Koninklijke Landmacht

Unclassified

Master's thesis

Investigating the possibilities to implement efficient transfer of (MS) Office documents within the RNLA

Thesis number: 530

By Michiel Broekman

Ministry of Defense

The Royal Netherlands Army

Command & Control Support Centre

command & control

C2SC

support centre

Preface

This thesis is the result of my graduation project that has been carried out at the Command and Control Support Centre of the Royal Netherlands Army. This project completes my study of Computer Science at the Radboud University of Nijmegen. The past seven months have been a great experience and expanded my horizon as a computer scientist. For the first time I was challenged to conduct my own research for an extended period of time. Although this has been an individual project, a number of people have been very helpful to me.

First I would like to thank Jeroen Bruijning (Radboud University Nijmegen) for his intensive guidance, support and feedback throughout this project. His clear interests and close involvement during the project made the weekly meetings very pleasant. I also wish to thank Marco van der Meijden (TNO FEL) for giving me more specific information about the systems that are deployed in the RNLA and for supporting me with the technical aspects of these systems. I would like to thank lieutenant-colonel Bert Smid (RNLA) and major Edward van Dipten (RNLA) for guiding me through the organization in order to get the information I required for my research. I wish to thank the people of C&CZ of the Radboud University of Nijmegen for helping me to overcome some technical problems during the work that had to be done at the workstations.

Above all, I would like to thank my family for supporting me throughout these years and for giving me the opportunity to fully develop myself both personally and professionally.

Michiel Broekman
August 28, 2004

Summary

The research in this project is aimed at the possibilities to implement efficient transfer of (MS) Office documents within the RNLA. Programs like MS Word, MS PowerPoint and MS Excel are amongst others being used to plan, interact and operate. Therefore it is important that military personnel can send or update documents with their colleagues. The highly distributed nature of the military information systems does not allow standard file synchronization tools to be used.

Principally there are two factors in achieving an efficient data transfer: (1) sending data as compactly as possible and (2) finding some efficient distribution of this data. The distribution part of the problem will be omitted and can be paid attention to later on in some other research. Therefore the scope of the project is limited to a specific part of the general problem.

A field investigation has been done to get a deeper understanding of what sort of documents are used. These documents have been categorized and realistic data sets have been developed to get a close imitation of what could be encountered in the field. Most of the MS Office documents that are created and sent over the network are relatively big in file size. Unfortunately the military networks that are deployed do not have enough bandwidth to accommodate such intensive network traffic as some part of the bandwidth should always be reserved for other activities. Therefore in this project the principal focus lies on compression tools that can reduce file sizes substantially.

Almost anyone is familiar with the concept of file compression and a well known tool for achieving this is called ZIP. However there are more specialized tools available that accomplish a special kind of compression called delta compression. Delta compression is mostly concerned with efficient file transfer over slow communication links in the case where a receiving party already has a similar file (or files). Delta compression tries to find a minimal set of differences between the old file and the new file and these differences are placed in a so called delta file. Later on, this delta, which is typically smaller than the new file, can for example be sent to another computer which already has got the old file present after which this computer can patch the old file with the received delta to construct the new file.

In order to get an indication of whether compression tools and delta compression tools can be of value to the RNLA a number of representative (delta) compression tools are examined in great detail: ZIP, XMill, diff, Xdelta, Vcdiff and DeltaXML. Their techniques are scrutinized and the characteristics are described with respect to the previously developed data sets.

The results of the tests are the basis of the recommendations that are made towards the RNLA. These recommendations fall apart in a technical and non-technical part and conclude this thesis. Delta compression tools combined with tools like ZIP and XMill can provide a big advantage in achieving the goal of efficient file transfer. File sizes can decrease dramatically which eventually results in a more efficient bandwidth utilization. However that may be, in the first place it is important to minimize the file size of the original uncompressed files. This can be done by following practical guide-lines of what should be avoided when creating an Office document. At the other end an alternative for MS Office called OpenOffice.org should be considered as its applications would certainly contribute to smaller file sizes than those achieved by MS Office.

Of course (delta) compression and alternatives for MS Office only solve one part of the general problem and in the future there should be paid attention to smart distribution algorithms for distributing these deltas across the network. However the RNLA should first investigate what part of the communication links is utilized by Office documents to estimate the profits of introducing these kind of changes.

Table of contents

1 Introduction

The Command & Control Support Centre (C2SC) of the Royal Netherlands Army (RNLA) has been busy producing operational information systems that are based on network centric warfare concepts. The military networks that are deployed have one thing in common: the bandwidth of the network connections is very limited due to low capacity wireless links. Furthermore the bandwidth is shared with multiple applications like Voice over IP and video conferencing.

The people in all layers of the military organization are (or should be) all familiar with the MS Office standard. Programs like MS Word, MS PowerPoint and MS Excel are amongst others being used to plan, interact and operate. Therefore it is important that military personnel can send or update documents with their colleagues.

Most of the MS Office documents which are created and sent over the network are relatively big in file size. Unfortunately the military networks which are in use at the moment do not have enough bandwidth to accommodate such intensive network traffic. Despite of the limited network resources there are circumstances in which MS Office documents should be sent as quickly and efficiently as possible.

1.1 C2 Support Centre

The project is carried out at the Command & Control Support Centre (C2SC) in Ede. As described in [6] C2SC is the software development and management house of the RNLA for tactical C2-systems. Its mission statement is formulated as follows:

"The Command & Control Support Centre digitizes the battlefield for Commanders in an innovative and controlled way".

The RNLA has explicitly chosen for in-house development, production and management of C2-systems. The benefit of this approach is that the RNLA remains maximally involved in the process during the development and production phase of C2-systems. If there are any changing user requirements or budgets then adjustments can be made more directly and costly negotiations with industry are not required. To benefit from the knowledge and experience of the civil world and to incorporate state of the art technology civilian contractors are hired and COTS products are used where possible.

The ministry of defense of The Netherlands decided in June 2001 that The Netherlands should become a leader in Europe in the field of Command, Control, Communications & Intelligence (C3I). With the development of a common, European C3I-architecture, the development of new C2 supporting systems and with training and implementation support, the RNLA C2SC should make a contribution to this European initiative. Eventually the Dutch Centre of Excellence (CoE) of which C2SC is a part, should evolve into a European Centre of Excellence of C2 support. This initiative was offered to the EU as part of the Dutch contribution to the European Strategic Defense Initiative (ESDI).

1.2 Outline

This thesis describes the outcome of the graduation project that has been carried out at C2SC. Chapter 2 gives some introductory information to the real problem definition which is treated in chapter 3. In this latter chapter the scope of the project will be defined. The problem and questions that have to be answered are precisely formulated. After the reader has gained a clear understanding of the motivations behind this project chapter 4 will provide some information of various systems that are used in the

military to give some idea of what role information systems play in the field. Further there will be a limited field investigation about what kind of data is representative in the informational need. Chapter 5 will go into the details of compression tools and particularly focuses on some delta compression tools. Their underlying techniques and most important characteristics are described in detail. The results from the field investigation of chapter 4 and the (delta) compression tools from chapter 5 are used for the benchmark that is described in chapter 6. The various tools are measured on their input to get an understanding of which techniques are favorable in certain situations and which not. Finally, chapter 7 will use the benchmark results and conclusions for recommendations towards the RNLA. These recommendations fall apart in technical and non-technical recommendations and are the end of this thesis.

1.3 CD-ROM

All deliverables that are produced during this project are delivered on a CD-ROM. The CD-ROM contains the following:

- Project Management Plan
- Field Investigation
- Test Plan
- Data Sets, Compression Tools
- Master's Thesis

2 Preliminary research

The notion of low capacity (wireless) links and related file transfer is of course not only limited to the military situation. All over the internet networks must cope with intense traffic which may be just beneath the capacity the networks can handle. The general problem as it has been defined at the beginning of the project can be formulated as follows:

"Investigating the possibilities to implement efficient transfer of (MS) Office documents within the RNLA."

In the military there are all sorts of situations which can be encountered with respect to data transfer. Essentially these can be split up into two main categories: (1) file distribution and (2) file replication. These two categories are related to each other, but there are some important differences which are explained beneath.

To put it in simple words file distribution is about distributing files over a network and focuses on how data must find its way from one location to another. Within the context of this project file distribution is concerned with getting all clients up-to-date whenever they are connected to the network.

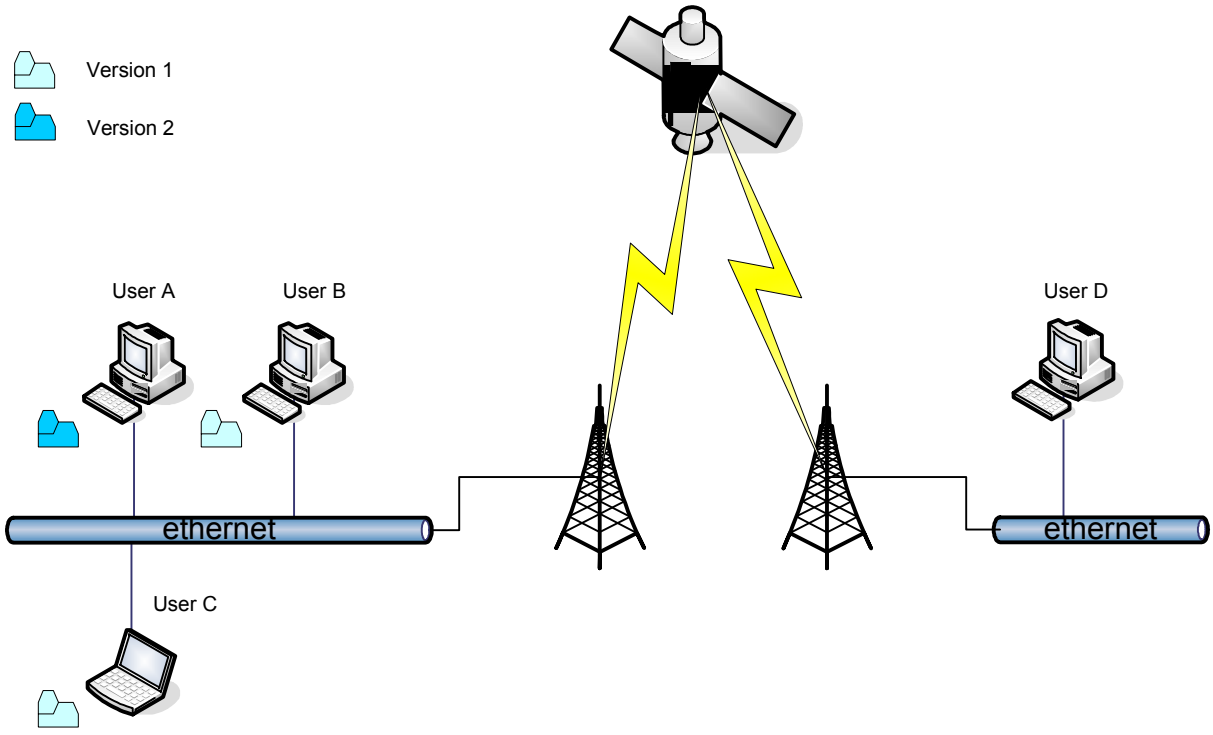

Figure 2-1: Four users want to get in sync with each other.

To get an idea of what this practically means figure 2-1 serves as an example. Suppose there are four users of whom user A, B and C already posses version 1 of a document. User A has made an update of the original document, so the newest version has become 2. The other users are all interested in this newest version and therefore want to synchronize, so in one way or another there must be found a way of providing the latest information to all users. It could be very inefficient to send version 2 to all interested parties as this could be very expensive in network bandwidth. For example consider the case where one wants to stay in sync with several clients that update their documents frequently. In the worst case sending complete documents simultaneously to all users could cause a network jam.

At this point terms like delta compression and file synchronization come in. The details of delta compression will be explained later but the idea can be formulated in the preceding context as follows. User A calculates the difference (delta) between versions 1 and 2 and sends this difference to all users who already have version 1 present, after which they can patch version 1 with the received delta. The users who do not have any version yet must get the complete version 2 or they can get version 1 from a neighboring host and successively patch version 1 with the difference between version 1 and version 2. So by sending only the essential differences, network bandwidth can be utilized more efficiently.

In our context file distribution is about finding smart distribution algorithms to figure out which users are up-to-date and which not. It is of overall importance to find a “smart” algorithm that finds out which users must be updated and subsequently searches for the best network path that should be followed to send the differential updates.

File replication is not really concerned with efficient distribution of data among several clients and usually does not make use of smart distribution algorithms to propagate data. For it is only focused on replicating data between two or more file servers. File synchronization tries to solve the problem of replicating data between two parties where the sender does not have a copy of the files held by the receiver. So the sender (server) does not have access to some sort of reference and uses a protocol between the parties which enables the receiver (client) to update its version to the newest one while minimizing communication between the two parties [1]. Because of the significant difference with delta compression the algorithms are rather different from those for delta compression.

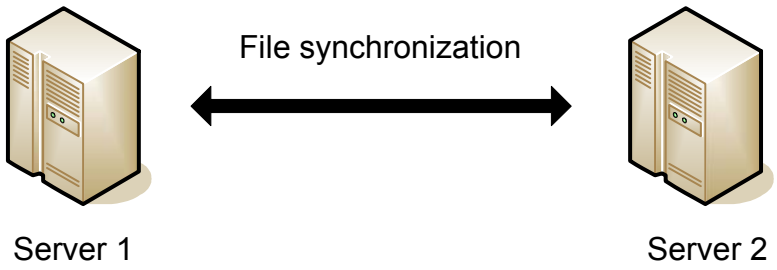

Figure 2-2: File synchronization between two servers.

A few relevant applications of file synchronization are: (1) Synchronization of user files, (2) Remote Backup of Massive Data Sets, (3) Web Access and (4) Distributed and Peer-to-Peer Systems. An example of a software package that enables synchronization of user files is Microsoft’s ActiveSync. It allows synchronization between several devices (like desktops and mobile devices). Synchronization can also be used for remote backup of data sets that have not changed too much between backups. In the case of Web Access file synchronization tries to achieve efficient HTTP transfer between clients and a server or proxy. By using file synchronization protocols the server does not need to keep track of the old versions held by the client. In distributed and peer-to-peer systems file synchronization can be used to update users who have been unavailable for some time and have to update their data while rejoining the system. Highly distributed data may be synchronized in the future, however at the moment there do not seem to be any promising applications that can cope with the complex networks and highly distributed data of the military. However it may be wise to watch the progress of these applications as they may become meaningful for the RNLA.

The famous file synchronization tool called rsync uses the rsync algorithm which provides a method for bringing remote files into sync [26]. It achieves this by sending just the differences in the files across the link, without requiring that both sets of files are present at one of the ends of the link beforehand. This tool however is not suitable for the highly distributed data in the military situation as rsync can not cope

with the inherent complex network structures and ways of data distribution. In the case where simple synchronization of two servers is of importance rsync will do its job fine.

3 Problem definition

The research in this thesis is aimed at the possibilities to implement efficient transfer of (MS) Office documents within the RNLA. In the preceding chapter an important point was made: the highly distributed nature of the military information systems does not allow standard file synchronization tools to be used. The scope of the project must then be limited to a part of the general problem which can be used for further research in this area.

Principally there are two factors in achieving an efficient data transfer: (1) sending data as compactly as possible and (2) finding some efficient distribution of this data. The distribution part of the problem will be omitted and can be paid attention to later on in some other research. With respect to the informational need of the RNLA the goal is to examine in what way data can be sent as compactly as possible. To structure the approach of this research a few questions need to be answered first and are formulated as follows.

- What sort of systems does the RNLA use for its purposes? There are many kind of systems that are used in the field. It is interesting to get some idea of what these systems do and what sort of information is needed.

- What sort of documents are we dealing with? In the beginning MS Office documents were explicitly mentioned as they play an important role in the planning, interaction and operation in the field. By doing a field investigation some deeper understanding of the actual documents may be gained.

- What are the exact criteria to categorize these documents? Of course there are many documents around and these should be categorized on certain criteria (like file size and frequency). It is of overall importance to get an good overview of the meta information concerning these documents.

- What sort of compression tools do exist? There are many sorts of compression tools and some promise very good compression rates. A special kind of compression called delta compression may be promising and deserves to be examined thoroughly.

- Which results are gained by using these compression tools on realistic data sets? All compression tools have their own characteristics and these must be examined. Therefore a benchmark must be developed for testing all representative compression tools on specially developed data sets.

- What recommendations can be made towards the RNLA? When all tests have been finished the results must be examined and interpreted. At the end of this project recommendations should be made towards the RNLA in the matter of what strategy should be handled when using compression tools. The results and conclusions can be used for further research. Besides the technical aspects of these recommendations there also should be paid attention to non-technical alternatives of improving efficient data transfer.

The next chapters will cover the listed items above and will try to answer the questions as completely as possible.

4 Field investigation

To get some knowledge about the kind of data that is sent it is necessary to investigate a real life situation. The military use several systems which all fulfill a specific purpose in the informational need. Before going into detail it is worth mentioning that all these systems are part of what is called network centric warfare.

4.1 Network Centric Warfare

Network centric warfare has got its roots in the 1990-1991 U.S. war against Iraq which is commonly considered the first information war [12]. This form of warfare is a concept that makes use of the advantages of the technology in the information age. Network centric warfare provides a force with a new sort of information that was previously an unavailable region of the information domain. Having access to this region gives modern warfighters a new type of information advantage. This is achieved by the advantage of information sharing which is made possible by networking. Due to this information sharing warfighters can get a better shared situational awareness and knowledge. The combat power that is realized by network centric warfare is related to the relationships between the physical, the information and the cognitive domains [13].

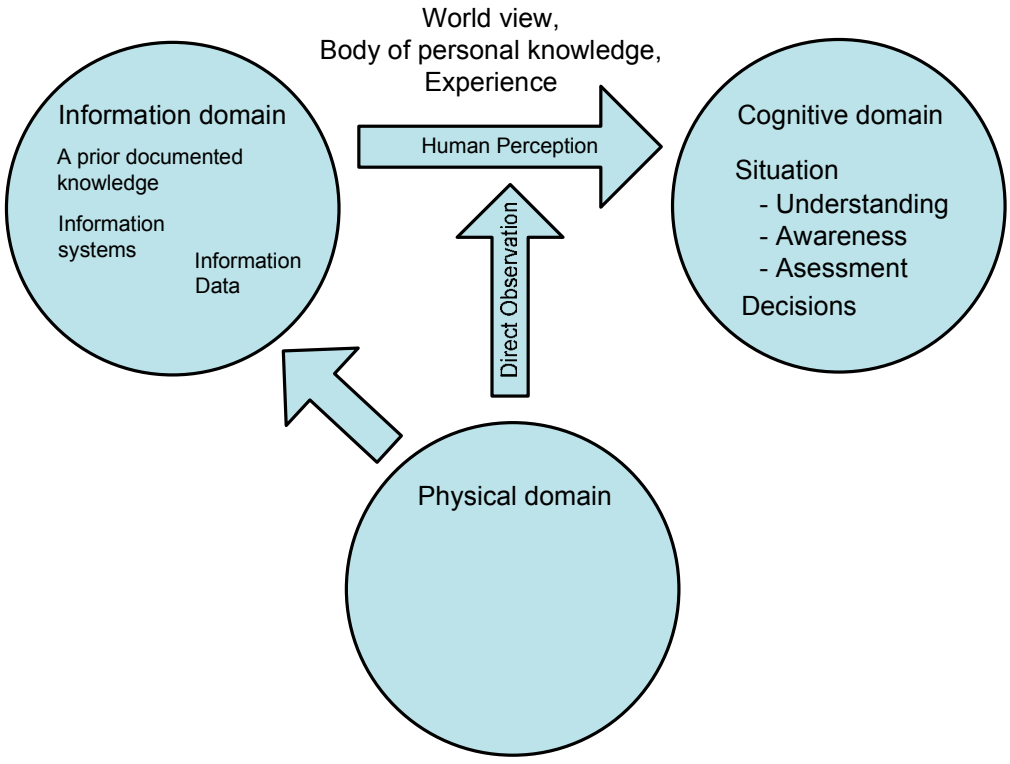

Figure 4-1: Domains concerning network centric warfare [13].

- Physical domain: the physical domain is the traditional domain of warfare. It is where real battles are fought and where all actions take place across ground, sea and air. It is also the domain where physical platforms and communication networks reside. The elements of this domain can be easily observed and combat power has in the past always been measured in this domain.

- Information domain: the information domain is the domain where all information resides. It is the domain where information is created, manipulated and shared. It is responsible for providing the

information to the warfighters. The command and control of modern military forces communicate in this domain. This domain is of utmost importance in the battle for information superiority.

- Cognitive domain: the cognitive domain is related to the mind of the warfighters. Leadership, morale, unit cohesion, level of training, experience, situational awareness and public opinion are some elements of this domain. The commander's intent, doctrine, tactics, techniques and procedures also reside here.

Battle space entities like aircrafts, ships, tanks and intelligence sensors are all connected to each other as well as to command and control centres. By connecting all these parts together a networking of platforms is achieved which can increase the effectiveness of a warfighter. The operational tempo is increased, accuracy is enhanced, survivability and lethality are increased.

Information technology is undergoing a change from platform-centric computing to network centric computing. This shift can be clearly observed in the growth of the internet, intranets and extranets [11]. All these technologies have led to the concept of network centric computing. Information is now distributed and can be used in a heterogeneous global computing environment. The military has also been influenced by these trends as battle time plays a very important role in the field. Battle time is heavily determined by the following two factors that are enabled by network centric warfare: (1) the measure of speed of command and (2) the measure of being able to organize forces from bottom up to meet the commander's intent. The forces get information superiority which results in a better awareness of the battle space. This means that excellent sensors, fast and powerful networks, display technology and sophisticated modeling and simulation capabilities are needed.

In figure 4-2 the emerging architecture for network centric warfare is depicted [13]. At the high end of the performance spectrum are cooperative sensing and engagement of high-speed targets. This can be realized by a high data rate and low latency of the information transport facilities. At the intermediate level there are several command and control activities, such as the coordination of tactical combat operations. These latter sort of operations can tolerate information delays on the order of some seconds. Other kinds of command and control and logistical operations, such as operational planning, are not that time sensitive. For example, if one would like to have information about a large container ship, which may need several days to get from one point to the other, delays on the order of minutes are tolerated. At the low end of the performance spectrum high level planning is of importance and therefore an overview of the total situation is necessary. The level of detail however is considerably smaller than that of the high end of the spectrum as the planning itself does not need a highly detailed overview of what happens in the field.

In the same way the wide variation in the importance and urgency is related to several levels of latency and priority. It is interesting to notice that there is a direct relationship between the velocity of information and the speed and tempo of operations in network centric warfare. Situational awareness should always be of great importance. To put it in simple words situational awareness is about three things: (1) Where am I? (2) Where are my buddies? and (3) Where is the enemy? Many more things can be said about network centric warfare and the interested reader is referred to a detailed overview in "Network centric warfare, Developing and Leveraging Information Superiority" [13].

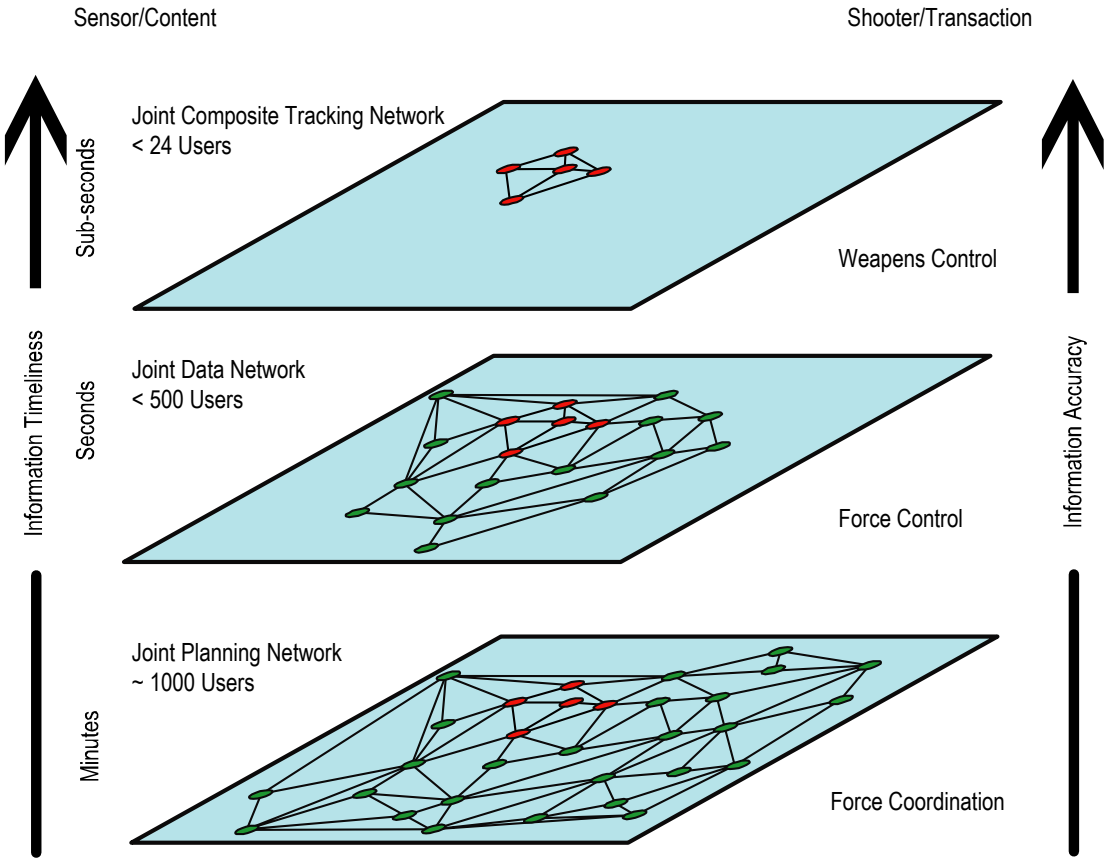

Figure 4-2: Architecture for network centric warfare [13].

4.2 File distribution

Realizing collaboration between several end systems is achieved by using hardware and software that can handle highly distributed data structures. In this section such a system is described briefly to give the reader some impression of what sort of systems within the RNLA exhibit these characteristics.

4.2.1 ISIS

The Integrated Staff Information System (ISIS) can be seen as an infrastructure, as well as a system, targeting a platform that can be used in the office and in the field [15]. It runs on MS Windows 2000 and MS Windows XP and brings MS Office applications, e-mail and specific C2 applications to the user. ISIS is meant to facilitate the informational need with respect to the decision making process and the forthcoming execution of plans. It is a distributed information system which enables the electronic communication of messages both in the office and in the field, at any location. To get some idea of what ISIS is all about some important facets will be briefly described beneath.

In every system information is of utmost importance and therefore also in the C2 workstation. The operations are highly dependant on the given information on activities in the area of operations, such as enemy sightings, status of bridges and roads and the own troop location. Planning information and orders must be exchanged along the command hierarchy. So it is necessary that everybody gets the information he needs. The information flow itself must be flexible which means that is should be possible to create a high level picture of the battle-zone, but at the same time detailed information should be available as well. This is where situational awareness comes in and is about the:

- Understanding of the locations, strength, intentions et cetera of the players in the area of operations.

- Ability to provide information about locations, strength, intensions et cetera of players in the area of operations.

- Ability to anticipate on changes of the players in the area of operations.

- Ability to plan an operation based on information in the area of operations.

Situational awareness is something that is within the perception of the mind. The Common Operational Picture (COP) is a way to provide this awareness. Principally the COP is a virtual storage place for all C2 information as it is distributed among several physical storage places at various organizational units. In the COP there are providers of information and consumers of information. When the consumer wants to consume information it needs to know what information is provided. The information provider is responsible for grouping and describing the information that is made available to consumers. The description of the information can be found in what is called the COP Catalog. The COP Catalog does not prescribe the content of the information but only prescribes that all information should be grouped and described. The idea behind the COP Catalog is that some user only needs the information he is interested in. So information is only sent when there is an explicit need for. When information at some point is requested the user will continuously be kept up to date with changes.

The idea above has both push- and pull-characteristics. First the user has to pull or define where he is interested in and has to make his interest available by using the COP Catalog. When the interest has been made clear the user will receive the information including all updates in the future. So the distribution is done on a need-to-know basis. The distribution mechanism that is used is called the publish/subscribe method. One major benefit of this method is that data will not be sent to users who are not interested, which eventually results in a better use of the bandwidth.

Besides that the COP Catalog owns the description of the COP information it also knows where to find the information because the location of the information and the provider's address are part of the description. The COP Catalog can basically be compared to the catalog of books in a library. Every organizational unit has at least one COP Catalog that can be accessed by other organizational units.

The 2D (as there is some progress in developing a 3D version) Geographical Information System looks like figure 4-3. It gives the reader an idea of how situational awareness is enhanced by maintaining an up-to-date overview of the battlefield. The exact details of ISIS will not be mentioned here and can be read in "C2WS3001 SUM ISIS 3.0 on C2WS" [15].

In ISIS it is also possible to make use of MS office applications. At the moment these documents are completely sent across the network by the publish/subscribe method. In the future a more efficient transfer could be achieved by using (delta) compression tools.

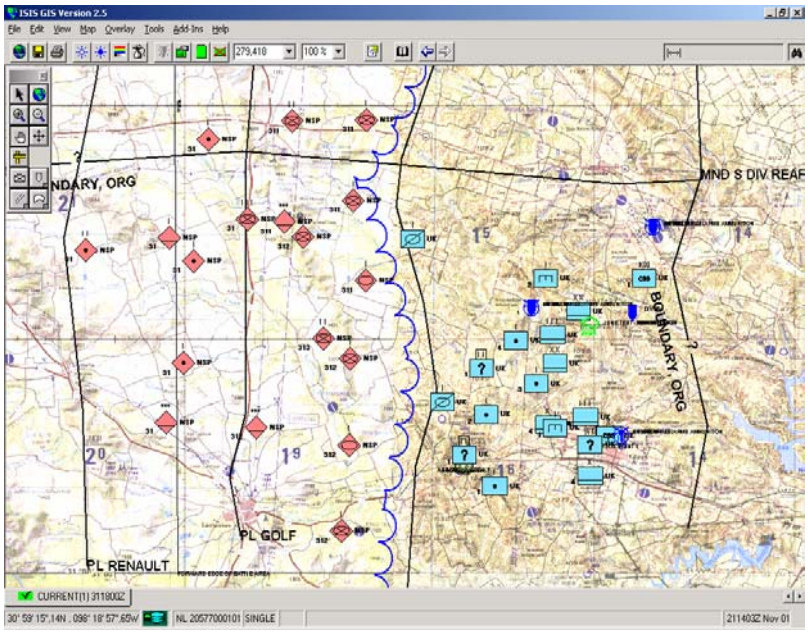

Figure 4-3: ISIS 2D GIS Viewer.

4.3 File replication

Realizing file replication between two or more servers is achieved by using some sort of file synchronization protocol. In the military servers need to stay in sync with each other and therefore a file synchronization tool is used. In the following section an example of such a case is given.

