A Survey Of Generic Architectures For Dependable Systems

by *Yang Liu and Purnendu Sinha*
Concordia University, Montréal, QC

Abstract

Dependability and availability of computing systems for critical real-time applications have been major concerns in development of different fault-tolerant architectures. With ever growing need to provide for reliable and timely services in varied applications, generic architectures for dependable systems are being developed that can adapt with ease to very diverse requirements of such applications. The main stream of building generic architectures for dependable real-time systems consists of the GUARDS architecture, the AQuA architecture, the FRIENDS architecture, and the Chameleon architecture. Their common goals are to achieve transparency, generality and adaptability. This survey paper compares the architectures in terms of their fault models, fault detection mechanisms and fault treatment schemes. A qualitative comparison between these architectures is summarized after evaluating them based on cohesion and coupling principle.

Sommaire

La sûreté de fonctionnement ainsi que la disponibilité de systèmes informatiques pour des applications critiques en temps réel ont été des soucis majeurs dans le développement de différentes architectures tolérant les fautes. Avec un besoin sans cesse grandissant de fournir des services fiables et à temps, dans diverses applications, des architectures génériques pour des systèmes sûrs de fonctionnement sont développées pour s'adapter avec facilité aux exigences fort variées de telles applications. Le courant principal dans la construction d'architectures génériques de systèmes sûrs de fonctionnement en temps réel est composé des architectures GUARDS, AQuA, FRIENDS et Chameleon. Leur but commun est l'atteinte de la transparence, de la généralité ainsi que la faculté d'adaptation. Le présent article donne une vue d'ensemble de ces différentes architectures et les compare en termes de leurs modèles de fautes ainsi que de leurs mécanismes et procédés pour détecter et traiter respectivement ces dernières. Une comparaison qualitative de ces architectures est présentée après les avoir évaluées selon les principes de couplage et de cohésion.

1.0 Introduction

Providing fault tolerance to distributed applications is a challenging and important goal. Towards this objective, in the past several years, fault-tolerant architectures, such as SIFT, FTMP, FTPP, MAFT, ERICA, Delta-4 [5,6] among others have been designed and implemented. The initial efforts in building software-implemented fault tolerance were put in integrating fault tolerance mechanisms into the operating system. These systems are limited in terms of accessibility and customization, and tend to only offer a static level of fault tolerance that remains fixed throughout the lifetime of the system. A better approach is to implement basic fault tolerance mechanisms in a library on top of the operating system. However, in this case, since the library functions are coupled with application source code, the mechanisms are still not independent.

Currently, efforts are being made to build generic architectures for fault-tolerant distributed systems. The FRIENDS [2], the AQuA [1], the GUARDS [4], and the Chameleon [3] represent the main streams of this effort. The common goals of these architectures are:

- **Transparency:** the fault tolerance mechanisms should be transparent to application programmer.
- **Generality:** a wide range of applications with different fault tolerance requirements can use the architecture.
- **Adaptability:** the architecture should be able to adapt to changing requirement both spatially and temporally.

The paper is structured as follows: Section 2 outlines the comparison criteria based on fault model, fault detection mechanisms, fault treatment strategies, and cohesion and coupling principles. Sections 3 to 6 summarize GUARDS, AQuA, FRIENDS, and Chameleon architectures respectively. In Section 7, we evaluate the above four architectures based on the criteria defined in Section 2.

2.0 Comparison Criteria

To evaluate the generic architectures, two system design and evaluation concepts are needed: cohesion within each module and coupling with the other modules:

- **Cohesion** is a qualitative indication of the degree to which a module focuses on just one thing, whereas,
- **Coupling** is a measure of the relative interdependence among modules,

Ideally, in a generic architecture, the cohesion within each module should be as high as possible and the coupling between modules should be as less as possible,

More concretely, the listed architectures will be evaluated on the basis of how they answer the following questions within these three subdomains:

1. **Fault-model:** What is the fault model of the architecture? Specifically, what are the faults handled in the architecture? Are crash faults, value faults, time faults and their combinations considered or just a subset of them are considered? The guiding principle is that the lesser the assumptions about faults that can occur, the better would be the architecture. Moreover, how many fault classifications are there in the architecture? Are the faults simply classified as crash faults, value faults, and time faults or the faults classified based on different basis such as location, duration, and criticality? More classifications mean that the architecture is capable of differentiating the faults in detail, thereby achieving high efficiency for fault tolerance.

