Chameleon: A Software Infrastructure for Adaptive Fault Tolerance

Zbigniew T. Kalbarczyk, *Member*, *IEEE*, Ravishankar K. Iyer, *Fellow*, *IEEE*, Saurabh Bagchi, *Student Member*, *IEEE*, and Keith Whisnant

Abstract—This paper presents Chameleon, an adaptive infrastructure, which allows different levels of availability requirements to be simultaneously supported in a networked environment. Chameleon provides dependability through the use of special *ARMORs*—Adaptive, Reconfigurable, and Mobile Objects for Reliability—that control all operations in the Chameleon environment. Three broad classes of ARMORs are defined: 1) **Managers** oversee other ARMORs and recover from failures in their subordinates. 2) **Daemons** provide communication gateways to the ARMORs at the host node. They also make available a host's resources to the Chameleon environment. 3) **Common ARMORs** implement specific techniques for providing application-required dependability. Employing ARMORs, Chameleon makes available different fault-tolerant configurations and maintains run-time adaptation to changes in the availability requirements of an application. Flexible ARMOR architecture allows their composition to be reconfigured at run-time, i.e., the ARMORs may dynamically adapt to changing application requirements. In this paper, we describe ARMOR architecture, including ARMOR class hierarchy, basic building blocks, ARMOR composition, and use of ARMOR factories. We present how ARMORs can be reconfigured and reengineered and demonstrate how the architecture serves our objective of providing an adaptive software infrastructure. To our knowledge, Chameleon is one of the few real implementations which enables multiple fault tolerance strategies to exist in the same environment and supports fault-tolerant execution of substantially off-the-shelf applications via a software infrastructure only. Chameleon provides fault tolerance from the application's point of view as well as from the software infrastructure's point of view. To demonstrate the Chameleon capabilities, we have implemented a prototype infrastructure which provides set of ARMORs to initialize the environment and to support the dual and TMR application execution modes. Through this testbed environment, we measure the execution overhead and recovery times from failures in the user application, the Chameleon ARMORs, the hardware, and the operating system.

Index Terms—Adaptive fault tolerance, high availability networked computing, software-implemented fault tolerance, COTS, extendible modular architecture.

1 Introduction

Traditionally, fault tolerance has been provided through dedicated hardware, dedicated software, or a combination of both. In the case of hardware-based fault tolerance, manufacturers such as Tandem provide stand-alone machines with high reliability through extensive hardware redundancy. Unfortunately, dedicated fault-tolerant architectures offer a static level of fault tolerance that remains fixed throughout the lifetime of the machine. Moreover, these architectures are often oriented toward specific classes of applications.

Distributed environments employ software-based solutions to provide dependability. Typically, services are replicated throughout the network to provide the requisite level of redundancy. The applications, however, must usually be written with the intent to run in such an environment, so the benefits of such a system go unnoticed to existing applications.

In contemporary networked computing systems, a broad range of commercial and scientific applications must coexist—each potentially requiring a different level of availability and reliability. It is not cost effective to provide dedicated hardware-based fault tolerance to each application, nor is it cost effective to rewrite each application to take advantage of the fault tolerance incorporated into a distributed network through specialized software. The pressing issue then becomes the best way to achieve high dependability for commercial off-the-shelf (COTS) applications running on off-the-shelf hardware.

It should be emphasized that although applications that require ultradependability may continue to operate on dedicated hardware, there is considerable interest in providing fault tolerance in networked systems using COTS hardware and software. For example, in the Remote Exploration and Experimentation (REE) Project undertaken by the Jet Propulsion Laboratory (JPL) and NASA, the key objective is to design and develop a computing architecture suitable for space flights, which offers software-implemented fault tolerance for executing NASA applications and which employs, to the extent possible, COTS technologies. Examples of candidate applications to execute in such an environment include:

- computation-intensive, hand-parallelized applications, e.g., image processing. The image processing is usually decomposed into several computation segments with potentially varying levels of reliability. In

- *The authors are with the Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 W. Main St., Urbana, IL 61801*
  E-mail: {kalbar, bagchi, kwhisnan, iyer}@crhc.uiuc.edu.

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 109049.

an example scenario, the collection and initial processing of raw image data is usually less critical and, consequently, a simplex execution mode may be sufficient for this computation segment. Further processing of the raw data converts the data into information and errors can damage the information. Consequently, this computation segment may require a high level of reliability in software execution. This could be achieved by executing the critical segment of the application in the Triple Modular Redundancy (TMR) mode.

- distributed data base applications that need reliable access to the information stored in the data base. In such applications data integrity and confidentiality must not be violated by any failures in the application's execution.

These two applications are example target areas for a software infrastructure capable of adapting to dynamically changing availability and reliability requirements. In this paper, we propose Chameleon, an adaptive infrastructure which allows different levels of availability requirements to be supported concurrently in a networked environment. Chameleon provides dependability through the use of ARMORs—Adaptive, Reconfigurable, and Mobile Objects for Reliability. ARMORs are components that control all operations in the Chameleon environment; they can be broadly classified into three groups:

1. **Managers**. Managers oversee other ARMORS and recover from failures in their subordinates. Primary managers include the *Fault Tolerance Manager* (FTM), the highest-ranking manager, and *surrogate managers,* which oversee the fault-tolerant execution strategy of specific user applications.
2. **Daemons.** Daemons allow Chameleon to access nodes in the network, provide ARMOR error detection, and provide the means through which ARMORs may communicate among themselves across the network.
3. **Common ARMORs.** Common ARMORs implement specific techniques for providing application-required dependability. Examples of common ARMORs include execution ARMORs, voter ARMORs, checkpoint ARMORs, and heartbeat ARMORs.

Throughout this paper, the term *ARMOR* refers to any instance of the above three groups (not only common ARMORs, but also managers and daemons).

We choose to provide dependability to a user application through ARMORs because they allow implementation flexibility. Since fault-tolerant techniques are encapsulated in ARMORs, Chameleon need only use those ARMORs that provide the required level of availability. For example, if checkpointing is not needed for a specific user application, the checkpoint ARMORs do not need to be present in the fault-tolerant execution strategy for the application, thus reducing the overhead of Chameleon as seen by the user application. In addition to providing a highly customizable fault-tolerant environment, ARMORs are location-independent—they can perform their actions while executing on any node in a heterogeneous network. This becomes crucial when tolerating node and hardware failures. The ARMORs also provide a convenient mechanism whereby new functionality can be introduced into the system at a later time without disturbing existing functionality.

ARMOR technology is a vehicle used for providing adaptive fault tolerance. By *adaptive fault tolerance* we understand the system's ability to adapt dynamically to changes in the fault tolerance requirements of an application. This is achieved by making the Chameleon infrastructure (by design) statically and dynamically reconfigurable. Static reconfiguration guarantees that ARMORs can be reused for assembling different fault tolerance execution strategies. Dynamic reconfiguration allows 1) an ARMOR's functionalities to be extended or modified during runtime by changing the ARMOR's composition and 2) ARMORs to be added to or removed from the system at runtime without taking down other, already active ARMORs.

Note that our notion of adaptive fault tolerance extends the concept of adaptation in fault tolerance used in the system (resource) management community, where issues such as availability or replica management are studied. While the early Chameleon implementation does not explicitly address the resource management issue, the ARMOR technology can be employed to implement resource management services as well.

We should emphasize that the objective of this paper is not to provide formal proofs of all concepts and ideas involved in building the Chameleon infrastructure. Rather, our primary goals are to present the system architecture and to demonstrate how this architecture serves our objective of providing an adaptive software infrastructure, one that offers varying levels of availability in application execution. We are engaged in an ongoing effort to develop formal specifications of the system.

The remainder of the paper is organized as follows: Section 2 discusses some related research. A behavioral overview of Chameleon is presented in Section 3. Section 4 provides an in-depth look at the functionality of the Chameleon ARMORs. Details of the error detection and recovery mechanisms incorporated into Chameleon are discussed in Section 5. Section 6 provides some experimental results from an early implementation of Chameleon. Section 7 discusses the underlying reconfigurable architecture of Chameleon and how this architecture can be used to provide adaptive fault tolerance. Section 8 provides an insight into Chameleon enhancement for supporting real-time applications. Section 9 concludes the paper.

2 Related Research

Most current approaches for providing fault tolerance in a network of unreliable components are based mainly on exploiting distributed groups of cooperating processes. Consequently, the primary focus is on providing a dedicated software layer to maintain and coordinate reliable communications among groups of processes. Over the last several years, the group communication paradigm has been employed as a key premise in designing and implementing many distributed systems.[1]

- ISIS provides tools for programming with process groups. By using these tools, a programmer may construct group-based software that provides reliability through explicit replication of code and data [6]. Horus, a new generation of ISIS, introduces a flexible group communication protocol that may be constructed by stacking well-defined microprotocols at run-time [29].
- Totem attempts to provide high performance and soft real-time guarantees to applications by providing a hierarchy of group communication protocols that is capable of delivering messages to member processes in the presence of communication and processor failures [24].
- Transis incorporates multicast services that are capable of recovering from network partition failures [12], [2].
- Rampart addresses security aspects of group communication by providing tolerance for malicious intrusions [28].

These systems are more concerned with group communication than with fault tolerance. Although reliability may be achieved through the use of these protocols, "fault tolerance," Birman notes [5], "is something of a side effect of the replication approach." There exist, however, examples of well-known systems which explicitly address the issue of fault tolerance.

- SIFT was one of the earliest attempts to propose a completely software-based approach to fault tolerance through loose synchronization of processors and memory, [34].
- Delta-4 sought to define and design an open, dependable, distributed architecture through the use of group communication layers built on top of an atomic multicast protocol. In addition, Delta-4 employs a specialized network attachment controller to support a fail-silent failure semantic [26], [27], [4].
- Piranha, an extension to Horus via the CORBA[2]-interface provided by Electra, addresses the issue of service availability in distributed applications by using a highly sophisticated ORB that provides failure detection [23].
- AQUA architecture provides a flexible approach to build dependable, object-oriented distributed systems while offering a standard CORBA interface to applications, [11].
- FRIENDS [13] and MAUD [1] advocate a software-based, metalevel architecture to realize goals of adaptation and reconfiguration for fault tolerance.
- ROAFTS offers a middleware architecture for real-time, object-oriented, adaptive fault tolerance [18].
- "Wolfpack," the Microsoft® clustering technology provides clustering extensions to Windows NT® for improving service availability and system scalability. Although this approach is not based on the process group paradigm, it maintains functions typical of the operation of a distributed environment, including maintaining cluster membership and sending periodic heartbeat messages to detect system failures [36].
- At Sun Microsystems, work has been done on Ultra Enterprise Cluster design to provide highly available data services. The Ultra Enterprise Cluster High Availability server provides automatic software-based fault detection and recovery mechanisms. Specialized software allows a pair of two computing nodes to monitor each other and redirect data requests in the case of software or hardware failure [33].
- Work at Lucent Bell Labs has focused on increasing the availability and data consistency of applications through the application's use of reusable components for automatic detection and recovery of failed processes and for preserving data consistency [17].

The process-group computing model has also been used in real-time systems such as the Advance Automated systems for the Air Traffic Control network being developed by IBM [10], and by MARS [19], a distributed system for real-time control. These technologies aim at providing strong real-time guarantees by employing accurate measuring of timing properties of the network and the hardware to achieve highly predictable behavior.

The issue of adaptation in fault tolerance has also been studied from the system resource management perspective.

