International Journal of Engineering Innovation & Research
Volume 1, Issue 2, ISSN : 2277 – 5668

Fault Tolerance Techniques in Distributed System

Sourabh Dave
sourabh.dave@gmail.com

Abhishek Raghuvanshi
abhishek.raghuvanshi@yahoo.com

***Abstract* — Fault tolerance is an important issue in distributed computing. Developers of early distributed systems took a simplistic approach to providing fault tolerance: They just used another copy of the same hardware as a backup. There are various factors & critical issues responsible for these overheads. This paper provides a study of fault tolerance techniques in distributed systems, especially replication and checkpointing. We also suggested fault tolerance by combining replication and checkpointing and implemented it in Java RMI. This work will provide a good reference for researchers.**

***Key Words* — Fault-tolerance, distributed system, Distributed Computing, Replication, Redundancy, High availability.**

I. INTRODUCTION

The increasing use of computers and our increasing reliance on them have led to a need for highly reliable computer systems. There are many areas where computer perform life critical tasks. Some examples of these are flight control systems, patient monitoring systems etc. Other application areas include banking and stock markets. In these systems, failure of computers may lead to catastrophe, great financial loss, or even loss of human life. In such applications, highly dependable systems are needed.

Dependability means that our system can be trusted to perform the service for which it has been designed [7]. Dependability can be decomposed into reliability, availability, safety and security. Where, reliability deals with continuity of service, availability with readiness of usage, safety with avoidance of catastrophic consequences on the environment, and security with prevention of unauthorized access and/or handling of information.

A system failure occurs when the system behaviour is not consistent with its specifications [13]. A system consists of several components, more the number of components; the more are the things that could be faulty.

Since failures are caused by faults, a direct approach to improve the reliability of a system is to try to prevent faults from occurring into a system. This approach is called fault prevention. The other approach is fault tolerance. The goal is to provide service despite the presence of faults in the system.

The fault prevention methods [14] focus on methodologies for design, testing and validation; whereas fault tolerant methods focus on how to use components in a manner such that failures can be masked. Here onwards, we will be discussing techniques for building fault tolerant distributed systems. Distributed computing is a field of computer science that studies distributed systems. The term distributed system is used to describe a system with following characteristics: it consists of several computers that do not share a memory or clock; the computers communicate with each other by exchanging messages over a communication network; and each computer has its own memory and runs its own operating system [9]. The resources owned and controlled by a computer are said to be local to it, while the resources owned and controlled by other computers and those that can only be accessed through the network are said to be remote or global.

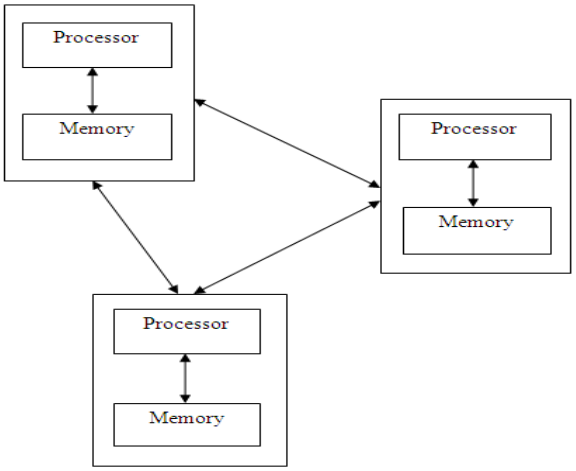


Fig. 1: Distributed Computing

II. COMMUNICATION AND EXECUTION

The physical and logical structures of distributed systems are discussed. Now, important communication constructs in a distributed system and their execution semantics will be discussed. It is necessary to understand the behaviour of a distributed system in order to understand many of the schemes for fault tolerance in such systems.

A. Interprocess Communication

In distributed systems, processes must communicate with each other in order to share information with each other. Synchronization is often needed between processes for communication. It is needed for controlled communication since a process cannot predict the speed of another process [6]. Communication and synchronization in distributed system, where there is no shared memory, are achieved by message passing.

