July 1997

UILU-ENG-97-2218
CRHC-97-13

University of Illinois at Urbana-Champaign

Chameleon: A Software Infrastructure and Testbed for Reliable High-Speed Networked Computing

R.K. Iyer, Z. Kalbarczyk, and S. Bagchi

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comment regarding this burden estimates or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE

Chameleon: A Software Infrastructure and Testbed for Reliable High Speed Networked Computing

5. FUNDING NUMBERS

NASA NAG-1-613

6. AUTHOR(S)

R.K. Iyer, Z. Kalbarczyk, S. Bagchi

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)

Coordinated Science Lab
University of Illinois at Urbana-Champaign
1308 W. Main Street
Urbana, IL 61901

8. PERFORMING ORGANIZATION REPORT NUMBER

(CRHC-97-13)
UILU-ENG-97-2218

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

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)

This report presents Chameleon, an adaptive infrastructure which allows different levels of availability requirements to be simultaneously supported in a single clustered environment. Fundamental components which constitute Chameleon are: (1) *Fault Tolerance Manager* (FTM) acting as an independent and intelligent entity capable of identifying and establishing the required fault tolerance strategy for executing the user application, (2) *Reliable, Mobile and Intelligent Agents* capable of migrating through the network and operating autonomously on behalf of the FTM according to built-in specifications and instructions, (3) *Surrogate Manager* operating as a pseudo-manager and capable of interacting with the user and supporting proper communications with the agents which monitor the application execution on remote hosts, (4) *Host Daemons* residing on each host and responsible for handshaking with the agents and monitoring the agents behavior, and (5) *Software Libraries* providing basic building blocks to create or re-engineer agents. A prototype implementation of Chameleon on a small LAN of heterogeneous machines connected to the high-speed Myrinet switch, is described.

14. SUBJECT TERMS

adaptive fault-tolerance, highly available networked computing, error detection and recovery.

15. NUMBER IF PAGES
28

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

Chameleon: A Software Infrastructure and Testbed for Reliable High-Speed Networked Computing

R.K.Iyer, Z.Kalbarczyk, S.Bagchi

Abstract

This paper presents Chameleon, an adaptive infrastructure which allows different levels of availability requirements to be simultaneously supported in a single clustered environment. Fundamental components which constitute Chameleon are: (1) *Fault Tolerance Manager* (FTM) acting as an independent and intelligent entity capable of identifying and establishing the required fault tolerance strategy for executing the user application, (2) *Reliable, Mobile and Intelligent Agents* capable of migrating through the network and operating autonomously on behalf of the FTM according to built-in specifications and instructions, (3) *Surrogate Manager* operating as a pseudo-manager and capable of interacting with the user and supporting proper communications with the agents which monitor the application execution on remote hosts, (4) *Host Daemons* residing on each host and responsible for handshaking with the agents and monitoring the agents behavior, and (5) *Software Libraries* providing basic building blocks to create or re-engineer agents. A prototype implementation of Chameleon on a small LAN of heterogeneous machines connected to the high-speed Myrinet switch, is described.

Keywords: adaptive fault-tolerance, highly available networked computing, error detection and recovery.

1 Introduction

In contemporary networked computing systems, a broad range of commercial and scientific applications which need varying degrees of availability must coexist. It is not cost effective to develop a reliable platform in each case. It is more efficient to build an infrastructure which provides the required levels of dependability based on application needs. It is also essential that the proposed alternatives be cost-effective. Hence, a primary issue to be addressed is how the envisioned infrastructure can leverage off-the-shelf components. There have been exhaustive studies on fault tolerance strategies (hardware and/or software implemented) capable of providing

efficient mechanisms to deal with system operational failures. Most of these works however focused on specific application needs and thus provided piecemeal solutions. Little work has been done in addressing how to build reliable networked computing system out of unreliable computation nodes. As a result there is no comprehensive solution for providing a wide range of fault-tolerant services in a single networked environment. The most feasible way of understanding how such a software environment would fit on top of existing layers (like the operating system, the network interfaces, etc.) is to implement the infrastructure for providing a range of reliable services. So it has been attempted to integrate a wide variety of existing strategies into a single environment on top of the hardware and software of heterogeneous workstations on a LAN. A prototype implementation of the envisioned approach is expected to provide useful insights into its feasibility.

This paper introduces and describes Chameleon, a software infrastructure capable of providing configurable (i.e., according to user specification) levels of fault tolerance in a networked computing environment. Methods and techniques supported by Chameleon are embodied in a set of specialized, reliable and intelligent *Agents* supported by a *Fault Tolerance Manager (FTM)* and its *Surrogates* to ensure that applications execute with the required levels of reliability. The components have been so designed that none of them is a single point of failure. Each of the components is active for a certain period e.g., during setting up the system configuration. In the case of a component failure during its active phase there is a provision for recovery, either by switching to backup or by regenerating the component. The broad goals of Chameleon are stated below:

- Dynamic handling of changing failure criticality requirements: certain segments of the application may be deemed more critical than others. The FTM is capable of recognizing such demarcations and executing different segments using different configurations tuned to provide required reliability levels.
- Adaptability to user-directed availability levels: the FTM, through the Agents and the available libraries of fault detection and recovery mechanisms, determines the user availability requirements and executes the application in the user-directed mode. Thus, unlike most

existing implementations, it can adapt to user availability requirements with just the amount of overhead required for supporting the particular level.

- Rapid error detection and recovery: Chameleon supports failure detection at three different levels:

  Application or process monitoring. The application executing at a remote host is monitored locally by the agents residing at the remote host. A misbehaving application raises an offending signal which is captured by the corresponding agent. Thus the detection is not tied up with the FTM.

  Agent monitoring. The agents are also monitored locally, by the Host Daemons residing on every host. The monitoring is through exceptions as well as *i_am_alive* messages.

  Node monitoring. Whether a particular computation node is up or down is monitored by the FTM through periodic heartbeat messages. The heartbeat scheme could be extended to include nodes sending out signatures containing information about the status of the various components, which would then be tallied at the FTM.

- Providing reliability with *unreliable* hardware and software components : Chameleon does not pre-suppose any special reliable platforms for execution or any specialized software. The computation nodes are mainly regular workstations or personal computers. All nodes are expected to have pre-installed C++ compiler and interpreter for a scripting language such as TCL. The Fault Tolerance Manager needs to run with a high degree of reliability (e.g., in a primary/backup mode). This would make the cost of maintaining the FTM high, but it would be distributed among all the end-users. This approach seems to be much cheaper and more efficient than to provide a separate fault-tolerant machine to individual user. The point to be stressed is that Chameleon uses components off the shelf. Thus there is no specialized hardware requirements for the user machines, nor any thick pre-installed software layer. This makes for cost effective services.

