Suporte Transaccional para o Sistema de Ficheiros Btrfs

João Eduardo Luís, João M. Lourenço, and Paulo A. Lopes*

CITI — Departamento de Informática,
Universidade Nova de Lisboa, Portugal
joao.luis@fct.unl.pt {Joao.Lourenco,pal}@di.fct.unl.pt

Resumo Em caso de falha abrupta de um sistema, é imperativo garantir a consistência do Sistema de Ficheiros (SF). Actualmente existem várias soluções que visam garantir que tanto os dados como os metadados do SF se encontram num estado consistente, mas que não garantem a consistência dos dados do ponto de vista das aplicações. Por exemplo, aplicações que pretendam alterar vários ficheiros de configuração terão de encontrar mecanismos para garantir que, ou todos os ficheiros são devidamente alterados, ou nenhum o é, evitando assim que numa situação de falha os ficheiros fiquem "aplicacionalmente inconsistentes". Do ponto de vista da aplicação, pode não ser simples implementar este comportamento sobre um SF típico; e pode também não ser razoável utilizar um Sistema de Gestão de Bases de Dados (SGBD), que oferece propriedades ACID. Neste artigo propomos, testamos e avaliamos uma integração das propriedades ACID num SF. Partindo do suporte para snapshots do sistema de ficheiros Btrfs, oferece-se uma semântica transaccional às aplicações que operam sobre volumes (sub-árvores) do SF, sem comprometer a sua semântica POSIX.

Palavras-Chave: Transacções; Sistema de Ficheiros; Linux; Kernel

1 Introdução

Os Sistemas de Ficheiros (SF) modernos incluem vários mecanismos que visam garantir a sua própria consistência em caso de falha abrupta do sistema. Ainda que seja essencial para uma aplicação que o SF se mantenha consistente, os requisitos de consistência de uma aplicação vão para além da consistência do SF. Por exemplo, enquanto que do ponto de vista do SF uma operação se poderá reduzir à escrita ou leitura de um bloco, para a aplicação uma operação poderá ser um vasto conjunto de leituras e escritas, sobre um ou mais ficheiros diferentes.

* Este trabalho foi parcialmente suportado pela Sun Microsystems ao abrigo do *Sun Worldwide Marketing Loaner Agreement #11497*, pelo Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL) e bolsa PEst-OE/EEI/UI0527/2011, e pela Fundação para a Ciência e Tecnologia (FCT/MCTES) no âmbito dos projectos PTDC/EIA/74325/2006, PTDC/EIA-EIA/108963/2008, e PTDC/EIA-EIA/113613/2009.

Ou seja, os SF nada garantem às aplicações cujo correcto funcionamento depende de uma visão lógica e consistente dos seus dados.

Esta consistência poderá depender, por exemplo, da garantia de que todo um conjunto de ficheiros de configuração são correctamente modificados em disco, ou que acessos em concorrência são efectuados como se a aplicação fosse a única a usar o SF, simplificando em muito a gestão de concorrência que tradicionalmente é obtida através de *file locking*.

A semântica transaccional associada às propriedades ACID dos SGBDs poderia suprir estas necessidades: *Atomicidade*, garantindo que todas as operações são aplicadas com sucesso ou nenhuma o é; *Consistência*, levando o SF a transitar sempre entre dois estados consistentes; *Isolamento*, permitindo à aplicação aceder ao SF como se fosse a única a aceder-lhe; e *Durabilidade*, garantindo que os dados ficam preservados de forma permanente. Contudo, para que seja possível usar um SGBD, para guardar neste os dados, a aplicação necessita de i) um SGBD disponível no sistema; e ii) de considerar o seu *overhead* (ainda que versões "leves" [2] o possam atenuar). Será contudo necessário ponderar as vantagens obtidas face ao esforço necessário na alteração da aplicação em tudo o que se refere a Entradas/Saídas, correspondendo a uma mudança substancial de paradigma.

É também possível oferecer um modelo transaccional às aplicações recorrendo a um SO que o suporte [7], mas tal acarreta o uso de uma versão modificada do núcleo do SO. Alternativamente, a semântica transaccional pode ser assegurada directamente pelo SF. Existem várias implementações com este propósito, ainda que seriamente limitadas: umas apenas suportam um modelo *single-writer/multiple-readers* e com níveis de isolamento mais relaxados [4]; outras recorrem internamente a SGBDs [5, 6]; outras ainda poderão apresentar *overheads* inaceitáveis para as aplicações [9].

