Transferência de Ficheiros Baseada em *Chunks* com Eliminação de Redundância

Diogo Paulo
GSD - INESC-ID Lisboa
Instituto Superior Técnico
Email: dpaulo@gsd.inesc-id.pt

João Barreto
GSD - INESC-ID Lisboa
Instituto Superior Técnico
Email: joao.barreto@inesc-id.pt

Paulo Ferreira
GSD - INESC-ID Lisboa
Instituto Superior Técnico
Email: paulo.ferreira@inesc-id.pt

***Resumo*—Neste artigo propomos uma nova técnica para a transferência de ficheiros através da qual conseguimos obter resultados comparáveis com as soluções actuais: eliminamos a redundância dos dados que enviamos; temos um protocolo que introduz poucos *overheads* na transferência, quer em termos de meta-dados como no número de rondas utilizadas pelo protocolo. A nossa solução combina uma nova técnica de eliminação de redundância baseada no conhecimento mútuo de informação que cada um dos intervenientes na transferência tem. O uso de *workloads* realistas nos nossos testes mostram que a nossa solução tem um bom desempenho quando comparada com soluções alternativas.**

I. Introdução

Actualmente, abundam aplicações de alto-desempenho que têm de guardar, recuperar, manipular e transferir grandes quantidades de dados. Os conjuntos de dados com que cada uma destas aplicações trabalha têm vindo a aumentar, estando agora muitas das vezes na ordem dos TeraBytes. Por exemplo, o TeraShake [1] é uma aplicação para a *Grid* que faz simulações de actividades sísmicas. Cada execução do TeraShake gera cerca de 50 TB de dados. Geólogos de todo o mundo podem estar interessados nos dados que o TeraShake produz. Não é desejável, por parte de quem utiliza estes dados, esperar durante muito tempo para que a transferência esteja concluída de forma a que consiga continuar o seu trabalho.

Existem várias técnicas para reduzir o tempo de transferência de um conjunto de dados, usadas em diferentes soluções. Uma família de soluções recorre a protocolos da camada de transporte alterados [2]. Outras soluções tentam fazer *tunning* de parâmetros dos protocolos de transporte que usam. Além destas soluções existem outras que recorrem a técnicas de mais alto-nível para obter melhores resultados. Este tipo soluções introduz paralelismo na transferência. Para tal, uma solução pode abrir vários canais entre máquinas e enviar os dados por esses canais em simultâneo ou usar várias máquinas para transferirem dados em paralelo.

A redução do volume de dados a enviar também pode ser visto como uma forma de melhorar o desempenho de uma ferramenta de transferência de dados. Nesta situação, uma solução reduz o tempo de transferência enviando menos dados. Para reduzir o volume de dados, podemos usar compressão ou eliminar a redundância dos dados. Na compressão os dados são guardados num arquivo numa forma comprimida. Existem três técnicas para eliminar a redundância dos dados: 1) *delta-encoding*; 2) *cooperating caches*; 3) *compare-by-hash*. As soluções que usam *delta-encoding* comprimem os dados guardando as diferenças (*deltas*) entre ficheiros. Estas soluções começam por escolher uma versão base de um ficheiro e, a partir desse momento, guardam apenas as diferenças entre as novas versões em relação à versão base do ficheiro. Na *cooperating caches* explora-se o facto de haver duas *caches* que guardam os mesmos dados em cada extremo da transferência. Assim, se um determinado conjunto de dados tiver presente numa *cache*, sabe-se que pode estar presente na outra e, se isso suceder, não necessita de ser enviado. As soluções que usam *compare-by-hash* dividem os ficheiros em *chunks* e comparam-nos, um a um, através das suas *hashes*. Sempre que se encontram dois *chunks* com *hashes* iguais descarta-se um, pois assume-se que não existem colisões de *hashes*.

