Fault Tolerant Distributed Computing

Mitvin Shah - mshah@cse.uta.edu
Department of Computer Science and Engineering
University of Texas at Arlington

Abstract

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable" Leslie Lamport, May 1987.

Fault tolerance is in the center of distributed system design that covers various methodologies. In past there have been cases where critical applications buckled under faults because of insufficient level of fault tolerance. Various issues are examined during distributed system design and are properly addressed to achieve desired level of fault tolerance. This paper defines various terminologies like failure, fault, fault tolerance, recovery, redundancy, security, etc and explains basic concepts related to fault tolerance in distributed environments. It also describes four kinds of fault tolerance and ways of achieving. The paper presents various solutions and architectures that implements fault tolerance in various facets of distributed computing. These solutions also cover few ongoing research works. Overall goal of this paper is to give understanding of fault tolerant distributed system and to familiarize with current research in this area.

1. Introduction

Distributed Computing Systems consists of variety of hardware and software components. Failure of any of these components can lead to unanticipated, potentially disruptive behavior and to service availability [2]. In past there have been cases where critical applications buckled under faults because of insufficient level of fault tolerance. Few of them just missed. In one such case the planned lift-off of the space shuttle Columbia on Oct. 9, 1981 was delayed due to a minor fuel spill and a few missing tiles, upon simulation they found that the code contained an uninitialized counter used in a "computed goto" command that resulted in all four of the redundant flight computers simultaneously branching off to a memory address containing no code. A subsequent investigation of the software using the specific knowledge gathered from this incident led to the discovery of 17 other similar systematic bugs in the flight control software, one of which could also have caused a catastrophic failure. Important lesson drawn from this case was to providing perfect solution to software fault tolerance over and above redundancy. Also the distributed systems are vulnerable to security threats because of their openness in operating environment. Chow [6] terms both of them as system faults. Providing proper fault detection mechanisms along with redundancy in the system by replication of data and resources can prevent failures. It also requires recovering by rolling back the execution of all the affected processes. Security is mainly concerned with issues like authentication and authorization and we drop this issue for discussions in this

paper. Also the aim of fault tolerant distributed computing is to provide proper solutions to these system faults upon their occurrence and make the system more dependable by increasing its reliability. The solutions to these system faults should be transparent to users of the system. Such fault tolerant behavior is extremely necessary in critical applications like flight control systems, hazardous industries, nuclear power plants, etc. as well as non critical ones like communications and transaction processing.

The field of fault tolerant computing has ever been specific to applications till attempts by Laprie [3] to organize concepts and terminologies. However later on Arora and Gouda [4] termed the field to be "fragmented". Since then more formal and abstract approach has led to better understanding of the problems faced and key to developing fault resilient systems. Researchers have undergone their work in various paradigms of distributed fault tolerance from failure detection to mobile security. One such approach by Moorsel [5] specifies action models and path based solution algorithm to provide an intuitive, high level, modeling formalism for fault tolerant distributed computing systems and to analyze the impact of fault tolerance mechanisms on the user perceived reliability. Kienzle [7] in his paper reviews the applicability of transactions and other fault tolerance mechanisms in concurrent programming language Ada. Numerous programming languages with support for fault tolerance have been developed like Arugus [18] and Plits [19]. Some of them are extensions of existing languages such as Fault Tolerant Concurrent C (FTCC) [20]. A new implementation of MPI called FT-MPI proposed by Fagg [21] allows the semantics and associated failure modes to be completely controlled by the application rather than just checkpoint and restart. Implementation of new Log Manager by Daniels [13] for shared logging service of the QuickSilver distributed operating system solves the problem of logging services shared by multiple resource managers. Yang [23] implemented SUVS (Simplified Unmanned Vehicle System) distributed real time test bed with system level fault tolerance techniques. Zhou [24] describes the design of a model that supports fault tolerant services, based on twin server model, of fault tolerant servers for the micro kernel based RHODOS distributed operating system. Goldreick [8] proposes a compiler that outputs a fault tolerant protocol given an input protocol for n semi-honest parties maintaining its correctness and privacy. Canetti [9] explains the relations between security and randomness in context of multiparty computations.