4.3.1 X-Share server

The structure of a military network is built up out of several basic components. The overall network however can have many forms as the network will be adapted to the hierarchical structure of the military organization on the spot. For instance there may be a military unit consisting of two command posts. Then the (simplified) structure of the network could be like the one depicted in figure 4-4.

Both command posts make use of a stand-alone X-Share server. The X-Share server is a special file server which is responsible for storing all files within the unit. It can be seen both as a central repository and a back up medium. The unit above consists of two command posts which must be able to operate independently. The two X-Shares of the unit are continuously being synchronized by using an Information Management Tool called Legato which is specialised in duplicating data between two file servers. The purpose of this synchronization is to get a full backup of the original file server in the case where the original file server falls out. The backup file server must then fully replace the one which is temporary out of operation.

4.4 Other systems

Of course there are many other applications that make use of the military network services. The examples above only give a partial impression of what sort of information is transferred over the network. Some (not yet mentioned) applications that have high data rates are for example VoIP and video conferencing. It may be clear that information superiority places a heavy burden on network

resources as all kind of military organizations, like the navy, army and the air force, claim their part of the available bandwidth.

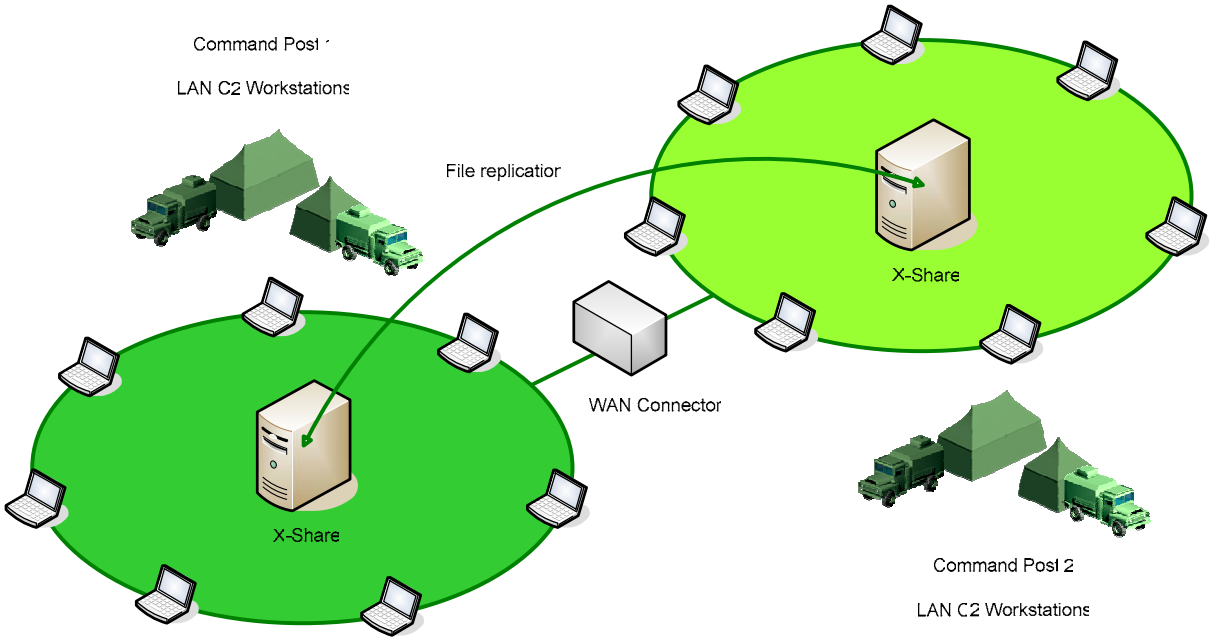

Figure 4-4: File replication between two command posts.

4.5 Field data

To get a more precise overview of the documents that are sent across the network it is important to investigate a real life situation. Therefore some servers of military data were queried to retrieve meta information of the files being stored. The CIS Control Centre in Stroe is responsible for delivering hardware and software support to military missions. The people of the centre have a real-time surveillance over all kind of operational systems in the field, so they are able to retrieve what sort of information is found at the various servers.

4.5.1 Queries

The queries beneath were all delivered to the CIS Control Centre and have been examined and answered where possible. Beforehand it is important to mention that the data in question is of a static nature: the servers that are queried are not part of a very complex distributed network. So the locally residing data is not used for complex file distribution. The following queries have been handed over to the CIS Control Centre:

- Get the total size of all files together.
- Get a listing of all types of files that are being stored.
- What is the minimum, average and largest size of those files that are being stored?
- How many duplicates can be found?

- What type of file is found most?
- What type of file is found least?
- What type of file is found regularly (between most and least)?
- Are there relatively many small files stored?
- How many large files are stored?
- Are there any extremely large files stored?
- What type of files are being updated most frequently (and what is their average file size)?
- What type of files are being updated least frequently (and what is their average file size)?
- How many files have (not) been modified in the last 24 hours?
- What type of files have (not) been modified in the last 24 hours?
- What is the relation between the update frequency and file size?
- Which part of the files being stored are not interesting in this investigation?
- If a file is older than for example 3 days, will this document then typically be updated in the future?
- Which part of the bandwidth is utilized by the transfer of documents?
- Are there any representative version histories and what is their content?

4.5.2 Results

The results are summarized in an Excel sheet. Due to confidentiality issues the exact overview is not shown in this document. Unfortunately the questions relating to the update frequency of documents and the bandwidth utilization could not be answered by the people who were responsible for retrieving the relevant data. By looking at the results in the Excel sheet the following insights can be obtained, taking into account that the situation is different for the various servers at different locations.

- The total size of all files together varies from 5 MB till 14 GB. 5 MB may not be very representative, so it is better to say the total size varies from 227 MB till 14 GB. It is important to understand that these numbers are only representative for the moment. File sizes grow when applications become more memory dependant (processing and data storage). This trend will hold on, so storage capacities must continuously be enlarged.
- The type of files that can be found at the file servers are: doc, txt, xls, ppt, mdb, rtf, jpg, tif, gif, bmp, mpg, avi, mov, exe, zip, html, lnk, bin/dbr/db, tmp, pdf, pst.

- The smallest file that is found is around 0 KB. The average file size varies from 188 KB till 17 MB. In almost any case the average file size is within the range of 188 KB till 1 MB. The largest stored files are above 50 MB and can even go up to 275 MB. Figure 4-5 gives an overview of the average file sizes on the servers. A remark should be made concerning the server that hosts 13 files. The average file size of this server is omitted as this size (17.430.652 bytes) is not representative for the whole.

- The amount of duplicates depends on the total amount of files stored at the specific server. Unfortunately the only information which is presented in the Excel sheet is the number of duplicates. This number does not say anything about how many duplicates there exist of a specific file. Having, for example, 1000 duplicates may mean that 1 file is duplicated 1000 times, but at the same time it could mean as well that 200 files are all duplicated 5 times. So the results of the CIS Control Centre do not contribute to a better understanding of the situation. Unfortunately more exact information could not be gained in time. Figure 4-6 shows the number of duplicates per server. Finding duplicates is done by using the CRC-check.

- The type of files that are found most are: doc, jpg, xls, ppt and html. pdf is also largely used on one server. pdf is probably stored on more servers, however the data are not complete for the pdf type.

- The type of files that are found least are: pst, bin/dbr/db, tmp, lnk, mov, avi, tif, mdb and mpg.

- The type of files that are found regularly are: zip, exe, bmp, gif, rtf, txt.

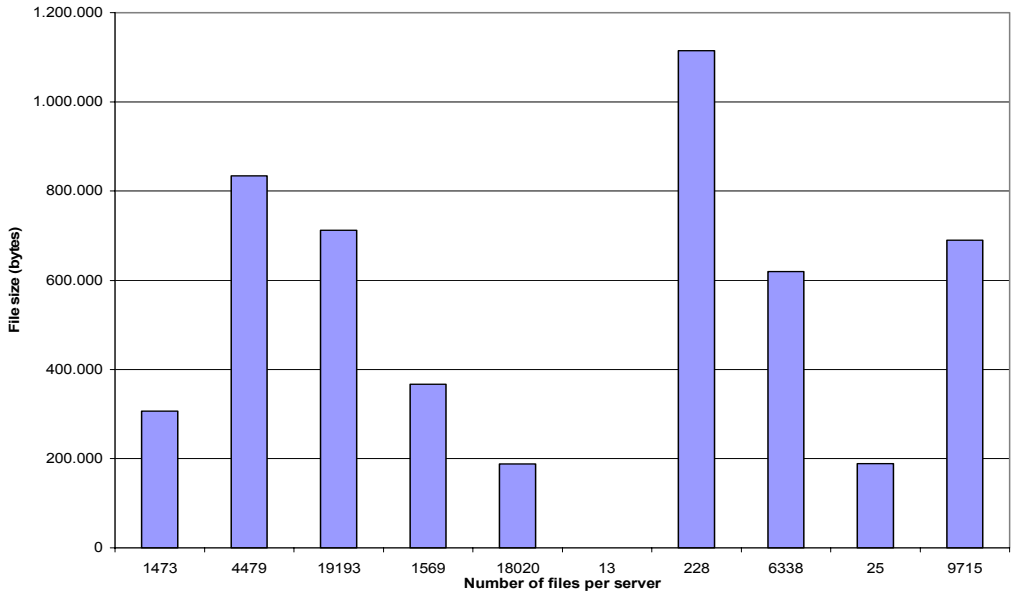

Figure 4-5: Average file size per server.

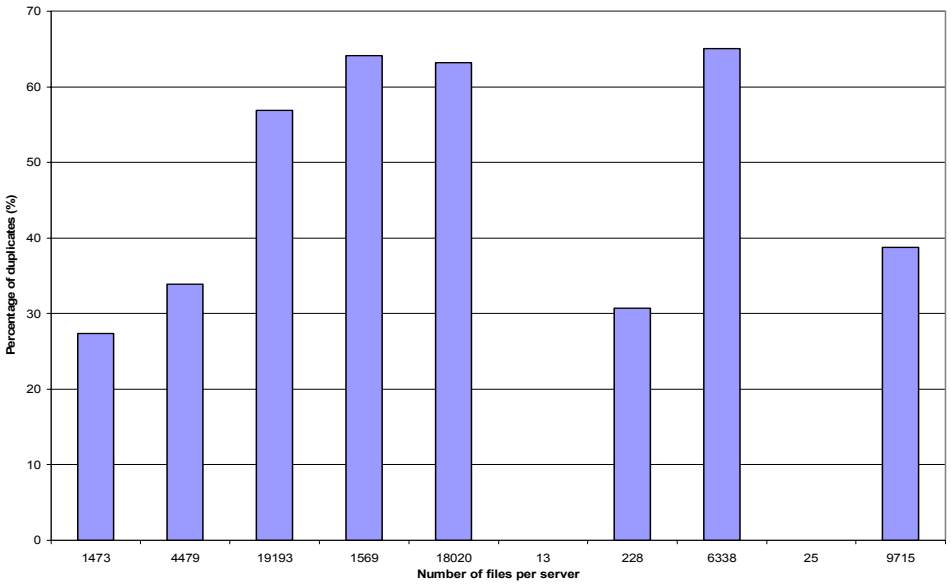

Figure 4-6: Number of duplicates per server.

- If you take the definition that a small file is less than 15 KB in file size then the percentage of small files varies from 0 % till 50 %. If the value 0 % would be excluded then 3 % may be a better lower limit. Figure 4-7 shows the various percentages.

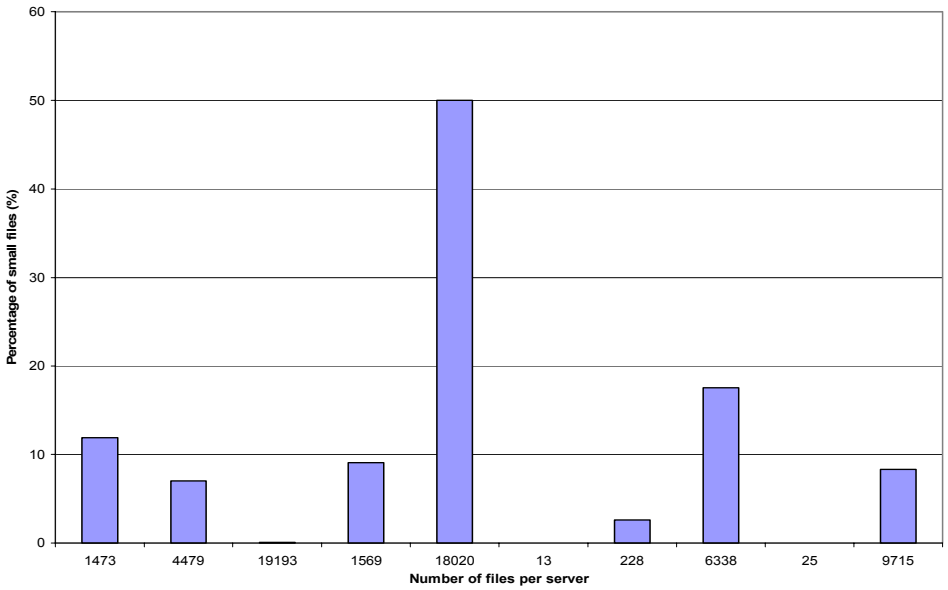

Figure 4-7: Percentage of small files per server.

- If you take the definition that a large file is bigger than 5 MB in file size then the number of large files essentially varies from 4.5 ‰ till 15 ‰ (the value 35 ‰ is excluded as it is relatively extreme). Of course some of the large files are even bigger than 50 MB (category extremely large). These files are fortunately very rare and can be discarded most of the time. A remark should be made concerning the server that hosts 13 files as all present files belong to the category big. This data is omitted as it would violate the layout of the figure 4-8.

- As said before there are files that are bigger than 50 MB. Such files are only found a couple of times on the server. On average these files may be found once or twice on a server with approximately 600 files. The type of files that are found in this case are: pst, ppt, mdb, avi, mpg, some doc files and some pdf files.

- It is without any doubt that doc files are updated the most frequently. The average file size of these frequently updated documents varies from 50 KB till 221 KB.

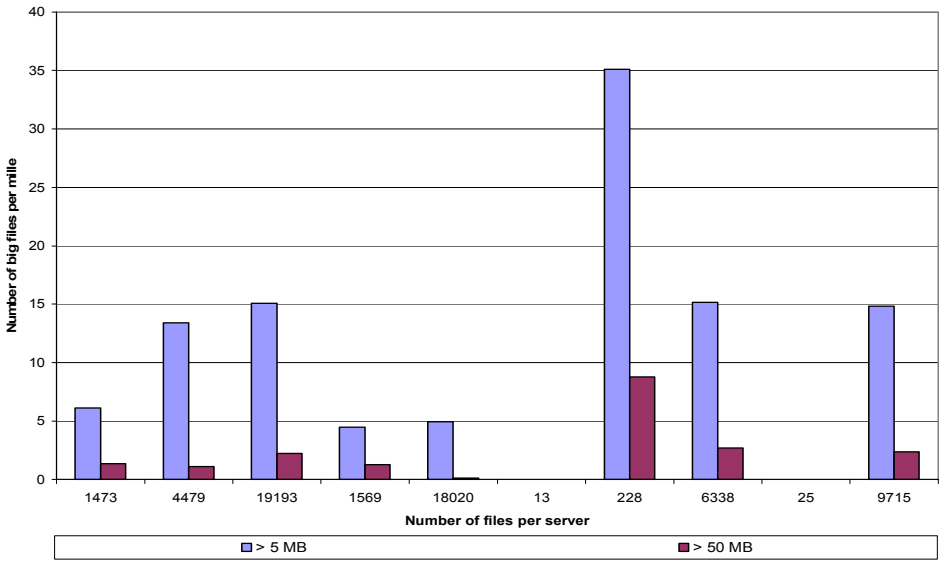

Figure 4-8: Number of big files per server per mille.

- Files that are updated least frequently are dbr files. There may be more files which are not updated regularly, however this data could not be retrieved from the Excel sheet.

- Files that have been modified in the last 24 hours are relatively rare. As can be observed in figure 4-9 there are cases in which there are no files modified in the last 24 hours. In all other cases the percentage of files that are modified in the last 24 hours ranges (all zeros excluded) between 0.2 % and 2.6 %. So in most cases files are not modified on a daily basis. Of course there are always files that will not be modified (and these are highly represented on the inspected servers), whereas doc files are more frequently changed.

- The type of files that have been modified in the last 24 hours are: doc, xls, jpg, gif, zip, html and mdb.

- All type of documents that are summarized in the beginning and that are not listed in the bullet above have not been updated in the last 24 hours.

- Unfortunately there has not been found any relation between the update frequency and the file size of a document: this data could not be extracted from the servers.

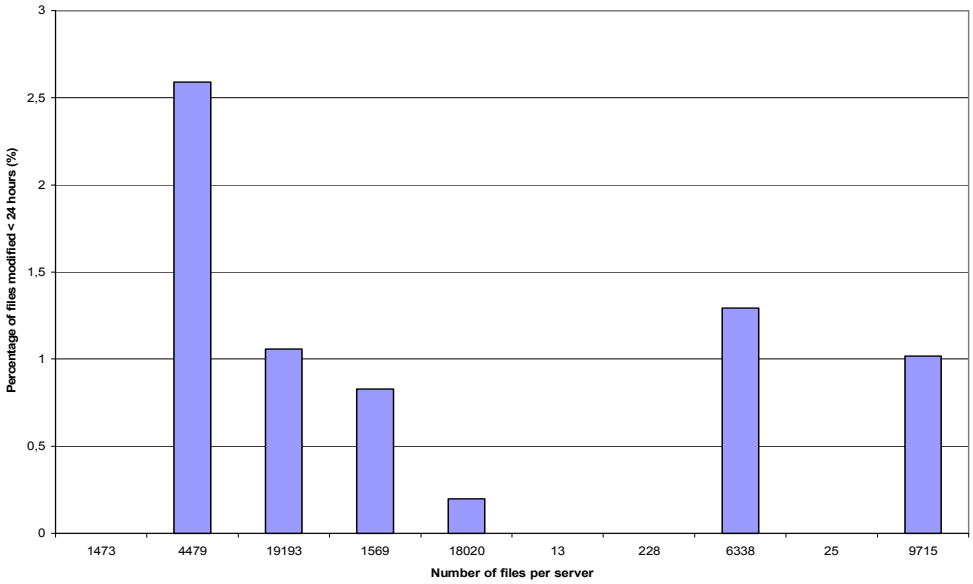

Figure 4-9: Percentage of files modified in the last 24 hours.

- The documents that have some version history are: daily INTSUMs (Intelligence Summary) which are presented in doc format, pst files, xls files and daily patrols and SITREPs (Situation Report) which are also in doc format.

- Another interesting point is that most files on the servers are between 15 KB and 5 MB in file size. This becomes clear when one looks at the statistical data: only a few files are smaller than 15 KB and even a much smaller amount of files are larger than 5 MB. So most files are in the 15 KB – 5 MB range.

5 Compression

Compression is widely used in computer network and storage systems in the case where efficient data transfer and storage is of major importance. Most of the techniques focus on the compression of individual files or certain types of data streams, like video or audio [1].

Nowadays, especially when one focuses on network-based environments, files are often widely replicated, frequently modified and cut and reassembled in different contexts and packages. There are many cases where the receiving party already has an earlier version of the transmitted file or some other file that is similar or where a few similar files are sent together. Examples are the distribution of software packages where the receiver already has an earlier version, the transmission of a set of documents which partially have the same structure or content (e.g. pages from the same web site) or remote file synchronization of a database. In these scenarios it may be clear that a more efficient strategy is feasible than that achieved by individually compressing files.

This chapter will first handle the tools and underlying techniques that enable individual file compression and will thereupon focus on the tools and underlying techniques that enable delta compression of which a first hint of its application area is already given away in the paragraph above.

5.1 Compression tools

In this section the well-known compression tool ZIP and its underlying technique is described. A more specialized tool called XMill will be treated in great detail as this tool may have certain important advantages over a standard general-purpose tool like ZIP.

5.1.1 ZIP

The ZIP file format is the most widely-used compression format in the world and makes use of the DEFLATE algorithm that combines LZ77 with Huffman coding [16].

The Lempel-Ziv compression methods are the most popular algorithms for lossless storage. LZ77 and LZ78 are the names for the two lossless data compression algorithms published in papers by Abraham Lempel and Jacob Ziv in 1977 and 1978. These two algorithms form the basis for most of the present LZ variations.

The LZ77 algorithm goes through the text from the beginning to the end and continuously maintains a history window of the most recent seen data and compares the current data that is being encoded with the data in the history window. So the previously seen text is then used as a dictionary. The compressed data stream contains references to the position in the history window, and the length of the match. In the case where a match could not be found the character itself is encoded in the stream after it has been flagged by a literal.

The main structure of LZ77 is a two-part sliding window of which the increasing part of the window represents the text that is already coded, while the decreasing part, which is called the look-ahead buffer, contains the text that is still to be encoded [17]. The incoming text is coded by tuples of the form (index, length, successor symbol). The index points to the location within the window on which a match is found with some of the text that is to be encoded. Length stands for the number of matching symbols and the successor symbol is the first symbol in the look-ahead buffer that does not match. At the moment the most popular LZ77 algorithm is DEFLATE which is the basis for ZIP. The other variant

called LZ78 will not be further described here but it is worth mentioning that instead of working with past data (like LZ77 does), LZ78 tries to work with future data and does this by forward scanning the input buffer and matching it against a dictionary. The LZ77 algorithm can be summarized as follows.

```
Fill the look-ahead buffer with symbols from the input stream;

while (look-ahead buffer is not empty)
{
        Find the longest match between the (start of the) look-ahead buffer and the already
        seen text. The length of the match is saved in the variable length;

        Output a tuple as described above to the output stream;

        Shift the contents of the window length + 1 symbols to the left, and fill empty bytes in
        the look-ahead buffer from the input stream;
}
```

The explained algorithm above might be somewhat difficult to understand and therefore the process is clarified by applying the LZ77 algorithm to the string “abracadabra”. Table 5-1 shows step by step what actions are taken to compress the string. POS relates to the coding position, MATCH shows the longest match in the window, CHAR shows the first char in the look-ahead buffer that does not match and OUTPUT presents the output in the form (index, length, successor symbol).

STEP	POS	MATCH	CHAR	OUTPUT
1	1	-	a	(0,0,a)
2	2	-	b	(0,0,b)
3	3	-	r	(0,0.r)
4	4	a	c	(3,1,c)
5	6	a	d	(2,1,d)
6	8	abra	EOF	(7,4,EOF)

Table 5-1: Output from and dictionary generated by LZ77.

So the table above gives an exact trace of how LZ77 operates on the input stream and what output eventually is returned. In figure 5-1 the window is shown at the final step of coding (step 5 to 6). The bold line in the window separates the already encoded text (dictionary) from the look-ahead buffer.

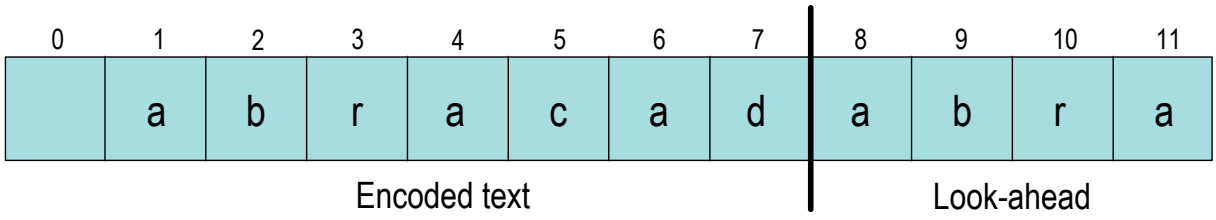

Figure 5-1: LZ77 Window at the final step of coding “abracadabra”.

Decoding is quite straightforward as the window is maintained in the same way. If P is the pointer to the match and C is the first character of the look-ahead buffer that did not match then in each step the algorithm gets a pair (P,C) from the input and outputs the sequence from the window specified by the pointer P and the character C.

As already explained before ZIP combines LZ77 with Huffman coding. In Huffman coding, codes are made by using a binary tree and this is done by the algorithm shown beneath. The statement in the while loop is executed until the number of nodes without a parent equals 1 as this will be the root of the tree.

Create a leaf node for every symbol, and let every node contain the probability of the occurrence of the symbol. The list of nodes is sorted on decreasing probability;

```
while (number_of_orphans > 1)
{
        Create a new node based on the two orphan nodes with lowest probability, and make it
        the parent of the two nodes. The content of the new node is the sum of probabilities for
        the previously orphan nodes;
}
```

Assign digits 0 and 1 to every left and right (or upper and lower, depending on the orientation of the tree) edge respectively;

To find the code of a symbol, follow each edge from the root node to the leaf node of the symbol, combining the digits on the passed edges;

This method has got the big advantage that the generated codes are directly decodable. Again the string “abracadabra” will be coded. If this string is coded without compression there are 3 bits for each symbol needed as the alphabet consists of 5 symbols. A simple calculation shows that the uncompressed string will occupy 11 x 3 = 33 bits. In information theory an information source has got symbols which are part of a finite alphabet. If information is to be measured in one or more symbols from a given source first one should know what is meant by information. Intuitively it is easy to understand that a seldom occurring symbol delivers more information than an often seen symbol. Therefore a measure for the information found by the occurrence of a given symbol is the inverse of the probability of the occurrence. Shannon has introduced the following formula which defines the information from the occurrence of symbol s_i as:

$$I(s_i) = \log \frac{1}{p(s_i)} = -\log p(s_i)$$

where $p(s_i)$ is the probability that symbol s_i occurs. The average information in a text is called the entropy and is defined as:

$$H = \sum_{i=1}^{n} p(s_i) I(s_i)$$

When using a base 2 logarithm the entropy formula returns the theoretical lower limit for the average number of bits per symbol that is needed to encode the stream of symbols. It would be out of scope to proof this assumption and a detailed description can be read in [18].

When returning to the example of “abracadabra” we can easily calculate the entropy of the string by filling in all variables in the formula above. Table 5-2 beneath first shows an overview of the information of each symbol.

SYMBOL S_l	COUNT	PROB $P(S_l)$	$I(S_l)$
a	5	0.455	1.138
b	2	0.182	2.459
r	2	0.182	2.459
d	1	0.091	3.459
c	1	0.091	3.459

Table 5-2: Statistical data for “abracadabra”.

When using the formula above the entropy of the string “abracadabra” becomes 2.040 and by executing the Huffman algorithm the following tree is built.

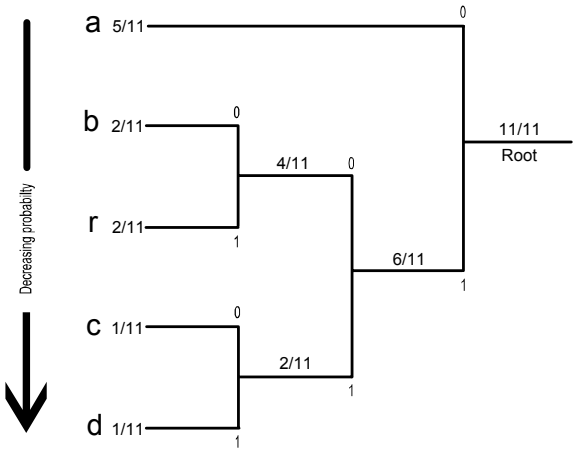

Figure 5-2: A Huffman tree for coding “abracadabra”.

The tree in figure 5-2 clearly shows that the nodes with the lowest probability should be combined. In the case of several nodes with equal probabilities there are multiple choices. It is easy to observe that characters with a high probability consume fewer bits whereas characters that have a low probability consume more bits. By going through the tree the following codes are obtained.

SYMBOL	CODE
a	0
b	100
c	101
d	110
e	111

Table 5-3: Huffman codes for “abracadabra”.

So the string “abracadabra” is compressed to the sequence “01001010111011001001010” which contains 23 bits. So on average one symbol consumes (23/11) 2.091 bits and this is pretty close to the entropy. If the string would be coded without compression then on average one symbol would consume (33/11) 3 bits, so Huffman coding certainly makes a difference.

So essentially ZIP makes both use of LZ77 and Huffman coding to compress data. LZ77 tries to seek for repeated data after which Huffman coding tries to reduce the number of bits necessary to represent an element of data. By using both techniques a more efficient compression ratio is realized.

5.1.2 XMill

The program XMill is a tool that is specially developed for compressing XML data. In [3] it is said that XMill usually achieves about twice the compression ratio of gzip (which is also based on LZ77 and Huffman coding) at roughly the same speed. XMill can be used with applications like XML data exchange and archiving. XML data may make use of a so called schema (such as a DTD or an XML-schema), however XMill does not necessarily need this schema information but it is able to exploit the present schema information to further enhance the compression ratio. In order to compress XML data XMill incorporates and combines several existing compressors: it makes use of zlib, the library function of ZIP, as well as a collection of data type specific compressors. It is also possible to fine-tune XMill by extending it with data specific compressors. There are already many specialized areas in which XML is used to exchange and store specific data: saving images and DNA sequences are some examples. Large organizations like the U.S. Army tend to migrate XML with their applications as well [19].

Before going into some of the details of XMill it may be wise to tell something about XML first. At the FAQ of the web site XML.com of O’Reilly [9] a simple introductory definition of XML is given: XML is a markup language for documents containing structured information. Structured information contains both content (words, pictures et cetera) and some indication of what role that content plays (for example, content in a section heading has a different meaning from content in a footnote). Practically all documents exhibit some sort of structure.

XML data is self-describing which means that the data describes itself by using a certain predefined schema. This feature increases the amount of data extremely but instead flexibility is gained. Although there have been many data exchange formats the last years XML will certainly become of great importance in the future (it already plays an important role). The reason XML has made such an amazing advance is because of its relation to the Web. Furthermore major companies like Microsoft have integrated XML in their applications which compels the world to make use of it. So its popularity is not because it is such a good and revolutionary idea, but because of its universal acceptance.

XML primarily consists of three kind of tokens: tags, attributes and data values. The following fragment gives an example of XML data:

```
<Book> <Title lang=”English”> Data compression </Title>
       <Year> 1995 </Year>
</Book>
```

In this example *Book*, *Title* and *Year* are tags. Each tag must be part of a begin-tag and an end-tag and this is clearly shown above. A begin-tag and an end-tag distinguish an element which on its turn can consist of elements as well (and/or data values). The element *Book* contains the elements *Title* and *Year*, and the content of *Year* is 1995. The data values are strings and an element may have a set of

attribute-value pairs. In the example above *@lang* is an attribute (to distinguish attributes from data values all attributes are preceded by the character @) of *Title* with as its value *English*. An XML document may also contain a processing instruction (PI), comments, CDATA values and a document type declaration (DTD).

It is important to understand that XMill is not a about a new algorithm. It is rather a new architecture which makes use of existing compressing algorithms and tools to compress XML data. Because of its open architecture XMill is extensible which means that users can add their self-developed compressors when needed. An example of this is when scientist must store data like DNA sequences or when medical personnel must store high resolution images of MRI scans. Specialized compressors can then be used for compressing parts that can not be compressed efficiently by using standard solutions. When knowledge about the XML data is gained, for example when two parties exchange XML data with a common DTD or XML-schema, settings of XMill can be changed to get an even better compression. An interesting fact is that by migrating data from other formats to XML the size of the compressed data decreases. When a native format is translated into XML the data usually expands because XML tags are verbose and must be repeated. However when XML data is compressed with XMill the compressed files are smaller than the gzipped data. The reason behind this is the way XMill groups data in so called containers which will be explained later. Of course the same compression rate can be achieved with a special purpose compressor, but such a compressor must be specially developed first. Thus by converting to XML both flexibility and efficiency (when compression is used) is gained.

