An Efficient Fault-Tolerant Algorithm for Distributed Cloud Services

Conference Paper · December 2012

DOI: 10.1109/NCCA.2012.21

CITATIONS
9

READS
601

3 authors:

Jameela Al-Jaroodi
Robert Morris University
183 PUBLICATIONS **5,294** CITATIONS
SEE PROFILE

Nader Mohamed
California University of Pennsylvania
236 PUBLICATIONS **7,180** CITATIONS
SEE PROFILE

Klaithem Nuaimi
United Arab Emirates University
24 PUBLICATIONS **702** CITATIONS
SEE PROFILE

An Efficient Fault-Tolerant Algorithm for Distributed Cloud Services

Jameela Al-Jaroodi, Nader Mohamed, and Klaithem Al Nuaimi
Middleware Technologies Lab, College of Information Technology, UAEU, PO Box 17551, Al Ain, UAE
jaljaroodi@gmail.com, nader.m@uaeu.ac.ae, klaithem.nuaimi@uaeu.ac.ae

***Abstract*—Several approaches for fault-tolerance in distributed systems were introduced; however, they require prior knowledge of the environment's operating conditions and/or constant monitoring of these conditions at run time. That allows the applications to adjust the load and redistribute the tasks when failures occur. These techniques work well when there is no high communication delay. Yet, this is not true in the Cloud, where data and computation servers are connected over the Internet and distributed across large geographic areas. Thus they usually exhibit high and dynamic communication delays that make discovering and recovering from failures take a long time. This paper proposes a delay-tolerant fault-tolerance algorithm that effectively reduces execution time and adapts for failures while minimizing the fault discovery and recovery overhead in the Cloud. Distributed tasks that can use this algorithm include downloading data from replicated servers and executing parallel applications on multiple independent distributed servers in the Cloud. The experimental results show the efficiency of the algorithm and its fault tolerance feature.**

Keywords*—*Cloud computing, heterogeneous distributed systems, fault-tolerance, load balancing

I. INTRODUCTION

Cloud service providers offer diverse information technology (IT) solutions as on-demand services. The Cloud clients range from individual users that use specialized software for designing their personal systems to small, medium, and large organizations that rely on the cloud for all or some of their IT needs. Cloud clients pay for utilizing the computing capabilities and this compels Cloud companies to offer capable, efficient, reliable, and highly available services that satisfy the clients' needs. These services usually operate on computing infrastructures that are globally distributed. In addition, data and services are usually replicated to provide scalable and efficient solutions to various clients at different locations. While for some client applications it is enough to utilize one Cloud site to get the required services, others using high performance applications may need to utilize multiple available replicated resources. Using multiple resources enhances performance in terms of execution time and availability. However, some of the challenges in utilizing replicated services among the distributed-resources in the Cloud are (1) the highly heterogeneous resources; (2) the highly dynamic loads on these resources; (3) the geographical distribution causing high communications delays; and (4) the extreme variations in communications delays and available bandwidth.

Several approaches for enhancing performance, reliability, and availability in distributed systems were introduced; however, most of them require prior knowledge of the environments operation conditions and/or constant monitoring of these conditions at run time. These techniques were designed with the assumption that there is no high communication delay in discovering and recovering from failures. This is not the situation in the Cloud due to the high communications delays among the replicated resources. In this case, discovering faults and recovering from server/network failures will take a significant amount of time. In this paper we develop a delay-tolerant fault-tolerant algorithm that can be used effectively for reducing the execution time of some distributed tasks by parallelizing the tasks, while minimizing the fault discovery and recovery overhead in the Cloud. Such tasks include downloading data from replicated servers and executing parallel applications on multiple independent servers. In our approach, fault-tolerance is automatically inherent from the technique.

The main idea of the algorithm is that when you have two servers, each server starts processing the task in an opposite direction from the other such that they will not have to contend on shared resources or coordinate their efforts. Thus they will work independently at their own speeds until they meet. As a result, we do not need to know any information about the resources used before hand and we do not have to monitor these resources during run time. Yet we still maintain good load balancing among all participating servers. In addition, if one of the servers fails before completing a task, the second server will continue its work until it reaches the point where the other server has stopped. It is like having a long fence that needs to be painted and two painters. You could just tell them to each take one half of the fence and this could be a problem if one of the painters is slow or stops painting because you will have to wait for him to finish his half. On the other hand, you could just start the painters at the opposite ends of the fence and get them to move towards each other. Here the painters will work independently and will continue until they meet somewhere along the fence and they will be done at the same time, while the meeting point will change depending on who was faster.

In this paper, we first offer an overview of some related work on fault-tolerance in the Cloud in Section II. Then we define the problem scope and describe the fault-tolerant algorithm in Section III. In Section IV we show examples of application domains for the proposed algorithm provide experimental evaluations of the algorithm. In Section V we conclude the paper.

978-0-7695-4943-9/12 $26.00 © 2012 IEEE
DOI 10.1109/NCCA.2012.21

II. Related Work

One way to improve service performance is to rely on replication and parallelization. However, in the Cloud, it is hard to rely on parallel models as the communication delays diminish any gains achieved. Yet, there are several areas where we could achieve good results if we reduce the overhead. In areas such as data transfer and compute-intensive processes, it is possible to achieve that, provided efficient approaches are introduced to achieve load balancing and fault tolerance while minimizing overhead.