- Availability management service ensures that the critical services of a distributed system remain continuously available to users despite of node failures. The availability manager is able to automatically enforce availability policies (e.g., primary-backup) and to reconfigure a system in the presence of failures [8], [7].
- Replica management service (subsystem) allows a programmer to specify the quality of service required for individual replicated objects in terms of availability and performance. The service is intended for long-running applications, which may require a dynamic, adaptive replication policy to ensure long-term availability [22].
- The object replication service, which takes advantage of the application-specific knowledge (e.g., interdependencies among the objects) achieves better performance and availability in a distributed system that supports replication [21]. The service is implemented as part of Arjuna, an object-oriented C++ based programming system that provides set of tools for the construction of fault-tolerant distributed applications [31].

1. The review presented here is not intended to be comprehensive. Rather, an attempt is made to illustrate the major trends in the area of distributed computations. For a thorough analysis of the issues related to group communications and for a more complete characterization of existing systems, refer to [4]. Issues related to distributed fault tolerance are covered in [9].
2. The Common Object Request Broker Architecture (CORBA) [25] is emerging as a major standard for supporting object-oriented distributed environments with the design goals of heterogeneity, interoperability and extensibility. While vendors are increasingly providing applications conforming to CORBA specifications, the vast majority of the existing applications are not built around CORBA objects.

Many of the systems proposed require a specialized and often a complex software layer and/or additional hardware in order to provide group communication and good coverage for fail-silent behavior. Most provide an *environment* through which a programmer can construct a distributed application and provide fault tolerance through replication. Chameleon explicitly provides fault tolerance through a wide range of error detection and error recovery mechanisms for both applications and Chameleon entities. Several of the above mentioned systems detect failures solely through the use of timeouts and some do not even mandate that recovery be initiated once failures have been detected. Finally, Chameleon tries not to make any assumptions concerning the fail-stop behavior of any of its entities. Of course, the coverage of our error detection mechanism is the overriding factor in determining whether such a claim is reasonable.

3 Overview of Chameleon

In this section, we examine the steps that Chameleon takes to execute a user application using a fault tolerance execution strategy. We also introduce the key components of the Chameleon environment. Note that this section is primarily concerned with introducing the error-free behavior of Chameleon; Chameleon's error detection and recovery techniques are discussed in detail in Section 5.

3.1 Initialization of the Chameleon Environment

Essentially any network of unreliable nodes can be configured to participate in the Chameleon environment. The highest ranking manager in the environment, called the Fault Tolerance Manager (FTM) is installed on an arbitrary node in the network. After being successfully installed, the FTM invokes a *daemon* (to handle communications with remote hosts) and a *heartbeat ARMOR* (to detect failures of remote nodes) on the local computation node. Finally, the FTM creates a *Backup FTM,* (if the FTM operates in primary-backup execution mode) which closely monitors the FTM and, upon detecting an error, promotes itself to become the new FTM. Once the Backup FTM is set up, we have a stable Chameleon environment ready to accept and serve user requests. Note that depending on the reliability requirements, the FTM can operate in different execution modes, e.g., the triple modular redundancy configuration.

Other nodes in the network can request to join the Chameleon environment through the FTM. Upon a new node request to join the infrastructure, the FTM sends the necessary code to compile and execute a *daemon* on the node wishing to join Chameleon.[3]

3.2 Interpreting User-Specified Dependability Requirements

The Chameleon environment allows the user to run several different applications in a fault-tolerant manner—each application potentially having different availability and reliability requirements. The user submits an application to the FTM with availability requirements specified in a semantic language that the FTM understands and the FTM selects an appropriate *fault tolerance execution strategy* (*FTES*) through which the fault tolerance requirements can be met. To accomplish this selection, the FTM has a registry of several FTES (e.g., dual execution mode with checkpointing, triple modular redundancy, etc.) and their associated semantic descriptions.

It is important to emphasize that these are not fixed execution strategies—other execution strategies can be easily developed using the Chameleon components and mechanisms to be described later. Section 7 discusses the reconfigurable architecture through which Chameleon can be extended.

After the FTM selects a particular execution strategy, it chooses or creates a corresponding surrogate manager to carry out the fault-tolerant execution of the user-supplied application. In general, there is one surrogate manager for each user application. The surrogate controls the fault-tolerant execution strategy, thus freeing the FTM from having to manage the application being executed and making the FTM more responsive to other events such as new user requests.

3.3 Invoking a Fault-Tolerant Execution Strategy

Once the FTM selects an appropriate surrogate manager to execute the application, the FTM *installs* the surrogate manager on a node in the Chameleon network (i.e., a node with a daemon installed as described in Section 3.1). Installing the surrogate manager consists of sending the surrogate manager code and required libraries of Chameleon components to the daemon of the node on which the surrogate manager is to be installed. The daemon receives the code, compiles it, and runs the resulting executable. Recompiling at the destination node gives the added flexibility of multiplatform support.

Surrogate managers use the common ARMORs introduced in Section 1. All Chameleon ARMORs are registered with the FTM and are available for use by any surrogate manager (or any manager for that matter). When a surrogate manager begins execution, it typically installs the common ARMORs needed to complete its assigned task. For example, the surrogate manager responsible for executing an application in TMR mode installs three copies of an Execution ARMOR (one for each application replica) and a Voter ARMOR. After the surrogate manager successfully installs the ARMORs necessary to execute the user application, it tells the FTM where the individual ARMORs are located. Consequently, the FTM has a global view of the system for use during error recovery.

3.4 Chameleon Software Architecture and the Overall System Architecture

Chameleon can operate over a network consisting substantially of commercial off-the-shelf (COTS) hardware (processing elements and network) and software (operating system). Fig. 1 places the Chameleon infrastructure in the context of the overall system architecture. The core of Chameleon is a reconfigurable ARMOR architecture consisting of a library of reusable fault tolerance techniques. This core constitutes the basis for creating different instances of ARMORs and configuring the system to

3. Since the node wishing to join the Chameleon environment must be able to communicate with the FTM, a lightweight background process called the *initialization process* exists for this purpose.

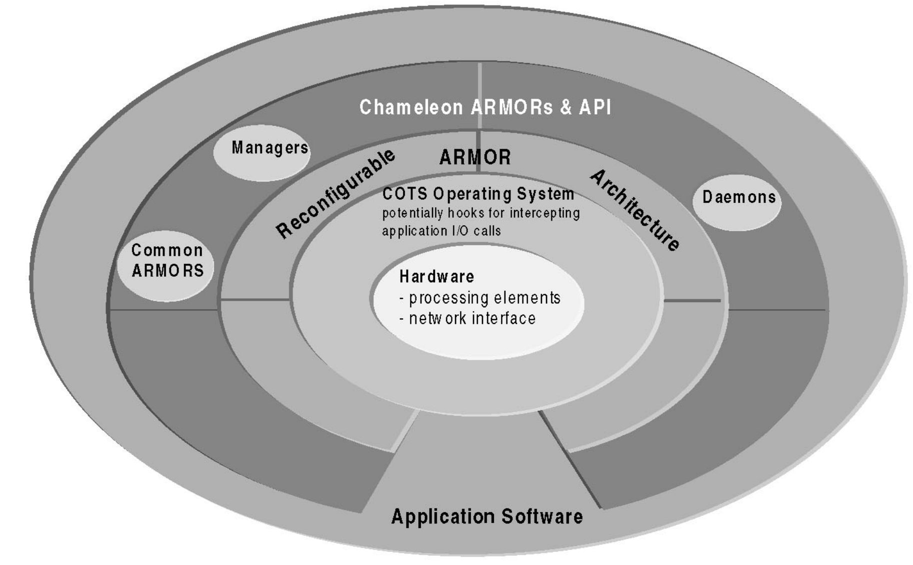


Fig. 1. Chameleon in the overall system architecture.

support specific fault-tolerant execution strategies. The three classes of ARMORs (managers, daemons, and common ARMORs) are employed to establish and to maintain the required system configuration. In addition, Chameleon offers an application programmer interface (API) available for software developers to use when implementing an application.

From the user perspective, the following operational modes are available for fault tolerance management using Chameleon:

Chameleon-unaware application

1. *Black box approach:* Chameleon treats the application as a black box. The fault tolerance can be achieved using application replication, voting on the computation results (fault masking), and detection of an abnormal termination and restart.
2. *Black box approach with use of hooks:* Chameleon uses hooks to the operating system for intercepting application I/O calls. The fault tolerance can be achieved using system checkpointing and rollback in addition to the techniques of the basic black box approach.

Chameleon-aware application

1. *Application written using Chameleon API*: This approach uses application-level checkpointing and rollback, voting on intermediate results or the application state, and built-in application error detection. The application notifies the Chameleon layer of any error.
2. *Application implemented as an ARMOR*: The application benefits from the entire spectrum of error detection and recovery mechanisms available to ARMORs.

A basic rule for ensuring the reliability of ARMORs in the Chameleon system is that every ARMOR is managed by another ARMOR. To this end, a chain of error detection and recovery is established between the ARMORs so that there is at least one ARMOR responsible for detecting errors in any given ARMOR and exactly one ARMOR for initiating recovery. More detailed discussion on the hierarchy of error detection and recovery in Chameleon is given in following sections.

4 Design and Implementation of Chameleon ARMORs

In this section, we discuss the specific responsibilities of the individual ARMORs in Chameleon and provide some details of the features that are implemented in the prototype Chameleon system. For some of the ARMORs, the implementation is more restrictive than the design and we present the broad design followed by what is implemented in the prototype.

4.1 Managers

Managers are specialized ARMORs that possess the following common capabilities:

- The ability to remotely install and uninstall ARMORs on other nodes. The manager-subordinate relationship is established at the time of ARMOR installation, with the installing entity being denoted as the manager. An ARMOR is uninstalled when we are guaranteed that its results will no longer be needed even under the most pathological error conditions. In the conservative case, this is when the application execution has finished.
- The ability to assign system-wide unique identifiers to the ARMORs installed by managers.
- The ability to maintain an up-to-date list of their subordinate ARMORs (i.e., ARMORs installed by the manager on remote hosts) and a mapping of their identification numbers to the nodes on which they are installed. For example, the FTM (and the Backup FTM) maintains a global table for all the

ARMORs in the system. The table contains the type and system-wide unique ID of the ARMOR, the manager of the ARMOR, and the location of the ARMOR.

The remainder of this section describes specific examples of managers in the Chameleon environment.

4.1.1 Fault Tolerance Manager (FTM)

The FTM is a centralized, key manager of the Chameleon environment. It has the following functionalities:

- Interfacing with the user to accept the application for the environment and communicating the final results of the run back to the user.
- Interpreting the user's dependability specifications for the application and mapping it into one of the available fault-tolerant strategies.
- Initializing parameters such as the system heartbeat interval, threshold number of failures (used by the Checkpoint ARMOR as a threshold for the number of local recovery attempts from application failures) and supporting runtime changes of these parameters.
- Determining the hosts that will be used to support the selected fault-tolerant execution strategy. Criteria used in the selection process may include the current load on a prospective node and the history log of failures of previous applications on the target node.[4]
- Maintain a list of available nodes in the system, along with properties like the platform, the OS running, the memory available. The list of hosts available for use by the FTM may change dynamically because of node failures, nodes voluntarily leaving the environment, or new nodes joining the environment.

4.1.2 Surrogate Manager (SM)

Surrogate managers are specialized managers with the following features:

- It is responsible for executing a particular application under a specific fault-tolerant execution strategy (FTES).
- It is capable of installing any other ARMORs required for executing an application under the FTES selected by the FTM.

4.1.3 Backup FTM

The Backup FTM is provided to guard against the FTM becoming a single point of failure. It has the following two functions:

- It is used to detect faults in the FTM and the daemon on the FTM's node and to initiate recovery by promoting itself to the FTM. This is done by invoking a heartbeat ARMOR (separate from the one used by the FTM) which is used to heartbeat just the FTM.
- It maintains its state consistent with that of the FTM by accepting updates from the FTM whenever there is a state change. The state of the FTM includes 1) global knowledge of all the ARMORs active in the system, 2) applications executing in the environment, and 3) history of hardware failures in the nodes in the environment.

