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

Název práce: Efektivní síťový zálohovací systém

Autor: Tomáš Filípek

Katedra / Ústav: Katedra aplikované matematiky

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

Abstrakt: [abstract of 80-200 words in Czech, but not a copy of the assignment of the bachelor thesis]

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: [abstract of 80-200 words in English, but not a copy of the assignment of the bachelor thesis]

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 chapter intruduces ... TODO

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

$$n=2^{16}$$

$$A=\sum_{i=k}^{k+N-1} data[i] (mod\, n)$$

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

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 in using a bigger constant $n = 2^{32}$.

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:

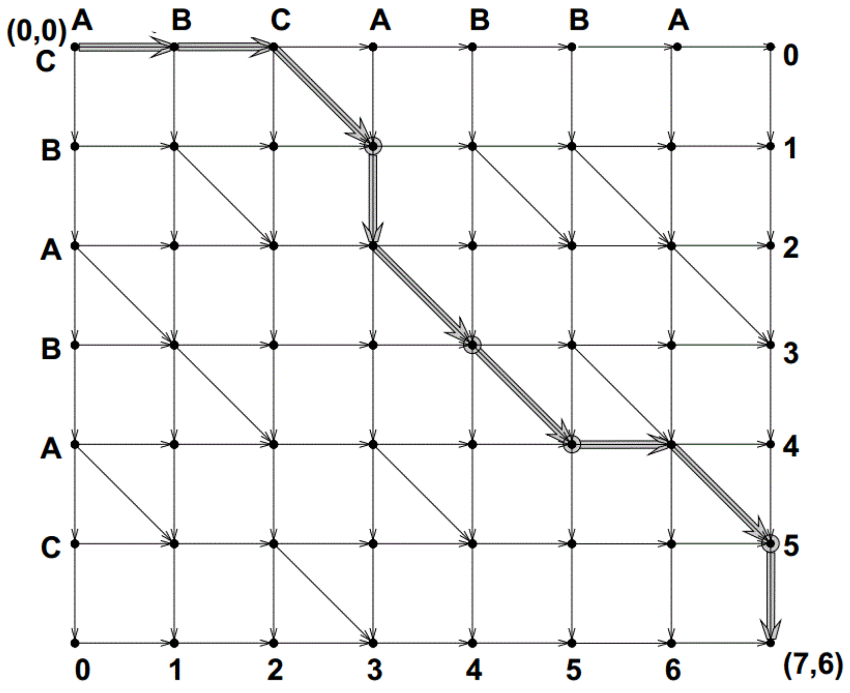
```
V := empty list of vectors
For each distance d, starting from 0, step 1:
    For each diagonal g,  from -d to d, step 2:
        find the longest reaching path LP from [0,0] , on the diagonal g
    Add a vector of the last snakes of the longest reaching paths to V.
    If the longest reaching path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



Picture 1 *(source: E.Myers [2])*

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].

The linear space refinement:

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  $(0..N)\times(0..M)$  :
    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  $(0..a)\times(0..b)$
    Y <-- shortest edit script path through  $(c..N)\times(d..M)$
    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  $(0..N)\times(0..M)$  :
    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
```

1.3 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 $(0..N)\times(0..M)$ :
      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 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)

LightDatabase:

- 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

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

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 Database contents, 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 copy of the Database object is sent to the client, to help it decide about existence and type of the file to download. Then a *get* / *get zip* request is sent to the server, followed by the path to the file in the server database and a version number. 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 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.*

GET_DB

- Server → Client : Database

GET_LIGHT_DB

- Server → Client : LightDatabase

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_light_db" followed by:
        - Server → Client : LightDatabase
    - 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)

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 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. During the window loop, each time a known block is encountered, a copy of the LightDatabase is retrieved from server, 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.