When working with data intensive applications or exchanging high volumes of data, it is important to have efficient and fast methods to move this data around and make it available for the applications and the clients. In addition, load balancing and fault tolerance become considerably important for such domains. Numerous models and techniques were introduced to achieve fast and reliable data handling including, but not limited to the following. GridFTP [4] extends FTP by adding features that allow for efficient mass data transfer using parallel TCP streams and data striping over multiple servers. It also implements the fault-tolerance feature based on checkpointing. One enhancement to GridFTP [20] offers a middleware framework to enhance reliability and performance. In BitDew [12] the support for reliable data and file transfers is offered through an abstract level over the Grid and the Desktop Grids. PFTP [7] uses striping and parallel connections to move data from cluster to cluster. PFTP does not offer fault tolerance, however, if the underlying parallel file system is fault tolerant, it will inherit the features. In [23] the authors present a parallel download scheme that guarantees that the file requested is received from the fastest available mirroring site. This model does not utilize parallel transfer, yet it offers fault-tolerance based on replication. Using a proxy to bring in file blocks from multiple replicas then delivering them to the client is also possible [14].

On the other hand, Cloud Computing defined in [16] as a model that enables ubiquitous, convenient, on-demand Internet access to a shared pool of configurable computing resources, has gained a strong popularity. According to the NIST Cloud computing has several essential characteristics such as on-demand self service, broad network access, resource pooling, rapid elasticity, and measured services. Due to the heterogeneity and high distribution of resources, one major issue with cloud services that need to be addressed effectively is fault handling. When Cloud resources fail, hundreds to millions of clients could be affected, leading to disastrous implications [25]. To offer such efficient and highly available services in the Cloud, the industry was leading the way with various tools and techniques that offer service registry, lookup, integration and measurement features. In addition several vendors also include features to provide scalability, dynamic resource allocation, load balancing, and reliability. For example RackSpace [21] offers load balancing and fault failover capabilities for service providers to ensure a balanced load on all servers and the reliability of the connections for the clients. RightScale [24] is a solution suite to help build and manage cloud infrastructures. It offers resource management and service development tools in addition to load balancing and reliability features. CloudLeverage [10] optimizes resource utilization across the cloud infrastructure to enhance performance. It ensures high resources availability using replication and failover mechanisms. Another example is GigeNet Cloud [15] which offers global level scalability, high availability and load balancing to incoming workloads on the Cloud.

Parallel to the industry advances, researchers have been working intensively to offer efficient tools and approaches that will enhance the Cloud infrastructures, services and facilities. One interesting approach is for handling high performance computing requirements in the cloud, in which case fault tolerance becomes a very important issue since the services in use would be handling large processing components and also high volumes of data. One approach that addresses this issue is by enhancing MapReduce to work in the cloud. In [26] the Hadoop MapReduce offers features to use MapReduce in the cloud with load balancing, while in [29] the fault tolerance aspect is also introduced. In addition, some researchers also introduced a technique to offer reliable intermediate data storage (ISS) [17] which complements Hadoop operations by providing a replication and recovery mechanism to ensure continuous availability of intermediate data even with failures. Along the same direction, VGrADS [22] offers qualitative resource abstractions to facilitate the development and execution of high volume scientific applications over the Grid and the Cloud. VGrADS offers deadline sensitive scheduling and fault tolerance based on automatic tasks replication. In addition, the authors in [19] use MODISAzure to experiment with Satellite data processing over the Azure Cloud Platform with added redundancy and fault tolerance capabilities based on monitoring and logging. Further along, another research group works with another example of data intensive computations based on parallel matrix multiplications (PMM) [11]. They devise fault tolerance models to support the processing of the PMM across multiple clouds. Approaching fault tolerance as part of a middleware framework is another possibility where the APIs for developing data intensive applications can be made to support fault tolerance. In [8] the authors introduce a new set of APIs to provide programmer-declared reduction objects. The fault tolerance is achieved by introducing periodical caching of the objects states on remote nodes which can later be used for recovery. Another middleware approach is described in [28] where fault tolerance is provided through LLFT middleware for data intensive applications executed in a cloud or data center environment. Fault tolerance is implemented based on a leader follower replication model where several replica groups are ordered in a hierarchy.

Generally, the Cloud offers a great environment to use the generic techniques and the Cloud-specific models. Since most resources and data centers rely on replication and redundancies, it becomes feasible to make use of that effectively. However, all proposed techniques involve a considerable level of overhead imposed by the features requirements. For example, checkpointing requires constant monitoring and exchange of status information, caching also requires storage capabilities and time to perform the task and monitor the cached data, while when replication is used high coordination efforts are needed to maintain consistency and orchestrate data transfers across the replicas. In the Cloud, the main

problem with any model that requires coordination or checkpointing is the communications delay. Trying to stay on top of things requires exchanging several messages to monitor and control the environment. However, this would take a long time given the large geographical distances covered. Consequently, trying to use these techniques in the cloud will magnify the problems and increase the overhead imposed due to the highly heterogeneous and distributed resources in use.