- Scaleable services : Chameleon provides a unified framework for allowing the system to scale in two major dimensions:

Physical scalability - the system structure is based on a high-speed network such as Myrinet or ServerNet. In this environment adding new computation nodes will improve the system performance (because of high bandwidth of the underlying network) while communication overheads will grow slowly with system size.

Fault tolerance scalability - Chameleon provides a general procedure for creating agents and extending functions of already existing agents. The FTM utilizes this procedure to generate agents for supporting the application dependent fault tolerance strategy. To support this approach, two basic libraries are provided: (1) *library of building blocks* and (2) *library of agents*. The building blocks are used in agent creation. The user is also allowed to develop his own basic building blocks which can be incorporated in an agent created by FTM.

2 Related Research

There are examples of highly fault-tolerant, distributed systems which have demonstrated that there are efficient solutions for providing highly reliable computations. For example, Tandem architecture groups servers (processing nodes) into failure masking server pair group, in Stratus architecture all components (e.g., CPU, I/O, communication controller) are implemented by paired microprocessors that execute identical instruction streams and continuously compare the results. These are, however, dedicated fault-tolerant architectures oriented to specific applications and offering static level of fault tolerance. In the common networked systems, the computation nodes are usually not fault-tolerant machines or have very little to support highly reliable services. The question to be asked is how to achieve high reliability with unreliable components?

To be cost effective and operationally efficient, a promising solution is to use a specialized software layer for organizing the system components into a reliable computation environment. By adding a dedicated software layer capable of providing predictable, fault-tolerant behavior, reliability of the overall system can be significantly improved. Current approaches to reliable distributed computing are mainly based on exploiting distributed groups of cooperating processes. There is a number of studies that address various aspects of this paradigm e.g., process synchronization, jobs distribution, fault tolerance strategies, e.g., [1], [3]. These studies result in a

number of tools to support the construction of reliable services, e.g., Isis [2], Totem [5], Delta-4 [7], Horus [8].

Most of these approaches require specialized and complex software layer which must be installed in each computation node. For example, they heavily rely on underlying protocols for supporting group membership and atomic broadcast. As a primary objective of developing these systems is to provide a software environment for constructing distributed applications, the service availability issue is not often of a primary concern. Consequently, there is no dedicated mechanism for error detection and the fault tolerance is somewhat of "a side effect" of the use of the group communication approach. The system usually relies on the error detection based on capturing the time-out in a response from one of the processes in the group. When the time-out is encountered, the group must reach a consensus as to which process failed and should be excluded from the group. Then, the process next in the hierarchy takes over.

Recently, the Piranha system presented in [4] is an attempt to address the issue of service availability in the execution of a distributed application. Piranha is a CORBA-based tool for attaining high availability in distributed systems. CORBA, a Common Object Request Broker Architecture is widely used distributed standard which provides interoperability between applications on different machines in heterogeneous distributed environments and interconnects multiple object systems [6]. CORBA does not address issues of availability and reliability and cannot consistently detect partial failures in distributed applications. Piranha exploits the dynamic replication of objects for achieving high availability. However, the necessary software layer is *thick* as Piranha runs on Electra, a CORBA Object Request Broker supporting the object group abstraction, reliable multicast, state transfer, and virtual synchrony. In addition Electra is built on the top of group communication subsystems such as Isis and Horus.

3 Chameleon Infrastructure - Overview

Chameleon (see Figure 1) provides an adaptive infrastructure which allows different levels of availability requirements to be simultaneously supported in a single, heterogeneous clustered environment. Fundamental components which constitute the Chameleon include: (1) *Fault Tolerance Manager* (FTM) acting as an independent and intelligent entity capable of identifying

and establishing the required fault tolerance strategy for executing the user application, (2) *Reliable, Mobile and Intelligent Agents* capable of migrating through the network and operating autonomously on behalf of the FTM according to built-in specifications and instructions, (3) *Surrogate Manager* operating as a pseudo-manager for one particular application and capable of interacting with the user and supporting proper communications with the agents which monitor the application execution on remote hosts, (4) *Host Daemons* residing on each host and responsible for handshaking with the agents and monitoring the agents' behavior, and (5) *Software Libraries* providing basic building blocks to create or re-engineer agents. The idea is that a failure of none of these individual entities shpuld compromise the whole system.

In the initialization phase, the FTM collects information about the system configuration and characteristics of individual nodes. Initialization agents are sent to hosts to obtain this data and to install the host daemons on participating machines. After successful initialization, Chameleon is ready for accepting the user requests. When a 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, types of results etc. Based on the collected information on the application, the FTM can identify the necessary fault tolerance strategy and can designate set of agents to initiate and monitor the application. Creation of agents is performed according to a predefined procedure which utilizes two software libraries: (1) library of building blocks and (2) 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 execution migrate through the network to the selected nodes, install themselves on the machines and initiate the application execution. In addition, a Surrogate Manager is spawned by the FTM and associated with each application. The Surrogate Manager can be allocated on any machine in the network (including the node running the FTM). It maintains copy of the system information supported by the FTM, provides reliable communications with the user and supervises the agents which monitor the application execution on the remote hosts.

To ensure a rapid reaction to the application failures, the application is watched by the agent which installed the application at the remote host and started execution. The agent communicates

to the appropriate Surrogate Manager detectable application misbehavior. As the agent itself may fail, it is watched by the host daemon which is capable of notifying the Surrogate Manager about agent failures. The Surrogate Manager can re-generate a new agent either to complete or to restart the application. Agents and Surrogate Managers once generated can act autonomously and the FTM is free to serve other user requests. As the Surrogate Managers are capable of performing the basic functions of the FTM, the application may complete even in presence of FTM failure. Errors and failures of the Surrogate Manager are directly reported to the FTM by the local Host Daemon monitoring the Surrogate Manager. The FTM, then re-initializes the application execution just as it would handle a new request. In order to detect node failures the FTM uses heartbeat messages which are sent with a predefined frequency. In the case of a 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. The possible solution is to support a passive backup FTM which supports the system information and is updated each time the system state changes (typically when a new application is admitted to the environment or when an application exits from the environment).