Contents of the testing data set:

- directory "similar"
  - 1,59 MB
  - 15 files
  - 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.
- directory "[DBI026] dbapl"
  - 3,42 MB
  - 12 files
  - Contains a set of files used during preparation for a CS course.
- directory "[NMAI062] Algebra I"
  - 1,43 MB
  - 156 files
  - Contains a set of files used during preparation for a CS course.
- directory "big_files"
  - 19 MB
  - 2 files
  - Contains two text files, the first is a series of 0s, the latter is a series of 1s
- directory "documents"
  - 2,03 MB
  - 12 files
  - Includes a few documents and images.
- directory "small_files"
  - 5,4 kB
  - 1000 files
  - Many small files holding just a few bytes.
- directory "WebApplication5"
  - 4,43 MB
  - 31 files
  - A typical Java web application project.
- file "blank"
  - 270000 of zero bytes
- file "blank2"

    - 271000 of zero bytes
- file “JavaApplication1.java”
    - 346 bytes long source file
- file “JavaApplication2.java”
    - a slightly changed version of JavaApplication1.java
- file “lipsum.txt”
    - 1,03 MB
    - Text file with random Latin words.
- file “lipsum2.txt”
    - 1,32 MB
    - Text file with random Latin words.

As there are many possible data sets that can be committed to the application, we decided to design three typical scenarios:

Scenario 1 includes the following operations:

- add big_files
- add big_files/file0 big_files/file1
- add big_files/file1 big_files/file0
- add small_files
- add "/[NMAI062] Algebra I"
- add "/[DBI026] dbapl"
- add "/WebApplication5"
- add documents
- get big_files/file0 ../data/fileA
- get_zip big_files/file0 ../data/fileB
- get small_files ../data
- get "[NMAI062] Algebra I" ../data
- delete big_files/file1 0

There is also an extra operation executed before anything else:

- add lipsum.txt

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

Scenario 2 includes:

- add "/lipsum.txt" lipsum
- add "/lipsum2.txt" lipsum
- add "/lipsum.txt" lipsum
- add "/lipsum2.txt" lipsum
- add "/blank" blank
- add "/blank2" blank
- add "/blank" blank
- add "/blank2" blank
- get lipsum ../data/lipsumA 0
- get lipsum ../data/lipsumB 1
- get lipsum ../data/lipsumC 2
- get blank ../data/blankA 0

- get blank ../data/blankB 1
- get blank ../data/blankA 2
- delete blank 1
- delete blank 0
- delete lipsum 2
- 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:

- add /similar/file0 similar
- add /similar/file1 similar
- add /similar/file2 similar
- add /similar/file3 similar
- add /similar/file4 similar
- add /similar/file5 similar
- add /similar/file6 similar
- add /similar/file7 similar
- add /similar/file8 similar
- add /similar/file9 similar
- add /similar/file10 similar
- add /similar/file11 similar
- add /similar/file12 similar
- add /similar/file13 similar
- add /similar/file14 similar

To warm-up the JVM, there is an extra pair of operations executed before anything else:

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

The three scenarios differ mainly in the utilization of versioning. In the first one, most additions to the database create new files, whereas in the last scenario all the additions, except for the first one, add just a new version.

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.

Because there is no such thing as a standard network speed or latency, we decided to run all the tests locally, i.e. the client and server are both on the same machine. It allows for measuring the actual performance of the application, with no unpredictable network delay.

4.3 Testing the block size

Tests over the three scenarios have been run using three different block sizes – 1024 B, 8192 B, 65536 B.

Machine used: Intel Core2 Quad Q9300 (4x2,5GHz)

Middle snake limit used (as mentioned in section 1.3): 1024 operations

Results on scenario 1:

Blocksize 1024B	**1**	**2**	**3**	**AVG**
add big_files/	13275	13322	13401	13333 (ms)
add big_files/file0 -> big_files/file1	7644	7706	7816	7722 (ms)
add big_files/file1 -> big_files/file0	7598	7645	7613	7619 (ms)
add small_files/	2512	2854	2698	2688 (ms)
add „[NMAI062] Algebra I/“	5164	5257	4945	5122 (ms)
add „[DBI026] dbapl/"	10873	11232	10327	10811 (ms)
add „WebApplication5/"	18080	18424	17566	18023 (ms)
add „documents/“	11076	11515	10483	11025 (ms)
get „big_files/file0“ ../data/fileA	1622	1638	1623	1628 (ms)
get_zip big_files/file0 ../data/fileB	7832	7878	7862	7857 (ms)
get small_files ../data	1841	2325	1857	2008 (ms)
get "[NMAI062] Algebra I" ../data	873	967	889	910 (ms)
delete big_files/file1 0	30102	30503	29742	30116 (μs)
total	**88420**	**90794**	**87110**	**88776 (ms)**
size:				**20,6 MB**

