一种新的面向移动云下载的预压缩文件同步机制

喻 宇[1] 胥钟予[1] 张 兴[1] 李永刚[2] 裴二荣[2]

[1](国家电网重庆电力公司信息通信分公司 重庆 401120)

[2](重庆邮电大学通信与信息工程学院 重庆 400065)

摘 要 远程文件的快速同步是移动云下载、Web 镜像、远程系统备份等网络场景的基础技术，当前，在 Linux 操作系统下广泛使用 Andrew Tridgell 提出的 rsync 算法。提出一种新的面向移动云下载的预压缩文件同步机制，通过在终端引入缓存及信息重用，能显著减少终端计算量并降低网络带宽需要。结果表明，在 rsync 基础上实现的改进算法，通过信息的重用和缓存，可以简化计算并显著减少传输的数据量，尤其适合于能耗敏感的移动云下载等远程文件同步场合。

关键词 移动云下载 文件快速同步 信息重用 能耗敏感

中图分类号 TP3 **文献标志码** A DOI:10.3969/j.issn.1000-386x.2024.12.004

A NOVEL PRE-COMPRESSED FILE SYNCHRONIZATION MECHANISM FOR MOBILE CLOUD DOWNLOADING

Yu Yu[1] Xu Zhongyu[1] Zhang Xing[1] Li Yonggang[2] Pei Errong[2]

[1](*State Grid Chongqing Electric Power Company Information Communication Branch, Chongqing 401120, China*)

[2](*School of Communication and Information Engineering, Chongqing University of Posts and Telecommunications, Chongqing 400065, China*)

Abstract Fast synchronization of remote files is the basic technology for mobile cloud download, Web mirroring, remote system backup and other network scenarios. At present, the rsync algorithm proposed by Andrew Tridgell is widely used in Linux operating system. This paper proposes a new pre-compressed file synchronization mechanism for mobile cloud download. By introducing cache and information reuse at the terminal, it can significantly reduce the terminal computation and reduce the network bandwidth requirement. The results show that the improved algorithm based on rsync can simplify the calculation and significantly reduce the amount of data transferred through information reuse and caching, especially suitable for remote file synchronization occasions such as mobile cloud download which is sensitive to energy consumption.

Keywords Mobile cloud downloading Quick file synchronization Information reuse Energy-sensitive

0 引 言

随着移动社交网络的蓬勃发展，人们越来越习惯于通过移动设备，包括智能手机、智能手表等，与朋友共享信息。移动云共享[1]不仅可以实现数据同步和备份，还提供了共享平台，其他用户可以实时获取自己共享的信息。在移动互联网时代，移动终端的普及和人们对共享的需求将极大地促进移动云共享的发展。但是，移动终端电池供电，受电池容量限制，现有的数据同步算法常常导致大量的带宽和流量需求，使得耗能过大。因而，针对移动终端使用特点对远程同步机制进行优化是必不可少的。

快速的数据同步使客户端和服务器上的相似文件可以通过最少的网络传输在最短的时间内保持一致。在低带宽和流量敏感的无线网络环境中，快速数据同

收稿日期：2021－06－13。重庆市基础研究与前沿探索项目(cstc2018jcyjAX0507)；国家电网有限公司科技项目(2021 渝电科技 23#)。**喻宇**，工程师，主研领域：可信软件工程与测试。**胥钟予**，工程师。**张兴**，高工。**李永刚**，副教授。**裴二荣**，副教授。

步技术可以解决移动云共享中遇到的主要问题。目前,快速数据同步算法主要包括 rsync 算法[2-3]及其改进算法[4-7]。该算法由 Andrew Tridgell 在 1996 年提出,已经被集成到 Linux 系统中,用于同步更新两处计算机的档案与目录,其思路是利用差分编码以减少数据传输。前期研究表明,对于内容相似性基本上与存储字节的相似性无关的文件(例如. rar 和. exe),rsync 算法存在性能退化问题。为此,本文提出一种预压缩的改进思路,通过在终端引入缓存及信息重用(充分利用已经计算得到的 checksum,避免重复计算)的思想,能显著减少终端计算量并降低网络带宽需要,尤其适用于能耗敏感的移动云同步场合。

