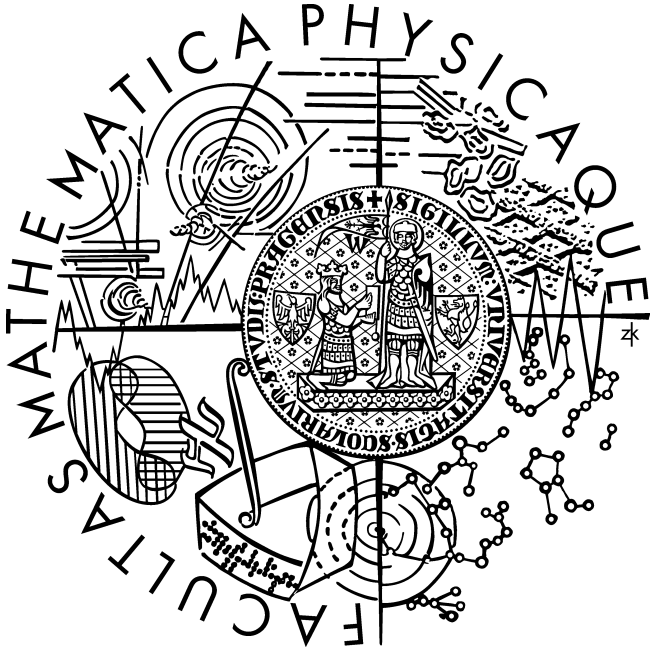
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Tomáš Filípek

Efficient Network Backup System

Department of Applied Mathematics (202. • 32-KAM)

Supervisor of the bachelor thesis: Mgr. Petr Baudiš

Study programme: Informatika

Specialization: Obecná informatika

Prague 2013

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature

Název práce: Efektivní síťový zálohovací systém

Autor: Tomáš Filípek

Katedra / Ústav: Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Petr Baudiš, Katedra aplikované matematiky

Abstrakt:

Pro ukládání vzájemně podobných souborů existují různé elegantní algoritmy, stejně jako pro jejich rychlé posílání po síti. Je však možné tyto algoritmy zkombinovat a vytvořit tak síťový zálohovací systém? Pokusili jsme se takový systém vytvořit, navíc s podporou verzování souborů a inteligentního udržování historie. Ve výsledném softwaru se toho podařilo dosáhnout, s logickým jádrem aplikace na serverové straně. Abychom mohli určit vhodné hodnoty některých parametrů použitých algoritmů, provedli jsme měření výkonu aplikace. S použitím naměřených optimálních hodnot program funguje rozumně efektivně v čase i prostoru.

Klíčová slova: rsync, záloha, Myers, síť

Title: Efficient Network Backup System

Author: Tomáš Filípek

Department / *Institute*: Department of Applied Mathematics

Supervisor of the bachelor thesis: Mgr. Petr Baudiš, Department of Applied Mathematics

Abstract:

There are elegant algorithms for storing similar files, as well as for sending them efficiently through a network. But can these algorithms be combined to form a back-up system? We tried to construct such a system, with the support for versioning and intelligent history keeping. The resulting software achieves it, with the logical core at the server side. To assess the right values for some of the algorithm parametres, a simple performance testing was performed. Using the chosen values, the application is reasonably efficient both in space and time.

Keywords: rsync, backup, Myers, network

Contents

Introduction / Preface

The main reason backup software exists, is to make sure you do not lose your data. Therefore, it would seem easy to say that its only important property is a high level of reliability. However, experience shows it is not a sufficient condition for making a system successful. The problem lies in making the software run efficiently both in space and time. Because, if every backup takes too much storage space, the user will think twice about doing the backup, which is bad. Backups should be done often and regularly, not only when it is absolutely necessary. The same applies to the case when the backup process takes too much time, which might happen when there is a slow connection between the client and server, for instance.

There are many decent solutions in the backup field available for free, some of which deal nicely with reducing the network traffic. This is usually done by employing the rsync algorithm. However, none of those makes use of the algorithm's internals to save disc space by finding similarities among files and storing the common sections only once.

The goal of this work is to provide a dependable backup system with a basic client – server architecture, capable of storing files paired with a reasonably long history. The fundamental aspect of the system will be its efficiency both in space and time, employing techniques from rsync and diff algorithms. Because many decisions about various algorithm parameters will have to be made, the program will be tested and according to the results, optimal parameters will be chosen. The testing methodology, results and conclusions will form an integral part of the work.

The thesis begins with an introduction to the algorithms and methods used in the system. Because no existing implementation of them proved useful, they have been completely reimplemented to match the needs of the system. Every change from the original algorithms will be thoroughly described in this section.

Next section introduces the project structure and implementation. Since we have chosen the client-server architecture, the used protocol specification is also included. Finally, as a part of the implementation section, the serialization process is described. The reason is, the application is built on the Java platform, which is object-oriented and the default serialization technique is not always the most efficient solution.

In the following section a simple user guide is presented, which describes how to run all the basic parts of the application – the server, client, scheduler ...

The last section presents the performance testing process. It begins with a discussion about what we measure and why, and follows with an overview of the employed testing methods and testing data. Finally, the testing results are discussed and it is explained what conclusions have been drawn from the results.

Related Works

rdiff-backup – A backup tool capable of copying an existing directory into another, possibly a remote one. It uses the rsync algorithm to reduce the network traffic. For each file, a simple history is maintained, which is achieved through usage of difference scripts. However, the software is not fully optimized for minimizing disc usage on the server side. There are no shared blocks of data among similar files. Also, the way difference scripts are used implies that only similarities among adjacent versions can be capitalized on. For example, if we change a file periodically in a way that all the odd and all the even versions are similar, no data redundancy is detected. Another minor issue is that rdiff-backup is not completely portable, since the Windows version has not gone through its testing phase yet.

rsync – The original program which introduced the rsync algorithm. It was intended to supersede *rcp*, so the main and only advantage over the prior is its bandwidth-efficient algorithm. No history is kept of the transferred files. Also, no solution for reducing the disc space usage is provided.

GNU diff – A classic utility used for comparing two text files and creating a difference script. Since 1980s, it employs Myers' algorithm, which is also used in this work. However, only text files are supported.

Duplicati – A complex back-up software suite, actually a .NET reimplementation of a different program Duplicity, which was limited to the Unix platform. It provides users with many useful features such as data encryption or support for cloud systems, but the core functionality is the same as with rdiff-backup. Thus, there is no extra emphasis on reducing disc space usage on the server side.

1. Overview of the employed algorithms

1.1 Rsync

Originally described by Andrew Tridgell in his thesis[1], rsync algorithm deals with efficient synchronization of files over a slow network. Its main advantage over a plain file copy program like *rcp* is that it sends just the parts of the file that have changed, taking advantage of an existing older copy of the file.