B. Asynchronous Message Passing

Communication and synchronization between processes is treated separately in shared memory. Communication is done through reading and writing shared variables and for synchronization access to shared data, different methods are employed.

Message passing is used for communication and synchronization where there is no shared memory in distributed system [6]. Communication is achieved by a process which sends some data to another process which receives that data. Synchronization is achieved, since message passing implies that the receiving of the message is done after it has been sent. A message is sent by executing the send command. A send command is of the form: send (data, destination), where data is being sent by the process and destination specifies the process to which the data is being sent. If the message passing is asynchronous message passing, then it is assumed that there is infinite buffer to store messages. In other words, with asynchronous message passing, senders can continue to send messages which will be saved in a buffer for the receiver process to consume [6]. In this, the sender never blocks, that is, a send command always succeeds immediately and a sender can be arbitrarily ahead of the receiver. However, the receiver process is not non-blocking. It will have to block if there is no message in the buffer waiting for it.

C. Synchronous Message Passing

Synchronous message passing has no buffering. In this form the execution of a send command is delayed till the corresponding receives command is executed. Hence every execution of a communication command represents a synchronization point where both the sender and the receiver process synchronize [6].

The main advantage of synchronous message passing is that for synchronization at each communication command it is easier to make assertions about processes. For example, in the sender process, when the send command finishes, the sender process can make some assertions about the state of the receiver process.

D. Remote procedure call

To program any type of message based interaction between processes send/receive primitives are sufficient. Programs using send/receive primitives will require a send followed by a "receive" by the client process and a "receive" followed by a send by the server process in the client/server type interactions. In this interaction, the client process is blocked till the service is complete, even though it uses asynchronous message passing, since the client cannot proceed until it receives the result. In RPC (Remote procedure call), the service to be provided by the server client process that wants the service simply makes call to the procedure. The implementation of the RPC takes care of the underlying communication. In RPC a client interacts with the server by means of the call statement, as is done in a sequential language. A call statement is of the form: Call service (value_args, result_args), where service is name of the remote procedure, value_args are the arguments that provide the parameter values to the remote procedure, and result_args are the argument in which the result of the remote procedure are returned. In this client server type of interaction after each call to the server by the client, the state of the server and the state of the client changes from some initial state to some final state. It simplifies the task of supporting fault tolerance. In an RPC two different processes are involved- one executing the client, the other executing the server- in this new issues related to fault tolerance arise.

Under failure condition the semantics of the RPC cannot be like that of the simple procedure in a sequential program, in which the failure of a node means the failure of the caller as well as the callee, and the failure of the communication network has no effect.

