Adaptive Fault Tolerance in Distributed Systems

Roger Bharath, Melanie Dumas, and Mevlut Erdem Kurul
Department of Computer Science
University of California, San Diego
La Jolla, CA 92193-0114
{*rbharath, mdumas, mkurul*}*@cs.ucsd.edu*

March 5, 2001

Abstract

Reliable distributed systems provide high availability for an important class of applications through a combination of software and hardware support. Redundancy and replication are essential features of these systems but both come with a high cost. One trend that promises to provide more intelligence to the allocation of resources in this environment is adaptation. Adaptive fault tolerance is the idea of adaptively configuring system resources to respond to environmental changes (i.e. faults). This paper presents an overview of several adaptive fault tolerant systems, and describes the challenges involved in their implementation.

1 Introduction

High availability is a desirable and sometimes crucial characteristic for some scientific and commercial programs. Applications like air traffic control systems, security monitoring systems and real-time systems provide motivation for assured reliability through distribution of processes. However, cost is a consideration in the design of any reliable system and efficient use of resources can be a determining factor of its viability.

Reliable distributed systems can benefit from *adaptive policies* that utilize resources more efficiently and can also achieve greater flexibility. To illustrate this further, consider that the level of reliability provided by a system is closely linked with the redundant resources that it uses. From this, it can be inferred that a configurable level of reliability (or resource usage) could potentially yield resource cost savings compared to a static plan of resource allocation which is constrained by the worst case. Applications can vary in their desired level of reliability, or Quality of Service (QoS) measure, and this variation is not leveraged with a static resource allocation policy.

Support for user-defined adaptive policies is usually implemented apart from the base operating system. A common approach is to incorporate adaptation into a middleware layer that also provides support for distributed computing. Some examples of middleware layers available commercially are DCOM [1], Java Beans [2] and CORBA [3]. Fault tolerance is not inherent in these products [4] and there is no inherent support for user-defined adaptive policies. The design problem is then to implement fault toler-

ance with adaptation starting with a foundation of distributed communication primitives.

This paper is intended as an introduction to adaptive fault tolerance and a survey of current representative systems. We present a theoretical framework for adaptive fault tolerance and apply these ideas to describe systems that feature adaptive fault tolerance.

The remainder of this paper has the following organization: Section 2 introduces the problem domain that is considered. Section 3 presents a model for adaptive fault tolerance. Section 4 presents adaptive fault tolerant systems evaluated upon the criteria proposed in the model, concluding with Section 5.

2 Problem Statement

When dealing with fault tolerance, it is important to characterize the types of faults a system may encounter. Much work on understanding faults (summarized in [5]) has provided the following model aimed at classifying faults.

Four broad types of failure classes are defined [5]. Omission failures occur when a process does not respond to an event. Timing failures occur when a process does not respond before a timeout has expired, or responds prematurely. Crash failures are unrecoverable states when a process completely stops executing. Byzantine, or arbitrary failures, are the class of failures that are unable to be accounted for in the design of a system.

An established approach to achieving fault tolerance is to have multiple copies of the same process running simultaneously, possibly in different environments. From this strategy it is evident that the true cost of a single process is actually that of the process and its copies plus overhead to coordinate their activities.

Typically, a fixed set of resources in a system are reserved to provide redundancy for a task. The reasoning for this is that failure is hard to predict and providing a resource unconditionally gives the best chance of avoiding failure. However, fixing the level of resource usage caters to the worst case, which can be expensive. Varying the level of redundancy allows for opportunities to reduce cost, suggesting an adaptive approach. Adaptive policies enable a system to change its level of redundancy in harmony with its environment, consistent with a user-defined level of granularity.

This gives motivation for the theme of adaptation as applied to fault tolerance. We define adaptive fault tolerance as the property that enables a system to maintain and improve fault tolerance by adapting to changes in environment and policy. Our problem domain focuses primarily on adaptive fault tolerance in distributed systems.