III. FAULT-TOLERENT ALGORITHM

The algorithm we propose to a specific set of problems with well defined properties. The main goal of this algorithm is to introduce efficient fault-tolerance and load balancing with minimal communication and coordination overhead while executing services in parallel over shared and dynamic heterogeneous distributed cloud infrastructures. We introduced a specialized version of this algorithm in [2], where it is designed to parallelize file downloads from replicated servers and provide efficient load balancing. Here we generalize the model for different types of problems and add the fault-tolerance feature. However, these problems should fit within a specific set of requirements. The main requirements are: easily parallelizable services; in a well defined geometric form; with well-defined boundaries; in addition to having replicated resources. Examples of such problems/services include large data set transfers, embarrassingly parallel computational problems like PMM and highly repetitive independent tasks. In addition, we extend the algorithm covers fault-tolerance and highly distributed environments with non-dedicated heterogeneous resources and operating conditions such as the Cloud.

A. *The Dual-Server Model*

Assuming we have two replicated servers, the client will need to obtain the problem properties and replica locations then divide the problem into multiple sub-tasks (blocks). Using control messages *Start*(*taskName*, *blockSize*, *firstBlock*, *counterMode)* and *End(taskName)* the client will get the servers started on processing and stop them when all blocks results are received. As each server starts from a different direction, they will both move towards each other and continue until the client tells both to stop when intermediate blocks overlap (See Figure 1).

To make sure all blocks are processed regardless of failure, the client keeps a current status record (CSR) to keep track of the progress of each server. This record includes for each server the *serverName*; the *LastBlock*, which is number of the last block received; the *counterMode*, which indicates the direction of process; the *partnerName*, which is the name of this server's partner (the other server in the pair); *totalBlocks*, which is the total number of blocks received from this server; *receiveTime*, which specifies the time when the last block was received; and the *serverStatus* which indicates that the server is available (*null*) or failed (*one*). A *cutOff* value can be set to help identify failed servers, which can either be fixed or dynamically adjusted during run time to reflect the current server's performance. All servers start with *serverStatus null* indicating they are all available. The *serverStatus* is changed to *one* due to two possible cases: (1) when the time since receiving the last block has exceeded the *cutOff* time, or (2) when the network fails and the TCP connection is dropped. In the dual-server case the last two parameters are relatively unnecessary; however, as we will explain the k-server case, they are important to maintain fault-tolerance with multiple failures.

As both servers are processing the client will need to keep track of received blocks to ensure all are received before requesting the servers to stop. This may result in some overlap in processing some blocks; however, this will also ensure that the client will not prematurely send the *End* messages. Therefore, if one server fails at some point the other one will pick up the slack without even knowing of the failure. In addition, this way all blocks will be processes even if one of the servers fails from the very beginning of the processing. In that case, the other server will simply do all the work. As a result there is no time wasted to discover the server failure and find an alternative for it as in many other fault-tolerance methods. Using the CSR and allowing the overlap to ensure delivery will guarantee that the problem at hand will be completed as long as there is one server still alive.

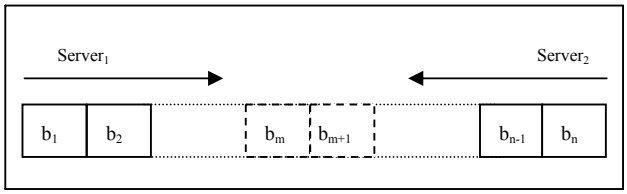

Figure 1. Block processing and downloading directions by dual-server case.

The implementation of the servers is relatively simple; each is multithreaded to handle multiple requests. In addition, each will simply execute the *Start* and *End* messages received from the client following the attributes included in these messages. On the client side, more control is needed thus the implementation is a bit more complex. The client is multithreaded such that it could manage the two servers and keep track of incoming blocks. When a client needs a task, it first finds the task information including the task size and the addresses of the servers it is available on. It will then prepare the *Start* messages and initialize the *CSR* for each server. During processing the client receives blocks and updates the *CSR*s, while keeping a lookout for the overlap in blocks received so it would send the *End* messages. Any extra arriving blocks after that will only overwrite their earlier version so the client does not need to verify duplicates. As a result, the client will ensure that it has received all blocks even in the presence of failure. There is also no need to add extra controls in the servers to compensate for failures.

B. *The k-Server Model*

The proposed algorithm in the dual-server case offers best possible load balancing in addition to fault-tolerance provided that at any point one server is working. The general technique for multiple servers still maintains efficient load balancing and affords multiple failures as long as at least one server is still working. Assume we have *k* servers and for clarity assume *k* is even. The servers are divided into *k/2* pairs. In addition, the requested task is divided into *n* equal-sized blocks, which are grouped into *k/2* partitions as

shown in Figure 2. Each pair will process one partition as described for the dual-server case. However, the difference will occur when pairs start finishing their work at different times, where a pair that finishes early is assigned to help other pairs in their partitions. That is if one pair completes its partition, we call it a *freePair* while a pair that did not finish its partition is called a *busyPair*. To provide the help, the partition of the *busyPair* will be divided into two partitions, *left*Part and *right*Part. And the servers of the *freePair* will be split such that each server will be paired with one of the servers in the *busyPair*. The new servers in each partition will start working from the new assigned point, while the original servers from the *busyPair* will continue their original work without any interruption or changes (see Figure 3). This process is repeated until there are no more partitions needing help. All this is controlled by the client using the *Start* and *End* control messages. For the case of an odd number of servers (*k* is odd), *k-1* servers are organized in pairs as described above and the last server spawns two threads and each is assigned a separate TCP connection such that the two threads of a virtual pair will work like the other pairs.

