A SURVEY OF SOFTWARE FAULT TOLERANCE TECHNIQUES

Zaipeng Xie*, Hongyu Sun* and Kewal Saluja*
*University of Wisconsin-Madison/Department of Electrical and Computer Engineering
1415 Engineering Drive, Madison WI 53706 USA

zxie2@wisc.edu, hongyusun@wisc.edu

Abstract: The paper surveys various software fault tolerance techniques and methodologies. The techniques include traditional techniques: recovery blocks (RcB), n-version programming, n self-checking Programming, retry blocks (RtB), n-copy programming and some new techniques: adaptive n-version systems, fuzzy voting, abstraction, parallel graph reduction, rejuvenation. The utility for each technique based on its attribution has also been presented.

1. Introduction

Software faults can be classified into two categories. Faults can be classified according to their phase of creation or occurrence, system boundaries (internal or external), domain (hardware or software), phenomenological cause, intent, and persistence. The discussion below is focused on software fault classification based on their recovery strategies. Gray [1] classifies software faults into Bohrbugs and Heisenbugs.

Software fault tolerance techniques are designed to allow a system to tolerate software faults that remain in the system after its development. Software fault tolerance techniques are employed during the procurement, or development, of the software. When a fault occurs, these techniques provide mechanisms to the software system to prevent system failure from occurring.

Software fault tolerance techniques provide protection against errors in translating the requirements and algorithms into a programming language, but do not provide explicit protection against errors in specifying the requirements. Software fault tolerance techniques have been used in the aerospace, nuclear power, healthcare, telecommunications and ground transportation industries, among others.

This paper includes four sections, major traditional software fault tolerance techniques are concluded in section2; and some new software fault tolerance techniques are discussed in section 3. This paper concludes in section 4.

2. Traditional Software Fault Tolerance Techniques

Software fault tolerance provides service complying with the relevant specification in spite of faults by typically using single version software techniques, multiple version software techniques, or multiple data representation techniques.

Single Version Software Environment: Monitoring techniques, atomicity of actions, decision verification, and exception handling are used to partially tolerate software design faults.

Multiple Version Software Environment: Design diverse techniques are used in a multiple version (or variant) software environment and utilize functionally equivalent yet independently developed software versions to provide tolerance to software design faults. Examples of such techniques include recovery blocks (RcB), N-version programming (NVP), and N self-checking programming (NSCP).

Multiple Data Representation Environment: Data diverse techniques are used in a multiple data representation environment and utilize different representations of input data to provide tolerance to software design faults. Examples of such techniques include retry blocks (RtB), N-copy programming (NCP) and N-selfchecking Programming.

It has been studied that the redundancy alone is not sufficient for tolerance of software design faults, so some forms of diversity must accompany the redundancy.

Diversity can be applied to several layers of the system- hardware, application software system software, operators, and the interfaces between these components. When Diversity is applied to more than one of these layers, it is generally termed multilayer diversity.

The main design diversity and data diversity techniques have been summarized in Table 1.

The environment diversity is to diversify the software operating circumstance temporarily. The typical examples of environment diversity technique are progressive retry, rollback rollforward recovery with checkpointing, restart, hardware reboot, etc.

2.1. Design Diversity

It is an identical service through separate design and implementations [2]. Since exact copies of software component redundancy cannot increase reliability in the face of software design faults, we need to provide diversity in the design and implementation of the software. Its goal is to make the modules as diverse and independent as possible to minimize the identical

error causes. We want the reliability of each variant as high as possible, so at least one variant will be operational at all times.

Design diversity begins with an initial requirements specification. The specification states the functional requirement of the software, when the decisions are to be made and upon which data the decision will be performed.

The variants perform their operations using these inputs. Since there are multiple results, this redundancy requires a means to decide which result to use. The variant outputs are examined by a decider or adjudicator. The adjudicator determines which variant result is correct or acceptable to forward to the next part of the software system. There are a number of decider algorithms available.

Recovery Blocks (RcB): The basic RcB scheme is one of the two original diverse software fault tolerance techniques. It was introduced in 1974 by Horning, et al. [3], with early implementations developed by Randell [A_3] in 1975 and Hecht [5] in 1981.The RcB is categorized as a dynamic technique. Its selection of a variant result to the output is made during program execution based on the result of the acceptance test (AT). The hardware fault-tolerant architecture related to the RcB scheme is stand-by sparing or passive dynamic redundancy.

RcB uses an AT and backward recovery to accomplish fault tolerance. We know that most program functions can be performed in more than one way, using different algorithms and designs. These differently implemented function variants have varying degrees of efficiency in terms of memory management and utilization, execution time, reliability, and other criteria. RcB incorporates these variants such that the most efficient module is located first in the series, and is termed the primary alternate or primary try block. The less efficient variant(s) are placed serially after the primary try block and are referred to as (secondary) alternates or alternate try blocks. Thus, the resulting rank of the variants reflects the graceful degradation in the performance of the variants.

The basic RcB scheme consists of an executive, an acceptance test, and primary and alternate try blocks (variants). Many implementations of RcB, especially for real-time applications, include a watchdog timer [6]. Figure 1 illustrates the structure and operation of the basic RcB technique with a watchdog timer.

the general syntax:

```
ensure      Acceptance Test
by          Primary Alternate
else by     Alternate 2
else by     Alternate 3
...
else by     Alternate n
else failure exception
```

The RcB syntax above states that the technique will first attempt to ensure the AT (e.g., pass a test on the acceptability of a result of an alternate) by using the primary alternate (or try block). If the primary algorithm's result does not pass the AT, then n-1 alternates will be attempted until an alternate's results pass the AT. If no alternates are successful, an error occurs.