2. **Fault-detection:** How are faults detected? Specifically, are the faults detected by operating system services or by system independent modules? Are there different modules for different faults? How long does it take for an error to be detected? How are different fault detection techniques organized? The principle is that a generic architecture should incorporate all available fault detection mechanisms into the architecture in a highly cohesive way. In addition, if new fault detection mechanism were invented, it should be easily adopted by the architecture.

3. **Fault-treatment:** How are detected faults treated? Specifically, How are faults masked? How are faulty components handled/eliminated? How are different fault treatment mechanisms organized? The basic idea is that a generic architecture should incorporate all available fault treatment mechanisms in a highly cohesive way and open to new technologies that might be invented in the future.

We emphasize that we focus only on the design of fault-tolerant mechanisms built into these systems for our comparative studies.

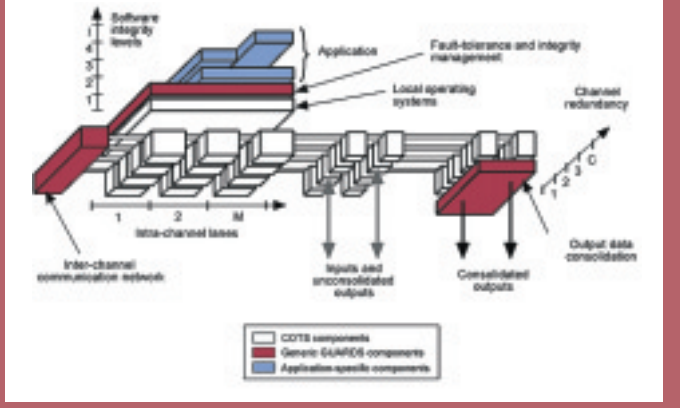

Figure 1: The Overall GUARDS architecture [4]

3.0 GUARDS

GUARDS (Generic Upgradable Architectures for Real-Time Dependable Systems) architecture provides a comprehensive framework from which specific instances can be derived to meet the dependability requirements of various application domains.

3.1 The Overview of the GUARDS Architecture

As Figure 1 shows, GUARDS uses a limited number of specific, but generic, hardware and software components to implement an architecture that can be configured into a wide variety of instances along three architectural dimensions - redundant channels, redundant lanes and integrity.

Channels provide the ultimate line of defense within a single instance for physical faults that affect a single channel. Multiple processors or lanes can be used to improve the capabilities for fault diagnosis within a channel, e.g., by comparison of computation replicated on several nodes. The integrity dimension aims to provide containment regions with respect to software design faults.

3.2 Fault model of GUARDS Architecture

The GUARDS architecture is capable of handling crash faults, value faults and time faults. Furthermore, the architecture considers both physical faults and design faults. Physical faults are assumed to occur independently on different components. The architecture differentiates channel-correlated faults, lane correlated faults, and globally correlated faults as well. However, most of the time, the faults in GUARDS architecture are classified as either temporary or permanent, because this classification is more pertinent to the efforts of masking faults in real time systems where temporal redundancy is not available.

3.3 Fault Detection and Diagnosis in GUARDS Architecture

Fault detections are done in two levels in the GUARDS architecture: intra-channel level and inter-channel level. First, each channel has local mechanisms to detect crash faults, value faults, and time faults that are from different sources such as nucleus, hardware, system, and application. In addition to the ability to locate faults, each channel is capable of diagnosing the fault as permanent or temporary by running a self-test algorithm. The self-test is carried out on a channel after it has been isolated from the pool. Secondly, the inter-channel fault detection depends on the number of operational channels in the system. As long as there are at least three operational channels, any errors due to a single faulty channel are detected by majority voting. In the case when two operational channels are available, a two-out-of-two vote is considered and single channel errors are detected. Any configuration with more than three channels is capable of tolerating arbitrary faults except a Byzantine fault leading to failure of the clock synchronization mechanism.

The efficiency of fault detection and diagnosis depends on the level of redundancy of the channels and the lanes. Moreover, the detections are achieved by the deliberate cooperation of channels and lanes. The components are tightly coupled together to provide efficient fault detection.

