The Reliable Hybrid Pattern
A Generalized Software Fault Tolerant Design Pattern

Fonda Daniels
Department of Electrical & Computer Engineering, Box 7911
North Carolina State University
Raleigh, NC 27695
email: fjdaniel@eos.ncsu.edu

Kalhee Kim and Mladen A. Vouk
Department of Computer Science, Box 8206
North Carolina State University
Raleigh, NC 27695
email: kkim1@eos.ncsu.edu, vouk@csc.ncsu.edu

Abstract

In order to make software applications more reliable, it may be necessary to incorporate into them a strategy for tolerating software faults. In the case of critical applications, this is essential. In this paper we describe a general pattern, which we call the Reliable Hybrid pattern, that can be used to design fault tolerant software applications. The pattern supports development of applications based on classical fault tolerant strategies such as N-Version Programming and Recovery Block, as well as those based on advanced hybrid techniques such as Consensus Recovery Block, Acceptance Voting, and N-Self Checking Programming. The Reliable Hybrid pattern structure reflects the fact that these advanced strategies can be constructed through recursive combination of N-Version Programming, Recovery Block, and hybrid components. Our pattern explicitly indicates how different advanced software voting techniques fit into the design structure. We use Consensus Recovery Block to illustrate the behavior of the designs that can be constructed using the Reliable Hybrid pattern, and we discuss some general issues related to the use of fault-tolerant software.

Key words: Reliable Hybrid pattern, fault tolerant software, N-Version Programming, Recovery Block.

1.0 Introduction

The technique of using redundant software modules as protection against software faults was inherited from hardware [Nels87]. However, unlike hardware failures, software faults are for the most part the result of software specification and design errors, thus simple replication of software components does not provide reasonable protection. This dictates designs and development methods that use diverse algorithms in the redundant components in order to minimize the potential for common-cause faults and failures. The use of redundancy requires the ability to judge acceptability of the outputs of several modules, either by direct evaluation or by comparison, and subsequent selection of one of the outputs as the "correct" output. The algorithm that compares, evaluates and selects the "correct" output is called an adjudicator.

Depending on the application, adjudication algorithms range from very simple to very complex. There are several adjudication algorithms which have been proposed. This includes voting, selection of the median value, and acceptance testing as well as more complex decision making. Two common fault-tolerant software schemes are N-Version Programming (voter-based) [Aviz85] and Recovery Block (acceptance test-based) [Rand75]. Both of these schemes are based on software component redundancy and on the assumption that correlated coincident failures of these components are rare.

Voter-Based

N-Version Programming (NVP) proposes parallel execution of independently developed functionally equivalent software versions with adjudication of their outputs by a voter. The voter accepts all N outputs as inputs and uses these to determine the correct, or best, output if one exists. Note that NVP is commonly used to refer to voting-based fault tolerance software. Table 1 lists several examples of voter-based algorithms. There are many other voting techniques that may be of practical interest to the reader [Lyu95, Lyu96].

Acceptance Test-Based

An acceptance test is an adjudicator which is a program-specific mechanism that checks the correctness of the program results. An acceptance test might only consists of bounds or simple tests to determine acceptability. An example of an acceptance test-based technique is listed in Table 1.

Hybrid

There are also ways of combine the preceding simple techniques to create hybrid fault tolerance approaches. Examples of hybrid strategies which combine both voter-based and acceptance-based techniques are listed in Table 1. There are many other hybrid techniques [Lyu95, Lyu96].

