CHAMELEON : A SOFTWARE INFRASTRUCTURE FOR ADAPTIVE FAULT TOLERANCE IN DISTRIBUTED SYSTEMS

BY

SAURABH BAGCHI

B.Tech.(Hons.), Indian Institute of Technology, Kharagpur, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

ACKNOWLEDGMENTS

I would firstly like to thank my family - my parents and my brother - without whose constant support, albeit long-distance, I would not have had the strength to weather graduate school.

I am grateful to my research adviser, Professor Ravi Iyer, for starting me on such an exciting research project, and giving me the right mix of freedom and guidance during the course of the project.

Chameleon is a joint design and development effort in which several members of our research group have contributed actively. Design ideas for the project were generated with Zbigniew Kalbarczyk, Erik Haakenson, Keith Whisnant, Jun Wang, Xiao Yuxiao, Mahesh Kalyanakrishnan and Mohammed Kaaniche. Erik and Keith provided valuable insight into the implementation in the early days with their simulation environment for Chameleon. Jun provided the checkpointing library support. Xiao integrated the simulation model of Chameleon with the detailed simulation model of a Myrinet switching network. Mahesh was responsible for the porting effort to Windows NT. Mohammed provided valuable suggestions during several discussion sessions. And last, but hardly the least, Zbigniew steered us splendidly through the thorny days, helped us design the system ground-up, and was a stabilizing presence during the especially chaotic days.

The project has benefited through some demonstrations and presentations given to people from the computer industry, and the discussions they generated. Of them, mention must be made of Pankaj Mehra of Tandem Labs, Roger Lee, Robert Ferraro and Jagdish Patel of the Jet Propulsion Laboratory, Larry Jack and Chet Markiewicz of Honeywell Inc., and Haim Levendel of Motorola.

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION 1

2 RELATED WORK 3

3 CHAMELEON OVERVIEW 14
3.1 Behavioral Overview 15
3.1.1 Initialization of the Chameleon Environment 15
3.1.2 Interpreting User-Specified Dependability Requirements 17
3.1.3 Invoking a Fault-Tolerant Execution Strategy 19
3.2 Functional Overview 20
3.2.1 Managers 20
3.2.2 Common ARMORs 22
3.2.3 Daemons 24
3.3 Structural Overview 25
3.3.1 ARMOR Customization 27
3.3.2 ARMOR Reuse 30

4 ERROR DETECTION AND RECOVERY 32
4.1 Failures in Participating Nodes 32
4.2 Failure in User Application 33
4.3 Failures in Chameleon ARMORs 34
4.4 Multiple Error Detection Paths 35
4.5 Failure Modes 35

5 IMPLEMENTATION OF CHAMELEON 38
5.1 Communication 39
5.2 Applications 41
5.3 Implementation of the ARMORs 44
5.3.1 Fault Tolerance Manager 44
5.3.2 Surrogate Managers 47
5.3.3 Common ARMORs 48
5.3.4 The Base ARMOR Class 53
5.4 Host Daemon 53

6 EXPERIMENTAL RESULTS 56
6.1 Application and Background Workload 57
6.1.1 Concurrent Application Execution 59
6.1.2 User Specification 59
6.1.3 Graphical Interface 60
6.2 Measurements 61

6.2.1 Time to Launch Basic ARMORs 62
6.2.2 Overhead in Application Execution and Recovery Times 63

7 CONCLUSIONS AND FUTURE WORK 67

REFERENCES 69

LIST OF TABLES

Table Page

5.1 Experimental Machine Configurations 38
5.2 Examples of Basic ARMORs 53

6.1 Time to Lauch Some Basic ARMORs 63
6.2 Time Overhead in Application Execution 64
6.3 Recovery Times 66

LIST OF FIGURES

Figure Page

2.1 Problem of GMS to GMS communication 11

3.1 Reliable Heterogeneous Computation and Communication 16
3.2 ARMOR Class Hierarchy 26
3.3 Hierarchy of Libraries 29
3.4 ARMOR Factory Skeleton 31

4.1 Chameleon Failure Modes and Recovery 36
4.2 (A) Error Detection (source node detects errors in sink node); (B) Error recovery (source node recovers from errors in sink node) 37

5.1 ARMOR Communication Mechanism 40
5.2 (a) A Chameleon Cell for Supporting Distributed Applications (b) Use of Chameleon Cell for Replicated Database Application 43
5.3 FTM: Flow chart of activities for a particular application 45
5.4 Pseudo code for the Dual Execution mode Surrogate Manager 48
5.5 Flowchart for the Execution ARMOR 50
5.6 Interaction of Checkpoint ARMOR, Execution ARMOR and Application 52
5.7 ARMOR Communication in Chameleon 55

6.1 Execution Configuration 58
6.2 Graphical Interface Snapshot 62

CHAPTER 1

INTRODUCTION

Distributed computing systems are increasingly playing a vital role in industry and society. Hence it comes as quite a surprise that the distributed systems generally suffer from a lack of reliability features. Since mission-critical and even life-critical applications are being supported by distributed systems nowadays, their reliability needs become critical. One might expect the reliability of a distributed system to correspond directly to the reliability of its constituents, but this is seldom the case. Since a distributed system, by its very nature, comprises several components, possibly resident at physically separate locations, coordinating to accomplish a task, mechanisms used to structure these various components together are as important as the reliability of the individual components.

Traditionally, fault tolerance for the individual components has been provided by running them on dedicated highly robust hardware platforms. To "glue" together the unreliable components, software based solutions have been employed. The problem with the hardware-based solutions has been that they essentially provide static levels of fault tolerance and can serve as the platform of choice for only a fixed set of applications. The software solutions require that the applications be written explicitly to take advantage of the underlying software infrastructure.

In contemporary networked systems, a broad range of commercial and scientific applications must coexist, with potentially varying reliability requirements. It is clearly not cost-effective to provide fault-tolerant hardware platforms tailor-made for each such application. At the same

time the traditional message- passing based software implemented fault-tolerance schemes require that the application source code be available and it be rewritten to make use of the provided layer. The pressing concern then becomes how to support off-the-shelf applications with widely varying reliability requirements in an environment consisting of off-the-shelf heterogeneous computing nodes connected by an interconnection network.

Chameleon is an attempt at addressing this issue. It is an adaptive infrastructure that allows different levels of availability to be supported in a single networked environment. Such an infrastructure can be expected to support emerging applications without substantial redesign of either the computing environment or of the applications themselves. Chameleon provides dependability through *mobile ARMORs (***A***daptive* **R***econfigurable* **M***obile* **O***bjects for* **R***eliability)* - entities that can migrate between the nodes in the environment, are intelligent enough to execute specific fault-tolerance strategies, and are made robust to handle various fault scenarios. We propose a hierarchy of ARMORs to coordinate the execution of applications in the Chameleon environment. We propose schemes to manage the communication between the ARMORs and the exchange of stimulus and information among them so as to have speedy detection and recovery. In order for the environment to be scalable, the ARMOR architecture is made open and extensible and provide hooks to facilitate incremental development of the ARMOR library. Approaches to re-engineering of existing ARMORs to create new ARMORs and their incorporation into the established base are investigated.

This thesis presents a discussion of the design of the Chameleon environment and then provides details about the initial prototype implementation that is currently deployed and is operational at the Center for Reliable and High Performance Computing. An experimental study on the prototype implementation is outlined and results provided under various loads and injection scenarios.

CHAPTER 2

RELATED WORK

A number of systems have been built for offering fault-tolerant services through software mechanisms. We outline some of the salient points of some of these systems. This is followed by a discussion of why we thought that none of these systems completely fulfilled our design goals.

SIFT - **S**oftware **I**mplemented **F**ault **T**olerance - [1] was one of the earliest systems to propose a completely software-based approach to fault tolerance through loose synchronization of processors and memory. The system used replication of processor-memory modules, with replicated inter-module busses. The multiple results are voted upon on completion of the tasks. This being an early system did not address issues of supporting distributed applications. The recovery techniques were also limited in scope. It did however mention the possibility of varying the level of reliability by varying the degree of replication. However it is not clear how much of this feature was incorporated in the final system. Also the system never reached the level of maturity where it could be deployed on computers as we know of them today and tested with a range of commercial workloads.

Delta-4 [2] was a highly successful project in Europe from 1986 to 1992 that addressed distributed software-implemented fault-tolerance. Off-the-shelf hardware nodes constituted the units of hardware redundancy and could communicate through a communication system built around a LAN. In order to increase the coverage of the fail-silence assumption on the individual

nodes, a trusted hardware module called the Network Adapter Card (NAC) was added to each host. A layer of system software was designed to run on top of the hosts' operating system that provided the run-time environment for heterogeneous distributed applications. A multi-point communication protocol stack, called AMp, was used to handle the distribution aspects of the system. The use of the NAC, and the specialized communication protocol stack, somewhat limited the applicability of the approach.The issue of validation of the components that implemented the multicast communication protocols was not addressed. It was not clear how applications with varying needs would coexist on the nodes of a Delta-4 system and how the system software would handle changing reliability QoS needs.

The theory of fault tolerance in networked systems is rich and varied. Cristian [3] provides a set of basic concepts that can be used to explain the architecture of fault-tolerant distributed systems. He also looks at the group membership problem and provides elegant theoretical formulations for the problem in synchronous and asynchronous systems [4]. The fault-tolerant community has also benefited significantly from the traditional work on distributed systems to maintain consistency across cooperating server replicas. These works include examination of the concepts of logical time [5], causal ordering in message delivery [6], consensus protocols [7]. What seems to be lacking is a systematic deployment of the theoretical techniques in real-world scenarios. It was not very clear how the various mechanisms would fit together to give a holistic view of a system capable of supporting off- the-shelf software on off-the-shelf hardware and negotiate reliability contracts with the applications. For example, interaction of the actual detection mechanisms (like disagreement among replicas) with the recovery mechanisms (like roll-back or roll-forward recovery) is not borne out in the literature.

Group Communication is an important and significant step taken in the direction of construction of reliable distributed software. The thesis behind the body of work is that the process of development of reliable distributed software can be significantly simplified using *process groups* and *group programming tools*. The work on Group Communication, starting with the V System around 1988, has generated a lot of interest in the community. There is vigorous research effort in the field at several places. We make a mention of a few of them here.

Rampart [8], being developed at AT&T Bell Labs, is a distributed system using the *virtual synchrony* execution model. [1] It attempts to provide security guarantees in the face of Byzantine failures of group members and is costly performance-wise. Totem [9] 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. Transis [10] looked at network partitioning environments formally for the first time and contributed several protocols for the environment. It has also refined the virtual synchrony execution model substantially. Current work is looking at multiplexing several applications onto a single but application-customized protocol stack. The V System [11] was arguably the first system that made use of the concept of process groups. It was a micro-kernel meant specifically for distributed environments. While the system did not bother itself with providing reliability guarantees, it used process groups for service replication at multiple locations in a distributed setting. Reliability in the system was purely on a best-effort basis, without providing any real guarantees.

[1] The model supports *view synchrony*, i.e. message reception is atomic with respect to group view changes, and *gap freedom*, i.e. the guarantee that if message m_i should be delivered before message m_j and some process receives m_j and remains operational, m_i will also be delivered to its remaining destinations. The synchrony is virtual because the ordering properties are weakened from the synchronous model in ways that do not affect the correctness of the algorithm.

There has been a volume of work on Group Communication protocols at the Cornell University. A line of groupware communication tools have come out of there from Ken Birman's research group - Isis, Horus and finally Ensemble [12, 13]. Each provides tools for working with process groups, with increasing sophistications like stacks of microprotocols that are stackable at run-time. Using the tools, a programmer may construct group-based software that provides reliability through explicit replication of code and data. The emphasis is on providing the underlying communication support for the replicas in the model. These systems are therefore not connected with fault tolerance *per se*. Indeed, as Birman notes [12] "fault tolerance is something of a side-effect of the replication approach." Building reliable systems using the primitives provided by the Group Communication libraries raises several issues which are under active investigation. While powerful communication primitives, like atomic multicast, can significantly ease the task of doing replication management, there are important issues of fault detection and recovery. For example, what are the recovery mechanisms that are to be set in motion, once a replica crash is detected. Moreover, supporting off-the- shelf applications was not a goal of the effort. This means that a vast majority of existing applications will not easily be able to benefit from the tools. The techniques embodied in Isis and its successors, were able to provide relatively high coverage since its fail silence assumptions were met most of the time [14]. However, our preliminary fault injection experiments [15] using applications and protocol stack provided by the Ensemble distribution indicated that when some of the group members are malicious, or members behave in ways other than fail-silent, we could make the entire system crash.

Piranha [16] is a tool to increase availability of CORBA applications. It uses the Electra ORB (Object Request Broker) that is built on top of Horus and Isis. Piranha also relies

primarily on replication for fault tolerance. The detection mechanism is primarily through timeout and it does not handle non-fail-silent failure semantic.

Approaches to providing fault-tolerance without using the Group Communication protocol stack can be found in Microsft's WolfPack clustering architecture [17], the Ultra Enterprise Cluster from Sun [18] and the software implemented fault tolerance work at Lucent Bell Labs [19]. Wolfpack provides a solution to manage clusters of Windows NT Servers in a transparent way to increase availability and scalability. It provides a cluster service (similar to any other NT service) to handle all the tasks of cluster operation on a cluster system. The Node Manager maintains a view of cluster membership. There are dedicated entities to maintain consistent database of machine configuration and resource status. Nodes run specified algorithms for joining a cluster, leaving a cluster, etc. The approach, naturally, does not have interoperability or heterogeneity as its design goals. Also the detection technique is simply heartbeating. The applications need to be "cluster aware". The failover mechanisms are as yet not transparent to the user. At Sun Microsystems, work has been done on the 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 set of two computing nodes to monitor each other and redirect data requests in the case of software and hardware failure. Work at Lucent Labs has focused on increasing availability and data consistency of applications through the application's use of reusable components for automatic detection and recovery of failed processes, and also for preserving data consistency. However, the holy grail of being able to support substantially off-the-shelf applications is no longer met. Also, the details of the Watchdog component are not very clear. It seems like it uses a simple heartbeat mechanism for detection. Also, the desirable

design philosophy of having each entity be monitored by another entity is not met. [19] says that the WatchDog watches itself, but the mechanism in which this seemingly non-trivial task is performed, is left unspecified. The interface between the user application and the Libft library [2] is also not clear. For example, in a distributed program, how is a globally consistent checkpoint taken.