XMill uses three principles to compress XML data:

- Separate structure from data: the structure consists of XML tags and attributes which form a tree. The data consists of a sequence of items (strings) representing element contents and attribute values. The structure and data are compressed separately.

- Group data items with related meaning: data items are grouped in containers, after which each container is compressed separately. For example, all <surname> items form one container and all <address> items form a second container.

- Apply different compressors to different containers: data items can consist of text, others of images, while others may consist of DNA sequences. Therefore XMill uses different specialized compressors (semantic compressors) for different containers.

To get an idea of what XMill can accomplish a simple example will be used as an illustration. Web Log files are compressed because these files increase in file size rapidly. A typical line (the two lines beneath are supposed to be on one line) in such a log file looks as follows:

```
202.239.238.16|GET / HTTP/1.0|text/html/200|1997/10/01-00:00:02|-|4478
|-|-|http://www02.so-net.or.jp/|Mozilla/3.01 [ja] (Win95; I)
```

Each line in the log file is a record consisting of eleven fields that are separated by a |: host, request line, content type et cetera. After a long time web logs of popular web sites can take huge amounts of space. A log file of 100.000 entries is approximately 16 MB in file size and can be reduced to 1.6 MB after it has been compressed with gzip.

weblog.dat	15.9 MB
weblog.dat.gz	1.6 MB

Programs that process a web log file are usually not very portable as different vendors use different formats. The same server can even be configured to generate different log formats. To get some flexibility the web log above can be converted into XML in the following way:

```
<apache:entry>
<apache:host>202.239.238.16</apache:host>
       <apache:requestLine>GET / HTTP/1.0</apache:requestLine>
       <apache:contentType>text/html</apache:contentType>
       <apache:statusCode>200</apache:statusCode>
       <apache:date>1997/10/01-00:00:02</apache:date>
       <apache:byteCount>4478</apache:byteCount>
       <apache:referer>http://www02.so-net.or.jp</apache:referer>
       <apache:userAgent>Mozilla/3.01 [ja] (Win95; I)</apache:userAgent>
</apache:entry>
```

Because field names and server type are very clear now an application can easily process the XML data. The major disadvantage of the XML format is that the file size increases dramatically, both for the XML file and its compressed version:

weblog.xml	24.2 MB
weblog.xml.gz	2.1 MB

The ultimate goal is to get the flexibility of XML without increasing the file size substantially. Therefore it is a good idea to compress data values separately. Compressing data values based on their tags results in all host values being compressed together, all request lines being compressed together et cetera. The nice thing is that gzip compresses better when it is applied to values of the same type than when applied to mixed values. The resulting compression will then become:

weblog.xml	24.2 MB
weblog.xmi	1.33 MB

In this case the XML file that is compressed with XMill is even smaller than the original gzipped file. But it is still possible to get an even better compression by giving XMill some explicit hints about the structure of the web log. The idea is to carefully inspect each field and use a specialized compressor for it. For example in the log file <apache:host> is an IP address and therefore can be stored by 4 unsigned bytes. The <apache:date> can also be stored more efficiently in binary. In some cases substrings can be taken out by closely looking at the structure of a certain value. In <apache:requestLine> most entries begin with GET and end in HTTP/1.0 (as HTTP/1.1 is also possible). The following hints can be added as an argument (settings.pz) to the XMill command line:

```
-p//apache:host=>seqcomb(u8 "." u8 "." u8 "." u8)
-p//apache:userAgent=>seq(e "/" e)
-p//apache:byteCount=>u
-p//apache:statusCode=>e
-p//apache:contentType=>e
-p//apache:requestLine=>seq("GET " rep ("/" e) " HTTP/1." e)
-p//apache:date=>seq(u "/" u8 "/" u8 "-" u8 ":" di ":" di)
-p//apache:referer=>or(seq("file:" t) seq("http://" or (seq(rep("." e) "/"
rep ("/" e)) rep ("." e))) t)
```

After settings.pz has been created the XMill command can be run in the following way:

```
xmill -f settings.pz weblog.xml weblog2.xmi
```

With these settings the compressed file size can be reduced to 0.82 MB. An overview of the file sizes is shown in table 5-4.

FILE	FILE SIZE
weblog.dat	15.9 MB
weblog.dat.gz	1.6 MB
weblog.xml	24.2 MB
weblog.xml.gz	2.1 MB
weblog.xml	24.2 MB
weblog.xmi	1.33 MB
weblog2.xmi	0.82 MB

Table 5-4: Overview of file sizes.

Of course the example of a web log is relatively very simple and in reality much more complex data can be compressed in the same way.

The architecture of XMill is based on the three principles described before. At first the XML file is parsed by a SAX (Simple API for XML) parser that sends tokens to the path processor. Each token (tags, attributes and data values) is thereupon assigned to a container. The tags and attributes which form the structure of the XML document are put in the structure container. Data values are put in various containers, in accordance with the path expressions, and the containers are compressed separately. Before a data value is put in a container a semantic compressor may compress the data value.

The core of XMill is the path processor which is responsible for mapping data values to containers. It is possible to control this mapping by adding some path expressions on the command line as an extra argument to XMill. For each XML data value the path processor checks its path against each container expression and so decides if the specific data value must be stored in an existing container or must be placed in a container that must be created beforehand. At the end each container is compressed separately with gzip after which it is stored in the output file.

Users have the possibility to associate semantic compressors with containers. Some atomic compressors can be used directly like differential compressors and binary encoding of integers. These simple compressors can further be used to form more complex ones. In special cases users can even build semantic compressors fully by their own, which is especially useful when XML data contains very specific data types like DNA sequences. It is not necessary to define semantic compressors and by default the text semantic compressor copies its input to the containers without any semantic compression.

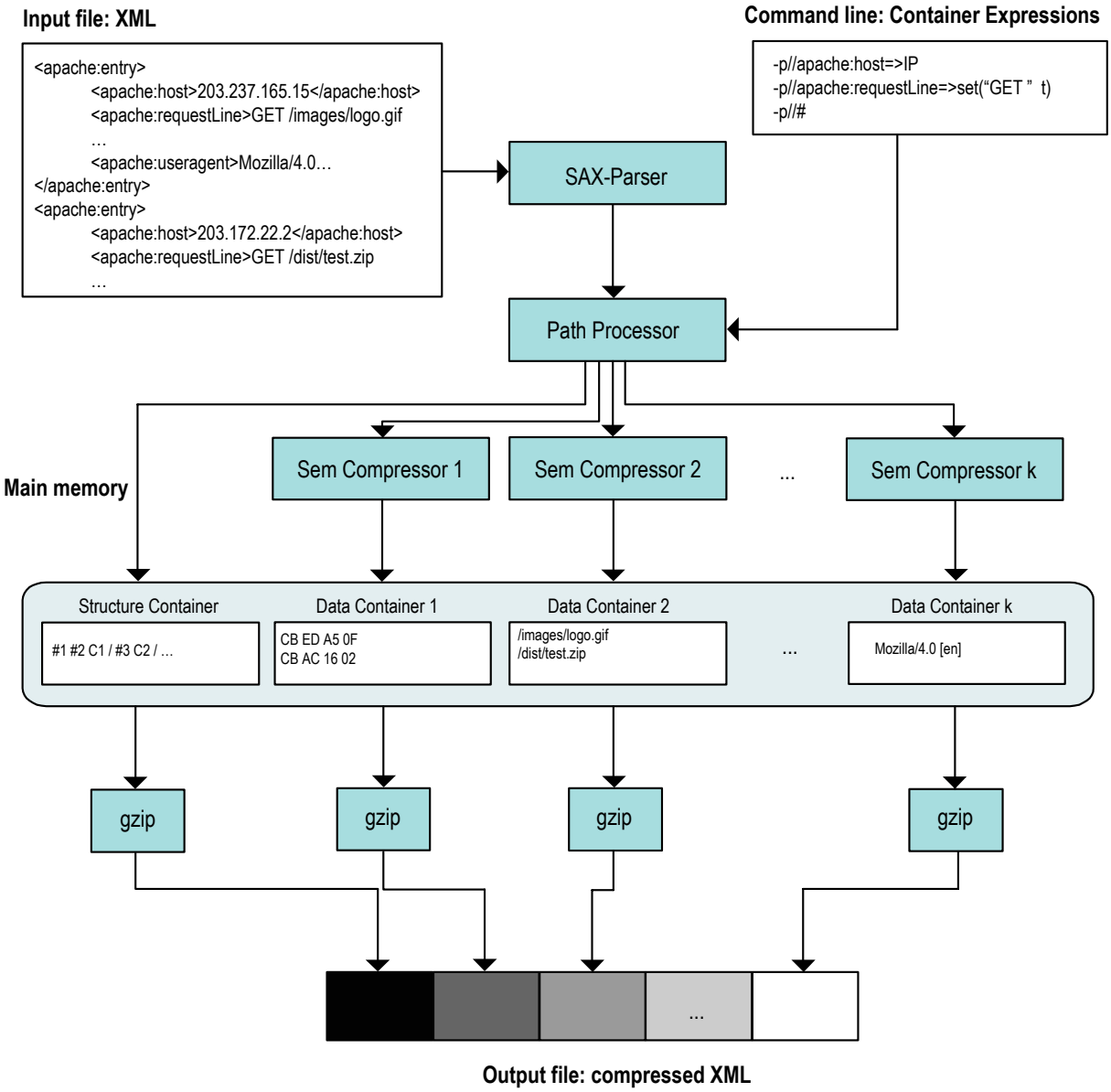

Figure 5-3: Architecture of XMill.

XML data consists of both structural elements and content and XMill separates these two to achieve an efficient compression. As described before XML uses tags and attributes to structure the content and this is tokenized in XMill as follows. Start-tags are encoded by assigning an integer value to them, while end-tags are replaced by the token /. Data values are replaced by their relating container number which can be illustrated with a simple example.

```
<Book> <Title lang="English"> Transaction Processing </Title>
       <Author> Gray </Author>
       <Author> Reiter </Author>
</Book>
```

Each data value is replaced by its container number and this number will be represented by a C followed by the container number.

```
<Book> <Title lang="C3"> C4 </Title> <Author> C5 </Author> <Author> C5
</Author> </Book>
```

From the example above it is clear that the @*lang* values are stored in container 3, the titles in container 4 and the authors in container 5. After replacing all tags and attributes the tokenized structure will be as follows:

```
Book = #1, Title = #2,  @lang = #3, Author = #4
Structure = #1 #2 #3 C1 / C2 / #4 C3 / #4 C3 / /
```

In reality the tokens above are encoded as integers (with 1, 2 or 4 bytes) so the structure above consumes 14 bytes. In the example above white spaces are ignored and the decompressor will produce standard indentation which is sufficient for most application. However it is also possible to preserve white spaces and in that case white spaces are stored in a special container. When preserving white spaces the structure above becomes as follows:

```
<Book> C1 <Title lang="C2"> C3 </Title> C1 <Author> C4 </Author> C1 <Author>
C4 </Author> C1 <Book>
```

By preserving white spaces the compressed file increases slightly: approximately 4 %. There are sorts of data, like a linguistic database, of which the increase is much higher (Treebank: 30 %) because of the deeply nested structures. Consider a large collection of books and for the sake of simplicity assume that they all share the same structure (*Title*, @*lang* attribute and two times an *Author*) then the structure container consists of repeated lines like the one beneath (white spaces are ignored).

```
#1 #2 #3 C1 / C2 /#4 C3 / #4 C3 //
#1 #2 #3 C1 / C2 /#4 C3 / #4 C3 //
…
```

The strings above can be compressed by gzip very well. A collection of 10.000 books can be compressed into 16 bytes if LZ77 uses a large enough window. With slight irregularities, like one author instead of two or a missing @*lang* good compression is still maintained. When the structure is regular the compressed structure will contain 1% - 3% of the compressed file, while an irregular structure (for example the linguistic database) will consume much more (Treebank: 20%).

Data values are uniquely assigned to one data container. This mapping is done by looking at the following information: (1) the data value's path and (2) the user-specified container expressions. The following example will clarify the two points above.

```
<Doc> <Book> <Title language="English"> Compression </Title>
             <Year> 1995 </Year>
       </Book>
       <Person> <Name> Tom </name>
                <Title> Mr. </Title>
                <Child> Tim </Child>
                <Child> Karen </Child>
       </Person>
</Doc>
```

In the example above the path to *Compression* is /Doc/Book/Title and the path to *English* is /Doc/Book/Title/@language. Container expressions are used to associate a data value with a container. One container is created for each tag or attribute. The container is specified by the last tag or attribute in the path, so there is one container for //Title and one for //@language. The expressions are XPath regular expressions. The mapping of tags and attributes to containers may be too restrictive and therefore a mapping is described from paths to containers with container expressions. This mapping is not treated here thoroughly , but a few simple examples may clarify its use. For example //Name creates one data container for all data values of which the path ends with *Name*. //Person/Title creates a container for all *Person Titles*. The // places all data values in one single container and //Person/# creates different containers for each tag under person.

Because certain values are not compressed very well by gzip semantic compressors are used. For example an IP address compressed by gzip does not come close to the standard 4 bytes per address. As XML data often comes with all sorts of values like integers, dates, airport codes et cetera special semantic compressors are used. As said before there are three sorts of semantic compressors: atomic, combined and user-defined. A brief description of these three will be given.

There are eight atomic compressors in XMill which are shown in table 5-5 beneath. The text compressor *t* only places the string to the container and leaves it unchanged after which it is compressed by gzip. The *u* compressor encodes positive integers in binary as follows: numbers less than 128 use one byte, those less than 16384 use two bytes and otherwise they use 4 bytes. The most significant one or two bits are used to indicate the length of the sequence. The integer compressor *u8* stores a number between 0 en 255 in one single byte. The other atomic compressors will not be described and can be read in [3].

COMPRESSOR	DESCRIPTION
t	default text compressor
i	compressor for integers
di	delta compressor for integers
e	enumeration (dictionary) encoder
u	compressor for positive integers
u8	compressor for positive integers < 256
rl	run-length encoder
"…"	constant compressor

Table 5-5: Atomic semantic compressors.

Combined compressors can be used where values exhibit some sort of structure. A nice example of this is the IP address which consists of 4 integers separated by a dot or the request line which always

begins with a GET followed by a string. XMill has three compressor combinators for compressing these sorts of values.

- Sequence compressor seq(s1 s2 …). For example seq(u8 "." u8 "." u8 "." u8) compresses an IP address as four integers. In order to simplify parsing all other semantic compressors (s1, s2, ….) must be constant.

- Alternate compressor or(s1 s2 …). For example take page references in a bibliography file. These can be 120–145 or a single page like 111. Then the composite compressor will be or(seq(u "-" u) u).

- Repetition compressor rep(d s). Here d is a delimiter and s is another semantic compressor. For example a sequence of comma separated keywords can be compressed by rep("," e).

User-defined compressors can be used when very specialized data types are part of the XML data (e.g. DNA sequences). A user can write its own compressor/decompressor and integrate it into XMill and XDemill, while making use of the Semantic Compressor API (SCAPI).

The SAX parser for XML uses callbacks that translate an XML file into a stream of events. There is a special event for each start-tag, end-tag, data value, attribute and attribute value. Each event, which essentially is a token, is sent to the path processor. The benefit of this way of parsing is that a complete internal representation of the XML file is not needed and when the window is full it will not result in any difficulties if the parsing is interrupted and at a later time resumed at the next token. The parser can buffer 64 KB of data to store the current token and data values that are too large are split up into several tokens. For example if *Author* in <Author> C5 </Author> would be too long to buffer the author would be split up into two or more values: <Author> C5 C5 </Author>. For each value the semantic compressor is evoked separately.

The path processor keeps track of the current path of each data value and evaluates each container expression on the path. This container expression is a regular expression and when successful the semantic compressor can be applied to the data value. This is the most time-critical part of the compressor and several different evaluation methods, which will not be treated, are used by XMill.

With respect to compression time the total processing time can be split up in two parts: (1) parsing and applying semantic compressors and (2) applying gzip. XMill saves time by applying gzip on smaller fragments and by regrouping data to further enhance the compression rate. There are four steps concerning decompression: (1) gunzip the containers, (2) interpret the XML structure and merge all data values, applying the appropriate semantic compressors which results in a stream of SAX events, (3) generate the XML-string (start-tags, end-tags, data values, attributes et cetera) and (4) output the uncompressed XML file. If an application would directly accept SAX events, instead of having to re-parse the XML-string then XMill would only need to go through step (1) and (2) which would make decompression much faster. Both compression and decompression time of XMill are linear in the size of the data.

In general XMill achieves better compression rates than gzip without any decreasing speed . It achieves better compression for data-like XML than for text-like XML. XMill is very suitable for data archiving purposes and in some cases data exchange. When data exchange is needed there are several factors that should be mentioned. Although XMill never looses to gzip its improvements depend on the following factors: (1) the type of exchange application and (2) the relative processor versus network speed. When facing a slow network XMill will always give some advantages as it compresses better than gzip. For a

fast network, one has to look at the following three exchange steps: (1) compression, (2) network transfer and (3) decompression. Compression is the most expensive and is approximately the same for XMill and gzip. Therefore the benefits rely more on the kind of application that is used. In the case of an end-to-end file transfer none of the two compressors is really better. When an XML file is published it only needs to be compressed once after which network travel en decompression are the final important factors that influence the total performance. XMill is only slightly faster than gzip. When data is directly imported into an application, then the decompression does not need to produce an output XML file. The only thing it has to do is to generate the SAX events. This makes XMill much faster than gzip. When network bandwidth is off overall importance XMill will surely help to gain some advantage by compressing the data as efficient as possible.

Some future improvements could be made to the compression/decompression time of XMill as the compression rate is already good. This seems very difficult as this could mean that zlib, the library of gzip, should be changed. Some research has been done by the inventors of XMill and it turned out that a time advantage would result in a less powerful compressor, so there should be a balance between time and space. It is interesting to mention that zlib consumes 50% of the compression time. If one would like to get a better time the path processor should be improved. In the case of decompression the bottleneck lies in merging the data from the different containers while interpreting the structure. So some future improvements could be achieved there. The interested reader can find a more detailed description of XMill in [3].

5.2 Delta compression tools

As already described in the beginning of this chapter there are cases where better compression should be achieved than that obtained by individually compressing each file.

Delta compression is mostly concerned with efficient file transfer over slow communication links in the case where a receiving party already has a similar file (or files) [1]. It becomes increasingly important in network-based applications where files are widely replicated, frequently modified and distributed over the network.

There will be a survey of software tools for delta compression. As told before delta compression can be used in the case where the sender knows all the files that are in the possession of the receiver. In reality however, there is the problem where the sender does not have a complete overview of the files held by the receiver. This is the field of remote file synchronization which will not be treated in this project and the interested reader is referred to [1]. Before going into the details of some representative delta compression tools it is interesting to get some insight of the fundamentals behind delta compression.

For example consider a server which distributes a software package. The client may already have some version of the software package present which enables an efficient distribution scheme. This scheme only sends a patch to the client that only contains the essential differences between the old version of the client and the new version which resides on the server. The server can compute the difference between the old version and the new version and outputs the difference which is called the delta. The computation of this delta or patch between two files is called delta compression or sometimes differential compression or delta encoding.

The problem of delta compression can be described mathematically as follows. The two files are represented by two strings f_{new} and $f_{old} \in \Sigma^*$ over some alphabet Σ (the methods described hereafter are character or byte oriented), and two computers C (client) and S (server) which are connected by a communication link. In the case of delta compression C has a copy of f_{old} and S has copies of both f_{new}

and f_{old}, so S must compute a file f^{δ} of a minimum size, such that C can build f_{new} from f_{old} en f^{δ}. The f^{δ} is also called the delta of f_{new} and f_{old}. In remote file synchronization, which will not be further treated, C has got a copy of f_{old} and only S has got a copy of f_{new}. So a protocol must be developed which results in C holding a copy of f_{new}. The communication costs between C and S must be minimized though. The old file f_{old} is also referred to as the reference file and the new file f_{new} is also called the current file.
The first work done in the area of delta compression is related to the string to string correction problem which is about finding the best sequence of insert, delete and update transformations that transform one string in the other (this idea is also exploited in the diff algorithm described next). The approach is about finding the longest common subsequence of the two strings. However this approach will not completely solve the problem of delta compression as it implicitly assumes that data common to f_{old} and f_{new} are in the same order in the two files. Another issue is that the string to string correction algorithm does not account for substrings in f_{old} which appear in f_{new} a couple of times (this will all become clear when the diff algorithm will be described in the next section).

At some point W. Tichy [20] came up with a string to string correction with block moves which resulted in a fundamental shift in the area of delta compression methods. In the string to string correction with block moves a block move is of the form of a tuple (p,q,l) such that $f_{old}[p,\ldots,p + l -1] = f_{new}[q,\ldots,q + l -1]$. This tuple represents a nonempty common substring of f_{new} and f_{old} of length l. The file f^{δ} can then be constructed as a minimal covering set of these tuples with the result that each element $f_{new}[i]$ ($f_{new}[i]$ defines the element at offset i of f_{new}) that is also part of f_{old} comes from exactly one block move. The f^{δ} which is the result of the longest common subsequence approach described above is a special case of a covering set of block moves. The question is how to minimize these block moves to construct a minimal f^{δ}. W. Tichy showed that a minimal cover set can be created and that this minimal set of block moves can be achieved in linear space and time. For more information about this subject the reader is referred to [20].

5.2.1 diff

The diff utility is a tool for keeping track of what has changed in a text file. As is described in [5] the diff command reports differences between files, expressed as a minimal list of line changes to bring either file into agreement with the other. It can also be used to distribute updates to files without redistributing the entire file which is the idea behind delta compression.

The diff command is also incorporated in a version control system that is called CVS [4]. The differences between different versions of a file in the CVS repository can be obtained by using the CVS (CVS diff) command. However, if you want to get the difference between two files that are not in the CVS repository, the stand-alone diff utility may be quite useful. The diff command is especially suitable for finding out the differences between the working copy and the backup copy of a specific source code file.

To give an illustration of the how diff operates the two text files beneath (text1.txt and text2.txt) are compared to each other.

```
Main Entry: jug·ger·naut
Pronunciation: 'j&-g&r-"not, -"nät
Function: noun
Etymology: Hindi JagannAth, literally, lord of the world, title of Vishnu
1 chiefly British : a large heavy truck
2 : a massive inexorable force, campaign, movement, or object that crushes
whatever is in its path <an advertising juggernaut> <a political juggernaut>
```

```
Main Entry: jug·ger·naut
Pronunciation: 'j&-g&r-"not, -"nät
Function: noun
Etymology: Hindi JagannAth, literally, lord of the world, title of Vishnu
1 chiefly American : a large heavy truck
2 : a massive inexorable force, campaign, movement, or object that crushes
whatever is in its path <an advertising juggernaut> <a political juggernaut>
```

The following command shows that files test1.txt and test2.txt differ at line number 5. The line starting with the less-than symbol is taken from the first file (test1.txt) and the line starting with the greater-than symbol is taken from the file test2.txt.

```
1 studs1>diff test1.txt test2.txt
5c5
< 1 chiefly British : a large heavy truck
---
> 1 chiefly American : a large heavy truck

2 studs1>
```

The first line of the output contains the character c (changed) that shows that line 5 in the first file is changed to line 5 in the second file. Of course much more complex comparisons can be done with the restriction that both input files are in plain text. The diff command can not show the differences between binaries, although it can return whether two binaries are different or not.

The technique behind diff is scrutinized by J.W. Hunt and M.D. McIlroy [5]. In their paper the basic aspects of differential file comparison with respect to the diff utility are discussed. It is very interesting to get some deeper understanding of how the underlying algorithm of diff operates. Therefore the most interesting insights that are explained in [5] will be treated now.

The program diff has been developed in such a way to make efficient use of both time and space on typically occurring inputs that are representative in version-to-version changes. Time and space usage vary about as the sum of the file lengths on real data. In the worst case however, they vary as the product of the file lengths.

The algorithm behind diff tries to find the “longest common subsequence” in order to find the lines that do not change between files. A subsequence can be obtained by deleting some or none symbols from a given sequence [23]. Given two sequences with lengths m and n, where $m \geq n$, the longest common subsequence problem is to find the common subsequence which has a maximal length among all common subsequences. For the sake of clarity the following example will show what is meant with a subsequence as the term may be still confusing. Consider the words “zen” and “zero knowledge”, then is the first word a subsequence of the second? The answer is yes and only in one way. This can be shown by capitalizing the subsequence: “ZEro kNowledge”. With respect to the longest common subsequence of diff each character in the example strings above are in reality separate lines of a text.

Efficiency is gained by focusing only on candidate matches between the files. To get a better performance techniques like hashing, presorting into equivalence classes, merging by binary search and dynamic storage allocation are incorporated.

To get to know what lines of one file have to be changed to bring it into agreement with the other file (or conversely) diff makes use of a list. Consider the two files that are horizontally listed beneath (the horizontally listed numbers indicate the line numbers of both files):

```
0     1     2     3     4     5     6     7     8

      t     r     m     q     o     p     h
      v     t     r     x     y     z     o     c
```

It is not difficult to see that the first file can be made into the second by the following procedure, in which an imaginary line 0 is at the beginning of each file.

```
append after line 0                     v
change lines 3 through 4, which were    m     q
into                                    x     y     z
delete lines 6 through 7, which were    p     h
append after line 7                     c
```

The first file can be obtained from the second in the following way:

```
delete line 8, which was                c
delete line 1, which was                v
change lines 4 through 6, which were    x     y     z
into                                    m     q
append after line 7                     p     h
```

The only actions that are used by diff are delete, change and append. The actions can be abbreviated by the letters d, c and a respectively. Another way of representing the operations above is shown beneath. The ‘<’ relates to the lines of the original file whereas ‘>’ relates to the derived file. It is easy to observe that the procedure of getting from one file to the other and vice versa is reversible: the append action can be exchanged with the delete action and the line numbers of the first file can be exchanged with those of the second file. It is an exercise for the reader to understand that when a longest common subsequence is found only three actions are required to transform the reference file into the current file.

```
0 a 1,1             1,1 d 0
> v                 < v
3,4 c 4,6           4,6 c 3,4
< m                 < x
< q                 < y
---                 < z
> x                 ---
> y                 > m
> z                 > q
6,7 d 7             7 a 6,7
< p                 > p
< h                 > h
7 a 8,8             8,8 d 7
> c                 < c
```

A simple idea of solving the longest common subsequence problem is to go through both files line by line until they differ, then search forward *in one way or another* in both files until a matching pair of lines is encountered and proceed in the same way. The problem lies in implementing *in one way or another*. Stripping matching lines from the beginning and the end may help when changes are not extremely pervasive as this will reduce some processing time. However the hard part of the problem (which is nonlinear) is still not solved of course.

There exists a very simple heuristic for *in one way or another*, which works pretty well when there are relatively few differences between files and relatively few duplications of lines within one file. When a difference is encountered, compare the k^{th} line ahead in each file with the *k* lines following the mismatch in the other for k = 1, 2 … until a match is found. When solving more difficult problems the method may not be appropriate. The value for k can be customarily limited to get some control over time and space. It is easy to understand that if the value for k is chosen too low then longer changed passages defeat resynchronization.

The following dynamic programming scheme makes use of a recursion to find the longest common subsequence. The lines of the first file are called A_i i = 1,…,m and the lines of the second file B_j j = 1,…,n. Then let P_{ij} be the length of the longest subsequence common to the first i lines of the first file and the first j lines of the second.

$$P_{i0} = 0 \quad i = 0,\ldots,m$$
$$P_{0j} = 0 \quad j = 0,\ldots,n$$

$$P_{ij} = \begin{cases} 1 + P_{i-1,\,j-1} & \text{if } A_i = B_j \\ \max\left(P_{i-1,\,j}, P_{i,\,j-1}\right) & \text{if } A_i \neq B_j \end{cases}$$

When looking at the recursive expression above it is easy to see that P_{mn} is the length of the longest common subsequence between the two files. If P_{mn} is calculated the indices of the elements of the longest common subsequence can be found quite easily. The time complexity of the program above is O(mn) and the space complexity is in the worst case even O(mn) which makes it less attractive. The algorithm above shows that each row P_i of the difference equation is gained from P_{i-1}.

The algorithm above can be improved by looking only at the essential matches. The essential matches that are called k-candidates by Hirschberg occur when $A_i = B_j$ and $P_{ij} > \max(P_{i-1,\,j}, P_{i,\,j-1})$. A k-candidate is a pair of indices (i,j) such that (1) $A_i = B_j$, (2) a longest common subsequence exists between the first i lines of the first file and the first j lines of the second, and (3) no common subsequence of length k exists when either i or j is reduced.

As a proof assume that both (i_1,j_1) and (i_2,j_2) with $i_1 < i_2$ are k-candidates, then $j_1 > j_2$. If $j_1 = j_2$ then (i_2, j_2) would violate condition (3). Also $j_1 < j_2$ would mean that the common subsequence of length k ending with (i_1, j_1) could be extended to a common subsequence of length k+1 with (i_2, j_2). The situation is visualized in the figure 5-4. The crossing lines indicate that (i1,j1) and (i2,j2) are part of two different common subsequences.

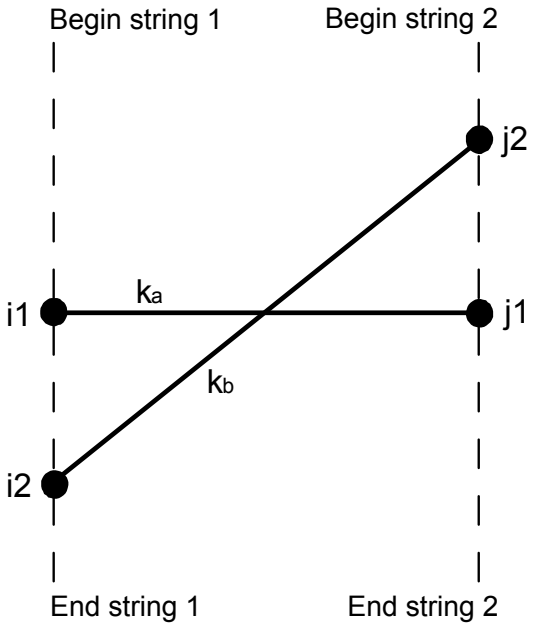

Figure 5-4: The pairs (i1,j1) and (i2,j2) are both k-candidates.

Figure 5-5 shows a graphical interpretation of the candidate methods. The dots represent grid points (i,j) for which $A_i = B_j$ is true. Any two horizontal or any two vertical lines in the figure have either no dots in common or have exactly identical dots, which means that the dots show an equivalence relation. In figure 5-5 a common subsequence is a collection of dots that is interconnected by a strictly monotone increasing (uninterrupted) curve. In total four of such curves are depicted in the figure. The values of k for these candidates are shown by the dashed curve that is strictly monotone decreasing. The total number of candidates is much less than mn and it turns out to be much less when real files are compared. So storing candidates will not be a big problem.

