AN ANALYSIS OF FAILURE HANDLING IN CHAMELEON, A FRAMEWORK FOR SUPPORTING COST-EFFECTIVE FAULT TOLERANT SERVICES

BY

ERIK EDWARD HAAKENSON

B.S., Rensselaer Polytechnic Institute, 1995

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 like to thank Zbigniew Kalbarczyk and Saurabh Bagchi, for help with providing ideas for the simulation and information about Chameleon. I would also like to thank Keith Whisnant for playing a major part in the development of the simulation. Finally, I would like to thank Professor Ravi Iyer, my research advisor, who was very helpful and provided encouragement, guidance, and advice when I needed them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The desire for low–cost reliable computing is increasing. Most current fault tolerant computing solutions are not very flexible, i.e., they cannot adapt to reliability requirements of newly emerging applications in business, commerce, and manufacturing. It is important that users have a flexible, reliable platform to support both critical and noncritical applications.

Chameleon [9], currently under development at the Center for Reliable and High–Performance Computing at the University of Illinois, is a software framework for supporting cost–effective, adaptable, networked fault tolerant service. Because of a desire for efficiency and adaptability, the Chameleon architecture is intended to support heterogeneity and scalability. Scalability here implies physical scalability, resource scalability, and fault tolerance scalability.

During the design of such complex systems as Chameleon, there is a need to validate the capabilities and measure the performance of the system. This can be done through analytical methods, experimentation, or simulation. Once a functional architecture is defined, simulation is often the most viable of these options. No working version of the system is necessary, and it allows for a more complex model of the system than mathematical analysis.

In the case of Chameleon, we are interested in analyzing the effectiveness of different types of fault detection and recovery strategies. We are also interested in measuring the overhead incurred by the fault detection and recovery mechanisms. These measurements will allow decisions to be made about which recovery strategies should be used in the actual implementation of Chameleon.

The goal of this thesis is to give a detailed description of the efforts to simulate fault injection, detection, and recovery in Chameleon, and of the results obtained from this simulation. This thesis is divided into seven chapters. The second chapter discusses related work in the area of distributed and reliable computing. The third chapter gives a general overview of Chameleon and its components. The fourth chapter describes the simulation. Chapter 5 gives a detailed description of the fault injection, detection, and recovery strategies that have been simulated. The experimental results obtained from the simulation are described the sixth chapter, and the conclusion is given in the final chapter.

CHAPTER 2

RELATED WORK

Current approaches to designing reliable networked computing environments from unreliable components are based primarily on taking advantage of distributed groups of cooperating processes. Most of these designs require a specialized, complex software layer that must be installed on each participating computation node. Several of these systems are focused on providing a software environment designed to handle distributed applications. Several of these approaches are discussed in this chapter.[1]

Isis [2] provides tools for managing and programming with process groups. Using these tools allows a programmer to construct group–based software that provides reliability through explicit replication of code and data.

Transis [6] is a multicast communication layer that enables the creation and execution of fault tolerant distributed applications in a networked environment. It supports reliable group communication for high–availability applications. Transis allows partitionable operation with the ability to reliably merge components when recovering.

Horus [18] also uses the group communication paradigm. It provides a framework for designing distributed applications at a minimal cost. The Horus tool can be used to aid in the construction of reliable services. It is a newer generation of the Isis toolkit.

[1]Cristian [4] provides more detail about the concepts behind distributed fault tolerant systems. Birman [1] discusses group communications and numerous existing fault tolerant distributed systems in more detail.

The systems outlined above are primarily concerned with the group processing paradigm rather than being particularly geared toward fault tolerance. A few systems described below did place a primary emphasis on fault tolerant and/or highly available computing.

Delta–4 [13] was one of the earlier efforts to build a dependable distributed system. It used an open architecture in which a trusted module was loaded on each participating host to execute a multipoint communication protocol. The protocol was used to coordinate process groups, process errors, and perform fault treatment. Delta–4 also required a specialized hardware network adaptor card to guarantee proper fail–silent behavior.

Some aspects of service availability are addressed in the Piranha [10] tool. Piranha acts as a fault tolerant process manager, exploiting the dynamic replication of objects to achieve high availability. It is designed to be a CORBA–based application–restart service and monitor. Piranha addresses needs for heterogeneity, interoperability, extensibility, and availability by making use of CORBA's Interface Description Language.

The Wolfpack [11] system from Microsoft® provides clustering extensions to Windows NT® for improving service availability and system scalability. Issues intended to be addressed in future versions of Wolfpack include distributed applications, higher performance interconnects, distributed storage, and load balancing.

At Sun Microsystems, work has been done on Ultra Enterprise Clusters [16], designed to provide highly available data services. The Ultra Enterprise Cluster High Availability 1.3 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 a software or hardware failure.

ServerNet [8] from Tandem Computers is a system area network designed to support reliable, efficient communications. It provides a combination hardware/software layer on which fault tolerant systems can be built. ServerNet is flexible in that the routers in the system can be configured in several different topologies. Error detection and recovery are also provided in the form of checksums on messages and an access validation and translation table for memory requests.

The systems described here that explicitly address fault detection and recovery each require a specialized and complex software layer and, in some cases, additional hardware. Also, many of the systems described above provide an environment for constructing distributed applications. Chameleon, on the other hand, explicitly provides fault tolerance through a wide range of error detection and recovery mechanisms. Not all of these systems have such explicit mechanisms, and many rely only on timeouts.

CHAPTER 3

CHAMELEON FRAMEWORK

3.1 Overview

Chameleon (Figure 3.1) is a network–based infrastructure with the capability of adapting to application–specific availability requirements. Primary issues addressed in designing Chameleon include efficient and rapid error detection and recovery techniques which provide a basis for implementing fault tolerance strategies required by each user application.

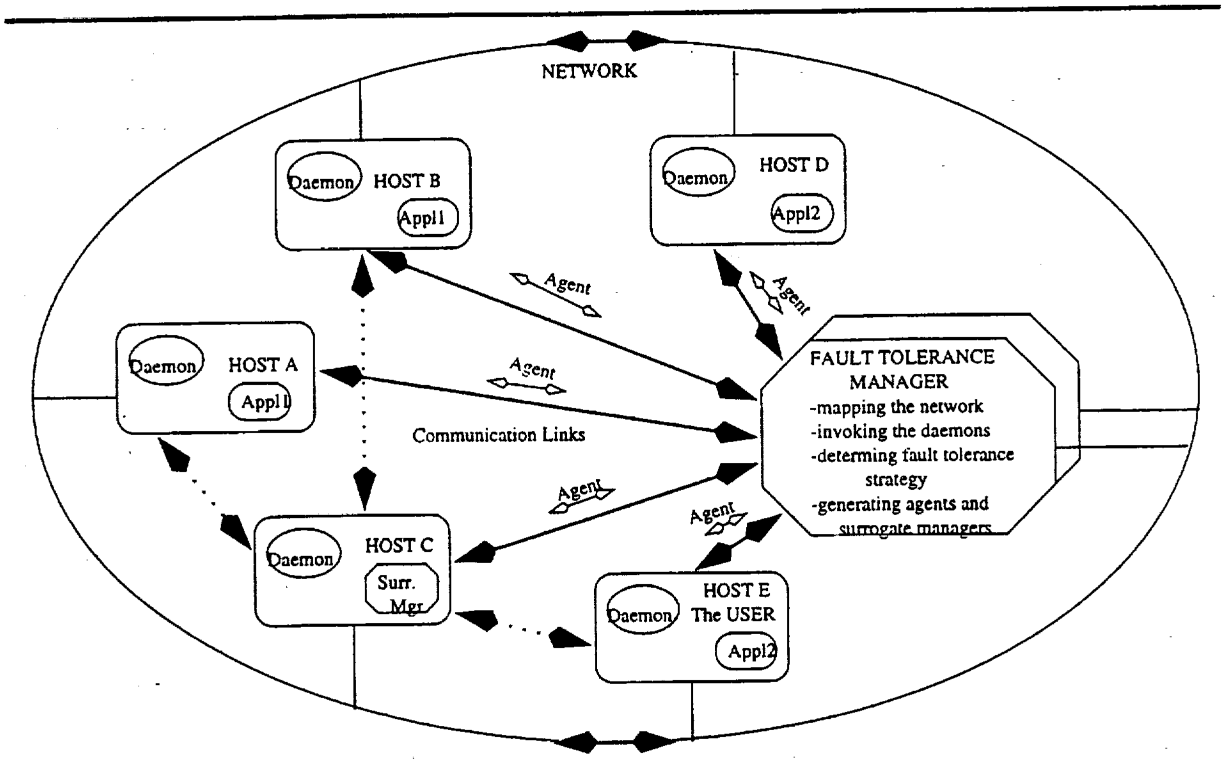

Figure 3.1 Chameleon: A reliable, networked computing environment (this figure is taken from Iyer, et. al. [9]).

To achieve these goals, Chameleon uses several specialized software components. These components include: (1) Fault Tolerance Manager (FTM), a specialized independent and intelligent entity capable of establishing an appropriate fault tolerance strategy complying to the required level of dependability for a given user request, (2) Reliable, Mobile, and Intelligent Agents capable of migrating through the networked environment and operating independently on behalf of the FTM according to built–in specifications and instructions, (3) Surrogate Managers operating as pseudomanagers for particular applications, capable supporting proper communications with the agents, which guard against faulty behavior of the application's execution on remote hosts, (4) Host Daemons residing on each node (throughout this thesis, the words *node* and *host* will be used interchangeably to refer to a machine participating in the reliable networked environment) and responsible for handshaking with the agents and managers and monitoring their behavior, and (5) Software Libraries providing basic building blocks to create or re–engineer agents. The goal of the system is to prevent any single point of failure from compromising the entire system.

The Chameleon implementation does not use a specialized language framework, rather it is based on widely available scripting languages, such as TCL, and high–level programming languages, such as C++. The goal is to provide a relatively thin software layer, which must be present in each machine in the structure. It should be noted that nothing prevents using a framework, such as CORBA (Common Object Request Broker Architecture) [12], for implementing some of the features of Chameleon. It is believed, however, that an implementation with CORBA, while providing for easier interoperability between processes executing on different machines in a heterogeneous environment, will increase Chameleon's complexity, at least in terms of the software that must be pre–installed on each node in the system. Chameleon attempts to maintain simplicity by allowing the user to develop the application in a regular fashion and to execute it with the user's desired level of dependability.