Conventional approaches to designing an adaptive fault tolerant system start with a means of providing rudimentary support for organized distributed computing. This is often done with a middleware technology. Fault tolerance is implemented on top of this middleware layer by some standard techniques that will be discussed in the next section. Finally, another layer of functionality is added to support adaptation, including an interface of some kind between the system and the user. This layered approach is beneficial for establishing correctness. However, an integrated approach has the advantage of escaping the temporal ordering imposed by layering. In particular, adaptivity may be thought about earlier in the design cycle of a system. This could result in more cohesive support for adaptive policies. Such support would become part of the lower level design of the system and could improve efficiency. We feel that this design strategy has been at least partially realized [6] [7] [8]. With this in mind, our goal is to provide an integrated model for adaptive fault tolerance.

3 An Adaptive Fault Tolerant Model

Several system characteristics have been identified as fundamental components of an adaptive fault tolerant system [5] [9]. The objective of this section is to present a unified picture of these required characteristics and to provide a justification for their inclusion in the model, as well as detail the types of problems and decisions that are required to architect an adaptive fault tolerant system.

3.1 Timing

Adaptive fault tolerant systems require different degrees of system responsiveness, based on the service they are providing to users. Real-time systems such as air traffic controllers require synchronous communication between processes, compared to an academic search engine for publications, where processes may communicate asynchronously. The complexity of a real time system, and the cost of providing a high availability service to users is a critical design decision that underlies development of all components of a system.

3.2 Replication

In a distributed system, groups of processes are often clustered together to create an abstraction for higher level processes, such as user applications. These groups look like a single process to the higher level applications, providing an interface for communication and supporting internal mechanisms for handling faults. Replication of processes across nodes provides redundancy to continue servicing requests despite partial system failures.

Replication strategies vary depending on the level of transparency required by particular applications. The degree of replication, or number of redundant processes, may be defined by user parameters, or dynamically configured by the application based on program requirements and specific knowledge of the environment.

3.3 Group Membership

Given a set of redundant processes, a policy for including or removing a particular process in a group must be considered. Upon partial system failures, members of a group must recognize which processes have been partitioned from the group, and each process' perspective of who is in the group must be reconciled. Upon group initialization, or overloading of a particular process, additional processes may be added, and a consistent state of the group must be passed to the new processes.

Modifications to group membership may be controlled by a leader process which was nominated by the other processes, or processes may be more democratic by broadcasting messages and agreeing upon the requested modifications to the group structure.

3.4 Group Communication

A critical component of group architecture is a strategy for reliable communication, which must include a mechanism for handling send omission failures. Policies for communication vary based on the group structure, depending upon whether a leader controls communication between members, or a more loquacious message broadcast protocol is used.

The advantages of using a leader process include simplicity in design and a constant point of contact, but the tradeoff introduces a single point of failure into the system, which must be detected and handled. Alternatively, broadcasting messages between processes requires additional network traffic and maintenance of a list of all active processes.

3.5 Group Agreement

A group of redundant processes must have a mechanism for agreeing on changes detected in an environment, and agreeing on a policy for action. A system that does not guarantee agreement between processes runs the risk of reaching an inconsistent state.

Typical strategies for maintaining agreement depend upon the structure of the group. The most common strategy for processes executing independently is to require a majority voter. This mechanism works because the number of failures tolerated by a system is less than three times the total number of processes, so the faulty processes may never hold the majority. An alternative strategy is that a leader may broadcast to its followers any changes detected, or any action requests.

3.6 Group Synchronization

Synchronization between members of a group is required in order to detect a faulty process and to avoid reaching an inconsistent state. In addition, processing dependent messages or executing sequential commands also demand synchronization between processes, otherwise the outputs of particular requests may vary between group members.

Strategies for handling synchronization typically require a mechanism for multicasting messages reliably between processes, and allowing only atomic actions. Reliable multicast restricts receipt of the message to group members only. Atomic actions guarantee that an action either completes executing, or does not execute at all. If a leader desires to broadcast a message, these mechanisms ensure that either the message was seen by all other processes, or none.

3.7 Fault Detection

Detection of faults is limited to the types of faults that a system may handle. The potentially exhaustive list of faults generally avoids the inclusion of arbitrary, or Byzantine faults, which simplifies the system design. The remaining send omission, crash, and timing failures may be detected by processes devoted to their detection, such as network monitors, or may be detected by any member of the group. It is the responsibility of the monitoring process to notify the group members so a decision can be made. In the case of crash failures, a crashed process may be detected when the process fails to respond to a message within a timeout period. The sending process then has the responsibility to notify all other members of the group that the process failed.

