Adaptive and Scalable RPC Timeout Mechanism

for Lustre File System

Yingjin Qian

Eric Barton

**ABSTRACT**

Timeouts are usually used for failure detection in RPC systems and are typically reported on a per-call basis. During pressure testing on a very large Lustre cluster, many timeouts are observed when a server is heavily loaded imposed by thousands of clients. Our experiments and investigation show that the clients’ preset fixed timeout value is not long enough to accommodate the performance changes with large scale RPC workloads, resulting in a lot of unnecessary timeouts. To solve the problem of the fixed timeout mechanism emerging in large scale HPC cluster systems running Lustre file system, this paper proposes an adaptive and scalable RPC timeout mechanism that considers network conditions, server loads, scalability, and performance. A series of simulation experiments to demonstrate that our adaptive and scalable RPC timeout mechanism is a more suitable failure detection mechanism for RPC models with timeouts, and it enhances the system responsiveness, reliability and stability without significant negative impact on performance even for very large scale HPC cluster systems.

**Keywords**

Lustre, RPC, failure detection, timeout, adaptivity, scalability, responsiveness, reliability.

# INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Conference’10*, Month 1–2, 2010, City, State, Country.

Copyright 2010 ACM 1-58113-000-0/00/0010 …$15.00.

There has been a clear trend in the recent years towards increasing large scale HPC supercomputers in the form of clusters. One can examine the historical data provided by the Top 500 List [6] to find a wealth of evidence to support this claim. The cluster is becoming the mainstream of supercomputer architecture. Such a cluster system usually adopts Remote Produce Call (RPC) for providing communication between nodes. RPC is a popular mechanism for structuring distributed system and has long been standard for implementing distributed service [1]. It is implemented on top of network transport protocol and provides a high-level communication mechanism for an application involved in a request/reply message exchange. As the cluster size increases, failures are not rare events any more [36]. A distributed cluster system, especially at a large scale, needs to be designed with a well thought out failure detection scheme. Timeouts are usually used for failure detection in RPC systems, which are typically reported on a per-call basis [23]. The RPC timeout mechanism directly impacts on many aspects of the entire RPC-based distributed systems especially responsiveness, reliability and stability [34].

Traditional RPC protocols [2], [3], [4], [5] usually use the simple timeout mechanism with a fixed timeout value (FIX mechanism for short). In this approach, the timeout value is immutable once set and used constantly through the entire communication session. If the reply does not arrive within the allotted time and the timeout reaches, the underlying RPC implementation will indicate that a failure occurred, and the client will be forced to take application-specific corrective action which might include terminating the client application, retransmitting the RPC request or at least informing the user the failure. The FIX mechanism has been applied in which the RPC requests submitted by a client can be handled quickly by the server. In such case, a reasonable timeout value, such as 30 seconds, provides ample time for the reply message to traverse the network between the server and the client. If a reply does not arrive within this time window, it is fairly safe for the client to assume that a network problem exists, or that the server system has crashed. However, it is not the case where the requested operation may require lengthy processing by the server, especially in large scale HPC clusters with heavy RPC workloads. In such environment, owing to the variation over time of network conditions and server loads, Round Trip Time (RTT) of a RPC is not varying within a predictable reasonable time window. Sometimes, it even reaches hundreds of seconds especially when the server loading is involved. At this time, it is hard to estimate the upper bound of the timeout value safely for an RPC. And the clients may determine the network or server has failed incorrectly by timeouts as the preset fixed timeout value by client is too small to adapt to the environment changes. This kind of inefficient timeouts degrades the performance of the entire system. Thus, the current FIX timeout mechanism cannot meet the requirement of applications running on large scale HPC cluster systems.

In this paper, we present an adaptive and scalable RPC timeout (AST) mechanism for large scale Lustre file system that can meet the following requirements: adjusting the client’s RPC timeout value with network and server congestion adaptively; no significant negative impact on performance for clusters at different scales under different workloads.

# OBSERVATION AND ANALYSIS

In this section, we will first introduce an observed phenomenon of the serious performance degradation resulting from inefficient timeouts with the FIX mechanism in a large scale HPC cluster under heavy load and then analyze the cause and the shortcomings of the FIX mechanism.

First, we define RPC RTT according to the processing flow of an RPC which will be used in the whole paper as follows: RTT=Tnet+Tservice；Where the RPC RTT is divided into two parts: Tnet and Tservice ; Tnet represents the network latency of delivering the RPC request and reply, and Tservice represents RPC’s service time on the server.

The RPC-based systems usually use queuing model to mange the RPC requests [24], [25], [26], [27], [28] and requests are serviced in FIFO order. In large scale HPC clusters, the nodes are usually connected with high speed links (i.e. Cray Jaguar SION can provide 889GB/s of bisectional bandwidth [30]) and the systems are not network bound. However, slow processors or disk systems will cause a RPC queue with long depth on the server and result in considerable queuing delay. And client RPC requests see huge variations in service time due to congestion and server loads.

Our observations were made on the large HPC clusters running Lustre clustered file system [7]. Lustre implements its own RPC protocol with timeouts, which is designed specifically for an implementation capable of supporting a parallel job running on tens of thousands of nodes and supports multiple implementations of underlying network protocols included TCP, InfiniBand (IB), Cray SeaStar, etc. Lustre is targeted at development of a next-generation file system supporting more than 100,000 of nodes. As HPC systems increase node counts to increase overall performance, Lustre is challenged to scale even further. And timeouts are the most complained about issues in Lustre mailing lists where most of them are caused by heave I/O loads especially on large scale HPC systems.

**Fig. 1**. Traces of the RPC workload over time on a Lustre server. The X axis is the time. Y axis of Fig. 1(a) is the queue depth of various RPC types at any instant. Y axis of Fig. 1(b) is the RTT of the finished RPC.

Fig. 1 illustrates the trace of RPCs over time on a server suffering concurrent writes from 1024 clients when we performed pressure testing on a Lustre file system. The write procedure in Lustre file system functions as shown in Fig. 2: the client sends a write RPC request to the server. Upon receipt, the request is enqueued, waiting for service. Then the server dequeues the request to execute it in the context of a service thread, and transfers the bulk RDMA data over the network and writes it to the disk. Upon the completion of the RPC request, the server sends a reply to the client. And each RPC usually writes 1MB data to the server. In Lustre, the maximal allowed number of concurrent I/O RPCs in flight between a client and a server is a tunable parameter. Its default value is 8 and we set it to 32 in our test. We use IOR [33] as our workload generator and the I/O mode is File Pre Processor (FPP) mode [9]. In the test, each client writes 512MB data to the server and the measured peak performance of the server was about 200 MB/s. The nodes were connected though high speed Infiniband switches which could provide 40 Gbps of end-to-end bandwidth. To study the normal I/O behavior under high load, we set the fixed timeout value to 300s, instead of the default 50s. From the graphs, we can see that the maximal number of queued write RPC requests reaches more than 32,000 and the maximal RPC RTT reaches as high as 197s. From the analysis of RPC trace logs on the Lustre cluster under heavy load, we also found that RPC’s Tnet was usually less than 1 second; while RPC’s Tservice occupied a relatively large portion, sometimes even reaching hundreds of seconds whereas RPC’s execution time Tp was very small and a lot of time was taken on waiting for service (Twait). It shows that in such kind of data intensive environments, heavy I/O load is the main reason caused the high latency.