Our focus is on understanding basic concepts of fault tolerance and implementation with the help of case studies one of them mentioned in Section 7. The paper is organized as follows: Section 2 introduces basic concepts and terminologies that are used, Section 3 broadly defines four forms of fault tolerances, Section 4 describes various phases of achieving fault tolerance like failure detection, redundancy, check pointing, etc., Section 5 describes achieving safety and liveness, Section 6 presents an overview of rollback and recovery mechanisms Section 7 presents various fault tolerant architectures covering various aspects of distributed computing and finally we conclude with Section 8.

2. Basic Concepts and Terminologies

Being fault tolerant is very much related to what are called dependable systems. A system is dependable when it is trustworthy enough that reliance can be placed on the service that it delivers. For a system to be dependable, it must be available (e.g., ready for use when we need it), reliable (e.g., able to provide continuity of service while we are using it), safe (e.g., does not have a catastrophic consequence on the environment), and secure (e.g., able to preserve confidentiality). Following are few terminologies that are very closely related to dependability of system and its behavior:

Fault – Can be termed as "defect" at the lowest level of abstraction. It can lead to erroneous system state. Faults may be classified as transient, intermittent or permanent. They can be of following types [6]:

1. Processor Faults (Node Faults): Processor faults occur when the processor behaves in an unexpected manner. It may be of classified into three kinds:
   a) Fail-Stop – Here a processor can both be active and participate in distribute protocols or is totally failed and will never respond. In this case the neighboring processors can detect the failed processor.
   b) Slowdown – Here a processor might run in degraded fashion or might totally fail.
   c) Byzantine – Here a processor can fail, run in degraded fashion for some time or execute at normal speed but tries to fail the computation.
2. Network Faults (Link Faults): Network faults occur when (live and working) processors are prevented from communicating with each other. Link faults can cause new kinds of problems like:
   a) One way Links – Here one processor can send messages to other is not able to receive messages. This kind of problem is similar to that faced due to processor slowdown.
   b) Network Partition – Here a portion of network is completely isolated with the other.

Error – Undesirable system state that may lead to failure of the system.

Failure – Faults due to unintentional intrusion.

Types of Failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure Receive omission Send omission	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure Value failure State transition failure	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Fault Tolerance – Ability of system to behave in a well-defined manner upon occurrence of faults.

Recovery – Recovery is a passive approach in which the state of the system is maintained and is used to roll back the execution to a predefined checkpoint.

Redundancy – With respect to fault tolerance it is replication of hardware, software components or computation.

Security – Robustness of the system characterized by secrecy, integrity, availability, reliability and safety during its operation.

3. Types of Fault Tolerance and Failure Detection

3.1 Types of Fault Tolerance

Lamport [1977] described system behavior with respect to its safety and liveness. For a distributed program to behave correctly, it must satisfy both the above properties. We will try to explain these concepts before we go ahead with explaining types of fault tolerances.

Safety means that some specific "bad thing" never happens within a system [1]. Formally, this can be characterized by specifying when an execution e is "not safe" for a property p: if e $\notin$ p there must be an identifiable discrete event e that prohibits all possible system executions from being safe. e.g.: simultaneous updating of a shared object. Distributed program is safe if system will always remain within set of safe states.

On the other hand, a liveness property claims that some "good thing" will eventually happen during the system execution [1]. Formally, a partial execution of a system is live for property p, if and only if it can be legally extended to still remain in p. "Legally here means that the extension must be allowed by the system itself. e.g. : a process waiting for access to a shared object will finally is allowed to do so.

To behave correctly, a distributed program A must satisfy both its safety and its liveness property. Now upon occurrence of fault, how is the property of A affected?

Forms of Fault Tolerance

	live	not live
safe	masking	fail safe
not safe	non-masking	none

The *masking* type of fault tolerance is most desirable but most expensive to implement. Applications with this kind of fault tolerance are able to tolerate faults in

transparent manner. While for last case where neither safety nor liveness is guaranteed is the most undesirable.

