensolaris.org/bin/view/Community+Group+zfs/source, Ottobre 2009. [Online; visitato Febbraio 2010].

[2] M. Ahrens. Is it magic? http://blogs.sun.com/ahrens/entry/is_it_magic, Novembre 2005. [Online; visitato Febbraio 2010].

[3] M. Ahrens. New scrub code. http://blogs.sun.com/ahrens/entry/new_scrub_code, Dicembre 2008. [Online; visitato Febbraio 2010].

[4] J. Bonwick. Raid-z. http://blogs.sun.com/bonwick/en_US/entry/raid_z, Novembre 2005. [Online; visitato Febbraio 2010].

[5] J. Bonwick. Seek_hole and seek_data for sparse files. http://blogs.sun.com/bonwick/en_US/entry/seek_hole_and_seek_data, Dicembre 2005. [Online; visitato Febbraio 2010].

[6] J. Bonwick. Zfs end-to-end data integrity. http://blogs.sun.com/bonwick/en_US/entry/zfs_end_to_end_data, Dicembre 2005. [Online; visitato Febbraio 2010].

[7] J. Bonwick. Smokin' mirrors. http://blogs.sun.com/bonwick/en_US/entry/smokin_mirrors, Maggio 2006. [Online; visitato Febbraio 2010].

[8] J. Bonwick. Zfs block allocation. http://blogs.sun.com/bonwick/en_US/entry/zfs_block_allocation, Novembre 2006. [Online; visitato Febbraio 2010].

[9] J. Bonwick. Space maps. http://blogs.sun.com/bonwick/en_US/entry/space_maps, Settembre 2007. [Online; visitato Febbraio 2010].

[10] R. Bourbonnais. The new zfs write throttle. http://blogs.sun.com/roch/entry/the_new_zfs_write_throttle, Maggio 2008. [Online; visitato Febbraio 2010].

[11] R. Corsanici C. Cigliola. Sun microsystems zfs, il filesystem con caratteristiche uniche. *Linux&C.*, 65:47–52, 2008. Tratto da riquadro: "Intervista a: Danilo Poccia".

[12] R. Elling. Zfs, copies and data protection. http://blogs.sun.com/relling/entry/zfs_copies_and_data_protection, Maggio 2007. [Online; visitato Febbraio 2010].

[13] Jeremy. Bsdcan 2008: Zfs internals. http://kerneltrap.org/FreeBSD/BSDCan_2008_ZFS_Internals, Maggio 2008. [Online; visitato Febbraio 2010].

[14] M. Leal. Zfs internals (part#6). http://www.eall.com.br/blog/?p=701, Dicembre 2008. [Online; visitato Febbraio 2010].

[15] B. Moore. Ditto blocks - the amazing tape repellent. http://blogs.sun.com/bill/entry/ditto_blocks_the_amazing_tape, Maggio 2006. [Online; visitato Febbraio 2010].

[16] Oracle. *Oracle SolarisZFS Administration Guide*, Settembre 2010. Online; visitato Febbraio 2010.

[17] Sun Microsystem. *ZFS On-Disk Specification draft*, 2006. Online; visitato Febbraio 2010.