January 2013

Delta Encoding Based Methods to Reduce the Size of Smartphone Application Updates

Nikolai Samteladze
University of South Florida, nikolay.samteladze@gmail.com

Delta Encoding Based Methods to Reduce the Size of Smartphone Application Updates

by

Nikolai Samteladze

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Kenneth J. Christensen, Ph.D.
Miguel A. Labrador, Ph.D.
Jay Ligatti, Ph.D.

Date of Approval:
April 1, 2013

Keywords: Compression, Traffic Reduction, Performance Evaluation, User Study, Savings Estimation

ACKNOWLEDGMENTS

I would like to thank my graduate adviser, Dr. Kenneth J. Christensen, for his guidance, support, and mentorship during the last two years. Working with him was a great learning experience for me and taught me a lot.

I am grateful for all the support I have received from my family and especially my mother throughout my studies at the University of South Florida.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In 2012 the two biggest smartphone application markets – the Google Play store and the Apple App Store – each had close to 700 thousand applications with approximately 2 billion downloads happening every month. The introduction of new features and correction of bugs and security vulnerabilities make it usual for mobile application developers to release new version of an application every month. Combined with the great smartphone popularity, it leads to approximately 400 PB annual traffic generated by app updates in the U.S. wireless networks alone. Being partially transmitted through cellular networks, mobile application updates traffic accounts to up to 20% of the annual cellular traffic in the U.S. This thesis presents delta encoding based techniques that significantly reduce update traffic by transferring only the changes (or patches) between two versions of an application. Such network bandwidth reduction enables savings for smartphone users, mobile operators, and data centers that serve app updates.

Two Android application update methods – called DELTA and DELTA++ - were developed, implemented, and evaluated. Both methods use delta encoding to transfer only the changes between application versions. DELTA++ improves on DELTA by exploiting the internal structure of APK packages, which are used to distribute Android applications. The APK file can be seen as a compressed archive of all the files contained in application. The DELTA++ algorithm unpackages APK and computes differences between decompressed application files, which allows it to produce much smaller

patches. Our experimental results show that DELTA++ reduces app update size by 77% on average. DELTA++ patches are twice smaller than those produced by the Google Smart Application Update method, which is currently used in the Google Play store. This reduction has a trade-off – increased complexity of generated patches makes patch deployment process more sophisticated. Consequently, more time has to be spent to apply the received patch in smartphone. Such delay can be considered acceptable as application update is a delay-tolerant process and smartphone users do not need an update immediately after its release.

In order to estimate how much savings can be achieved with DELTA++, a study of Android smartphone users was conducted. The results show that if DELTA++ is used in Google Play instead of the Google Smart Application Update method, then 32 PB or 1.7% of annual traffic can be saved every year in cellular networks in the U.S. The Apple App Store currently does not use any method based on delta encoding to reduce application updates traffic. Usage of methods similar to DELTA++ in the App Store can further increase the savings up to the 12% of yearly cellular traffic in the U.S., which equals to more than $2 billion cost savings a year.

CHAPTER 1

INTRODUCTION

1.1 Motivation

In late 2012 more than 114 million people owned a smartphone in the U.S., which is approximately half of all mobile device users in America [1]. Such popularity has prompted massive development of various smartphone applications. By the end of 2012 the two biggest application markets – the Google Play store (distributes applications for Android operating system) and the Apple App Store (distributes applications for iOS operating system installed on iPhone) – both had more than 700 thousand applications with approximately 2 billion downloads happening each month [2, 3].

The existence of a sole application market for each operating system and the availability of low-cost high-speed wireless networks provided an opportunity to update mobile applications more frequently than desktop software. Figure 1 shows a system-level view of application updating for smartphones showing an application market with its data centers that serve application updates, wireless networks that transport information between data center and end-user, and user smartphones. The opportunity to deliver updates easily to all the application users is used by developers to introduce new features, add new content, or fix bugs and security vulnerabilities. It is common for a popular application to have updates released every month (see Section 2.5.1). Frequent application updates increase smartphone users download traffic, which is usually limited

by the mobile operator. These updates also increase traffic in wireless networks, and outgoing traffic in data centers that serve app updates.

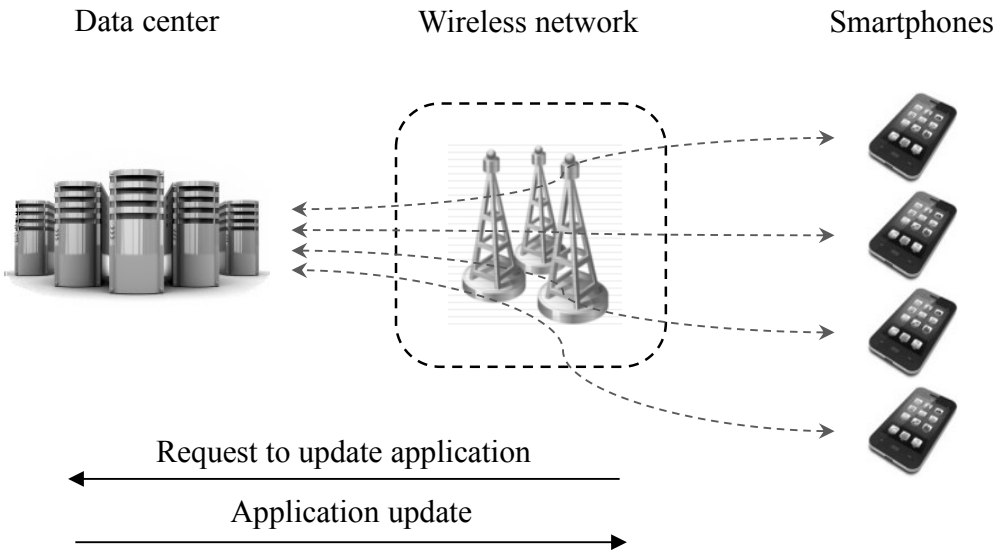

Figure 1. System View of Application Updating

The current amount of wireless traffic and its growth rate is a big challenge for data centers and mobile operators in the U.S. Mobile operators, for example, constantly spend billions of dollars to upgrade and expand their networks in order to keep up with the increasing amount of traffic [5, 6]. The majority of IT companies agree [7] that the growing demand for network bandwidth is the most serious challenge for their data centers, half of which are currently being updated to improve I/O performance. With the fast growing popularity of smartphones, mobile application updates significantly increase wireless traffic, which is already hard to manage. The question arises, is there a better way to update mobile applications, which would reduce the traffic generated by app updates?

Smartphone applications are mostly updated incrementally – by extending functionality, adding new content, fixing existing problems, etc. Thus, the significant part

of the application remains unaffected during an update. Delta encoding [15] is a technique that is used to compute the difference, or patch, between two files. This patch is then can be used to construct the newer file version from the old one. Delta encoding can be used to shrink the size of an application update by transmitting only the changes between the old application version and the new one.

This thesis focuses on Android operating system and its applications distributed through the Google Play store. Two delta encoding based methods – called DELTA and DELTA++ – are presented. They significantly decrease the traffic generated by application updates and enable savings for mobile operators, data centers, and smartphone users.

1.2 Problem Statement

The primary purpose of developing a better method to update mobile applications is to reduce the traffic generated by updates. However, it is not enough to shrink the update size as much as possible. Any savings achieved by reduction of traffic can become negligible due to the costly changes that need to be made in the current infrastructure or the software used to distribute application updates. Thus, it is very important to develop new updating methods that can be easily implemented on top of the existing infrastructure and with minimum changes in the software used.

Along with low implementation costs, a better updating method should not exacerbate user experience both for smartphone users and application developers as it can cause their transition to another application market or mobile operating system. Thus, any new method to update smartphone applications should not require any additional effort

from application developers or affect the way users update applications installed on their smartphones.

1.3 Contributions

The primary contributions of this thesis are as follows:

- Design, implementation and evaluation of DELTA – a method that uses delta encoding to update smartphone applications. This work was presented at IEEE LCN 2012 [11] and shows that an updating approach based on delta encoding can be successfully used to reduce the traffic generated by application updates.
- Design, implementation and evaluation of DELTA++ updating method that improves on DELTA and further reduces the size of application update. DELTA++ is compared with the state-of-the-art Smart Application Update method [9], which is used to update Android applications distributed through the Google Play store. This work was submitted to IEEE Internet Computing magazine [12].
- A study using feedback from volunteer Android users that allows to estimate large-scale savings that can be achieved with a full-scale deployment of DELTA++.
- A study of the Google Play store and the Apple App Store that provides valuable statistics about the most popular free applications.
- A first order estimate of savings that can be achieved with full deployment of DELTA++ and similar methods in the Google Play store and the Apple App Store.

1.4 Outline

The remainder of the manuscript is organized as follows:

- Chapter 2 discusses the literature related to this thesis.
- Chapter 3 describes the DELTA and DELTA++ methods to update Android applications and implementation of the latter in details.
- Chapter 4 presents the study of Android smartphone users. This study was conducted to characterize how users update applications installed on their smartphones. The obtained statistics allows estimation of the potential savings from DELTA++ in terms of reduced traffic.
- Chapter 5 shows experimental results of the proposed methods and comparison of their performance to Google's Smart Application Update.
- Chapter 6 presents a first order estimate of the savings that can be achieved with DELTA++ deployment.
- Chapter 7 concludes and discusses possible future work.

CHAPTER 2

RELATED WORK

This work is related to the literature on network traffic reduction. Constantly improving computer performance leads to the growth of the amounts of processed data. Consequently, the size of data that needs to be transferred between different machines also increases. Thus, it is always important to develop methods and algorithms that transfer the same information faster and use less network bandwidth.

2.1 Taxonomy of the Traffic Reduction Approaches

Traffic reduction methods can be divided into 3 categories:

1) *Traffic compression*. These methods process the transferred data and to decrease its size by eliminating redundancy in it.
2) *Traffic offloading*. This category includes methods that use less expensive mediums to partially transfer data.
3) *No Transmission.* This category includes methods that eliminate the necessity to transfer data where possible.

Traffic compression methods can be further divided in 3 sub categories:

1) *Lossless compression*. These methods propose different algorithms that can decrease the data size by eliminating the redundancy inside of it in a way that allows reconstruction the original from the compressed data. More CPU resources are spent in an attempt to save network bandwidth.

2) *Lossy compression*. These methods decrease the size of data by disregarding some of its parts, which makes it impossible to reconstruct the original from the compressed data. Usually, quality is traded to save network bandwidth.

3) *Delta Encoding*. This subcategory contains methods that eliminate redundancy between files and attempt to transfer only the difference between current files and the files that have been already transferred. Network bandwidth is saved by using more CPU resources and more storage space.

Traffic compression methods can be further divided in 3 sub categories:

1) *Caching*. These methods determine what data will be needed in the future and store it thus eliminating the necessity to transmit it again. Storage on the receiving side is traded to the network bandwidth.

2) *Reduction of Control Traffic*. This subcategory includes methods that try to reduce the amount of control traffic where possible, i.e., by aggregating multiple transfers and transmitting them all at once. It is assumed that transmission can be delayed in order to save network bandwidth.

Figure 2 shows the taxonomy of the traffic reduction approaches in a form of a tree. This thesis presents methods that use delta encoding to reduce traffic. Thus, literature on various delta encoding algorithms and their usage for traffic reduction will be a focus of this chapter and will be overviewed in Section 2.2. Section 2.3 summarizes all other traffic reduction approaches.

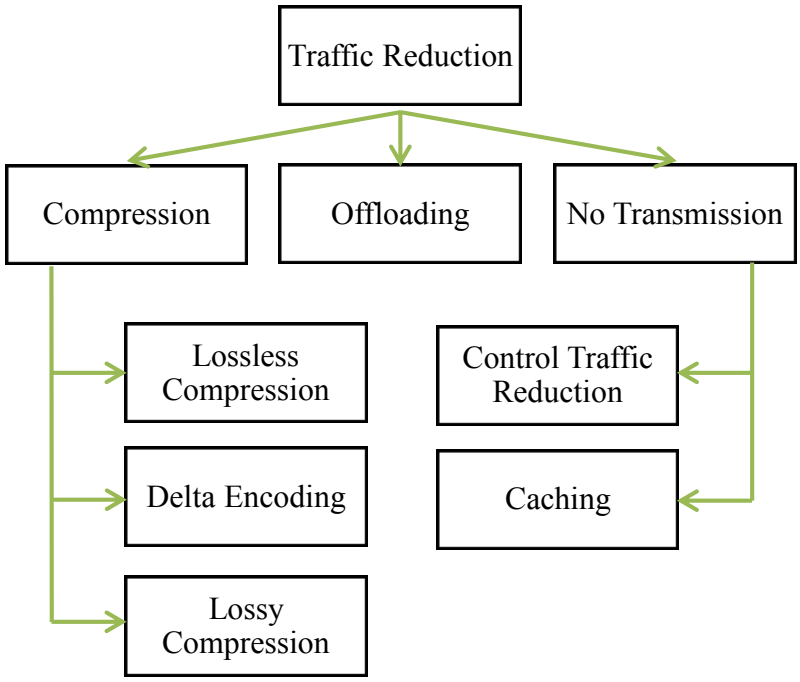

Figure 2. Taxonomy of the Traffic Reduction Approaches

2.2 Delta Encoding

Delta encoding is an approach to storing or transferring data using calculated differences between files rather than the files themselves. Such difference data, which will be referred to as a patch, can be used to construct the target file from the source file. There are three terms that are widely used to describe this approach – delta differencing, delta compression, and delta encoding. Within this manuscript, these terms are used as follows:

- *Delta encoding* generally defines a broad field of algorithms that computes a difference between the target file and the reference file. Output is a patch that can be used to construct the target file from the reference file if the last one is available. It is worth noting that though most of the algorithms use a single file as a reference, multiple reference files can be also used.