4. For now, we are using the simple metric of the total number of ARMORs installed on the node to ascertain the node's workload.

4.2 Common ARMORs

4.2.1 Heartbeat ARMOR

This is an elementary common ARMOR invoked by the FTM to query the status of nodes in the environment. The Heartbeat ARMOR, in its simplest incarnation, can use a simple ICMP (Internet Control Message Protocol) ping message to determine if the node is alive or not. At the other end, the heartbeat message can be quite sophisticated and can encapsulate within it information about the health of the node being monitored. For example, the heartbeat can collect information about the number of errors in memory or I/O in the last heartbeat interval.

The host daemon on a particular node responds to the Chameleon heartbeat. The absence of a heartbeat acknowledgment from a node may be due to reasons other than the node being down. For example, the machine may be too slow to respond within the specified heartbeat interval. The heartbeat message, therefore, must fail to reach the node a certain number of times before the Heartbeat ARMOR declares the machine as being down.

4.2.2 Execution ARMOR

This is the basic ARMOR responsible for installing an application on a particular host, overseeing its execution, and communicating the result of the application back to the manager. The flowchart representing its actions is presented in Fig. 2. Blocks 5-10 denote actions performed by the Execution ARMOR.

4.2.3 Checkpoint ARMOR

The Checkpoint ARMOR provides API calls (*take_checkpoint()* and *recover()*) and interacts with the Execution ARMOR to enable the checkpointing and recovery of an application running on a particular node. Pseudocode for interaction between the Execution ARMOR, the Checkpoint ARMOR, and the application is presented in Fig. 3. The application is linked with the Checkpoint ARMOR and is run by the Execution ARMOR. The Checkpoint ARMOR can be invoked in one of the following ways:

- The user makes explicit calls to the Chameleon checkpoint function in the body of his application.
- The user provides markers in his application denoting appropriate places to take checkpoint. This is then modified, transparent to the user, to insert calls to the checkpoint function.
- The Execution ARMOR calls the Checkpoint ARMOR with a default time interval. In this case, the user application is modified to generate a signal every *default checkpoint interval*. In the signal handling routine, we then make the calls to take checkpoint.

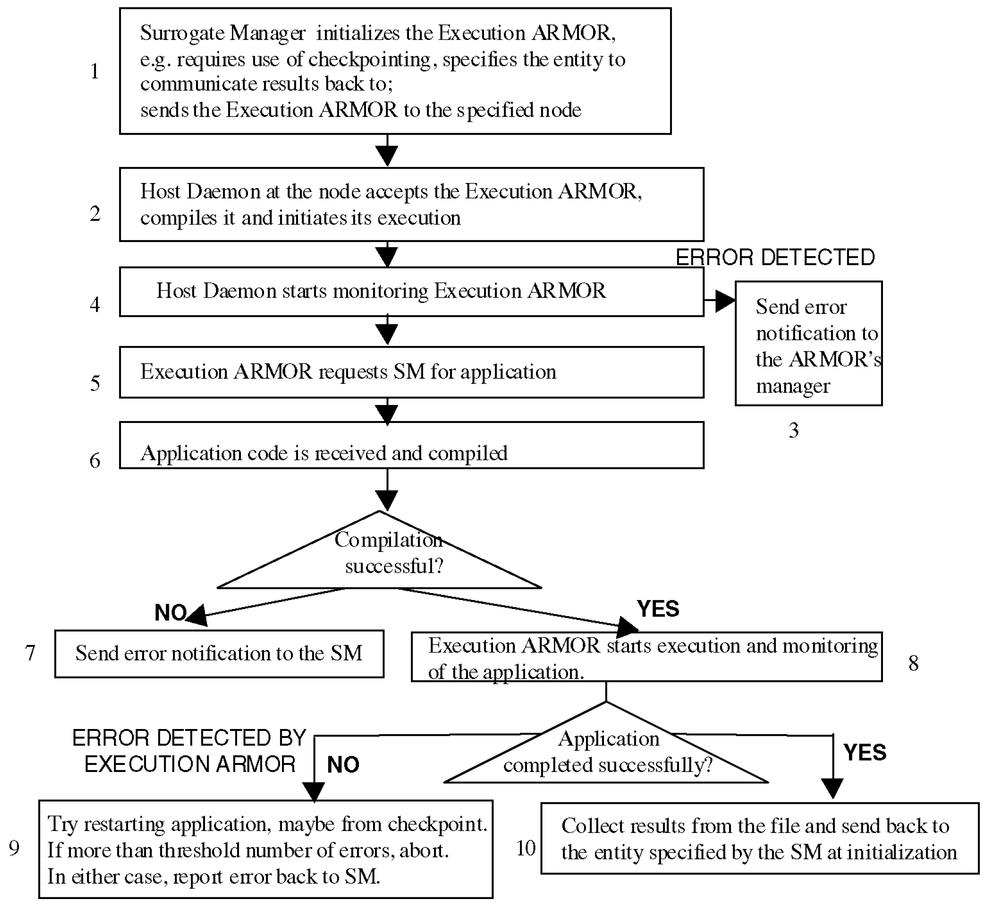

1 Surrogate Manager initializes the Execution ARMOR, e.g. requires use of checkpointing, specifies the entity to communicate results back to; sends the Execution ARMOR to the specified node
2 Host Daemon at the node accepts the Execution ARMOR, compiles it and initiates its execution
ERROR DETECTED
3 Send error notification to the ARMOR's manager
4 Host Daemon starts monitoring Execution ARMOR
5 Execution ARMOR requests SM for application
6 Application code is received and compiled
Compilation successful?
NO
YES
7 Send error notification to the SM
8 Execution ARMOR starts execution and monitoring of the application.
ERROR DETECTED BY EXECUTION ARMOR
NO
Application completed successfully?
YES
9 Try restarting application, maybe from checkpoint. If more than threshold number of errors, abort. In either case, report error back to SM.
10 Collect results from the file and send back to the entity specified by the SM at initialization

Fig. 2. Flowchart for installation and execution of an application on a remote node.

The Checkpoint ARMOR is also used during recovery. When the Execution ARMOR detects an application failure, it passes control to the checkpoint ARMOR which rolls back to the last saved checkpoint available in stable storage and restarts the application from there. If local recovery fails a certain threshold number of times, the Execution ARMOR notifies its manager to initiate recovery on a different node. To support checkpointing on different platforms (e.g., UNIX, Windows NT) Chameleon provides a library of Checkpoint ARMORs implemented for different platform and available to the application.

4.2.4 Voter ARMOR

The voter can be used to vote on arbitrary number of inputs and return the majority for an arbitrary k-of-n match. Different voting strategies and characteristics may be obtained by overriding the default behavior of the voter ARMOR (e.g., success on exact match or success on match within 10 percent variation). At initialization of the voter, the surrogate manager specifies the entities from which it is expecting results, the timeout period, and the entity to which it has to communicate its result.

A critical voter parameter is the timeout interval for which the voter waits for the results to come from the ARMORs. Initially, the timeout is determined based on the specifications from the user who supplied the application. During run-time, the timeout is tuned depending on the relative speed of machines which execute the application. For example, consider an application which is executed in TMR mode with additional checkpointing to support recovery from errors. In this scenario, each Execution ARMOR measures the application execution time. After a fixed number of checkpoints, the ARMOR sends the measured time to the voter. The voter compares the collected times and readjusts the voting timeout according to the ratio of the worst (i.e., the longest) to the best (i.e., the shortest) time reported by the ARMORs. For example, if the timeout was initialized to 5 sec and the three measurements (arrived from the three ARMORs) are 1.3 sec, 1.1 sec, and 2.2 sec, the voting timeout is readjusted to the value of 10 sec (i.e., 5 * (2.2 / 1.1)) to take into account the performance of the slowest machine.

4.2.5 Initialization ARMOR

After the FTM has installed a host daemon on a node, the FTM sends the Initialization ARMOR to a particular node. During execution, the Initialization ARMOR collects information about the OS type, the hardware type, memory resources available. It then brings this information back to the FTM so that the FTM can update its table of the functional hosts in the environment.

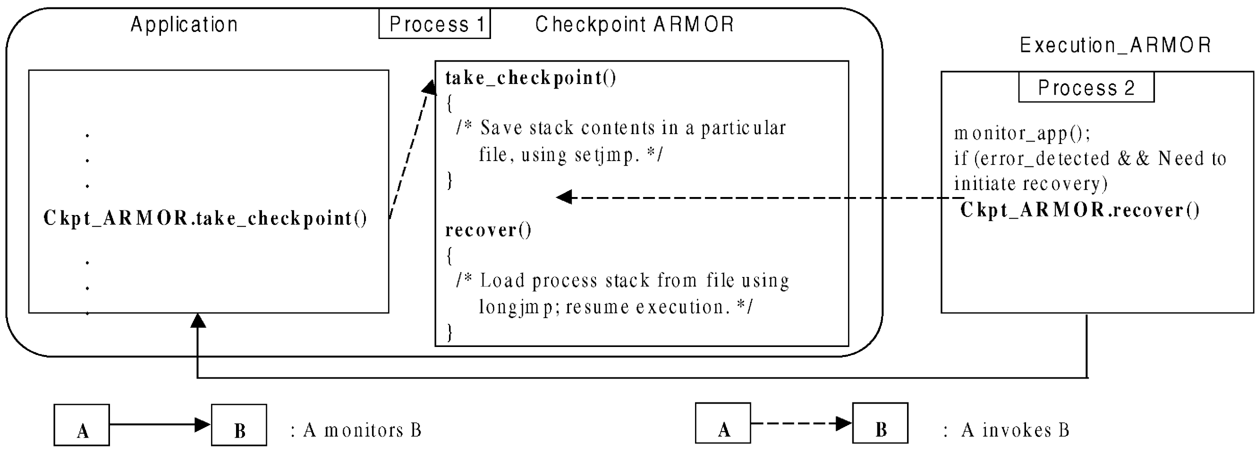

Application
Process 1
Checkpoint ARMOR
Ckpt_ARMOR.take_checkpoint()
take_checkpoint()
{
/* Save stack contents in a particular file, using setjmp. */
}
recover()
{
/* Load process stack from file using longjmp; resume execution. */
}
Execution_ARMOR
Process 2
monitor_app();
if (error_detected && Need to initiate recovery)
Ckpt_ARMOR.recover()
A
B
: A monitors B
A
B
: A invokes B

Fig. 3. Interaction of checkpoint ARMOR, execution ARMOR and application.

TABLE 1
Examples of Primitive Functions and their Use

Primitive function	Invoked by	Function
*install_* ARMOR	Surrogate manager	Installs ARMOR on a particular host
get_appinfo	FTM	Collects information from the user specification about the requirements of the application
voter	Surrogate manager	Votes on a parameterized number of results. Can be passed the entities from which results are expected and the comparison scheme to be followed.
*monitor_* ARMOR	Host Daemon	Monitor one of the ARMORs installed on the host. The monitoring in the current implementation is done by trapping illegal signals raised and by timeout intervals.
send_ready_notif	Execution ARMOR	Send notification to the SM that it is ready to accept application fo execution.

4.2.6 Fanout ARMOR

The Fanout ARMOR provides consistent data distribution among multiple replicas. It uses a commit protocol and a totally ordered multicast protocol to ensure that either all the replicas see the identical sequence of inputs or none of them sees it. Details of the application of this ARMOR are provided with the discussion on application support in Section 4.6.

