DFD Design Document

PREFACE

Change History

Version	Date	Description
1.00	14/11/2010	Initial Version

Open questions

Nr.	Question	Assumption
1	DFD Client: what “tuning” parameters supports?	Another than *blocksize*
2	DFD Client: how will be displayed error messages?	DFD Client should return an error code that has a corresponding message (in the future displayed by a GUI)
3	DFD Server: how is authenticated the client?	A MOREDEVS network's user
4	DFD Server: On what port it listen to connections?	It's configurable through a conf file.
5	How DFD Server and MOREDEVS Storage servers communicates?	Defining a protocol and via tcp socket

Answered questions

Nr.	Answer

References

R Sync Algorithm http://samba.anu.edu.au/rsync/tech_report/tech_report.html

Qt Framework http://qt.nokia.com/

Table of Contents

1 Overview

The rsync algorithm is used when we want to update a file on one machine to be identical to a file on another machine. The machine where the source file is, we shall consider it as *rsync client*, and the destination machine we shall consider it as *rsync server*. The rsync client identifies the parts that are present also on the rsync server's version of the file, and send to the server only the parts that are different. So with this algorithm, instead of just copy all the content of the file, only the differences are sent to the server, and by updating the file with these differences, the files will become identical.

The DFD project is a set of two applications that enables the implementation of the rsync algorithm. These applications are the DFD client and the DFD server. The DFD client is a Windows command line application, and the DFD server is a server daemon installed on a Linux machine.

The main objective of these two applications is to synchronize the content of a specific file.

In the first case (and the most common one) the DFD client has a newer version of the file than the version present on the DFD server. In order to update the version present on the DFD server, the rsync algorithm is used. This is called the back-up operation.

In the case presented above, the DFD client plays the role of the rsync client (it has the source file), and the DFD server plays the role of the rsync server (it has the destination file).

The second case is when we want to update the file present on the DFD client, also known as the restore operation. In this case the roles are inverted; the DFD client plays the role of the rsync server, and the DFD server plays the role of the rsync client.

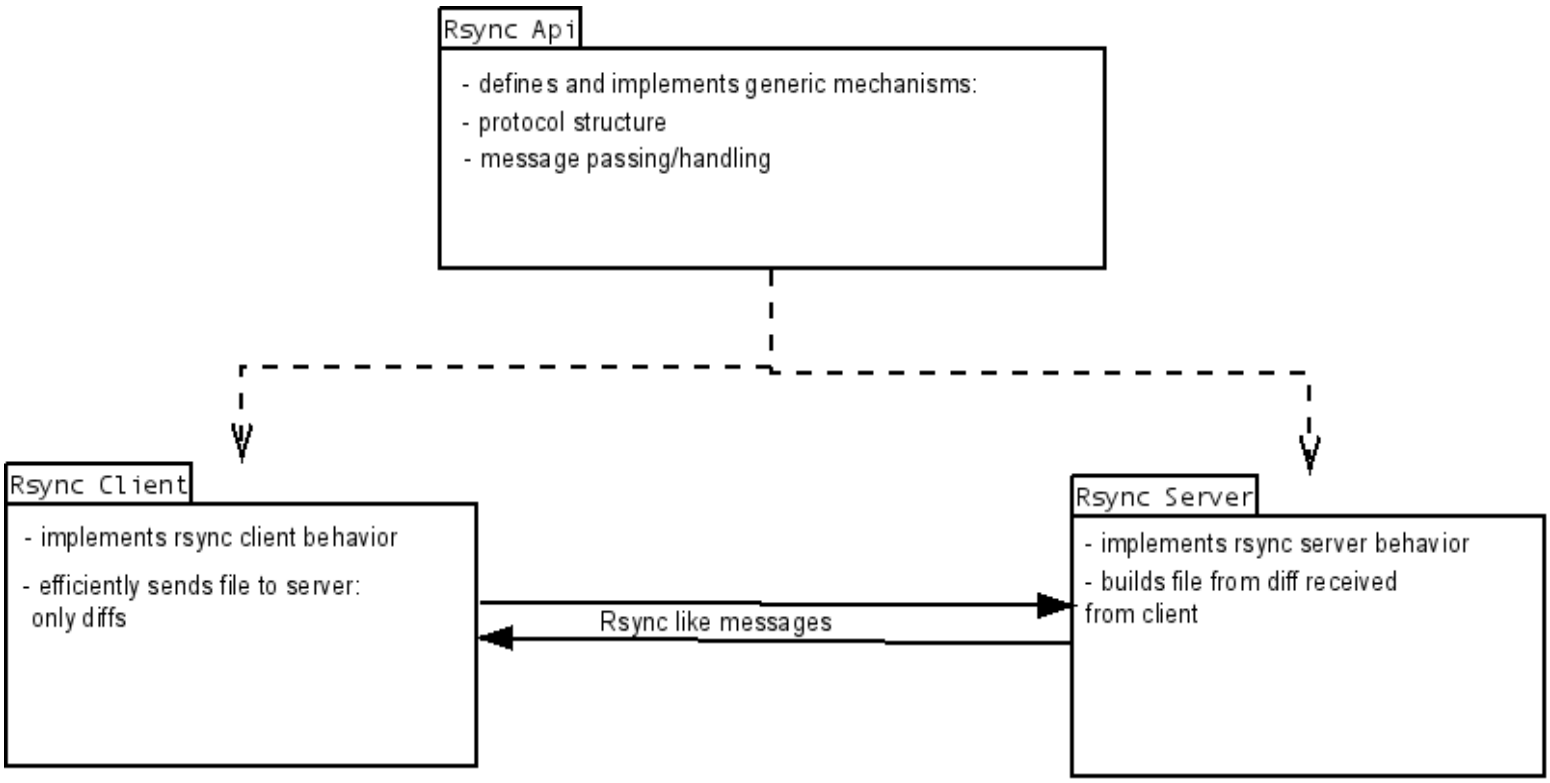
In the following chapters are described in detail the implementation of the rsync algorithm, the DFD client and the DFD server.