Fig. 2 Lustre Write RPC flow.

To achieve high reliability, HPC systems usually need to checkpoint system memory periodically. For example, Cray Jaguar is required to checkpoint 20% of total system memory, once per hour, using no more than 10% of total compute time [30]. To gain aggregative bandwidth, the checkpoint data is usually striped across many data servers of the shared Lustre file system. During the simulation tests of checkpointing, we discovered the phenomenon that the RPC queue depth on each server reached tens of thousands in a short time, and a lot of timeouts were triggered, sometimes even resulting in sharp degradation of the performance. Via investigation, we found that the more servers the file data is striped across, the more to cause the huge number of I/O requests piling up at the contention data servers; and the data server got backed up processing RPC requests generated by a huge number of involved clients due to the slow speed disk systems, and a long queue built up as the load increased, resulting in considerable queuing delay. But the timeout value set by clients was not long enough to accommodate 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 further contributed to the server’s workload. They would also timeout, preventing any real forward progress and creating a further backlog on the Lustre servers, resulting in serious performance degradation. In even worse case, it may crash the entire system if not treated appropriately. In the Cray Jaguar system, to prevent timeouts and retries, the predefined fixed timeout value increased as high as 600 seconds to account for worst-case situations [12]. But this solution using large timeout values has drawbacks. When a server’s workload becomes less busy or the network becomes less congested, 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 [2]. And long timeouts also increase the recovery and failover time [31]. For example, the request/reply message loss of an RPC, which is caused by a temporary network failure, may block the client’s process until the long timeout reached. This makes fast transparent recovery of an RPC via retrying impossible and will slow the entire process down. If it is a nested RPC or other RPCs are dependent on it, the large timeout value may even lead to cascading timeouts. On large Lustre clusters, the observed reboot recovery/failover time often reaches tens of minutes due to large fixed timeout values. And during the recovery phase, the system can not process any new requests, which decreases the high availability of the HPC systems. All these obviously hurt the performance of the entire system. Thus, with the clusters scaling up, it is clear that the FIX mechanism has severe scalability problem.

Since the development of RPC, little has changed in the functionality it offers. But according to the observations and analysis above, for late-model large scale HPC cluster systems, the FIX mechanism is no longer suitable for the failure detection of the RPC model. In the following sections, we will introduce how our AST mechanism to reduce and avoid timeouts, and how to improve the responsiveness of the system. Our AST mechanism includes two strategies: adaptive timeout strategy and early reply strategy. Most of our algorithms are based on Lustre clusters and part of the work has already merged into 1.8 version of Lustre which has just added about 500 LOC into Lustre RPC library and been used as the default timeout setting mechanism. However, some algorithms proposed in the paper can be also generally used in many other RPC-based systems, especially in the environments with intensive access to the shared resources through a slow speed component (i.e. data intensive supercomputing).

# ADAPTIVE TIMEOUT MECHANISM

The principle of the adaptive timeout strategy (AT for short) is that the RPC’s timeout value setting by clients is adapted and adjusted in a dynamic fashion according to the network conditions and server loads. As the network or servers become congested, RPC RTT increases, and the client’s timeout value must grow to match. Similarly, as server load or network traffic decreases, RPC RTT decreases, and the client’s timeout value must reduce. In a word, as the RTT varies, the timeout should ideally follow.

Our adaptive timeout strategy decouples the network and server loading from the fixed timeout value. The basic idea is as follows. RPCs are tracked by both clients and servers; The server estimates the current RPC’s server time (EST) according to the history of RPCs’ *Tservice*, the current load, etc, and then feeds back the EST to the client in the reply message; The client estimates the current RPC’s network latency (ENT) according to the history of RPCs’ *Tnet* between the client and the server; The client sets the timeout value of an RPC ready for sending the RPC request targeted to the server according to the feedback EST (FEST) and ENT.

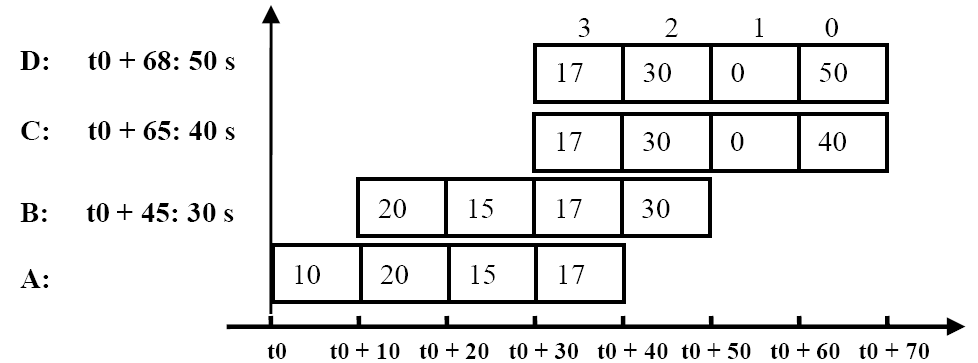
## Sliding Time Window Algorithm

At the heart of the adaptive timeout strategy is the Sliding Time Window (STW) algorithm. Each STW records the history of the latest time length *H*. The whole STW is divided into multiple Sub Sliding Time Windows (SSTWs) each with equal time length *L*, thus there are *N* SSTWs in total where *N=H/L*. Each SSTW has a record. It stores the record value adding in its own time window according to various algorithms’ requirements. The whole time window slides forwards at the integer multiple of the time step *L*. Each time more than one SSTWs slide, the SSTWs without any new records are reset. We use STWs to track the variant history of RPCs’ Tservice and Tnetcaused by the variation of network conditions and server load in the last period time window *H*. According to the records in the STW, we estimate the network latency and the service time.