- *Delta differencing* describes algorithms that compute a difference between files as a set of instructions on how to construct the target file from the reference file. This set of instructions is contained in the resulting patch.
- *Delta compression* covers algorithms that construct a patch in two steps. First, a difference between files is computed as a set of instructions on how to construct the target file from the reference file. Second, the computed instruction set is compressed using some compression algorithms. Thus, the patch in this case contains a compressed set of instructions rather than instructions themselves.

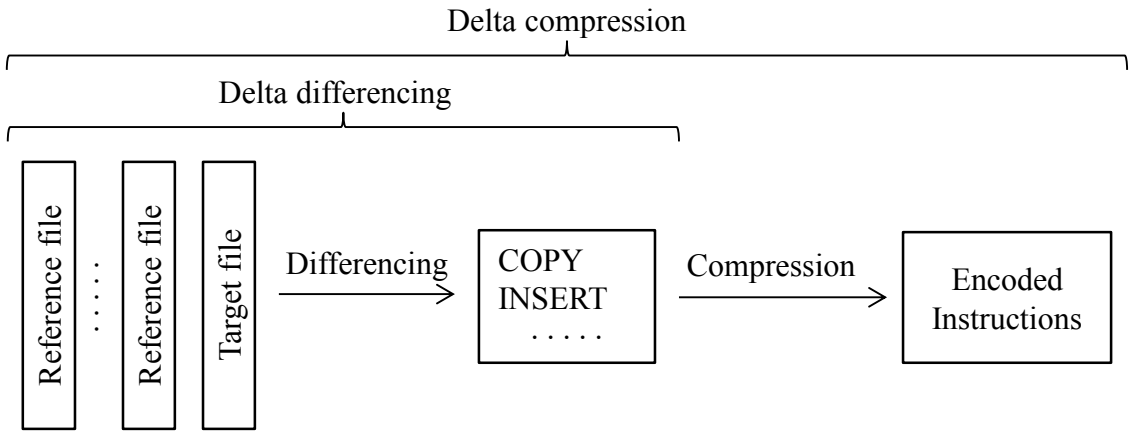

Figure 3. Distinction between Delta Differencing and Delta Compression

Figure 3 shows the distinction between delta differencing and delta compression. The result of delta differencing is a set of instruction on how to construct the target file from the reference files. Most of the algorithms use COPY and INSERT instructions. Delta compression includes delta differencing, but also adds an additional step during which the computed instruction set is compressed. Both not encoded and encoded instruction sets can be used as a patch to construct the target file from the reference files.

Delta encoding is used as a general term and refers to both delta differencing and delta compression.

2.2.1 Delta Encoding Algorithms

Delta encoding algorithms aim to compute a difference between files and efficiently encode this difference in a way that allows using the resulting patch to construct the target file from the reference file. File differencing problem arises from the string-to-string correction problem, first studied by Levenshtein in 1965 [20]. Many different algorithms have been developed since then. These algorithms mainly differ in the used data structures, file comparison algorithms, and difference encoding methods. As a result, they have different running time, space usage, and compression rate.

The delta differencing algorithm was first proposed and implemented in 1976 in UNIX diff tool [15], which was used to efficiently store multiple versions of text documents. The diff tool uses a dynamic programming based algorithm [24] to find the longest common subsequence in two text files on a line-by-line basis. Difference between two files contains a minimal set of lines that must be changed to construct the newer file from the old one. Thus, the resulting patch has a human-readable format. It is worth noting that the diff algorithm, unlike the majority of delta encoding algorithms after it, does not use any compression to reduce the constructed patch size.

Later delta encoding algorithms started to use techniques similar to the Ziv-Lempel compression algorithm [28], which allowed them to achieve significantly better compression rates. Bdiff was developed in 1986 [45] and used a modified Tichy's block-move algorithm [23], which works with file's blocks instead of lines, and thus can be efficiently used not only for text files. Bdiff can be considered a delta compression

algorithm as it first computes the difference between files as a set of COPY and INSERT instructions and then encodes them. At the first stage, the longest match is found using a greedy algorithm with a suffix tree to speed up the blocks look up. During the second phase, copy blocks are encoded using their offsets and lengths. Then all the instructions are compressed using a common splay tree algorithm [32].

Vdelta algorithm, which was described in [17], is also a modification of the Tichy's block-move algorithm and uses a greedy algorithm to match file blocks. Hash tables are used to look up blocks during the matching process instead of suffix trees used in bdiff. Block matching is done both between the target file and the reference file and within the target file itself, which allows vdelta to eliminate redundancy inside the target file as well. Also, the rules of the greedy matching algorithm are relaxed, which decreases vdelta's running time and memory usage, but it lessens its compression rate. Difference is represented as ADD and COPY instructions, which are then encoded by using an internal binary format. Two caches are used during encoding and decoding processes to minimize the memory required to store the address information for each instruction. Vdelta became the basis for the general vcdiff delta encoding format [18]. An approach similar to vdelta's was used in zdelta [19], which built up on the zlib [27] compression library, and in xdelta presented as a part of XDFS file system [22]. Both zdelta and xdelta are based on vcdiff format. Another widely used delta encoding format is gdiff described in [14].

Baker et al. in [16] noted that size of the delta differences computed between binary files can be significantly reduced by taking into the account platform-dependent structure of executables. Binary files contain compiled code that has numerous data

pointers. Even small changes in the source code cause modification of all the following pointers and, thus, result in significant changes in the compiled binary. The authors presented the exediff algorithm that consists of two main steps: preliminary matching and value recovery. Pre-matching finds matched blocks in two files but does not include in the patch changes that are likely to be caused by the compilation process. Such changes are predicted during the value recovery step. Predictions are then evaluated; changed bytes are stored explicitly if the prediction is inaccurate. During the patch deployment, the target file is constructed by copying the unmatched blocks and then by recovering the secondary changes using the predictions.

The bsdiff algorithm used in this thesis was described in [13]. It also exploits the structure of executables as exediff, but it does not require disassembling and, thus, is platform-independent. The author of the bsdiff algorithm observes that if two blocks in binary files were compiled from the same source code, then the bytewise difference between them will be mostly zero, although it will be difficult to find a long common sequence of bytes in them. Such sparse difference is highly compressible. The bzip2 [26] algorithm is used to compress the constructed patch, which in size is slightly greater than the size of the patches generated by platform-dependent exediff.

Table 1 lists delta encoding algorithms overviewed in Section 2.2.1 and the key papers where these algorithms were presented.

Table 1. Significant Delta Encoding Algorithms

Algorithm	Year	Key Work
Diff	1976	[15], [24]
Bdiff	1985	[45], [23]
Vdelta	1996	[17], [18]
Gdiff	1997	[14]
Exediff	1999	[16]
Xdelta	2000	[22]
Zdelta	2002	[19]
Bsdiff	2003	[13], [21]

2.2.2 Delta Encoding to Reduce Traffic

Delta encoding has been successfully used to reduce the amount of traffic in various systems including the distribution of software update, internet browsing, file systems, and versioning systems. The remainder of this sections overviews the most significant work on usage of delta encoding methods to reduce traffic. Such work can be divided into 3 broad categories:

- Software Updating
- HTTP Communications
- File Storage and Replication

Currently, many popular software is updated using delta encoding that fits naturally in the updating process as the main goal of it is to replace the old version with the new one, which is usually very similar.

Microsoft created the Binary Delta Compression method [36] and then the Delta Compression API [35] to distribute compact updates for its various Windows operating systems. Binary Delta Compression was first used in 1998 with Windows NT 4.0 Service

Pack 4 and was extended to all updates downloaded from the Windows Update web site in 2003. Delta compression API was presented in 2009 and allows any developer to benefit from delta encoding. It also offers special handling of Portable Executable files, which leads to smaller deltas between them. The mobile operating system manufactures update software on the end-user devices using Over-the-air (OTA) wireless downloads [38], which can significantly reduce wireless traffic, which is usually limited and expensive for users.

Internet browser is another widely used software that constantly needs to respond to new security threats and to improve in order to keep up with the increasing capabilities of modern Internet. Google developed Courgette [37] for compressing patches for its Chrome browser. The Courgette method uses delta encoding to transmit only changes that were made in the browser and not a full new version of Google Chrome. It decreases the size of a patch by taking into account the platform-dependent structure of a compiled application file in a similar way to the exediff algorithm, proposed by Baker et al. in [16]. Executable files contain a lot of internal references, which completely change their values from even small changes in the source code. Courgette uses disassembling to find all the internal pointers and recover the values of these. This optimization allows Courgette to achieve a 10 times smaller patch size for the Google Chrome browser than that of bsdiff.

Wireless sensor networks recently have gained much popularity. Due to the limited resources, these networks need an efficient mechanism to distribute updates. Many methods based on various delta encoding algorithms have been proposed. Stolikj et al. [49] overviewed the most wide-spread approaches to efficiently reprogram a

wireless sensor network and concluded that vcdiff and vsdiff algorithms provide the best overall performance.

A lot of work has been presented on usage of delta encoding to reduce the traffic and latency of HTTP-based communications. Housel et al. [39] first proposed to use delta encoding – called form differencing – in their WebExpress system. It uses a client/intercept model and does not require any changes in server or client. Two components – or interceptors – are installed: one on the client's side and one on the server's side. These components perform different optimizations to reduce traffic and latency. Such optimizations include caching, delta encoding, and eliminating redundant HTTP control traffic. Caches are maintained both in the Client Side Interceptor (CSI) and Server Side Interceptor (SSI), which enables significant savings when previously transferred files are requested. WebExpress uses delta encoding to transmit an updated version of a file that is already cached both by CSI and SSI. The authors observe that information from the same application server (e.g. stock queries replies) is usually very similar. Thus, if CSI and SSI both already have old information in the cache, then SSI can transmit the updated information as a difference between the old and the new data. In order to reduce the control traffic, WebExpress uses a single connection between SSI and CSI, which allows it to avoid a connection establishment overhead. SSI also maintains information about the client browser, which does not change within one connection between CSI and its SSI. This information is sent by CSI only once, which further decreases the amount of control traffic. Housel et al. experimented with two application servers and concluded that the proposed WebExpress system enables 60% to 99% traffic reduction and 37% to 97% latency reduction due to the usage of caches in SSI and CSI.

Mogul et al. published a thorough study [40] of the potential benefits from delta encoding in HTTP communications. The authors collected and analyzed traces from two real-world web sites and used diff, vdelta and gzip to reduce traffic. As in WebExpress system, the delta difference was computed between the new version of file and the old one, which both the server and the client had in cache. During the experiments, vdelta significantly outperformed diff and enabled approximately 85% savings in transmitted bytes for the cached files. Overall traffic reduction for the 2 studied web sites was 31% and 9%. The authors also compared vdelta delta encoding and gzip compression by conducting a simple experiment where several plain-text files were sent multiple times. The results suggest that gzip compression works better than vdelta in terms of saved bandwidth when small files are sent. In the case of large files, vdelta provides a better compression rate than gzip. Mogul et al. also published RFC 3229 [44] that proposes a compatible extension to HTTP/1.1 protocol that allows the server to send delta encoded responses if the client supports particular delta encoding algorithms.

Chan et al. [41] explored the idea of using multiple files as references when the target file is encoded. The proposed method is the called cache-based compaction and includes an algorithm to select reference files and an encoding/decoding algorithm to compress the target file based on the selected references. The authors presented a heuristic based on the URL structure to determine the similarity between files. It explores the idea that Web files, which can be considered close to each other in the hierarchy formed by their URLs, have more similarities than files that are farther from each other in such hierarchy. The proposed encoding/decoding algorithm is similar to gzip compression and is based on the Liv-Zempel compression algorithm. Chan et al.

presented experimental results to justify that the proposed selection algorithm works and to evaluate the savings that can be achieved with deployment of their cache-based compaction method. Experiments are done using only text objects because graphical objects are already compressed and do not have many non-trivial similarities between them. The proposed method – called npact(n), where n is the number of reference files – is compared to the gzip compression and the diff delta encoding. The authors use the npact(3) version, which takes 3 files as references, as it proved to be the most efficient in terms of compression rate and compression time. The results show that npact(3) provides a much better compression rate than gzip and diff. None of the experiments compared npact to any of the modern delta compression algorithms (e.g. vdelta), which also proved to be much better than gzip and diff [17].

Spring et al. [43] showed that there is redundancy not only between Web caches but also in streaming media, dynamically generated content, and other non-cached traffic. The authors studied similarities within one file and between files and concluded that there is 50% redundancy within dynamically generated Web documents (such as CGI and search results), while the redundancy among the documents that servers are prohibited to cache is 76%. Experimental results showed that for different protocols from 4% to 34% bytes can be saved.

