XRootD Monitoring Scale Validation

Diego Davila & Derek Weitzel

Summary

After a first validation study, it was clear that the main reason for data loss was due to having UDP packets longer than the MTU being dropped. This issue has been addressed in 2 different ways:

1. Limiting the maximum size of the UDP packets being sent by the XRootD Server. This approach is not the favorite of the XRootD developers because:
    a. Is less efficient having to chop packets in little pieces
    b. They think this could cause issues when the minimal data unit in a package, e.g. the LFN, is longer than the maximum size allowed per packet. We have tested this (Section 3) with a very simplistic approach and looks like the data gets processed correctly even when the LFN is greater than the MTU
2. Making sure that jumbo frames (MTU=9000) are supported between the XRootD server(s) and the Collector. This implies a change in deployment model, and operations responsibility. The sites that operate XRootD servers are responsible for operating a collector "nearby", where nearby means on the network reachable via jumbo frames from the server. This was successfully tested on Section 2

This study describes scale tests of the XRootD monitoring when either of the 2 solutions above were employed. We expected significant data loss when numerous servers sent data to a single monitoring collector. In our tests, we did **not** observe significant data loss when scaling to 50 very high traffic servers.

Using solution (1) described above, we ran a set of scale tests in Section 2 and showed that the collector can process around 100 records per second. Also, the current processing rate on the production system is very close to the observed maximum processing rate. Since we have observed data loss in the production monitor, we expect eliminating that data loss will raise the incoming messages above the maximum processing rate.

Recommendations

We observed the maximum processing rate of the monitor which is close to the current production processing rate. We **must** improve the processing rate of the monitor or split the monitor into multiple instances. We **must** ensure there is no data loss between the servers and the monitoring collector. Proposed solutions:

1. Remove the possibility of data loss by developing a light-weight monitoring flow message bus, converting the UDP messages to TCP or use a production Message Bus. **And** increase the processing rate of the monitor by parallelizing the processing.

2. Deploy multiple monitoring collectors, assigning servers to a “nearby” collector. It is the responsibility of the site to choose the location of the collector. OSG or PATh are not responsible for these collectors. We just provide software and documentation on the deployment.

Given that the scale of the current collector seems very close to its limit and that the deployment of the New Generation collectors is very simple, the idea of having multiple collectors deployed seems like the obvious approach. In this case any site or VO operating an XRootD server should operate their own collector or make sure that jumbo frames are supported between their XRootD server and a nearby collector and that the server is configured to limit the maximum size of the UDP packets sent to the collector to a max of 9000.

Tests Performed

Section 1. Scale tests

After the first study we carried out, it was clear that the main reason for the observed data loss was related to the drop of UDP packages larger than the minimum MTU within the network, but the above did not discard the possibility of another source of data loss due to scale, i.e., many servers generating more monitoring data than what the collector is able to process.

In order to discard the possibility of data loss due to scale we modify the scripts used in the first study to be able to work with multiple servers at a time and executed 3 repetitions of the 8 tests in Table 1.

On each test a client will request ‘N’ number of random files to each of the ‘M’ servers, then wait for a second and repeat until a total amount of ‘O’ files is reached where:
N - Req. rate
M - Num. Servers
O - Total files req.

Every file request is uniquely identified and stored in a json file that will be used later to compare with the data that gets pulled from the rabbitMQ queue. The comparison is made on 3 attributes:
1. Filename
2. Server name
3. Request Id

If all 3 attributes are found to be the same in a record both in the file with the requests and the data pulled from RabbitMQ, then the file is considered correct. The column “Success %” shows the percentage of correct files found in each test averaged over 3 repetitions. As it can be seen in *Table 1* in all cases this percentage is higher than 99%.

Id	Num. Servers	Files req. per server	Total files req.	Req. rate	Files recorded rep. 1	Files recorded rep. 2	Files recorded rep. 3	Files recorded avg.	Success %
ff_10000	2	100	200	20/s	200	200	200	200,00	100,00%
ff_20000	4	100	400	20/s	400	400	400	400,00	100,00%
ff_30000	8	100	800	20/s	800	800	800	800,00	100,00%
ff_40000	32	100	3200	20/s	3200	3200	3190	3196,67	99,90%
ff_50000	50	100	5000	20/s	5000	5000	5000	5000,00	100,00%
ff_60000	50	200	10000	50/s	10000	10000	10000	10000,00	100,00%
ff_70000	50	400	20000	80/s	19977	20000	20000	19992,33	99,96%
ff_80000	50	800	40000	100/s	39981	40000	39992	39991,00	99,98%

Table 1. First set of tests

A second set of tests was executed with the objective of finding out how much the current implementation of the collector could scale under pressure.