Table 1: Fault Tolerance Techniques

Adjudication Method	Algorithm	Description	Reference
Voter	NVP with Majority Voting (MV)	In an m out of N fault tolerant software system, m is the agreement number or the number of matching outputs which the adjudication algorithm requires to make a decision. $m = \left\lceil \frac{N+1}{2} \right\rceil$	[Aviz85, Lyu96]
	NVP with Consensus Voting (CV)	Generalization of majority voting. In consensus voting the result with the largest agreement number is chosen as the correct answer. If there are several outputs with the same agreement number, the tie can be broken in several ways.	[McAl90, Vouk93]
	NVP with Maximum Likelihood Voting (MLV)	Takes the reliability of each software version into consideration and determines the most likely correct answer. Practical implementation of MLV requires certain modification to the MLV algorithm.	[Leun95, Kim96]

Table 1: Fault Tolerance Techniques

Acceptance-Test	Recovery Block (RB)	Independently developed functionally equivalent versions are executed in sequence or in parallel, and the output of each is adjudicated by an acceptance test. The first version that passes the acceptance test is chosen as the correct answer. The complexity of the acceptance test can vary widely.	[Rand75, Lyu95, Lyu96]
Hybrid	Consensus Recovery Block (CRB)	Executes NVP first, followed by RB. If NVP fails the system resorts to RB in an attempt to find the correct answer. The strategy fails if both NVP and RB fail.	[Scot87]
	Acceptance Voting (AV)	The output of each module is first presented to an acceptance test and only those modules that pass the acceptance test are used to vote on the correctness of the answers.	[Bell91, Lyu96]
	N-Self Checking Programming (NSCP)	N modules are executed in pairs for all N. Outputs of each pair are tested and if they do not agree with each other, the responses of the pair are discarded. If they agree the outputs are compared again.	[Lapr90]

An object oriented approach to software fault tolerance has previously been discussed in [Xu95]. NVP and RB have been previously implemented as Master-Slave [Busc95] and Backup [Subr95] design patterns respectively. The pattern design presented in this paper is more general. It allows reuse of these two patterns, as well as construction of more complex hybrid solutions. We call it the Reliable Hybrid pattern. The Reliable Hybrid Pattern is based on the use of the Composite design pattern to recursively combine the Master-Slave and Backup design pattern. Combining recursive definition of the adjudicator instantiations permits a range of fault tolerance strategies, from the very simplest to the much more complex solutions.

2.0 The Pattern

2.1 Intent

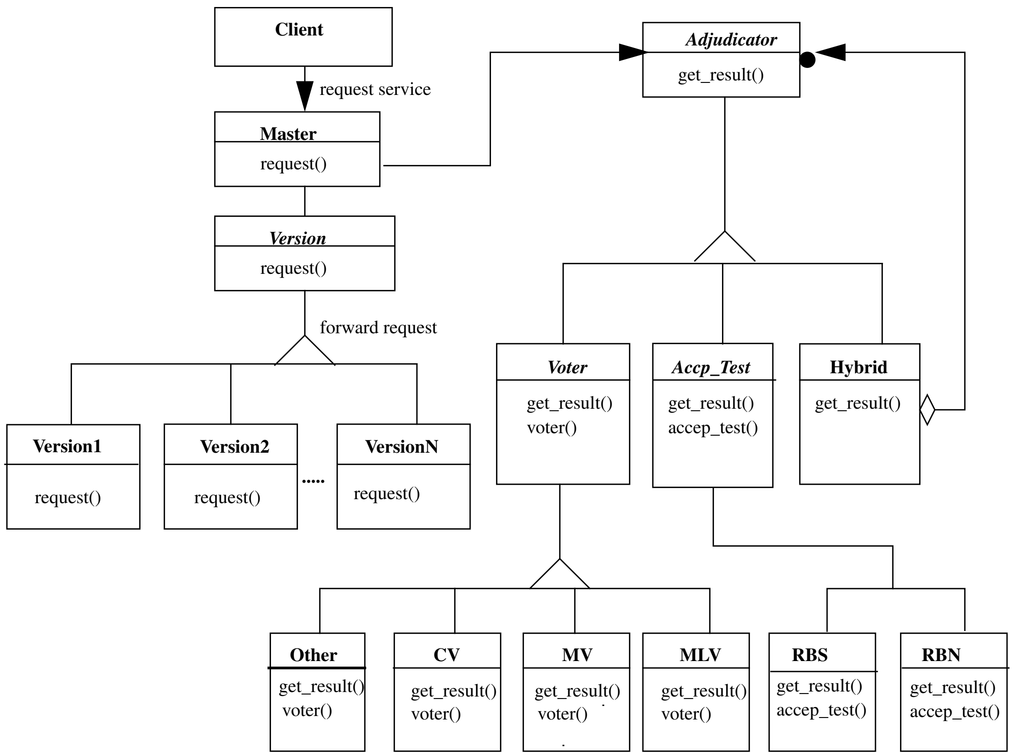
The Reliable Hybrid pattern provides a general object-oriented framework for fault-tolerance which can range from basic approaches (e.g. NVP and RB) to complex hybrids (e.g. CRB, AV, NSCP).