2 Rsync Algorithm

The rsync algorithm is used when we want to synchronize the content of two files that are present on two different machines. The rsync client and the rsync server are exchanging messages in order to identify the differences between the two files, and the rsync server is updating the file with the differences received.

The algorithm is written in C/C++ using Qt Framework.

Below is the structure diagram that implements the Rsync Algorithm.



The above diagram it will be explained in details in the following sections.

2.1Rsync Algorithm API

TBD

2.2Rsync Client

DFD client receives from the server a list of checksums (for a series of non-overlapping blocks), both weak and strong checksums.

Before going any further, let's take a look at these checksums:

The weak checksum is based on adler-32 checksum algorithm, but it is an improved algorithm since adler-32 checksum algorithm is not so efficient on small amount of data, so we will define a so-called “rolling checksum” which is used in the rsync algorithm because it has the property tat it is very

cheap to calculate the cehcksum of a buffer X2...Xn+1 given the checksum of buffer X1...Xn and the values of the bytes X1 and Xn+1. Our checksum is defined by:

$$a(k,l)=(\sum_{i=k}^{l} X_i) \bmod M$$

$$b(k,l)=(\sum_{i=k}^{l} (l-i+1)X_i) \bmod M$$

$$s(k,l)=a(k,l)+2^{16} b(k,l)$$

where $s(k,l)$ is the rolling checksum of the bytes $X_k \dots X_l$ and $M=2^{16}$

The important property of this checksum is that successive values can be computed very efficiently using the recurrence relations:

$$a(k+1,l+1)=(a(k,l)-X_k+X_{(l+1)}) \bmod M$$

$$b(k+1,l+1)=(b(k,l)-(l-k+1)X_k+a(k+1,l+1)) \bmod M$$

The strong checksum is a classic md5 checksum.

Please note that these checksums formulas are used for both client and server checksum computations.

After DFD client receives the list of weak rolling checksums and the strong checksums from the server, it opens the file and searches to find all blocks of length *blocksize* (at any offset, not just multiples of *blocksize*) that have the same weak and strong checksum as one of the blocks from the server.

The basic strategy is to compute the 32-bit rolling checksum for a block of length *blocksize* starting at each byte from client's file in turn, and for each checksum, search the list for a match. To do this our implementation will use a three level searching scheme.

The first-level uses a 16-bit hash of the 32-bit rolling checksum and a 2^{16} entry hash table. The list of checksum values (i.e., the checksums from the blocks from server) is sorted according to the 16-bit hash of the 32-bit rolling checksum. Each entry in the hash table points to the first element of the list for that hash value, or contains a null value if no element of the list has that hash value. At each offset in the file the 32-bit rolling checksum and its 16-bit hash are calculated. If the hash table entry for that hash value is not a null value, the second-level check is invoked.

