Journal of Computer Science 7 (2): 159-166, 2011
ISSN 1549-3636

A Robust Byzantine Fault-Tolerant Replication Technique for Peer-to-Peer Content Distribution

[1]Ayyasamy Sellappan and [2]Sivanandam Natarajan
[1]Department Information Technology, Tamilnadu College of Engineering,
[2]Department Computer Science and Engineering, PSG College of Technology,
[1,2]Anna University of Technology, Coimbatore, Tamil Nadu, India

Abstract: Problem statement: In peer-to-peer networks, Byzantine fault tolerance refers to the capability of a system to tolerate Byzantine faults. It can be achieved by replicating the server and by ensuring all server replicas reach an agreement on the input despite Byzantine faulty replicas and clients. Since malicious attacks and software errors can cause faulty nodes to exhibit Byzantine behavior, Byzantine-fault-tolerant algorithms are increasingly important. **Approach:** In the study, we wish to develop a robust Byzantine Fault-Tolerance Replication (BFTR) technique for peer-to-peer content distribution systems which contains fault detection and fault recovery. It is based on collaborative monitoring of each node to detect the occurrence of a fault. Already we proposed a QoS based overlay network architecture (QIRM) involving an intelligent replica placement algorithm to improve the network utilization of the P2P system. **Results:** By simulation results, we show that the proposed technique involves less overhead and recovery time with increased accuracy. **Conclusion/Recommendations:** Here the result obtained is that BFTR Technique is much efficient than the QIRM with respect to packet drop ratio, average end-to-end delay, throughput and overhead.

Key words: Fault-tolerance, Internet Protocol (IP), replication technique, content distribution, Byzantine Fault Tolerance (BFT), Peer to Peer (P2P), Origin Server (OS)

INTRODUCTION

A peer-to-peer, commonly abbreviated to P2P, is any distributed network architecture composed of participants that make a portion of their resources such as processing power, disk storage or network bandwidth directly available to other network participants, without the need for central coordination instances such as servers or stable hosts. Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model where only servers supply and clients consume. Peer-to-peer systems often implement an Application Layer overlay network on top of the native or physical network topology. Such overlays are used for indexing and peer discovery. Content is typically exchanged directly over the underlying Internet Protocol (IP) network (http://en.wikipedia.org/wiki/Byzantine_fault_tolerance). Anonymous peer-to-peer systems are an exception and implement extra routing layers to obscure the identity of the source or destination of queries.

Fault detection is a subfield of control engineering which concerns itself with monitoring a system, identifying the fault when it occurs and pinpoint the type of fault and its location (Simon, 2009). If the detector determines that some node has become faulty, it notifies the application software, which can then take appropriate action. For example, nodes can cease to communicate with the faulty node; once all correct nodes have followed suit, the faulty node is isolated and the fault is contained.

Fault detection is insufficient for dealing with faults that have serious and irreversible effects, such as deletion of all copies of an important document. However, detection may offer an efficient and scalable alternative to Byzantine Fault Tolerance (BFT) for faults that have limited or recoverable effects, including freeloading, censorship and denial-of-service. Using fault detectors, each action is undeniably associated with the identification of the node that has performed the action, allowing the system to gather irrefutable evidence of faulty behavior. The fault detection systems we consider should guarantee at least two properties. The system should be complete: whenever a correct node observes the effects of faulty behavior, the system

Corresponding Author: Ayyasamy Sellappan, Department Information Technology, Tamilnadu College of Engineering, Anna University of Technology, Coimbatore, Tamil Nadu, India

eventually generates evidence against at least one faulty node. Also, the system should be accurate: it never generates valid evidence against a correct node (Petr and Druschel, 2006).

Byzantine Fault Tolerance (BFT): In a distributed system, an arbitrary fault occurs during the execution of an algorithm which is known as the Byzantine fault. Both omission failures like crash failures, failing to receive a request, or failing to send a response and commission failures like processing a request incorrectly, corrupting local state and/or sending an incorrect or inconsistent response to a request are included in the Byzantine fault (Petr and Druschel, 2006). The potential of a system to tolerate Byzantine faults is the Byzantine fault tolerance. By replicating the server and ensuring all server replicas reach an agreement on the input in spite of Byzantine faulty replicas, Byzantine fault tolerance can be achieved. This agreement is known as Byzantine agreement (Zhao, 2007). Importance of Byzantine-fault-tolerance algorithms is increasing since faulty nodes are caused by malicious attacks and software errors which can exhibit Byzantine behavior (Castro and Liskov, 1999).