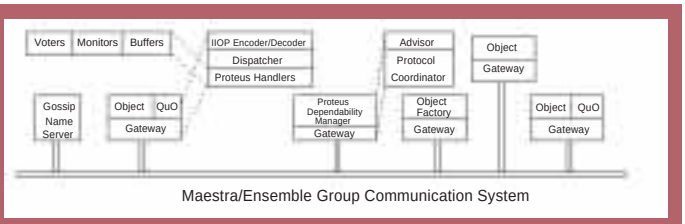

Figure 2: The Overall AQuA Architecture [1]

3.4 Fault treatment in GUARDS Architecture

Detected errors trigger fault diagnosis to determine which channel is faulty. The faulty channel is then isolated from the operational channels to execute a self-test aimed at determining whether the fault is permanent or temporary. If the fault is judged to be permanent, an explicit repair action must be carried out whereas in the case of a temporary fault, certain fault treatment strategies may authorize automatic re-integration of the faulty component. Furthermore, Correlated faults at integrity level 1 should be confined to that level by the integrity policy (IP) enforcement mechanisms.

Essentially, the faults, if any, will be first contained within the channel dimension; if the efforts fail, attempts are made to contain the faults within the lane dimension; finally the integrity dimension tries to prevent the software faults from propagating.

4.0 AQuA Architecture

The AQuA architecture, which stands for Quality of Service for Availability, allows distributed applications to request and obtain a desired level of availability by configuring the system in response to outside requests and changes in system resources due to faults.

4.1 The Overview of the AQuA Architecture

Figure 2 shows the different components of the AquA architecture in one particular configuration. In the AQuA Architecture, the application uses “Quality Objects (QuO)” to specify dependability requirement. The AQuA framework employs the QuO runtime to handle these requests, and the Proteus dependability manager to configure the system in response to faults and availability requests. In addition, a CORBA interface is provided to application objects using the AQuA gateway. The AQuA architecture is heavily dependent on the Ensemble protocol stacks to provide group communications services.

4.2 Fault Model in AQuA architecture

The AQuA architecture handles object faults of three types: crash failures, value faults and time faults, but imposes the following assumptions:

- All faults occur in nodes not in links,
- Value faults occur within objects themselves, not in the links,
- Value faults occur only in the application and/or QuO runtime, thus not in AQuA gateway.

The assumption about no value faults in links holds only when the conventional coding/correction techniques (such as hamming code) are available. If the conventional coding/correction techniques do not exist in the underlying infrastructure, then the consequence of this assumption is that either some faults that cannot be tolerated or they will be tolerated in a very inefficient way.

4.3 Fault Detection in AQuA architecture

The crash failures are detected by Ensemble. Among the elements composing the object, only the gateway process is an Ensemble process. However, since the crash of the application process or the QuO runtime process leads directly to the crash of the gateway process, Ensemble can detect the crash of any element of an object. The failure of any replica will cause a view change of the group composition. This view change is communicated to the Proteus manager through the Proteus Communication Service (PCS) group. The comparison between the old structure of the group and the new composition allows the dependability manager to detect the crash failure.