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 aim is to provide a relatively thin software layer which must be present in each machine in the environment. The implementation of Chameleon is *not* based around CORBA, the Object Management Group's standard for building distributed systems with the design goals of heterogeneity, interoperability and extensibility. The motivations behind this design choice were multiple. To build Chameleon around CORBA would necessitate a full CORBA implementation around all the nodes in the system. The CORBA software would include an ORB with its name registry, support of IDL (Interface Definition Language) stubbers, etc. While increasingly vendors are providing applications conforming to CORBA specifications, still the vast majority of the existing applications were not built around CORBA objects. For instance, Piranha [4] requires a highly sophisticated CORBA ORB implementation capable of supporting object groups and failure detection. Hence in an environment like Chameleon which addresses availability needs in a general system of networked workstations (as opposed to specialized systems built with CORBA objects, for example), it was not considered desirable to impose the overhead of CORBA object

handling. The cost of CORBA standardization is paid in the form of performance degradation because of intermediate layers of interfaces that an application must go through. This conflicts with our other design goal of high speed error detection and recovery. The important point however is that we do not preclude the use of CORBA in future versions (or alternate versions whose goal may be slightly different eg. to support CORBA applications only). We can port some of the CORBA layer along with the Host Daemons and the nodes would then be able to support CORBA applications. The ORB can be a centralized one residing on the dedicated FTM machine. The operative word is to support user-defined levels of availability without the burden of specialized software for prior installation at the user nodes. However we do not rule out CORBA extensions to the project in the future.

The key features of Chameleon can be stated as follows:

- Chameleon is the first infrastructure which explicitly addresses the dynamic adaptation to the user required availability level in a heterogeneous, multi-node system
- Chameleon, by employing Reliable Agents and Surrogate Managers, maximizes the chances of the application to complete even if an entity of the infrastructure fails
- Chameleon does not require thick software layer to be maintained by each computation node

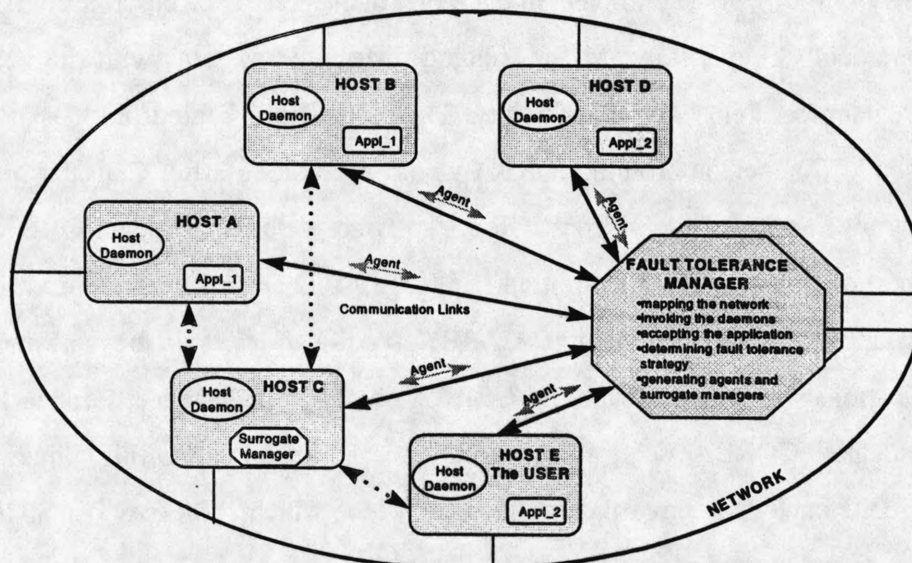


Figure 1: Reliable, Networked Computing Environment

4 Components of the System

This section presents a detailed description of components of the Chameleon. Basic functions, operational modes and component interactions are identified and characterized.

4.1 Fault Tolerance Manager (FTM)

The Fault Tolerance Manager is an intelligent, independent, active entity which operates in the networked environment with the task of providing continuous execution of the user request despite errors. The FTM is built as a reliable unit operating on a dedicated server. The decision mechanism at the FTM is responsible for identifying the configuration to execute the user application (e.g. single machine with recovery), determining machines to be used for the chosen configuration and finally setting up the environment for the particular mode of execution. The primary functions of the FTM can be summarized as follows:

- mapping the network, i.e., identification of the network configuration and collecting information about the nodes in the system. The data about the nodes are collected in the designated data structure which is updated when the nodes join or depart the network.
- invoking a daemon process on each node in the network for supporting node's communication with FTM
- collecting application specifications from the user
- determining fault tolerance strategies for providing the application required availability level. The decision procedure is based on the application requirements and a data on prior history of failures and uses software libraries on error/failure detection and recovery techniques to compose final fault tolerance strategy.

- generating a set of Agents and an associated Surrogate Manager for supporting the application execution according to earlier determined fault tolerance strategy

```
struct        config_info   {
      string        machine_name,
      string        net_address,
      string        architecture,
      string        OS,
      string        memory
};
struct        appl_info {
      string        user_name,
      string        user_machine,
      string        appl_name,
      string        agents_used[]
};
config_info system_config_table[];
appl_info           runtime_table[];
appl_spec           applSpec_table[];
```

```
struct appl_spec   {
        string specScope;
        string reliability,
        string results[]
        string sysResource,
        string #machines,
        string vote,
        string agree,
        string timeOut
}
```

Figure 2: Data Structures for Maintaining System Information

In order to support these functions, the FTM utilizes and maintains information about the system. Three tables are used to support system specific information: (1) **system configuration table** which contains information regarding the various hosts which are participating in the environment and (2) **runtime application table** which contains information related to the various applications which are currently active in the environment, and (3) **application specification table** which contains information on the user application. The basic data structures used to keep the information are given in Figure 2.

Network mapping and host daemons initialization

The network mapping and the host daemons initialization is accomplished using initialization agents - ***init_agent().*** Figure 3 presents the general structure of the initialization agent. It is assumed that functions in the agent body, e.g., **`get_sys_info()`** are already implemented and exist in one of the software libraries supported by the FTM (software libraries are presented in the further sections).

```
init_agent  (destination_host) {

config_info systemTable;

get_sys_info(systemTable);
init_sys_daemon();
go_back_to_FTM();
put_info(system_config_table);

}
```

```
get_sys_info(systemTable) {

system();   // obtaining the system
specific info

systemTable->machine_name = name;
systemTable->net_address = addr;
systemTable->architecture = arch;
systemTable->OS = SUN
systemTable->memory = mem_size;

}
```

```
init_sys_daemon() {

start(); //install the daemon
init_watchdog();
set_dedicated_port();
waiting_for_agents();

}
```

```
init_watchdog()  {
if (monitor_notification) {
      watch_agent(agent_id);
      if (agent_terminated) {
       notify_FTM(agent, OK);
      }
      else if (agent_failure) {
       notify_FTM(agent, failure);
      }
 }
}
```

Figure 3: Basic Structure of the Initialization Agent

The initialization agent is responsible for setting up the system with the participating hosts. Whenever a host is plugged into the network, this agent is invoked and it executes steps necessary for the host to be recognizable by the FTM. The agent also sets up the Host Daemon (`init_sys_daemon()`), to handle communication with the FTM. A host on being plugged into the network needs to have an interface (e.g., a TCP configured port) capable of accepting the initialization agent from the FTM. The initialization agent queries for system parameters from the hosts (`get_sys_info` ()) such as the memory availability, the native operating system, and fills up the information in the system configuration table at the FTM (`put_info(system_config_table)`). This data structure is later used by the FTM for issuing system specific commands at the remote host as well as determining the availability of required resources for running a particular application at the machine.

Communication with the user

The input to the system are the user's requirements gather by the specialized query agent - **user_query_agent()** (see Figure 4). The information is specified in a predefined format given by the **app_spec** data structure (see Figure 2). The meaning of fields available in the format is discussed below.

```
user_query_agent (user_host)  {

appl_info          applInfo;
appl_spec          applSpec;

      get_appl_info(applInfo);
      get_user_spec(applSpec);
      go_back_to_FTM();
      put_info(runtime_table);
      put_info(applSpec_table);
}
```

Figure 4: Basic Structure of the User Query Agent

Scope of the specifications(specScope) - determines parts of the application which must be executed with different availability levels. This parameter can take two values: *entire_appl* and *partial_appl*. In the latter case, the user must provide explicit demarkation to identify individual segments of the application corresponding to the specified level(s) of reliability.

*Required reliability level(**reliability**)* - specified as an integer starting from 1 (for the lowest reliability level) up to n. The predefined fault tolerance strategies are described in Section 4.2.

*Results of interest (**results**)*- specified as variables of interest at the end of the execution of the application or as the output file into which the results of the application are saved.

*System resources required (**resources**)* - specified as an amount of runtime memory required. Other applications required resources might include ghostscript, gnuplot for example.

*Number of machines (**#machines**)* - determined as a number of machines on which copies of the application are to be executed.

*Voting strategy (**vote**)* - specified as n_of_m, i.e., n among m machines must agree for application to succeed (this also includes "no_voting" strategy).

*Agreement criterion (**agree**)* - specified as an exact match or match in a range of values defined in terms of absolute value or percentage variation from mean.

*Application time-out (**timeOut**)* - an upper bound on the execution time of the application provided by the user. If the application does not terminate within this period, it is taken to have failed.

Identification of Fault Tolerance Strategy

The FTM determines the required fault tolerance strategy based on the data collected during communication with the user. In order to facilitate this decision, the FTM utilizes the three system tables (*system_config_table[], runtime_table[], applSpec_table[]).*The general procedure in selecting/identifying a required fault tolerance strategy is presented in Figure 5.

First, the FTM calls *FT_strategy* to identify whether the fixed level of availability is specified for the whole application. If so, the FTM calls *FT_config* to find which fault tolerance strategy to use. If the user specifies application regions with varying availability levels, the FTM calls *instrument_appl()* to identify the criticality levels, collects the data in a table and calls *FT_config* for determining appropriate fault tolerance strategies. The *FT_config* allows to distinguish among predefined strategies as well as the user might define his own approach(es). Functions *single_machine_nr(), single_machine_r(), dual_exec(), tmr_exec()* are used (by *FT_config)* to set up default fault tolerance strategies, single machine execution without recovery, single machine execution with recovery, duplicated execution, triple modular redundancy, respectively. The latter option in the *FT_config* is for supporting user specified strategy which can be incorporated into an agent. Each of these functions support a procedure for creating the necessary set of agents and surrogate manager capable of supporting the application execution.

Examples of agent creations and characterizations of different fault tolerance strategies are given in the next section.

```
FT_strategy(*applSpec_table) {

  if (specScope == entire_appl)    {
      FT_config();
  }
  else {
      instrument_appl(appl_name);
      FT_config();
  }
}
```

```
FT_config (*applSpec_table, userId){

   switch (reliability) {
     case(1) {
      single_machine_nr();
    }
     case(2) {
       single_machine_r();
     }
     case (3) {
       dual_exec();
     }
     case (4) {
       tmr_exec();
     }
     default  {
       user_define();
     }
}
```

```
user_define(*applSpec_table) {
if (#machines == n) {
  if (voting == kOFn) {
    if (agree == EXACT) {
      voter(k, n, EXACT);
    }
    else if (agree == INEXACT) {
      voter(k, n, agreeRange);
    }
    else {
      userVoteFunc();
    }
  else {
    userVoteStrategy(m1, m2,…, mn);
  }
}
else  {
  error(unidentified_strategy);
}
```

Figure 5: Procedure for Identifying the Fault Tolerance Strategy

4.2 Reliable Agents

The Agents are seen as a failure resilient carriers of information/stimulus to/from the Fault Tolerance Manager. They are designated by the FTM to perform the actions/operations needed for successful completion of the application in the designated mode. An agent is expected to be sufficiently intelligent to execute specified functions in an autonomous fashion. This is to prevent overload the FTM with too many tasks, which could reduce the FTM performance and thus decrease the utility of the environment. The primary characteristics of agents are: (1) mobility, (2) reliability, and (3) scalability (see Figure 6).

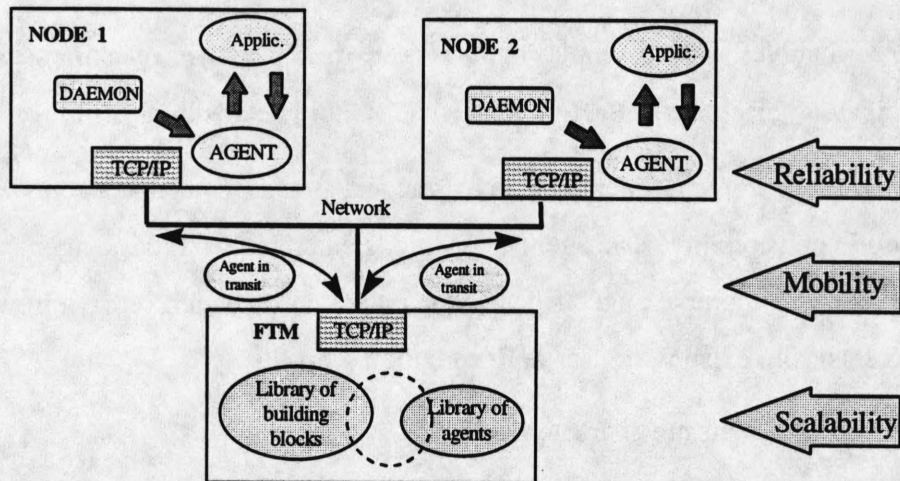


Figure 6: Primary Characteristics of Agents

Mobility. Agents migrate through a computer network in order to accomplish actions specified by the FTM. Well-known communication protocols like TCP/IP may be used to support the mobility.

Reliability. Agents are resilient to the network or node failures. To achieve this, the agent code in the already existing libraries is tested rigorously against erroneous execution. It is also important to ensure that failure in the agent does not cause a crash of the application which it was in charge of executing or that the agent crash does not propagate out of the node. To meet this requirement, agents are watched by host daemons (the host daemon is notified which agent it will have to monitor for possible crash by the micro-operation *monitor()*) and if an agent fails, the daemon notifies the Surrogate Manager or the FTM (in the case of the Surrogate Manager failure). The agent is protected from the network corruption by guarding it with check sum.

Scalability. Agents are easy to create or re-engineer using elementary blocks or already existing agents. The Chameleon provides a unified, general framework for creating new agents or extending functions of already existing agents. Consequently, the user can actively participate in developing 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) library of building primitives, and (2) library of agents.

- *Library of building blocks* contains micro- and macro-operations for supporting application execution in the distributed environment. Agents can be created, modified, re-engineered

using these bubbles. An example bubble is `notify`(target_machine, execution_mode, list_participating_machines). Section 3.4 provides more details on this library.

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

Primary functions of Agents are characterized below.

Configuring the system according to the selected fault tolerance strategy. Once the FTM decides to run an application in a particular mode, the set of agents and an associated surrogate manager are invoked to set up the environment to support the selected execution mode e.g., triple modular redundant execution. The agents and the surrogate manager take over from there. The agents have the tasks of porting the application code or executable to the three machines, executing them there, monitoring their status during execution (for example trap any segmentation violation signals, etc.) and finally returning the results to the surrogate manager. Predefined example configurations include:

Single machine with no recovery : in this most unreliable mode of execution, the user application is run on a single machine with no recovery information being maintained.

Single machine with recovery: in this mode the application is run on a single machine but provision is made for recovery if the application terminates abnormally. For example, checkpointing information is kept and in case of crash, if a homogeneous machine is available the code is re-executed from the checkpoint. If a homogeneous machine is not available, then the application is ported to another machine, recompiled and restarted.

Dual execution: the application is executed concurrently on two machines, (a primary and a secondary). An agent resides at each machine and monitors the application. In a simple operational mode, if the primary terminates normally, its result is taken and the secondary process and agent are killed. If the primary fails, the result from the secondary is accepted. In the case of double failure the application is taken as having failed. In another possible operational mode the

results from the two executions are compared to produce final output from the application execution. In the case of any discrepancy between the two outputs, a failure is signaled.

Triple modular redundancy (TMR): This mode corresponds to the highest reliability requirements. The application is run on three machines and their results are voted on based on the simple majority principle. The voting is performed by the Surrogate Manager.

Monitoring the system. Chameleon supports three levels of system monitoring: (1) application monitoring, (2) agent monitoring and (3) node monitoring. The agents are responsible for monitoring the application execution and notifying the associated Surrogate Manager on a detected error. The software library available with the FTM provides series of detection mechanisms implemented as a macro-operations and used for building agents. Example detection mechanisms include:

Heartbeat: The FTM sends out heartbeats at regular intervals to ascertain whether a particular node is up or down. The heartbeat interval should be carefully chosen as too high a frequency can flood the system and too low a frequency can lead to imprecise monitoring. On receiving a heartbeat query from the FTM, the host responds to let it know its status. An alternative to this simple heartbeat scheme would be for the host to send out a signature message which is tallied at the FTM.

Application time out: The user can specify an upper bound on the time to completion of his application when he submits it. When the Surrogate Manager does not get notification of completion of the application from the agent in charge of monitoring it, it concludes that the application has failed.

System panic: The agent monitoring an application at a remote site can detect a system panic or a signal indicating a misbehaving application, trap the signal and notify the FTM or Surrogate Manager.

Voter: When an application is executed in the TMR mode, then a conflict in the voting results can point to an error in the application.

Restoring the application. In the case of an error in application execution or a node failure, Chameleon is capable of restoring the application on the same machine or other available

machines. The software library with FTM provides set of recovery mechanisms implemented as macro-operations for restoring the application. Example recovery mechanisms include:

Process Migration: In the case of node failure the application must be migrated to another node. FTM or Surrogate Manager designate a new machine to run the application and restart (via an agent) the execution. The process migration between homogeneous nodes utilizes checkpointing and rollback and process migration between heterogeneous nodes is based on restarting the application after recompilation.

Checkpointing: Homogeneous checkpointing is supported in the system. This allows a rapid process recovery by not necessitating recompilation and re-execution from the beginning of the process. Heterogeneous checkpointing is considered as a valuable option, however, support of checkpoints on different platforms characterize large complexity, particularly when a checkpoint requires storing the entire system state which has different representations on different platforms.

4.3 Surrogate Manager

A Surrogate Manager is spawned by the FTM after the required fault tolerance configuration has been determined. It is created using procedure similar to the one employed for creating agents. Each Surrogate Manager is associated with an application (or for simple applications, several of them can also share the same Surrogate Manager). The Surrogate Manager can be seen as a "super agent" or "pseudo-manager". It is capable of acting as a regular agent e.g., it can travel through the network to the designated nodes, 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 which are designated to control the application, it can re-generate agents which failed during the 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 is located. By this means, the application can survive even in the case of FTM failure. The system information which must be available to the Surrogate Manager include: full specification of the application and access to the software libraries used to create or re-engineer the agents. It is worth noting that Surrogate Manager can be allocated on the same machine as the one on which the FTM is running. An example of the Surrogate Manager and its role in the application execution scenario is presented in section 5.

4.4 Software Library

The Software Library contains basic building blocks/objects for implementing a variety of agents. Two basic classes of objects are distinguished: (1) micro-objects which represent elementary actions and (2) macro-objects which provide a full implementation of more complex mechanisms. An example of the first category is the *notify()* function which provides a mechanism used by agents for notifying about various events that happen in the system. Macro-operations basically provide implementations of various detection and recovery techniques. The predefined mechanisms for error detection and recovery are implemented and collected in the library as macro-operations e.g., different voting algorithms. The library is an open entity and can be extended with new components. The user is allowed to actively participate in developing new strategies which can then be included to the library and used for creating agents (however the scope is restricted to that user's applications only). An example is a customized acceptance test suited for checking the correctness of computation results.

4.5 Host Daemon

The host daemons are entities at each of the hosts that are responsible for handling communication with the FTM or the Surrogate Manager via the agents. The daemon processes accept the agents and interact with them to accomplish their task. The daemon processes have intelligence to recognize the type of agent being sent over and have a well-defined handshaking protocol for communicating with each of the agents. The daemon process is also capable of monitoring Agents and Surrogate Managers behavior. When the host daemon detects a malfunctioning of the agent it notifies the Surrogate Manager. An error encountered in the Surrogate Manager is communicated directly to the FTM. The FTM then sends over clones of the Agents and/or the Surrogate Manager to the particular host which complete the execution of the application in the desired mode.

5 Creation of Agents & Surrogate Managers- example scenarios

The FTM utilizes software libraries to compose Agents and Surrogate Manager for satisfying application requirements. The FTM gathers the user requirements (through the user query agent) and then checks in the agent library to see if an agent or surrogate manager with the required functions is available. If it is, the FTM invokes the appropriate component otherwise it creates the

agent or the surrogate manager either from scratch or by specializing an already existing component. Examples of both methods are presented below. Example 1 presents a method of constructing agents and the surrogate monitor from scratch and Example 2 demonstrates creation of the surrogate monitor by specializing an already existing component.

Example 1.

Lets assume that the user wants to execute the application ***my_app.c*** in the centralized duplicated mode with the fail-safe failure semantic. By centralized we mean that the results from the two machines will be gathered and voted upon at the Surrogate Manager. Fail-safe failure semantic implies that if the results of the primary and the secondary are in conflict the application is taken to have failed. The user specifies that the results of the application are stored in a file called ***output***. The FTM after taking the user specs decides on two machines, say ***x*** and ***y***, to run the application and generates the following two agents and a single Surrogate Manager from the building blocks. In this case all the basic building blocks are already existing in library.

Surrogate Manager#1

The Surrogate Manager can reside in any node in the network including the node which is running the FTM. This entity does the work of setting up two hosts to take part in the duplicated execution mode, notify the two host daemons to monitor the relevant agents, sends the application code to the two machines, then waits for the results to be brought back from the two hosts and finally compares the two results for passing back the final result of the application (or error status) to the user. The pseudo-code of the Surrogate Manager is shown in Figure 7.

```
notify(mc x,dup,mc y);                                              (1)
notify(mc y,dup,mc x);                                              (2)
install(mc x,agent#1);                                              (3)
install(mc y,agent#2);                                              (4)
monitor(mc x,agent#1);                                              (5)
monitor(mc y,agent#2);                                              (6)
wait(agent#1,request_to_send_code)                                  (7)
wait(agent#2,request_to_send_code)                                  (8)
send(my_app.c,mc x,src);                                            (9)
send(my_app.c,mc y,src);                                            (10)
wait(response_from_agent#1);                                        (11)
wait(response_from_agent#2);                                        (12)
if (read_failure(agent#1,errno1) || read_failure(agent#2,errno2))
      notify_FTM_failure(surrogate#1,errno1||errno2)                (13)
else
      { // Both hosts executed application correctly
        a = get_result(agent#1);                                    (14)
        b = get_result(agent#2);                                    (15)
        final_result = vote(a,b,2,2,EXACT);                         (16)
        if (final_result == NULL)
             notify_FTM_failure(surrogate #1,NO_AGREEMENT);         (17)
        else
             notify_USER(surrogate #1,final_result);                (18)
      }
```

Figure 7: A Surrogate Manager#1 - pseudo-code

Lines (1) & (2) notify the hosts *x* and *y* that they will be participating in duplicated execution with each other. (3) & (4) install the agents #1 and #2 in hosts *x* and *y.* (5) & (6) instruct the host daemons at machines *x* and *y* to monitor the agents #1 & #2. (7) & (8) wait for the agents #1 and #2 to request to send the application code from *x* and *y,* respectively. (9) & (10) send the application code **my_app.c** to the two hosts. The src in the parameter tells the daemons that it is a source file that is being sent over that needs to be compiled and then executed. (11) & (12) wait for responses from agents #1 and #2 from *x* and *y,* respectively, either about correct termination or an error flag detected during execution. If either of the agents flags an error, in (13) FTM is notified of the failed execution along with the error code. If both hosts ran the application succesfully, in (14) & (15) the result is collected from the two machines. In (16) the voter is invoked with the two results and specifying that a 2 out of 2 exact agreement is required for the voter to succeed. If the voter fails, NULL value is returned. In that case (17) i.e., the two executions returned different values, the FTM is notified of the failure. Otherwise, (18), the final result agreed upon by both machines is communicated to the user.

Agent#1

The *agent#1* is sent over to the host daemon on machine ***x*** where it will monitor the running of the application that will be sent over there by Surrogate Manager. This agent will detect faults if any during the running of the application and will notify the Surrogate Manager. If the application terminates normally, the agent sends the result back to Surrogate Manager. The pseudo-code of the *agent#1* is shown in Figure 8.

```
 send(surrogate#1,request_to_send_code,my_app.c);                    (1)
 install(my_app.c);                                                  (2)
 monitor(my_app.c);                                                  (3)
 if (detect_error(my_app.c, errno) != 0)
   notify_failure(surrogate#1, errno);                               (4)
else
   { // The application executed correctly
    a = collect_result(output);                                      (5)
    return_result(surrogate#1, a);                                   (6)
    terminate(monitor,agent#1);                                      (7)
    exit();                                                          (8)
   }
```

Figure 8: Agent#1 - pseudo-code

Line (1) sends the request to the *Surrogate Manager#1* to send the source **my_app.c.** (2) installs *my_app.c* i.e., compiles the source and starts execution. (3) starts the monitoring function of the agent. If an error is detected during execution of the application, the error number is returned in `errno` and *Surrogate Manager#1* is notified about the failure. If the application terminated normally, then the result is collected from file *output*, (5), returned to the *Surrogate Manager#1*, (6), the watchdog process of the host daemon that was monitoring this agent is terminated, (7), and finally the agent itself is terminated, (8).

Agent#2

The *agent#2* agent is sent over to machine ***y***. This agent does exactly the same functions as *agent#1* but at machine ***y***.

Example 2.

Let assume that the user specifies the application to be run on 3 machines and the voting strategy to be followed is simple majority. The agents and the Surrogate Manager for this mode are built by modifying already existing components.