Though some part of replicas fail, a service replicated over several BFT servers are able to survive and so affecting an overall application security has a very small probability. BFT replication can be protected against malicious or incompetent machine operators, only when replicas are separately administered.

Linearizability and liveness are the two main properties of BFT protocols. In Linearizability, the service appears to all clients for executing a sequence of requests which preserves the temporal order of non-concurrent operations. Improvement in executing client's requests in a system with at-least some weak assumption about eventual message delivery is known as Liveness (Li and Mazieres, 2007). As more faults are tolerated, the performance of the Byzantine fault-tolerant agreement-based approaches drops rapidly due to server-to-server broadcast communication and the requirement that all correct servers process every request (El Malek *et al*., 2005).

Byzantine fault tolerant replication scheme must meet the following objectives:

- Ensure that all non-faulty replicas have equivalent logical state such that they will return the same answer to any given query
- Ensure that the client always gets correct answers to queries belonging to transactions that commit, even when up to f replicas are faulty
- Detect faulty replicas and flag them for repair

Proposed work: We wish to develop a fault-tolerance replication technique for peer-to-peer content distribution systems which contains fault detection and fault recovery. It is the extension of our previous study (Ayyasamy and Sivanandam, 2009) which presents a QoS based intelligent replica placement algorithm.

Two fundamental classes of replication techniques ensure linearizability: primary-backup and active.

This primary Back-up Replication technique uses one replica, the primary that plays a special role: it receives invocations from client processes and returns responses. Server x's primary replica is denoted prim(x); other replicas are backups. Backups interact directly only with the primary replica, not the client processes.

The proposed technique involve less overhead and recovery time with increased accuracy. It is based on collaborative monitoring of each node to detect the occurrence of a fault.

Related work: Clement *et al*. (2009) have describe Aardvark, a new BFT replication protocol that guarantees good performance during uncivil periods, when the network is reliable but when up to *f* servers and any number of clients are faulty. Aardvark gives up some performance compared to protocols that focus on optimizing for the best case, but Aardvark's peak throughput of 40527 requests per second seems sufficient for many applications. Because Aardvark is less aggressively tuned for the fault free case, it is guaranteed to remain within a constant factor of 40527 when faults occur.

Zhijun and Minghong (2005) Evolutionary Overlay Service in Peer-to-Peer Systems can evolve the overlays based on many factors, such as peers' reliability; peers' capacity; peers' ability to judge the correctness of the information, etc. The evolved overlay can improve the system performance, which is proved by theoretical analysis and verified by experimental results.

Amir *et al*. (2010) have presented the first hierarchical Byzantine fault-tolerant replication architecture suitable to systems that span multiple wide area sites. The architecture confines the effects of any malicious replica to its local site, reduces message complexity of wide area communication and allows read-only queries to be performed locally within a site for the price of additional hardware. A prototype implementation is evaluated over several network topologies and is compared with a flat Byzantine fault-tolerant approach.

Kotla *et al*. (2007) have presented Zyzzyva, a protocol that uses speculation to re-duce the cost and simplify the design of Byzantine fault tolerant state machine replication. In Zyzzyva, replicas respond to a client's request without first running an expensive three-phase commit protocol to reach agreement on the order in which the request must be processed. Instead, they optimistically adopt the order proposed by the primary and respond immediately to the client. Replicas can thus be-come temporarily inconsistent with one another, but clients detect inconsistencies, help correct replicas converge on a single total ordering of requests and only rely on responses that are consistent with this total order.

Pathak and Iftode (2006) have described Byzantine Fault Tolerant Authentication, a mechanism for public key authentication in peer-to-peer systems. Authentication done without trusted third parties, tolerates Byzantine faults and is eventually correct if more than a threshold of the peers is honest. They addressed the design, correctness and fault tolerance of authentication over insecure asynchronous networks. An anti-entropy version of the protocol is developed to provide lazy authentication with logarithmic messaging cost.

Singh *et al*. (2009) have presented a novel BFT state machine replication protocol called Zeno that trades consistency for higher availability. In particular, Zeno replaces strong consistency (linearizability) with a weaker guarantee (eventual consistency): clients can temporarily miss each other's updates but when the network is stable the states from the individual partitions are merged by having the replicas agree on a total order for all requests.