4.3 Daemons

Daemons are entities resident on every participating node and perform the following functions in the Chameleon environment:

- Install ARMORs locally on the node. Daemons perform the low-level installation of an ARMOR on a node, i.e., spawn a new process, set up an appropriate communication channel between itself and the ARMOR, notify the FTM as to the location of the newly installed ARMOR, etc. This is exemplified in block 3 of Fig. 2.
- Monitor all locally installed ARMORs. Details of the error detection provided by the daemons can be found in Section 5.3. Block 4 of the Fig. 2 represents this activity.
- Serve as the primary gateway for all communication between ARMORs in the Chameleon environment. Section 4.5 describes the ARMOR communication process in more detail.
- Respond to heartbeat messages from the heartbeat ARMOR.

4.4 ARMOR Structure

The ARMORs are composed of a set of primitive functions. Table 1 provides examples of some primitives used in the current Chameleon implementation. Note that the basic functions can be reused by multiple ARMORs. For example, the *install_*ARMOR primitive can be used by all managers. Similarly, the same common ARMORs can be reused in multiple FTES. Thus, the same Execution ARMOR is put to use in the Dual as well as the TMR mode of execution.

4.5 ARMOR Communication

The host daemons perform the actual network communication in the Chameleon environment. Hence, their implementation is specific to a particular network protocol. Our current implementation is built on TCP/IP because of the portability and ease of implementation it offers.

The communication infrastructure is schematically presented in Fig. 4. Chameleon uses the event-driven model for cooperation between its ARMORs. The events are triggered by messages arriving at the message queues of the entities. To send a message, an ARMOR passes the message to the local host daemon through an IPC (Inter-Process Communication) channel. The daemon does a lookup and translates the ARMOR ID into the location of the node on which the ARMOR resides. Before sending the message over a TCP socket, the daemon adds a header to the message which

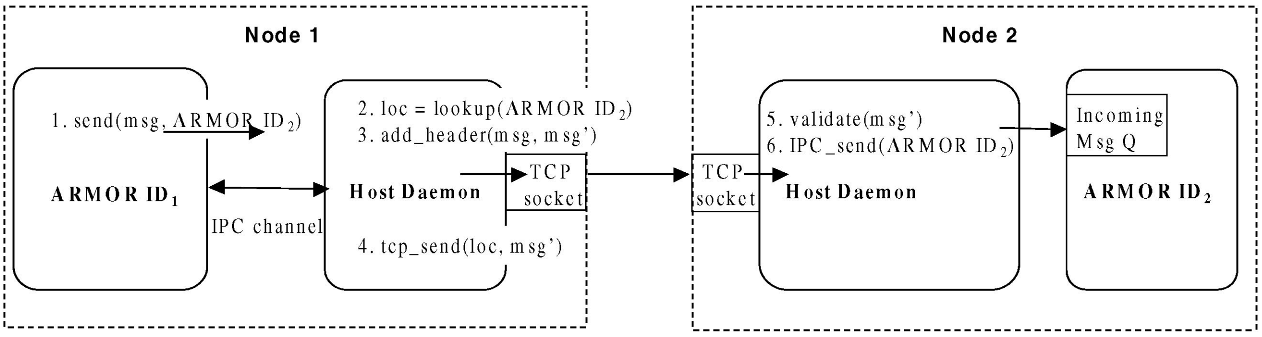


Fig. 4. ARMOR communication mechanism.

contains source and destination identifiers and, possibly, a checksum or a CRC to provide an extra layer of protection. When receiving messages, the daemon does validity checks on the message (such as checking that the destination ARMOR of the message is locally resident or not). The daemon then forwards the message to the appropriate local ARMOR over the IPC channel.

4.6 Application Support

Chameleon seeks to support off-the-shelf, stand-alone, as well as distributed applications. Stand-alone applications are those in which different tasks constituting the application do not exchange state. In this scenario, the stand-alone application may, however, take intermediate inputs. This may lead to the problem of replica nondeterminism, i.e., replicas of the stand-alone application may take different execution paths leading to results that cannot be voted upon meaningfully. In order to ensure that all the replicas see identical inputs in identical order, the *Fanout ARMOR* is interposed between the source of the input and the replicas. The *Fanout ARMOR* uses totally ordered multicast to deliver the inputs to each of the replicas.

For distributed applications, the synchronization between the tasks presents a more challenging problem. Consider a case in which an application is parallelized into several tasks and these tasks are communicating amongst them. This communication must be visible to the Chameleon entities so that they can take appropriate actions, such as sending the communication to multiple replicas. This can be achieved in one of the following two ways. The application is aware that it is executing in the Chameleon environment. So, when it is the sender, it routes its communication through a Chameleon ARMOR and, when it is the receiver, it expects the communication to come through an ARMOR. This requires the application to be written for the environment. Alternatively, we can insert hooks in the operating system to trap all network I/O calls, inspect such calls and, if found to be from a Chameleon application, invoke appropriate actions. Such action can be to send the communication to a Voter ARMOR and, after voting, to send the result to a Fanout ARMOR for multicasting to the multiple replicas of the task for which the communication was meant. The trapping of the network I/O calls at a host can be accomplished through a specialized driver installed into the host operating system. The drawback of this approach is that we no longer have a truly off-the-shelf OS and the driver may not be easily portable across platforms. Note that the OS still supports off-the-shelf applications.

Fig. 5 illustrates a common application scenario and how the replication management is done using *Fanout ARMOR*. Suppose we have a client-server based application where the client is updating a replicated database. For a particular fault tolerance execution strategy, we replicate the querying clients, the database servers, and the database itself. To synchronize updates to the database, we interpose the Chameleon *Fanout ARMOR*, which is responsible for providing consistent information to all replicas of the database application.

In our current implementation, we support off-the-shelf stand-alone applications and a limited set of distributed applications where the tasks do not share state but can synchronize among themselves at marked points in their execution. At these synchronization points, we can interpose voters to vote on the intermediate results. The results to be voted upon are required to be in files in the current implementation.

5 Error Detection and Recovery

Chameleon is designed to recover from transient and permanent hardware faults. Such faults can affect the hardware or operating system (e.g., node crash, link failure), application (e.g., abnormal termination), or Chameleon ARMORs (e.g., an ARMOR crash). The question to be asked here is: Do these faults occur in practice? Our field experience in analysis of failure data on real systems (e.g., Tandem [20], [15], IBM [32], and networks of workstations) demonstrates that these faults do occur in real operational environments and that they can impact the dependability. The remaining part of this section provides a complete picture of the detection and recovery techniques in Chameleon from a design perspective. A comprehensive description of hierarchy of ARMOR error detection and recovery can be found in [35].

5.1 Failures in the Hardware and the Operating System

In most cases, a failure in the operating system or hardware associated with a particular computational node will make that node inaccessible to the rest of the network. To detect a node failure, the Heartbeat

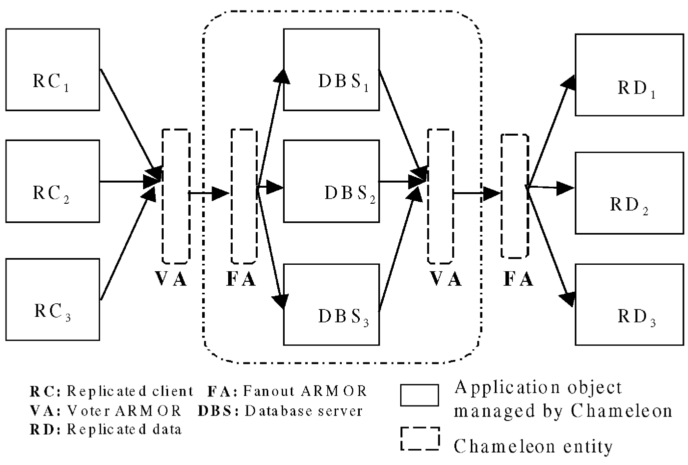

RC1
RC2
RC3
VA
FA
DBS1
DBS2
DBS3
VA
FA
RD1
RD2
RD3
RC: Replicated client FA: Fanout ARMOR
VA: Voter ARMOR DBS: Database server
RD: Replicated data
Application object managed by Chameleon
Chameleon entity

Fig. 5. Use of Chameleon for a replicated database application.

ARMOR periodically sends heartbeat messages to each daemon in the network. When the Heartbeat ARMOR fails to receive an acknowledgment from the daemon, the Heartbeat ARMOR notifies the FTM of a possible node/daemon failure.[5] Upon receipt of an MSG_NODE_DOWN notification message, the FTM first checks to see if any of its immediately subordinate ARMORs (e.g., surrogate managers) are installed on the failed node. If so, the FTM initiates recovery by reinstalling these ARMORs on different nodes. It then systematically sends MSG_NODE_DOWN messages to all subordinate managers, who in turn reinstall any affected ARMORs and propagate the notification message downward so that all managers in the Chameleon environment receive notification of the failed node.

In the case of non-fail-silent node behavior, the node may continue to send heartbeats despite the failure. The resulting application misbehavior can be captured by 1) the local Execution ARMOR if the node failure did not compromise the execution ARMOR, 2) the Host Daemon on the remote computation node to which the faulty node sent a message (e.g., a message sent to an ARMOR that does not exist on the node which received the message), and 3) the Voter ARMOR which detects incorrect computation results.

Note that Chameleon does not distinguish between node and network failures. An efficient means to cope with link and switch failures is to use the redundant network. In this respect, ServerNet from Tandem is the only, commercially available fault-tolerant network [16].

5.2 Failures in the User Application

The user application may experience three kinds of errors:

- **Abnormal termination error.** The application may terminate due to an not handled exception. These are detected by the Execution ARMOR.
- **Value-domain error.** Applications that appear to execute normally may produce incorrect values due to data corruption. These value errors are detected by the Voter ARMOR.
- **Time-domain error.** Applications that make no forward progress due to an otherwise undetected error will be detected by the timeout mechanisms in the Voter ARMOR. Section 4.2 provides a detailed description of how timeouts are implemented in Chameleon to detect application errors.

After an error in the application has been detected, the Execution ARMOR is ultimately responsible for restarting the user application (possibly from a previously saved checkpoint). The request to recover the application may come from the Execution ARMOR, the Surrogate Manager, or the Voter ARMOR, depending on how the error is detected.

5.3 Failures in the Chameleon ARMORs

To ensure the correct execution of a Chameleon ARMOR, the local daemon begins monitoring the ARMOR after the ARMOR is installed. The implemented ARMOR error detection techniques employed by the daemons can be found in Section 4. Once a failed ARMOR is detected, the daemon notifies the ARMOR's manager to initiate recovery. Managers have several recovery options available to them. For example, they may try to reinstall the ARMOR on the same node, on a different node, or on a different platform. They may also try using another ARMOR with similar functionality (more on this in Section 7).

5.3.1 FTM Failure

Every ARMOR in Chameleon has a direct manager except for the FTM. For this reason, a special ARMOR called the *Backup FTM* closely monitors the FTM for errors and recovers from any errors it detects. As introduced in Section 4.1, the Backup FTM keeps its state synchronized with the FTM so that it may assume the role of FTM at any time.