1 相关算法介绍

目前快速文件同步主要的方法包括 rsync[2-3]、multiroundRsync[4]、unison[5]、tpsync[6-7]、相关改进算法[7]。其主要原理是计算客户端和备份存储服务器中文件的差异,然后传输差异来更新服务器文件。文件的改变可能由许多原因导致,比如,对文件系统的操作方式包括,如插入、删除、复制、创建、更名等。

1.1 校验算法

校验算法的主要思想是通过对源端旧文件进行固定长度分块并计算每个块的哈希值发送至目的端,目的端利用弱校验和强校验方法寻找匹配块。对于不匹配的块实行差量压缩传输来达到同一文件相似版本之间的同步。

1.1.1 弱校验方法:rolling checksum 算法

在 rsync 算法中使用的弱校验方法为 rolling checksum 算法[3],它是根据马克艾德勒(Mark Adler)的 adler-32 校验和算法计算而来。使用该算法,只要给出数据块 $X_1,X_2,\cdots,X_n$ 的 rolling checksum 值以及字节 X_1 和 X_{n+1} 的值,就可以快速算出 $X_2,X_3,\cdots,X_{n+1}$ 的 rolling checksum 值。rolling checksum 算法定义为:

$$s(k,l)=a(k,l)+2^{16}b(k,l) \tag{1}$$

式中:$a(k,l)=(\sum_{i=k}^{l}X_i)\bmod M$;$b(k,l)=(\sum_{i=k}^{l}(l-i+1)X_i)\bmod M$,$s(k,l)$ 是数据块 $X_k,X_{k+1},\cdots,X_l$ 的 rolling checksum 值。出于简单及速度考虑,令 $M=2^{16}$。

rolling checksum 算法的重要性质在于可以利用如下关系式快速算出下一个数据块的 rolling checksum 值:

$$a(k+1,l+1)=a((k,l)-X_k+X_{l+1})\bmod M,$$

$$b(k+1,l+1)=(b(k,l)-(l-k+1)X_k+a(k+1,l+1))\bmod M$$

根据上述关系式我们可以快速算出数据块 $X_{k+1},X_{k+2},\cdots,X_{l+1}$ 的 rolling checksum。该算法虽然运算速度快,但碰撞率较高。

1.1.2 强校验方法:MD5 算法

MD5(Message-Digest Algorithm 5)即信息-摘要算法 5,用于确保信息传输完整一致。是计算机广泛使用的哈希算法之一。该算法输入任意长度的消息,输出 128 位消息摘要,处理以 512 位输入数据块为单位。MD5 算法可简要地叙述为:MD5 以 512 位分组来处理输入的信息,且每一分组又被划分为 16 个 32 位子分组,经过了一系列的处理后,算法的输出由四个 32 位分组组成,将这四个 32 位分组级联后将生成一个 128 位散列值。MD5 算法的优势为其碰撞率极小几乎可忽略不计,缺点为计算复杂、开销大。

1.2 rsync 算法

1.2.1 rsync 算法同步更新流程

文件同步的场景是,客户端(C 端)向服务器端(S 端)请求文件下载,服务器端返回相应文件数据。rsync 算法的同步更新流程共分为三个阶段,如图 1 所示。

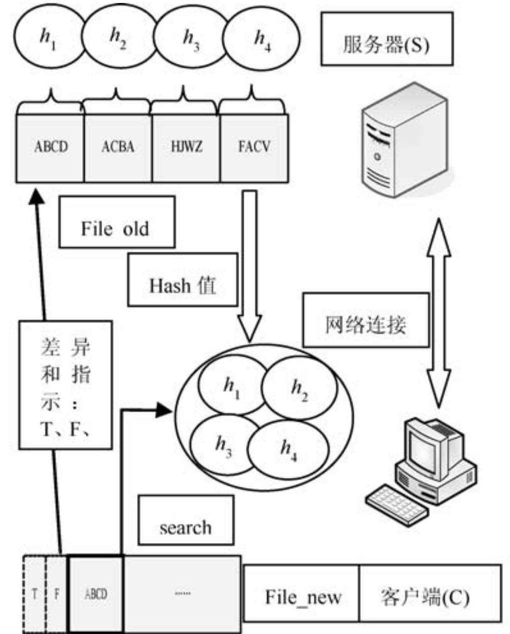


图 1 rsync 算法同步更新流程

第一阶段(S 端):S 端将 file_old 分块,计算每个 block(块)的弱、强检验值(h_1,h_2,h_3,h_4),并将这些检验值信息发送到 C 端。