Among the two intermediate *fail safe* is favorable (and is active area of research) over *non-masking* because of the importance of leaving the system in safe state. In case of *non-masking* type the output of the system may not be desirable or correct but still the result is delivered. Recently specialization of *non-masking* fault tolerance called *self-stabilization* is actively worked after. Programs of this kind are able to withstand any kinds of transient faults. However programs of such kind are difficult to construct and test.

3.2 Failure Detection

Failure detection as you will see later in Section 5 is extremely essential in achieving safety and liveness property of the system. Various researchers for efficiently detecting and determining the type of fault in the system have made numerous efforts. One of the significant contributors has been Chandra [26] to solving consensus and atomic broadcast problem by unreliable failure detectors by use of rigorous formalization. Gallet [27] have further investigated their efficiency with measures using longest message chain before decision and greatest lower bounds for main class of failure detectors. Yang [28] related different system models and failure detectors by converting consensus algorithm proposed for one model to another. Beauquier [29] have presented hierarchy of transient failure detectors that detect occurrence of transient faults and the resources required for implementing them.

4. Redundancy

No matter how well you design your system to tolerate fault, it is always possible that is fails under repeated or severe attacks of faults. Gartner [1] claims that redundancy is necessary, though not sufficient condition for fault tolerance. It states two forms of redundancy, first that in *space* and second that in *time*.

Redundancy in space is meant by set of configurations in a program that are never reached in absence of faults while *redundancy in time* is meant by set of actions of a program that are never executed in absence of faults.

It is important to note that proper fault detection is very essential for the system to lead it to safety. Detection needs information about state space and /or program actions. Correction upon detection of fault ensures liveness property of the system. So we can understand that ensuring safety (by detection) is easier to achieve than liveness (by correction). It is also necessary to add that these detection and correction mechanisms should themselves be fault tolerant.

Redundancy in space is achieved by replication of components. Examples of space redundancy in hardware it may be tandem systems or in software may be like adding parity bits in transmission. *Redundancy in time* is like repeating the computation

in the same system. Examples of time redundancy might be calculation of result more than once.

Hardware redundancy is in the form of *replaceable hardware units* meaning the units that fail independently of other units (or can be added or removed independently) [2]. The service provided by the hardware servers in each replaceable unit should have very good failure semantics (crash and omission). The *replaceable hardware units* can be having *coarse granularity architecture* or *fine granularity architecture*. Prior architectures e.g. Tandem, the DEC, VAX Clusters and the IBM MVS/XRF have some replaceable units package together several elementary hardware servers such as CPUs, memory, I/O controllers and communication controllers. Later architectures e.g. Stratus and Sequoia have each elementary hardware server as a replaceable unit by itself. Certain assumptions, e.g. memory having read omission failure semantics, regarding hardware failure semantics are made by designers to use known hierarchical masking techniques. Failure is detected in hardware by either error detection codes (with error detecting circuitry is used) or by duplication with comparison (where hardware duplication with comparison logic is used). Error detection codes are used in storage and communication hardware servers while duplication and matching is used in complex circuitry [2]. Hardware server failures are masked in hardware itself by implementing redundancy management mechanisms (mostly multiplexing). But such mechanisms are not able to eliminate the need for handling at application software levels processor service failure, operating system level failures and application level failures.

Software redundancy in similar way to hardware redundancy should provide good failure semantics. These failure semantics depends on the persistent state of the service. Cristian [2] mention the use of simplicity and clarity during design, hierarchical design methods based on information hiding and abstract data types, rigorous design specification and verification techniques, systematic identification, detection, and handling of all exception occurrences and use of modern inspection and testing methods to prevent design faults in program. Software Servers must provide concurrency control and recovery support. Software group masking techniques based on Tandum and IBM XRF use members with crash/performance failure semantics. Any software service should replicate its resources in order to mask service failures. Several issues arise such as consistency of states, communication, replication management while maintaining software server groups.