III. FAULT TOLERANCE

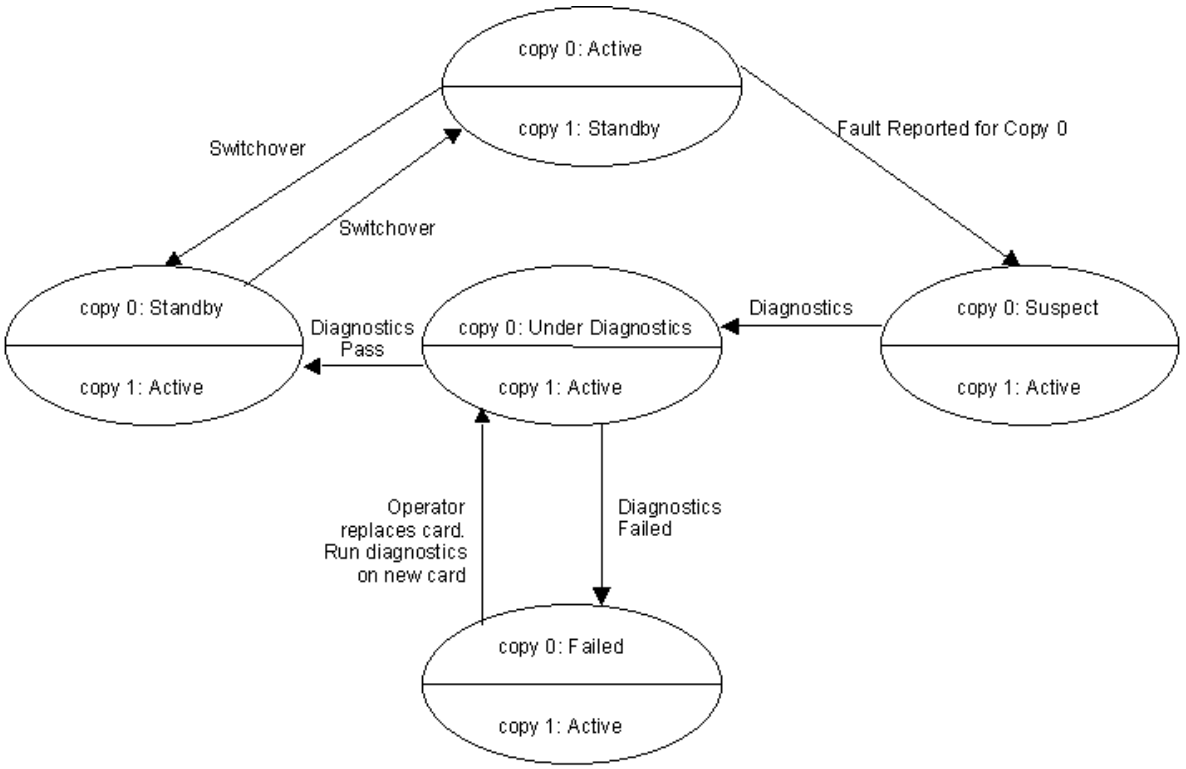
As already discussed, computing systems consist of a multitude of hardware and software components that are bound to fail eventually. In many systems, such component failures can lead to unanticipated, potentially disruptive failure and to service unavailability. Some systems are designed to be fault-tolerant: they either exhibit well-defined failure behaviour when components fail or mask component failures to users, that is, continue to provide their specified standard service despite the occurrence of component failures. To many users temporary errant system failure behaviour or service unavailability is acceptable. There is, however, a growing number of user communities for whom the cost of unpredictable, potentially hazardous failures or system service unavailability can be very significant [6]. Examples include the on-line transaction processing, process control, and computer-based communications user communities. To minimize losses due to unpredictable failure behaviour or service unavailability, these users rely on fault tolerant system. With the ever increasing dependence placed on computing services, the number of users who will demand fault-tolerance is likely to increase.

The task of designing and understanding fault-tolerant system architectures is notoriously difficult: one has to stay in control of not only the standard system activities when all components are well, but also of the complex situations which can occur when some components fail. The difficulty of this task is also due to lack of structuring concepts and use of different names for the same concepts. For example, what one person calls a failure, a second person calls a fault, and a third person might call an error.

A. Faults, Errors, and Failures

The definition of fault tolerance specifies the correct behaviour that is expected from the system. A failure occurs when an actual running system deviates from this specified behaviour. The cause of a failure is called an error. An error represents an invalid system state, one that is not allowed by the system behaviour definition. The error itself is the result of a defect in the system or fault. In other words, a fault is the root cause of a failure [6]. That means that an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures.

Faults can be characterized as transient or permanent. Transient faults are fault of limited duration, caused by



temporary malfunction of the system or due to some external interference. They can cause a failure, or an error, only in the duration for which they exist [6]. These errors caused may also exist only for a short duration, which makes detecting such faults very hard and expensive.

Permanent faults are those in which once the component fails, it never works correctly again. Many techniques for fault tolerance assume that the components fail permanently.

B. Fault Handling Lifecycle

A typical fault handling state transition diagram is as shown in Figure 4. The assumption made here is that the system is running with copy-0 as active unit and copy-1 as standby [10].