First, we propose a simple STW algorithm used by the latter adaptive timeout algorithm, called MAX algorithm. In this algorithm, the saved record in each SSTW is the maximal value adding in its own time window and the estimated value is the maximal value of the ***N*** records in the STW. For ease of presentation, we define a STW for the MAX algorithm as a 3-tuple STW=(*H*, *T*, *v[N]*)where *H* denotes the whole time window length; *T* denotes the time range on time axis covered by the STW; *N* denotes the number of the SSTWs; and *v* is an *N*-dimensional vector and each element represents a SSTW used to store its own record value. The formula to calculate the estimated value in MAX algorithm is as follow:





****

**Fig. 2**. Sliding time window.

Fig. 2 shows an example using the MAX algorithm to track the variation of the RPCs’ service times. In this case, *H* is 40s; *L* is 10s; *v* is initialized with zero, and *N* is 40/10=4. The start time is t0. At the time of t0+40, T=[t0, t0+40], and the values of vector *v* are shown in row A: <17,15,20,10>. At time t0+45, the RPC’s service time 30s is added to the STW, resulting in sliding forwards one time step; at this time, T = [t0+10, t0+50], *v* = <30,17,15,20>; At time t0+65, a record with value 40s is added, resulting in sliding forwards 2 time steps; at this time, T = [t0+30, t0+70], as there are no records added in time range [t0+50, t0+60], thus *v*[1] = 0, and *v* = <40,0,30,17>. At time t0 + 68, a service time 50s is added, because new value is bigger than the old one located in the same SSTW, so we need to update the record value *v*[0] to 50s.

## Adaptive Timeout Algorithm

In our AT algorithm, each server has an EST STW to track the variation of RPCs’; while each client maintains ENT and FEST STWs to track the variation over time of the network latency and the feedback EST between each client/server pair, respectively. The EST can be tracked per server or even per type of RPC services provided by the server. Each RPC tracks its own sending time (), arrival time () and service time (). Upon the completion of an RPC, the server calculates the RPC’s and add it to the EST STW; When received a RPC reply, for the corresponding client/server pair, the client calculates the RPC’s , add it to the corresponding ENT STW and add the FEST fed back from the server to the corresponding FEST STW. Client sets the optimal RPC timeout value by using the estimation functions of history records in ENT and FEST STWs according to the formula below:

Where is the setting timeout value; is an estimation function of *N* history records in the STW where the formula of the MAX algorithm in Subsection A is a kind of such estimation functions; represents the estimated value of the network latency; represents the client-side estimated service time according to recorded FESTs in the FEST STW; is the amplification factor of ), and should meet due to the feedback latency. In order to avoid the timeout value too large or too small, the upper and lower bounds are defined: .

According to the processing flow of an RPC, the AT algorithm involves four processes: setting timeout value when sending an RPC request, and adding the measured network latency and FEST into corresponding STWs when received a reply on the client side; RPC request receipt and sending the reply on the server side. To describe the algorithm exactly, we define an RPC as a quintuple RPC = <Tsend, Tarrival, *deadline*, Tservice, FEST> where the definitions of , and are same as before; is RPC’s expiration time and its value is the sum of RPC’s and ; FEST is the feedback EST from the server. The client/server pair on the client is defined as a couple CSP =(stwNet, stwFEST) where *stwNet* denotes the STW for the network latency; *stwFEST* denotes the STW for the FEST. It is used to manage the communication channel between a client and a server. The STW to track RPCs’ service times on the server is defined as *stwEST*. The pseudo code for the MAX algorithm is described as follows.

**Algorithm 1** RPC adaptive timeout algorithm based on STW

1: **Procedure** SendRPCRequest(rpc)

2: // @now: current time

3:

4: Send the RPC request message to the server;

5: **end procedure**

6: **Procedure** ReceiveRPCRequest(rpc)

7:

8: Enqueue the new RPC request, waiting for service;

9: **end procedure**

10: **Procedure** SendRPCReply(rpc)

11:

// add the service time value to the *stwEST* on the server

12: AddRecord(stwEST, );

13: rpc.FEST = (stwEST);

14: Send the RPC reply message with FEST to the client;

15: **end procedure**

16: **Procedure** ReceiveRPCReply(rpc)

17:

18: AddRecord(CSP.stwNet, netlatency);

19: AddRecord(CSP.stwFEST, rpc.FEST);

20: Process the RPC reply message;

21: **end Procedure**

## Service Time Estimation Algorithm

In our implementation, ENT and FEST STWs on the client both adopt the MAX STW algorithm, and the STW length on the servers and clients are set to the same value to simplify the configuration. As the heavy load on the server is the main reason caused inefficient timeouts. Thus, the estimation of RPC’s service time is particularly important.

The optimal EST fed back to the client should be the of the current new arrival RPC request. For the stable RPC workload flow, RPC’s is usually constant and the service time estimation algorithm based on the MAX STW algorithm (MAX algorithm for short) can correctly estimate the EST on the server. But its EST is maximal RPCs’ in the STW added upon the completion of RPC requests. It does not consider the variation of the server load during the considerable queuing delay of the RPC with the maximal service time. In this time phase, RPC requests may keep arriving and cause the dramatic variation of the service time. Thus, it does not work so well when the RPC workload varies dramatically.

We propose a line least square curve fitting service time estimation algorithm (LCF algorithm for short). It extends the record value to track service times in each EST SSTW, and the record value is a time-value pair in the form where is RPC’s and is the corresponding RPC’s ; each SSTW records the value with max service time adding in its own time window. It determines a curve fitting function which has the best fit to the series of N discrete time-value records by using line least square curve fitting method and calculates the estimation value at current time according to the function. The estimation function for current EST *v* is given below.



As the server load has a direct impact on the service time, thus we propose another service time estimation algorithm (LOAD algorithm for short). It is based on the producer-consumer queuing model and estimates the service time based on the server loads. It works as follows. The server tracks the number of queued and serviced RPC requests (denotes as *Q*), over time, upon the receipt and completion of an RPC on the server. *Q* is used as the metric to measure the server load. And the RPC consuming rate (denoted as *C*) is time-averaged measured periodically, and its value is made equal to the division of finished RPCs and efficient working time in certain time window. Upon the completion of an RPC, the returned EST is set to *Q/C* directly.

MAX algorithm is mainly applied to the environment with stable RPC workloads; while LCF algorithm can perform better for dynamic RPC workloads. For LOAD algorithm, it can work well if the fluctuation of RPC consuming rate *C* on the server is small; otherwise, it will generate large estimate deviation. In the section of simulation evaluation, we will carry out a series of experimental comparisons for these algorithms.

# EARLY REPLY STRATEGY

As analyzed in Section 2, a congestion server sometimes may result in a lot of inefficient timeouts. Although the AT strategy can adjust the timeout value setting to accommodate the environment changes, but it cannot avoid timeouts totally. The reasons include the following aspects: (1) it exist the estimation deviation; (2) if a client keeps inactive for a long time, the timeout value set by the client will be out of track of the real network latency and service time; (3) The adaptive timeout strategy is not so sensitive to detect and accommodate burst workloads. At this time, it’s hard to distinguish whether the timeout results from heavy load on the server, or from the server death or network problem, and the client may incorrectly believe that the server or network failed. This kind of inefficient timeouts may lead wrong recovery actions and turn the entire system into an unstable state. This is clearly harmful to the entire system.



