Build a Cloud Server

Here we list all the commands used (and their relative specifications) for the creation of a simple Cloud server. In order to do so we use REDIS as the middleware structure which is needed to build the cloud architecture on top.

0.1 PART A

First we download and install REDIS with:

```
>> wget http://download.redis.io/releases/redis-5.0.5.tar.gz
>> tar -xvzf redis-5.0.5.tar.gz
>> cd redis-5.0.5
>> make
>> make test
```

These commands let us download and build the libraries of REDIS on our machine and the command **make test** tests the successful installation.
Now we can access the executables to initialize both the server and the client but before we have to export the PATH variable to make them accessible from other locations:

```
>> export PATH=$DIRECTORY/redis-5.0.5.tar.gz/src:$PATH
```

Now from two different shells we simulate both the client and the server on the same machine connected at the localhost address 127.0.0.1:PORT .

0.2 PART B

Now we proceed in implementing the *cloud.sh* library to create basic communication protocols for our distributed cloud database.
We construct it by using a "Distributed Hash Table" (DHT) to assign hash spaces to each database server. We use the first letter of the SHA1 filename hash as hashkey and, since we are dealing with 8 different databases, we need to convert the hexadecimal value into a 8 base one. All of this is done using the following commands:

```
>> hash=$(sha1string /etc/passwd)
>> hashkey=${hash:0:1}
>> h8d $hashkey
```

Successively we create a **cloud_upload** and **cloud_download** function to make the file transaction procedure automatized with the hashkey extraction process.

```
# ------------------------------------------------------------------------
# cloud_upload    : upload a file to the cloud storage
# ------------------------------------------------------------------------
# args: <source file> <cloud file>
# <source file>   : source file name to upload
# <cloud file>    : file name in the cloud system
# return 0 if success
function cloud_upload() {

   # check if the source file exists
   if [ ! -f "$1" ] ; then
      echo "error: source file <$1> does not exist!";
      return 1;
   fi

   hash=`sha1string $2`;
   hashkey=${hash:0:1}
```

```
    hashvalue=‘h8d $hashkey‘;
    echo "==> Uploading $1 with hash $hash to DHT location $hashvalue"
    upload $1 $hashvalue $2
 }

# ------------------------------------------------------------------------
# cloud_download  : download a file from the cloud storage
# ------------------------------------------------------------------------
# args: <cloud file> <local file>
# <cloud  file>   : cloud file name to download
# <target file>   : target file name in the local filesystem
# return 0 if success
function cloud_download() {

    # check that the target is not yet existing
    if [ -f "$2" ]; then
        echo "error: target exists";
        return 1;
    fi

    hash=‘sha1string $1‘;
    hashkey=${hash:0:1}
    hashvalue=‘h8d $hashkey‘;
    echo "==> Downloading $1 with hash $hash from DHT location $hashvalue"
    download $hashvalue $1 $2

    # check if something was actually downloaded
    if [ ! -f $2 ]; then
        echo "error: download failed";
        return 1;
    fi
}

# ------------------------------------------------------------------------
```

0.2.1 Test Upload

Then we write a simple script to simulate the upload of several files. To do so, we generate a token 1kb text file using the command:

```
>> dd if=/dev/zero of=1k.txt bs=1k count=1
```

which creates a one text file filled with zeros of the size of 1kb. This file can than be uploaded several times with different ”target names” in order to trick the database and make it treat them as different files.

```
# ------------------------------------------------------------------------
source cloud.sh

# Create 1kb file if it doesn't exist
if [ ! -f "1k.txt" ] ; then
    dd if=/dev/zero of=1k.txt bs=1k count=1;
fi

# Upload files
for ((i=1; i<=10; i++))
    do echo "Iteration: $i";
    cloud_upload 1k.txt $(uuidgen) > /dev/null;
    done;
# ------------------------------------------------------------------------
```

The **source cloud.sh** command is very important inside the script, because it grants its corresponding subprocess to have access to the functions written inside **cloud.sh** library as executables.