Ayyasamy and Sivanandam (2010a) have proposed a cluster based replication architecture for load-balancing in peer-to-peer content distribution systems. In addition to an intelligent replica placement technique, it also consists of an effective load balancing technique. In the intelligent replica placement technique, peers are grouped into strong and weak clusters based on their weight vector which comprises available capacity, CPU speed, access latency and memory size. In order to achieve complete load balancing across the system, an intra-cluster and inter-cluster load balancing algorithms are proposed.

Ayyasamy and Sivanandam (2010b) have presented a trust based content distribution for peer-to-peer overlay networks, which is built on the trust management scheme. The main concept is, before sending or accepting the traffic, the trust of the peer must be validated. Based on the success of data delivery and searching time, they calculated the trust index of a node. Then the aggregated trust index of the peers whose value is below the threshold value is considered as distrusted and the corresponding traffic is blocked.

Previous work on replica placement:

System model and overview: In our QOS aware topology, nodes are grouped into strong and weak clusters based on their weight vector which comprises the following parameters:

- Available capacity
- CPU speed
- Memory size
- Access Latency

In the replica placement algorithm, we classify the content as Class I and Class II, based on their access patterns. (i.e.,) The most frequently accessed contents are ranked as Class I and the less frequently accessed contents as Class II. Then more copies of Class I content are replicated in strong clusters (having high weight values) (Ayyasamy and Sivanandam, 2009). Routing is performed hierarchically by broadcasting the query only to the strong clusters. Thus the proposed architecture achieved Low bandwidth Consumption, Reduced Latency, Reduced Maintenance Cost, Strong Connectivity and Query Coverage.

Let us consider a collection of N server nodes which forms a Peer to Peer (P2P) overlay network. In addition to being part of the overlay, each node functions as a server responding to requests (queries) which come from clients outside of the overlay network. An example could be that each node is a web server with the overlay linking the servers and clients being web browsers on remote machines requesting content from the servers.

We assume each node always stores one copy of its own content item which it serves to clients and that it has additional storage space to store k replicated content items from other nodes which it can also serve (Hales *et al*., 2007). The object is associated with an authoritative Origin Server (OS) in the network where the content provider makes the updates to the object. The object copy located at the origin server is called as origin copy and the object copy at any remaining server is called a replica.

Intelligent replica placement algorithm:

Clustering the node:

For each node N_i, $i = 1,2.....n$, let

Where:

BW_i = Available bandwidth
SP_i = CPU speed
AL_i = Access latency
MZ_i = Memory size

The weight of the node Ni can be calculated as:

$$W_i = (BW_i + SP_i + MZ_i) / AL_i$$

Form the vector W = {S_i, W_i}, which denotes the node ids and their corresponding weight values, sorted on the descending order.

Let {Sk} denote the set of strong cluster nodes (0<=k<n), which satisfies the following condition $W_k \geq \beta$, where β is the minimum threshold value for the weight.

Then the set {W_j} = {N_i} – {S_k}, denote the set of weak cluster nodes (0<=j<n), which satisfies the condition $W_k < \beta$.

Replica placement: Let QS be the query server which registers the query of each client. The query server stores the cluster information of each node along with the node id as "S" or "W" for strong and weak clusters, respectively:

- At time Tk, let m clients generates query requests {Qm} of the form q{nid, ckwd}, where nid is the node id of the client and ckwd is the keyword of the content to be retrieved
- The queries {Qm} are registered in the query server QS
- The requested content of the queries are classified and categorized as Class 1 or Class 2, depending on the access frequencies:
  (i.e.) A query Q_j, j<m, is considered to be Class 1
  If n (Q_j) >= A_{min} and Class 2,
  If n (Q_j) < A_{min}
  Where:
  n(Q_j) = The no. of access of the content pattern for the given query
  A_{min} = The minimum access threshold value
- Then the query server QS assigns the class1 contents to the strong cluster nodes and class2 contents to the weak cluster nodes
- After the assignment, QS transmit these replication pattern information to the origin server OS
- OS performs the replication placement, according to the pattern information obtained from QS. The weight value Wi of each node is stored along with the content
- OS then broadcasts the replication information to the respective clients in the following format:

{Nid, Clid ("S" or "W"), c1, c2 …}

Where:
N_{id} = The node id
C_{lid} = The cluster id and c1, c2, are content database ids

MATERIALS AND METHODS

Byzantine Fault Tolerance Replication (BFTR) Technique:

Architecture model: In our proposed System, clients do not interact directly with the database replicas. Instead they communicate with the agent, which acts as a front-end to the replicas and coordinates them. The agent is replicated for increased fault tolerance. The following Fig.1 shows the agent system architecture. The architecture requires 2n + 1 database replicas, where n is the maximum number of simultaneously faulty replicas the system can tolerate. It requires only 2n + 1 replica because the database replicas do not carry out agreement; instead they simply execute statements sent to them by the agents. The Fig.1 illustrates a system in which n = 1.