The process of coping a file from Sender to Recipient roughly consists of the following steps:

- If there is an existing older version of the file at the recipient machine, it parses the file into equally sized, non-overlapping blocks. Otherwise, a simple copy is carried off.
- For each block of data, a weak and strong hash values are computed and the sets of all the weak hash and strong hash values are sent to the Sender, along with the chosen block size N.
- Sender reads the input file with the “window“ method. That means at one time, only N adjacent bytes are looked at, starting with the section from position 0 (counting from 0) of length N. The window is gradually moved by one byte at a time to the next position, meaning the first byte is removed from the window and the first byte to the right of the window is added. The process ends when the right side of the window has reached the end of the file.
- For each position of the window, a weak hash value is computed. If it is found in the set received from the Server, the strong hash value is computed and searched for in the second set. If a match is found, it is assumed that the contents of the window are the same as that of a block on the Server. The strong hash function is assumed to be safe enough not to treat possible hash collisions.
- If the aforementioned match is encountered, all the previous unmatched data is sent to the Server. Then a pair of hash values identifying the current window contents is sent instead of the actual data. This step is essential to saving network bandwidth.
- The Server reconstructs the file and saves it.

Of course, for the algorithm to work reasonably fast, a special weak hash function must be utilized. In the original rsync, Adler-32 checksum has been used. Because it is computed very frequently – at almost every position of the input file, it is beneficial that the value can be obtained just from the following (k – position of the current window, N – length of the window):

- Adler32(k-1)
- data[k-1]
- data[k+N-1]

Therefore, it is not necessary to go through all the bytes in the block when computing weak hash value of the window moving to a next position in the input file.

The Adler-32 checksum is computed by keeping two auxiliary values A and B. Let

$$n = 2^{16}$$

$$A = \sum_{i=k}^{k+N-1} data[i] \pmod n$$

$$B = \sum_{i=k}^{k+N-1} (k+N-i) * data[i] \pmod n$$

Note that these values can be easily updated when moving the window by one byte:

$$A_{k+1} = A_k + data[k+N] - data[k]$$

$$B_{k+1} = B_k - (N * data[k]) + A_{k+1}$$

The actual checksum is then obtained by:

$$Adler32(k) = A_k + (n * B_k)$$

Note: The implementation used in the subject of this thesis slightly differs from the original rsync. Differences:

- n is set to $n = 2^{32}$
- Only weak hash values are downloaded to the Sender. Strong hash values are then compared online, through a special type of request to the Receiver.

1.2 Myers' Diff

Problem: Let A and B be two files. Find a shortest edit script, i.e. at least one shortest sequence of character deletions from A, character additions from B and no-operations, that will transform file A into file B.

Eugene Myers solved this problem using dynamic programming. We will begin with some definitions.

File A and B – finite sequences of characters

N (M, resp.) - the length of file A (B, resp.)

Operation – an addition of a character from B into A, a deletion of a character from A or an empty operation doing nothing

Shortest edit script – a shortest such sequence of operations that transforms A into B. There can be multiple shortest edit scripts, this denotes any of them.

Edit graph – as illustrated in picture 1, it is a lattice graph of size (M+1)*(N+1) with characters from A being paired with x-axis units and characters from B paired with y-axis units.

k-th diagonal – a line defined as $y = x - k$

The goal of this algorithm is to find a shortest path through the graph from [0,0] to [N,M], where only the following moves from point [x,y] are allowed:

- to [x+1,y] (a deletion from file A)
- to [x,y+1] (an insertion into A from B)
- if A[x] is equal to B[y], then a move to [x+1,y+1] is allowed (a diagonal move)

length of a path – the number of deletions and insertions along the path.

snake – a single deletion or insertion followed by zero or more diagonal moves. Thus, we have another definition of the path length: the number of snakes on the path

distance of [x,y] – a minimal number of snakes required to move from [0,0] to [x,y]The basic form of the algorithm:

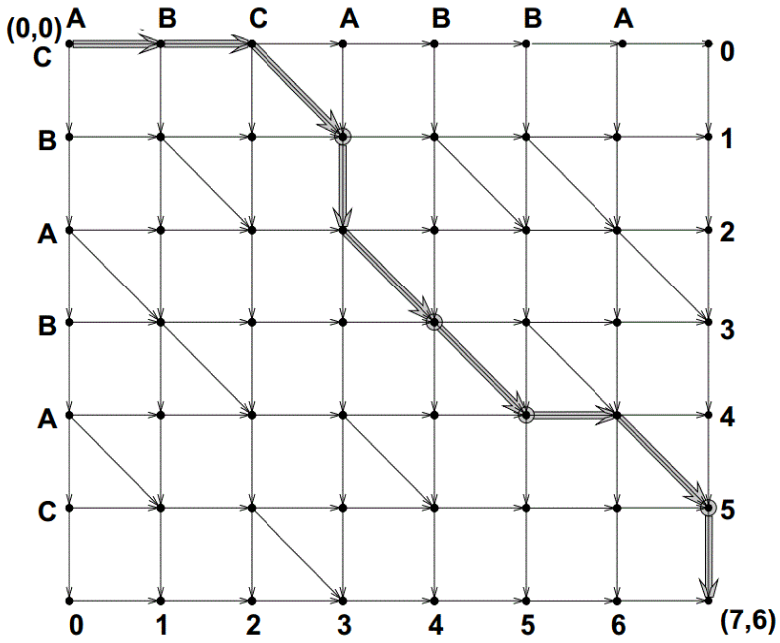
```
V := empty list of vectors
For each distance d, starting from 0, step 1:
      For each diagonal g,  from -d to d, step 2:
            find the longest reaching path LP from [0,0] , on the diagonal g
      Add a vector of the last snakes of the longest reaching paths to V.
      If the longest reaching path on any diagonal crosses the point [N,M], break
```

Construct the path from [0,0] to [N,M] by moving backwards through V and looking for the previous snakes that end in the start point of the current snake.

Example:

File A : ABCABBA

File B: CBABAC



How to find the longest reaching path on diagonal k and on distance d? Because a move along a snake changes the current diagonal by one, the last snake on the path must start at either (k-1)-diagonal or (k+1)-diagonal. Thus we can use the knowledge of the longest reaching paths on those diagonals at distance d-1.

Note: because at the odd (even) distances we can only reach to odd (even) diagonals, we can copy the furthest reaching snakes at odd (even) diagonals at an even (odd) distance from the vector at the previous distance. This allows us to take steps by 2 in the inner loop.

There is a drawback to this algorithm, though. Given length D of the shortest edit script, the algorithm needs to store D vectors of snakes, each containing at most N+M members. That gives us a space complexity of O(N+M)*D). For a rigorous analysis, see Myers [2].

1.3 A Linear Space Refinement for Diff

Let's define a middle snake as the $\frac{n+1}{2}$ -th snake on the path describing the shortest edit script. If we can find a middle snake, it is possible to construct the shortest edit script recursively, by using the divide-and-conquer method:

```
Shortest edit script path through (0..N)×(0..M) :
        If (M == 0) or (N==0), solution is trivial
        M <-- middle snake (suppose it start at [a,b], ends at [c,d])
        X <-- shortest edit script path through (0..a)×(0..b)
        Y <-- shortest edit script path through (c..N)×(d..M)
        Return concat(X,M,Y).
```

It can be seen that this algorithm only requires O(N+M) memory.