Blocksize 8192B	**1**	**2**	**3**	**AVG**
add big_files/	6786	6739	6817	6781 (ms)
add big_files/file0 -> big_files/file1	1654	1670	1669	1664 (ms)
add big_files/file1 -> big_files/file0	1654	1654	1654	1654 (ms)
add small_files/	2871	2824	2730	2808 (ms)
add „[NMAI062] Algebra I/“	1576	1514	1497	1529 (ms)
add „[DBI026] dbapl/"	2621	2386	2371	2459 (ms)
add „WebApplication5/"	3401	3182	3152	3245 (ms)
add „documents/“	1560	1529	1498	1529 (ms)
get „big_files/file0“ ../data/fileA	842	842	827	837 (ms)
get_zip big_files/file0 ../data/fileB	7083	7847	7738	7556 (ms)
get small_files ../data	1107	1092	1092	1097 (ms)
get "[NMAI062] Algebra I" ../data	514	515	515	515 (ms)
delete big_files/file1 0	12954	12843	12730	12842 (μs)
total	**31682**	**31807**	**31573**	**31687 (ms)**
size:				**13 MB**

Blocksize 65536B	**1**	**2**	**3**	**AVG**
add big_files/	6383	6365	6349	6366 (ms)
add big_files/file0 -> big_files/file1	1622	1623	1638	1628 (ms)
add big_files/file1 -> big_files/file0	1638	1622	1607	1622 (ms)
add small_files/	3853	3791	3744	3796 (ms)
add „[NMAI062] Algebra I/“	1389	1404	1419	1404 (ms)
add „[DBI026] dbapl/"	1654	1638	1622	1638 (ms)
add „WebApplication5/"	2449	2418	2449	2439 (ms)
add „documents/“	967	952	967	962 (ms)
get „big_files/file0“ ../data/fileA	765	780	749	765 (ms)
get_zip big_files/file0 ../data/fileB	7083	7410	7098	7197 (ms)
get small_files ../data	905	827	904	879 (ms)
get "[NMAI062] Algebra I" ../data	468	468	468	468 (ms)
delete big_files/file1 0	7740	7397	6119	7085 (μs)
total	**29184**	**29305**	**29020**	**29171 (ms)**
size:				**12,2 MB**

Results on scenario 2:

Blocksize 1024B	**1**	**2**	**3**	**AVG**
add lipsum.txt lipsum	2013	2045	2028	2029 (ms)
add lipsum2.txt lipsum	2528	2542	2480	2517 (ms)
add lipsum.txt lipsum	1186	1201	1185	1191 (ms)
add lipsum2.txt lipsum	1482	1435	1435	1451 (ms)
add blank blank	234	234	234	234 (ms)
add blank2 blank	343	343	327	338 (ms)
add blank blank	327	328	327	327 (ms)
add blank2 blank	343	328	327	333 (ms)
get lipsum ../data/lipsumA 0	203	203	202	203 (ms)
get lipsum ../data/lipsumB 1	219	218	234	224 (ms)
get lipsum ../data/lipsumC 2	203	187	203	198 (ms)
get blank ../data/blankA 0	109	110	94	104 (ms)
get blank ../data/blankB 1	125	125	109	120 (ms)
get blank ../data/blankA 2	110	109	93	104 (ms)
delete blank 1	8242	8343	8232	8272 (µs)
delete blank 0	340213	341102	342690	341335 (µs)
delete lipsum 2	6391	7096	6323	6603 (µs)
delete lipsum 0	119540	120299	123037	120959 (µs)
total	**9899**	**9885**	**9758**	**9850 (ms)**
size:				**1,72 MB**

