Exercise 3 : asynchronous I/O and caching

Sébastien Ponce

May 22, 2015

1 Goals of the exercise

- Do network measurements
- Show synchronous I/O (in)efficiency
- Play with asynchronous I/O
- See client caching benefits

2 Setup

2.1 introduction

In this exercise, we will play with client-server interaction over wide area networks. The wide area network will actually be faked : we will connect to ourselves via the loopback device and configure it to have WAN behavior by adding some large delay.

Concerning the client-server architecture, we will mainly use 2 kinds :

- home cooked scripts as in previous exercises to demonstrate concepts
- the xrootd framework to go further. Xrootd is a powerful framework and tool suite based on the xrootd protocol and that can provide efficient and scalable data transfer solutions.

Depending on the exercises, we will either use the xrdcp command line tool that is able to read/write files from/to an xrootd server, or we will interact directly with the client API of xrootd, and more specifically with its python binding.

2.2 Basics

- open a terminal
- go to the exercise3 directory

```
cd /data/io/exercise3
```

- copy the datafile used for exercise1 into the local directory (or recreate it)

```
cp ../exercise1/datafile .
```

3 Setting up Network

3.1 Latency of real WAN

- use ping to measure the round trip time to the CERN data center

```
ping -c 5 <CERN machine>
```

- try to find out what builds up this latency by using mtr

```
sudo mtr <CERN machine>
```

  - analyze the physical path of the data
  - see the number of hops, estimate the time spent in routers/switches
  - use google maps to have a rough estimate of the cable length (take fastest road length, that is highways, the fibers follow these)
  - check whether the order of magnitude fits or whether congestions in the network are predominant

3.2 Faked network

From now on, we will not use the WAN but fake it. The main reason being that it's not realiable, neither powerful enough.

In order to fake it, we will add delay and throughput restrictions to the local loopback device so that connecting to ourselves looks like connecting to CERN through the wide area network.

- create some delay of 15ms on the loopback device, with a normal distribution centered on 15ms and having 5ms of sigma

```
sudo tc qdisc add dev lo root handle 1: netem delay 15ms 5ms \
  distribution normal
```

- check that latency is correct

```
ping -c 10 localhost
```

- measure the throughput is using iftop in a separate shell
  - in one shell

    ```
    sudo iftop -i lo -B
    ```

  - in a second shell, start a netcat server

    ```
    nc -l -p 12345 > /dev/null
    ```

  - in a third shell, send zeroes to the netcat server

    ```
    dd if=/dev/zero bs=4096 count=1048576 | nc localhost 12345
    ```

- you should get around 80MB/s which means that you have an actual network speed of 1Gbit/s and that you get limited by the loopback device

3.3 Setup Xroot

- in a separate shell, go to exercise3 directory and launch an xrootd server

```
cd /data/io/exercise3
xrootd /data/io
```

- create an empty big file to be transfered

```
dd if=/dev/zero of=bigfile bs=1024 seek=1048576 count=1
```

- note by the way how the file system optimizes such 'sparse' file

```
ls -l bigfile
du -h bigfile
du -h --apparent-size bigfile
```

- read the big file through the loop back device and check that network throughput is ok via the iftop (should still be running)

```
xrdcp -f -DICPParallelChunks 1 \
  root://localhost//data/io/exercise3/bigfile /dev/null
```

- xrdcp also gives some throughput numbers. They should match the iftop ones

4 synchronous I/O

In this part, we will exercise the synchronous remote reading of the big file with different parameters.

4.1 Manual example

You will find server.py and readFile.py files in the exercise 3 directory. They implement a very simple (and inefficient) client-server mechanism to transfer file pieces.

- Have a look at the code. It's using http protocol and very simple URL parameters to retrieve a piece of a file. See how readFile.py accesses the file, reading one buffer at a time.

- start the server part in a separate shell

```
cd /data/io/exercise3
python server.py
```

- read the file using readFile with different buffer sizes and write down the speed you get using iftop. Ctrl-C the transfers once the speed is stable, as it would takes ages to complete ! In case the reading is stuck, check the server's output. Error handling is not implemented so nicely, so that code stays simple.

```
python readFile.py localhost 10000 bigfile /dev/null
python readFile.py localhost 100000 bigfile /dev/null
python readFile.py localhost 1000000 bigfile /dev/null
python readFile.py localhost 10000000 bigfile /dev/null
```

- analyze your data : compute efficiency and try to work out the latency and network speed from what you got.
  Does it match with what we set up ?