To find a middle snake, we can use a modified basic version of the Myers' algorithm. The principal differences are:

- We construct the longest reaching paths simultaneously both from [0,0] and [N,M] for each distance d.
- During the construction of the longest reaching paths, we only remember one vector of distances on the diagonals. This is important, because since each time the method is called, one snake of the final path is found, and so the method is called at most D times at any moment. This gives us intuitively a O(N+M) space complexity. For full analysis, see Myers[2].

The algorithm:

```
Find a middle snake through (0..N)×(0..M) :
        L <-- empty vector
        R <-- empty vector
        For each distance d, starting from 0, step 1:
                For each diagonal g,  from -d to d, step 2:
                        find the longest reaching path LP from [0,0] , on the diagonal g
                        L[g] <-- last snake on the path LP
                For each diagonal g,  from -d to d, step 2:
                        find the longest reaching path RP from [N,M] , on the diagonal g
                        R[g] <-- last snake on the path RP
                For each diagonal g,  from -d to d, step 2:
                        if L[g] and R[g] cross at any point, return L[g] as the middle snake
```

1.4 A Heuristic for Myers' Diff

The diff algorithm, as described in section 1.2, is good for finding the shortest edit script. However, in practice it is often preferable to find an edit script that is not the shortest, but can be computed much faster. In this work, a heuristic developed by Paul Eggert is used, which guarantees that a middle snake is found within a constant distance from [0,0] in the edit graph. However, the returned middle snake may not conform to the definition, because it may either be another snake on the shortest path through the edit graph or it can even lie on a different (and longer) path.

The principal idea is that the search for the middle snake is conducted up to a certain distance from [0,0], and if the middle snake is not found until then, the search is stopped. For each diagonal, the longest reaching snakes are checked and the one that reaches the furthest from [0,0] gets returned as the middle snake, even though it may be a suboptimal result.

Here is the modified algorithm for finding a middle snake:

```
Lim <- a constant
Find a middle snake through  (0..N)×(0..M)  :
       L <-- empty vector
       R <-- empty vector
       For each distance d, starting from 0, step 1:
              For each diagonal g,  from -d to d, step 2:
                     find the longest reaching path LP from [0,0] , on the diagonal g
                     L[g] <-- last snake on the path LP
              For each diagonal g,  from -d to d, step 2:
                     find the longest reaching path RP from [N,M] , on the diagonal g
                     R[g] <-- last snake on the path RP
              For each diagonal g,  from -d to d, step 2:
                     if L[g] and R[g] cross at any point, return L[g] as the middle snake
              If d >= Lim then :
                     ls <- furthest reaching snake (measured from [0,0]) out of L
                     rs <- furthest reaching snake (measured from [N,M]) out of R
                     if (ls.end - [0,0]) >= ([N,M] - rs.end)
                            then return ls
                            else return rs
```

2. Description of the Implementation

2.1 Overview

The application is divided into two parts – one is server-side, the other is on the client machine. The former is mainly used for handling the data storage, i.e. dealing with blocks, log files, versions of files and composition and decomposition of files into/from blocks of data. It is operated through a network connection by the client part, which is, in contrast, more user-centric. Its main responsibilities include inspecting the newly inserted files, determining the parts of files to be sent and, importantly, communicating with the user.

The basic operations available at the client side:

- add a new file or directory into the system
- add a new version of an already present file
- obtain a file, either the latest or an older version
- delete a version of a file
- list all the contents

The server is able to store multiple versions of a file. Thus, it is essential to use an intelligent way of storing the file data, because various versions of a file will probably have some common data sections. Two approaches to storing file data are considered:

- blocks of data
- edit script

If the block-form is used, file data are parsed into blocks of a constant size, that is used globally for all files. These blocks and their hash values are useful when adding new files to the system - the incorporated rsync algorithm uses them to find matching sections in new files. The contents of each block are stored as a separate file. The process of determining the block size is described in a separate section.

If the script-form is used, the new version of the file is stored as an edit script relative to an existing older version of the file. The classic Myers' algorithm is implemented to compute the differences, along with a simple heuristic. However, the script-form is used only if the difference is small, otherwise the block-form seems more useful.

It is possible so specify a limit on the server database size. That means, if an addition of a new file should make the database size grow over the limit, the addition is rejected by the server and the client is informed. However, before that happens, the server tries to do a limited clean-up, which deletes all the block-form versions that are not the latest ones, including all the dependent script-form version. Since data blocks can be shared among all the present files and versions, it is necessary to monitor the reference count for each block to allow for deletion of unused blocks. The aforementioned cleanup first decrements the appropriate reference count for some blocks, and only after this phase is finished proceeds to the actual deletion of nowhere referenced blocks.

The server is able to serve multiple clients in parallel, where each client is served by a single thread. However, because of the character of the application, the space for parallelization is limited – critical sections including access to the file database must always run in exclusive mode.

The client part of the application consists of a functional core, bundled with a simple command line interface and a more user-friendly graphical interface.

On the client side, there is also a scheduler, which posts a set of previously defined requests to the server in fixed time intervals.

2.2 Format of the Serialized Data Structures

The software is intended to be fully platform independent, so the format for data interchange over a network does not preserve any Java-specific or platform-specific features.
Variables of primitive types are serialized the following way:

- *byte*: byte variable is simply represented by its value
- *short, int, long* : each type is represented as consecutive bytes in big-endian order - *short* as two consecutive bytes, like as if the *short* has been cut into two bytes where the most important bits are in the first byte. *int* as four bytes and similarly, *long* as eight bytes.
- *float, double*: float variable is at first converted to the IEEE 754 format. Then, this *float* is looked upon as an *int* value containing the exactly same bits. The *int* variable is then used for serialization. d*ouble* is similarly converted to a *long* variable.
- *char*: variables of *char* type are handled the same way as *short* variables
- *boolean*: *true* is represented the same way as a *byte* variable of value 1, *false* corresponds to zero *byte.*

There is also an alternative way to serialize *int*s and *long*s. If it is used anywhere, it is explicitly noted so. This serialization is optimized for small values stored in the variable. The procedure is:

- The int/long variable is parsed into 7-bit blocks, starting at the least-important bits side.
- A bit (set to 1) is added to the left (most-important) side of each block, thus creating a byte from each block. The exception is the byte containing the most important bits of the original variable, where a zero bit is set, thus marking it is the final block.
- The bytes are then used in the little endian order, i.e. the byte containing the least important bits is sent first. This allows for sending just the non-zero blocks of the variable.

Enum variables are represented by their ordinal value, which is of *int* type. The actual ordinal values for the Requests class are mentioned later in this section.

The description of how reference types are handled follows.
String variables are handled in a special way. These steps are followed:

- String length (int value) is at the beginning. Each character of the String is represented as a series of bytes, where each byte is of a special format: the first one has special values at the two leftmost bits – the left bit signalizes that the string is encoded in UTF8, the right signalizes that another byte follows. The other six bits hold the actual data (the rightmost six bits of the int value). Next bytes are of this format: leftmost bit signalizes that another byte follows, the other seven bits hold the actual data (bits 12+k*7 to 6+k*7, where k=number of byte in the sequence minus two).
- The actual String value is sent as chars, one by one. Each char is sent as consecutive bytes: if the char is smaller than 0x007F, just one byte is sent. If the char is greater than 0x007F but smaller than 0x07FF, the sent bytes equal (0xC0 | char >> 6 & 0x1F) , (0x80 | char & 0x3F) . Else, if the char is greater than 0x07FF, three bytes are sent: (0xE0 | char >> 12 & 0x0F) , (0x80 | char >> 6 & 0x3F) , (0x80 | char & 0x3F) .
- If the String is known to be in ASCII format, a more efficient method is used. Each char is

serialized as a byte, with the leftmost bit set to 0. The end of String is signalized as an extra byte at the end, with the value 0x80 .

