# Dynamic I/O Congestion Control in Scalable Lustre File System*

Yingjin Qian*, Ruihai Yi
Satellite Marine Tracking & Control Department of China,
Jiangyin Jangsu, China

Yimo Du, Nong Xiao, Shiyao Jin
State Key Laboratory of High Performance Computing,
NUDT, Changsha Hunan, China

***Abstract*—This paper introduces a scalable I/O model of Lustre file system and propose a dynamic I/O congestion control mechanism to support the incoming exascale HPC systems. Under its control, clients are allowed to issue more concurrent I/O requests to servers, which optimizes the utilization of the network/server resources and improves the I/O throughput, when servers are under light load; on the other hand, it can throttle the clients' I/O and limit the number of I/O requests queued on the server to control the I/O latency and avoid congestive collapse, when the server is overloaded. The results of series of experiments demonstrate the effectiveness of our congestion control mechanism. It prevents the occurrence of congestive collapse and on this premise it can maximize the I/O throughput for the scalable Lustre file system.**

***Keywords—scalability; clustered file system; Lustre; HPC; I/O congestion control; response time; QoS; I/O intensive***

## I. INTRODUCTION

The incoming exascale computing systems pose serious scalability challenges for any data storage system. The design of traditional network file systems usually doesn't consider the congestion issue raised by systems scaling up. And file system clients remain oblivious to continually injecting I/O requests to the network and server I/O system regardless of the congestion conditions. In the HPC environments, a very large number of file system clients may impose a heavy I/O load on the shared file system. The load condition also varies considerably over time, introducing high, variable response time. All these may cause more serious congestion problems to the storage systems and sometimes even result in congestive collapse [1]. Thus, it is important to design an I/O congestion control mechanism to control and coordinate the I/O behaviors of individual clients in the storage systems, especially at large scale.

This paper first introduces a scalable I/O model of Lustre[2,3] file system, then presents a distributed dynamic I/O congestion control mechanism. On the premise of avoiding congestive collapse, it adaptively achieves two conflicting goals - maximizing the throughput and low controllable latency, according to the load conditions.

## II. LUSTRE I/O MODEL AND CONGESTION PROBLEM

As the leading clustered file system in HPC market, Lustre can scale effectively to support systems with tens of thousands of compute nodes. Our research is aiming at the scalable clustered file system Lustre. Figure 1 shows Lustre's scalable I/O model.

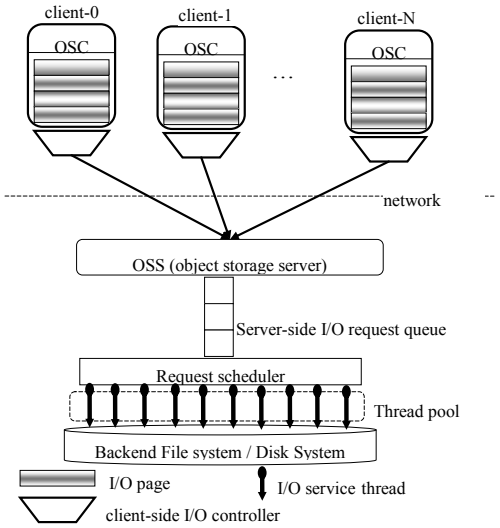


Fig. 1. Lustre I/O Model

Like most network file systems, Lustre uses an RPC model with timeouts for implementing distributed services. Fig. 1 illustrates its client-server I/O model. The design has already taken congestion into consideration. On the client side, Lustre defines tunable client-controlled maximal cached dirty data per OSC(Object Storage Client) and I/O request concurrency credits (RCC) to conduct the I/O behaviors of an individual client. (Please see Reference[4] for glossary definitions of OSC,OSS,etc) When the amount of dirty data exceeds the predefined threshold that clients are allowed, I/O becomes synchronous. Lustre prefers large bulk I/O, and the maximal size for each bulk data transfer is 1 MB due to the router limitation. For buffered I/O, the reads and writes are performed at page granularity. For each OSC, once assemble enough I/O pages for a data object, it groups them, builds an optimized I/O RPC request and sends it to OSS(Object Storage Server) over wire immediately. Due to write behind and read ahead, the optimized RPCs are mostly built with 1M bulk data. Thus, 1M I/O is the most common I/O in Lustre. Each client-side OSC has an I/O controller, functioning as the client I/O request dispatcher. It controls the fan-out concurrency of I/O requests – the number of concurrent I/O requests in flight between a client

*Corresponding author coolqyj@163.com. This research is supported by NSFC61025009 and NSFC61232003.