When the copy-0 fails, copy-1 will detect the fault by any of the fault detection mechanisms that are implemented by the system. At this point, copy-1 takes over from copy-0 and becomes active. The state of copy-0 is marked suspect and for the time being diagnostics is pending. The system raises an alarm, notifying the operator that copy-0 is in stand-by mode and diagnostics are to be done. Diagnostics are now scheduled on copy-0. This includes power-on diagnostics (to check power failure) and hardware interface diagnostics (to check failure of hardware components). If the diagnostics on copy-0 pass, copy-0 is brought in-service as standby unit. If the diagnostics fail, copy-0 is marked failed and the operator is notified about the failed card. The operator replaces the failed card with a new one and the system diagnoses the new card to assure that it is healthy. Once the diagnostics pass, copy-0 is marked standby. The copy-0 now monitors the health of copy-1 which is currently the active copy. The operator now restores the original configuration, i.e. copy-0 becomes active and copy-1 standby.

C. Phases in Fault Tolerance

In general, the implementation of fault tolerance in any particular system is closely linked with the system, its architecture and design. Just like designing a system is depend on the properties/requirements of the system, designing a fault tolerant system is also dependent on the needs and functionality of the system [8]. Thus, no general technique can b e proposed for “adding” fault tolerance to a system. However, some general principles which are useful in designing fault tolerant systems have been identified.

The four phases that are general when designing fault tolerance in a system are:

(1) Error detection
(2) Damage confinement
(3) Error recovery
(4) Fault treatment and continued system service

1) Error Detection:

The first step to any fault tolerance activity is error detection. Faults and failures cannot be observed directly and thus first state of the system is checked to see if an error has occurred or not, after which failures and faults can be deduced [6]. Hence, error detection mechanisms are also referred to as “failure/fault detection”.

Since error detection is the first and foremost step of fault tolerance, there are some important properties that an error detection check should satisfy. Firstly, the check should never be influenced by the internal design of the system, it should be determined from the specifications of

the system. Any influence of the system on the check can cause same error in the check as is present in the system.

Secondly, an ideal check should be complete and correct, i.e. the check should be able to detect all possible errors in the behaviour of the system and should not detect any error when none is present. If the check is not complete, then some errors may remain undetected in the system thereby later causing failure of the system.

2) Damage Confinement and Assessment:

There is always a time difference between when the failure occurred and when the error was detected. This delay can cause the error to spread to other parts of the system. The goal of this phase is to determine the boundaries of corruption of the system, before the error id detected and corrected.

Errors spread when different components of the system communicate with the faulty component. So, to determine the amount of damage in the system after an error has been detected, the flow of information between the faulty component and other components is examined [6]. The boundaries are identified beyond which no information exchange occurred and it is implied that the damage is limited to this boundary.

The boundary can be identified dynamically by recording and examining the information flow that occurred, but this method is a little complex. Another way is to statically include firewalls in the design of the system to ensure that no information flow takes place outside these walls. Thus, if an error is detected within this defined area then it can be assumed that it has not spread beyond the fire walls.

3) Error Recovery:

This is the phase when the error is removed from the system, after it has been detected and its extent identified. If the error is not removed, it may cause failure of the system in future. Thus, in this phase the system state is made error-free and the system is restored to a consistent state. There are two general techniques of error recovery: Backward recovery and Forward recovery

In Backward recovery, the system state is restored to an earlier state that is error-free. However, this requires that the state of the system be periodically saved on stable storage (check-pointing) that is not affected by failure. When some error or failure is detected, the system is rolled back to the last checkpointed state [6]. Since the failure occurred after the checkpoint was done, so the checkpointed state will be error free and thus, after the rollback, the state of the system will also be error-free.

In Forward recovery, no previous state of the system is available and thus, the system does not roll back. Instead, the goal is to go forward and reach a consistent state, which is error free. This form of recovery seems very promising in terms of overhead and efficiency but it requires thorough assessment of the damage to the state. And the error can be removed only if exact nature of the error is known, which requires good diagnosis of the reason of failure. The diagnosis has to be system and application dependent, which makes forward recovery a system and application dependent approach. Due to this, it is not as commonly used as backward recovery.