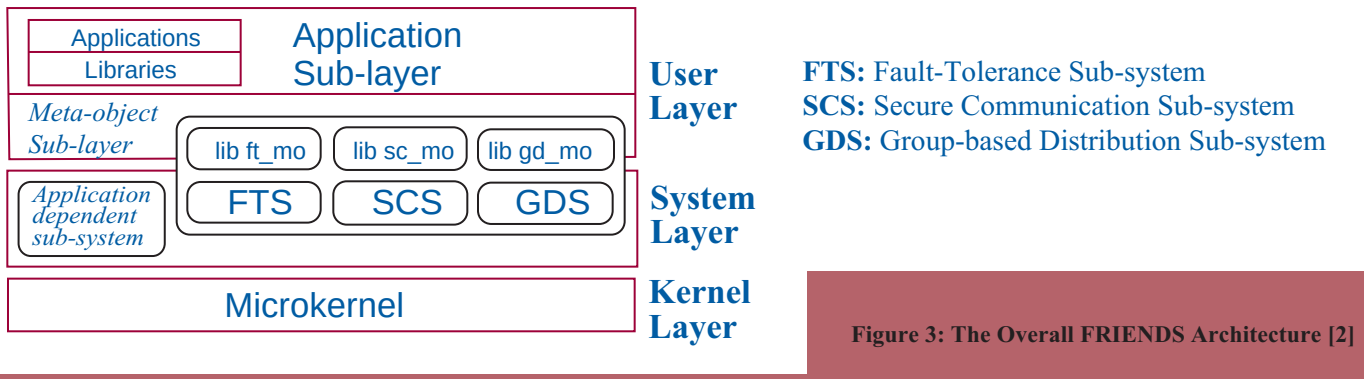

Figure 3: The Overall FRIENDS Architecture [2]

The voter that is implemented in the gateway part of the leader object detects value faults. Although each object has a voter implemented, only the voter present in the leader of the replication group is active. When an object on the client side sends out a request it sends it to the leader of its replication group. Then the voter of the leader votes on the requests. If some requests differ from the majority, a single or multiple value fault has occurred. In this case, the leader gateway process of the replication group joins the PCS group to notify the Proteus manager about the value fault. Equivalently, on the server side, after a request has been processed by the different replicas of the replication group, all replicas send back their reply to the leader of the server replication group. The voter of the leader then votes on the different replies. A value fault has occurred if one or more replies differ from the majority. The leader gateway process then joins the PCS group to complain about the value fault. Proteus thus detects a value fault by the communication of the complaint when the leader joins the PCS group.

Time errors are detected by monitors that record information regarding various times and omissions. Where and how the timers operate depends on the type of faults that are being tolerated. A monitor is implemented in the gateway part of each object. When tolerance to time faults is required, all monitors of the object members of replication groups activated time faults are communicated to the Proteus manager using the PCS group structure described earlier.

The separation of detection mechanisms of crash faults and detection mechanisms of value faults and time faults makes the fault detection of AQuA less cohesive. Furthermore, voters and monitors are not independent components themselves; they are parts of the AQuA gateway whose main function is however to interface between Ensemble and application level components. Such coupling of error detection mechanisms and interfaces makes the cohesion within AQuA gateway less efficient.

4.4 Fault Treatment in AQuA architecture

The Proteus manager advisor, using fault information communicated from the gateways, makes decisions regarding fault treatment. After a decision is reached, the object factories and gateways make the configuration change, under control of the protocol coordinator.

For crash failures, since the number of replicas may need to be maintained, a new object may be started either on the same host or on another host. The advisor decides when and where the new object is to be started. The new object joins the replication group and the state of the leader is transferred to the new replica.

For value and time faults, the fault treatment consists of two phases. First, the source of the fault is determined, based on information provided to the advisor. Using complaints from the various object monitors and voters, the advisor decides whether to kill suspected replicas, or start new replicas, and where to start the new replicas. Second, the replicas for which a value or a time fault has been detected may be killed and new replicas started, if mandated by the advisor, in order to maintain the global number of replicas. The newly created objects then join the replication groups from which objects have been killed, and the leaders of these replication groups transfer their state to the new objects.

As the decision on how to treat faults and the action of fault treatment are done by the advisor and the coordinator, respectively, the separation of decision-making and action makes both advisor and coordinator more cohesive as well as less coupling with the other modules. This is advantageous in a sense that further modification of the advisor and the coordinator is transparent to other components.

5.0 FRIENDS

The FRIENDS architecture, which stands for Flexible and Reusable Implementation Environment for your Next Dependable System, achieves fault tolerance by providing a library of meta-objects that can be recursive to add new properties to distributed applications in an object-oriented manner.

5.1 The Overview of the FRIENDS Architecture

The FRIENDS consists of three layers and a protocol:

4. The kernel layer which can be either a Unix kernel or a microkernel, like Chorus,
5. The system layer composed of several dedicated sub-systems,
6. The user layer dedicated to the implementation of applications and mechanisms as meta-objects, and
7. A customizable meta-object protocol (MOP) that defines how the application objects and meta-objects interact.

A simplified static view of the overall system architecture is given in Figure 3.