There has been considerable interest in the relatively recent area of using reflection and metalevel architecture for fault-tolerant distributed systems. In such work (notably the FRIENDS work from LAAS [20], and the Actor-based MAUD work at the University of Illinois [21]), base application objects interact with their metaobject counterparts for providing various services like fault-tolerance and security. The metalevel approach has given rich theoretical formulations of interactions of objects and metaobjects through metalevel protocols. However, for our purposes, we would like to move away from some of the formalisms and concentrate on the deployment of our fault-tolerance entities in realworld distributed systems and to investigate the effects of their interactions. The performance measurements that are provided in the related literature - both the FRIENDS work and the MAUD work - deal more with the costs of metaobject installation and metalevel protocols and less with the overall system cost under various fault scenarios. The reflective architecture does not take away from the fact that several basic services must still be implemented at the system level, like error detection (to ensure high coverage of the failure mode assumptions).

Having looked at some of the specific approaches to distributed software fault tolerance, we can step back and analyze some of the more general issues that we feel need to be addressed in the field. It is not as if the community is unaware of these issues. In fact, they have been

[2] *Libft* is a user-level C library that can be used in application programs to specify and checkpoint critical data.

raised by several researchers in the field at different places. For example, the problem of making progress in different partitions of a group in the event of a network partition was handled by the work on Totem. However, for the purpose of completeness and for providing a concise discussion of the state-of-the-art in fault tolerant distributed systems, these issues are presented here.

We focus on the Group Communication approach because it presents an attractive approach to distributed computing and the body of work has reached a certain level of maturity. The main contribution of the Group Communication body of work seems to be in the field of communication protocols. To build a truly fault-tolerant environment out of these protocols still requires substantial effort. Powerful multicast primitives ease the task of replica management, and takes the headache of tricky synchronizations away from the application developer. However, the integral issues of fault-tolerance - error detection and recovery - must still be dealt with. The goal is to support these fault tolerance services in a transparent, flexible and adaptive manner. A critical issue in middleware for robust systems is to validate the components of the middleware itself. An added replica manager means an added point for the fault to hit. Thus, the implicit assumption of fail-silence of Group Communication (GC) components seems like an obvious weakness. The protocols are silent on the implications of violation of the assumption, nor is there an easy way to assess the coverage of this assumption.

Let us consider a situation where the assumption of a basic transaction of a GC system can be violated. GC systems use two phase commit protocols for guaranteeing all or none semantics. Suppose that the data to be committed has arrived and is sitting at a buffer at a particular node and the node has replied to the initiator of the transaction in the first phase of the protocol. Now before the initiator's command to the node to commit the data arrives at the node, the data in the buffer is hit by a transient and undergoes bit flip. In that case we

will be committing a wrong value and the integrity of the transaction as a whole is violated. More generally, one has to be extremely cautious about exposed time windows where a fault occurring could violate the integrity of a system.

Again GC systems use the Group Membership service (GMS) to keep a consistent view of group membership across all the members of the group. If it so happens that the GMS integrity is compromised, then different processes may have different views of the group. This may lead to anomalous situations with the group lacking any group leader, for example. If in an attempt to make the GMS more fault-tolerant, we replicate the server hosting the service, further complexity is added to handle the replication management. Generally speaking, a system's robustness is increased if we have managerial entities that are active (and thus, in the critical path) for short time periods, and are under careful scrutiny for the period of activity.

A group member sending inconsistent replies to other members can jeopardize the system reliability. One fundamental problem is that for purposes of fault containment, the communication and detection of failure should be synchronized. This is not the case in most existing systems. The general need is to have fault containment within specified boundaries of the system, say a node in a network of workstations and PCs.

Consider an extra level of complexity that wil have to be incurred in the standard case of three replicas and three voters (fig. 2.1). Logically, the three replicas will form a group, and the three voters will form another group. Now each voter expects three inputs. If one replica fails (R3 in figure), then the GMS in the group G_1 will detect it. But for efficiency reasons (so that the voters don't wait forever, or till timeout occurs), we would like to propagate this information to the GMS of group G_2 also. Thus, we have a case here where one of the GMS' failure detection has to be propagated to another GMS. This points to the more encompassing

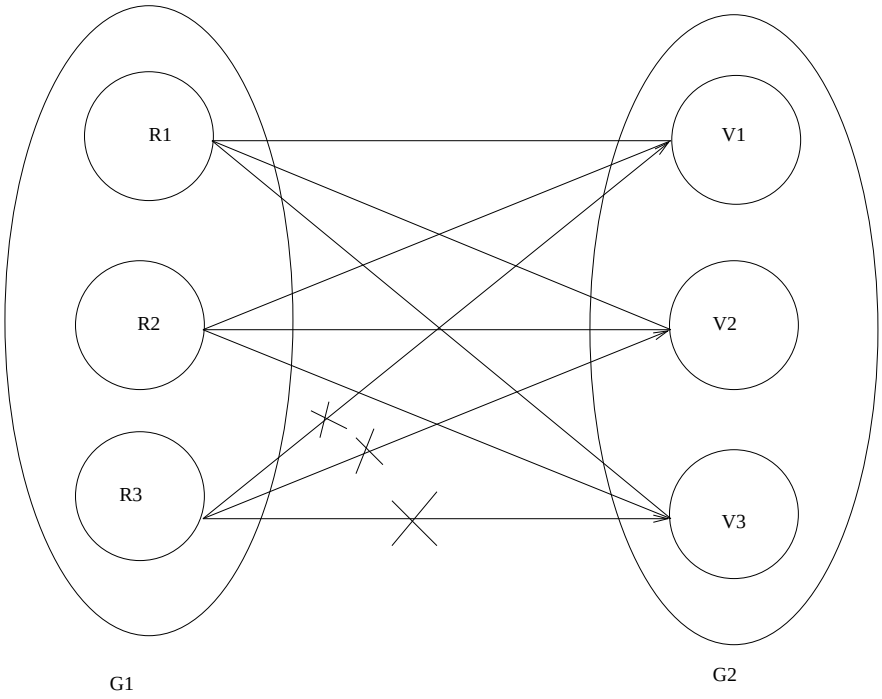

Figure 2.1 Problem of GMS to GMS communication

issue of how the entities are to be grouped together for purposes of providing a coherent robust system.

The replicas that are considered in GC systems are primarily stateless. Because of this, they can be brought back up at a different node (potentially) afresh without causing any inconsistency. This is a valid model for services that exist forever, accept client queries/requests, services them, and then goes back to wait for the next client. However consider an application that computes matrix inversion on some sensor data. Consider one of the replicas crashed, and another was brought back up in its place. Then mechanisms will have to be built in to either recreate the state the earlier replica was in, or to forward the input to the new replica.

The primary mechanism for failure detection is through timeout. This does not differentiate between failure of a process and that of a node. Thus it is possible that the entity in charge of sending the *i_am_alive* messages is dead, but the node is not. Existing systems will take the conservative view that the node is lost, and will try to move over all the entities of the node to some other node. This may be an overkill and can involve substantially inflated costs.

We sum up this section by outlining some of the fundamental issues that future efforts to providing fault-tolerance will need to look at. The systems of today don't strive to support off-the-shelf, or even substantially off-the-shelf, applications. This is of course a very challenging problem, and the solution to which could reap rich dividends. Dynamic adaptation to changing fault scenarios is another outstanding issue. This involves changing the fault-tolerance techniques either to handle changing environment, or changing application needs. The level of fault-tolerance desired may change from one application to another, and this needs to be supported in a single environment. Also, the reliability level may change within one application. It is not unreasonable for parts of the application to be deemed critical, and some other parts less so. Then, the system will need to provide differing reliability levels within one single application. It is important to have the system be customizable. This would be beneficial in allowing the system to operate in different environments. Thus, a system which inherently operates in a non-real-time setting may need to be moved over to a real-time environment. A desirable property of the system will be support for heterogeneity. This involves being able to execute on different hardware platforms as well as on different operating systems. To be cost-effective, the platforms need to be substantially off-the-shelf. Allied with the adaptivity needs is the need for scalability of the system. The system should be able to scale well with respect to the number of nodes supported. Then nodes can be integrated in the environment, for reasons of performance

or reliability, without too much effort. Scalability in another dimension is scaling with respect to the fault-tolerance techniques incorporated. An example of this, would be going from the TMR mode to the Quad mode of execution by modification of some of the entities that are in charge of providing the fault-tolerance services.

This thesis presents Chameleon which is a potential framework for addressing the issues raised above and an early testbed design for performing experiments to investigate the interaction of applications and the system entities.

CHAPTER 3

CHAMELEON OVERVIEW

Chameleon is a network-based infrastructure that has the capability of adapting to application-specified availability requirements. The basis for implementing the different fault tolerance strategies lies in the specialized entities called ARMORs - Adaptive Reconfigurable Mobile Objects for Reliability. We have developed three broad classes of ARMORs [1] :

- **Managers.** Managers oversee other ARMORs and recover from failures in other ARMORs. They are responsible for setting up the environment in the appropriate configuration to run the application and for coordinating the efforts of low-level or common ARMORs (see below). Managers are of two broad categories: The Fault Tolerance Manager, the highest- ranking manager that accepts the user application and specification, and coordinates activities till the result is communicated back to the user; The Surrogate Managers, which are selected and deployed by the FTM to run the application in its specific mode.

- **Daemons.** Daemons allow Chameleon to access a node in the network, provide ARMOR error detection, and provide the means through which ARMORs may communicate amongst themselves.

[1] Notice that there is distinction between ARMORs and Common ARMORs. Common ARMORs are a subclass of the ARMORs which is the generic term used for a Chameleon entity.

- **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.

The above entities are available in the form of several libraries. At the base, there is the library of the basic-building blocks. These contain the routines to implement basic functionality in a distributed environment like send a value to a particular component, query a location service for the location of its manager, etc. These routines are utilized in creating the common ARMORs. Thus we have a library of common ARMORs which may be invoked by the surrogate managers. The surrogate manager library consists of ARMORs responsible for application execution in a specific configuration like TMR. The library of surrogate managers consists of managers for supporting various levels of reliablity requirements of the applications. Examples would include the Dual execution mode, the TMR mode, and the Quad mode.

A high-level schematic of the environment is presented in fig. 3.1

3.1 Behavioral Overview

In this section, we discuss the steps encountered during the execution of an application in the Chameleon environment. Here we focus on the path followed for an error-free run. The path followed when there are errors in one or more of the components is discussed in Chapter 4 .

3.1.1 Initialization of the Chameleon Environment

Essentially any network of unreliable nodes may be configured to participate in the Chameleon environment. A system administrator (or some user with comparable network privileges) man-

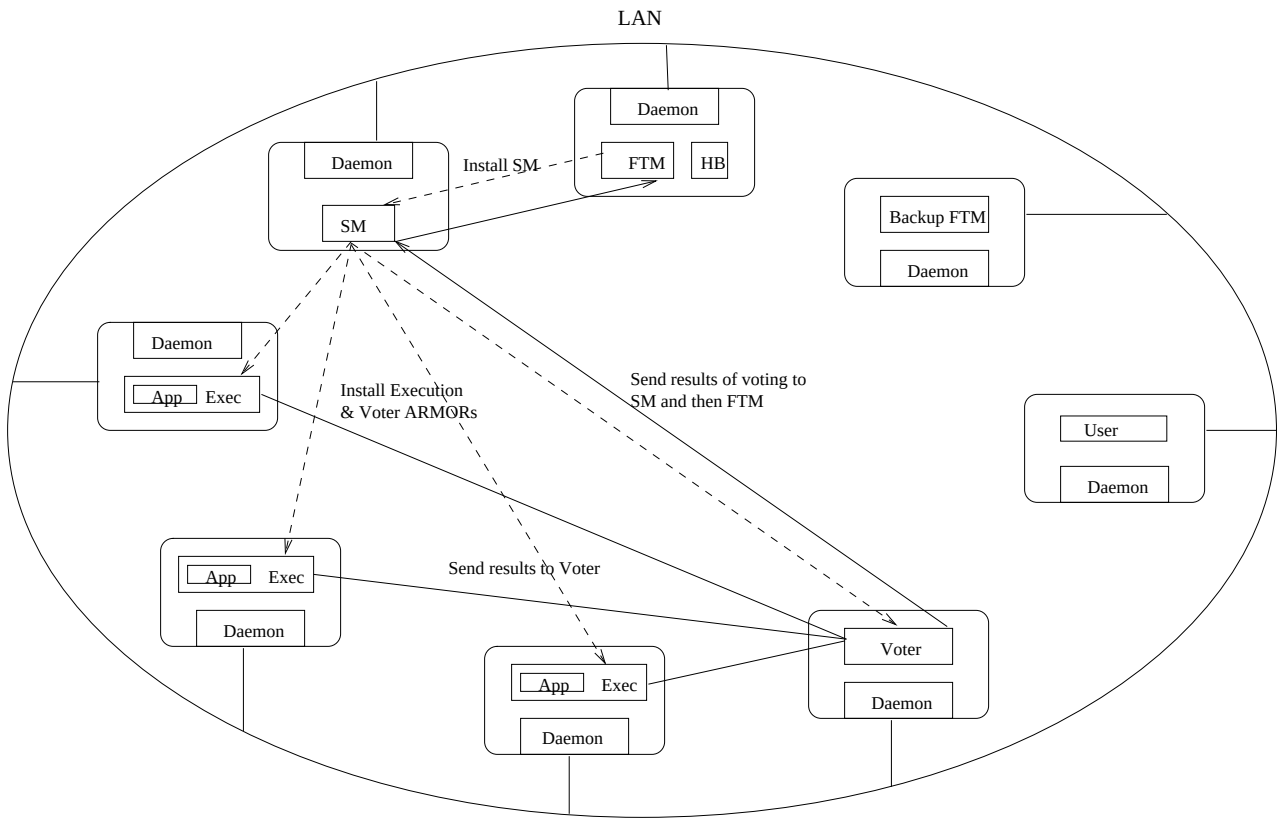

Figure 3.1 Reliable Heterogeneous Computation and Communication

ually installs a program known as the Fault Tolerance Manager (FTM) on an arbitrary node. The FTM oversees the Chameleon environment by executing as a background process that handles user requests to run an application and that configures additional nodes to participate in the Chameleon environment. Successfully installed FTM, invokes a host daemon (to handle communications with remote hosts) and a heartbeat ARMOR (to detect failures of remote nodes) on a local computation node. Finally the FTM creates a backup FTM which closely monitors the FTM and, upon detecting an error, promotes itself to become the new FTM. Because the backup FTM's primary responsibility is to monitor the FTM, rapid error recovery is possible in the event of a failed FTM. Once the backup FTM is set up we have a stable

Chameleon environment ready to accept and serve user requests. Other nodes in the network may 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 host daemon on the node wishing to join Chameleon . Daemons constitute an important part of the Chameleon architecture, as all communication between ARMORs occurs through the daemons. The node, therefore, becomes a fully-functioning member of the Chameleon environment after it has a daemon installed.