Este artigo propõe o TxBtrfs, um Sistema de Ficheiros Transaccional que garante às aplicações as propriedades ACID nas operações sobre o SF através de uma interface explícita disponibilizada ao programador, implementado como um módulo para o Linux Kernel. Relativamente a trabalhos similares desenvolvidos no passado, a nossa proposta, que usa o Btrfs [1] como base para a implementação, apresenta as seguintes contribuições: usa versões standard do núcleo do Linux; não necessita de recorrer a software externo; e apresenta *overheads* pequenos para a grande generalidade das operações.

O restante artigo estará organizado da seguinte forma: na Secção 2 apresentamos a nossa implementação, discutindo os resultados preliminares na Secção 3; concluímos na Secção 4 com a apresentação do trabalho para o futuro mais imediato.

2 *Transactional Btrfs* (TxBtrfs)

O Btrfs suporta nativamente os conceitos de *subvolume* e *snapshot*. Ambos devem ser vistos como sub-árvores independentes, pertencentes ao Sistema de Ficheiros, que apresentam todas as características de um SF singular (p.e., política

de atribuição de números de *inodes* própria, representação por árvores independentes da do SF original). Contudo, um subvolume é criado vazio, enquanto que um snapshot é uma cópia lógica de um subvolume existente. Suportada nesses conceitos, a nossa implementação introduz o conceito de *Subvolume Transaccional* (TxSv), no qual todos os acessos serão transaccionais, e que poderá coexistir com subvolumes e snapshots nativos. Cada transacção iniciada no SF irá na altura ser mapeada num snapshot do TxSv vigente. Cada processo pertencente à transacção será mapeado, de forma transparente, no seu snapshot aquando do acesso ao TxSv. Qualquer tentativa de acesso ao snapshot por parte de processos alheios à transacção será negada. Sempre que uma transacção fizer um *commit* com sucesso, as suas alterações serão retratadas no TxSv; transacções iniciadas posteriormente tomarão o TxSv vigente como estado inicial do SF, enquanto que qualquer transacção que pretenda fazer *commit* terá de se validar com o estado do TxSv actual com o intuito de detectar conflitos antes de poder ser terminada com sucesso.

Com o intuito de manter a compatibilidade com a norma POSIX, decidimos que a forma mais correcta de programar com transacções seria usar *ioctl's* para especificar cada operação, seja esta de início, final ou cancelamento de uma transacção. Dado não alterarmos o comportamento das chamadas ao sistema suportadas pelo Btrfs, a nossa implementação satisfaz todos os testes do pacote *pjd-fstests* [8] também satisfeitos pela versão 0.19 do Btrfs, i.e., todos excepto um teste relativo ao `truncate(2)`.

2.1 Processos e Transacções

Quando um processo inicia uma transacção sobre o SF ambos ficam associados a um novo snapshot. No entanto admite-se que uma transacção e o seu respectivo snapshot possam estar associados a múltiplos processos. Tal situação é apenas possível em três casos, todos envolvendo um processo e seus filhos, e visa manter as "expectativas habituais" da semântica operacional de partilha de recursos.

O primeiro caso é aquele no qual um processo inicia uma transacção e posteriormente invoca um `fork(2)`: o processo filho resultante será executado no âmbito da transacção do pai (se aceder ao subvolume transaccional). Desta forma, um processo filho nunca entrará em conflito (na perspectiva transaccional) com o seu progenitor. Se, por outro lado, o `fork(2)` ocorrer antes do inicio da transacção, o processo filho resultante já não estará abrangido pela transacção do pai, e irá dispôr de uma transacção própria.

No segundo caso, o processo filho, criado após o pai ter iniciado uma transacção, inicia ele mesmo uma nova transacção. Como o filho partilha a transacção do pai, estabelece-se que qualquer transacção por si iniciada (ou por qualquer um dos seus descendentes) irá ser considerada como *aninhada* segundo o modelo *flat nesting*, i.e., todas as operações feitas na transacção aninhada são parte integrante da transacção do progenitor.