5.2 Fault Model and Detection in FRIENDS Architecture

The FRIENDS architecture currently deals with only physical crash faults and assumes that application objects have a deterministic behavior. Concurrency and other sources of non-determinism have not been considered yet. The fault-tolerance mechanism for other types of faults (e.g., software faults) can be added to an application by connecting the appropriate meta-objects to the application objects. Note that meta-objects behavior depends on information provided by the application objects, thus differing from mechanisms used for handling physical faults.

The FRIENDS architecture mainly employs watchdog timer, fail silent network attachment controllers and double memory boards to implement its fault detection mechanisms. Fault detection mechanisms are provided in the library of fault tolerance meta-object classes (libft_mo). In turn, the libft_mo builds these mechanisms on top of the basic service provided in fault tolerance subsystem. New meta-objects for different fault detection mechanisms can easily be added into the library in a cohesive manner.

5.3 Fault Treatment in FRIENDS Architecture

Basically, three fault tolerance mechanisms are implemented in the form of meta-object classes: a mechanism based on stable storage, a primary-backup replication protocol and a leader-follower replication protocol. Specifically, the stable storage is achieved by two meta-objects: STABLE_STORAGE and DOMAIN. During the object method execution, an error can be detected by the error detection system service

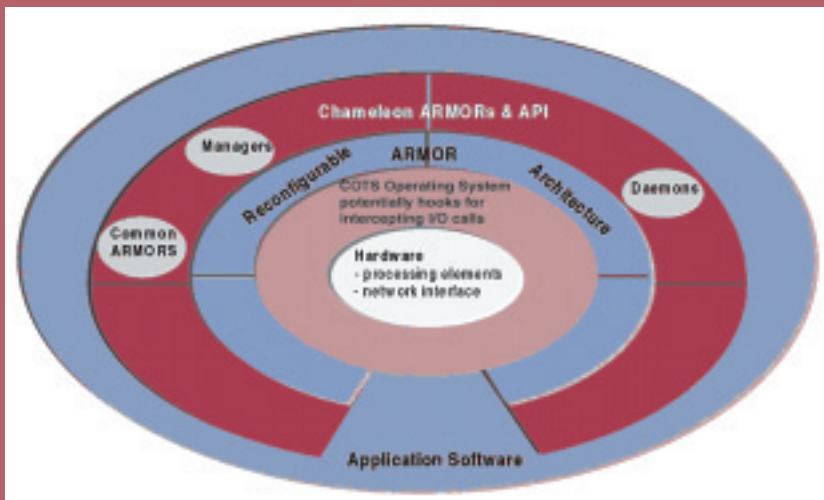

Figure 4: The Overall Chameleon Architecture [3]

either before the new state is saved to stable storage or in between the time it is backed up and the time the server sends the reply to the client. In both cases the error is signaled to the client meta-objects, and the STABLE_STORAGE will identify the recovery point and the DOMAIN will invoke its method to maintain stable storage.

The primary-backup replication mechanism is implemented in two meta-objects: PBR_CMO and PBR_SM. In this strategy, all replicas of a server belong to the same atomic multicast group and, thus, receive the same input messages in the same order. Among the replicas, only the primary handles the client requests and checkpoints its new state at the end of every method executed to the backups. Any primary errors will be detected by backups, which then choose a new primary among them. This new primary restores the last checkpointed state and, if necessary, executes the current request before returning the reply to the client.

The leader-follower replication mechanism is achieved by two meta-objects: LFR_CMO and LFR_SMO. In this mechanism, all replicas process input messages, but only the leader sends output messages. All replicas of a server belong to the same atomic multicast group and receive the same messages in the same order. The leader first executes the request and then notifies it to the other replicas that in turn execute the request. Only the leader returns the reply to the client.

6.0 Chameleon

The Chameleon architecture provides a powerful framework through which fault tolerance execution strategies may be constructed and reused to provide dependability to substantially off-the-shelf applications.

6.1 The Overview of the Chameleon Architecture

Chameleon provides the runtime environment for reliability through the use of ARMORs that stand for Adaptive, Reconfigurable, and Mobile Objects for Reliability. ARMORs are components that control all operations in the Chameleon environment and can be classified into three categories: Managers, Daemons, and Common ARMORs.

There are three kinds of managers in the Chameleon architecture: Fault Tolerance Manager (FTM), Surrogate Manager (SM), and Backup Fault Tolerance Manager (Backup FTM). There are six kinds of common ARMORS in Chameleon environment: Heartbeat ARMOR, Execution ARMOR, Checkpoint ARMOR, Voter ARMOR, Initialization ARMOR and Fanout ARMOR. Daemons are entities resident on every participating node.