The following synchronization algorithms are implemented to keep the FTM and the Backup FTM in loose synchrony. When a new ARMOR is installed, the Surrogate Manager that installs the ARMOR sends an update message to both the FTM and the Backup FTM updating both their tables of ARMORs, their location map, and the manager relationship table. The two updates are performed using an

5. See Section 4 for implementation details as to how we are currently differentiating from a node and daemon failure.

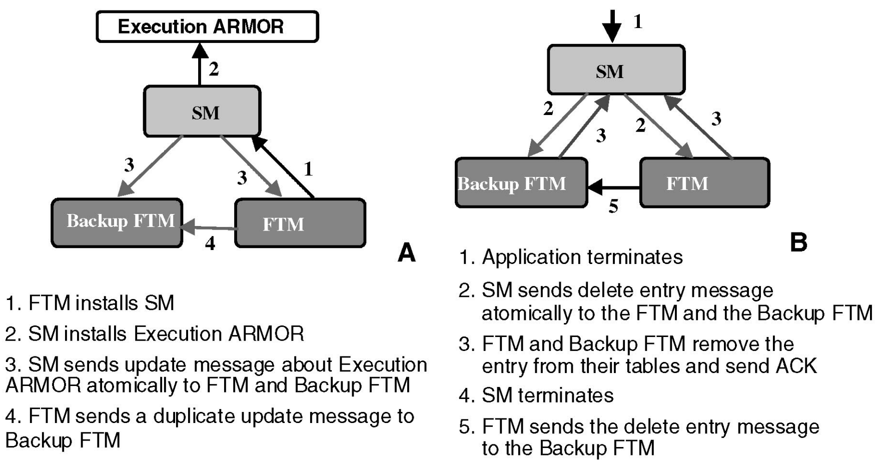

Execution ARMOR
SM
Backup FTM
FTM
1
2
3
4
A
1. FTM installs SM
2. SM installs Execution ARMOR
3. SM sends update message about Execution ARMOR atomically to FTM and Backup FTM
4. FTM sends a duplicate update message to Backup FTM
B
5
1. Application terminates
2. SM sends delete entry message atomically to the FTM and the Backup FTM
3. FTM and Backup FTM remove the entry from their tables and send ACK
4. SM terminates
5. FTM sends the delete entry message to the Backup FTM

Fig. 6. Maintaining consistency among FTM and backup FTM.

atomic commit protocol that ensures neither or both semantic (see Fig. 6a). When an application terminates, the Surrogate Manager uninstalls all its ARMORs, sends notification to the FTM and the Backup FTM to delete corresponding entries from their tables, and then terminates itself (see Fig. 6b). There are duplicate uninstall messages being sent to the Backup FTM – from the FTM and the SM.

An FTM crash failure is detected by the lack of response to the heartbeat sent out by the Backup FTM. The more challenging case occurs when the FTM fails in a Byzantine manner. We outline one such case to show how our synchronization mechanisms help to maintain consistent state. Suppose the FTM starts deleting entries from its global table of ARMOR entries. In such a case, when the SM sends the *delete entry* message to the FTM (Fig. 6b, step 2), the FTM cannot acknowledge it because the entry is lacking for the corresponding ARMOR (Fig. 6b, step 3). In that case, the Surrogate Manager flags an error in the FTM, and the Backup FTM promotes itself to FTM and notifies all subordinate managers.

5.4 Handling of Software Faults

A related issue is the problem of software faults. Again, analysis of software faults from actual systems provides evidence that remaining faults in well-tested software manifest most often as timing or synchronization problems/errors [20], [32]. It was shown, for example, that using process pairs in Tandem systems, which were originally intended for tolerating hardware faults, allows the system to tolerate about 70 percent of reported faults in the system software that caused processor failures. The major reason for this observed fault tolerance is the loose coupling between processors, which results in the backup (retry) execution (the processor state and the sequence of events) being different from the original execution [20], [14]. While process pairs may not provide perfect software fault tolerance, the implementation is relatively cheap. Chameleon takes advantage of the fact that in a replicated application execution on different machines, a software fault is unlikely to manifest in the same way in all replicas or cause the same incorrect result.

5.5 Failure Modes

Table 2 presents the primary failure modes for Chameleon environment. The table is intended to be self-explanatory and, hence, we will make only general remarks about its contents. Each failure mode is characterized by a brief description of the consequences on the environment. In addition, the table identifies the ARMOR responsible for detection of a particular failure and, finally, the table gives a detailed description of the fundamental steps in recovery from the detected error. From Table 2, one can observe that there exists a certain hierarchy in error detection and recovery. This hierarchy is illustrated in Fig. 7, which provides primary paths of error detection and recovery activities. The secondary paths, such as detection of an erratic node behavior by a Voter ARMOR, are not depicted to preserve clarity of the figure.

6 Experiments and Results

The experimental applications are run on our prototype Chameleon implementation that exists on a testbed of heterogeneous computing nodes at the Center for Reliable and High Performance Computing at the University of Illinois. The nodes comprise both workstations and PCs, and the environment can run on various flavors of UNIX—SunOS, Solaris, and HP-UX—as well as Windows NT. The nodes communicate using TCP/IP over 10 Mbps Ethernet. The prototype implementation supports the following ARMORs:

- Managers: FTM and Surrogate Managers—1) Single node execution offering baseline reliability, 2) Dual execution where the first result is accepted, 3) Dual execution where the two results are compared and only a match is accepted, 4) TMR execution, and 5) Quad execution
- Daemons: for SunOS, Solaris, HP-UX, and Windows NT,
- Common ARMORs: 1) Execution ARMOR, 2) Voter ARMOR, 3) Checkpoint ARMOR, 4) Heartbeat ARMOR, 5) Initialization ARMOR.

TABLE 2
Chameleon Failure Modes and Recovery

	Failure Mode	Consequence	Detection	Recovery
Node	Crash	All ARMORs lost on the node	Heartbeat ARMOR	• HB ARMOR notifies FTM • FTM removes node from list of registered nodes • FTM restarts any affected ARMORs it manages to a new node • FTM notifies immediate managers of the crashed node; these managers restarts any of their ARMORs and recursively notify all subordinate managers
Network	Link down	Unreachable node	Same as node crash	Same as node crash if a redundant link is not available; No actions are necessary if a redundant link is available
	Switch down	Network down	Heartbeat ARMOR	Cannot recover if a redundant switch is not available; No actions are necessary if a redundant switch is available
Application	Abnormal termination	Program fails to complete normally	Execution ARMOR	• Notify the execution ARMOR s manager • Restart the application (with assistance from a checkpoint ARMOR if enabled)
	Livelock	No forward progress made in the application	Execution ARMOR through a user-supplied timeout	• Kill application • Restart the application. • If repeated restarts result in livelock, notify execution ARMOR s manager • Manager may elect to reinstall the execution ARMOR on a node with a different platform • If installing the ARMOR on a new platform fails, the user will be notified of an apparent software bug
	Erroneous computation	Incorrect results	Voter ARMOR	• Dual mode: restart the application and notify the user; • TMR mode: mask the error (optionally notify the user)
Common ARMOR	Crash	Lost ARMOR	Daemon	Notify the crashed ARMOR s manager (recovery as described in section 5.3).
	Process alive, but unresponsive	ARMOR cannot process incoming messages	Daemon	• Kill the unresponsive ARMOR • Notify the ARMOR s manager to reinstall theARMOR
Daemon	Crash/Unresponsive	All ARMORs on the same node cannot communicate with remote ARMORs	Heartbeat ARMOR	• Notify the daemon s manager (the FTM) • Most likely, the daemon s manager will treat a daemon failure as if the entire node has crashed and recovers as for the node failure.
FTM	Crash/Unresponsive	Environment without overseeing manager	Backup FTM (designated surrogate manager)	• Backup FTM promotes itself to become the FTM • New FTM notifies all its managed ARMORs of the change; all subordinate managers recursively notify managed ARMORs • New FTM promotes a new backup FTM from one of the surrogate managers
FTM Daemon	Crash/Unresponsive	FTM unreachable	Backup FTM	Assume the FTM has crashed and recover as above

6.1 Application and Background Workload

Applications can be submitted to the environment and different applications can be executed in different FTES using the Chameleon ARMORs. The benchmark application used in our experiments is a distributed matrix multiplication that has been split a priori into two tasks. Results presented in the next section are for a run with two matrices of sizes 200*400 and 400*200; the size of the executable file for each subpart of the application is 33.5k. It employs the simple matrix multiplication algorithm,[6] distributed over two machines. Each part at the end of its computation dumps the result into a file. The results are combined at the voter and then the combinations are voted upon. The application is run in a TMR mode, which involves three independent pairs of machines, and each pair executes a replica of the distributed application. We have one Execution ARMOR on each of the six machines, monitoring the execution of each part of the distributed application. The FTM, the Surrogate Manager, and the Voter ARMOR run on a separate computation node. Since we wish to use application checkpointing and make measurements of recovery times for application failure, we are constrained to run the six copies on Solaris machines.

6. If $C_{m,n} = A_{m,z} * B_{z,n}$ then $c(i,j) = \Sigma_{k=1,\ldots,z} a(i,k) * b(k,j)$, where A and B are input matrices and C is the resulting matrix. The distribution is such that each part computes one horizontal strip of the result matrix consisting of half the rows.

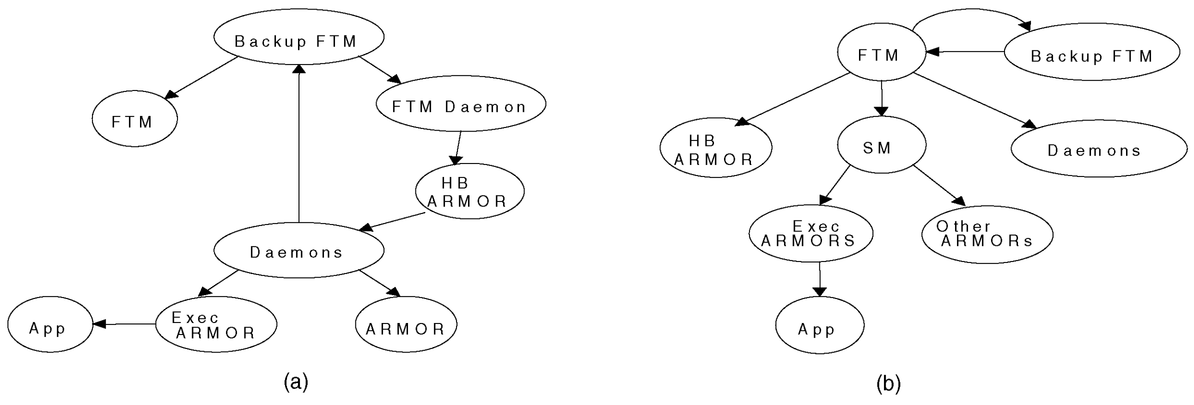


Fig. 7. (a) Error detection (source node detects errors in the sink node); (b) Error recovery (source node recovers from errors in the sink node).

The background workload for the experiment is varied from the baseline case of normal background on our network of machines to one or more copies of a computationally intensive task of a factorial calculation, which is executed in machines, participated in the Chameleon environment. We do not mandate an idle workload because we felt that the normal workload would be more representative of the workload that will be experienced in a cluster where the nodes on the cluster are not dedicated ones.

Since different FTES reuse the same common ARMORs and primitive functions, we have been able to scale our environment with respect to the configurations supported quite easily. Thus, our experience in developing the TMR configuration from the Dual configuration was quite encouraging. Also, as we scale the configurations supported and the degree of parallelization of the application, we scale up the number of machines on which we run Chameleon. In the above configuration, we are using seven machines, but if the application were to be parallelized into four tasks in place of two, our infrastructure would not need to be changed. Also, we found that the additional message passing in scaling up the number of active entities in the system is not prohibitive.

