International Journal of Advances in Electronics Engineering– IJAEE
Volume 6 : Issue 1 [ISSN : 2278-215X]
Publication Date: 18 April, 2016

FOTA Delta Size Reduction Using File Similarity Algorithm's

[Shyjumon N, Shivansh Gaur, Surendra Gadwal]

***Abstract*— In today's word, mobile devices are changing regularly. Thus the supporting software and applications needs to be regularly updated. Firmware over the air (FOTA) updates is a fresh technology for easy updation of mobile devices. However the FOTA delta size for updates is increasing considerably. Thus optimizing the network consumption for downloading the updates is a big challenge. Large FOTA delta size implies greater network bandwidth consumption while downloading the updates. This paper addresses this problem and aims to reduce the delta size for the FOTA updates. Various algorithms were implemented and their performance was analyzed. The analysis shows that the percentage reduction in delta size is reasonable.**

***Keywords*— FOTA, delta, fingerprints, chunking, hashing, simHash, TTTD.**

I. Introduction

Over-the-air (OTA) is the method of making data transfers or transactions wirelessly using the cellular network instead of a cable or other local connection. It refers various methods of distributing new software updates, configuration settings, and even updating encryption keys to devices like cell phones, set-top boxes or secures voice communication equipment. In the context of the mobile this OTA is called FOTA (Firmware-Over-The-Air). On modern mobile devices, an over-the-air update may refer simply to a software update that is distributed over Wi-Fi or mobile broadband using a function built into the operating system, with the "over-the-air" aspect referring to its use of wireless internet instead of requiring the user to connect the device to a computer via USB to perform the update. FOTA facilitates the following:

- Allows manufacturers to repair bugs
- Allows manufacturers to remotely install new software updates, features and services - even after a device has been purchased.

Firmware updates through FOTA involves use of tools that enable developers to identify the essential changes from an existing firmware version to a new, updated version and automatically create an extremely compact package (delta update) of the updated firmware. A delta update is an update that only requires the user to download the content that has been changed, not the whole new firmware version. In this work, we aim at generating lesser size delta by efficiently eliminating the redundant content to save network bandwidth and decrease the download time.

In this paper, we introduce a three stage approach, explained in Sec. Ⅲ, to find out the similarity between two files. To further reduce the delta size, we have put many constraints on files that should be matched, based on their size, parent directory, extension, etc. Hence, using the technique of file similarity algorithms along with the different constraints on the files based on the kind of data set we have, our system is able to reduce the delta size by a significant amount.

To evaluate the effectiveness of our system, we have conducted many experiments with different types of FOTA files. Compared with the existing tools, our system performs competitively when the content of two FOTA files differ significantly.

II. Related Work

There have been a large number methods dealing with the problem of finding similar files in a large collection of files using different kind of file similarity algorithms. Many of these existing systems are designed for a single collection of files and not for the two set of files where a file in one set shouldn't be compared with the files in the same set.

As in the domain of data deduplication, chunking algorithms can be mainly divided into two categories: Fix-Size Chunking (FSC) and Variable-Size Chunking (VSC). FSC is simple and faster. It breaks the input stream into fix-size chunks. FSC is used in rsync [1]. A major problem with FSC is the editing (i.e., insertion/deletion) even a single byte in a file will shift all chunk boundaries. A storage system Venti [2] also adopts FSC chunking for its simplicity. VSC derives chunks of variable size and addresses boundary-shifting problem by posing each chunk boundary only depending on the local data content.

VSC was first used to reduce the network traffic required for remote file synchronization. Spring et al. [3] adopts

Shyjumon N, Senior Development Manager
Advance Solutions Team, Samsung Research Institute Noida, Noida-201301, Uttar Pradesh, India

Shivansh Gaur, Engineer
Advance Solutions Team, Samsung Research Institute Noida, Noida-201301, Uttar Pradesh, India

Surendra Gadwal, CS Engineer
Former student from, Department of Computer Science & Engineering, Indian Institute of Technology Roorkee, Roorkee-247667, Uttarakhand, India