0.2.2 Remove and ls

Successively we write the functions **cloud_rm** and **cloud_ls** in the form:

```
# ------------------------------------------------------------------------
# cloud_rm        : remove a file from the cloud storage
# ------------------------------------------------------------------------
# args: <cloud file>
# <cloud file>    : file name to delete
function cloud_rm() {
   hash=‘sha1string $1‘;
   hashkey=${hash:0:1}
   hashvalue=‘h8d $hashkey‘;
   echo "==> Deleting $1 with hash $hash from DHT location $hashvalue"
   delete $hashvalue $1
}

# ------------------------------------------------------------------------
# cloud_ls        : list files on the cloud storage
# args: <none>
# ------------------------------------------------------------------------
function cloud_ls() {
   tmpfile="/tmp/.cloud_ls.$RANDOM"
   for name in 1 2 3 4 5 6 7 8; do
      list $name >> $tmpfile
   done
   sort $tmpfile
   unlink $tmpfile
}
# ------------------------------------------------------------------------
```

The second function creates a temporary file and stores the output of the **list** function inside it. Then it uses the sort function to both sort the filenames line by line and print them. The last command deletes the temporary file.
Then we can easily count the number of files inside a given DB by calling:

```
>> cloud_ls | wc -l
```

Here the *wc -l* command makes it print only the newline counts.

0.3 PART C

Now we test the performances of the database protocol by uploading different amount of files and record the performance of the **cloud_ls** command.

Listing 1: **part_c.sh**

```
# ------------------------------------------------------------------------
source cloud.sh
# Delete preexisting files
redis-cli -h ‘hostmap 1‘ -p ‘portmap 1‘ -n ‘dbmap 1‘ FLUSHALL;
# Create 1kb file if it doesn’t exist
if [ ! -f "1k.txt" ] ; then
    dd if=/dev/zero of=1k.txt bs=1k count=1;
fi

# Loop over the copies
for n_files in 100 500 1000 5000 10000 100000
    do echo "Uploading $n_files files";
    # Reinitialize the database
    redis-cli -h ‘hostmap 1‘ -p ‘portmap 1‘ -n ‘dbmap 1‘ FLUSHALL > /dev/null;
    TIMEFORMAT=%R
    # Upload files
    for ((j=1; j<=$n_files; j++))
        do cloud_upload 1k.txt $(uuidgen) > /dev/null;
```

```
        done;
    # Store times
    (printf %s "$n_files; ") >> time.txt;
    (time cloud_ls | wc -l) 2>> time.txt;
    echo "Done!"
    done;
# ------------------------------------------------------------------------
```

Note that the distributed database is reset at the end of each batch size test by means of the command:

```
>> redis-cli -h ‘hostmap 1‘ -p ‘portmap 1‘ -n ‘dbmap 1‘ FLUSHALL
```

At the end, times and their matching batch sizes are written in a .txt file using the append method.
In the following, we plot the recorded times results:

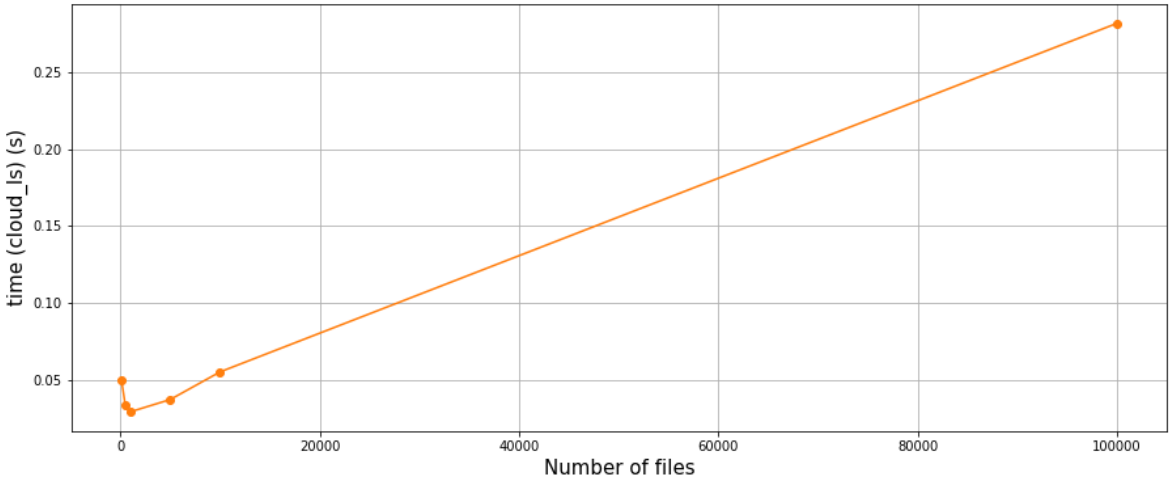

Figure 1: Timings of **cloud_ls** with respect to the number of uploaded elements

As we can see, the behaviour seems to be linear, however the data spacing is not well distributed. Therefore we proceed with a slightly different script and record the **cloud_ls** execution times but this time we perform it cumulatively measuring it each time with the same increment in the uploaded files number. This procedure was repeated ten times for each different file number increment size. Every time the ten increments were exhausted, the database is reset with *FLUSHALL* command.

Listing 2: **part_c.sh**

```
# ------------------------------------------------------------------------
source cloud.sh
# Delete preexisting files
redis-cli -h ‘hostmap 1‘ -p ‘portmap 1‘ -n ‘dbmap 1‘ FLUSHALL;
# Create 1kb file if it doesn’t exist
if [ ! -f "1k.txt" ] ; then
    dd if=/dev/zero of=1k.txt bs=1k count=1;
fi
# Loop over the copies
for n_files in 100 500 1000 5000 10000 100000
    do echo "Uploading $n_files files";
    # Reinitialize the database
    redis-cli -h ‘hostmap 1‘ -p ‘portmap 1‘ -n ‘dbmap 1‘ FLUSHALL >/dev/null;
    for ((i=1; i<=10; i++))
            do echo "Iteration: $i";
            # Upload files
            for ((j=1; j<=$n_files; j++))
                    do cloud_upload 1k.txt $(uuidgen) >/dev/null;
                    done;
            # Store times
            TIMEFORMAT=%R
            (printf %s "$n_files; ") >> time.txt;
```

```
            (time cloud_ls | wc -l) 2>> time.txt;
            done;
    done;
# ------------------------------------------------------------------------
```