As described in Chapter 2, most of the current approaches used in distributed computations require a specialized, complex software layer that must be installed in each computation node, e.g., sophisticated and complex underlying protocols for supporting a group membership and atomic broadcast. Because a primary objective of developing these systems is to provide a software environment for executing distributed applications, the service availability issue is not often considered to be critical. Consequently, there is no dedicated mechanism for error detection, and the fault tolerance is somewhat a side effect of the use of the group communication approach. The system usually relies on error detection that is based on capturing the timeout in a response from one of processes in the group.

3.2 Components of Chameleon

The five main components of Chameleon, as stated above, are described in more detail in this section. Each component's responsibilities, modes of operation, and communication patterns are discussed. Table 3.1 gives a brief summary of the main components.

3.2.1 Fault Tolerance Manager

The Fault Tolerance Manager (FTM) is the component of Chameleon that is responsible for interfacing between the user and the system internals. The FTM has four main functions: (1) mapping the network, i.e., identifying the network configuration and collecting information about the nodes in the system. The FTM maintains an internal data structure that contains this data and is updated when nodes are added to or removed from the system, (2) invoking a

Component	Task	Recovery
Fault Tolerance Manager (FTM)	Oversees execution environment	Backup FTM takes over
Agent	Implements specific techniques providing application–required dependability	Host daemon notifies surrogate manager
Surrogate Manager	Oversees execution of a particular application	Host daemon notifies FTM
Host Daemon	Provides communication gateway to agents and makes resources at a host available to the Chameleon environment	Heartbeat agent notifies FTM
Agent Library	Provides predefined agents and agent building blocks	N/A

Table 3.1 Chameleon system components, their tasks, and recovery mechanisms.

daemon process on each node in the network to support communication with the FTM, (3) collecting information about applications from users, and (4) determining fault tolerance strategies to allow the application to execute at the required level of dependability. The FTM's decision may be based on a history of failures in the system as well as on application requirements. Detection and recovery techniques are taken from the agent libraries to construct the agents necessary to implement the chosen fault tolerance strategy.

In the initialization phase, the FTM collects information about the system configuration and characteristics of individual nodes, such as type of architecture, operating system, size of the RAM, etc. Initialization agents are sent to the hosts to obtain this data and to install the host daemons on participating machines. After successful initialization, Chameleon is ready to accept user requests.

When a user request arrives, the FTM designates a query agent to acquire the necessary information on the application specifics, such as the required availability level, needed system resources, type of results, etc. Based on information collected about the application, the FTM can identify the necessary fault tolerance strategy and can designate a set of agents to initiate and monitor the application. Creation of agents is performed according to a predefined procedure that uses two software libraries: (1) a library of building blocks and (2) a library of agents. The FTM may create new agents from the basic building blocks or may re-engineer already existing agents to extend their functions. Agents designated to support the application's execution migrate through the network to the selected nodes and initiate the application's execution. One of the designated agents is resident in FTM and is responsible for supporting proper communications with the agents that monitor the application on the remote hosts. To ensure a rapid reaction to the application's failures, the application is watched by the agent that evoked it. The agent communicates to the FTM any detectable application misbehavior. As the agent itself may fail, it is watched by the host daemon, which is capable of notifying the FTM about agent failures. The FTM can regenerate a new agent either to complete or to restart the application (if the application failed). It should be emphasized that agents, once generated, can act autonomously, and the FTM is free to serve other user requests. In order to detect node failures, the FTM uses heartbeat messages, which are sent with a predefined frequency. In the case of node failure, the application(s) executed on the node are migrated to other available nodes. To operate reliably, the FTM must be resilient to errors. Consequently it must provide a sufficient level of redundancy to cope with errors. To handle FTM failures, a backup FTM is used. The role of the backup FTM is to periodically send a heartbeat to the FTM to determine whether it is alive. In the case of an FTM failure, the backup FTM has the capability to act as the FTM until the primary FTM can be recovered.

3.2.2 Reliable, Mobile Agents

The agents in Chameleon are designed as fault resilient carriers of information to and from the FTM and other managing entities. They are designated by their manager to perform the actions and operations needed for successful completion of an application, while adhering to the user's needs for dependability. Each agent is designed to be sufficiently intelligent to execute specified functions in an autonomous fashion. This autonomous nature of the agents aids in offloading much of the processing from the FTM. This decreasing of the burden on the FTM enables the FTM to concentrate on its primary tasks as described above. The primary characteristics of agents are (1) mobility, (2) reliability, and (3) scalability.

Mobility: Agents migrate through Chameleon's network in order to accomplish their tasks as defined by their manager. Well-known communication protocols such as TCP/IP can be used to support this mobility.

Reliability: It is imperative that agents are resilient to network and node failures. To achieve this, the agent code in the existing agent libraries is tested rigorously against erroneous execution. It is also important to ensure that a failure in the agent does not cause a crash of the application it was in charge of executing and that the agent's crash does not propagate out of the node on which the agent currently resides. To prevent such behavior, agents are watched by host daemons. The host daemon is notified which agents it will have to monitor for possible crashes when each agent is installed on its host. If an agent fails, the daemon notifies the agent's manager. To protect the agent from corruption in the network it and its transmissions are guarded with a checksum.

Scalability: It is simple to create or re-engineer agents using elementary building blocks or already existing agents. Chameleon provides a unified, general framework for creating new agents or extending functions of already existing agents, e.g., the user might provide an application–specific detection mechanism to be incorporated into an agent. Two basic software libraries support this approach: (1) a library of building primitives and (2) a library of agents. These libraries are discussed in more detail below.

3.2.3 Surrogate Managers

A surrogate manager is spawned by the FTM after the required fault tolerance configuration has been determined. It is created using a procedure similar to the one employed for creating agents. Each surrogate manager is associated with an application (or possibly several applications may share the same surrogate manager). The surrogate manager can be seen either as a "super agent" or pseudomanager. It is capable of acting as a regular agent, e.g., it can travel through the network to the designated nodes, and it is recognized and monitored by the host daemon. At the same time, it is capable of operating as a manager, i.e., it supervises agents designated to control the application, and it can regenerate agents that failed during operation. To facilitate autonomous and independent operation of the surrogate manager, a portion of the system information maintained by the FTM is also kept with the surrogate manager. By this means, the application can survive even in the case of FTM failure. The system information that must be available to the surrogate manager includes full specification of the application and access to the software libraries used to create and re–engineer the agents.

3.2.4 Host Daemons

The host daemons are entities at each of the hosts which are responsible for handling communication between agents, surrogate managers, and the FTM. The daemon processes are responsible for accepting and installing any agents sent to their host; they interact with the agents to accomplish their task. The daemon processes have the intelligence to recognize the type of agent being sent over, and they have a well-defined handshaking protocol for communicating with each of the agents. The daemon process is also responsible for monitoring the behavior of agents and surrogate managers. When the host daemon detects a malfunctioning agent, it notifies the agent's manager. In the case of an error encountered in a surrogate manager, the FTM is notified directly of the failure. The manager of the malfunctioning component then sends over a clone of the agent or surrogate manager to the host on which the erroneous behavior occurred (alternately, a new host could be chosen for the regenerated agent or manager). The new agent/manager becomes a part of the process to complete the execution of the application in the mode defined by the FTM.

3.2.5 Agent Libraries

The agent libraries are used to construct different types of agents for executing user requests in the required mode of dependability. There are two distinct libraries for agents: the library of building blocks and the library of agents. The agent building-block library contains basic building blocks used to implement the different types of agents required by Chameleon. The library of agents contains already constructed agents ready for use by the FTM.

The library of building blocks contains micro- and macro-operations for supporting application execution in the distributed environment. Agents can be created, modified, and re-engineered using these operations. These operations may include capabilities such as installing a user application, comparing two or more results files, or notifying another component of a specific event (e.g., an application failure).

The library of agents contains hierarchically arranged, already available agents, which have the flexibility of extension: (1) basic agents, (2) agents extended from the basic agents using primitive building blocks, (3) complex agents derived from the combination of existing agents, and (4) user–defined agents from existing or user–defined building blocks.

3.3 Fault Tolerance Strategies

Once the FTM has decided to run an application using a particular fault tolerance strategy, the set of agents and an associated surrogate manager are invoked to set up the environment to support the selected strategy (e.g., triple modular redundant mode). The surrogate manager takes over management of the application from this point, and the agents begin their duty of installing, executing, and monitoring the application. When the application completes, the agents are responsible for notifying the surrogate manager of the final results.

There are five predefined modes of application execution: single machine with no recovery, single machine with recovery, duplicated execution, triple modular redundant execution, and quad execution. Each of these is outlined below.

In *single machine with no recovery* mode, the user application is run on a single node with no special recovery steps taken in the event of failure. This mode is the least reliable of the predefined modes of execution.

Single machine with recovery mode also executes the application on a single system node, but it provides recovery in the event of application failure. This recovery includes restarting the application (possibly from a checkpoint) in the event of abnormal application termination. The monitoring and restarting is performed by a specialized execution agent.

In *duplicated execution* mode, the user application is executed concurrently on two separate machines. When results are obtained from each application, they are compared by a specialized voter agent. If the voter agent finds a discrepancy between the two results, it notifies the surrogate manager that the application failed; otherwise, the application is considered to have completed successfully.

In *triple modular redundant* mode, the application is executed on three separate machines, with the results compared by a voter agent when all execution has completed.

Quad execution mode provides the highest level of dependability of the predefined execution modes. The application is run on four nodes as two sets of duplicated applications. Each set has a voter agent, which compares the results of the two applications. If no discrepancies are found, another voter agent compares the results from these two voters and notifies the surrogate manager of the results.

3.4 An Example Application

This section describes the steps taken to execute a user application in duplicated execution mode. The figures below provide a graphical portrayal of this process. It is assumed that all initialization has completed at the beginning of the application's execution.

Figure 3.2 shows the processing of the user request. The FTM is notified of the request and sends a query agent to the user's machine to obtain information about the request. The query agent is installed on the user's machine and collects information about the application from the user. The query agent then returns with the query results and application code to the FTM.