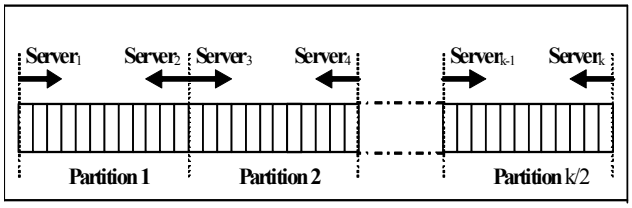

Figure 2. k-server case.

As in the dual-server case, fault-tolerance is achieved in this technique by allowing the servers in a pair to slightly overlap their work such that the client does not issue the *End* messages to any pair until it has completely received all required blocks in the partition. Therefore, if any server from one pair fails, the other one will continue sending blocks of the partitions until it reaches the point where the other server in the pair has stopped. Incase both servers in a pair fail, the client will be aware of the last block numbers received, thus when another pair becomes free, it will be assigned to pick up the work left out by the failed pair (as it is treated as a busy pair). In this case as soon as the two new servers start working with the failed pair, block will start reaching the client and the failed server in each of the newly formed pairs will be identified and treated accordingly. Thus, regardless where the failure occurred, we do not need to know about it, but we will just notice some degradation of the performance of the pair with failed server (or two). This degradation is directly proportional to the amount of remaining work to be done by the surviving server in the pair or the newly assigned pair to help the failed pair. However, the client, using the *receiveTime* value in the CSR can assume a failure if the time since the last received block is higher than the cutoff value. In that case it changes the *serverStatus* to *one* to indicate that this server is not active. This information can be used at the reassignment stage to decide what to do with the pair.

The technique allows the completion of the problem even if only one server remains alive. Generally, we can assume single or multiple failures in the system; therefore, we must compensate in different ways. If a failure occurs in one server in a pair, the other server will continue until the partition is finished. This is true if several servers, yet only one in any given pair fails. If a pair with a failed server (as indicated by the *CSR* information) becomes a *freePair* while there are still some *busyPairs*, we need to adjust the reassignment process. First, the client will try to restart the failed server, if the fault was transient. If the server restarts, the client updates the *serverStatus* in the *CSR* and the pair is reassigned normally to help one of the *busyPairs*. If the failed server does not restart, then the client splits the active server to form a virtual pair and reassign it to help a *busyPair*. However, if at the same time another *freePair* is available with a similar situation, the two active servers from both pairs are grouped to form a new *freePair*. The new *freePair* is then assigned to help a *busyPair*. The second possible failure case is when both servers in a pair fail. In this case the partition will not be completed and the pair will remain a *busyPair* until another pair becomes a *freePair* and is reassigned to help this pair. Here if multiple *busyPairs* are available, the *freePair* is assigned to help the slowest *busyPair*. This will allow us to single out a *busyPair* where there is a possible failure either in one or both of its servers. As the reassignment process continues, we will help all *busyPairs* until all partitions are completed.

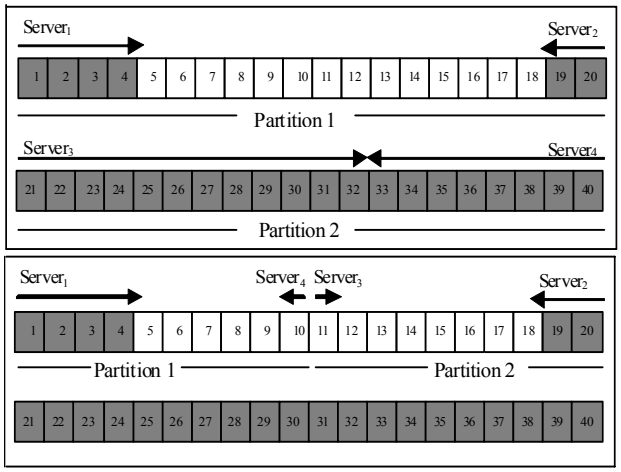

Figure 3. Top: a snapshot of four servers processing two partitions, the second partition is completed by $Server_3$ and $Server_4$ while the first partition is still being processed. Bottom: the rearrangement of partitions and Servers when $Server_3$ and $Server_4$ start helping $Server_1$ and $Server_2$. The $Server_3$ and $Server_2$ pair gets more blocks to process based on the partitioning heuristics.

IV. Example Applications and Experiments

Here we provide experimental evaluations and demonstrate the proposed fault-tolerance algorithm.

A. Data Service

For this set of experiments we use a modified version of DDFTP [2]. We introduced the fault tolerance feature and used that to evaluate the proposed algorithm. We evaluated and compared the results with a modified version of Dynamic Adaptive Data Transfer Model (DADTM) [27] to match the operations of

our algorithm and facilitate the comparisons we added a fault-tolerance feature to DADTM such that the adaptation process takes into consideration the discovery of faults after each rescheduled monitoring round.