Neste artigo apresentamos uma solução para a transferência de ficheiros que recorre à eliminação de redundância. A nossa solução não altera o formato dos dados nem requer o uso de aplicações exteriores sobre os dados após a sua transferência para que possam ser usados. A nossa solução tem também um protocolo que não necessita de trocar grandes quantidades de informação sobre os dados redundantes (ao contrário das outras soluções).

Este artigo está dividido em cinco secções. A segunda secção fala do trabalho relacionado. Na terceira secção descrevemos como é que a nossa solução elimina a redundância dos dados que envia. Na quarta secção avaliamos a nossa solução. Finalmente, na sexta secção finalizamos este artigo com algumas conclusões.

II. Trabalho Relacionado

Vamos descrever algumas soluções para a transferência de dados. Começaremos por descrever soluções que reduzem o volume de dados a transferir como forma de optimizar o processo de transferência, passando de seguida para as soluções que usam paralelismo para optimizar a transferência.

A. Redução do Volume de Dados

Podemos usar técnicas de compressão para reduzir o volume de dados a enviar. Aqui os dados são comprimidos para um arquivo. Quando os queremos usar novamente, o arquivo tem de ser descomprimido. A diferença entre as várias soluções de compressão é o algoritmo que cada solução usa para fazer

os arquivos. Muito basicamente, a ideia de cada algoritmo é encontrar no ficheiro um símbolo (conjunto de bytes) e atribui-lhe um código de menor tamanho que o símbolo. Na descompressão esses códigos são substituídos pelos símbolos correspondentes. Um exemplo de uma solução de compressão é o Zlib [3].

Outra técnica usada na redução do volume de dados é *delta-encoding*. Esta técnica começa com uma versão base de um ficheiro. A partir daí, todas as versões posteriores são guardadas como um conjunto compacto das diferenças (*deltas*) para a versão anterior. As semelhanças entre versões são encontradas através de algoritmos especializados, mas estes só são aplicáveis a um par de versões. Esta técnica é usado em aplicações como o CVS [4], SVN [5], Porcupine [6], BitKeeper [7] e XDFS [8].

O grande problema desta técnica é o facto de só permitir a eliminação de redundância entre pares de ficheiros. Por esse motivo a gestão dos deltas gerados entre ficheiros não é trivial, sendo por isso usada em versões do mesmo ficheiro consecutivas.

Spring e Watherall [9] propuseram uma técnica baseada em duas *caches* cooperativas, situadas em cada extremo da comunicação. A ideia desta técnica é que ambas as *caches* são iguais e contêm os mesmos dados. Assim, antes de enviar, vê se os dados se encontram na cache e, no caso de ser verdade, estes não necessitam ser enviados.

Esta solução sofre de alguns problemas por usar *caches*. Nomeadamente as *caches* têm um tamanho muito inferior à quantidade de dados que se envia. Outro problema tem a ver com o facto da cache estar isolada ou não. Se for uma cache usada normalmente pela máquina corremos o risco das *caches* entre as máquinas ficarem inconsistentes num espaço de tempo relativamente curto.