Fig. 3 Nested timeout.

In the distributed system, servers may be clients of other servers to allow chains of RPCs. It may cause the nested timeout problem when distributed operations involve multiple nodes. Fig. 3 shows an example of a chain of RPC invocations that may cause nested timeouts. It involves three nodes: A client, B client and server, C server. First A sends the RPC request r1 to node B, and during r1’s execution it invokes RPC r2 targeted at node C. In the chains of invocations, the timeout of any interaction leads the failure of the entire distributed operation. And r1’s timeout value must be no less than r2’s timeout value, or A may incorrectly believe that B has failed. But it’s usually hard to ensure r1’s timeout value to be larger than r2’s timeout value in distributed systems.

To distinguish the server congestion from server death or network failure, and to resolve the nested timeout problem, an early reply strategy is proposed to further reduce the occurrences of timeouts. When a server realizes that it cannot meet an expected response time for a client RPC request, it will send an early reply to the client indicating a more accurate estimation of completion time. Then the client will wait the indicated amount of time for a response, rather than an arbitrary fixed value. In our implementation, each server has an early reply timed list to manage buffered RPC requests on the server sorted by their deadline, together with a corresponding timer. Upon receipt, the RPC request is first inserted into the timed list, and then the server adjusts expiration time of the timer to the value of the difference of subtracting the minimal RPCs’ deadline from the reserved time (denoted as epReserve) which is used for delivering the early reply message. Upon completion of an RPC request, it is deleted from the timed list. When the timer expires, the server checks the timed list and sends early reply messages to clients for the RPCs which are about to time out and meet the following condition: now – rpc.deadline < epReserve where now is the current time. This kind of messages returns the extra server times and current ESTs to clients. And then the server adjusts the deadline value of the server-side RPC request and inserts it to the timed list again; when received the early reply message, the client adjusts the timeout value of the corresponding RPC request on the client side and waits for an extra time for a normal response. The algorithm above is called ERP for short.

However, it is hard to determine a reasonable extra server time for an RPC when the early reply is triggered. In our implementation, we set it to a preconfigured static value. An early reply quick checking scheme upon receipt (ERPCK for short) is proposed. When a server receives an RPC request, it first calculates the maximal service time () allowed by the client. If this value is lower than current EST, set the estimated extra service time to directly, and send the early reply immediately. By this way, the EST is also returned, so that the client can be aware of the variation of the EST quickly.

In early reply strategy, the early reply message usually can reach the client to adjust the timeout value before the timeout occurs if epReserve is set reasonably. As a result, client can assume a failure of the server or network if not receive such a message. For the nested timeout problem in Fig. 3, the early reply strategy can adaptively adjust r1’s timeout value bigger than r2’s timeout value by the early reply message if r1 needs a long processing time and the original timeout value of r1 is too small. It solves the nested RPC timeout problem in distributed systems. As is known, the responsiveness is an essential criterion to optimize RPC applications. Timeouts may result when clients make requests to servers, or when servers make requests to clients. For the former a long timeout is acceptable. However, for the latter the servers expect a response in a much shorter time to retain responsiveness of the cluster. At this time, the early reply strategy can be used to adjust the original small timeout values of server requests and its role to improve the system response speed is more obvious.

In patent [2] it presents a polling based mechanism (POLL mechanism for short) for handling timeout in a standard RPC connection. After submitting an RPC request, the client periodically makes secondary request to the server to determine if the server is still actively processing the primary RPC request. If the secondary request is processed successfully and the server indicates that the primary request is still in progress, the client will continue to wait until either the primary request completes or enough time elapsed to warrant another secondary request. If the secondary request fails, this failure is treated as a sign that there is either a network or a server problem, and the client is triggered into taking appropriate corrective action. Although this mechanism can also solve the problems mentioned above, but it is client-driven and exists scalability issue. Firstly, it needs twice message passing for each poll. Secondly, the polling interval is hard to determine. If set too large, it may wait for a long unnecessary time before detecting failure, reducing the responsiveness. If set too small, on the other hand, it may load the network with excessive unnecessary traffic and this clearly hurts performance. Thus it is not suitable for large scale cluster systems. Our early reply strategy is server-driven. The sever sends early reply messages intelligently according to the RPCs’ deadline. Client adjusts the timeout value according to the indicated extra service time in the early reply. Compared to the existing strategy, it generates much less extra messages and it is much more suitable for large scale cluster systems.

# SIMULATION EVALUATION

## Expermient Setup

In order to evaluate the performance of our newly proposed mechanism and compare with existing ones, the clustered file system Lustre that has been widely used by many HPC supercomputers was chosen as the evaluation platform. We use the Lustre simulator [13, 29] to evaluate our mechanism. Lustre simulator is developed as a Lustre simulation platform to study its scalability, analyze I/O behaviors, and design various algorithms at a large scale. It simulates the disk systems, packet-level network, Lustre’s RPC protocol and three Lustre subsystems: Client, Metadata Server and Object Storage Server [7]. It can simulate concurrent operations of 100,000 clients.

The main metrics that were measured includes the timeout rate (denoted as ), responsiveness to detect failures, adaptability of the algorithm, scalability and performance, etc. We use the ratio of RPC’s RTT to RPC’s , denoted as , to measure the responsiveness and adaptability, where the measured set is the RPCs that finished before timeouts reach. Obviously, and the larger is, the better responsiveness and adaptability is achieved, and the better the response speed of failure detection it will get.

The overall performance of our mechanism is measured by simulating concurrent write RPCs from a large number of clients destined to a server using IOR FPP mode (to multiple servers is just a simple extension of this case as I/O RPCs to different servers are usually independent). To simplify simulation experiments, the maximal number of RPC requests in flight between a client and a server is set to 1. By this way, each client sends I/O RPC requests one by one with a synchronized manner. The simulator is configured as follows. The simulated network can provide 10 Gbps of end-to-end bandwidth. The backend disk bandwidth of the server is about 300MB/sec. The total client number is 32,000. If not specified, the default common parameters in experiments are as follow: AtMax = 600s; for STWs on both the server and clients, H is set to 40s, L is set to 5s and records in SSTWs are all initialized to 0. To make some noises, we set the time skew between clients to start I/O processes, and the clients’ start times are a uniform distribution in the overall time skew. When timeouts are triggered, no any recovery actions are taken, and the server and clients perform as normal. Two kinds of RPC workloads are simulated: burst workload and stable workload. As the observed RPCs’ are all less than 1s, we mainly focus on the evaluation and analysis of service time estimations.