System failure semantics is implicit upon failure semantics of all levels of abstraction of the system i.e. hardware, operating system and application. The goal is to make hardware having crash of omission failure semantics and make software be totally or at least partially correct.

5. Safety and Liveness

As we discussed earlier safety and liveness is essential for a program to work correctly. In this section we will look at how can we achieve them.

5.1 Achieving Safety

Faults must be detected and actions must be taken to remain safe. Detection is easy locally, but in distributed settings it requires intrinsic fault tolerant mechanisms like consensus algorithms and failure detectors [1]. Formally detection always includes checking whether a certain predicate Q over the extended systems state holds. If the type and effect of faults from F are known, it is easy to specify Q. In distributed settings the detectability of Q depends on different factors and is constrained by system properties. Consensus algorithms are used to detect this predicate Q to be holding. Consensus can be viewed as agreement by set of processes on a common value. Fault tolerant consensus algorithms "diffuse" initial values within the system to all nodes. All the nodes decide upon the next course of action depending upon the diffused information. Different forms of unreliable failure detectors are used to solve failure detection problem in asynchronous systems [10].

5.2 Achieving Liveness

Liveness is essential for a system to be masking fault tolerant apart from safety. It involves not only fault detection but also correction. Correction implies recovery from bad state to good state. Correction may be achieved by various ways like *error correction codes, rollback recovery* or *roll-forward*. Formally on detecting a "bad" state through detection of predicate Q the system must try to impose a new target predicate R into the system. The choice of R is difficult to decide because choice that is good for one process may not be same for the other. This problem in generally solved in distributed settings by majority voting. A particular node may also take the decision by first received vote. Schneider [11] proposed *state machine approach* to ensure liveness in distributed systems. Here servers are replicated and their behavior is coordinated via consensus algorithms. *Fault tolerant broadcasts* also enforce fault tolerant services [12].

6. Rollback and Recovery

Rollback and recovery mechanisms are essential for a system to maintain its masking fault tolerant property. The recovery manager of the system to bring it back to consistent state from where all its processes can start again implements various such techniques. A checkpoint is an entry made in the recovery file at specific interval of time that will force all the current committed values to the stable storage [25]. Upon failure the process will rollback to the latest checkpoint made and will also force all other processes depending on it to do so. It is the responsibility of the recovery manager to implement proper check pointing algorithm so that system is brought to globally consistent state upon recovery from failure. Various check pointing and rollback recovery mechanisms are used to ensure reliable distributed environment. We will focus on one of the check pointing method proposed by Park [17] that is based on communication pattern of the processes.

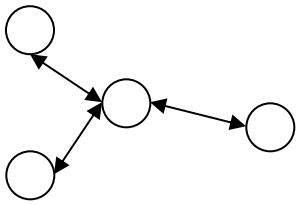
Park [17] presents check pointing coordination scheme in which the application controls the coordination activity by utilizing the communication pattern of application program. It classifies communication patterns of the processes as follows: centralized, serial, circular, hierarchical, irregular, rare and partitioned. Though the patterns may

change upon the coarse of execution, in most cases one or a few patterns dominate the inter-process communication. There is certain fixed property of the length domino cycle formed under each pattern and it is found that most patterns have domino cycle with limited length. So it suggests two check pointing coordination schemes to avoid the domino cycle with reduced coordination for domino cycle with length two. For the communication patterns producing the domino cycle with length longer than two, limited coordination scheme is suggested. The process follows loose check pointing coordination in this case with limited target processes. Performance evaluation of the schemes evaluates two performance measures, namely average number of additional checkpoints taken at each process and rollback distance. Results claim to reduce number of checkpoints and coordination overhead. It however says that in some cases the rollback distance might get longer.

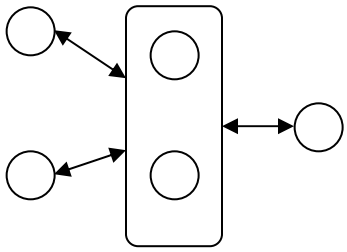
7. Case Study : Somersault

Somersault is a middleware for developing and integrating distributed fault-tolerant software components. It provides a fault-tolerant communication transport protocol that can be combined with ORB to achieve replication transparency. Design goals are aimed to serve applications that require continuous availability, high message throughput, maintaining consistent state and guaranteed message delivery. Murray [15] uses “roll forward” approach where it replicates process and makes it highly available upon failure of primary. It claims to survive hardware failures, operating systems failures and non-replicated application failures. It provides C++ middleware library to manage each pair of replica processes and to maintain identical computation and consistent state in each. This library can be integrated with CORBA.

Somersault consists of two units: simple non-fault-tolerant and recovery unit of replicated processes distributed across the network. Somersault implements n to m connection oriented messaging protocol where recovery unit acts as a single entity.



Communication Patterns in Regular Distributed Systems



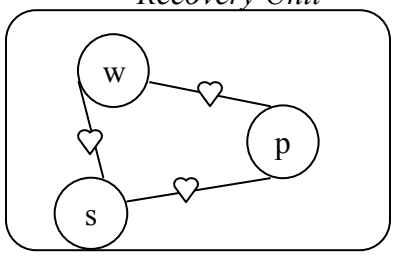
Communicating Units in Somersault

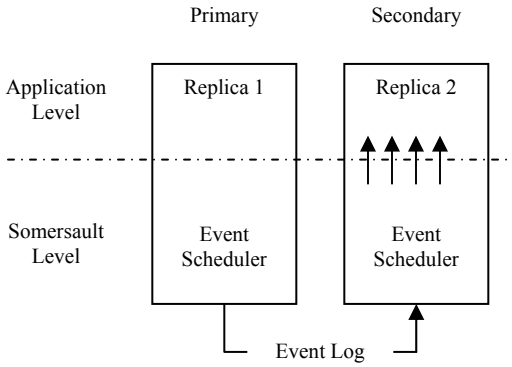
For our understanding we will consider a minimal case of three processes: the Primary, Secondary and Witness. Primary and Secondary replicate the application and

are involved in failure detection. Witness acts as a tiebreaker. Somersault goes through following steps :

Failure Detection:
Processes communicate by passing heartbeat messages to one other. Failure is detected upon missing heartbeat.

Recovery Unit

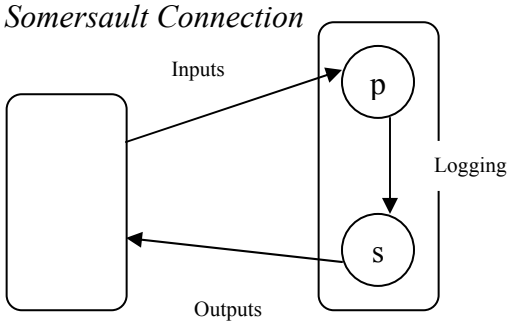




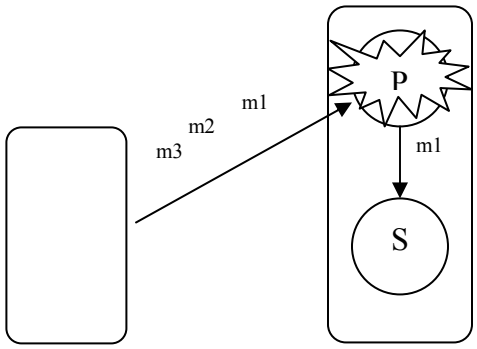
Replication:
Process pairs acts as replicas with logging channel between primary and secondary. Primary performs non-deterministic events that are made to perform by secondary identically.

Unit Communication and Secondary Sender Protocol:
All replicas consume input message and generate output messages but only one copy of each output is sent. The order of message is as shown in the figure besides.

Somersault Connection



Failure of Primary



Failover:
Windowing protocol is used at unit-to-unit communication level that ensures that messages are not lost or re-ordered in the event of the failing of primary or secondary. After failover of primary the secondary will receive the same input message and repeat the work

Recovery:
Upon failure of primary, secondary will become primary. Immediately it will replicate itself to create another secondary that will be synchronized with it.

8. Conclusion

Increasing application of critical application have lead to giving more focus and importance to fault tolerance. The area have gone more and more complex with advent of networked system and distributed application. Even after substantial research in theoretical as well as practical implementation this area is still is far from convincingly explored. Though there are some inherent limitations, fault tolerance is not impossible to implement. As we have seen formal approaches to deal with the problem using safety and liveness have helped us understand the concept well to implement it with exact understanding. Researchers have worked hard to formalize this field to make it more understandable. Queinnec [30] have tried to derive certain basic properties of a system to be fault tolerant. Arora [29] have tried to formalize building masking fault tolerant programs by use of component based method. They used stepwise approach by adding components to fault intolerant programs to transform it to non-masking fault tolerant and finally to masking one. But there are certain cases where detection is not possible before state of system transits to unsafe state and only its liveness may be guaranteed. Fault detection algorithms have successfully been implemented to solve consensus problems efficiently. In practice designing fault tolerance systems is more of compromise of design decisions with respect to replication level, protocols, fault detection algorithms, etc. Babaoğlu [31] has used probability of correct output as a criteria for evaluating and coming to decision regarding design.

Current practical implementation efforts are focused on areas such as providing fault tolerant middleware abstraction to better overall fault tolerance. Distributed Object Oriented Reliable Service (DOORS) developed by Natrajan [22] is a prototype implementation of Fault Tolerant CORBA Specification proposed by OMG. Current CORBA has few limitations in implementing processor based failure detection and recovery like overly coarse granularity, inability to restore complex object relationships and restrictions on process check pointing and recovery. DOORS implements subset of the FTC standards to achieve fault tolerance via replication, fault detection and recovery. Recent work also includes work on Cassel's [16] proposal to develop fault tolerant distributed computing and database system to solve extremely challenging problems of event processing, analysis and simulation of large data sets. The system is proposed to be scalable to hundreds of workstations and to expand fault tolerance and process group communication tools to wide-area networks. The project is currently under implementation and they have specified plan of action for five years. They plan to build prototype to understand the system working. Later they plan to concentrate on complex areas of fault tolerance, high-speed communication and performance measurements.

Future work needs to be done in implementation of methodologies described by Agha [32] for developing adaptable dependable systems. These systems may function over long duration despite a changing execution environment. It presents language framework for describing dependable systems. Finally we would like to say that task of designing and understanding fault tolerant systems are notoriously difficult and predicting and dealing with behavior of system activities and components is challenge which lies ahead.

References

[1] Felix C. Gartner - Fundamentals of Fault Tolerant Distributed Computing in Asynchronous Environments, ACM Computing Surveys (CSUR) March 1999

[2] Flavin Cristian – Understanding Fault-Tolerant Distributed Systems, Communications of the ACM, 1993

[3] Laprie J. C. 1985 – Dependable computing and fault tolerance: concepts and terminologies. In FTCS-15, 15th Symposium on Fault Tolerant Computing Systems (June 1985), pp. 2-11.

[4] Arora A. and Gouda M. 1993 – Closure and Convergence: a foundation of fault tolerant computing. IEEE Transactions on Software Engineering 19,11, 1015-1027.

[5] Aad P. A. van Moorsel – Action Models: A Reliability Modeling Formalism for Fault Tolerant Distributed Computation System, IPDS 1998

[6] Randy Chow and Theodore Johnson – Distributed Operating Systems and Algorithms, Addison Wesley Longman Inc. 1997.

[7] Jorg Kienzle – Combining Task and Transactions.

[8] Oded Goldreick, Silvio Micali and Avi Wigderson – How to Solve any Protocol Problem.

[9] Ran Canetti, Eyal Kushilertz, Rafail Ostrovsky and Adi Rosen – Randomness vs. Fault Tolerance, Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing August 1997.

[10] Tushar Deepak Chandra and Sam Toueg - Unreliable Failure Detectors for Reliable Distributed Systems, Journal of the ACM (JACM) March 1996

[11] Schneider, F. B. 1990. - Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Survey 22, 4 (Dec. 1990), 299 ± 319.