3.1.2 Interpreting User-Specified Dependability Requirements

The Chameleon environment allows the user to run several different applications in a fault-tolerant manner at the same time. Each application can have different availability and reliability requirements. From the user's standpoint, he submits an application to the FTM with availability requirements and possibly, hints for the efficient utlization of fault tolerance techniques, specified in a semantic language that the FTM understands. The availability requirements can be in the form of a certain percentage figure which would be mapped to an available strategy by a mapping available at the FTM. It could also be explicitly specified by the user, say he wants the application to be executed in the quad mode. The FTM uses the user input to select an appropriate execution strategy through which the fault-tolerant requirements can be met. To accomplish this selection, the FTM has a registry of several different execution strategies in the form of a library of Surrgate Managers. There is a tradeoff to be made in the complexity of the specification language. A very complex language may suffer from difficulty of usage by the user, and a correspondingly sophisticated parser or interpreter at the FTM which would add to the overhead of the environment. On the other hand, too simple a semantic may not

afford the user the level of control he might wish to have on the execution of his application. A case in point here is whether the language should allow user specification of different reliability levels for different parts of his application. This would not only add to the complexity of the grammar of the language, but also supporting such a requirement would be quite difficult. However, it is not unreasonable to imagine scenarios where such a functionality will be useful. For example, in a long-running control application, some aspects like collecting the data from the sensors may be done in triplicate, while some other modules like computing on the sensor inputs may be done on a single node. It is important to emphasize that the set of execution strategies is not fixed.Other execution strategies may be easily developed using the Chameleon components and mechanisms to be described later. As will be seen later, Chameleon utilizes a well-structured ARMOR class hierarchy composed of substitutable components through which new execution strategies may easily be constructed. Once new execution strategies are developed, they only need to be registered with the FTM to become available for use. 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 (the creation of a surrogate manager is discussed later). In general, there is one surrogate manager for each user application. The surrogate manager utilizes Chameleon ARMORs to realize the execution strategy, thus freeing the FTM from having to manage the application being executed. Since several user-supplied applications may simultaneously execute in the Chameleon environment, the fact that the FTM and the executing application are loosely coupled allows the FTM to be more responsive to other operations (such as responding to additional user requests, overseeing the recovery from a failed network node, overseeing the recovery of a surrogate manager, etc.)

3.1.3 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. 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 the code, and runs the resulting executable file. As a result, the FTM must have access to the source code for the surrogate managers and the associated libraries of Chameleon functions. Recompiling at the destination node gives the added flexibility of multi-platform support, assuming the surrogate manager source code and Chameleon libraries are written to be platform-independent. [2] Surrogate managers utilize Chameleon components called ARMORs to meet the user-specified levels of fault tolerance for a given application. ARMORs are specialized entities that perform much of the work in the Chameleon environment, and they are designed in such a way that no individual ARMOR is a single point of failure. As discussed later, ARMORs are constructed from a common library of basic building blocks and are intended to be reusable and extendible. ARMORs supply much of the functionality to the Chameleon system. All ARMORs in Chameleon fall into one of three broad categories: managers, daemons, and common ARMORs. We have already seen examples of managers (FTM and the surrogate managers) and daemons. From the standpoint of the Chameleon architecture, managers and daemons are merely specialized ARMORs. The generic term ARMOR, therefore, refers to the common ARMORs as well as managers and

[2] This was a major consideration during the development of the code for the Chameleon entities. The platform-specific portions of the entities have been marked out explicitly in the code. At the beginning of each entitity we determine the OS platform on which the entity is running, and we use a variable to switch between the OS dependant part of the code depending on this knowledge. We have had encouraging experience with supporting several flavors of UNIX, Linux and Windows NT.

daemons. Common ARMORs are those ARMORs that the surrogate managers use to attain a particular level of dependability. All 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 necessary ARMORs on other nodes 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. It is important to emphasize that after the surrogate manger successfully installs ARMORs necessary to execute the user application, it notifies the FTM where the individual ARMORs are located. Consequently, the FTM is able to initiate correct recovery actions in the case of a surrogate manager failure. As can be seen, Chameleon provides for the reliable execution of a user application through the use of ARMORs. These ARMORs are closely monitored through the techniques described in Chapter 4.

3.2 Functional Overview

Here we look at the complete list of functionalities provided by each of the entities in the Chameleon environment. As mentioned in the previous section, all ARMORs in the Chameleon environment may be classified into one of three groups: managers, common ARMORs, and daemons. A description of each group and representative examples of each group are given below. Note that the set of specific ARMORs found in this section may be expanded to incorporate new fault-tolerance features into the Chameleon environment.

3.2.1 Managers

Managers are specialized ARMORs that possess the following common capabilities:

- The ability to remotely install ARMORs on other nodes.
- The ability to assign system-wide identifiers to the ARMORs that it installs. Each ARMOR has a unique identification number that allows it to be distinguished from any other ARMOR in the Chameleon system.
- The ability to maintain an updated list of its subordinate (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. At this level, nodes are identified by the identification number of the daemon installed on the node. Since all ARMORs (and hence daemons) have unique identification numbers, these number uniquely identify all nodes in the network.

The remainder of this section describes specific examples of managers in the Chameleon environment. Note that this does not represent a static list of managers, but rather examples of managers that are included in the initial implementation of Chameleon.

Fault Tolerance Manager (FTM). The FTM is a centralized, key manager of the Chameleon environment. It has the following functionality:

- 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.
- 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 application on the target node . The

list of hosts available for use by the FTM may change dynamically because of node failures, because of nodes voluntarily leaving the environment, and because of new nodes joining the environment.

Surrogate Manager (SM). Surrogate managers are specialized managers with the following features:

- It is responsible for executing a single user application under a specific fault-tolerant strategy. Currently, surrogate managers exist for dual, TMR, and quadruple execution modes.
- The surrogate manager is capable of installing any other ARMORs required for executing an application under the fault-tolerant strategy selected by the FTM.
- When assigning identification numbers to the ARMORs it installs, the surrogate manager must query the FTM for a list of valid identification numbers (numbers not previously assigned to other active ARMORs).

3.2.2 Common ARMORs

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 be a simple ping message to determine if the node is alive or not. At the other end, the heartbeat ARMOR can be quite sophisticated and encapsulate within it information about the health of the node being monitored. Our prototype heartbeat ARMOR can monitor the number of errors in the various components of a node (e.g., errors in memory and errors in I/O). The absence of a heartbeat from a node may not indicate that the node is being down, but rather the ARMOR

that responds to the heartbeat may be down, or the heartbeat interval is too short and the machine (might be a slow machine) is unable to respond within the specified heartbeat interval. Therefore, the heartbeat ARMOR pings the machine a fixed number of times before it declares the machine as being down.

Execution ARMOR. This is the basic ARMOR responsible for installing an application on a particular host, overseeing its execution, and finally, communicating the result of the application back to the manager.

Voter ARMOR. This is a generalized ARMOR capable of majority voting on the results obtained from other ARMORs (such as execution ARMORs or voter ARMORs). Different voting strategies and characteristics may be obtained by overriding the default behavior of the voter ARMOR (e.g., n of m voting, self-checking voting, etc.). 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 hints from the user who supplied the applications. During the 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 - worst (i.e., the longest), best (i.e., the shortest) time reported by the ARMORs. For example if the timeout was initialized to 5s and the three measurements (arrived from the three ARMORs) are 1.3, 1.1, and 2.2, the voting timeout is readjusted to the value of 10s (i.e., 5 * (2.2/1.1)) to take into account the performance of the slowest machine .

Checkpoint ARMOR. The checkpoint ARMOR interacts with the execution ARMOR to enable the checkpointing of applications running on a particular node. This ARMOR is crucial for application recovery on homogenous nodes. On detection of an application failure, an initial attempt is made to restart the application on the same node from the last checkpoint. If this fails, then the checkpoint ARMOR notifies its manager to initiate recovery on a different node.

3.2.3 Daemons

Daemons are an important class of ARMORs that perform the following functions in the Chameleon environment:

- Installs ARMORs locally on the node. This is in contrast to managers, which remotely install ARMORs by sending messages to a daemon. Daemons perform the low-level installation of an ARMOR on a node (i.e., spawns a new process, sets up an appropriate communication channel between itself and the ARMOR, notifies the FTM as to the location of the newly-installed ARMOR, etc.).

- Monitors all locally-installed ARMORs. Details of the error detection provided by the daemons can be found in Chapter 5.

- Serves as the primary gateway for all communication between ARMORs in the Chameleon environment. 5.1 describes the ARMOR communication process in more detail.

- Responds to heartbeat messages from the heartbeat ARMOR.

Because of the vital role that daemons play in the providing communication capabilities to locally- installed ARMORs, the loss of a daemon renders the entire node unusable - the locally-installed ARMORs effectively become isolated from the rest of the Chameleon environment.

Care must be taken, therefore, to accurately detect daemon failures and rapidly recover from such failures. Chapter 4 discusses how the heartbeat ARMOR and FTM work together to provide daemon error detection and recovery. Since the daemons perform the actual network communication in the Chameleon environment, they must be specific to a particular network protocol. Our current implementation is built on TCP/IP because of the portability and ease of implementation it offers, but there is parallel effort to use MPI as well (we have some early implementation of some Chameleon features based on the MPICH libraries, [22]).

3.3 Structural Overview

In this section we present a detailed description of the class hierarchy of the ARMORs in the Chameleon environment. The entities in the environment have been structured around object-oriented principles. Some of the specific advantages that can be derived out of this can be stated as follows. The hierarchy allows us to keep the individual entity's complexity within bounds thus making validation that much easier. Code duplication is avoided and since in our environment, the code often needs transportation from node to node, this is a major gain. The class structure allows us to spread the detection and recovery responsibilities in a logically consistent manner. For example, when an error message arriving at a particular ARMOR cannot be handled by the methods implemented in that ARMOR, it passes the message up the hierarchy for handling by a base class. During detection, when a fault notification has to be propagated, this tree structure (arising out of the class hierarchy) provides an efficient way of dissemination of the knowledge from one entity which detects the fault. This hierarchy has advantages in the addition of functionalities to the environment. Thus, for example, if we decided to add a new exotic configuration to our environment (say, half of the application in