The experiment was conducted to measure the performance with fault-tolerance of DDFTP and DADTM. Server permanent faults were created to the first server after 20, 40, 60, and 80 seconds from the beginning of the file transfer. We experimented with fault-tolerance on a WAN using an emulator that mimics WAN features such as high round trip time (RTT) delays. We used 0.240 second RTT, which is close to the RTT for a signal traveling between USA and Europe. Figure 4 shows the result when the WAN is used and the superiority of DDFTP becomes obvious with the existence of long delays. These delays make it harder for DADTM to reflect environment changes and faults quickly in the load balancing steps and also makes it take longer to discover faults and compensate. Therefore, the overall performance of DDFTP is better than DADTM in all cases.

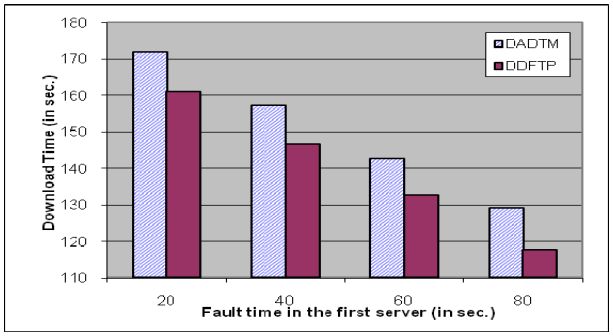

Figure 4. Fault-Tolerance Performance in WAN.

Another experiment was conducted to further illustrate the effect of long RTT delays on the performance of both DDFTP and DADTM with the existence of faults. A 500MB file was also used for this experiment. All experiments started with two servers and ended with one server. The server fault was intentionally created after around 40 seconds of starting the download process. The results are shown in Figure 5, where DDFTP download time does not increase much as RTT increases. However, with DADTM the file download time is significantly increased as RTT increases. This is due to the adaptive algorithm used in DADTM which needs to collect information from the servers. This becomes harder to collect and takes longer as the RTT value increases. For example a faulty server or network will take more time to be discovered if there are long RTT delays in the environment. This makes the adaptive algorithm less effective for load balancing as well as for fault discovery and resolution. In DDFTP, both the load balancing and the fault-tolerance mechanisms are naturally inherited in the download process as both servers download from different ends. This dual-direction process eliminates the need for collecting information about the servers and their associated networks and also eliminates the need to continuously monitor the servers' changes during the download. Therefore, we can provide a fast fault-tolerant mechanism that compensates for the faults without even having to discover them. Therefore, as the RTT increases, the download time will not increase much.

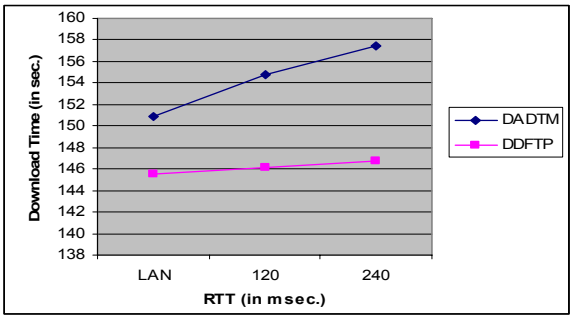

Figure 5. Impact of RTT on downloads with server faults.

Another set of experiments was conducted for the k-server case. In the experiments, 8 servers were used and the file size was 500MB. In the first experiment, the impact of multiple faulty servers was measured using DDFTP and DADTM. Server faults in up to half the number of servers were intentionally enforced after around half the no-fault download time has passed (that is the first half of the download is always fault free). An RTT delay of 240ms was used. The result is shown in Figure 6. Both DDFTP and DADTM can handle multiple faults; however, DDFTP provides better handling and better performance in all cases.

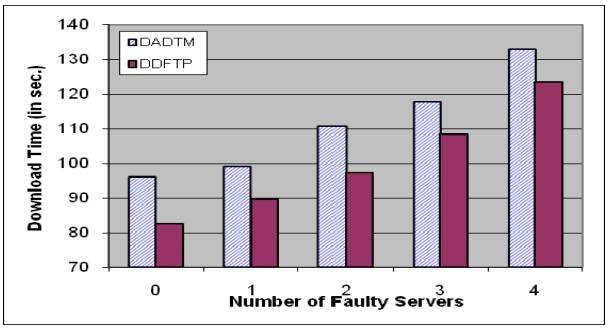

Figure 6. Impact of multiple server faults.

B. *Computation Services*

In this section, we will evaluate the proposed algorithm for computation services through parallel PMM. Several PMM algorithms are available [1] [9] [13] [18]. Beaumont & Boudent [6] developed a PMM algorithm for dedicated heterogeneous environments. This algorithm aims to enhance load balancing for heterogeneous environments and reducing communication overhead. The algorithm works well as long as the environment's operational conditions do not change during run time. However, it is not suitable for shared heterogeneous environments with some possibility for server or network faults because it cannot change the load distribution during run time. Alonso [5] experimented with different implementations of PMM on heterogeneous clusters of multi-core processors to find the best scheduling strategy. Al-Jaroodi [3] experimented with PMM using the master/slave model on heterogeneous clusters where the matrices are divided into small matrices that are distributed among available slaves based on their dynamic

speed. These and other approaches rely on some knowledge either prior to or during run time to adjust the load or adopt the master/slave model to divide the task into smaller partitions to be executed among the slaves. In both cases, a sort of load balancing may be achieved but at the expense of high overhead. Therefore, in our approach we limit the overhead while achieving efficient fault tolerance and good load balancing.