Array variables are serialized as the following items in succession:

- an “alternatively” serialized integer marking the size of the array
- all the items in the array in their serialized forms (ordered the same way as in the source array)

Other reference types are never serialized in this application, but there is an exception – the Database class and all the supporting classes, which are handled is a special way. The serialization format of those classes follows – the items always come in succession, with primitives being serialized the usual way.

DBlock:

- int (the position in the hash collision list)
- int (reference count)
- int (block size)
- int (number of valid/used bytes)
- long (weak hash)
- String (strong hash)

DDirectory:

- String (name)
- int (number of items contained in the directory)
- for every item contained in this directory:
  - String (name)
  - boolean (is it a directory?)
  - the serialized DFile/DDirectory object

DFile:

- int (number of versions the file contains)
- for each version: the serialized DVersion object
- int (number of items on the path to this file)
- for each item on the path to this file: String (the name of the item)

DVersion:

- long (date&time as milliseconds since 01-JAN-1970, 00-00-00)
- String (file name)
- int (block size)
- String (strong hash of the version contents)
- long (size of the version contents, in bytes)
- boolean (is it in a script form?)
- int (number of blocks)
- for each block: the serialized DBlock object

Database:

- int (the number of present blocks)
- for each block: the serialized DBlock object
- int (the number of distinct weak hash values used)
- all the weak hash values that are used – each as a long
- int (the number of distinct strong hash values used)
- all the strong hash values that are used – each as a String

- int (total number of regular files in the database)
- for each file: the serialized DFile object
- int (number of items in the root directory)
- for each item:
  - String (name)
  - boolean (is it a directory?)
  - the serialized item (DFile/DDirectory)

The *Requests* enum class has the following ordering of its members:
0 = ITEM_EXISTS, 1 = CREAT_FILE, 2 = CREAT_VERS, 3 = CHECK_CHANGES, 4 = CREAT_DIR, 5 = DEL_VERS, 6 = GET_FILE, 7 = GET_ZIP, 8 = END, 9 = GC, 10 = GET_D_ITEM, 11 = GET_FS

2.3 Server

The core functionality of the server part of the application is contained in the Server class. When run, the program accepts connections from possibly multiple clients. Each client is served by a dedicated thread. For the specification of the communication protocol between client and server, see section 2.3. In the following, we will discuss all the important data structures used on the server.

Most of the administrative data is stored in an instance of the Database class, with only the edit scripts being kept in an extra structure. The reason is that unlike some of the Database members, scripts never need to be sent across the network.

Most important Database class members:

- blockSet – a set containing weak hash values for each present data block.

  Main usage: during the rsync algorithm “window” phase on the client side, to quickly rule out presence of the current block

- blockSet2 – a set containing strong hash values for each present data block

  Main usage: to prevent problems arising from hash collisions in the weak hash

- regularFiles – a collection of all data structures representing regular files (not directories)

  Main usage: makes it fast to go through all the versions of all the files when determining what to delete, in case of disc space shortage

- blockMap – maps names of the files containing the block data to the corresponding objects that represent the blocks

  Main usage: speeds up searching for a block object, for example when receiving new version of a file that has not changed much

- fileMap – represents the file system structure of the files committed to the program. Contains top-level files and directories.

  Main usage: preserving a hierarchical file system of the committed files

The fileMap structure is worth further discussion. Generally, it simulates a simple file system, where only files and directories are allowed (no special files like a pipe or symlink). As the client uploads a file to the server, it gets added to this filesystem.

The directory nodes are implemented using the DDirectory class, which contains the directory name and a structure holding the directory contents, just as fileMap holds the root directory contents.

Regular files are represented by DFile class, which contains its name, path to the file, and a list of all the file versions.

File versions are implemented using DVersion class. It includes information about its size, date of creation, strong hash of the contents, name of the file, whether it is stored in the script-form, and finally, a list of blocks that contain the version data (if block-form is used).

Data blocks also have an abstract representation, which is provided by the DBlock class. It contains the weak and strong hash values of the block data, reference count, size of the window (as described in rsync section) when the block was created. Further data elements include the byte count, which can be different from the window size, because if the file is not exactly aligned to blocks, the remainders are saved as standalone blocks, albeit smaller. Finally, also the block's position in the weak hash collision list is also included.

There is also an important data structure included in the Server class:

- scripts – maps versions to the edit scripts that represent them.

  Main usage: for each version stored in the script-form, it contains the script, which is used to reconstruct the actual data contents of the version.

In the following, we will discuss in depth all the common tasks and how they are dealt with on the server.

- Adding a new file/version

At first, the client checks whether the file to be added is already present. That means finding the target path in fileMap. Then, if needed, a request to create a directory or file object is sent to the server. What follows is the request to create a new version of the file. It is first checked whether the file contents have changed since the last uploaded version. This is done by comparing the SHA-256 checksum values. If the checksums do not match, the process continues. A request to add a new version is sent to the server, followed by its size. If there is not enough disc space, the server tries to do some clean-up by deleting some less important older versions of big files (see next section). Next, client sends some identification data about the version it is about to upload. The window loop, as described in the rsync section, begins. After the complete version data is obtained, it is intelligently determined whether to save the version in blocks or as an edit script – for details see following sections.

- Clean-up of the server database

First, a list of all versions stored in the block form is constructed. Those that are the latest block-form versions of the corresponding file are left out. Then the version objects are deleted, along with all the script-form versions depending on it. For each block that any of the deleted versions contained, its reference count is decremented. Finally, a simple garbage collection is executed over the present blocks, to prevent storing unused blocks in the server storage.

- Determining how to store a version (edit script / blocks)

First it is checked whether there exists a version of the specified file, that is stored in the block form. If not, block form is chosen. It is then determined whether the new version contains any newly parsed blocks, or just some of the previously present blocks. If there has been a new block parsed, it is assumed that script form is preferred and the script creation begins. However, if the script size grows over a limit, which is defined equal to the block size, the computations terminates and block form is used instead. The edit script is always relative to the last block-form version.

The diff-ing mechanism uses basically the same algorithm as *GNU diff*, but contains a few customizations. First, it works exclusively with binary data. The elementary units, upon which the differences are calculated, are bytes. Another major modification is, it is possible to monitor the process of building the difference structure so that if the edit script grows over a specified limit, it can stop the construction and return appropriate status message.

- Obtaining a version of a file

At first a get request is received from a client. Then a copy of the DItem object is sent to the client, to help it decide about existence and type of the file to download, as well as about version numbers. If the *get zip* request was posted, the server delivers all the data as a single zip archive, otherwise a simple binary form is chosen. If the user wants to download a directory with all its contents, the client side of the application divides the request into multiple single file *get* requests. As a result, after receiving *get / get zip* command, the server sends contents of exactly one file.