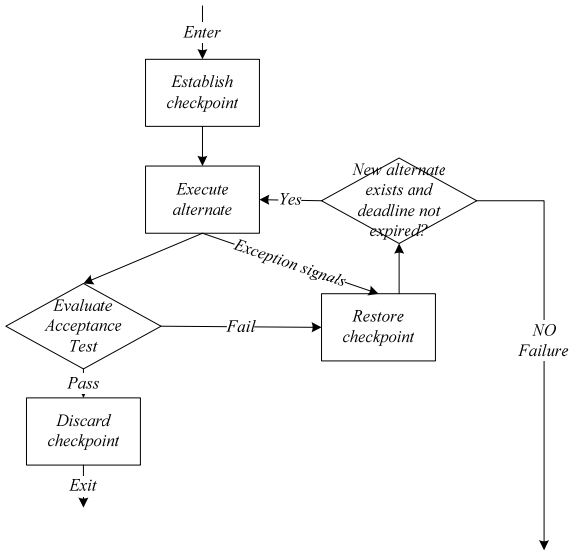

Figure 1: Recovery block structure and operation.

N-Version Programming: The NVP is one of the original design diverse software fault tolerance techniques. NVP was suggested by Elmendorf in 1972 [7] and developed by Avizienis and Chen [8, 9] in 1977–1978. Compared with RcB, NVP is s a static technique. That means a task is executed by several processes or programs and a result is accepted only if it is adjudicated as an acceptable result, usually via a majority vote. The hardware fault tolerance architecture related to the NVP is N-modular. The processes can run concurrently on different computers or sequentially on a single computer.

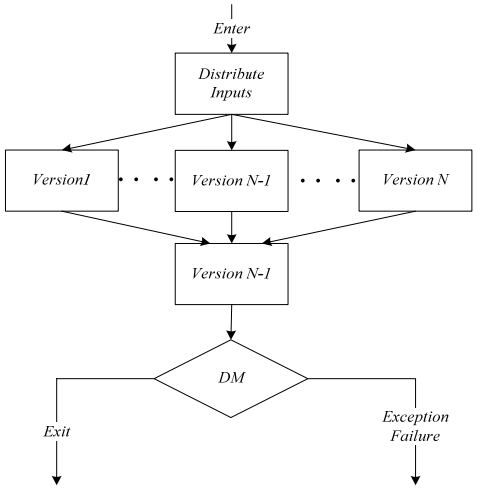

Figure 2: N-version programming structure.

The NVP technique uses a decision mechanism (DM) and forward recovery to accomplish fault

tolerance. The technique uses at least two independently designed, functionally equivalent versions (variants) of a program developed from the same specification. The variants are run in parallel and a DM examines the results and selects the "best" result, if one exists. There are many alternative decision mechanisms available for use with NVP.

General syntax:
run Version 1, Version 2, ..., Version n
if (Decision Mechanism (Result1, Result2,...,Result n))
    return Result
  else failure exception

The NVP syntax above states that the technique executes the n versions concurrently. The results of these executions are provided to the DM, which operates upon them to determine if a correct result can be adjudicated. If one can (i.e., the Decision Mechanism statement above evaluates to TRUE), then it is returned. If a correct result cannot be determined, then an error occurs.

*N Self-Checking Programming:*NSCP is a design diverse technique developed by Laprie, et al. [10, 20]. The hardware fault tolerance architecture related to NSCP is active dynamic redundancy. A self-checking program uses program redundancy to check its own behavior during execution. It results from either the application of an AT to a variant's results or from the application of a comparator to the results of two variants.

The NSCP hardware architecture consists of four components grouped in two pairs in hot standby redundancy, in which each hardware component supports one software variant. NSCP software includes two variants and a comparison algorithm or one variant and an AT on each hardware pair. When the NSCP executes, one of the self-checking components is the "active" component. The other components are "hot spares." Normally, the N in NSCP will be even, with the NSCP modules executed in pairs. But N can be odd, for instance, in the case where one variant is used in both pairs. Since the pairs are executed concurrently, there is an executive or consistency mechanism that controls any required synchronization of inputs and outputs. The self-checking group results are compared or otherwise assessed for correction. If there is no agreement, then the pair results are discarded. If there is agreement, then the pair results are compared with the other pair's results. NSCP failure occurs if both pairs disagree or the pairs agree but produce different results. NSCP is thus vulnerable to related faults between the variants.

The NSCP technique consists of an executive, n variants, and comparison algorithm(s). The executive orchestrates the NSCP technique operation, which has the general syntax (for n = 4):

run Variants 1 and 2 on Hardware Pair 1,
        Variants 3 and 4 on Hardware Pair 2
compare Results 1 and 2 compare Results 3 and 4
    if not (match)   if not (match)
        set NoMatch1   set NoMatch2
    else set Result Pair 1   else set Result Pair 2
if NoMatch1 and not NoMatch2, Result = Result Pair 2
    else if NoMatch2 and not NoMatch1, Result = Result Pair 1
    else if NoMatch1 and NoMatch2, raise exception
    else if not NoMatch1 and not NoMatch2
        then compare Result Pair 1 and 2
        if not (match), raise exception
        if (match), Result = Result Pair 1 or 2
    return Result

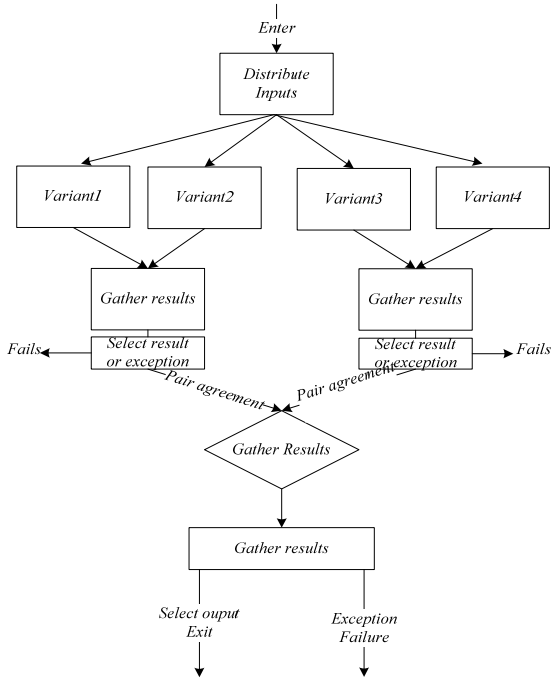

Figure 3: N self-checking programming structure.