Considere-se finalmente o terceiro caso, no qual o processo filho inicia uma nova transacção no contexto da transacção criada pelo seu pai, mas em que a transacção aninhada termina somente após o fim da transacção envolvente. Para

eliminar os problemas potencialmente criados por esta situação, consideramos que a transacção apenas será realmente finalizada quando o número de *commits* invocados for igual ao número de *starts*.

2.2 Commit de Transacções e Reconciliação de snapshots

Durante a execução de cada transacção, todas as operações por esta efectuadas no SF serão registadas com o intuito de a validar e reconciliar com o TxSv aquando do *commit*. Actualmente contemplamos a validação das escritas sobre um mesmo ficheiro por parte de transacções distintas, através da comparação das leituras da transacção a finalizar com as escritas de todas as transacções que tenham sido aplicadas ao TxSv após o início dessa transacção. O processo de reconciliação actual cinge-se à cópia de intervalos de blocos entre os ficheiros validados, o que poderá ser moroso para ficheiros de grande dimensão; uma alternativa em estudo passa por aplicar deduplicação aos ficheiros partilhados entre as transacções.

Contudo, o processo de validação e de reconciliação terá de contemplar também todas as operações sobre metadados do SF (incluindo criação, remoção e renomeação de ficheiros e directórios). Estes temas ainda se encontram em estudo, com o intuito de aferir quais as relações entre operações, de forma a que seja feita uma correcta validação das transacções.

3 Validação

Como a versão actual do TxBtrfs ainda não detecta eficazmente os conflitos entre transacções, o processo de validação consiste, de momento, em aferir o impacto que toda a transparência no mapeamento dos processos nos snapshots e o registo completo das operações têm no desempenho do SF. Os testes foram executados num sistema com Debian Linux 6.0, com um processador dual-core i5 650 a 3.20 GHz, 4GB de RAM e um disco rígido SATA Samsung P80 SD.

Assim, executou-se o benchmark *IOzone* [3], com os testes automáticos para leitura e escrita, sobre uma versão pura do Btrfs (v0.19) e sobre o TxBtrfs. No caso do TxBtrfs, todo o benchmark foi executado no contexto de uma única transacção. Cada teste do benchmark consiste no acesso a ficheiros com tamanhos entre 64KB e 4GB usando registos cuja dimensão varia de 4KB a 16MB. Os resultados obtidos para a largura de banda (LB) vêm expressos em KB/s. O limite superior de 4GB para o ficheiro foi por nós imposto, uma vez que por omissão o *IOzone* só cria ficheiros até 512MB. Este limite superior, igual à dimensão da RAM da máquina, visa garantir que o teste executa realmente operações de I/O que exercitam o disco físico, não se ficando apenas pela *page cache* do Linux.

De forma a sintetizar os resultados e calcular o *overhead* imposto, fizemos a média dos valores obtidos em execuções sucessivas (forçando a invalidação da *page cache* entre execuções), tanto para o Btrfs como para o TxBtrfs. Em ambos os testes verificou-se que tanto o Btrfs como o TxBtrfs obtêm resultados na ordem dos GB/s quando os ficheiros são de dimensão inferior a 2GB. Tal fenómeno

é facilmente explicado pela capacidade do Linux Kernel em manter esses ficheiros na *page cache*. Para ficheiros de 4GB, a situação inverte-se completamente, obtendo-se valores entre os 40 MB/s e os 70 MB/s. Verificámos também que os acessos a um ficheiro de 2GB ora exibem largura de banda na ordem dos GB/s (indiciando que este coube na *page cache*) ora exibem largura de banda em tudo similares à obtida para ficheiros de 4GB. Para ficheiros de 2GB só foram considerados os "runs" que exibiram largura de banda consentâneas com acessos ao disco, i.e., da ordem dos MB/s.