Figure 3.3 portrays the selection of fault tolerance strategy by the FTM and the installation of the appropriate agents and associated surrogate manager. In this case, the FTM has chosen

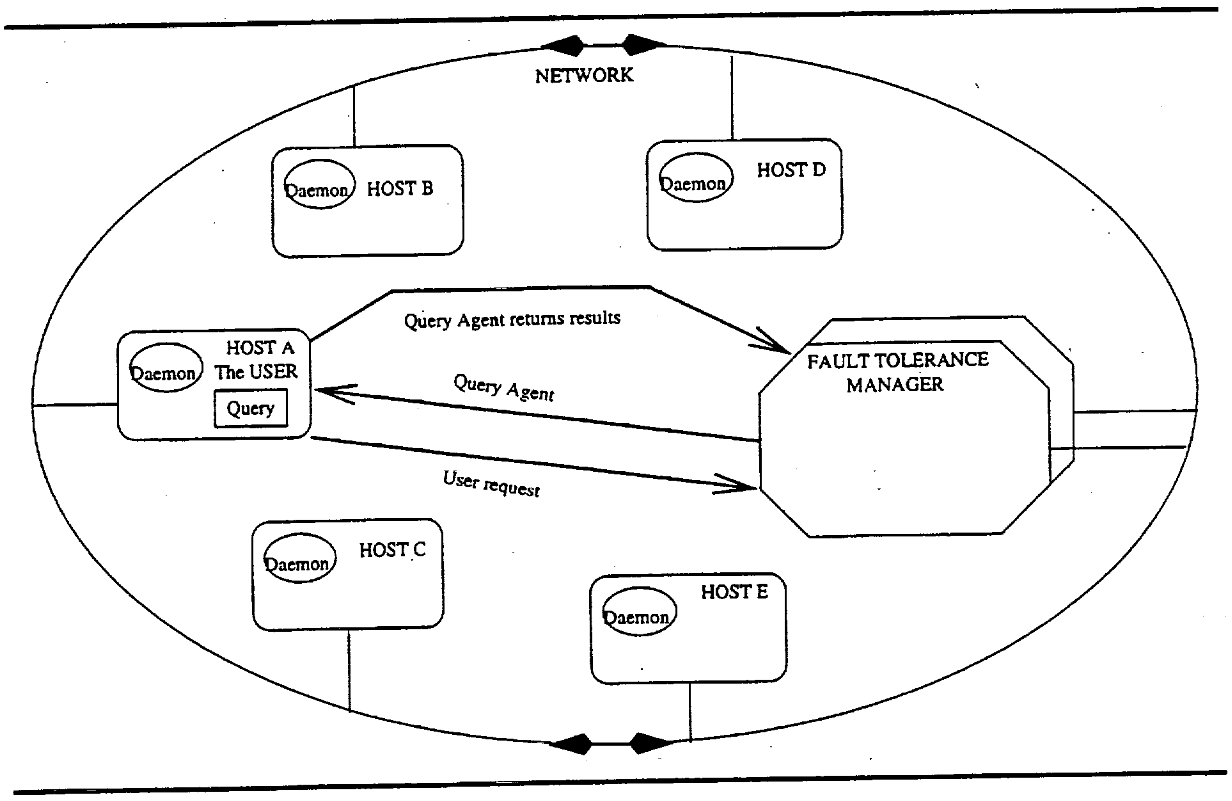

Figure 3.2 User communication and query agent.

to execute the application in duplicated execution mode, as described above. The FTM is responsible for retrieving and/or constructing the proper agents from the agent libraries. Each host daemon receives, installs, and initiates execution of its incoming agent or manager. It is worthwhile to note that the user's host need not be a part of the execution environment.

The application installation and execution is depicted in Figure 3.4. Once the execution agents have been installed, they request the application code from the FTM. After receiving this code, it is compiled by the agent and executed. During the execution, the application is periodically monitored by the agent, which is in turn periodically monitored by the host daemon.

When the applications have completed, the sequence of events shown in Figure 3.5 occurs. The results from each application are sent to the voter agent. Once all results have been

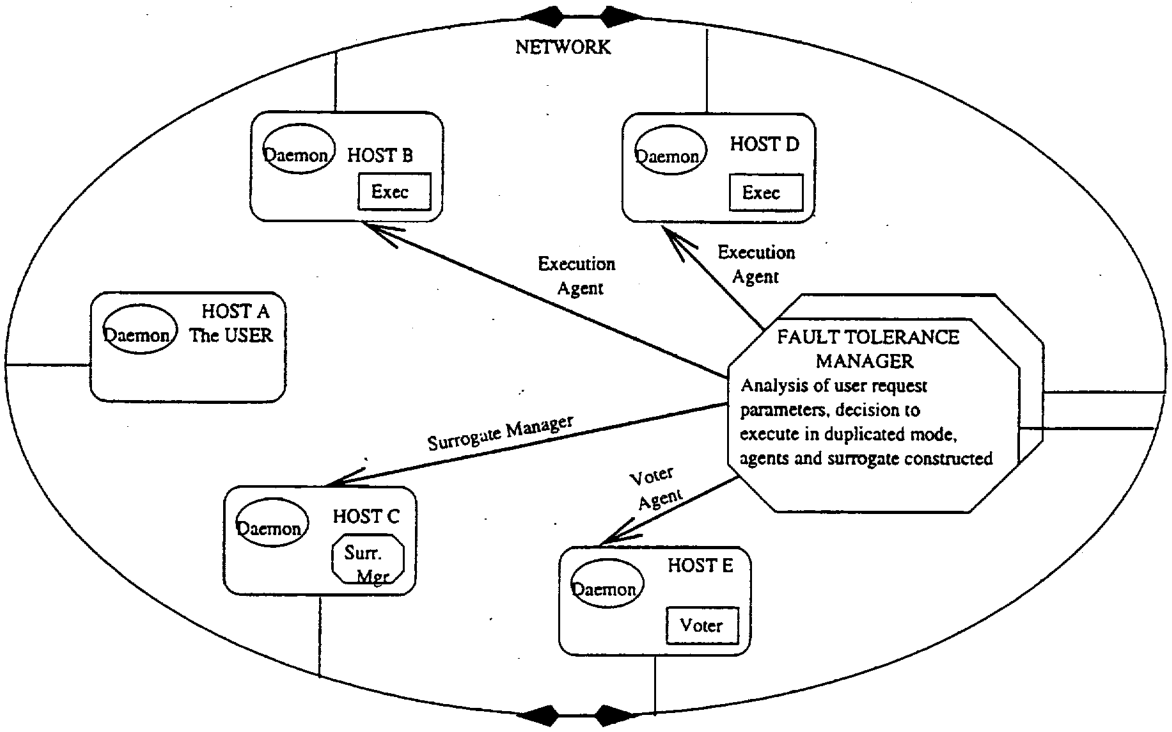

Figure 3.3 Agent and surrogate manager installation.

obtained by the voter agent, a comparison is done. This comparison may check for an exact match, or it may check that the differences lie within an allowable range as specified by the user to the query agent at the beginning of the user request. After a successful comparison, results are sent by the voter agent to the surrogate manager, which in turn sends them to the FTM. When the FTM receives the results of the application, it notifies the user and sends and recursively uninstalls the surrogate manager and its agents. (Recursive uninstallation means that each manager uninstalls those entities directly under its command, e.g., in under duplicated execution, the FTM uninstalls the surrogate manager, and the surrogate manager uninstalls the two execution agents and the voter agent.)

The scenario described above does not show system behavior under failures. A similar scenario detailing the steps taken to detect and recover from a failure is detailed in Chapter 5.

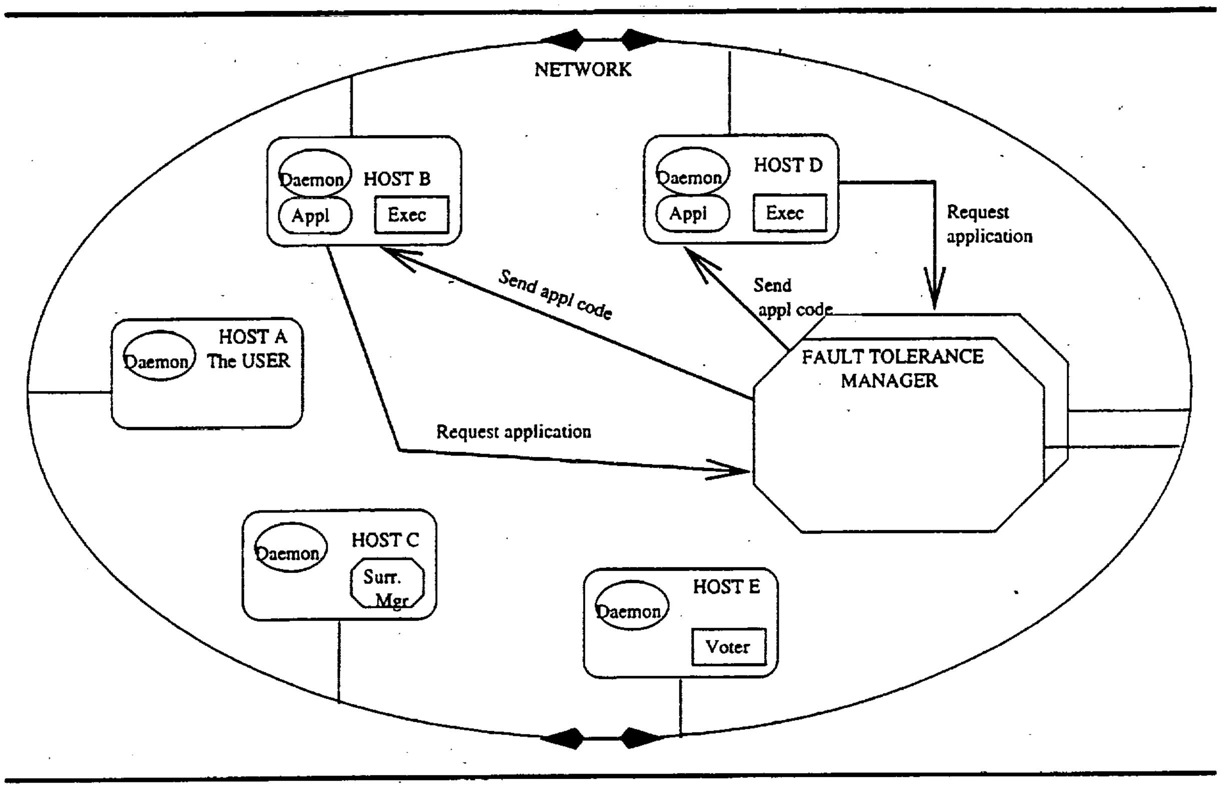

Figure 3.4 Application installation and execution.

3.5 Summary

Chameleon provides an environment for efficient creation and execution of dependable applications. It provides various mechanisms to detect and recover from failures in a dynamically changing networked environment. Chameleon allows for applications with varying dependability requirements to be efficiently executed on the same reliable networked platform using an application–specific fault tolerance strategy for each user–submitted application.