Butler et al. [42] from Google proposed Shared Dictionary Compression over HTTP (SDCH) – an HTTP/1.1-compatible extension that also aims to eliminate redundancy between dynamically generated content (e.g. search results) and is used in Google Chrome browser. The authors proposed compressing data using a dictionary that is shared between the server and the client. Such a dictionary is downloaded by the user's

browser from the server and contains blocks that are likely to appear in later responses from the server. The server can then code repeated blocks in its responses using references to the shared dictionary, which are smaller than the blocks themselves. After receiving an encoded response, the client can decode it using the downloaded dictionary. The delta encoding algorithm used in SDCH is designed based on the vcdiff format. HTTP format is extended by adding a sdch keyword to the Accept-Encoding header of the HTTP request and the Get-Dictionary header to the HTTP responses. The Get-Dictionary header points to the URL where a shared dictionary can be obtained. In current implementation SDCH cannot eliminate redundancy between Javascript and CSS files, but Butler et al. point to it as a future work direction.

Delta encoding can be used not only to reduce network traffic, but also to save storage space when multiple file versions are stored. The Revision Control System [45] created in 1984 uses a line-based delta encoding algorithm to store only the changes between versions rather than file versions themselves. The most recent file version is stored intact. All the old versions are stored as reverse patches. Previous version of a file can be retrieved by applying the appropriate patch to the successor version. This implementation makes the retrieval of the latest version simple and fast. The presented usage statistics show that in 95% of all cases the most recent version is needed. MacDonald presented the Xdelta File System (XDFS) [22], which uses several delta encoding algorithms with various performance to compute changes between files. XDFS is not solely a version control system, but can be used as one. The experimental results show that although XDFS is slower that RCS when only a few versions per file are stored, it significantly outperforms RCS when there are a sufficient number of versions

per file. Modern versioning systems such as Github [62] also benefit from usage of delta encoding.

Different network file systems [47, 48] use delta encoding algorithms for file replication, which allows them to reduce network traffic and to use available bandwidth more efficiently. Modern cloud-based personal storage software such as Dropbox [46] also uses delta encoding algorithms to update files in the cloud and on the user's machine.

Table 2 categorizes the key work on usage of delta encoding for traffic and occupied storage space reduction.

Table 2. Key Work on Usage of Delta Encoding

Category	Subcategory	Key Work
Software Updating	Operating Systems	[35], [36]
	Internet Browsers	[37]
	Wireless Mobile Updates	[9], [38]
	Wireless Sensor Networks	[49]
HTTP Communications		[39], [40], [43]
File Storage and Replication	Versioning Systems	[22], [45]
	Network File Systems	[47], [48]

2.3 Other Methods of Traffic Reduction

2.3.1 Lossless Compression

Various lossless compression algorithms have been developed [26-29] to eliminate redundant information within a single file. These algorithms rely on the assumption that files contain sufficient amount of repetitive information that can be encoded in an efficient manner. Such encoding reduces the file size and also allows

future reconstruction of the original file from the compressed file. MPEG video codec [31] and JPEG image format [30] use lossless compression algorithms to efficiently store and transmit media content when no information can be disregarded. Limited resources of wireless sensor networks also facilitate usage of compression during transmissions [31] because data processing consumes much less power than data transmission.

2.3.2 Lossy Compression

Lossy compression explores the idea that some data can be disregarded if the changes will not be noticed by the user. Real-time internet video relies on bandwidth-scalable lossy compression algorithms to enable low-latency video streaming by sacrificing video quality when bandwidth is scarce [34].

Fox et al. proposed [53] to use lossy compression, which they referred to as distillation, in an HTTP proxy to save network bandwidth. The authors' implementation – called Pythia – uses real-time lossy compression of images and PostScript documents to reduce network traffic. Pythia also implements a refinement algorithm that allows it increasing the reduced quality of media content by user request. Although Fox et al. reported a 5 times increase in the Web browsing speed based on the Pythia users feedback, no experiments were conducted to quantify the bandwidth savings.

Screens of smartphones and other mobile devices are limited in size and display capabilities. Thus, bandwidth can be saved by transmitting only the information that the end-device can display. Lossy compression of media content in proxy servers is used by Amazon [54] to reduce traffic in Amazon Silk browser installed on the Kindle Fire tablets. In 2012 Microsoft introduced Data Sense technology [52] that uses in proxy compression to save traffic for Windows 8 smartphones. Lossy compression of media

content (e.g. images) is used together with lossless compression of other web content such as JavaScript and text. When Microsoft Data Sense is used, the requested web address is sent to both the intended website and to the Microsoft Data Sense optimization server. Browser receives information from both the website and the optimization server and then decides where to fetch the requested page in order to minimize data usage. It is worth noting that only HTTP traffic is optimized; web addresses accessed using HTTPS protocol are not sent to the optimization server and the corresponding web pages are fetched directly from the intended website. A similar technology is used in the Google Chrome for Android browser [50].

2.3.3 No Transmission

In some cases, network bandwidth can be saved by eliminating the necessity to transfer some data. Probably the most popular approach is caching that is widely used to store files, which are likely to be needed again in the future in order to reduce network traffic [61]. The amount of stored data depends on the cache capacity. Thus, storage is traded to save network bandwidth. It was shown [59] that caches can significantly reduce traffic and decrease latency. However, it was pointed [60] that caches can provide only limited savings.

Another approach to eliminate data transmission is to reduce the amount of control traffic in network. Falaki et al. published a study [57] of 35 Android smartphones and 8 Windows smartphones, which showed that in the case of mobile networks traffic is dominated by small size packets. The authors reported that half of all smartphone transmissions headers accounted for 12% of the total traffic. If the SSL transport security is used to transfer a packet, control traffic overhead is 40% for half of the captured

transfers. Falaki et al. proposed to aggregate small transfers in a cloud proxy server and then transmit them all at once to the end-user device in order to reduce the control traffic overhead. However, no experiments were conducted to measure the possible savings. It is also worth noting that the proposed approach can work only for delay-tolerant transmissions.

2.3.4 Traffic Offloading

In the case when multiple networks are available, traffic can be partially offloaded to the network with the lowest transmission cost, but software usually must be changed to include network switching and load dispatching algorithms. Balasubramania et al. [55] studied Wi-Fi offloading of mobile 3G traffic. They created a system – called Wiffler – that implements a fast switching algorithm and uses a simple model of environment to predict future Wi-Fi connectivity. Based on the predictions, Wiffler delays some data transfers in order to transmit them over Wi-Fi. Evaluation was done using traces from transit buses in 3 cities. The results show that 45% traffic reduction can be achieved for applications that can tolerate 60 seconds of delay.

Different results were presented by Lee et al. in [56]. They studied 100 iPhone smartphones owned by people in major cities in South Korea over an 18 day period in 2010. The results showed that with a 100 second delay only 2-3% traffic can be offloaded to Wi-Fi. But for applications that can tolerate a delay of 1 hour and longer, 29% traffic reduction can be achieved. This work analyzes traces that are closer to the real-world smartphone usage than data collected from transit buses in [55].

Yap et al. [58] explored an idea of using multiple networks simultaneously to increase network throughput, improve connectivity, and reduce transmission cost. They

created an Android implementation that can connect to multiple networks at a time and use them to transmit data. Using 10 available networks simultaneously, the authors were able to achieve more than 3 times throughput of the fastest network.

2.4 Distribution of Android Applications

2.4.1 Application APK Package

The Android application package file (APK) is the file format used to distribute and install applications in Google's Android operating system. APK files are based on the JAR file format used in Java applications. An APK is essentially a ZIP archive that contains all parts of an Android application such as program byte code, resources, assets, certificates, and manifest file. The APK file is created by compiling the application's code and resources and compressing all of its files into one package. An APK package usually contains the following:

- META-INF directory contains the application's manifest (MANIFEST.MF), certificate (CERT.RSA) and a list of resources (CERT.SF). The manifest file lists all the files included in APK packages together with their checksums (SHA-1 digest is used).
- classes.dex is the compiled application's code in .dex file format, which was designed specifically for the Android operating system and is usually two times smaller than a .jar (Java Archive) file derived from the same code (such savings are partially enabled by using shared strings pools).
- lib directory includes the processor-specific compiled code.
- resources.arsc contains compiled application resources (e.g., XML files).

- res is a directory with all the application's resources that cannot be compiled into resources.arsc (e.g., icons or pictures used in the application).
- AndroidManifest.xml file serves as an additional Android manifest file and contains information about the distributed application. This includes the application's name, version, and referenced third-party libraries, and required access rights.

2.4.2 Updating Android Applications

The easiest way to update a smartphone application is to send the latest application version to the end device where it will be installed instead of the previous application version. One advantage of such application updating method is its simplicity – only the latest application version needs to be stored in a data center and no extra operations are required to install it on a smartphone once it is received. A major drawback is that even small changes in an application cause transmission of the full new application version, which in size in some cases can reach hundreds of megabytes (e.g. for games). This simple method to update application was used in the Google Play store until late 2012 and is used in the Apple App Store today.

At the Google I/O developer conference held in June 2012, Google announced that its Smart Application Update technology [9] had been introduced to the Google Play store and would be seamlessly used to update all applications on Android devices. The Google Smart Application Update goes unnoticed to application developers and Android users. To enable the Google Smart Application Update, changes were made in the Google Play application, which comes with the Android operating system, and to the server software that handles users' requests. The Google Play application is now able to

construct new versions of updated applications by applying a received patch to an old version of an application installed on an Android device.

Another method to update Android applications is Update Direct for Android [8], which is commercially distributed by Pocket Soft. It also uses delta compression to reduce the update size. Pocket Soft provides a library that must be included in the application. This library allows the application to download updates from the host server and to deploy then. Although Update Direct for Android can shrink the update size, it has some significant drawbacks. First, it requires changes in the application source code and a server to serve app updates. Thus, updates are no longer distributed through the Google Play store, which decreases security and significantly increases maintaining and updating costs. Also in its current version, Update Direct for Android does not allow developers to use the Google App Engine, which provides useful services and scalability though Google's infrastructure. Such restrictions make applications that use Update Direct for Android less trust-worthy for users (because application updates are downloaded from some server and not from a well-known Google Play store) and harder to develop and maintain for application developers.

2.5 Application Markets Statistics

2.5.1 Google Play

Google Play statistics were collected in November 2012 and were used to estimate savings that can be achieved with DELTA++ deployment. Information was manually collected for the top 110 free applications, which are tabulated in the market. It is worth noting, that application's popularity is not determined solely by the number of downloads, application's ratings, or any other parameters. Table 3 shows a summary of

the obtained statistics. To get the statistics for each application, the application main page on the Google Play web site was parsed. Number of downloads for each application was calculated as the average of the range given in Google Play. Time since last update was calculated as the difference between the statistics collection data and the date when the last application update was posted in the Google Play store.

Table 3. Statistics for the Top 110 Free Applications in Google Play

Measurement	Value
Average size	6.21 MB
Average number of downloads	58 million
Average time since last update	29 days

2.5.2 The App Store

Apple App Store statistics were collected in January 2013 and were used to estimate savings that can be achieved with the deployment of a method similar to DELTA++ in the application store. Information was manually collected for the top 110 free iPhone applications. As in the Google Play store, an application's popularity is not determined solely by the number of downloads, application's ratings, or any other parameters. To get the statistics for each application, the application main page on the App Store web site was parsed. The Apple App Store does not provide any information about the number of application downloads for each application. Time since last update was calculated in the same way as for the Google Play store - as the difference between the statistics collection data and the date when the last application update was posted in the App Store.

Table 4 shows a summary of the obtained statistics. It is worth noting that the average size of the iOS applications in the Apple App Store is more than four times larger

than the average size of a popular Android application in the Google Play store. This might be related to the higher popularity of games among iPhone users. Games accounted for more than 90% of applications with a size greater than 30 MB.

Table 4. Statistics for the Top 110 Free Applications in the App Store

Measurement	Value
Average size	25.94 MB
Average number of downloads	unknown
Average time since last update	46 days

The large size of smartphone applications in the Apple App Store might be the reason why applications are not updated as frequently as Android apps in the Google Play store. The average time since last update for popular applications in the Apple App Store is 46 days, which is more 50% longer than the average time since last update for Android Apps. Unfortunately, statistics about the number of downloads are not presented in the market.

CHAPTER 3

DELTA AND DELTA++

3.1 Introduction

Android applications are distributed as APK packages, which can be seen as compressed ZIP archives of all files application contains. DELTA [11] was developed in May 2012 and showed that delta encoding can be successfully used to reduce traffic generated by app updates. Generally DELTA works as the Smart Application Update method [9], a similar method later developed by Google and announced in August 2012. DELTA++ [12] improves on DELTA by exploring specifics of APK files. Its main idea is to unpackage APK file and compute difference the between raw application files rather than between compressed archives of files. This approach, in combination with several optimizations, allows DELTA++ to construct much smaller patches, thus reducing app update traffic.

Any delta encoding based application updating method can be divided into four main parts, which are:

1) Patch computation from the old and the new application versions.
2) Patch transmission from the server to the smartphone.
3) Patch deployment by applying it on the old version of the updated application.
4) Installation of the new application version instead of the old version.

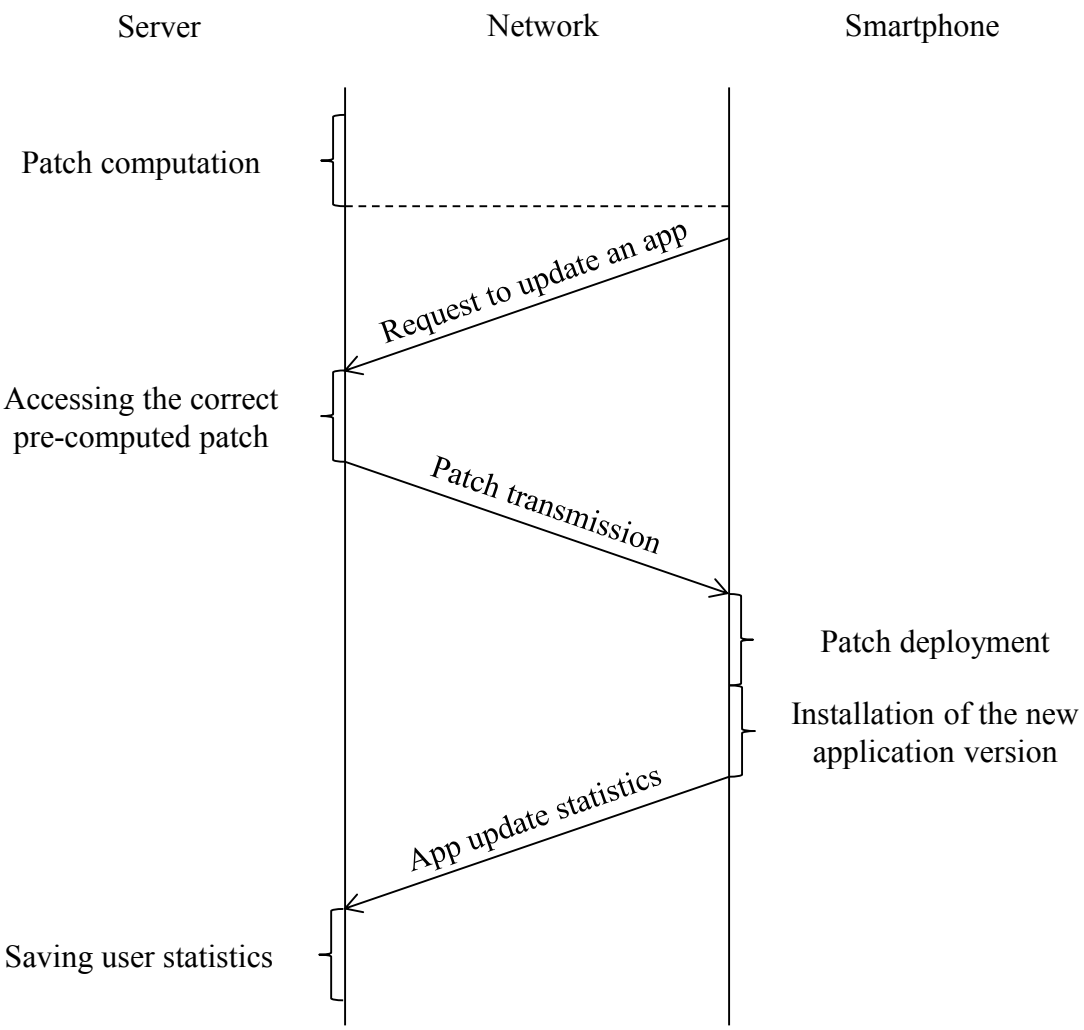

Figure 4. Scheme of an Application Updating Method Based on Delta Encoding

Figure 4 shows the scheme of a smartphone application updating method based on delta encoding. Patch computation is done in advance on the server side in the data center as soon as the new application version is available. This computation needs to be done only once for each application version. Within this thesis it is assumed that the server always has a pre-computed patch to update to the latest application version from any earlier version installed on the user smartphone. This assumption is reasonable because in real life the server can notify the end devices that a new update is available only after all the patch computations have been completed. The smartphone's request to update an

application is triggered manually by user or by a server notification if automatic updates are allowed for this particular app. Patch deployment and application installation is done on the user's smartphone and is done each time an application is updated. Both DELTA and DELTA++ do not specify how patch is sent to the user smartphone or how the obtained new application version is installed. Implementation details can be found in Section 3.4 of this chapter. Updating statistics is sent to the server after the installation of the new application version is complete (either successfully or unsuccessfully).

The bsdiff delta encoding algorithm [13] is used both in DELTA and DELTA++ to compute patches between files. This algorithm was proven to be one of the most cost effective delta encoding algorithms in various usage scenarios [13, 37, 49]. Patches created by bsdiff can be deployed using the bspatch algorithm. Bsdiff and bspatch source code in C++ is available from [66]. Although the current implementation of both DELTA and DELTA++ methods uses bsdiff, any other platform-independent delta encoding algorithm can be used to compute the differences between files.

3.2 DELTA

The DELTA method generates a patch simply as a delta difference between the application's old version APK file and the new version APK. Figure 5 shows the DELTA patch construction algorithm.

1) The old application version APK and the new application version APK files are given as an input to the bsdiff delta encoding algorithm that constructs the patch as a difference between the provided versions.

Figure 5. DELTA Patch Construction Algorithm

On the smartphone bspatch algorithm is used to apply the received patch. The DELTA patch deployment algorithm is shown on Figure 6.

1) The APK package of the current application version is located using ApplicationInfo class.
2) Bspatch algorithm is used to obtain a new version of an application from the received patch and the old version of the app.

Figure 6. DELTA Patch Deployment Algorithm

Simplicity of the DELTA method allows to use it not only for Android APK files but for any application package format in general.

3.3 DELTA++

The size of a patch computed by a delta differencing algorithm primarily depends on the amount of differences between two files. There are some specifics that might affect the resulting patch size, one of them is use of compression. For example, if we have two text files with only very few differences it is possible that the compressed versions of these files might be highly different on binary level because of the ways they were processed during compression. The same happens with the distributed APK application package which is basically just a compressed archive of all the files that comprise an Android application. The main idea of DELTA++ is to compute the difference between the application files within an APK and not between the compressed APK packages themselves as even small differences in the application's files can cause significant difference between the application's APK packages.

Figure 7 shows the step-by-step DELTA++ patch construction algorithm. Delta difference between small files can be sometimes larger than the files themselves due to the control information contained in the patch file. It usually the case for small images that generally hardly benefit from delta encoding algorithm due to their two dimensional nature and usage of compression to store image data. Therefore, at Step 6 of the patch

construction algorithm the computed difference is compared with the new file version itself and the smallest one is used.

1) The APK packages of the old and the new versions of an application are decompressed. During decompression APK files are treated like ZIP archives.
2) The manifest files of both versions are traversed to get the names, relative paths, and SHA-1 hash digests of all the files (in the APK) included in both versions.
3) The files contained in the new version are marked as NEW (if the file is present in the new version but not present in the old one), UPDATED (if the file is present in both versions but its SHA-1 sums differ) or SAME (if the file is present in both versions and the SHA-1 digests are the same). It is worth noting that the application's metafiles (contained in META-INF directory inside APK package) are not presented in manifest, but are also marked as UPDATED.
4) The files contained in the old version are marked as DELETED if the file is present in the old version but was deleted in the new one.
5) The files from the latest version that are marked as NEW are just copied into the constructed patch.
6) The files from the latest version that are marked as UPDATED are given as input to the bsdiff delta encoding algorithm to compute differences between the old and new versions. This difference is then copied into the constructed patch. Sometimes the difference between small files can be greater than size of the files themselves because of the overhead associated with the delta file creation. In such cases, the new file is re-marked as NEW and is copied into the patch.
7) The files that are marked as SAME remain untouched.
8) PatchManifest.xml file is created and included in the patch. It serves as a patch description and comprises information about which application version can be updated using the patch and what NEW files and delta differences between UPDATED files are contained in the patch. Information about files marked DELETED is also included in PatchManifest.xml.
9) Finally, the constructed patch is compressed into a ZIP archive using bzip2. The compressed patch is then ready to be sent to an Android device for deployment.

Figure 7. DELTA++ Patch Construction Algorithm

Figure 8 shows the file marking process that takes place during Step 3 and Step 4 of the DELTA++ patch construction algorithm. DELTA++ patch deployment in the user smartphone (Android device) is shown on Figure 9.

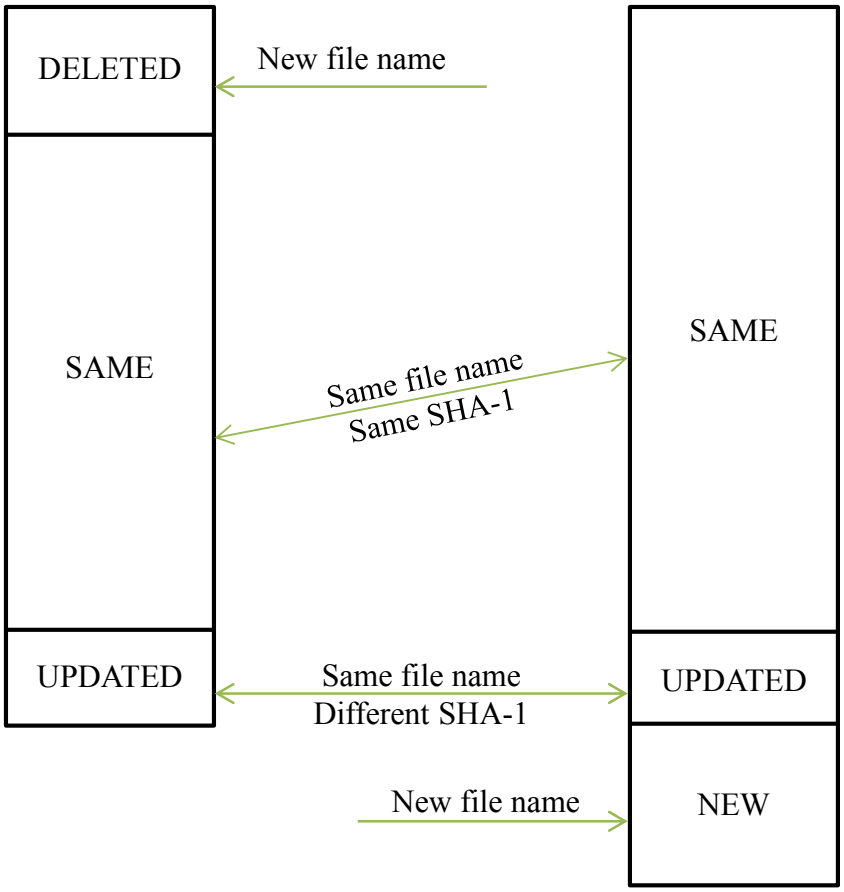

Figure 8. Files Marking During DELTA++ Patch Construction

1) The received patch is decompressed using an Android built-in Zip class.
2) The APK package of the current application version is located using ApplicationInfo class.
3) The PatchManifest.xml file is accessed to obtain file marking information (names of the files that are marked DELETED, NEW, SAME or UPDATED.
4) All the files in the old application APK marked DELETE are no longer required and are deleted.
5) Proper differences in the patch are applied to the files marked UPDATED, thus updating them.
6) All NEW files from the patch are copied to the old application version. At this point, the old version contains exactly the same files as the new version of application.
7) The APK package is constructed by compressing all the files into a ZIP archive with an .apk extension.

Figure 9. DELTA++ Patch Deployment Algorithm

3.4 Implementation of DELTA and DELTA++

DELTA and DELTA++ was implemented as server side software, which constructs patches and serves them by request, and as an Android application that deploys the received patches and updates the installed applications. The updating algorithm (DELTA or DELTA++) is explicitly set in the server configuration and in the Android application settings. The developed software provides several levels of abstraction allowing effortless addition of new delta encoding and patch construction algorithms. Source code for all the software mentioned in this thesis is available from [64].

Server side software was written in C# and uses the bsdiff delta encoding algorithm to calculate the delta difference between application packages.

An Android app was developed to apply patches locally on a smartphone and to install the obtained APK packages after patch deployment. It uses bspatch algorithm to deploy the received patch and to construct the new application version. The Linux version of bspatch was ported to Android using Android Native Development Kit (NDK), which allows executing C code directly on the computing platform rather than on the Dalvik Virtual Machine. Another Android-friendly version of the bspatch source code can be found in the Android operating system source tree [10]. However, it was not suitable for the developed application because it does not have an adapter layer that provides an interface to call C functions from the Java code.

In smartphone the constructed APK package of the new application version is installed using the Android PackageInstaller build-in application. It is worth noting that the same built-in application is used by the Google Smart Application Update method.

CHAPTER 4

EVALUATION OF DELTA AND DELTA++

4.1 Methodology

In order to evaluate DELTA and DELTA++, experiments were conducted using the 110 most popular free Google Play applications in November 2012 (available from [65]). For these applications, delta patches were generated and deployed based on previous versions of the apps. Previous versions were collected manually and locally archived in a period from November 2012 till January 2013. Patches were generated and deployed using DELTA, DELTA++ and Google Smart Application Update. If more than two versions of an application were available, computations were done for each consecutive pair of application versions and an average was taken. For each time measurement 5 repetitions were done and the average was taken.

In all the experiments a PC with an Intel Core i5 2.30 GHz processor and 8 GB RAM was used to generate delta patches. An HTC Thunderbolt smartphone with a single core, 1000 MHz Snapdragon processor, and 768 MB RAM was used to deploy patches.

The response variables of interest were patch size and the time to apply the received patch and to install the constructed application. Key measures were:

- Size of a patch compared to size of the new application version.
- Improvement compared to Google Smart Application Update.
- Time to deploy a patch and install the new application version instead the old one.

Improvement compared to Google Smart Application Update was measured as a fraction of the patch generated by Google Smart Application Update that can be saved if DELTA or DELTA++ is used. The Google Smart Application Update improvement upon DELTA++ was calculated as a fraction of the patch deployment time that can be saved. The improvement was calculated for every application and an average was taken.

4.2 Experimental Results

4.2.1 DELTA Patch Size

Figure 10 shows the size of the patch, as generated by DELTA and by Google Smart Application Update. For Google Smart Application Update, the average patch size was 46% of the latest application version's size, the minimum was 4% (for Bike Race Free application) and the maximum was 100% (for the Adobe Reader application). For DELTA the average patch size was 39%, the minimum was 4% (for ESPN Fantasy Football application) and the maximum was 93% (for Adobe Air application).

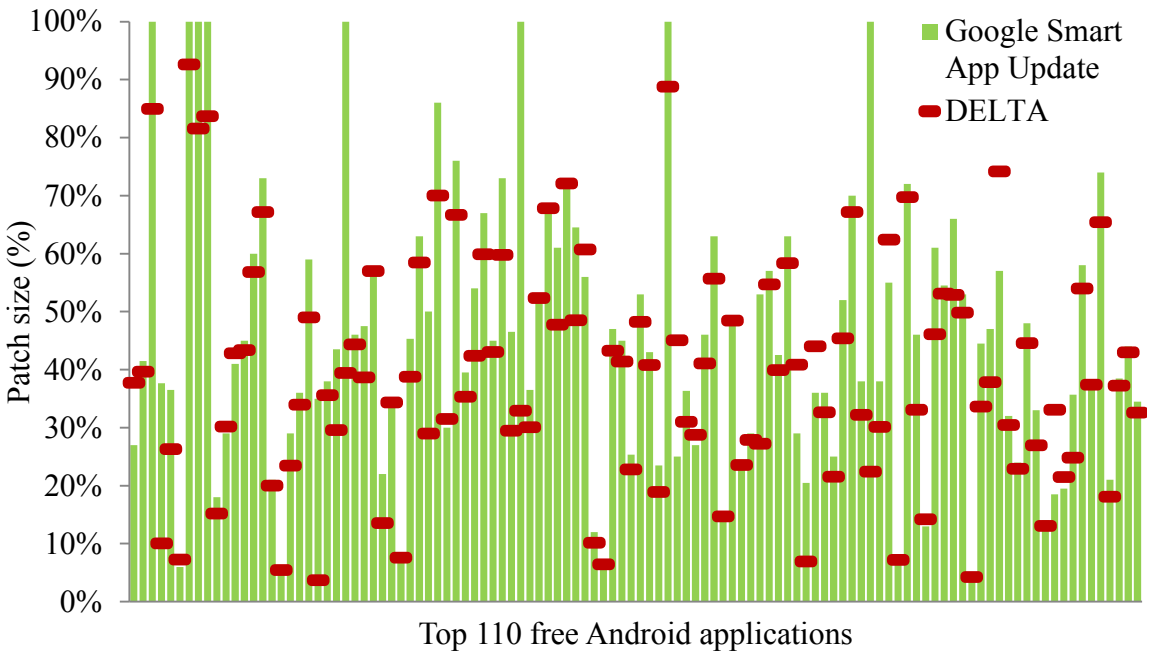

Figure 10. Patch Size for DELTA Compared to Google Smart Application Update

Figure 11 shows the improvement of DELTA on the Google Smart Application Update updating method in terms of traffic reduction. Negative improvement means that patch size for DELTA is actually larger than patch size for Google Smart Application Update. Average savings were 10%, the minimum was 89% (for ESPN Fantasy Football application) where patch size increased, and the maximum was -80% (for Linked In application). It is worth noting that in 47% of all the cases the difference in patch size between DELTA and Google Smart Application Update is less than 10%.

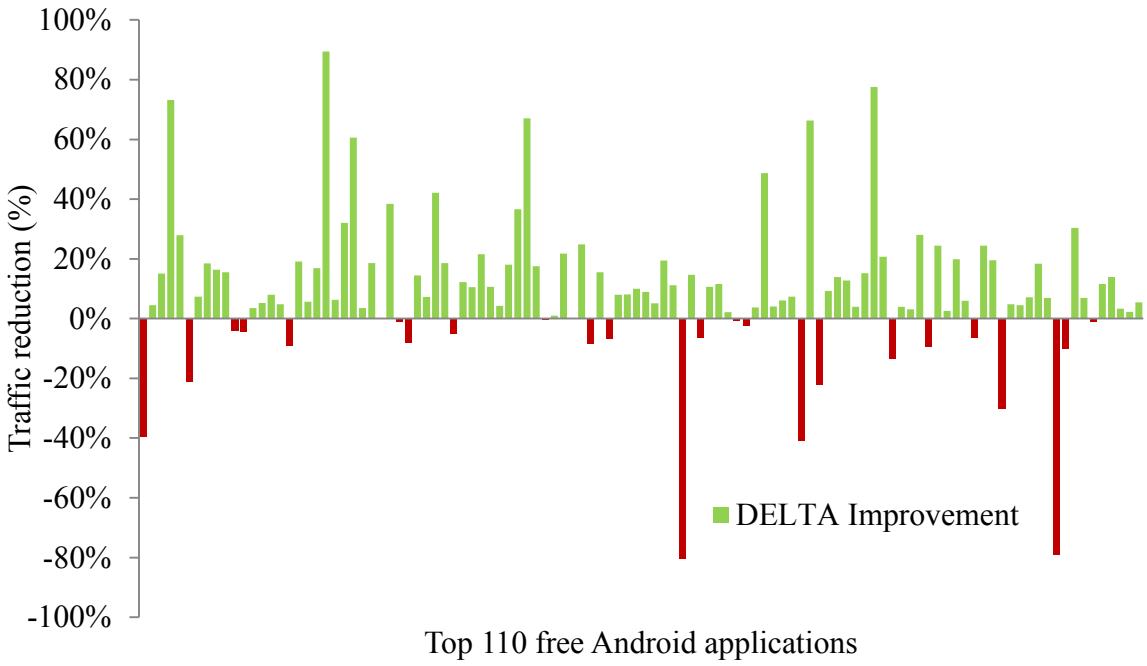

Figure 11. DELTA Improvement on Google Smart Application Update

4.2.2 DELTA++ Patch Size

Figure 12 shows the size of the patch, as generated by DELTA++ and by Google Smart Application Update for the top 110 most popular apps. For Google Smart Application Update, the average patch size was 46% of the latest application version's size, the minimum was 4% (for Bike Race Free application) and the maximum was 100% (for the Adobe Reader application). For DELTA++ the average patch size was 23% of

the latest version size, the minimum was 0.1% (for Brightest Flashlight Free application) and the maximum was 81% (for WatchESPN application).

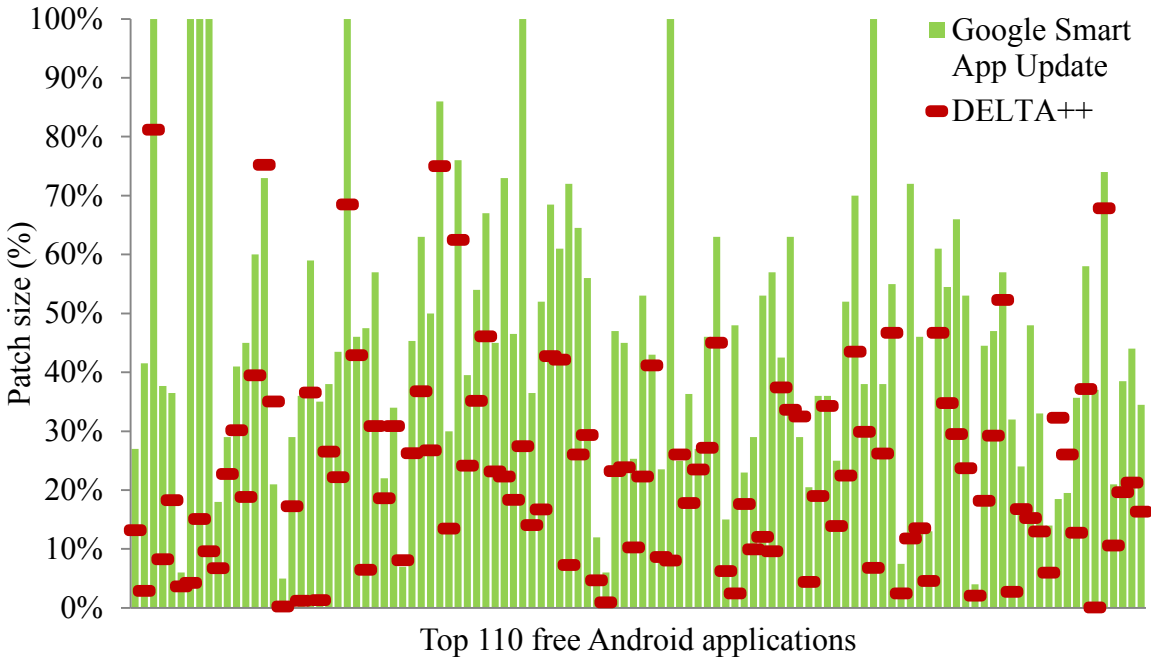

Figure 12. Patch Size for DELTA++ Compared to Google Smart App Update

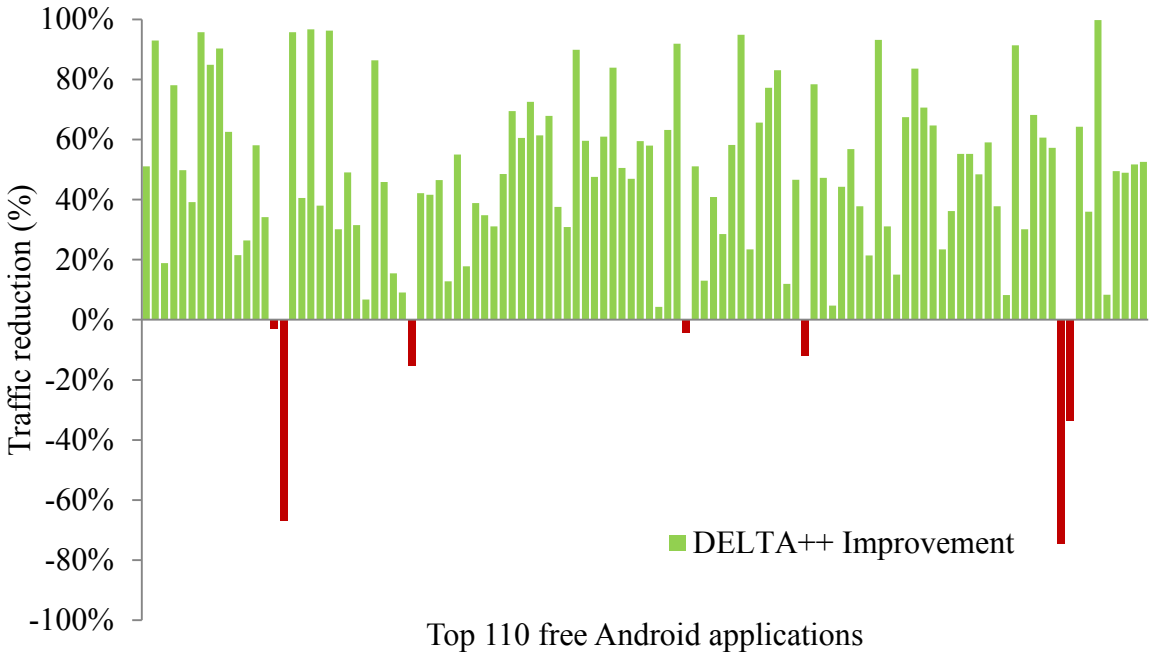

Figure 13. DELTA++ Improvement on Google Smart Application Update

Figure 13 shows the improvement of DELTA++ on the Google Smart Application Update method in terms of reduction of transmitted data. Average savings were 46%, the minimum was -75% (for Scramble with Friends application) where patch size increased, and the maximum was 99.8% (for Brightest Flashlight Free application). Thus, DELTA++ significantly reduces applications updates traffic comparing to the Google Smart Application Update method.

In some cases both methods produced patches that were only slightly smaller than the full version of the application. Such large patches occur when numerous new resource files (e.g., images, video files, or third-party libraries) have been added in the new version of the application. The size of other patches is significantly affected by the differences in the application code. As we compute patches between consecutive application versions, this may be due to the use of tools such as ProGuard, which is included in the Android development environment, that obfuscates byte code making it harder to decompile. Such obfuscation of byte code introduces new differences between two files on binary level and leads to larger patches produced by delta encoding algorithm.

4.2.3 DELTA++ Update Time

As it was mentioned in Section 3.1, application updating can be divided into four steps, which are 1) patch construction, 2) transmission of patch, 3) patch deployment on the device, and 4) installation of the updated application version. DELTA++ decreases the transmission time by reducing the transferred file size but requires more time to deploy a patch. Figure 6 shows the time to apply a DELTA++ patch and install an updated application compared to the same time for Google Smart Application Update.

For DELTA++, the average time was 62.5 s, the minimum was 4.6 s (for Barcode Scanner app) and the maximum was 212.3 s (for Angry Birds app). For Google Smart Application Update, the average time was 12.3 s, the minimum was 1.0 s (for ESPN Fantasy Football app) and the maximum was 57.5 s (for Temple Run app).

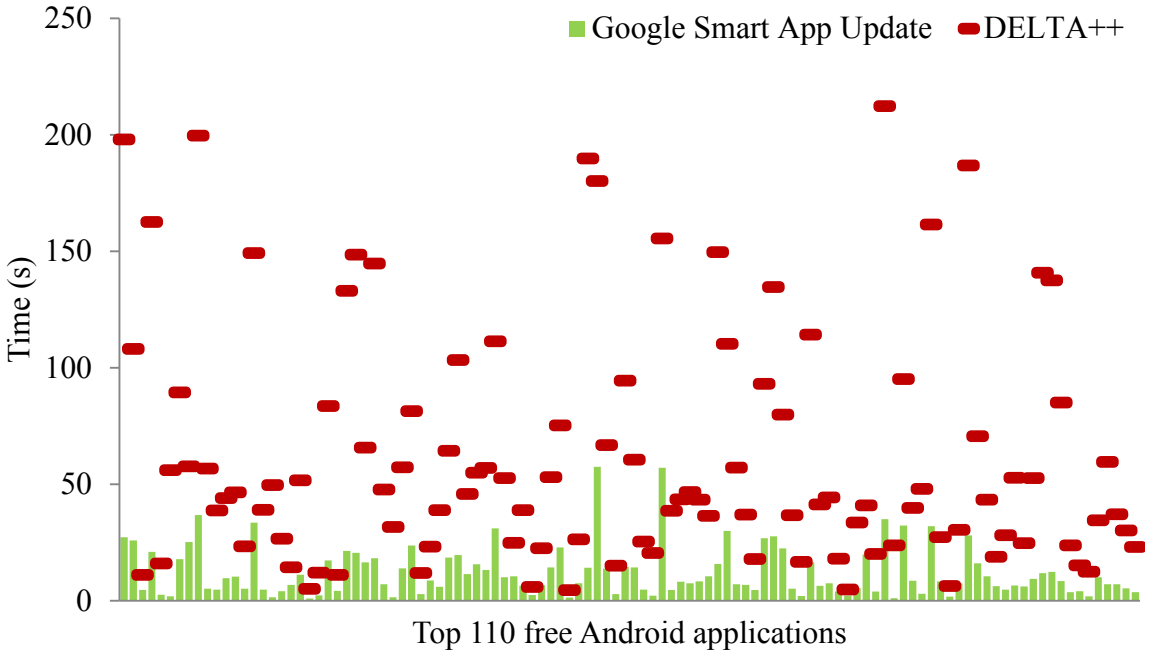

Figure 14. Patch Deployment and Installation Time for DELTA++ Compared to Google Smart Application Update

Figure 15 shows the improvement of the Google Smart Application Update method on DELTA++ in terms of reduction of the time it takes to deploy the patch and install the new application in smartphone. Average savings were 80%, the minimum was 51% (for Gun Disassembly application), and the maximum was 97% (for Defender 2 application). Thus, deployment of the patches generated by DELTA++ takes significantly longer comparing to the Google Smart Application Update method.

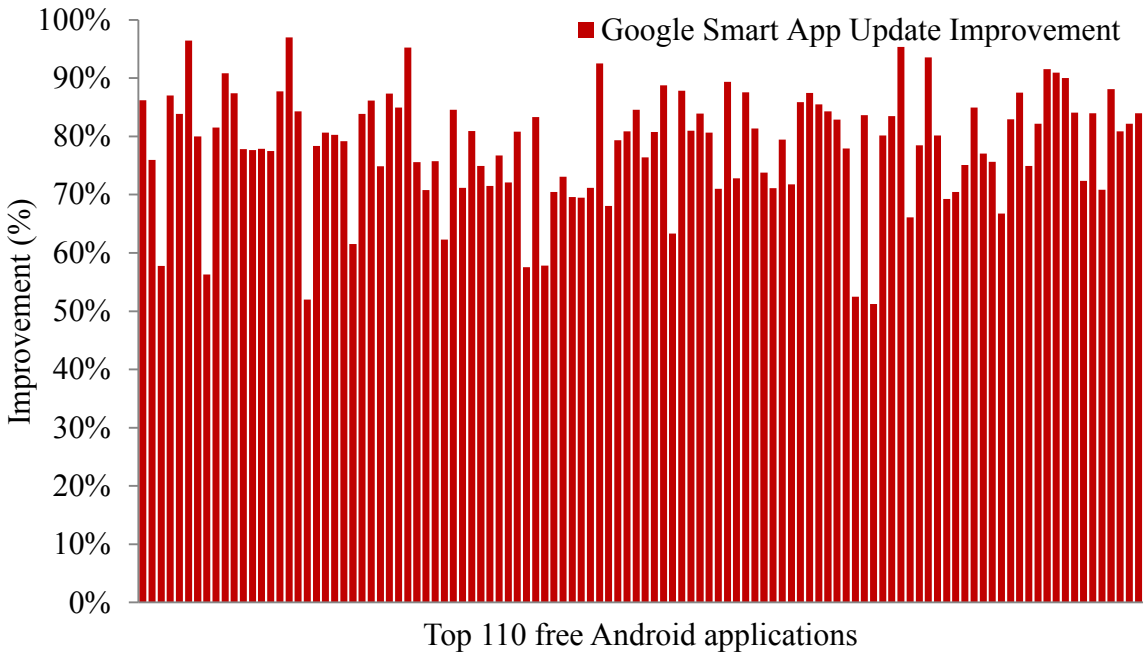

Figure 15. Google Smart Application Update Improvement on DELTA++

4.3 Discussion

Table 5. Summary of the Patch Size Measurements

Measurement	DELTA	DELTA++	Google Smart Application Update
Average	39.42%	23.36%	45.63%
Minimum	3.68%	0.08%	4.00%
Maximum	92.61%	81.17%	100.00%
Median	38.24%	21.71%	44.25%
Standard deviation	19.85%	17.25%	23.35%

Table 5 summarizes the results of the experiments where the patch size was measured. All the patch sizes are given relatively to the size of the new version of a particular application. Both DELTA and DELTA++ generate patches that are smaller than those generated by the Google Smart Application Update method. Patches generated by the DELTA method are smaller by 6.21 percentage points in average. Patches generated by DELTA++ are smaller by 22.27 percentage points in average. DELTA++

also significantly improves on DELTA and constructs patches that are smaller by 16.06 percentage points in average.

Table 6. Summary of the Time Measurements

Measurement	DELTA++	Google Smart Application Update
Average	62.5s	12.3s
Minimum	4.6s	1.0s
Maximum	212.3s	57.5s
Median	45.1s	8.4s
Standard deviation	50.9s	10.8s

Table 6 summarizes the results of the experiments where the patch deployment and application installation time was measured. In both methods the new application version is installed using the built-in Android application. Thus, the application installation takes the same time with DELTA++ and Google Smart Application Updates. The time difference occurs because the construction of the new application version (patch deployment) takes much more time when DELTA++ patches are used due to the complexity of the DELTA++ patch. Application of the Google Smart Application Update patches is faster by 50.2 seconds in average.

Experimental results show that DELTA++ significantly improves on the currently used Google Smart Application Update method in terms of the generated patch size, which is reduced by two fold. However, DELTA++ constructs more complex patches, increasing patch deployment and application installation time by 5 and, thus, increasing CPU consumption on smartphone. DELTA++ trades CPU cycles and application updating time to reduce the bandwidth usage.

Although DELTA++ patches take much longer to deploy, savings from traffic reduction exceed these costs. Modern smartphones have enough CPU power to update applications using DELTA++ patches in approximately 1 minute on average. Also, application updating is a delay-tolerant process because users do not need updates immediately after they are released. Thus, delay in application updating can be tolerated by smartphone users. However, download traffic is usually limited and can cost as much as $50 a month for 5 GB of data (for example, a 5 GB data plan from AT&T as of January 2013). Thus, the amount of traffic generated by app updates can be considered the biggest concern for smartphone users, data centers and mobile operators.

Other problem that might arise with usage of delta encoding is that patch can be applied only on unmodified application. If application files were corrupted (either by virus or user), then delta update will fail while a full version of application can be still installed.

CHAPTER 5

CHARACTERIZATION OF ANDROID SMARTPHONE USERS

5.1 Introduction

To estimate how much actual savings could be achieved with DELTA++ it is necessary to understand how users update applications on their devices. To characterize user behavior an Android application, named DELTA Statistics, was created. This application collects statistics on how applications are updated on the end-user smartphones. All the collected statistics are sent to a configured server, where it is aggregated and analyzed. Users can also receive the statistics obtained from their devices using the application's functionality. DELTA Statistics application is freely available from Google Play. Source code is available from [64].

5.2 Methodology

The key questions to help estimate savings from traffic reduction are:

- How much traffic is generated by application updates in smartphone.
- What fraction of application updates traffic is transmitted through Wi-Fi, which usually can be used limitless by smartphones users (e.g. home Wi-Fi network) without extra charges.

DELTA Statistics collects the following information:

- How many applications a user has installed on his or her Android device.
- How frequently applications are updated.

- What wireless technology is used to receive an update (for example, Wi-Fi, 3G, 4G, etc.)

DELTA Statistics was installed on 20 Android devices and information was collected during three months – from November 2012 until January 2013. The 20 devices were owned by students at the University of South Florida.

5.3 DELTA Statistics Application Implementation

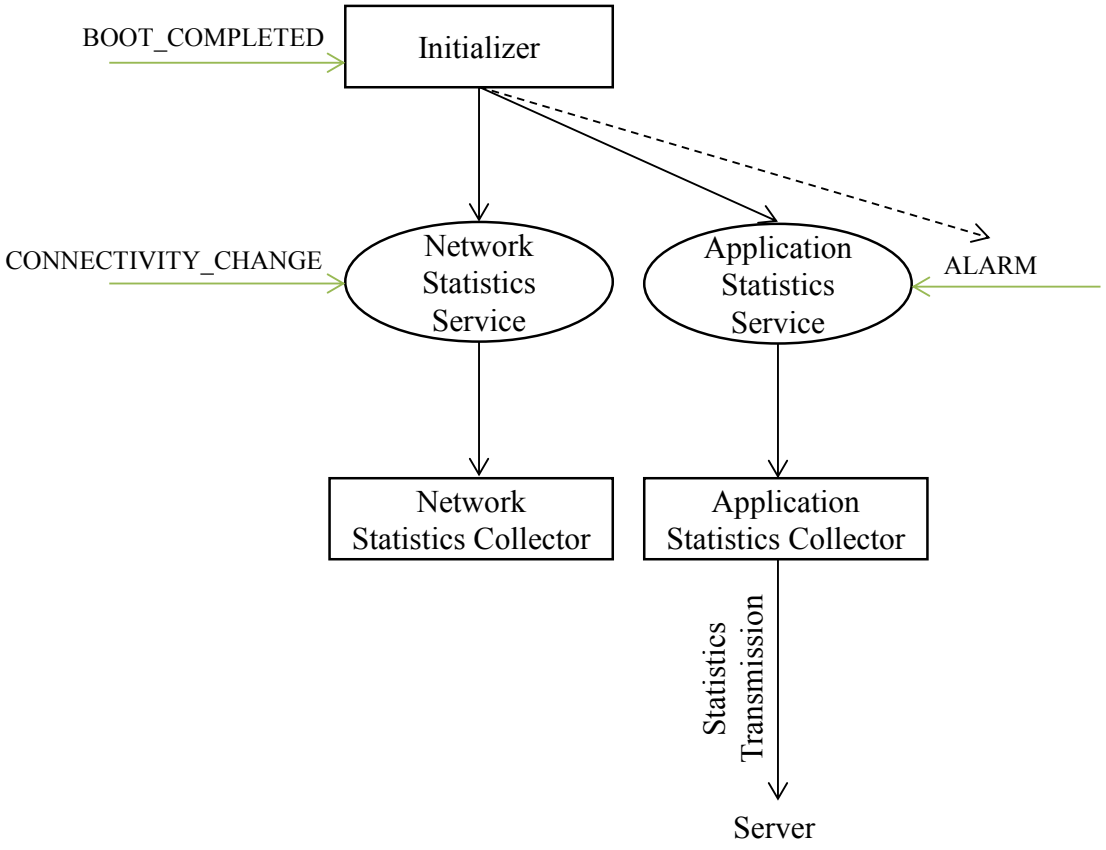

Figure 16. Design of the DELTA Statistics Application

Figure 16 shows the general application design. The arrows on the figure represent the special events – called intents – distributed by the Android operating system. Intents are used as triggers to start particular processes in the application.

Rectangles on the figure represent parts of the application that are invoked to do some work and terminate after this work is completed. Ovals represent services – processes that do work in background and go into the sleep mode when they complete their tasks.

DELTA Statistics application starts immediately after the smartphone finishes its booting process or when it is opened by the smartphone user. BOOT_COMPLETED intent propagated by the operating system invokes the Initializer program block. Initializer starts the Application Statistics Service, the Network Statistics Service, and creates a custom ALARM that will go off every 24 hours. The Network Statistics Service is responsible for the collection of the network connectivity statistics. It receives the CONNECTIVITY_CHANGE operating system intent, which is broadcasted every time wireless connection is changed. The intent includes the name of the changed wireless technology (Wi-Fi, 3G, etc.), its new state (CONNECTED, DISCONNECTED, SUSPENDED, etc.) and a timestamp. When the CONNECTIVITY_CHANGE intent is received, the service invokes the Network Statistics Collector block that saves all the information about the device's network connections from the received intent.

The Application Statistics Service is responsible for the collection of the statistics about every non-system application installed on the device. The service wakes up every time it receives the scheduled ALARM (every 24 hours) and starts the Application Statistics Collector that gathers and saves information about all the installed non-system application. FLAG_UPDATED_SYSTEM_APP and FLAG_SYSTEM flags provided by the ApplicationInfo system class are used to detect system applications. For each application the Application Statistics Collector saves the application's package name and the time of the last update.

DELTA Statistics aggregates the collected information and sends it to the server during the daily communication that occurs every time the Application Statistics Collector finishes its work. On the server side, a history of the devices connectivity is maintained. An application update is determined by comparing the time of the last update for each application with the same data received during the previous communication. When the server detects that application was updated since the last transfer, it uses the devices connectivity history to determine what technology was used to receive update.

5.4 Study Results

Table 7. Characterization of Android Users in the U.S.

Measurement	Value
Number of participants	20
Average number of apps per smartphone	47
Average days between updates	41 days
Fraction of updates deployed via Wi-Fi	37 %

It was found that in average an Android smartphone has 47 applications installed varying from 31 to 97 applications for the 20 studied devices. It is worth noting that the average time since last update obtained from Google Play (see Table 3) is 30% smaller than the same data obtained from the conducted users study (29 days versus 41 days). This shows that users tend not to update their application immediately after the new version release. This notion is used during savings estimation in Chapter 6 where it is assumed that in practice applications are updated with 41% delay comparing to the statistics collected from the application store. It was also found that 37% of all updates are downloaded via Wi-Fi, which correlates to the overall Wi-Fi usage statistics published by CISCO [4]. Table 7 summarizes the statistics obtained from this user study.

CHAPTER 6

SAVINGS ESTIMATION

6.1 Methodology

This chapter presents a first order estimate of the traffic generated by app updates in the U.S. and the savings that could be achieved with full deployment of our proposed DELTA++ scheme in the Google Play store and a method similar to DELTA++ in the Apple App Store. It is assumed that delta encoding methods can achieve the same benefits for iPhone app updates as for Android app updates.

The average time between updates for Android applications is taken from the conducted study of Android users (see Chapter 5). For iPhone applications this time is taken from the obtained market statistics (see Section 2.5.2) and then scaled-up by 41% based on the observation that users tend to update their applications with a delay (see Chapter 5). In order to do this, time between updates is increased by 41% made during the conducted study of Android users.

A fraction of overall traffic that is downloaded via Wi-Fi is taken from the Android users study and assumed to be true for both Android and iPhone users. The number of smartphone users and operating systems market shares are taken from the comScore report [1] and CISCO Visual Networking Index [4].

A first-order estimate of the cost savings is made based on the cost of data service plans as of January 2013. A 5 GB data plan from AT&T is used.

V1: Number of smartphones in the U.S. = 114 million (See [1])

V2: Market share = See [1]

V3: Average time since last update = See Table 3 for Google Play and Table 4 for the App Store

V4: Fraction of updates deploy via Wi-Fi = 37% (See Table 7)

V5: Updating delay in practice = 41% (See Section 5.4)

V6: Number of smartphones = V1 × V2

V7: Number of apps per smartphone = 47 (See Table 7)

V8: Average size of application = See Table 3 for Google Play and Table 4 for the App Store

V9: Average days between updates = V3 × (1 + V5)

V10: App update traffic per year per phone = V7 × V8 × (365 / V9)

V11: Total app update traffic = V6 × V10

V12: Total app update traffic with Google Smart Application Update = V11 * 46% (See Table 5)

V13: Total app update traffic with DELTA++ = V11 * 23% (See Table 5)

V14: Extra savings with DELTA++ = V12 – V13

V15: Extra savings with DELTA++ in cellular = V14 * (1 – V4)

Figure 17. Formulas Used During the Estimation of Annual Traffic Reduction in the U.S.

V1: Estimated cost of downloading 1 GB of data = $10 (See AT&T 5 GB data plan)

V2: Traffic reduction with DELTA++ (%) = 77% (See Table 5)

V3: App update traffic per year per phone = See Table 8 for Android smartphones and Table 10 for iPhone smartphones

V4: Annual savings with DELTA++ per phone = V2 × V3

V5: Cost savings with DELTA++ per phone = V1 × V4

V6: Number of smartphone in the U.S. = See Table 8 for Android smartphones and Table 10 for iPhone smartphones

V7: Total cost savings with DELTA++ in the U.S. = V5 × V6

Figure 18. Formulas Used During the Estimation of Annual Cost Savings in the U.S.

All the formulas used during the estimation of annual traffic reduction in the U.S. both for Google Play and for the App Store are presented on Figure 17. Formulas used during the estimation of annual cost savings in the U.S. both for Google Play and for the App Store are presented on Figure 18.

6.2 Google Play

Table 8. Estimate of Annual Traffic Reduction in the U.S. for Google Play

Measurement	Estimate
Number of Android smartphones	60 million
Number of apps per smartphone	47
Average size of application	6.2 MB
Average days between updates	41 days
App update traffic per year per phone	2.5 GB
Total app update traffic	144 PB
Total app update traffic with Google Smart Application Update	65 PB
Total app update traffic with DELTA++	33 PB
Extra savings with DELTA++	32 PB
Extra savings with DELTA++ in cellular	20 PB

There were more than 114 million smartphone users in the U.S. in 2012 with 52% of smartphones running the Android operating system [1]. The average application size for the top 110 free apps in Google Play is 6.2 MB, which leads to approximately 2.5 GB in yearly applications update traffic for each user considering the average of 47 applications on an Android smartphone updated every 41 day. These updates add up to the total of 144 PB yearly traffic for all users in the U.S. Google Smart Application Update provides 54% savings and reduces this traffic down to 65 PB, while DELTA++ enables 77% savings that further decreases application updates traffic by 32 PB resulting in 33 PB yearly traffic. Users study (see Table 7) shows that 37% of updates are done

using Wi-Fi, which means that extra savings in cellular networks is approximately 20 PB. Table 8 summarizes the savings that can be achieved in the Google Play store with full deployment of the DELTA++ application updating method.

Table 9. Estimate of Annual Cost Savings in the U.S. for Google Play

Measurement	Estimate
Estimated cost of downloading 1 GB of data	$10
Traffic reduction with DELTA++ (%)	77%
App update traffic per year per phone	2.5 GB
Annual savings with DELTA++ per phone	0.58 GB
Cost savings with DELTA++ per phone	$5.80
Number of Android smartphones in the U.S.	60 million
Total cost savings with DELTA++ in the U.S.	$350 million

A first-order estimate of the cost savings of DELTA++ compared to Google Smart Application Update can be made based on the cost of data service plans. At about $10 per GB for large carriers in the U.S., the cost savings per year would be about $5.80 per user per year (in reduced data downloaded) for Android users. Multiplied by the number of Android smartphone, the total savings are about $350 million per year. Estimate of annual cost savings for Google Play is shown in Table 9.

6.3 The App Store

Currently Apple does use any technology to reduce the size of transmitted package for application updates. iPhone applications are distributed as IPA packages, which can be generally seen as ZIP archives of all the files contained in the application. The DELTA method does not depend on the internal structure of the application package and, thus, can be used for iPhone application updates without any changes. DELTA++ exploits the internal structure of the Android application packages and cannot be used for

iPhone applications. However, iPhone IPA packages are very similar to Android APK packages. Thus, a method similar to DELTA++ can be implemented in the Apple App Store for iPhones.

According to [1] Apple market share in 2012 was 33% of all smartphones. The study of the top 110 application in the App Store shows that the average iPhone application size is 26 MB with 64 days from the last update this leads to 6.8 GB annual update traffic for each user. Multiplied by the number of iPhones this results in 247 PB yearly traffic in the U.S. from which 190 PB could be saved with delta encoding. Considering that 37% of updates occur via Wi-Fi, savings in cellular networks would be about 120 PB. Table 10 summarizes the savings that can be achieved in the Apple App Store if a method similar to DELTA++ will be used to update applications. Deployment of DELTA++ can provide much more savings in the App Store than in Google Play because no method is currently used in the App Store to reduce application updates traffic while Google Smart Application Update is used in Google Play.

Table 10. Estimate of Annual Traffic Reduction in the U.S. for the App Store

Measurement	Estimate
Number of iPhone smartphones	38 million
Number of apps per smartphone	47
Average size of application	26 MB
Average days between updates	64 days
App update traffic per year per phone	6.8 GB
Total app update traffic	247 PB
Total app update traffic with Google Smart Application Update	NA
Total app update traffic with DELTA++	57 PB
Extra savings with DELTA++	190 PB
Extra savings with DELTA++ in cellular	120 PB

Table 11. Estimate of Annual Cost Savings in the U.S. for the App Store

Measurement	Estimate
Estimated cost of downloading 1 GB of data	10$
Traffic reduction with DELTA++ (%)	77%
App update traffic per year per phone	6.8 GB
Annual savings with DELTA++ per phone	5.24 GB
Cost savings with DELTA++ per phone	$52.40
Number of iPhone smartphones in the U.S.	38 million
Total cost savings with DELTA++ in the U.S.	$2 billion

A first-order estimate of the cost savings with the deployment of a method similar to DELTA++ in the Apple App Store is made based on the same 5 GB data plan from AT&T used in Section 6.2. At about $10 per GB, the annual cost savings in reduced data downloaded would be about $52.4 per user per year for iPhone users. Multiplied by the number of iPhone smartphone, the total savings are about $2 billion per year. Cost savings from DELTA++ deployment for iPhone users are greater than for Android users because of the larger size of iPhone applications and the absence of any traffic reduction method in the Apple App Store. Estimate of annual cost savings for the Apple App Store is shown in Table 11.

6.4 Conclusions

It was estimated that annual application update traffic in the U.S. wireless networks is approximately 390 PB. If it is assumed than 37% of this traffic goes through Wi-Fi networks, then cellular traffic generated by smartphone application updates can be estimated at 246 PB a year, which according to the CTIA statistics [63] is 20% of the total annual cellular traffic in the U.S.

With deployment of DELTA++ and similar application updating methods, more than 140 PB can be saved in the U.S. cellular networks alone. According to [63], the total U.S. wireless data traffic in the first half of 2012 was 590 PB (so about 1180 PB per year). Thus, roughly 140 PB savings represents over 11.5% of all data – this is a significant savings. A first-order estimate of the cost savings shows that approximately $2.3 billion can be saved annually in the U.S.

For a single smartphone user DELTA++ provides an opportunity to trade application updating time to less bandwidth usage, which leads to cost savings. In case of Android smartphone user, 0.58 GB or $5.80 can be saved annually if 50 seconds increase of the application updating time can be tolerated.

Table 12 summarizes the tradeoffs that datacenters, mobile operators and smartphone users would face with the deployment of the DELTA++ application updating method.

Table 12. Tradeoffs from the DELTA++ Deployment

Affected Party	Pros	Cons
Smartphone user	Possibly decreased price of the data plan from the mobile operator, which will benefit from the decreased traffic	Increased energy usage because of the additional CPU usage during the patch deployment
		Increased CPU usage because of the additional CPU cycles spent during the patch deployment
		Increased total application updating time because of the significantly increased patch deployment time
		Possibly decreased user experience because of the increased total updating time and the increased energy consumption
Mobile operator	Decreased network bandwidth usage because of the traffic reduction	Possibly lost customers because of the decreased user experience
	Increased profit because less data needs to be transmitted through the network	
	Decreased energy usage because less data needs to be transmitted	
Datacenter	Decreased bandwidth usage because of the decreased update size	Increased space usage because of the necessity to store the computed patches
		Increased CPU usage because of the necessity to compute patches

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

The growing popularity of mobile devices that host multiple applications leads to significant network traffic from application updates. The annual application update traffic in the U.S. wireless networks alone is approximately 390 PB. If 37% of this traffic goes through Wi-Fi networks, then yearly traffic generated by smartphone application updates in cellular networks in the U.S. is approximately 246 PB, which is up to 20% of the total annual cellular traffic in the U.S.