- Zipping a file/directory

If the client requests a file/directory to be delivered as a ZIP archive, zipping process takes place after the relevant data are gathered. The built-in Java library implementation is used, and the archive is created in a ByteArrayOutputStream. Finally, the contents of the underlying byte array in the stream are sent to the client.

- Deleting a version of a file

After receiving version identification from the client, it is checked whether we are not trying to delete the last remaining version of the file. If that is the case, deletion is aborted. Otherwise, deletion proceeds. If there are any script-form versions that depend on this one, they are all converted to be relative to the adjacent version.

- Listing the contents of the server file database

Server sends the database file system root element object to the client. Further processing is done locally.

All of the server data, both administrative and file contents, are stored in a single directory, called “home directory” in the following sections. The server data structures are saved to disc each time a client disconnects. For serialization process specification, see section 2.2. There are two files for storing administrative data structures of the application. The first file, named “index”, includes an instance of the Database class. The second file is named “scripts” and contains the “scripts” data structure, as described above.

The contents of a data block are saved in a separate file. Its name consists of a hexadecimal representation of the block's weak hash value. However, if there are multiple different blocks with identical weak hash values, a suffix is added to the file name. Its form is “vXX”, where XX is a natural number denoting the position in the weak hash collision list. However, as blocks can be deleted, the hash collision list can have “holes” in it – once the position in the hash collision list is determined, the number never changes.

To send anything over the network to a client, the application uses an external serialization framework named Kryo. The main main reason for choosing Kryo over the default Java serialization functionality was performance – Kryo framework is very minimalistic, and if we provide extra serialization procedure for each custom (not from Java library) class, it does no extra work than specified in the serialization procedures. The exact specification of how the custom classes are serialized, see section 2.2.

2.4 Communication Protocol

The communication between a client and the server consists of multiple independent requests, that are sent to the server. Each request begins with an instance of Requests type, and according to its value, further steps differ. In the following section, we will discuss these steps for each possible value of Requests type.

If an object instance is sent, its serialized form is used, as described in section 2.2.

Example:

Server → Client : long ... The server sends a long variable to the client.

ITEM_EXISTS

- Client → Server : String array (a path in the server database)
- Server → Client : boolean (whether the specified path points to an existing item)

DEL_VERS

- Client → Server : String array (a path in the server database)
- Client → Server : int (version number)
- Server → Client : boolean (whether the deletion succeeded)

CREAT_FILE

- Client → Server : long (size of the file to be added)
- Client → Server : boolean (whether to check the available disc space)
- Client → Server : String array (a target path in the server database)
- If the available space is being checked:
  - Server → Client : boolean (whether the file creation succeeded)

CREAT_VERS

- Client → Server : boolean (whether to check the available disc space)
- Client → Server : long (size of the file to be added)
- Server → Client : int (block size used)
- If the available space is being checked:
  - Server → Client : boolean (whether it is possible to add the version)
    ..if negative, request handling ends.
- Client → Server : String array (a path to a file in the server database)
- Client → Server : String (strong hash of the file contents)
- A loop begins, with the client each time sending one of the following:
  - String “raw_data“ followed by a byte array containing new data
  - String “hash“ followed by:
    - long : the primary hash of an already present block
    - String : the secondary hash of the block
  - String “get_vals“ followed by:
    - Server → Client : number of all weak hash values in server database
    - Server → Client : all the weak hash values (as long variables)
  - String “check” followed by:
    - Client → Server : String (a strong hash)
    - Server → Client : boolean (the block specified by the strong hash exists)
  - String “end“ , which marks the end of communication.

GET_FILE

- Client → Server : String array (a path in the server database)
- Client → Server : int (version number)
- Server → Client : boolean (whether the specified version is present)
- If the version is present:
  - Server → Client : byte array (contents of the file)

END

- Nothing follows, server shuts down.

CREAT_DIR

- Client → Server : long (size of the directory contents)
- Client → Server : boolean (whether to check the available disc space)
- Client → Server : String array (path on the server to be created)
- If the available space is being checked:
  - Server → Client : boolean (whether the directory creation succeeded)

GET_ZIP

- Client → Server : String array (a path in the server database)
- Client → Server : int (version number)
- Server → Client : boolean (whether the specified version/directory is present)
- If the version/directory is present:
  - Server → Client : byte array (a ZIP archive)

CHECK_CHANGES

- Client → Server : String array (a path in the server database)
- Client → Server : String (a content hash)
- Server → Client : boolean (whether it is reasonable to upload a new version)

GC

- No communication follows, only garbage collection over the server data blocks is performed.

GET_D_ITEM

- Client → Server : String array (a path in the server database)
- Server → Client : byte (0 .. the desired file does not exist, 1 .. the file is a directory, 2 .. the file is a regular file)
- Server → Client : the serialized DItem object (if it exists)

GET_FS

- Server → Client : Map<String,DItem> (the “fileMap” object, serialized the same way as inside a Database object)

2.5 Client

On the client side of the applications, the following functionalities are present:

- Command line client
- Client with GUI
- Scheduler

The functional core of the client is included in the Client class, together with a simple command line interface.

When run, the Client class checks for program parameters containing the identification of the server to connect to. If it is not found, the program interactively asks the user to enter the data.

User requests are then read from the standard input in a loop, in work() method. After some pre-processing in serveOperation(..), the requests are distributed to specialized methods in switchToOperation(..).

In the following, we will discuss how the user requests are dealt with on the client side:

- Adding a new file

Adding a new file or version is utilized by the serveAdd(..) method. First, it is checked whether the file to be added exists and is not a symbolic link – if it is, the process is aborted. Symbolic links are not allowed to store in the system, because they are not a platform-independent feature. Next, a request is sent to server to determine, whether the file to which we want to add the version to, is already present in the database. If it is not, it must be created first. Then, it is determined whether the file is a regular file or a directory. Directories are handled differently – the process of adding is run recursively on each file the directory contains. If we deal with a regular file, its contents are read into memory and a strong hash is computed on it. It is compared with the contents hash of the last version on server. If it has not changed, nothing more is done. Otherwise, a request to add a new version is sent to the server with some information about the version and the window loop, as described in rsync section, starts – it is implemented in the windowLoop(..) method.

- Obtaining a file from server

The process is implemented in the serveGet(..) method. It checks whether the requested file is present on the server, whether the local target is directory when downloading a directory, and does other similar preprocessing of the request. The actual transmission is handled by methods receiveVersion(..) and receiveDirectory(..). There, in receiveVersion(..), the target file is created (if a directory has been defined as the target). The raw data is retrieved from the server and written into the target file.

- Deleting a version from server

A request to delete the specified version is sent to the server.

- Obtaining the database contents

An instance of the database filesystem root element object is retrieved from the server and printed in a structured way to the standard output.

The graphical interface client functionality relies on the Client class, described above. The GUI is implemented using standard Java libraries Swing and AWT. There are two main frames used - “connectFrame” provides options to connect to a server, whilst “browserFrame” is opened only after a connection is established. It gives the user a tree-like view of the server database filesystem, along with basic options to manipulate the files – add a file/version/directory, obtain file/version/directory plainly or as a ZIP archive.

