Incorporating Reconfigurability, Error Detection and Recovery into the Chameleon ARMOR Architecture


Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
Chameleon is a software infrastructure for providing adaptive fault tolerance in distributed systems. In Chameleon, reliability is provided through the use of entities called ARMORs, independent processes that control all Chameleon operations. In this report we present: (i) A detailed description of the reconfigurable architecture of Chameleon with emphasis on static and dynamic reconfigurability of the system. The reconfigurability features allow new, uploaded functionality to be incorporated dynamically into active ARMORs. (ii) A hierarchical framework of error detection mechanisms/algorithms for making ARMORs failure resilient, thus providing fault tolerance to the SIFT layer of a distributed environment. Different levels of error detection can be selectively turned on or off, depending on the amount of overhead that can be tolerated by the application. (iii) The results of availability analysis of Chameleon using Markov modeling, which shows that the Chameleon architecture satisfies the key design principle, i.e., there is no single point of failure. The analysis also demonstrates that the Daemon is the most critical component in Chameleon, from the point of view of application availability.
Incorporating Reconfigurability, Error Detection and Recovery into the Chameleon ARMOR Architecture


Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign

1998
# TABLE OF CONTENTS

## INTRODUCTION ........................................................................................................... 5

## OVERVIEW OF THE CHAMELEON INFRASTRUCTURE .................................................. 7
1.1 Hierarchical Composition: A Basis for Reconfigurability ........................................... 7
1.2 Message Subscription and Delivery .............................................................................. 9
1.3 Inter-Armor Communications ......................................................................................... 10
  1.3.1 ARMOR and Daemon Interactions ........................................................................... 10
  1.3.2 Daemon-to-Daemon Interactions ........................................................................... 12
1.4 Message Acknowledgement Strategies ........................................................................... 13

## IMPLEMENTATION OF SPECIFIC ARMORS ................................................................. 16
1.5 Daemon Implementation ................................................................................................. 16
  1.5.1 ARMOR Process Management (daemon_armor_proc_mgmt_t) .................................... 16
  1.5.2 Named Pipe Management Element (daemon_np_mgmt_t) ......................................... 17
  1.5.3 TCP/IP Connection Element (daemon_tcp_mgmt_t) .................................................. 17
  1.5.4 Remote ARMOR Location Cache Element (daemon_armor_loc_mgmt_t) .................... 18
  1.5.5 ARMOR Installation Coordinator (daemon_armor_install_t) ..................................... 19

1.6 Manager Implementation ............................................................................................... 19
  1.6.1 General Manager Functionality ................................................................................. 20
    1.6.1.1 ARMOR Location Table (mgr_armor_loc_db_t) ................................................... 20
    1.6.1.2 ARMOR Installation Coordinator (mgr_armor_install_t) ...................................... 21
  1.6.2 Surrogate Manager Functionality .............................................................................. 21
    1.6.2.1 Application Configuration Parameters Element (app_param_t) ........................... 21
    1.6.2.2 Execution ARMOR Management Element (sm_exec_mgmt_t) ............................. 21

1.7 Execution Armor Implementation ................................................................................... 22
  1.7.1 Application Configuration Parameters Element (app_param_t) .................................. 22
  1.7.2 Application Process Management (app_spawn_t) ....................................................... 23
  1.7.3 Application Named Pipe Management (app_np_mgmt_t) .......................................... 23

## EXTENSIONS FOR MPI .................................................................................................... 24
1.8 Chain-of-Command for MPI Applications ....................................................................... 24
1.9 Launching an MPICH Application ................................................................................ 25
1.10 MPI Execution Armor Management Element (mpi_sm_exec_mgmt_t) ........................... 27

## CONCEPTS IN ERROR DETECTION .................................................................................. 30
HIERARCHICAL ERROR DETECTION IN ARMORS

1.11 ARMOR SIGNATURES
1.11.1 Control Signatures
1.11.1.1 Control Signature Generation and Propagation
1.11.2 Data Signatures
1.12 ARMOR HIERARCHY
1.12.1 ARMOR State
1.12.2 Superset Relationship in ARMOR State
1.12.3 Active or Passive Replicas
1.12.4 Distribution of ARMOR Signatures within the Hierarchy

AVAILABILITY ANALYSIS OF CHAMELEON

1.18 AN OVERVIEW OF THE MODEL
1.19 DESCRIPTION OF THE MODEL
1.19.1 Model with all Failures Recoverable
1.19.2 Model with Unrecoverable Failures
1.20 ANALYSIS

RELATED WORK

REFERENCES
APPENDIX A: INSTANTIATING ELEMENT AND ARMOR OBJECTS ......................................... 63
A.1 INSTANTIATING WITHIN AN ARMOR ........................................................................... 63
A.2 ELEMENT AND ARMOR FACTORIES ......................................................................... 63
A.3 ADDING ELEMENTS TO A COMPOUND ....................................................................... 64
APPENDIX B: MESSAGE ALIASING ................................................................................. 65
B.1 A REVIEW OF MESSAGE DELIVERY ............................................................................ 65
B.2 “CONNECTING” ELEMENTS TOGETHER ..................................................................... 65
1 Introduction

A common thread that runs throughout all discussions of the Chameleon architecture is the notion of flexibility. By having a flexible architecture, we allow the basic services and service providers of the SIFT layer to change over time. Being that Chameleon offers its fault tolerance services through ARMORS, our notion of flexibility essentially extends to having a flexible ARMOR architecture.

Since fault tolerance requirements may change from mission to mission, we allow the specific ARMORS that you use and the configuration of these ARMORS to change from mission to mission as well. For example, in one mission, you may choose to have the Fault Tolerance Manager (FTM) run in a primary-backup configuration, while in another mission, you may choose to have the FTM triplicated; in others, you may choose to distribute much of the FTM functionality among several surrogate managers. Fundamentally, we strive to give you the flexibility to compose ARMORS from the collection of services that we provide, tailoring your suite of ARMORS to match the requirements of a given mission. In particular, we attempt to avoid situations in which we mandate that one and only one type of ARMOR be capable of providing a specific service. Rather, we try to encapsulate the service into modules that may be included in virtually any ARMOR.

To help make the task of assigning functionality to ARMORS more manageable, we provide three groups of ARMORS that comprise the core functionality of Chameleon needed by any particular set of ARMORS:

- **Managers.** Managers are ARMORS that can initiate the remote installation of other ARMORS, as well as recover from failures in subordinate ARMORS.
- **Daemons.** Daemons are ARMORS that are installed on each node in the Chameleon network. They locally-install new ARMORS, serve as routers for ARMOR-to-ARMOR communications, and interact with the locally-installed ARMORS (such as detecting a failed ARMOR process and notifying the failed ARMOR’s manager).
- **Other ARMORS.** These ARMORS which are neither managers nor daemons provide other services in the Chameleon environment, such as the monitoring of a user application process (Execution ARMOR).

Extending the notion of flexibility even further, we not only allow the makeup of ARMORS to change from mission to mission, but also allow the ARMORS to dynamically reconfigure themselves during run-time. For example, we would like to allow the FTM to change from simplex to TMR-execution and back to simplex during the lifetime of the mission without having to take down the FTM. Reconfigurability could also be used to change the way managers recover from failures over time. The reconfigurability features also allow new, uploaded functionality to be dynamically incorporated into active ARMORS without bringing those ARMORS down.

Since ARMORS are the key components through which much of the functionality and fault-tolerance of the Chameleon environment is derived, it is necessary to make the ARMORS, themselves, resilient to failures. The second half of the paper introduces a hierarchical approach to error detection in ARMORS. A wide range of failures—from benign to malicious—are considered and a framework presented to tolerate the whole range of failures using techniques of varying latency, overhead and periodicity of invocation. In keeping with the philosophy of flexibility in Chameleon, the different levels of error detection may be selectively turned on or off depending on the amount of overhead that can be tolerated by the application. Generally, the addition of the fault-tolerance features involves addition of element(s) to corresponding ARMORS which respond to certain activation triggers, and which may be switched on or off depending upon the application needs.

This paper begins by describing some of the significant features of the reconfigurable and composable Chameleon architecture. We follow this section with a general discussion of ARMOR communications, a detailed look at some specific ARMORS that have been implemented using the reconfigurable and
composable architecture, and an overview of some of the extensions that have been made to support MPI applications in the Chameleon environment. Finally, we delve into our support for error detection and diagnosis at the ARMOR level, and we show how these mechanisms are constructed in a hierarchical manner that take advantage of the reconfigurable and composable architecture of Chameleon.
2 Overview of the Chameleon Infrastructure

As mentioned in the introduction, Chameleon provides fault tolerance services to an application through the use of ARMORS. Several different kinds of ARMORS exist in the Chameleon system with varying behavior. Some are managers that must be capable of installing other ARMORS and recover from failures in subordinate ARMORS. Some are daemons that must route messages to other ARMORS. Some are Execution ARMORS that oversee an application process and provide a Chameleon interface to the application process. And these are just the ARMORS found in the early implementation of Chameleon—as the environment matures and becomes more sophisticated, the library of ARMORS will most likely grow to be large and feature-rich.

Despite the diversity among ARMORS in the Chameleon environment, a concerted effort has been made to standardize some key characteristics of all ARMORS. A few of these characteristics follow:

- ARMORS should be reusable across several different fault-tolerant execution strategies wherever possible. For example, the same Execution ARMORS ideally should be able to work with MPI processes, standalone processes, replicated processes, and so on with little customization.

- ARMORS should be highly modular, not only for reusability and debugging purposes, but also to support the notion of ARMOR reconfigurability. Reconfiguring an ARMOR with different behavior obviously becomes much easier if most of the ARMOR’s behavior is modularized into separate components.

- ARMORS should interact with each other through message passing. Message passing gives us a sufficiently generic interface through which existing and future functionality may be invoked easily.

To meet these requirements, we have devised a hierarchical ARMOR architecture that provides a framework for implementing ARMOR behavior. Under this architecture, messages are processed by replaceable objects called elements. Elements form the foundation for more complex components including compounds and, ultimately, ARMORS. This section presents a high-level overview of the Chameleon infrastructure, including the hierarchical composition of ARMORS and ARMOR communications.

2.1 Hierarchical Composition: A Basis for Reconfigurability

The key components in the ARMOR architecture can be represented in the object-oriented class hierarchy found in Figure 1. Note that element_t is the root of the entire class structure. This is important, because all Chameleon components share the same interface—namely the interface of an element. The derived classes in Figure 1 can be viewed as simply being specialized elements.

Simple elements (instances of the element_t class) are the most basic building blocks in the ARMOR architecture. The primary responsibility of an element is to process specific messages—different types of elements process different types of messages. Several elements may be grouped together to form compounds. In fact, a compound groups together objects of any class derived from element_t, so a compound may be made up of simple elements, other compounds, or even entire ARMORS. It is the potential nested nature of compounds that give Chameleon ARMORS their hierarchical composition.

![Class hierarchy for the ARMOR architecture](image)
An instance of the `armor_t` class is a special type of compound that may be executed as a separate process and that may send and receive messages to other ARMORS. The other remaining classes in Figure 1 are specific kinds of ARMORS and will be discussed in later sections.

Operational elements in the Chameleon environment cannot exist in isolation—they must be part of an `armor_t` object in order to be “executing” (i.e., capable of processing messages). When an element is created, it is created as part of an existing compound. For typical ARMORS consisting of a single-level composition hierarchy, the `armor_t` object serves as the sole compound in the ARMOR. This setup is depicted in Figure 2(a). A multi-level composition hierarchy—one where an `armor_t` object is made up of one or more compounds—is shown in Figure 2(b). Because an element (be it an instance of `element_t` or `compound_t`) must be created as part of a parent compound, the outermost `armor_t` “shell” is usually created first and then populated with its constituent elements. It so happens that the constituent elements can completely specify the ARMOR’s behavior, thus giving a convenient form for specifying ARMOR behavior.

![Figure 2: (a) Typical single-level composition of an ARMOR; (b) Multi-level composition using a combination of compounds and elements](image)

Now we have the fundamental concept of having ARMORS that are composed of one or more basic building blocks. You will recall that another design goal of Chameleon is to allow an ARMOR’s composition to dynamically change at run-time. Most of the issues that concern this reconfigurability property can be found at the `compound_t` level of the class hierarchy. Simply put, a compound has ultimate responsibility over its constituent elements, and an element can only be accessed through its parent compound. With this in mind, a compound has two primary responsibilities:

1. Manage its composition through the insertion and removal of constituent elements.
2. Deliver incoming messages to one or more of its constituent elements.

To support dynamic reconfigurability, a compound must coordinate all composition changes so that its elements are not removed or changed while being used. Requiring that all element interactions be done through the parent compound ensures that this condition holds. In fact, the level of indirection that compounds provide to their elements is a fundamental concept in the overall ARMOR architecture:

*Elements within a compound cannot directly access other elements, even within the same compound. All access must be through the parent compound. Since all functionality is accessed indirectly, the underlying behavior (element composition) of the ARMOR may be dynamically changed without affecting other elements.*

This concept also has another ramification for the local ARMOR and remote ARMORS alike—a uniform interface to an ARMOR’s functionality. Specifically, the ARMOR architecture forces that most (if not all) of the ARMOR’s behavior be accessible through message passing. The reason for having a message passing interface for use by remote ARMORS should be fairly obvious. But the fact that elements within the same ARMOR cannot directly interact with each other mandates that a similar interface be provided for intra-ARMOR behavior invocation. We will see later that these intra-ARMOR messages simply amount to an
indirect function call; nevertheless, the important point to take away from this discussion is that messaging provides a universal interface to functionality within the ARMOR and to functionality residing in remote ARMORS.

### 2.2 Message Subscription and Delivery

All of this reliance on messaging leads us to the second primary responsibility of compounds—message delivery. When elements are added to a compound, the elements typically subscribe to the message types that they want to process. Then, when the parent compound receives an incoming message, it consults its subscription list to forward the message to the appropriate elements. Forwarding to an arbitrary element is possible because all instances of `element_t` (and derived classes) have a virtual `process_message()` function that may be overridden to give an element the ability to process specific message types.

But how does the `compound_t` object receive the incoming message to deliver to its constituent elements? Since a compound is a derived type of `element_t`, it also has a `process_message()` of its own. So, messages are delivered to compounds just like any other element—from the parent compound. In Figure2(b), for example, the middle-level compound has messages delivered to it by the parent `armor_t` object. Note for this to work, all subscription requests are propagated up the parental chain. If Element 1 subscribes to message types M1 and M2, Element 2 subscribes to M3, Element 4 subscribes to M4, and Element M5 subscribes to M5, then Table 1 lists the subscription lists for each compound.

**Table 1: Example subscription list for Figure 2(b)**

<table>
<thead>
<tr>
<th>ELEMENT</th>
<th>SUBSCRIPTION LIST</th>
</tr>
</thead>
</table>
| Compound A | M1 → Element 1  
            | M2 → Element 1  
            | M3 → Element 2 |
| Compound B | M4 → Element 4 |
| ARMOR     | M1 → Compound A  
            | M2 → Compound A  
            | M3 → Compound A  
            | M4 → Compound B  
            | M5 → Element 5 |

Now we just have to worry about how the top-level `armor_t` object receives messages—once the `armor_t` gets a message, it simply propagates the message down the composition hierarchy. Messages arrive at the top-level `armor_t` object either from within the local ARMOR or from a remote ARMOR. In both cases, the message is passed to the `armor_t::process_message()` function. For intra-ARMOR messages, elements directly pass the message to the `armor_t::process_message()` function. For messages from remote ARMORS, specialized elements within each ARMOR read incoming messages from some sort of interprocess communication channel (e.g., named pipes for ARMORS and daemons, TCP/IP sockets for daemons, etc). These same elements then directly pass the message to the `armor_t::process_message()` function. More details about this procedure will follow, but the most important observation to take away from this discussion is that the `armor_t::`-
process_message() function is called for both intra- and inter-ARMOR messages, thus providing a centralized location to handle all incoming messages.

2.3 Inter-ARMOR Communications
We have seen how ARMORS are composed of modular elements and how these elements process incoming messages. Now, we will give a high-level overview of inter-ARMOR communication—specifically, we will take a look at the role that daemons play in the Chameleon message passing system.

A daemon executes on every node in the Chameleon network, and are typically launched at boot time by an initialization script. Since a unique daemon executes on each node, the daemon ID number serves as a convenient, network-independent way of identifying nodes. Keep this in mind when you encounter statements such as “a manager installs an ARMOR on Daemon X” and “the FTM allocates Daemons X, Y, and Z for use in executing an application.”

![Figure 3: System view of typical ARMOR installations, complete with ARMOR-to-daemon and daemon-to-daemon interconnects](image)

As daemons are just special kinds of ARMORS, they are built around the ARMOR architecture presented earlier and contain specialized elements that give them daemon-specific functionality. From a messaging point of view, these elements make the daemons behave as routers that pass messages back and forth between sending and receiving ARMORS.

All non-daemon ARMORS that execute in the Chameleon system are connected to the local daemon through a named pipe. Daemons connect to each other through the network as illustrated in Figure 3. At the highest level, message routing is conceptually simple. If ARMOR A1, for example, wants to send a message to ARMOR A12, then the message will take the path A1 -> Daemon 1 -> Daemon 6 -> A12. We will now take a closer look at this process, beginning with the steps taken by the sending ARMOR to transmit an outgoing message.

2.3.1 ARMOR and Daemon Interactions
Non-daemon ARMORS take the basic armor_t object and add an element that makes them capable of sending and receiving messages. This is the only mandatory element for non-daemon ARMORS, and the nondaemon_t class in Figure 1 automatically adds this element to the basic armor_t structure. The extra element, np_mgmt_t, oversees all activity associated with the named pipe connecting the ARMOR to the local daemon. As Figure 4 illustrates, the np_mgmt_t is the only element within the ARMOR directly connected to the named pipe.
During its initialization, the np_mgmt_t element creates a new thread to monitor the named pipe for incoming messages. Upon receiving a message, the element passes the message to the armor_t::process_message() function to initiate the proper message delivery as described above.

For sending outgoing messages, other elements within the ARMOR indirectly invoke the message sending capabilities of the np_mgmt_t element, thus keeping with the spirit of the ARMOR architecture so that the np_mgmt_t element may be replaced with another element in the future (possibly so that another form of IPC can be used between the ARMOR and daemon). To initiate a message send, an element calls its element_t::send_message() function. This function simply passes the message to the top-level armor_t::process_message() function for further processing. At this point, the message appears to the ARMOR exactly as if it came from the daemon over the named pipe connection.

![Figure 4: ARMOR connection to the local daemon](image)

The armor_t::process_message() function first checks to see if the final destination of the message is the local ARMOR or a remote ARMOR (remember, source and destinations are specified by ARMORS, not by specific elements within the ARMOR). If the message is intended for the local ARMOR, then the armor_t::process_message() passes the message to top-level ARMOR's compound_t::process_message() function for delivery to the appropriate element(s).

If the message is to be sent to a remote ARMOR, then the armor_t::process_message() function must forward the message to the np_mgmt_t element (or whatever element interacts with the local daemon). You will recall that we must access this element through the message delivery services of the compound in order to provide the proper level of indirection and abstraction required by the reconfigurable ARMOR architecture. With this in mind, Chameleon mandates that the np_mgmt_t element (or equivalent) subscribe to the MSG_TRANSMIT_MESSAGE message type. Now, whenever the top-level armor_t object receives a message destined for another ARMOR (such as when one of armor_t's elements sends an outgoing message), the armor_t::process_message() function creates a new message of type MSG_TRANSMIT_MESSAGE and delivers it to the appropriate element using the normal message delivery services of the underlying compound. The body of the new MSG_TRANSMIT_MESSAGE is simply the original outgoing message.

So, to summarize, an element passes a message to the top-level armor_t::process_message() function through its own element_t::send_message() function. The ARMOR then delivers the message to all child elements for an intra-ARMOR message; otherwise, the ARMOR encapsulates the outgoing message in a MSG_TRANSMIT_MESSAGE message for delivery to the element responsible for transmitting outgoing messages.
2.3.2 Daemon-to-Daemon Interactions

Note that all of the routing of local and remote messages occurs at the armor_t level, implying that these rules apply equally well to daemons and non-daemons alike. Indeed, this is the case. Daemons, however, have an additional routing destination to consider. Whereas non-daemons simply forward all outgoing messages to the np_mgmt_t element (or equivalent), daemons can either forward outgoing messages through one of two channels:

1. Across one of its named-pipe connections to a locally-installed ARMOR
2. Across the network to a remote daemon for further routing.

As you might expect, daemons contain two mandatory elements to assist them in this routing task: daemon_np_mgmt_t and daemon_net_mgmt_t. Conceptually, Figure 5 illustrates these elements and their external connections in the Chameleon environment. These two elements receive messages and forward them to the armor_t level of the daemon in much the same way that the np_mgmt_t element receives and forwards incoming messages to the armor_t level for non-daemons. The daemon elements, however, each have several connections to monitor. Presently, each connection has a separate thread that monitors for incoming messages. Although inefficient, this solution is readily portable to all of the target platforms for Chameleon. For performance or resource usage concerns, we may develop platform-specific optimizations of these two elements in the future.

Having several connections makes the problem of routing outgoing messages a little more difficult, but we follow the same general strategy used for non-daemon ARMORS. When the armor_t: :-

---

1 Although the daemons that actually execute in the Chameleon environment have network-specific elements for remote communication (e.g., daemon_t_tcp_mgmt_t), we will abstractly talk about a daemon_net_mgmt_t element, as the exact protocols and networks are unimportant to the understanding of the high-level message routing in Chameleon.
process_message() function receives a message destined for a remote ARMOR (i.e., not the daemon), it still encapsulates the message in a MSG_TRANSMIT_MESSAGE message as before. For daemons, however, the daemon_np_mgmt_t element (or whatever element that is responsible for the local connections) subscribes to this message. Upon receipt of the MSG_TRANSMIT_MESSAGE message, the daemon_np_mgmt_t element checks to see if the destination ARMOR is one of the locally-installed ARMORS. If so, then the element transmits the message across the appropriate named pipe connection. If not, the element assumes the message is destined for a remote ARMOR and encapsulates the original message in a MSG_TRANSMIT_MESSAGE_REMOTE message. As expected, the daemon_net_mgmt_t element handles this message and selects the appropriate daemon through which the message should be routed2. Figure 6 flowcharts the entire procedure of routing an incoming message (regardless of whether the message came from within the daemon or from one of the ARMORS connected to the daemon). Keep in mind that the compound_t and armor_t objects in Figure 6 both refer to the daemon object—since daemon_t derives from armor_t and compound_t, the daemon can freely access the compound part of its functionality, as well as its base ARMOR functionality.

2.4 Message Acknowledgement Strategies

Now that we have a high-level overview of the message routing in the Chameleon environment, we can look at the four different kinds of acknowledgement models that are available for ARMOR-to-ARMOR communications. They are:

- non-blocking, unacknowledged send (element_t::send_message())
- receipt-acknowledged send (element_t::rsend_message())
- completion-acknowledged send (element_t::csend_message())
- request/reply semantics (element_t::send_request() and element_t::send_reply())

We will consider each of these in turn, initially focusing only on inter-ARMOR messaging.

Obviously, the non-blocking, unacknowledged send is the simplest of the four kinds of message sends. Using these semantics, the element_t::send_message() function returns as soon as the message has been sent across the local IPC channel. For non-daemon ARMORS, this means that send_message() returns after the np_mgmt_t element writes the message to the named pipe. This fact becomes apparent by looking at a trace of the function calls that occur when sending an outgoing message:

```cpp
element_t::send_message (msg)
parmor->process_message (msg)
compound_t::deliver (MSG_TRANSMIT_MESSAGE)
np_mgmt_t::process_message (MSG_TRANSMIT_MESSAGE)
named_pipe_t::write (msg)
```

Since these are all nested function calls, element_t::send_message() will not return until named_pipe_t::write() returns. Tracing the element_t::send_message() for elements within a daemon show similar results—the function does not return until the data is committed to the outgoing IPC channel (be it a named pipe or network connection). True to its intended behavior, the

---

2 Selecting the appropriate intermediary daemon will be discussed later in the document.
caller of `element_t::send_message()` does not know if the destination ARMOR received the message, or even if the message made it past the local daemon.

To address this problem, Chameleon offers two specific forms of acknowledged sends through the `element_t::rsend_message()` and `element_t::csend_message()` functions. The `rsend_message()` function will send an acknowledgment to the sender once the destination ARMOR has received the message, and `csend_message()` will send an acknowledgement to the sender after the destination ARMOR has completed processing the message. Neither of these functions returns until the acknowledgement has been received from the destination ARMOR.

Ideally, we would like the same message to be capable of being sent using any one of these three acknowledgement semantics. To do this, we need to consistently send and check for acknowledgements for all messages. We put this functionality at the `armor_t` level in the form of message pre-processing and post-processing.

The complete processing of inter-ARMOR messages can be divided into three important phases:

1. Message pre-processing (done in `armor_t::preprocess_message()`)
2. Message processing (done in `armor_t::process_message()`)
3. Message post-processing (done in `armor_t::retire_message()`)

Phase two consists of what we normally consider the true processing of the message (i.e., it during this phase that the `compound_t` functionality of the ARMOR delivers the message to the appropriate elements). During pre-processing of an incoming inter-ARMOR message, the following operations are performed:

- If the ARMOR is the final destination of the message and the message was sent using the `rsend_message()` function (indicated by a field in the message header), then send an acknowledgement to the sender.
- If the incoming message is an acknowledgement (as indicated by a field in the message header), then unblock the sender's `rsend_message()` or `csend_message()` function call.

Similarly, the ARMOR sends an acknowledgement back to the sender during post-processing if the message was sent using the `csend_message()` function.

Finally, the request/reply semantics extend the `csend_message()` acknowledgement scheme to include response data in the acknowledgement. Typically, an element will need to request information from another ARMOR (e.g., a daemon may request that the FTM provide it with the location of a remote ARMOR). For instances in which a response is needed before the element may continue, the `element_t::send_request()` function may be used. When the element calls `send_request()`, it specifies a buffer into which the reply of the request should be copied. The remote element processing the request uses the special `element_t::send_reply()` function when replying to a request.

During the pre-processing phase of the reply message, the ARMOR copies the response data from all replies into the proper buffer (if any). The pre-processing phase then unblocks the caller of `send_request();` thus, the data is freely available to the requestor when the `send_request()` function returns.

For inter-ARMOR messages, the calling thread of `rsend_message()`, `csend_message()`, and `send_request()` must block until the ARMOR receives the associated acknowledgement or reply. For intra-ARMOR messages, the message transmission, delivery, and processing all occur within the same thread of execution as the calling thread. When the function returns, therefore, the caller knows that the
processing of the intra-ARMOR message has completed. Again, tracing through the function calls will help clarify this point:

\[
\begin{align*}
\text{element}_t & \text{::send}_\text{message} (\text{msg}) \\
\text{parmor}\rightarrow & \text{process}_\text{message} (\text{msg}) \\
\text{compound}_t & \text{::deliver} (\text{msg}) \\
\text{element}_t & \text{::process}_\text{message} (\text{msg})
\end{align*}
\]

When \text{element}_t\text{::process}_\text{message}() returns (i.e., after the element that ultimately receives the message has completed processing the message), all higher-nested functions return. Under these circumstances, the behavior of the \text{rsend}_\text{message}() is not that interesting; indeed, for intra-ARMOR messages, \text{send}_\text{message}(), \text{rsend}_\text{message}(), \text{csend}_\text{message}() all behave like a \text{csend}_\text{message}() function—the functions only return when the message has been completely processed by the receiving element.
3  Implementation of Specific ARMORS

The previous section gave a general overview of the underlying Chameleon infrastructure. In this section, we will look at some of the specific ARMORS that have been implemented using this infrastructure. Specifically, we will look at the elements that have been designed to provide the bulk of specific ARMOR functionality.

3.1  Daemon Implementation

In addition to routing messages as presented in the previous section, the daemon also locally installs new ARMORS. To fulfill its responsibilities, each daemon contains the following elements:

- ARMOR Process Management Element
- Named Pipe Management Element
- TCP/IP Connection Element
- Remote ARMOR Location Cache Element
- ARMOR Installation Coordinator

Each of these elements will be considered in turn.

3.1.1  ARMOR Process Management (daemon_armor_proc_mgmt_t)

This element spawns a new ARMOR process whenever a new ARMOR needs to be installed on the local node. In order to install any arbitrary ARMOR, the daemon_armor_proc_mgmt_t element uses an ARMOR factory and an ancillary program called create_arm. The exact flow-of-control follows:

1. The daemon_armor_proc_mgmt_t receives the MSG_DAEMON_SPAWN_ARMOR message that contains the ID to give the new ARMOR, the ID of the new ARMOR’s manager (usually the ARMOR making the request to install the new ARMOR), and the type of the new ARMOR.

2. The element spawns a new process that executes the create_arm program, passing the information found in the MSG_DAEMON_SPAWN_ARMOR message as command-line arguments.

3. The create_arm program uses an ARMOR factory to create a new ARMOR object (a descendant of the arm_t class in Figure 1).

4. The create_arm program then calls the arm_t::init() function for the new ARMOR object, causing the ARMOR to be initialized. Usually, the initialization routine adds the elements needed to become a specific ARMOR and “connects” the elements together so that they function properly. See Appendix A.3 for more details on the role that the init() function plays during element creation (and, in this case, ARMOR creation).

5. The create_arm program exits only after the ARMOR object has “terminated,” which usually happens when the daemon uninstalls the ARMOR.

Note that since the create_arm program instantiates the new ARMOR object in a child process, the ARMOR executes in its own address space (under Linux, at least). After spawning the new process, the daemon_armor_proc_mgmt_t launches a new thread that calls waitpid() on the child process to detect termination of the ARMOR process. Finally, the element stores relevant information about the new ARMOR process (process ID, thread ID of the waitpid() thread, etc.) in a table for future use.
3.1.2 Named Pipe Management Element (daemon_np_mgmt_t)

The daemon_np_mgmt_t element performs the following functions for the daemon:

- Establishes a named pipe connection to a new ARMOR process.
- Maintains a table of all ARMORS installed on the node through the daemon.
- Monitors each named pipe for incoming messages from the ARMORS.
- Forwards inter-ARMOR messages to locally-installed ARMORS when applicable.
- Monitors an auxiliary named pipe for incoming messages from the outside world.

The first three actions are triggered by a new ARMOR being installed by the daemon. When the element receives the MSG_DAEMON_CONNECT_NEW_ARMOR message, it:

1. Creates and opens a new named pipe for communicating with the new ARMOR process.
2. Adds the ARMOR and named pipe pointer to its list of locally-installed ARMORS.
3. Spawns a new thread to monitor the named pipe for incoming messages from the ARMOR (a new thread is needed because we are currently using a blocking read on the named pipe).

Most of the behavior associated with writing to and reading from the named pipe connections has already been covered in section 2.3.

In addition to the standard named pipes connecting the local ARMORS to the daemon, the daemon also creates an auxiliary named pipe for receiving incoming messages from the outside world (from non-ARMOR processes). For example, an early implementation of the Chameleon API for REE applications used the auxiliary named pipe to send information to the Chameleon system before it connected to the Execution ARMOR. Although the Chameleon API no longer uses auxiliary pipe for this purpose (in fact, nothing in the current implementation uses it at all), the ability for non-ARMOR processes to have a well-known entry point into the Chameleon system may prove to be beneficial in the long run. In any case, it can easily be removed if it proves to be unnecessary or insecure.

3.1.3 TCP/IP Connection Element (daemon_tcp_mgmt_t)

The daemon_tcp_mgmt_t element is a specific instance of the generic daemon_net_mgmt_t element described in section 2.3.2. As you might expect, this element manages the TCP/IP connections to other daemons in the Chameleon environment.

The central data structure for the daemon_tcp_mgmt_t element is a table that maps daemon IDs to IP addresses and port numbers. The element updates this table on demand with help from the master daemon. By our current conventions, the first daemon installed in the Chameleon environment becomes the master daemon. All other subsequently-installed daemons have the master daemon as their manager.

When the daemon needs to route an outgoing message to a remote daemon, it attempts to locate the destination daemon’s ID in its connection table. If it cannot find the remote daemon in its connection table, it requests the remote daemon’s IP address and port number from its manager (the master daemon). Under our current implementation, therefore, the master daemon should have an entry for each daemon in its connection table to satisfy location requests from other daemons.

As a result of this need, all daemons must register with the master daemon upon being installed (registration consists of sending the daemon’s IP address and port number on which the daemon accepts new connections). Consequently, the entity installing the daemons—be it the user, script, or other means—must pass the location of the master daemon as command-line parameters to the daemon installation program. After registration, the newly-installed daemon will know how to reach the master...
daemon (necessary for determining the location of other daemons for future communications) and the master daemon will have a record of the location of the newly-installed daemon.

In a somewhat separate issue, the TCP/IP protocols mandate that a separate connection be established for communicating with each remote daemon. In the interest of keeping a bounded number of open connections, we intend to have only \( n \) connections be active at any point in time. Although it is not currently implemented that way (a connection remains open once established), the idea is to have an older, infrequently-used connection be closed when a new connection needs to be made. With this in mind, the connection table has a field that stores the socket associated with open connections for each daemon in table. It should be noted that although connections may be opened and closed throughout the lifetime of the daemon, we do not envisage having to remove the IP address and port number associated with a remote daemon from the connection table.

Most of the other responsibilities of the `daemon_tcp_mgmt_t` element have already been outlined in section 2.3.2. To summarize, the `daemon_tcp_mgmt_t` performs the following functions and services:

- Keeps a table of the TCP/IP parameters needed to establish a connection with remote daemons; this table is updated from the master daemon on an as-needed basis.
- Keeps a small cache of open connections to remote daemons.
- Routes outgoing messages to remote daemons.
- Monitors each open connection for incoming messages.
- Accepts new connections from remote daemons.

As with the `daemon_np_mgmt_t` element, the `daemon_tcp_mgmt_t` element creates a per open connection that monitors for incoming from remote daemons. There is also a single thread that blocks on an `accept()` call to handle new connections from remote ARMORS.

### 3.1.4 Remote ARMOR Location Cache Element (daemon_armor_loc_mgmt_t)

The previous section and section 2.3.2 describe a significant portion of the message routing services that daemons perform for locally-installed ARMORS. There remains one detail in this procedure—namely, how the daemon on the sending end of an inter-ARMOR message knows the remote daemon to which the message should be forwarded. Remember, the message header contains only the ID of the destination ARMOR, not the ID of the daemon through which the destination ARMOR may be reached. The `daemonArmorLoc_mgmt_t` element fills this void.

Simply put, the `daemon_armor_loc_mgmt_t` element maintains a cache of frequently-used ARMOR-to-daemon mappings. Note that this information is also available to the destination ARMOR’s manager (i.e., a manager knows where it has installed its ARMORS). In this respect, the `daemon_armor_loc_mgmt_t` serves merely to optimize the routing of messages through daemons. If this element were not present, the daemon would be forced to request that the destination ARMOR’s manager provide this information for every message going to a remote node.

Although not currently implemented, we will need to make sure that an ARMOR’s entry in all `daemon_armor_loc_mgmt_t` caches be removed or invalidated when the ARMOR is uninstalled.

It should also be noted that since the information found in this cache is a duplicate of information already found in one or more managers in the Chameleon system, the daemons can easily regenerate the ARMOR-to-daemon mappings if needed. In fact, the cache is initially empty. When the daemon needs to route a message to a remote ARMOR for the first time, it makes a request to its manager that the ARMOR-to-daemon mapping be provided. Recall that the master daemon is the manager for most daemons. If the
master daemon cannot find the ARMOR-to-daemon mapping in its daemon_armor_loc_mgmt_t element, it asks its manager (usually the FTM) for the information. Since the FTM is the highest-ranking manager, it contains the ARMOR-to-daemon mappings for every ARMOR in the Chameleon environment.

3.1.5 ARMOR Installation Coordinator (daemon_armor_install_t)

Finally, the daemon_armor_install_t element oversees the installation of a new ARMOR by the daemon by performing the following steps upon receiving a MSG_DAEMON_ARMOR_INSTALL message:
1. Creates a new ARMOR process by sending a MSG_DAEMON_SPAWN_ARMOR message to itself.
2. Connects the new ARMOR process to the daemon through the MSG_DAEMON_CONNECT_NEW_ARMOR message.

![Diagram showing Coordinator-invoked functionality and Domino-invoked functionality](image)

These two steps indirectly invoke the services of the daemon_proc_mgmt_t element and the daemon_np_mgmt_t element. Graphically, the role of the coordinator may be depicted in Figure 7(a).

As an alternative to having a coordinator oversee the functionality invocation, we could have the daemon_proc_mgmt_t element subscribe to the MSG_DAEMON_INSTALL_ARMOR message as its primary input. The daemon_np_mgmt_t element—the element whose functionality must be invoked based on information obtained from the first element—can then subscribe to the completion message of the daemon_proc_mgmt_t element. The latter technique has the added advantage of one fewer element, but the disadvantage in that the functionality of the daemon_proc_mgmt_t element cannot be invoked without triggering the daemon_np_mgmt_t element—the domino effect, so to speak. Although the domino effect would be acceptable for this specific example (very rarely would you want to launch a new ARMOR process without it establishing a connection via a named pipe), the same cannot be said for any arbitrary multi-element interaction.

To clarify, the ARMOR installation currently utilizes a coordinator element—the alternative is presented here mainly as another design option.

3.2 Manager Implementation

In the Chameleon environment, managers are responsible for initiating the installation of ARMORS and for recovering from failed subordinate ARMORS. Although there are several specific kinds of managers, all managers must provide this functionality and, therefore, contain the following elements:

- ARMOR Location Table Element

---

3 Recovery of failed ARMORS has not been fully implemented.
• ARMOR Installation Coordinator

We have also implemented a specific type of manager—the surrogate manager—for overseeing an application executing in the Chameleon environment.

3.2.1 General Manager Functionality

In this section, we will discuss the two elements that are common to all managers. In fact, adding these elements to a generic arm_t “skeleton” turns the ordinary ARMOR into a manager.

3.2.1.1 ARMOR Location Table (mgr_amror_loc_db_t)

Managers keep the following information for all subordinate ARMORS:

• ARMOR ID
• Daemon on which the ARMOR is installed
• Immediate manager of the ARMOR

Note that by subordinate ARMORS, we mean not only those ARMORS directly installed by the manager, but all descendant ARMORS. For example, Figure 8 illustrates a typical manager-subordinate relationship. Here, we have an MPI application overseen by Surrogate Manager A, and a replicated MPI application overseen by Surrogate Manager B. The managerial scope of Surrogate Manager A contains Execution ARMORS 1 – 3. The scope of Surrogate Manager B, however, contains Surrogate Manager C, Surrogate Manager D, and the Voter ARMOR (those ARMORS directly installed by Surrogate Manager B), as well as Execution ARMORS 4 – 9. As a result, the mgr_armor_loc_db_t element for Surrogate Manager B will contain entries for all of these ARMORS within its scope. The FTM, by the same logic, will contain entries for every ARMOR installed in the Chameleon environment.4

Upon installing a new ARMOR, the manager not only updates its own mgr_armor_loc_db_t table, but also sends an update message to its manager, which recursively sends an update message to its manager, and so on, until the message reaches the FTM.

---

4 It is this feature that allows the master daemon to make requests to the FTM to determine an ARMOR-to-daemon mapping for any ARMOR in the system. Section 3.1.4 describes how the daemon uses this information.
3.2.1.2 ARMOR Installation Coordinator (mgr_armor_install_t)

Like daemons, managers contain an element that oversees the installation of a new ARMOR. Specifically, the mgr_armor_install_t element performs the following steps upon receipt of a MGR_MANAGER_INSTALL_ARMOR message:

1. Sends a MSG_DAEMON_INSTALL_ARMOR message to the appropriate daemon.
2. Updates the manager’s ARMOR location table via the MSG_MANAGER_REGISTER_ARMOR message.

3.2.2 Surrogate Manager Functionality

Surrogate managers behave like ordinary managers with the additional ability to oversee an application under the Chameleon environment. Typically, a single application will consist of several processes—either an MPI application running in simplex mode, for example, or a standalone application that is replicated across multiple nodes. In both of these cases, the surrogate manager will need to install several Execution ARMORS, configure the Execution ARMORS, start application execution, and (possibly) stop application execution.

In this section, we will examine the surrogate manager functionality that applies to a generic application—MPI or otherwise. Specific extensions to the surrogate manager for supporting MPI applications will be considered in section 4.

A generic surrogate manager contains the following elements:

- Application Configuration Parameters Element
- Execution ARMOR Management Element (sm_exec_mgmt_t)

3.2.2.1 Application Configuration Parameters Element (app_param_t)

The app_param_t element stores the executable name, working directory, and command-line arguments for application(s) overseen by the surrogate manager. Although we mostly speak of the surrogate manager as overseeing a single application, we have designed all of the surrogate manager elements to accommodate more than one application. To distinguish these applications overseen by the same surrogate manager, the manager gives each application an application context ID.

The app_param_t element creates a new application context ID (reserves data structure space for a new application) upon receipt of a MSG_APP_CREATE_NEW_APP_CONTEXT message. After allocating a new ID, the app_param_t publishes a MSG_APP_ANNOUNCE_NEW_APP_CONTEXT message. Other elements within the surrogate manager may subscribe to this message to allocate their own data structures for use with a new application.

3.2.2.2 Execution ARMOR Management Element (sm_exec_mgmt_t)

Surrogate managers work closely with subordinate Execution ARMORS to execute the user application under a particular fault-tolerant execution strategy. Because of the important role that Execution ARMORS play, an element in the surrogate manager dedicates itself to keeping important information about the Execution ARMORS. Currently, the element keeps the following information for each Execution ARMOR overseen by the surrogate manager:

- Daemon on which the ARMOR is installed.
- Surrogate manager’s application context ID (the identifier the surrogate manager uses to distinguish between multiple applications that it oversees).
Execution ARMOR’s application context ID (to distinguish between the applications for which the individual Execution ARMOR is responsible).

Status flags for the Execution ARMOR (idle, executing application, application terminated, etc.).

Although some of this information can also be found in the mgr_armor_loc_db_t element (such as the daemon on which the ARMOR is installed), the sm_exec_mgmt_t has its own copy mainly for performance reasons. It keeps its tables updated by subscribing to relevant messages that the manager already generates (e.g., it can update its status flags by subscribing to the MSG_APP_NOTIFY_TERMINATION message).

3.3 Execution ARMOR Implementation

Each user application process executing in the Chameleon environment is overseen by an Execution ARMOR. To deal with these application-related duties, Execution ARMORS are composed of the following elements:

- Application Configuration Parameters Element
- Application Process Management Element
- Application Named Pipe Management Element

3.3.1 Application Configuration Parameters Element (app_param_t)

Execution ARMORS use the same app_param_t element as surrogate managers to store relevant application parameters. Consequently, Execution ARMORS also have the ability to oversee more than one application, each application being identified through an application context ID allocated by the app_param_t element.

It should be noted that at the present time there is no correlation between the application context IDs kept by the surrogate manager and the application context IDs kept by the Execution ARMORS. The surrogate manager, for example, may refer to a specific application (consisting of a set of processes) by application context ID 2 within its own internal data structures. When it assigns one of these application processes to an Execution ARMOR, however, the Execution ARMOR may allocate application context ID 1 for the process. An example of this can be found in Figure 9.

In any case, the most important point is that both surrogate managers and Execution ARMORS use the app_param_t to store application configuration information, and both use application context IDs to identify their applications.

<table>
<thead>
<tr>
<th>Surrogate Manager Application Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM App ID</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution ARMORS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: kmeans (1)</td>
</tr>
<tr>
<td>ARMOR A</td>
</tr>
<tr>
<td>2: kmeans (2)</td>
</tr>
<tr>
<td>ARMOR B</td>
</tr>
<tr>
<td>1: abft_mm (1)</td>
</tr>
<tr>
<td>ARMOR C</td>
</tr>
<tr>
<td>2: abft_mm (2)</td>
</tr>
</tbody>
</table>

Figure 9: Sample Application Context ID Assignments
3.3.2 Application Process Management (app_spawn_t)

The app_spawn_t performs many of the tasks that directly work with the application process, including:

- Spawning a new child application process.
- Registering an existing application process with the Execution ARMOR.
- Forcibly terminating an application process.

For the most part, this element handles anything that deals with the process ID of a specific application process. Upon request, the app_spawn_t will spawn a new child process to execute an application. The element can also "register" an existing application process (one that wasn't spawned through Chameleon) by adding its process ID to the app_param_t element's table. After a process has been registered (either as a result of it being created by the Execution ARMOR or as a result of the Execution ARMOR attaching itself to an existing application), the app_param_t publishes an MSG_ANNOUNCE_NEW_APP_PROCESS message. Other elements may subscribe to this message if they wish to be notified whenever the Execution ARMOR begins to oversee a new application process.

3.3.3 Application Named Pipe Management (app_np_mgmt_t)

Chameleon intends to export an API to the user application through which the application can notify Chameleon of any errors it detects, suggest recovery strategies, change the fault-tolerant execution strategy under which it is running, etc. To support this API, the application must be able to contact Chameleon in some way. In our current design, the application will interface only with the Execution ARMOR through a dedicated named pipe.

The app_np_mgmt_t creates and maintains this dedicated connection with the application. Upon being notified of a new application process through the MSG_ANNOUNCE_NEW_APP_PROCESS message, the app_np_mgmt_t creates a named pipe based on the process ID of the application. Once the application process opens the other end of the named pipe, a connection is successfully established.

Since the Execution ARMOR must respond to requests by the application, the app_np_mgmt_t has many of the message processing capabilities of the np_mgmt_t element. Some similarities include:

- Monitoring the named pipe for incoming messages from the application (this is done in a separate thread).
- Forwarding all incoming messages to the Execution ARMOR's armor_t::process_message() function. Since this is same entry point used for other intra-ARMOR and inter-ARMOR messages, the message will be routed to its appropriate destination with no further involvement on the part of app_np_mgmt_t.

When the app_np_mgmt_t element is included in the Execution ARMOR, it becomes responsible for detecting an application process failure. If the application process terminates abnormally, then the blocking read that a thread in the app_np_mgmt_t element performs on the named pipe will return with an error value.

The app_np_mgmt_t serves as a good example for the composable ARMOR architecture of Chameleon. Configuring an ordinary Execution ARMOR to support an application that uses the Chameleon API is as easy as adding the app_np_mgmt_t element to the Execution ARMOR. Other elements do not need to be concerned with the presence or absence of app_np_mgmt_t—all of its functionality is "added on" to existing functionality without the other elements even knowing it.
4 Extensions for MPI

The previous section outlined the specific elements that provide functionality found in various types of ARMORS. Wherever possible, these elements were designed to handle MPI and non-MPI applications alike. As it turns out, the current incarnation of Chameleon only makes extensions to the surrogate manager to handle MPI applications. This section describes the new elements that need to be added to a surrogate manager for it to oversee MPI applications. It also gives us the opportunity to discuss a particular feature of the ARMOR architecture that facilitates extensions such as these.

It should be noted that these elements have been developed for MPICH, a portable implementation of MPI that uses TCP/IP to communicate among the MPI processes. Features of the elements that are specific to MPICH will be noted when applicable.

4.1 Chain-of-Command for MPI Applications

Being that an MPI application typically consists of several processes, actions that involve the entire application must be coordinated among all processes. A surrogate manager provides this coordination for a single MPI application, implying that there may be several surrogate managers active at any one time in the Chameleon system. Each surrogate manager uses an Execution ARMOR to oversee each MPI process. Like generic Execution ARMORS, they reside on the same node on which the MPI process is running.

![Figure 10: Typical manager-subordinate relationships for MPI applications](image)

Execution ARMORS, in fact, are the only direct contact that the MPI application has with the Chameleon environment. Execution ARMORS consult the surrogate manager when necessary (such as upon detecting an application error), and the surrogate managers consult their manager (most likely the FTM) only when necessary (such as when the surrogate manager needs to restart the application on a different set of nodes).
Figure 10 shows what these manager-subordinate relationships might look like if two MPI applications were running in the Chameleon environment.

4.2 Launching an MPICH Application

The MPICH implementation of MPI provides the `mpirun` script for starting MPI applications. We will briefly review the steps that MPICH takes to start an MPI application and then see what role Chameleon plays in this process.

When using `mpirun`, the user typically specifies the MPI application to run and the number of nodes. Using this information, the following sequence of operations are performed:

- `mpirun` consults a *machine list* file (a file that contains all of the machine names available for executing MPI applications) and extracts the appropriate number of machine names. Using these machine names, `mpirun` generates a *procgroup* file that resembles:

```
cism1 /home/ree/apps/kmeans/kmeans
```

```
cism2 /home/ree/apps/kmeans/kmeans
```

```
cism4 /home/ree/apps/kmeans/kmeans
```

```
cism6 /home/ree/apps/kmeans/kmeans
```

The important things to notice about the procgroup file are that the file contains the machines on which to execute the MPI application (cism1, cism2, cism4, cism6) and the executable name that can be used for the specific machine.

`mpirun` then starts the MPI application with the following command:

- `<program-name> <program-args> -p4pg <procgroup-filename>

  - The master MPI process begins executing; one of the first functions that it calls will be the `MPI_Init()` function.

  - In `MPI_Init()`, the master process uses the procgroup file identified on the command line to get the list of machines on which the slave processes should be executed.

  - For each target machine, `MPI_Init()` uses `rsh` to remotely-execute the slave MPI processes. The exact command that the `rsh` executes on the remote nodes resembles:

    - `<program-name> <master-node> <master-port> -p4amslave`

    - The slave processes begin executing, again calling their `MPI_Init()` function. This time, the `MPI_Init()` function can detect that they are slave processes by the "-p4amslave" command-line argument. Using the `<master-node>` and `<master-port>` arguments, `MPI_Init()` establishes a connection with the master MPI process and retrieves the application command-line arguments.

In order to fully execute within the Chameleon environment, we need to have an Execution ARMOR oversee each application process. For non-MPI applications, we can simply have the Execution ARMOR spawn the application process, itself. But such a solution for MPI applications would be rather intrusive—we would have to circumvent the call to rsh used by MPICH.

Rather than doing this, Chameleon allows the MPI application to launch as usual and then attaches to the application processes. By attaching, we mean that the Execution ARMOR creates and connects to the named pipe that the application will use for the Chameleon API. In order to successfully attach to the Execution ARMORS, the MPI application must make a call to `chm_init()` rather than `MPI_Init()`.
So, under Chameleon, MPI applications are executed as follows:

The surrogate manager is provided with the application’s executable name and a list of daemons on which to run the application.

1. The surrogate manager translates the daemon list into a list of machine names suitable for the procgoup file. Using this list and the application’s executable name, a procgoup file is generated.

2. The surrogate manager installs an Execution ARMOR on each node specified in the daemon list. These Execution ARMORs will each eventually oversee an individual application process.

3. The surrogate manager instructs the Execution ARMOR to begin executing the MPI application. The Execution ARMOR does this by using the same command that `mpirun` uses (see step 2 above).

4. The master MPI process makes a call to `chm_init()`.

5. The master MPI process calls `MPI_Init()` to launch the slave processes as before.

6. The master MPI process then calls `MPI_Recv()` once for each slave process to retrieve the machine name on which the slave process is executing and the slave process ID.5

7. The master process attaches to the Execution ARMOR by opening the named pipe identified by its process ID. Note that since the Execution ARMOR directly spawned the master MPI process and is aware of the process ID of the master process, the Execution ARMOR has already created the application named pipe (see section 3.3.3).

8. After collecting the machine name and process ID from the slave processes, the master MPI process forwards the information to the surrogate manager through the application named pipe.

9. Upon executing, the slave processes call `chm_init()` which then calls `MPI_Init()` as usual.

10. The slave processes use `MPI_Send()` to send their machine name and process ID to the master MPI process.

11. Upon receiving the slave process information from the master MPI process, the surrogate manager notifies the Execution ARMOR on each of the specified nodes as to the process ID of the locally-executing slave process. Using this information, the local Execution ARMOR can successfully create and open its application named pipe.

12. The slave processes attach themselves to the pre-installed Execution ARMOR by opening the named pipe associated with their process ID.

To summarize the important points:

- Only the master process is directly spawned by an Execution ARMOR; therefore, the master MPI process is the only true child process of an Execution ARMOR.
- The slave processes attach themselves to the local Execution ARMOR.
- The Execution ARMORs and slave process agree to use process IDs to reference the application named pipes. Because of this, the Execution ARMORs become aware of the slave process IDs through the `chm_init()` function.

---

5 Remember, to attach to the Execution ARMOR the application must open the Execution ARMOR’s named pipe. This named pipe is referenced by the process ID of the application (see section 3.3.3); hence, the Execution ARMOR must know the application’s process ID in order correctly create the named pipe.
4.3 MPI Execution ARMOR Management Element (mpi_sm_exec_mgmt_t)

For the time being, generic surrogate managers only need to add the mpi_sm_exec_mgmt_t element in order to manage an MPI application. The mpi_sm_exec_mgmt_t element is a special kind of element designed to augment the sm_exec_mgmt_t element.

Recall that the sm_exec_mgmt_t stores information relevant to each Execution ARMOR managed by the surrogate manager. The mpi_sm_exec_mgmt_t takes this base data and adds the machine name on which the Execution ARMORS are installed (for use when generating the procgroup file). This is in stark contrast to other elements whose operations are wholly-independent of one another. If the sm_exec_mgmt_t element and the mpi_sm_exec_mgmt_t element were to be truly independent, then most of the information in the sm_exec_mgmt_t element would have to be duplicated in the mpi_sm_exec_mgmt_t element. Although in this case the amount of overlap would be small, the circumstances gave us the opportunity to explore what may be an important design alternative in the future.

The relationship between the sm_exec_mgmt_t element and the mpi_sm_exec_mgmt_t element bears some resemblance to “friend” classes in C++. In this case, the sm_exec_mgmt_t elements allows other elements to access its “private” data structures. This access, however, is done in a controlled manner to ensure that reconfigurability properties of the ARMOR are not sacrificed. An explanation of this procedure follows.

For the sake of clarity, let us call sm_exec_mgmt_t element the “base” element and the mpi_sm_exec_mgmt_t element the “extension” element. The base element must, in effect, open itself up to the extension elements by providing the following features:

The base element’s data structures must have user-defined pointer fields so that the extension elements may augment each entry in the data structure. For example, the sm_exec_mgmt_t data structure for each entry in its table may look like:

```c
typedef struct
{
    armor_id_t idArmor;        // Execution ARMOR ID
    daemon_id_t idDaemon;      // Daemon on which ARMOR resides
    dword dwStatus;           // Status flags
    app_id_t idAppMgr;        // Surrogate manager's application ID
    app_id_t idAppExec;       // Execution ARMOR's application ID
    void *pvExtension;        // Points to data specific to
                              // extension elements
} entry_t;
```

Here, pvExtension may point to a linked list of structures, each structure belonging to a different extension element (there may be several elements extending a base element).

The base element must have an “access” message through which the extension elements may access the base element.

The last bullet will be explained through a rather trivial example. Suppose that the mpi_sm_exec_mgmt_t element (the extension element) appends the machine name to the entries for each of the Execution ARMORS in the sm_exec_mgmt_t element. Let’s say that the mpi_sm_exec_mgmt_t now needs to retrieve the machine name that corresponds to a particular

---

6 See section 3.2.2.2 for a more thorough discussion of the core data fields within the sm_exec_mgmt_t table.
Execution ARMOR. To do this, mpi_sm_exec_mgmt_t defines a special function, say get_machine_name(), that will operate directly on sm_exec_mgmt_t’s data structure.

This point is important enough to justify repeating—extension elements assume direct knowledge of the implementation of their base elements. The primary advantage is, of course, performance. It is, however, markedly different than treating other elements within the ARMOR as black-box entities.

In order to preserve the reconfigurability properties of the ARMOR, all access to the elements must be through message passing (and, hence, through the parent compound of the element). Herein lies the role of the “access” message identified in the previous bullet—to provide a message-passing interface from the extension element’s point of view.

Concretely speaking, the base element must subscribe to and handle a specific access function—in our trivial example, MSG_SM_EXEC_ACCESS_TABLE. A parameter in this message specifies the extension element’s function that should be applied to the base element’s data structure. In our example, the relevant code would look something like this:

```c
sm_exec_mgmt_t::process_message (msg)
{
    case MSG_SM_EXEC_ACCESS_TABLE:
        *(msg.user_func) (table, msg.param); // Call extension’s function
        // on the table (NOTE: The
call is made from “within”
// the base element.)
}
```

```c
mpi_sm_exec_mgmt_t::get_machine_name (table_t *ptable, param)
{
    for each entry in ptable
        if (entry.idArmor == param.idArmor) // Directly examine entries
            copy entry.machine_name to param.buffer // in base element’s table
}
```

Then, to actually call the get_machine_name() function on sm_exec_mgmt_t’s table, mpi_sm_exec_mgmt_t would send the MSG_SM_EXEC_ACCESS_TABLE message as an intra-ARMOR message, specifying the get_machine_name() function and other specific parameters in the payload area of the message. This has the following advantages and ramifications:

Since we are still going through the normal message delivery services, the parent compound is still the only entity that directly accesses the sm_exec_mgmt_t element.

- Since we never directly interact with the sm_exec_mgmt_t element (only through the MSG_SM_EXEC_ACCESS_TABLE message), the mpi_sm_exec_mgmt_t is not dependent on the specific element type sm_exec_mgmt_t being present in the ARMOR—only an element with the same base data structure and one that subscribes to the MSG_SM_EXEC_ACCESS_TABLE message.

- The previous two points imply that the reconfigurability properties of the ARMOR remain intact.

- The extension element must have intimate knowledge of the base element’s data structure (e.g., the mpi_sm_exec_mgmt_t knows that the sm_exec_mgmt_t’s table entries contain an idArmor field).

- Since the MSG_SM_EXEC_ACCESS_TABLE is sent as an intra-ARMOR message, only an indirect function call results.
• Not every element may automatically be a base element—they have to be designed that way, although making the necessary changes to existing elements is not prohibitive.

• Extension elements are dependent upon the base element existing in the ARMOR; the reverse is not true.

• By using access messages, the extension elements essentially have to gain "permission" from the base element before manipulating the base element's data structures.

We feel that extension elements may prove to be a valuable trade-off between efficiency and pure element "isolation." In essence, we keep the elements isolated enough to provide reconfigurability by mandating that extension elements access base elements through the message passing interface, while we keep the extension elements efficient by allowing them to directly manipulate the data structures of the base element.
5 Concepts in Error Detection

All the functionality in Chameleon is derived from ARMORS. For the environment to be resilient to faults, the ARMORS themselves have to be made resilient. In the sections that follow, we propose a hierarchical structure for error detection in Chameleon ARMORS. The hierarchy of detection techniques is arranged in four logical levels as shown in Figure 11.

![Figure 11: Hierarchy of ARMOR Error Detection](image)

The error detection mechanisms in Chameleon are encapsulated in four logical levels. A level is denoted lower than another if it is implemented closer to the ARMOR being monitored. The chain of error detection starts from the lowest level, and errors not captured at the lower levels bubble up to the higher levels. Because each level may incorporate a suite of detection techniques, a level is characterized by the participating ARMORS. Some level—or, more precisely, some techniques within a level—may be active all the time with a certain periodicity of activation (e.g., the smart heartbeat from the daemon to the local ARMORS), while some others are activated when some trigger event happens (e.g., the consistency protocol in Level 4). The sites at which each of the four levels execute are shown in Figure 12.
Level 1 (lowest level) consists of detecting errors locally and internally in the ARMOR. Support for this level is provided at the time of design of the ARMORS—either by building some capability (like signature generation) into each ARMOR, or by providing a specialized element (like the Monitor Element in each ARMOR). Level 2 consists of detection by the daemon performed on the same node as the ARMOR. The same daemon is responsible for monitoring all the locally-installed ARMORS. Levels 3 and 4 utilize multiple nodes for doing the error detection. They use message passing between distributed processes to perform the detection. The Level 3 message exchanges take place between an ARMOR and its replicas, while those of Level 4 take place between an ARMOR and its replicas, and a manager and its replicas.

Our discussion begins by introducing ARMOR signatures, an integral component of Levels 2, 3 and 4. Next, we present the ARMOR hierarchy with an ARMOR identification system, concept of ARMOR state and its relation to an ARMOR’S position in the ARMOR hierarchy. Then, we present the detection Levels 1 and 2 which execute internally in the same node as the ARMOR. Next, we describe the specification of the protocols in Levels 3 and 4 along with the assumptions and some suggested optimizations and recovery strategies. Finally, we present the interaction between the different detection levels through the example of the lifetime of a synthetic single-task application running in the Chameleon.

### 5.1 ARMOR Signatures

An ARMOR uses signatures to validate the integrity of another ARMOR in Chameleon. A signature generally consists of a data portion and a control portion. For purposes of detection, a receiver has to re-create a golden signature of the sender during comparison. The control part of the signature is pre-loaded in the receiver in the form of a set of possible patterns from the sender. The data portion of the signature is computed by the receiver prior to the comparison. It is meaningful to compute the data signature only on a shared and consistent data structure. Therefore, for interactions between certain ARMORs with no such shared data structure, the signature only consists of the control portion.

#### 5.1.1 Control Signatures

The control part of the signature identifies the control flow of the ARMOR. This can be computed from the control flow graph of the ARMOR. For example, consider the control flow graph of an ARMOR in Figure 13, where each rectangle is a logical block of execution in the ARMOR. Valid control flows for the ARMOR can be specified by the regular expression A.(B.(C.DIE))*. The sender sends the control flow state at the time of sending the signature. This is checked for truth or falsity according to the regular expression pre-loaded in the receiver. Since an ARMOR is composed of elements executing in no fixed
order, the control part of the ARMOR signature is derived by combining the control signatures from its constituent elements. When an element is invoked by the message delivery function of its parent compound, it writes its signature in the data structure in which the incremental ARMOR signature is constructed. Naturally, this data structure is shared among all the elements in the ARMOR.

![Sample control flow graph of an element used in control signature computation](image)

Signatures are unique to an ARMOR. A particular ARMOR, independent of whether it is acting as a primary or a backup, will have a unique golden signature which is a set of its possible valid control flows. If the replicas are passive, then the signature generated by them will obviously be much simpler than that generated by the primary. However, it will still match one of the valid control-flow signatures stored at the receiver. The utility of treating primary and backups alike from the point of view of signature is two-fold. First, in the implementation of the system, the primary and the backup will have the same ARMOR code. A flag determines if the ARMOR behaves as a primary or a backup. If it acts as a passive backup, there will be a separate element, with the functionality of keeping consistent state, that will be activated and all other elements will be inactive. Second, the backup ARMORS may switch from a passive to an active role (say, for some critical operations like deciding to terminate an application). In that case, having the same signature for the ARMOR in both active and passive role, obviates the necessity of reloading the signature for the replica, that has just turned active, at the receiver.

Since there is no inherent ordering between the activations of elements that compose an ARMOR, the control flow of an ARMOR is essentially the combination of the control flows of its constituent elements. The control flow of an element can be computed at a coarse or a fine level of granularity. At a coarse level, the element signature consists of the valid input and output message pairs for the element. In practice, this is expected to capture most control flow errors in an ARMOR. It is less probable that an element fails in such a way that it accepts a valid input message, and produces a valid output message but the intermediate processing is erroneous. In case such a failure mode also needs to be covered, an element can produce signature at the finer level of granularity. At the finer level of granularity, the control flow within the element is also captured. Either the designer of the element inserts code in the element at appropriate points, or a tool decomposes the element into its basic blocks and assigns unique IDs to each basic block. The latter strategy is dependent on obtaining a tool which would do the decomposition of the element and also append the signature at the basic block level. The first strategy places the responsibility on the ARMOR designer for identifying appropriate points (at which points the control flow is sought to be tracked) in the element code and inserting code for the signature.
5.1.1.1 Control Signature Generation and Propagation

The signature is not very useful unless it can be propagated to other ARMORs which are responsible for error checking. Also, the checker must have a golden copy of the signature against which the received signature can be compared. Thus, methods have to be defined for pre-loading the golden copy of the signature of an ARMOR into possible checkers and to distribute the ARMOR's signature to its checkers during execution when one of the error detection levels requires signature checking. When an ARMOR is installed, in the initialization phase, all its constituent elements register the messages they want to subscribe to with the compound level of the ARMOR. This is augmented to make the element also register the output message it generates for each of the input messages. It may be recollected that this pair is precisely the coarse-grained control signature for the element. After initialization, this signature is available internally in the ARMOR. The ARMOR forwards this signature to the local daemon. This is used by the daemon to check the integrity of the ARMOR in Level 2 of error detection (See section 6.2 for details of the detection). To enable Levels 3 and 4 of detection, the signature must also be available with some remote ARMORs (e.g., the replicas of an ARMOR for Level 3 of detection). The transmission of the control signature occurs to these other ARMORs through the daemon. Along with the forwarding of the signature to the daemon, the ARMOR distributes its valid signatures to the other appropriate ARMORs through the daemon. The finer-grained control-flow signature is also propagated in a similar manner. Here, the element's initialization phase involves additional code through which the element registers its signature with the ARMOR at the compound level. This then forwards it to the daemon, which then distributes it to the appropriate ARMORS.

**Figure 14: Control Flow Signature Generation and Propagation**

During execution of the ARMOR, its signature is computed and propagated to the checkers. For the coarse-grained signature, this occurs as a natural consequence of the message handling by the element. When a message is delivered to an element, and when a message is generated by an element, this information is
registered at the compound level in the ARMOR. This information is propagated to other ARMORS (like the local daemon and other replicas) when queried as part of detection at Levels 2, 3 or 4. For the fine-grained control flow signature in the element, the element designer calls a function that writes the signature in a shared data structure at the ARMOR level. After that, the propagation is done on receiving a query just as in the previous case. The signature is also propagated to the daemon when the ARMOR generates a message to be routed through the daemon. The scheme of signature generation and propagation as well as the concepts of associating both coarse and fine-grained signatures with threads within an element, an element within an ARMOR as well as an ARMOR in a node are all shown schematically in Figure 14.

Figure 14 shows an example of an ARMOR A1 and the local Daemon D. A1 has two elements—E1 and E2. Element E1 has two threads of execution—T1 and T2. The running example for signature generation is given for the thread T1 in the element E1 of the ARMOR A1. The checker for the signature is Daemon D. Element E1 is initialized after the ARMOR A1 is installed. During initialization, E1 registers the following information with the compound level of ARMOR A1:

- The set of valid input-output messages for T1. In the example, this is (I1,O1),(I2,O2). [Note that a single element can subscribe to multiple messages and can produce multiple output messages in response to an input message.]
- The valid signature of T1. In the example, this is given by the regular expression

\[
[ S1 . ( ( S2 . S3 ) | S4 ) ] *
\]

This registration information comprises the Registration Information Table (RIT). The registration of the acceptable input messages by an element occurs in the Chameleon environment, irrespective of any error detection mechanisms, for its natural message delivery functionality. Hence, piggybacking the extra signature information at the registration phase does not incur any substantially additional overhead. After registering this information internally in the ARMOR, the ARMOR propagates its RIT to the local daemon. This RIT sent to the daemon forms part of the Golden Signature Table (GST) for ARMOR.

The above discussion covers events that take place at the time of initialization of the ARMOR. During the execution of the ARMOR, the runtime signature has to be generated and made available to any checkers like the daemon. This is achieved by incrementally constructing the signature that is being generated in a data structure called the Runtime Signature Table (RST). This RST is communicated to the checkers either synchronously (e.g., when the daemon checks for ARMOR signature with a certain periodicity) or asynchronously (e.g., the case where the signature is checked on activation of a detection protocol). In the example, the thread T1 has executed the blocks whose signatures are S1, S2 and S3; it had been activated by an input message I1 and generated an output message O1. The Emit routine which is called within a thread of an element writes into the RST at the compound level. Observe that at the current control flow signature of T1 (in E1 which is in A1) is valid. If this holds for every thread in the ARMOR, then the ARMOR will be certified as functioning correctly by the local daemon.

### 5.1.2 Data Signatures

The data part of the signature is obtained by executing a hashing function on some data structure that is shared among a set of ARMORS. In the sections that follow, we will consistently use the ARMOR Location Table found in managers as the representative state of an ARMOR. Please refer to section 5.2 for more details about the ARMOR Location Table.

In order to use the ARMOR Location Table as a basis for verifying the integrity of the armor through data signatures, all participants must have the same consistent view of the data so that the data signatures computed by each participant will match for the same set of data.
5.2 ARMOR Hierarchy

This section describes the ARMOR hierarchy as it relates to the error detection algorithms. We also introduce some of the relationships between the various levels of the ARMOR hierarchy and the pertinent information that they keep for the error detection algorithms. ARMOR Parent-Child-Sibling Relationship

The ARMOR hierarchy presented here is according to the manager-subordinate relationship among the ARMORS. The hierarchical arrangement of ARMORS in different ranks is shown in Figure 15. The figure also shows replicated ARMORS. Note that the replication degree of ARMORS of any rank is unconstrained and is determined by the desired reliability level. Also, within the same rank, the replication degree of different ARMORS may vary.

![Figure 15: ARMOR Hierarchy](image)

In the ARMOR hierarchy, a lower rank number denotes a higher ranking manager. Thus, rank 1 consists of the highest ranking manager—the FTM and its replicas. The rank of a manager is one less than its immediate subordinate.

The ID assigned to an ARMOR is a 3-tuple. The first element of the tuple is the rank of the ARMOR, the second element is the ID unique to the logical ARMOR in that rank\(^7\), and the third element is the ID unique to the particular ARMOR replica within the logical ARMOR. The 3-tuple uniquely identifies all ARMORS in the Chameleon environment.

The first ARMOR within a logical ARMOR (i.e., the ARMOR placed first in a round-edged rectangle, or the ARMOR whose ID has its third tuple being 1) is the primary and all others are backups (active or passive depending upon the particular operation or execution strategy). The ARMORS which constitute a logical ARMOR are referred to as siblings of each other. The parent-child relationship among ARMORS is defined as follows: In Figure 15, the primary in the logical ARMOR at the source (or head) of an arrow is referred to as the parent of all the replicas within the logical ARMOR at the sink (or tail) of the arrow. The child relationship is the reverse of the parent relationship. For example, in Fig.15, A(1,1,1) is the parent of A(2,1,1) and A(2,1,2). A(2,1,1) and A(2,1,2) are siblings, and so are A(1,1,1), A(1,1,2) and A(1,1,3). Note that because of this definition, A(1,1,2) is not a parent of A(2,1,1).

---

\(^7\) A particular ARMOR and its replicas constitute a logical ARMOR. In Fig.15, all ARMORS within a single round-edged rectangle form a logical ARMOR. This is because all of these ARMORS are visible to the outside world as a single ARMOR except for the error detection protocols. For error detection, distinction has to be made among those ARMORS so as to diagnose a faulty replica.
Generally,

- Necessary and sufficient condition for \(A(i_1, j_1, k_1)\) and \(A(i_2, j_2, k_2)\) to be siblings is \(i_1 = i_2\) and \(j_1 = j_2\).

Necessary (though not sufficient) condition for \(A(i_1, j_1, k_1)\) to be the parent of \(A(i_2, j_2, k_2)\) is \(i_2 = i_1 + 1, k_1 = 1\).

### 5.2.1 ARMOR State

The state maintained at each ARMOR may be quite complicated. It is also dependent on the position of the ARMOR in the ARMOR hierarchy. For example, state at the FTM will include a list of functioning hosts participating in the Chameleon environment and the respective loads on these hosts. This state information need not be present at any other ARMOR. Again, the state at a manager ARMOR will include diagnostic information such as the number of failures of a subordinate ARMOR on a particular node so that migration to another node may be initiated if it exceeds a certain threshold. This will not be part of the state at an ARMOR which is a leaf node in the hierarchy. For the purpose of this discussion, however, we will use the ARMOR Location Table as our representative example of ARMOR state.

### 5.2.2 Superset Relationship in ARMOR State

The state maintained at the ARMORS follows a superset relation according to its placement in the hierarchy tree denoted in Figure 15. The parent, along with its siblings, contains a superset of the state maintained at its child and its siblings. In the example from Figure 15, the ARMOR Location Table at \(A(1,1,1)\) (the primary FTM) will contain location information for all the other ARMORS; the ARMOR Location Table at \(A(2,1,1)\) will contain location information for \(A(3,1,1)\) and \(A(3,1,2)\). Applying the transitive relation to ARMORS at subsequently lower levels, we can conclude that the state at an ARMOR is the superset of any of its subordinate ARMORS (direct or indirect subordinates).

### 5.2.3 Active or Passive Replicas

The replicas may be configured as active or passive replicas. They may also switch roles at runtime depending on failures being encountered (a higher fault rate may necessitate running the replicas as active replicas) or on a per-operation basis (a critical operation like terminating an application may have to be taken by FTM replicas acting in active mode). The same elements are installed in all ARMOR replicas. Based on a flag that is set in the ARMOR, either all the elements containing the normal functionality of the ARMOR are activated (active replica), or only the element responsible for maintaining consistent state and participating in the error detection protocols is activated (passive replica).

### 5.2.4 Distribution of ARMOR Signatures within the Hierarchy

The data part of the signature does not have to be explicitly distributed among any of the entities. To recall, the data part of the signature is computed on some common data structure (like the ARMOR Location Table). By the consistency protocol described in section 6.2, the ARMOR Location Table is made the same in all the ARMOR replicas; therefore, we have effectively distributed the data signature among all ARMOR replicas. Again, because of the superset relationship for the ARMOR state among ARMORS and their children the data signature of an ARMOR is available to all its managers (direct or indirect managers).

In contrast to the data signature, the control signature has to be explicitly loaded into other ARMORS. To minimize the overhead of the loading operation, the control signature should be distributed to the minimal set of ARMORS. In this section, we present the set of ARMORS to which the control signature has to be loaded. A mention is made of when it comes of use in one of the different error detection levels. This will be clarified during the detailed discussion of the detection protocols in sections 6.1 - 6.3.

The control signature of an ARMOR A [CSigA] is loaded into:

- The local daemon [DA] (used in Level 2)
• The set of replicas of A \( [R_A] \) (used in Levels 3 and 4)
• The set of manager replicas of A \( [M_A] \) (used in Level 4)
The set of child replicas of A \( [C_A] \) (used in Level 4)
• The set of daemons on which A has installed children. This is loaded when A installs the subordinate ARMORS and is used during Level 4.

In addition, the control signature of the FTM is loaded into all the Daemons in the Chameleon environment. This is loaded when a node registers itself with the FTM, and the FTM installs a Daemon on that node. This is used in executing Level 4 when the FTM installs a Surrogate Manager and involves the Daemon as a participant.

6 Hierarchical Error Detection in ARMORS

This section describes the hierarchical error detection techniques for ARMORS that are available at each of the levels.

6.1 Level 1: Internal ARMOR Techniques

The Level 1 error detection techniques are built into each ARMOR at design time. The goals are to make the ARMOR self-checking and to enable detection techniques external to the ARMOR. This level consists of three basic detection techniques currently.

1. **Assertion checks.** Assertion checks are built into the elements. The assertion check may take one of several forms. It may be a check for a type of message the element has not subscribed to, or it may be a validation check on some internal data structure of the ARMOR (e.g., check to see that the location table size of a manager is not zero). When an assertion check fails, element notifies the local daemon of the error. A typical assertion check in an ARMOR element is shown in Figure 16.

   ```java
   Element e_i of ARMOR A_i resident on node with Daemon D_j
   switch (IncomingMessageType)
   case M1: ... break;
   case M2: ... break;
   default: // Error. Wrong message type.
               D_j.notify({e_i, A_i});
   
   Figure 16: Assertion check in ARMOR element
   ```

2. **Livelock detection.** Livelock detection can also be done within an ARMOR. Because of the multithreaded nature of the ARMOR architecture, an element typically acquires a mutex lock before accessing its local data structure. Since must livelock conditions will occur while processing the local data structure, a separate thread in the ARMOR can watch the mutex locks to detect livelocks. In our current design, we are planning to have a dedicated element (a Monitor Element) that executes this dedicated thread to detect livelocks. If a lock is observed to be held for greater than a threshold period of time, the Monitor Element notifies the daemon about the error. A fallout of this design is that the Monitor Element has to be informed of the installation or uninstallation of any other element.
(or, at least, those elements willing to have their mutex locks monitored). This is achieved by having the Monitor Element subscribe to the MSG_CMP_ADD_ELEMENT and MSG_CMP_DELETE_ELEMENT message. Also, the Monitor Element has to be informed of the lock variable of the other elements. An element can provide this information in its element_t::init() function called during the initialization of a newly-instantiated element.

3. **Signature computation.** This enables error detection at other levels. As discussed in section 5.1.1.1, the signature gets computed within the ARMOR and is generated in an internal data structure available for use by a higher detection level.

### 6.2 Level 2: Detection by the Local Daemon

Level 2 consists of the error detection done by the local daemon. There are two basic techniques available at this level:

The daemon captures the exit status of a child ARMOR process. This relies on the malfunctioning ARMOR generating an exception in the underlying operating system, which is captured by the daemon.

The daemon heartbeats each ARMOR that is installed on the local node. If a manager entity needs to be updated about the health of a subordinate entity, it may send a request to the daemon to heartbeat the specific ARMOR outside of the regular heartbeat period. Thus the heartbeats from the daemon may be synchronous (when it follows the specific period) or asynchronous (when it is done pre-emptively by a message from a manager entity). In response to these “smart” heartbeats, the ARMOR activates its constituent elements to do a consistency check. This it does by activating each of the constituent elements with a test message such that the response of the element to the test message is known by the daemon. The daemon generates a syndrome information for the ARMOR and concludes if any of the constituent elements is faulty and needs to be reinstalled.

Levels 1 and 2 interact to provide errors from propagating outside the node. As discussed in the previous section, a Level 1 detection technique may notify the local daemon of an error. In this case, recovery is handled by the daemon as if the detection had been done at Level 2. In addition, when an ARMOR wants to communicate with another ARMOR, the daemon can check the signature of the sending ARMOR before forwarding the message to verify that the source ARMOR is functioning correctly. This provides error containment within the local node using the daemon as the barrier.

### 6.3 Level 3: Detection among ARMOR Replicas

Level 3 is the first level in which detection does not occur on the local node on which the ARMOR is executing. Thus, the mechanisms are this level are not susceptible to malfunctioning of the node. One of the mechanisms envisaged at this level involves exchange of signatures among ARMOR replicas. As discussed in section 5.2.4 both control and data signature are available with all the replicas of an ARMOR. Hence comparison of the received value and the “golden” value can be done at each of the replicas.

This protocol involves message exchanges among replicas of an ARMOR. The messages contain both the control and the data signature of an ARMOR. The protocol executes with a certain periodicity. The trigger to activate the protocol is a timer that runs locally on each ARMOR replica. For the data signatures to be consistent across replicas, their data must have been made consistent. Hence, this protocol cannot execute in between a consistency protocol. Hence, the Level 3 and the Level 4 protocols are atomic with respect to each other. This is achieved by using the same element within each ARMOR for both the Level 3 and Level 4 protocol. The element acquires its mutual exclusion lock at the beginning of either of the protocols and releases it on termination of the protocol.
Protocol has periodicity $\tau$
Self.ID = (M,N,P)
Set of replicas = (M,N,x), x=1,...,R

\[\text{do} \{
\text{Set timer} = \tau; \\
\text{Start timer;}
\text{When (timer expires) \{} \\
\text{Lock element to ensure protocol atomic w.r.t. state change;}
\text{For each ARMOR (M,N,x), x=1,...,R, x*P \{} \\
\text{Send(SigSelf} = \text{CSigSelf + DSigSelf});
\text{\} Unlock element;}
\text{\} Set timeout_timer = N * \tau; \\
\text{Start timeout_timer;}
\text{Wait for Sig(M,N,x), } \forall x=1,...,R, x\neq\text{P till timeout_timer expires;}
\text{For each ARMOR (M,N,x), x=1,...,R, x\neq\text{P \{} \\
\text{If (Sig(M,N,x) == SIG(M,N,x)) Then}
\text{ OKAY!}
\text{Else}
\text{ Notify Manager of (M,N,x) of fault;}
\text{\} while (TRUE);}
\]

\[\text{CSig(x,y,z) & DSig(x,y,z) : Control and Data Signature respectively of ARMOR (x,y,z)}
\text{SIG(x,y,z) : Golden signature of ARMOR (x,y,z)}
\]

Figure 17: Pseudo-code for the Level 3 Protocol

The Level 3 protocol pseudo code is given in Figure 17. When the timer for the default periodicity of the protocol expires, each ARMOR computes its data signature and control signature and sends it to each of its replicas. Notice that the actual time between the timer being started and the protocol starting execution may be greater than the default periodicity ($\tau$ in Figure 17) because the attempt to gain the lock for the element may fail if the Level 4 protocol is executing. After it is done with sending its signature, it waits to receive the signatures from all of its replicas. If no signature is received from a replica, it flags a missing value. At time of comparison all missing values and all signatures that don’t compare with the golden copy are flagged as errors. It is to be noted that the protocol executes in close synchrony among the replicas because the timer $\tau$ is started only after one round of comparisons is completed. Thus the skew between two replicas can never become greater than (Timeout for flagging missing value + Skew between the times to do comparisons at each replica).

When a manager is notified of an error through this protocol, it checks to see that all of its replicas have been notified of the potentially faulty ARMOR. Only then does it initiate recovery. Otherwise, it forwards the possible fault to the next level by running the diagnostic phase of the Level 4 protocol. This is because if we assume that only a single ARMOR among the ARMOR replicas fails (and sufficiently high coverage for a faulty ARMOR being unable to produce a valid signature), then all the replicas other than the faulty ARMOR will detect a mismatch in the signature of the faulty ARMOR and execute the symmetric action of notifying the manager. Observe that for the common case of an ARMOR having two replicas and one of them failing, this protocol will be unable to diagnose the faulty replica and will trigger the Level 4 protocol.
6.4 Level 4: Detection Among Adjacent Rank ARMORs

This is the highest level of error detection in Chameleon. This level incurs the most overhead among the four levels, and is also least frequently invoked. Like any of the other levels, the mechanisms at this level may be turned on or off depending on the overhead that can be tolerated. The Level 4 error detection involves the participation of ARMORs from at least two adjacent ranks in the ARMOR hierarchy. It is invoked on state changes in the system or when an error remains undiagnosed at the lower levels. ARMOR replication introduces the problem of keeping the replica states consistent. The goal of keeping consistent replica state coupled with checking integrity of the replicas is sought to be achieved through the Level 4 protocol. The two goals are mapped to two logical phases of the protocol.

6.4.1 Protocol Requirements

The Level 4 protocol (L4) needs to satisfy the following requirements:

1. The protocol causes the correctly functioning ARMORs which participate in the protocol to agree upon a consistent update information.
2. Assuming that at most one of the participating ARMORs is in error, the protocol is able to diagnose the faulty ARMOR.

6.4.2 Protocol Participants

The protocol is modeled as a Byzantine General's protocol augmented with error detection capabilities. The Authenticated Messages variant of the protocol is used. The theoretical results show that to tolerate \( m \) Byzantine faults, there need to be at least \( (m+2) \) participants. (The case of agreement becomes vacuous if there are less than 2 correct entities.) Therefore, to tolerate a single fault, there need to be at least 3 participating ARMORs. Generally, L4 is executed among a parent and its replicas and one of its children and its replicas. For example, in Figure 18, \( A(1,1,1) \), \( A(1,1,2) \), \( A(1,1,3) \), \( A(2,1,1) \), \( A(2,1,2) \) could participate in one run of the protocol.

Figure 18: Example of Participating ARMORs in a Level 4 Protocol

However, if 3 ARMORs are not found among 2 adjacent ranks, ARMORs from other ranks will have to be involved. This can be solved by involving ARMORs from 3 adjacent ranks (and this is guaranteed to provide the minimum number of replicas, i.e., 3). For example, if ARMORs at ranks 1, 2 and 3 are all unreplicated, then ARMORs from all 3 ranks participate in one run of the protocol. If an unreplicated Execution ARMOR (EA) installs an application and wants to make this state consistent with its manager, an unreplicated Surrogate Manager, it will initiate L4 with participants from 3 adjacent ARMOR layers—
the Execution ARMOR itself, the Surrogate Manager (SM), and the FTM. The authentication mechanism in L4 is through the use of signatures – both control and data. For the data signatures to be meaningful, the participants must have some consistent data that they share. When ARMORs from multiple ranks are involved, then the subset of state that they share is used for computation of the data signature. For higher coverage from the signature, the data signature must be computed on the largest subset of data possible. The subset becomes smaller as ARMORs from ranks which are farther apart are involved. Thus, for example, as in the case above, if the EA, SM and FTM are involved, the data signature will be computed on a small subset of the total data in the FTM, and hence the detection of a FTM malfunction has lesser coverage. Therefore, as far as possible, the protocol involves ARMORs from 2 adjacent ranks.

There is a case where it is not possible to involve ARMORs from 3 adjacent ranks. That is when the FTM installs a Surrogate Manager. Even if the FTM is replicated in a primary-backup configuration, it does not have 3 ARMORs with whom to run L4. In this special case, the protocol is run between the replicas of the FTM and the Daemon on the node on which the SM is being installed. This points to a requirement to have the FTM signature loaded onto all Daemons in the environment and this is done when a node registers itself with the FTM and the FTM installs a Daemon on the node. Obviously, since the Daemons do not share the Location Table with other ARMORs, the signature exchange that goes on in this special case is only the Control Signature.

6.4.3 Protocol Triggers
The Level 4 protocol is triggered in the following cases:

State change in an ARMOR. The participants on such a trigger are the ARMOR in which the state change has occurred, its replicas and its parent and its replicas (all of whom to which the state change must be communicated). A state change takes place, for example, when the ARMOR installs or uninstalls another ARMOR. It is to be noted that the protocol is triggered before the actual state change has occurred. Thus, it will be triggered when a particular ARMOR is about to uninstall one of its children. After L4 executes and all the participants update their state, the actual uninstallation occurs. Otherwise, a faulty ARMOR could go ahead and uninstall another ARMOR and only after that (when L4 is executed) would it be known that the ARMOR was faulty to begin with. As opposed to that, in our design, the ARMOR has to run L4 first and only if it is certified to be fault-free is it allowed to do the uninstall.

Error propagation. In addition, if a Level 3 error detection mechanism indicates an error, but is unable to diagnose the errant ARMOR, then we invoke phase 2 of our Level 4 protocol.

6.4.4 Protocol Phases
The Level 4 protocol has two logical phases. During any particular run, either of the two phases may be omitted depending on the invoking condition. In the first phase, the participants exchange state update messages in an effort to reach a consistent state. In the second phase, the participants try to determine the erroneous entity, if any, among them. This is achieved by examining the messages exchanged in the first phase followed by exchange of the signatures among the participants.

6.4.5 Protocol Specification
The Level 4 protocol is specified in the lines of a Byzantine Generals (BG) protocol augmented to do error diagnosis. (Note that “phase” and “round” are used in two different senses in this discussion. “Phase” is explained in section 6.4.3, “round” refers to a stage of message exchange in a BG protocol.)

A few words about the mapping of the BG protocol for our environment. In a practical use of the protocol, it is important to define the abstract state that is exchanged between the participants. For the first

---

8 In order to avoid the overhead, an optimization is suggested in the section to reduce the number of times the protocol is actually triggered in the system.
logical phase, the state should be the one on which the participants need to come to an agreement (i.e., the state that must be maintained consistent across the replicas). In the Chameleon environment, an example of this is the ARMOR Location Table. Since this information is propagated up the hierarchy of ARMORS and the protocol participants are also ARMORS from two adjacent ARMOR ranks, it is suitable for defining the ARMOR Location Table as the state to be exchanged. For the second phase of the Level 4 protocol, the requirements are quite different. In this phase, we are trying to diagnose the erroneous ARMOR among the participants by looking at the state exchanged among them. Here, the state has to be indicative of the health of the entity. In other words, the coverage that if the entity is erroneous, then the state that it will transmit will be in error should be as high as possible. The signature of the ARMOR constitutes precisely such a state. Since part of the signature is derived from the internal state of the ARMOR, this indicates a natural dependency between the two phases (the consensus phase and the error detection phase) of the Level 4 protocol—the phase for attaining consensus should precede the signature checking.

As may be familiar to the reader, interactive consistency protocols can use either Oral Messages or Signed Messages. Oral Messages denote that there is no authentication of messages coming from an entity, while Signed Messages assume that the message from an entity can be authenticated. The advantage of using the Signed Messages protocol is a reduction in the number of participants to tolerate a certain fixed number of faults. Specifically, for tolerating a single fault, Signed Messages would require three participants, while OM would require four. Since we have introduced the concept of signatures, we can use them also for the purpose of authenticating the messages. This is attractive in our case because if we can restrict the number of participants required, then ARMORS from multiple ranks will not have to be involved in a run of L4. As mentioned in Sec.6.4.2, this has the advantage of giving higher coverage for the associated signatures. Since we already have signatures of each participant loaded in the other and if we assume high coverage for the signature to be non-forgable, then these signatures can be used to sign messages in the traditional BG protocol. Since the receiving ARMORS store signatures on a per logical ARMOR basis, therefore, to differentiate between the replicas of an ARMOR, the ARMOR Id is appended to the signature. Hence, we use the Signed Message variant of the BG protocol in our scheme.

Another issue in the mapping of the interactive consistency protocol is to synchronize the stages of the protocol. Once the protocol is initiated, each participant can set local timeouts and if no message is received within the timeout, it can conclude a missing message. However, all entities’ timeout clocks need to be triggered at the start of the first round. This implies that the trigger for the protocol has to be reliably communicated to all the participants. To recall, triggers for the Level 4 protocol can be installation, uninstallation, or a detected error from another level. Let us say that the protocol is being run among two replicas of the FTM (F1 & F2) and two replicas of the surrogate manager (S1 & S2), and the trigger in this case is an installation of an ARMOR by S1. Once it installs the ARMOR, S1 acts as the transmitter and sends out the first round message to S2, F1 and F2. This message is used to flag the start of the protocol. If the message is not received at a particular ARMOR before a message from a subsequent round is received, then the entity flags the default missing value for the message from S1 and uses the subsequent message to start its timeout clock. So, as long as the transmitter does not remain silent to all the other participating ARMORS, the timeout clock can be started. The problem of how to deal with the case when S1 does not initiate the protocol at all and unilaterally tries to force a state change is discussed in Sec.6.4.6.2.

6.4.6 Protocol Description

We describe the protocol Z for the most common case when the participants belong to two adjacent ARMOR ranks. For the two special cases when one ARMOR from each of 3 ARMOR ranks is involved, and when the FTM replicas and a Daemon on which SM is to be installed participate, the protocol follows the same general structure. The deviations are mentioned in Sec.6.4.6.2.

Participants:  A(I, J, 1), A(I, J, 2), ..., A(I, J, M) : Parent; lower rank number ARMOR replicas
A(I+1, K, 1), A(I+1, K, 2), ... , A(I+1, K, N) : Child; higher rank number ARMOR replicas

Since the second identifier parameter for the ARMORS is immaterial in our discussion, we abbreviate the representation as: A(I,1), and A(I+1,1), and so on.

Assumption: We assume that at most one entity among the participants can fail. This is a weaker assumption than only one entity in the whole environment being allowed to fail. Under our assumption, more than one entity can fail as long as they are not participating in the same run of the protocol. Later, we identify the time window within which this assumption of single ARMOR failure needs to be satisfied.

Trigger Events (TE):
1) ARMOR A(I+1,1) wanting to install or uninstall another ARMOR9.
2) An error being detected but not diagnosed among the child ARMOR replicas and flagged to A(I+1,1).

All the trigger events are from the perspective of A(I+1,1) being the initiator of the protocol. TE1 & TE2 cause phase 1 of the protocol to be executed. Whether it will be followed by phase 2 or not depends on whether confidence can be placed on the integrity of the participating ARMORs (perhaps because of the diagnostic having been run in the near past). TE3 causes phase 2 of the protocol to be initiated by A(I+1,1).

Note that these trigger events are for the case when we are considering ARMOR installation or uninstallation as the state change. In the more general case, state change may be more complicated and may depend upon the position of the ARMOR in the hierarchy. In the FTM, for example, a change occurs when its list of functional nodes in the environment changes. Thus, for the general sense, the trigger events for the protocol, in place of TE1-TE4, will be events that cause a change in ARMOR state that needs to be propagated up the ARMOR hierarchy.

Protocol Description

Phase 1

t0 : Suppose the ARMORs to be installed or uninstalled are A(I+2,1), ... , A(I+2,P) and the corresponding Daemons are D(I+2,1), ... , D(I+2,P) (maybe not all distinct). A(I+1,1) forms a message of the form Ψ=M:S:I, where

M is of the form [{A(I+2,1),D(I+2,1)},...,{A(I+2,P),D(I+2,P)},F], where F is a Boolean flag that denotes if the tuples have to be added to the LT (installation), or deleted from the LT (uninstallation). This is the new update to the Location Table that the participants are trying to agree upon. Thus, by a sequence of L4 protocols, the state can be maintained consistent across replicas of a particular ARMOR and can be consistently propagated up the parent chain.

- S is the concatenation of the Control and Data signature of A(I+1,1) with one difference for the data signature. Let LT_{I+1} be the Location Table of A(I+1,X) (i.e., any of the ARMORs A(I+1,1), ..., A(I+1,N)) before the update and LT'_{I+1} be the Location Table if it were updated with V (if the transmitter i.e. A(I+1,1) were faulty, then there is no guarantee that this will happen). Then, instead of

9 Depending on whether the replication scheme is active or passive, A(I+1,1) may take this decision unilaterally, or it may be arrived on after voting on decisions of all the child replicas. In either case, it is A(I+1,1) which initiates the rounds of message exchange for the protocol.

10 Here also, depending on active or passive replication among the parent replicas, the command may be given unilaterally by A(I,1) or after voting on the decisions of all the replicas.
using $LT_{i+1}$, $A(I+1,1)$ uses $LT'_{i+1}$ to compute its data signature. The reason for this is discussed after the protocol description.

- $I$ is the ARMOR identifier for $A(I+1,1)$

The message type $M:S:I$ is equivalent to the form of message $m:s$ in the traditional BG protocols with Signed Messages, where $M=m$ is the payload and $S:I=s$ is the non-forgeable signature of the participant. It may be mentioned that the unique ARMOR identifier $I$ is necessary to form $s$ so that the receiver can distinguish between ARMOR replicas for the same logical ARMOR. Remember that receiver ARMORs store a single copy of the golden signature for all the replicas of a particular ARMOR.

After forming $\Psi$, $A(I+1,1)$ sends it to each of the other participants, in the child rank as well as the parent rank.

$t_1$: All other participating ARMORs when they receive the message from $A(I+1,1)$ form a message with the same payload and append their signature and ID to that of $A(I+1,1)$. Consider $A(I,j)$. It receives $M:S_{M(i+1,j)}:I_{M(i+1,j)}$ and forms $M' = M: S_{M(i+1,j)}:I_{M(i+1,j)}:S_{M(i,j)}:I_{M(i,j)}$. $M'$ is formed by concatenation of the control signature and the data signature, where the data signature is computed on the following data structure. The Location Table of $A(I,j)$ will be a superset of that at any of the child replicas. The Location Table has the structure $[ARMOR\_ID, Daemon\_ID, Manager\_ID]$. $A(I,j)$ forms a subset of its entire LT restricting to tuples for which the manager ID is the logical ARMOR $A(I+1,X)$. Using the restrict operator $\cdot$ from database, we have $LT'_{j} = \cdot_{Mgr\_ID=A(I+1,X)}LT_{j}$. $A(I,j)$ appends the received payload $M$ (containing the new tuples which may or may not be finally updated in the LT) to LT" and forms $LT"$. The data signature is computed on LT".

After forming $M'$, $A(I,j)$ sends one copy of it each to all the participants, except the transmitter, i.e., $A(I+1,1)$.

For a child replica $A(I+1,j)$ $[j\not=1]$, the steps are same as above except $LT'=LT$

**Phase 2**

$t_2$: Each ARMOR has $M+N-1$ values with it. If it does not receive a communication from any of the other participants within the timeout interval, it assumes the default value from that ARMOR. It filters out the malformed values from the received messages. A value is malformed under the following conditions:

- If the message is $M:\{CS1,DS1\}:I1:\{CS2,DS2\}:I2$ and applying the data signature hashing function on $LT\cup M$ does not generate $DS1$ or $DS2$,
- If $I1$ or $I2$ is not a protocol participant

The ARMOR does a vote on the remaining $M+N-1$ values. The decision is used to update the LT of the ARMOR. If no decision can be reached, then the transmitter $A(I+1,1)$ is in fault. Phase 2 of the protocol is no longer required. The post-phase (Sec.6.4.6.1) of the protocol is initiated. Each ARMOR after receiving all the messages examines them in order to identify the faulty ARMOR if any.

To recapitulate, three of the assumptions for the signed messages used in the traditional BG protocol (which also hold, albeit with a certain coverage, for Z) are:

A. A loyal general's signature cannot be forged,

B. Any alteration of the contents of his signed messages can be detected.

C. Anyone can verify the authenticity of a general's signature.

A. is sought to be satisfied by the following mechanisms:

- Each ARMOR receives the signature of other ARMORs in a specific data structure which is cleared after specific periods of time thereby making recreating later in time more difficult.
period after which the receiving structure is cleared is determined by how long after a diagnostic is run can confidence be placed on the integrity of the ARMOR. This is a tunable parameter for each logical ARMOR.

- The entries in the Golden Signature Table (GST) that the ARMOR has for checking upon receiving a signature is in a different form from the form of the received signature. For example, the GST may have a regular expression denoting all valid signatures, while the received signature will be one specific instance. While the ARMOR can check if the received signature satisfies the regular expression, it is assumed unlikely to generate a valid signature by expanding the regular expression.

B. is sought to be achieved by having the data signature that is appended in the message not be computed only on the existing LT (that does not include the new tuples). Instead, the signature is based on the message itself (i.e., the new tuples). Therefore, if a malicious ARMOR changes any part of the message from a loyal general, it will also have to appropriately modify that loyal general’s data part of the signature which can be assumed to be a sufficiently unlikely event.

C. is achieved because each participant has the golden signature of all other participants, and knows the ID of all other participants.

Under these assumptions being satisfied, an ARMOR can detect a faulty ARMOR under the following conditions:
- It receives a malformed message as discussed above.
- It receives different values in messages from the same source. Suppose, A(I+1,1) is faulty and it sends \([(A_1,D_1),\text{UPDATE}]\) to A(I+1,2), \([(A_2,D_2),\text{UPDATE}]\) to A(I,1). Then, after A(I+1,1) forwards its received message to A(I,1), A(I,1) would have the following two messages (among others):

  \[
  [(A_1,D_1),\text{UPDATE}]: S_{A(I+1,1)}: I_{A(I+1,1)} \\
  and [(A_2,D_2),\text{UPDATE}]: S_{A(I+1,1)}: I_{A(I+1,1)}: S_{A(I+2,1)}: I_{A(I+2,1)}
  \]

Since both of the messages are well-formed, and because of assumption B above, it concludes that the source A(I+1,1) is faulty. It is to be noted that because there is only one faulty ARMOR, a majority of the ARMORs will detect and flag the faulty ARMOR even if the faulty ARMOR had manifested its fault to only one of them.

- The signatures in the message do not match with the golden signatures at the receiving ARMOR. The assumption about signatures is that a faulty ARMOR can never generate a correct signature. Under that assumption, for an ARMOR to be diagnosed as faulty, its signature must have mismatch in every message and also this will then be raised by all the other ARMORs.

Alternate version of Phase 2

If the protocol is invoked in response to a flagging of error from a lower level (and not due to a state change), then there really is no state update that has to be communicated consistently to all the participants. Therefore, there is no need to run the phase 1 of message exchange detailed earlier. Since the error detection mechanism described above is dependent upon these rounds of message exchange, it is necessary to provide an alternate mechanism for diagnosis if phase 2 has to be invoked in isolation. This protocol takes the form of exchange of signature messages among the participants.

\textit{t2.1:} To start with, each participant reads in its control signature from its Runtime Signature Table and computes its data signature from the current copy of its Location Table. It forms its composite signature by appending the control and the data signatures. It then sends out a copy to each of the other participants in the protocol.
In the next time step each of the participating ARMORs has $M+N-1$ signatures from each of the other ARMORs. (If it does not receive a signature message from any of the other ARMORs, it flags it as the missing default value, which is of course different from its Golden Signature.) The ARMOR then locally compares each of the received signatures with the golden copy of that ARMOR's signature. The golden copy is obtained by concatenating the golden control signature from the Golden Signature Table, and it generates the data signature by performing the hashing function on its entire Location Table (same rank replica), or on a subset thereof (a parent rank replica). A mismatch between the golden and the received signatures causes the ARMOR to flag an error in the source ARMOR. Note that this assumes sufficiently high coverage for the ARMOR signatures such that an incorrect ARMOR cannot generate a correct signature even once.

If no fault is detected, then each ARMOR updates its LT and the initiator generates the permission bits to execute the installation or uninstallation.

### 6.4.6.1 Post Phase of Level 4 Protocol: Recovery

On detection, the following recovery strategy is executed.

- If any of $A(I+1,J)$ (say, specifically $A(I+1,j_0)$) is in error, that will be detected by all $A(I,K)$ (among others). Each $A(I,K)$ sends an authenticated message to $A(I,1)$ asking for recovery. $A(I,1)$ then starts the recovery by sending the RECOVER message to $D(I+1,j_0)$. This message carries with it the authenticated RECOVER messages sent by all of $A(I,K)$, $K=2,...,M$. $D(I+1,j_0)$ does a this validation check and only if successful initiates recovery of the failed ARMOR $A(I+1,j_0)$. This extra round of authorizing recovery by all ARMORs at the I level is to prevent a rogue I level primary from initiating recovery of a fault-free $(I+1)$-level ARMOR.

- If any of $A(I,J)$ (say, specifically $A(I,j_0)$) is in error, that will be detected by all $A(I+1,K)$ (among others). Each $A(I+1,K)$ sends an authenticated message to $A(I,1)$ asking to initiate recovery of $A(I,j_0)$, if $j_0 \neq 1$; else each sends the recovery message to $A(I,2)$.

- If $A(I+1,1)$ was not the erroneous entity, then the temporary update agreed upon after phase 1 is made permanent. This can be looked upon as a two-phase update of the state of the ARMORs, the two phases being mapped to the two phases of the protocol described above.

### 6.4.6.2 Special Cases

1. **ARMORs from 3 adjacent ranks are involved.** The control signature of an ARMOR is loaded only in ARMORs whose ranks differ from its own by not more than 1. Therefore in this case, the control signature is not available at all the participants. So, the signatures that are appended to the messages for this case are only the data signatures.

2. **FTM installs or uninstalls SM.** Since the FTM is the highest-ranking manager without any rank of ARMORs above it, it cannot run the typical case of protocol Z. However, the goals of consistently updating all the FTM replicas and diagnosing any faulty FTM replica must still be satisfied. Therefore, the Daemon on the node on which the Surrogates Manager is to be installed is involved in the run of the protocol. Since the Daemon does not have a shared LT with the FTM, the only signatures appended to the messages are the control signatures. The control signature of the FTM is available with all the Daemons, as well as all the Daemons' signatures are loaded into the FTM. Both of this is done when the FTM installs the Daemon on a node after the node has registered with the FTM. The payload contains a single tuple $(SM,D_{SM})$ which is updated on all the FTM replicas.
6.4.6.3 Daemon Participation

It is the daemon that ultimately handles the installation or uninstallation of an ARMOR on the local node. Hence, the daemon has to be involved in the Level 4 protocol. The daemon's involvement in the protocol occurs under the following cases:

1. When the daemon receives a MSG_DAEMON_INSTALL_ARMOR message from a manager, it also receives with it the Golden Signature of the manager. It stores the signature in its Golden Signature Table (GST) and adds this information in its Local ARMOR Management table as [ARMOR ID, Manager ID, Pointer to the GST entry for the Manager].

2. When a manager sends an MSG_DAEMON_UNINSTALL_ARMOR message to a daemon, it has to furnish proof that it has taken part in Level 4 prior to the operation and that it has the consent of all the other replicas of itself for the state change. This proof is in the form of permission bits that get set after termination of Level 4. The uninstalling manager sends the permission bits along with its MSG_DAEMON_UNINSTALL_ARMOR message to the daemon. It then clears out the permission bits. Actually, the clearing of the bits is not done immediately. If a probabilistic analysis enables us to conclude about an ARMOR's validity for a time T after having passed Level 4 or Level 3, then the permission bits are cleared after time T. This takes care of the situation where one ARMOR does not initiate Level 4 and unilaterally decides to perform an uninstall. It also sends the signature of itself and all its replicas along with the message. These signatures are checked by the daemon with the entry in the GST. In case of a mismatch, the daemon does not carry out the uninstallation.

It is to be noted that the daemon can do the checks only at uninstallation time, not at the time of install, because at that time it does not have the signature of the installing entity. It can be argued that this may be considered tolerable because the worst scenario is installation of some useless ARMORS by a malfunctioning manager, while uninstallation is definitely not tolerable.

6.4.7 Optimizations

Even without doing an experimental evaluation, it is quite intuitive that the above protocol will be quite costly. The following optimizations are suggested.

1. **Probabilistic confidence estimation.** Say Level 3 or Level 4 has been run at time \( t_0 \) and all participating ARMORS have been found to be valid. Now, a trigger event for Level 4 occurs at time \( t_1 \). Phase 2 of the protocol need not be run at time \( t_1 \) if enough confidence can be placed on the diagnosis at \( t_0 \).

   \[
P(\text{ARMOR is okay at } t_1 \mid \text{ ARMOR was found okay at } t_0) \propto 1/(t_1 - t_0)
   \]

   Such that if \( t_1 = t_0 \), then \( P = 1 \). For every ARMOR, there is a confidence threshold (CTh) that can be set such that if

   \[
P(\text{current time} > CTh)
   \]

   Then it can conclude that the other ARMOR is okay at current time. If among the participants of Level 4, above condition holds for every ARMOR with respect to each other ARMOR, then phase 2 of the protocol is omitted.

2. Phase 2 of iteration \( I \) of the protocol is run in parallel with phase 1 of iteration \( I-1 \). This raises the problem of race condition in the update of the ARMOR Location Table, as illustrated in Figure 19.

   Note that "Update ARMOR Location Table" in phase 1 is the update to the temporary ARMOR Location Table, but the signature computation is also done on this temporary ARMOR Location Table. Therefore, in the above case the signature will not match even for a correct entity.
To overcome this problem, we will have to store two LTs and update each alternately. This is similar to the principle applied in distributed checkpointing where two checkpoints are maintained and each is written to alternately.

![Figure 19: Possible race condition in parallel execution of phases 1 and 2](image)

3. The triggers for the protocol have been mentioned above. We can specify aggregation criteria in addition to these trigger events. The aggregation criteria will enable the protocol to be executed only when a certain threshold number of events satisfying any of the trigger events occurs. For example, rather than executing the protocol on every single ARMOR installation, it is executed when a bunch of ARMORS is installed. This is particularly significant because a common scenario in Chameleon is a surrogate manager being initialized and installed by the FTM, and the surrogate manager installing a group of ARMORS to set up the execution environment. In such a scenario, we would want the protocol to be executed once when the initial group of ARMORS have been installed by the surrogate manager rather than for every single installation.

4. The signatures are sent out by each of the participants at the beginning of the second phase of the protocol. The reason why this has to follow the first phase is to have the ARMOR Location Table s updated. However, the transmitter in phase 1 A(I+1,1) can send out its signature piggybacked on the messages at time T2 of phase 1, because it need not wait for termination of phase 1 to decide on its update. This would not save on the number of rounds of the protocol, but would save M+N-1 messages.

5. Consider the condition that the participants cannot agree on a common value after the voting at the end of phase 1 (in which case in the traditional interactive consistency protocol, they would choose the default value). Such a condition is unique to the case where the transmitter of phase 1 A(I+1,1) is faulty. Therefore, phase 2 can be avoided altogether and recovery under the case of error in A(I+1,1) can be started.

6.5 Interaction Among Detection Levels

The four different detection levels that have been presented do not work in isolation, but interact among each other to provide a framework for composing resilient ARMORS. To understand their interactions, we first consider each of their internal trigger conditions (i.e., trigger is not from another detection level) and then, their external trigger conditions. Finally, we give an example with a replicated, single-threaded application to show how an error may propagate up the hierarchy, or get trapped at a certain level. But first, we consider events in a sample Chameleon configuration and how the different error detection protocols are activated with respect to these events.

Figure 20 shows the configuration where there is the FTM in primary-backup, a single Surrogate Manager, a single Execution ARMOR overseeing a standalone application running in simplex mode. Let us assume that the periodicity of the signature exchange in L3 is τ. The timestamps and the associated events that occur as part of the normal functioning of the Chameleon environment are presented in
columns 1 & 2 respectively, while column 3 shows events associated with the detection protocols that are triggered by the events in column 2.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Normal Event</th>
<th>Error Detection Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>F1, F2 exchange signatures (L3). They will participate in L3 at t0+τ, t0+2τ, ...</td>
<td>F1,F2 exchange signatures (L3). They will participate in L3 at t0+τ, t0+2τ, ...</td>
</tr>
<tr>
<td>T1</td>
<td>F1 installs SM</td>
<td>F1 loads SigF1 to DSM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L4 between DSM, F1, F2. {SM, D_SM} entry added consistently to F1, F2</td>
</tr>
<tr>
<td>T2</td>
<td>SM installs Execution ARMOR</td>
<td>L4 between SM, F1, F2. {SM, D_SM} entry added consistently to SM, F1, F2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SM loads SigSM to DEA</td>
</tr>
<tr>
<td>T3</td>
<td>Application starts executing</td>
<td>L4 between EA, SM, F1, F2. L4 between SM, F1, F2. {App, D_EA} entry added consistently to EA, SM, F1, F2.</td>
</tr>
<tr>
<td>T4</td>
<td>Application has abnormal termination</td>
<td>EA detects Application terminated abnormally. It sends notification to the SM.</td>
</tr>
<tr>
<td>T5</td>
<td>Application restarted</td>
<td>EA restarts Application from checkpoint.</td>
</tr>
<tr>
<td>T6</td>
<td>Application terminates normally</td>
<td>EA detects Application terminated normally. Notifies SM with result.</td>
</tr>
<tr>
<td>T7</td>
<td>SM uninstalls EA</td>
<td>L4 between SM, F1, F2. {EA, D_EA} removed consistently from SM, F1, F2. DEA checks SigSM and Permission Bits.</td>
</tr>
<tr>
<td>T8</td>
<td>F1 uninstalls SM</td>
<td>L4 between D_SM, F1, F2. {SM, D_SM} removed consistently from F1, F2.</td>
</tr>
</tbody>
</table>

Figure 20: Sample Chameleon Configuration

The Level 1 mechanisms are executed whenever the appropriate code in the ARMOR is encountered, without having any specific periodicity. For example, in the case of internal assertion checks, whenever there is an assertion check built into an element in the ARMOR, Level 1 is taken to be invoked. Some general rules for this technique can be postulated. When a particular message type gets delivered to an element, it should be because the element subscribed to that message type and the process_message function of the parent compound forwarded the message appropriately. So, quite logically, an assertion check is inserted into the message processing switch statement of an element to catch when a message not covered by any of the cases is received. When an error is diagnosed because of an assertion check failing, the corresponding element notifies the daemon. From this point forward, it is identical to the daemon detecting an error as part of Level 2 error detection.
The Level 2 mechanisms are executed by the local daemon. As we saw in the preceding discussion, this may be triggered by a Level 1 method. The daemon also verifies an ARMOR is functioning correctly by checking the signature that is generated by the ARMOR within Level 1. This check is performed by the daemon when it receives a message from the ARMOR for routing to another ARMOR. Again, in response to a smart heartbeat sent by the daemon, diagnostic techniques internal to the ARMOR may be triggered.

The Level 3 signature exchange actions also happen with a certain periodicity. This is the only trigger for this level currently. When an error is detected by any of the lower levels, the manager of the entity diagnosed as faulty is notified for initiating recovery. There is a possibility after Level 3 that an error is detected, but not identified to any particular ARMOR. In such a case, the phase 2 of the Level 4 protocol is invoked. This serves as the only interaction between Level 3 and Level 4.

Now let us consider a replicated, single-threaded application running on a single node in the system. Let us consider the various entities that may fail and could affect successful completion of the application. The application may terminate abnormally. The first level of defense against this is the Execution ARMOR which directly monitors the application for any illegal signals that are raised by it. If the application exhibits a value-domain error, then this would be captured by the Voter ARMOR at the time of voting on the results of the application, either intermediate or final voting. The Execution ARMOR may itself fail. The first line of defense is the Level 1 error detection. If the monitor thread detects this failure (by detecting lack of progress, etc.) then it informs the daemon and the daemon informs the ARMOR’s manager for recovery. If an assertion check put in the ARMOR code captures the error, it again flags it to the daemon. If internal techniques fail, the daemon may detect the failure of the ARMOR either because it does not respond to the daemon’s heartbeat, or the signature that the ARMOR sends out does not tally with the entry in the repository with the daemon. At the next level, if the ARMOR is replicated, it and its replicas take part in signature exchange rounds. The ARMOR on its own sends out its signature to all its replicas. Also on receiving signatures from its replicas, it does a local comparison and can flag an error. Finally, whenever the ARMOR installs (or uninstalls) another ARMOR, it acts as a transmitter for Level 4 protocol messages and then participates in diagnostic rounds. If at any level, the ARMOR is detected as faulty, its functioning is suspended by its manager and recovery initiated. In this manner, the whole suite of detection techniques can hierarchically provide a very high coverage for the ARMOR’s fault tolerance. Note that when the application is executing, it can execute for periods when the surrogate manager, or some other manager is absent (because it is in the process of being recovered). Therefore, the application may be oblivious to errors being detected and recovered in some of the ARMORS.

To conclude, we have seen that several errors in an ARMOR may be detected at a level quite close to the ARMOR, in which case the higher overhead upper levels do not see the error at all, or the error may bubble up to a higher level. Also, a lower level may provide notification to a higher level about any potential malfunction in an ARMOR, so that the more powerful higher level may execute a higher assurance detection diagnostic. These different error detection levels working in cooperation are expected to provide a suitably high coverage for the Chameleon ARMORS.
7 Availability Analysis of Chameleon

This section describes the availability analysis done on the Chameleon architecture. The analysis determines the availability of the system from the application point of view. The analysis was done using Markov models for the various entities in Chameleon and getting the availability of each of these entities. The analysis was required to prove design objectives. It tests whether the current Chameleon architecture satisfies the goals of the design. The presented availability analysis does not take into account the multiple error detection strategies that were described earlier in this report.

7.1 An Overview of the Model

To keep the task of constructing the Markov model manageable, we restrict the models as follows:

- Availability is assessed from the perspective of the user application.
- Undetected errors cause the system to restart (possibly with a high latency). An uncovered error refers to any error that goes undetected by any entity in the Chameleon environment.

From the time the user makes an execution request to the FIM to the time he collects the results, the Chameleon system goes through various stages. Once the FIM selects and installs the Surrogate Manager, the FIM initiates the flow-of-control graphically represented in Figure 21. Figure 21 shows an example execution strategy for an application run in TMR.

Much of the time, however, is spent in the “TMR Exec” stage in the above figure. The other states simply install s, uninstall ARMORS, send the application code, and send results. While these actions do take a finite amount of time, they add little to the overall execution time of the application. The Markov model, therefore, concentrates on the time spent in “TMR Exec.” Stage.

When Chameleon enters the “TMR Exec” stage as described above, the Surrogate Manager has already installed the three Execution ARMORS and the Voter ARMOR necessary to realize the required level of fault tolerance. The “TMR Exec” stage, itself, may be decomposed into two phases: the execution phase and the voting phase. Of the two, the execution phase is the most complex and will be analyzed in the model.

![Flowchart](image)

Figure 21: Flow-of Control for the TMR execution strategy in Chameleon

7.2 Description of the Model

Two models are developed for Chameleon.
1. A model where all failures are recoverable. This means that the availability of the system will be a steady state one, and will not degrade with time. This model typically stands for what Chameleon hopes to achieve. In an ideal environment, all the errors will be recovered from by using a variety of error detection and recovery techniques. The availability of the system will remain the same over a wide range of time, making it possible to give availability guarantees when used in long running missions.

2. A model with unrecoverable failures. This model is similar to the earlier model, except that there is an absorption state in the Markov model for each entity. The availability of such a system will decrease with time.

The model consists of Markov descriptions of all the entities in Chameleon. The entities include the application, Execution ARMOR, Heartbeat ARMOR, Daemon, Surrogate Manager and FTM.

7.2.1 Model with all Failures Recoverable

The application. Applications are managed by the Execution ARMOR. When an application fails, the failure is detected by the Execution ARMOR. The Execution ARMOR is also the one, which does the recovery of the application. We have created two descriptions for the application, one for the application running in the simplex mode and another for the application running in TMR mode. The description of the application executing in simplex mode is shown in Figure 22. The raw failure rate of the application is $\lambda_{app}$. This is taken to be a constant and is an input to the model. The rate at which detection is done by the Execution ARMOR is $\lambda_{A}$ and is given by the expression:

$$\lambda_{A} = \lambda_{app} * c_{app} * P_{exec}. $$

Where $c_{app}$ is the coverage that an error will be detected by one of the detection levels and $P_{exec}$ is the availability of the Execution ARMOR. The recovery of the application is done by the Execution ARMOR with the recovery rate $\mu_{A}$

$$\mu_{A} = \mu_{app} * P_{exec}. $$

Where $\mu_{app}$ is the application repair rate. Here, repair time, which is the inverse of the repair rate, corresponds to the time from detection of failure to the time when the application is ready to run, say from the last checkpoint.

In the Markov description, the application is available if it is in the OK state. State A denotes the state of application failure, state E denotes the Execution ARMOR failure, H - the host failure, D - Daemon failure,
and state U denotes an undetected failure. We assume that all failures are recoverable. The failure rate for uncovered failures is,

$$\lambda_U = \lambda_{app} + \lambda_{exec} + \lambda_{host} + \lambda_{daemon} - (\lambda_A + \lambda_E + \lambda_H + \lambda_D).$$

For the application running in TMR mode, the model is shown (partly) in Figure 23. It is not shown in entirety due to lack of space. The model has four stages and each consists of a number of states as follows:

1. The OK state, where no failures have occurred.
2. The states A, D, E, H and U correspond to single failures. The system is available if it is in any of these states or in the OK state.
3. The states AA, AD, AE, AH, AU, DD, DE.....UH, UU, model states with two failures in different nodes. As an example, in Figure 23, the states originated from A and state U are shown.
4. The states AAA, AAD....UUH, UUU, model states with three failures in different nodes. As an example, in Figure 23, the states originated from AU are shown.

**The Execution ARMOR.** The Execution ARMOR (which manages the application) is supervised by the Surrogate Manager. The Execution ARMOR failure is detected by the Daemon on the same node and the recovery is done by the Surrogate Manager. The expressions for the failure and recovery rates of the Execution ARMOR are:

$$\lambda_E = \lambda_{exec} \cdot c_{exec} \cdot P_{daemon}.$$  
$$\mu_E = \mu_{exec} \cdot P_{SM}.$$

The description of the Execution ARMOR is similar to the description of the application, with states OK, E, D, H and U. Execution ARMOR is available if it is in the OK state. This Markov description is valid for any ARMOR, which is managed by the Surrogate Manager.

**The Daemon.** The Daemon failures are detected by the Heartbeat ARMOR. The Daemon is repaired by the FTM. The expressions for the failure and recovery rates are,

$$\lambda_D = \lambda_{daemon} \cdot c_{daemon} \cdot P_{heartbeat}.$$  
$$\mu_D = \mu_{daemon} \cdot P_{FTM}.$$

The description for the Daemon has the states OK, D, H and U. The Daemon is available if it is in OK state.

**The Surrogate Manager:** The ARMORS like Surrogate Manager and the Heartbeat ARMOR are managed by the FTM. Their failures are detected by the local Daemon and their recovery is done by the FTM. The expressions for the failure and recovery rates of the Surrogate Manager are,

$$\lambda_{SM} = \lambda_{surrogate} \cdot c_{SM} \cdot P_{daemon}.$$  
$$\mu_{SM} = \mu_{surrogate} \cdot P_{FTM}.$$

The expression for the Heartbeat ARMOR is similar. The description of the Surrogate Manager has the states OK, SM, D, H and U. The Surrogate Manager is available if it is in the OK state.
The FTM: The FTM can operate in several execution modes, e.g., TMR mode on a dedicated machine with a single voter or a distributed TMR with one or duplicated voter. Hence, the availability of FTM (\(P_{\text{FTM}}\)) can be easily obtained by solving a Markov model, corresponding to the selected execution mode of the FTM. For the analysis purposes, it is assumed that the FTM operates in a primary backup mode. The FTM cannot recover if the primary and the backup fail at the same time. Detection of the failure of a copy of the FTM is done by the Daemon. Hence, the failure expressions for the FTM are,
The description of FTM has the following states OK, Fail_1, Fail_2, D, H and U. Fail_1 corresponds to the state with one replica failing. Fail_2 corresponds to the state with both replicas failing (it is an absorption state).

Finally the host/node failures are detected by the Heartbeat ARMOR and are recovered by the FTM. The host recovery is typically done by migrating all ARMORS to a new host/node. The expressions for failure and recovery rates are

\[
\lambda_H = \lambda_{host} \cdot c_{host} \cdot P_{heartbeat}.
\]

\[
\mu_H = \mu_{host} \cdot P_{FTM}.
\]

7.2.2 Model with Unrecoverable Failures:
The Markov descriptions for the model with unrecoverable failures is the same as those of the model with all recoverable failures with addition of an absorption state in each of the descriptions. There is a transition from the failure state of an ARMOR (e.g., State A in the case of application) to the absorption (fail) state, with rate \( \lambda_{fail} \). This rate is constant and is typically low and causes the availability of the system to degrade with increase in time.

7.3 Analysis
An iterative method is used to compute components’ availability. In this method, an initial assumption is made about the values of \( P_{FTM} \), \( P_{heartbeat} \) and \( P_{daemon} \). With these values, availability of other entities is found out. These values are used in the next iteration. A convergence condition is checked to end the iterations. The analysis of the models was done with the help of SHARPE, a tool for analytical analysis of models. The steps in this iterative method are:

1. Assign initial values to \( P_{FTM} \), \( P_{heartbeat} \), \( P_{daemon} \).
2. Calculate \( P_{SM} \), \( P_{daemon} \), \( P_{heartbeat} \), \( P_{exec} \) and \( P_{app} \) in that order.
3. With the new availabilities, recalculate \( P_{FTM} \).
4. Check for convergence of \( P_{app} \). If not converged, go to step 2 for next iteration.

The convergence criterion is that the change in \( P_{app} \) after an iteration should be less than 0.0001.

\[ \text{i.e., } |P_{app,i} - P_{app,i+1}| < 0.0001. \]

Where, \( P_{app,i} \) is the value of \( P_{app} \) got after \( i \) iterations. The convergence checking was done manually. It was found that convergence takes place within 10 iterations.

Using the two models the following analysis was conducted:

- **Analysis of effect of change in failure rates.** We need to know the dependence of the availability of the system on the failure rates of different entities. This is required so that we can get an insight into how critical an entity is to the system. To do this, we assume default values for the failure rates for various entities. Then, a failure rate for one entity is decreased by a factor of 10. In our analysis, we kept the default values of failure rates to be 0.1/hr, and decreased each entity’s failure rate to 0.01/hr. This is done for all the entities in the system. The availability of the system is computed for each of
the configurations (as given in section 7.2). The availability was calculated over a period of three years. The results are shown in Figure 24 and Figure 25.

- **Analysis of the effect of the application configuration.** The application can run in several modes and it is critical to know in which configuration the application should execute to get maximum availability. We compared the application running in TMR mode and running in simplex mode. The availability was calculated for a period of three years. The input parameters were taken to be the default values. The results are shown in Figure 26 and Figure 27.

### Parameters used

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_{app}, \lambda_{exec}, \lambda_{heartbeat}$</td>
<td>0.1/hr</td>
</tr>
<tr>
<td>$\lambda_{daemon}, \lambda_{host}$</td>
<td>0.1/hr</td>
</tr>
<tr>
<td>$\lambda_{SM}$</td>
<td>0.1/hr</td>
</tr>
<tr>
<td>$\lambda_{FTM}$</td>
<td>0.01/hr</td>
</tr>
<tr>
<td>$\lambda_{fail( unrecoverable)}$</td>
<td>0.0001/hr</td>
</tr>
<tr>
<td>All coverage factors</td>
<td>0.9</td>
</tr>
<tr>
<td>$\mu_{app}, \mu_{exec}, \mu_{heartbeat}$</td>
<td>5 sec</td>
</tr>
<tr>
<td>$\mu_{daemon}$</td>
<td>5 sec</td>
</tr>
<tr>
<td>$\mu_{SM}$</td>
<td>5 sec</td>
</tr>
<tr>
<td>$\mu_{FTM}$</td>
<td>5 sec</td>
</tr>
<tr>
<td>$\mu_{H}$</td>
<td>5 min</td>
</tr>
<tr>
<td>$\mu_{U}$</td>
<td>30 min</td>
</tr>
</tbody>
</table>

![Figure 24: Variation of Availability for the Model with all Recoverable Failures](image)

Figure 24: Variation of Availability for the Model with all Recoverable Failures
Figure 25: Variation of Availability for the Model with all Unrecoverable Failures

Figure 26: Impact of Application Configuration on the Availability for the Model with all Recoverable Failures

Figure 27: Impact of Application Configuration on the Availability for the Model with Unrecoverable Failures
The following are the results obtained from the conducted analysis:

- The availability of the application is maximum if the availability of the Daemon and the Execution ARM increases (see Figures 24 and 25). The entities affecting the availability of the system in decreasing order are Daemon, Execution ARMOR, application, FTM and SM. The Daemon affects the most as its failure brings down all the entities in the node. The Execution ARMOR is the one, which is closest to the application and is the only one, which communicates with the application. It is also responsible for recovery in case of application failures. Hence, decrease in failure rate of Execution ARMOR increases the availability of the system to a large extent. To increase the availability of the system, it is more efficient to make the Daemon more failure resilient rather than the SM or the FTM. In the future,

- The availability of the application increases more if the failure rate for the Daemon is decreased, than if the failure rate for the application itself is decreased. This is a non-intuitive result and points to the fact that the Daemon failure can affect several entities, which affects the system availability more than the application failure itself.

- The dependency of application on SM and FTM is not high. This reinforces the design principle that the system is able to continue despite failures of managers, and that no one component failure can bring down the system entirely. The managers are required only in case of failures of ARMORS (e.g., Execution ARMOR), and do not play a major role when the application is running. They are crucial in setting up the system configuration and starting the execution, but do not affect the availability of the system heavily from then on.

- Application running in TMR mode has higher availability than application in simplex mode in the case of recoverable failures. This is not the case when we consider the model with unrecoverable failures. An application running in TMR is supported by more ARMORS and consequently there is a higher probability of unrecoverable failures. Hence, the application running in simplex mode has a higher availability than application running in TMR mode (see Figure 27). This leads to the conclusion that replication not necessarily guarantees increase in the application availability.

In summary, this work shows that the Chameleon architecture satisfies the design principles that the system should be able to run despite failures of individual entities, i.e., there is no single point of failure. We found that that the Daemon is the most critical component in Chameleon, from the point of view of the application availability. This work does not include the multiple levels of error detection mentioned earlier in the report. We do not provide insights into the overhead of the error detection mechanisms and into the error latencies in the system due to the various error detection mechanisms. The model needs to be improved to include non-exponential distributions for error recovery times.
8 Related Work

Current approaches for providing fault tolerance in a network of unreliable components are based mainly on exploiting distributed groups of cooperating processes. Consequently, the primary focus is on providing a dedicated software layer to maintain and coordinate reliable communications among groups of processes.

ISIS [BIR94] and Horus [BIR96], [REN96], provide tools for programming with process groups. By using these tools, a programmer can construct group-based software that provides reliability through explicit replication of code and data. Totem attempts to provide high performance and soft real-time guarantees to applications, [MOS96]. Transis incorporates multicast services that are capable of recovering from network partition failures, [AMI92], [DOL96]. Rampart addresses security aspects of group communication by providing tolerance for malicious intrusions, [REI93]. Although reliability may be achieved using these approaches, "fault tolerance," Birman notes [BIR93], "is something of a side effect of the replication approach."

There exist, however, examples of systems, which explicitly address the issue of fault tolerance. SIFT was one of the earliest attempts to propose a completely software-based approach to fault tolerance through loose synchronization of processors and memory, [WEN72]. Delta-4 sought to define and design an open dependable distributed architecture through the use of group communication layers built on top of an atomic multicast protocol, [POW94]. Piranha, a CORBA-based, [OMG95], extension to Horus addresses the issue of service availability in distributed applications by using a highly sophisticated ORB that provides failure detection, [MAF97]. AQUA architecture provides a flexible approach to building dependable, object-oriented distributed systems while offering a standard CORBA interface to applications [CUK98].

"Wolfpack", Microsoft clustering technology provides clustering extensions to Windows NT for improving service availability and system scalability, [MIC97]. At Sun Microsystems work has been done on Ultra Enterprise Cluster design to provide highly available data services, [SUN97]. Work at Lucent Bell Labs has focused on providing availability of applications through reusable components for automatic detection and recovery of failed processes, [HUA93].

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

Many techniques have been proposed to monitor the inter-instruction control flow using signature monitoring. The reason for this interest is that it has been shown that control flow errors account for between 33% and 77% of all errors [OHL92, MAD92], depending on the fault assumption (e.g., register bit-flip or pin-level stuck-at-x) and system characteristics (e.g., RISC or CISC processor), making it important to detect this class of errors. In general, signature monitoring involves: (i) monitoring the instructions that the processor is executing, and (ii) having access to pre-computed signatures. There are two broad classes of techniques with regard to (ii) above - those which store the checking information in a separate watchdog memory [EIF84, MIC91], and those which embed the signatures in the application, the granularity generally being a single basic block [OHL95]. The Embedded Signature Monitoring [ESM] techniques suffer from loss of performance and are unable to handle off-the-shelf applications. The
former however suffers from high hardware complexity for the watchdog processor. In Chameleon environment, we do not have any specialized processor that can act as the watchdog processor. Also we are trying to do control flow checking of the ARMORs, which can be modified by the system designers, and not the user application. Therefore, the Embedded Signature Monitoring technique is suitable for our purposes. Rather than using a unique signature per basic block, the signatures can be inserted into the ARMOR code according to the ARMOR designer's judgement. This is meant to reduce the overhead associated with generating a signature for every basic block and to free us from the constraint of writing a tool to analyze arbitrary code into basic blocks for heterogeneous platforms. Also, with ESM techniques, there is generally a watchdog timer which is used to detect the absence of a signature checking point. In Chameleon, the equivalent functionality is provided by a Monitor Element internal to the ARMOR which does detection for livelock in ARMORs.

The problem of guaranteeing consistent information among multiple replicas is a much-studied problem in the design of reliable distributed systems. This was called the "Byzantine Generals Problem" by Lamport et al in their seminal paper [LAM82] which introduced a completely unconstrained fault model. The solution with signed messages was also proposed in the same paper. It briefly hinted at detection of faulty processors through examination of their signatures, but did not expand on this topic. Optimizations to the general algorithm were proposed under multiple failure modes in [THA88] where Thambiduraj and Park proposed an algorithm for Interactive Consistency that retained resilience to the arbitrary fault mode, while tolerating more faults of simpler kinds than the standard Byzantine-resilient algorithms. The problem of locating the faulty processors in a Byzantine protocol is a less-studied problem. Some previous work has been done by Walter [WAL90], Shin and Ramanathan [SHI87], and more recently by Ayeb and Farhat [AYE98]. However, the algorithms proposed suffer from one or more of the following problems: they are off-line, additional rounds of message exchange are required, or there is no deterministic upper bound by when all malicious entities are detected. For our purposes, we have the simplifying assumption of a single failure among the participants in one run of the protocol, and we have a signature mechanism that imposes certain restrictions on faulty ARMORs (such as a faulty ARMOR never being able to generate its correct signature). Using these devices, we propose a simpler algorithm for error detection. An illuminating example of application of the Interactive Consistency algorithms in a real-world distributed system was provided by GUARDS [POW98], a system currently being developed by a consortium of European companies and academic partners.

The issue of replica consistency occurs quite frequently in the literature of reliable distributed systems. These works include examination of the concepts of logical time [LAM84], causal ordering in message delivery [BIR87] and consensus protocols [CHA96]. Applications for such work have been found in a large volume of work on Group Communication. Mention may be made of Rampart [REI96], Totem [MOS96] and Horus and Isis [BIR93, BIR96]. Our approach augments the data consistency approach with diagnosis performed during the process of consistently distributing data. Also, we incorporate data distribution across replica groups whereby a select subset of state is made consistent between replicas of different ARMORs.

References


Appendix A: Instantiating Element and ARMOR Objects

This appendix discusses the actions taken whenever a new element object is created. In this context, "created" refers to the process of instantiating an element object as part of a compound, not the act of designing (and implementing) a new element. Once the instantiation process has been described for elements, it can easily be extended to include compounds and ARMORS. Special attention will be given to ARMOR instantiation when appropriate.

A.1 Instantiating Within an ARMOR

As mentioned in section 2.1, a single element cannot exist in isolation—it must ultimately be part of a top-level armor_t object that is capable of sending and receiving messages. After reading section 2.3, it should be clear that:

The armor_t "layer" contains the necessary functionality to distinguish between intra- and inter-ARMOR messages.

- The armor_t "layer" relies on an element to forward outgoing messages and to receive incoming messages from the local daemon.

Elements, themselves, do not possess these capabilities and rely on the message delivery services of their parent compounds to provide them with messages. Because of this, a complete ARMOR is created by instantiating an armor_t object and then populating the armor_t object from the top down. All elements, therefore, know their parent upon instantiation.

The element_t constructor has the following prototype to reflect these observations:

```
element_t::element_t (element_id_t idElement, compound_t *pParent, armor_t *pArmor);
```

Note that when creating an element, you must provide the compound to which the element will belong and the top-level armor_t object (these may be the same object for a single-level hierarchical composition as illustrated in Figure 2[a]). Compounds have a similar constructor with the same restriction that they must be created as part of an existing compound.

Conceptually, armor_t objects are nothing more than a specialized element_t object. But because of the special role they play (not the least of which involves being the only element_t object that can be created without a parent), a special constructor exists for ARMORS:

```
armor_t::armor_t (ARMOR_id_t idArmor, ARMOR_id_t idManager, daemon_id_t idDaemon);
```

Having a constructor like this underscores two important facts:

- The ARMOR must have a manager.
- The ARMOR must be associated with a daemon.

The reliance on this special armor_t constructor, however, has certain side-effects that we will touch upon in the next section.

A.2 Element and ARMOR Factories

Compounds use an element factory to instantiate a specific type of element on demand. Element factories have a create_element() function that takes the same parameters as the element_t constructor in addition to the type of element to instantiate.
Because we only support ARMOR instantiation through the special armor_t constructor presented above, ARMORS cannot be created through element factories—special ARMOR factories exist for this purpose. Segregating ARMOR instantiation from element instantiation implies that elements cannot add entire ARMORS to their composition. As a result, you cannot "fuse" two whole ARMORS together under a top-level armor_t shell. Although not particularly common, such a feature would be nice in a highly-composable infrastructure. But this is really less of a limitation than it would first appear, even if you desired this rather esoteric feature. Since all of the ARMOR functionality exists in its constituent elements and compounds, you can simply add these elements and compounds from the formerly-separate ARMORS into a single ARMOR.

While we are on the topic of element and ARMOR factories, it should be noted that we only support static factories at this point (the object code for the elements in the factories must be linked into the executable that makes use of the factory). In the future, we plan to support dynamic factories that make use of dynamic linking (through dlopen() and related functions that work with shared libraries under most versions of UNIX). With dynamic factories, the executable program does not need to contain any elements at all—they can all reside in dynamic libraries that are loaded upon demand. This setup gives the ARMOR designer the utmost in flexibility and provides the perfect complement to the reconfigurable ARMOR architecture—now, for instance, you could upload new elements during the course of the mission and dynamically substitute these elements into executing ARMORS without having to bring those ARMORS down. Dynamic factories also help keep the memory usage to a minimum—only those elements that are truly being used need to reside in memory.

A.3 Adding Elements to a Compound

All of the composition management functions of a compound are made available through the standard message-passing interface. For example, the compound_t::process_message() function handles messages such as MSG_ADD_ELEMENT, MSG_REMOVE_ELEMENT, MSG_REPLACE_ELEMENT, etc. In this section, we will look at the steps that a compound takes to add a new element to itself:

Using the element type found in the MSG_ADD_ELEMENT message, the compound uses its element factory to instantiate a new element_t object.

- The compound calls the element_t::init() function of the new element, thus invoking the initialization routines specific to each element.

We have already seen that each element should override the element_t::process_message() in order to handle specific types of messages. Similarly, each element should override the element_t::init() function to perform its own initialization routine. For most elements, this simply consists of subscribing to the appropriate messages and of initializing its data structures.

Objects of the armor_t class also have their own init() function. As mentioned in section 3.1.1, the create_armor ancillary program calls the init() function after creating the armor_t object. It is fairly easy to see that the create_armor program is the equivalent of the compound_t "layer" as far as ARMOR instantiation and initialization are concerned.

---

11 Section 3.1.1 gives a fairly thorough discussion of the role that ARMOR factories play when a daemon installs a new ARMOR process.
Appendix B: Message Aliasing

This appendix describes a feature that is closely related to the message delivery services of compounds, and that plays an important role in integrating element functionality. We will begin reviewing some of the key steps in the message delivery process before diving into the message aliasing capabilities of compounds.

B.1 A Review of Message Delivery

Recall that compounds keep a subscription list (section 2.2) that they consult whenever they receive an incoming message. If an element has subscribed to the incoming message type, then the compound forwards the message to the element through the element's `process_message()` function. Most elements implement their `process_message()` virtual function as a switch statement:

```cpp
switch (msg.type) {
    case MSG_TYPE_A:
        // Handle message type A here.
        break;
    case MSG_TYPE_B:
        // Handle message type B here.
        break;
}
```

Note that the programmer must “hard-code” the message type into the `process_message()` function if he wants to use the familiar switch-statement construct.

B.2 “Connecting” Elements Together

In order to provide the utmost in flexibility, it would be nice to have an element’s services automatically invoked in response to some other action taken by the ARMOR. As a concrete example, consider the Execution ARMOR that has been configured to attach itself to the user application through a named pipe (i.e., the Execution ARMOR that has added the app_np_mgmt_t element to itself [section 3.3.3]). Although we did not mention it when discussing the implementation of app_np_mgmt_t, the element subscribes to the message MSG_APP_CONNECT_TO_APP message to begin the processing of opening the named pipe and establishing a connection with the user application.

Under normal circumstances, this would imply that some other element (be it in the same ARMOR or a different ARMOR) would have to send the MSG_APP_CONNECT_TO_APP message to the Execution ARMOR to get the ball rolling. Of course, this means that the ARMOR sending the message must necessarily know that the Execution ARMOR has been configured to connect to the user application. It would be far better to have this functionality automatically invoked if the app_np_mgmt_t element exists in the Execution ARMOR.

The MSG_APP_CONNECT_TO_APP (hereafter MSG_CONNECT) requires that the application’s process ID be provided in the payload area of the message so the app_np_mgmt_t element knows what named pipe to open. By design (not by coincidence), the app_spawn_t element generates a MSG_ANNOUNCE_NEW_APP_PROCESS (hereafter MSG_ANNOUNCE) message whenever a new application process is registered with the Execution ARMOR. As you may very well guess, the MSG_ANNOUNCE message contains the process ID of the application that the ARMOR has just registered.
Since both the MSG_ANNOUNCE and MSG_CONNECT have the same payload fields, we should be able to have the app_np_mgmt_t element act upon the MSG_ANNOUNCE message.

Unfortunately, to do this would require that we manually modify the app_np_mgmt_t::process_message() function by adding another case statement for the MSG_ANNOUNCE message. Although this involves adding only an extra line into the app_np_mgmt_t source code, we would like to do as little tweaking as possible, especially for features that should supposedly espouse the flexible nature of the Chameleon architecture.

Herein lies the role of message aliasing (for lack of a better name)—the output of one element can become the input message of another element. This, in effect, "connects" the elements together. To support this, compounds add another field to their subscription list to contain the alias message type under which the message should be delivered to the element.

In our above example, the app_np_mgmt element would subscribe to the MSG_ANNOUNCE message under the alias of MSG_CONNECT. Note that this does not involve any changes to the source code, as the compound's subscription services can be invoked through the compound's message passing interface, even by a remote ARMOR (such as the manager of the Execution ARMOR, perhaps). Now, the compound will have an entry in its subscription table stating that all MSG_ANNOUNCE messages should be delivered to app_np_mgmt_t as if it were a MSG_CONNECT message. So, during delivery, the compound simply overwrites the message type with the alias message type when delivering to the app_np_mgmt_t element—a fairly efficient means by which two elements may be connected.

A salient point of this ability rests in the fact that this special form of subscription can be performed by sending messages to the compound. Also you will note that it is the compound layer—not the element who sends the message nor the element who ultimately receives the message—that coordinates the message aliasing. From the app.spawn_t element's perspective, it has sent a MSG_ANNOUNCE message and from the app_np_mgmt_t element's perspective, it has received a MSG_CONNECT message.