978-1-4799-0218-7/13/$31.00 ?2013 IEEE

and a server. Each client import is given an I/O service quota, namely, I/O RCCs. It is a tunable parameter. The I/O request is allowed to build and put into wire until available credits are consumed; otherwise, it must wait until more credits are available. The previous versions of Lustre file system use static fixed RCC strategy. For efficient use of I/O pipeline, Lustre keeps 8 I/O RPC requests in flight at most (8 RCCs), by default.

Lustre uses an out-of-band data transfer mode. Under Lustre's scalable I/O model, the utilization of network resource and server memory is well under control due to separating the bulk RDMA data transfer from the initial I/O request [5]. However, there is still congestive collapse problem, which is very similar with network congestion phenomenon Nagle discovered in 1987 [6,7].

Like most distributed systems, timeouts are used for failure detection in the RPC-based Lustre file system, which are reported on a per-call basis [8]. When a timeout occurs, certain forms of recovery action such as reconnections or retries are triggered. In large scale Lustre clusters suffering heavy I/O loads, it was observed a huge number of I/O requests queued on the server. Sometimes, it even triggers lots of the timeouts and degrads the performance sharply. This scenario often happens during checkpointing 100% of memory into a shared storage for HPC clusters. Via investigation, we found that the server got backed up processing RPC requests due to the slow speed disk systems, and a long RPC queue builts up as the load increased, resulting in considerable queuing delay. But the timeout value set by clients was not long enough to accommodate the workload changes. And RPCs have already timed out (repeatedly) and retries had been sent by the time the RPC request got to the front of the queue. The subsequent retires exacerbated the network/server loads. They would also timeout, preventing any real forward progress and creating a further backlog on the Lustre servers, resulting in serious performance degradation. Even worse, it may crash the entire system if not treated appropriately. In the Cray Jaguar system, to prevent timeouts and retries, the predefined timeout value was increased as high as 600 seconds to account for worst-case situations [9]. But this solution has drawbacks. When a server's workload becomes less busy, the large timeout value causes the failure detection mechanism to be less responsive. The client may need to wait for an excessive time period before reaching a timeout when the server fails to respond for any reason, making failure detection promptly impossible. And long timeouts also increase the recovery and failover time. These obviously hurt the performance of the entire system. Thus, a mechanism to limit the number of outstanding I/O requests on the server is needed to control the latency not exceeding the timeout value, and thereby to prevent the occurrence of congestive collapse.

## III. DYNAMIC I/O CONGESTION CONTROL

### A. I/O Model Analysis

We begin with a theoretical analysis of the I/O model shown in Fig. 1. It is obvious a multiple producers (clients)-single consumer (disk) queuing model. The consuming rate is the *IOPS* (I/O Operations Per Second) of underlying disk systems.

In HPC environments, disk performance greatly lags behind that of CPU, memory, and interconnects. The slow speed disk systems are the major obstacle to achieving high performance. For example, in the Cray Jaguar system, its scalable I/O network can provide over 889 GB/s of bisectional bandwidth using a high performance IB DDR network; while the peak bandwidth of a single storage server is only about 400 MB/s. In such environments, network related latencies can be ignored (usually less than 1 second); but RPC service time on the server is so large that lots of time is taken waiting for I/O service in the long queue. For a full pipeline, $D$ is equal to the sum of $Q$ and $N$ (where $D$ is the number of queued and serviced requests, $Q$ is the queue depth, and $N$ is the maximal number of I/O service threads on the server), and the maximal $D$ can theoretically reach RCC×C under the static RCC scheme, where $C$ is the number of I/O active clients. Thus, the I/O latencty $L$ can be approximated as

$$L=(Q+N)/IOPS=D/IOPS=RCC\times C/IOPS. \quad (1)$$

Eq. (1) obviously shows that $L$ increases linearly with $C$. Thus, the simple static RCC control scheme, using the pre-configured value, can not achieve good control effect. We propose a dynamic adaptive congestion control mechanism to regulate flow based on current congestion levels and control objective of low controllable I/O latency. The basic idea is that the server tracks congestion in real time to determine how many current I/O requests can be issued by a client according to the latency bound and then return a client a RCC value to control the its I/O behavior.

### B. Distributed I/O Congestion Control Algorithm

Lustre uses an import/export pair to manage the stateful connections and communication between a client and a server. Through the import an OSC can send requests and receive replies to/from an OSS while an OSS can receive, processes requests and send replies through the corresponding export. In order to simplify the description, the following data structures are defined. An import is defined as *import* = (*pages*, *rcc*, *urc*) where *pages* is the number of pending I/O pages managed by the OSC import; *rcc* represents the current RCC assigned by the server; *ucr* represents used credits by the import. A server is defined as a tetrad *Server* = ($D$, $D_{low}$, $L_{max}$, $C$) where $D$ and $C$ are same as the previous definitions; $D_{low}$ presents the low watermark of the number of queued and serviced requests, and it is used to determine whether the server is under light load and set to $N$ by default; $L_{max}$ is the coarse-grained latency bound. An I/O RPC is defined as a tetrad *RPC* = ($T_a$, $T_s$, *cnr*, *arc*)where $T_a$ is the time that the RPC request arrivals at the server; $T_s$ is the RPC service time; *cnr* is the client requested credits, and it is estimated according to the division of the pending I/O pages of the import and maximal pages per I/O RPC; *arc* is the returned *RCC* by the server and it is piggybacking to the client in the reply message.

The algorithm is showed in Fig. 2. When try to make I/O to a server, the client checks whether there are still credits left first. Only after acquired the credit, the client is allowed to build an optimized I/O RPC by grouping a vector of I/O pages and put it into wire through the corresponding import. Upon the completion of an I/O request, the server calculates the RCC

assigned to the client according to certain control scheme, then returns it to the client. Upon receipt of an I/O reply from the server, the client first releases the used credit, then updates its RCC as the newly feedback value from the server. If current used credits (*ucr*) are less then the updated RCC, the client will keep making I/O requests targeted to the server until no optimized I/O RPC can be built or use out all available credits.

```
Algorithm 1 RCC-based Congestion control algorithm
// now: current time.
1: Procedure ClientSendIORequest(import)
2:          if import.urc < import.rcc then
3:             rpc = BuildIORPC(import);
4:             rpc.cnr = import.pages / maxPagesPerRPC;
5:             import.urc = import.urc + 1;
6              Send the I/O RPC request to the server.
7:          end if
8: end procedure

9: Procedure ServerRecvIORequest(server, rpc)
10:          rpc.T_a = now;
11:          Enqueue the new I/O request, waiting for service;
12:          server.D = server.D + 1;
13: end procedure

14: Procedure ServerSendIOReply(server, rpc)
15:          rpc.T_s = now - rpc.T_a;
16:          server.D = server.D - 1;
17:          rpc.arc = CalcRCC(Server, rpc);
18:          Send the I/O reply message with returned RCC;
19: end procedure

20: Procedure ClientRecvIOReply(import, rpc)
21:   import.rcc = rpc.arc;
22:   import.urc = import.urc - 1;
23:   while import.urc < import.rcc && import.pages > 0 do
            ClientSendIORequest(import);
24:   end while
25: end procedure
```

Fig. 2. Congestion control algorithm

### C. RCC Assignment Algorithm

Our congestion control algorithm at a high level detects overload at the server based on the predefined $L_{max}$. The control principle is that within the latency bound, the server assigns RCC to clients large enough to maximize the throughput and stable enough to provide fair I/O service among clients. According to section *A* we get following equations:

$$L_{max} = D/IOPS = RCC \times C/IOPS,\ D_{max} = L_{max} \times IOPS,\ RCC = L_{max} \times IOPS/C. \quad (2)$$

For the persistent stable workload (*C* and *IOPS* are almost constant), the maximal deviation $\Delta L$ of controlled I/O latency can be obtained as follow

$$\frac{dL}{dRCC} = \frac{C}{IOPS}; \Rightarrow \Delta L = \frac{C}{IOPS} \times \Delta RCC \quad (3)$$

Here $\Delta RCC$ is the RCC fluctuation size. And the ratio of the deviation, denoted as *P*, is

$$P = \frac{\Delta L}{L_{\max}} = \frac{C}{IOPS \times L_{\max}} \times \Delta RCC \quad (4)$$

However, in the storage clusters with large number of nodes, the clients participate or depart I/O processes dynamically and the *IOPS* is affected by the workloads presented upon the underlying disk systems. *C* and *IOPS* are both not constant. To resolve this problem, for the number of I/O active clients *C*, each export tracks the number of its own pending I/O requests (*q*) to determine whether it is I/O active. Once the export becomes active when *q* becomes 1 for the first time, server's *C* is increased by 1. But *C* is not decreased immediately when the number drops to zero, as the corresponding client may remain I/O active when its RCC is 1. Instead, *C* is updated until the export receives the periodical ping message from the client per 25s and detects that it lasts I/O inactive status for more than a certain time length (denoted as *STL*). *IOPS* is time-averaged measured periodically and its value is made equal to the division of finished I/O operations and the efficient I/O time in certain time window. Combined with preconfigured $L_{max}$ and the bound of *IOPS*, the server can theoretically bound the maximal allowed *D*, thereby limiting the server's memory used to buffer initial I/O requests. And under these bounds, we perform a best-effect RCC assignment.

```
Algorithm 2 RCC assignment algorithm
1: Procedure CalcRCC(server, rpc)
2:          if server.D < server.D_low then
3:               rcc = rpc.cnr;
4:          else
5:               rcc = server.L_max *IOPS/server.C;
6:               L_e = server.D / IOPS;
7:               L_c = max(L_e, rpc.T_s);
8:               if L_c > server. L_max then
9:                  rcc = rcc – 1;
10:              end if
11:         end if
12:         rcc = max(min(rcc, RCC_max), RCC_min);
13:         return rcc;
14: end Procedure
```

Fig. 3. RCC assignment algorithm

The algorithm to assign *RCC* is described in Fig. 3. When *D* is lower than the preconfigured $D_{low}$ which indicates that the server is under light load, give credits with the client required value *cnr*. Otherwise, according to the preconfigured $L_{max}$ and measured *IOPS*, first calculate the maximal allowed *D*: $L_{max} \times IOPS$ and then obtain *RCC* in average via dividing by *C* to maintain fairness. When the estimated I/O latency $L_e$ or the measured RPC's service time $T_s$ exceeds $L_{max}$, the returned *RCC* is slightly decremented by 1. At last, in order to avoid the *RCC* value too big or too small, the upper and lower bounds are defined as: $RCC_{min} \le RCC \le RCC_{max}$ and $RCC_{min} \ge 1$. And the RCC for each import is initialized as $RCC_{min}$.

In the following, we give guideline on how to set the timeout value based on the preconfigured latency bound. According to the RCC assignment algorithm, we can get $\Delta RCC = 1$ for the stable workload. Thus:

$$\Delta L = \frac{C}{IOPS}, P = \frac{C}{IOPS \times L_{\max}}. \quad (5)$$

Then the timeout value $T_{timeout}$ can be set as follow.

$$T_{timeout} = \lambda \times L_{max} + L_{net} \quad (6)$$

Where $\lambda$ is the amplification factor and $\lambda \ge 1+P$; $L_{net}$ is the maximal network-related latency and its value is set based on the known capacity of the networking; $L_{max}$ is set according to

the system scale, required responsiveness and performance. They are all set by the adminstrator. Theoretically, Eq. (6) ensures that most of I/O RPCs' timeouts will not be expired.

## IV. EXPERIMENTAL EVALUATION

Based on the simulations on Lustre simulator [10], we implement our congestion control mechanism and evaluate it on the Tianhe-1 supercomputer system [11].

For easy of description, we use FIX($n$,$c$) to represent the static *RCC* control scheme that the *RCC* is fixed at $n$, and use CC($L_{max}$,$c$) to represent adaptive *RCC* control scheme that the latency bound is set to $L_{max}$. In both definitions, $c$ represents the number of the clients taking part in the experiment.

All our experiments involve synthetic workloads. We mainly focus on writes, using IOR benchmark as the workload generator to simulate the checkpointing process (the reads are similar). The default IOR setting is that keep the aggregate file size constant at 512G and I/O per client is 512G/c in average; the I/O mode is File Per Processor (FPP); the transfer size is 1M and perform fsync after write close. If not specified, the common parameters for congestion control are set as follows:

$RCC_{min}$=1，$RCC_{max}$=32，$D_{low}$=128，$STL$=60s.

First we evaluate the impact of variety of RCC on the performance by two series of experiments.

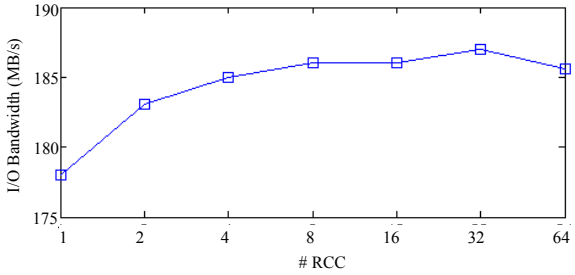


Fig. 4. Evaluation of the static RCC scheme with different $n$ values (c=1)

The first experiment is made by individual client writing data with static RCC varying from 1 to 64, as shown in Fig. 4. The results illustrate that the I/O bandwidth increases slightly from 178MB/s to 187MB/s with the increasing RCC. The reason is that the number of working I/O service threads follows with the increasing of the value of static RCC. And this increases the I/O concurrency between a client and a server, thus improve the performance.

Fig. 5 shows the second experiment results. When RCC is set to 1, the I/O bandwidth of FIX(1,1024) and FIX(1,512) drops to as low as 105MB/s and 103 MB/s respectively; While the RCC increases to 32, the bandwidth improves to 192MB/s and 177MB/s, respectively. The results clearly illustrate that the RCC has great impact on the performance. The higher RCC is, the better it ensures the stream sequentiality. The low RCC destroys the sequentiality of client's I/O, resulting in excessive disk seeks. It is the main factor leading to performance degradation. From Fig. 5(a) and 5(c), we can also observe that the increasing queue depth on the server has no negative impact on performance under Lustre's scalable I/O model. However, with 32 RCC 1024 clients, the maximal queue depth increases to more than 32,000 and the average I/O latency increases to considerable 145s. Thus, the considerable high latency introducing by high RCC must be controlled.

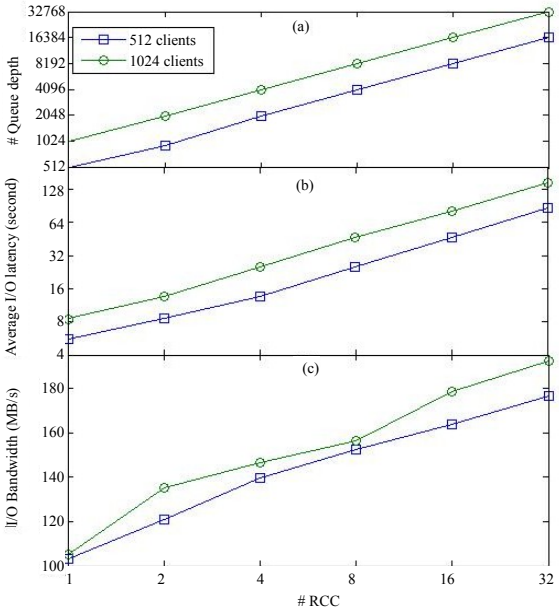


Fig. 5. Evaluation of the static RCC scheme with different $n$ values at large scales (c=512,1024).

Then, we evaluate the congestion control mechanism using adaptive RCC scheme.

First, we measure the performance for individual client comparatively with two test cases FIX(8,1) and CC(25,1). From the RPC trace, we observed that the client's RCC of CC(25,1) reaches to 32 immediately after the first I/O RPC. The performance data is nearly same as the result with 32 RCC in Fig. 4 and has slight improvement compared FIX(8, 1). The results show that the efficiency of the storage server generally increases as the concurrency on the server is increased and the I/O congestion control mechanism can optimize performance under light load.

Another set of experiments were carried out to evaluate the congestion control mechanism at large scale.

Fig. 6 shows the trace of CC(60,1024), compared with the one of FIX(8,1024). It plots the dynamic variation over time of various measured items including the queue depth on the server, I/O latency, the measured IOPS, the number of I/O active clients $C$ and the assigned RCC by the server, etc, as shown in Fig. 4 (a), (b), (c), (d), (e), respectively. Table 1 presents the collate information of 2 test cases, including the start and end time, mean/standard deviation/maximal values of I/O latencies in the stable phase (where $C$ is almostly constant.) and the overall bandwidth. The results illustrate that the maximal deviation of controlled latency follows Eq. (5), and the I/O latency is well under control. Compared with FIX(8,1024), the performance improves 9%. And compared with FIX(4,1024) and FIX(1,1024) as shown in Fig 5, the performance improves more than 15% and 62%, respectively.

TABLE I. I/O LATENCY STATISTICS IN THE STABLE PHASE FOR VARIOUS TEST CASES AND OVERALL BANDWIDTH

| Test Case | Stable Phase | | | | | | | | Overall Bandwidth (MB/s) |
|---|---|---|---|---|---|---|---|---|---|
| | Start | End | I/O Latency | | | | | | |
| | | | Mean. | Std. | Max. | $L_{max}$ | $\Delta L$ | $P$ | |
| FIX(8,1024) | 522 s | 1893 s | 55.8 s | 1.90 s | 59 s | N/A | N/A | N/A | 156.13 |
| CC(60,1024) | 409 s | 1659 s | 58.3 s | 3.25 s | 66 s | 60 s | 6 s | 10% | 170.13 |

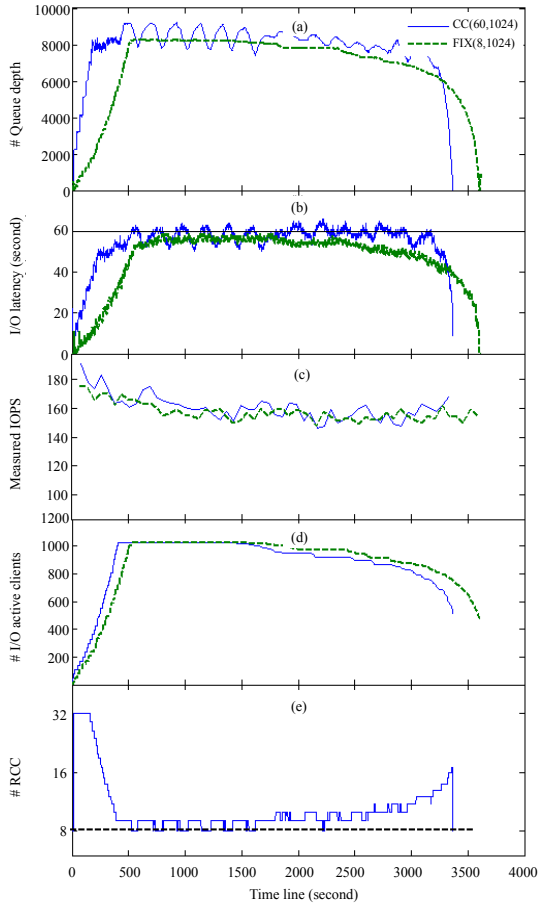


Fig. 6. CC(60,1024) trace

In the end, we conduct a set of experiments to evaluate our mechanism involving multiple OSSs. The procedure is as follow: A shared file is striped over 32 OSSs; $L_{max}$ is set to 60s; total client number is 1024; use IOR shared access mode and perform fsync after write close; I/O per client is 512M, to each OST is 16M; two cases were carried out: CC(60,1024) and FIX(8,1024); all the other setting are same as previous. We observed that the I/O behavior of each storage server was similar to previous tests. And the measured overall I/O performance of CC(60,1024) is 5.66 GB/s. The performance improves about 10% in contrast to 5.16 GB/s of FIX(8, 1024).

To avoid unnecessary timeouts which may induce congestive collapse under heavy load in experiments, for static RCC scheme the timeout value is set to 300s; while for the congestion control mechanism with adaptive RCC scheme λ is set to 1.5 and $L_{net}$ is set to 5s. They both don't trigger any timeouts. But according to Eq. (6), we know the timeout value of the latter is much less than the former. This proves indirectly that the mechanism can improve the system responsiveness. All these experimental results demonstrate that our algorithm can maximize the throughput on the premise of avoiding congestive collapse and verify the effectiveness of the congestion control mechanism.

## V. CONCLUSIONS

This paper proposes a RCC-based I/O congestion control mechanism for the scalable Lustre file system. Experimental evaluations indicate that under light load, the server can optimize performance by maximizing the concurrency level between a client and a server; under heavy load, it avoids congestive collapse by controlling the RCC assignment based on the latency bound to throttle clients' I/O, in the meanwhile it maximizes the throughput by minimizing interference among parallel I/O streams. Further challenges lie ahead as data intensive supercomputer systems scale from petaflops to exaflops over the coming decade or two. However, the larger the scale is, the more prone to induce I/O congestion problem. This paper provides file system designers with insight into how I/O congestion of network file system is addressed and how scalability is achieved in the Lustre file system.

## REFERENCES


[1] S. Floyd, K. Fall, "Promoting the use of end-to-end congestion control in the Internet," IEEE/ACM Transactions on Networking 7(4)(1999) 458-472.

[2] Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System White Paper, http://www.sun.com/offers/details/LustreFileSystem.html.

[3] Peta-Scale I/O with the Lustre File System White Paper, http://www.sun.com/offers/details/Peta-Scale_wp.html.

[4] Lustre Manul. https://wiki.hpdd.intel.com/display/PUB/Documentation.

[5] Yingjin Qian, Eric Barton, Tom Wang, et al, "A Novel network request scheduler for a large scale storage system," J. Computer Science – Research and Development 23(2009) 143-148.

[6] J. Nagle, "On Packet Switches with Infinite Storage," IEEE Transactions on Communication 35(4)(1987) 435-438.

[7] Andrew S. Tanenbaum, Computer Networks, Fourth Edition, Prentice Hall, 2003, Section 5.3.

[8] Kenneth P. Birman, Bradford B Glade, Consistent Failure Reporting in Reliable Communication Systems, Tech. Rep. TR93-1349, May 1993.

[9] Nicholas Henke, "Stabilizing Lustre at Scale on the Cray XT," in: CUG 2008 proceedings, http://cug.org/5-publications/proceedings_attendee_lists/2008CD/S08_Proceedings/pages/Authors/16-19Thursday/Henke-Thursday16B/Henke-Thursday16B-paper.pdf.

[10] Lustre simulator, https://bugzilla.lustre.org/show_bug.cgi?id=%2013634.

[11] TOP 500 Supercomputers, http://www.top500.org.