As últimas técnicas de redução do volume de dados têm a ver com os princípios da compressão por *hash*. Existem técnicas que, para eliminar a redundância dos dados que enviam, dividem o ficheiro em *chunks* com um tamanho fixo. Após a divisão do ficheiro é calculada uma assinatura para cada um dos *chunks* resultantes. Os *chunks* são depois comparados e, no caso de haver dois iguais, estes são considerados como redundantes e um deles pode ser apagado. Um exemplo de uma solução que usa esta técnica é o rsync [10]. Outra forma de identificar e eliminar redundância entre dados é através de "*content-defined chunks*", usado no LBFS [11] e no haddock-FS [12]. Nesta técnica, o tamanho dos *chunks* pelo qual o ficheiro vai ser dividido varia, ao contrário da anterior. O tamanho de cada *chunk* é determinado através de *Rabin fingerprint* [13]. É calculada a *hash* de cada *chunk* e esta é guardada, normalmente numa *hash table*. Esta *hash table* guarda apenas uma entidade de cada *chunk*. Isto é, sempre que for inserido um *chunk* que tenha a mesma *hash* que um *chunk* já presente na *hash table* este último *chunk* é descartado. Em soluções que usam esta técnica, a transferência dos dados é feita enviando as *hashes* dos *chunks* que constituem o ficheiro. No lado do receptor faz-se uma pesquisa na *hash table* pelas *hashes* que recebeu, requerendo todos os *chunks* para os quais não existe nenhuma entrada válida na sua *hash table*. Existem também soluções que usam várias das técnicas mencionadas para eliminar redundância. Um exemplo disso é o TAPER [14]. Esta solução usa diferentes técnicas para diferentes granularidades de procura de redundância, desde o sistema de ficheiros até chegar a comparação byte-a-byte. Em [15], os autores propõem uma nova técnica para a sincronização de sistemas de ficheiros distribuídos. O mecanismo de sincronização apresentado elimina a redundância dos dados que envia. Nesta técnica as máquinas trocam vectores de versões entre si, que representam o estado do sistema de ficheiros. Recorrendo aos vectores de versões cada máquina consegue, localmente, eliminar a redundância dos dados que envia. Tal é possível pois através dos vectores de versões uma máquina consegue saber os ficheiros que a outra máquina, com quem está a sincronizar, tem.

Todas estas soluções assumem que não existe colisões nas *hashes*. No entanto há soluções que, para evitar este hipotético problema, usam comparação byte-a-byte para confirmar que os *chunks* são redundantes; mas essas soluções sofrem a nível do desempenho pois a comparação byte-a-byte é mais lenta que a comparação das *hashes*. Outro problema deste tipo de soluções tem a ver com o protocolo que elas usam para transferir os dados. A quantidade de meta-dados e *round trips* que estes protocolos têm de usar é demasiado grande introduzindo atrasos. Outra desvantagem presente em algumas destas soluções é que são usadas para sincronizar um sistema de ficheiros ou conjunto de dados. Este comportamento não é aceitável num sistema de transferência de ficheiros, que deve permitir ao utilizador transferir um número de ficheiros arbitrário e à escolha.

B. Transferência de ficheiros

Neste tipo de soluções, a melhoria do tempo de transferência está muito ligada à utilização de vários canais em paralelo. O que vai diferir em grande parte, como veremos, é o local onde se vão buscar os dados e como. No caso de os dados estarem localizados num único local, pode-se abrir vários canais entre as máquinas que estão a transferir os dados e enviar por estes canais o ficheiro fragmentado. Como exemplo de uma solução que usa esta técnica temos o bbcp [16]. Também podemos ter o caso onde os ficheiros ou *chunks* desse ficheiro se encontram replicados por várias máquinas. Neste tipo de situação, quem quer os dados (cliente) cria uma ligação com diferentes máquinas e vai pedindo diferentes *chunks* a cada uma; tal comportamento está presente no famoso BitTorrent [17], mas neste caso particular, e nos que derivam dele, o cliente também envia dados. Em [18] é estudado o facto de os ficheiros estarem replicados por vários servidores e o cliente pedir em paralelo a cada servidor para lhe enviar vários *chunks*.

Estas técnicas não estão preocupadas com a redução do volume de dados que enviam, na verdade isto tem de ser uma opção do utilizador que disponibiliza a partida os dados comprimidos ou através de comandos diz a solução para enviar os dados comprimidos.

III. Protocolo de Transferência

Vamos de seguida explicar como é que a nossa solução elimina a redundância dos dados que envia.

A. Estado da Redundância

Quando um ficheiro entra no nosso sistema, passa a ter um identificador único e é colocado numa lista de ficheiros que ainda não foram analisados. Quando é chamada a função que é responsável por eliminar a redundância dos dados, esta vai à lista de ficheiros por analisar e começa a processá-los um a um. A eliminação de redundância na nossa solução segue os princípios da eliminação de redundância do LBFS [11]. Assim, o ficheiro é dividido em *chunks* e depois é colocado num repositório e, no caso de já existir um *chunk* com o mesmo conteúdo, este último *chunk* a ser inserido é considerado redundante.