We utilize the proposed algorithm for PMM in the Cloud as shown in Figure 7, which represents the result matrix. The result matrix is divided into blocks of equal size and each block consists of a number of cells to be calculated for the result matrix. Each server in the system is considered a resource that can be used to solve a part of the problem. In the dual-server case, one server will start calculating the result blocks starting from the first block (top left corner) while the second server will start from the last block (bottom right corner). The client will ask the servers to stop processing and sending results as soon as it has the full matrix results as explained in Section III. If the resulting matrix is of size *m* by *p* and each block consists of *l* cells, then there are *(m*p)/l* blocks in the result matrix. In the k-server case the result matrix is divided into *k/2* partitions and each pair of servers works on one. To evaluate the algorithm on PMM, four multi-core machines were used with the specifications listed in Table 1.

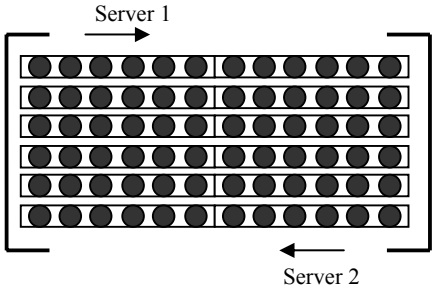

Figure 7. Result matrix with two servers.

These machines were connected by a dedicated LAN using Dell 2324 Fast/Gigabit Ethernet switch. A number of experiments were conducted to evaluate the load-balancing and fault-tolerance mechanisms.

TABLE 1. MACHINE SPECIFICATIONS.

Machine	Specifications
M1	Desktop, Microsoft Windows XP Professional, Intel® Core 2 CPU 6400 @ 2.13GHz, 3.00 GB of Ram, Gigabit Ethernet NIC
M2	Laptop, Microsoft Windows XP Professional, Intel® Core 2 Duo T7300 @ 2.00GHz, 0.99 GB of RAM, Gigabit Ethernet NIC
M3	Laptop, Microsoft Windows 7, Intel® Core i5 CPU M450 @ 2.4 GHz, 4.00 GB of RAM, Fast Ethernet NIC
M4	Laptop, Microsoft Windows 7, Intel® Core i7CPU Q740 @ 1.73GHz, 6.00 GB of RAM, Fast Ethernet NIC

The first set of experiments was conducted to evaluate the performance of the algorithm without faults in the k-server case (k = 4 and k = 8) on a LAN and a WAN with 0.400 seconds RTT (see Figure 8) using 2000x2000 matrices. M4 was used as a client. The experiment was conducted for both sequential and parallel processing. For sequential processing, the performance of each individual machine was measured for processing the PMM. These are indicated by Mat(M1), Mat(M2), Mat(M3), and Mat(M4) in Figure 8. One thread server was used from each of the machines for the 4-server case, DDPar(4), and two thread servers were used from each of the machines for the 8-server case, DDPar(8). The speedup is 4.38 for DDPar(4) and 7.9 for DDPar(8) with respect to the slowest machine M3; and 3.66 for DDPar(4) and 6.59 for DDPar(8) with respect to the fastest machine M1. The performance results of DDPar(4,400) and DDPar(8,400) on the WAN with an average RTT of 0.400 seconds slightly increases compared to the LAN's results. This good result is mainly due to the minimization of the coordination messages in DDPar. Eq-Part(4) is the result when the matrix is divided into four equal parts each is solved by one server thread from the four machines. In addition, Eq-Part(8) is the result when the matrix is divided into eight equal parts each two solved by two server threads from the four machines. As we can see, DDPar provides better performance due to the dynamic load balance mechanism that is inherited in the proposed algorithm.

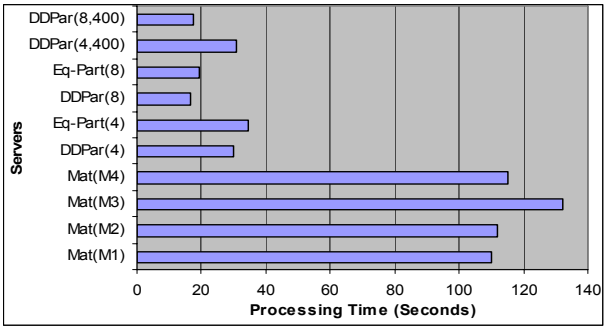

Figure 8. DDPar performance of PMM, k-server case.

Another set of experiments was conducted to compare the performance of PMM using DDPar and the M-Blocks master/slave approach where the client divides the matrices into multiple blocks. These blocks are assigned by the client to the server each time they finish from a block. In this case the faster servers will solve more blocks, but will communicate more with the master. The experiment was conducted between the client and two servers while an emulated WAN was embedded between them to measure the effect of long delays. The aim of this experiment is to evaluate the effect of long RTT on the performance of both DDPar and M-Blocks. As shown in Figure 9, as the RTT increases the processing time in DDPar slightly increases, while it increases significantly in M-Blocks. This is due to the cost of coordination in the M-Blocks approach, where servers need to go back to the client every time they finish a block to get the next one. The cost of coordination in DDPar is kept at minimum since the servers work independently so they are less affected by the delays. Although, this experiment shows that B-Blocks approach provides good performance for LAN as well as it can easily cover server or network faults, it can not be used for WAN. This is mainly due to the high-cost of coordination inflected by the high delays between the distributed servers and the client.