Blocksize 8192B	**1**	**2**	**3**	**AVG**
add lipsum.txt lipsum	702	827	609	713 (ms)
add lipsum2.txt lipsum	1763	1794	1591	1716 (ms)
add lipsum.txt lipsum	218	219	218	218 (ms)
add lipsum2.txt lipsum	266	265	281	271 (ms)
add blank blank	156	187	156	166 (ms)
add blank2 blank	109	141	109	120 (ms)
add blank blank	78	78	78	78 (ms)
add blank2 blank	78	78	62	73 (ms)
get lipsum ../data/lipsumA 0	124	109	110	114 (ms)
get lipsum ../data/lipsumB 1	141	140	140	140 (ms)
get lipsum ../data/lipsumC 2	109	109	109	109 (ms)
get blank ../data/blankA 0	62	62	62	62 (ms)
get blank ../data/blankB 1	78	78	141	99 (ms)
get blank ../data/blankA 2	62	62	78	67 (ms)
delete blank 1	6115	6237	6186	6179 (µs)
delete blank 0	5717	5499	5181	5466 (µs)
delete lipsum 2	4912	4931	4860	4901 (µs)
delete lipsum 0	22132	22554	22339	22342 (µs)
total	**3985**	**4188**	**3783**	**3985 (ms)**
size:				**1,17 MB**

Blocksize 65536B	**1**	**2**	**3**	**AVG**
add lipsum.txt lipsum	437	468	484	463 (ms)
add lipsum2.txt lipsum	4009	4009	4165	4061 (ms)
add lipsum.txt lipsum	203	219	219	214 (ms)
add lipsum2.txt lipsum	265	265	250	260 (ms)
add blank blank	156	156	156	156 (ms)
add blank2 blank	125	109	125	120 (ms)
add blank blank	78	78	78	78 (ms)
add blank2 blank	78	78	78	78 (ms)
get lipsum ../data/lipsumA 0	94	93	93	93 (ms)
get lipsum ../data/lipsumB 1	125	109	109	114 (ms)
get lipsum ../data/lipsumC 2	94	94	94	94 (ms)
get blank ../data/blankA 0	63	63	62	63 (ms)
get blank ../data/blankB 1	78	78	62	73 (ms)
get blank ../data/blankA 2	63	63	62	63 (ms)
delete blank 1	5796	5608	5607	5670 (µs)
delete blank 0	4624	4820	5073	4839 (µs)
delete lipsum 2	4939	4702	4875	4839 (µs)
delete lipsum 0	10009	9975	9941	9975 (µs)
total	**5893**	**5907**	**6062**	**5955 (ms)**
size:				**1,41 MB**

Results on scenario 3:

Blocksize 1024B	**1**	**2**	**3**	**AVG**
add /similar/file0 similar	141	141	140	141
add /similar/file1 similar	765	780	546	697
add /similar/file2 similar	140	140	141	140
add /similar/file3 similar	156	141	141	146
add /similar/file4 similar	156	141	156	151
add /similar/file5 similar	140	125	141	135
add /similar/file6 similar	141	125	140	135
add /similar/file7 similar	125	125	125	125
add /similar/file8 similar	125	140	124	130
add /similar/file9 similar	140	125	140	135
add /similar/file10 similar	187	171	172	177
add /similar/file11 similar	172	171	172	172
add /similar/file12 similar	172	171	172	172
add /similar/file13 similar	172	171	172	172
add /similar/file14 similar	172	171	172	172
total	**2904**	**2838**	**2654**	**2800 (ms)**
size:				**432 kB**

Blocksize 8192B	**1**	**2**	**3**	**AVG**
add /similar/file0 similar	93	94	78	88
add /similar/file1 similar	234	250	250	245
add /similar/file2 similar	78	78	78	78
add /similar/file3 similar	78	78	78	78
add /similar/file4 similar	78	78	78	78
add /similar/file5 similar	187	202	202	197
add /similar/file6 similar	250	265	249	255
add /similar/file7 similar	234	249	234	239
add /similar/file8 similar	249	250	234	244
add /similar/file9 similar	187	172	171	177
add /similar/file10 similar	234	234	250	239
add /similar/file11 similar	656	655	671	661
add /similar/file12 similar	671	671	671	671
add /similar/file13 similar	686	671	671	676
add /similar/file14 similar	671	670	671	671
total	**4586**	**4617**	**4586**	**4597 (ms)**
size:				**237 kB**