第二阶段(C 端):C 端接收 S 端发送的 blocks 的检验值对信息,并以 rolling checksum 的 16 位哈希值建立哈希索引表,而后对本地的 file_new 执行三级匹配检索(rsync 算法核心,计算量密集),最后将 blocks 的匹配信息以及 missed string(file_new 中新增的不存在于 file_

old 的数据块,即图 1 中的字符 T、F)发送到 S 端。

第三阶段(S 端):利用 C 端发送回来的匹配内容及本地的 file_old 重构出 file_new。

1.2.2　rsync 算法三级匹配检索流程

Rsync 算法采用三级匹配检索流程,如图 2 所示。

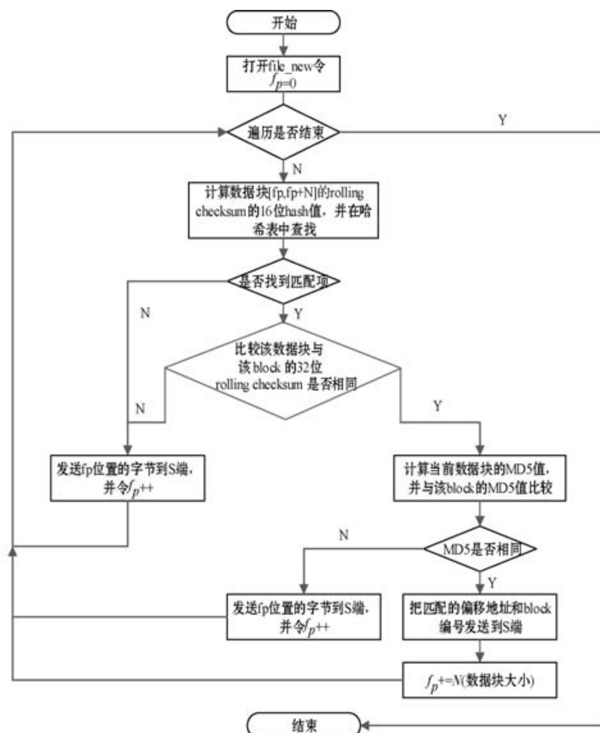


图 2　rsync 算法三级匹配检索流程

Rsync 算法执行三级匹配检索时,遍历 file_new,以字节为单位,从偏移地址 fp = 0 开始反复执行如下步骤:

计算从偏移地址开始大小为 N 的 block 的 rolling checksum 及其 16 位 hash 值。

从旧文件 blocks 的哈希表中查找是否有匹配的 16 位快速哈希值。

如果找到匹配的项,则比对 32 位 rolling checksum,判断是否相同。

若 rolling checksum 也相同则计算其 md5 值,如果也匹配,则发送该 block 的偏移地址 j 和该 block 的编号给 S 端,并对偏移地址 j 进行加 N 操作。

如果没有找到,或者 md5 值不匹配,则传输偏移地址 f_p 处字节给 S 端,f_p + = 1。

1.2.3　rsync 算法性能分析

rsync 算法对文件切块大小参数 N 比较敏感,不同的 block 大小可能导致最后传输总流量的较大差异。块长度(粒度)越小, 系统越复杂, 检测数据的开销越大;块粒度越大,重复数据被漏检的概率就越大。rsync 实验结果表明,当其分块大小参数取默认值 700 Byte 时综合性能最优。

算法的运算量较大:首先,S 端需要计算整个文件的 MD5 值;其次,C 端执行的三级匹配检索运算量较大,是整个算法运算量最集中的部分。

2　rsync 算法移植及改进

目前基于 rsync 算法及其改进的快速数据同步算法主要针对 pc 领域,其改进思路也是引入更复杂的机制来提高 rsync 算法的性能、进一步降低网络传输流量,较少有针对移动终端设备进行的优化改进。考虑到移动终端设备对电池续航、带宽及流量的敏感性,我们从另一个方向提出对 rsync 算法的改进思路,即通过在终端引入缓存及信息重用(充分利用已经计算得到的 checksum,避免重复计算)的思想以达到简化终端计算量并降低网络流量的目的。