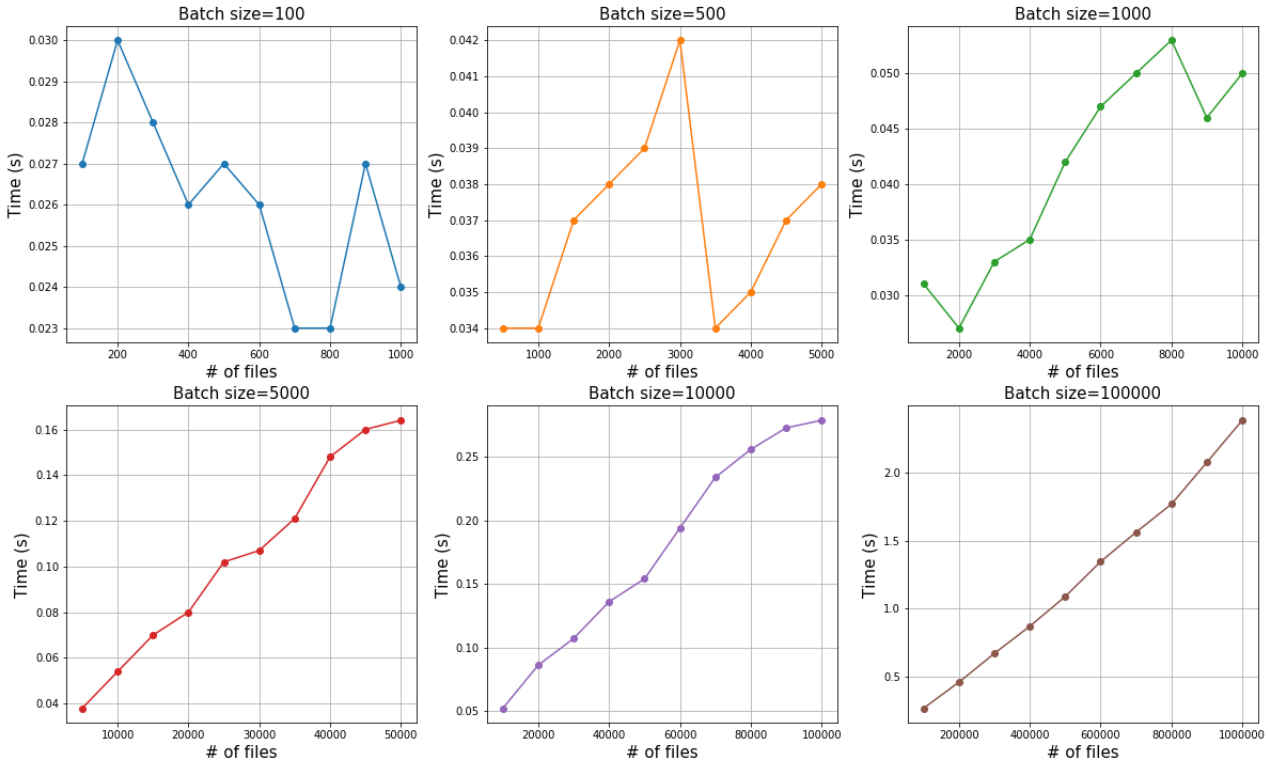

Figure 2: Timings of **cloud_ls** with respect to the number of uploaded elements. Different plot corresponds to different batch sizes tests.

As we can see the last plots seem to be more informative. We see an oscillating behaviour for the first smaller sizes and we start to see a linear increment when we reach considerable difference in number of files from one point to the other. We clearly see a linear behaviour in the last plot.

0.4 PART D

Here we test the upload function performance by uploading 1000000 files and recording their upload times:

Listing 3: **part_d.sh**

```
# ------------------------------------------------------------------------
source cloud.sh
# Deleate preexisting files
redis-cli -h ‘hostmap 1‘ -p ‘portmap 1‘ -n ‘dbmap 1‘ FLUSHALL;
# Create 1kb file if it doesn’t exist
if [ ! -f "1k.txt" ] ; then
    dd if=/dev/zero of=1k.txt bs=1k count=1;
fi

n_files=1000000
echo "Uploading $n_files files";
# Upload files
TIMEFORMAT=%R
for ((j=1; j<=$n_files; j++))
    do echo "$j";
    (printf %s "$j; ") >> time_1mill.txt;
    (time (cloud_upload 1k.txt $(uuidgen) > /dev/null)) 2>> time_1mill.txt
```

```
done;
time (cloud_ls | wc -l > /dev/null) 2> temp_final.txt
# ------------------------------------------------------------------------
```