Para a nossa solução existem dois tipos de *chunks* redundantes: *root chunk* e *child chunk*. Os *root chunks*, ou *chunks* pai, têm a particularidade de estarem sempre acessíveis localmente, logo os *child chunks*, ou *chunks* filho, podem ser apagados. Assim, dado um conjunto de *chunks* redundantes, nós consideramos um *chunk* como *root chunk* e os restantes como *child chunks*.

Estando a análise e divisão dos *chunks* feita, estamos em condições de guardar os ficheiros com a sua redundância eliminada. Cada ficheiro é reescrito sem os *child chunks* que este tem. Em lugar de cada *child chunks* é escrito, no início do ficheiro, um conjunto de meta-dados que permitem identificar o "pai" desse *child chunk*. Estes meta-dados contêm o *offset* no ficheiro onde começa o *child chunk*, o tamanho do *chunk*, o identificador do ficheiro que tem o *root chunk* e o *offset* do *root chunk* no seu ficheiro. A este conjunto de meta-dados presente em cada ficheiro damos o nome de *Lista de referências para chunks redundantes*. Além desta lista, guardamos também uma estrutura de dados auxiliar que contém pares ⟨identificador do ficheiro que tem o child chunk; identificador do ficheiro que tem o respectivo root chunk⟩ para cada *child chunk*. Esta estrutura diz que para um dado ficheiro existe, pelo menos, um *root chunk* num outro ficheiro. Esta estrutura serve apenas para optimizar o processo de identificação de *chunks* redundantes quando estivermos a eliminar a redundância. A esta estrutura damos o nome de *Tabela de similaridades*.

B. Protocolo de Eliminação de Redundância

Antes de explicarmos como funciona o protocolo de eliminação de redundância, temos de introduzir mais uma estrutura de dados auxiliar. Esta estrutura serve para guardar os ficheiros que já foram transferidos entre pares de máquinas. Assim, uma máquina necessita de uma destas estruturas por cada outra máquina com que comunica. Nós usamos os identificadores dos ficheiros nesta estrutura para identificar os ficheiros que já foram transferidos. Chamamos a esta estrutura *mapa das transferências efectuadas*. Além desta estrutura existem mais dois conjuntos que temos de introduzir. São eles: o conjunto de ficheiros já transferidos, que iremos designar por *C*; o conjunto de ficheiros a transferir, que iremos designar por *T*. *C* é identificado pelo mapa das transferências efectuadas.

Estamos agora em condições de explicar o protocolo de eliminação de redundância. Quando uma máquina recebe um pedido de transferência de um ficheiro, tem de identificar quais são os dados redundantes entre *T*, os ficheiros que tem de enviar, e *C*, os ficheiros que sabe que a outra máquina tem. A redundância entre os conjuntos *C* e *T* pode aparecer em três casos, ilustrados na figura 1: i) Um *chunk* em *T* tem um *root chunk* em *C*; ii) Um *chunk* em *T* é um *root chunk* de um *child chunk* em *C*; iii) Um *chunk* em *T* tem um *root chunk* num ficheiro presente na máquina que, por sua vez, tem outro *child chunk* em *C*. O caso i é o mais fácil de identificar, iterando a lista de referências de *chunks* redundantes que cada ficheiro tem, saber se um *child chunk* é redundante ou não, consiste em ver se o identificador do ficheiro que tem o *root chunk* pertence a *C*. Para os casos ii e iii temos de recorrer à tabela de similaridades. Para o caso ii temos de ver se existe algum identificador em *C* que tenha um *root chunk* em *T*. Ou seja, se existe algum par $\langle c, t \rangle$ onde *c* e *t* identificadores que pertencem a *C* e *T*, respectivamente. O terceiro e último caso é identificado percorrendo a lista de referências para *chunks* redundantes, para cada ficheiro em *T*, e em cada *chunk* dessa lista temos de saber qual o identificador do ficheiro que tem o *root chunk*. Sabendo esse identificador e recorrendo à tabela de similaridades, temos de ver se existe algum par que diga que um ficheiro em *C* tem um *root chunk* no ficheiro com este identificador.