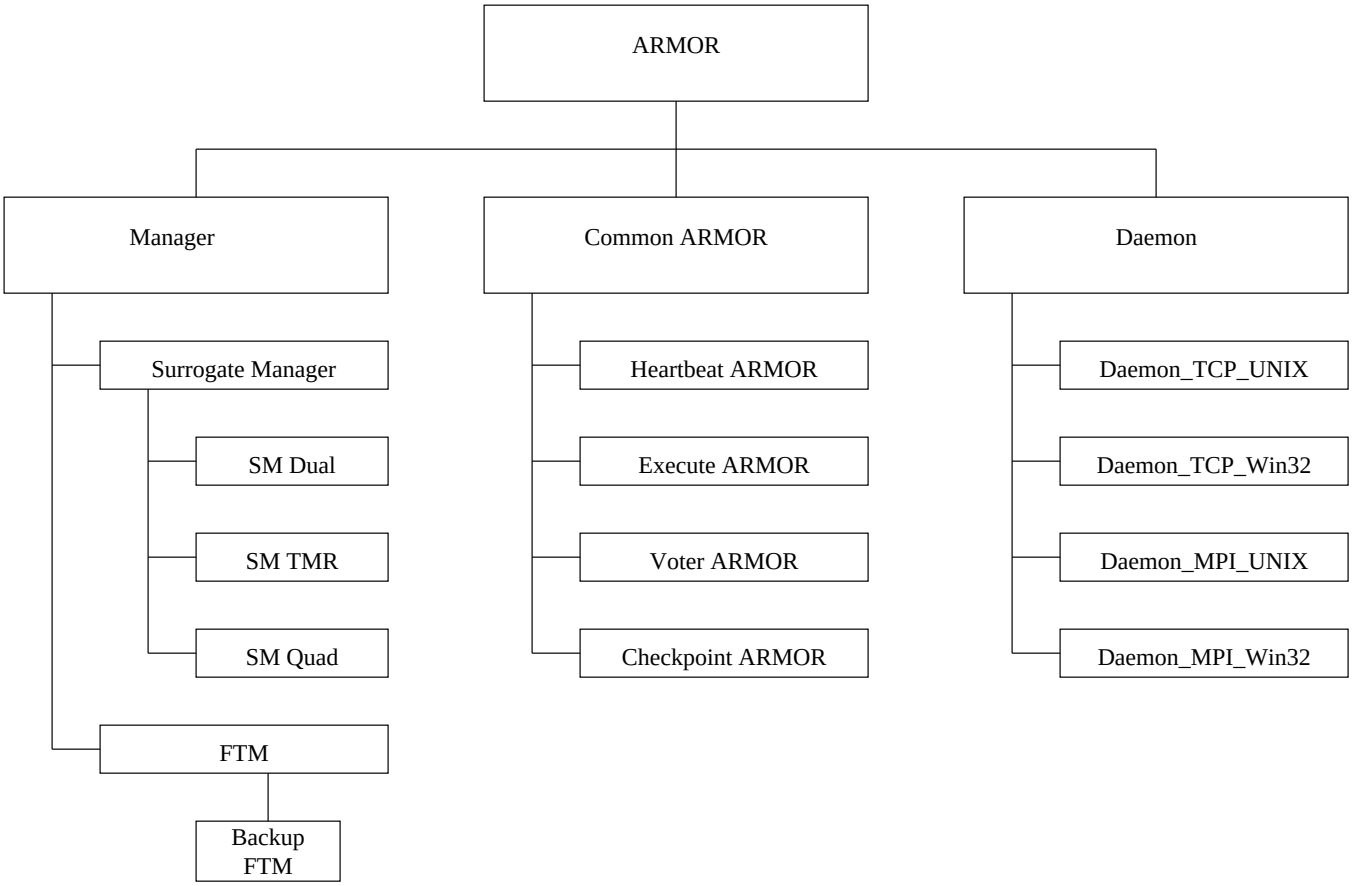

Figure 3.2 ARMOR Class Hierarchy

TMR, and a less critical half in dual with result taken from the first run to complete), we would create the added functionalities for it and to inherit all the base functionalities of a surrogate manager, just hang it from the class tree below the Surrogate Manager class.

The class hierarchy of the ARMORs can be succintly presented through the figure 3.2.

The base ARMOR class corresponds to the basic building blocks that are accessible by all entities. For example, the `get_manager_id` primitive. On the other hand, some primitive like `install_ARMOR` is useful only to the manager class and is therefore not put in the base class, but

in the manager class. The Daemon class has two classifications, according to the OS platform and also the communication layer it uses.

3.3.1 ARMOR Customization

We propose an automated method for modifying or extending the functionalities of the ARMORs. An important goal of Chameleon is to provide a middle layer which would be easily extensible and thus flexible enough to adapt to changing application needs. This is sought to be achived by being able to incrementally add to the functionalities of the existing ARMORs on a *as-needed basis*, without a need for complete rewriting. It would be ideal to be able to do this in an automated manner. This would involve the following three basic steps, each of which is a challenging problem in itself. First, the available functionalities in the current libraries have to be characterized in a manner that is easy to manipulate (i.e. do matching operation, and to reason about). Second, the user specification has to be mapped to a representation that is easy to match with the above representation, and in the case of no match, should lend itself to easy synthesis of the new ARMORs. Third, the new ARMORs will have to be synthesized, hopefully not from scratch, but by re-engineering of some existing ARMOR.

Initially, we have simplified the first two steps, by making the specification language quite rigid. The available configurations are represented by discrete integers. The user specification of the configuration is given in terms of dicrete numbers, like the number of basic ARMORs that it requires, and say for the voting strategy, a number denoting which of the three available voting strategies to employ. The re-engineering is now made simpler because we can have a mapping from these discrete levels to the parameters to be passed to the entities. As shown in

figure 3.3, we have three libraries : library of basic building blocks, of common ARMORs, and of surrogate managers. Each of them provide support for certain levels of customization.

Let us look at an example for a simple re-engineering of an existing configuration. Suppose we have a Surrogate Manager for the base TMR configuration, with three Execution ARMORs and a single Voter ARMOR. Now suppose to make the configuration more robust, we decide to replicate the voter and have three voters with each of the voters voting on all three values and only in case of agreement at all three voters we accept the results of the application. Also suppose the application is a simulation run and two correctly-running simulations will also have some discrepancy in the results. Thus we would like to provide some slack to the voter ARMOR. The first level of customization occurs at the level of the Surrogate Managers. Here, we create the new Surrogate Manager that instantiates three voter ARMORs instead of one. Also to each of the voter ARMORs, it passes a parameter to indicate the type of voting strategy to employ. For example, we have a strategy for approximate match. It specifies that and indicates that a match range of 10% is allowed. Also to each entity we provide at the time of instantiation, the id of the entity to which it has to convey the result. In the base TMR case, the voter would have communicated its result to the SM. Here each of the voter's result is communicated to each of the other two voters. The voter ARMOR at a lower level uses a base ARMOR class functionality called `compare`. This compares two values and returns a boolean value indicating a match or mismatch. The voter ARMOR when it invokes this functionality passes it the slack value of 10% for doing its comparison. In this way, we have created a new Surrogate Manager for the new configuration desired by the user.

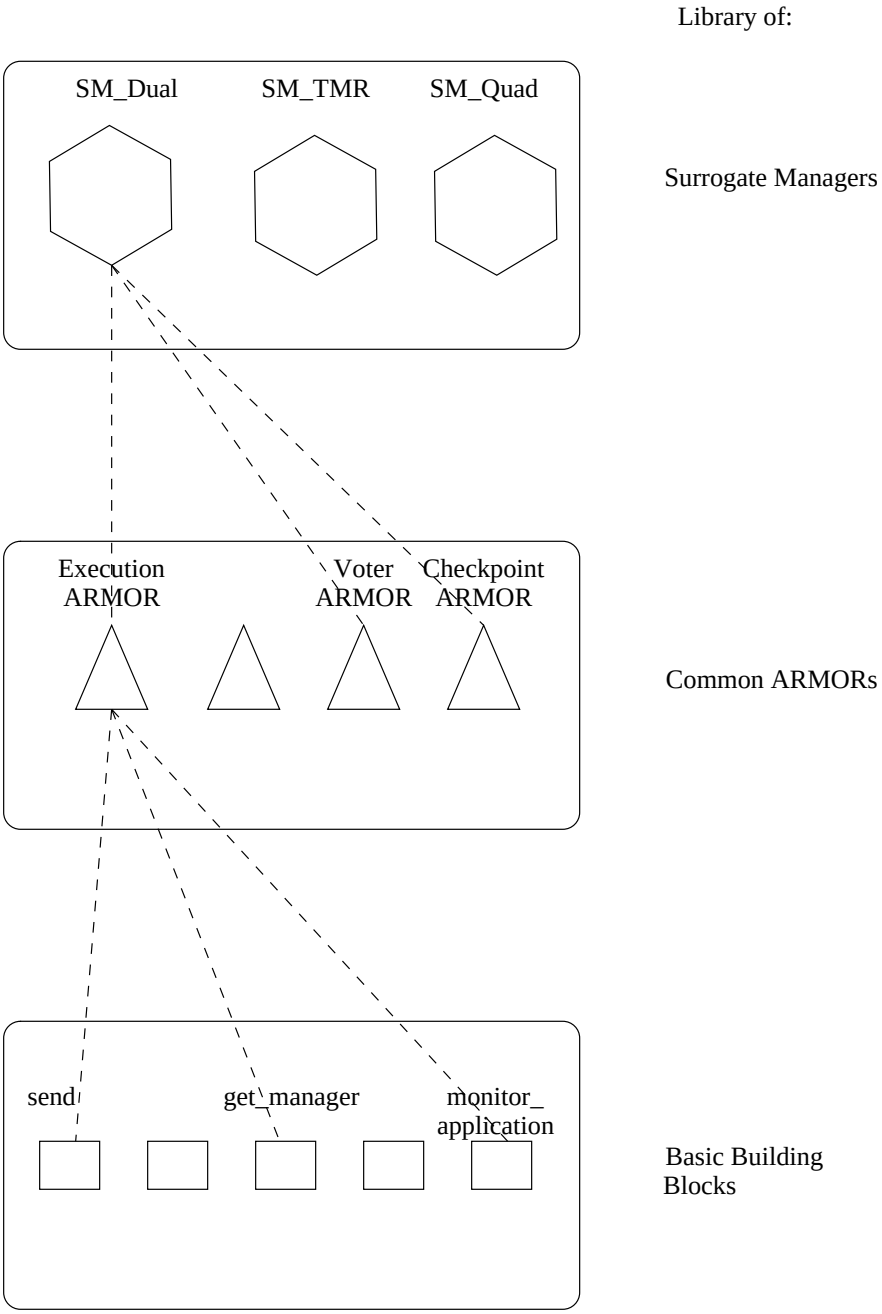

Figure 3.3 Hierarchy of Libraries

3.3.2 ARMOR Reuse

The next step is to incorporate this added functionality in the current library. This new surrogate manager is inserted into the class tree, a description of the new functionality is generated [3] and a new discrete number is allotted to this configuration.

To use the newly created ARMORs, we use an object called the *ARMOR Factory*. In an abstract sense, its function is quite simple. Being passed the type of ARMOR to create (e.g. Surrogate Manager) and an identifying number alotted above to the configuration, it returns a handle to the particular type of ARMOR. This handle can then be used to invoke the functionalities of the ARMOR. A sample code fragment for the ARMOR creation in the factory is provided in figure 3.4.

[3] This is a challenge in itself to represent the functionality in a non- ambiguous way.

```
Agent *Agent_Factory::Create_Agent(int AGENT_TYPE)
{
        switch(AGENT_TYPE)
        {

        case SM_Dual:
                        return new SM_Dual;
                        break;
        case SM_TMR:
                        return new SM_TMR;
                         break;
        case SM_TMR_Special:
                         return new SM_TMR_Special;
                          break;
                .
                .
                .

        }
}
```

Figure 3.4 ARMOR Factory Skeleton

CHAPTER 4

ERROR DETECTION AND RECOVERY

The Chameleon system is designed to detect and recover from failures in any of the components. In some instances, multiple detection and recovery paths are provided (see 4.4). The design tries to get away from any of the components from becoming a single point of failure. The components in the environment from the point of view of failure analysis can be classified as follows:

(1) The participating nodes [1]

(2) Chameleon ARMORs

(3) User application

Below we present the details of the detection and recovery of components from each of these classes. A more concise tabular representation is given in Fig. 4.1.

4.1 Failures in Participating Nodes

The hardware crash of a participating node is detected by the Heartbeat ARMOR. When the Heartbeat ARMOR does not elicit a response from a node for a certain threshold number of times, then the FTM concludes that the node has failed. All the Chameleon entities running on that node as well as any application are presumed to be lost. The recovery that is undertaken

[1] This encompasses both the hardware platform of the node and the operating system running on it.

is that the FTM informs all the Surrogate Managers that had ARMORs on the node. If the Surrogate Manager itself was on the node, then the FTM reinstalls the SM on another node.

In case the node behaves in an erratic manner (responding to some queries and remaining silent for some others - a situation that might arise due to an unreliable network connection), then it might be detected in one of several different ways depending on which entity it affects. If it affects only some of the local entities, then the entity overseeing that will detect the malfunction. For example, if the application executing on that node is affected, then the Execution ARMOR will detect it. On the other hand, if all the local entities are affected, then the misbehavior can be detected on comparison of the results of the computation from various nodes, or during the validation check that is done on the messages arising out of that node. For example, a misbehaving node may send out messages to another node's Host Daemon destined for an ARMOR that does not exist on that node. Note that we are not trying to provide 100% coverage for the most general of fault models, but provide protection from a range of faults that seem likely in a practical situation.

4.2 Failure in User Application

The Execution ARMOR monitors the application executing at a particular node under a given fault-tolerant strategy. The detection mechanism is through a signal that will be raised by the application in case of malfunction. The signal is captured by the ARMOR and by looking at the type of the signal, the ARMOR diagnoses the problem. As recovery, the application will be restarted by the ARMOR a certain number of times. Of course, the Surrogate Manager for the execution of the application will be informed of the application's fault. In case checkpointing is enabled, the application will be restarted from the last saved checkpoint, otherwise it will be

restarted from the beginning. If the restart of the application fails a certain number of times, then the Surrogate Manager will kill the application and the ARMOR on the particular node and migrate the application to some other node. Also, in case of any application failure, the Voter ARMOR is notified to adjust its timeout value accordingly.

4.3 Failures in Chameleon ARMORs

The Chameleon ARMORs are rigorously tested against design defects and the code of each individual ARMOR is kept simple enough to aid debugging efforts. As such, any failure of Chameleon ARMORs can reasonably be expected to be due to transient errors in the platform on which the ARMOR is executing. In the first path of detection, [2] error in any ARMOR is detected by the Host Daemon on the local machine. The Host Daemon in its data structure maintains the ID and location of the manager of any Chameleon entity executing at the node (Refer to the data structure presented in 5.4. On detection, it informs the manager of the ARMOR that has been affected. The recovery action initiated by the manager is usually to re-install the ARMOR either on the same node or on a separate node (depending on the recurrence of the fault). It then updates the ID to location mapping of the ARMORs accordingly.