The second-level check involves scanning the sorted checksum list starting with the entry pointed to by the hash table entry, looking for an entry whose 32-bit rolling checksum matches the current value. The scan terminates when it reaches an entry whose 16-bit hash differs. If this search finds a match, the third-level check is invoked.

The third-level check involves calculating the strong checksum for the current offset in the file and comparing it with the strong checksum value in the current list entry. If the two strong checksums match, we assume that we have found a block on the client which matches a block on the server. These blocks can be different but the probability is very very low, and in practice it is reasonably accepted.

When a match is found, the client sends to server the data between the current file offset and the end of the previous match, followed by the index of the block on the server that matched. This data is

sent immediately a match is found, which allows us to overlap the communication with further computation.

If no match is found at a given offset in the file, the rolling checksum is updated to the next offset and the search proceeds. If a match is found, the search is restarted at the end of the matched block. This strategy saves a considerable amount of computation for the common case where the two files are nearly identical.

2.3Rsync Server

The server (from rsync algorithm point of view) has two main tasks: first is to send to the client information about the file that it is going to back up, and the second is to reconstruct the file based on client's data and directions:

i)Send file information:

DFD server has to open the file and splits it into a series of non-overlapping fixed-sized blocks of size *blocksize*. For each block it has to perform two kind of checksums: **weak checksum** (which is based on adler-32 checksum) and a **strong checksum** (which is an md5 checksum).

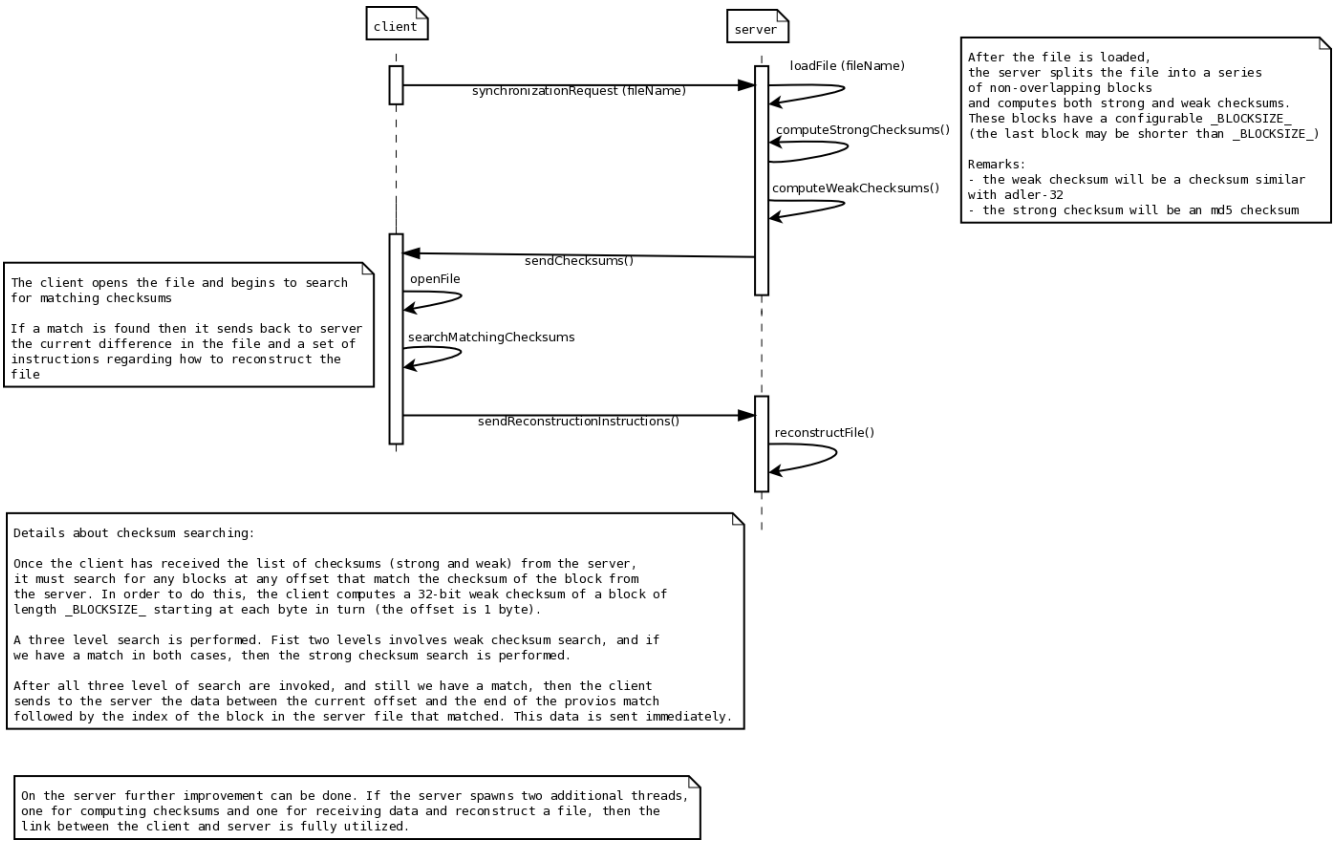
After these checksums are computed, DFD server sends these checksums to the DFD Cliet