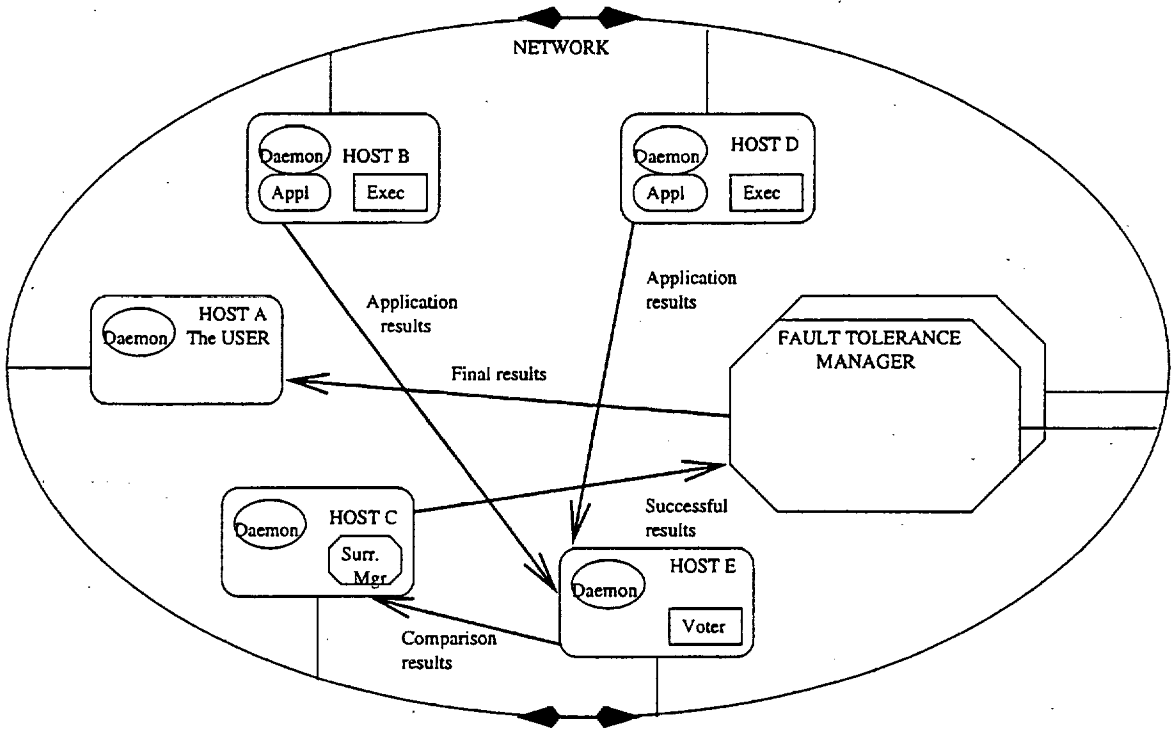

Figure 3.5 Completion of application.

CHAPTER 4

SIMULATION OF CHAMELEON

4.1 Overview

A simulation has been constructed to model the Chameleon system. The simulation models the behavior of the FTM, host daemons, surrogate managers, and agents and their interactions. The goal of the simulation is to obtain information about the effect of fault tolerance detection and recovery strategies in terms of fault coverage and performance degradation. This information can in turn be used to guide the implementation of Chameleon. The simulation was written in DEPEND [7], a functional, process-based simulation tool. DEPEND was chosen particularly for its emphasis on modeling fault tolerant systems.

4.2 DEPEND

DEPEND is a simulation–based CAD environment built on top of CSIM [15] that helps computer systems designers study the behavior of a system in detail. DEPEND is designed to be a joint performability and dependability analysis tool. DEPEND provides an object–oriented C++–based framework which allows for the evaluation of dependable computer systems. The tool provides facilities to model components often found in fault tolerant systems and allows for automated fault injection. By an acceleration technique, DEPEND allows its users to obtain a detailed and statistically valid dependability analysis of a given system.

The methodology of DEPEND is a three-level hierarchy. *Simulation objects* are used to describe models. *Process entities* in each of these simulation objects represent schedulable units. Finally, *simulation constructs* in each of the processes provide for communication, synchronization, and resource allocation.

DEPEND is a hybrid simulation engine, taking advantage of the flexibility of process–based simulation and the speed of event–based simulation. Compiler–based techniques are used to translate from a process–based model to a hybrid process–based/event–driven model to increase simulation speed. Process-based techniques are used because process interaction is generally a better model of system behavior.

DEPEND supplies a library of C++ objects that simulate the functional behavior of components often found in fault tolerant systems. These objects also inject faults, initiate repairs, compile statistics, and generate reports. To use DEPEND, a user writes a control program in C++ with the objects provided by DEPEND. The program is then compiled and linked with DEPEND objects and the run-time environment. The model can then be executed in a simulated parallel environment. In this environment, all objects execute simultaneously to simulate the functional behavior of the architecture.

4.3 Simulated System Description

The system being modeled by this simulation consists of eleven hosts (one host dedicated to the FTM). Each host is simulated as having a single processor with a round–robin scheduling policy with a specified time slice. The hosts are attached to a Myrinet switch [3] through which all communication takes place.

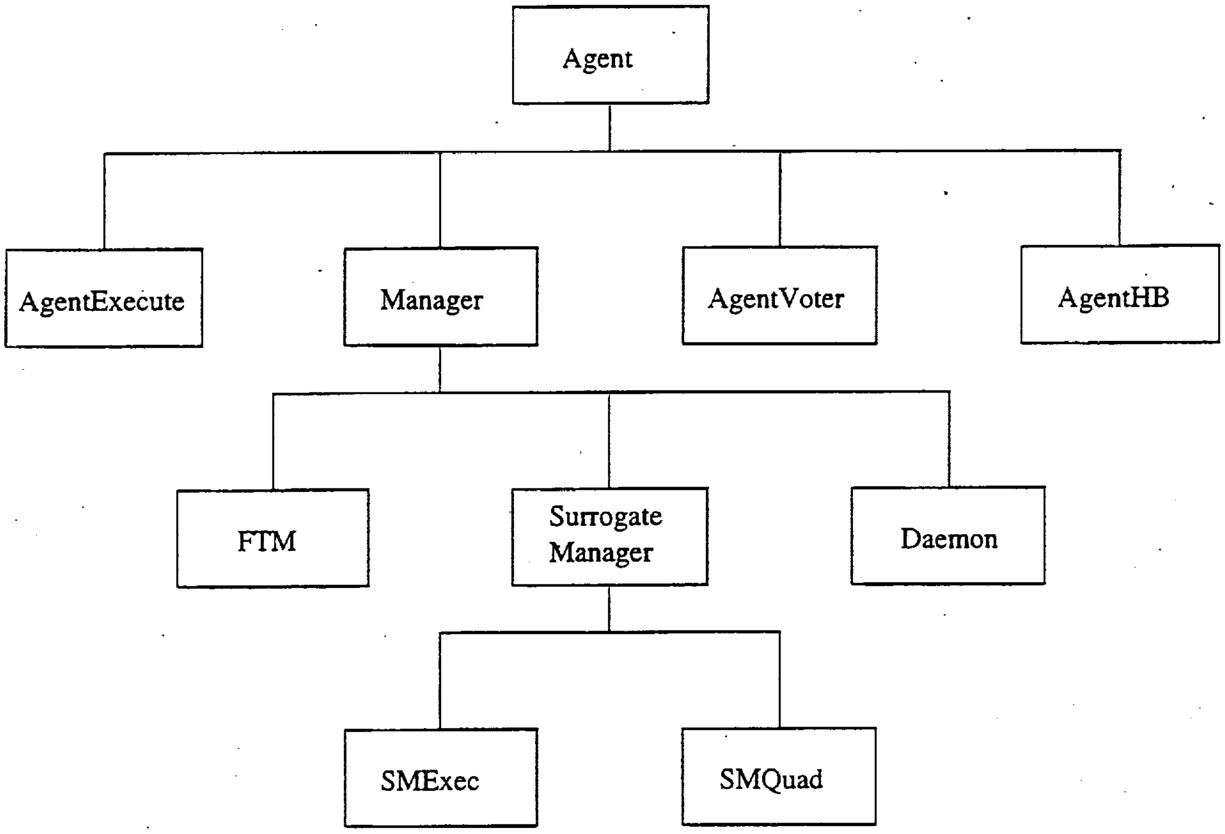

Figure 4.1 Class hierarchy of simulated Chameleon components.

4.4 Data Structures

The major data structures used in the simulation are C++ classes modeling the main components of Chameleon. All major Chameleon components except as the host data structure are derived from the Agent base class. The class hierarchy is depicted in Figure 4.1. The host class, not shown, is derived from the FT_server class in the DEPEND class library. This hierarchical design of the simulation allows for new types of agents or other simulated Chameleon constructs to be integrated into the simulation code relatively easily.

4.4.1 Agent Classes

The Agent base class includes methods common to nearly all the system components, such as a message sending method, a message processing method, an execute method, and fault injection methods.

Three types of agents are modeled in the simulation: the execution agent, the heartbeat agent, and the voter agent. Neither the heartbeat agent nor the voter agent have any additional methods; the methods from the Agent base class are simply specialized for the behavior of the agents. The execution agent has one additional method, which allows it to monitor its application. Each of the inherited methods is also specialized for the execution agent.

4.4.2 Manager Classes

The Manager class is derived from the Agent class. Additional functionality of the Manager class includes the ability to maintain lists of hosts and agents associated with the manager and the ability to install and uninstall agents. Additional message–processing capabilities are also included.

The host daemon and FTM classes are both derived from the Manager base class. Additional methods used by the daemon class include those to monitor for incoming messages, to dispatch messages to the appropriate agents, and to monitor the agents residing on the daemon's host. The FTM's class has an additional method to process incoming user requests, as well as methods to execute the requests using the proper fault tolerance strategy.

4.4.3 Surrogate Manager Classes

The SurrogateManager class is derived from the Manager class. Additional functionality includes the ability to process more types of messages than the Manager base class. Both the replicated surrogate manager class (used for executing applications in duplicated mode and TMR mode) and the quad surrogate manager class (used for executing applications in quad mode) are derived from the base SurrogateManager class. Each one has its inherited methods specialized to perform its designated tasks.