The FTM is the only entity that does not have a direct manager. It is monitored by the backup FTM through a heartbeat. If it does not detect a heartbeat from the FTM for a certain number of times, then it promotes itself to be the new FTM and informs all ARMORs of the location of the new FTM. [3]

[2] We talk of multiple error detection paths in 4.4.

[3] This propagation of information flows down the tree denoting the manager-ARMOR relationship.

The failure of the backup FTM is detected by the usual mechanism of the Host Daemon. On being informed the FTM just installs a new backup FTM on another node, updates the new backup's state and makes it consistent with its own current state.

4.4 Multiple Error Detection Paths

In Chameleon, we provide multiple paths by means of which errors in some entities may be discovered. For example, take the case of a misbehaving application. Ordinarily, the application error will be detected by the Execution ARMOR monitoring it. In case this detection fails (possible when the application has a value error), then the error will get detected at the Voter ARMOR when the results from multiple runs of the application will be compared.

4.5 Failure Modes

Figure 4.1 presents the primary failure modes for the Chameleon environment. The table is intended to be self-explanatory, and hence we will make very generic remarks about the table 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, it gives the fundamental steps involved in the recovery from the detected error.

From Figure 4.1, one can observe that there exists a certain hierarchy in error detection and recovery. This hierarchy is illustrated in Figure 4.2, which provides the primary paths of error detection and recovery activities. The secondary paths are not depicted in the figure to preserve its clarity.

	Failure Mode	Consequence	Detection	Recovery
Node	Crash	All agents lost on the node	Heartbeat agent	• HB agent notifies FTM • FTM removes node from list of registered nodes • FTM restarts any affected agents it manages to a new node • FTM notifies immediate managers of the crashed node; these managers restarts any of their agents 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 agent	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 agent	• Notify the execution agent's manager • Restart the application (with assistance from a checkpoint agent if enabled)
	Livelock	No forward progress made in the application	Execution agent through a user-supplied timeout	• Kill application • Restart the application. • If repeated restarts result in livelock, notify execution agent's manager • Manager may elect to reinstall the execution agent on a node with a different platform • If installing the agent on a new platform fails, the user will be notified of an apparent software bug
	Compilation error	Application executable not generated	Execution agent	• Retry compilation • If retry repeatedly fails, request a fresh copy of the source code from the execution agent's manager • If new source code cannot compile, notify the execution agent's manager to try installing the execution agent on a node with a different platform • If the application will not compile on the new platform, notify the user of an unrecoverable error
	Erroneous computation	Incorrect results	Voter agent	• Dual mode: restart the application and notify the user; • TMR mode: mask the error (optionally notify the user)
Common Agent	Crash	Lost agent	Daemon	Notify the crashed agent's manager (recovery as described in Chapter on Error Detection and Recovery).
	Compilation error	Agent not installed	Daemon	• Ensure that all Chameleon libraries are present on the node. If not, request the appropriate libraries from the agent's manager. • If re-compilation repeatedly fails, request a fresh copy of the source code from the agent's manager • If new source code cannot compile, notify the agent's manager to try installing the agent on a node with a different platform • If the agent will not compile on the new platform, notify the user of an unrecoverable error
	Process alive, but unresponsive	Agent cannot process incoming messages	Daemon	• Kill the unresponsive agent • Notify the agent's manager to reinstall the agent
Daemon	Crash/Unresponsive	All agents on the same node cannot communicate with remote agents	Heartbeat agent	• 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 agents of the change; all subordinate managers recursively notify managed agents • New FTM selects a new backup FTM.
FTM Daemon	Crash/Unresponsive	FTM unreachable	Backup FTM	Assume the FTM has crashed and recover as above

Figure 4.1 Chameleon Failure Modes and Recovery

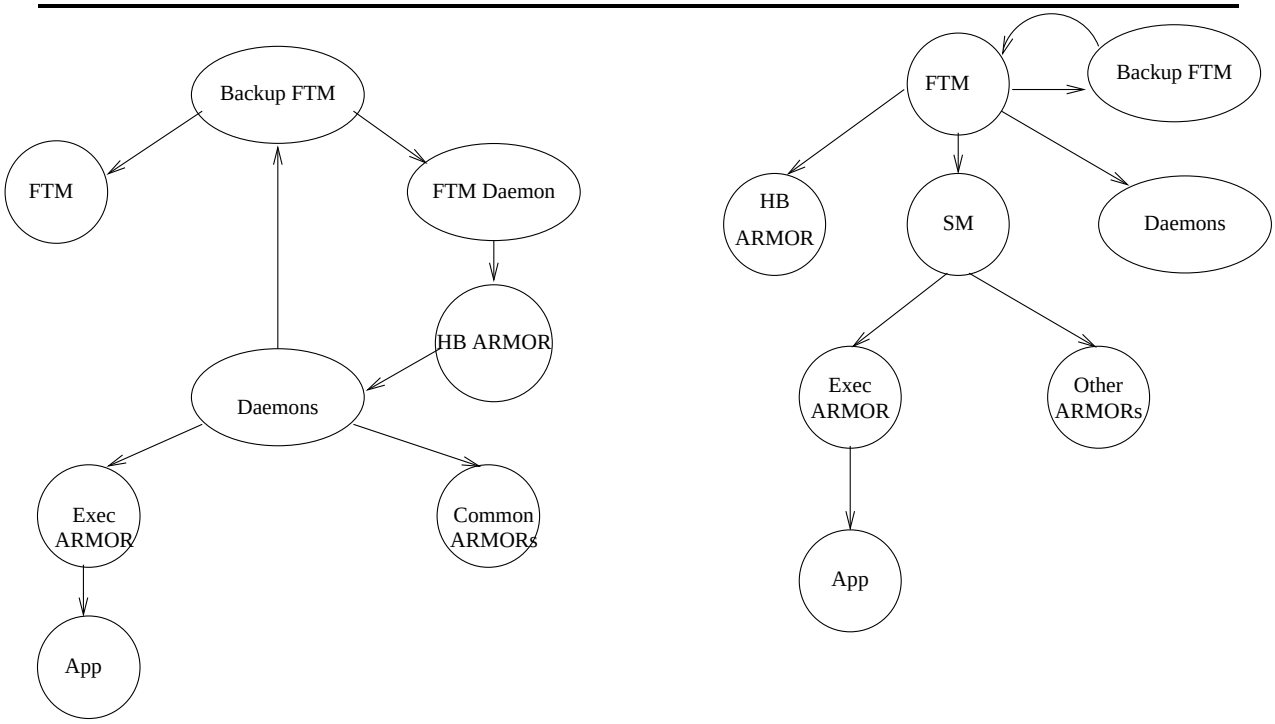

Figure 4.2 (A) Error Detection (source node detects errors in sink node); (B) Error recovery (source node recovers from errors in sink node)

CHAPTER 5

IMPLEMENTATION OF CHAMELEON

We have a prototype implementation of Chameleon on a testbed of heterogeneous computing nodes at the Center for Reliable and High Performance Computing at the University of Illinois. The computation nodes communicate with one another using TCP/IP protocol over the 10Mb/s Ethernet. The software has been ported to Sun OS, Sun Solaris, HP-UX and Windows NT. The configuration of the machines on which the environment is running is summarized in table 5.

Name	**Manufacturer/ Model**	**Memory**	**Disk**	**OS**
tyagaraj	HP9000/715	32M	2.0G	HPUX9.05
mozart	Sun Ultra1-170	128M	2.1G	Sol2.5
nahoona	Sun Ultra1-170	64M	2.1G	Sol2.5.1
dvorak	HP9000/C160	64M	2.1G	HPUX10.20
franck	Sun Ultra1-170	64M	2.1G	Sol2.5.1
bizet	Sun Ultra1-140	64M	2.1G	Sol2.5.1
karl	Sun Ultra1-200	192M	4.2G	Sol2.5.1
bernstein	Sun Ultra1-140	64M	2.1G	Sol2.5.1
wolf	Sun Ultra1-170	128M	2.1G	Sol2.5.1
berg	SunUltra1-140	64M	2.1G	Sol2.5.1
intel1	IntelP6-200	64M	4.2G	Windows NT4.0

Table 5.1 Experimental Machine Configurations

The execution modes which are supported are: (1) A single node execution offerog baseline reliability; (2) Dual mode execution where the first result is accepted; (3) Dual execution with

the requirement that both the results must agree for success; (4) TMR execution with majority voting. All of the above modes can optionally utilize the functionality provided by a checkpoint ARMOR. Currently, however, the checkpointing library has been developed only for the Solaris platform. Consequently, if the application wishes to take advantage of rollback recovery, it has to be running on the Solaris nodes from among the machines listed above.

5.1 Communication

The environment uses a message passing model of communication. Messages contain payload like the source code of applications and ARMORs as well as control messages like notification of some error. The TCP/IP communication layer is used in the current implementation, and there is a parallel effort in implementing some of the ARMORs using MPI [22]. For the TCP/IP communication, all communication passes through the Host Daemon's TCP port. The Daemon communicates to the ARMORs installed at its node using the interprocess communication device of pipes. So for sending message to any entity in the environment, the destination entity's ID [1] is specified in the header and the message is passed to the Host Daemon IP address and TCP port.

The reason behind using the Host Daemon as the gateway for communication is to prevent a maverick ARMOR from sending corrupt information out of the node. At the Host Daemon, we can implement various schemes to check on the validity of the message, or the integrity of the source of the message before the message is allowed to be transmitted to a remote node. Protecting the message against network errors by appending CRC bits can also be done at the

[1] A system-wide unique identification of the active entities in Chameleon.

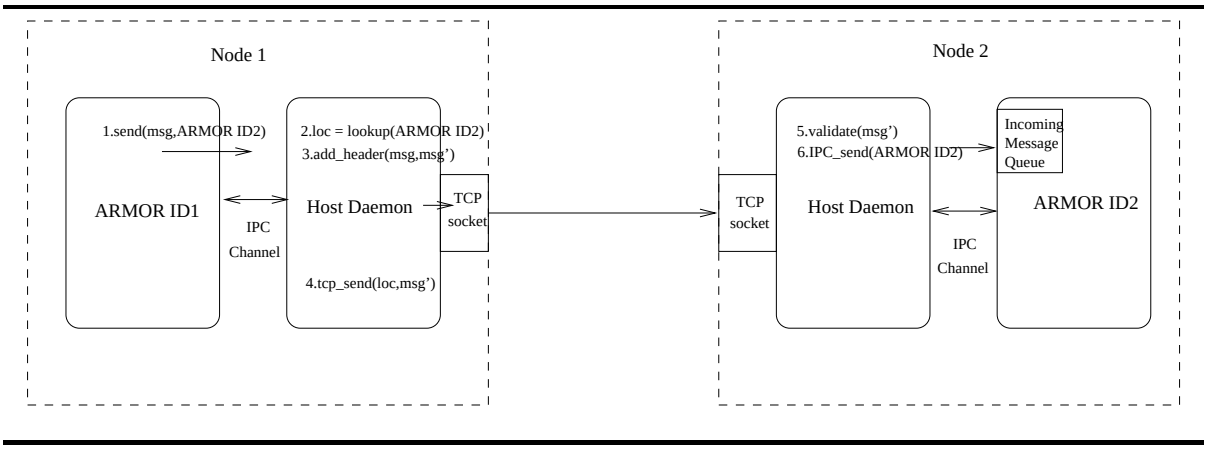

Figure 5.1 ARMOR Communication Mechanism

Host Daemon. Rather than requiring all the ARMORs to have CRC computation capability, this calls for only the Host Daemon to have such a facility.

The communication infrastructure is schematically presented in fig. 5.1. 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 out a message, an ARMOR passes on the message to the local Host Daemon through an IPC (Inter Process Communication) channel. The daemon does a lookup and translates the Chameleon ID of an ARMOR into the location of the node on which the ARMOR is resident. Before sending out the message over a TCP socket, it adds header to the message which contains source and destination identifiers, and possibly, a checksum or a CRC to provide an extra layer of protection. When receiving messages it also does a validity check on the message like checking that the ARMOR for which the message is meant is locally resident or not. It then forwards the message to the appropriate local ARMOR over the IPC channel.

5.2 Applications

Chameleon seeks to support off-the-shelf standalone as well as distributed applications. For the standalone applications, we have used some nested loop-based application. For the distributed application, we have used a matrix multiplication application, hand parallelized to run on two nodes. The applications so far have operated in the batch mode with it taking the inputs at the beginning of the run, and outputing the results of the run at the end into a file. However, an extension is not difficult to implement. For a standalone application, all the inputs can be specified in a file at the beginning and read off periodically by the application during various stages of its run. However, this is still not basically an interactive application. To get the flavor of an interactive application, the communication path between the Execution ARMOR and the FTM which is the point with which the user makes its communication, will have to be traversed more often. The requirement in our application is that the result to be compared or voted upon be available in a file. This could be an output file which is generated by the application itself, or the run's output could be piped to an output file explicitly. For a distributed application, the environment currently supports applications where there is not much sharing of state and synchronization between the individual tasks. The application that we have consists of two sub-tasks (hand parallelized a priori). Each sub-task computes the matrix product of one half of the result matrix. At the end of the computation the result is output to a file. This file is collected by each Execution ARMOR running on the corresponding node and sent to the Voter ARMOR. At the voter, the two parts are combined and the result is voted upon. Actually to optimize the voting time, we do not vote upon the whole result matrix, but only on the checksum that is computed for each row and column of the matrix.