The diff utility stores the dots in linear space as follows:

(1) At first it creates lists of the equivalence classes of elements in the second file. The space complexity of these lists is O(n). This can be done by sorting the various lines of the second file. Table 5-6 represents the equivalence classes of “cbabac”.

CHARACTER	EQUIVALENCE CLASS
a	3,5
b	2,4
c	1,6

Table 5-6: Equivalence classes of “cbabac”.

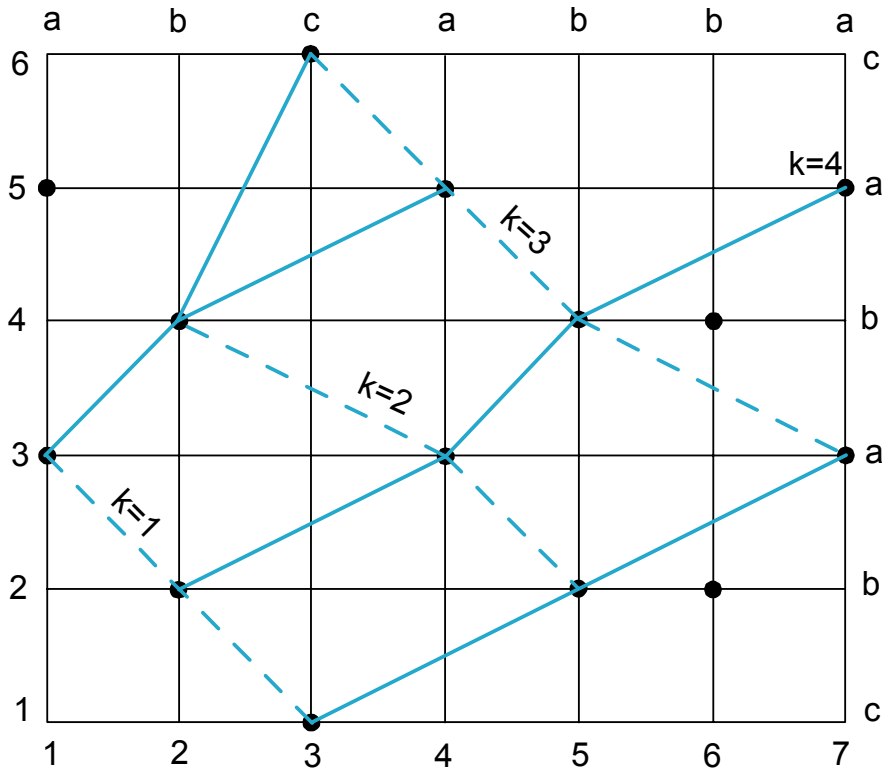

Figure 5-5: Common subsequences and candidates in comparing "abcabba" with "cbabac".

(2) Next the equivalence class must be associated with the elements of the first file. This association can be done in O(m) space and these relations are shown in the table 5-7. This results in a list of dots for each vertical.

ELEMENTS FIRST FILE	EQUIVALENCE CLASSES SECOND FILE
1	3,5
2	2,4
3	1,6
4	3,5
5	2,4
6	2,4
7	3,5

Table 5-7: Associations of equivalence classes.

After this the candidates are generated from left-to-right. Let K be a vector indicating the rightmost k-candidate that is yet seen for each k. The vector also includes a 0-candidate (0,0) and for all k that do not have a candidate yet a fence candidate (m+1,n+1). So K begins with (0,0) as its only value and is updated while moving right. Figure 5-6 clarifies all updates that are made during this process.

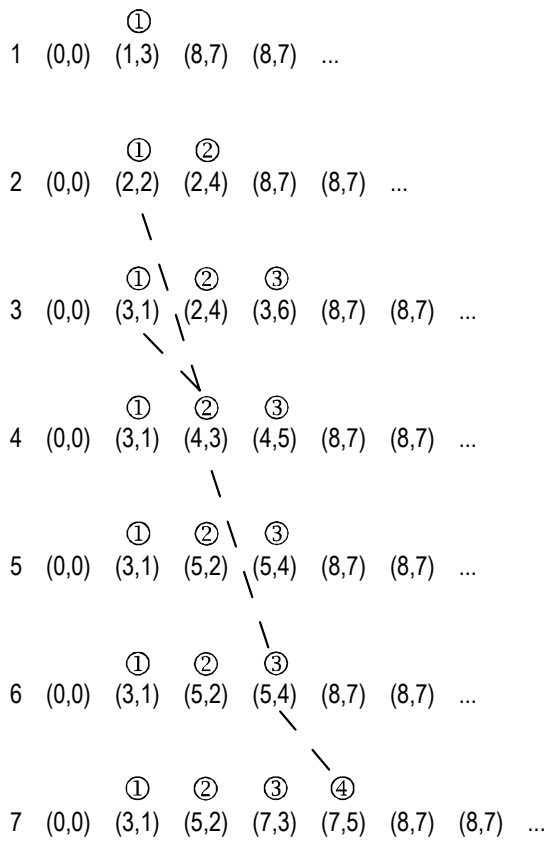

Figure 5-6: Generating candidates from left-to-right.

Each step corresponds with the number of the vertical in figure 5-5. So after processing the 4th vertical the list of rightmost candidates is visualized in step 4 above. A new k-candidate on the next vertical is the lowest dot that is located between the ordinates of the previous (k-1)- and k-candidates. In figure 5-5 two of those dots can be observed. These two dots replace the 2-candidate and 3-candidate and the following vector K is shown in step 5. The two dots on the 6th vertical do not fall between the ordinates in the list and are therefore no candidates. In figure 5-6 the method of finding a chain of candidates is visualized. Each new k-candidate is chained to the previous (k-1)-candidate to recover the longest common subsequence and this done as follows.

Begin by taking the k-candidate (i_k,j_k) with the largest value for k and find a (k-1)-candidate (i_{k-1},j_{k-1}) that satisfies the following rules: (1) $i_k > i_{k-1}$ and (2) $j_k > j_{k-1}$. So in figure 5-6 (7,5) (5,4) (4,3) (3,1) (the string “abac”) and (7,5) (5,4) (4,3) (2,2) (the string “abab”) are the only two longest common subsequences between the two given strings. So finding a chain of consecutive candidates is about comparing the coordinates of the candidates to each other. Figure 5-6 also shows that, while updating K, the value for i in some k-candidate (i_k,j_k) does not become smaller while the value for j in the same k-candidate does not become larger. This feature can be exploited in finding the longest common subsequence. Candidates are determined on a certain vertical by a specialized merge of the list of dots on that vertical into the current list of rightmost candidates. This merging, of which the exact details can be read in [5], dominates the worst case time complexity of diff.

It would be very inefficient to compare large files (with thousands of lines) without hashing each line in random access memory. Therefore diff hashes each line into one computer word. The drawback of this is that lines that are different may be looked at as if they were equal. If the hash function would be really random the probability of a false equality on a given comparison is 1/M when the hash values range

from 1 to M. So a longest common subsequence of length k which is determined by using hash values can contain k false matches. Diff solves the jackpot problem by checking the generated longest common subsequence against the original files, so false equalities are deleted.

In the worst case the diff algorithm is not really better than the simple dynamic program of before. The worst case complexity is for a large part the result of the merging and is O(mn log m). The worst case space complexity is dominated by the space that is used for the candidate list, which is O(mn). In reality diff works much better than it does in these worst cases and only in rare situations more than min(m,n) candidates are found. In most of the cases diff needs only linear space. Concerning time complexity the algorithm of diff is so fast that half the time is needed for simple character handling for hashing, jackpot checking and other simple operations, which are linear in the amount of characters in the two files. Sometimes diff loses from the simple algorithms (like the one described earlier) in trivial cases but in more difficult cases diff is a winner. For a complete description of the diff algorithm the reader is referred to [5].

5.2.2 LZ77-Based Delta Compressors

The best general-purpose delta compression tools are at the moment copy-based algorithms based on the Lempel-Ziv approach described earlier [1]. Two examples of such tools, which will be further looked at in this chapter are Xdelta and Vcdiff (the newer variant of Vdelta). These copy-based algorithms are based on a modification of zlib (general-purpose data compression library) and anyone who has read the section of ZIP must be able to understand the following description.

The idea behind LZ77-based compressors is to encode the current file by pointing to substrings in the reference file as well as in the already encoded part of the current file. To identify suitable matches during coding, two hash tables are used of which one is for the reference file, T_{old}, and the other for the already coded part of the current file, T_{new}. The table T_{new} is essentially built up in the same way as the table is done in LZ77, where new entries are inserted while encoding f_{new}. The table T_{old} though is completely built at the beginning by scanning f_{old}. By searching in both tables the algorithm tries to find the best match. A substring is hashed by looking at its first 3 characters, with chaining inside each hash bucket. First a short intermezzo concerning hashing is presented.

> Assume that a hash table maintains two arrays, one for keys, and one for values [16]. The elements of these arrays are referred to as buckets. To find the associated value for a certain key, a key is given to the hash function which outputs an integer (hash value). This integer is then the index to the associated value.
>
> A collision occurs when two or more different keys hash to the same integer and one technique for dealing with this is called chaining. Chaining is about using each bucket as a pointer to another structure, like an array, a linked list or another hash (preferably with another size and hash function).
>
> Chaining has got a big disadvantage because when more and more elements are added to the hash the O(1) complexity of hashes is lost. This in turn can be partially solved by rehashing: by increasing the table size and recomputing the hash values with respect to the table size the O(1) complexity can be remained.

If both the reference and current file fit into main memory and the hash tables are initially empty the algorithm basically goes as follows.

1) Preprocessing the reference file:

```
for (i = 0; i <= length(f_old) – 3; i++)
{
        hi = h(fold[i, i + 2]);

        Insert a pointer to position i into hash bucket hi of Told;
}
```

2) Encoding the current file:

Initialize pointers $p_1,\ldots, p_k$ to zero, say for k = 2

```
j = 0;

while (j <= length(f_new))
{
        h_j = h(f_new[j, j + 2]);

        Search hash bucket h_j in both T_old and T_new to find a "good match", that is, a substring in
        f_old or the already encoded part of f_new that has a common prefix of maximum length
        with the string at position j of f_new;

        Insert a pointer to position j into hash bucket h_j of T_new;

        If the match is of length at least 3, encode the position of the match relative to j if the
        match is in f_new, and relative to one of the pointers p_i if the match is in f_old. If several
        such matches of the same length are found then choose the one that has the smallest
        relative distance to position j in f_new or to one of the pointers into f_old. Also encode the
        length of the match and which pointer was used as reference. Increase j by the length
        of the match and updating some of the pointers p_i may give some better performance;

      If no match of length at least 3 is found then write out character f_new[j] and increase j by 1;
}
```

The whole algorithm is visualized in figure 5-7 on the following page. Of course there are still some issues concerning the implementation above. For example there are various ways of updating the pointers p_i in the case where the match is in f_{old}. The motivation for updating these pointers lies in the fact that in many cases the location of the next match from f_{old} is a short distance after the location of the preceding one. This is especially so if the two files are very similar. Vcdiff which will be described later uses a special method of updating these pointers.

In figure 5-7 both the reference file and the current file are depicted. The two tables are hash tables that are compared to each other to find suitable matches. The dashed lines represent the pointers which point to a specific location of the file in question. In the example there are exactly 4 matches to be observed. A remark should be made that in table T_{new} there are two matches with respect to the same hash bucket (h_9). However the best match is chosen and this match is between the current file and the old file as the length of the match is the longest.

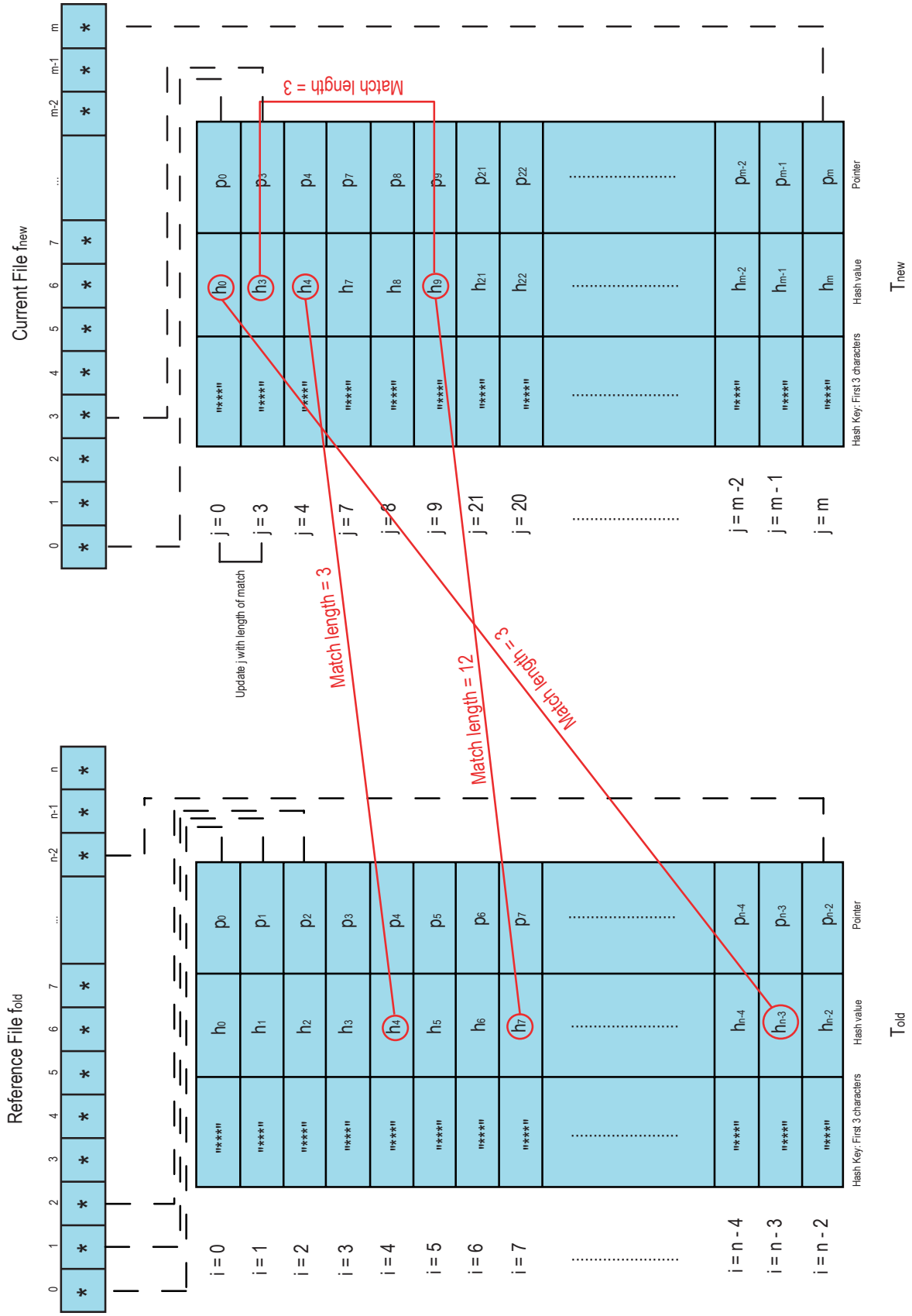

Figure 5-7: Visualization of LZ77-based copy-based delta compression algorithms.

5.2.3 Xdelta

The program Xdelta is a general-purpose delta compression tool engineered by the eXperimental Computing Facility (XCF) at Berkeley. Version 2 of Xdelta presents an application-level file system that is based on the Berkeley Database and is called the Xdelta File System (XDFS). XDFS is a solution for delta-compressed storage.

The file delta problem is about constructing a file t^d from the set of k^d *From* files $F^d = \{f^d_0 \ldots f^d_{k^d}\}$ [21]. The definition is more general than other presentations as F can be a set of files instead of only a single file. A file is defined as a sequence of bytes, so for a file f, size(f) specifies the length of f in bytes and for a set of files F, size(F) = $\sum_{f \in F} size(f)$. The notation f[i] specifies the byte at offset i of f, where $0 \leq i < size(f)$. The delta is split up into two parts: the Control C^d and insert-data I^d. The control contains a sequence of instructions $c^d_1 \ldots c^d_{z^d}$. The insert-data is a file which contains the concatenation of the data inserted by each insert instruction. Each instruction c_i can be one of the following two operations: copy(m,o,l) and insert(l). These operations are executed to create the *To* file, where copy inserts the substring $f_m[o] \ldots f_m[o+l-1]$ and insert inserts the next l bytes from I^d.

The program Xdelta uses an algorithm which achieves a linear space and time complexity. It uses a fingerprint function which is a hashing function for fixed length strings for which collisions are unlikely. The fingerprint algorithm uses a hashed index of the value of a fingerprint function at regular offsets in the reference file to achieve string matching. After this the current file is scanned while executing the copy/insert algorithm (this copy/insert algorithm looks like the LZ77-based delta compression approach described earlier and therefore will not be treated here).

Let s be a constant and a small power of 2 then the algorithm calculates a fingerprint on segments of length s in each file and at all offsets i divisible by s except, possibly, the last if s does not divide size(f) which means that the segment is too short. In each file the fingerprint function a_{fi} = fingerprint(f,i) is evaluated and inserted in a hash table B with a hash function H. A single pair (f,i) or a nil is placed in each hash bucket. Because the hash table does not chain hash collisions an array c_f is used for each file to index $c_f[i] = a_{fi}$, which makes it possible to detect a fingerprint collision. The complete algorithm is shown at the next page. The m in the algorithm refers to a specific f that is part of the set F as defined above.

The algorithm computes the fingerprint while going through the current file. If a large value for segment s is chosen the work done and space required by the algorithm is reduced. However less matches will be found in this case. A value that is chosen too small will result in copies that are so small that the efficiency will be less than a similar insert encoding.

Further it is worth mentioning that Xdelta itself does not compress the generated delta and leaves it up to the user. The interested reader should read [21] for a comprehensive description of Xdelta.

```
x ← fingerprint(t,i)                                    # Step 1: Fingerprint

if (B[H(x)] = nil)                                      # Step 2: Lookup
        return no match
else
        (m,o) ← B[H(x)]
If ( cfm [o] ≠ x)                                       # Step 3: Test for collision
        return no match
l ← length of longest matching substring at offsets o and i   # Step 4: Grow
if (l<s)
        return no match
return (m,o,l)
```

5.2.4 Vcdiff

Like Xdelta the program Vcdiff is a general-purpose delta compression tool that is based on the LZ77 copy-based algorithm. The program Vcdiff is also known as a Generic Differencing and Compression Data Format [22]. In many cases files are transported to machines with different architectures and performance characteristics, so the data should be encoded in a form that is portable and can be decoded with little or no knowledge of the encoders. This portability is the aim of Vcdiff.

Data differencing and data compression are traditionally treated as distinct types of data processing. However in Vdelta, the predecessor of Vcdiff, compression is seen as a special form of differencing in which the source data is empty (data compression is principally similar to differencing without the use of a source data). The idea is to combine the string parsing scheme used in the LZ77 style compressors with the block-move technique of W. Tichy [20]. This can be roughly summarized as follows: (1) Concatenate source and target file, (2) Parse the data from left to right as in LZ77 and make sure that a parsed segment starts the target data, (3) Start to output when the target data is reached.

Parsing is based on string matching algorithms which becomes a problem for large files. The way to solve this memory limitation problem is to split up the input file into windows. Still little has been done on researching efficient window schemes. The techniques that are also used in Vdelta use source and target windows that correspond to positions of the source and target files. The string matching and windowing algorithms have a big effect on the compression rate of delta and compressed files. Vcdiff however does not focus on such algorithms and uses a portable encoding format that is not dependant of such algorithms. This means that Vcdiff could be of great importance in the area of client-server applications where a server does not know the computing characteristics of the client it communicates with. Vcdiff is the first encoding format that addresses these issues and the following characteristics are achieved:

- Output compactness: the basic encoding format compactly represents compressed or delta files. The basic encoding format can be extended with secondary encoders to enhance the compression rate.

- Data portability: The basic encoding format is free from machine byte order and word size issues. This enables data to be encoded on one machine and decoded on a different machine with a different architecture.

- Algorithm generality: The decoding algorithm is not dependent of string matching and windowing algorithms. This allows competition among implementations of the encoder while keeping the same decoder.

- Decoding efficiency: Except for secondary encoders, the decoding algorithm runs in time proportional to the size of the target file and uses space proportional to the maximal window size. Vcdiff is different from more conventional compressors as it uses only byte-aligned data, thus avoiding bit-level operations, which improves decoding speed at the slight cost of compression efficiency.

As already mentioned above the basic data unit is a byte. Because of portability reasons Vcdiff limits a byte to its lower eight bits and this also counts for machines with larger bytes. There are specific ways in which Vcdiff deals with bytes and for detailed information the reader is referred to [22].

A large target file is split up into several target windows which are processed separately. The order of processing is based on the sequential order of the windows in the target file. A target window T of length t can be compared with a source segment data S with length s. This segment S has to come either from the source file or from a part of the target file that is earlier than the target window T. The values of T, t, S, and s are all the result of the window selection algorithm which has a big influence on the size of the encoding. Vcdiff encodes previously made choices which means that during decoding no knowledge of the window selection algorithm is required.

The j^{th} byte in S is written as S[j] and T[k] represents the k^{th} byte of T. With respect to the delta instructions the windows S and T are treated as substrings of a superstring U which is the result of the concatenation of S and T: S[0]S[1]...S[s-1]T[0]T[1]...T[t-1]. The instructions to encode the reconstruction of the target window are called delta instructions. There are three kind of delta instruction:

- ADD: This instruction contains two arguments, a size x and a sequence of x bytes that should be copied.

- COPY: This instruction contains two arguments, a size x and an address p which relates to the string U. The arguments specify the substring of U that should be copied. The substring must be fully present in either S or T.

- RUN: This instruction contains two arguments, a size x and byte b that is repeated x times.

The following example shows a source and target window in order to clarify the instructions above. The source string consists of 16 characters and the target string consists of 28 characters.

```
a b c d e f g h i j k l m n o p
a b c d w x y z e f g h e f g h e f g h e f g h z z z z
```

The delta instructions which encode the target window in terms of the source window are shown beneath.

```
COPY   4, 0
ADD    4, w x y z
COPY   4, 4
COPY   12, 24
RUN    4, z
```

The first letter a of the target window is at position 16 (the start position is 0) in U. The fourth instruction may be confusing: it copies from T itself, and position 24 in U relates to position 8 in T. The instructions above show that it is no problem that data to be copied is overlapped with data being copied from with the condition that the latter starts earlier. This also enables the efficient encoding of periodic sequences. The reconstruction of the target window is done by processing one delta compression after the other and copying the data from either the source window or the target window that is being reconstructed. This is all done by looking at the instruction type and the associated addresses.

A Vcdiff delta file starts with a header section after which a sequence of window sections follow. The header section uses specially reserved bytes to identify the file type and information which relates to data processing that goes beyond the basic encoding format. The window sections contain the encoded target windows. The exact way of how Vcdiff makes use of the header section and window sections is very interesting, but it would go too far to explain the mechanism in detail. The interested reader is referred to [22].

The addresses of COPY instructions are locations of matches and often occur a short distance from or exactly equal to one another. This is the result of that data in local regions is often replicated with some small changes. Therefore it would give some advantages to code a newly matched address against a recently matched address. Vcdiff maintains two different address caches to encode addresses of COPY instructions more efficiently. Both the encoder and the decoder are fully aware of these caches which means that the caches of the decoder stay synchronized with the caches of the encoder. The exact way Vcdiff handles these caches can be read in [22].

The matches between two files are most of the time short in lengths and are separated by small amounts of data that did not match. This practically means that the lengths of the COPY and ADD instructions are most of the time small. This scenario can for example be observed in binary data or structured data like HTML and XML. The compression can be enhanced by combining the encoding of the sizes and the instruction types as well as by combining the encoding of adjacent delta instructions with data sizes that are small enough. When such combinations should be performed depends on many factors like the data which is being processed and the string matching algorithm which is used. For example if a reasonable amount of COPY instructions have the same data size it may be a good idea to encode these instructions more compactly than other cases.

Vcdiff is specially designed so that a decoder does not have to know anything of the choices that are made in the encoding algorithms. This is realized by using a so called instruction code table. To get detailed knowledge of the format of a code table and how encoding and decoding are related to each other the reader is referred to [22].

Finally there can be said something about the performance of Vcdiff. The encoding format is compact and the string parsing strategy is based on LZ77 without any secondary compressors. The compression rate is close to gzip and for differencing decoding speed and encoding efficiency is quite good when compared to already existing methods. Vcdiff can be run in several modes which have their own specific performances:

- Vcdiff: Vcdiff is used as a compressor only.

- Vcdiff-d: Vcdiff is used as a differencer only which means that it compares target data against source data. Because files are large they are split up into windows. This means that each target window starting at some file offset in the target file is compared against a source window with the same file offset. The source window is kept somewhat larger than the target window to increase the possibility of matches.

- Vcdiff-dc: It is the same as Vcdiff-d but this version also compares target data against target data. This results in Vcdiff computing both differences and compressing data. The windowing algorithm stays the same with the exception of the hint given above.

- Vcdiff-dcw: It is the same as Vcdiff-dc, but now the windowing algorithm makes use of a content-based heuristic to select a source window that probably will make a better match with a given target window. In this case the source data segment that is selected for a target window will most of the time not be aligned with the file offsets of this target window.

In some cases the compression rate of Vcdiff is a little bit worse than gzip. However when Vcdiff is used as a delta compression tool it can produce very small deltas between two files versions that are very similar. Vcdiff-d and Vcdiff-dc both use the same window selection algorithm of aligning by file offsets but Vcdiff-dc also compresses the data which results in a smaller output. Vcdiff-dcw uses a content-based algorithm as well to find the best matches between the two file versions. Although it does its job pretty well, the algorithm is not always able to find the best matches.

5.2.5 DeltaXML

DeltaXML is a tool for comparing, merging and synchronizing XML documents. When comparing two XML documents DeltaXML generates a delta that is XML formatted and can be viewed instantly. The program itself is written in Java and therefore it can be easily integrated into any application. The Java API is based on SAX/JAXP/TrAX (which are Java APIs for XML processing) industry standards which make integration for developers possible [24]. The reader should be familiar with the basics of XML to understand what is explained next.

Nowadays it comes in quite handy to manage XML data as XML is increasingly used for all kind purposes like data exchange. Standard tools fail to identify changes precisely and meaningfully. The problem of finding changes can be solved more intelligently by closely looking at the XML structure. DeltaXML represents deltas in an XML format that allows downstream processing in an XML pipeline. According to a definition from [27] a pipeline is an XML vocabulary for describing the processing relationships between XML resources. A pipeline document specifies the inputs and outputs to XML processes and a pipeline controller uses this document to figure out the chain of processing that must be executed in order to get a particular result.

The reason behind XML change control is that in various fields changes to data must be identified, tracked, communicated and synchronized. Typical applications are related to content management, data versioning, data synchronizing and merging. Finding differences between files has already been done in many ways, but applying a simple diff on XML data is not very efficient. The rules needed to identify changes in XML files are very different from those that are needed for unstructured files. Tools that are specially made for one sort of document structure do not make use of the advantages of XML such as openness, flexibility and standardization.

Finding differences between XML files is not really straightforward. First of all it is important to get some understanding of what a change to an XML document really means. Traditional string-based change control will not be efficient as a large amount of insignificant changes will be found. These insignificant changes must then be ignored by an XML aware comparison. For example the following two files are identical to each other.

```
<record xmlns=http://www.myco.com/records id="b123">
<name>Michael Brown</name><born>1984-03-08</born>
<sex>M</sex>
</record>
```

```
<staff:record id="b123" xmlns:staff="http://www.myco.com/records">
       <staff:name>Michael Brown</staff:name>
       <staff:born>1984-03-08</staff:born>
       <staff:sex>M</staff:sex>
</staff:record>
```

Although both examples above are XML-identical they are also different from each other. The second file has got another declaration of the namespace, the elements look different and some white spaces were added. Other differences than these are significant. For example if the order is important then the following two files are different and if the order is not important the only difference is the two added elements.

```
<record id="b123">
       <name>Michael Brown</name>
       <born>1984-03-08</born>
       <sex>M</sex>
</record>
<record id="b124">
       <name>Gillian Bryan</name>
       <born>1951-03-06</born>
       <sex>F</sex>
</record>
```

```
<record id="b124">
       <employee-no>BR12</employee-no>
       <name>Gillian Bryan</name>
       <born>1951-03-06</born>
       <sex>F</sex>
</record>
<record id="b123">
       <employee-no>BR24</employee-no>
       <name>Michael Brown</name>
       <born>1984-03-08</born>
       <sex>M</sex>
</record>
```

A text based comparison like diff will most of the time report a huge change when only some small changes have been made to an XML document. The identification of a minimal set of actual changes is not an easy job.

In evaluating an XML change control solution there are principally three criteria that should be taken into account:

- Accuracy of the result: does the tool accurately identify changes to the XML data? A tool should not return changes because of a difference in the order of elements or the use of white spaces or namespace prefixes or any other insignificant change in XML. It is also very handy if a tool can be configured for example in the case where specific elements or attributes should be ignored. The handling of white spaces is also something that should be configurable as for some users white spaces will be relevant and for others not.

- Representation of the result: can the change information be used? It should for example be possible to automate change processing.

- Usability of the solution: the solution should be both efficient and usable. For example can the solution be used in the case of fast changing XML data or (very) large XML data sources? It is also very useful if the solution can be integrated in other applications.

The challenge lies in how changes to XML documents and data files can be represented in XML. In [24] a proposal of a delta format for XML is outlined and a short summary will be given here.

The first decision that arises when one would like to represent the differences between two files in XML is whether new elements should be used to indicate the changes or whether attributes should be used on the existing elements. There are people who like to use XML attributes to contain data and those who use XML as a markup language and use tags or elements to distinguish data items. However attributes provide order-independence and also use less file space. The disadvantage of using attributes is that it is impossible to add attributes to attributes. Attributes can be used for meta-data or information that can be applied to all kind of elements. That is why attributes are used as a method of specifying why an element is present in some delta file. As it is impossible to add attributes to attributes another mechanism must be used to identify the changes to attributes.

It is less difficult to extend an XML definition that is content-based than one that is attribute-based. Attributes are leaf elements which can not easily be extended. The use of attributes to identify the reason for the inclusion of an element in the delta file can be explained by a simple example. Consider an XML file that has no attributes and DTD. The basic structure of the delta file for the content-based XML file can be created by adding a single attribute, the delta attribute. This is an optional attribute which can have one of the following values: modify, add or delete. Consider the following XML fragment:

```
<X> <Y/>
</X>
```

Then consider a small change to this file:

```
<X>
</X>
```

Then the delta fragment is as follows:

```
<X d:delta="modify"> <Y d:delta="delete"/>
</X>
```

Of course the example above is very simple, but it gives an idea of the basic principles. The basic structure of the delta file is the same as the files that are compared to each other.