6.2 Measurements

In order to determine the effectiveness of the Chameleon environment, it is essential to demonstrate the system capability of providing the fault tolerance against different failure modes (i.e., application, hardware, and ARMOR failures) while preserving acceptable level of performance overhead. We have, therefore, conducted direct measurements in the prototype implementation of Chameleon to obtain the overhead in the application execution and recovery times for various failure scenarios. Note that the entities on which measurements are made are in an active phase of development; hence, the numbers do not reflect any of the optimization techniques (chiefly with respect to the number of hand shaking messages being exchanged) that we plan to apply to reduce the communication overhead. The measurements provided are averaged over six machines with processor speeds from 140 to 200 MHz.

6.2.1 Time to Launch Basic ARMORS

To gather a sense of the time overhead which is involved in launching different components of the Chameleon infrastructure, we compiled some measurements which are provided below.

- Installing the Execution ARMOR: 71 msec
- Setting up a node to participate in Chameleon: 2,245 msec
  - Installing the Host Daemon: 40 msec
  - Collecting information about the platform through the Initialization ARMOR: 2,205 msec

6.2.2 Overhead in the Application Execution and Recovery Times

To quantify the time overhead in the application execution, we have conducted several experiments to measure this overhead.

First, we give time to detect node (or host daemon) and application failures:

- Time for local detection of failure by trapping abnormal signals, (as is done by the *Execution ARMOR* while monitoring an application if it misbehaves): 928 ms
- Overhead of Heartbeat ARMOR implemented as ICMP pings:
  10.494 s (if node is failed; the default timeout period of 10 s dominates); 2.716 ms (if node is okay).

Second, we provide measurements of the time overhead in the application execution for a fault-free execution without checkpointing and an execution with a fault injection to the application and recovery from a checkpoint. The measurements (averaged out over five application runs) are given in Table 3 as a function of the background workload. The workload varies from zero to three additional processes per node (each process executes a factorial computation program).

Specific comments to execution times given in Table 3 are provided below:

TABLE 3
Time Overhead in Application Execution

Number of workload processes	Time for standalone execution [s]	Time for fault-free execution in Chameleon [s]; [overhead %]	Time for execution in Chameleon with fault injection and recovery [s]; [overhead %]
0	38.01	45.85 (20.6%)	60.14 (31.2%)
1	49.36	50.31 (1.9%)	69.50 (35.4%)
2	54.87	56.23 (2.5%)	94.09 (64.3%)
3	73.52	76.17 (3.6%)	95.62 (23.9%)

1. The execution times do not include the time for compiling the two parts of the application (scenario where we are executing on homogeneous nodes),
2. The fault-free execution in Chameleon encompasses the time to launch the necessary ARMORs for supporting the application (i.e., Execution ARMORs, the Surrogate Manager, and the Voter ARMOR), time to vote upon results from the application, and time to communicate the results to the surrogate manager,
3. The execution with fault injection and recovery comprises (in addition to the times described above in point 2) time to launch the checkpoint ARMOR, the time to modify the application to incorporate the appropriate checkpoint function call and the time link the Checkpoint ARMOR with the application. The execution time also involves the time spent on setting checkpoints (the overhead for each checkpointing operation is 70 ms and the frequency of checkpointing is 1.5 sec). Finally, the execution time includes the time for error detection and recovery from the last checkpoint.

For the readings presented in Table 3 we injected a single fault at a random offset from the start of the application. The observed overhead in the application execution is about 30 percent as compared to the fault-free execution in Chameleon. A higher overhead of 64 percent is measured for the application execution when two background processes are running on each node. This higher overhead is due to changes in the load on the individual nodes and in the network traffic (recall that the application is executed in the network of regular workstations, which are used by other users).

Third, we present times for recovery from failures of various components under the normal workload case (i.e., no copy of our workload process running). The times are estimated from the summation of the times for each of the suboperations and are given in Table 4.[7] Note that for the *Host Daemon* recovery, the assumption is that the node is alive (i.e., only the daemon process crashed) and a single execution ARMOR and application were on the host which needs to be restarted. The installation times for the application, the Execution ARMOR, the Voter ARMOR and the Host Daemon are averaged out over six machines.

Finally, we present some preliminary results from the execution of a *simple iterative application having three nested loops* in the duplicated mode with the FTM running on a Windows NT machine. The two machines selected for executing the application are Solaris workstations: *wolf* (Sun Ultra1-170, 64 M memory) and *monn* (Sun Ultra1-140, 64 M memory).

- Time to transmit, install and get an acknowledgment from the Execution ARMOR:
  Average (taken from six measurements) 1,462 ms (on *wolf*); 1,717 ms (on *monn*)
- Time to send the application to the Execution ARMOR and get the results back:
  Average (taken from six measurements) 1,056 ms (on *wolf*); 1,059 ms (on *monn*)

7 Reconfigurable ARMOR Architecture

Results obtained from the design and evaluation of our early Chameleon system demonstrate the effectiveness of using the same ARMORs in several different FTES—the same kind of Voter ARMOR, for example, may be used in both dual execution mode and TMR execution mode. Encouraged by these results, we wish to extend this reusability and flexibility to include not only the construction of fault-tolerant execution strategies, but also the construction of ARMORs. Taken a step further, we also wish that the ARMORs were able to dynamically change their composition at run-time, thus allowing ARMORs to adapt to changing dependability requirements. Intuitively, the following attributes should be present in a flexible and reusable architecture:

1. A well-defined concept of an *ARMOR type*. ARMORs of the same type behave similarly and are

TABLE 4
Recovery Times

Entity to be recovered	Recovery time (ms)
Execution ARMOR	2428
Host Daemon	1891
Surrogate Manager	1857
Voter	1510

7. The recovery times are estimated from the times for the constituent operations of the recovery. For example, for the common ARMOR (specifically, the Execution ARMOR), the steps are: Detection by Daemon (928 msec); Notification to SM (141 usec); Execution ARMOR restarts application from checkpoint (1.5 sec = Interval between checkpoints).

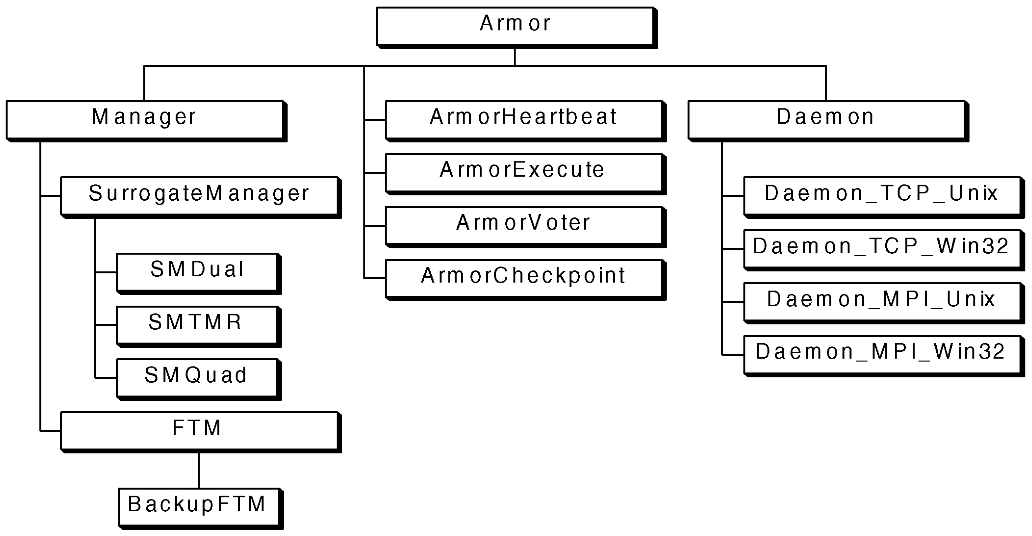


Fig. 8. ARMOR class hierarchy.

substitutable for one another. For example, the Surrogate Manager responsible for the TMR execution strategy should be able to utilize *any* ARMOR of the type "Voter ARMOR." An ARMOR of type "Voter ARMOR" can simply be defined as an ARMOR that takes in results, votes upon the results, and reports the outcome of the voting.

2. A well-defined procedure by which new ARMOR types may be derived from existing ARMOR types. Extending the behavior of an ARMOR consists of inheriting the functionality of an existing ARMOR type and then either adding to or removing from the derived functionality. Creating a new ARMOR consists of inheriting base functionality common to all ARMORs (e.g., message sending, message handling, etc.).
3. A systematic means by which newly created or extended ARMORs may be integrated into the library of existing ARMORs. Once registered in the library, the new ARMOR may be used by any FTES needing ARMORs of the new ARMOR's type.

In the succeeding sections, we will demonstrate the provisions of the Chameleon architecture that satisfy these attributes and, as a result, provide a reconfigurable and flexible infrastructure.

7.1 ARMOR Class Hierarchy

Using established object-oriented constructs, the first attribute may be realized through an *ARMOR class hierarchy*. In addition to providing for ARMOR types, the ARMOR class hierarchy in Fig. 8 partially addresses the second attribute—descendant ARMOR types inherit all of the base functionality from their ancestor classes and may define additional functionality of their own. For example, the instances `SurrogateManager` class possess all the capabilities of the `Manager` class (see Table 5 for a listing of these capabilities) and add the capability of overseeing the execution of a single application in the Chameleon environment. Equally important is the fact that instances of `SurrogateManager` may be used wherever instances of the `Manager` class are expected. Since managers are pervasive throughout the operations of the Chameleon environment, the ability of having all instances of `SurrogateManager` be seamlessly treated as managers is crucial in providing an extendible and reusable architecture. As an example of this extendibility and reusability, consider case of a daemon notifying a manager of a failed subordinate ARMOR. No code needs to be rewritten to have daemons send failure notification messages to instances of `SurrogateManager` (or any descendant of `Manager`, for that matter), and no additional code needs to be added to have instances of `SurrogateManager` recover from subordinate ARMOR failures.

Although the class hierarchy in Fig. 8 provides the concept of a well-defined ARMOR type and provides for the foundation of ARMOR extendibility, the hierarchy does not directly address *how* functionality of an ARMOR is specified or how this functionality may dynamically adapt in response to an external request. These issues will be addressed in the following two subsections.

7.2 Encapsulating Functionality in Basic Building Blocks

From the ARMOR's perspective, dynamic adaptivity can only occur if the ARMOR is able to reconfigure itself—specifically, if it is allowed to change its functionality. In order to support dynamic reconfiguration, the ARMOR's functionality must be modular and encapsulated. Then, modules with similar behavior may be substituted for one another to alter the functionality of the overall ARMOR. For example, managers may have their recovery functionality encapsulated in a `Recovery` module. To provide for different forms of recovery (e.g., forward recovery), the manager only needs to replace its existing `Recovery` module with a `Recovery` module that provides the appropriate level of functionality.

In Chameleon terminology, ARMORs are composites of modules called *basic building blocks* and basic building blocks, themselves, may be composites of other basic building blocks. By associating a set of basic building

TABLE 5
Basic Building Blocks Common to all ARMORs

Basic Building Block	Functionality
`MessageDispatch`	Dispatches incoming messages to the appropriate basic building block
`MessageSend`	Sends a message to an ARMOR
`CompositionController`	Responsible for dynamically adding and removing basic building blocks in an ARMOR

blocks with a specific ARMOR type, all descendant ARMOR types automatically inherit those basic building blocks and, hence, their functionality. For example, all descendants of `Manager` automatically contain a default `Recovery` block as described above. Descendant classes are free to override the inherited basic building blocks or add basic building blocks to augment the ARMOR's functionality.