Border's [4] approach to devise the very first chunking algorithm. It aims at identifying redundant network traffic. Muthitacharoen et al. [5] presented a VSC based file system called LBFS which extends the chunking approach to eliminate data redundancy in low bandwidth networked file systems. You et al. [6] adopts VSC algorithm to reduce the data redundancy in an archival storage system. Some improvements to reduce the variation of chunk sizes for VSC algorithm are discussed in TTTD [7]. Recent research TAPER [8] and REBL [9] demonstrate how to combine VSC and delta encoding [10] for directory synchronization. Both schemes adopt a multi-tier protocol which uses VSC and delta encoding for multi-level redundancy detection. In fact, resemblance detection [4, 11] combined with delta encoding [12] is usually a more aggressive compression approach. However, finding similar or near identical files [13] is not an easy task. Comparison results on delta encoding and chunking are provided in [14, 15]. In this paper, we adopt the concept of TTTD discussed in [TTTD paper] along with the simHash discussed in [simHash paper] to identify near identical files.

III. Methodology

The proposed approach involves computation of similarity among files. Files which are similar to each other are patched during the formation of the delta. This section covers the algorithms used for computation of similarity among files.

Subsection *A* gives a brief overview about the methods used for similarity computation. Subsection *B* covers the implementation details of the algorithm.

A. *Overview*

The method for creating FOTA delta involves various stages as shown in figure 1. The source FOTA and the target FOTA are taken as input. The files in the two folders that match in name are picked up and their patch is created directly. This is done on assumption that the files that have same name will have a large amount of redundant content. The files with dissimilar name are processed to find the pair of files with highest similarity. The pair of files that exceed the similarity threshold are sent for patching, while the files that do not qualify the threshold constraint are sent as it is in the FOTA delta.

The proposed approach consists of three stages: **(1) Chunking** or **Feature Extraction, (2) Hashing, (3) Comparison**. The approach involves similarity matching, which is a process of finding pairs of files in a large collection of files that are similar to each other. As the amount of data an investigator has to deal with is growing rapidly, it is not possible to look at each file by hand. Hence a smaller representation of the file needs to be made.

Chunking: is a way of making a representative of a file of smaller size by breaking the file into a sequence of chunks. The well-known chunking approaches are fixed-length chunking and variable-length chunking (content-defined chunking). As names suggest, extracted chunks are of fixed size in fixed-length chunking and in variable-length chunking, chunk boundaries are determined by the local contents of the file.

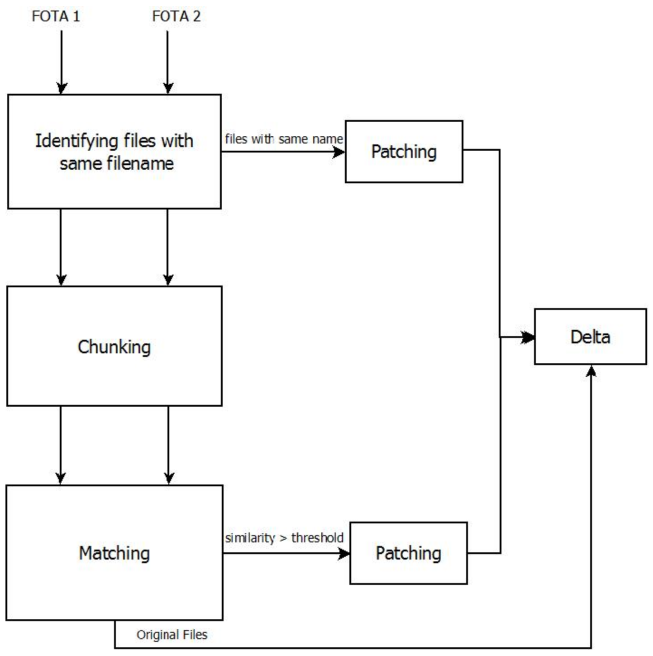

Figure 1: Overall algorithm pipeline

Hashing: Chunks or features selected from a large file needs to be matched from other files. Matching the chunks directly bit by bit is not convenient and will be complex. So we use similarity preserving hash functions. The chunks are hashed to an integer value using similarity preserving hash function. Hashing with similarity preserving hash function has the interesting property that hashes of similar features will be similar. Hashed chunks make comparison of files faster and easier.