In order to be complete it is important to identify changes to PCDATA and to identify pairs of elements that have been exchanged. Both changes are represented by using special elements. This is roughly all there is needed. However the exact rules for representing the data elements are more complex.

For well-formed XML the delta file must be built without having any clues of the structure of the file which is given in a DTD or XML schema specification. Although the structure of the file is not explicitly given it is still possible to make a comparison between two XML files and to traverse the XML tree of the two files in a synchronized manner. A specific element of one file is seen as a modification of an element in the other file if they are the same type which means that the elements have the same name space (if any) and local name. The best match can be realized for elements that have the same name space and local name, the same attributes and sub elements that are all similar right down through the tree structure. These equal elements are used as an anchor point while the two files are matched, which eventually results in a minimum delta. Only a few attributes and elements are used to represent the changes. These attributes are added to the existing elements of the input files. The additional attributes are:

- d:delta to specify how the containing element has been changed.
- d:new-attributes and d:old-attributes specify changes to attributes.

The new delta elements are as follows:

- d:PCDATAmodify to specify a change to PCDATA in an element.
- d:exchange to specify an element exchanged with another or an element exchanged with PCDATA.

The attribute d:delta is found regularly in the delta file. It specifies why a certain element is present in the delta file, for example because it has been modified, added or deleted. The root element of the delta file has got a d:delta attribute with a value “WFmodify” if something has changed. The “WF” stands for “Well Formed” which separates it from a modification that is based on the structure of the DTD. Below any element with a d:delta with value “WFmodify” each element also contains a d:delta attribute with one of the following values:

- “add” if the specific element has been added.

- “delete” if the specific element has been deleted.
- “unchanged” if the specific element is unchanged.
- “WFmodify” if the attributes and/or content of the specific element have been modified.

There are some restrictions on how these attributes with their values are nested and this can be read in [24].

If a PCDATA item has been changed there will be a d:PCDATAmodify element which contains the old data within a d:PCDATAold element and the new data within a d:PCDATAnew element. If an element in one file is replaced by a different element or PCDATA in the other file, the delta file will contain a d:exchange element. This element will contain the data from the first file within a sub element d:old and the data from the second file within a sub-element d:new. If a d:exchange occurs in the delta file the two elements in the input file must be of a different type.

If there are any changes to XML attributes these should be detected and shown in the delta file. These changes are represented by using two special attributes in the delta file: d:old-attributes and d:new-attributes. The d:old-attributes contains all values of attributes that existed in the old file and which have been changed or deleted in the new file. The d:new-attributes contains all values of attributes that appear in the new file and which have either been changed or added in the new file. If attributes have not changed they are not included in the delta file, except in some special cases which will not be treated here.

The encoding of the values of the delta attributes is done by their attribute values which are changed, added or deleted. For example consider the attribute “Juggernaut” which has changed then it will be included as “Juggernaut=’old-value’ ” in the delta attribute d:old-attributes and as “Juggernaut=’new-value’ ” in the delta attribute d:new-attributes. A deleted attribute will only exist in d:old-attributes and an added attribute will only appear in d:new-attributes. The following example shows that on an element a, the value of the attribute href has been modified from href=’www.run.nl’ to href=’http://www.run.nl’. Furthermore the attribute xx=’value of xx’ has been deleted and the attribute yy=’value of yy’ has been added.

```
<a d:old-attributes= "href='www.run.nl' xx='value of xx'"
   d:new-attributes= "href='http://www.run.nl' yy='value of yy'"/>
```

There are other ways of treating changed attribute values but the way it is done above is how it is handled in DeltaXML.

XML documents can be compared as well-formed XML, but better results can be gained by exploiting the structure of the XML data. A DTD provides DeltaXML with knowledge of the allowed structure of an XML file such that a better and more efficient comparison can be done which usually results in a smaller delta file. The DTD defines which elements are required, which are optional and/or repeated. With this knowledge two files can be compared better as it is now more clear how the two files correspond with one each other. For example consider the two XML fragments beneath.

```
<fragment>
       <firstName>John</firstName>
       <lastName>Smith</lastName>
       <address>21 High Street</address>
       <address>Malvern</address>
       <address>Worcester</address>
</fragment>
```

```
<fragment>
       <firstName>John</firstName>
       <lastName>Smith</lastName>
       <firstName>Mike</firstName>
       <lastName>Jones</lastName>
       <address>Worcester</address>
</fragment>
```

When comparing these two fragments, without any knowledge of the structure, the following delta may be produced:

```
<fragment d:delta="WFmodify">
       <firstName d:delta=unchanged"/>
       <lastName d:delta="unchanged"/>
       <d:exchange>
       <d:old>
       <address>21 High Street</address></d:old>
       <d:new><firstName>Mike</firstName></d:new> </d:exchange>
       <d:exchange>
       <d:old><address>Malvern</address></d:old>
       <d:new><lastName>Jones</lastName></d:new>
       </d:exchange>
       <address delta="unchanged"/>
</fragment>
```

This delta file that is generated without any knowledge of the DTD could be useful to check if two files are the same or to provide an update for changes. Without the DTD the comparator uses the following structure:

```
<! ELEMENT fragment (firstName | lastName | address)*>
```

However if the definition of the fragment was as follows:

```
<! ELEMENT fragment (firstName | lastName | address*)*>
```

then the modification could be identified as it is originally meant. By using the information of the DTD the comparator can understand that the address elements have been deleted instead of that they have been changed to another type of element. With the new information a smaller delta file can be generated:

```
<fragment d:delta="modify">
       <firstName d:delta="unchanged"/>
       <lastName d:delta="unchanged"/>
       <address d:delta="delete">21 High Street</address>
       <address d:delta="delete">Malvern</address>
       <address d:delta="delete">Worcester</address>
       <firstName d:delta="add">Mike</firstName>
       <lastName d:delta="add">Jones</lastName>
       <address d:delta="add">Worcester</address>
</fragment>
```

The delta above is more meaningful. So by making use of a DTD better results are achieved, although in some cases the DTD will not help much in producing an adequate delta. This is true for a relatively unstructured DTD and for data of which the order is strictly defined.

Many more things come along in order to produce an efficient comparison between XML files. For more detailed information the reader is referred to [24].

At last some important characteristics of DeltaXML are listed beneath:

- The possibility to define how precisely changes should be handled.
- Representation of deltas in XML which can be processed by both humans and automata.
- Extensive configuration options designed for pipeline architectures.
- Scalability: large source files can be compared.
- Java API with a comprehensive documentation.

6 Benchmark

This chapter describes the structure of the developed data sets and the tests that were carried out to measure the performance of the described (delta) compression tools. The results of the field investigation were needed to get a realistic picture of which data can be encountered and what their related characteristics are. This chapter is an important part of the project as it attempts to find out which tools are preferable with respect to the data sets.

6.1 Data Sets

Before the tests can be run some realistic data sets must be developed first. In the design of these data sets the results of the field investigation were used as a guide-line. The sort of documents fall apart in the categories described next.

6.1.1 MS Office/OpenOffice.org Documents

- MS Word documents are widely used for all kind of purposes and are relatively frequently modified. Therefore these documents are very appropriate to undergo some extensive testing. The average file size of MS Word documents varies between 50 KB and 221 KB. Therefore it is particularly interesting to focus the tests on this range of file sizes. In the preliminary research some testing is already done, even for MS Word documents that well exceed 50 MB in file size. Parallel to the MS Word documents there should be made similar OpenOffice.org Writer documents.

- MS Excel documents are regularly used and their file size will (on average) certainly not exceed that of MS Word documents. The file size of MS Excel documents ranges from 50 KB till 221 KB as well. Parallel to the MS Excel documents there should be made similar OpenOffice.org Calc documents.

6.1.2 Data characteristics

With respect to Word/Writer documents it is interesting to develop data sets that are fully text-based and data sets that contain both text and graphical elements. Graphical content may be in the form of embedded jpeg, gif, bmp, eps and various other formats. Other sorts exist but in this test the graphical elements are limited to a few types only. With this information it is possible to get an idea of what contribution is made by graphical content to the calculation time and the size of the delta.

6.1.3 Data Sets structure

There are several data sets: MS Word XML Graphics Data Sets, MS Word Doc Only Text Data Sets, MS Excel xls Data Sets, OpenOffice.org Doc Graphics Data Sets et cetera. Each data set consists of 12 files that are part of a version history. The data sets and the file sizes of the individual files can be observed in the Excel sheets in the appendix.

6.1.4 XML vs Native format

MS Office 2003 (with MS PowerPoint as an exception) documents can be saved in the native format but also in XML. As open standards become more popular data standards migrate to XML. Therefore it

would certainly be useful to test all Office documents (except MS PowerPoint) in the XML format. XML documents can be gained by converting the native format to the XML format, such that the content of the two documents will be identical. OpenOffice.org saves documents in a rather different way. Writer documents for example are by default saved in a ZIP compressed format. If this compressed file is unzipped several files can be distinguished: XML files, CSS files and various other elements can be encountered.

6.2 Benchmark tool

The main benchmark script is written in Perl and automatically runs all tests after being initiated (see the appendix for the source code). The benchmark primarily consists of two parts: (1) the delta part and (2) the patch part. The delta part is responsible for computing the deltas by subsequently running the specific compression tools on the files of the various data sets. The patch part computes the target files by making use of the old files and the associated deltas that were generated in (1). Multiple runs are performed for each file in the data sets such that an average value of the calculation times can be computed. Both the delta part and the patch part return unique files which contain average values of the measurements.

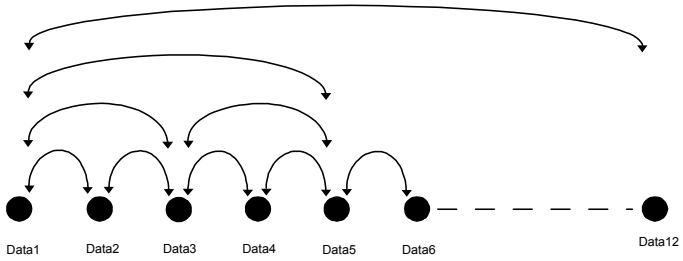

Figure 6-1: Permutations of deltas between the files of the data sets.

As mentioned before each data set consists of 12 files that represent a version history. The first file Data1 is the oldest file in the history and the newest file is Data12. Each file is compared to each other file to include all possible scenarios of delta compression. The number of possible comparisons can be calculated by filling in n =12 and k =2 in the following formula:

$$\frac{n!}{(n-k)!k!} = \frac{12!}{(12-2)!2!} = 66$$

So there are 66 possible comparisons to be done. Each measurement is repeated 10 times, so altogether there are 66 x 10 = 660 comparisons per delta compression tool (per delta or patch part). So at the end the benchmark has performed 66 x 10 x 2 x 4 = 5280 comparisons.

The delta part outputs files that contain average values of the performed measurements and each line consists of the following fields (the names reveal their meanings) :

```
size_old_file
size_new_file
size_new_file_zipped
size_uncompressed_delta
size_zipped_delta
average_time_delta_uncompressed
average_time_delta_zipped
average_time_total_computation
```

The patch part also outputs files, although the fields are somewhat different:

```
size_old_file
size_new_file
size_zipped_delta
size_uncompressed_delta
average_time_unzipping
average_time_patching
average_time_total_computation
```

The benchmark code is properly structured and comments are placed where necessary, so an average programmer should be able to read the code easily. Some additional benchmark scripts have been made to measure other specific values, however these are not included in this paper and can be found on the CD-ROM.

6.3 Hardware and software specifications

The major benchmark script was run on a workstation with the following specifications (all data sets were directly available on the local hard disk):

OS:	Sun Microsystems Solaris 8
Processor:	Sparcv9 502 MHz
Memory:	256 MB
Hard Disk:	7200 rpm

The benchmark script for comparing XMill to ZIP was run on a PC with the following specifications (all data sets were directly available on the local hard disk):

OS:	Windows XP SP1
Processor:	AMD Athlon 2100+, 1.73 GHz, L2 On-board Cache 256 KB
Memory:	512 MB SD RAM
Hard Disk:	7200 rpm

The various tools that are used in the benchmark are only CPU-bound. No continuous disk activity was observed and memory is not a problem as the tools work with a certain window size to maximize memory allocation.

6.4 Results

The results of the benchmark will be presented with the help of Excel sheets that are derived from the benchmark data. All (delta) compression tools were evaluated to enable a comprehensive comparison.

6.4.1 ZIP vs XMill

Both ZIP and XMill are tools to compress individual files. ZIP can be used on any given input type whereas XMill is only applicable to XML files, so a suitable comparison between the two can only be achieved on XML files. Therefore the two tools are compared on both the MS Word XML Graphics Data Set and the MS Word XML Only Text Data Set. This makes it possible to find out how the two tools behave on data-like XML files (which contain binaries that are encoded by using the base64 encoding scheme) and text-like XML files. The results are shown in figure 6-2.

A first observation clearly indicates that uncompressed XML files are relatively large in file size. The uncompressed XML files fully correspond with the last 11 (the first one is excluded) files that are shown in the MS Word XML Data Sets in the appendix. It does not make any significant difference (although any reduction in file size is desirable) if the files of the MS Word XML Graphics Data Set are compressed by ZIP or XMill. The XMill compressed XML files are only a few thousands of bytes smaller than the ZIP compressed XML files. However a more significant difference can be observed between the XML files and the XMill compressed files of the MS Word XML Only Text Data Set. XMill is here a clear winner and in some cases the XMill compressed files turn out to be 23 % smaller than the ZIP compressed files. This difference would be even bigger if XMill would be extended with some specialized compressors such that some structural knowledge of MS Word XML files can be exploited.

Figure 6-3 shows how the original file sizes of the data sets relate to the file sizes of the ZIP and XMill compressed files. It is particularly interesting to observe that ZIP and XMill behave in the same way: their curves look quite similar (although the curves of XMill are a little bit less steeper, which means that it continually compresses better than ZIP). The fact that the curves look similar can be explained by the fact that XMill, just as ZIP, uses the zlib library to compress data. XMill however stores data beforehand in so called containers that are later compressed separately by gzip. As already claimed by the inventors of XMill, in almost any case XMill performs better than ZIP. The inventors also stated that XMill would produce better results when compressing data-like XML files instead of text-like XML files. This does not seem to be true when looking at the benchmark results, for in this benchmark XMill compresses text-like XML files better than data-like XML files. This may be due to the way MS Word formats XML data.

Figure 6-4 shows the computation times of ZIP and XMill. This graphic indicates that the difference in computation time between ZIP and XMill looks like some initiation time. The behavior of both tools is further quite similar.

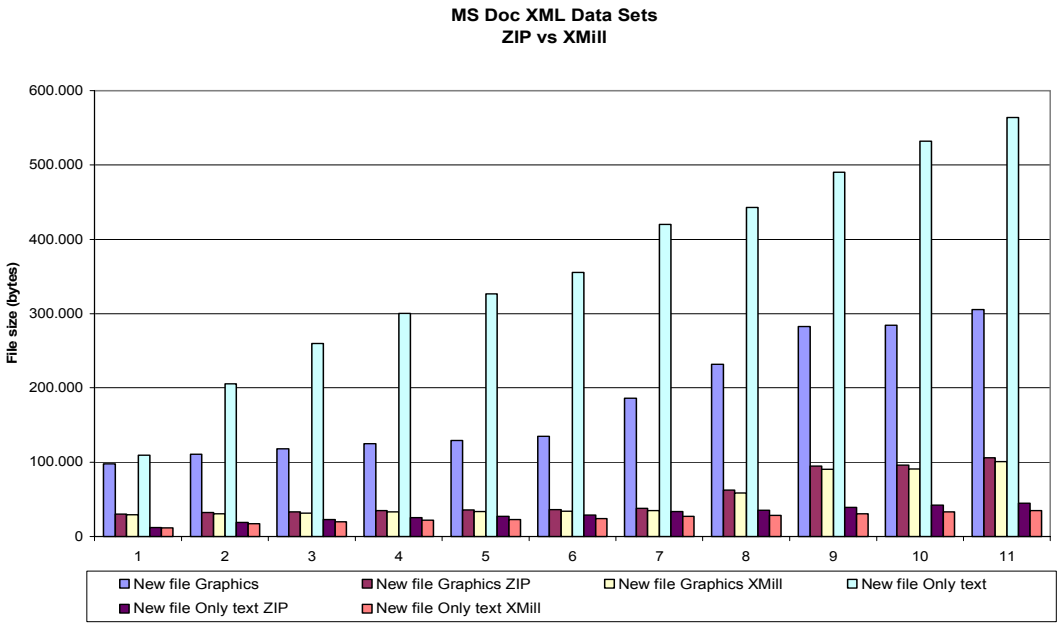

Figure 6-2: ZIP versus XMill MS Doc XML Data Sets File Compression.

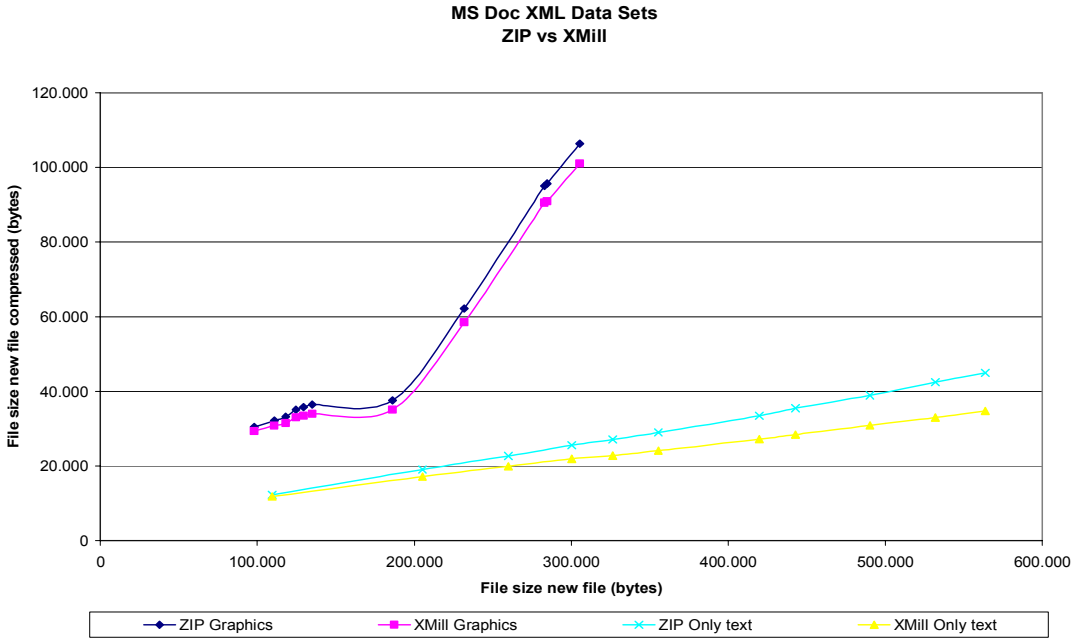

Figure 6-3: ZIP versus XMill MS Doc XML Data Sets File Compression.

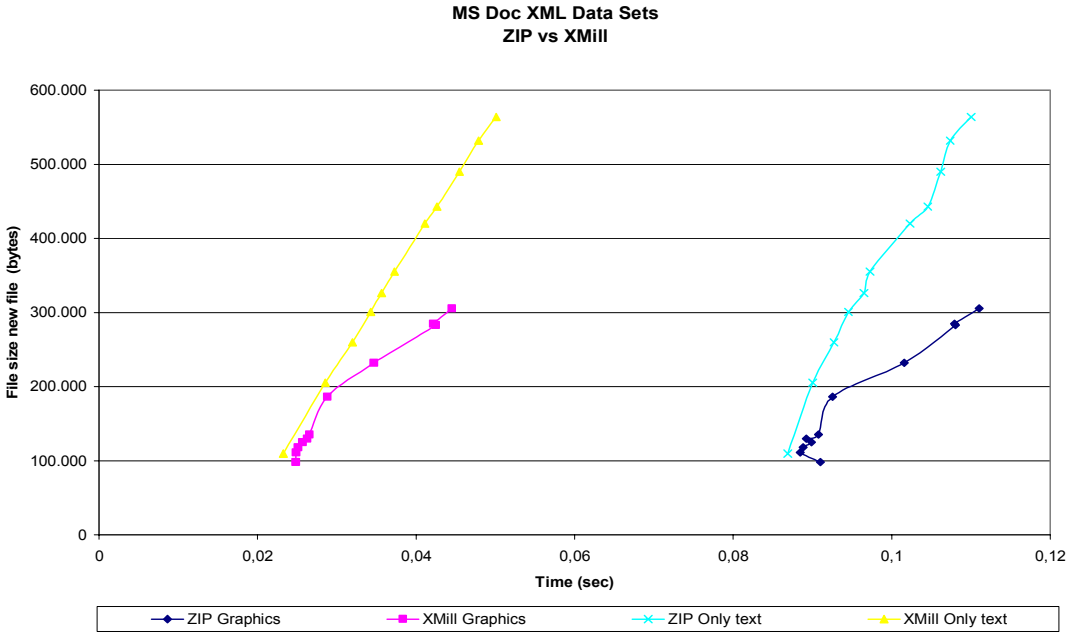

Figure 6-4: ZIP versus XMill MS Doc XML Data Sets Compression Time.

6.4.2 Diff vs Xdelta vs Vcdiff vs DeltaXML on XML

As open standards are becoming more important nowadays it is interesting to know how the delta compression tools perform on XML data. The data sets used are the same as in the previous comparison between ZIP and XMill.

Figure 6-5 gives an impression of what results are gained when the delta compression tools are evaluated on the MS Word XML Graphics Data Set. As already expected the deltas that are generated by diff are extremely inefficient. The diff algorithm is not a good delta compressor as it implicitly assumes that data common to f_{old} and f_{new} are exactly in the same order in the two files and it also does not account for substrings in f_{old} which appear in f_{new} a couple of times. DeltaXML does not return any efficient deltas as well, which may be caused by the (not so efficient) XML code that is generated by MS Word. However better results would be expected of a tool that is specially designed to generate accurate deltas between XML files. Xdelta and Vcdiff are much more efficient in generating deltas and without compressing the delta, Xdelta returns the best results among the delta compression tools tested here. In general the uncompressed deltas can also be further compressed by using ZIP (XMill is not used here as it did not work on Solaris 8). Although Xdelta does not compress its output it would not make any difference to zip the output as the format is already compressed enough. The output of Vcdiff can still be compressed though as Vcdiff was run without compressing the delta (vcdiff –d).

By zipping the output of Vcdiff the file size comes pretty close to that of the uncompressed output of Xdelta. The zipped output of diff remains the most inefficient of the tools. Figure 6-5 shows the size of the zipped output of DeltaXML which is smaller than that of the zipped output of Vcdiff. In the previous comparison between ZIP and XMill, XMill turned out to be a better XML compressor than ZIP as XMill compressed files were sometimes 23 % smaller than those that were zipped. So DeltaXML combined with XMill may certainly result in very small deltas which may be even smaller than those of Xdelta.

Figure 6-6 shows the same comparison only then related to the MS Word XML Only Text Data Set. The uncompressed deltas of diff and DeltaXML are much larger than those of the MS Word XML Graphics

Data Set. Xdelta and Vcdiff return very small deltas without using any compression. Further the same trend can be observed as for the MS Word XML Graphics Data Set, although the deltas are generally smaller.

> Note: The curves that are related to the file size of the deltas consist of 11 points that are formed by the following deltas: 1-2, 1-3, …, 1-11, 1-12.

Figure 6-7 shows the relation between the uncompressed deltas and the calculation time of the tools on the MS Word XML Graphics Data Set whereas figure 6-8 shows the relation between the compressed deltas and the calculation times of the tools on the MS Word XML Graphics Data Set. Without compressing the delta with ZIP the following trend can be observed: Xdelta is the fastest among them, diff and Vcdiff do not differ much in speed and DeltaXML is clearly the slowest. When compressing the delta (which is recommended!) Xdelta and Vcdiff are very similar in speed and diff is significantly slower which is due to the fact ZIP needs more time to compress the large deltas that are generated by diff.

Figure 6-9 shows the relation between the uncompressed deltas and the calculation times of the tools on the MS Word XML Only Text Data Set whereas figure 6-10 shows the relation between the compressed deltas and the calculation times of the tools on the MS Word XML Only Text. The difference with the previous Graphics Data Set is that the files of the MS Word XML Graphics Data Set consume more time which means that graphical elements (base 64 encodings) require more calculation time. It is also interesting to mention that diff operates relatively better on text-like XML files: although the deltas are still the largest diff operates faster than Xdelta and Vcdiff on the Only Text Data Set, see figure 6-9. However when the deltas are zipped Xdelta is approximately as fast as diff and Vcdiff becomes the one that is slower, while DeltaXML remains the slowest delta compression tool among all tools.

Figure 6-11 and 6-12 relate to the MS Excel XML Data Set and are quite similar to the graphics of the MS Word Only Text Data Set. No special attention will be paid to the patching part and therefore only figure 6-13 will give some idea of how the tools patch the old file with the delta to generate the target file. Although some tools are faster, the difference in speed is most of the time a matter of a fraction of a second, with DeltaXML as an exception because it is significantly slower than all other tools.

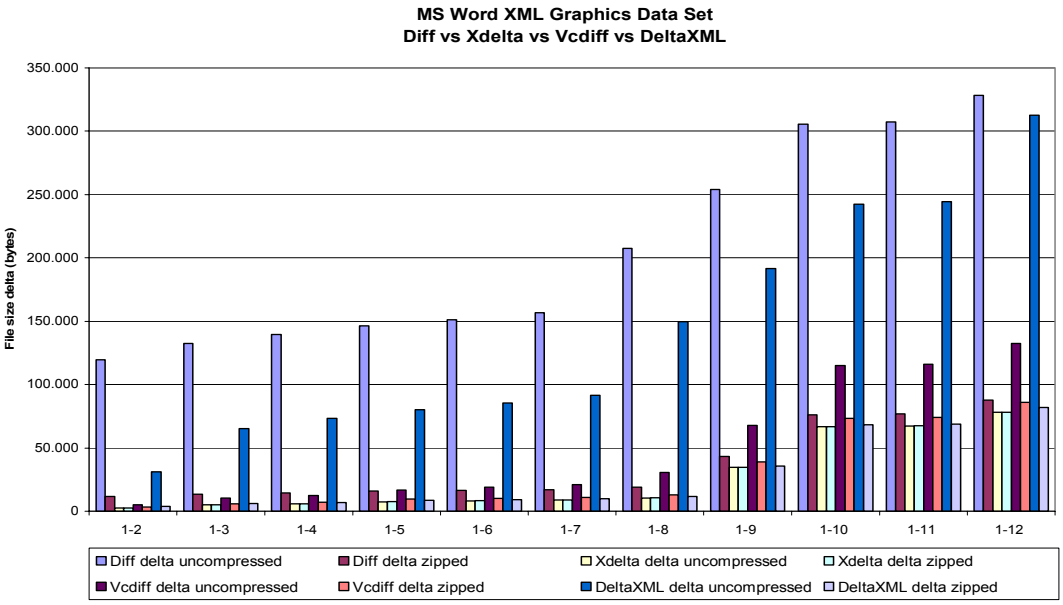

Figure 6-5: (Delta) compression tools on MS Word XML Graphics Data Set.

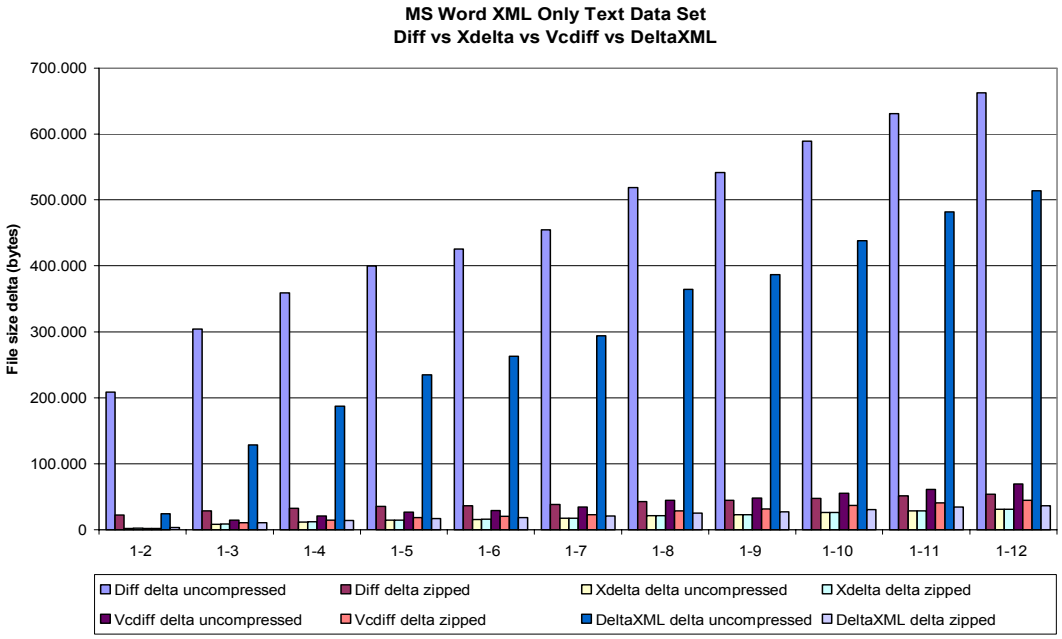

Figure 6-6: (Delta) compression tools on MS Word XML Only Text Data Set.

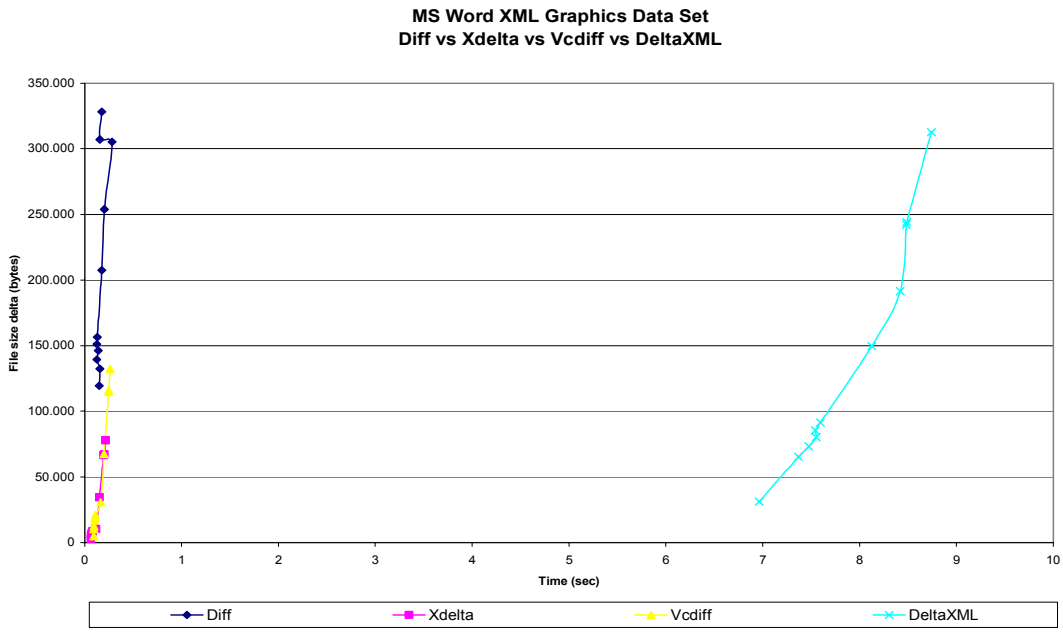

Figure 6-7: Delta compression tools on MS Word XML Graphics Data Set.