## Evaluation of Adaptive Timeout Strategy

To evaluate our AT strategy and compare with the FIX mechanism, we simulate a RPC workload instance that the variation trend of the number of queued RPCs on a server over time presents a wave crest shape. The experiment is designed as follows: the time skew is set to 120s; each client write 4MB data, that is to say, each client sends 4 synchronous RPCs one by one to the server; AtMin = 0. For easy of description, we use *f* to represent the fixed timeout value. As the timeout value of the first RPC on each client is out of control of our strategy, thus we divide the set of overall RPCs (S) into two subsets: S1 and S2. The first RPCs of all clients belong to S1; all the other belong to S2. In S1, the timeout value of the first RPC under the adaptive timeout strategy is set to *f* mandatorily. And we only analyze the values for the RPCs that do not trigger timeouts in S2.

Fig. 4 RPC traces of FIX mechanism with *f* = 50s.

First, the FIX mechanism is evaluated. Fig. 4 illustrates the RPC traces of FIX mechanism with *f* = 50s. Fig. 4(a) shows the variation of the RPC queue depth over time and maximal queue depth reaches almost 30,000. It could be used as the metric to measure the server load. Fig. 4(b) shows the variation of RPC RTT over time and maximal value reaches 113s. And the two corresponding curves of the AT strategy are almost same. Fig. 4(c) depicts the variation of timeouts per second over time. It shows that timeouts are triggered from about 100s and continued to the end, where most of RPCs are destined to timeout due to the already existed large queuing delay when they are sent. The final timeout rate is 76%. Fig. 5 shows the timeout rates for various *f* values. It shows that with *f* = 25s the timeout rate reaches as high as 87%. It prevents timeouts until *f* increases to 125s. Fig. 6 plots its corresponding variation over time where the X axis is the time that the RPC reply arrivals at the client and the Y axis is the RPC’s . It shows at the beginning is very low; and then as RTT increases it is increasing accordingly, and at about 300s it reaches the maximal value 0.9; after that, it decreases as RTT reduces. The mean is 0.61. And we believe that the higher *f* is set to, the lower the mean we will get.



Fig 5. Timeout rate for FIX mechanism with various f values.



Fig. 6 variation over time for the FIX mechanism with *f* = 125s.

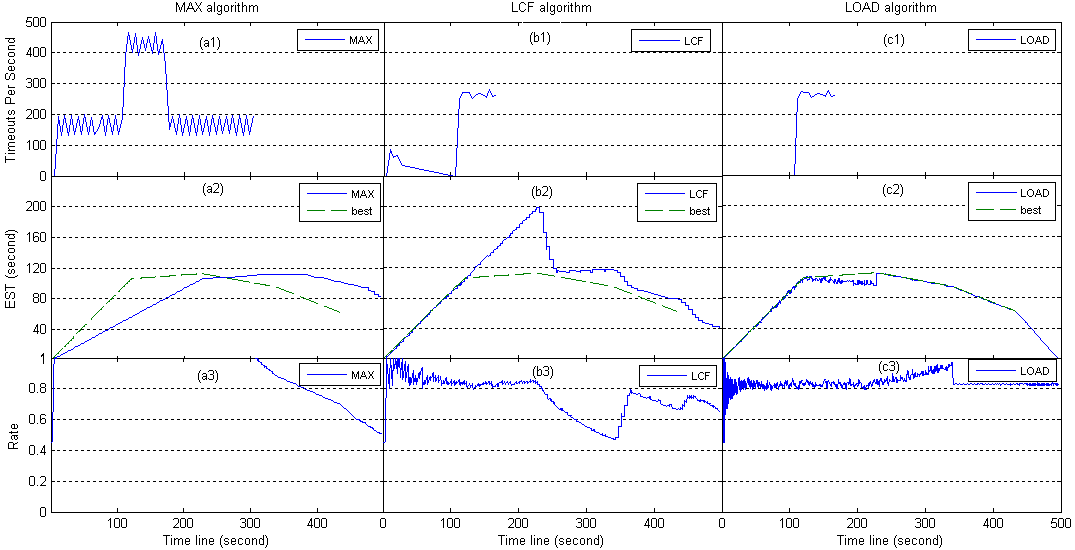


Fig. 7 RPC Traces of the AT strategy with various service time estimation algorithms. . Graphs in the first row illustrate the variation of timeouts per second over time; Graphs in the second row illustrate the EST variation over time; Graphs in the third row illustrate the variation over time.

Table 1 and mean statistics for various service time estimation algorithms with different values in S1 and S2, respectively.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| λ | | | 1.0 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
| MAX | S | *Rto* | 63% | 56% | 52% | 48% | 45% | 43% | 41% | 39% | 36% | 18% | 13% |
| S2 | *Rto* | 67% | 57% | 52% | 47% | 43% | 40% | 37% | 34% | 31% | 7% | 1% |
| *Rra* | 0.82 | 0.79 | 0.75 | 0.72 | 0.69 | 0.66 | 0.63 | 0.61 | 0.60 | 0.67 | 0.66 |
| LCF | S | *Rto* | 27% | 15% | 14% | 13% | 13% | 13% | 13% | 13% | 13% | 13% | 13% |
| S2 | *Rto* | 19% | 3% | 2.5% | 0.6% | 0.5% | 0.4% | 0.3% | 0.3% | 0.2% | 0.2% | 0.2% |
| *Rra* | 0.82 | 0.78 | 0.72 | 0.67 | 0.62 | 0.58 | 0.54 | 0.52 | 0.49 | 0.46 | 0.44 |
| LOAD | S | *Rto* | 41% | 18% | 13% | 13% | 13% | 13% | 13% | 13% | 13% | 13% | 13% |
| S2 | *Rto* | 38% | 7% | 0.1% | 0.1% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| *Rra* | 0.98 | 0.91 | 0.84 | 0.78 | 0.72 | 0.68 | 0.64 | 0.60 | 0.57 | 0.54 | 0.51 |

And then, we evaluate our adaptive timeout strategy. In the rest of experiments in this paper, *f* is fixed at 50s. Fig. 7 shows the various traces of our strategy with by using MAX, LCF and LOAD algorithms, respectively. For graphs in the second row, the curves “best” are plotted by using RPCs’ as the X axis and RPCs’ as the Y axis. It is used as the best EST to determine the effects of server time estimations. We start with the analysis about EST illustrated in these graphs. For MAX algorithm, before 260s its EST is all underestimated and almost all of the timeouts occurred in this period; afterwards its EST is overestimated. It prevents timeouts from about 300s. But as it lags to reflect the decrease of the service time, its is decreasing gradually from 1 to 0.5 with the decreasing of RPC RTT. For LCF algorithm, before 120s the curves “LCF” and “best” are almost overlapped, and is above 0.8 in this period; From 120s, the RPC queue depth starts to decrease as illustrated in Fig. 4(a) and the server load reduces gradually, but its EST can not follow immediately until the time that completed RPCs update the records in the EST STW, resulting in the degradation of ; however, the experiment results show that it doesn’t result in any new timeouts and its EST can decrease promptly to adapt to the workload changes once detect at about 200s. For the LOAD algorithm, the observed RPC consuming rate fluctuates between 250~280; Fig. 7 (c2) and (c3) show that in almost all of the time, its EST is correct estimated and its is above 0.8. Finally, the timeout rates of MAX, LCF and LOAD algorithms are 52%, 14% and 13%, respectively.