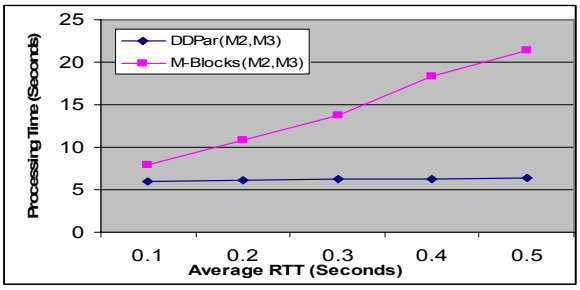

Figure 9. Effects of long RTT in PMM.

The third experiment was conducted to evaluate the performance of the proposed algorithm with the existence of faults. The experiment was done for both LAN and WAN with an average of 400 milliseconds between the servers and the client. 2000x2000 PMM was used over M1, M2, M3 and M4 machines. The experiment was conducted without any faults, with M2 fails at 15 sec. from the starting of the process, and with both M2 & M3 fail at 15 sec. from the starting of the process. The results, in Figure 10, show that the proposed algorithm provides good fault tolerance in WAN with little overhead. Although there is around 400 milliseconds delay between the sever and the client, there is only around 2.7% increase in the processing time between fault tolerance handling in WAN compared to LAN. This is because the algorithm provides efficient fault tolerance over WAN without much overhead. This makes the algorithm very useful for highly distributed replicated Cloud services.

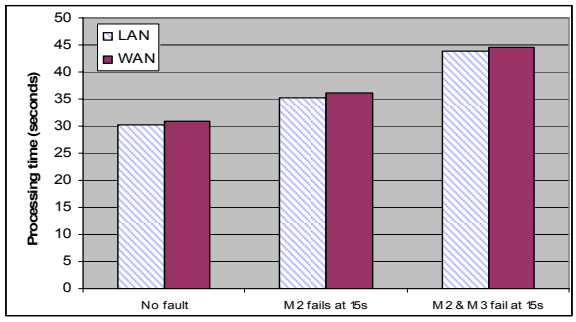

Figure 10. Impact of faults in LAN and WAN with 400 msec RTT.

V. Conclusion

Services in the Cloud can be replicated over multiple servers distributed over large geographical areas with high communication delays among them. One of the main challenges in this case is how to utilize these replicated services for enhancing the performance of cloud applications. This paper proposed an efficient algorithm that provides fault-tolerance and load balancing mechanisms that utilize existing replicated cloud services to enhance availability and performance of the service provided. This algorithm is delay-tolerant in which it can provide fault-tolerance and load balancing even with high and dynamic delays in the communication among the replicated service and the client. While this algorithm offers efficient fault-tolerance, it minimizes the coordination and synchronization efforts needed to implement the fault discovery and handling feature. This is done by dividing the work among the servers grouped in pairs such that each pair will work on its part from opposite directions. This allows each server to continue its work without having to consult with the client or other servers. In addition, as each server in a pair works towards the other, the load will be automatically evenly distributed among the two based on their speed and they will both finish at the same time. In case one server fails, the other will continue processing the rest of the subtasks with minimal coordination effort.

References

[1] Agarwal, R., F. Gustavson, and M. Zubair, "A High Performance Matrix Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication," IBM J. Research and Development, vol. 38, no. 6, pp:673-681, 1994.

[2] Al-Jaroodi, J. and N. Mohamed, "DDFTP: Dual-Direction FTP," in Proc. of 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011), May 2011.

[3] Al-Jaroodi, J., N. Mohamed, H. Jiang, and D. Swanson, "Middleware Infrastructure for Parallel and Distributed Programming Models on Heterogeneous Systems," in The IEEE Transactions on Parallel and Distributed Systems – Special Issue on Middleware Infrastructures, 14(11), pp: 1100-1111, November 2003.

[4] Allcock, B., J. Bester, J. Bresnahan, AL. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, I. Foster, "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing," in proc. IEEE 18th Symposium on Mass Storage Systems and Technologies, San Diego, CA, USA, April 2001.

[5] Alonso, P., R. Reddy, A. Lastovetsky, "Experimental Study of Six Different Implementations of Parallel Matrix Multiplication on Heterogeneous Computational Clusters of Multicore Processors," in proc. 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010.

[6] Beaumont, O., V. Boudet, "Matrix Multiplication on Heterogeneous Platforms," IEEE TPDS, 12(10), pp:1033-1051, 2001.

[7] Bhardwaj, D. and R. Kumar, "A Parallel File Transfer Protocol for Clusters and Grid Systems," in proc. 1st International Conference on e-Science and Grid Computing, 2005.

[8] Bicer T., W. Jiang, and G. Agrawal, "Supporting Fault Tolerance in a Data-Intensive Computing Middleware," in proc. IEEE Int'l Symposium on Parallel Distributed Processing, pp:1-12, April 2010.