Surrogate Manager#2

The pseudo-code *Surrogate Manager#2* is shown in Figure 9. A *Surrogate Manager#2* is created by including new primitives to the *Surrogate Manager#1* (introduce changes are marked as a bold text)

```
notify(mc x,trip,mc y,mc z);                                  (1)
notify(mc y,trip,mc x,mc z);                                  (2)
notify(mc z,trip,mc x,mc y);                                  (3)
install(mc x,agent#1);                                        (4)
install(mc y,agent#2);                                        (5)
install(mc z,agent#3);                                        (6)
monitor(mc x,agent#1);                                        (7)
monitor(mc y,agent#2);                                        (8)
monitor(mc z,agent#3);                                        (9)
wait(agent#1,request_to_send_code);                           (10)
wait(agent#2,request_to_send_code);                           (11)
wait(agent#3,request_to_send_code);                           (12)
send(my_app.c,mc x,src);                                      (13)
send(my_app.c,mc y,src);                                      (14)
send(my_app.c,mc z,src);                                      (15)
wait(response_from_agent#1);                                  (16)
wait(response_from_agent#2);                                  (17)
wait(response_from_agent#3);                                  (18)
if((read_failure(agent#1,errno1) && read_failure(agent#2,errno2))
|| (read_failure(agent#2,errno2) && read_failure(agent#3,errno3))
|| (read_failure(agent#3,errno3) && read_failure(agent#1,errno1)))
/* all this implying that at least two of the 3 have failed */
      notify_FTM_failure(surrogate#2, errno1 || errno2 || errno3); (19)
else
      { // The application terminates normally
      a = get_result(agent#1);                                (20)
      b = get_result(agent#2);                                (21)
      c = get_result(agent#3);                                (22)
      final_result = vote(a,b,c,2,3,EXACT);                   (23)
      if (final_result == NULL)
            notify_FTM_failure(surrogate#2,NO_AGREEMENT);     (24)
      else
            notify_USER(surrogate#2,final_result);            (25)
      }
```

Figure 9: A Surrogate Manager#2 - pseudo-code

Agents#1-3

The *agents#1-3* will execute the code in each of the three machines (in the current configuration, *x, y* and *z*). The code for each of these agents is the same as that of the *agent#1*.

6 Prototype System Implementation

To demonstrate the capabilities of Chameleon, the prototype of the environment with the envisaged structure and a subset of the agent library has been implemented around four machines connected to the high-speed Myrinet LAN. The small network used consists of two Sun

workstations running Solaris OS and two Intel PCs running Linux and connected via a Myrinet switch. One of the machines in the network has been dedicated for running FTM.

Figure 10 and Figure 11 illustrate a complete execution scenario of the user application in the Chameleon environment. Figure 10 gives the existing testbed configuration. Once the environment is activated, the *system initialization agent* is invoked by the FTM which creates the configuration table. The FTM also invokes the *heartbeat agent* with a default granularity of a fixed time interval and installs the *host daemons* on the 3 hosts in the network. This setup is shown in Figure 11(ii).

Figure 11(iii) shows the user communication with the FTM when he submits an application for execution. The application source is brought to the FTM which decides on the application execution mode. The Figure 11(iv) shows the decisions taken by the FTM and the generation of the *requisite agents* for setting-up the desired execution mode. In the example, the FTM decides on a duplicated execution mode in the fail-safe (ie. both primary and secondary must agree for success) and centralized (ie. voting done at the FTM, contrary to agent at primary collecting the results and deciding) mode. Two agents (agent#1-2) and the Surrogate Monitor are invoked. In this implementation, the Surrogate Manager is located on the same node as FTM (Sun Workstation - "dusek"). Since all the agents are already existing in the library, the FTM does not need to build up or reengineer any existing agent. The agents are installed at the appropriate machines (Intel PC - "intel2" and Sun Workstation - "mahler" on Figure 11(iv)), the application code is taken over to the two machines participating in the duplicated execution mode and execution is started.

The agents monitor the application while the host daemons monitor the agents. This scenario is shown in Figure 11(v). Figure 11(vi) shows the case when the application terminates normally. The results are brought back to the Surrogate Monitor, where it is voted upon and the finally output is communicated to the user. Figure 11(vii) demonstrates the case when the program has an abnormal termination on one of the machines (intel2) and returns an error code. Since the decided mode was fail-safe the application is deemed to have failed and user is notified accordingly. The described scenario demonstrates already implemented features of Chameleon.

Not all details are included (e.g., nodes communicate via TCP/IP protocol). The intention is to demonstrate the overall concept and to show the feasibility of the proposed approach.

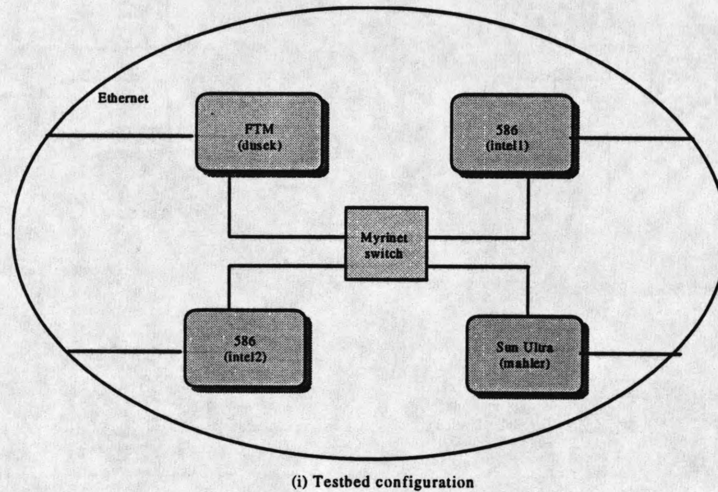
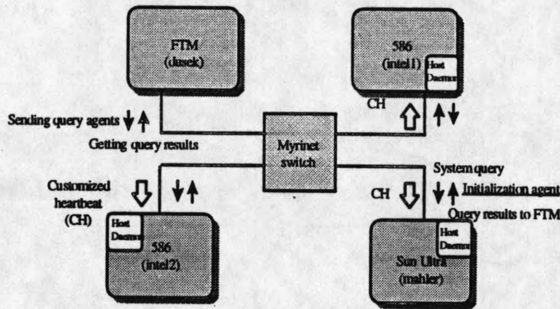


Figure 10: The Application Execution in Chameleon Environment - The Environment

System Configuration Table

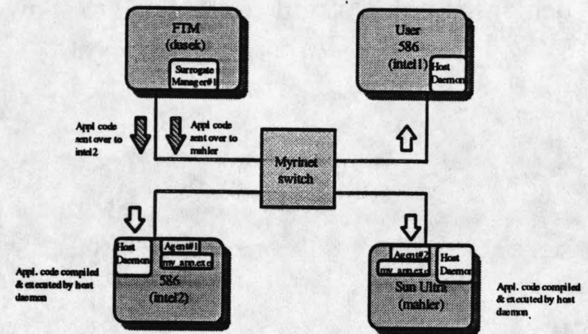
Name	Address	Architecture	OS	Physical Mem	VM
Intel1	192.17.5.30	586	Linux	48 MB	240 MB
Intel2	192.17.5.147	586	Linux	32 MB	240 MB
Mahler	192.17.5.126	Sparc	SunOS5.5	64 MB	263 MB