Blocksize 65536B	**1**	**2**	**3**	**AVG**
add /similar/file0 similar	109	109	109	109
add /similar/file1 similar	234	234	234	234
add /similar/file2 similar	109	109	109	109
add /similar/file3 similar	94	109	109	104
add /similar/file4 similar	141	141	140	141
add /similar/file5 similar	234	249	234	239
add /similar/file6 similar	297	296	296	296
add /similar/file7 similar	297	297	281	292
add /similar/file8 similar	296	296	296	296
add /similar/file9 similar	218	219	234	224
add /similar/file10 similar	312	297	296	302
add /similar/file11 similar	219	218	218	218
add /similar/file12 similar	312	296	296	301
add /similar/file13 similar	296	296	312	301
add /similar/file14 similar	343	344	296	328
total	**3511**	**3510**	**3460**	**3494 (ms)**
size:				**447 kB**

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.

In all scenarios the 1024 B block size variant is the slowest, whilst the 8192 B and 65536 B show comparable time performance. Because in the second and third scenarios the space efficiency is a little bit better with the 8192 B variant, we decided to set it as the default block size in the application.

4.4 Testing the middle snake heuristic limit

The same three scenarios have been used in this case. The tested values for the middle snake heuristic limit were: 128 operations, 1024 ops., 8192 ops.

The block size used: 8192 B

Machine used: Intel Core2 Quad Q9300 (4x2,5GHz)

Results on scenario 1:

Middle snake limit 128 ops	**1**	**2**	**3**	**AVG**
add big_files/	6801	6771	6817	6796 (ms)
add big_files/file0 -> big_files/file1	3541	3542	3526	3536 (ms)
add big_files/file1 -> big_files/file0	3822	3635	3651	3703 (ms)
add small_files/	2730	2715	2714	2720 (ms)
add „[NMAI062] Algebra I/“	1482	1467	1482	1477 (ms)
add „[DBI026] dbapl/"	2402	2418	2371	2397 (ms)
add „WebApplication5/"	3916	3854	3885	3885 (ms)
add „documents/“	1498	1466	1482	1482 (ms)
get „big_files/file0“ ../data/fileA	858	842	842	847 (ms)
get_zip big_files/file0 ../data/fileB	7410	7269	7020	7233 (ms)
get small_files ../data	1155	1217	1139	1170 (ms)
get "[NMAI062] Algebra I" ../data	515	515	514	515 (ms)
delete big_files/file1 0	13700	13024	13096	13273 (µs)
total	**36144**	**35724**	**35456**	**35774 (ms)**
size:				**13 MB**

Middle snake limit 1024 ops	**1**	**2**	**3**	**AVG**
add big_files/	6848	6755	6755	6786 (ms)
add big_files/file0 -> big_files/file1	3947	3947	3931	3942 (ms)
add big_files/file1 -> big_files/file0	4041	4025	4041	4036 (ms)
add small_files/	2730	2777	2761	2756 (ms)
add „[NMAI062] Algebra I/“	1498	1466	1497	1487 (ms)
add „[DBI026] dbapl/"	2418	2372	2387	2392 (ms)
add „WebApplication5/"	3869	3884	3946	3900 (ms)
add „documents/“	1482	1545	1451	1493 (ms)
get „big_files/file0“ ../data/fileA	827	827	842	832 (ms)
get_zip big_files/file0 ../data/fileB	7005	7301	7223	7176 (ms)
get small_files ../data	1186	1139	1060	1128 (ms)
get "[NMAI062] Algebra I" ../data	515	515	515	515 (ms)
delete big_files/file1 0	13029	13172	12944	13048 (µs)
total	**36379**	**36566**	**36422**	**36456 (ms)**
size:				**13 MB**