2.2 Applicability

Use the Reliable Hybrid pattern in situations when the need for reliability of an object-oriented software application outweights both the cost of obtaining or developing multiple implementations versions and the associated processing and memory costs.

2.3 Structure

Figure 1: Reliable Hybrid Structure



2.4 Participants

Client

- invokes services of the (Reliable Hybrid) master component when a fault tolerant service is requested

Master

-requests service from two or more functionally equivalent but diverse software versions
-sends the results of the version executions to the adjudicator subsystem
-receives results back form adjudication subsystem and reports results back to client

Version

- abstract class the declares the common interface for software versions

Version1..N

executes the requested service and reports the results of execution back to the master

Adjudicator

- abstract class that declares the common interface for adjudication algorithms

Voter

- abstract class that declares the common interface for objects that implement voter-based techniques

CV, MV, MLV, and Other
- implements CV, MV, MLV any other voting technique suitable for use with voting interface

Accp_Test
- abstract class that declares the common interface for objects that implement RB

Recovery Block Simple (RBS)
- implements RB and runs a simple acceptance test on the given version outputs

Recovery Block Version (RBN)
- implements RB and uses version N + 1 as the acceptance test on the given version outputs

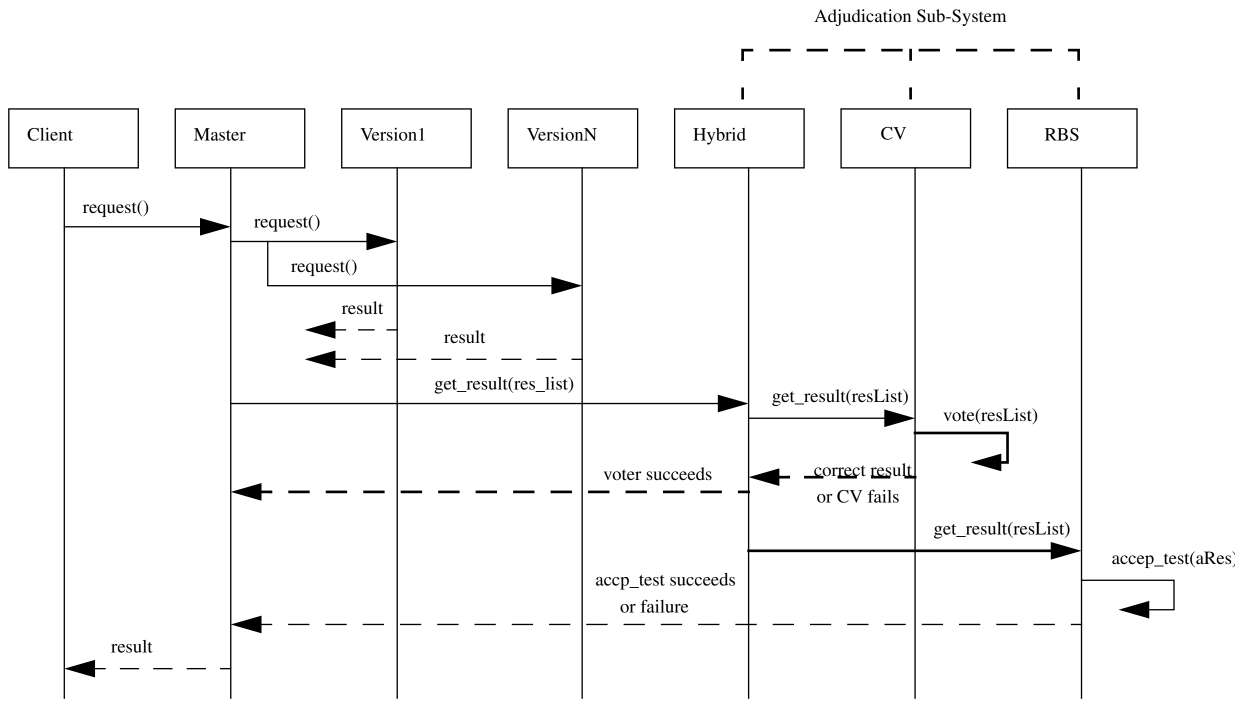
Hybrid
- declares and implements the common interface for objects that implement fault tolerant techniques by recursively combining, NVP, RB and already generated hybrid solutions