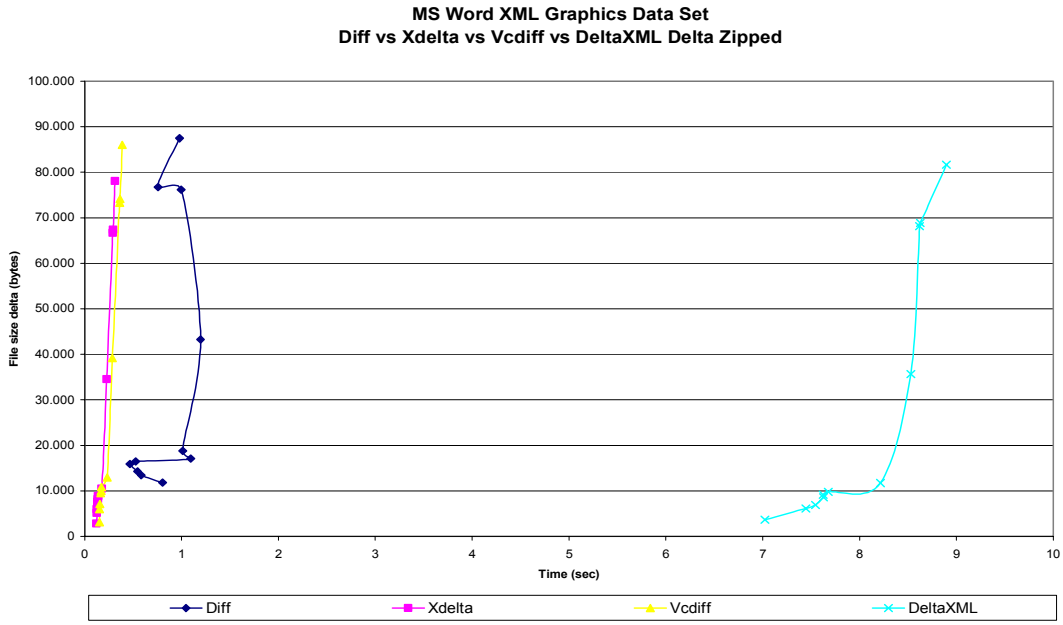

Figure 6-8: Delta compression tools on MS Word XML Graphics Data Set Delta Zipped.

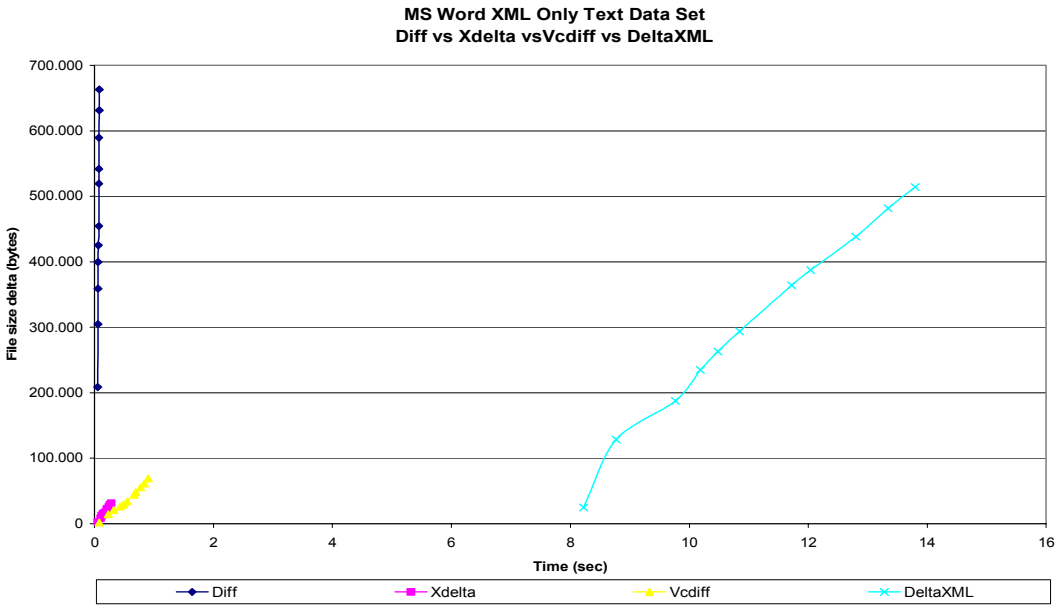

Figure 6-9: Delta compression tools on MS Word XML Only Text Data Set.

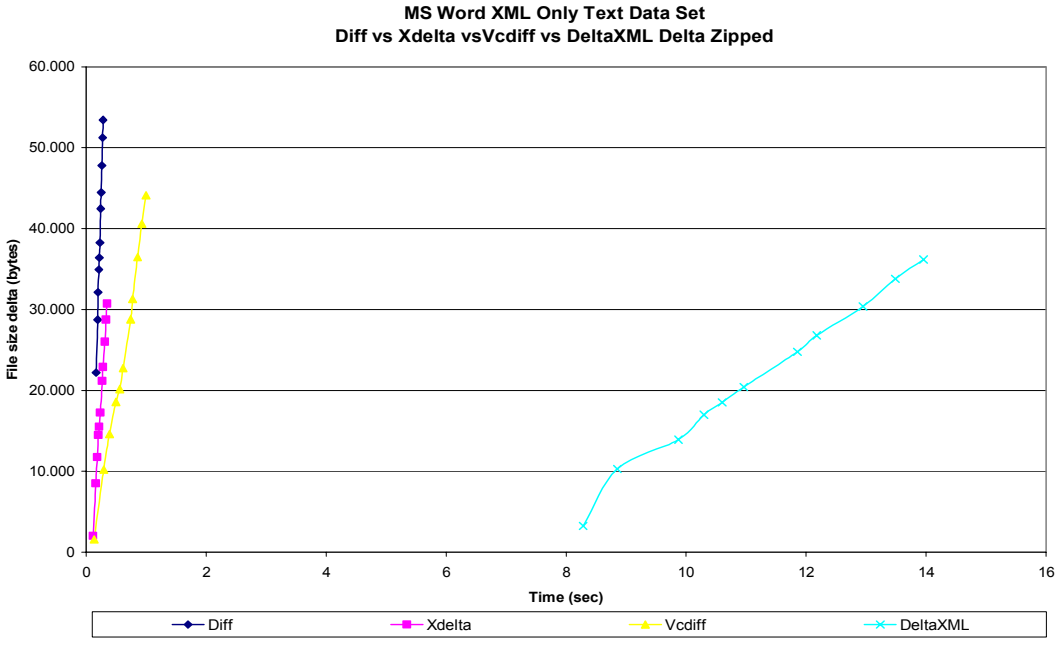

Figure 6-10: Delta compression tools on MS Word XML Only Text Data Set Delta Zipped.

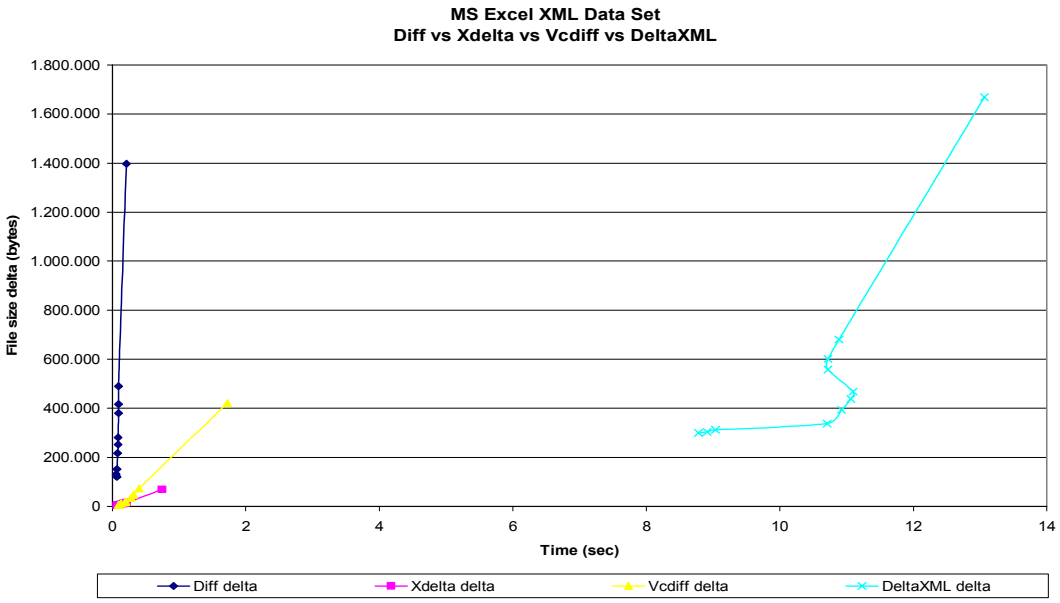

Figure 6-11: Delta compression tools on MS Excel XML Data Set.

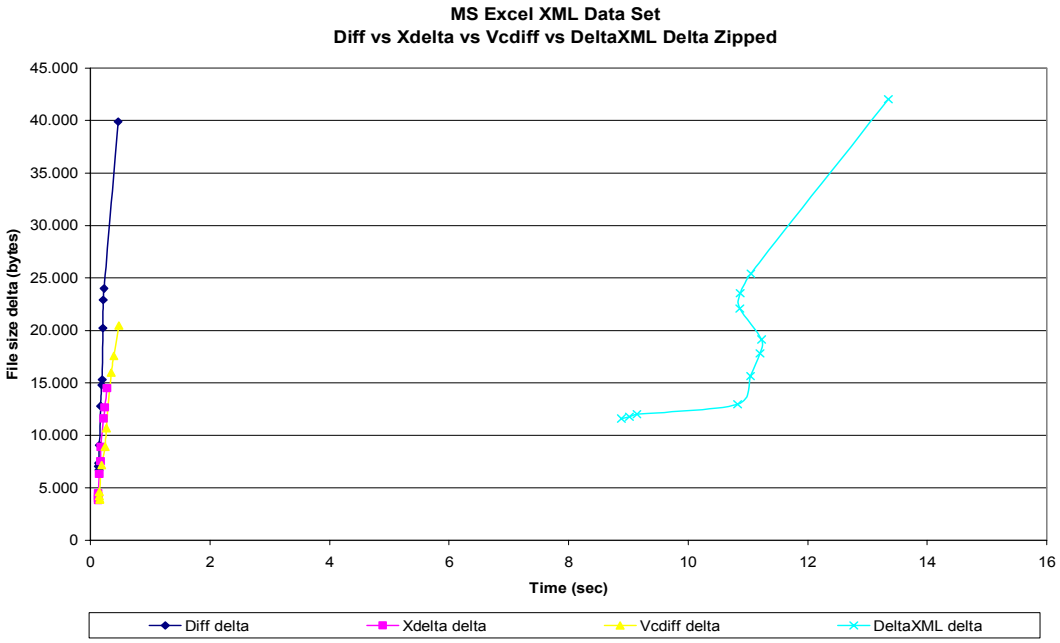

Figure 6-12: Delta compression tools on MS Excel XML Data Set Delta Zipped.

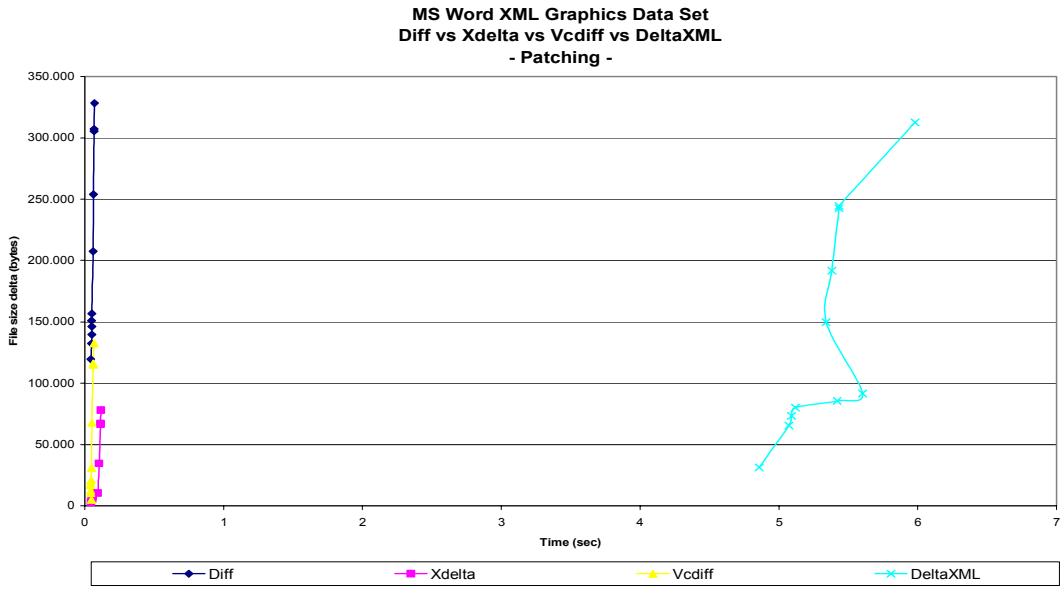

Figure 6-13: Patching of delta compression tools on MS Word XML Graphics Data Set.

6.4.3 Xdelta vs Vcdiff on Doc/xsw

Figure 6-14 shows how Xdelta and Vcdiff behave on the MS Word Doc Data Sets. Xdelta performs the best as it is produces the smallest deltas in a considerably shorter time than Vcdiff. The strange twist in the blue curve is something which could not be explained. By looking at figure 6-15 it is easy to see that some profit can be gained by compressing the deltas with ZIP. The last two figures 6-16 and 6-17 relate to the OpenOffice.org Writer Doc Data Sets. The uncompressed deltas that were generated between the files of the OpenOffice.org Writer Doc Data Sets are a little bit smaller than those generated between the files of the MS Word Doc Data Sets. As can be observed by comparing figure 6-15 with 6-17 the difference between the zipped deltas of the MS Word Doc files and the OpenOffice.org Writer Doc files is quite small. However the deltas with respect to the OpenOffice.org Writer Doc Data Sets are still smaller and the time required to generate these compressed deltas is relatively much shorter. It is not interesting to show any information about the patch part as patching is achieved in a very short time.

The results with respect to the OpenOffice.org Writer Native Data Sets (xsw) are unexpected and can be observed in figure 6-18. Here Vcdiff performs the best: the deltas of Xdelta and Vcdiff are quite similar while the time required to generate these deltas is shorter for Vcdiff (the time difference is the largest with respect to the graphics data set). Further compressing the deltas will not contribute to a yet more efficient delta.

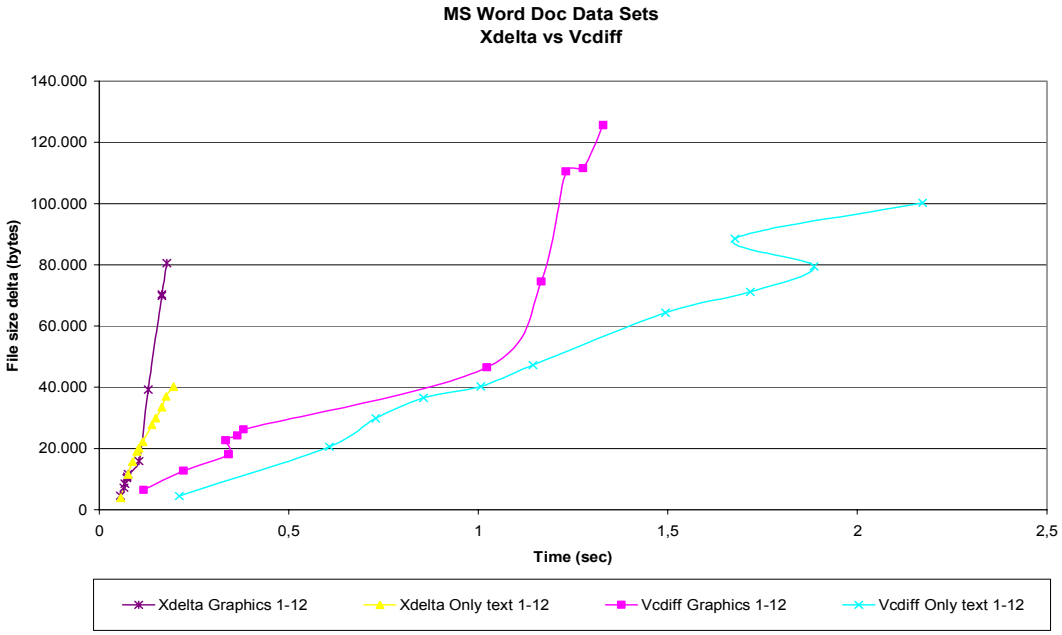

Figure 6-14: Xdelta vs Vcdiff on MS Word Doc Data Sets.

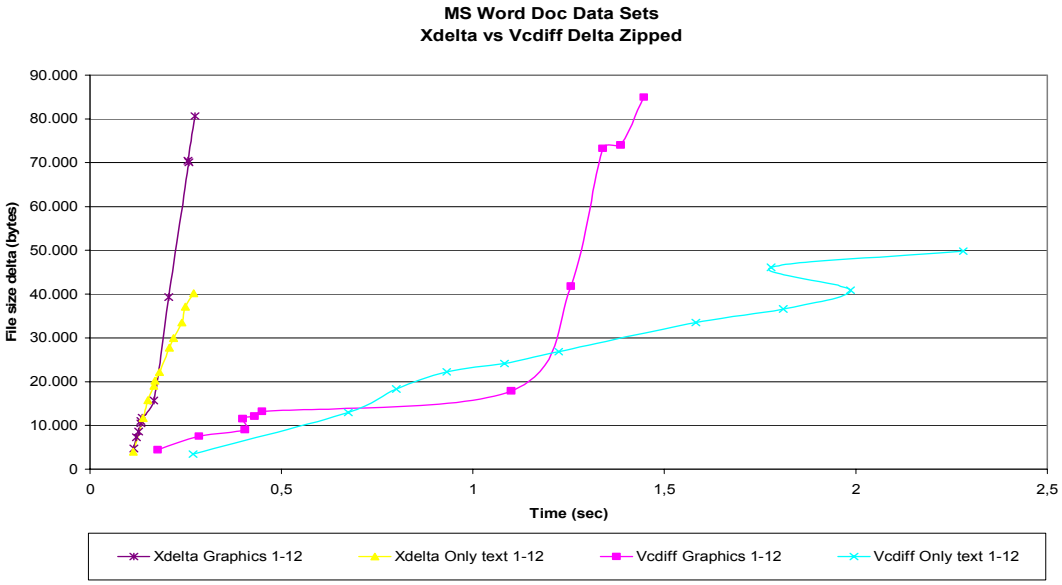

Figure 6-15: Xdelta vs Vcdiff on MS Word Doc Data Sets Delta Zipped.

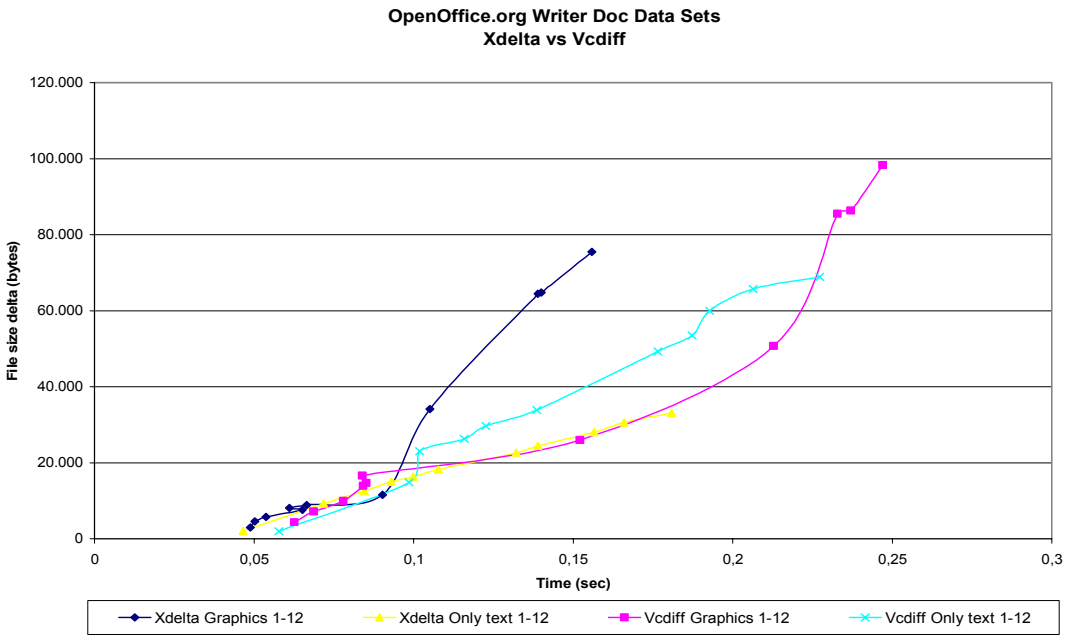

Figure 6-16: Xdelta vs Vcdiff on OpenOffice.org Writer Doc Data Sets.

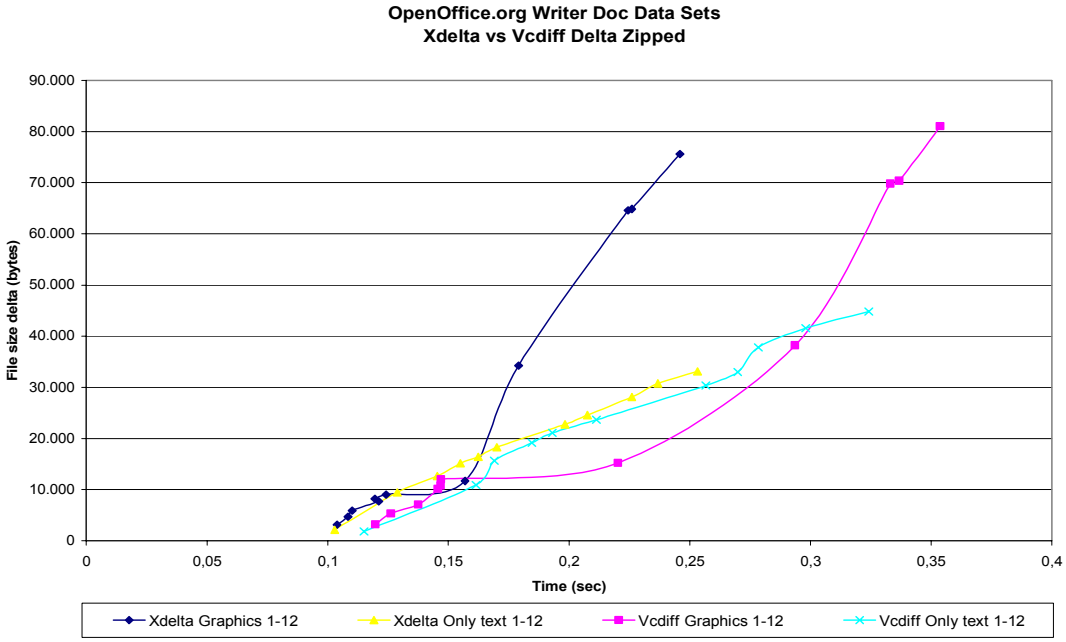

Figure 6-17: Xdelta vs Vcdiff on OpenOffice.org Writer Doc Data Sets Delta Zipped.

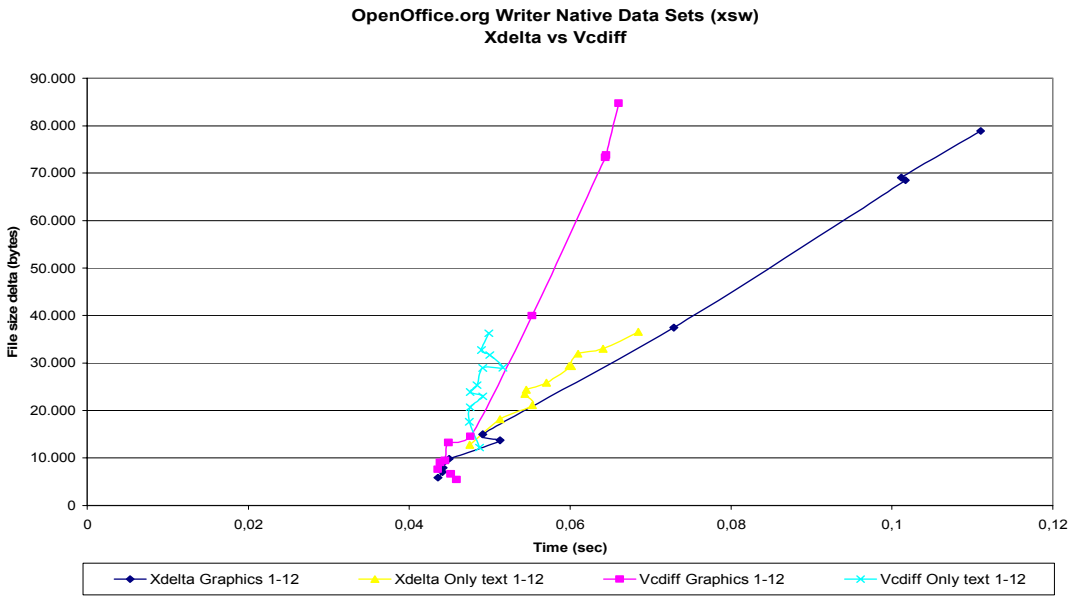

Figure 6-18: Xdelta vs Vcdiff on OpenOffice.org Writer Native Data Sets (xsw).

6.4.4 MS Word and OpenOffice.org Writer: Doc vs XML vs xsw

Figure 6-19 and 6-20 should give some impression of what a specific data format and compression method can contribute to the total file size of an Office document. In figure 6-19 it is clearly visible that the MS Word Doc and MS Word XML files are relatively very large. The OpenOffice.org Writer Doc format and especially the OpenOffice.org Writer Native format (xsw) create much smaller files (at least 50 % till 80 % smaller). All formats, except the OpenOffice.org Writer Native files which are already compressed by ZIP, can be further compressed to get an even smaller file size. The most efficient files are gained by zipping the OpenOffice.org Writer Doc files, after which the OpenOffice.org Writer Native files and the XMill compressed MS Word XML files are the most efficient. It is worth mentioning that the XMill compressed XML files from MS Word are smaller than the ZIP compressed Doc files from MS Word. In the section about XMill it was already stated that when XML files are compressed with XMill these compressed files will typically be smaller than the zipped original data.

When comparing figure 6-19 with figure 6-20 it is clear that text-like XML files consume more memory than data-like XML files. Data-like XML files are only 30 % larger than the original MS Word Doc files, whereas text-like XML files are more than 100 % larger than the original MS Word Doc files. This is the consequence of the verbose nature of text-like XML data. Fortunately compressors are very suitable for compressing text-like XML and this is reflected in the small file sizes of the compressed files (figure 6-20). This time MS Word XML files that are compressed with XMill are the smallest files that can be produced. These files are even smaller than the compressed OpenOffice.org Writer formats.

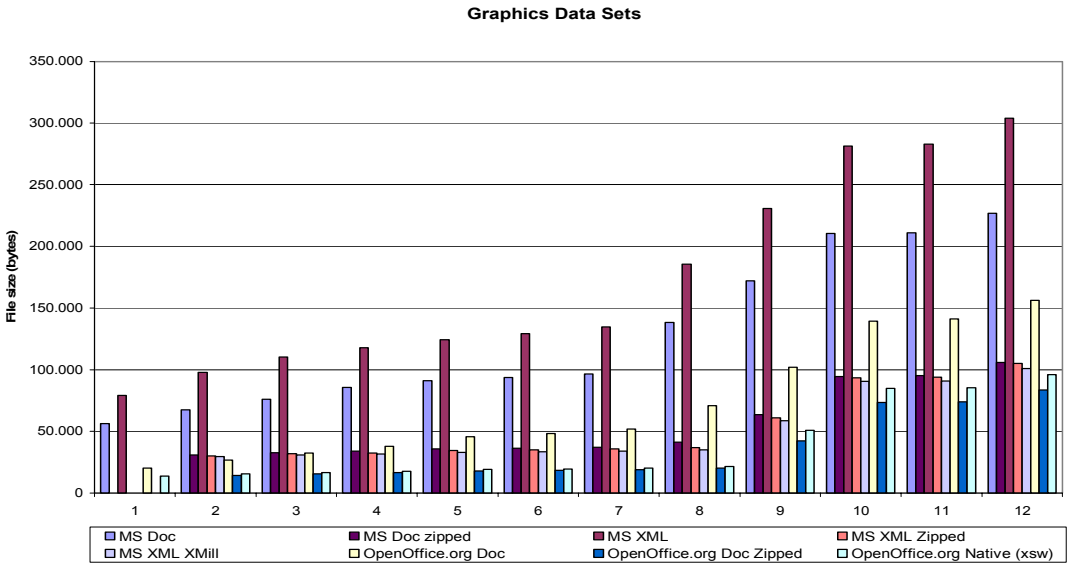

Figure 6-19: Comprehensive overview Graphics Data Sets.

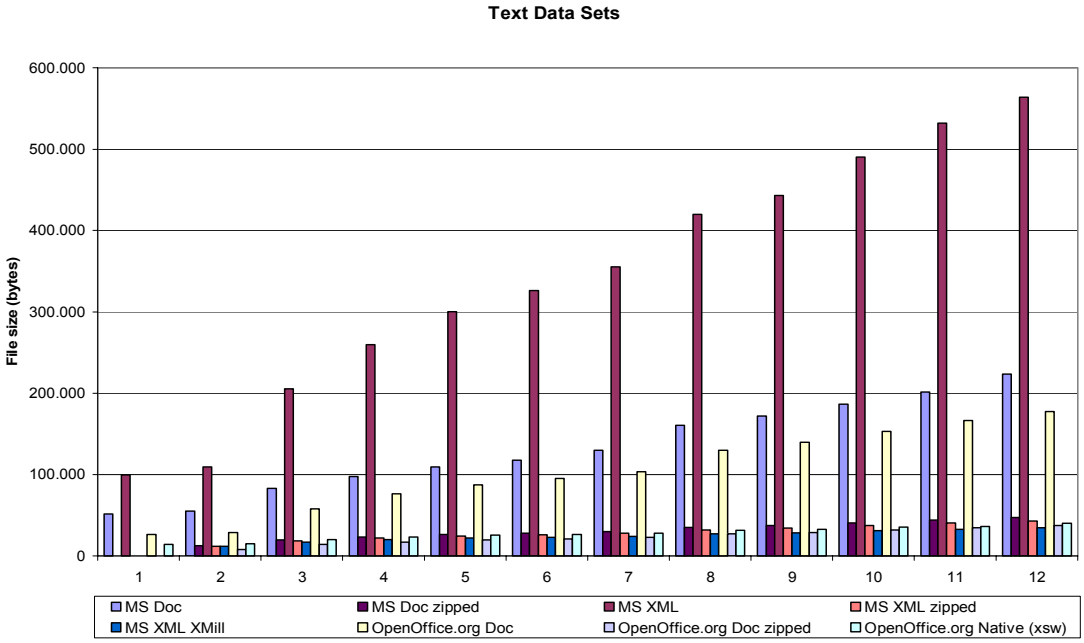

Figure 6-20: Comprehensive overview Doc Only text Data Sets.