ii)Reconstruct the file:

DFD server receives from DFD client a sequence of instructions for constructing a copy of the client's file. Each instruction is either a reference to a block of the server, or literal data. Literal data is sent only for those sections of the client's file which did not match any of the blocks on the server.

The result is that the server gets a copy of the client's file, but only the pieces of the client's file that are not found on the server (plus a small amount o data for checksums and block indexes). Based on these information, DFD server reconstructs the file.

The following diagram shows a sequence of how a file gets backed-up.



2.4Rsync Messages

The communication between the client and server is using an **rsync** based protocol. The (binary) messages which can be exchanged between the client and the server (and sender – receiver) are going to be defined in a **message set**.

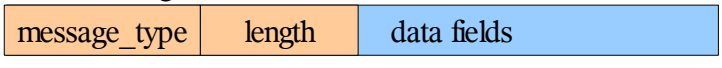
All messages will contain in their first byte the **message_type**.

Messages can be either of **fixed length** or **variable length size**. The distinction between fixed lenght/variable length can be done on the *message_type* value.

The general structure of a message looks like:

fixed length:

variable length:



In case of a *fixed length* message the only mandatory field is the *message_type (i.e. msgType)*. The rest of the data fields are optional: their presence depends on the nature of the message which is exchanged. There might be a message which does not require any additional fields (for example a client could ask the server what *protocol version* it knows: would be enough to send a request with only one

byte set: e_MsgType_GET_PROTOCOL_VERSION, this being one of the possible values for *message_type (msgType)*).

If one peer wants to send a variable length message (for example a message which contains a filename) then the second field in the message will be the **length** encoded as a *16 bit unsigned integer*. The *length* represents the total length of the message: each variable length will be preceded by a *field_length,* if needed.

Internally the messages are going to be encoded as plain **struct types**.

For example the first message which will be exchanged between the client and server is the protocol version which is supported by them:

```
struct ProtocolVersion {
  E_MsgType   message_type;
  unsigned short version;
};
```

```
enum E_MsgType{
  e_MsgType_PROTOCOL_VERSION=1,
  e_MsgType_FILE_DETAILS=2,
  e_MsgType_SUM_CHUNK=3,
  e_MsgType_SUM_FILE=4,
...
};
```

The rest of messages to be exchanged need to be defined: this is more like an organic process: as progress is made new messages will be added, existing ones might be modified ==> protocol version will change.