We assume that the agent itself is trusted and does not have Byzantine faults, though it might crash. Since the complexity and amount of code in the agent is orders of magnitude smaller than in the replicas, we believe that assuming non-Byzantine behavior is reasonable.

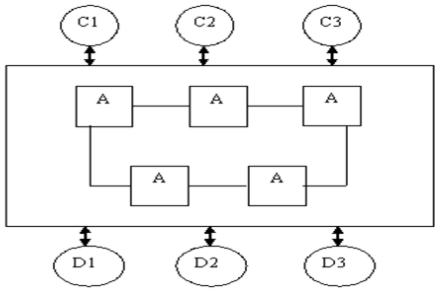

Fig. 1: Architecture

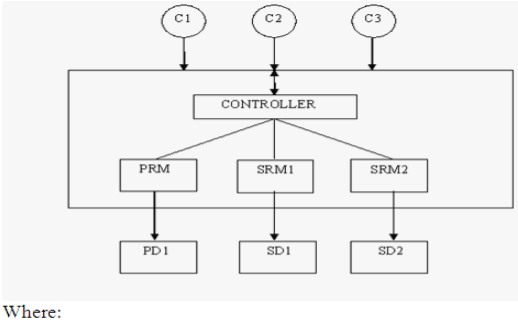

Fig. 2: Basic Design

Basic design and implementation: As shown in Fig. 2, the agent runs a single controller and one replica manager for each back-end replica. The controller receives statements from clients and forwards them to the replica managers. Replica managers execute statements on their replicas and send answers back to the controller. The controller sends results back to the clients, compares query answers for agreement and determines when it is safe for transactions to commit. The controller may also decide to abort and retry transactions or initiate repair of faulty replicas.

In BFTR technique, one replica is designated to be primary and runs statements of the transactions slightly in advance to the other secondary replicas. The order in which the transactions are completed on the primary determines a serial order. BFTR ensures that all the non-faulty secondary's commit transactions in an order equivalent to that at the primary. Furthermore, the agent achieves good performance by allowing queries to execute concurrently on the secondary's where it observes the queries executing concurrently on the primary. When the primary is non-faulty, the system performs well. A faulty primary can cause performance to degrade but BFTR maintains correctness.

If a non-faulty database replica completes executing two statements without an intervening commit then the two statements do not conflict.

By using this property, the control try to extract concurrency information by observing the primary execute transactions: if the (non-faulty) primary allows a set of queries to run without conflicting, then the secondary's can run these queries concurrently, or in any order, without yielding a different equivalent serial ordering from the primary. Execution schedules are generated by a faulty primary where conflicting statements execute concurrently.

Notations used: Global commit delay counters-D_c, Query-q, q's delayer-q_d, Commit-c_0, Replica delayer-r_d, Transaction-t, ABORT-A_{RT}, PR-Primary Replica, SR-Secondary Replica, PRM-Primary replica manager, SRM-Secondary replica manager, Query answer- q_a, Acknowledge-ack and Client-cl.

Commit delayer principle: We implement the commit-ordering and transaction-ordering rules using the commit delayer principle. The controller maintains a global commit delay counter, D_c and SRM maintains a replica delayer r_d.

The following steps are involved in the commit delayer principle:

1. If the controller receives a response to query q from PR, then:
   $q_d = D_c$
2. If the controller commits a transaction t, then:
   $t_d = D_c$
3. $D_c = D_c+1$
4. Send c_0 to the replica managers.
5. While $q_d \geq r_d$ (SRM waits to send query q to SR)
6. If $r_d = t_d$, then
   SRM commit transaction t
7. $r_d = r_d+1$
8. If SR has executed all queries of t
   It commits transaction t
9. $r_d = D_c$

The following algorithms provide the steps for handling Byzantine faulty primary replicas. A failure of primary replica is indicated by returning a wrong answer and it is handled by detecting that the client received an incorrect answer at the transaction commit point and aborting the transaction.