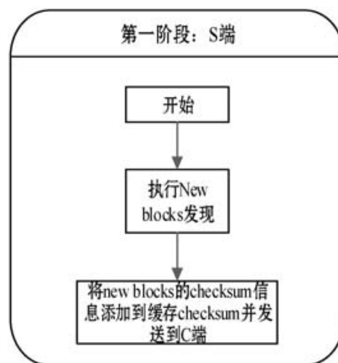
2.1　rsync 算法改进思路

改进 rsync 算法的出发点:各版本文件的相似度为逐渐退化的过程。实际中同一个文件的相近几个版本间的差别可能并不大,即被用于重构 version1 文件的 version0 文件中的重用 blocks 也很有可能无修改地被其后的新版本文件所重用。

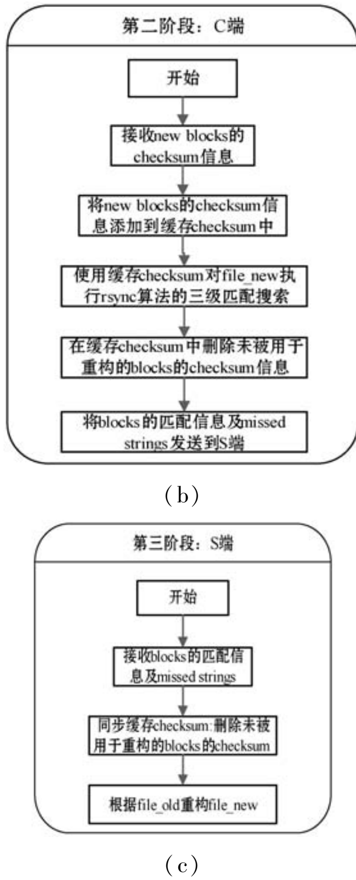
当前 rsync 算法只考虑相邻两个版本文件间的同步更新,对于同一文件多个更新版本的问题,其做法依旧是每轮同步更新时都将相邻旧版本文件分块并计算每个 block 的 checksum。考虑到大部分上几轮已计算的 blocks 很可能无修改地被其后多个新版本文件所重用,当前 rsync 算法显然做了很多无用的重复计算。故我们可考虑重用旧信息(blocks 的 checksum)以简化计算量并加速同步过程,即重用旧 version 的 checksum 以达到简化计算的目的。

2.2　rsync 算法改进技术实现

我们的改进想法主要是引入缓存 checksum(即保存第一次使用 rsync 算法时产生的 blocks 的 checksum),并实时在两端更新维护缓存 checksum,即对 rsync 算法的三阶段同步更新流程按图 3 进行修改。



(a)



(b)

(c)

图 3 改进 rsync 算法同步更新流程

在图 3 的同步更新流程中，第一阶段中我们不再去将 file_old 分块并计算每个 block 的 checksum，取而代之我们执行 new blocks 的发现过程，并将 new blocks 的 checksum 写入到缓存 checksum 中同时传送到 C 端。在第二阶段的开始与结束处，C 端实时更新缓存 checksum（添加 new blocks 的 checksum，删除无用 blocks 的 checksum）。最后的第三阶段，S 端也添加进缓存 checksum 更新模块。

其中，new blocks 的发现过程如图 4 所示。

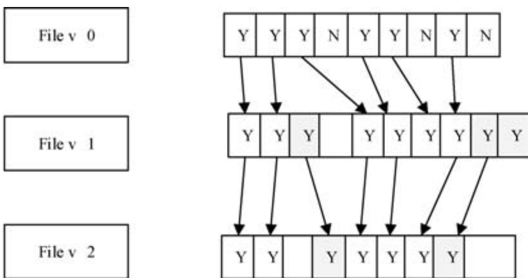


图 4 new blocks 的发现过程图示

图 4 中 Y 和 N 分别表示被重用和未被重用的 block，深色标识的 Y 表示在 new blosks 发现过程中新计算的 blocks。

3 rsync 算法及改进算法的实验性能测试

在实验中我们将 Linux 平台下的开源 rsync 使用 C ++ 语言移植到 Windows 平台，并且我们成功实现了使用 rsync 算法及本文改进算法重构出新版本文件。为了简化实现，我们只移植了 rsync 软件的核心部分（单文件同步更新流程的三个阶段），同时省去了对网络中传输数据进行压缩的流程。