C. Envio dos Dados

Identificados os *chunks* redundantes entre *C* e *T*, estamos em condições de enviar os dados. Antes de enviarmos os dados não redundantes, temos de enviar uma lista com os meta-dados relativos aos *chunks* redundantes entre *C* e *T*. Esses meta-dados contêm o identificador do ficheiro e o seu tamanho total, mais a lista de todos os *chunks* redundantes para esse ficheiro, similar à lista de referências para *chunks* redundantes que cada ficheiro tem. Enviados estes meta-dados, são depois enviados os *chunks* não redundantes e o ficheiro é reconstruído no receptor.

IV. Avaliação

Nesta secção avaliamos a nossa solução; tivemos em atenção o volume de dados transferidos e o tempo que cada solução necessita para enviar esses dados.

A. Experiência

Para o teste foram usadas duas máquinas, uma com um processador Intel (R) Pentium(R) 4 a 3.20GHz e 2GB de RAM, a outra com um processador Pentium(R) 3 (Coppermine) e 500MB de RAM. Ambas as máquinas estão ligadas por um switch de 100Mb/s.

As *workloads* usadas são constituídas por dois ficheiros, para testar o protocolo de eliminação de redundância entre ficheiros. Quanto às *workloads* usadas, usámos três *workloads* reais: i) *gentoo*, este conjunto de ficheiros de testes é constituído por duas imagens de um disco de instalação do

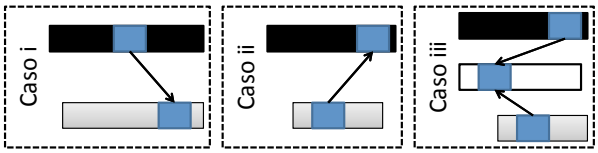

Figura 1. Os três casos possíveis de redundância entre chunks de ficheiros nos conjuntos *T* e *C*. O ficheiro a perto está contida em *T*,enquanto que o ficheiro cinzenta pertence ao conjunto *C*

sistema operativo Gentoo Linux (iso). As imagens são de uma versão do Gentoo para a arquitectura x86 e são de builds mensais deste sistema operativo. A primeira versão diz respeito a uma imagem de 1 de Abril de 2009 e a segunda versão é uma imagem de 15 de Abril do mesmo ano; ii) *eclipse*, esta *workload* é constituída pela versão 3.4.0 e 3.4.2 do IDE eclipse. A versão 3.4.0 corresponde à primeira versão e a 3.4.2 à segunda versão de teste. Cada versão do eclipse é um arquivo constituído por ficheiros de texto em ASCII, código Java compilado (jars), algumas imagens (jpgs) e executáveis; iii) *sdes*, obtivemos o código dos projectos de dois grupos de alunos das cadeiras de Sistemas Distribuídos e de Engenharia de Software do Curso de Engenharia Informática e de Computadores do Instituto Superior Técnico. Para além dos projectos, juntámos também algum material das cadeiras como slides, exemplos de código e guias.

As soluções avaliadas foram: i) para a redução do tamanho dos dados usando *delta-encoding*, utilizámos a ferramenta xdelta3 [19], versão 3.0; ii) na avaliação da nossa solução, FTPChunk. Foram executadas várias versões deste sistema e cada execução varia no tamanho médio dos *chunks* e no uso ou não de compressão. Por uma questão de simplicidade, vamos só apresentar os valores obtidos para a versão da nossa solução a correr com *chunks* de tamanho 2KB; iii) *scp* é uma ferramenta para transferência de ficheiros presente nos sistemas operativos Linux. Esta vai ser usada para transferir dados com e sem compressão; iv) Usámos o rsync [10] para testar uma ferramenta que eliminasse a redundância através de um algoritmo de comparação por *hash*, com *chunks* de tamanho fixo. Foi usada a versão 2.6.9 deste software. O rsync foi executado de duas maneiras: com e sem compressão.