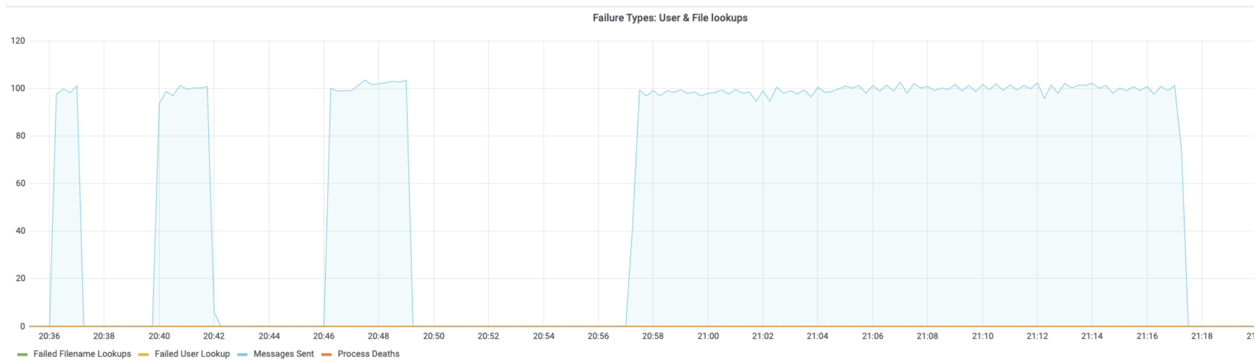
For these tests a single server was used to request a large amount of files and then we observed how long it would take to the collector to process all of these requests. In *Table 2* we can see that the collector is capable of processing around 100 messages per second. We compared the above with what is currently observed at the production collector (see *Plot 2.*) and concluded that the current scale at which the production collector is operating is very close to its limit.

Id	Total files	Transfers time	Collector time	MPS
ff_01	6000	20.7	61	98.4
ff_02	12000	42.7	120	100.0
ff_03	18000	65	180	100.0
ff_04	120000	446.2	1203	99.8

Table 2. Second set of tests

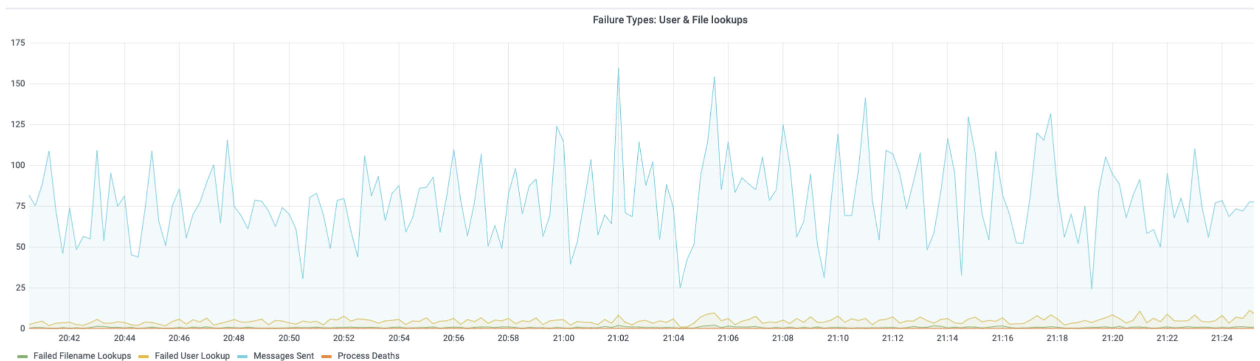
Id - the identificator of the table
Total files - the amount of files to be requested
Transfer time - the time it took for the files to be transferred to the client
Collector time - the time it took to the collector to process all the requests
MPS - Messages per second (here a message is the full: *open -> read N bytes x M -> close*)

By looking at the columns 3 and 4 of *Table 2* it can be observed that, as the number of requests (column*: Total files*) increases, the difference in time between the transfer of files being made (column: *Transfers time*) and their monitoring data being processed (column: *Collector time*) also increases. If this delay grows enough the initial state of a given request, e.g., a *file open* might be dropped by the collector before its corresponding *file close* is processed, thus loosing monitoring data.