6.2 Fault Model in Chameleon Architecture

Chameleon makes no assumptions about faults that may occur in the system. This makes it applicable to wider range of applications. Furthermore, Chameleon differentiates the faults that might occur in application, its own components and underlying network components. This differentiation makes it more efficient to locate the faults and subsequently to recover from them.

6.3 Fault Detection in Chameleon Architecture

The detection mechanisms are implemented in Heartbeat ARMOR, Execution ARMOR, Voter ARMOR, Backup FTM and the Daemon. The Heartbeat ARMOR is responsible for detecting crash faults that occur in nodes, links, and daemons. If necessary, Heartbeat ARMORs can collect information sufficient to determine the health of the node being monitored.

The Execution ARMOR is responsible for detecting abnormal terminations (crash failures) and live-locks in applications by overseeing application executions. The erroneous computations (value faults) of applications are detected by voter ARMORs. The value faults and time faults in applications originated in underlying hardware or operating system can also be detected by execution ARMORs and voter ARMORs in the same manner.

The Backup FTMs are able to detect the crash faults in FTMs and the crash faults in Daemons on FTMs' nodes. Daemons are capable of detecting crash faults in common ARMORs by monitoring all the ARMORs installed by them.

Having different fault-tolerant mechanism modules inherited from the same parent class ARMOR, the Chameleon architecture achieves high cohesion and low coupling harmoniously.

6.4 Fault Treatment in Chameleon Architecture

Heartbeat ARMOR detects a crash failure in a node, link or daemon, and then it notifies the FTM, which in turn removes the node from the list of registered nodes and restarts any affected ARMORs it manages on a new node. After that, the FTM notifies its immediate managers of the crashed node; these managers restart any of their ARMORs and recursively notify all subordinate managers.

When an erroneous computation (value fault) is detected, the voter ARMOR has two choices: if the application is in dual mode, the voter ARMOR will restart the application and notify the user; if the application is in TMR mode, the voter ARMOR will mask the error and optionally notify the user.

When a crash failure occurs in common ARMORs, the corresponding Daemon will notify the crashed ARMOR's manager. The manager will try to reinstall the ARMOR on the same node, on a different node or on a different platform depending on the situation. The manager may also try to employ another ARMOR with similar functionalities. When a common ARMOR is found alive but unresponsive, the corresponding daemon will kill the ARMOR and then notify the ARMOR's manager; and the manager will reinstall the ARMOR accordingly.

When a crash fault is detected in a daemon, the querying Heartbeat ARMOR notifies the daemon's manager. The daemon's manager will treat a daemon failure as if the entire node has crashed and recovers as for the node failure.

A crash fault in FTM and a crash fault in FTM Daemon are treated in the same way. The corresponding backup FTM promotes itself to become the FTM and notifies all ARMORS that it manages directly of the change. All subordinate managers recursively notify the ARMORs managed by them respectively. Finally, new FTM promotes a new backup FTM from one of the surrogate managers.

As different managers and ARMORS handle different faults, fault treatment mechanisms are organized in such a way that high cohesion and low coupling are achieved without sacrificing each other.

7.0 Discussion

The GUARDS is a real-time-oriented architecture, which is naturally determined by its three major applications that are from railway, nuclear propulsion, and space domain. Consequently, GUARDS aims at containing and masking various faults because time redundancy is assumed unavailable, and more emphasis are given to fault diagnosis and fault

Table 1: A Comparison of Fault Tolerance Mechanisms of Different Architectures

	FRIENDS	AQuA	CHAMELEON	GUARDS
Fault Models	Physical crash failures	• Crash failure • Value and time faults • Fault-free links • No value faults in AquA gateway	• No specific assumptions • Differentiates faults occurring at different locations	Based on: • Location • Duration • Interdependence
Fault Detection Mechanisms	Organized into the library of meta-object classes	• Crash failures detected by Ensemble • Value and time faults detected by voters and monitors, respectively	Implemented in different ARMOR, Backup FTM, and Daemon	Achieved by the deliberate cooperation of redundant channels and lanes
Fault Treatment Mechanisms	Implements separate detection and handling mechanisms as meta-object classes	• Supports both active and passive replication • Differentiates between decision making and fault handling actions	Different managers and ARMORs handle different faults	Focuses on fault containments and fault masking

containment instead of fault recovery. These aspects make GUARDS more suitable for highly critical systems.