B. Resultados

Vamos agora analisar os resultados obtidos para as *workloads* e soluções avaliadas. Neste artigo só analisaremos os dados correspondentes à transferência do segundo ficheiro de cada *workload*, pois é a partir desse momento que as soluções de eliminação de redundância começam a ter resultados significativos. Nos gráficos apresentados as soluções que tiverem no nome a palavra *nocompress* não usam compressão enquanto que as restantes comprimem os dados antes dos enviarem.

Começando por analisar o volume de dados que cada solução transfere, figura 2, vemos que para a *workload sdes* a nossa solução é a que envia a menor quantidade de dados. Até mesmo a versão que não usa compressão envia menos dados que as restantes soluções avaliadas. Ao contrário das restantes *workloads*, a solução xdelta tem um fraco desempenho nesta *workload* sendo só melhor que a solução que envia os dados comprimidos (scp) e a solução que envia os dados sem qualquer tipo de redução no volume (scp nocompress). Nesta *workload* a nossa solução é superior as restantes devido a quantidade de *chunks* redundantes presentes no próprio ficheiro. O xdelta não consegue encontrar este tipo de redundância e é por isso que envia uma quantidade de dados superior. Na *workload gentoo* temos um comportamento semelhante ao da *workload* anterior, mas nesta o xdelta envia menos dados que o rsync com e sem uso de compressão. Contudo, esta é uma diferença marginal. Ambas as versões da nossa solução também enviam sensivelmente a mesma quantidade de dados. Na *workload* eclipse, a melhor solução passa a ser o xdelta, mas, neste caso, a nossa solução é a melhor a seguir ao xdelta.

Vemos agora os resultados obtidos para o tempo que cada solução leva a fazer a transferência, figura 3. É claro que ambas as versões do rsync são as piores ao nível do seu desempenho no envio dos dados. Isto deve-se ao facto do rsync ser uma ferramenta pesada computacionalmente e das máquinas onde foram feitos os teste não corresponderem ao poder computacional normal para uma máquina da actualidade. Além disso, o rsync faz todo o processo de redução do volume de dados em tempo de transferência, enquanto que nós assumimos que esse tempo não seria considerável. Isto porque nos casos reais, quando vamos transferir um ficheiro de um servidor, não esperamos que este comprima e envie os dados, mas sim que já os tenha comprimidos em disco e, assim, é só enviar quando recebe o pedido. Por estas razões vamos olhar com mais detalhe para o resto dos resultados, com excepção do rsync. Para a *workload sdes*, vemos que é a solução com uso de compressão que necessita de menos tempo a transferir os dados. Contudo, a nossa versão sem compressão tem uma diferença marginal em ralação a outra, transferindo ambas os dados na casa dos 11 segundos, seguem-se as soluções xdelta e scp. Na transferência da *workload gentoo*, o xdelta transfere os dados mais rapidamente que qualquer outra solução, enviando em 5 segundos. Já as duas versões da nossa solução enviam os dados em 6 segundos. Ambas as versões do scp enviam os dados em 7 segundos. Na última *workload* o xdelta é um claro vencedor enviando os dados em metade do tempo que as restantes soluções necessitam, excluindo o rsync com e sem compressão.