The scheduler is operated from the command line. The following information is required to run the scheduler:

- Address of the server, including the used port
- Path to a text file which contains user requests, one per line
- Time interval between consecutive executions of the scheduled code
- Maximum number of executions of the scheduled code before the scheduler shuts down. If this parameter is set to 0, no limit is set on the number of executions.

In regular intervals, it connects to the server and executes the user commands found in the input file. It uses the Client class for communication with the server.

3. User Guide

3.1 Command-Line Client

The command-line client can be run by executing the following scripts, found in the bin directory:

- console_client.bat (on Windows)
- console_client.sh (on UNIX)

However, it is usually necessary to specify program parameters, such as the address of the server. To do this, open the aforementioned script in a text editor and edit some of the following variables:

- COMP – the address of the server
- PORT – port used by the server
- LOCALE – localization option, possible values: EN, CZ

The script is then in charge of correctly passing these values as program parameters to the Client class. After the applications starts, it starts to ask for user commands in a loop. The following user commands are supported:

- **add** <SOURCE_FILE> [TARGET_FILE]

Purpose: To add a new file, directory or a new version of an already present file to the server file database.

Notes: <SOURCE_FILE> denotes the name of the local file to be added, [TARGET_FILE] is the path on the server machine (default value is equal to the filename of the source file). If the target file is already present, a new version is added, if not, a whole new record is created.

- **get** <SOURCE_FILE> <DESTINATION> [SOURCE_VERSION_NO]

Purpose: To obtain a file, version or directory from the server.

Notes: <SOURCE_FILE> is the name of the file to get. [SOURCE_VERSION_NO] is an optional argument. If it is not specified, the current version is fetched. If its value is a negative integer n, (current-n)-th version is obtained, if the value is a non-negative integer n, n-th version is requested (counting from zero as the first version inserted). <DESTINATION> denotes a path on the local machine, to which the downloaded file(s) will be saved.

- **get_zip** <SOURCE_FILE> <DESTINATION> [SOURCE_VERSION_NO]

Purpose: To obtain a file, version or directory from the server as a ZIP archive.

Notes: usage is exactly the same as in case of the "get" command.

- **delete** <PATH> <VERSION_NO>

Purpose: To delete a version of a file.

Notes: Deletion is possible only if there are at least two versions of the file present. The reason is to make sure that at least a single version stays available. <PATH> denotes a file in the server database, <VERSION_NO> is the number of the version to be deleted.

- **list** [verbose]

Purpose: To view the contents of the server database.

Notes: If "verbose" is specified, more information is printed, including the usage of every data block.

- **exit**

Purpose: To quit the client program and terminate the connection to the server.

3.2 GUI Client

The client with graphical user interface can be run by executing the following scripts:

- gui_client.sh (on UNIX)
- gui_client.vbs (on Windows)

No program parameters have to be specified before running these scripts. The server address and port, as well as localization of the client, can be specified in the main window of the application.

After starting the application, user enters the computer URI and port number of the desired server into the designated fields. A new window appears, consisting of a tree-like directory structure of the server file database.

To download a file from server, follow these steps:

- Select the source file or directory to download.
- Click the “Get“ button.
- In the file selection dialog, select the destination file or directory. In case the source is a directory, destination can only be a directory.
- Wait until the operation completes.

To download a file or directory as a ZIP item:

- Same as in the case above, just use the “Get ZIP“ button.
- It is possible to download a directory into a file, since it is downloaded in the form of a ZIP file.

To upload a file to server, follow these steps:

- Select the destination in the server browsing window. If the destination is a regular file, you can only upload a regular file. It would become a new version of the destination file regardless of it's name.
- Click the “Add“ button.
- Select the source file or directory.
- Wait until the operation completes.

3.3 Scheduler

The scheduler can be run by executing the following scripts from the bin directory:

- scheduler.bat (on Windows)
- scheduler.sh (on UNIX)

However, before running the scripts, it is usually necessary to specify the scheduler parameters, such as the scheduled commands. To do this, open the script in a text editor and edit some of the following variables:

- COMP – the IP address of the server
- PORT – number of the port used by the server
- TIME – time interval (in seconds) between two executions of the scheduled code
- INPUT_FILE – path to a text file which contains user requests, one per line
- COUNT – the maximum number of executions of the scheduled code. Zero value means there is no limit set.

The file pointed to by the INPUT_FILE variable contains user requests, as specified in the section 3.1 – one per line.

3.4 Server

The server can be run by executing the following scripts from the bin directory:

- server.bat (on Windows)
- server.sh (on UNIX)

It is usually necessary to specify program parameters such as the used port number, before running the script. To do this, open the script in a text editor and edit some of the following variables:

- PORT – port number used for accepting connections from clients
- HOME_DIR – directory to which all of the server data will be saved
- RESERVED_SPACE – Total space (in bytes) reserved for the application. The total space the server occupies will never exceed this limit.

4. Measuring Runtime Properties

4.1 Reasons for Measuring

During designing of the application, decisions had to be made about the size of certain parameters, notably:

- In the window loop: the block size used for a file
- In the heuristic in the diff-ing algorithm: the maximal distance to which the search for the middle snake spreads

Without measuring, it is unclear what values to use, since in both cases “bigger” values as well as “smaller” values have both advantages and disadvantages. Lets see what happens when choosing different block sizes:

Rather small leads to:

- Smaller space requirements, which leads to the ability to store more data in case a constant data space is dedicated to the application.
- Possibly worse dependability, as block sharing is utilized more than in the case of bigger blocks. I.e., if a block gets damaged, more files can get corrupted.
- If there are too many blocks of data, a time penalty may show up when uploading a new version of a file. Before the window loop, a set of all used weak hash values is downloaded, which can take some time. Also, if the weak hash values set gets too large, hash collisions may become more frequent.

Conversely, rather big leads to:

- Worse chances of sharing blocks between different files, which leads to bigger space requirements.
- Because the upper limit on the size of edit script on the server is set to be equal to the block size, bigger block size often makes the script computation take longer.
- The number of blocks is smaller than in the case of small block sizes, which possibly leads to a faster addition of files to the database.

As for the heuristics parameter, rather small limit leads to faster computation, but it is much further from optimal result than if a bigger value was used. If we get an extremely suboptimal result, the edit script takes more space and it also takes longer to apply it on a file.

4.2 Testing Methods

To measure the time and space requirements of the application when different block sizes are chosen, we prepared a set of testing data. An overview of the testing data is charted in the following table:

Name	Dir / File	Size	No. of files inside	Description
similar	dir	1,59 MB	15	In the first five files, every two files differ by at most 1 kB. In the next five files, every two files differ by at most 8 kB. In the last five files every two files differ by at most 60 kB.
[DBI026] dbapl	dir	3,42 MB	12	Contains a set of files used during preparation for a CS course.
[NMAI062] Algebra I	dir	1,43 MB	156	Contains a set of files used during preparation for a CS course.
big_files	dir	19 MB	2	Contains two text files, the first is a series of 0s, the latter is a series of 1s
documents	dir	2,03 MB	12	Includes a few documents and images.
small_files	dir	5,4 kB	1000	Many small files holding just a few bytes.
WebApplication5	dir	4,43 MB	31	A typical Java web application project.
blank	file	263 kB	-	270000 of zero bytes
blank2	file	264 kB	-	271000 of zero bytes
JavaApplication1.java	file	346 B	-	346 bytes long source file
JavaApplication2.java	file	346 B	-	a slightly changed version of JavaApplication1.java
lipsum.txt	file	1,03 MB	-	Text file with random Latin words.
lipsum2.txt	file	1,32 MB	-	Text file with random Latin words.