The AQuA architecture is unique in that it supports runtime reconfiguration of the system; whereas other architectures only support reconfiguration at compile time. As a result AQuA is able to offer changing Quality of Service (QoS) at runtime. AQuA is heavily dependent on the Ensemble system that is based on group communication methodology. Consequently there is no central control point in AQuA at all, which makes the decision making in AQuA inefficient. Currently, AQuA is still programming language dependent, which is due to his dependence on the Maestro system that is based on C++ language.

The FRIENDS architecture's fault tolerance ability is still very limited due to its weak fault model. However, FRIENDS holds a different philosophy on how to separate fault tolerance mechanisms from application programmers. While other architectures try to make the fault tolerance mechanisms totally transparent to application programmers, FRIENDS tries to distinguish fault tolerant programmers from application programmers; and the fault tolerant programmers' task is to customize meta-object classes and meta-object protocols that should be highly reusable. By this approach, the FRIENDS architecture will be open to any new technologies in the field and also more extensible. Another strength of FRIENDS is that it is fully object-oriented. FRIENDS supports C++ application only.

The Chameleon is a centralized architecture. The fault tolerance manager is able to oversee the whole system. The hierarchical arrangement of managers makes the system more efficient. Chameleon is object-oriented as well, providing the same level of extensibility as the FRIENDS architecture. It has a much better fault coverage than the AQuA and FRIENDS architectures, at the same time, it is not dedicated to critical real-time applications only. It suits a wider range of applications with different dependability requirements. Chameleon makes no assumption about programming language.

Table 1 shows the comparison of the four architectures in fault models, fault detection mechanisms and fault treatment mechanisms. We emphasize that this comparison in no way undermines other effective and efficient mechanisms being supported by each of these individual architectures.

8.0 References

[1]. M. Cukier, et al. "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," Proc. of IEEE Symposium on Reliable Distributed Systems, 1998.

[2]. J.-C. Fabre, T. Perennou, "A Meta-object Architecture for Fault-Tolerant Distributed Systems: the FRIENDS Approach," IEEE Transactions on Computers, 47(1), Jan. 1998.

[3]. Z. Kalbarczyk, R.K. Iyer, S. Bagchi, K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," IEEE Transactions on Parallel and Distributed Systems, 10(6), pp. 560-579, June 1999.

[4]. D. Powell, et al. "GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems," IEEE Transactions on Parallel and Distributed Systems, 10(6), pp. 580-599, June 1999.

[5]. D. Pradhan, Fault-Tolerant Computer System Design, Prentice-Hall, NJ, 1996.

[6]. D.P. Siewiorek, R. Swarz, Reliable Computer Systems: Design and Evaluation, 3/e, A.K. Peters, MA, 1998.

9.0 Acronyms

ERICA: Error-Resistant Interactively Consistent Architecture

FTMP: Fault Tolerance Multiprocessor

FTPP: Fault Tolerance Parallel Processor

MAFT: Multi-computer Architecture for Fault Tolerance

SIFT: Software Implemented Fault Tolerance

About the authors

Yang Liu holds a Bachelor Degree, with high distinction, in Computer Science from Concordia University. He is currently completing his M.A. Sc. in Telecommunication Software Engineering under the supervision of Professor Ferhat Khendek at Concordia University. Yang Liu won FCAR B1 scholarship in 2002.

Purnendu Sinha is an Assistant Professor in the Department of Electrical and Computer Engineering at Concordia University, Montreal. He obtained his Ph.D. in Computer Engineering from Boston University, Boston, MA. He received his M.S. degree in Computer Science from the New Jersey Institute of Technology, Newark, NJ, and also his M.E. degree in Electrical Engineering from the Stevens Institute of Technology, Hoboken, NJ. His research interests include design and analysis of distributed dependable and real-time algorithms, embedded systems, formal methods based verification and validation (V&V) of fault-tolerant and real-time protocols, fault-injection based validation, and real-time imaging.