4) Fault Treatment and Continued Service

In the first three phases, the focus is on errors. The error is detected, its extent determined and then it is removed, after which the system becomes error-free. This works when the error is caused by some transient fault (exist for a small duration). After error recovery, the system restarts from an error free state.

But, if the faults are permanent, then the one that caused the error and failure still remains in the system and may cause the system to fail again. Thus, it is required that the faulty component be recognized and should not be used after error recovery phase. The goal of this phase is to replace the faulty component in such a manner that the computation of the system is not hindered. This phase has two sub phases Fault location and System repair.

In system repair, the system repaired such that either the faulty component is not used or used in a different configuration [7]. One of the simplest and most commonly used strategies for system repair is standby spare strategy. In this, there is a standby component in the system which is used if the main component fails. The state of the stand by component is made consistent with the state of the rest of the system.

IV. REPLICATION AND CHECKPOINTING

A. Replication

In distributed systems if the data resides on one single node only then nothing can be done to successfully complete that action which will need that data if any kind of failure occurs [1]. Hence if we want to complete the action which was stopped due to failure we need to replicate that data. Replication in simple definition means to make several copies of that data and keep them on several nodes [3]. Therefore if we replicate the data then if failure on one node occurs then it will not be inaccessible to the user. Replication is one of the key concepts in distributed systems, introduced mainly for increasing data availability and performance. It consists in maintaining multiple copies of data items (objects) on different servers. However, replication introduces some serious problems like consistency.

Replication is a key to effectiveness of distributed systems in that it can provide enhanced performance, high availability and fault tolerance [5]. For example, the caching of resources from web servers in browsers and proxy servers is a form of replication, since the data held in caches and at servers are replicas of one another. Hence the main motivations of replication are to improve:

1) *Performance enhancement:* Replication increases performance with little cost to system. Replication of changing data of web leads to overheads in the form of protocols. These protocols ensure that users receive up to date data. But performance enhancement using replication has its own limitations.

2) *Increased availability:* Users require 100% availability of the data they want to access with reasonable response times. Apart from failures other conflicts do occur when data is not available to user. Some other factors or conflicts are: Sever Failure, Process Failure, Network partitions that leads to communication failure.

3) *Fault tolerance:* The high availability [13] of data does not mean that data is correct or the recent updated data that is it may be out of date. But fault tolerance service always guarantees that the data is recent and correct despite of various faults. This is useful in fields like air traffic control where correct data is needed on short time scales.

Centralized systems have only one version of every object, and its state at any time reflects the last write operation [14] on that object. In a distributed system the notion of last is not so obvious because of the lack of a common clock. Every system using replication must therefore use some kind of consistency protocol that will arrange communication between replicating servers.

B. Types of Replication

In the distributed systems replication is mainly used to provide fault tolerance. Two replication protocols have been used in distributed systems: Active and Passive replication [10].