Middle snake limit 8192 ops	**1**	**2**	**3**	**AVG**
add big_files/	6771	6865	6802	6813 (ms)
add big_files/file0 -> big_files/file1	4336	4321	4336	4331 (ms)
add big_files/file1 -> big_files/file0	4368	4353	4400	4374 (ms)
add small_files/	2683	2746	2777	2735 (ms)
add „[NMAI062] Algebra I/“	1482	1497	1513	1497 (ms)
add „[DBI026] dbapl/"	2355	2418	2387	2387 (ms)
add „WebApplication5/"	3869	3915	3869	3884 (ms)
add „documents/“	1482	1497	1545	1508 (ms)
get „big_files/file0“ ../data/fileA	858	842	827	842 (ms)
get_zip big_files/file0 ../data/fileB	7207	7971	7160	7446 (ms)
get small_files ../data	1046	1029	1139	1071 (ms)
get "[NMAI062] Algebra I" ../data	515	514	515	515 (ms)
delete big_files/file1 0	13056	12962	12901	12973 (µs)
total	**36985**	**37981**	**37283**	**37416 (ms)**
size:				**13 MB**

Results on scenario 2:

Middle snake limit 128 ops	**1**	**2**	**3**	**AVG**	
add lipsum.txt lipsum	593	593	624	603	(ms)
add lipsum2.txt lipsum	1030	889	998	972	(ms)
add lipsum.txt lipsum	218	219	218	218	(ms)
add lipsum2.txt lipsum	265	265	265	265	(ms)
add blank blank	156	156	140	151	(ms)
add blank2 blank	93	94	94	94	(ms)
add blank blank	78	94	78	83	(ms)
add blank2 blank	78	78	78	78	(ms)
get lipsum ../data/lipsumA 0	109	109	109	109	(ms)
get lipsum ../data/lipsumB 1	125	125	125	125	(ms)
get lipsum ../data/lipsumC 2	109	109	110	109	(ms)
get blank ../data/blankA 0	62	63	62	62	(ms)
get blank ../data/blankB 1	78	78	78	78	(ms)
get blank ../data/blankA 2	62	78	78	73	(ms)
delete blank 1	6527	6536	6516	6526	(µs)
delete blank 0	3421	3452	3957	3610	(µs)
delete lipsum 2	3323	3570	3342	3412	(µs)
delete lipsum 0	17456	18596	17897	17983	(µs)
total	**3087**	**2982**	**3089**	**3052**	**(ms)**
size:				**1,17**	**MB**

Middle snake limit 1024 ops	**1**	**2**	**3**	**AVG**	
add lipsum.txt lipsum	764	577	546	629	(ms)
add lipsum2.txt lipsum	1670	1466	1326	1487	(ms)
add lipsum.txt lipsum	218	218	218	218	(ms)
add lipsum2.txt lipsum	265	265	265	265	(ms)
add blank blank	156	141	140	146	(ms)
add blank2 blank	109	109	110	109	(ms)
add blank blank	78	78	78	78	(ms)
add blank2 blank	78	78	62	73	(ms)
get lipsum ../data/lipsumA 0	109	109	110	109	(ms)
get lipsum ../data/lipsumB 1	124	125	125	125	(ms)
get lipsum ../data/lipsumC 2	93	109	109	104	(ms)
get blank ../data/blankA 0	62	63	62	62	(ms)
get blank ../data/blankB 1	78	78	78	78	(ms)
get blank ../data/blankA 2	63	62	63	63	(ms)
delete blank 1	6484	6419	6835	6579	(µs)
delete blank 0	3360	3420	3611	3464	(µs)
delete lipsum 2	3226	3282	3272	3260	(µs)
delete lipsum 0	17695	17670	17876	17747	(µs)
total	**3867**	**3509**	**3324**	**3577**	**(ms)**
size:				**1,17**	**MB**