[12] Hadzilacos, V. and Toueg, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425. Department of Computer Science, Cornell University, Ithaca, NY.

[13] Dean Daniels, Roger L. Haskin, Jon Reinke, and Wayne Sawdon: Shared Logging Services for Fault-Tolerant Distributed Computing. Operating Systems Review 25(1): 65-68 (1991)

[14] James S. Plank, Henri Casanova, Jack J. Dongarra and Terry Moore – Netsolve: An Environment for Deploying Fault-Tolerant Computing, FTCS-28: 28th International Symposium on Fault-tolerant Computing, Munich, June, 1998

[15] Paul T. Murray, Roger A. Fleming, Paul D. Harry, Paul A. Vickers – Somersault: Enabling Fault-Tolerant Distributed Software Systems, HP Labs Technical Reports, Tech Reports: HPL-98-81.

[16] David G. Cassel – Distributed Computing and Databases for High Energy Physics, University of Cornell, Research Index Article.

[17] Taesoon Park and Heon Y. Yeom – Application Controlled Checkpointing Coordination for Fault-Tolerant Distributed Computing Systems, Parallel Computing 26(4), March 2000,pp.467-482.

[18] N. Liskov – "The Argus language and System. In Distributed Systems: Methods and Tools and Specification, LNCS< Vol 190 (M. Paul and H. Siegert, eds.), ch 7, pp. 343-430 Berlin: Springer-verlag, 1985.

[19] C. Ellis, J. Feldman and J. Heliotis, "Language constructs and support systems for distributed computing", in ACM Symposium on Principals of Distributed Computing, pp. 1-9, August 1982.
[20] R. Cmelik, N. Gehani and W.D. Roome – "Fault Tolerant Concurrent C: A tool for writing fault tolerant Distributed programs", in Proc. 18th Fault-Tolerant Computing Symposium, pp. 55-61, June 1988.
[21] Graham E. Fagg and Jack J. Dongarra - FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world, Lecture Notes in Computer Science, University of Tennessee, 2000.
[22] Balachandran Natrajan, Aniruddha Gokhale, Shalini Yajnik, Douglas C. Schmidt – DOORS: Towards High-Performance Fault Tolerant CORBA, Proceedings of the 2nd Distributed Applications and Objects (DOA) conference, Antwerp Belgium, Sept. 21-23, 2000.
[23] S.M. Yang, K.M. Kavi, A. Agarwalla, M. Reddy and S. Anam – SUVS: A Distributed Real-Time System Test bed for Fault Tolerant Computing, Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing, March 1992.
[24] Zhou and Andrzej Goscinski – Fault Tolerant Servers for the RHODOS Systems, Journal of Systems Software, 37(3), pp. 201-214, June 1997.
[25] Priya Venkitakrishnan – Rollback and Recovery Mechanisms in Distributed Systems, University of Texas at Arlington, Research Paper - March 2002.
[26] Deepak Chandra – Unreliable Failure Detectors for Asynchronous Distributed Systems, PhD Dissertation, Cornell University, May 1993.
[27] Carole Delporte-Gallet and Hugues Fauconnier - Greatest lower bounds for consensus using unreliable failure detectors, Research Index Article
[28] Jiong Yang, Gil Neige, Eli Gafni – Structured Derivations of Consensus Algorithms for Failure Detectors, Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing, June 1998
[29] Anish Arora , Sandeep S. Kulkarni - Designing masking fault-tolerance via nonmasking fault-tolerance, IEEE Transactions on Software Engineering, June 1998, Volume 24 Issue 6
[30] Philippe Queinnec and Gerard Padiou – Derivation of fault tolerance properties of distributed algorithms, Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing August 1994.
[31] Özalp Babaoğlu - On the reliability of consensus-based fault-tolerant distributed computing systems, ACM Transactions on Computer Systems (TOCS) October 1987.
[32] Gul Agha and Daniel C. Sturman – A Methodology for Adapting to Patterns of Faults, Foundations of Ultra dependability, Vol 1., Kluwe Academic, 1994.