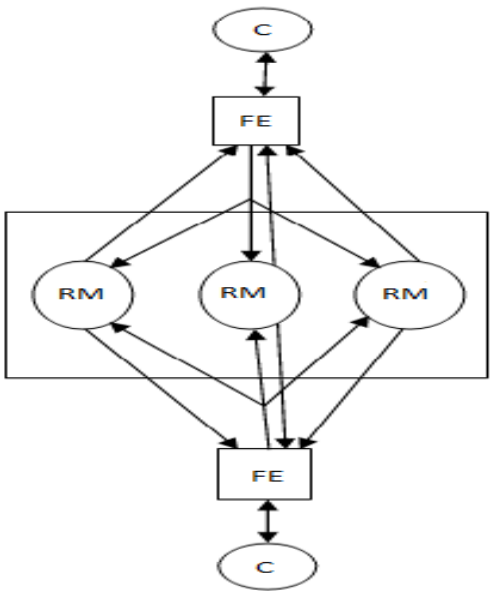


Fig. 4: Active Replication

In active replication each client request is processed by all the servers. It is also known as state machine replication [4]. This requires that the process is deterministic. Deterministic means that, given the same initial state and a request sequence, all processes will produce the same response sequence and end up in the same final state. In order to make all the servers receive the same sequence of operations, an atomic broadcast protocol must be used. An atomic broadcast protocol guarantees that either all the servers receive a message or none will receive, and they all will receive messages in the same sequence. In the above active replication diagram, C refers to the clients, FE are the front end nodes and RM be the replica managers. Here the replica managers are state machines [4] that are organized as groups. Front ends multicasts their request to replica managers and all the replica managers process the request independently and the reply. When the replica manager crashes there is no impact on the performance because the remaining replica managers continue to respond. Active replication can control byzantine faults because the front ends can collect the different replies of replica managers and compare them. The sequence of events when a client requests an operation to be performed [15] is as follows:

1) *Request:* The front end multicasts the request, containing a unique identifier to the group of replica managers. It does not issue another request until each and every replica manager responds to the request.
2) *Coordination*: With the help of group communication the request is delivered to the replica managers in the order.
3) *Execution*: Every replica manager executes the request independently and the response contains the client's unique identifier.
4) *Agreement*: This phase is not there in active replication since multicast delivery is used.
5) *Response*: Each replica manager responds to the front end which then sends the request back to the client. The number of replies the front end collects depends upon the failure state and the multicast algorithm used.

In passive replication [10] (Primary backup) there is only one server (primary) that processes client requests. After processing a request, the primary server updates the state on the other backup servers and sends back the response to the client. If the primary server fails, one of the backup servers takes its place. Passive replication may be used even for non-deterministic processes. The disadvantage of passive replication compared to active is that in case of failure the response is delayed. In the above diagram of passive replication, C refers to the clients, FE are the front end nodes and RM be the replica managers. The sequence of events [10] when a client requests an operation to be performed is as follows:

1) *Request:* The front end issues the request, containing a unique identifier to the primary replica manager.
2) *Coordination*: The primary takes each request one at a time, in the sequence in which it has received the request. It checks the unique identifier if it has already executed the request and if it is so it simply resends the request.
3) *Execution*: The primary executes the request and stores the request.
4) *Agreement*: If the request is update then the primary sends the updated state, response and the unique identifier to all the backups. Then the backups send the acknowledgement.

5) *Response*: The primary responds to the front end which then sends the request back to the client.

C. Checkpointing

Fault tolerance techniques enable systems to perform tasks in the presence of faults. Fault tolerance can be achieved through some kind of redundancy. The most common method used is checkpoint-restart [17]; an application is restarted from an earlier checkpoint or recovery point after a fault. This may result in the loss of some processing and applications may not be able to meet strict timing targets.

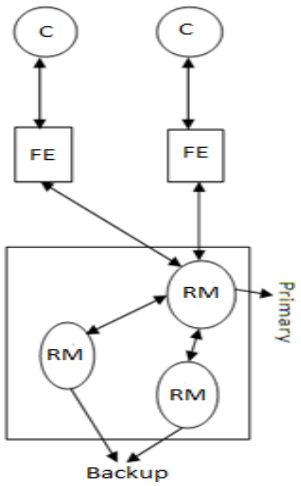


Fig. 5: Passive replication

Checkpointing is primarily used to avoid losing all the useful processing done before a fault has occurred [11]. Checkpointing consists of intermittently saving the state of a program in a reliable storage medium. Upon detection of a fault, previous consistent state is restored. In case of a fault, checkpointing enables the execution of a program to be resumed from a previous consistent state rather than resuming the execution from the beginning. In this way, the amount of useful processing lost because of the fault is significantly reduced.