4.5 Simulation Behavior

The simulation need not simulate all behavior of Chameleon. It is only useful to simulate behavior related to the measurements the simulation will provide. Failure to abstract away some system behavior could cause significant performance degradation in the simulation. The initialization procedure (i.e., the handshaking procedure to install a daemon and register a host with the FTM) was not modeled in the simulation, as it only affects the system startup cost and not the execution of user requests. Also, actual execution of code is not simulated. Only the use of a host's processor is simulated. The simulation concentrates on system–level behavior rather than modeling the program counter, caches, etc., on each host.

Only three modes of execution are simulated: duplicated, TMR, and quad mode. These are the modes that require the most overhead and are the most interesting for the results obtained in this thesis. All agents are assumed to exist in the agent library, no user–specified agents have been simulated, and agents are never built from building blocks in the simulation.

CHAPTER 5

FAULT HANDLING IN THE SIMULATION

5.1 Overview

To see how Chameleon handles failures it was necessary to simulate failures to various components during execution of the simulation. There are three main parts to the failure process: fault injection, fault detection, and fault recovery. This chapter provides a detailed description of the injection, detection, and recovery strategies used for each of the simulated components.

Table 5.1 briefly summarizes the detection and recovery process required for each component into which faults are injected in the simulation. This is discussed in more detail below.

5.2 Fault Injection Strategy

Fault injection in the simulation is implemented using the DEPEND fault injector object. The components that may fail are: Hosts, Execution Agents, Voter Agents, Heartbeat Agents, Surrogate Managers, User Applications, and the FTM. Each instantiated object of these types has an internal fault injector, which is started at the time the component begins executing. For each component, faults are assumed to occur according to an exponential distribution.

The fault injection strategies for the different components are very similar. All faults injected are *permanent* and *fail-silent*, with the exception of faults in the FTM, which are *transient* and

Failure	Consequence	Detection By	Recovery
Agent Crash	Agent lost	Resident Host Daemon	1. Agent's manager notified. 2. Manager constructs new agent. 3. Manager installs new agent.
Surrogate Manager Crash	Application running without supervision	Resident Host Daemon	1. FTM notified. 2. FTM constructs new surrogate. 3. FTM installs new surrogate.
Host Crash	All agents on host lost	Heartbeat Agent	1. FTM notified. 2. FTM deregisters host. 3. FTM migrates affected agents it manages to new host. 4. FTM notifies immediate managers of failed host; these managers migrate their agents and recursively notify all subordinate managers.
User Application Crash	Program fails to complete normally; no results produced	Execution Agent	1. Execution agent's manager notified. 2. Restart application.
FTM Crash	Chameleon environment proceeds without supervision	(Transient failure)	N/A

Table 5.1 Simulated fault injections, detections, and recoveries.

fail–silent. Permanent failures are failures that exist indefinitely until some corrective action is taken. Transient failures exist only for short periods of time. A fail–silent fault is one in which the failed component stops communicating with other components rather than continuing to send possibly faulty communications.

A fault injected into a host in this system will cause the host to become unreachable, and all applications executing on that host will terminate and be lost. A fault injected into any agent or manager will cause execution of that component to terminate, and it will be incapable of receiving or sending any sort of communication. A user application that fails will immediately stop executing and will produce no results.

One component into which faults are not injected is the host daemon. The reasoning behind this is that host daemon failures have the same effect as node failures and are handled in the same fashion as node failures. A failed host daemon prevents its host from receiving or sending Chameleon–related communications.

5.3 Fault Detection Strategies

Chameleon has several built–in failure–detection capabilities to cover the different components that may fail. This section describes how each type of failure is detected in the simulation.

5.3.1 Agent

To detect agent failures for each type of agent, the daemon residing on that agent's host periodically polls each agent to see whether it is still alive. This is meant to simulate a process table lookup or something similar. It is simulated by checking a field in the agent class indicating whether the agent is alive. If the daemon determines the agent is no longer alive, it removes the agent from its list of agents to monitor and sends a message to the agent's manager (surrogate manager or FTM) indicating the failure. This polling method is used rather than attempting to capture signals because it can be used to simulate detection of failures like livelock (by checking the process's program counter) as well as abnormal termination failures.

5.3.2 Surrogate Manager

Detection of failures in a surrogate manager is much the same as detection of agent failures. Each surrogate manager is polled periodically by the resident host daemon. When the daemon detects a failure, it sends a notification message to the FTM.

5.3.3 User Application

To detect failures in the user application, the application is periodically monitored by its execution agent in much the same way agents are monitored by the resident host daemons. When a user application has failed, the execution agent sends a message alerting the surrogate manager of the failure.

5.3.4 Host

To detect host failures, a heartbeat agent resides on the same host as the FTM and periodically sends heartbeat messages to all hosts registered with the FTM. Each host is expected to respond to a heartbeat within the defined heartbeat timeout interval. Since a failed host is incapable of communicating, it will not respond to heartbeat messages. Once a heartbeat timeout is detected, the agent stops sending heartbeats to that host and notifies the FTM of the failed host.

5.3.5 FTM

Implementation of failures in the FTM was done quite differently from the other failure detection and recovery mechanisms. Several methods for detecting FTM failures have been conceived, such as running the FTM in triple modular redundant mode, having a backup FTM which is alerted of updates to the primary FTM's data structures, and executing the FTM on a dedicated, highly reliable computation node. None of these was implemented in the simulation. Instead, when a failure occurs in the FTM, detection is assumed to occur after a random amount of time (based on an exponential distribution), and the FTM recovers after that time. Essentially, FTM failures are modeled as transient failures.

This transient failure model is used for the FTM because, at the time the simulation results were taken, the detection and recovery for the FTM in the Chameleon implementation had not yet been established. The transient model should be sufficient to examine the effects to the system when the FTM fails, however that failure may be handled.

5.4 Fault Recovery Strategies

Once a failure has been detected using the methods described above, the next step is to recover from the failure. Chameleon has the capability to recover from each of the faults injected in the simulation. These recovery techniques are described below. There is no subsection for the Fault Tolerance Manager in this section because of the transient nature of its faults.

5.4.1 Execution Agent

When an execution agent fails, its surrogate manager is responsible for its recovery. Once the surrogate manager receives a failure notification from the agent's resident daemon, it uses information it has about the failed agent to reconstruct a new execution agent. The surrogate manager must maintain information about the voter to the execution agent sends its results and the application which the execution agent was monitoring. It then installs this new agent on the same host and sends it the application to be restarted. Once the new execution agent is installed and running, a message is sent to its voter notifying it of the new agent from which it should expect results.

5.4.2 Heartbeat Agent

The FTM is the entity responsible for recovering a failed heartbeat agent. Once the FTM is notified about the failure, it reconstructs a new heartbeat agent. The FTM provides the new heartbeat agent with its list of registered hosts. Once the agent has been reconstructed with this information, it is installed and begins sending heartbeats to the hosts provided to it by the FTM.

5.4.3 Voter Agent

When a voter agent fails, its surrogate manager is responsible for its recovery. Once the surrogate manager receives a failure notification from the agent's resident daemon, it uses information it has about the failed agent to reconstruct a new voter agent. The surrogate manager must maintain information about the entity to which the voter agent was intended to send its results (the surrogate manager itself, or possibly another voter agent) and the agents from which the voter agent expected to receive results. It then installs this new agent on the same host. The destination entity for the new voter's results (in the case that it is not the surrogate manager itself) is then notified of the new source by the surrogate manager.

Once the voter agent has been installed, it sends a message to each of the agents on whose results it is voting, instructing them of the new destination for their results. Upon receiving this message, an agent will send its results, if they have already been computed, as well as updating its destination agent. This allows the voter to recover any results sent during the period between when the fault occurred and recovery completed, or to immediately receive any results the failed voter had already received.

5.4.4 Surrogate Manager

In the case of a surrogate manager's failure, the FTM is responsible for its recovery. Once the FTM has received a notification of the failure, it reconstructs the surrogate manager, providing it with the list of agents the failed manager was overseeing and the expected communications flow between the agents (which agents expect results from which other agents). When the reconstruction is complete, the surrogate manager is installed on the host on that the failed one was running. After installation, the new manager sends a request to its primary voter (the voter which sends its results to the surrogate manager) to send any results it has. As with voter agent recovery, if the primary voter has not computed results yet, it ignores the request.

In the special case that the surrogate manager was in the process of overseeing the installation of its agents when it failed, a new surrogate manager is created from scratch. All agents whose installation had completed under the failed surrogate manager are then uninstalled by the FTM.

5.4.5 User Application

In the case of a user application's failure, the execution agent is in charge of recovery. Once the failure is detected by the execution agent, it sends a message to notify the surrogate manager and restarts the application from the beginning (or from the most recent checkpoint, if one exists). The surrogate manager's notification allows the manager to notify the voter agent in case the voter agent has a specified timeout period, or if the application has failed multiple times, the surrogate manager may decide to migrate it to a different host.

5.4.6 Host

The FTM is the entity responsible for recovery from a host failure. When the FTM is notified of the failure by the heartbeat agent, the failed host is deregistered from the FTM's table of participating nodes. After deregistering the host, a message is sent to each surrogate manager notifying it of the host failure. When a surrogate manager receives such a message, if it determines an agent it is managing was executing on the failed host, the agent is reconstructed and reinstalled on a new host. This reinstallation procedure is similar to the one described above for either an execution agent or a voter agent.

The FTM may recognize that a surrogate manager was residing on the failed host. In that case, it restores the surrogate manager on a different host using much the same method as described above for recovery from surrogate manager failures.

5.5 An Example Failure Scenario

This section outlines an example of a failure in duplicated execution mode and the steps taken to detect and recover from the failure. The failure shown here is a node failure that affects the voter agent. The steps from detection of the failed host to completion of the application are detailed below and in Figures 5.1 through 5.3.

Figure 5.1 shows the detection of the host failure by the heartbeat agent. After not receiving a heartbeat from the host within the specified timeout period, the heartbeat agent assumes the unresponsive node has failed. The FTM is promptly notified of the failure and removes the host from its system configuration file.