An alternate model is to designate a group as a monitor for another group. This model defines a dependence relation between objects where an object A is dependent on an object B if B is responsible for monitoring the state of A. An example of this is to consider when a parent process monitors the state of the child. The advantage of this approach is a single point of responsibility, but the disadvantage is that if an error occurs in a parent process, the state and data of a child process may be irrevocably separated from the system. More robust relations can be defined by introducing cyclic dependencies.

3.8 Fault Transparency

Two models are generally applied to handle actions when faults occur and have been detected in a system. The first model views faults as exceptions to be propagated to the higher layers of the system for handling, with the reasoning that the fault would be reprocessed at a higher layer.

The objective of most fault-tolerant models, however, is to provide uninterrupted service to users and handle all faults within the lower lev-

els of the system, thereby masking that the failure ever occurred. This process of handling and masking failures is called fault transparency, and many systems will continue service based on sheer numbers of redundant processes. This (second) model allows a certain number of faults to be tolerated, and above the threshold the system may notify the user or discontinue service.

3.9 Environment Awareness

This rather undeveloped area of fault tolerant adaptive systems includes the detection of changing resources automatically and dynamically. Many fault tolerant systems require the user to notify the system when additional resources become available, since most of the models handle only the case when resources fail and become unavailable. Automatic detection of additional resources would provide additional flexibility in a system, and allow a system to continue running longer without user intervention. This strategy could be viewed as a proactive attempt to effectively utilize all available resources, and should be examined carefully in future systems.

4 Systems

The purpose of this section is to survey representative systems based on the criteria developed in our model.

4.1 Electra

Electra [4] [10] provides an infrastructure for users to develop reliable distributed computing applications, including an adaptive component for dynamic updates. Electra is built using CORBA, and addresses a number of issues resulting from CORBA's inherent inability to provide reliable distributed communication.

The core CORBA technology does not include a mechanism for multicasting requests between objects in a group, which is a critical component of distributed computing. Landis and Maffeis [4] provide a solution by developing a CORBA Object Group, which is a group of replicated processes that maintain state consistency. An object group allows the user to define the number of processes within a group, dynamically adjusting the level of replication of an application.

Synchronization among members of a group is maintained by an active replication object within the group, which monitors each member to provide assurance that members are actively maintaining replicated state. Active replication ensures consistency within a group, and allows service requests to a group as long as one of any of its members is accessible. Active replication also provides the foundations for load balancing and object migration across servers.

Addition of new processes to a group are requested by the active replication object. After a new process is started, its state is synchronized by receiving a state transfer from the active replication object. This transfer of state may only occur when the existing members of a group are in a consistent state. Each object within a group maintains a view, which is a dynamic list of the objects within its group. To complete the addition of a new group member, an external Object Monitor process notifies each existing group member of the membership change.

CORBA Object Groups are able to ensure consistent states by requiring atomic requests to a group. In addition, the user may request a particular level of synchronization by specifying either total or causally ordered message passing [4]. These conditions coupled with reliable multicasting of all messages facilitate reliable communication that is seen in the same order by all processes within a group. Since all processes are running from the same code base, and they

all see the same messages, agreement is implicit. A difference in the results of processes indicate that a process has gone out of synch with respect to the group and the process is halted.

A user-specified parameter establishes the maximum number of process failures, allowing Electra to determine the size of the process group that will ensure a majority of nonfaulty processes. Faults within a group are transparent outside the group, since the nonfaulty processes will agree on actions, and external processes will see only a single resultant value.

Electra provides an infrastructure that allows a programmer to easily write and customize an application given a reliable, distributed framework. Therefore, constraints upon the timing of an application must be imposed and implemented by the user. Electra allows asynchronous and synchronous communication between processes, which may require an upper bound on the length of a communication transaction. In addition, the programmer is also responsible for implementing the degree of environmental awareness in an application, since this is not inherently provided within this framework.

4.2 AFTM

AFTM [8] is an adaptive fault-tolerant middleware, which uses a CORBA-compliant object request broker. The objective is to provide complex, mission critical applications with a uniform interface between the application and the underlying software layers that transparently monitors and adaptively reconfigures system resources. Additionally, the system incorporates a generic, scalable adaptation policy.