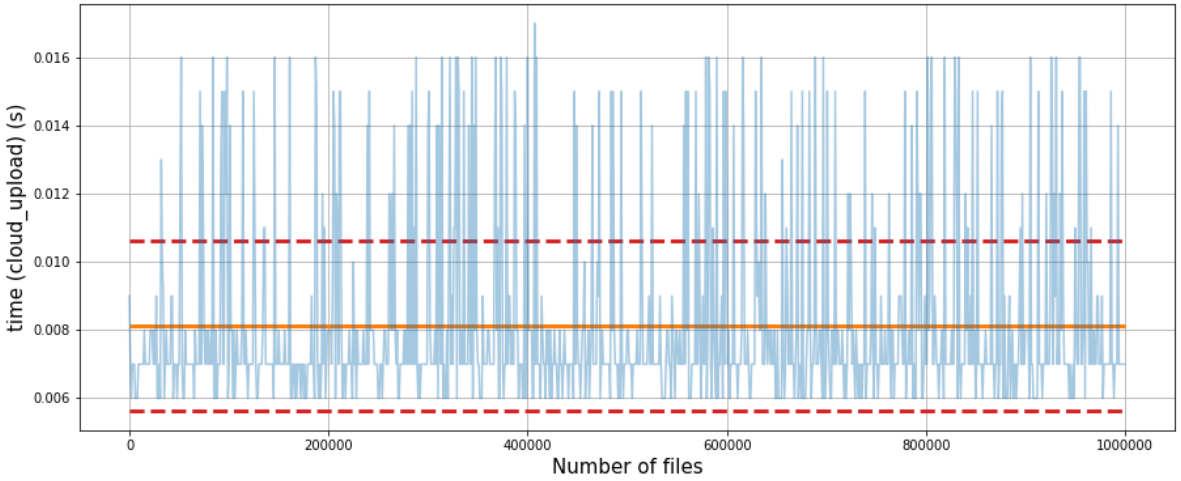

Figure 3: Timings of **cloud_upload** with respect to the number of uploaded elements.

As we can see the times are very variable and asymmetrically distributed around their mean value. Moreover, we show the record of the **cloud_ls** command over the 1000000 files distributed in the database added to the previously taken records.

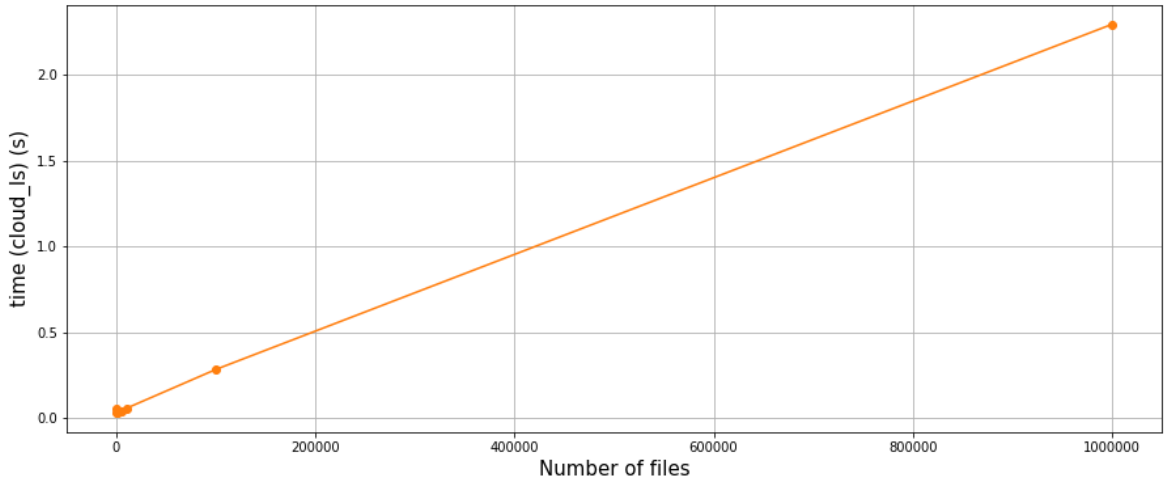

Figure 4: Timings of **cloud_ls** with respect to the number of uploaded elements.

Here we can see the linear behaviour again.