The NSCP syntax above states that the technique executes the n variants concurrently, on n/2 hardware pairs. The results of the paired variants are compared. If any pair's results do not match, a flag is set indicating pair failure. If a single pair failure has occurred, then the nonfailing pair's results are returned as the NSCP result. If both pairs failed to match, then an exception is raised. If pair results match then the results of the pairs are compared. If they match, then the result is set as one of the matching values and returned as the NSCP result. If the result of the pair matches does not match, then an exception is raised. NSCP operation is illustrated in Figure 3.

2.2. Data diversity

Due to the limitations of some design diverse techniques, it led to the development of data diverse software fault tolerance techniques. The data diverse

techniques are meant to complement, rather than replace, design diverse techniques.

Ammann and Knight [A-11-23] proposed data diversity as a software fault tolerance strategy to complement design diversity. The employment of data diversity involves obtaining a related set of points in the program data space, executing the same software on those points, and then using a decision algorithm to determine the resulting output. Data diversity is based on a generalization of the works of Gray, Martin and Morris [15-17], which utilize data diverse approaches relying on circumstantial changes in execution conditions. These execution conditions can be changed deliberately to effect data diversity. This is done using data expressions to obtain logically equivalent variants of the input data. Data diversity use data re-expression algorithms (DRAs) to obtain their input data.

Data re-expression Algorithm: The performance of data diverse software fault tolerance techniques depends on the performance of the re-expression algorithm used. There are several ways to perform data re-expression and provide some insight on actual re-expression algorithms and their use. DRAs are very application dependent. Development of a DRA also requires a careful analysis of the type and magnitude of re-expression appropriate for each data that is a candidate for all re-expression.

Data re-expression is used to obtain diverse input data by generating logically equivalent input data sets. Given initial data within the program failure region, the re-expressed input data should exist outside that failure region. A re-expression algorithm, R, transforms the original input x to produce the new input, y=R(x). The input y may either approximate x or contain x's information in a different form. The program, P, and R determine the relationship between P(x) and P(y).

Not all applications can employ data diversity. Those that cannot do so include applications in which an effective DRA cannot be found. This may include: applications that do not primarily use numerical data, some that use primarily integer data, some for which an exact re-expression algorithm is required, those for which a DRA that escapes the failure region cannot be developed, and those for which the known re-expression algorithms that escape the failure region are resource-ineffective.

Retry Blocks (RtB): The basic RtB technique is one of the two original data diverse software fault tolerance techniques developed by Ammann and Knight [18-29]. The RtB technique is also categorized as a dynamic technique. The hardware fault tolerance architecture related to the RtB technique is stand-by sparing or passive dynamic redundancy. It is the data diverse complement of the recovery block (RcB) scheme.

The RtB technique uses acceptance tests (AT) and backward recovery to accomplish fault tolerance. A watchdog timer(WDT) is also used and triggers execution of a backup algorithm if the original algorithm does not produce an acceptable result within a specified period of time. The algorithm is executed using the original system input. The primary algorithm's results are examined by an AT. If the algorithm results pass the AT, then the RtB is complete. However, if the results are not acceptable, then the input is re-expressed and the same primary algorithm runs again using the new, re-expressed, input data. This continues until the AT finds an acceptable result or the WDT deadline is violated. If the deadline expires, a backup algorithm may be invoked to execute on the original input data.

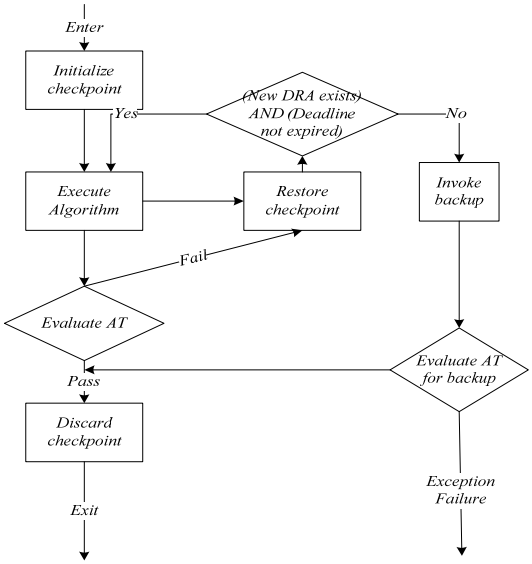

Figure 4: Retry block structure and operation.

The RtB technique consists of an executive, an AT, a DRA, a WDT, and primary and backup algorithms. The executive orchestrates the operation of the RtB, which has the general syntax:

```
ensure      Acceptance Test
by          Primary Algorithm(Original Input)
else by     Primary Algorithm(Re-expressed Input)
else by     Primary Algorithm(Re-expressed Input)
...
...         [Deadline Expires]
else by     Backup Algorithm(Original Input)
else failure exception
```

The RtB syntax above states that the technique will first attempt to ensure the AT (e.g., pass a test on the acceptability of a result of the algorithm) by using the primary algorithm. If the primary algorithm's result does not pass the AT, then the input data will be re-expressed and the same algorithm attempted until a result passes the AT or the WDT deadline expires. If the deadline expires, the backup algorithm is invoked with the original inputs. If this backup algorithm is not successful, an error occurs.