https://gracc.opensciencegrid.org/dev/d/UEicZJWGk/xrootd-monitor?orgId=1&from=1614371745673&to=1614374565673&var-Server=validation

Plot 1. Messages processed by the collector over time, the 4 different sections correspond to the 4 tests described in *Table 2*



https://gracc.opensciencegrid.org/dev/d/UEicZJWGk/xrootd-monitor?orgId=1&from=1614372060688&to=1614374880688&var-Server=xrootd-mon.unl.edu

Plot 2. Messages processed by the production collector over time.

Section 2. Comparison between Gled and New Generation collectors.

One of the goals of this validation is to show that the New Generation collector performs at least as good as the old Gled collector, for that reason we have carried out a set of tests to show the data loss between these 2 collectors.

Accessing 90 files with a separation of 60 seconds between one file and the next one, we recorded the number of records, out of these 90, that were lost on their way to the monit DB at CERN and the results are shown in the next table

	Gled	NG at Nebraska	NG UCSD	NG UCSD w/MTU 9000 in host	NG UCSD w/MTU 9000 in both host and container
Lost records	0	9	8	9	0

Notice that for the New Generation collector deployed at UCSD we have 3 different setups:

1. **NG UCSD**. In this case the collector is deployed unmodified and the host has a default MTU of 1500
2. **NG UCSD w/MTU 9000 in host.** In this case we have set the MTU on the host to be 9000 but left the container with its default MTU, as it can be observed, this is useless.
3. **NG UCSD w/MTU 9000 in both host and container.** In this case both the host, and the container are configured with a MTU of 9000 (see section “Setting MTU in Docker“ below)

Setting MTU in Docker.

In order for the MTU increase to take effect within the Docker container the following needs to be done apart from setting the MTU in the host.

First Modify the *docker.service* file.
Either directly or by making a copy of such file and modifying the copy(recommended):

```
cp /lib/systemd/system/docker.service /etc/systemd/system/docker.service
```

And making the following line:

```
ExecStart=/usr/bin/dockerd -H fd:// --containerd ...
```

To look like:

```
ExecStart=/usr/bin/dockerd --mtu 9000 -H fd:// --containerd ...
```

Then reload the sytemctl daemon and restart docker

```
sudo systemctl daemon-reload
sudo service docker restart
```

Finally, on the docker-compose file make sure you have something like this:

```
networks:
```

```
default:
  driver: bridge
  driver_opts:
    com.docker.network.driver.mtu: 9000
```

Section 3. A LFN longer than the “fbsz” attribute

In the previous study it was found that monitoring data losses could be avoided by limiting the size of the UDP packets but a question raised:

What happens when the minimal data unit in a package, e.g., the LFN is longer than the maximum size allowed per packet?

To test the above, we created the following LFN which is made of more than 1638 characters and requested such file from a server which was configured to keep a maximum packet size of 1400, i.e., **fbsz = 1400**

The collector processed such requests normally and the record was found in the RabbitMQ later on.

The monitoring directive was:

```
xrootd.monitor all flush 30s fbsz 1400 window 5s fstat 60 lfn ops xfr 5 \
   dest fstat files info user io 192.168.2.139:9930
```

And the LFN used:

```
/store/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaa/bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbb/ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
cccccccccccccccccccccccc/ddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddd/eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee/fffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffff/ggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggggggggggggggggggggggggggggggggggggg/hhhhhhhhhhhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
```

```
hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh/iiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii/jjjjjjjjjjjjjjjjj
jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj
jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj/dummy.root
```

We used tracepath to verify the MTU between the 2 hosts (XRootD server and Collector) and it showed a MTU of 1440

```
[root@xrootd-server-59b69f645d-9t4tt xrootd]# tracepath river-c013.ssl-hep.org
 1?: [LOCALHOST]                                                  pmtu 1440
 1:
192-170-236-133.prometheus-operator-kube-proxy.kube-system.svc.cluster.local
0.045ms
 1:
192-170-236-133.prometheus-operator-kube-proxy.kube-system.svc.cluster.local
0.037ms
 2:  192-170-236-113.prometheus-operator-kubelet.kube-system.svc.cluster.local
0.337ms reached
     Resume: pmtu 1440 hops 2 back 2
```