Na Tabela 1 apresentamos o *overhead* do TxBtrfs face à versão do Btrfs do qual partiu o desenvolvimento, tanto para leituras como para escritas. Seguindo o que foi discutido anteriormente, ao invés de enunciar os tamanhos dos ficheiros testados, optámos por usar a notação *Cache* e *Disco*, uma vez que as diferenças de resultados (para cada uma das categorias) só são relevantes de acordo com o tamanho do registo usado. *Overheads* negativos indicam que o TxBtrfs obteve um melhor desempenho, o que só se pode justificar por variações no estado do sistema, pois o *overhead* do TxBtrfs deveria ser sempre positivo.

Tabela 1: Overhead do TxBtrfs face ao Btrfs, em percentagem.

	Overhead de Leituras (%)						Overhead de Escritas (%)					
Tamanho	Cache			Disco								
do Registo	Mín	Méd	Máx	Mín	Méd	Máx	Mín	Méd	Máx	Mín	Méd	Máx
4K-64K	-12	27	67	N/D	N/D	N/D	-5	10.62	34	N/D	N/D	N/D
128K-2M	-19	4.7	39	0	0.2	1	-14	2.85	12	1	1.6	3
4M-16M	-16	-1	16	0	0.6	1	-17	-1.38	17	1	1.6	3

Poder-se-á verificar que os resultados para as leituras e escritas, ainda que diferentes, são da mesma ordem de grandeza. Quando o acesso faz um maior uso da *cache*, o TxBtrfs apresenta um *overhead* consideravelmente superior para registos de tamanho reduzido (4KB a 64KB), *overhead* este que decresce à medida que o tamanho do registo aumenta. Tal não se verifica, contudo, para acessos que façam efectivamente maior uso do disco, sendo o *overhead* para qualquer tamanho de registo muito próximo de zero. Este comportamento é consistente com a nossa implementação, que é toda ela suportada em memória. Quando o I/O é também muito dependente da memória (e do CPU), como aquando do uso de registos de pequena dimensão, a nossa implementação apresenta um comportamento pior. Contudo, à medida que a dimensão do registo aumenta ou que o acesso aos dados transita da memória para o disco, o *overhead* introduzido pelo acréscimo de uso do CPU é atenuado, tornando-se virtualmente negligenciável.

4 Conclusões

O nosso objectivo final é o suporte da semântica transaccional no Btrfs, sendo para isso necessário i) registar as operações realizadas no contexto de transacções, ii) implementar um algoritmo de validação dos registos e detecção de conflitos entre transacções, e posteriormente iii) proceder a reconciliações de snapshots de transacções concorrentes compatíveis. Até ao presente desenvolvemos o módulo de registo das operações, mantendo a conformidade total com a norma POSIX, e avaliámos a robustez e desempenho do protótipo através de um conjunto de testes baseados no *benchmark* IOZone. Como ficou patente na Secção 3, o uso adicional de CPU e memória pelo TxBtrfs demonstram algum impacto nalguns resultados, face à implementação original do Btrfs, sendo no entanto negligenciável quando o I/O se efectua sobre disco físico.

Num futuro próximo, esperamos conseguir reduzir este impacto em ficheiros de pequena dimensão, evitando a introdução de mais *overhead* aquando da finalização das transacções. Iremos também implementar um algoritmo de validação das transacções sobre o Sistema de Ficheiros, e proceder à reconciliação dos snapshots. A validação deste trabalho deverá incluir, para além da avaliação do desempenho, um conjunto de testes visando a verificação da conformidade do TxBtrfs com as propriedades ACID.

Referências

1. Btrfs wiki at kernel.org. `https://btrfs.wiki.kernel.org/`, last modified on 11 January 2011.
2. D. Richard Hipp. SQLite software library. `http://sqlite.org/`, last checked on 25 January 2011.
3. IOzone. IOzone File System Benchmark Home Page. `http://www.iozone.org/`, last checked on 17 June 2011.
4. Microsoft. Basic TxF Concepts. `http://msdn.microsoft.com/en-us/library/dd979526%28v=VS.85%29.aspx`, last checked on 14 June 2011.
5. Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system, 2002.
6. Michael A. Olson and Michael A. The design and implementation of the inversion file system, 1993.
7. Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 161–176, New York, NY, USA, 2009. ACM.
8. Tuxera. Posix test suite. `http://www.tuxera.com/community/posix-test-suite/`, last checked on 17 June 2011.
9. Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *Trans. Storage*, 3(2):4, 2007.