Algorithm for the controller:

1. If controller receives q from cl.
   1.1 Send q to PRM.
   End if
2. If controller receives response for q from PRM, then
   2.1 Send the response to cl.
   2.2 $q_d \leftarrow D_c$
   2.3 Record response as q_a.
   2.4 Send q to SRM.
   End if
3. If controller receives response for q from SRM, then
   3.1 Add response to votes (q).
   End if
4. If controller receives A_{RT} from cl, then
   4.1 Send A_{RT} to RM.
   4.2 Send ack to cl.
   End if
5. If controller receive c_0 for t from cl., then
   5.1 While f+1 replica is ready to commit t
      5.1.1 Delay processing c_0
   5.2 End While
   5.3 If the response q_a of cl does not contain f votes in votes (q), then
      5.3.1 Send A_{RT} to the RM
      5.3.2 Inform cl of the A_{RT}.
   5.4. Else
      5.4.1 $t_d \leftarrow D_c$;
      5.4.2 $D_c \leftarrow D_c + 1$.
      5.4.3 Send ack to cl.
      5.4.4 Send c_0 to RM.
   5.5 End if
6. End if

Algorithm for the SRM:

1. For each query q in the collection, determine
    1.1 All earlier queries from q's transaction have completed processing.
    1.2. $q_d \geq r_d$
    1.3. Execute each query q that is ready on the replica
    1.4. Send the result to the controller.
   End For
2. If all queries of t have completed processing at SR and
    $r_d = t_d$, then
    2.1 RM issue a c_0 to the replica
   End if
3. If c_0 completes processing, then
    $r_d = r_d + 1$
   End if
4. for each A_{RT} for a transaction t in the collection,
    4.1 discard any queries of t that have not yet been sent to the replica
   End For

Fault recovery:

Recovery of a crashed replica:

Step 1: The replica rejoins the system after recovery from offline. Transactions, which are not committed before the failure gets cancelled and these transactions must be repeated for updating the replica. Statements for recovering the replica are contained in the collection of the manager.

Step 2: When a database connection breaks, the replica manager considers that a replica had crashed. When the connection breaks, the manager sends a $c_0$to the replica and when no reply is sent back to the manger, the knowledge of processing c_0 before the replica went down can't be known. The transaction cannot re-run, if it's already committed.

Step 3: A transaction log table which consists of a row for each committed transaction with commit delayer for t_d, is added to each reply so that the replica manager determines the situation. The transaction log table contains an entry for the transactions committed before failure. The missing committed transactions can be re-run using the information from transaction log table.

If a connection gets re-established, the replica manger reads the table, compares the list of committed transactions with agent information and re-runs it. After processing all the committed transactions, the manger runs the in-progress transactions.

Recovery from agent crash:

Step 1: The log should be written prior so that crashes can be endured. When a controller determines that it can commit a transaction, it writes transaction queries along with the c_0 to the log and the controller forces the log to disk before replying to the cl. After the replica's transaction log table information is written, the log is also written to the disk and then the table is truncated.

Step 2: Crash can be recovered by the agent. It reads the log and identifies all committed transactions and initializes the statement collections at the managers to contain the statements of these transactions. The replica's transaction log table is examined by the controller and included in the replica manager's collection. When the primary replica implements all the statements in the collection, new cl transactions are accepted by the agent.

Step 3: The transaction information committed at all the replicas can be discarded so that the agents log can be shortened. Nearly all the replicas run properly so that there is no need of very large log.

RESULTS AND DISCUSSION

Simulation setup: Here we discuss about the experimental performance evaluation of our algorithms through simulations. In order to test our protocol, the NS2 simulator (http://www.isi.edu/nsnam/ns) is used. NS2 is a general-purpose simulation tool that provides discrete event simulation of user defined networks.

We have used the Bit Torrent packet-level simulator for P2P networks (Eger *et al*., 2007). The network topology is used only for the packet-level simulator. Based on the assumption that the bottleneck of the network is at the access links of the users and not at the routers, we use a simplified topology in our simulations. We model the network with the help of access and overlay links. Each peer is connected with an asymmetric link to its access router. All the access routers are connected directly to each other, modeling only an overlay link. This enables us to simulate different upload and download capacities as well as different end-to-end (e2e) delays between different peers.