6.4.5 Xdelta vs Vcdiff on xls/sxc

Xdelta and Vcdiff are the only tools that can generate deltas between MS Excel xls documents or OpenOffice.org Calc xls/sxc documents. The uncompressed deltas that are generated by Xdelta and Vcdiff on the OpenOffice.org sxc Data Set are really small which is the result of the fact that the original files are already small. This also reduces the total calculation time required for generating the delta. So if processor speed is expensive this OpenOffice.org format is recommended. Further compressing of the deltas of the OpenOffice.org sxc format will not contribute to a smaller delta: the original format is already compressed by ZIP. See also figure 6-21 and 6-22.

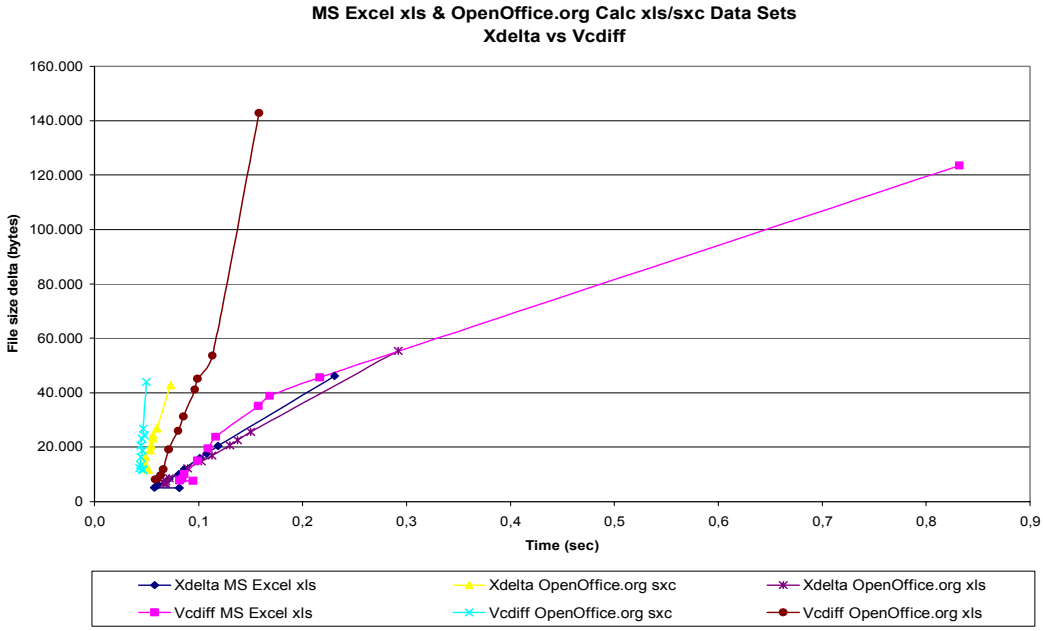

Figure 6-21: Xdelta vs Vcdiff on Excel/Calc Data Sets.

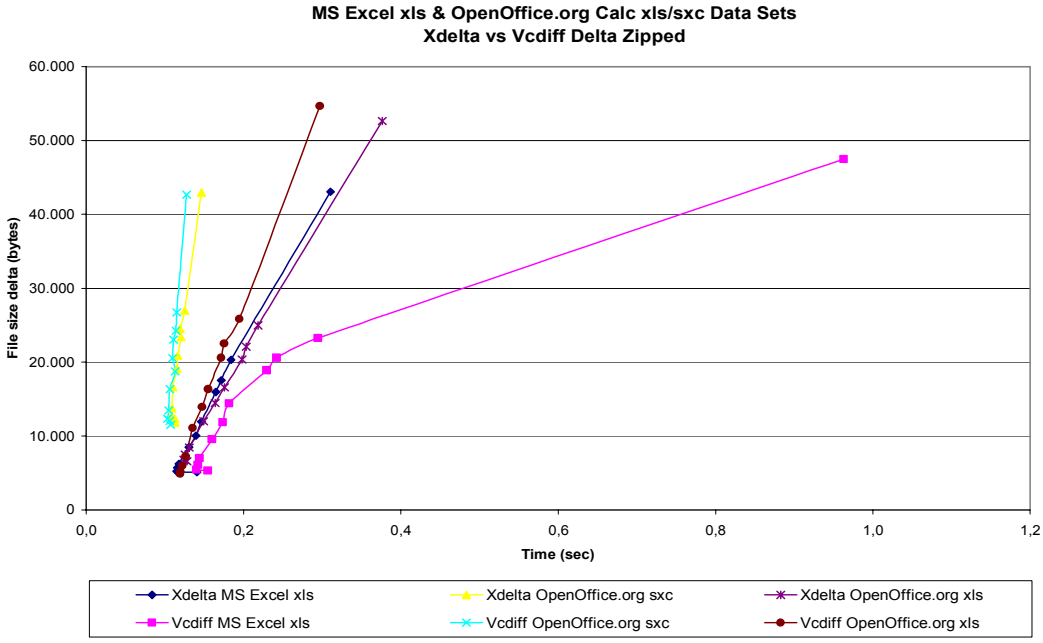

Figure 6-22: Xdelta vs Vcdiff on Excel/Calc Data Sets Delta Zipped.

6.4.6 MS Excel and OpenOffice.org Calc: xls vs XML vs sxc

In figure 6-23 all Excel/Calc formats are compared to each other. As already stated before uncompressed XML files are very large. In the section about XMill it is said that XMill compressed XML files are usually smaller than the zipped original files. Again it can be shown this claim is true as the large XML files are reduced to file sizes that are the smallest among all file sizes. So XMill can compress XML data very efficiently and performs somewhat better than ZIP. It can also be observed

that the OpenOffice.org Calc Native file format is always smaller than the OpenOffice.org Calc xls alternative.

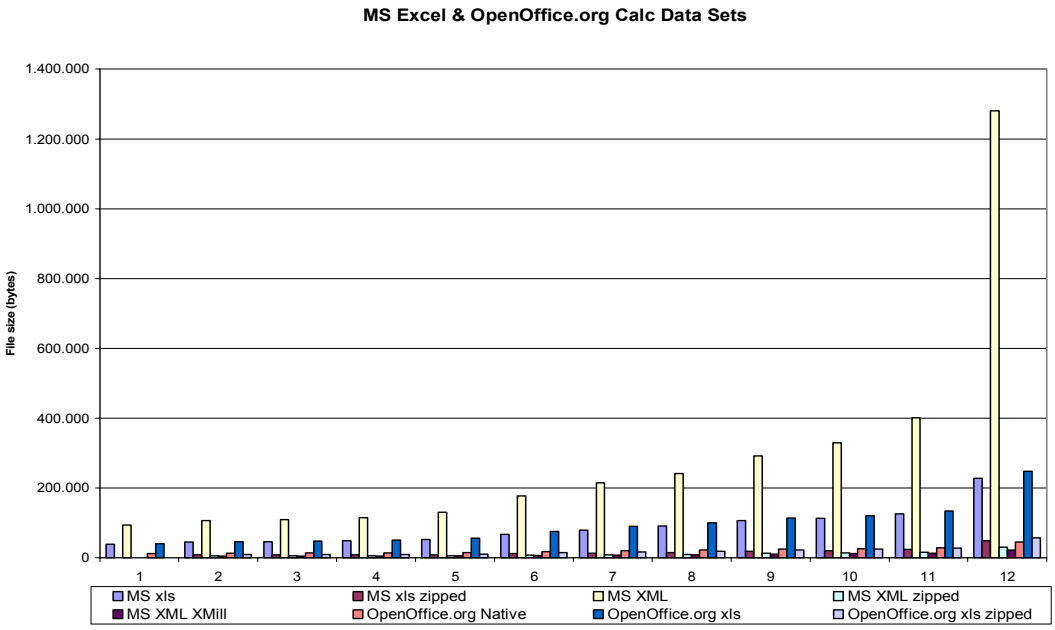

Figure 6-23: Comprehensive overview Excel/Calc Data Sets.

6.4.7 The benefits of a delta

In papers about delta compression it is often stated that deltas are usually much smaller than the individually compressed files. The benchmark results consist of many data that support this claim and figure 6-24 and 6-25 show that, although the benefits are not always that large, the deltas are smaller than the individually compressed files.

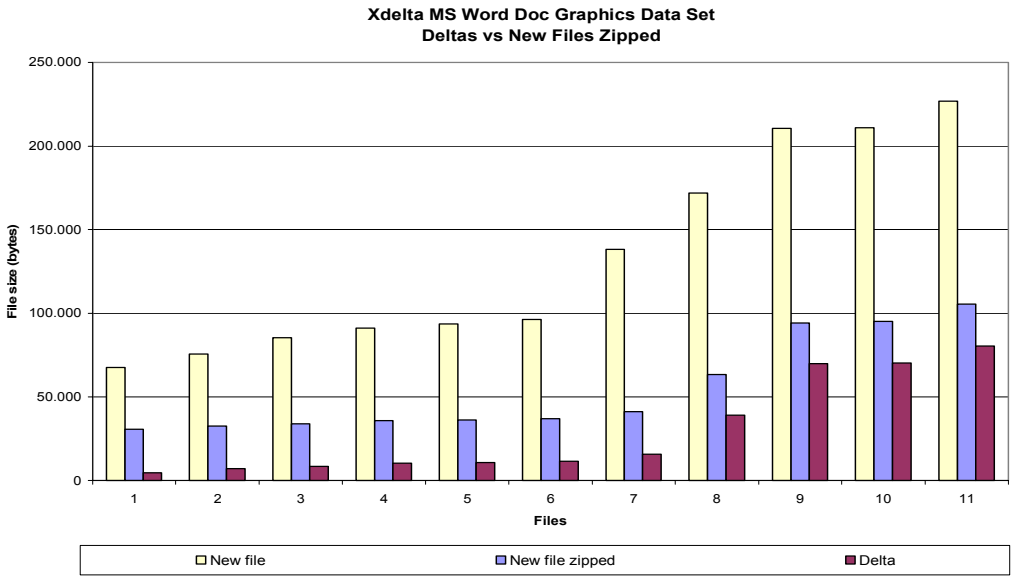

Figure 6-24: Xdelta Delta vs New File Zipped on MS Word Doc Graphics Data Set.

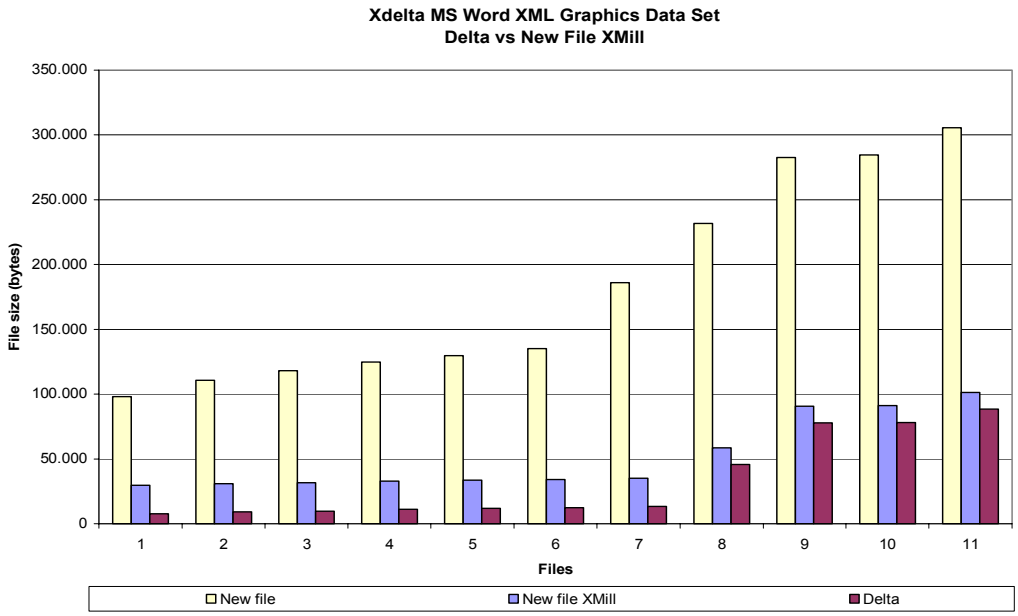

Figure 6-25: Xdelta Delta vs New File Zipped on MS Word XML Graphics Data Set.

6.4.8 Delta sequences

Many of the previous figures show curves that consist of 11 points. These 11 points are the file sizes of the following deltas: 1-2, 1-3, …, 1-11, 1-12.

The benchmark has computed many more deltas though and these are: 2-3, 2-4, …, 2-11, 2-12 and 3-4, 3-5, …, 3-11, 3-12 and 4-5, 4-6, …, 4-11, 4-12 and 5-6, 5-7, …, 5-11, 5-12 and 6-7, 6-8, …, 6-11, 6-12 and 7-8, 7-9, …, 7-11, 7-12 and 8-9, 8-10, 8-11, 8-12 and 9-10, 9-11, 9-12 and 10-11, 10-12 and finally 11-12. Figure 6-26 shows 7 curves that are related to the first 7 sequences of deltas described above. The last point of the some curve n ($2 \leq n \leq 7$) lies less higher than the last point of the n-1th curve which is the result of the fact that for example the delta 1-12 is bigger than the delta 8-12. The strange twists in the curves could not be explained. Figure 6-27 shows the same sequences of deltas only then for Xdelta instead of Vcdiff.

It is possible to calculate the average file size of the deltas for each sequence (there are 11 sequences altogether) and these are shown in figure 6-28 for both Xdelta and Vcdiff. This graphic demonstrates that both tools share some common characteristic: if the average delta of Xdelta of sequence n is bigger than that of sequence n+1 then the same thing counts for Vcdiff. This practically means that if for example one of the tools produces larger deltas than expected for some sequence of deltas than the same can be expected for the other tool.

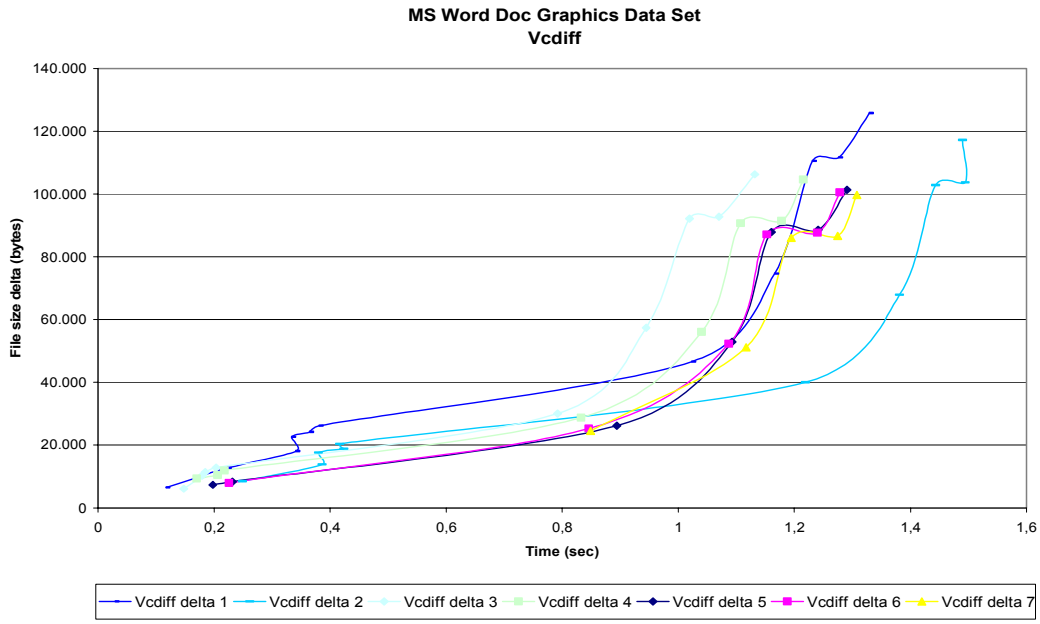

Figure 6-26: Vcdiff Deltas on MS Word Doc Graphics Data Set.

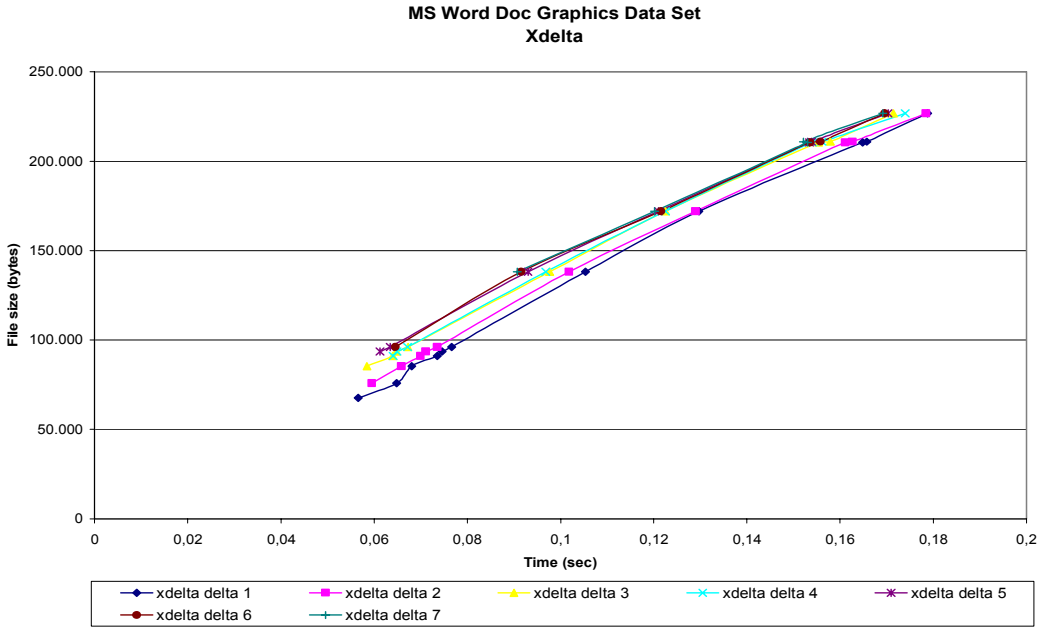

Figure 6-27: Xdelta Deltas on MS Word Graphics Data Set.

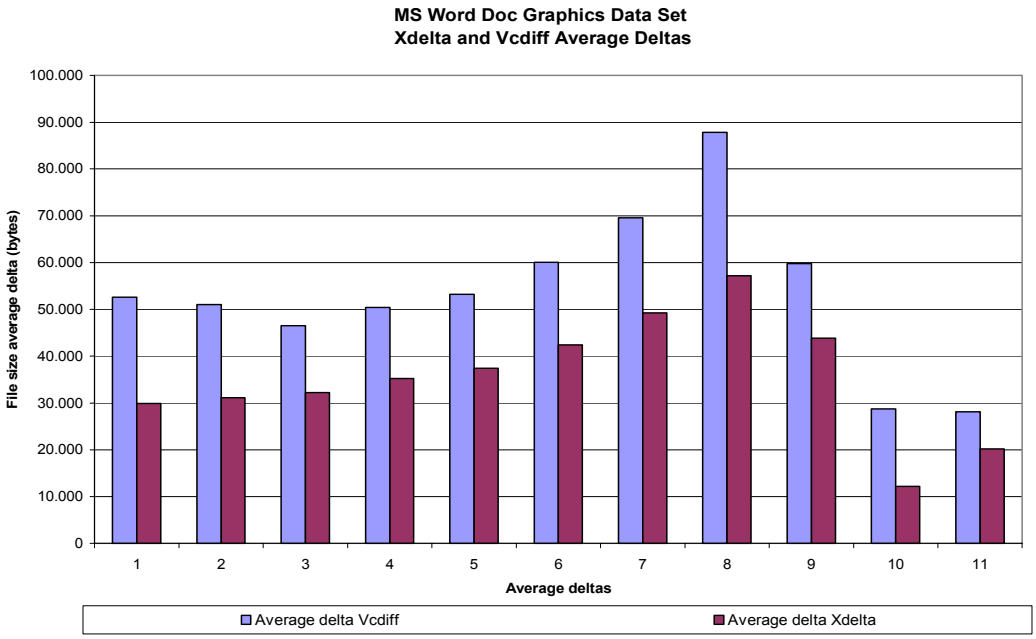

Figure 6-28: Average deltas of Xdelta and Vcdiff on MS Word Doc Graphics Data Set.

7 Recommendations

This chapter is based on the results and insights that were obtained during this project. Several recommendations can be made towards the RNLA and these fall apart in a technical and non-technical part.

7.1 Technical part

The results of the tests give some clues of how the problem of efficient transfer of (MS) Office documents can be tackled by using a technical approach. Several approaches can be considered and these are listed beneath.

7.1.1 File synchronization

Highly distributed data may be synchronized in the future, however at the moment there do not seem to be any promising applications that can cope with the inherent complex networks and highly distributed data of the military. However it may be wise to watch the progress of these applications as they may become meaningful for the RNLA.

7.1.2 Open standards

The advantages of XML as a primary source of information have been stressed a couple of times during this project. Terms like openness, flexibility and standardization are characteristics that refer to XML. Some benefits of openness are [28]:

- Availability: open standards are available for all people to read and implement.
- Maximize end-user choice: open standards do not force the customer to choose for a particular vendor.
- No royalty: everyone can implement an open standard without any fee. However certification of compliance by a standards organization may involve a fee.

At the moment the RNLA frequently uses COTS products to incorporate state of the art technology in their systems [6]. As open standards have matured to the point where they are a serious competitor (see the section about OpenOffice.org in the non-technical part of this chapter) to their commercial counterparts, open standards may offer the same or even better performance than the commercial products that are not free, unavailable for people to read and implement and less easily extensible. Open standards may make things more easy when one for example would like to extend software to include techniques for a more efficient file transfer.

7.1.3 XMill vs ZIP

XMill only compresses XML files and can be extended with self-made compressors in order to enhance the compression rate. This means that military systems which use XML to store or exchange data can reduce file sizes by using XMill as a primary compression tool. By compressing files with XMill good results are gained. Furthermore it is possible to extend XMill with specialized compressors and/or to give it some clues of how certain data items are structured such that XMill outperforms compression tools like ZIP. XMill is slightly faster than ZIP but this difference is not really significant. As explained

earlier decompressing XMill compressed files can be done very fast if an application would directly accept SAX events, instead of having to re-parse the XML-string. When using XML data it is recommendable to use XMill as there is always gained some extra compression. The benchmark showed that files from the MS Word XML Only Text Data Set are compressed very efficiently by XMill. In the case where simple structured data has to be exchanged XMill can achieve very good compression rates by exploiting the structural knowledge of the data.

7.1.4 Delta compression tools

Some tools perform well only on specific data types and others perform well on a wider range of data types. The benchmark results of the preceding chapter clearly indicate that there are some differences between the performances of the various tools.

Table 7-1 gives a summary of the measured performances of the tools on the various data sets. The following notation is handled: xx/yy, where xx is the score for the size of the delta and yy is the score for the time that is required to calculate the (compressed) delta. The score is relative and a + + is assigned to a tool which performs the best and - - is assigned to a tool that performs the worst. A single + or – is also possible of course. The blue areas relate to the smallest deltas that can be achieved.

	Diff	Diff + ZIP	Xdelta	Vcdiff	Vcdiff + ZIP	DeltaXML	DeltaXML + ZIP
MS Word XML Graphics	- - / + +	+ / -	+ + / + +	+ / +	+ / + +	- - / - -	+ + / - -
MS Word XML Only Text	- - / + +	- - / + +	+ + / + +	+ / +	+ / +	- - / - -	+ / - -
MS Word Doc Graphics			+ + / + +	- - / - -	+ / - -		
MS Word Doc Only Text			+ + / + +	- - / - -	+ / - -		
OpenOffice.org Doc Graphics			+ + / + +	+ / - -	+ / - -		
OpenOffice.org Doc Only Text			+ + / + +	- - / +	+ / +		
Open Office.org xsw Graphics			+ + / +	+ / + +	+ + / +		
Open Office.org xsw Only Text			+ + / +	+ / + +	+ + / +		
MS Excel xls			+ + / + +	- - / - -	+ / - -		
MS Excel XML	- - / + +	- - / + +	+ + / + +	+ / +	+ / +	- - / - -	+ / - -
OpenOffice.org Calc xls			+ + / +	- - / + +	+ + / + +		
OpenOffice.org Calc sxc			+ + / + +	+ + / + +	+ + / + +		

Table 7-1: Comprehensive score card for the tools with respect to the data sets.

In almost any case Xdelta turns out to be the best delta compression tool with respect to the developed data sets. Xdelta is continuously the fastest and delivers the smallest deltas. Compressing the output is even unnecessary as the deltas are already saved efficiently enough.

Vcdiff returns (vcdiff –d) deltas that are usually larger than those of Xdelta and it is usually somewhat slower than Xdelta. However Vcdiff can also be run with the option to compress the delta (vcdiff –cd), however this option was not included in the benchmark and instead the deltas were zipped.

DeltaXML does not produce any small deltas (although some of the ZIP compressed deltas are small too) and it really needs much processing time to generate a delta. DeltaXML may not be very suitable for Office documents, but according to the developers it may certainly be meaningful for data exchange applications.

The famous diff is very impractical for delta compression purposes as the deltas are not efficient enough. Furthermore diff can only be applied to plain text files which makes it useless for many formats.

Except for Xdelta it is strongly recommended to further compress the delta. By using ZIP or XMill (XMill can only be used in the case where deltas are XML formatted) deltas can decrease in file size

significantly. The largest decrease can be observed when XML formatted deltas are compressed by XMill.

The advantage of using (delta) compression tools is smaller when OpenOffice.org is used because the native file formats are already saved very efficiently. By using delta compression some reduction in file sizes can even still be gained though.

The best combination possible is to create the Office documents in OpenOffice.org after which Xdelta can compare the documents and generate highly efficient deltas.

7.2 Non technical part

This part of the recommendations will treat the non-technical aspects that can influence the efficiency of the transfer of (MS) Office documents. It tries to inform the reader which factors can contribute to a more efficient transfer even before any technical solutions have been applied.

7.2.1 Control user behavior

A MS PowerPoint presentation tends to be a large consumer of memory. Strict standards should be designed to overcome this practical problem. At the moment some people create very fancy presentations to impress their superiors. Adding useless animations and large pictures makes presentations needlessly large. Special protocols, that are principally used everywhere in the army, can be designed to create some delimitation of what is allowed to be made in MS PowerPoint. This will certainly result in a heavy decrease of the file sizes which on its turn results in a better bandwidth utilization. At the other end strict rules also cause presentations to be more uniform which makes them better to read. Use of colors, pictures, animations and sounds which do not contribute to a better understanding will then be omitted. People also like to scan pictures from paper and place it directly into an application. Resizing and compressing with appropriate image compression formats like png and jpeg decreases the total file size too. Superiors should play an exemplary role and should discourage the needless use of memory consuming elements.

7.2.2 OpenOffice.org

At the official FAQ of the web site of OpenOffice.org [8] OpenOffice.org is described as the open source project through which Sun Microsystems has released the technology for the popular StarOffice Productivity Suite. All of the StarOffice source code is available under the GNU Lesser General Public License (LGPL) as well as the Sun Industry Standards Source License (SISSL). Sun is participating as a member of the OpenOffice.org community.

From the unofficial FAQ [8] the role of OpenOffice.org is explained as follows: OpenOffice.org is the project behind the multi-platform, free office package called OpenOffice.org 1.1 consisting of applications such as a word processor, spreadsheet and presentation software, that has a similar codebase to Sun Microsystem's StarOffice. OOo as its commonly called is the alternative to using a paid package like Microsoft Office. It currently runs on Windows, Linux, Mac OS X, FreeBSD, and Solaris.

Because the data sets used in the benchmark indicate that OpenOffice.org documents are much more efficiently saved then their MS Office counterparts it may be particularly interesting to compare the two packages with each other.

In an article from eWEEK [7] much time has been spent with comparing the overall accepted MS Office 2003 to the less known OpenOffice.org. In last years, open-source alternatives to MS Office have become more sophisticated and IT managers are seriously looking at the possibility of moving from the Microsoft Office suite to a license-free alternative. In the article of eWEEK a user-based comparison between the OpenOffice.org project's OpenOffice.org suite and Microsoft's Office 2003 is made.

In this user-based comparison most users already use MS Office 97 or MS Office 2000. During the tests most of the users had almost no trouble moving from MS Office to OpenOffice.org. However advanced users, particularly those who work with MS Excel were of the opinion that OpenOffice.org was less ideal. This is because advanced users already came close to the limits of MS Office and some specialized features were just not part of OpenOffice.org.

Users who tested MS Office 2003 said that the suite was more polished and easy to use than MS Office 97 and 2000. The advanced users of MS Excel found out that MS Excel 2003 provides significantly more functionality than the preceding one.

All users liked MS Office 2003 and said it would be the smoothest upgrade as the user interface of MS Office is found the most user friendly. However for the average user some training will help to get used to the interface of OpenOffice.org soon. There are some differences between MS Office and OpenOffice.org which must be overcome, like different key combinations and other small things. OpenOffice.org Writer presented the fewest file-format-compatibility problems. Many users agreed that familiarity with a MS Office product will minimize the time required to get used to the OpenOffice.org alternative.

It is worth mentioning that OpenOffice.org and MS Office differ in the case where fancy markup is used. Very specific elements that are used in MS Office are not always understood by OpenOffice.org which does not always make converting a very easy task. Some people who are getting used to OpenOffice.org even prefer the way the applications are organized. Furthermore OpenOffice.org produces relatively small files that do not necessarily have to be compressed further which reduces processing time. At the other end some users complained that it took more time to load a similar file in OpenOffice.org than in MS Office 2003. This may be the result of the fact that OpenOffice.org makes use of Java for some features.

People who are used to work with macros in MS Office must be aware of the fact that OpenOffice.org uses a different version of Basic than MS Office, so macros created in the MS Office will not work in OpenOffice.org.

7.3 Further research

In this project the emphasis lies clearly on compression tools and techniques to decrease file sizes which eventually will contribute to a decrease in network traffic. Besides compression another important factor is related to the distribution. The compressed data should be distributed by using an algorithm that tries to find an efficient way of propagating data from one point to the other through the network. Already existent protocols can be investigated and perhaps these can be adapted to the specific needs of the RNLA. When distributing data clients and servers must be aware of the hosts in the network. The hosts need to communicate with each other to get an up to date picture of the network.

Unfortunately due to a lack of time MS PowerPoint and OpenOffice.org Impress data sets could not be developed. As MS PowerPoint is used for all kind of purposes in the army it would be interesting to

develop a realistic version history which can be used as an input for the (delta) compression tools of the benchmark. By making some small adjustments to the benchmark (see appendix) the performance of the various tools on MS PowerPoint and OpenOffice.org Impress documents can be measured.