In order to support tasks that share more state and need to synchronize at a more fine-grained level, we might have to use techniques like those provided by Group Communication (like atomic broadcast, causal boradcast). However, we believe that the entire functionality provided by Group Communication would not be useful in our setting. Moreover, we cannot violate the basic tenet of not requiring the applications to be rewritten. So it is a better alternative to add to the functionality of the execution ARMORs to trap the results and communicate them to the other replicas, maintaining any guarantees that may be required. For example, if the required synchronization be that all functioning replicas need to receive a value generated by an entity which may itself fail at any time after it has started the process of transmission of the result (i.e., a reliable multicast protocol) we can add to the functionality of the Execution ARMOR to implement a three-phase commit protocol. The problem in this approach is for the Execution ARMOR to be able to trap network I/O calls by the entities and interpose its own protocol message passing. We believe that this requires some hooks in the kernel.

Chameleon seeks to support off-the-shelf standalone as well as distributed applications. Replication management in standalone applications does not pose much of a problem. For distributed applications, the synchronization between the tasks presents a more challenging problem. Consider a case where we have an application that is parallelized into several tasks and these tasks are communicating amongst themselves. We need the communication to be visible to the Chameleon entities so that they can take appropriate actions, like sending the communication to multiple replicas , or doing validation check on the communication. This can be enforced by one of the following two ways. One, the application is aware that it is executing in the Chameleon environment. So, when it acts as the sender, it routes it 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 rewritten for the environment. Alternately, 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 and have the Execution ARMOR forward the message to an appropriate ARMOR, say the Voter ARMOR for voting on the message. The drawback of that is that we no longer have off-the-shelf OS and it will not be easily portable across platforms. The Chameleon cell shown in Fig. 5.2(a) is the typical support provided for distributed applications. The Fan-Out ARMOR broadcasts a message to all the replicas, while the Voter ARMOR votes on the incoming messages from all the replicas and passes on a single value. We exemplify use of the cell in Fig. 5.2(b). Suppose we have a client-server based application where the client is updating a replicated database. For a particular FTES, we have replicated the client, the server and the actual database threefold. Now to synchronize updates to the database we interpose the Chameleon cell as shown in Figure 5.2(b)

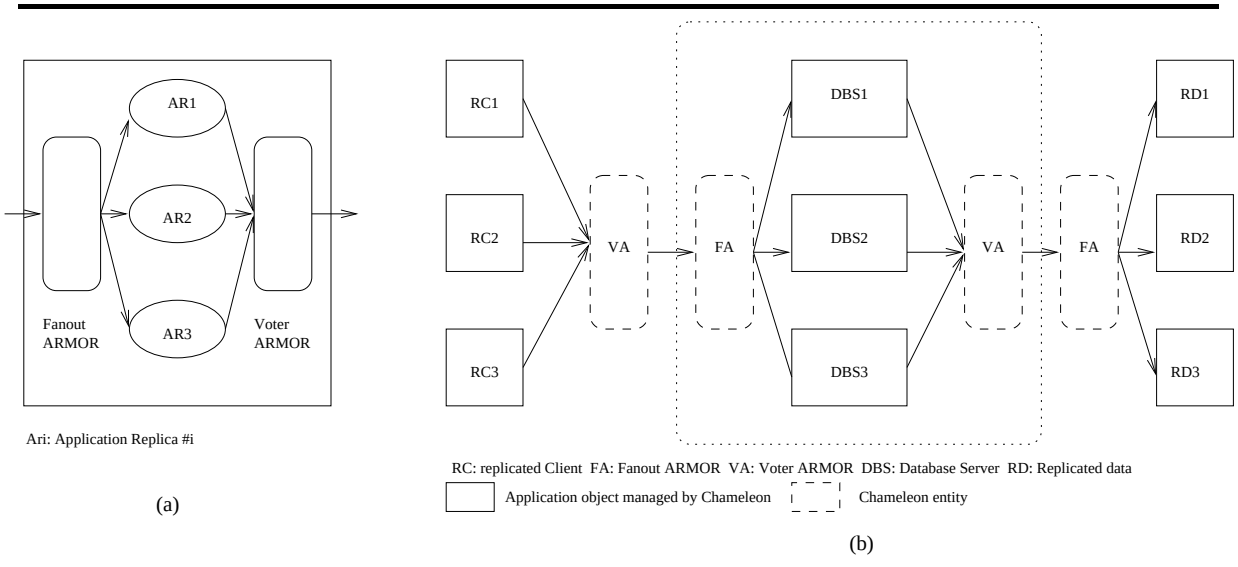

Figure 5.2 (a) A Chameleon Cell for Supporting Distributed Applications (b) Use of Chameleon Cell for Replicated Database Application

5.3 Implementation of the ARMORs

The ARMORs of Chameleon have been implemented in C/C++. In all cases, the source code of the entities is available for transporting to remote nodes and possible compilation. The environment specific variables, like the location of the FTM, is specified in a specification file `Chameleon.h` which is added to the header of the entities before compilation. Following the design of the class hierarchy outlined in 3.3, we build the ARMORs around the basic building blocks. Some samples of such usage are given in table 5.3.4. All the entities use message passing style of communication, the messages being routed through the Host Daemon on the local machine.

5.3.1 Fault Tolerance Manager

This is the controlling manager in the Chameleon environment. The FTM is the entity with which the user has any direct communication. The FTM cycles through the following actions during the course of execution of an user application [Figure 5.3].

It is important to note that the FTM is built to handle more than one user task. For every user task that it accepts it forks off a a process and then in the main process it waits for subsequent inputs. The FTM, though is the controlling entity in the environment, is not over loaded with activities. After a particular Surrogate Manager has been instantiated, the FTM is essentially idle except for its monitoring task. The FTM has the responsibility of monitoring the hosts. This it does by sending a periodic heartbeat message. When there is a timeout of the heartbeat to a particular node and it exceeds a certain threshold limit, then the FTM concludes that the node is down and initiates recovery actions. This consists of letting the

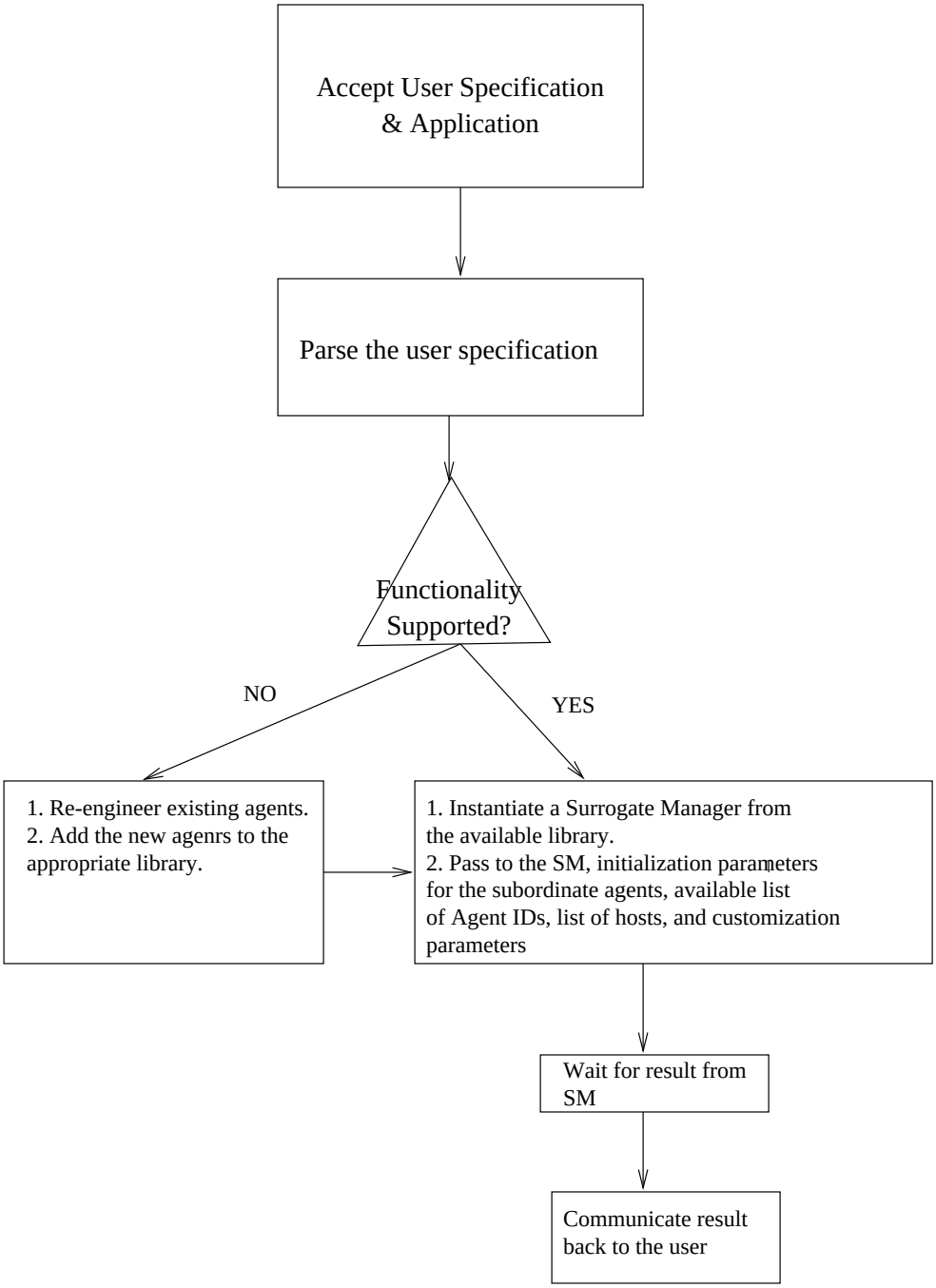

Figure 5.3 FTM: Flow chart of activities for a particular application

Surrogate Managers who had ARMORs on that node know of the failure, or in case the failing node housed a Surrogate Manager itself, instantiate another copy of the Surrogate Manager.

The FTM is installed on a well-defined port and IP address in the environment. It is built to handle TCP/IP communication, and can handle multiple communication channels concurrently. (Current implementation limits the concurrency to 10 active channels.) All entities in the environment have to be aware of this location of the FTM. Initially, a skeleton initialization process running on a node wishing to join the environment, sends a request to the FTM. In response the FTM sends the Host Daemon source code to the node and it is compiled there and installed. Subsequently the node can take part in this environment, i.e. can host ARMORs on itself.

The FTM has to keep global knowledge of the location of the currently active entities in the environment. For this purpose it maintains the following data structure.

```
struct FTM_Location_Map {
struct ARMOR_Type type;
struct ARMOR_ID   id;
struct Location   loc;
}
```

The `ARMOR_Type` denotes the integer identifying each type of ARMOR. Thus, we have a unique number for the Surrogate Manager for the TMR mode, and one for the Dual mode. The `ARMOR_ID` is the unique id that is assigned by the Surrogate Manager to each of its subordinate ARMORs at the time of installation. This ID therefore has to be communicated back to the FTM. The ID for the Surrogate Managers is assigned by the FTM itself. The location variable refers to the communication port of the corresponding Host Daemon. (Remember that the

communication is directed through the Host Daemon [5.1].) For the TCP/IP case, it will refer to the IP address of the Daemon and the port on which it is installed.

When the FTM installs a Surrogate Manager it provides to it a list of IDs from among which it can assign to its subordinate ARMORs. Also the SM queries it for a list of machines on which to execute the application in the desired configuration. The FTM can provide the machines so as to balance the loads on the machines, or on the basis of history of failures of the machines from which heuristic estimates of the reliability of the machines can be made. In the current implementation, it is decided upon arbitrarily, with the potential of making use of user hints if given. The user in the specification of his application can provide a list of platforms on which the application can execute.

5.3.2 Surrogate Managers

The current implementation provides four surrogate managers. They are all installed on the same node as the FTM and run as child processes of the FTM. Note that this is just the current state of implementation, and in the design the SMs and the FTM are much more separate and will execute on different nodes.

The SMs provided are:

(1) Single execution for the basline reliability case

(2) Dual execution where the result is taken from the first run to complete

(3) Dual execution where the results from both the runs are expected, voted upon and success is flagged upon match

(4) Triple modular redundant execution

```
Dual_SM_With_Voting(...)
{
  ftm_id = get_manager_id();
  get_list_of_machines(ftm_id);
  get_list_of_ids(ftm_id);

  For ARMOR_Exec(i)  {
    ARMOR *ARMOR_Exec(i) = ARMOR_Factory.Create_ARMOR(EXEC_ARMOR_TYPE);
     ARMOR_Exec(i)->destination_ARMOR = ARMOR_Voter;
     ARMOR_Exec(i)->  id = ARMOR_Exec_ID(i);
      send(ARMOR_Exec(i));
  }

  ARMOR *ARMOR_Voter = ARMOR_Factory.Create_ARMOR(VOTER_ARMOR_TYPE);
  ARMOR_Voter->destination = Dual_SM;
  ARMOR_Voter->source_ARMORs = (ARMOR_Exec(1), ARMOR_Exec(2));
  ARMOR_Voter->id = Voter_ID;
  ARMOR_Voter-> timeout = Timeout;
  send(ARMOR_Voter);

  WAIT(Message);
  if (Message = "Execution was a success")
     notify_FTM(SUCCESS);
  if (Message = "Voter reports an error")
     notify_FTM_failure(FAILURE);
  if (Message = "Invalid message")
     discard_message();
  if (Message = "Execution ARMOR reports failure") {
     send(ARMOR_Voter_Adjust_TIMEOUT);
     get_list_of_machines(ftm_id);
     send(ARMOR_Exec_New);
     goto WAIT;
  }
}
```

Figure 5.4 Pseudo code for the Dual Execution mode Surrogate Manager

Depending on the user specification, each of these modes could additionally use the checkpointing ARMOR in an individual run of the application.

Pseudo code for the SM for the third mode is provided below [Fig 5.4].