0.5 PART E

0.5.1 Buckets

In order to implement the option to use buckets we download some already written user friendly commands typing:

```
>> wget https://apeters.web.cern.ch/apeters/csc2018/_downloads/
            de5792e3bdec6634fbdbdb0b943e810d/cloudset.sh
```

Successively the **cloud.sh** library has been modified to implement the **cloudset.sh** commands if the file is present. We also modified the **cloud_upload** and **cloud_ls** functions to make use of buckets.

```
# ------------------------------------------------------------------------
```

```
# ---------------------------------BUCKETS-------------------------------
# -----------------------------------------------------------------------
# -----------------------------------------------------------------------

# Include cloudset for buckets
if [ -f "cloudset.sh" ]; then
    . cloudset.sh
    echo "cloudset loaded successfully!"

    # personal bucket initialization
    bucketname="$USER"
    buckethash=`sha1string $bucketname`
    buckethashkey=${buckethash:0:1}
    bucketindex=`h8d $buckethashkey`
    # -----------------------------------------------------------------
    # cloud_upload    : upload a file to the cloud storage
    # -----------------------------------------------------------------
    # args: <source file> <cloud file>
    # <source file>   : source file name to upload
    # <cloud file>    : file name in the cloud system
    # return 0 if success
    function cloud_upload() {

        # check if the source file exists
        if [ ! -f "$1" ] ; then
            echo "error: source file <$1> does not exist!";
            return 1;
        fi

        hash=`sha1string $2`;
        hashkey=${hash:0:1}
        hashvalue=`h8d $hashkey`;
        echo "==> Uploading $1 with hash $hash to DHT location $hashvalue"
        upload $1 $hashvalue $2
        # Associate buckethashkey
        set_add $bucketindex $bucketname $2
    }
    # -----------------------------------------------------------------

    # -----------------------------------------------------------------
    # cloud_ls        : list files on the cloud storage
    # args: <none>
    # -----------------------------------------------------------------
    function cloud_ls() {
        set_ls $bucketindex $bucketname | sort
    }
    # -----------------------------------------------------------------
fi

if [ ! -f "cloudset.sh" ]; then
    echo "ATTENTION cloudset.sh MISSING, cannot use buckets!"

    # -----------------------------------------------------------------
    # cloud_upload    : upload a file to the cloud storage
    # -----------------------------------------------------------------
    # args: <source file> <cloud file>
    # <source file>   : source file name to upload
    # <cloud file>    : file name in the cloud system
    # return 0 if success
    function cloud_upload() {
```

```
        # check if the source file exists
        if [ ! -f "$1" ] ; then
            echo "error: source file <$1> does not exist!";
            return 1;
        fi

        hash=‘sha1string $2‘;
        hashkey=${hash:0:1}
        hashvalue=‘h8d $hashkey‘;
        echo "==> Uploading $1 with hash $hash to DHT location $hashvalue"
        upload $1 $hashvalue $2
    }
    # ------------------------------------------------------------------------

    # ------------------------------------------------------------------------
    # cloud_ls        : list files on the cloud storage
    # args: <none>
    # ------------------------------------------------------------------------
    function cloud_ls() {
        tmpfile="/tmp/.cloud_ls.$RANDOM"
        for name in 1 2 3 4 5 6 7 8; do
            list $name >> $tmpfile
            done
        sort $tmpfile
        unlink $tmpfile
    }
    # ------------------------------------------------------------------------

fi
```

We see that dependently on the presence of the supplementary library different kinds of functions are defined. It simply uses the first character of the hash string of the variable $USER content as bucket identifier.