2.5 Collaborations

The client invokes the services of the master when run-time fault tolerance is required. The master component then delegates the requested service to one or more of the functionally equivalent software versions. Each version computes the request and returns its results to the master. This operation can be preformed sequentially or in parallel. The master then forwards the outputs to the adjudication subsystem. The adjudication subsystem attempts to adjudicate the results and returns one of them as the correct answer, or it signals a failure. The master then forwards the result back to the client.

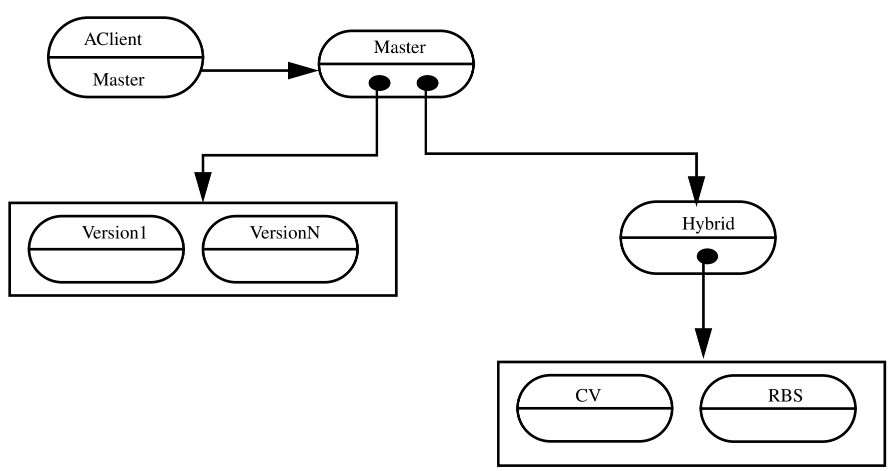
Figure 2 shows the hybrid that represents CRB. Versions are executed in parallel and CV is used to select the correct answer among the results (resList) generated from N versions. If the adjudication (CV) succeeds, that result is returned to the client. If the adjudication fails then the version results (resList) are passed to the RBS adjudicator. RBS first acceptance tests a version result (aRes) that, by some criterion, is considered the “best”. If this result passes the simple test, it is returned back to the client. If it does not pass the test, the next “best” result is tested, and so on. If none of the results pass the tests, the RBS signals a failure.