5.3.3 Common ARMORs

The current implementation supports the following common ARMORs:

(1) Execution ARMOR

(2) Voter ARMOR

(3) Checkpoint ARMOR

(4) Initialization ARMOR

(5) Heartbeat ARMOR

For all these ARMORs, the ARMOR source code is available. Before the source code is transported to a different node for installation there, the user specified locations of the FTM (its IP address and port number), which are available in a separate header file are concatenated to the ARMOR source code. In this way the ARMOR is made aware of the FTM location, as well as we get away from having to hardcode this location in the body of every ARMOR's code. Later, if the FTM has to be migrated, this single header file needs to be modified. Also the already active ARMORs will have to be informed of this migration.

In the initialization portion of the ARMOR, it figures out the platform type on which it is running and uses this knowledge to switch between one of the various implementations in the platform specific parts of the code.

The ARMORs during initialization are made aware of the entity to which it has to communicate the result. It is made aware of the system-wide ID of the entity and just before communicating, it does a lookup on the actual location of the entity.

Execution ARMOR The activities in the lifetime of the Execution ARMOR are presented in Fig. 5.5

Voter ARMOR The voter ARMOR can be used to vote on arbitrary number of inputs and return majority for an arbitrary k-of-n match. The match criterion can also be specified. The core functionality of this ARMOR is derived from the base class primitive `compare`. At

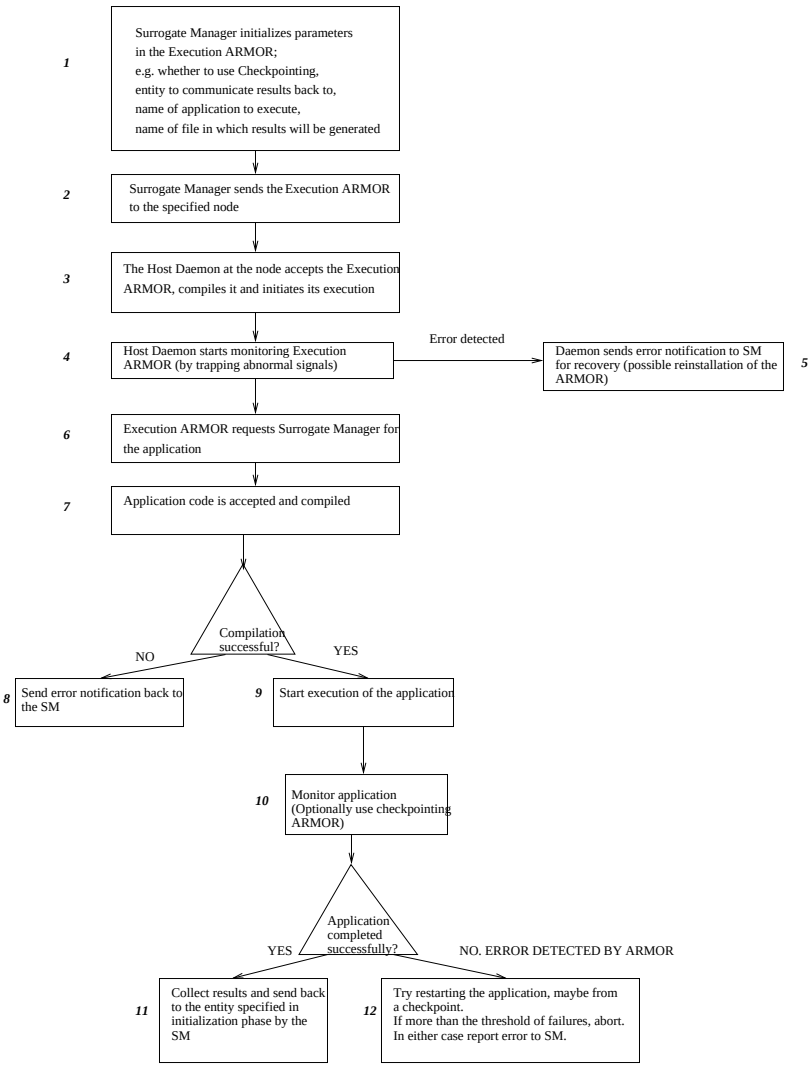

Figure 5.5 Flowchart for the Execution ARMOR

initialization, the IDs of the entities from which the voter is expecting results, the timeout period, and the ID of the entity to which it has to communicate the result are specified to the Voter ARMOR. Thus when it receives inputs to be voted upon, it validates the host address from which the value is coming.

Since the voter is a critical component in any replication scheme, an effort is made to keep it barebone and simple.

Checkpoint ARMOR The checkpoint ARMOR interacts with the execution ARMOR to enable the checkpointing of applications running on a particular node. Psudo code for interaction between the execution ARMOR, the checkpoint ARMOR and the application is presented in Fig. 5.6. Our current checkpoint scheme involves taking a dump of the process stack to a file using the UNIX call setjmp and restoring the process stack using longjmp. The checkpoint ARMOR is invoked by the Execution ARMOR and the application is linked with the Checkpoint ARMOR and forms a single process which the Execution ARMOR executes under its monitoring. The Checkpoint ARMOR can be invoked in one of the following three 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, and in the signal handling routine we make the calls to take checkpoint.

The last two methods are less intrusive because the application programmer doesnot have to be aware of the Checkpoint ARMOR. The checkpoint ARMOR is also used during recovery. When the execution ARMOR dete cts an application failure, it passes control to the checkpoint ARMOR which rolls back to the last saved checkpoint available in stable storage and restarts

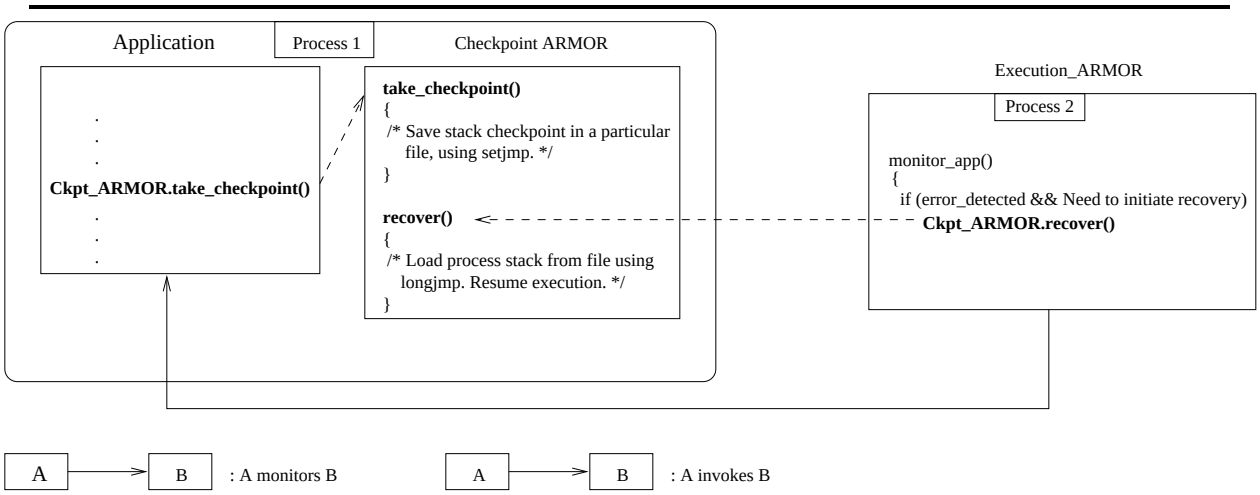

Figure 5.6 Interaction of Checkpoint ARMOR, Execution ARMOR and Application

the application from there. If local recovery fails a certain threshold number of times, then the execution ARMOR notifies its manager to initiate recovery on a different node.

Initialization ARMOR After the FTM has installed a Host Daemon on a node, this is the first ARMOR that it invokes to query for some system parameters of the particular node. The initialization ARMOR code gets taken to a particular node, compiled there and it executes there. During execution, it runs scripts like `sysinfo` and `uname` (available as shell commands in UNIX) and collects information about the OS type, the hardware type, the memory resources available. This information then it brings back to the FTM for it to update the table of all functional hosts in the environment that is available at the FTM.

Heartbeat ARMOR The heartbeat ARMOR is invoked by the FTM periodically to test the status of a node in the Chameleon environment. In the current implementation, it is as simple as a ICMP `ping` message that gets sent to the nodes and gets echoed from the hosts. If the heartbeat times out three times in succession, the FTM presumes the node to be down, removes it from its list of active nodes, and initiates recovery actions for all the ARMORs that were

resident on that node. As discussed in the previous chapter, it does not burden the FTM with actually recovering all the ARMORs. Rather it informs all the Surrogate Managers that had ARMORs on the affected node and it is up to them to initiate recovery.

5.3.4 The Base ARMOR Class

Following in the lines of the hierarchical approach outlined in 3.3 we have developed our ARMORs around the base class primitives. A representative list of such primitives is provided in Table 5.3.4.

Base class primitive	**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
compare	Voter ARMOR	Matches two input values and returns a boolean. The match may be based on various criterion
monitor_ARMOR	Host Daemon	Monitor one of the ARMORs installed on the node
send_ready_notif	Execution ARMOR	Send notification to the SM that it is ready to accept application for execution

Table 5.2 Examples of Basic ARMORs

5.4 Host Daemon

The Host Daemon process runs continuously on any node that is participating in the Chameleon environment. This is installed on a node by the FTM in collaboration with an initialization process. The initialization process is a barebone process that first sends a request to the FTM to join in the environment, then opens a TCP channel for accepting the Host

Daemon code, compiles it and installs it on the node. This initialization process persists in the node even after this initial operation is completed. This is reactivated in case the Host Daemon process crashes.

The Host Daemon performs handshake with the Surrogate Manager and installs any ARMOR locally after compiling their source code. It maintains the following data structure.

```
typedef struct {
struct Chameleon_type ag_type; // Type of the ARMOR, e.g. Surrogate Manager, Voter ARMOR, etc.
struct ARMOR_Id       ag_id;   // ID of the ARMOR installed on the host
struct ARMOR_Id       mgr_id;  // ID of the manager of this ARMOR
struct Location       mgr_loc; // Location of the manager of this ARMOR
String                app;     // Name of the application (if any) the ARMOR is monitoring
} Host_Daemon_db_elem;

typedef struct {
 Host_Daemon_db_elem el[10];
} Host_Daemon_db;
```

When a local ARMOR wishes to communicate with a remote ARMOR, it performs the following actions.

(1) Looks up the location of the destination ARMOR. This it does by querying the manager of the local ARMOR. It caches this information for subsequent retrieval.

(2) Creates a header for the message (which includes fields like the ID of the source ARMOR, etc.).

(3) Computes the CRC on the message + header.

(4) Sends it to the destination node.

This is schematically shown in figure 5.7

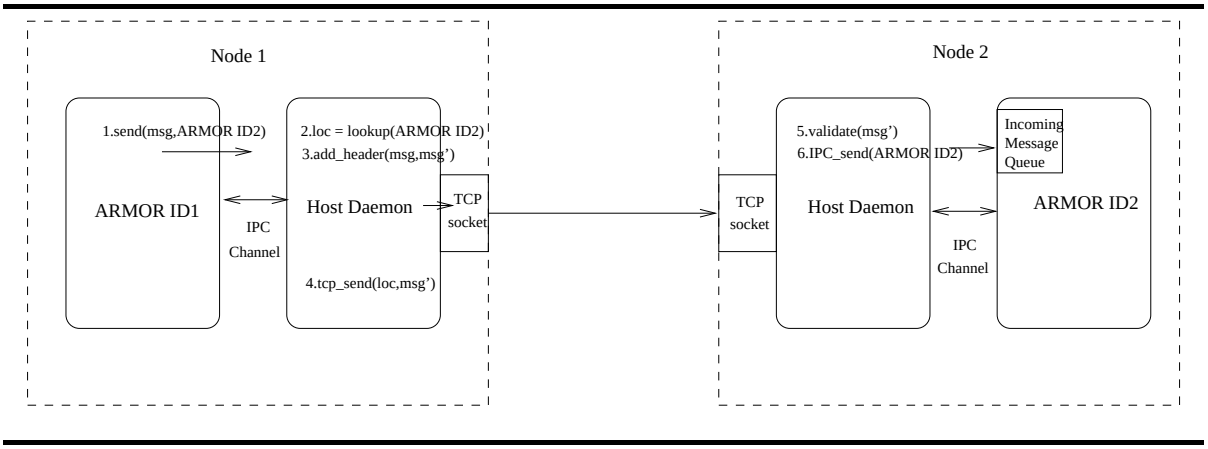

Figure 5.7 ARMOR Communication in Chameleon

CHAPTER 6

EXPERIMENTAL 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 - Sun OS, 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.

(1) Managers

  (a) FTM

  (b) Surrogate Managers:

    (i) Single node execution offering baseline reliability

    (ii) Dual execution where the first result is accepted

    (iii) Dual execution where the two results are compared and only a match is accepted

    (iv) TMR execution

    (v) Quad execution

(2) Daemons : For Sun OS, Solaris, HP and Windows NT

(3) Common ARMORs:

  (a) Execution ARMOR

(b) Voter ARMOR

(c) Checkpoint ARMOR

(d) Heartbeat ARMOR

(e) Initialization ARMOR

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 apriori 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 sub-part of the application is 33.5k. It employs the simple matrix multiplication algorithm [1], 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 combined result matrices 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 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. 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

[1] If $C_{m,n} = A_{m,z} * B_{z,n}$ then $c_{i,j} = \sum_{i=1}^{z} a_{i,k} * b_{k,j}$ where A and B are input matrices and C is the resulting matrix. The distribution is such that each task computes one horizontal strip of the result matrix consisting of half the rows.