4.2 Using xrootd

- if it is not already running, start an xrootd server in a separate shell

```
xrootd /data/io
```

- exercise synchronous read with 10K buffers

```
xrdcp -DICPParallelChunks 1 -DICPChunkSize 10000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
```

- try other buffer sizes (100K, 1M, 10M) like with the manual example. Compare results

5 asynchronous I/O

- if it is not already running, start an xrootd server in a separate shell

```
xrootd /data/io
```

- exercise asynchronous read with 10K buffers and different level of parallelism

```
xrdcp -DICPParallelChunks 4 -DICPChunkSize 10000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
xrdcp -DICPParallelChunks 16 -DICPChunkSize 10000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
xrdcp -DICPParallelChunks 64 -DICPChunkSize 10000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
xrdcp -DICPParallelChunks 255 -DICPChunkSize 10000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
```

- See how asynchronicity and parallelism allows to work around latency issues
- Try again with different buffer sizes for an average level of parallelization (16)

```
xrdcp -DICPParallelChunks 16 -DICPChunkSize 1000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
xrdcp -DICPParallelChunks 16 -DICPChunkSize 10000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
xrdcp -DICPParallelChunks 16 -DICPChunkSize 100000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
```

- do the same for 4 and 64 parallel threads

```
xrdcp -DICPParallelChunks 4 -DICPChunkSize 100000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
```

```
xrdcp -DICPParallelChunks 4 -DICPChunkSize 1000000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
xrdcp -DICPParallelChunks 64 -DICPChunkSize 10000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
xrdcp -DICPParallelChunks 64 -DICPChunkSize 100000 \
  -f root://localhost//data/io/exercise3/bigfile \
  /dev/null
```

- compute in each case the amount of "flying data" (= chunkSize * nbParallelChunks) and the transfer efficiency. See that the "flying data" size is more or less proportional to the achieved speed and that it is of the order of magnitude of the datasize needed for a similarly efficient synchronous transfer.

6 caching

Note that we go back to synchronous I/O here.

- if it is not already running, start an xrootd server in a separate shell

  ```
  xrootd /data/io
  ```

- check that datafile is in the kernel page cache. If it's not, cat it into /dev/null to make this happen

  ```
  linux-fincore datafile
  cat datafile > /dev/null
  linux-fincore datafile
  ```

- Have a look at the code of plotCaptor.py. It is basically the same as in exercise one, except for the file opening/reading that is using the xrootd API to open a remote file

- start iftop in another shell in order to monitor I/O. Also add show-totals:yes into its config file so that we can see total I/O

  ```
  echo 'show-totals : yes' > .iftoprc
  sudo iftop -i lo -B -c .iftoprc
  ```

- launch plotCaptor.py for a given captor and check timing

  ```
  python plotCaptor.py root://localhost//data/io/exercise3/datafile 40
  ```

- check how much was read from network. It's in iftop, the first column on the right of the host names

- relaunch iftop and plotCaptor.py for a given captor and its neighbour

  - in separate shell

    ```
    sudo iftop -i lo -B -c .iftoprc
    ```

  - in main shell

    ```
    python plotCaptor.py root://localhost//data/io/exercise3/datafile 40,41
    ```

- see that twice more was read from network and the effect on timing
- check the implementation of plotCaptorCached.py. Actually you could look at the diff with plotCaptor.py, as it's quite limited

  ```
  diff plotCaptor.py plotCaptorCached.py
  ```

- use plotCaptorCached.py for a given captor and its neighbour. Relaunch iftop first to have clean totals
  - in separate shell

    ```
    sudo iftop -i lo -B -c .iftoprc
    ```

  - in main shell

    ```
    python plotCaptorCached.py root://localhost//data/io/exercise3/datafile 40,
    ```

- see the improvement in timing. It does not do more than for a single captor. Understand the I/O number : what are we actually reading ?
- do it again for 10 captors
  - in separate shell

    ```
    sudo iftop -i lo -B -c .iftoprc
    ```

  - in main shell

    ```
    python plotCaptorCached.py root://localhost//data/io/exercise3/datafile 40,
    ```

- I/O is unchanged ! (100 values ar cached for each read)