Fig. 8 variation over time for MAX mechanism with .

Table I shows and statistics for various service time estimation algorithms with varying from 1.0 to 2.0 in RPC set *S* and *S2*, respectively, where the recorded is the mean. In the experiments, each test case generates 128,000 RPCs in total. The timeout rates in S1 for all test cases above are almost constant at 13%. Table 1 illustrates that when increases to 1.3, unnecessary timeouts are nearly eliminated by using LCF and LOAD algorithms; increasing value continually can only reduce . As the EST of MAX algorithm can not well adapt to the dramatic changes of RPC workloads, thus timeout rate in S2 drops to 1% until increases to 2.0; the corresponding variation over time is shown in Fig. 8 and the mean is 0.66. It is better than the result of the FIX mechanism with *f* = 125s

Finally, we carry a series of experiments to evaluate the impact of different STW’s settings (the STW length *H* and SSTW length *L*) on the results. In the experiments, the RPC workload is same as previous and is fixed at 1.2. First, we fix *H* to 40s and set *L* to 20s, 10s, 5s and 2s, respectively. The results of MAX and LOAD algorithms are not affected by the changes of *L*. But for LCF algorithm, although the values have nearly no changes but the timeout rates decrease from 18% to 14%. It illustrates that the decrease of *L* increases the total records (*N*) in the STW and it can result in better estimation by curve fitting. Second, we fix *L* at 5s and set *H* to 160s, 80s, 40s, 20s, respectively. The results show that the timeout rates for various algorithms have almost no changes. And the of the MAX algorithm slightly decreases from 0.75 to 0.73; the one of LCF algorithm decreases from 0.72 to 0.65; the one of LOAD algorithm decreases from 0.85 to 0.76. It illustrates that the increase of *H* decreases the adaptivity and responsiveness of the algorithms. However, the changes of settings have less effect on the final results. Thus, the results with *H* = 40s and *L* = 5s can present a common significance of our algorithms.

## Evaluation of Early Reply Strategy

Based on the experiments in the previous subsection, we evaluate our early reply strategy and compare with the POLL mechanism. In the experiments, the extra message size is 4K; *epReserve* is 5s; AtMin is set to 10s; the preconfigured static extra service time is 20s; the polling interval is also 20s; all the other settings are same as previous. The experiment results show that all early reply messages arrive at clients in the reserved time to adjust the timeout value successfully; for some extreme RPCs, the client’s timeout value can be amended several times by multiple early reply messages passing to avoid the occurrences of inefficient timeouts; and the timeout rates combined with early reply strategy all drop to 0%. In the following, we mainly focus on analyzing the extra traffic generated by various strategies.

Fig. 9 depicts the variation of the outgoing network traffic of the server NIC over time for different strategies. In the experiments, the network bandwidth occupied by normal RPC replies is almost constant at 0.5 MB/s. Table 2 shows the statistics of extra messages generated by various strategies.

First, we make a comparison between ERP and ERPCK algorithms based on Fig. 9 and Table 2. From the table, we can see that for ERP lots of RPCs need extra message passing more than one time or even 5 times to avoid inefficient timeouts.

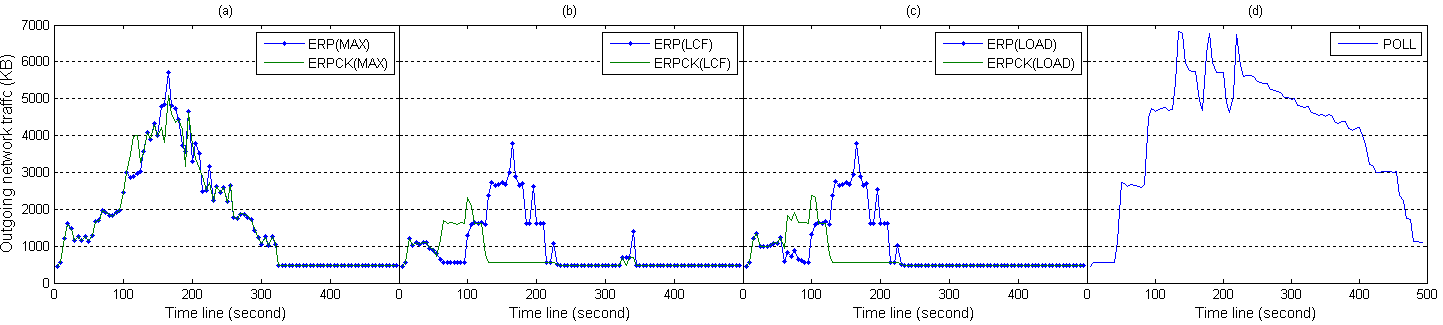


Fig. 9.Traces of the network traffic for various strategies with a dynamic RPC workload. ERP(*name*) means the early reply strategy combined with #*name* service time estimation algorithm; similarly, ERPCK(*name*) represents the enhanced early reply strategy (with quick checking upon RPC receipt) combined with #*name* service time estimation algorithm. It plots the time on the X axis and the network traffic per second on the Y axis.

Table 2 Extra message statistics.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Number of RPCs | | Number of extra messages per RPC | | | | | Total extra messages |
| 1 | 2 | 3 | 4 | 5 |
| ERP | MAX | 17972 | 24705 | 22287 | 4535 | 601 | 155388 |
| LCF | 11526 | 4579 | 4552 | 4564 | 559 | 55427 |
| LAOD | 11355 | 4549 | 4517 | 4574 | 629 | 55445 |
| ERPCK | MAX | 18023 | 24697 | 22334 | 5095 | 0 | 154799 |
| LCF | 25485 | 0 | 0 | 0 | 0 | 25485 |
| LOAD | 24452 | 0 | 0 | 0 | 0 | 24452 |
| POLL | | 11839×2 | 11867×2 | 20080×2 | 25053×2 | 47339×2 | 432720×2 |

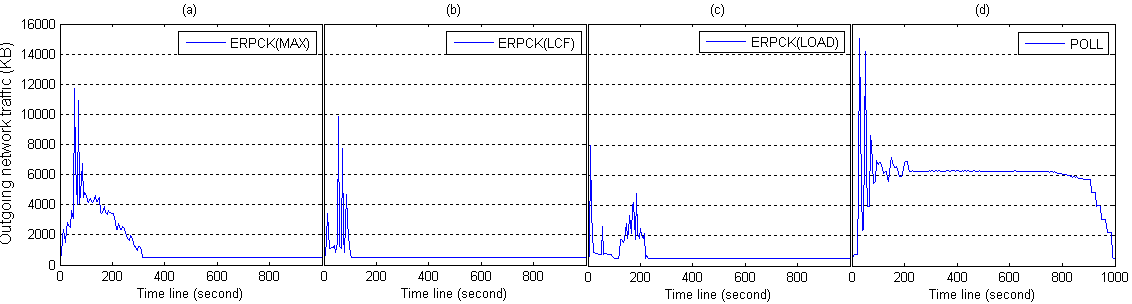
Fig. 11. Network traffic comparison between the early reply strategy and the POLL mechanism under a relatively stable RPC workload.