To ensure that basic building blocks are truly substitutable components, all basic building blocks must conform to a consistent interface. Generally speaking, basic building blocks must be able to have their functionality invoked through a message. As such, each basic building block must contain a function of the following prototype:

```
BasicBuildingBlock::process_message (Message *pmsg);
```

Most importantly, the `process_message` function should be the only interface to the basic building block's functionality. Since all basic building blocks conform to the same interface—and since all basic building blocks with similar behavior should respond to the same commands—-semantically similar basic building blocks may be substituted for one another within the ARMOR framework without any changes made to the code that invokes the services of the basic building block. What constitute "semantically similar" blocks? Semantically similar blocks must minimally process the same types of messages and send the same types of messages (said another way, they must have at least the same inputs and outputs). For example, substituting the `ForwardRecovery` block for the `Recovery` block should be transparent to all `Manager` ARMORs, as long as both basic building blocks initiate recovery when invoked with the appropriate message through `process_message`.

7.3 ARMOR Composition to Support Flexible Reengineering

At the very least, all ARMORs must be composed of the basic building blocks found in Table 5. The first two basic building blocks—`MessageDispatch` and `MessageSend`—reflect the fact that the Chameleon environment is driven through message passing among ARMORs. Although ARMORs appear to be the endpoints of message communication, all messages must ultimately originate from basic building blocks and be processed by basic building blocks. Under our current specifications, all incoming messages to an ARMOR are sent to the `MessageDispatch` block. The `MessageDispatch` block must then forward the message to the appropriate basic building block by invoking the `process_message` function of the destination block. Since all message communication is done in an ARMOR-to-ARMOR manner as opposed to a block-to-block manner, nothing in the message identifies the basic building block that should receive the incoming message. As such, all basic building blocks must *subscribe* to messages they wish to receive. To do this, a basic building block sends a message to `MessageDispatch` for each type of message it wishes to receive (e.g., all `MSG_ARMOR_INSTALL` messages or all `MSG_HEARTBEAT` messages).

The counterpart to the `MessageDispatch` block is the `MessageSend` basic building block. As its name suggests, the `MessageSend` block is responsible for routing an outgoing message to its appropriate destination. Typically, if the message is being sent to another ARMOR, the message must be forwarded to the local daemon for further routing.

Finally, the `CompositionController` block is unique among all other basic building blocks—it is the only block that is not dynamically substitutable. This is done for good reason, as the `CompositionController` is responsible for carrying out the dynamic substitution of *other* basic building blocks. To effectively replace an existing block with another block during run-time, the `CompositionController` must ensure that the ARMOR is not currently using the block to be replaced (i.e., the block is not currently processing a message). This is guaranteed by mandating that all interaction with basic building blocks go through the `CompositionController`. Messages are not directly passed to a specific basic building block—instead, a request is made to the `CompositionController` to forward the message to the correct basic building block.

In summary, the flow of incoming messages through an ARMOR can be described as follows:

1. After the local daemon delivers an incoming message to an ARMOR, the ARMOR's `MessageDispatch` basic building block is notified of the incoming message.
2. The `MessageDispatch` block forwards the message to all basic building blocks that have subscribed to the message type. If no basic building blocks have subscribed to the incoming message type, the message is dropped. To perform the actual forwarding, the `MessageDispatch` basic building block notifies the `CompositionController` to deliver the message to the appropriate block through the block's `process_message` function.

Likewise, the flow of outgoing messages can be described as follows (see Fig. 9):

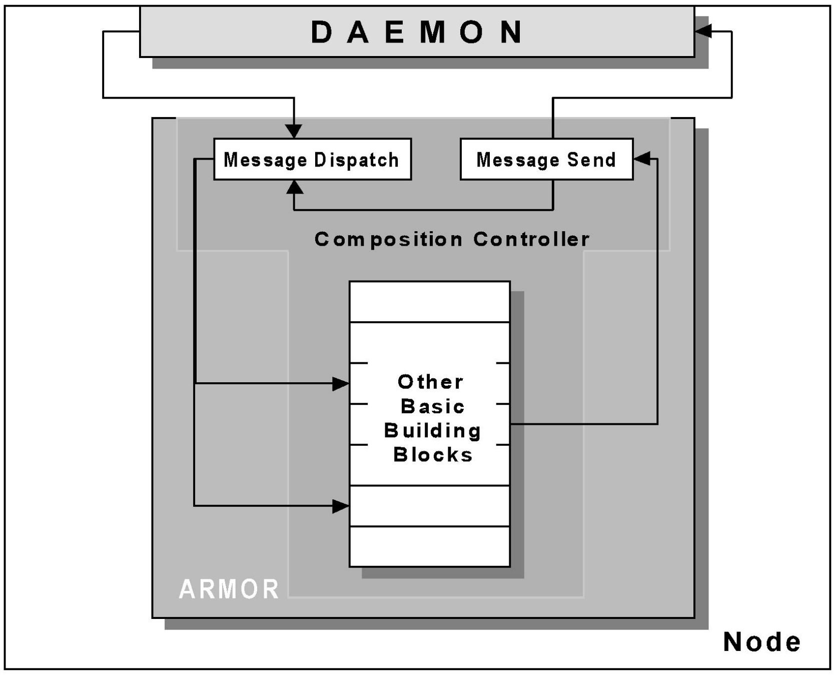


Fig. 9. Message flow among basic building blocks in an ARMOR.

1. The basic building block requests that the ARMOR send a message to a destination ARMOR through the `MessageSend` basic building block.
2. If the message is an intra-ARMOR message, then `MessageSend` simply forwards the message to `MessageDispatch`. If the message is being sent to another ARMOR, `MessageSend` forwards the message to the local host daemon for further routing.

Having highly substitutable components and a means to perform this substitution at run-time allows an ARMOR to dynamically adapt to changing dependability requirements. To initiate the configuration change, an external source (be it the user, the application, or another ARMOR) must send a message to the ARMOR's `CompositionController` telling it what basic building blocks to add or replace.

7.4 Factories

In the previous subsections, we have seen provisions that provide an extendible and reusable ARMOR infrastructure. To complete the picture, however, the third and final attribute from the section introduction must be addressed—namely, the need for integrating new and extended ARMORs and basic building blocks into the Chameleon environment. Obviously, very little has been gained if existing fault-tolerant execution strategies cannot use newly created ARMORs, and if existing ARMORs cannot use newly created basic building blocks.

After a new ARMOR or building block has been created, it needs to be registered in the Chameleon environment. Our current specifications dictate that the FTM will be responsible for registering Chameleon components and for making them available for general use. Once registered, a specific ARMOR type or basic building block type may be identified through a unique identification number.

To produce an instance of a specific ARMOR type or basic building block type, the type's identification number is passed to a *factory*. A factory's sole responsibility is to create components of a specific type. Take, for example, the situation in which a manager wishes to replace its current `Recovery` block with a `ForwardRecovery` block. To do so, the manager must only know the identification number of the `ForwardRecovery` block and pass this identification number to the basic building block factory. As expected, there are two kinds of factories—an ARMOR factory and a basic building block factory. These factories may be implemented in one of two ways:

- For the most flexible approach, the factory may use dynamic linking to create an instance of any component available in the FTM's library at run-time.
- For systems that cannot take advantage of dynamic linking, the factories can statically include only those components that are known a priori to be needed throughout the lifetime of the application.

7.5 Benefits of Hierarchical Composition

Coupled with the reusability of ARMORs in several FTES, the ARMOR architecture outlined in this section provides for a highly flexible and reconfigurable infrastructure for Chameleon. The resulting architecture contains several levels of reconfiguration granularity—FTES can be customized by selecting specific ARMORs via an ARMOR factory and specific ARMORs may be customized even further by composing them from specific basic building blocks via a basic building block factory.

Much like the COMPOSITIONCONTROLLER controls the makeup of an ARMOR in terms of its building blocks,

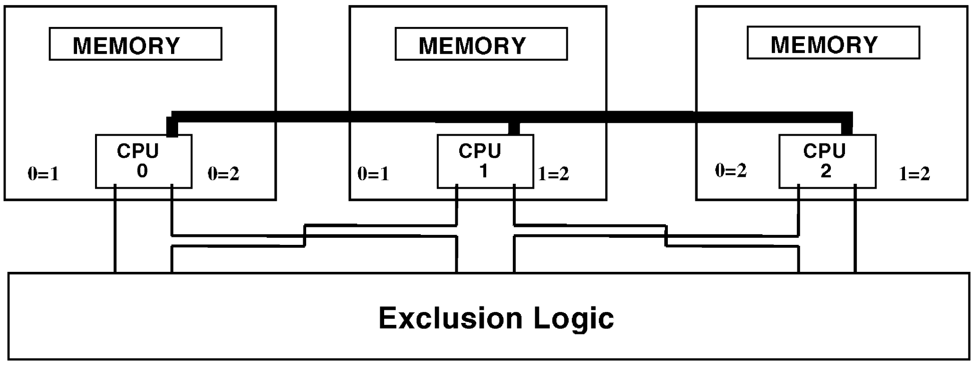


Fig. 10. Simplified architecture of the railway control system.

Surrogate Managers control the makeup of FTES by selecting the ARMORs that implement the execution strategy. Since the Chameleon architecture is completely message-driven, changing the composition of a Chameleon entity can be initiated by sending a reconfiguration message to either the COMPOSITIONCONTROLLER or surrogate manager. For instance, if the application reaches a critical section where TMR execution mode is needed, it may send a message to the surrogate manager requesting the higher level of fault tolerance.

8 Chameleon Enhancement to Handle Real Time Applications

The attribute of a key importance in real-time applications is timeliness, i.e., the ability to meet real-time deadlines (hard or soft) in producing responses. The difficulty in providing real-time guarantees to the application increases significantly if the application executes in a distributed environment in the presence of failures which affect system components (i.e., some of the system resources, e.g., computation nodes, become unavailable). To provide timeliness in such an environment, we need error detection and recovery techniques with estimatable/predictable recovery times.

An example real-time application which Chameleon is intended to support is the railway control system described and analyzed in detailed in [30]. The railway control system employs TMR with a fail-safe logic (usually implemented in hardware) to achieve a high level of dependability (see Fig. 10). The three processing units execute identical control software. The voting procedure consists of two steps: 1) distributed voting—conducted in each processing unit, and 2) centralized majority voting—conducted in the exclusion logic unit. Outputs from the first step are used as inputs for the second one. The faulty processing unit is marked as faulty and is recovered and reintegrated into the system in the next cycle.

Using the ARMOR technology real-time functions can be encapsulated in basic building blocks. These real-time basic blocks can then be used to reengineer ARMORs to add timeliness in ARMORs behavior. The following outlines the type of enhancements and provisions envisioned for supporting real-time applications

- Employ a real-time operating system (such as VxWorks from WindRiver Systems, Chorus from Sun Microsystems, or Lynx from Lynx Real Time Systems) which provides a POSIX-compliant environment and the system services (e.g., operating system hooks) to support implementation of fault tolerance. Note that all basic building blocks used to create ARMORs are designed to be POSIX compliant.
- Provide a set of basic building blocks (e.g., process creation) with predictable execution times. Observe that real-time is treated as another attribute that can be used in selecting basic building blocks (i.e., some blocks may have real-time and non-real-time versions, others may have only one design).
- Provide a Real-time Manager, an ARMOR responsible for configuring Chameleon to operate under real-time constraints. The Real-time Manager would encompass most of functions supported by FTM (e.g., keep a repository of a global information about the system resources) and would assemble the fault tolerance execution strategies using ARMORs with guaranteed real-time behavior (i.e., ARMORs composed of basic building blocks with predictable execution times).
- Preinstall ARMORs and keep them persistent throughout the system lifetime (with backups for timely switchover in case of a failure). This will allow us to avoid frequent ARMORs installing/uninstalling operations.
- Support a mechanism/algorithm for clock synchronization. This service could be implemented as a basic building block and embedded into the Daemon ARMOR on each node participated in the Chameleon environment.
- Select fault tolerance strategies for which recovery time bounds can be relatively easy estimated (at least for the worst scenarios). For example, dual and TMR execution modes can be considered as two candidate fault-tolerant schemes for real-time application execution.
- Use a dedicated network/communication subsystem for which it is possible to estimate the worst time delays in message transmission. Timeliness in message transmission in combination with real-time

basic building blocks for sending and receiving messages (`MessageDispatch` and `MessageSend` building blocks) can ensure predictable time bounds on delivery and initial processing of messages.