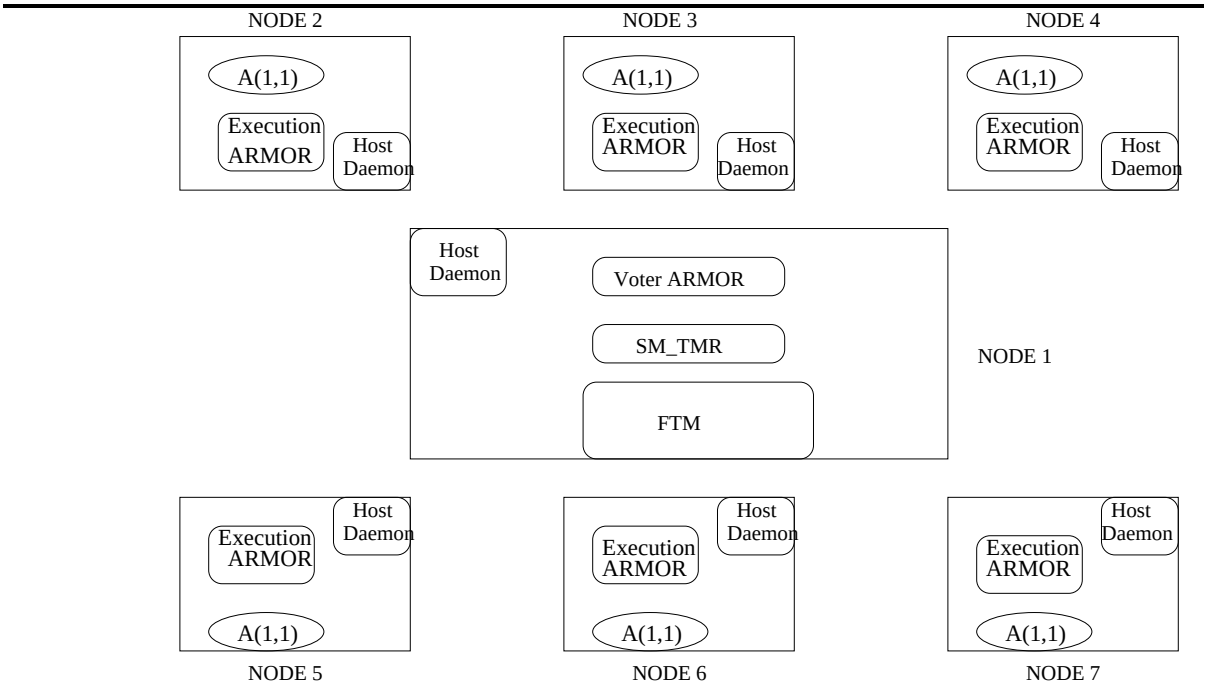

Figure 6.1 Execution Configuration

of a computationally intensive task of a factorial calculation which is executed in machines participating 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. In addition to the measurements on the Chameleon environment for the matrix multiplication application running in a TMR mode with checkpointing, we also present times for some basic tasks in the environment, like time for detection of node failure, and time for recovery from a failed node. A schematic representation of the configuration in which the application is executing and the distribution of the Chameleon ARMORS is shown in figure 6.1.

Since different FTES reuse the same common ARMORs and the basic building blocks, we have been able to scale our environment with respect to the configurations supported quite eas-

ily. 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 parallization of the application, we scale up the number of machines on which we run Chameleon. Thus, in the above configuration, we are using 7 machines. If the application were to be parallelized into 4 tasks in place of 2, 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. We have ported the environment to Sun OS, Solaris, HP-UX and Windows NT (a subset of ARMORs).

6.1.1 Concurrent Application Execution

The environment can handle multiple applications submitted by the user from different hosts. If there are not enough number of hosts in the environment to use separate nodes for each fault-tolerant execution mode, then multiple applications, potentially running in different configurations, may coexist on the same node. All the operations in the environment are tagged using an *application index* which is simply a monotonically increasing integer value. Using this the operations are associated with a specific application and hence we can have monitoring by the ARMORs on a per application basis.

6.1.2 User Specification

We have a text-based user interface through which the user submits his application for execution in the environment, and also provides specifications about the application. Minimally, he provides:

- The number of tasks in the application, the names of each of the tasks, their command line parameters, and their path names.
- The file name into which the output of the tasks will go.
- A timeout estimate value within which if the application does not respond then it is taken to have failed.
- A specification of the reliability desired. For now, it is an integer value which is mapped to one of the supported execution strategies.
- Specification about the checkpointing to be used, i.e. whether rollback recovery option is desired or not, if it is, then, whether the application is checkpoint-aware and itself makes calls to take checkpoint at the appropriate places, or whether checkpoint needs to be taken using a default timer interval.

There may be other hints provided by the user. For example, a list of the platforms for which the application is known to run. This cab be valuable for the FTM, when it decides to migrate the application to another node.

6.1.3 Graphical Interface

This was not a keen concern in the initial prototype. A rudimentary Graphical User Interface, written in Java, is provided. This is displayed at the FTM site. It displays configuration information about the machines which are in the environment. The information is both of the static sort (machine configuration like IP address, memory availability) as well as of the dynamic sort (the number of ARMORs existing on a particular node). The interface allows selection of the machines on which a particular application may run (possibly by a system administrator).

It also provides a default selection strategy according to the load on the machines, the load being estimated by the number of ARMORs on the node. Sample snap shots of the interface are provided in fig. 6.2 The sample shot shows that there are five machines currently configured to run in the environment. Note that it may include machines from different domains. The bottom left window shows information about the machine *monn.crhc.uiuc.edu*. The bottom right window shows the input window for a particular application *mm1.c* that is to be executed in the Quad mode.

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. Therefore we have conducted direct measurements, in the prototype implementation of Chameleon, to obtain the overheads 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 the development, and 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 out over six machines with processor speeds from 140 MHz to 200 MHz.

Figure 6.2 Graphical Interface Snapshot

6.2.1 Time to Launch Basic ARMORs

To gather sense of the time overhead which is involved in launching different components of the Chameleon infrastructure we conducted corresponding measurements which are provided in Table 6.2.1.

Installing the execution ARMOR	71 msec
Setting up a node to participate in Chameleon	2245 msec
Installing the Host Daemon	40 msec
Collecting information about the platform through the Initialization ARMOR	2205 msec

Table 6.1 Time to Lauch Some Basic ARMORs

6.2.2 Overhead in 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 the node (or host daemon) and the 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 requests and echoes) (from the FTM): 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: (1) a fault free execution without checkpointing and (2) 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 6.2.2 as a function of the background workload. The workload varies from 0 to 3 additional processes per node (each process executes a factorial computation program).

Number of workload processes	**Time for standalone execution(sec)**	**Time for fault-free execution in Chameleon(sec)(Overhead %)**	**Time for execution in Chameleon with fault injection & recovery (sec)(Over head %)**
0	38.01	46.88 (23.3%)	61.17 (30.5%)
1	49.36	51.34 (4.0%)	70.53 (37.4%)
2	54.87	57.26 (4.4%)	95.12 (66.1%)
3	73.52	77.20 (5.0%)	96.65 (21.2%)

Table 6.2 Time Overhead in Application Execution

Specific comments to execution times given in Table 6.2.2 are provided below:

(1) The execution times do not include the time for compiling the two parts of the application (scenario where we are exeuting 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 voting ARMOR), time to vote upon results from the application and 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, modify the application

(i.e., to incorporate the appropriate function call for invoking checkpointing) and link the checkpoint ARMOR with the execution ARMOR in the node.

(4) The execution time involves also the time spent on setting checkpoints (the overhead for each checkpointing operation is 70 ms, and the frequency of checkpointing is 1.5 s).

(5) The execution time includes the time for error detection and recovery from the last checkpoint.

For the readings presented in Table 6.2.2 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% as compared to the fault-free execution in Chameleon. A higher overhead of 66%, is measured for the execution while 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 sub- operations and are given in Table 6.2.2. 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.[2]

[2]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 μsec); SM resends ARMOR (71 msec); Execution ARMOR restarts application from checkpoint (1.5 sec = Interval between checkpoints).

Entity to be recovered	Recovery Time (msec)
Execution ARMOR	2499
Host Daemon	1921
Surrogate Manager	1928
Voter	1581

Table 6.3 Recovery Times

Finally, we present some preliminary results from the execution of a simple loop application in the duplicated mode with the FTM running on a Windows NT machine (intel1), 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) 1462 ms (on wolf); 1717 ms (on monn) Time to send the application to the exec ARMOR and get the results back: Average (taken from six measurements) 1056 ms (on wolf); 1059 ms (on monn)

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

The thesis presents the design of the Chameleon system and an initial prototype implementation of the system. The design is prompted by the goals of making the entities simple and hierarchical. Also the Chameleon ARMORs should lend themselves to easy customization to support the adaptivity with respect to the reliability requirements. We propose a hierarchy of Chameleon entities which cooperate among themselves to set up the environment to run off-the-shelf applications in a fault-tolerant mode. We also provide for monitoring of each of the entities, and in some cases, multiple detection paths. Through the implementation, we have shown proof of concept for the dynamic environment. Some simple detection and recovery mechanisms have been implemented. Preliminary results have shown the feasibility of supporting varying reliability needs of an application in the same environment. Some measurements have also been provided to get an insight into the costs associated with Chameleon operations.

The scope of the Chameleon system is very broad. It is conceivable that it will reach its maturity and be able to substantially meet all its design goals after a sutained and prolonged research effort. There are therefore lot of interesting issues to be looked into. The entire issue of runtime reconfigurability of the entities is yet to be investigated. Further automation of the processes of ARMOR manufacturing and reengineering is required. The scope of applications supported in the environment will have to be broadened to include substantially off-the-shelf distributed applications, which do intensive communication and share state. This would pose

several challenges as to how to interpose the Chameleon software layer between such an application and the runtime system in a transparent manner. This would most possibly require some aspects of Chameleon functionalities to be moved to the kernel level where Chameleon entities would have more control on the application execution.

The environment's fault detection capabilities will have to be strengthened to optionally include techniques provided by the application, like assertion checks. The coverage of existing strategies needs to be measured. The behavior of the environment needs to be validated under more stressful fault injection, for example, causing the operating system of a node to misbehave. The checkpointing scheme can be made more intelligent, to store the entire memory image, and do that in an incremental fashion. Also, for soft real-time systems, the crossover point between roll-forward and roll-back recovery techniques would be interesting to look at.

Finally, it needs to be mentioned that Chameleon is in an active phase of development. At the time of writing, the design is going through some overhaul to better modularise the components to allow efficient reconfiguration. The development work is also underway, with features like common ARMOR recovery being incorporated currently. The environment is thus in its infancy. It is hoped that the research will bring up lots of issues of interest to the fault-tolerant community and be able to give reasonable solutions to the problem of providing adaptive reliability in contemporary networked systems.

REFERENCES

[1] J. H. Wensley, "SIFT - Software Implemented Fault Tolerance," in *Proceedings of AIPS*, vol. 41, pp. 243–253, 1971.

[2] D. Powell, "Distributed Fault Tolerance - Lessons Learnt from Delta-4," Tech. Rep. 93192, LAAS-CNRS, 31077 Toulouse France, December 1993.

[3] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM*, pp. 57–78, February 1991.

[4] F. Cristian, "Synchronous and Asynchronous Group Communication," *Communications of the ACM*, pp. 88–97, April 1996.

[5] L. Lamport, "Using Time instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Transactions on Programming Languages and Systems*, pp. 254–280, April 1984.

[6] T.A. Joseph. K.P. Birman, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems*, pp. 47–76, February 1987.

[7] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the Impossibility of Group Membership," *Proceedings of the ACM Symposium on Principles of Distributed Computing*, May 1996.

[8] M. Reiter, "Distributing Trust with the Rampart Toolkit," *Communications of the ACM*, pp. 71–75, April 1996.

[9] L. Moser, P. Melliar-Smith, D. Agarwala, R. Budhia, and C. Langley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System," *Communications of the ACM*, pp. 54–63, April 1996.

[10] D. D. ad D. Malkhi, "The Transis Approach to High Availability Cluster Communication," *Communications of the ACM*, pp. 64–70, April 1996.

[11] D. Cheriton and W. Zwaenwpoel, "Distributed Process Groups in the V Kernel," *ACM Transactions on Computer Systems*, pp. 77–107, May 1985.

[12] K. Birman, "The Process Group Approach to Reliable Distributed Computing," *Communications of the ACM*, pp. 37–53, December 1993.

[13] R. van Renesse, K. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Communications of the ACM*, pp. 76–83, April 1996.

[14] K. Birman and R. van Renesse, "Reliable Distributed Computing with the Isis Toolkit," *IEEE Computer Society Press*, 1994.

[15] M. Kalyanakrishnan and S. Bagchi, "Group Communication Protocols: Survey and Measurements," Tech. Rep. Project Report for CS436, Fall 1997, Center for Reliable and High Performance Computing,
University of Illinois at Urbana-Champaign, 1308 W. Main Street, Urbana, IL 61801., December 1997.

[16] S. Maffeis, "Piranha: A CORBA Tool for High Availability," *IEEE Computer*, vol. 30, pp. 59–66, April 1997.

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

[18] "The Ultra Enterprise Cluster Ha 1.3 Architecture," February 1998. URL - http://www.sun.com/clusters/ha/arch_wp/.

[19] Y. Huang and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," *Proceedings of the Fault Tolerant Computer Symposium 23*, pp. 2–9, 1993.

[20] J.-C. Fabre and T. Perennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS approach," *IEEE Transactions on Computers*, pp. 78–95, January 1998.

[21] G. Agha and D. Sturman, "A Methodology for Adapting to Patterns of Faults," in *Foundation of Ultradependability* (G. Koob, ed.), vol. 1, Kluwer Academic Press, 1994.

[22] "MPICH - A Portable Implementation of MPI." URL - http://www.mcs.anl.gov/mpi/mpich.