AFTM is implemented on a network of Solaris workstations due to ease of implementation and support for real-time threads. Real-time facilities are provided by a layer which implements RTO.k objects, where each RTO.k object uses a time-triggered method that is associated with a completion deadline, a concurrency constraint, and an upper bound on the life of the object data.

The components of the AFTM architecture include the Adaptation Manager (AM), Resource Allocation Executive (RAE), Network Reconfiguration Manager (NRM), and System Monitor (SM). Each of the components of AFTM are dedicated to a specific operation such as fault detection, decision making, or reconfiguration. Through cooperation, these components support resource management, fault tolerance, response to environmental changes, and application-specific adaptation. The user may specify the level of adaptation and can be optionally informed of the health of system resources through the SM. However, task communication and synchronization are handled by the underlying CORBA and application layers.

Three databases are used to implement the AFTM adaptation. These are the environmental database which stores environmental conditions and demands maintained by the sensors and SM, the internal state database which maintains the recent failure history as well as the history of system resources maintained by NRM and RAE, and user requirement database which is maintained by the user for application specific replication and adaptation policies. An Adaptation Manager uses these databases to select a suitable fault-tolerance and resource allocation strategy, which is implemented by the RAE.

The leader node makes group decisions by requesting services from local AM and RAE components, to distribute the workload evenly for reliability and efficiency. The leader also implements voting policies within the group. In the case of a leader failure, a follower node detects the failure through a timeout and is promoted to a new leader. Determination of faulty processes is accomplished through timeouts and the NRM, with the leader coordinating removal of group members and reallocation of new members if necessary.

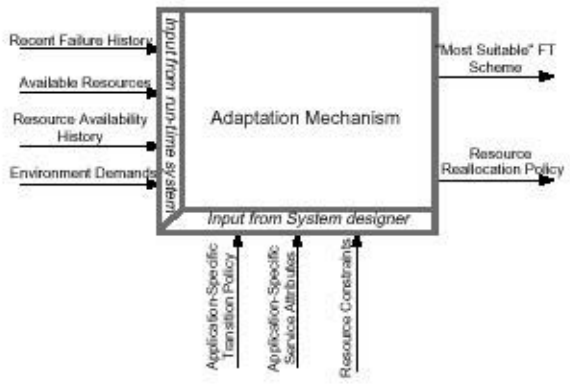

Figure 1: AFTM Adaptation Mechanism

The AFTM middleware enhances the system reliability, performance, survivability and effectiveness through the use of its components. The "Most Suitable" fault tolerant and resource allocation scheme is selected dynamically through user requests and the parameters in the three databases (Figure 1), which is coordinated through the Adaptable Distributed Recovery Block (ADRB). The ADRB is replicated across nodes, tolerating failures and providing a specific fault tolerant execution mode for each node. These services provide automatic reconfiguration of the groups, transparently masking faults from the user.

4.3 Proteus

AQuA is an operating system architecture, which provides a flexible infrastructure upon which a distributed, fault-tolerant system may be built. AQuA is supported by CORBA, and provides dynamic replication of objects satisfying the dependability requirements of the users.

The AQuA infrastructure contains Proteus [11], which is the fundamental component providing dynamic fault tolerance for user-specified faults through adaptive reconfiguration. Proteus provides applications the means to specify their level of dependability through the use of Quality Objects (QuO). The QuO runtime application is responsible for interfacing with AQuA and detecting and handling faults.

Proteus is organized into three modules (see Figure 2), which are connected to AQuA via the QuO runtime. The first module is the dependability manager, which consists of an advisor and a protocol coordinator. The Advisor makes system-wide decisions based on the system information and application requests, and determines a strategy for handling faults. Advisor requests are propagated to the other components of Proteus using the Protocol Coordinator. The Protocol Coordinator uses the Ensemble group communication system to ensure reliable message passing between groups.

The Object Factory module is instantiated on each local node, and is responsible for monitoring, creating and killing local processes. Requests for process replication originate at the Advisor, and instantiation and registration of processes is provided by the Object Factory. The factory is assumed to maintain handles on all local processes, and provide any membership services as requested from the Advisor or from within the local processes.