9 Conclusions

This paper presents Chameleon, an adaptive software infrastructure that allows different levels of availability requirements to be supported concurrently in a networked environment. Chameleon provides a highly reconfigurable ARMOR architecture. Carefully defined and implemented error detection and recovery techniques play a key role in attaining high availability for substantially off-the-shelf applications. The system is designed to recover from faults in the application, hardware, operating system, and Chameleon components themselves. Chameleon, by design, ensures that there is no single point of failure.

The following features differentiate Chameleon from other software-implemented approaches to fault tolerance:

- Construction of fault-tolerant execution strategies from a comprehensive set of ARMORs. These fault-tolerant execution strategies can then be reused by different applications without modification.
- Creation of ARMORs from a library of reusable basic building blocks and the ability to seamlessly integrate new ARMORs into existing fault-tolerant execution strategies through ARMOR factories.
- Dynamic adaptation to changing fault tolerance requirements achieved through the run-time reconfiguration of ARMORs.
- Flexibility of using the same computation nodes to concurrently execute applications with different availability requirements.
- Hierarchical error detection and recovery whereby every ARMOR and user application is overseen by another ARMOR so that failure of a single ARMOR does not compromise the dependability of the system.
- Operation in a network of heterogeneous computation nodes, including UNIX and Windows NT platforms.

The initial implementation of Chameleon demonstrates the feasibility of the proposed ARMOR-based infrastructure in a heterogeneous network. The experimental results show that overhead in application execution and recovery times are acceptable for computationally intensive applications. The first prototype provides very useful insight into further system development and implementation, e.g., it was observed that we need to have efficient and fast error detection techniques within ARMORs capable of capturing errors in application, as well in the ARMOR. It was also concluded that an API for the application programmer would be very useful for getting better controllability over the application. We are aware that much work needs to be done on extending the Chameleon, particularly in enhancing our support for distributed applications and addressing real-time issues. We demonstrated that Chameleon does address the problem of building a fault-tolerant, networked system from unreliable computation nodes. The question of determining the limits of infrastructures such as Chameleon in providing high availability and fault tolerance can best be answered through a combination of theory and experimentation.

Acknowledgments

This work was supported in part by Jet Propulsion Laboratory (JPL)—NASA under contract JPL961345 and in part by a grant from Tandem Computers (now a division of Compaq). We thank B. Horst (formerly of Tandem Computers) and our colleagues on REE team from JPL (especially R. Lee, J. Beahan, R. Some, D. Rennels, and F. Mathur) for many insightful discussions. We would like to thank J. Wang and M. Kalyanakrishnan for their contributions in developing the application and porting the environment to Windows NT.

References

[1] G. Agha and D.C. Sturman, "A Methodology for Adapting to Patterns of Faults," G. Koob, ed., *Foundation of Ultradependebility*, vol. 1. Kluwer Academic, 1994.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A Communication Sub-System for High Availability," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing (FTCS-22)*, pp. 76-84, 1992.

[3] P.A. Barrett et al., "The Delta-4 Extra Performance Architecture," *Proc. 20th Int'l Symp. Fault-Tolerant Computing (FTCS-20)*, pp. 481-488, 1990.

[4] K.P. Birman, *Building Secure and Reliable Network Applications.* Manning Publications Co., 1996.

[5] K.P. Birman, "The Process Group Approach to Reliable Distributed Computing," *Comm. ACM*, vol. 36, no. 12, pp. 37-53, 1993.

[6] K.P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, Los Alamitos, Calif.: IEEE CS Press, 1994.

[7] F. Cristian and S. Mishra, "Automatic Service Availability Management in Asynchronous Distributed Systems," *Proc. Second Int'l Workshop on Configurable Distributed Systems*, pp. 58-68, Pittsburgh, 1994.

[8] F. Cristian, "Automatic Reconfiguration in the Presence of Failures," *Software Eng. J., IEE*, pp. 53-60, Mar. 1993.

[9] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Comm. ACM*, vol. 34, no. 2, pp. 57-78, 1991.

[10] F. Cristian, B. Dancey, and J. Dehn, "Fault Tolerance in Advanced Automation System," *Proc. 20th Int'l Symp. Fault-Tolerant Computing (FTCS-20)*, pp. 6-11, 1990.

[11] M. Cukier et al., "AQUA: An Adaptive Architecture that Provides Dependable Distributed Objects," *Proc. Symp. Reliable Distributed Systems (SRDS-17)*, pp. 245-253, 1998.

[12] D. Dolev and D. Malkhi, "The Transis Approach to High Availability Cluster Communication," *Comm. ACM*, vol. 39, no. 4, pp. 64-70 1996.

[13] J.-Ch. Fabre and T. Perennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach," *IEEE Trans. Computers*, vol. 47, no. 1, pp. 78-95, Jan. 1998.

[14] J. Gray, "Why Do Computers Stop And What Can We Do About It?" *Proc. Fifth Symp. Reliability in Distributed Software and Database Systems*, pp. 3-12, 1985.

[15] J.P. Hansen and D.P. Siewiorek, "Models for Time Coalescence in Event Logs," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing (FTCS-22)*, pp. 221-227, 1992.

[16] R.W. Horst, "TNet: A Reliable System Area Network," *IEEE Micro*, pp. 37-45, Feb. 1995.

[17] Y. Huang and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing (FTCS-23)*, pp. 2-9, 1993.

[18] K.H. Kim, "ROAFTS: A Middleware Architecture for Real-time Object-oriented Adaptive Fault Tolerance Support," *Proc. High-Assurance Systems Eng. Symp.*, pp. 50-57, Washington D.C., 1998.

[19] H. Kopetz et al., "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach," *IEEE Micro*, vol. 9, no. 1, pp. 25-40, 1989.

[20] I. Lee, "Software Dependability in the Operational Phase," PhD thesis, Univ. of Illinois at Urbana-Champaign, 1994.

[21] M. Little and S. Shrivastava, "Using Application Specific Knowledge for Configuring Object Replicas," *Proc. Third Int'l Conf. Configurable Distributed Systems,* May 1996.

[22] D. McCue and M. Little, "Computing Replica Placement in Distributed Systems," *Proc. Second IEEE Workshop Replicated Data,* Nov. 1992.

[23] S. Maffeis, "Piranha: A CORBA Tool for High Availability," *Computer,* vol. 30, no. 4, pp. 59-66, 1997.

[24] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos, "Totem: A Fault- Tolerant Multicast Group Communication System," *Comm. ACM,* vol. 39, no. 4, pp. 54-63, 1996.

[25] Object Management Group, *The Common Object Request Broker: Architecture and Specification (CORBA),* Revision 2.0. Inc. Publications, 1995.

[26] D. Powell, "Lessons Learned from Delta-4," *IEEE Micro,* vol. 14, no. 4, pp. 36-47, 1994.

[27] *Delta-4: A Generic Architecture for Dependable Distributed Computing,* vol. 1, D. Powell, ed., ESPRIT Research Reports. Springer-Verlag, 1991.

[28] M.K. Reiter, "Distributing Trust with the Rampart Toolkit," *Comm. ACM,* vol. 36, no. 12, pp. 71-74, 1993.

[29] R. van Renesse, K.P. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Comm. ACM,* vol. 39, no. 4, pp. 76-83, 1996.

[30] L. Romano, Z. Kalbarczyk, R. Iyer, A. Mazzeo, and N. Mazzocca, "Behavior of the Computer Based Interlocking System under Transient Hardware Faults," *Proc. Pacific Rim Int'l Symp. Fault Tolerant Systems,* pp. 174-179, Taiwan, 1997.

[31] S.K. Shrivastava, G.N. Dixson, and G.D. Parrington, "An Overview of the Arjuna Distributed Programming System," *IEEE Software,* pp. 66-73, Jan. 1991.

[32] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability—A Study of Field Failures in Operating Systems," *Proc. 21st Int'l Symp. Fault-Tolerant Computing (FTCS-21),* pp. 2-9, 1991.

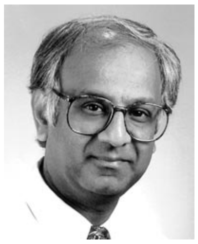
[33] *Sun RAS Solutions for Mission-Critical Computing.* White Paper, http://www.sun.com/cluster/wp-ras/, Oct. 1997.

[34] J.H. Wensley, "SIFT Software Implemented Fault Tolerance," *Proc. Fall Joint Computer Conf., AFIPS,* vol. 41, pp. 243-253, 1972.

[35] K. Whisnant, S. Bagchi, B. Srinivasan, Z. Kalbarczyk, and R.K. Iyer, "Incorporating Reconfigurability, Error Detection and Recovery into the Chameleon ARMOR Architecture," technical report, Univ. of Illinois at Urbana-Champaign, 1998.

[36] "Wolfpack,"Microsoft Clustering Architecture. White Paper, http://www.microsoft.com/ntserver/info/wolfpack.htm, May 1997.

Zbigniew T. Kalabarczyk holds the PhD in computer science from the Technical University of Sofia, Bulgaria. After receiving his doctorate, he worked as an assistant professor in the Laboratory for Dependable Computing at Chalmers University of Technology at Gothenburg, Sweden. Currently, he is the principal research scientist at the Center for Reliable and High-Performance Computing in the Coordinated Sciences Laboratory of the University of Illinois at Urbana-Champaign. He is a project leader of a DARPA-sponsored research project that is developing a framework for design and validation of highly dependable systems. Dr. Kalbarczyk's research interests are in the area of automated design, implementation, and evaluation of dependable computing systems. He is a member of the IEEE and the IEEE Computer Society.



Ravishankar K. Iyer is the George and Ann Fisher Distinguished Professor of Electrical and Computer Engineering and holds appointments in the Department of Computer Science and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. He is also co-director of the Center for Reliable and High-Performance Computing. Professor Iyer's research interests include reliable computing, measurement and evaluation, and automated design. He was general chair of the 19th Annual IEEE International Symposium on Fault-Tolerant Computing (FTCS-19) and program co-chair for FTCS-25, the Silver Jubilee Symposium.

Professor Iyer is an IEEE Computer Society distinguished visitor, and associate fellow of the American Institute for Aeronautics and Astronautics (AIAA), a fellow of the IEEE, and a member of the ACM, Sigma Xi, and the IFIP technical committee (WG 10.4) on fault-tolerant computing. In 1991, he received the Senior Humboldt Foundation Award for excellence in research and teaching. In 1993, he received the AIAA Information Systems Award and Medal for "fundamental and pioneering contributions towards the design, evaluation, and validation of dependable aerospace computing systems."

Saurabh Bagchi received the BTech degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1996, and the MS degree in computer science from the University of Illinois in 1998. He is currently pursuing a PhD in the DEPEND research group of Prof. Ravishankar K. Iyer. His research interests include software-based fault tolerance and distributed systems. He is an IEEE student member. He has worked as a summer intern at Tandem Research and with the IBM T.J. Watson Research Laboratory.

Keith Whisnant received his BS in computer engineering from the University of Illinois in 1997. He is now in his second year of graduate school and he has recently been awarded the NASA/JPL fellowship to pursue his PhD at the Center for Reliable and High-Performance Computing. His research work focuses on designing reconfigurable software architectures for reliable networked systems.