D. Types of checkpointing

Depending on the programmer's intervention in process of checkpointing, it can be classified as follows:

1) *User triggered checkpointing:* These checkpointing schemes [18] require user interaction. These are generally employed where the user has the knowledge of the computation being performed and can decide the location of the checkpoints. The main problem is the identification of the checkpoint location by a user. This approach is well suited for long-running, computation-intensive parallel applications, because of the minimal fault-free overhead. Indeed, there is no overhead during the normal execution of the application between the moments that the checkpoints are taken.
2) *Uncoordinated Checkpointing:* In uncoordinated or independent checkpointing [16], processes do not coordinate their checkpointing activity and each process records its local checkpoint independently. In this way, each process becomes independent in deciding when to take checkpoint, i.e., each process may take a checkpoint when it is most convenient. It eliminates coordination overhead all together and forms a consistent global state on recovery after a fault. After a failure, a consistent global checkpoint is established by tracking the dependencies [11]. It may require cascaded rollbacks that may lead to the initial state due to domino-effect, i.e. the processes may resume from the beginning. It requires multiple checkpoints to be saved for each process and periodically invokes garbage collection algorithm to reclaim the checkpoints that are no longer needed. In this scheme, a process may take a useless checkpoint that will never be a part of global consistent state. Useless checkpoints incur overhead without advancing the recovery line.
3) *Coordinated Checkpointing:* In coordinated [16] or synchronous checkpointing, processes take checkpoints in such a manner that the resulting global state is consistent. Mostly it follows two-phase commit structure. In the first phase, processes take tentative checkpoints and in the second phase, these are made permanent. The main advantage is that only one permanent checkpoint and at most one tentative checkpoint is required to be stored. In case of a fault, processes rollback to last checkpointed state. A permanent checkpoint cannot be undone. It guarantees that the computation needed to reach the checkpointed state will not be repeated [11]. A tentative checkpoint, however, can be undone or changed to be a permanent checkpoint.
4) *Message Logging based checkpointing:* Message-logging protocols [18] are popular for building systems that can tolerate process crash failures. Message logging and checkpointing can be used to provide fault tolerance in distributed systems in which all inter-process communication is through messages. Each message received by a process is saved in message log on stable storage. No coordination is required between the checkpointing of different processes or between message logging and checkpointing. When a process crashes, a new process is created in its place [11]. The new process [12] is given the appropriate recorded local state, and then the logged messages are replayed in the order the process originally received them. All message logging protocols require that once a crashed process recovers, its state needs to be consistent with the states of the other processes.

V. OUTCOME

A systematic investigation is carried out:

1) To find pros and cons of different fault tolerance techniques in distributed system.

2) To address the consistency issue in replication based fault tolerance technique.
3) To explore the way to reduce the overhead of check-pointing technique.

Another important issue relating to a checkpointing server is the overhead of time delay while retrieval of checkpoints. When an independent process crashes, it has to retrieve only its own last consistent state. But in case of a process, which is communicating with several other processes, is crashed the retrieval has to be done for several processes. This results in time delay at server, since it has to process several requests at the same time. To resolve this issue, the states have been saved at all the replicas of the node where the process is running. The solution of both these issues is combined to create a consistent multiple fault tolerant system.

CONCLUSION

We have proposed an improved method to ensure the consistency by simulating the distributed environment using java RMI. This work reveals to lock leased protocol for write-write or read-write operation. Concurrent read operations can be performed simultaneously. This algorithm is very simple and ensures the consistency in a very simple manner. Checkpointing overhead is reduced by saving the checkpoints on local hard disk instead of SNA (Storage Network Area) or DFS (Distributed File System) following the assumptions given by John Paul Walters. John Paul Walters only addressed replication placement but he has not addressed consistency issue of replica management.

This work is one step ahead as a control frame is suggested that is work as a coordinator for interactive consistency model of checkpointing replication. Multiple fault capability is managed by this controller although it is a single point failure due to master controller but in future (as future work) controller can be replicated to protect from single point failure. This work will definitely work as a reference for researcher and practitioner to design and develop high performance multiple fault tolerance.

.

ACKNOWLEDGMENT

We would like to thank the all faculty members of the institute who helped us lot in calculating the facts and figures related to our paper. I would also like to thank the anonymous reviewers who provided helpful feedback on my manuscript.