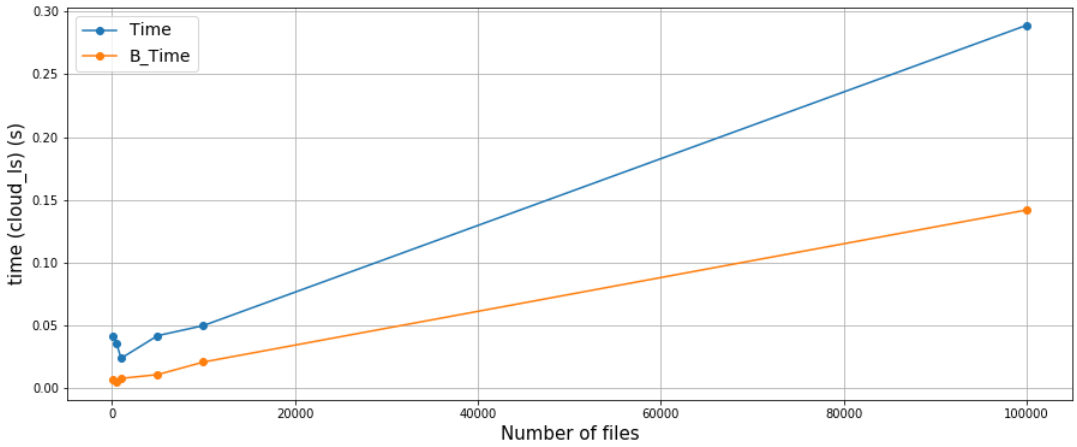

Figure 5: Timings of **cloud_ls** with respect to the number of uploaded elements

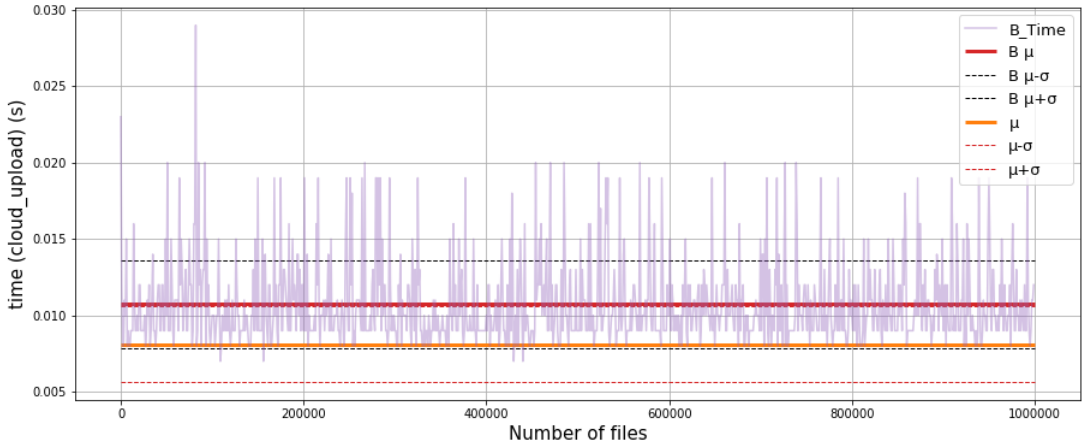

Figure 6: Timings of **cloud_upload** with respect to the number of uploaded elements

The B character in the legend of Figure(5) and Figure(6) labels the bucket version. From the results we see that with the use of buckets the **cloud_ls** improves significantly using nearly half of the time needed by the normal implementation. The **cloud_upload** function however prolongs its computation time due to the additional task to associate the bucket hashkey to the file upload.

0.5.2 Security Overhead

The addition of a security protocol for the communication with the database would introduce time delays due to the time needed to verify and authenticate the user and the files. We can simulate such supplementary option by inserting a **sleep 1** command in the bash script. This would lead to a linear increment of the execution timings which will scale by the time needed for the security authentication task.
A possible solution to this problem would be to implement security verification only at the start of the connection between the client and the server like a login prompt which will grand different access to different individuals. Moreover it would be useful to request an additional verification when a big file is requested for upload/download.

0.5.3 Data Redundancy

Adding redundancy in our database would considerably increase command execution times in the read/write process, however it makes the database more resilient to data loss or corruption. There is different options for the redundancy schemes such as RAID1, RAID5 or RAID6. Either one implements simple redundancy of files with RAID1, copying the whole disk content on a second one, or one can use more sophisticated error correction techniques, such as parity check, with RAID5.
The use of redundancy, depending on the disk configuration can affect both write and read speeds since the system has to gather files scattered on different physical discs.