Middle snake limit 8192 ops	**1**	**2**	**3**	**AVG**	
add lipsum.txt lipsum	609	624	593	609	(ms)
add lipsum2.txt lipsum	1857	1731	1810	1799	(ms)
add lipsum.txt lipsum	218	203	219	213	(ms)
add lipsum2.txt lipsum	265	265	265	265	(ms)
add blank blank	141	156	140	146	(ms)
add blank2 blank	109	109	110	109	(ms)
add blank blank	78	78	78	78	(ms)
add blank2 blank	78	78	93	83	(ms)
get lipsum ../data/lipsumA 0	124	109	125	119	(ms)
get lipsum ../data/lipsumB 1	125	141	125	130	(ms)
get lipsum ../data/lipsumC 2	109	110	109	109	(ms)
get blank ../data/blankA 0	62	62	62	62	(ms)
get blank ../data/blankB 1	78	62	78	73	(ms)
get blank ../data/blankA 2	78	62	62	67	(ms)
delete blank 1	6489	6459	6604	6517	(µs)
delete blank 0	3730	3394	3416	3513	(µs)
delete lipsum 2	3353	3231	3241	3275	(µs)
delete lipsum 0	18768	18434	18135	18446	(µs)
total	**3963**	**3822**	**3900**	**3894**	**(ms)**
size:				**1,17**	**MB**

Results on scenario 3:

Middle snake limit 128 ops	**1**	**2**	**3**	**AVG**	
add /similar/file0 similar	93	94	94	94	
add /similar/file1 similar	187	187	187	187	
add /similar/file2 similar	63	78	78	73	
add /similar/file3 similar	63	62	63	63	
add /similar/file4 similar	62	63	47	57	
add /similar/file5 similar	63	78	78	73	
add /similar/file6 similar	62	78	78	73	
add /similar/file7 similar	78	78	78	78	
add /similar/file8 similar	78	62	62	67	
add /similar/file9 similar	78	62	78	73	
add /similar/file10 similar	78	62	78	73	
add /similar/file11 similar	156	141	140	146	
add /similar/file12 similar	140	141	140	140	
add /similar/file13 similar	140	156	141	146	
add /similar/file14 similar	125	141	140	135	
total	**1466**	**1483**	**1482**	**1478**	**(ms)**
size:				**237**	**kB**

Middle snake limit 1024 ops	**1**	**2**	**3**	**AVG**
add /similar/file0 similar	93	94	78	88
add /similar/file1 similar	234	250	250	245
add /similar/file2 similar	78	78	78	78
add /similar/file3 similar	78	78	78	78
add /similar/file4 similar	78	78	78	78
add /similar/file5 similar	187	202	202	197
add /similar/file6 similar	250	265	249	255
add /similar/file7 similar	234	249	234	239
add /similar/file8 similar	249	250	234	244
add /similar/file9 similar	187	172	171	177
add /similar/file10 similar	234	234	250	239
add /similar/file11 similar	656	655	671	661
add /similar/file12 similar	671	671	671	671
add /similar/file13 similar	686	671	671	676
add /similar/file14 similar	671	670	671	671
total	**4586**	**4617**	**4586**	**4597 (ms)**
size:				**237 kB**

Middle snake limit 8192 ops	**1**	**2**	**3**	**AVG**
add /similar/file0 similar	93	78	94	88
add /similar/file1 similar	265	249	250	255
add /similar/file2 similar	94	78	78	83
add /similar/file3 similar	93	94	93	93
add /similar/file4 similar	78	78	78	78
add /similar/file5 similar	453	452	452	452
add /similar/file6 similar	422	437	437	432
add /similar/file7 similar	437	452	405	431
add /similar/file8 similar	406	421	405	411
add /similar/file9 similar	406	405	405	405
add /similar/file10 similar	702	702	718	707
add /similar/file11 similar	780	765	780	775
add /similar/file12 similar	780	764	780	775
add /similar/file13 similar	780	780	795	785
add /similar/file14 similar	780	765	780	775
total	**6569**	**6520**	**6550**	**6545 (ms)**
size:				**237 kB**

Since in all scenarios the smallest limit was the fastest and, at the same time, the space requirements were very similar, we decided to use the 128 operations variant as the default value.

Epilogue / Conclusion

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].