在实验测试中我们选择 3 个. doc 文件进行测试（文件大小分别为 180 KB，1 700 KB，9 000 KB），每个文件均有 4 个版本。测试时，我们分别用 rsync 算法及本文改进算法对每个文件进行 3 次文件版本更新（版本 v0 -> 版本 v1，版本 v1 -> 版本 v2，版本 v2 -> 版本 v3）。每个版本文件更新方法为：在旧版本文件中随机选择多处进行修改、增加、删除内容，产生新版本文件。表 1 至表 3 为本次实验测得的统计结果。

表 1 file1（180 KB）的同步更新统计结果

版本更新	Rsync 算法/改进 Rsync 算法					
	Total_blocks	Blocks_computed	Tag_hits	Matchs	Missed_string_len/KB	Data_transfer/KB
file1_v0（178 KB）-> file1_v1（178 KB）	260/260	260/260	356/356	198/198	42.6/42.6	57.9/57.9
file1_v1（178 KB）-> file1_v2（179 KB）	261/259	261/61	478/375	204/202	39.5/40.4	55.6/46.2
file1_v2（179 KB）-> file1_v3（192 KB）	262/258	262/56	638/453	192/192	61.4/60.8	77.4/66.2

版本更新	Rsync 算法/改进 Rsync 算法		
	Reuse_rate/%	Resue_blocks_v0	Resue_rate_v0/%
file1_v0（178 KB）-> file1_v1（178 KB）	76.0/76.0	198/198	76.0/76.0
file1_v1（178 KB）-> file1_v2（179 KB）	78.3/77.6	-/192	-/73.7
file1_v2（179 KB）-> file1_v3（192 KB）	73.3/73.3	-/185	-/71.0

表 2 file2（1 700 KB）的同步更新统计结果

版本更新	Rsync 算法/改进 Rsync 算法					
	Total_blocks	Blocks_computed	Tag_hits	Matchs	Missed_string_len/KB	Data_transfer/KB
File2_v0（1 695 KB）-> File2_v1（1 697 KB）	2 480/2 480	2 480/2 480	6 516/6 516	2 313/2 313	116.2/116.2	280.2/280.2
File2_v1（1 697 KB）-> File2_v2（1 730 KB）	2 483/2 449	2 483/136	10 245/9 116	2 198/2 204	227.8/223.7	389.4/264.6
File2_v2（1 730 KB）-> File2_v3（1 636 KB）	2 531/2 464	2 531/260	7 314/5 358	2 185/2 194	142.0/136.0	305.6/183.1

续表 2

版本更新	Rsync 算法/改进 Rsync 算法		
	Reuse_rate/%	Resue_blocks_v0	Resue_rate_v0/%
File2_v0(1 695 KB) -> File2_v1(1 697 KB)	93.3/ 93.3	2 313/ 2 313	93.3/ 93.3
File2_v1(1 697 KB) -> File2_v2(1 730 KB)	88.5/ 88.8	-/ 2 198	-/ 88.6
File2_v2(1 730 KB) -> File2_v3(1 636 KB)	86.3/ 86.7	-/ 2 055	-/ 82.9

表 3　file3(9 000 KB)的同步更新统计结果

版本更新	Rsync 算法/改进 Rsync 算法								
	Total_blocks	Blocks_computed	Tag_hits	Matchs	Missed_string_len/KB	Data_transfer/KB	Reuse_rate/%	Resue_blocks_v0	Resue_rate_v0/%
File2_v0(8 849 KB) -> File2_v1(8 883 KB)	12 945/ 12 945	12 945/ 12 945	506 006/ 506 006	9 002/ 9 002	2 729.3/ 2 729.3	3 537.3/ 3 537.3	69.5/ 69.5	9 002/ 9 002	69.5/ 69.5
File2_v1(8 883 KB) -> File2_v2(9 421 KB)	12 995/ 12 701	12 995/ 3 699	472 228/ 448 160	10 085/ 10 079	2 527.3/ 2 531.1	3 360.7/ 2 880.2	77.6/ 77.5	-/ 8 873	-/ 68.5
File2_v2(9 421 KB) -> File2_v3(9 189 KB)	13 782/ 13 410	13 782/ 3 331	548 276/ 497 005	9 348/ 9 345	2 798.3/ 2 800.3	3 654.2/ 3 119.5	67.8/ 67.8	-/ 8 403	-/ 64.9

上述测试中所选择的块大小 Block_size =700 Byte。

第一轮同步更新（版本 v0 -> 版本 v1）时，由于 S、C 两端均无缓存 checksum 故本文改进算法首次调用 rsync 算法，第一轮同步更新结束后缓存 checksum 用于今后更新。

表 1 – 表 3 中统计参数说明如下：

Total_blocks：旧版文件在本轮更新中产生的 blocks 的总个数。

Blocks_computed：本轮更新中需 S 端计算 checksum 的 blocks 的个数。

Tag_hits：C 端进行三级匹配检索时，rolling checksum 的 16 位 hash 值匹配的次数。

Matchs：C 端进行三级匹配检索时，重用 block 的个数（即新、旧文件中具有完全相同 block 的个数）。

Missed_string_len：新文件中未找到匹配的数据块的长度。

Data_transfer：本轮更新中总共需在网络中传输的数据量（包括 Missed_string_len 及交互信息）。

Reuse_rate：本轮更新中旧文件的重用率。

Reuse_blocks_v0：本轮更新中重用版本 v0 文件中 block 的个数。

Reuse_rate_v0：本轮更新中版本 v0 文件的重用率。

综合每个文件的 3 轮同步更新结果，我们得出表 4。

表 4　各文件同步更新统计结果合计

文件	Rsync 算法/改进 Rsync 算法			
	Total_blocks_computed	Average_matchs	Total_data_transfer/KB	Average_reuse_rate/%
File1	783/ 377	198/ 197	190.9/ 170.3	75.9/ 75.6
File2	7 494/ 2 876	2 232/ 2 237	975.2/ 727.7	89.4/ 89.6
File3	39 722/ 19 975	9 478/ 9 475	10 552.2/ 9 537	71.6/ 71.6

分析上述统计结果，我们可得出如下结论：

（1）对比统计参数 block_computed 可发现在后续轮数的更新中，本文改进算法相比 rsync 算法对终端计算量的需求大大降低。计算量降低的程度取决于各版本文件间的相似度，与上一轮 missed_string_len 成负相关关系（即当 missed_string_len 越大需计算的 blocks 就越多）。

（2）对比统计参数 data_transfer 可发现改进算法相比 rsync 算法所需的网络流量也有一定程度的降低。网络流量的降低主要来自两端交互信息量（由 S 端传输到 C 端的 blocks 的 checksum 以及 C 端传回 S 端的匹配结果）的减少。

（3）对比统计参数 matchs 和 reuse_rate 可发现我们的改进算法在降低终端计算量和网络流量的同时，几乎不降低 rsync 算法的匹配效果（即对旧文件的重用率）。

（4）统计参数 reuse_blocks_v0 和 reuse_rate_v0 说明了本文改进算法的合理性，即同一个文件的相近几个版本间的差别并不大，被用于重构 v1 文件的 v0 文件中的重用 blocks 也很有可能无修改地被其后的新版本文件所重用。

另外，在实验中我们还发现了一个有趣的现象：我

们将 File1_v0. doc 和 File1_v1. doc 分别压缩成 File_v0. rar 和 File_v1. rar,然后使用 rsync 算法对压缩后的文件进行同步更新,但结果显示对旧文件的重用率很不理想。

这种现象似乎令人难以置信,但是经过仔细的考虑和分析,我们发现它与文件压缩机制以及 rsync 算法的匹配检索原理有关:rsync 算法使用字节级匹配检索,这需要类似版本的文件内容。相似的性能导致文件存储字节的相似性。如果压缩了文件,尽管压缩后文件的两个版本的内容级别的相似性保持不变,但是由于压缩算法,存储字节级别的相似性大大降低了。这意味着这两个文件基本上不再与基于字节级匹配检索的 rsync 算法相关。