Comparison: involves matching of set of hash values of files. Set intersection method is the most commonly used method for comparison of two files. The greater the intersection between the two sets, greater is the similarity of two files. The similarity score between two files is thus obtained.

B. *Implementation*

In this section we present implementation details of our proposed approach. We performed chunking of files using *TTTD* algorithm proposed by HP laboratory [16] at Palo Alto, California and to get break points of chunks and to get chunk-hash values, we used *simHash* proposed in [17].

TTTD algorithm picks chunks out of the original text using a variable window and divisors to find trigger points or break points. The break points mark the boundary of the chunk. It makes sure that the size of the chunk is neither very large nor very small. For this it uses thresholds. TTTD Algorithm uses four parameters, maximum threshold, minimum threshold, main divisor, and second divisor. We set their values to 200, 180, 540, and 270 respectively to get chunk size between 180 to 200 bytes. The maximum and minimum thresholds are used to eliminate very large-sized and very small-sized chunks. The main divisor can be used to make the chunk-size close to our

International Journal of Advances in Electronics Engineering– IJAEE
Volume 6 : Issue 1 [ISSN : 2278-215X]
Publication Date: 18 April, 2016

expected chunk-size. In usual, the value of the second divisor is half of the main divisor.

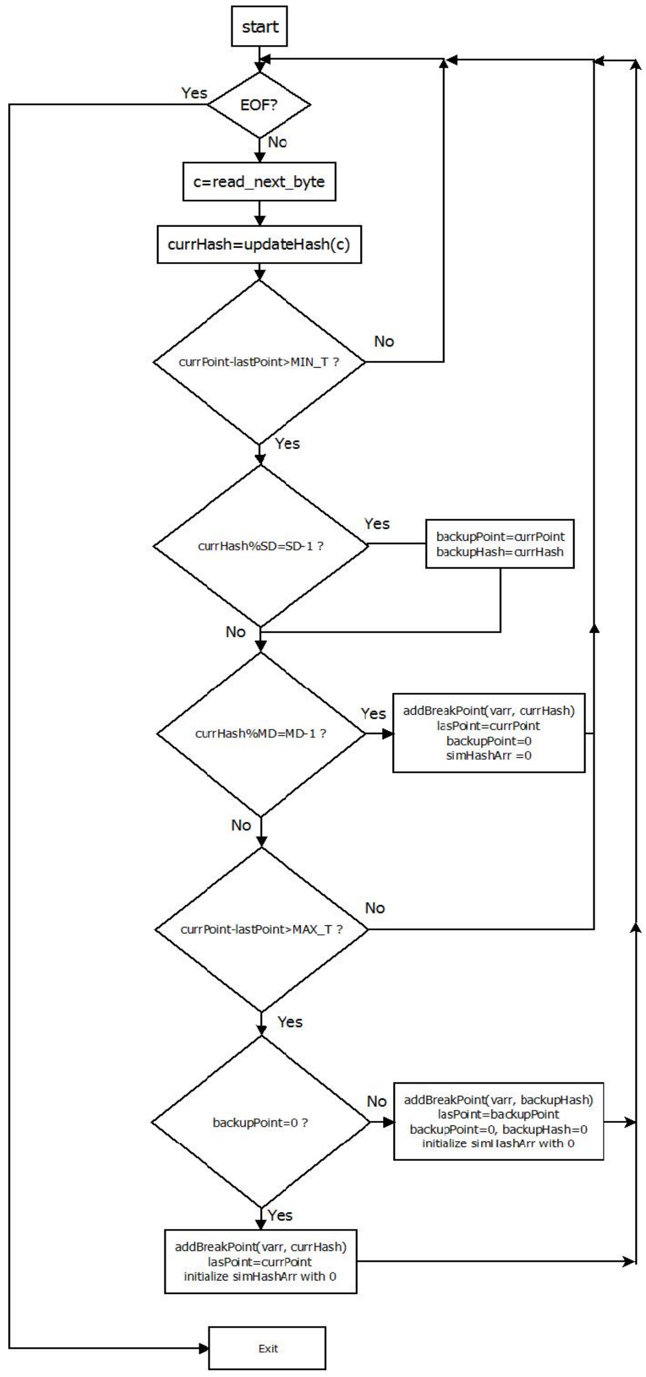

Figure 2: Block Diagram of TTTD Algorithm

Processing of each file for fingerprint calculation is done as shown in Fig. (2). Aalgorithm starts by taking the very first byte in the window and it hashes it using simHash. It adds subsequent bytes one at a time to the window and computes the hash value using simHash. If the size from last breakpoint to current position is larger than minimum threshold, it starts to determine the breakpoint by second and main divisors. Before reaching the maximum threshold, if it can find a breakpoint by main divisor, then uses it as the chunk boundary. The sliding window starts afresh at this position and repeats the computation and comparison until the end of file. But when the window reaches the maximum threshold, it uses the backup breakpoint (the very last one) if it finds any one, otherwise it uses the maximum threshold as a breakpoint. The new window now starts from this position and the computation is repeated.

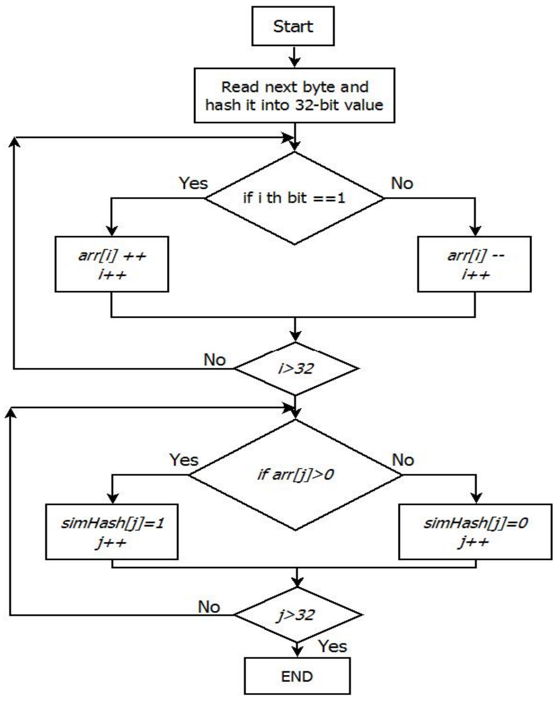

Figure 3: Block Diagram of SimHash Implementation

We used simHash for hashing purpose. SimHash is a similarity preserving hash function. In simHash very similar files map to very similar, or even the same, hash key, and distance between keys give measure of the difference between files. In simHash, integer valued hash keys of substrings are produced. As shown in figure 3, every character picked up by the window is mapped to its ASCII value and represented in binary form. We call this *hash* of a character. A 32-bit array is maintained and for each i^{th} set bit of the *hash*, *arr[i]* is incremented by one. Every time a character is added in the window this 32-bit array (*arr[]*) is updated. Finally the simHash value of the window is again a 32-bit number. Simhash bit *i* is 1 if *arr[i]* is positive, else it is 0.

For each chunk, its hash code is recorded. This way a set of simHash values for a file is obtained. This set matched with the simHash set of another file using set intersection method. Comparison by set intersection method involves sorting of the two sets of fingerprints of files and then matching their integer values linearly. Thus similarity scores are obtained.

IV. Complexity Analysis

In this section, we present the complexity analysis of the proposed approach- chunking part is linear in terms of the size of the text in each file. $O(C_{avg}.lg(C_{avg}))$ for sorting of fingerprints for each file before comparison, where C_{avg} is the average number of fingerprints selected from each file. Set intersection time is linear in average set size for each pair of files. There are several advantages of using this approach, like it performs better than the basic sliding window algorithm for chunking. We can control the average chunk size by changing the value of main divisor. Different tests can be performed with variable chunk sizes according to the data set we have. Along with these we have some limitation also, like size of a few percent of total chunks will be near to maximum threshold when main divisor or main divisor and second divisor are unable to find break point.

V. Results and Performance

The performance on given two firmware versions ranges widely depending on the average chunk size, which is a controllable parameter. In practice, we often start with a relatively small chunk size, e.g. 180-200 bytes, to get the better and true similarity scores. So this analysis will detect pairs of files that are similar due to shorter common sequences as well as larger common sequences.

A complete performance characterization with all implemented approaches is difficult to present. Results were compared with currently used tool at the Advance Software Lab, SRI-Noida, GOTA (Google-over-the-Air). We ran several tests for three FOTA pairs:

1) FOTA_I747MVLUFNE5_1718113_REV04_user_low_ship and FOTA_I747MVLUEMK5_2140838_REV04_user_low_ship shown as NE5 and MK5 in Table 2. FOTA delta size of this pair using GOTA engine was **408MB** (242MB patch files and 166MB original files).
2) (OXXCNG1 + XXUCNG1) and (VIMBNA1 + XXUBML4) shown as (NG1+NG1) and (NA1+ML4) in Table 2. FOTA delta size of this pair using GOTA engine was **565MB** (312MB patch files and 253MB original files).
3) FOTA_I9192DDUCNG1_2053775_REV01_user_low_ship_MULTI_CERT and FOTA_I9192DDUCNF6_2053775_REV01_user_low_ship_MULTI_CERT shown as NG1 and NF6 in Table 2. FOTA delta size of this pair using GOTA engine was **555KB** (496KB patch files and 59KB original files).

We used two different patching tools, ***bsdiff*** and ***xdelta*** to make patch or diff files from one source and one target file. In GOTA engine, they are using ***sbdiff*** patching tool that is a bit more efficient then above mentioned two tools. Here we present some performance measures:

Name of the FOTA pair	**TTTD+simHash results**			**Threshold value (in %)**	**Overall percentage reduction**
	Patch files size (in MB)	Original files size (in MB)	Total size (in MB)		
NE5 and MK5	243	53	296	50	27.45
NE5 and MK5	231	88	319	70	21.81
(NG1+NG1) and (NA1+ML4)	347	89	436	50	22.83
(NG1+NG1) and (NA1+ML4)	343	98	441	70	21.96
NG1 and NF6	1.8	0	1.8	50	-230
NG1 and NF6	1.8	0	1.8	70	-230

TABLE I . FOTA delta size and percentage reduction

As we can see in the Table (1), size of patched files is almost similar or slightly more than to the one we got from GOTA engine. So FOTA delta applying time on the device won't increase by an unacceptable factor.

These measurements were performed on a SAMSUNG NP300E5V-A06NG Core i3 Laptop with a 2.50 GHz Intel processor and 4 GB RAM. As results are shown, we got significant amount of reduction in FOTA deltas as compared to GOTA engine.

VI. Conclusions and Future Directions

Various file similarity algorithm were studied and implemented. For chunking Two Threshold Two Divisor (TTTD) algorithm, Rabin's fingerprinting algorithm and selective fingerprinting algorithms were implemented. With Rabin's fingerprinting, rolling hash function was used and with other chunking algorithms simHash was used. We applied the algorithms on different Firmware-over-the-Air (FOTA) update files, which gave us the similar files among the two folders. So the similar files were sent to a patch making tool to create a Delta between the two FOTA folders. The size of the delta folder created was considerably reduced. We performed extensive experimental comparisons and the performance for each method was evaluated. The results obtained were promising and there is a great scope for future improvements also.

Proposed approach for delta generation can further be improved by using alternative *Matching Algorithms*. One is to use **Bloom Filters** to store information about chunk-hash values of each file. A hash value can be in the range of *0* to $2^{32}-1$, so a bloom filter of size 2^{32} or a bloom filter of lesser size with *P* hash values will do the work. Comparison between two files will be faster using bloom filter than the set intersection but the implementation and handling of multiple bloom filters for each file will increase the complexity. This approach can also be integrated with Google's GOTA to keep device side work (applying patch) unchanged.

Acknowledgment

This work had been at Samsung R&D Institute Noida. We thank our colleagues, Mr. Ankit Singh and Mr. Mandeep Singh for helping us whenever we needed it. We also want to thank all System Memory team members for the support they gave us. We would also like to thank our organization for creating such an informative atmosphere and helping us in whatever way they could.

References

.

[1] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Technical report TR-CS-96-05, Deparment of Computer Science. 1996)

[2] Sean Quinlan, Sean Dorward. Venti: a New Approach to Archival Storage. In Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'02). 2002.

[3] Neil T. Spring, David Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'00). 2000, pp. 87-95.

[4] Broder, Andrei Z. On the resemblance and containment of documents. In Proc. of compression and complexity of sequences (SEQUENCES'97). 1997.

[5] A. Muthitacharoen, B. Chen, and D. Mazi`eres. A low-bandwidth network file system. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01). October 2001, pp. 174-187.

[6] Lawrence L. You, Kristal T. Pollack, Darrell D. E. Long. Deep Store: An Archival Storage System Architecture. In Proceedings of the 21st International Conference on Data Engineering. April 2005, pp. 804--815.

[7] K Eshghi, HK Tang. A Framework for analyzing and improving content-based chunking algorithms. Hewlett-Packard Labs Technical Report TR. 2005.

[8] Navendu Jain, Mike Dahlin, Renu Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica synchronization. In Proceedings of the 2005 USENIX Conference on File and Storage Technologies (FAST'05). 2005.

[9] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy Elimination within Large Collections of Files. In Proceedings of 2004 USENIX Technical Conference. 2004.

[10] Calicrates Policroniades, Ian Pratt. Alternatives for detecting redundancy in storage systems data. In Proceedings of the 2004 Usenix Conference. June 2004.

[11] Manber, U. Finding similar files in a large file system. In Proceedings of the USENIX Winter 1994 Technical Conference. January 1994, pp. 1-10.

[12] Fred Douglis, Arun Iyengar. Application-specific Delta-encoding via Resemblance Detection. In Proceedings of 2003 USENIX Technical Conference. 2003, pp. 113-126.

[13] Deepak R. Bobbarjung, Suresh Jagannathand and Cezary Dubnicki. Improving Duplicate Elimination in Storage Systems. ACM Transaction on Storage. 2006, Vol. 2, 4.

[14] Lawrence L. You, Christos Karamanolis. Evaluation of efficient archival storage techniques. In Proceedings of the 21st IEEE Symposium on Mass Storage Systems and Technologies (MSST). April 2004.

[15] Nagapramod Mandagere, Pin Zhou, Mark A Smith, Sandeep Uttamchandani. Demystifying data deduplication. In Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion. 2008.

[16] Eshghi, Kave; Tang, Hsiu Khuern, A framework for analyzing and improving content-based chunking algorithms. Tech. Rep. HPL-2005-30(R.1), Hewlett Packard Laboratories, Palo Alto, 2005.

[17] Bingfeng Pi, Shunkai Fu, Weilei Wang , and Song Han Roboo Inc., Suzhou, P.R.China, SimHash-based Effective and Efficient Detecting of Near-Duplicate Short Messages. In Proceedings of the Second Symposium International Computer Science and Computational Technology (ISCSCT '09) Huangshan, P. R. China, 26-28,Dec. 2009, pp. 020-025

About Authors:

Shyjumon N did his B.Tech in Electrical & Electronics Engineering from Govt. Engineering College, Thrissur, Kerala in the year of 2002. He worked with Various R&D centers such as Samsung, Toshiba, Cranes, Otwo in the field of Embedded Systems for the development of handheld devices. He is having more than 12 years of experience in the Linux Kernel and device drivers and also worked on different Bootloader Developments. He is specialized in BSP & Peripheral bus technologies such as SDIO, USB and presently mastering the System Memory area.

Shivansh Gaur dis his B.Tech in Computer Science and Engineering from Indian Institute of Technology, Roorkee. His research interests include artificial intelligence, Data Mining and Graphical Models, Cryptology and Information Security. He developed a music recognition application on JAVA that could identify a music track from small audio recording. He has been involved in development of applications for Windows phone also. Presently working with Samsung.

Surendra Gadwal did his B.Tech. from Indian Institute of Technology, Roorkee. His researches interests include image processing, Information & Network Security and Distributed Computing. In the field of image processing, he has developed an application for Detecting Texts from Natural Images. He is an active Android and Windows phone app developer. He is an Executive Member of IIT-Roorkee ACM Student Chapter.