The final Gateway module provides critical pieces of the group infrastructure. The Gateway implements four different groups, managing replication, connection, point-to-point, and broadcast communication services. The Gateway ensures agreement between this set of groups by implementing a simple voting mechanism and communicating the results to all nodes through group leaders. Each group specifies a leader that sends a message, which is ensured to be reliably multicast and atomic using Ensemble. Synchronization of processes is maintained through message passing and acknowledgments between the leader and follower processes.

Faults may occur at either the leader or follower processes. A faulty follower process is detected by the leader when a process failed to send an acknowledgment to a message within a

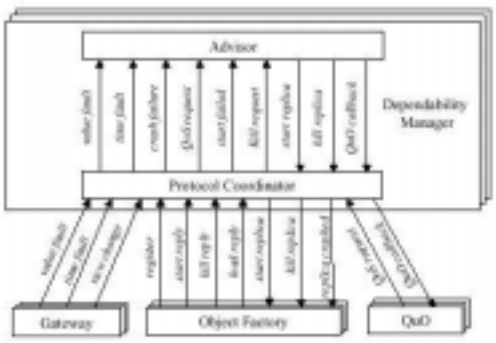

Figure 2: Proteus Object Model

timeout period, or when the leader detects an error in the state of a process. If the leader fails, the follower process that detected the failed leader becomes the new leader, and updates its view of the system.

Adaptive responses to the environment are not dynamic, but Proteus provides the user with the ability to manually register additional resources through the QuO runtime to the Advisor. The system is not designed to provide real-time services, instead focusing on visibility and providing the user with finer grained control of dependability requirements.

4.4 Chameleon

Chameleon [6] is a reconfigurable software layer that employs ARMORs (Adaptive, Reconfigurable, Mobile Objects for Reliability) to provide fault tolerance and adaptability. An ARMOR is composed of *basic building blocks* which are objects that correspond to low level mechanisms. Some examples are MessageSend and MessageDispatch which are responsible for sending and receiving messages to and from other ARMORs. Blocks support host-independent primitives which define the functionality of ARMORs.

An ARMOR abstractly represents a characteristic type of dependability. A base set of ARMORs are defined for the architecture with well defined semantics. They can be extended into applications by the application programmer. Table 1 summarizes the functionality of each type of ARMOR.

ARMOR	Functionality
Fault Tolerance Manager (FTM)	Coordinates application requests.
Backup FTM	Monitors health of FTM. Assumes duties if FTM fails.
Daemon	Monitors Execution ARMORS. Coordinates node communications.
Heartbeat	Monitors health of Daemons via heartbeat messages.
Surrogate Manager (SM)	Coordinates the execution of an application under a particular strategy (FTES).
Initialization	Obtains node-specific information upon Daemon start-up, sending it to the FTM.
Execution	Installs, executes and cleans up the application. Returns results to FTM via SM.
Fanout	Coordinates multicast and distribution of data within a replication group.
Voter	Collects and determines group agreement based on a user-defined voting policy.
Checkpoint	Provides checkpointing and recovery mechanisms for an application.

Table 1: Chameleon ARMORs

The Chameleon architecture is composed of two layers. The upper layer is a dynamic collection of ARMORs that may be distributed and

replicated over a networked system. The lower layer is the Chameleon API that supports ARMORs and direct calls from applications. Both layers are supported by the OS. Further details can be found in [6].

Support for real-time applications is provided in Chameleon through the use of subclasses. A subclass of FTM can be instantiated to support real-time operation. This ARMOR, called a real time manager (RTM), provides similar functionality to the FTM but is composed of different building blocks. The distinction is basic building blocks for real-time operations have predictable (bounded) latencies. Using these building blocks, subclasses of all ARMORs can be constructed resulting in predictable execution times.

Chameleon provides a language with which users can devise fault tolerant execution strategy (FTES). These plans detail precise numbers and distribution of group members. A registry of these plans is maintained by the FTM. Common schemes can be tuned for performance or reliability. For example, the first result produced by a group member can be returned (for performance) or a majority voter can be used for reliability.

Faults experienced by a group member may require the addition of a new group member and a transfer of state, which is handled by FTES. Several possible situations may occur. The functional ARMOR that coordinates the recovery is the Surrogate Manager (SM). Techniques for recovery may include process migration, application restart or the creation of a new group member. Transfer of state is required for the last option and can be done with a checkpointed copy of a current group member.