4 结 语

随着移动互联网的持续快速发展,移动云共享将逐步融入人们的生活。本文使用 C ++ 语言将 Linux 平台下的 rsync 移植到 Windows 平台上,并提出一种基于 rsync 的改进算法,该算法使快速文件同步和共享更适合于移动设备。当移动设备充当 C 端从云服务器下载并更新共享文件的最新版本(充当 S 端)时,本文改进的算法可以大大减少移动设备的计算,而不会影响快速 rsync 算法的同步效果。实验结果表明,在三轮同步更新的条件下,移动终端设备的计算量减少了约 50%,所需的网络流量也减少了约 14%,旧文件的重用率几乎没有变化。显然,随着同时更新轮数的增加,本文改进算法的效果将更加明显。

在实验测试中,我们还发现了一些新问题。例如,对于内容相似性基本上与存储字节的相似性无关的文件(例如. rar 和. exe),rsync 算法的快速文件同步效果非常不令人满意。在后续工作中,我们将探索如何解决. rar 和其他类型文件的快速同步和更新问题。此外,我们将继续研究如何降低 C 端(计算强度最高的三级匹配检索过程)的计算复杂度。

参 考 文 献

[1] Zhang J, Liu L, Wang K, et al. A cache-based pre-downloading scheme for HSR communication using C-RAN[C]// IET 8th International Conference on Wireless, Mobile & Multimedia Networks, 2019: 55 - 59.

[2] 张静,王少奎,郝希亮,等. 一种基于 Rsync 算法的改进型两轮同步算法[J]. 数据通信, 2017(1): 52 - 54.

[3] Matsuzawa K, Hayasaka M, Shinagawa T. Practical quick file server migration[J]. ACM Transactions on Storage (TOS), 2020, 16(2): 1 - 30.

[4] Yan H, Irmak U, Suel T. Algorithms for low-latency remote file synchronization[C]// IEEE INFOCOM 2008-The 27th Conference on Computer Communications, 2008: 156 - 160.

[5] Bolso E I. File synchronization with unison[J]. Linux Journal, 2005, 132: 6.

[6] Sheng Y, Xu D, Wang D. A two-phase differential synchronization algorithm for remote files[C]// 10th International Conference on Algorithms and Architectures for Parallel Processing, 2010.

[7] 徐旦,生拥宏,鞠大鹏,等. 高效的两轮远程文件快速同步算法[J]. 计算机科学与探索, 2011, 5(1): 38 - 49.

[8] 胡晓勤,卢正添,刘晓洁,等. 远程文件快速同步方法[J]. 电子科技大学学报, 2008, 37(4): 594 - 597.

(上接第 22 页)

[22] 单锦辉,张路,王金波,等. 实时嵌入式软件时间抽象状态机的扩展[J]. 北京大学学报(自然科学版), 2019, 55(2): 197 - 208.

[23] Huang S, Huang J, Dai J Q, et al. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis [C]//26th International Conference on Data Engineering Workshops, 2010.

[24] Abulnaja O A, Ikram M J, AI-Hashimi M, et al. Analyzing power and energy efficiency of Bitonic Mergesort based on performance evaluation[J]. IEEE Access, 2018, 6: 42757 - 42774.

[25] Hashmi J M, Chu C H, Chakraborty S, et al. FALCON-X: Zero-copy MPI derived datatype processing on modern CPU and GPU architectures[J]. Journal of Parallel and Distributed Computing, 2020, 144: 1 - 13.

[26] Valery O, Liu P F, Wu J. A collaborative CPU-GPU approach for deep learning on mobile devices[J]. Concurrency and Computation: Practice and Experience, 2019, 31(17): 5225.

[27] Tang X Y, Fu Z J. CPU-GPU utilization aware energy-efficient scheduling algorithm on heterogeneous computing systems[J]. IEEE Access, 2020, 8: 58948 - 58958.

[28] Cini N, Yalcin G. A methodology for comparing the reliability of GPU-based and CPU-based HPCs[J]. ACM Computing Surveys, 2020, 53(1): 1 - 33.

[29] Riahi A, Savadi A, Naghibzadeh M. Comparison of analytical and ML-based models for predicting CPU-GPU data transfer time[J]. Computing, 2020, 102(9): 2099 - 2116.

[30] Smirni E. Practical reliability analysis of GPGPUs in the wild: From systems to applications[C]//ACM International Conference on Performance Engineering, 2019.