In Figure 5.2, restarting of the agent on the failed node is portrayed. First, the FTM searches its list of surrogate managers to determine whether the failed host was home to a surrogate

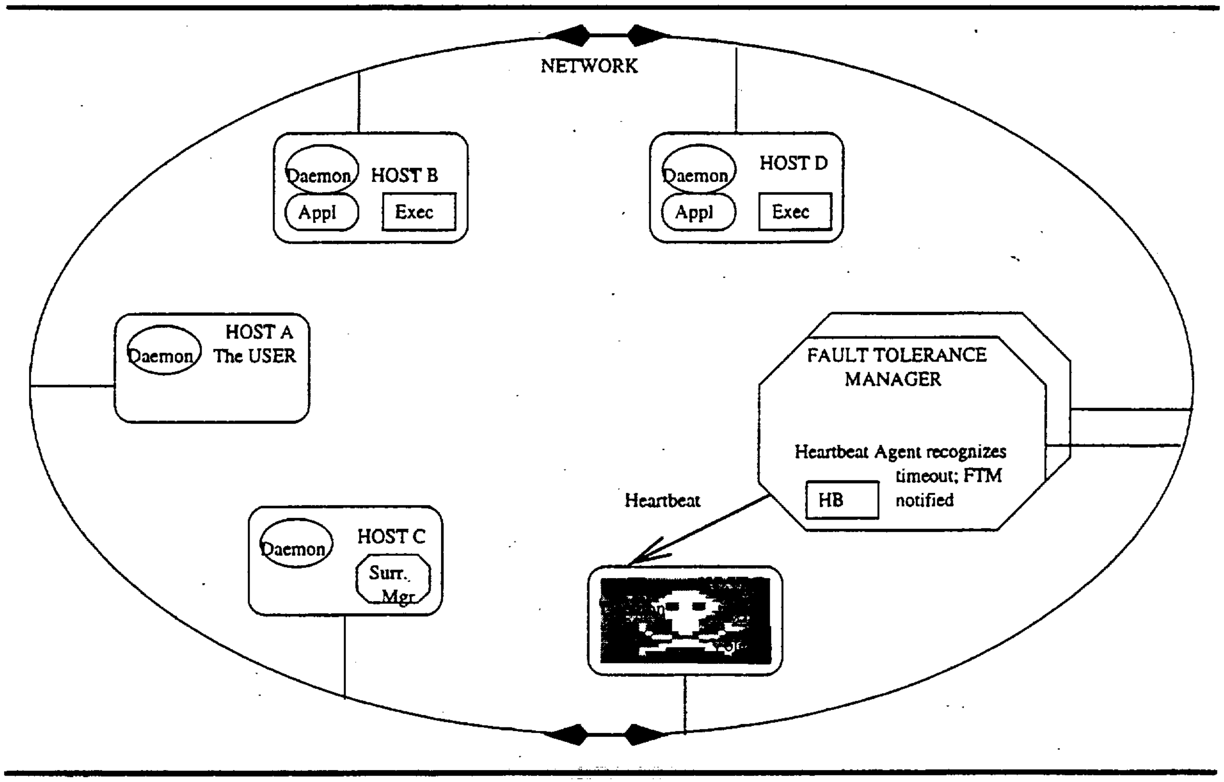

Figure 5.1 Host failure detected by heartbeat agent.

manager. After that, the FTM notifies all of its surrogate managers of the host failure. When a surrogate manager receives a host failure notification, it determines whether any of its agents were located on that host. In this case, the surrogate manager notices the voter agent was executing on the failed host. A new voter agent is created using the information the surrogate manager maintained about the unreachable voter agent. Once the voter is ready to be deployed, the surrogate manager requests a new host from the FTM. After receiving information about the host, it installs the regenerated voter agent on this new host.

Figure 5.3 shows the final step in the recovery process. The voter agent notifies the execution agents on whose results it will vote of its new location. Upon receiving this notification, the execution agents modify their results destination accordingly. If the application being monitored

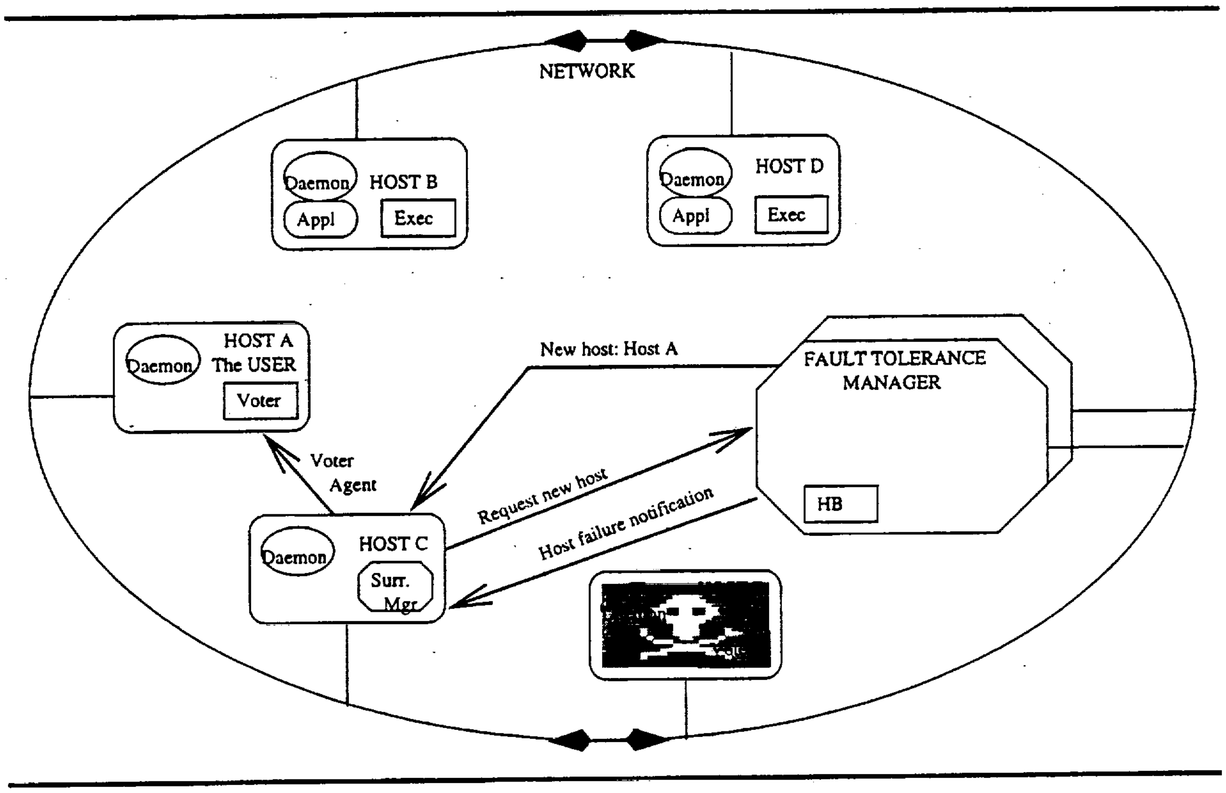

Figure 5.2 Voter agent regeneration.

by the execution agent has already successfully terminated, the results are immediately sent to the new voter agent.

5.6 Summary

Chameleon supports a number of mechanisms to detect and recover from failures of various components. The injection, detection, and recovery strategies described above are those that have been implemented in the simulation. All simulation results described in this thesis incorporate the strategies outlined in this chapter.

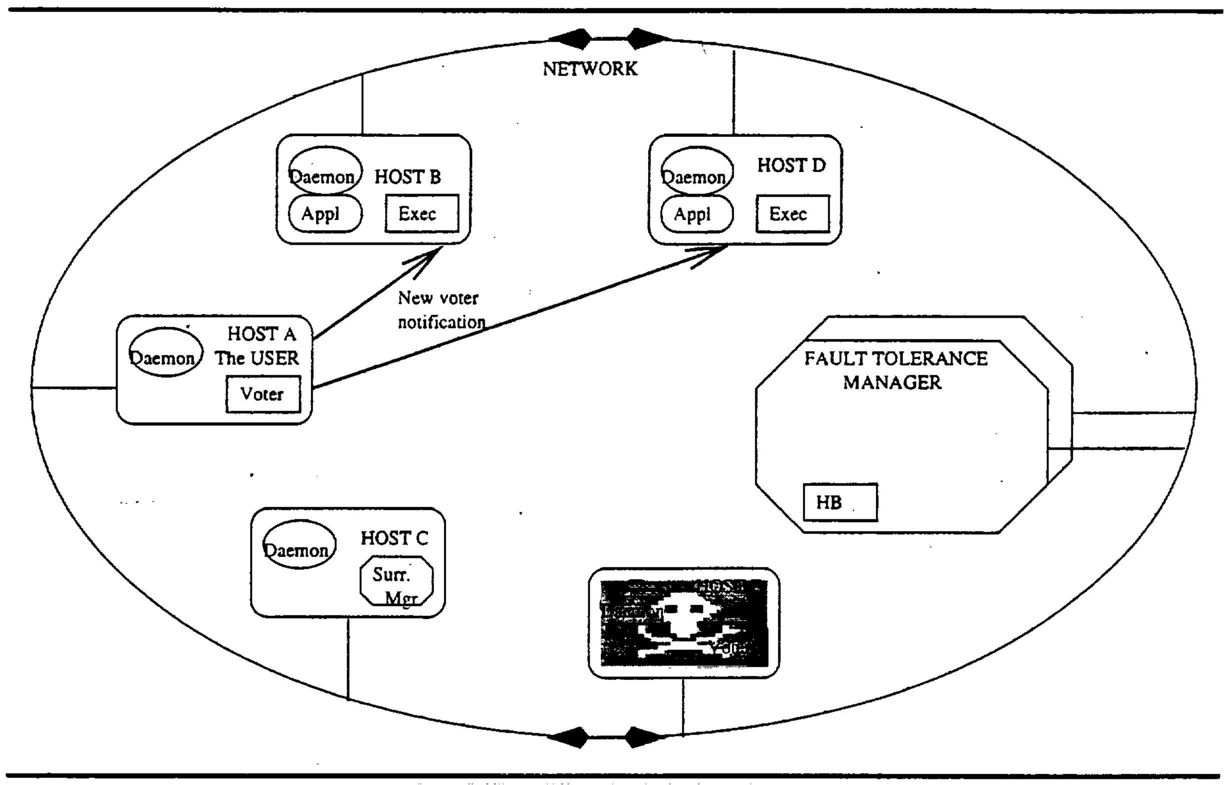

Figure 5.3 Execution agents notified of new voter agent.

CHAPTER 6

EXPERIMENTAL RESULTS

6.1 Overview

Using the simulation described in the previous two chapters, several simulated scenarios were run and analyzed. The main focus of the results is on Chameleon's fault–handling capabilities. In this chapter, the evaluated scenarios and the results obtained are discussed.