N-Copy Programming: NCP, also developed by Ammann and Knight [18-29], is the other original data

diverse software fault tolerance technique. NCP is a data diverse technique, and is further categorized as a static technique. The hardware fault tolerance architecture related to the NCP is N-modular or static redundancy. The processes can run concurrently on different computers or sequentially on a single computer, but in practice, they are typically run concurrently. NCP is the data diverse complement of N-version programming (NVP).

The NCP technique uses a decision mechanism (DM) and forward recovery to accomplish fault tolerance. The technique uses one or more DRAs and at least two copies of a program. The system inputs are run through the DRA(s) to re-express the inputs. The copies execute in parallel using the re-expressed data as input. A DM examines the results of the copy executions and selects the "best" result, if one exists. There are many alternative DMs available with NCP.

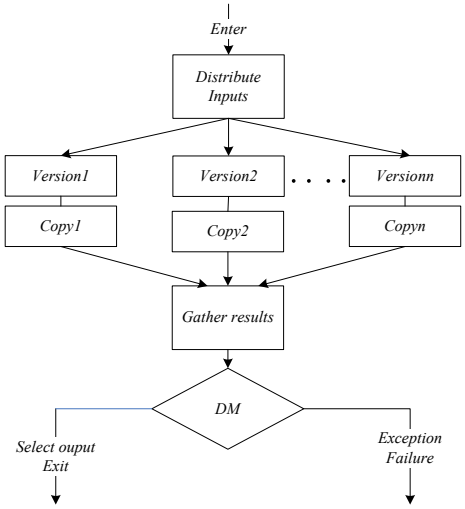

Figure 5: N-copy programming structure.

The basic NCP technique consists of an executive, 1 to n DRA, n copies of the program or function, and a DM. The executive orchestrates the NCP technique operation, which has the general syntax:

```
run DRA 1, DRA 2, ..., DRA n
run Copy 1(result of DRA 1),
   Copy 2(result of DRA 2), ...,
   Copy n(result of DRA n)
if (Decision Mechanism (Result 1, Result 2, ...,
         Result n))
   return Result
else failure exception
```

The NCP syntax above states that the technique first runs the DRA concurrently to re-express the input data, then executes the n copies concurrently. The results of the copy executions are provided to the DM, which operates upon the results to determine if a correct result can be adjudicated. If one can (i.e., the Decision Mechanism statement above evaluates to TRUE), then it is returned. If a correct result cannot be determined, then an error occurs.

3. New Software Fault Tolerance Techniques

The new software fault tolerance techniques could be either improvement versions of some traditional techniques (such as adaptive N-version systems, fuzzy voting and Byzantine fault tolerance with abstract specification encapsulation, graph reduction) or techniques based on new concepts that are not categorizable as redundancy or diversity (such as rejuvenation).

Adaptive N-version programming: A modification of classical n-version systems was discussed in [12], which considers an adaptive approach to model and manage different quality levels of the versions by introducing an individual weight factor to each version of the n-version system. This weight factor is then included in the voting procedure, i.e. the voting is based on a weighted counting of the number of monitored events for the deviation behaviour of the individual version. The voting procedure can be adaptively modified and tailored to the fault state of the overall system (Figure 6). The philosophy is mainly adopted from the self-purging redundancy, so the developed approach is also called Adaptively-Purging Redundancy (APR).

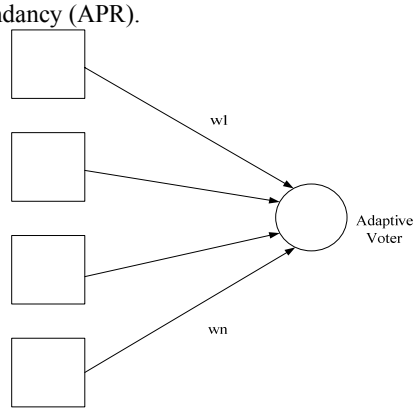

Figure 6: Adaptive voting for *n*-version programming: versions have dynamically changeable weights.

The reliability of the *n*-version system with adaptively adjusted weight factors is given by

$$R_c = R_v\left(\sum_{k=0}^{K} \sum_{j_1,..,j_k,..j_n} (1-R_{j_1})..(1-R_{j_j})R_{j_{k+1}}...R_{j_n}\right)$$

Where R_v is the reliability of the voter and the $K = \max\{k\}$ when the sum of the weights of the n-k correct versions is greater than that of the k faulty version, i.e.,

$$\sum_{r=1}^{K} w_{j_r} < \sum_{r=k+1}^{n} w_{j_r}$$

The numerical simulation results (Figure 7) for the example of a satellite control system consisting of seven components which are all implemented by 3 versions indicates that the change of weight improves the software reliability.

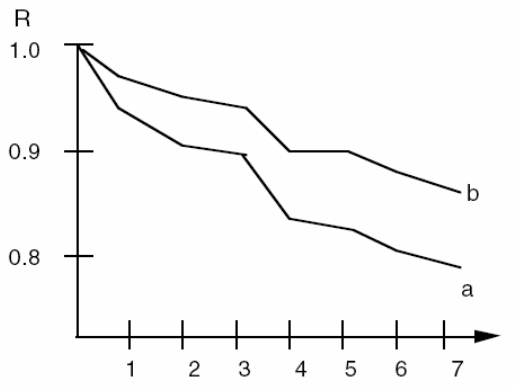

Figure 7: Cumulated reliability of 7 components of a software system for a satellite control application [12]. a) classical 3-version system b) 3-version system with weight factors adapted to monitored failures.