Table 1 - Contents of the testing data set

Various types of data are included in the testing data set. The “similar” directory contains similar files, that will be used to test the performance of the edit script algorithm. Files from the “small_files” directory test the application performance if the input data are scattered across a bigger number of files. However, we believe that the most common usage of the system will be to back-up directories containing usual text document, spreadsheets, images, or programmer's source files. As a result, we have included mostly files of these types in the testing data set.

In the following by “scenario” we mean a set of user operations executed on the client side of the application. Considering the possible scenarios when using the application, we have concluded that there is a factor that distinguishes the scenarios most – the level of versioning functionality utilization. With this in mind, we have designed three typical scenarios. The first one, employes no versioning, but includes adding many different types of input data, from a few big files to a large number of very small files. The next scenario adds a little use of versioning, where around a half of the added versions will likely be stored in the script form and the other will likely be scripted. The last scenario employs the versioning functionality heavily. Many versions are added to the same file, with the difference between the consecutive versions gradually becoming bigger.

A summary of the designed scenarios:

Scenario 1 includes the following operations:

No.	Operation
1	add big_files
2	add big_files/file0 big_files/file1
3	add big_files/file1 big_files/file0
4	add small_files
5	add "/[NMAI062] Algebra I"
6	add "/[DBI026] dbapl"
7	add "/WebApplication5"
8	add documents
9	get big_files/file0 ../data/fileA
10	get_zip big_files/file0 ../data/fileB
11	get small_files ../data
12	get "[NMAI062] Algebra I" ../data
13	delete big_files/file1 0

In addition, there is an extra pair operations executed before anything else:

- add "/JavaApplication1.java" japp
- add "/JavaApplication2.java" japp

However, it is not included in the results of the measuring, because its only purpose is to warm-up the JVM.

Scenario 2 includes the following operations:

No.	Operation
1	add "/lipsum.txt" lipsum
2	add "/lipsum2.txt" lipsum
3	add "/lipsum.txt" lipsum
4	add "/lipsum2.txt" lipsum
5	add "/blank" blank
6	add "/blank2" blank
7	add "/blank" blank
8	add "/blank2" blank
9	get lipsum ../data/lipsumA 0
10	get lipsum ../data/lipsumB 1
11	get lipsum ../data/lipsumC 2
12	get blank ../data/blankA 0
13	get blank ../data/blankB 1
14	get blank ../data/blankA 2
15	delete blank 1
16	delete blank 0
17	delete lipsum 2
18	delete lipsum 0

Again, there is an extra pair operations executed before anything else:

- add "/JavaApplication1.java" japp
- add "/JavaApplication2.java" japp

Scenario 3 includes the following operations:

No.	Operation
1	add /similar/file0 similar
2	add /similar/file1 similar
3	add /similar/file2 similar
4	add /similar/file3 similar
5	add /similar/file4 similar
6	add /similar/file5 similar
7	add /similar/file6 similar
8	add /similar/file7 similar
9	add /similar/file8 similar
10	add /similar/file9 similar
11	add /similar/file10 similar
12	add /similar/file11 similar
13	add /similar/file12 similar
14	add /similar/file13 similar
15	add /similar/file14 similar

Again, there is an extra pair operations executed before anything else:

- add "/JavaApplication1.java" japp
- add "/JavaApplication2.java" japp

We have decided to run the performance tests locally. If the connection from a client to the server is led through a network, the main bottleneck, compared to the local connection, is the initial phase of the addition process. The set of weak hash values of all the present blocks is sent to the client. According to the total size of the server database, the estimated size of the set is charted in table 5.

DB Size	Appr. Set Size
10 MB	1,2 kB
100 MB	12 kB
1 GB	125 kB
10 GB	1,2 MB
100 GB	12 MB

It should be noted that the set size is directly dependent on the used block size. In table 5, the block size is chosen the same as in section 4.3, i.e. 64 kB.

To record the elapsed time per user operation, a small helper class cz.filipekt.Benchmark is used. It only calls the appropriate Client methods and records the time data.

For each scenario, the actual user commands are contained in a script file, called “benchmarkXX.bat”, with XX substituted for the number of the scenario. The script contains calls to the Benchmark class, thus measuring the elapsed time, but also manages to start up and shut down the server.

Java VM usually runs all methods in an interpreted mode at first. Later, after some statistics are collected, which is usually after a fixed number of executions, the methods get compiled. This is not a very convenient practice for benchmarking, so we have followed a special procedure. The JVM is run with a non-standard switch -XX:CompileThreshold=1, which causes all methods to get compiled after just one execution. A mock request is then handed to the server to ensure the methods are compiled and the JIT process will not skew the performance measuring tests.

We have chosen three distinct block sizes to be tested:

- 1024 B
- 8192 B
- 65536 B

To eliminate other interfering elements, the middle snake limit has been fixed at 1024 operations.

To determine the optimal middle snake limit, we have in turn fixed the block size at 65536 B and tested the following limits:

- 128 operations
- 1024 operations
- 8192 operations

As a result, we have tested 5 different configurations in total.

The machine used for testing: Intel Core2 Quad Q9300 (4x2,5GHz), 8GB DDR2 RAM

4.3 Discussion of the Testing Results

Complete results of the tests carried out can be found in the Attachments section in tables A to O. Summary of the testing results is charted in the following table.

Set No.	Block Size	Heuristics Limit	Total Size	Time Elapsed (ms)
1	1024 B	1024 ops	20,6 MB	83842
2	1024 B	1024 ops	1,72 MB	8770
3	1024 B	1024 ops	432 kB	2824
1	8192 B	1024 ops	13 MB	29438
2	8192 B	1024 ops	1,17 MB	3360
3	8192 B	1024 ops	237 kB	4656
1	65536 B	1024 ops	12,2 MB	27686
2	65536 B	1024 ops	1,41 MB	5873
3	65536 B	1024 ops	447 kB	3269
1	65536 B	128 ops	12,2 MB	27634
2	65536 B	128 ops	1,41 MB	2855
3	65536 B	128 ops	447 kB	1536
1	65536 B	8192 ops	12,2 MB	27730
2	65536 B	8192 ops	1,41 MB	33518
3	65536 B	8192 ops	448 kB	41220