Fig. 10 RPC Traces for the stable RPC workload.

In contrast, for ERPCK, especially combined with the LCF or LOAD algorithm, one extra message passing is usually enough. Fig. 9 and Table 2 show that ERPCK generates less network traffic than ERP. The results demonstrate that with the feature of early reply quick checking, the server can provide reasonable extra service time rather than a fixed value and make the client notice the EST change quickly, and the enhanced ERPCK strategy can reduce extra messages significantly.

And then we compare our early reply strategy with POLL mechanism. Table 2 illustrates that POLL mechanism generates 432720×2 extra messages in total. The amount of extra network traffic reaches as high as 3380MB. In contrast, with the early reply strategy, the worst ERP(MAX) generates 155388 extra messages and the overall extra network traffic is 606MB. And the ERPCK(LCF) and the ERPCK(LOAD) only generate about 100MB extra network traffic. Another set of experiments are carried out to present the advantage of our strategy under a relatively stable RPC workload as follows. We only increase the I/O amount per client to 8MB and the time skew is shortened to 10s; all other configurations are some as previous. Figure 10 depicts the RPC traces of the experiments. It shows that RPC queue depth reaches 32,000 in a short time and is keeping at this high value for a long time. Fig. 11 shows the corresponding network traffic comparison between ERPCK strategy and POLL mechanism. It shows that in most of the time the network bandwidth occupied by the client-driven polling is more than 6MB/s. If count the network traffic caused by the extra second request messages, it is doubled to 12MB/s. And the total number of extra messages reaches 2670584 and the amount of extra network traffic is 10,432MB. In the early reply strategy, it doesn’t generate any early reply messages from about 300s. For ERPCK(MAX), the total extra messages is only 215007, and the amount of network traffic is 839M. According to the results above, for the POLL mechanism, when hundreds of server nodes in large HPC clusters suffer such workloads, with a huge number of RPCs in progress, the extra polling messages can cause very considerable network traffic, and the occupied network bandwidth can even reach several GB/s. It obviously hurts performance and has serious scalability problem. In contrast, in our early reply strategy, for relatively stable RPC workloads, the upper bounds of the timeout value can be correctly estimated using the adaptive timeout strategy, and it rarely generates extra early reply messages.

The current HPC systems are usually required to recover automatically after failure for providing resilient and continuous service [35]. It is usually implemented by retires when timeouts reach and a suspected failure is detected. However, early replies are better than reties. First, from the above comparison experiments, we can know that if the fixed timeout value is set to same as polling interval, it will generate almost same amount of retry messages and may make the server busy on handling the duplicated retry requests. Early replies can make clients notice the changes of network latency and service time, and the timeout value of subsequent requests can be correctly estimated, thereby reduce timeouts and the followed reties. Second, early replies can avoid unnecessary timeouts by the notification of the early reply and further reduce the network traffic due to the smaller early replies’ size compared with the normal requests’. More importantly, it can reduce consumption of server’s CPU/memory resources to handle the duplicated retries. From these viewpoints, early replies can even help to improve the performance compared with retires.

# RELATED WORKS

As we known, failures such as lost messages, network failure and node crashes can cause reliability problems in a distributed system. The detection and treatment of failures [16], [17], [23] is one of main issues that requires close attention in an RPC design. Most distributed systems that are based on RPC protocols use timeouts for failure detection [3], [4], [14], [15], [18], [19], [20], which is especially useful when the underlying protocol is unreliable. They usually used fixed timeout mechanism.

However, the research on the failure detection with timeouts in distributed systems is relatively rare. For large scale cluster systems, to the best of our knowledge, our mechanism is the first failure detection mechanism for RPC model with timeouts that considers network conditions, server loads, scalability and performance, etc.

Delaney, William P. et al. [2] propose a similar adaptive timeout setting method that takes into consideration the communication time between a client and a server. In the method, an RPC’s response time is tracked by the client and recorded in a storage array. An optimal timeout value is determined from recorded response times by the client. Our adaptive timeout strategy uses STWs to track the variation of RPC’s and over time. The RPC’s is estimated by clients while the RPC’s is estimated by the server and fed back to clients. The setting of the timeout value is calculated based on these two estimated times. As the server is more intelligent to estimate the server time according to dynamic changes of its load, thus it can adapt to the environment changes more sensitive especially in the environment with large scale dynamic RPC workloads. In the protocol of Linux SUN RPC which is mainly used by NFS, the client also sets a request’s timeout value according to the measured RTT and retry count. The timeout value is calculated by the formula: . Although this method can reduce the number of extra messages, but it reduces the speed of failure detection compared with our mechanism.

Some RPC protocols have no timeout mechanism limiting the duration of a remote call. They use other mechanisms to implement the failure detection. In Cedar RPC [1], the client periodically sends a probe packet to the server which is expected to acknowledge from the server. This allows the client to notice if the server has crashed or there is some serious communication failure and to notify the user of an exception; NCA/RPC [21], [22] provides extra routine used by the client to send “ping” packets to inquire an outstanding request and “quit” packets to inform the server that it is abort processing of the remote call. In Sprite RPC model with timeouts [20], if the remote computation takes a long time and result in a timeout, the client resends the request message to the server and requests an explicit acknowledgement. The server responds with an explicit acknowledgment to indicate that the remote computation is in progress. All mechanisms above are similar to the polling mechanism in patent [5]. In Section 5, we have already evaluated that they exit scalability and performance problems and our AST mechanism is a more advantage mechanism to implement the failure detection.