6.2 Simulation Parameters

To obtain results from the simulation, it was necessary to choose values for certain critical parameters (Table 6.1). The link bandwidth value was measured from preliminary implementation results using TCP/IP, not the Myrinet API. Other parameters necessary in the simulation included CPU times required for various tasks, such as the time it takes a voter to compare results, the time it takes a daemon to process a message and send it to the appropriate agent, etc. Many of these times were taken from preliminary implementation measurements, others were estimated.

6.3 Scenarios Evaluated and Measurements Taken

A primary goal of these measurements was to determine the performance degradation caused by running an application in Chameleon. Another goal was to measure the time to recover from

Parameter	Value
Heartbeat interval	5 s
Heartbeat timeout	20 s
Daemon monitor interval	20 s
Agent monitor interval	20 s
Link bandwidth	25KBytes/s

Table 6.1 Parameters used in the simulation.

single points of failure in various system components. A third goal was to analyze Chameleon's capability to handle multiple simultaneous failures.

To find the performance degradation with no failures, single user requests were simulated running in duplicated mode, TMR mode, and quad mode. Each of these scenarios was simulated with background workloads of 0, 1.5 (two background jobs on half of the hosts, and one on the other half), and 3.0 (three jobs on each host).

No background network traffic was simulated. In several runs of the simulation with significant network traffic, only minimal performance degradation was noticed. The number of background jobs running on the nodes had a much more profound effect on the time required to execute user requests.

To measure the performance under single failures, the same three execution modes and the same three background workloads were used as for the measurements with no failures. In addition, a single fault was injected during each execution of a user request. The faults were injected into six different components: execution agents, heartbeat agents, surrogate managers, hosts, user applications, and voter agents. The times required to complete these user requests were compared with the times to completion without failures. All user requests were assumed to require 50 seconds of CPU time. The faults were injected into each component according to an exponential distribution with a mean of 20 seconds. Each scenario was run with 10 different seeds to the fault injector, and the results were averaged.

To see how well Chameleon handles multiple simultaneous failures, the simulation was run with faults being injected into all components (each instance of the six components injected in the single failure runs and the FTM). Along with this, the simulation was run with two fault injections deliberately coinciding with several different pairs of faults.

6.4 Analysis of Results

This section contains the results obtained from running various simulated scenarios. The results obtained are discussed and analyzed. Both single– and multiple–failure scenarios are presented, as well as the performance degradation in a fault–free environment.

Table 6.2 shows the overhead required for running an application under three different fault tolerance strategies when no faults occur. The times are compared to the actual execution time of the application, taking the system load into consideration. The Chameleon overhead ranges from 6.5% to 11.8%, certainly reasonable to ensure that a critical application completes with the correct results. Since the amount of overhead changes only slightly with the execution time of the application, the percentage overhead should be smaller for applications requiring more CPU time than 50 seconds. Under the same conditions with 1000–second applications, the Chameleon overheads ranged from 2.1% to 2.8%.

Tables 6.3 through 6.5 show the times required to recover from six different types of failures for three different fault tolerance strategies. FTM failures were not considered because no recovery mechanism for them has been modeled.

Execution Strategy	Average Load	Time	Chameleon Time	Increase
Duplicated	0	50.00s	54.06s	8.1%
Duplicated	1.5	100.00s	107.38s	7.4%
Duplicated	3.0	200.00s	213.00s	6.5%
TMR	0	50.00s	54.74s	9.5%
TMR	1.5	100.00s	108.76s	8.8%
TMR	3.0	200.00s	215.44s	7.7%
Quad	0	50.00s	55.89s	11.8%
Quad	1.5	100.00s	111.72s	11.7%
Quad	3.0	200.00s	218.74s	9.4%

Table 6.2 Overhead incurred by Chameleon under various loads and fault tolerance strategies.

Failed Component	Average Load	Time to Complete	Increase over Fault-Free Execution	Percent Change
Execution Agent	0.0	81.87s	27.81s	51.4%
Execution Agent	1.5	134.09s	26.71s	24.9%
Execution Agent	3.0	237.36s	24.36s	11.4%
Surrogate Manager	0.0	54.06s	0.0s	0.0%
Surrogate Manager	1.5	109.12s	1.74s	1.6%
Surrogate Manager	3.0	219.37s	6.37s	3.0%
Voter Agent	0.0	54.22s	0.16s	0.0%
Voter Agent	1.5	107.53s	0.15s	0.0%
Voter Agent	3.0	213.16s	0.16s	0.0%
Host	0.0	76.33s	22.27s	41.2%
Host	1.5	162.82s	55.44s	51.6%
Host	3.0	232.21s	19.21s	9.0%
Heartbeat Agent	0.0	53.89s	-0.17s	0.0%
Heartbeat Agent	1.5	107.11s	-0.27s	0.0%
Heartbeat Agent	3.0	212.76s	-0.24s	0.0%
Application	0.0	83.28s	29.22s	54.1%
Application	1.5	136.00s	28.63s	26.7%
Application	3.0	240.66s	27.66s	13.0%

Table 6.3 Performance degradation caused by single failures in duplicated execution mode.

For execution agent and application failures, the performance degradation includes the time required to detect and recover from failure, as well as the time lost due to the fact that, the application is restarted from the beginning. The results show that in the case of an execution agent failure or an application failure, Chameleon may require approximately 50% or more extra time to complete the application. Most of this time is due to the fact that the application must be restarted from the beginning. If a failure occurred when an application was 99% complete, it would take about twice as long by Chameleon to process the user request. The average time taken for Chameleon to detect and recover from an application failure is 10.0 seconds; from an execution agent failure, the time required is 10.8 seconds. These times were measured with no load on the system and are constant with respect to the running time of the application. This implies that most of the overhead is being caused by the application catching up to the point of the failure.

In practice, overhead when there is an application or execution agent failure will average approximately 50% of execution time. For applications requiring a large amount of CPU time, these overheads become very large. To prevent restarting the application from the beginning, application checkpointing could be implemented. In the event of a failure in the application or its monitoring agent, the application could be restarted from the most recent checkpoint. As long as the checkpointing interval is not too large, this would solve the problem of the recovery time increasing with the execution time of an application.

Each set of results shows extremely minimal overhead for recovering from voter agent failures. In general, a voter agent failure should cause very little overhead. However, a failed voter agent can cause more delay when the time between its failure and regeneration overlaps with the completion of the application. All execution agents must stall until the voter is regener-

Failed Component	Average Load	Time to Complete	Increase over Fault–Free Execution	Percent Change
Execution Agent	0.0	81.87s	27.13s	49.6%
Execution Agent	1.5	134.09s	25.33s	23.3%
Execution Agent	3.0	237.36s	21.92s	10.2%
Surrogate Manager	0.0	54.74s	0.0s	0.0%
Surrogate Manager	1.5	124.26s	15.50s	14.3%
Surrogate Manager	3.0	223.24s	7.80s	3.6%
Voter Agent	0.0	54.90s	0.16s	0.0%
Voter Agent	1.5	108.92s	0.16s	0.0%
Voter Agent	3.0	215.60s	0.16s	0.0%
Host	0.0	81.38s	25.49s	45.6%
Host	1.5	167.98s	56.26s	50.4%
Host	3.0	237.59s	18.85s	8.6%
Heartbeat Agent	0.0	53.89s	-0.17s	0.0%
Heartbeat Agent	1.5	107.11s	-0.27s	0.0%
Heartbeat Agent	3.0	212.76s	-0.24s	0.0%
Application	0.0	83.31s	28.57s	52.2%
Application	1.5	136.06s	27.30s	25.1%
Application	3.0	240.72s	25.28s	11.7%

Table 6.4 Performance degradation caused by single failures in TMR mode.

ated before sending the application results. This is a relatively uncommon occurrence, and the overhead is fixed with respect to the application's running time.

Surrogate manager failures are similar to voter agent failures in that the overhead is minimal except when the failure overlaps with the completion of the managed application. There is one other exception for surrogate managers. A surrogate manager may fail during the agent installation process. In this case, the new surrogate manager must restart the entire installation process, since no record is kept of which agents have been installed. This could be the cause of significant overhead, especially in applications being executed with fault tolerance strategies requiring a large number of agents.

Failed Component	Average Load	Time to Complete	Increase over Fault-Free Execution	Percent Change
Execution Agent	0.0	83.78s	27.13s	49.6%
Execution Agent	1.5	137.38s	25.33s	23.3%
Execution Agent	3.0	241.96s	21.92s	10.2%
Surrogate Manager	0.0	59.67s	3.78s	6.8%
Surrogate Manager	1.5	146.00s	34.28s	30.7%
Surrogate Manager	3.0	227.13s	8.39s	3.8%
Voter Agent	0.0	55.91s	0.02s	0.0%
Voter Agent	1.5	111.73s	0.01s	0.0%
Voter Agent	3.0	218.75s	0.01s	0.0%
Host	0.0	81.91s	26.02s	46.6%
Host	1.5	158.21s	46.49s	41.6%
Host	3.0	243.44s	24.70s	11.3%
Heartbeat Agent	0.0	55.71s	-0.18s	0.0%
Heartbeat Agent	1.5	111.48s	-0.24s	0.0%
Heartbeat Agent	3.0	218.62s	-0.12s	0.0%
Application	0.0	83.78s	27.89s	49.9%
Application	1.5	137.38s	25.66s	23.0%
Application	3.0	241.96s	23.22s	10.6%

Table 6.5 Performance degradation caused by single failures in quad mode.

It is interesting to note that in the case of a heartbeat agent failure Chameleon applications require slightly less time to complete. This is because the host daemons do not need to process heartbeat messages while the heartbeat agent is not alive. This decreases the number of jobs vying for each processor slightly and allows the applications to finish a little bit more quickly. Problems may occur when a host failure overlaps with a heartbeat agent failure. In this case the time required to detect a host failure will markedly increase, potentially causing applications to be stalled. This may not be realistic in the actual implementation, as the application may be required to stall until the heartbeat agent can be recovered.

Host failures are another significant source of overhead. The average host failure takes about the same time to detect as an agent failure because the heartbeat timeout interval is the same as the agent monitoring interval. Recovery time is slightly more because the surrogate manager needs to request a new host from the FTM. The reason the results show such significant overheads is that when a host with an execution agent on it fails, it requires more time to recover from than an execution agent failure or a simple application failure. Again, this could be helped by using a checkpointing scheme, as long the scheme is architecture–independent or there is another host with the same architecture available for migrating the application.