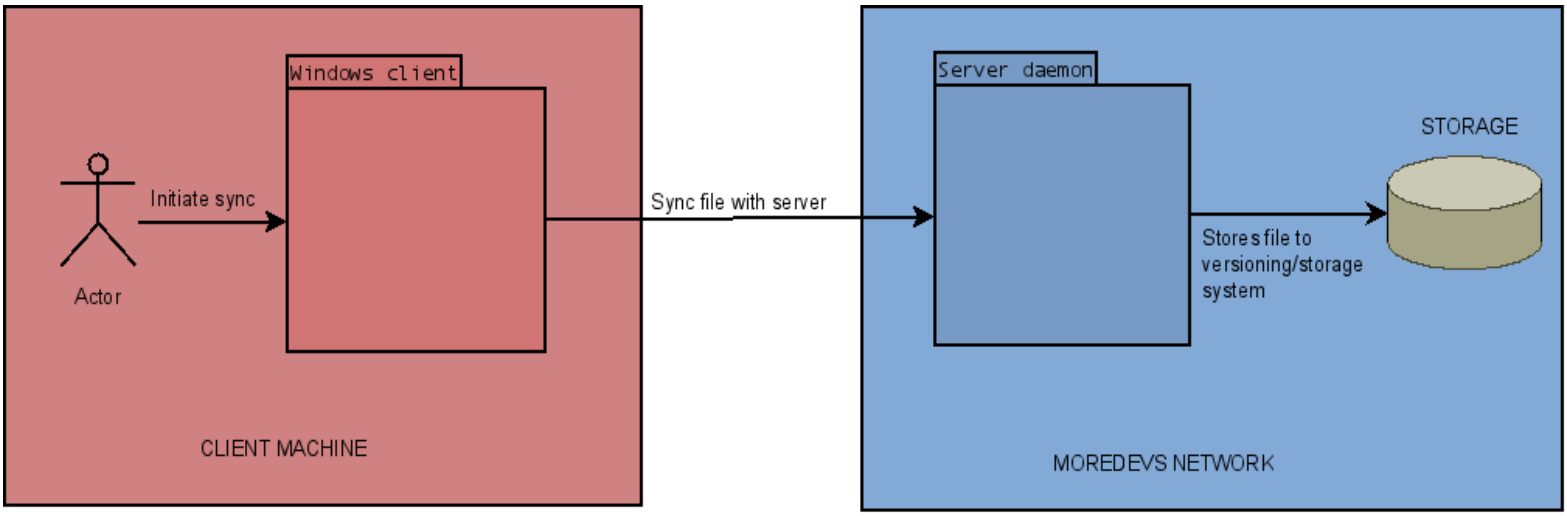
We find that at minimum we need to have messages for:

- *file details*: name of file, length of file, MD5 sum of file, [maybe details about creation time, owner, ...]
- *chunk details*: offset in file, length of chunk, simple sum, MD5 sum.
- *list of chunks:* file length, number of chunks, list of *chunk details*.
- *chunk data:* offset in file, chunk size, chunk data

3DFD Client and DFD Server

DFD Client and DFD Server are two applications written in C++ using Qt Framework.

DFD Client is a Windows command line application that connects to DFD Server in order to synchronize the content of a file. The synchronization is done using the rsync algorithm.



3.1DFD Client

DFD Client is Windows command line application. It support various parameters in order to synchronize the file with the version on the server.

./DFDClient [--backup|--restore] [--blocksize=SIZE] [--recurse] SRC user@HOST:DEST [--version=VERSION]

TBD more parameters

--backup|--restore = this parameter is optional. By default it has the value*--backup*

--blocksize=SIZE = force the rsync algorithm to be implemented using a specific check-sum block size

--recurse = to be used when an entire folder is synchronize

SRC = the file or folder on the local machine

user@HOST:DEST = the user name: the hostname of the DFD Server: the destination file or folder

DFD Client is responsible to initialize the connection with the DFD Server. It opens a TCP/IP connection with the *HOST* and authenticates the *user*.

Depending on the *--backup--restore* parameter, it initiate the rsync algorithm, with the corresponding role: when *--backup* is present, the DFD Client has the role of rsync client; when --restore is present, the DFD Client has the role of the rsync server.

If *--recurse* is prezent, then for each file in the folder, and recursive for the sub folders, the rsync algorithm is initiated.

When *--blocksize=SIZE* parameter is present, an extra message is sent to the server, to begin the rsync algorithm with the *SIZE* value for the check-sum block size.

The rsync algorithm is initiated with the *SRC* as local file, and *DEST* as the remote file. These two files has to be synchronized. If the *--recurse* parameter is used, then *SRC* and *DEST* represents a folder that has to be synchronized.

When the *--restore* parameter is present, then the *DEST* file has to be specified witch version is restored. For this is present the *--version=VERSION* parameter. By default it is the last version available.

After the rsync algorithm is completed, a corresponding error code is returned.

3.2DFD Server

DFD Server is a daemon that runs on a Linux machine. When launched, it reads a DFDServer.conf file and with the corresponding configuration it runs and listen for incoming connection.

The DFDServer.conf contains a lot of parameters, like port number, and **TBD**.

When a DFD Client is connected, it expects that the client initialize the rsync algorithm. If DFD Server has the rsync server role, after the algorithm is finished, it communicates with the MOREDEVS Storage server in order to update the new file.

If DFD Server has the rsync client role (DFD Client initiated a restore operation), then, before the rsync algorithm begins, it ask the MOREDEVS Storage server the corresponding version of the file.

TBD communication between DFD Server and MOREDEVS Storage server

4Review comments

Page	Paragraph begins ...	Comment
8	Section 3.1	*Additional parameters: Compression* (true/false), *Encryption* (true/false)