No entanto existe um aspecto que temos de ter em conta em relação a estes resultados. O esquema dos testes está a beneficiar o xdelta. Isto porque só estamos a transferir um ficheiro e não um conjunto deles. No ambiente mais realista com vários ficheiros, o xdelta ia, provavelmente, atingir piores resultados,

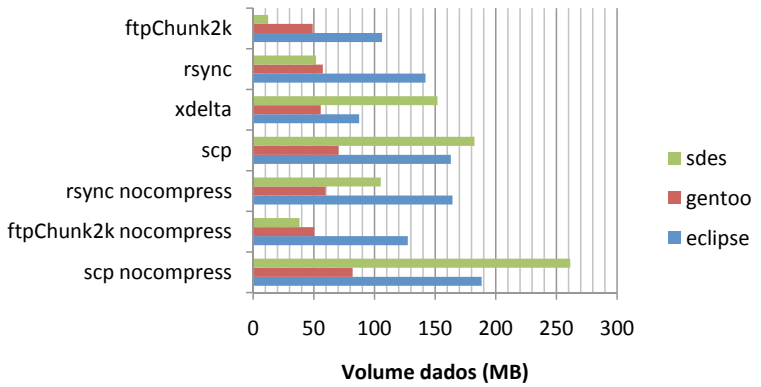

Figura 2. Volume de dados transferido para as diferentes *workloads* pelas diferentes soluções

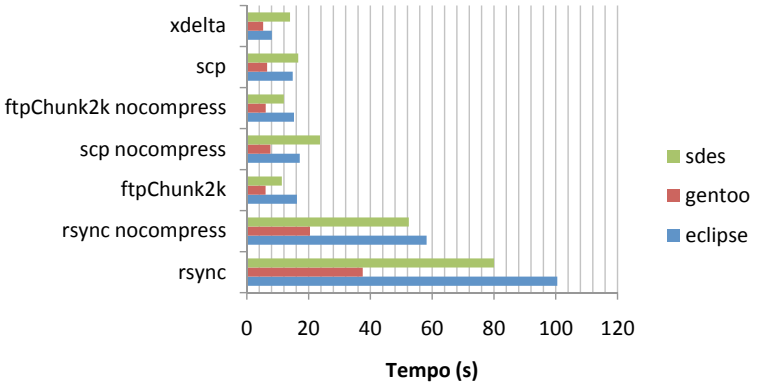

Figura 3. Tempo total da transferência para as diferentes workloads pelas diferentes soluções

pois tinha de recorrer a uma heurística para saber que pares de ficheiros devia compara para ver as diferenças. Este facto pode levar a que o xdelta encontre menos redundância, uma vez que só pode comparar pares de ficheiros e pode-se dar o caso de um par não ser o mais adequado. Apesar deste facto a nossa solução consegue ter resultados comparáveis ao xdelta.

V. Conclusão

Foi apresentada uma solução para a transferência de ficheiros com eliminação de redundância, que usa princípios da cache cooperativa para identificar os ficheiros que cada interveniente tem na transferência e consequentemente os *chunks* redundantes. Esta solução elimina a redundância no local onde os dados são enviados e pode ser feita antes da transferência acontecer. Assim, quando se está a realizar uma transferência, apenas é necessário verificar, dos *chunks* a serem enviados, quais os que já estão presentes do outro lado da ligação e, como tal, não necessitam de ser enviados. Através disto, conseguimos criar uma aplicação que consegue fazer a transferência de ficheiros individuais e que não se limita a sincronizar pastas ou sistemas de ficheiros como na maioria das soluções que usam eliminação de redundância. Também não necessita de um grande número de comunicações e troca de meta-dados entre os vários intervenientes para que possa ser feita a eliminação da redundância dos dados que envia. Além disso, mostrámos as potencialidades de uma ferramenta de transferência de ficheiros que use tais mecanismos para enviar os dados.

Vimos também que a nossa solução tem bom desempenho na transferência de ficheiros sendo, na maioria das *workloads* testadas, tendo resultados comparáveis com a melhor solução, tanto a nível da quantidade de dados que envia como do tempo que necessita para os enviar. Apesar de não ter sido uma avaliação extensa, estes resultados mostram que existem certos tipos de dados onde soluções de transferência de ficheiros baseada em *chunks* podem ser aplicadas com sucesso.