This thesis is focused on Android and application updates as served by the Google Play store. Delta encoding based methods to update Android application were described, implemented and evaluated. The proposed DELTA++ application updating method significantly improves upon Google Smart Application Update in generating 50% smaller patches. This thesis also presents a study of Android smartphone users and a first order of magnitude estimation of possible savings. The deployment of delta encoding methods similar to DELTA++ for Android and iPhone application updates could reduce yearly traffic in cellular networks by 140 PB or by more than 11.5%. This would lead to significant savings for mobile operators and data centers as less resources (both network bandwidth and number of servers) would be required for serving patches.

DELTA and DELTA++ introduce a new trade off: network bandwidth can be saved at the expense of the smartphone CPU cycles and the total application updating

time. For smartphones users DELTA++ can provide significant traffic reduction and cost savings if 50 seconds delay can be tolerated during application updating.

Future research directions include:

- Developing a delta encoding based method to update App Store applications.
- Studying how much savings can be achieved in the data centers that serve application updates if these updates can be distributed from one smartphone to another using a P2P protocol. An application can be developed and used to upgrade apps without downloading them from a data center once a sufficient number of devices received an update.
- Studying similarities between different applications and developing methods to eliminate redundancy between them. This will allows to further reduce the size of application update or even the first application download.
- Developing a framework that will combine all the designed methods and will provide users an easy way to efficiently update all the applications installed on their devices. This framework can also include further optimizations such as pre-fetching of the available updates via Wi-Fi network when possible and then instantly updating applications by user request. Availability of different updating methods will allow users to specify different policies and decide whether they want to minimize traffic or update application as fast as possible.

LIST OF REFERENCES

[1] "comScore Reports July 2012 U.S. Mobile Subscriber Market Share," comScore, September 4, 2012. URL: http://www.comscore.com/Insights/Press_Releases/2012/9/comScore_Reports_July_2012_US_Mobile_Subscriber_Market_Share.

[2] K. De Vere, "Android Reaches 25 Billion App Downloads, 675,000 Total Apps Available," September 26, 2012. URL: http://www.insidemobileapps.com/2012/09/26/android-reaches-25-billion-app-downloads-675000-total-apps-available/.

[3] D. Reisinger, "Apple App Store Hits 40 Billion Downloads; 20 Billion in 2012, Alone," CNET, January 7, 2013. URL: http://news.cnet.com/8301-13579_3-57562400-37/apple-app-store-hits-40-billion-downloads-20-billion-in-2012-alone/.

[4] "CISCO Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2012-2017," CISCO, February 3, 2013. URL: http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html.

[5] J. Wortham, "Customers Angered as iPhones Overload AT&T," September 26, 2009. URL: http://www.nytimes.com/2009/09/03/technology/companies/03att.html.

[6] T. Simonite, "The Great Bandwidth Brawl," MIT Technology Review, May 30, 2012. URL: http://www.technologyreview.com/news/427795/the-great-bandwidth-brawl/.

[7] A. Venkatraman, "Need for Increased Network Bandwidth is IT's Biggest Datacentre Pain-Point," Computer Weekly, September 26, 2012. URL: http://www.computerweekly.com/news/2240163911/Need-for-increased-network-bandwidth-is-ITs-biggest-datacentre-pain-point.

[8] Update Direct for Android, Pocket Soft. URL: http://pocketsoft.com/android_updatedirect.html

[9] S. Musil, "Google Play Enables Smart App Updates, Conserving Batteries," CNET News, August 16, 2012. URL: http://news.cnet.com/8301-1023_3-57495096-93/google-play-enables-smart-app-updates-conserving-batteries/.

[10] Android operating system. Source code. URL: http://source. android.com

[11] N. Samteladze and K. Christensen, “DELTA: Delta Encoding for Less Traffic for Apps,” *Proceedings of IEEE Conference on Local Computer Networks*, pp. 212-215, October 2012.

[12] N. Samteladze and K. Christensen, “DELTA++: Reducing the Size of Android Application Updates,” submitted to *IEEE Internet Computing* in February 2013.

[13] C. Percival, “Naive Differences of Executable Code,” draft paper dated 2003 and available from: http://www.daemonology.net/bsdiff.

[14] A. van Hoff and J. Payne, “General Diff Format Specification,” August 21, 1997. URL: http://www.w3.org/TR/NOTE-gdiff-19970901

[15] J. W. Hunt and M. D. McIlroy, “An Algorithm for Differential File Comparison,” Bell Laboratories, 1976.

[16] B. S. Baker, U. Manber, and R. Muth, "Compressing Differences of Executable Code," *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*, pp. 1-10. 1999.

[17] J. Hunt, K.-P. Vo, and W. Tichy, “An Empirical Study of Delta Algorithms,” *Software Configuration Management*, pp. 49-66, 1996.

[18] D. G. Korn and K.-P. Vo, “Engineering a Differencing and Compression Data Format,” *Proceedings of the Usenix Annual Technical Conference*, pp. 219-228, 2002.

[19] D. Trendafilov, N. Memon, and T. Suel, “zdelta: An Efficient Delta Compression Tool,” Technical Report TR-CIS-2002-02, Polytechnic University, June 2002.

[20] V. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals,” *Soviet Physics-Doklady*, vol. 10, no. 8, 1966.

[21] C. Percival, “Matching with Mismatches and Assorted Applications,” Ph.D. Dissertation, University of Oxford, 2006.

[22] J. MacDonald, “File System Support for Delta Compression,” Master’s thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[23] W. F. Tichy, “The String-to-String Correction Problem with Block Moves,” *ACM Transactions on Computer Systems (TOCS)*, 2, no. 4, pp. 309-321, 1984.

[24] J. W. Hunt and T. G. Szymanski, “A Fast Algorithm for Computing Longest Common Subsequences,” *Communications of the ACM,* 20, no. 5, pp. 350-353, 1977.

[25] R. C. Burns and D. Long, "In-place Reconstruction of Delta Compressed Files," *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 267-275. ACM, 1998.

[26] J. Seward, "The bzip2 and libbzip2 Official Home Page," 2013. URL: http://www.bzip.org.

[27] J. Gailly, "Zlib Compression Library". Available at http://www.gzip.org/zlib/.

[28] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-rate Coding," *IEEE Transactions on Information Theory,* 24, no. 5, pp. 530-536, 1978.

[29] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE* 40, no. 9, pp. 1098-1101, 1952.

[30] D. S. Taubman and M. W. Marcellin "JPEG2000: Standard for Interactive imaging," *Proceedings of the IEEE,* 90, no. 8, pp. 1336-1357, 2002.

[31] D. Le Gall, "MPEG: A Video Compression Standard for Multimedia Applications," *Communications of the ACM*, 34, no. 4, pp. 46-58, 1991.

[32] D. W. Jones, "Application of Splay Trees to Data Compression, "*Communications of the ACM*, 31, no. 8, pp. 996-1007, 1988.

[33] N. Kimura and S. Latifi, "A Survey on Data Compression in Wireless Sensor Networks," *IEEE International Conference on Information Technology: Coding and Computing 2005*, vol. 2, pp. 8-13, 2005.

[34] W.-T. Tan and A. Zakhor, "Real-Time Internet Video Using Error Resilient Scalable Compression and TCP-Friendly Transport Protocol," *IEEE Transactions on Multimedia,* 1, no. 2, pp. 172-186, 1999.

[35] "Delta Compression Application Programming Interfaces," Microsoft, October 2009. URL: http://msdn.microsoft.com/en-us/library/bb417345.aspx

[36] "Binary Delta Compression", Microsoft, March 2004. Available at http://www.microsoft.com/en-us/download/details.aspx?id=1562

[37] "The Chromium Project, Software Updates: Courgette," URL: http://dev.chromium.org/developers/design-documents/software-updates-courgette.

[38] B. Bing, "A Fast and Secure Framework for Over-the-Air Wireless Software Download Using Reconfigurable Mobile Devices," *Communications Magazine, IEEE,* 44, no. 6, pp. 58-63, 2006.

[39] B.C. Housel, G. Samaras, and D. B. Lindquist, "WebExpress: a Client/Intercept Based System for Optimizing Web Browsing in a Wireless Environment," *Mobile Networks and Applications*, 3, no. 4, pp. 419-431, 1998.

[40] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential Benefits of Delta Encoding and Data Compression for HTTP," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 4, pp. 181-194, ACM, 1997.

[41] M. C. Chan and T. Woo, "Cache-based Compaction: A new Technique for Optimizing Web Transfer," *Proceedings of Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, pp. 117-125, IEEE, 1999.

[42] J. Butler, W.-H. Lee, B. McQuade, and K. Mixter, "*A Proposal for Shared Dictionary Compression over HTTP,"* Google Inc., Tech. Rep., 2008.

[43] N. T. Spring and D. Wetherall, "A Protocol-Independent Technique for Eliminating Redundant Network Traffic," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4, pp. 87-95, ACM, 2000.

[44] J. Mogul, B. Krishnamurthy, and F. Douglis, "*RFC 3229: Delta encoding in HTTP*," IETF RFC, 2002.

[45] W. F. Tichy, "RCS - A System for Version Control," *Software: Practice and Experience*, 15, no. 7, pp. 637-654, 1985.

[46] I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding Personal Cloud Storage Services," *Proceedings of the 12th ACM SIGCOMM Conference on Internet Measurement*, pp. 481-494, 2012

[47] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey, "Redundancy Elimination Within Large Collections of Files," *Proceedings of the USENIX Annual Technical Conference*, pp. 59-72, 2004.

[48] A. Muthitacharoen, B. Chen, and D. Mazieres, "A Low-Bandwidth Network File System," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 174-187, ACM, 2001.

[49] M Stolikj, P. Cuijpers, and J. J. Lukkien, "Efficient Reprogramming of Wireless Sensor Networks Using Incremental Updates and Data Compression," 2012.

[50] M. Welsh, "Data Compression in Chrome Beta for Android," The Chromium Blog, March 5, 2013. URL: http://blog.chromium.org/2013/03/data-compression-in-chrome-beta-for.html

[51] N. Kimura and S. Latifi, "A Survey on Data Compression in Wireless Sensor Networks," *International Conference on Information Technology: Coding and Computing,* vol. 2, pp. 8-13, 2005.

[52] C. Welch, "Microsoft's Data Sense for Windows Phone 8," The Verge, October 29, 2012. URL: http://www.theverge.com/2012/10/29/3572048/microsoft-data-sense-windows-phone-8-announced.

[53] A. Fox and E. A. Brewer, "Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation," *Computer Networks and ISDN Systems,* 28, no. 7-11, pp. 1445-1456, 1996.

[54] S. Wolpin, "Amazon's Kindle Fire Is Blazingly Fast for a Mobile Device", January 24, 2012. URL: http://www.popsci.com/gadgets/article/2011-12/amazons-kindle-fire-blazingly-fast-mobile-device

[55] A. Balasubramanian, R. Mahajan, and A. Venkataramani, "Augmenting Mobile 3G Using Wi-Fi," *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 209-222, 2010.

[56] K. Lee, I. Rhee, J. Lee, S. Chong, and Y. Yi, "Mobile Data Offloading: How Much Can Wi-Fi Deliver?," *Proceedings of the 6th International Conference*, p. 26, 2010.

[57] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, "A First Look at Traffic on Smartphones," *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, pp. 281-287, November 2010.

[58] K.-K.Yap, T.-Y. Huang, M. Kobayashi, Y. Yiakoumis, N. McKeown, S. Katti, and G. Parulkar, "Making Use of All the Networks Around Us: A Case Study in Android," *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*, pp. 19-24, 2012.

[59] B. M. Duska, D. Marwood, and M. J. Feeley, "The Measured Access Characteristics of World Wide Web Client Proxy Caches," *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pp. 23-35, 1997.

[60] T. M. Kroeger, D. DE Long, and J. C. Mogul, "Exploring the Bounds of Web Latency Reduction From Caching and Prefetching," *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, vol. 12, pp. 13-22, 1997.

[61] J. Wang, "A Survey of Web Caching Schemes for the Internet," *ACM SIGCOMM Computer Communication Review*, 29, no. 5, pp. 36-46, 1999.

[62] "Github Book", Github, 2013. URL: http://git-scm.com/book/en/Git-Internals-Packfiles.

[63] "Background on CTIA's Semi-Annual Wireless Industry Survey," CITA, 2012. URL: http://files.ctia.org/pdf/CTIA_Survey_MY_2012_Graphics-_final.pdf.

[64] N. Samteladze, DELTA Software, January 2013. URL: https://github.com/NikolaiSamteladze.

[65] N. Samteladze, Previous Versions of Top 110 Free Android Applications, January 2013. URL: http://www.csee.usf.edu/~nsamteladze/.

[66] C. Percival, bsdiff/bspatch Source Code, January 2013. URL: http://www.daemonology.net/bsdiff/.