Here is a hint: MS PowerPoint documents are also widely used and their file size is relatively big. It is not extraordinary to find a 10+ MB document. Pictures and even animations are not seldom integrated in a standard presentation. Some MS PowerPoint files can be used in the benchmark and of course the file size may exceed 10+ MB. Parallel to the MS PowerPoint documents there should be made similar OpenOffice.org Impress documents. The latter can be saved as a compressed format that also makes use of XML data parts, contrary to MS PowerPoint 2003.

In the field investigation one of the questions was: “Which part of the bandwidth is utilized by the transfer of documents?”. This question could not be answered at the time as this would implicate that much monitoring work should be done to retrieve the information. It is important to understand that the (delta) compression tools used will only substantially contribute to a better bandwidth utilization if the documents themselves utilize a considerable part of the bandwidth. For example assume that MS Office documents are responsible for approximately 60 % of the total available bandwidth of a specific link, then any possible reduction in file size of these documents would be a great benefit to the total availability of the communication link. The previous chapter shows that the combination of compressing (ZIP and XMill) and differencing (diff, Xdelta, Vcdiff, DeltaXML) can largely reduce a file size (by at least 50%).

Acronyms

C2	Command & Control
C2SC	C2 Support Centre
C2WS	C2 Work Station
CIS	Communication & Information Systems
COTS	Commercial Off The Shelf
CVS	Concurrent Versions System
C3I	Command, Control, Communications & Intelligence
DTD	Document Type Definition
ESDI	European Strategic Defense Initiative
FAQ	Frequently Asked Questions
ISIS	Integrated Staff Information System
JAXP	Java API for XML Processing
RNLA	The Royal Netherlands Army
SAX	Simple API for XML
TrAX	Transformation API for XML
VoIP	Voice over IP

References

[1] Algorithms for Delta Compression and Remote File Synchronization
Torsten Suel & Nasir Memon

[2] XML Sizing and Compression Study for Military Wireless Data
Michael Cokus & Daniel Winkowski

[3] XMill: an Efficient Compressor for XML Data
Hartmut Liefke & Dan Suciu

[4] Manual CVS, Concurrent Versions System

[5] An Algorithm for Differential File Comparison
J.W. Hunt & M.D. McIlroy

[6] Web Site Command & Control Support Centre
http://www.c2sc.org

[7] Web Site Article eWEEK
http://www.eweek.com

[8] Web Site Open Office
http://www.openoffice.org

[9] Web Site O'Reilly XML.com
http://www.xml.com

[10] Web Site rsync
http://samba.anu.edu.au/rsync/

[11] Web Site Naval Institute
http://www.usni.org

[12] Web Site Iconmedia
http://www.iconmedia.com

[13] Network Centric Warfare, Developing and Leveraging Information Superiority
David S. Alberts, John J. Garstka and Frederick P. Stein

[14] TMS 4 "The TITAAN (phase 2/3) mobile messaging system"
RNLA

[15] C2WS3001 SUM ISIS 3.0 on C2WS
RNLA

[16] Web Site TheFreeDictionairy.com
http://www.encyclopedia.thefreedictionary.com

[17] Web Site University Oslo
http://www.ifi.uio.no/

[18] Information Theory for Information Technologists
M. J. Usher

[19] XML Sizing and Compression Study For Military Wireless Data
Michael Cokus & Daniel Winkowski

[20] The string-to-string correction problem with block moves
W. Tichy

[21] Versioned File Archiving , Compression, and Distribution
Josh MacDonald

[22] RFC 3284 The VCDIFF Generic Differencing and Compression Data Format
David G. Korn, J. MacDonald, Jeffrey C. Mogul, Kiem-Phong Vo

[23] An Approach for Solving the Constrained Longest Common Subsequence Problem
Chao-Li Peng

[24] Change Control for XML: Do It Right
DeltaXML

[25] A Delta Format for XML: Identifying Changes in XML Files and Representing the Changes in XML
DeltaXML

[26] Web Site Rsync
http://samba.anu.edu.au/rsync/

[27] Web Site World Wide Web Consortium
http://www.w3.org

[28] Web Site Bruce Perens
http://www.perens.com

A. Appendix

Benchmark script

```
#!usr/bin/perl -w

use Time::HiRes qw(gettimeofday tv_interval);
use Cwd;

# In all data sets there are 12 files

$max_comparisons = 12;

# To get an average value of all measurements, every measurement is done
# several times.

$iterations = 10;

# Delta part

sub delta
{
   $tool = shift(@_);

   for ($i=1; $i <= $max_comparisons-1; $i++)
   {
      for ($next=$i+1; $next <= $max_comparisons; $next++)
      {
         $cwd = getcwd();

         if ($cwd =~ /XML/)
         {
            $filename1 = "Data".$i.".xml";
            $filename2 = "Data".$next.".xml";
         }
         elsif ($cwd =~ /Doc/)
         {
            $filename1 = "Data".$i.".doc";
            $filename2 = "Data".$next.".doc";
         }
         elsif ($cwd =~ /Native/)
         {
            $filename1 = "Data".$i.".sxw";
            $filename2 = "Data".$next.".sxw";
         }

         @filestat1 = stat ($filename1);
         $size_old_file = $filestat1[7];

         @filestat1 = stat ($filename2);
         $size_new_file = $filestat1[7];

         # Zipping the new file

         $file_new_zipped = "file_new_".$next.".zip";
         $command1 = "zip $file_new_zipped $filename2";
         system($command1);

         @filestat1 = stat ($file_new_zipped);
         $size_new_file_zipped = $filestat1[7];

         # Configuring arguments for the specific tool and giving
         # the statistics file the right name

         $uncompressed_delta = "uncompressed_delta_".$i."-".$next.".txt";

         if ($tool eq "diff")
         {
            $command1 = "diff $filename1 $filename2 > $uncompressed_delta";
            $datafile = "delta_diff_data.txt";
         }
         elsif ($tool eq "xdelta")
         {
            $command1 = "xdelta delta $filename1 $filename2 $uncompressed_delta";
            $datafile = "delta_xdelta_data.txt";
         }
         elsif ($tool eq "vcdiff")
         {
            $command1 = "vcdiff -d $filename1 < $filename2 > $uncompressed_delta";
            $datafile = "delta_vcdiff_data.txt";
         }
         elsif ($tool eq "deltaxml")
         {
            $command1 = "java -jar /home/infstud/mbroekma/benchmark/Tools/deltaxml/DeltaXMLAPI-2_8_1/command.jar compare --
                  raw-xml-output --changes-only $filename1 $filename2 $uncompressed_delta";
            $datafile = "delta_deltaxml_data.txt";
         }

         # Every measurement is done several times

         for ($j=1; $j <= $iterations; $j++)
         {
            # The delta is written to an output file

            $time0 = [gettimeofday];
```

```
            system ($command1);

            $time1 = [gettimeofday];

            # The ouput file is zipped

            $zipped_delta = "zipped_delta_".$i."-".$next.".zip";
            $command2 = "zip $zipped_delta $uncompressed_delta";
            system($command2);

            $time2 = [gettimeofday];

            # Inspecting the file sizes

            @filestat1 = stat ($uncompressed_delta);
            $size_uncompressed_delta = $filestat1[7];
            @filestat1 = stat ($zipped_delta);
            $size_zipped_delta = $filestat1[7];

            # Inspecting time intervals

            $time_delta_uncompressed = tv_interval ($time0, $time1);
            $time_delta_zipped = tv_interval ($time1, $time2);
            $time_total_computation = tv_interval ($time0, $time2);

            # Writing all data to arrays

            $definitive_values[$i][$next][0] = $size_old_file;
            $definitive_values[$i][$next][1] = $size_new_file;
            $definitive_values[$i][$next][2] = $size_new_file_zipped;
            $definitive_values[$i][$next][3] = $size_uncompressed_delta;
            $definitive_values[$i][$next][4] = $size_zipped_delta;

            $temp_values[$i][$next][$j][0] = $time_delta_uncompressed;
            $temp_values[$i][$next][$j][1] = $time_delta_zipped;
            $temp_values[$i][$next][$j][2] = $time_total_computation;
         }

         # Calculating average time values

         $total = 0;

         for ($k=1; $k <= $iterations; $k++)
         {
            $total = $total + $temp_values[$i][$next][$k][0];
         }

         $average_time_delta_uncompressed = $total/$iterations;

         $total=0;

         for ($k=1; $k <= $iterations; $k++)
         {
            $total = $total + $temp_values[$i][$next][$k][1];
         }

         $average_time_delta_zipped = $total/$iterations;

         $total=0;

         for ($k=1; $k <= $iterations; $k++)
         {
            $total = $total + $temp_values[$i][$next][$k][2];
         }

         $average_time_total_computation = $total/$iterations;

         $definitive_values[$i][$next][5] = $average_time_delta_uncompressed;
         $definitive_values[$i][$next][6] = $average_time_delta_zipped;
         $definitive_values[$i][$next][7] = $average_time_total_computation;
      }
   }

   # Writing all benchmark data to a text file

   open(FILE,">$datafile") or die "$datafile could nog be created\n";

   for ($m=1; $m <= $max_comparisons-1; $m++)
   {
      for ($n=$m+1; $n <= $max_comparisons; $n++)
      {
         print FILE "$m-$n \t
               $definitive_values[$m][$n][0] \t
               $definitive_values[$m][$n][1] \t
               $definitive_values[$m][$n][2] \t
               $definitive_values[$m][$n][3] \t
               $definitive_values[$m][$n][4] \t
               $definitive_values[$m][$n][5] \t
               $definitive_values[$m][$n][6] \t
               $definitive_values[$m][$n][7] \n";
      }
   }

   close(FILE);
}

# Patch part

sub patch
{
   $tool = shift(@_);
```

```
for ($i=1; $i <= $max_comparisons-1; $i++)
{
      for ($next=$i+1; $next <= $max_comparisons; $next++)
      {
         $zipped_delta = "zipped_delta_".$i."-".$next.".zip";
         $uncompressed_delta = "uncompressed_delta_".$i."-".$next.".txt";

         $cwd = getcwd();

         if ($cwd =~ /XML/)
         {
            $filename1 = "Data".$i.".xml";
            $filename_new = "Data".$next."b.xml";
         }
         elsif ($cwd =~ /Doc/)
         {
            $filename1 = "Data".$i.".doc";
            $filename_new = "Data".$next."b.doc";
         }
         elsif ($cwd =~ /Native/)
         {
            $filename1 = "Data".$i.".sxw";
            $filename_new = "Data".$next."b.sxw";
         }

         if ($tool eq "diff")
         {
            $command1 = "patch -o $filename_new $filename1 $uncompressed_delta";
            $datafile = "patch_diff_data.txt";
         }
         elsif ($tool eq "xdelta")
         {
            $command1 = "xdelta patch $uncompressed_delta $filename1 $filename_new";
            $datafile = "patch_xdelta_data.txt";
         }
         elsif ($tool eq "vcdiff")
         {
            $command1 = "vcundiff $filename1 < $uncompressed_delta > $filename_new";
            $datafile = "patch_vcdiff_data.txt";
         }
         elsif ($tool eq "deltaxml")
         {
            $command1 = "java -jar /home/infstud/mbroekma/benchmark/Tools/deltaxml/DeltaXMLAPI-2_8_1/command.jar
                         recombine-forward $filename1 $uncompressed_delta $filename_new";
            $datafile = "patch_deltaxml_data.txt";
         }

         for ($j=1; $j <= $iterations; $j++)
         {
            $time0 = [gettimeofday];

            # The delta is unzipped

            $command2 = "unzip -o $zipped_delta $uncompressed_delta";
            system($command2);

            $time1 = [gettimeofday];

            # The old file is patched to get the new file

            system ($command1);

            $time2 = [gettimeofday];

            # Inspecting the file sizes

            @filestat1 = stat ($filename1);
            $size_old_file = $filestat1[7];

            @filestat1 = stat ($zipped_delta);
            $size_zipped_delta = $filestat1[7];

            @filestat1 = stat ($uncompressed_delta);
            $size_uncompressed_delta = $filestat1[7];

            @filestat1 = stat ($filename_new);
            $size_new_file = $filestat1[7];

            # Inspecting time intervals

            $time_unzipping = tv_interval ($time0, $time1);
            $time_patching = tv_interval ($time1, $time2);
            $time_total_computation = tv_interval ($time0, $time2);

            $definitive_values[$i][$next][0] = $size_old_file;
            $definitive_values[$i][$next][1] = $size_new_file;
            $definitive_values[$i][$next][2] = $size_zipped_delta;
            $definitive_values[$i][$next][3] = $size_uncompressed_delta;

            $temp_values[$i][$next][$j][0] = $time_unzipping;
            $temp_values[$i][$next][$j][1] = $time_patching;
            $temp_values[$i][$next][$j][2] = $time_total_computation;
         }

         # Calculating average time values

         $total = 0;

         for ($k=1; $k <= $iterations; $k++)
         {
            $total = $total + $temp_values[$i][$next][$k][0];
```

```
            }

            $average_time_unzipping = $total/$iterations;

            $total=0;

            for ($k=1; $k <= $iterations; $k++)
            {
               $total = $total + $temp_values[$i][$next][$k][1];
            }

            $average_time_patching = $total/$iterations;

            $total=0;

            for ($k=1; $k <= $iterations; $k++)
            {
               $total = $total + $temp_values[$i][$next][$k][2];
            }

            $average_time_total_computation = $total/$iterations;

            $definitive_values[$i][$next][4] = $average_time_unzipping;
            $definitive_values[$i][$next][5] = $average_time_patching;
            $definitive_values[$i][$next][6] = $average_time_total_computation;
         }
   }

   # Writing all benchmark data to a text file

   open(FILE,">$datafile") or die "$datafile could nog be created\n";

   for ($m=1; $m <= $max_comparisons-1; $m++)
   {
      for ($n=$m+1; $n <= $max_comparisons; $n++)
      {
         print FILE "$m-$n \t
            $definitive_values[$m][$n][0] \t
            $definitive_values[$m][$n][1] \t
            $definitive_values[$m][$n][2] \t
            $definitive_values[$m][$n][3] \t
            $definitive_values[$m][$n][4] \t
            $definitive_values[$m][$n][5] \t
            $definitive_values[$m][$n][6]\n";
      }
   }

   close(FILE);
}

# Main procedure which calls all other procedures

sub doEverything
{
   $time_start_benchmark = [gettimeofday];

   # diff

   @diff_dir = ('/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_XML/Graphics',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_XML/Only_text');

   foreach my $dir (@diff_dir)
   {
      chdir $dir;
      delta("diff");
      patch("diff");
   }

   # xdelta

   @xdelta_dir = ('/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_XML/Graphics',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_XML/Only_text',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_Doc/Graphics',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_Doc/Only_text',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/Open_Office_Writer_Native/Graphics',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/Open_Office_Writer_Native/Only_text',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/Open_Office_Writer_Doc/Graphics',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/Open_Office_Writer_Doc/Only_text');

   foreach my $dir (@xdelta_dir)
   {
      chdir $dir;
      delta("xdelta");
      patch("xdelta");
   }

   # vcdiff

   @vcdiff_dir = ('/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_XML/Graphics',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_XML/Only_text',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_Doc/Graphics',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_Doc/Only_text',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/Open_Office_Writer_Native/Graphics',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/Open_Office_Writer_Native/Only_text',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/Open_Office_Writer_Doc/Graphics',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/Open_Office_Writer_Doc/Only_text');

   foreach my $dir (@vcdiff_dir)
   {
      chdir $dir;
      delta("vcdiff");
      patch("vcdiff");
```

```
   }

   # deltaxml

   @deltaxml_dir = ('/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_XML/Only_text',
         '/home/infstud/mbroekma/benchmark/Word_Files_Data_Sets/MS_Word_XML/Graphics');

   foreach my $dir (@deltaxml_dir)
   {
      chdir $dir;
      delta("deltaxml");
      patch("deltaxml");
   }

   $time_end_benchmark = [gettimeofday];

   # The benchmark is completed and the total computation time is calculated

   $total_time_running = tv_interval ($time_start_benchmark, $time_end_benchmark);

   print "\nBenchmark process completed in $total_time_running seconds\n";
}

doEverything;
exit(0);
```

Word/Writer Data Sets

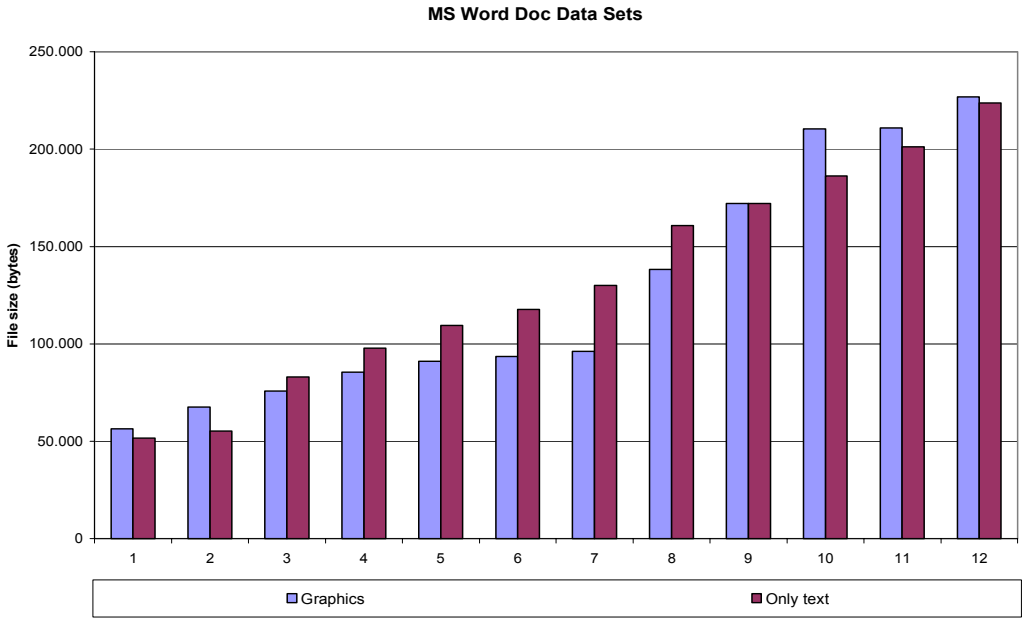

Figure 1: MS Word Doc Data Sets

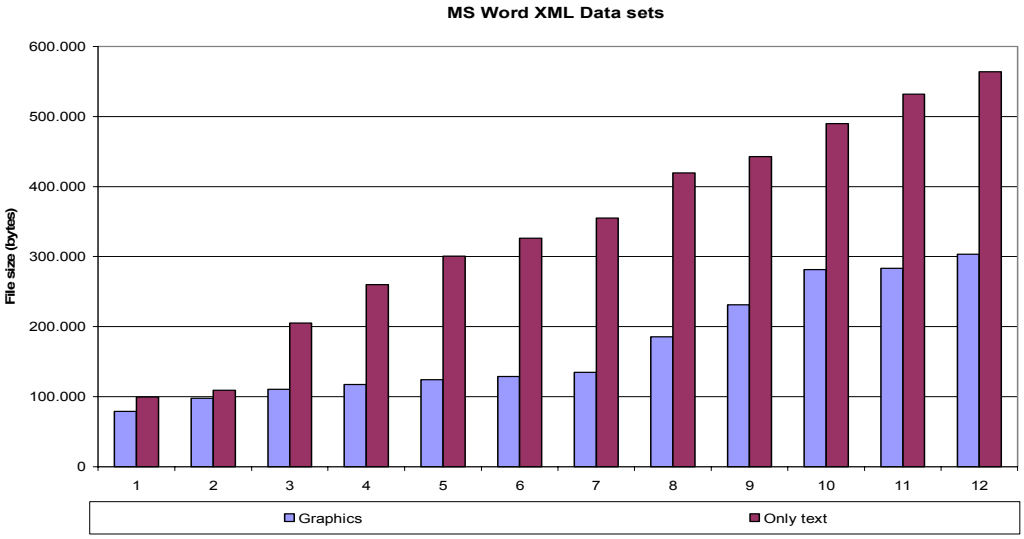

Figure 2: MS Word XML Data Sets.

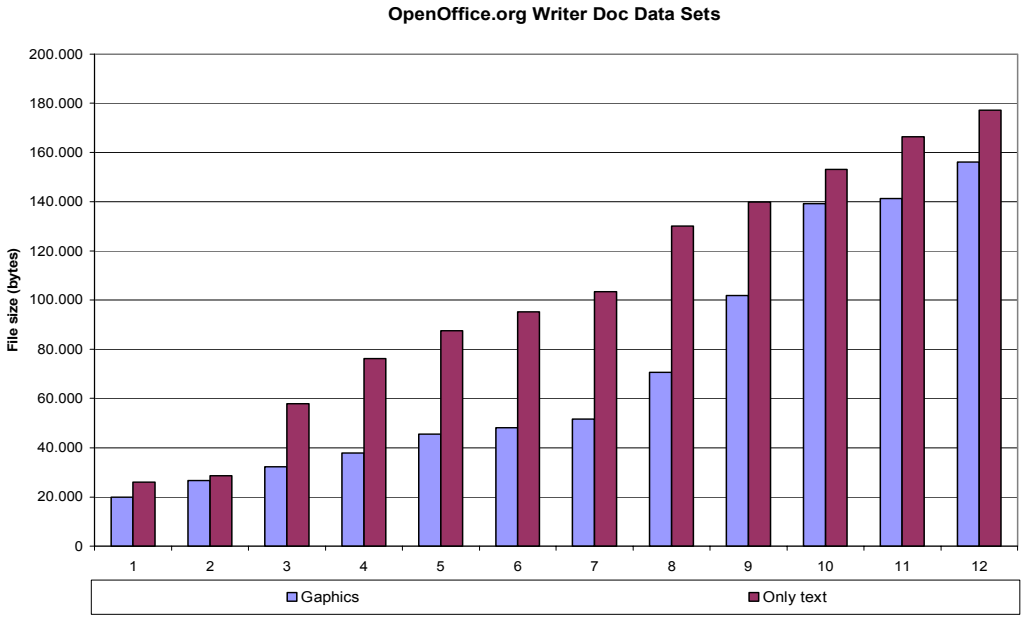

Figure 3: OpenOffice.org Writer Doc Data Sets.

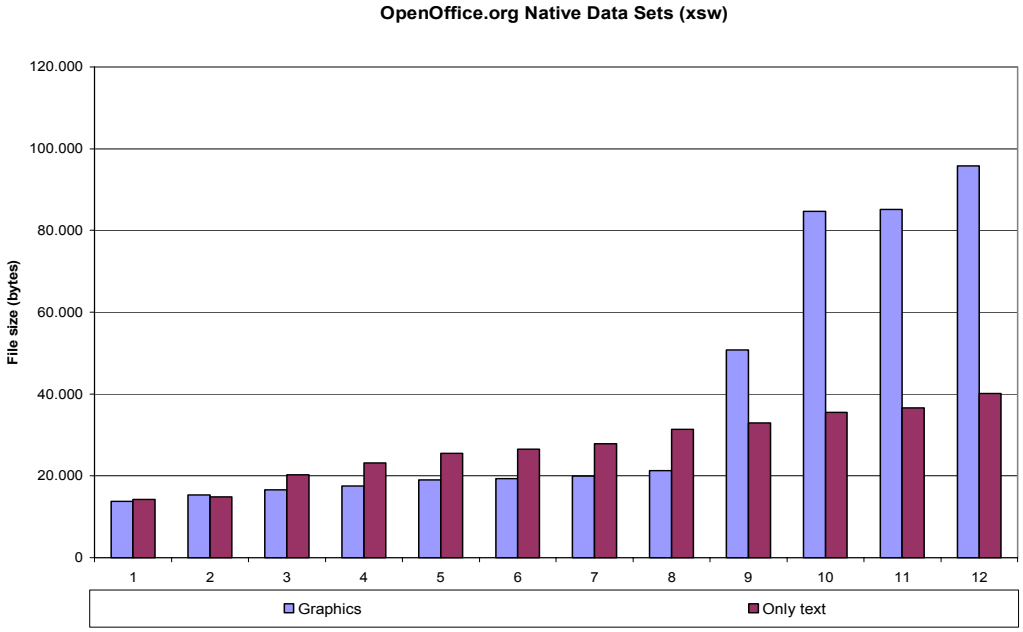

Figure 4: OpenOffice.org Writer Native Data Sets (xsw).

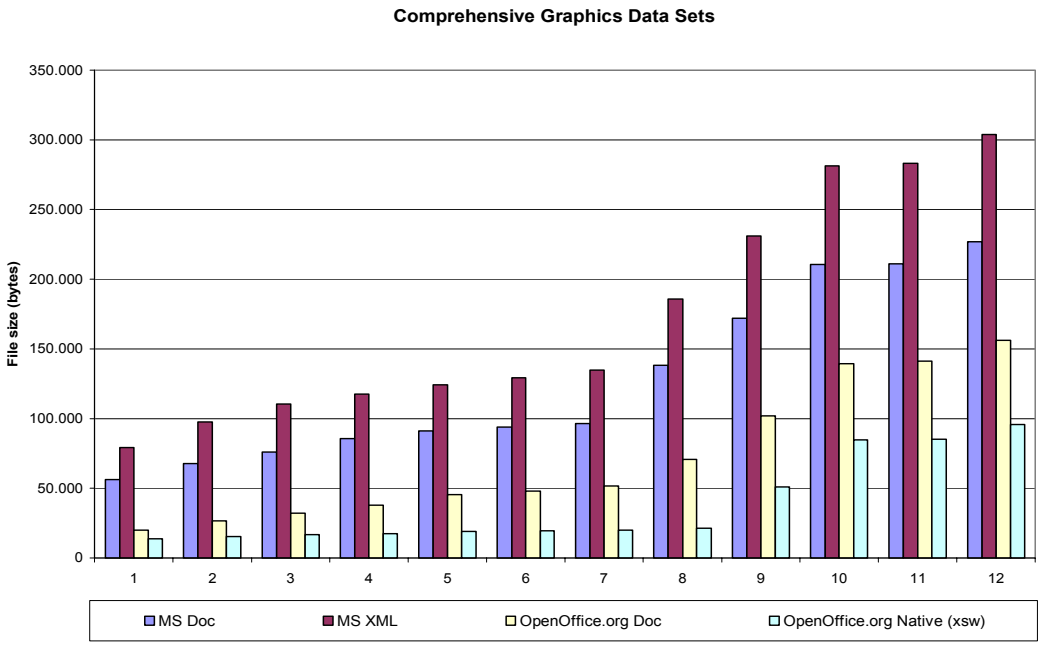

Figure 5: Comprehensive Graphics Data Sets.

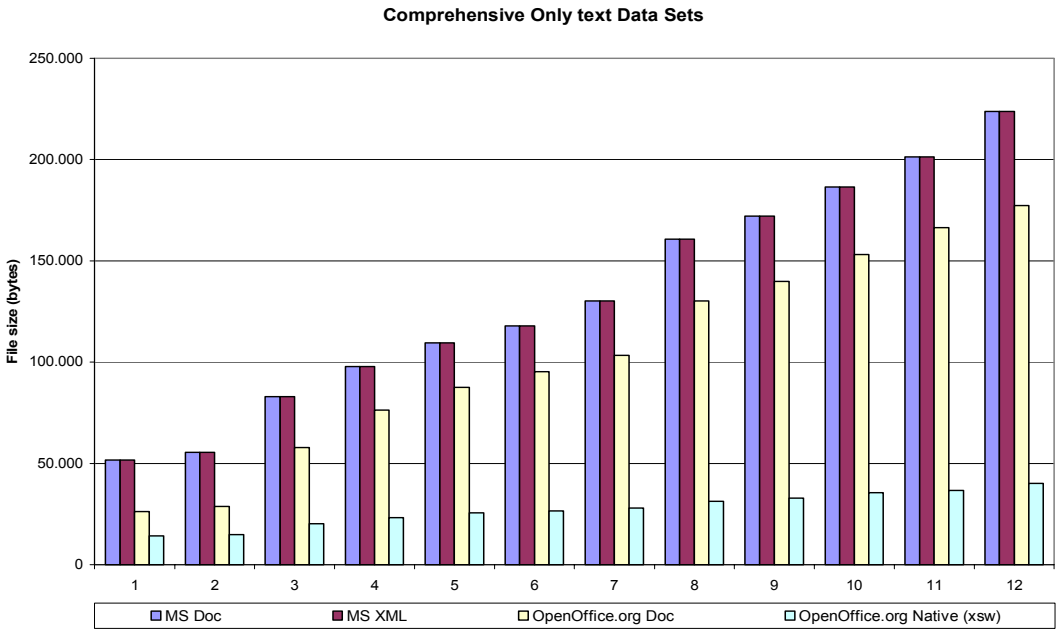

Figure 6: Comprehensive Only text Data Sets.

Excel/Calc Data Sets

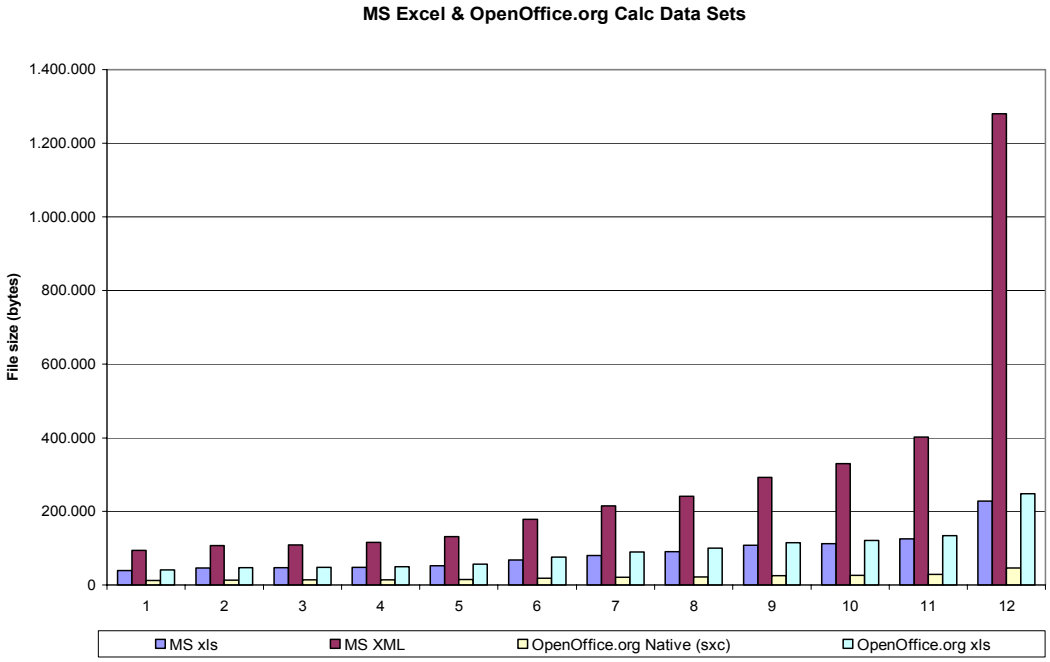

Figure 7: MS Excel & OpenOffice.org Calc Data Sets.