Additionally, the adaptive approach was extended to a component-based *n*-version system. Considering a system with a number of *I* tasks each of which assigned to each module *i* (*i*=1,…,*I*) and each module has *n* versions, the selected *n* versions of module *I* together form the module stage *i* of the system. The reliability of the *I*-stage modular n-version system is

$$R_{\mathrm{mod},I,n} = \prod_{i=1}^{I} R_{mi,n}$$

where $R_{mi,n}$ is the reliability of module stage *i* comprising *n* version modules.

By introducing adaptive voting to the output of each module stage, the reliability can be further increased (Figure 8).

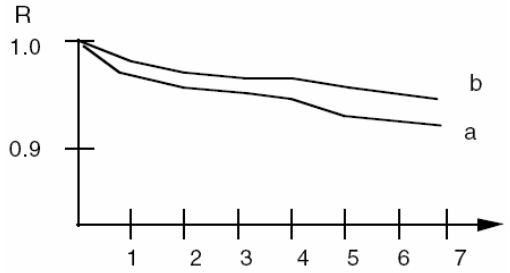

Figure 8: Cumulated reliability of 7 components of a software system for a satellite control application [12]. a) 3-version system with modular construction of the versions b) 3-version system with modular construction of the version and additional adaptive adjustment of module weights.

Fuzzy Voting: Voting strategies are used to select the correct output from different outputs obtained from different redundant software versions. Traditional voting methods are based on an output classification or partitioning of the outputs into disjoint subsets. Elements within these subsets are "equal" within certain tolerance. N version programming with majority voting (NVP-MV) selects the correct answer based on majority number of members. N version programming with consensus voting (NVP-CV) selects the correct answer based on largest number of members. Maximum likelihood voting (MLV) selects the output with the largest success likelihood.

Methods not based on output classification use convergence functions: midpoint, median, average, weighted average etc..

All concepts assume the independence of faults in redundant software and sufficiently small probability of coincidental faults. In traditional voting, equality relation regards two real numbers as equal if their difference is smaller than fixed tolerance ε. For different version outputs that are "closer" to each other than the fixed threshold there is no gradual comparison. As a result, certain interconnection of faults could incur incorrect selection. [22] proposed a fuzzy extension of classical numerical equivalence relation to overcome those potential problems of traditional equivalence relation, applied to real numbers.

The equality relation of a version output set can be represented by the agreement matrix, which is a Boolean matrix with each element is defined as

$$r_{i,j} = \begin{cases} 1, & if\ |x_i - x_j| \le \varepsilon \\ 0, & otherwise \end{cases}$$

The fuzzy logic maps the input vector into an output nonlinearly (Figure 9), which can be defined as:

$$\mu_{A_i}(x_i) = \begin{cases} 1 - \dfrac{|a_i - x_i|}{\varepsilon/2}, & if\ |x_i - a_i| \le \varepsilon/2 \\ 0, & otherwise \end{cases}$$

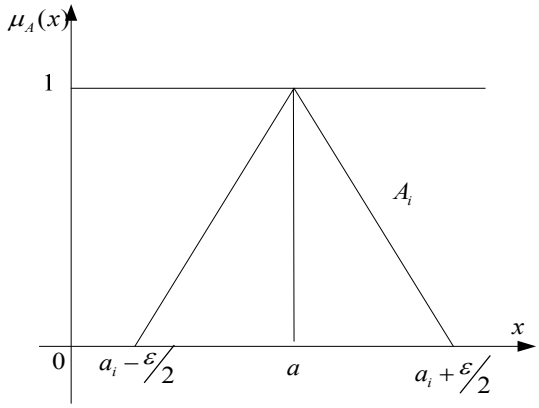

Figure 9: Triangular shape of fuzzy set outputs.

Fuzzy relation is the degree of interconnecting elements of sets that comprise the relation. A fuzzy relation is a fuzzy equivalence relation if and only if all three properties of fuzzy relations are satisfied: reflexivity, symmetry and transitivity.

Fuzzy equivalence relation results in more reliable systems [22].

Reconfiguration and Rejuvenation: Reconfiguration and Rejuvenation are complementary ways of software fault tolerance (Figure 10). Reconfiguration is reactive, analogous to event-driven interruption process in computer communications; Rejuvenation is proactive, analogous to polling process.

Software reconfiguration can use redundant resources for real-time recovery while dynamically considering a large number of factors (operating system services, processor load, and memory variables among others).

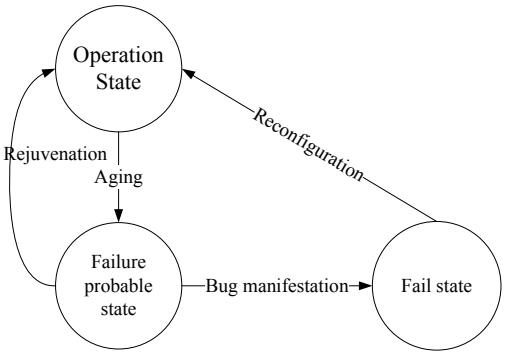

Figure 10: Complementary nature of reconfiguration and rejuvenation [23].

Rejuvenation: A novel approach to handle transient software failures due to software aging is called software rejuvenation, which can be regarded as a preventive and proactive solution that is particularly useful for counteracting the aging phenomenon. It involves stopping the running software occasionally, cleaning its internal state and restarting it. Cleaning the internal state of a software system might involve garbage collection, flushing operating system kernel tables, reinitializing internal data structures, etc. An extreme, but well known example of rejuvenation which has been around as long as computers themselves is a hardware reboot.

A fault-tolerant software system with two-version redundant structure and random rejuvenation schedule was studied in [15] and the steady-state system availability was evaluated quantitatively based on the familiar Markovian analysis. Figure 11 demonstrated a Markov transition diagram for a single-version software system with rejuvenation proposed by Huang et al. [6]. The states are defined as:

State 0: highly robust state (normal operation state)

State 1: failure probable state

State 2: system failure state

State 3: software rejuvenation state.

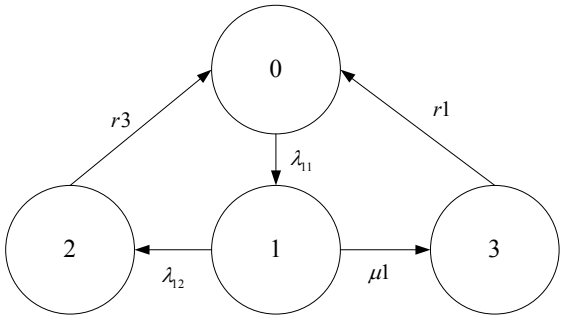

Figure 11: Markovian transition diagram for a single version software system with rejuvenation [15].

The steady-state system availability was derived as:

$$A = \frac{\frac{1}{\lambda_{11}} + \frac{1}{\lambda_{12}+\mu_1}}{\frac{1}{\lambda_{11}} + \frac{1}{\lambda_{12}+\mu_1} + \frac{\lambda_{12}}{(\lambda_{12}+\mu_1)r_3} + \frac{\mu_1}{(\lambda_{12}+\mu_1)r_1}}$$

Figure 12 shows the numerical simulation results which indicate that the steady-state system availability increases with software rejuvenation rate for a single version software system.

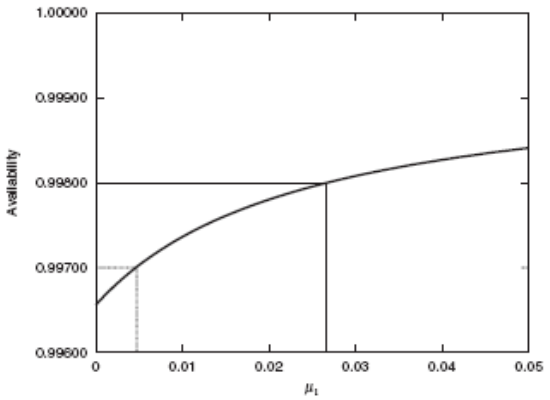

Figure 12: The steady-state system availability versus software rejuvenation rate for a single version software system [15].

Similarly, the results for a two version software system show that its steady-state system availability increases with software rejuvenation rate (Figure 13).

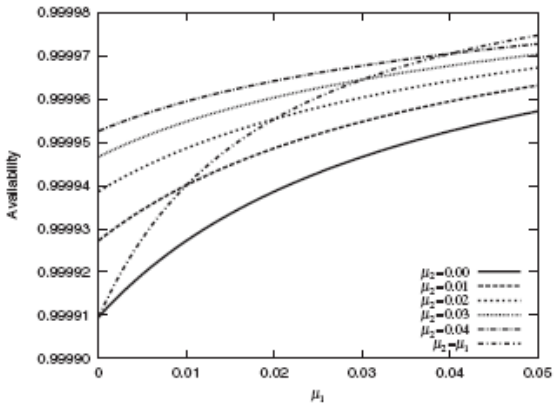

Figure 13: Dependence of software rejuvenation rate on the steady-state system availability for two version software system [15].

Abstraction: Software fault tolerance using replication is expensive to deploy. [17] proposed a replication technique, Byzantine fault tolerance (BFT) with Abstract Specification Encapsulation (BASE), which uses abstraction to reduce the cost of Byzantine fault tolerance and to improve its ability to mask software errors.

Byzantine fault tolerance allows a replicated service to tolerate arbitrary behaviour from faulty replicas—behaviour caused by a software bug or an attack. Abstraction hides implementation details to enable the reuse of off-the-shelf implementations of important services (e.g., file systems, databases, or HTTP daemons) and to improve the ability to mask software errors.

BASE reduces cost because it enables reuse of off-the-shelf service implementations. It improves availability because each replica can be repaired periodically using an abstract view of the state stored by correct replicas, and because each replica can run distinct or nondeterministic service implementations, which reduces the probability of common mode failures.

There is also a proactive recovery mechanism for BFT that recovers replicas periodically even if there is no reason to suspect that they are faulty. This allows the replicated system to tolerate any number of faults over the lifetime of the system provided fewer than one-third of the replicas become faulty within a window of vulnerability.

A global view of all BASE functions and upcalls that are invoked is shown in Figure 14.

BASE implements a form of state machine replication that requires replicas to behave deterministically. Our methodology uses abstraction to hide most of the nondeterminism in the implementations it reuses.

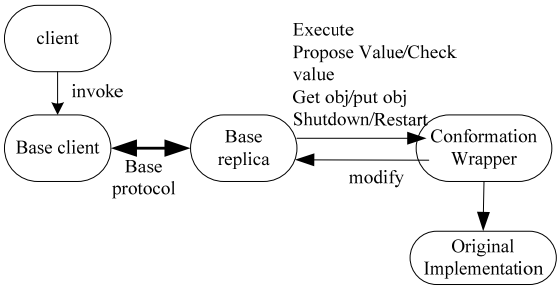

Figure 14: A global view of all BASE functions and upcalls that are invoked.

However, many services involve forms of nondeterminism that cannot be hidden by abstraction. For instance, in the case of the NFS service, the time-lastmodified for each file is set by reading the server's local clock. If this were done independently at each replica, the states of the replicas would diverge.

Instead, BASE allows the primary replica to propose values for non-deterministic choices by providing the propose value upcall, which is only invoked at the primary. The call receives the client request and the sequence number for that request; it selects a non-deterministic value and puts it in non-det. This value is going to be supplied as an argument of the execute upcall to all replicas.

The protocol implemented by the BASE library prevents a faulty primary from causing replica state to diverge by sending different values to different backups. However, a faulty primary might send the same, incorrect value to all backups, subverting the system's desired behavior. The solution to this problem is to have each replica implement a check value function that validates the choice of non-deterministic values that was made by the primary. If one-third or more non-faulty replicas reject a value proposed by a faulty primary, the request will not be executed and the view change mechanism will cause the primary to be replaced soon after.

Proactive recovery periodically restarts each replica from a correct, up-to-date checkpoint of the abstract state that is obtained from the other replicas. Recoveries are triggered by a watchdog timer. When this timer fires, the replica reboots after saving to disk the abstract service state, and the replication protocol state, which includes abstract objects that were copied by the incremental checkpointing mechanism. The library could invoke get obj repeatedly to save a complete copy of the abstract state to disk but this would be expensive. It is sufficient to ensure that the current concrete state is on disk and to save a small amount of additional information to enable reconstruction of the conformance representation when the replica restarts. Since the library does not have access to this representation, the service state is saved to a file by an additional upcall, shutdown, that is implemented by the conformance wrapper. The conformance wrapper also implements a restart upcall that is invoked to reconstruct the conformance representation from the file saved by shutdown and

from the concrete state of the service. This enables the replica to compute the abstract state by calling get obj after restart completes.

The performance results indicate that the overhead introduced by this abstraction technique is low [17].

Parallel Graph Reduction: Recently, parallel computing is popularly applied to many systems. Functional programming is suitable for parallel programming because of its referential transparency and the independency among each program. Referential transparency means that all references to the value are therefore equivalent to the value itself and the fact that the expression may be referred to from other parts of the program is of no concern. It is applied to symbol processing systems and parallel database systems. Programs of some functional programming can be regarded as graphs and are processed in terms of reduction of the corresponding graphs.

A fault tolerance scheme based on parallel graph reduction in functional programming was proposed. The method is a class of receiver-based message logging and time overhead of fault tolerance is reduced by taking advantage of referential transparency.

This new approach based on the graph reduction stores the received graph as a message log and the erroneous task is recovered by using the checkpoint and the stored graph.

The program is executed by reducing the graph, which is divided into subgraphs. The subgraph is assigned to each node and is reduced in parallel. Figure below shows an example of parallel graph reduction.

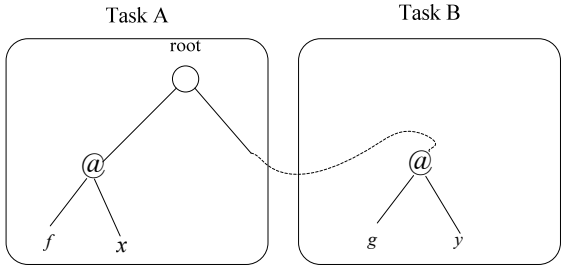

Figure 15: An example of parallel graph reduction.

The model assumes that

- There exists at most one faulty task each time.
- The faulty task can be detected.
- Only tasks can be faulty. Network is error free.

Each task not only reduces the assigned graph but also has the backup of other nodes. The backup of the subgraph corresponds to the log in the message logging method. However, the order of the received subgraph needs not to be registered because of referential transparency.

(1) Transmission. A task transmits a subgraph of its graph to another task for parallel reduction. The task receiving the subgraph recognized the transmitted messages as a subgraph by the tag included in the message.
(2) Backup. The task sends the received subgraph to its backup task. The task stores the received subgraph into its storage area.
(3) Reduction. The task begins to reduce the received subgraph.

In addition to the backup, each task takes its check-point constantly. The checkpoint consists of all graphs reduced by the task, information about the reduction process and the relationship between its graphs and the one in the other tasks. The check point is sent to the backup task of the task.

An error recovery uses a checkpoint and a subgraph stored in the erroneous task. The erroneous task is restored by the checkpoint and the subgraph is retransmitted to the restored task. Then the restored task begins to reduce the subgraph.

(1) Roll back. The checkpoint stored in the backup task of the erroneous tasks is sent to the erroneous task and the erroneous task is restored.
(2) Retransmission. The backup of the subgraphs for the erroneous tasks is transmitted by the backup task. The subgraphs are the ones which the erroneous tasks received after the last checkpoint was taken.
(3) Checkpointing. The checkpoint of the restored task is taken and sent to the backup task.
(4) Reduction. The restored task begins to reduce the subgraph.

4. Conclusions

We surveyed and compared various software fault tolerant techniques. First, we summarized traditional techniques with diversity implementations. Then, we addressed some new techniques which either improved the traditional techniques or took a new approach to solve the problem of software fault tolerance.

In conclusion, a lot of techniques have been developed for achieving fault tolerance in software. The application of all of these techniques is relatively new to the area of fault tolerance. Furthermore, each technique will need to be tailored to particular applications. This should also be based on the cost of the fault tolerance effort required by the customer. The differences between each technique provide some flexibility of application.

References

[1] GRAY, J. (1986): 'Why Do Computers Stop and What Can Be Done About It?', Proc. Fifth Symp. Reliability in Distributed Software and Database Systems, Jan. 1986, pp. 3-12

[2] AVIZIENIS, A., and J. P. J. KELLY, "Fault Tolerance by Design Diversity: Concepts and Experiments," IEEE Computer, Vol. 17, No. 8, 1984, pp. 67–80.

[3] HORNING, J. J., et al., "A Program Structure for Error Detection and Recovery," in E. Gelenbe and

C. Kaiser (eds.), Lecture Notes in Computer Science, Vol. 16, New York: Springer-Verlag, 1974, pp. 171–187.

[4] RANDELL, B., "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, 1975, pp. 220–232.

[5] HECHT, M., AND H. HECHT, "Fault Tolerant Software Modules for SIFT," SoHaR, Inc. Report TR-81-04, April 1981.

[6] HECHT, H., "Fault Tolerant Software for Real-Time Applications," ACM Computing Surveys, Vol. 8, No. 4, 1976, pp. 391–407.

[7] ELMENDORF, W. R., "Fault-Tolerant Programming," Proceedings of FTCS-2, Newton, MA, 1972, pp. 79–83.

[8] AVIZIENIS, A., "On the Implementation of N-Version Programming for Software Fault-Tolerance During Execution," COMPSAC '77, Chicago, IL, 1977, pp. 149–155.

[9] CHEN, L., and A. AVIZIENIS, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Proceedings of FTCS-8, Toulouse, France, 1978, pp. 3–9.

[10] LAPRIE, J. -C., et al., "Hardware and Software Fault Tolerance: Definition and Analysis of Architectural Solutions," Proceedings of FTCS-17, Pittsburgh, PA, 1987, pp. 116–112.

[11] LAPRIE, J. -C., et al., 'Definition and Analysis of Hardwareand Software-Fault-Tolerant Architectures', IEEE Computer, Vol. 23, No. 7, 1990, pp. 39–51.

[12] KANOUN, K., et al., "Reliability Growth of Fault-Tolerant Software," IEEE Transactions on Reliability, Vol. 42, No. 2, 1993, pp. 205–129.

[13] TOMEK, L. A., J. K. MUPPALA, and K. S. Trivedi, "Modeling Correlation in Software Recovery Blocks," IEEE Transactions on Software Engineering, Vol. 19, No. 11, 1993, pp. 1071–1086.

[14] LYU, M. R. (ed.), Software Fault Tolerance, New York: John Wiley & Sons, 1995.

[15] TRIVEDI, T. S., Probability and Statistics with Reliability, Queuing, and Computer Science Applications, Englewood Cliffs, NJ: Prentice-Hall, 1982.

[16] DEB, A. K., and A. L. GOEL, "Model for Execution Time Behavior of a Recovery Block," Proceedings COMPSAC '86, Chicago, IL, 1986, pp. 497–502.

[17] DEB, A. K., "Stochastic Modeling for Execution Time and Reliability of Fault-Tolerant Programs Using Recovery Block and N-Version Schemes," Ph.D. thesis, Syracuse University, 1988.

[18] AMMANN, P. E., "Data Diversity: An Approach to Software Fault Tolerance," Proceedings of FTCS-17, Pittsburgh, PA, 1987, pp. 113–117.

[19] AMMANN, P. E., "Data Diversity: An Approach to Software Fault Tolerance," Ph.D. dissertation, University of Virginia, 1988.

[20] AMMANN, P. E., and J. C. KNIGHT, "Data Diversity: An Approach to Software Fault Tolerance," IEEE Transactions on Computers, Vol. 37, No. 4, 1988, pp. 418–416.

[21] GROSSPIETSCH, K.E. and SILAYEVA, T.A. (2003): 'An adaptive approach for n-version systems', Parallel and Distributed Processing Symposium, 2003. Proceedings. International 13-17, April 2003

[22] MANIC, M. and FRINCKE, D. (2001): 'Towards the fault tolerant software: fuzzy extension of crisp equivalence voters', Industrial Electronics Society, 2001. IECON '01. The 18th Annual Conference of the IEEE Vol. **1**, 29 Nov.-2 Dec. 2001 pp.84 - 89.

[23] YURCIK, W. and DOSS, D. (2001): 'Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration', IEEE software, July/August 2001, pp.48-52.

[24] RINSAKA, K. and AND DOHI, T. (2005): 'Behavioral Analysis of a Fault-Tolerant Software System with Rejuvenation', Autonomous Decentralized Systems, 2005. ISADS 2005. Proceedings, 4-8 April 2005, pp. 159 – 166.

[25] HUANG, Y., KINTALA, C., KOLETTIN, N., and FUNTON, N. D. (1995): 'Software rejuvenation: analysis, module and applications', Proc. 16th Int'l Symp. on Fault Tolerant Computing, IEEE CS Press, 1995, pp. 381-390.

[26] CASTRO, M., RODRIGUES, R., and LISKOV, B. (2003): 'BASE: Using Abstraction to Improve Fault Tolerance', ACM Transactions on Computer Systems, Vol. **12**, No. 3, August 2003, pp.236-179.

Table 1. Summary of Traditional Software Fault Tolerant Techniques

Software Fault tolerance Techniques		Abbreviation	Error Processing Techniques
Design Diversity	Recovery Blocks	RcB	Error detection by acceptance test (AT) and backward recovery
	N-Version Programming	NVP	Vote
	Distributed Recovery Blocks	DRB	Error detection and result switching, can be detected by AT, or by comparison
	N Self-Checking Programming	NSCP	Error detection by AT and forward recovery
	Consensus recovery Block	CRB	Vote then AT
	Acceptance Voting	AV	AT first then Vote
	Retry Blocks	RtB	Acceptance test and backward recovery
Data Diversity	N-Copy Programming	NCP	Run the same process concurrently or sequentially
	Two-Pass Adjusdicators	TPA	Use both data and design diversity to handle multiple correct results (MCR)