[9] Blackford, L.S., J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, "ScaLAPACK Users' Guide," SIAM, 1997.

[10] CloudLeverage: Global LB, URL: http://cloudleverage.com/global-load-balancing/, viewed November 2011.

[11] Deng, J., S. C-H. Huang, Y. S. Han, and J. H. Deng, "Fault-Tolerant and Reliable Computation in Cloud Computing," in proc. IEEE GLOBECOM Workshops, Miami, FL, USA, pp:1601-1605, December 2010.

[12] Fedak, G., H. He and F. Cappello, "BitDew: A data management and distribution service with multi-protocol file transfer and metadata abstraction," in The Journal of Network and Computer Applications, Vol. 32, No. 5, pp:961-975, September 2009.

[13] Fox, G., et. al., "Matrix Algorithms on a Hypercube i:Matrix Multiplication," Parallel Computing, vol. 3, pp: 17-31, 1987.

[14] Funasaka, J., A. Kawano and K. Ishida, "Implementation Issues of Parallel Downloading Methods for a Proxy System," in proc. 4th

International Workshop on Assurance in Distributed Systems and Networks, ICDCSW'05, vol. 1, pp: 58 - 64, 2005.

[15] GigeNET Cloud Servers, URL: http://www.gigenetcloud.com/web_load_balancing_and_scalability.html, viewed Nov. 2011.

[16] Grance, T., PM. Mell, "The NIST Definition of Cloud Computing," NIST Special Publication 800-145, September 2011. http://www.nist.gov/customcf/get_pdf.cfm?pub_id=909616 and http://www.nist.gov/manuscript-publication-search.cfm?pub_id=909616, viewed November 2011.

[17] Ko, SY., I. Hoque, B. Cho, and I. Gupta, "Making cloud intermediate data fault-tolerant," in proc. 1st ACM symposium on Cloud computing (SoCC), ACM, New York, NY, USA, 2010.

[18] Kumar, V., A. Grama, A. Gupta, and G. Karypis, Introduction to Parallel Computing. Benjamin/Cummings Inc., 1994.

[19] Li, J., M. Humphrey, Y-W. Cheah, Y. Ryu, D. Agarwal, K. Jackson, and C. van Ingen, "Fault Tolerance and Scaling in e-Science Cloud Applications: Observations from the Continuing Development of MODISAzure," in proc. IEEE e-Science 2010 Conference, Brisbane, Australia, December 2010.

[20] Lim, SB.,G. Fox, A. Kaplan, S. Pallickara and M. Pierce, "GridFTP and Parallel TCP Support in NaradaBrokering," in Distributed and Parallel Computing, Lecture Notes in Computer Science, Vol. 3719/2005, pp: 93 - 102, 2005.

[21] RackSpace load balancer, URL: http://www.rackspace.com/cloud/cloud_hosting_products/loadbalancers/, viewed Nov. 2011.

[22] Ramakrishnan, L., D. Nurmi, A. Mandal, C. Koelbel, D. Gannon, TM. Huang, Y-S. Kee, G. Obertelli, K. Thyagaraja, R. Wolski, A. YarKhan, and D. Zagorodnov, "VGrADS: Enabling e-Science Workflows on Grids and Clouds with Fault Tolerance," in proc. ACM/IEEE Int'l Conference on High Performance Computing and Communication (SC09), Portland, Oregon, USA, Nov. 2009.

[23] Rao, G.N. and S. Nagaraj, "Client Level Framework for Parallel Downloading of Large File Systems," in International Journal of Computer Applications, Vol. 3, No. 2, pp:0975–8887, June 2010.

[24] RightScale, "Load Balancing in the Cloud: Tools, Tips, and Techniques," white paper, RightScale, 2010, URL: http://www.rightscale.com/ and http://www.rightscale.com/lp/load-balancing-in-the-cloud-whitepaper.php, viewed Nov. 2011.

[25] Vazquez-Poletti, JL., "Toward a Fault-Tolerant Cloud," article, HPC in the Cloud, June 2011, URL: http://www.hpcinthecloud.com/hpccloud/2011-06-23/toward_a_fault-tolerant_cloud.html, viewed Nov. 2011.

[26] Venkatesh, KA., et. al., "Using MapReduce and load balancing on the Cloud," white paper, DevelopersWorks, IBM, July 2010, URL: http://www.ibm.com/developerworks/cloud/library/cl-mapreduce/index.html?ca=drs-, viewed Nov. 2011.

[27] Zhang, Q. and Z. Li, "Data Transfer Based on Multiple Replicas in the Grid Environment," in proc. 5th Annual ChinaGrid Conference, pp:240–244, 2010.

[28] Zhao, W., P. M. Melliar-Smith and L. E. Moser, "Fault Tolerance Middleware for Cloud Computing," in proc. IEEE 3rd International Conference on Cloud Computing, pp:67-74, Miami, FL, USA, July 2010.

[29] Zheng, Q., "Improving MapReduce fault tolerance in the cloud," in proc. IEEE International Symposium on Parallel Distributed Processing Workshops and PhD Forum, pp:1-6, Atlanta, GA, USA, April 2010.