Group communication is handled by the Daemon. A Daemon is installed on each node and implements at least one network protocol. All intra-node communication is directed first to the Daemon and from the Daemon to the destination. This is done using standard inter-process communication facilities. Inter-node communication is directed through the Daemon on the sender's node to the Daemon on the receiver's node.

Synchronizing group members provides challenges that require help from the application to be effectively performed. The Fanout ARMOR is the primary instrument of synchronization providing atomic multicast and total message ordering facilities. Applications that use nondeterminism need to devise protocols to ensure group consistency. Different nodes could have different pseudo random seeds and algorithms; with some coordination, a single random value can be broadcast to all group members so that a meaningful result is produced. Another problem occurs with cooperating processes. Interposing the Fanout ARMOR between the communications of the processes is an adequate solution.

Fault detection in Chameleon is accomplished by timeouts and a dependence relation. The relation specifies which component of the system is responsible for detecting faults in other components. The dependency is cyclic, which enables knowledge of fault discoveries to propagate backwards, notifying other components of the failure. Omission, crash and timing failures can be detected per normal operation. In addition, it is possible for users to code a notion of forward progress and with this user-supplied procedure, an ARMOR can detect livelock (a specific form of omission fault).

Much of an application's desired behavior when faulting is specified in its FTES. In particular, transparency can be implemented when it is possible to gain meaningful results from some group members. The Voter ARMOR can implement a user specified voting policy for the situations when group members return different results (i.e. value fault). Crash and omission faults can be masked when at least one group member continues executing. With the aid of the Checkpoint ARMOR, it is possible to mask failure of

an application with only one group member provided that results are returned to the user only after a checkpoint completes.

Chameleon relies on explicit representation of adaptive policies providing some support for environmental awareness.

5 Conclusions

In this paper, we presented a unified model highlighting fundamental components in the design of an adaptive fault tolerant system. We used our model to describe a selection of recent representative systems and expose the design decisions made during their construction.

Adaptive fault tolerance can increase availability, reliability and decrease cost in a distributed computing environment. Present-day AFT systems are mature in their use of redundancy, communication and synchronization but to further the goal of reliability other directions need to be explored. Environment awareness and other proactive measures are features of AFT that we believe future systems will attempt to leverage.

Acknowledgements

We would like to thank Keith Marzullo for his feedback on our approach to the topic of reliability in distributed systems.

References

[1] Microsoft Corporation

[2] Sun Microsystems

[3] The Object Management Group

[4] S. Landis, S. Maffeis. ”Building Reliable Distributed Systems with CORBA,” in Theory and Practice of Object Systems, John Wiley, New York, April, 1997.

[5] F. Cristian. ”Understanding Fault-Tolerant Distributed Systems,” in Communications of the ACM, 34(2):56-78, 1991.

[6] Z. Kalbarczyk, R.K. Iyer, S. Bagchi, K. Whisnant. ”Chameleon: a software infrastructure for adaptive fault tolerance,” in IEEE Transactions on Parallel and Distributed Systems 10(6):560-579, June, 1999.

[7] J. Ren, M. Cukier, P. Rubel, W. Sanders, D. Bakken, D. Karr. ”Building Dependable Distributed Applications using AQuA,” in Proceedings of the 4th IEEE International Symp. on High-Assurance Systems Engineering, pp. 189-196, November, 1999.

[8] E. Shokri, H. Hecht, P. Crane, J. Dussault, K. Kim. ”An Approach for Adaptive Fault-Tolerance in Object-Oriented Open Distributed Systems,” Workshop on Object-Oriented Reliable Distributed Systems, February, 1997.

[9] M. Hiltunen, R. Schlichting. ”Adaptive Distributed and Fault-Tolerant Systems,” in International Journal of Computer Systems Science and Engineering, 11(5):125-133, September, 1996.

[10] S. Maffeis. ”A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming,” in Proceedings of the ECOOP 1993 Workshop on Object-Based Distributed Programming,

[11] C. Sabnis, M. Cukier, J. Ren, P. Rubel, W. Sanders. ”Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA,” in Proceedings of the 7th IFIP International Working Conference in Dependable Computing for Critical Applications, pp. 137-156, January, 1999.