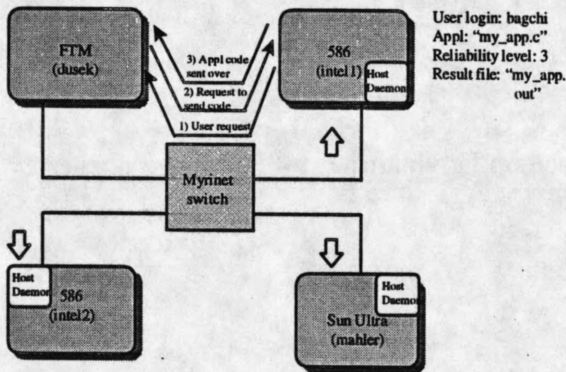
(ii) After initialization & heartbeat agent activation

Runtime Application Table

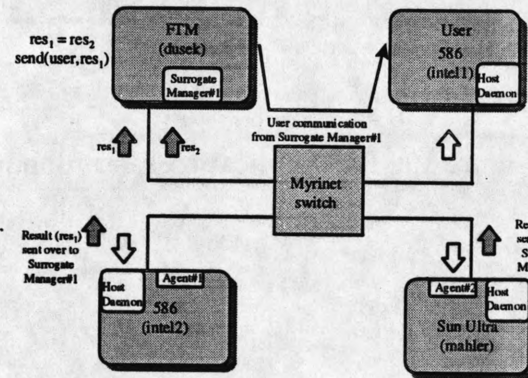
User	Application	Surrogate	Agents
bagchi	my_app.c	#1	#1, #2



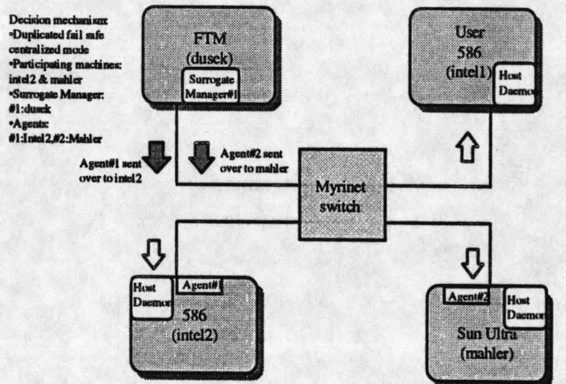
(v) Program Execution and Monitoring via Agents



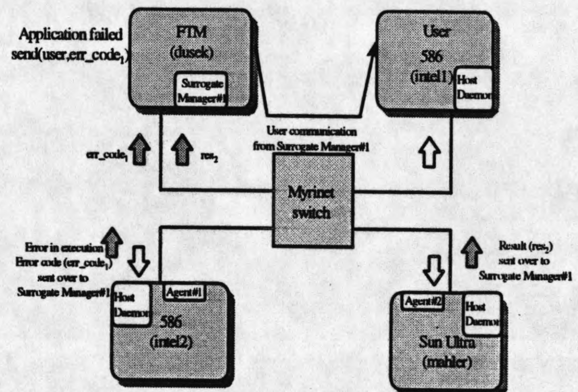
(iii) User Communication & Query Agent



(vi) Normal Program Termination



(iv) Decision at FTM & Creation and Deployment of Agents (for Duplicated Execution)



(vii) Abnormal Program Termination

Figure 11: An Example Application Execution Scenario in Chameleon

7 Summary and Scope for Future Work

In the paper we have presented Chameleon, a flexible fabric for providing configurable fault-tolerance levels for user applications in a heterogeneous networked environment. Chameleon employs an unique master entity for coordinating application execution at the desired quality of service (QoS) called the Fault Tolerance Manager (FTM). The FTM identifies the configuration to execute the user application and designates reliable Agents and Surrogate Manager which migrate through the network with the task to set up the environment for the particular execution mode and to monitor the application execution. The environment is dynamic and adaptive to varying criticality requirements from one application to another as well as within the same application. It offers flexibility in employing detection and recovery techniques by allowing the provided software libraries of elementary building blocks and agents to be incrementally upgraded by both the Fault Tolerance Manager and the user.

As existing machine and network configurations need to support a broad range of applications with different reliability requirements, environments like Chameleon would be powerful in embedding such support in already existing systems. The environment has the added attraction of being able to support a wide spectrum of heterogeneous platforms and networks. We believe that the mobile agents provide a useful abstraction for migration of execution to remote platforms for supporting the user's needs. At the same time the environment code is "thin" enough to be integrated as a layer on top of existing operating systems.

The work on Chameleon is still in an initial phase. We have a prototype implementation with the FTM, a limited set of agents and host daemons. Further work needs to be done on the process of re-engineering of available agents to create new agents which will meet different user demands. Substantial future work will be directed towards setting up the general agent framework to make it more flexible to the application requirements. A graphical interface needs to be incorporated into the environment which will enable the user to monitor his application as well as interact with the FTM. The possibility of migrating the environment to different network protocols with varying reliability levels depending on the application needs opens up a broad scope of future work. Simultaneously with the actual implementation, work is also going on in simulating the environment using the process based simulation environment, DEPEND.

ACKNOWLEDGMENT

This research was supported by NASA under grant NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

REFERENCES

[1] Birman K.P., "The Process Group Approach to Reliable Distributed Computing," Communications of the ACM, Vol.36, No.12, 1993.

[2] Birman K.P., R. van Renesse, "Reliable Distributed Computing with the Isis Toolkit," IEEE Computer Society Press, Los Alamitos, California, 1994.

[3] Dolev D., D. Malki, "The Transis Approach to High Availability Cluster Communication," Communications of the ACM, Vol. 39, No. 4, 1996.

[4] Maffeis S., "Piranha: A CORBA Tool for High Availability," IEEE Computer, April 1997.

[5] Moser L.E., P.M.Melliar-Smith, D.A.Agarwal, R.K.Budhia, C.A.Lingley-Papadopoulos. "Totem: A Fault-Tolerant Multicast Group Communication System," Communications of the ACM, Vol. 39, No. 4, 1996.

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

[7] Powell D., ed., "Delta-4: A Generic Architecture for Dependable Distributed Cluster Communication," Springer-Verlag, Berlin and New York, 1991.

[8] van Renesse R., K.P. Birman, S. Maffeis, "Horus: A Flexible Group Communication System," Communications of the ACM, Vol. 39, No. 4, 1996.