Figure 2. Interaction model of CRB hybrid with CV



A typical object structure for CRB with CV looks like Figure 3:

Figure 3. Object instance diagram of CRB hybrid



2.6 Consequences

<u>Improved Reliability</u>:

The reliability of fault tolerant software approaches has been studied extensively [Vouk93, Luy95, Lyu96, Kim96]. Studies have shown that depending on the diversity and dependability of the components the reliability gains of a N-version system can be 30% to several orders of magnitude. A criticality and cost analysis should be used to decide if the gains are sufficient and cost effective.

Cost:
It is obvious that development of several functionally equivalent version, will cost more than development of a single version. However, reuse of existing but diverse object libraries can go a long way to mitigate that cost. Practical experience with the cost of development of redundancy-based fault tolerant software solutions indicates that, in a well designed multiversion system, each additional component costs about 75-80% of a single version [Lyu96]. This means that the cost of multiversion software increases sublinearly with the number of components. Additional memory may be required due to the added complexity of the fault tolerant mechanism. Demand on processing resources increases proportional to the number of software versions. Of course, the overall increase in the cost needs to be balanced by the potential for a superliner increase in the reliability and safety of the services a fault tolerant system can provide.

Flexibility:
As mentioned earlier, the Reliable Hybrid pattern generalizes the Master-Slave and the Backup patterns. This flexibility derives from the recursive definition of the adjudicator instantiations which permits a range of fault tolerance strategies, from the very simplest to the much more complex solutions. Furthermore, the Reliable Hybrid pattern explicitly addresses the placement and the mixing of more advanced voting and acceptance testing approaches.

2.7 Implementation

Sequential /Parallel Behavior:
The Reliable Hybrid pattern can be implemented sequentially or in parallel. In implementing sequential behavior the software versions are executed one at a time. The results can then be processed either sequentially (e.g. acceptance testing) or collectively. An alternative is to execute all the versions concurrently and present the results to the adjudication algorithms either in an asynchronous fashion as each version completes, or in a synchronous manner. In some situations, such as in the case of plain acceptance testing, execution of all versions may not be necessary and could be considered and unwanted overhead.

Diversity of Versions:
Modules which are used to compose a multiversion fault tolerant system should be as diverse as possible to mitigate the situation where two versions may contain common cause faults, or for some other reason may fail coincidentally to produce identical but incorrect responses. The latter situation confuses adjudicators and may result in the return of and incorrect response as the “correct” answer. This suggests use of design and development methods that diversify algorithms implemented in the redundant components and actively seek to eliminate causes of correlated runtime failures of the versions. In [Zim95], relationships between design patterns are classified into categories that can aid in design diversity of software versions. For example, design patterns that possess similar relationships can be used to implement solutions to similar problems. Using diverse design patterns to implement functionally equivalent versions is very desirable. The issue of diversity has been studied in great detail in the literature [Lyu95, Lyu96].

2.8 Known Uses

Fault tolerance has been considered for use, and is actually being used, in a number of application areas.

In most cases these are critical application areas such as nuclear power plants, railway systems, and aerospace systems [Lyu95, Luy96].

2.9 Related Patterns

The most closely related are, of course, the Master-Slave pattern that emulates NVP and the Backup pattern that emulates RB. Both of these patterns are subsumed by the Reliable Hybrid pattern and their solutions can be reused to implement NVP and RB adjudication classes of the Reliable Hybrid pattern.

The other two patterns that are related to the general idea of redundancy-based fault tolerance are the Strategy pattern, and the Composite pattern [Gamm94]. In the Reliable Hybrid pattern each diverse software version is an example of Strategy. The Composite design pattern is used when one wished to represent part-whole hierarchy and dependence of objects, some of which are primitive and some of which are composites. In the Reliable Hybrid pattern, NVP and RB are the primitive objects from which once can construct a whole hierarchy of hybrid objects in a recursive fashion.

3.0 Conclusions

The Reliable Hybrid pattern supports development of applications based on basic fault tolerant strategies such as N-Version Programming and Recovery Block, as well as those based on advanced hybrid techniques such as Consensus Recovery Block, Acceptance Voting, and N-Self Checking Programming. The Reliable Hybrid pattern structure provides a generalized framework of designing fault tolerant software using the Backup and Master-Slave design pattern. In addition, the Reliable Hybrid pattern also explicitly indicates how different advanced software voting techniques fit into the design structure.

References

[Aviz85] A. Avizienis, “The N-Version Approach to Fault-Tolerant Software,” IEEE Transactions on Software Engineering, Vol. SE - 11 (12), pp. 1491-1501, 1985.

[Bell91] F. Belli and P. Jedrzejowicz, “Comparative Analysis of Concurrent Fault-Tolerance Techniques for Real-Time Applications”, Proc. Intl. Symposium on Software Reliability Engineering, Austin, TX, 1991.

[Busc95] F. Buschmann, “The Master-Slave Pattern”, Pattern Languages of Program Design, J.Coplien and D.C. Schnidt, eds. Addision-Wesley, pp. 133-142, 1995.

[Gamm94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[Kim96] K.Kim, M.A. Vouk, and D. F. McAllister, “An Empirical Evaluation of Maximum Likelihood Voting in Failure Correlation Conditions,” Proc. ISSRE 96, pp. 330-339, 1996.

[Lapr90] J.C. Laprie, J. Arlat, C. Beounes, K. Kanoun, “Definition and Analysis of Hardware and Software Fault-Tolerant Architectures,” IEEE Computer, Vol 23(6), July 1990, pp. 39-51.

[Leun95] Y. Leung, “Maximum Likelihood Voting for Fault-Tolerant Software with Finite Output Space”, IEEE Trans. Rel, Vol. 44(3), 1995.

[Lyu95] M. Lyu, B. Krishnamurthy, eds. Software Fault Tolerance, Johm Wiley & Son, New York, 1995.

[Lyu96] M. Lyu, eds., Handbook of Software Reliability Engineering, McGraw-Hill and IEEE Computer Society Press, New York, 1996.

[McAl90] D.F. McAllister, C.E. Sun and M.A. Vouk, “Reliability of Voting in Fault-Tolerant Software

Systems for Small Output Spaces", IEEE Trans. Rel., Vol-39(5), pp. 524-534, 1990.

[Nels87] V.P. Nelson and B.D. Carroll, eds, "Tutorial; Fault-Tolerant Computing", IEEE Computer Society Press, 1987.

[Rand75] B. Randell, "Systems structure fro software fault-tolerance", IEEE Trans. Soft. Eng., Vol. SE - 1, pp. 220-232, 1975.

[Scot87] R.K. Scott, J.W. Gault and D.F. McAllister, "Fault Tolerant Reliability Modeling", IEEE Trans. Soft. Engr. Vol. SE-13, No. 5, pp. 582-592, 1987.

[Subr95] S. Subramanian, and W. Tsai, "Backup Pattern: Designing Redundancy in Object-Oriented Software", Pattern Languages of Program Design, J.Coplien and D.C. Schnidt, eds. Addision-Wesley, 1996.

[Vouk93] M.A. Vouk, D.F. McAllister, D.E. Eckhardt, Kalhee Kim, "An Empirical Evaluation of Consensus Voting and Consensus Recovery Block Reliability in the Presence of Failure Correlation," Journal of Computer and Software Engineering, Vol1(4), pp. 367-388, 1993.

[Xu95] J. Xu, B. Randell, C.M. F. Rubira-Calsavara and R.J. Stroud, "Toward an Object-Oriented Approach to Software Fault Tolerance," in Fault-Tolerant Parallel and Distributed Systems, eds. D. Pradham and D. Avresky, IEEE CS Press, pp. 226-233, 1995.

[Zimm95] W. Zimmer, "Relationships Between Design Patterns", Pattern Languages of Program Design, J.Coplien and D.C. Schmidt, eds. Addison-Wesley, pp. 345-364 1995.