# CONCLUSIONS

Due to the scale and complexity, current supercomputers have frequent failures. The speed of failure detection, reaction and recovery is becoming more critical in effectively supporting fault tolerance in large Petascale systems and future exascale systems [35]. In this paper, we have investigated the timeout mechanism widely used for failure detection in RPC-based distributed systems. To solve the problem of the fixed timeout mechanism emerging in large scale HPC cluster systems, we present an AST mechanism that considers the network conditions, server loads, scalability and performance, etc. It includes two strategies: STW-based adaptive strategy, early reply strategy. The results from series of evaluation experiments demonstrate that (1) the adaptive strategy can adapt the setting timeout value to the dynamic changes of network conditions and server loads, reducing timeouts significantly and it even eliminates the occurrences of timeouts combined with the early reply strategy; More importantly, it increase the speed and accuracy of failure detection using timeouts; (2) in large scale clusters under heavy RPC workload, other failure detection mechanisms, such as client-based polling and probing, may load the network with considerable unnecessary traffic and has scalability problem; while our early reply strategy is more attractive that generates much less network traffic and improves the system responsiveness. And with AST mechanism, reboot recovery or failover time on lightly-loaded Lustre clusters drops significantly [31], [32]. All these prove that our AST mechanism is a more suitable failure detection mechanism for RPC models with timeouts than traditional fixed timeout mechanism. Furthermore, it enhances the system responsiveness, reliability and stability without significant negative impact on performance even for large scale cluster systems..

# REFERENCES

1. B. VI. VII. [1] G. O. Young, A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. ACM Transactions on Computer Systems, 2(1):39-59, February 1984.
2. US Patent 6536433. Adaptive timeout value setting for distributed computing environment (DCE) applications. http://www.patentstorm.us/patents/6526433/claims.html.
3. B. McClure and D. O’Dea. Development of an Interoperable, Survivable Pilot DCE Application. Scientific Publication. Report Number: DSTO-TN-0025. AR number: AR-009-480.
4. SUN RPC, RFC 1831, RPC: Remote Procedure Call Protocol Specification Version 2.
5. US Patent 7146427. Polling-based mechanism for improved RPC timeout handling. http://www.freepatentsonline.com/7146427.html.
6. TOP 500 Supercomputers, http://www.top500.org/.
7. Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System White Paper, http://www.sun.com/offers/details/LustreFileSystem.html
8. Peta-Scale I/O with the Lustre File System White Paper, http://www.sun.com/offers/details/Peta-Scale\_wp.html
9. GFahey, M. I/O performance on a massively parallel Cray XT3/XT4. In Parallel and Distributed processing, 2008.
10. N. Nieuwejaar, D. Kotz. File-access characteristics of parallel scientific workloads. In IEEE Transactions on Parallel and Distributed Systems, 7(1): 1075-1089, October 1996.
11. Feng Wang, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D.E. Long, File System Workload Analysis for Large Scale Scientific Computing Applications, in: the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004), pp. 139-152.
12. 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.
13. Lustre simulator. https://bugzilla.lustre.org/show\_bug.cgi?id=%2013634.
14. F. Panzieri, S. K. Shrivastava. Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing. Jan. 1988 IEEE Transactions on Software Engineering.
15. Aurelien Bouteiller, Frederic Desprez. Fault Tolerance Management for a Hierarchical GridRPC Middleware. In Cluster computing and the Grid, 2008, 484-491.
16. Sacha Krakowiak. Middleware Architecture with Patterns and Frameworks. http://sardes.inrialpes.fr/~krakowia/MW-Book/Chapters/Intro/intro-body.html
17. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 34(1):225-267, March 1996.
18. Chris Mayers. ANSAwise – Remote Procedure Call in Distributed Systems. http://www.ansa.co.uk/ANSATech/94/Training/134401.pdf
19. S Smith, M Takatsuka. Towards a Standards-based, Message-oriented Advanced Collaboration System. Journal of Research and Practice in Information Technology, 2007.
20. Brent B. Welch. The Sprite Remote Procedure Call System. Report No. UCB/CSD 86/302, June 1986. Computer Science Division (EECS) University of California.
21. B.H. Tay, A. L., Ananda. A survey of remote procedure calls. ACM SIGOPS Operating Systems Review, Volume 24, Issue 3 (July 1990), Pages: 68-69, 1990.
22. Dineen, T. H., Leach, P.J., et al, "The Network Computing Architecture and System: An Environment for Developing Distributed Applications", In Proc. of the 1987 Summer USENIX Conference (Phoenix, Ariz., June),USENIX Association, Berkeley, Calif., 1987, pp. 385-398.
23. Kenneth P. Birman, Bradford B Glade. Consistent Failure Reporting in Reliable Communication Systems. Technical Report: TR93-1349, May 1993.
24. Mitsuhisa Sato; Motonari Hirano; Yoshio Tanaka; Satoshi Sekiguchi. OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP. In International Workshop on OpenMP Applications and Tools, WOMPAT 2001, Jul 30-31, 2001, West Lafayette, IN, USA.
25. Yusheng Liu Hoang, D.B. A performance model for the OSI RPC architecture. In Proceedings of IEEE Singapore International Conference on Networks, 1995. Theme: 'Electrotechnology 2000: Communications and Networks'. [in conjunction with the] International Conference on Information Engineering, Jul 1995, pages: 31-35.
26. Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy. Lightweight remote procedure call. ACM Transactions on Computer Systems (TOCS), Volume 8, issue 1, 1990, pages: 37-55.
27. A. M. Khandker, P. Honeyman, T. J. Teorey. Performance of DCE RPC. In 2nd International Workshop on Services in Distributed and Networked Environments (SDNE), 1995, pp.2.
28. Marc F. Pucci, J. L. Alberi. Optimized Communication in an extended remote procedure call model. ACM SIGARCH Computer Architecture, Volume 16, Issue 4, 1998, pages: 37-46.
29. Yingjin Qian, Eric Barton, Tom Wang, Nirant Puntambekar, Andreas Dilger. A Novel network request scheduler for a large scale storage system. In the Proceedings of the International Supercomputing Conference, 2009 (ISC’09). Published in Computer Science – Research and Development, Volume 23, Number 3-4, 143-148.
30. Sarp Oral, Feiyi Wang, David Dillow, Galen Shipman, Ross Miller, Oleg Drokin. Efficient Object Storage Journaling in a Distributed Parallel File System. In proceedings of 9th USENIX Conference on File and Storage Technologies (FAST’10).
31. Peter J. Braam. Scalable locking and recovery for network file systems. In Proceedings of the 2nd international workshop on Petascale data storage, 2007, pages:17-20.
32. Snappy recovery. https://bugzilla.lustre.org/show\_bug.cgi?id=18948.
33. IOR HPC Benchmark. http://sourceforge.net/projects/ior-sio/.
34. H. Shan, J. Shalf. Using IOR to analyze the I/O performance of XT3. In the Proceedings of 49th Cray User Group (CUG) Conference, Seattle, WA, 2007.
35. Frank Cappello, Al Geist, Bill Gropp, Sanjay Kale, Bill Kramer, Marc Snir. Toward Exascale Resilience. Technical Report of the INRIA-Illinois Joint Laboratory on PetaScale Computing, TR-JLPC-09-01.
36. B. Schroeder, G. Gibson. Understanding Failures in Petascale Computers. In SciDAC 2007, Journal of Physics, Conference Series 78 (2007) 012022.