Table 6 - Testing results

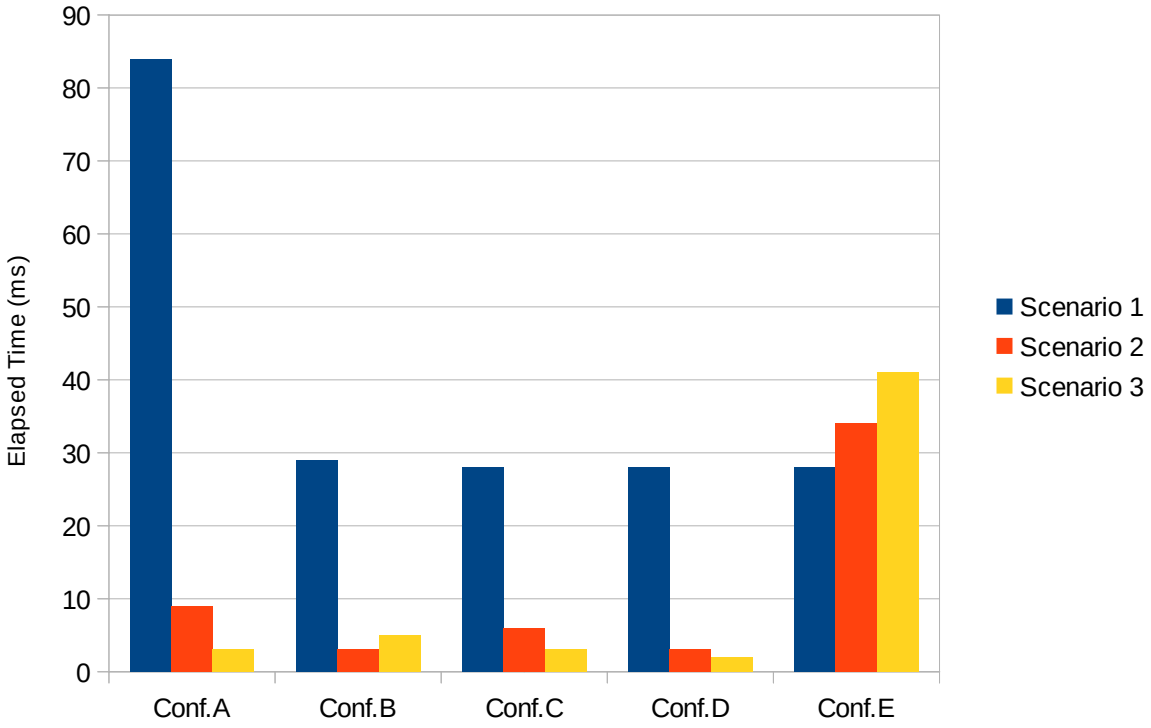
For each configuration mentioned in table 6, tests have been run with 50 iterations for each of the three scenarios. The average elapsed time is included in the last column of table 6. Total size, as charted in the fourth column of table 6, denotes the total size of the server database directory after executing the user commands the specific scenario. In scenario 3, an additional garbage collection over the unused blocks was performed before measuring the database size, since this space would nonetheless be freed the first time it is needed.

As for scenario 1, the only impractical configuration is the first one, i.e. block size 1024 B and heuristics limit 1024 operations. On the other hand, all the configurations with block size equal to 64 kB seem to be quite equivalent, considering both time and space efficiency. This is understandable, because the only part of the application to be affected by varying the heuristics limit is the versioning mechanism. Scenario 1 does not utilize versioning, with the exception of the two warm-up operation at the beginning.

In scenario 2, the situation is different, since versioning functionality is now utilized. Right away we see that the last configuration, block size 65536 B and heuristics limit 8192 operations, is very slow, compared to the others. It is theoretically balanced by the fact that the created edit scripts are closer the optimum, since the heuristics limit is quite high. But compared to the smaller heuristics value configurations, the database size is almost not bigger at all, so there is no need to chose big heuristics limit, based on this scenario. The best-performing configurations are 8192 B block size, 1024 operations limit and 64 kB block size and 128 operations limit, based on this scenario. The slight rise in time complexity with a greater block size is presumably caused by the fact that the limit on the script size is set equal to the block size, thus the script computation takes more time before it gives up (or eventually succeeds).

The third scenario tests almost exclusively the versioning functionality. Similarly to the previous scenario, the configuration with a big heuristics limit is quite slow and is not much more economic in space, either. As expected, the fastest configuration is the one with the smallest heuristics limit value, i.e. 128 operations. However, the lowest space demands has, surprisingly, the configuration with 8 kB block size and 1024 heuristics limit. This is caused by the specific structure of the files that are added. The files begin with a series of 100000 zero bytes. As a result, the space efficiency gain in using scripts for bigger differences (8kB vs 64 kB) is outweighed by the effect of storing the initial monotonic series of bytes with a much smaller (single) block. Also, when a block-form is used, the smaller block size allows for greater amount of block sharing.

A graphical view of the measured time values follows in the graph 1. The configurations, as introduced in table 6, are named in order A,B,C,D,E,F.



It is interesting that in the first scenario the space demands actually decrease with the increasing block size. We believe that it is caused by the decreased time required to work with all the blocks, since there is a considerably smaller number of blocks present when a big block size is used.

We have decided to choose the optimal configuration as 64 kB block size and 128 heuristics limit (the D variant), because it was the fastest configuration. Although it was not the most space efficient one, the space demands remain quite low.

Epilogue / Conclusion

The thesis covered the algorithms and principles behind the creation of an efficient network back-up system. An integral part of the work is the actual implementation, which is included as an attachment in the electronic form.

The first chapter introduced the rsync and diff algorithms, which are indispensable for the application to work efficiently in space and time. Since the original version of Myers' diff algorithm produces optimal results only with a non-trivial time complexity, a heuristic is also introduced in the chapter.

The second chapter described the implementation, starting with a high level overview of the system and following with a detailed account on each part of the system.

Next chapter provided the reader with a simple user guide, containing instructions on how to run the attached software.

In the final chapter we presented the testing process, including the reasoning about what to measure, the actual methodology description and the testing outcomes and discussion. We also provided the optimal values for the tested parameters, based on the testing results.

We believe there is a space for further work on making the application suitable for holding larger amounts of data. In the current state, the application works smoothly when up to 50-100 GB of data is kept, but the addition process slows down in a linear fashion with the database size.

Bibliography

[1] TRIDGELL, Andrew. Efficient algorithms for sorting and synchronization. Canberra 1999. PhD thesis. Australian national university. Thesis supervisors R.Brant, P.Mackerras, B.McKay.

[2] MYERS, Eugene W. An O(ND) difference algorithm and its variations. Tucson 1986. Department of Computer Science, University of Arizona.

[3] HUNT, Charlie and JOHN, Binu. Java Performance. 1st ed. Michigan: Addison-Wesley Professional, 2011. ISBN 978-0137142521.

[4] EVANS, Benjamin and VERBURG, Martijn. The Well-Grounded Java Developer. 1st ed. Manning: Shelter Island, 2012. ISBN 978-1617290060.

[5] KARP, Richard M. and RABIN, Michael O. Efficient randomized pattern-matching algorithms. Berkeley 1987. University of California, Berkeley.

[6] DEUTSCH, L. Peter and GAILLY, Jean-Loup. ZLIB Compressed Data Format Specification version 3.3. Request for Comments: 1950 [online]. 1996. Available at: http://tools.ietf.org/html/rfc1950 [2013-05-05].

List of Tables