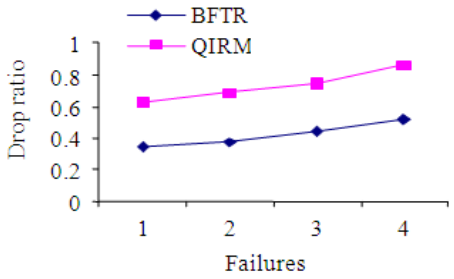

Fig. 3: No. of failures Vs Packet drop ratio

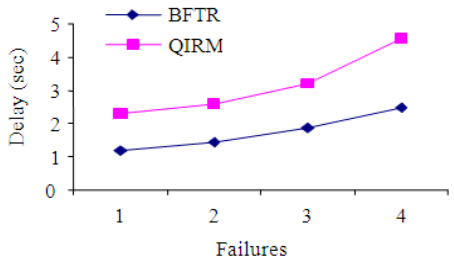

Fig. 4: No. of failures Vs end-to-end delay

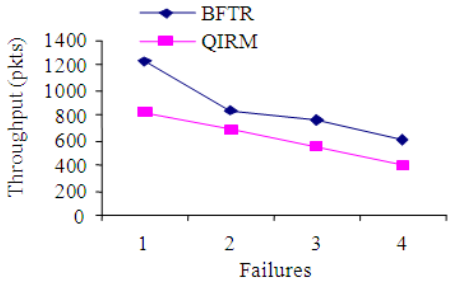

Fig. 5: No. of failures Vs packets received

Performance metrics: In our experiment, we measure the following metrics:

- Packet Drop Ratio: It is the ratio of number of dropped packets to the total number of packets sent
- Average end-to-end delay: The end-to-end-delay is averaged over all surviving data packets from the sources to the destinations
- Throughput: It is the number of packets received successfully
- Overhead: Routing and control overhead in terms of packets.

Simulation results: We have compared our BFTR architecture to our previous QIRM architecture (Ayyasamy and Sivanandam, 2009) with no fault-tolerant capabilities. In the experiment, we vary the number of failures of replicas as 1-4 and measure the above metrics.

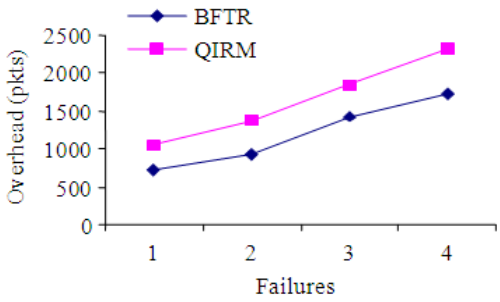

Fig. 6: No. of failures Vs overhead

Fig. 3 shows the packet drop ratio obtained with our BFTR technique compared with the QIRM technique. It shows that the packet drop ratio is significantly less than the default technique, when number of failures increases. Fig. 4 shows the end-to-end delay occurred for the number of failures. It shows that the delay of BFTR is significantly less than the QIRM technique.

Fig. 5 shows the throughput occurred for various failures. As we can see from the figure, the throughput is more in the case of BFTR when compared to QIRM technique.

The communication overhead is represented in Fig. 6. It shows that the overhead is comparatively less for BFTR technique than the QIRM technique.

CONCLUSION

In this study, we have developed a Byzantine Fault-Tolerance Replication (BFTR) technique for peer-to-peer content distribution systems which contains fault detection and fault recovery. Initially we have proposed a QoS based overlay network architecture involving an intelligent replica placement algorithm to improve the network utilization of the P2P system. In our BFTR technique, clients do not interact directly with the database replicas. Instead they communicate with the agent, which acts as a front-end to the replicas and controls them. The agent is replicated for increased fault tolerance. In this technique, one replica is designated to be the primary and runs statements of transactions slightly in advance to the other secondary replicas. The commit-ordering and transaction-ordering rules are implemented using the commit delayer principle which ensures correctness while handling the faulty primary replica. By simulation results, we have shown that the proposed technique involves less overhead and recovery time with increased accuracy.