REFERENCES

[1] Data Replication strategies in wide area Distributed Systems. Sushant Goel, Grid Computing and Distributed Systems (GRIDS) LaboratoryDepartment of Computer Science and Software Engineering, The University of Melbourne, Australia.

[2] A Concept of Replicated Remote Method Invocation Jerzy Brzezinski and Cezary Sobaniec, Institute of Computing Science, Poznan University of Technology, Poland {Jerzy.Brzezinski, Cezary.Sobaniec}@cs.put.poznan.pl

[3] Replication-Based Fault Tolerance for MPI Applications John Paul Walters and Vipin Chaudhary, Member, IEEE

[4] A Fusion-based Approach for Tolerating Faults in Finite State Machines Vinit Ogale , Bharath Balasubramanian and Vijay K. Garg Parallel and Distributed Systems Laboratory, Dept. of Electrical and Computer Engineering, The University of Texas at Austin. IBM India Research Lab (IRL),Delhi, India.

[5] Protocols for maintaining consistency of replicated data Ricardo Anid and N.C. Mendonca.

[6] Jalote, P. Fault Tolerance in Distributed Systems, (Prentice Hall, 1994).

[7] http://www.ibm.com/developerworks/rational/library/114.ht ml.

[8] A survey on fault-tolerance in Distributed Network systems.Naixue Xiong, College of Computer Science, Wuhan Univ. of Science and Engg,, China Yan Yang, Center of Asian and Pacific Studies, Seikei Univ., Tokyo, Japan.

[9] http://en.wikipedia.org/wiki/Distributed_computing.

[10] http://www.eventhelix.com/faulthandling/faulthandlingtechn iques.

[11] Checkpointing Based Fault Tolerance in Mobile Distributed Systems Parveen Kumar1, Rachit Garg.

[12] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," Proc. Supercomputing Symp., pp. 379-386,1994.

[13] Rachid Guerraoi, and André Schiper, "Software-based replication for fault tolerance," Journal of the ACM, Vol. 30, issue 4, April 1997.

[14] Halpern, J. and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," Proc. of the 3rd ACM Symposium on Principles of Distributed Systems, 1984, pp. 50-61 and Lamport, L., R. Shostak, and M. Pease, "The Byzantine Generals Problem," ACM Transactions on Programming Languages and Systems, Vol. 4 No. 3, July 1982, pp. 382-401.

[15] Andrews, Gregory R. (2000), Foundations of Multithreaded, Parallel, and Distributed Programming, Addison–Wesley, ISBN 0-201-35752-6.

[16] Guohong Cao and Mukesh Singhal , "On Coordinated Checkpointing in Distributed Systems", IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 12, December 1998.

[17] Mohamed-Slim Bouguerra, Thierry Gautier, Denis Trystram, and Jean Marc Vincent, "A Flexible Checkpoint/Restart Model in Distributed Systems", Springer-Verlag Berlin Heidelberg 2010, R. Wyrzykowski et al. (Eds.): PPAM 2009, Part I, LNCS 6067, pp. 206–215, 2010

[18] Liu, Y., Nassar, R., Leangsuksun, C., Naksinehaboon, N., Paun, M., Scott, S.: An optimal checkpoint/restart model for a large scale high performance computing system. In: IEEE International Symposium on Parallel and Distributed Processing, pp. 1–9 (2008)

Sourabh Dave received his B.E. degree in Computer Science from MITM, Indore in the year 2006. He is currently pursuing M.Tech in Information Technology from MIT, Ujjain. His research interest includes Distributed Systems, Ad-Hoc Networks and Compiler Design.

Abhishek Raghuvanshi received his B.E degree in (ter Science from IETE, New Delhi in the year 2 d M.Tech in Computer Science from Technocrats Institute of Technology Bhopal in the year 2010. He is an Assistant Professor of Computer Science department at MIT Ujjain. His research interest includes Distributed Systems, Databases and Network Security.