VI. Trabalho Futuro

Como trabalho futuro iremos estudar melhor o impacto que diferentes tamanhos de *chunks* têm na nossa solução. Outro tipo de testes que queremos fazer têm a ver com diferentes larguras de banda. Queremos ver os ganhos que a nossa solução consegue ter em diferentes tipos de rede. Especialmente em redes de baixa velocidade, onde o peso de enviar 1MB é totalmente diferente de uma rede mais rápida. Também queremos testar a nossa solução com um maior numero de ficheiros e ficheiros com um maior tamanho. Queremos também alargar o número de soluções testadas.

Referências

[1] Y. Cui, R. Moore, K. Olsen, A. Chourasia, P. Maechling, B. Minster, S. M. Day, Y. Hu, J. Zhu, A. Majumdar, and T. Jordan, "Enabling very-large scale earthquake simulations on parallel machines," in *ICCS (1)*, ser. Lecture Notes in Computer Science, Y. Shi, G. D. van Albada,

J. Dongarra, and P. M. A. Sloot, Eds., vol. 4487. Springer, 2007, pp. 46–53. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72584-8_7

[2] E. He, P. V.-B. Primet, and M. Welzl, "A survey of transport protocols other than "standard" tcp," 2005.

[3] L. P. Deutsch and J.-L. Gailly, "ZLIB compressed data format specification version 3.3," Internet RFC 1950, United State, may 1996.

[4] P. C. et al., "Version managemnet with cvs," [Online Manual]http://www.cvshome.org/docs/manual/, 1993.

[5] M. Pilato, *Version Control With Subversion*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2004.

[6] Y. Saito, B. N. Bershad, and H. M. Levy, "Manageability, availability, and performance in porcupine: a highly scalable, cluster-based mail service," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, p. 298, 2000.

[7] V. Henson and J. Garzik, "Bitkeeper for kernel developers," http://infohost.nmt.edu/ val/ols/bk.ps.gz, 2002.

[8] J. P. Macdonald, "File system support for delta compression," Department of Electrical Engineering and Computer Science, University of California at Berkeley, Tech. Rep., 2000.

[9] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *In Proceedings of ACM SIGCOMM*, 2000, pp. 87–95.

[10] A. Tridgell, "Efficient algorithms for sorting and synchronization," Ph.D. dissertation, Australian National University, 1999.

[11] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2001, pp. 174–187.

[12] J. Barreto and P. Ferreira, "A replicated file system for resource constrained mobile devices," March 2004.

[13] M. O. Rabin, "Fingerprinting by random polynomials," Harvard Aiken Computation Laboratory, Tech. Rep. TR-15-81, 1981.

[14] N. Jain, M. Dahlin, and R. Tewari, "Taper: tiered approach for eliminating redundancy in replica synchronization," in *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2005, pp. 21–21.

[15] J. Barreto, "Optimistic replication in weakly connected resource-constrained environments," Ph.D. dissertation, Instituto Superior Técnico - Universidade Técnica de Lisboa, 2008.

[16] A. Hanushevsky, A. Trunov, and L. Cottrell, "Peer to peer computing for secure high performance data copying," presented at CHEP'01: Computing in High-Energy Physics and Nuclear, Beijing, China, 3-7 Sep 2001.

[17] B. Cohen, "Incentives build robustness in bittorrent," in *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, 2003. [Online]. Available: http://citeseer.nj.nec.com/cohen03incentives.html

[18] C. Gkantsidis, M. Ammar, and E. Zegura, "On the effect of large-scale deployment of parallel downloading," in *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*. Washington, DC, USA: IEEE Computer Society, 2003, p. 79.

[19] J. Macdonald, "xdelta, http://xdelta.org/," World Wide Web electronic publication, 2009. [Online]. Available: http://xdelta.org/