REFERENCES

Amir, Y., C. Danilov, D. Dolev, J. Kirsch and J. Lane *et al*., 2010. Steward: Scaling byzantine fault-tolerant replication to wide area networks. IEEE Trans. Dependable Sec. Comput., 7: 80-93. DOI: 10.1109/TDSC.2008.53

Ayyasamy, S. and S.N. Sivanandam, 2009. A QoS-aware intelligent replica management architecture for content distribution in peer-to-peer overlay networks. Int. J. Comput. Sci. Engg., 1: 71-77. ISSN: 0975-3397

Ayyasamy, S. and S.N. Sivanandam, 2010a. A cluster based replication architecture for load balancing in peer-to-peer content distribution. Int. J. Comput. Networks communi., 2: 158-172. DOI: 10.5121/ijcnc.2010.2510

Ayyasamy, S. and S.N. Sivanandam, 2010b. Trust based content distribution for peer- to- peer overlay networks. Int. J. Network Security Appl., 2: 134-145. DOI: 10.5121/ijnsa.2010.2211

Castro, M. and B. Liskov, 1999. Practical byzantine fault tolerance. Proceeding of the 3rd Symposium on Operating Systems Design and Implementation, (SOSDI'99), New Orleans, USA., pp: 1-14.

Clement, A., M. Marchetti, E. Wong, L. Alvisi and M. Dahlin, 2009. Making byzantine fault tolerant systems tolerate byzantine faults. Proceeding of the 6th USENIX Symposium on Networked Systems Design and Implementation, (NSDI'09), USENIX Association, Berkeley, CA, USA., pp: 153-168.

Eger, K., T. Hobfeld and R. Binzenhofer, 2007. Efficient simulation of large-scale p2p networks: Packet-level vs. flow-level simulations. Proceedings of 2nd Workshop on the Use of P2P, GRID and Agents for the Development of Content Networks, (UPGADCN'07), ACM, New York, NY, USA., pp: 9-16. DOI: 10.1145/1272980.1272986

El Malek, M.A., G.R. Ganger, G.R. Goodson, M.K. Reiter and J.J. Wylie, 2005. Fault-scalable byzantine fault-tolerant services. Proceedings of the 20th ACM symposium on Operating systems principles (SOSP'05), ACM, New York, NY, USA.,pp: 59-74. DOI: 10.1145/1095810.1095817

Hales, D., A. Marcozzi and G. Cortese, 2007. Towards cooperative, self-organised replica management. Proceeding of the 1st IEEE International Conference on Self - Adaptive and Self-Organizing Systems, July 9-11, Cambridge, MA, USA., pp: 367-370. DOI: 10.1109/SASO.2007.62

Kotla, R., L. Alvisi, M. Dahlin, A. Clement and E. Wong, 2007. Zyzzyva: Speculative byzantine fault tolerance. Proceeding of the 21st ACM SIGOPS on Operating Systems Principles, (SOSP'07), ACM New York, NY, USA., pp: 45-58. DOI: 10.1145/1294261.1294267

Li, J. and D. Mazieres, 2007. Beyond one-third faulty replicas in byzantine fault tolerant systems. Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation, (USNSDI'07). VMware Inc. and Stanford University, CA., pp: 131-144.

Pathak, V. and L. Iftode, 2006. Byzantine fault tolerant public key authentication in peer-to-peer systems. Comput. Network, 50: 579-596. DOI: 10.1016/j.comnet.2005.07.007

Petr, A.H. and K.P. Druschel, 2006. The case for byzantine fault detection. Proceedings of the 2nd Conference on Hot Topics in System Dependability, (HOTDEP'06), SENIX Association Berkeley, USA., pp: 5-5. http://portal.acm.org/citation.cfm?id=1251019

Simon, L.M., 2009. Fault Detection: Theory, Methods and Systems. Nova Science Publishers, USA., ISBN-10: 161728291X

Singh, A., P. Fonseca, P. Kuznetsov, R. Rodrigues and P. Maniatis, 2009. Zeno: Eventually consistent byzantine-fault tolerance. Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, (NSDI'09), USENIX Association, CA., pp: 169-184.

Zhao, W., 2007. Byzantine fault tolerant coordination for web services atomic transactions. Proceedings of the 5th International Conference on Service-Oriented Computing, (ICSOC '07), Springer-Verlag Berlin, Heidelberg, pp: 307-318. DOI: 10.1007/978-3-540-74974-5_25

Zhijun, L.I. and L. Minghong, 2005. EOS: Evolutionary overlay service in peer-to-peer systems. Am. J. Applied Sci., 2: 1401-1406. DOI: 10.3844/ajassp.2005.1401.1406