It is not guaranteed that Chameleon is capable of recovering from multiple overlapping failures. To test how well Chameleon fares with overlapping failures, the simulation was run in a scenario in which all components could fail as described in the previous section. Running 100 user requests in this scenario yielded 93 completed requests. The remaining seven requests did not complete because not enough hosts were available. No overlapping failures caused an application to be lost.

Because the coverage of the scenario above may not have been complete, a few scenarios were run where two failures were explicitly injected at about the same time. This resulted in an FTM failure overlapping with a surrogate manager failure, which in turn caused the user request to be lost. A few more runs showed that a user request will not complete in Chameleon if an entity and its managing component fail at the same time.

Figure 6.4 shows the chain of events when an agent and its surrogate manager fail simultaneously. The double failure shown results in a race condition between the notification of the execution agent failure and the regeneration of the failed surrogate manager. If the surrogate manager is successfully restarted before the notification of the failed execution agent, the recovery will be successful. If the notification of the execution agent failure arrives to the surrogate

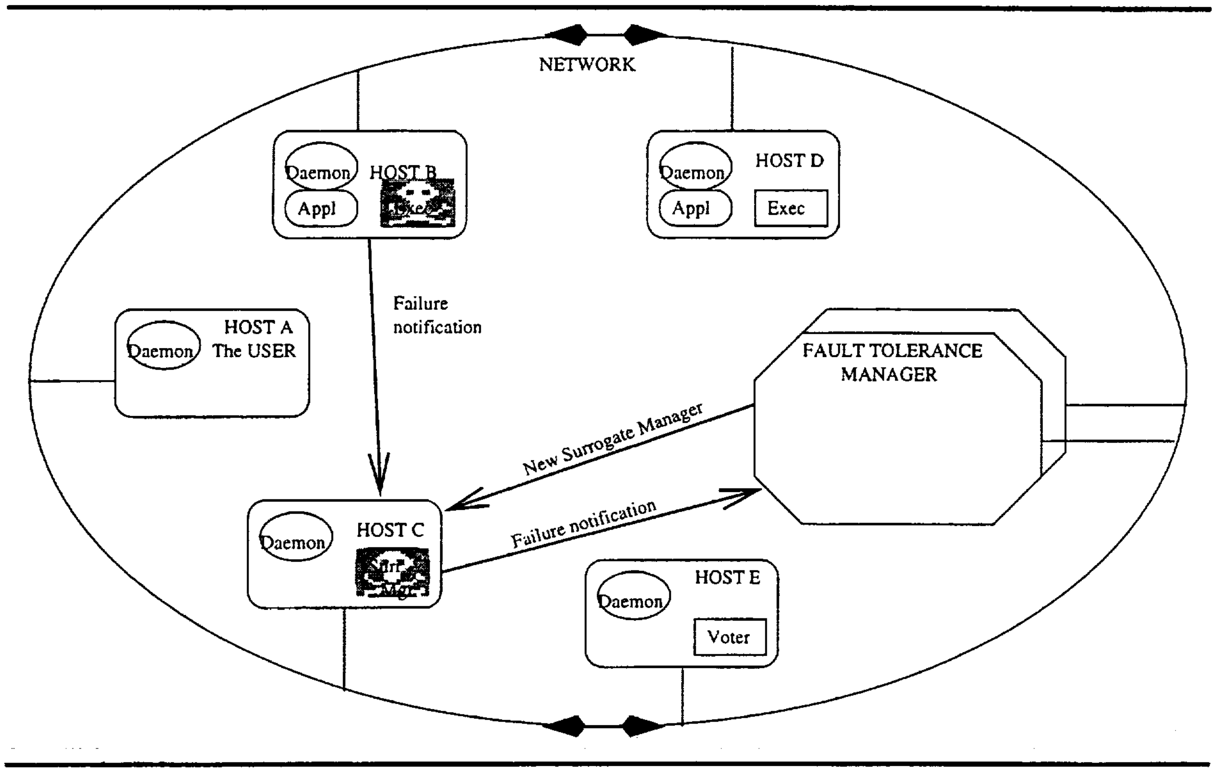

Figure 6.1 Simultaneous failures in an execution agent and its surrogate manager.

manager's node while the surrogate manager is down, the message is dropped and there is no acknowledgement of the execution agent's failure.

There are a few ways of correcting this problem, but they are not ironclad, and any attempt to design mechanisms to recover from double failures will only add to the detection and recovery overhead. It is hoped that simultaneous failures are sufficiently rare that they need not be considered when devising fault recovery schemes. Agents, managers, and daemons in Chameleon are designed to be compact and simple so they can be thoroughly tested. This will help in maintaining fault resilience and preventing such overlapping failures from occurring.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis, the simulation of fault injection and recovery in Chameleon, a framework for supporting cost–effective fault tolerant computing services, was described in detail. The simulation results have shown that Chameleon is an efficient environment for executing both critical and noncritical applications, especially applications requiring large amounts of CPU time.

The simulation showed that there is some room for improvement in some areas. A check-pointing scheme could be implemented to more efficiently recover from execution agent failures and user application failures. With such a scheme in place, the recovery overhead would no longer be dependent on the application's running time. Chameleon is capable of recovering from some kinds of overlapping failures. However, some overlapping failures may cause user requests to be lost. As long as these failures are relatively rare (i.e., recovery time is very short compared to mean time between failures for each component), it is not necessary to develop special recovery mechanisms. It is believed that Chameleon components are resilient enough for this to be unnecessary. In summary, the simulation shows that Chameleon is capable of providing a cost–effective, reliable, networked environment.

Future simulation work may include simulating additional failure modes, such as faults in the communications medium, transient faults in components besides the FTM, and modes other than fail–silent. Each of these modes will provide more insight into Chameleon's fault–handling capabilities. In addition, various methods for detecting and recovering from failures in the FTM (e.g., the FTM running in TMR mode) should be modeled. The FTM is the most critical piece of Chameleon, and it should be modeled very accurately to show that a single failure to the FTM will not be catastrophic. The effectiveness of checkpointing should be analyzed through simulation. Checkpointing will certainly help make recovery more efficient, but it is not known how much of an effect it will have on normal system behavior and performance. Finally, methods for handling parallel and distributed applications submitted by a user should be analyzed. Since it is increasingly common for user applications to be of a parallel or distributed nature, Chameleon should be able to handle these types of requests from users as well as it handles single–threaded applications.

REFERENCES

[1] K. P. Birman, *Building Secure and Reliable Network Applications.* Greenwich, CT: Manning Publications Co., 1996.

[2] K. P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit.* New York, NY: IEEE Computer Society Press, 1994.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.–K. Su, “Myrinet: A Gigabit–Per–Second Local–Area Network,” *IEEE Micro*, vol. 15, pp. 29–36, February 1995.

[4] F. Cristian, “Understanding Fault–Tolerant Distributed Systems,” *Communications of the ACM*, vol. 34, pp. 57–78, February 1991.

[5] H. M. Deitel, and P. J. Deitel, *C++ How to Program.* Englewood Cliffs, NJ: Prentice Hall, 1994.

[6] D. Dolev, and D. Malki, “The Transis Approach to High Availability Cluster Communication,” *Communications of the ACM*, vol. 39, pp. 64–70, April 1996.

[7] K. K. Goswami and R. K. Iyer, “DEPEND: A Simulation–Based Environment for System Level Dependability Analysis,” *IEEE Transactions on Computers*, vol. 46, pp. 60–74, January 1997.

[8] R. W. Horst, “TNet: A Reliable System Area Network,” *IEEE Micro*, vol. 15, pp. 37–45, February 1995.

[9] R. K. Iyer, Z. Kalbarczyk, and S. Bagchi, "Chameleon: A Software Infrastructure and Testbed for Reliable High-Speed Networked Computing," Center for Reliable and High-Performance Computing Tech. Rep. 13, University of Illinois, Urbana, IL, 1997.

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

[11] Microsoft Clustering Architecture "Wolfpack," White Paper, May 1997. *http://www.microsoft.com/ntserverenterprise/guide/wolfpack.asp*

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

[13] D. Powell, "Lessons Learned from Delta-4," *IEEE Micro*, vol. 14, pp. 36–47, August 1994.

[14] D. K. Pradhan ed., *Fault–Tolerant Computer System Design.* Upper Saddle River, NJ: Prentice Hall, 1996.

[15] H. D. Schwetman, "Introduction to Process–Oriented Simulation and CSIM," in *Winter Simulation Conference*, pp. 154–157, 1990.

[16] Sun RAS Solutions for Mission–Critical Computing, White Paper, October 1997. *http://www.sun.com/clusters/wp-ras/*

[17] A. S. Tanenbaum, *Computer Networks.* Upper Saddle River, NJ: Prentice Hall, 1996.

[18] R. van Renesse, K. P. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Communications of the ACM*, vol. 39, pp. 76–83, April 1996.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB NO. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 1/30/98	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
An Analysis of Failure Handling in Chameleon, a Framework for Supporting Cost-Effective Fault Tolerant Services

5. FUNDING NUMBERS
DABT63-94-0045
NASA NAG 1-613

6. AUTHOR(S)
Erik Haakensen

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
Coordinated Science Laboratory
University of Illinois
1308 W. Main St.
Urbana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER
[illegible] (CRHC-98-01)
UILU-ENG-98-2204

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA/ITO
3701 N. Fairfax Dr.
Arlington, VA 22203-1714

NASA Langley Research Center
Hampton, VA 23681

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The desire for low-cost reliable computing is increasing. Most current fault tolerant computing solutions are not very flexible, i.e., they cannot adapt to reliability requirements of newly emerging applications in business, commerce, and manufacturing. It is important that users have a flexible, reliable platform to support both critical and noncritical applications. Chameleon, under development at the Center for Reliable and High-Performance Computing at the University of Illinois, is a software framework for supporting cost-effective adaptable networked fault tolerant service. This thesis details a simulation of fault injection, detection, and recovery in Chameleon. The simulation was written in C++ using the DEPEND simulation library. The results obtained from the simulation included the amount of overhead incurred by the fault detection and recovery mechanisms supported by Chameleon. In addition, information about fault scenarios from which Chameleon cannot recover was gained. The results of the simulation showed that both critical and noncritical applications can be executed in the Chameleon environment with a fairly small amount of overhead. No single point of failure from which Chameleon could not recover was found. Chameleon was also found to be capable of recovering from several multiple failure scenarios.

14. SUBJECT TERMS
adaptive fault tolerance, highly available networked computing
error detection and recovery

15. NUMBER IF PAGES
48

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL