November 2000

UILU-ENG-00-2216
CRHC-00-05

University of Illinois at Urbana-Champaign

A Smart Voting Subsystem for Distributed Fault Tolerance

G. Rotondi and R. K. Iyer

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

SF 298 MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 2000	

4. TITLE AND SUBTITLE

A Smart Voting Subsystem for Distributed Fault Tolerance

5. FUNDING NUMBERS

6. AUTHOR(S)

G. Rotondi, R. K. Iyer

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

Coordinated Science Laboratory
University of Illinois
1308 W. Main Street
Urbana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER

UILU-ENG-00-2216

(CRHC-00-05)

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

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Chameleon is an adaptive software infrastructure which allows different levels of availability to be supported simultaneously in a networked environment [KALB99]. In this paper, we present a smart voting architecture designed to extend the Chameleon functionalities by a set of services aimed at collecting and validating data across multiple replicas of the same application. The voter topology presented here can be effectively employed to provide an increased security to the environment itself, because it supports the replicated execution of ARMOR objects. Our original approach is in the introduction of an asymmetric signature generation to efficiently validate data coming from the system periphery.

14. SUBJECT TERMS

smart voting, distributed fault tolerance

15. NUMBER IF PAGES

41

16. PRICE CODE

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

TABLE OF CONTENTS

Chameleon is an adaptive software infrastructure developed at the Center for Reliable and High Performance Computing of University of Illinois, which allows different levels of availability to be supported simultaneously in a networked environment [KALB99]. In this paper we present a smart voting architecture designed to extend the Chameleon functionalities by a set of services aimed to collect and validate data across multiple replicas of the same application. The voter topology presented here can be effectively employed to provide an increased security to the enviroment itself, because it supports the replicated execution of ARMOR objects. Our original approach stands in the introduction of an asymmetric signature generation to efficiently validate data coming from the system periphery.

Foreword

The study of fault tolerant computing takes the origins from the past as a separate assessed discipline from the aerospace industry: the first book on the subject dates 1965 [PIER65]; from these pionieristic age, the research has proposed various models and techniques to validate and successfully design fault-tolerant systems focusing on their behavior in presence of malfunctioning induced by hardware components; in order to provide continuation of service or fail-safe operation such techniques try to indentify the origin of faults and the palliative operation to guarantee the design specifications [JOHN89].

As the research has progressed in defining new fault tolerant topologies, the reliability of components has improved over the years leaving even less responsibility to the hardware design, because most of the causes of a system malfunctioning may arise from a software problem, which is of different orders of magnitude more complex than hardware and thence very difficult to validate with formal methods [RAND75]. The problem emerged as a consequence of a common trend to replace hardware functionalities by firmware [OSBO78] started in 1971 with the introduction of the first general purpose microprocessor chip [ASPR97]: most of nowadays control systems performing critical operations rely their functionality on the massive usage of embedded software [BRIE93 ESCH97 LION96 VOAS97a YEH97].

Although what is or is not a "critical system" is often debated, a critical software system is simply a system, where failure, denial of service and so forth could have expensive consequences, such as loss of life, a loss of business, lost property or financial interests [VOAS98b].

The result is that the traditional definition of "fault-tolerance", typical of electrical and computer engineering, which refers to building subsystems from redundant components, is inappropriate when applied to systems performing most of their functionalities in software. Many authoritative fonts have pointed out that software errors are consequence of a lack in specifications [SIWI98]: indeed, considering negligible the occorrence of errors in the "manufacturing" process, the software is subject only to design errors, or to misunderstandings of such formal specifications; consequently, more than 50 statistical models have been developed for the estimation of software reliability over the past 20 years [PHAM95], but comparatively a very little contribution has been given in the techniques for achieving software fault tolerance [LITT91].

The formalization of *software fault tolerance* is a matter of recent years [MARC94]: a computer program is considered failure-tolerant, if and only if [VOAS97a]:

1. is able to compute an acceptable result, even if the program itself suffers from incorrect logic;

2. whenever correct or incorrect, it is able to perform a safe computation, even if the program itself receives corrupted or malicious data during execution.

where the concept of *safe computation* reflects the incidental damages, which can occur to the whole embedded system (hardware and software) as consequence of a software failure.

The last frontier of the research on software dependability is to employ for design commercial off-the-shelf components (COTS) in place of custom embedded systems, with immediate benefits in terms of reduced developement times, availability and maturity of many commercial tools and operating systems [Iyer, 1999 #238].

Although this last aspect of dependable software environments is still at an embrional stage, we can distinguist three main guidelines followed over the years:

① Direct design of failure tolerant software [VOAS97c] running on the top of fault-tolerant operating systems; such systems rely on specialized hardwares to provide a dependable framework, which includes as native features replicated execution [BORG89], reliable group communication, checkpointing and roll-back recovery.

② A separate fault tolerant engine, which implements in software the basic building blocks to provide extra dependability to applications: the services are accessible in a separate process or thread through an interface libray, which maps the calls a proprietary API. These dependable libraries provide high-availability networking services and supply algorithms to identify (and recover) from a dead node. Piranya is a superset of the CORBA environment, which implements dependable functionalities [MAFF97], while Wolfpack [MS97] is an extension of the Microsoft NT Server cluster architecture. RAS is the clustering architecture proposed by Sun Microsystems [SUN97].

③ A total networked environment, which provides fault tolerant services to the application standing on the top, without mandatory request for running on specialized hardware. In order to exibit a failure-tolerant behavior, some environments require a slight modification of the application source code [CUKI98], some others provide the fail-safe functionalities free of charge and at a price of less flexibility, because they put strict requirements on the end-user application, they implement a limited set of built-in redundant execution polices.

Chameleon [WHIS98] is a software infrastructure, that standing between the last two solutions is aimed to integrate operating system features and to provide at system level high prerequisites for failure tolerant execution. It is the research response, like other experimental architectures [CUKI98], to commercial systems, which still do not fit to networks with a large number of nodes.

INTRODUCTION

Chameleon is an adaptive software infrastructure, which allows different levels of availability to be supported simultaneously in a networked environment; its leitmotiv is to protect a distributed application running over a network by the definition of additional fault tolerance polices, which cannot be covered by conventional hardware fault tolerance techniques.

This paper describes our proposal for an adaptive voting system, which integrates other recovery strategies still existing in Chameleon by providing a set of services aimed to collect and to match data produced by the ARMOR architecture or available externally from multiple replicas of a client application; our design involves three main areas, which strictly depend one each other, as we will explain in this report:

1 - Application Program Interface (API)

2 - Smart voting system design & implementation

3 - Chameleon Messsage Dispatching

The Chameleon interface API has been extended with the intending to allow an easy porting of existing applications written in ANSI C to assimilate the environment fault tolerant layer with slight patches to the original source code; this objective forced the creation of an interface following standard C library conventions and using no special C++ features or inheritance, but function wrappers compliant to ANSI C. These aspects are presented in the following section of this paper.

We observe that because this extension impacts only the interface API, it will not prevent at a same future to expand the Chameleon ARMOR architecture with a C++ interface library from which safety requirements could be inherited as multiple properties of basic building blocks; however, we feel that this is a long time goal, which will be worth as applications will be designed having Chameleon as target environment, i.e. making the application a resilient ARMOR of the environment. Actually, our main concern is to set a flexible design philosophy for porting existing sofware under Chameleon.

The central part of this paper covers the design issues of the smart voting system, which is composed by a set of ARMOR elements, whose responsibility is to assemble data from the periphery through the Fault Tolerance Manager (FTM) node, where stands the centralized voter ARMOR. The communication among the various objects, which make the voting subsystem is hierarchical and makes use of an asymmetric signature generation to minimize the exchanged data-flow, which is handled asynchronously with the outputs rising the application front-end.

Another important aspect described in the last section is related to Chameleon message dispatching system: we introduced the concept of dynamic messages leaving an addressing space available for their registration; such messages will exist for the duration of specific services and will allow a very fast and reliable reconfiguration of the environment releasing the overhead on some centralized managers.

1 - Chameleon Application Interface (API) In developing the program interface, we posed from the end-user developer perspective, who wants to modify a complex application designed with no reliability requirements to run it under Chameleon, in order to benefit of the safety services provided by the fault tolerance layer. Believing that the effectiveness of an Application Program Interface for such external environment, as Chameleon is intended to be at this early stage, stands in its simplicity, we put a great effort in designing an interface that fits transparently on the top of the original application concentrating all the extra information required by the fault tolerance layer in some global properties, which must appear at the very beginning of the application source code and could be declared in a global header file.

Our point to support this design philosophy is that, if a developer has to introduce so many patches to his or her own application, which may impact the consistency of the activation tree, a full re-design could be less error prone, than an aggressive reshaping.

For the sake of clarity, as we refer to "the application", we intend the whole set of processes and threads, which compose a distributed application specifically assuming that different fault tolerance strategies may be associated to distinct processing sets. The above assumption allows the dynamic reconfiguration of a distributed application by swamping down and restarting parts that cannot be removed at the same time. The latter requirement is typical in enviroments performing safety-critical tasks, where a remote unit, which usually drives a shuttle system, collects data to be subsequently validated at a control site [VOAS98a].

Under the above consideration, we designed a hierarchical voting architecture, that matches the relation process-thread and allows the dynamic reconfiguration by a distributed topology of centralized voting collectors, which perform group-voting operations [AGRA91].

In this section we present our extension to Chameleon interface API, which is composed of an external pre-compiled library linked to the target application and of some ARMOR elements, which act as front-end to the environment.

In the following figure 1.1 we introduce the objects that concur to realize, inside Chameleon, the client application front-end: we use the convention to represent *element objects* by square boxes and *ARMOR objects* by rectangles. An *element* is an object resilient in its ARMOR (Adaptive Reconfigurable and Mobible Objects for Reliability), which provides the execution thread, the message dispatching service and manages all the elements allocated in its name space. An istance is represented by a single line sketch, where a double line indicates a class object, which admits multiple instances of the same type. With the above conventions, the elements of figure 1.1 are dynamically allocated in the *armor_exec_t* name space, which is a class ARMOR, whose instance statically lives in the environment initialization code; the

dotted objects have been specifically developed to design the voting subsystem. The elements at the top realize the application front-end by subscribing to a set of messages dedicated to manage the data flow coming or going through the application interface: each stub implements a peer-to-peer connection with the corresponding function API designed for handling the specific flow; the stubs are instantiated under request of the managements units (*_mgmt*), which supervise the estabilished links and hand over the control to the specific stub under request of the application interface. Each element will be described in more detail at the end of this section.

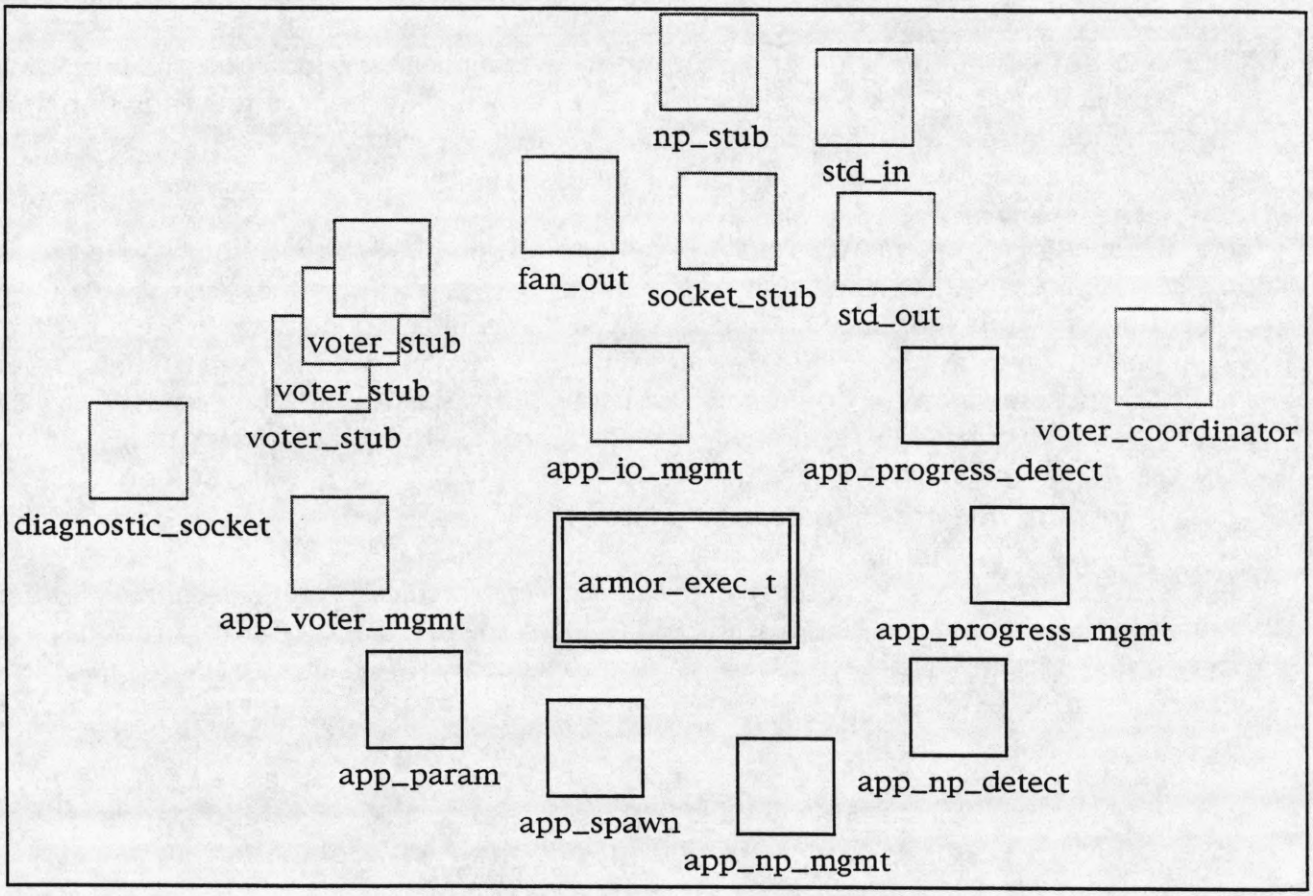

Figure 1.1: armor_exec_t population

We originally introduced the concept of *signed flow* to manage the data exchanged at various levels of the above interface.

If we look at the end-user application as a black box, we can identify a certain number of data flows coming from the external world inside the application by mean of standard pipes, operating system messages, etc. and a certain number of data crossing the application boundaries to reach the operating system.

A conventional operating system with no fault tolerant polices would treat all the incoming and outgoing flows as raw data, because there is no knowledge at that level of any internal representation and thus each request of service merely dispatches the information to the parties devices; this situation is depicted in figure 1.2 with the black arrows crossing the os boundaries.

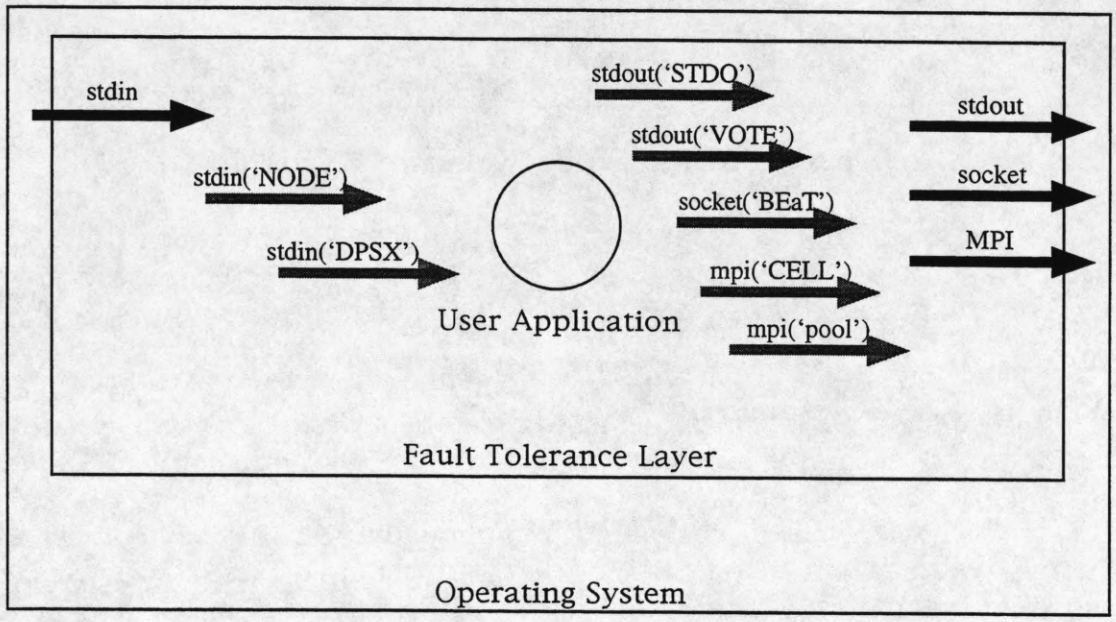

Figure 1.2: concept of flow; unsigned and signed flows.

On the other hand, when a fault tolerance layer intercepts the delivered information, to match it against different replicas of a voting topology, in order to assure the correcteness of the "driving" results, it could be very useful to select different types of data, which rise up from the application front-end: this is the purpose of the multiple stub elements introduced in figure 1.1.

Some flows may be subject to different voting polices, some other of exclusive pertinence of the operating system layer, some inter-process communication, some diagnostic flows, which we like to be forwarded to the *fault tolerance layer* to keep it informed of the "application health", the remaining ones just for stilistic purposes and carring no critical information: all this information is sometimes packed in a single transmission unit, expecially when the original application has been designed with no fault tolerance strategies in mind.

To properly dispatch the information supplied from the client application, our interface provides an *adaptation layer* and the mechanism of *signed flows* to deliver typified information across the fault tolerant layer boundaries; it is responsibility of the application developer to instruct the environment of the different kind of data to be exchanged with the external world: the association is performed via a

declaration block, which has to be called at the very beginning of the application and is compliant to ANSI C (figure 1.3).

```
// This handler initializes the io interface and has to be called at the very beginning
// of the application

chm_define_interface(chmio, stdin, stdout, stdout('VOTE'), socket('vot2'), \
                          vgrep('stuf', "(.*)(\t\t*)(.*)","\2",'vot2'));
```

Figure 1.3: an example of interface definition

The above statement initializes the mandatory fault tolerance control flow (*chmio*) and defines three types of exchanged data: a standard stream pipe, a socket stream and a *virtual flow*; each flow is "signed" to identify it from other flows of the same type.

The introduced syntaxt resembles the standard C files descriptors with the semantic extension to mark each *flow* by a four *char* signature; example: *stdout('SiGN')*.

Note that, while this syntaxt is actually resembling the C files conventions, a *flow* descriptor is valid only in the context of the Chameleon API and is not replaceable outside these bounds, because it uses a fully different mechanism to expand the namespace [KOEN89].

The ability to split uniquely defined flows in different signed types is accomplished by the way of *virtual flows*, barely a set of filter functions, which perform a re-routing to a list of pre-defined ending points on the basis of unix regular expressions. While different kind of virtual flows may be supplied as part of standard Chameleon API, the application developer has still the ability to write his/her own interface wrappers and to link them to the interface library.

The abstraction of *virtual flows* has been introduced to allow an easily port under Chameleon of existing applications, that may require various voting topologies: a*virtual flow* abstracts critical information, even if originally delivered over a single channel and splits it to different receiving parties (namely *stubs*), which can be dynamically configurated.

The folloving listing (figure 1.4) is an example of how an application can benefit of such interface layer.

```
// Example of a telecom application: we would like to select
// the destination field for delivering it to a voting flow

fprintf(stream, "DEST: %d SOURCE: %d PAYLOAD: %s CRC: %d ACCOUNTING: %d",\
            dst,src,data,crc(data),acct);

// Could be rearranged in:
// At the very beginning:
chm_define_interface(chmio, stdin, stdout, stream('VOTE'), stream('PASS'), \
                    vgrep('stuf', "(.*)(\t\t*)(.*)","\1",'VOTE', "*", 'PASS'));

........

// End replacing each occorrence of the above fprintf with:

chm_fprintf(vgrep('stuff'),"DEST: %d SOURCE: %d PAYLOAD: %s" \
            CRC: %d ACCOUNTING: %d",dst,src,data,crc(data),acct);
```

Figure 1.4: a telco interface

Please, note that where the example above refers to a simple stream flow realized by the way of the library function *printf*, the concept of *virtual flow* may be used to implement more complex substitutions: in a very complex protocol, the virtual "wrapper" could be a filter written in the Abstract Syntax Notation (ASN.1) [CASN1] to address the problem of extracting critical information from existing protocols. Moreover, *virtual flows* may be used to act as *software wrappers* [VOAS98b], in order to reduce the I/O sets to an external program.

While it can be a concern, if the filtering process has to run in the application thread or as a separate process on the fault tolerant layer, actually we want to emphasize that the *signed flows* give the ability to summarize in a unique point different behaviors related to the information delivered or received at the application front end.

Another leitmotiv, which has driven our approach, is the consideration that the application interface is a way to provide extra synchronization points to the end-user application, because all the communication to and from the fault tolerance layer should be handled by blocking calls: a developer may submit the execution of the application threads to the results of a certain voting policy; this aspect will be discussed further in the following section 2.

As complement to the io interface file, we plan to use a preprocessor program to automatize the port of existing code, as the supplied interface will be enough mature and proofed to be effective: this tool will receive in input the existing unpatched source code and a safety specifications file, which will address the requirements for

the fault tolerance layer and it will produce an ANSI C program suitable for recompilation and for running under Chameleon. This scenario is sketched in figure 1.5.

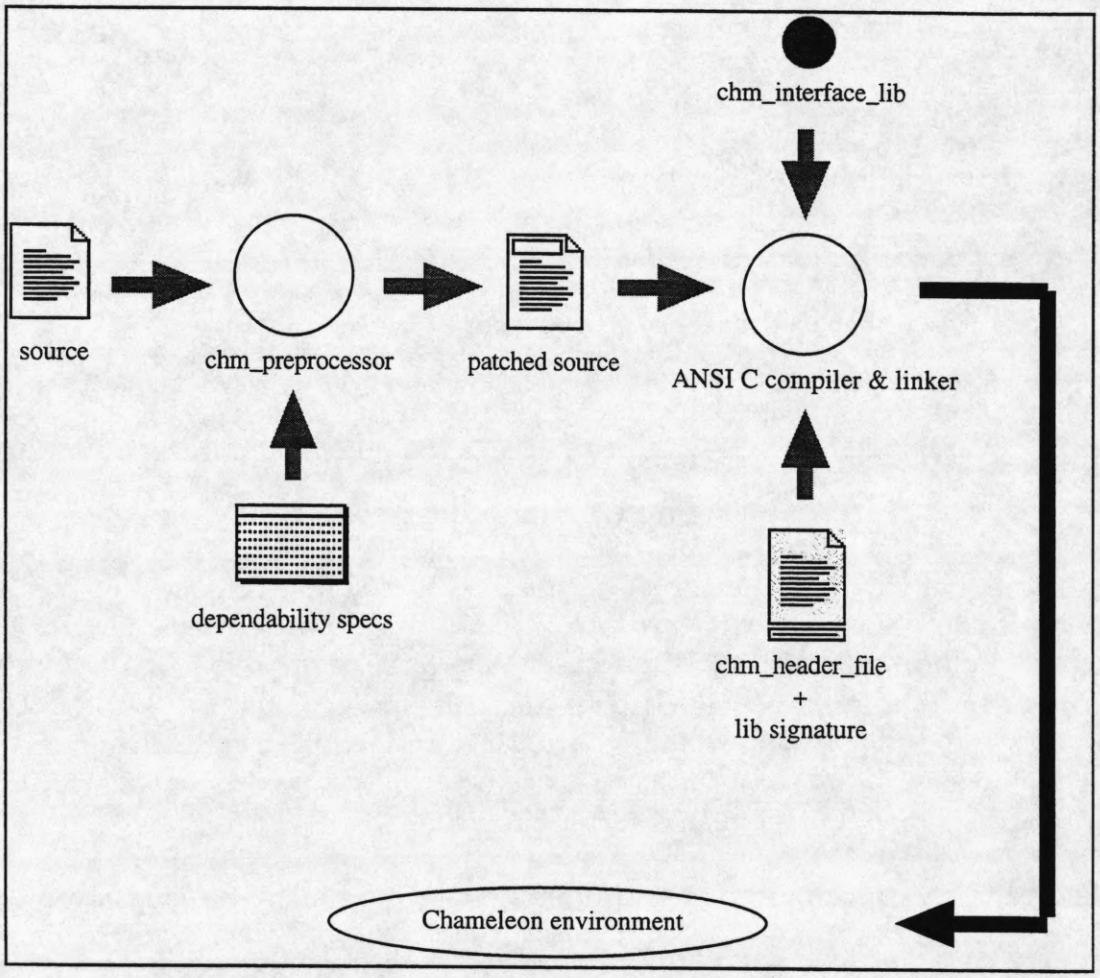

Figure 1.5: the Chameleon porting process

We strongly believe that this ending goal can be achieved effortlessly, because we have designed the io interface as much simple, semantically speaking and close to the standard C interface, so a one-pass lexical analyser shall fulfill the job.

At the actual design stage the application delivers its flows by the way of proprietary functions, which resemble the standard file operators, or the standard notation used in other communication handlers: this behavior is realized via a set of macro dispatchers, which maps standard C library functions to preassigned flows and hydes at the same time any other flow not specifically declared in the initialization section. The library interface will be provided with a public signature to assure the compatibility by matching it with the environment core object code.

The listing below illustrates the replacement rules that act at preprocessing time:

```
// Examples of std flow function replacements

// ORIGINAL SOURCE

printf("Hello World");
sprintf(fileout, "SOME STUFF");

// REPLACEMENT RULES

char stuff[4] = 'STUF';
char vote[4] = 'mark';

chm_fprintf(stdout('FLOW'), "Hello Flow");
chm_fprintf(stdout(stuf), "Hello stuff   SELECT marker");
chm_fprintf(vgrep(stuf, "MARK EXPRESSION", vote);
```

Figure 1.6: example of function replacements

for every virtual flow we have defined in the above interface, there is a specific built-in element in Chameleon, that is responsible for the delivery of information and synchronization with other objects: each interface function uses a named pipe to communicate with the ARMORs and from there the proprietary message pump.

The io elements are part of a dynamic load library, which is handled by a local manager (*app_io_mgmt*), whose responsibility after the initial allocation is to supervise the behavior of the io subsystem; it also de-allocates unused elements or dynamically reconfigurates the system, if required by the selected fault tolerance policy.

To efficiently perform the allocation and subsequent addressing of the interface stubs, that are instantiated at run-time, we have extended the original message pump allowing the registration of *dynamic messages*; this extension is explained briefly in the ending section of the present paper.

The following list is a description of the various elements internal to the ARMOR architecture, which compose the io interface: for each element is given a brief description of the functionalities along with the message subscription list.

app_io_mgmt : this element instantiated in the *armor_exec_t* name space is responsible for the initialization and supervision of Chameleon-application io subsystem. Because there is an interface stub for each registered application flow, the purpose of this manager is to allocate and properly connect the various stubs to the application interface; such "connection" is peformed via a dynamic message path: as well as this object receives a MSG_REGISTER_FLOW_(TYPE) message, it allocates

the stub matching the request type, then it contracts a new *dynamic message* for the instantiated element. The new message is registered as *private dynamic* and notified to the interface function for subsequent delivery of the defined flow.

The *app_io_mgmt* element subscribes to the following message types:

• MSG_APP_IO_INIT	• MSG_CREATE_LOCK
• MSG_APP_SIGNATURE	• MSG_REGISTER_FLOW_STDIN
• MSG_REGISTER_IO_GROUP	• MSG_REGISTER_FLOW_STDOUT
• MSG_UNREGISTER_IO_GROUP	• MSG_REGISTER_FLOW_PIPE
• MSG_ENABLE_FLOW	• MSG_REGISTER_FLOW_SOCKET
• MSG_DISABLE_FLOW	• MSG_REGISTER_FLOW_MPI
• MSG_SUBMIT_FLOW	• MSG_REGISTER_VIRTUAL_FLOW
• MSG_UNSUBMIT_FLOW	• MSG_REGISTER_USER_FLOW
• MSG_FORWARD_FLOW	

Figure 1.7: *app_io_mgmt* message subscription list

std_in_stub : as allocated, this element registers a dynamic message (*DYNA_RECEIVE_FLOW*) to communicate with its end party, which handles the *chm_stdin* flow; it subscribes also to the following messages:

• DYNA_RECEIVE_FLOW	• (MSG_LOCK)
• MSG_CREATE_LOCK	• MSG_SIGNAL_RECEIVE

Figure 1.8: *std_in_stub* message subscription list

note that the *MSG_LOCK* message is subscribed conditionally, if a *MSG_CREATE_LOCK* is received.

std_out_stub : element responsible for delivering the standard output flow, which is wrapped by the interface function *chm_stdout*(); it subscribes to messages:

• DYNA_TRANSMIT_FLOW	• (MSG_LOCK)
• MSG_CREATE_LOCK	• MSG_SIGNAL_RECEIVE

Figure 1.9: *std_out_stub* message subscription list

The dynamic message *DYNA_TRANSMIT_FLOW* is defined as the element is allocated.

np_stub : creates a named pipe to communicate with the end-user application; the named pipe flow is reached by the *chm_fprintf(chm_np(),...)* wrapper. The subscribed message list is:

• DYNA_TRANSMIT_FLOW	• MSG_CREATE_LOCK
• MSG_CONNECT_FLOW	• (MSG_LOCK)
• MSG_DISCONNECT_FLOW	• MSG_SIGNAL_RECEIVE

Figure 1.10: *np_stub* message subscription list

where *DYNA_XXX* are dynamic messages contracted as the element is allocated.

socket_stub : intercepts a socket data flow and forwards it to the final destination; the subscribed message list is:

• DYNA_TRANSMIT_FLOW	• MSG_BIND_FLOW
• DYNA_RECEIVE_FLOW	• MSG_CREATE_LOCK
• MSG_CONNECT_FLOW	• (MSG_LOCK)
• MSG_DISCONNECT_FLOW	• MSG_SIGNAL_RECEIVE

Figure 1.11: *socket_stub* message subscription list

mpi_stub : intercepts any mpi call and delivers it to the appropriate group; the subscribed message list is:

• DYNA_TRANSMIT_FLOW	• MSG_BIND_FLOW
• DYNA_RECEIVE_FLOW	• MSG_CREATE_LOCK
• MSG_CONNECT_FLOW	• (MSG_LOCK)
• MSG_DISCONNECT_FLOW	• MSG_SIGNAL_RECEIVE
TO BE COMPLETED	

Figure 1.12: *mpi_stub* message subscription list

vgrep_stub : this is an example of virtual flow: actually it relies on the unix grep command to perform the intended task; soon will be delivered a fully internal version of the stub. Additional virtual flows will be implemented later.
The *vgrep_stub* subscribes to the following messages:

• DYNA_TRANSMIT_FLOW	• MSG_DEFINE_VFLOW
• DYNA_RECEIVE_FLOW	• MSG_NEGATE_VFLOW
• MSG_CONNECT_FLOW	• MSG_TEST_VFLOW

Figure 1.13: *vgrep_stub* message subscription list

2 - Smart Voting Topology In this section we present the smart voting system, which poses on the previously described interface; we designed it having in mind three main concerns [Xu, 1998 #236]:

- dynamic reconfigurability, i.e. the ability to change the voting topology during the application life, for example switching from a TMR execution to a DUP mode; such reconfigurability, also requires that different application processes can be managed separately, i.e. a recovery of an application process does not necessary involve the restart of the remaining parts, which appear to work properly.

- total reusability of the voting elements to provide a redundant execution of the ARMORS objects inside Chameleon: each ARMOR may be instantiated as *logical* redundant object; the logical ARMOR still resembles the behavior of the basic building armor, but it provides extra reliability, because each output message is the result of a matched voting agreement among different replicas of the same base object (figure 2.1);

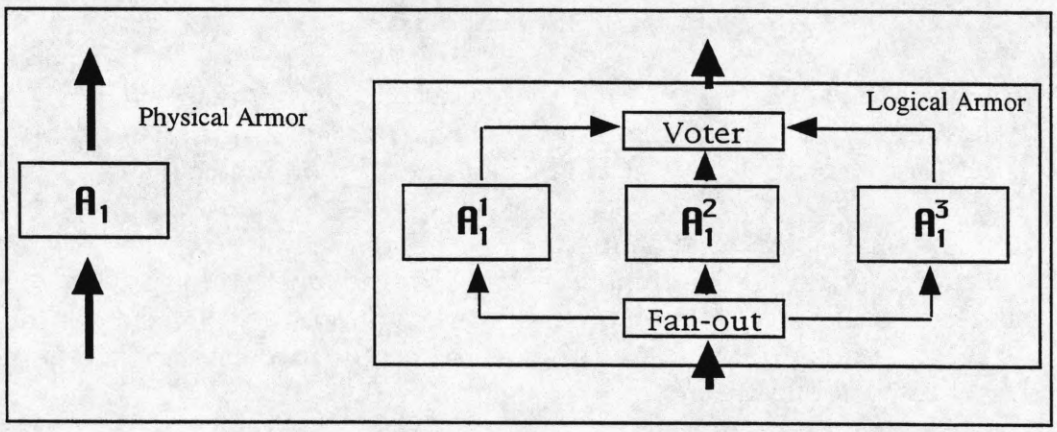

Figure 2.1: physical vs logical armor

- manageability at the application front-end: more than a requirement, this is an assumption, which states the total responsibility for instructing Chameleon of a requested voting policy at total charge of the end user developer. We strongly believe that while the fault tolerance layer is the supplier of services, that could make an application dependable over a computer network, the end-user developer has to set the timing requirements and to choose the right policy (among a library supplied by the environment), that best fits his/her own application.

The configuration of the fault tolerant layer is accomplished via the io interface file introduced in the previous section with a semantic extension, the *safety block*, which allows the client application to contract the required fault tolerance policy: the configuration statements take place at the very beginning of the application in a

mandatory sequence of ANSI C functions, we called *Chameleon Interface Block* and *Chameleon Safety Block*. The introduction of such interface wrapper will be performed automatically in some future: as we explained in figure 1.5, a one pass compiler will parse the application source code and a specification header (consisting of the above two blocks) and it will return a patched code, suitable for recompilation under an ANSI compliant compiler.

To clarify how a general application written with no fault tolerance polices in mind could be patched to run under Chameleon with a preassigned voting topology, we will refer to a pseudo-code (figure 2.2) that performs some calculations and gives some output results, which we will consider our *driving results*, i.e. the information to be validated before delivering it to the intended destination.

For our example, we assume to request:

- a TMR validation on all *driving results*;
- an intermediate checking point (TMR) on internal state data;
- a synchronization point, where the application thread waits for the delivery of some remotely validated information, before performing any local voting operation on internal sets.

Please, note that the last requirement is a very handy way to lock the execution of a thread waiting for delivery of remote data produced and validated by some other actor, thing necessary, if no previous synchronization scheme was built in the application; this is a common assumption, if the application has not been specifically built for a redundant execution.

```
// Voting sample
// This pseudo code show the interaction points of a generic appl and how
// the patching is performed to run it under chameleon with a voting scheme.

main(int arc, char  *argv[])
{

    printf("Hello APPL starts here \n");

   // Appl initialization stuff

   .......

   // Read input parameters

   for (i = 1; i < argc; i++)
    {
         // Get arg i
```

```
        char * x = argv[i];
        // Perform calculations
        ........
        //Output intermediate result
        printf("Status of elaboration cycle %d reading %f\n", i, status);
        .......
        //Contact another party
        putc(msg, socket_file);
        ........
        // Collect result from a remote party
       getc(result, socket_in);
       // Other processing
       ..........
       // Add local result to a file
       fprintf(fileout, result);
   } // END loop
}
```

Figure 2.2: a generic program eligible for voting

The diagram in figure 2.3 translates the above listing in terms of control flow among the io interface points: from such activation graph, we can argument that a generic application performs the io operations at scattered intervals, which are function of hardware performance and input data; some io calls may result in a blocking thread, some other not, depending on the nature of io operation and of the underlying operating system software / hardware architectures.

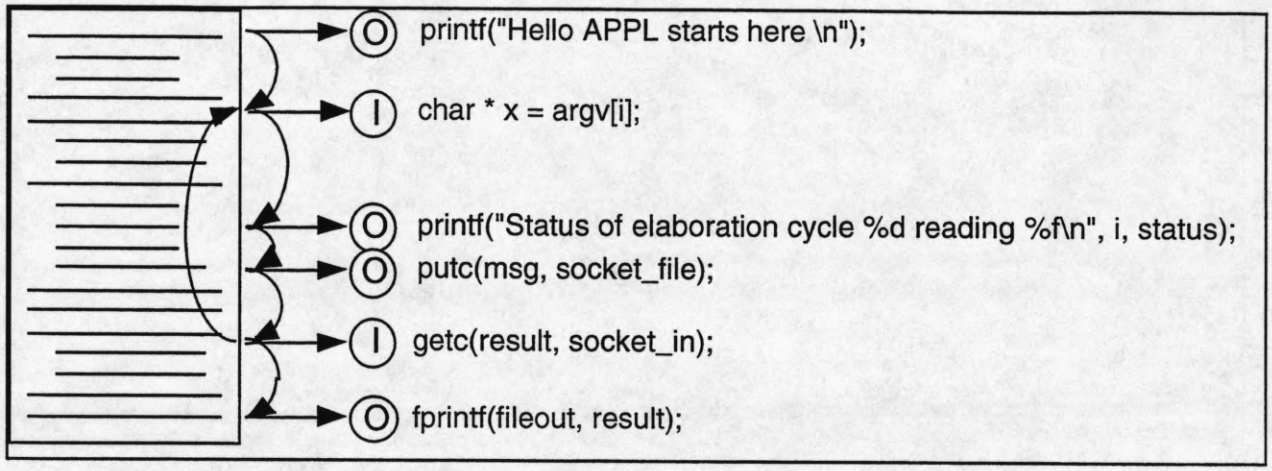

Figure 2.3: original source code control flow

Before the introduction of the voting layer, Chameleon was used to replicate the program execution simply by issuing the same control flow multiple times: this raw control scheme cannot be used in a voting redundant execution, because data coming from different instances of the same execution thread need to reach the voting periphery at the same time. In order to provide the extra synchronization points required by a voting thread and to keep the interface simple, we have choosen to implement the semaphores inside the interface wrappers: in this way, the original user source code gains a certain number of synchronization placeholders uniformely distributed and sematically joined to the voting flows.

In the *safety block* interface, the end-user developer instructs the environment of which flows are critical and which ones require special handling attributes to perform synchronization: Chameleon configures the interface stubs for a flow with the *critical* qualifier to collect the data associated with that flow across different replicas taking care of the necessary synchronization; an additional "submit" attribute may lock the execution every time the flow is referenced waiting for the synchronization flows specified in the submit list.

This methodology gives the ability to perform very complex synchronization schemes, without strongly impacting the original application source code.

An example of the *safety block* interface is given in the following listing (figure 2.4), which is the result of the above extensions to the previously presented pseudo-code.

```
// Voting sample
// This pseudo code show the interaction points of a generic appls and how
// the patching is performed to run it under chameleon
// HERE PATCHING IS DONE SUPPOSING WE REQUEST A TMR EXECUTION
```

```
main(int arc, char  *argv[]
{

    // BEGIN of Chameleon INTERFACE

    chm_define_interface(chmio, stdin, sdtout, stdout('VOTE'), \
                         socket('rmin'), socket('rout'), file, voter);

    // BEGIN of Chameleon SAFETY BLOCK

    // Instruct the environment of which redundacy is required for
    // each appl process; a missing specification will result in a
    // single application run.

    chm_run_process(main, TMR);

    // [optional]: one per each thread/process supervised:

    chm_hearth_beat_rate("15 sec", main);

    // Specify which flows has to been marked as "critical", i.e.
    // a voter stub is allocated for such critical flows

    chm_define_critical(chmio, stdout('VOTE'), socket('rmin'),
                        socket('rmout'), file, voter);

    // Voting is performed on the basis of the source flow
    // redundancy, if not otherwise specified; as a voting
    // operation occurs, the flow will be cleared to reach
    // the intending destination, if no submitions has been
    // previously registered in the following section.
    // A voter flow performs by default a voting operation
    // with the above policy and synchronizes the execution
    // as the voter has reached an agreement. The synchro-
    // nization happens on a total agreement, if not other-
    // wise specified. Partial agreement require a time-out
    // parameter to specify when a partial agreement can be
    // taken in account, if no total agreement is reached.
    // Example:
    //           chm_define_policy(voter, "2/3"); // Unlocks as soon as
    //                                            // 2 flows agree.
    //           chm_define_policy(voter, "3/3"); // error if not matchs
    //                                            // execution.
    //           chm_define_policy(voter, "5");   // 5/(flow redundancy)
    //
    //           chm_define_policy(voter, "2/3", "5 sec");
    //           // The above statement means: 2/3 agreement unlocks
    //           // as well as timeout is reached, or as soon as a
    //           // a total agreement is reached. Timeout units may
    //           // be expressed in terms of absolute time, or relative
    //           // to samples times (collected from the voter).
```

```
chm_submit_critical("2/2 BEFORE", socket('rout'), \
                    voter, stdout('VOTE'));

// Flow socket('rout') is submitted to the successful agreement
// (based on respective registered agreement policy) to flows
// voter and stdout('VOTE'). The flow locks BEFORE
// performing any operation on the "submitted" flow, until an
// agreement on both the flows "2/2" is reached.

// This function introduced here, just for example, is very
// useful to synchronize different application processes or threads.

// One or more per critical flow:
// format string syntaxt: "check-rate UNIT [time-out(min, MAX) UNIT]"
// suffix "ALL" marks every-flow with the same timing specs; any
// further declaration will produce a run-time error.

chm_check_rate("15 sec", ALL);

// END of Chameleon SAFETY BLOCK

if !(chm_app_signature('TEST'))
{
// END of Chameleon INTERFACE


chm_printf("Hello APPL starts here \n");

// Appl initialization stuff
.......
// Read input parameters
for (i = 1; i < chm_argc; i++)
 {
      // Get arg i
      char * x = chm_argv[i];

     // A bare voting flow: if an agreement is reached it goes on

     chm_fprintf(voter, x);

     // Perform calculations
     ........

     //Output intermediate result
     chm_fprintf(stdout('VOTE'), \
            "Status of elaboration cycle %d reading %f\n", i, status);

     // The programs waits on the above statement until all the replicas
     // have reached an agreement, beacuse stdout('VOTE') has been
     // marked as "critical flow".
```

```
        .......

        //Contact another party
        chm_putc(msg, socket_file);

        // if socket_file is a critical flow, it waits until agreement

        ........

        // Collect result from end party

       chm_getc(result, socket_in);

       // Waits before calling chm_getc until both the submitted flow
       // have reached an agreement. Actually they are part of the same
       // execution thread, so agreement is achieved, if the control
       // flow reaches this point. The function performs the same
       // voting operation, if the flow has been registered as critical.

       // Other processing

       ..........

       // Add local result to a file

       chm_fprintf(fileout, result);

   } // END loop

   } // Chamemeon Safe Execution block if !(chm_app_signature())
}
```

Figure 2.4: the example pseudocode patched to run under Chameleon

In the beginning of the main application process all the io flow are declared, even if not dealing with the voting subsystem; the *safety block*, which follows the interface configuration section, contains hadlers to instruct the environment of the fault tolerance requirements for the reliable execution of end-user code.

The first call specifies how many replicas to run for each application process: the function *chm_run_process()* accepts a function pointer and a policy label; in this way the application developer may request different redundancies for different application processes or threads. The execution redundancy also affects the default voting redundacy, if no other behavior is specified.

The *hearth_beat_rate()* takes a format parameter, which can have different tags for specifing an hearth beat rate, i.e. a timed check performed on the function code given as second parameter.

All the flows intended for voting, must be marked as *critical*: for this purpose, the function *chm_define_critical()* uses the same syntax of *chm_define_interface();* in order to be marked as "critical" a flow has to be previously registered via the *chm_define_interface()* with the only exception of a *voter* flow, which by default ends into a voting device.

The *chm_submit_critical()* wrapper connects a *critical flow* to any previously defined flow, in a way that stops the thread execution until the submitted flows become available; a *critical flow* becomes "available" as well as a voter has reached an agreement (partial or total) upon it. The first parameter of *chm_submit_critical()* interface is a format string, which gives the ability to specify the synchronization policy, i.e. if a flow is *preposted* or *deferred* to the submitted list; another optional parameter is the number of flows, which unlock the semaphore, as they result available: a total agreement is the default. Using this function a developer may realize very complex voting and synchronization polices just slightly modifing an existing application.

If no *chm_check_rate*() is called on the registered flows, the default check-rate will be automatically defined at run-time by Chameleon, with a safety margin to prevent a too fitted grid; the first parameter is a format string, other acceptable parameters are the registered flows, or the label ALL, which applies the issued timings to all the previously registered flows. The check-rate parameter is explained in detail later: it represents the amout of time that occurs between two subsequent inquiries from the central voting collector.

In our example, function *chm_app_signature()* closes the interface block: this function will delivery a signature to the voting subsystem; the central voting collector employs such signature to validate data samples delivered from the voting periphery. To generate the signatures we can define a one-way function, for example a polinomial used in Cyclic Redundancy Check (CRC), which can be effortlessly inplemented in hardware and will be subject of a further report.

All the interface functions have strict requirements upon the accetable parameter list and previous mandatory calls: each time an interface function is called, it checks for the consistency of the whole interface context and if something is missing, or wrong it returns a non null value. This diagnostic feature gives the ability to perform a check on the last interface call to prevent the execution of the application, if something went wrong in the interface definition sequence.

Table 2.5 summarizes the requirements for every interface function.

Interface function	Mandatory	Defaults	Parameters	Requirements
chm_define_interface	yes		chmio mandatory, then v.a.	none
chm_run_process	no if single execution		thread ptr, policy	prev defined function thread
chm_hearth_beat_rate			format string, thread ptr	prev defined function thread
chm_define_critical	for voting		chmio mandatory, then v.a.	can reference ony prev registered flows and voter flows.
chm_define_policy			flow, format string, opt format string	only prev registered critical flows
chm_submit_critical			format string, flow, v.a.	first flow critical, other registered flows
chm_check_rate			format string, flow, ALL	registered flows
chm_app_signature	yes for voting		signature string	

Table 2.5: Chameleon Interface requirements

To synchronize all the independent voting threads with the user-application under the specified fault tolerance policy, we have chosen to collect the application io flows at specific time intervals; such sample interval is passed to Chameleon in the *safety block* interface, via the *chm_check_rate*() function call.

Figure 2.6 shows the interaction path between the end-user application and the fault tolerance environment: the control flow on the end-user side is subject to some synchronization points, which match the io interface placeholders for the registered voting flows.

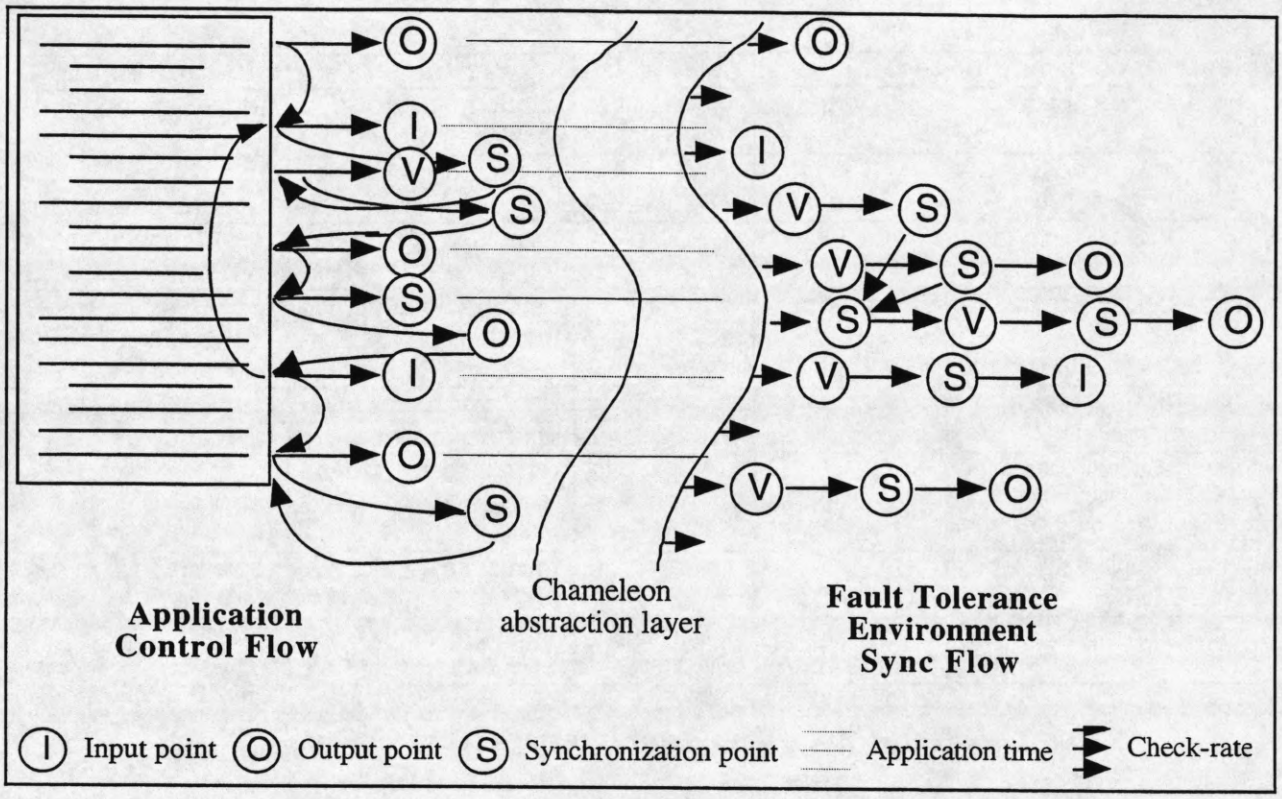

Figure 2.6: IO and control flow

The *abstraction layer* supplies a continous flow to the interface stubs, as well as the voter stubs: as a voting operation is performed, the output drives the synchronization lock and goes through the output channel (if any has been registered as end party of that flow). The real synchronization is performed at specific sample intervals on the Chameleon io interface layer, which is the central collector of all the incoming flows (synchronization and dispatching). The synchronization intervals are under control of the user application, which instructs the environment of the bounds and of slacks in which each window can grow or stretch.

Such synchronization scheme came us in mind to prevent a deadlock condition and by argumenting that if some data cannot be collected upon a certain time for subsequent voting, we would like to reach a partial agreement or a disagreament, if there are no enough data for voting, which would trigger some recovery operation; this aspects is peculiar of software fault tolerance, which in case of an exception need to perform some recovery tasks asynchroneously, where an hardware solution usually would raise an error signal to bring the system in a fail-safe condition.

With the above intending, the fault tolerance environment is under total control of the application interface and it is developer's responsability to issue the dependability requirements to the platform: he or she really knows, which are the critical timings and when is best to give-up, then wait indefinitively.

In the following figure 2.7 we introduce the smart voting topology outlining the exchanged data flow between the client application and the Chamelon architecture: the example refers to a choosen TMR topology, where three replicas of the same application process are matched to produce a voted *driving output* at the environment front-end; although the diagram refers to a specific fault tolerance setting, it can be easily extended to different voting polices, mixted as well. Each arrow represents a vector data flow produced at a certain end and delivered to the corresponding party; in figure we distinguish the following kind of objects:

Physical elements, appearing as square boxes represent a single object instance and make integrant part of the voting system.

Logical elements are single functional units, however their functionalities result from a collection of objects, which handle separately different messages, upon a shared architecture; logical elements are marked as a continuos circle.

A *group* is a compount of many elements (logical and physical) intended to fulfill a specific job; as groups, we have the *io subsystem*, which has been explained in the first section of this report and the *voter subsystem*, whose composition will be given later.

The same conventions introduced for the elements apply to the groups: a *physical group*, marked as a continous line, lives on a sigle node of the chameleon architecture.

A *virtual group* collects similar functionalities and gives the abstraction of the globally exchanged flows.

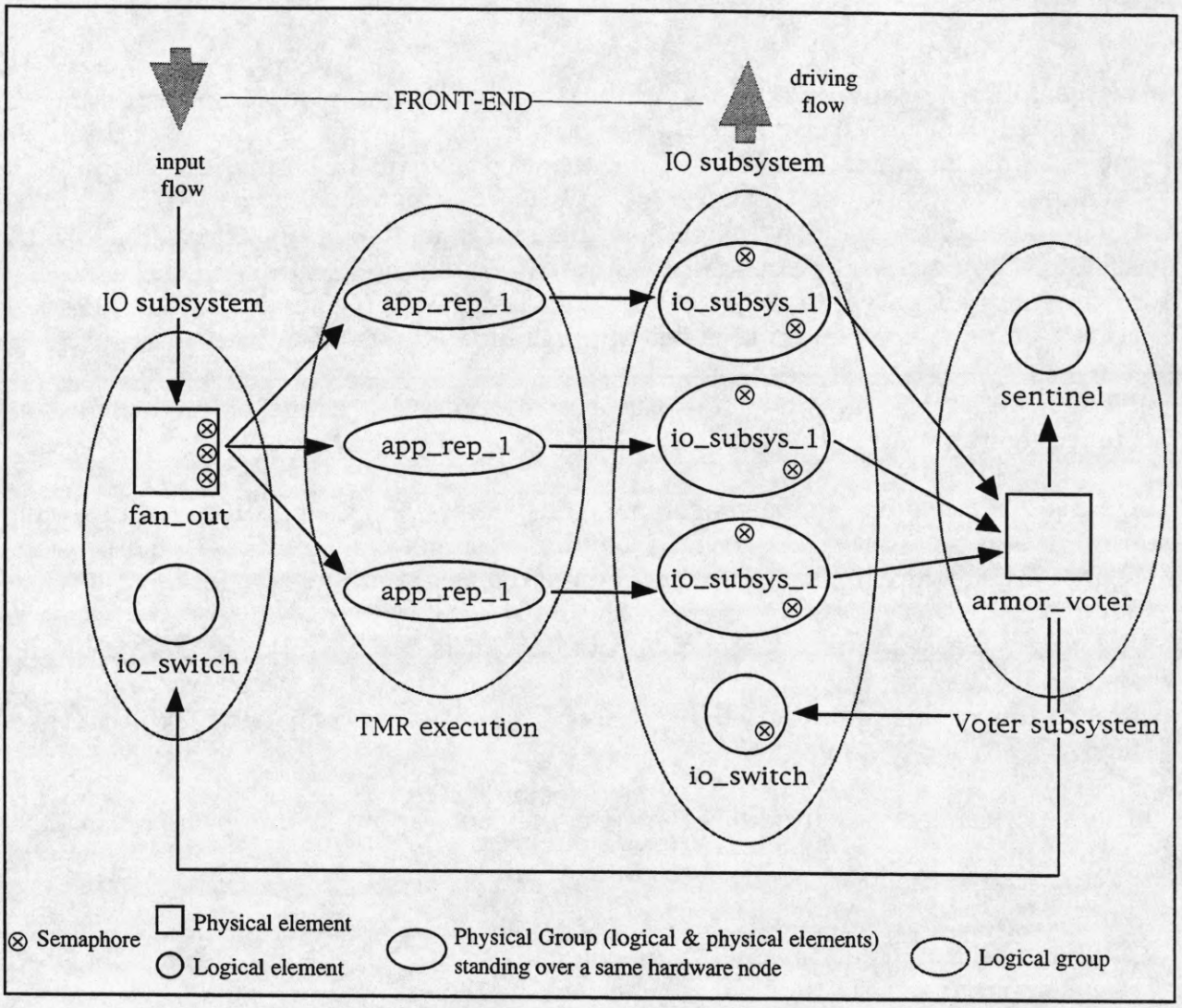

Figure 2.7 Chameleon voter control flows

To understand the role of the various objects, which concur to realize the voting topology presented here, we can follow the data flow of figure 2.7 from the top left to the top right.

As the input data originating at the client application reaches the Chameleon front-end, it encounters first the *IO subsystem*; where it is dispatched to the object *fan_out*, whose responsibility is to provide the different application replicas with the same driving input; the *fan_out* element is equipped with a blocking mechanism, which prevents the delivery of further data, if the subsequent blocks of the voting chain have not performed the evaluation of previous data chunck.

The *io_switch* appearing in the *IO subsystem* group has the purpose to adapt the driving flow to the required voting parallelism, according to the commands received

from the central voter ARMOR (right). We introduced this closed control loop, because the number of voting parties may change during the application life, due to a specific request to switch the redundancy level, or as conseguence of a permanent failure in the voting chain. Such failure may result from an hardware problem at the application node, where the program is actually running, if no spare nodes are available for resuming the execution of the missing replica; alternatively, a permanent failure may occur, when different versions of the same application running in multiple mode bring to mismatching results, because of an error in the control flow of one of them.

The output of the application replicas passes through the *IO subsystem* (centre), which acts, as explained in the first section of this paper, as intermediate collector of voting data: the flows marked*critical* are separated from other registered data flow and redirected to the voting subsystem. There are to distinct data paths, one for the external output, namely *drive data* , one for the*voting data.*; both use semaphores to assure the proper timing to the end parties.

As the application replicas do not drive straightly the operating system devices, but the output goes through the io_subsystem, which acts as intermediate collector of voting data. and *voting data* are delivered through different channels by the *io_subsystem*.
The paths

Voting data are delivered to the voter ARMOR, as well as it has fullfilled the previous voting cycle and as the incoming channels are marked "active" (an incoming flow may become "unavailable", if the associated replica has signaled REPLICA_MAX_FAILURES times a voting mismatch under a TMR or more redundant topology); this parameter prevents the rejection of a voting flow, due to temporany jamming on its associated channel.

The *driving output* is selected on the basis of the voting agreement polices and in any case after the voting cycle has been completed reaching an agreement, although partial; a partial agreement does not affect the driving output, but the abnormal situation is signaled to the *fault tolerance manager* (FTM). It is the *io_switch* in the (central) *IO subsystem*, that controls the delivery of driving outputs to the external environment.

The *voter subsystem* collects data asynchronously from the application replicas and synchronizes the entire environment: as a new voting cycle is fulfilled at the voter end, a message unlocks the io switches and the subsequent data chunk is delivered; after that, a signal notifies to the *sentinel* element the successful completion of the voting cycle. The *sentinel* element acts as a monitoring process of the voter health and implements some recovery polices: if an internal watchdog reaches zero, an inquiry to the FTM checks for a possible deadkock condition; upon a certain time

from the first signal another watchdog resets the voting system under the assumption that it hung for some reason.

In the above discussion we analyzed the data-flow of the voting subsystem; in the remaining part we will focus on the interaction among the different elements, which compose the voting subsystem, as shown in figure 2.8:

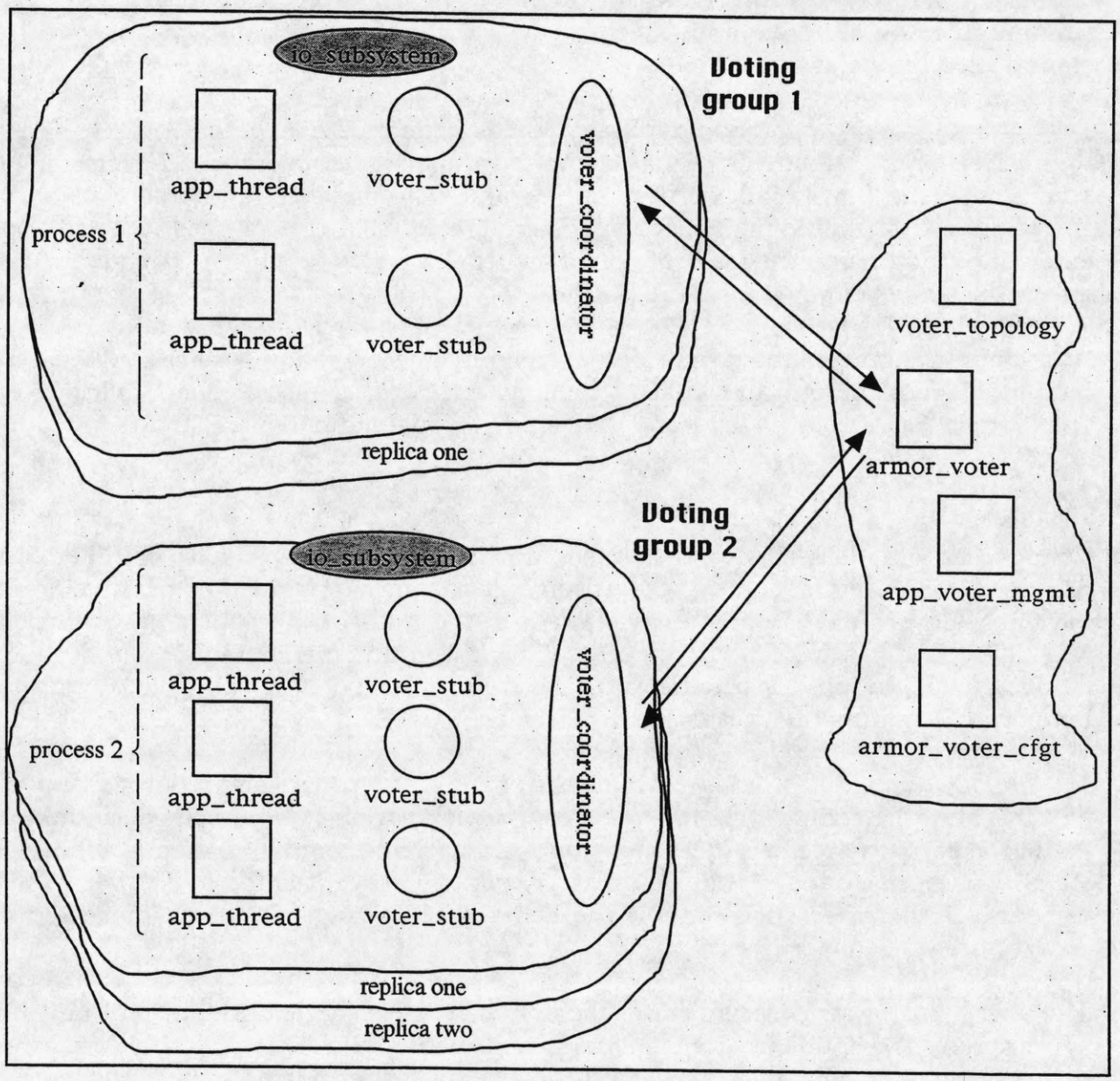

Figure 2.8: voter topology

The figure 2.8 refers to an application running over Chameleon composed of two processes: the first process shares two application threads and the second one three;

moreover, the first process has required a duplicated mode execution, while the second group uses a triple module redundancy (TMR). Please, observe that the creation of two separate voting groups is the conseguence of two different voting topologies requested for the application processes: whenever both processes should be subject to the same voting topology, they can coexist in the same voting group.

As the io interface stubs are responsible for capturing the application data flow and delivering it across Chameleon, the *voter_stub* elements collect the information marked as *critical* for subsequent dispatching: the flows are transferred to a *voter coordinator*, which is a local collector of the information coming asynchronously from the application interface; the coordinator acts as rate adapter between the incoming flow and the samples delivered at a fixed check-rate upon request from the central manager.

There is a *coordinator* per replica and different application threads may share the same *voter coordinator*. To provide the ability of handling separately different part of the application, we have introduced the concept of *voting group*: a *voting group* is a set of monitored user application threads, which can be subject to the same voting and recovery policy.

The example presented in figure 2.8 shows two process of the same user application: the first process owns two threads, which produce two *voting flows* registered to a coordinator as "group 1"; there are two coordinators associated to voting "group 1", because the first process instance is replicated in duplicated mode. The second process, which executes in TMR, has three threads and produces three *voting flows* (one per thread) registered as "group 2" to the local cordinator.

Please, note that the concept of *voting flow* is independent from the location of the source data: more execution threads of the same process may register different flows, just because the requirements in terms of redundacy are different.

The *voter coordinators* deliver the collected information to a central element, which periodically triggers this end parties for a new data sample.

Before delivering the voting information, the coordinator objects perform one important operation intended to keep small the voting overhead: they assemble the data collected from the *voter_stubs*, assign them an unique sequence number and generate on the whole data chunk a *digital signature*, which is delivered to the central voter in place of original data (figure 2.9). In this way, the voter element has to compare small signatures, instead of the complete data set from which come from. Different flows share the same signature, if belonging to the same voting group, because the delivered information triggers the same recovery operation, in case of a continous disagreement at the voter end.

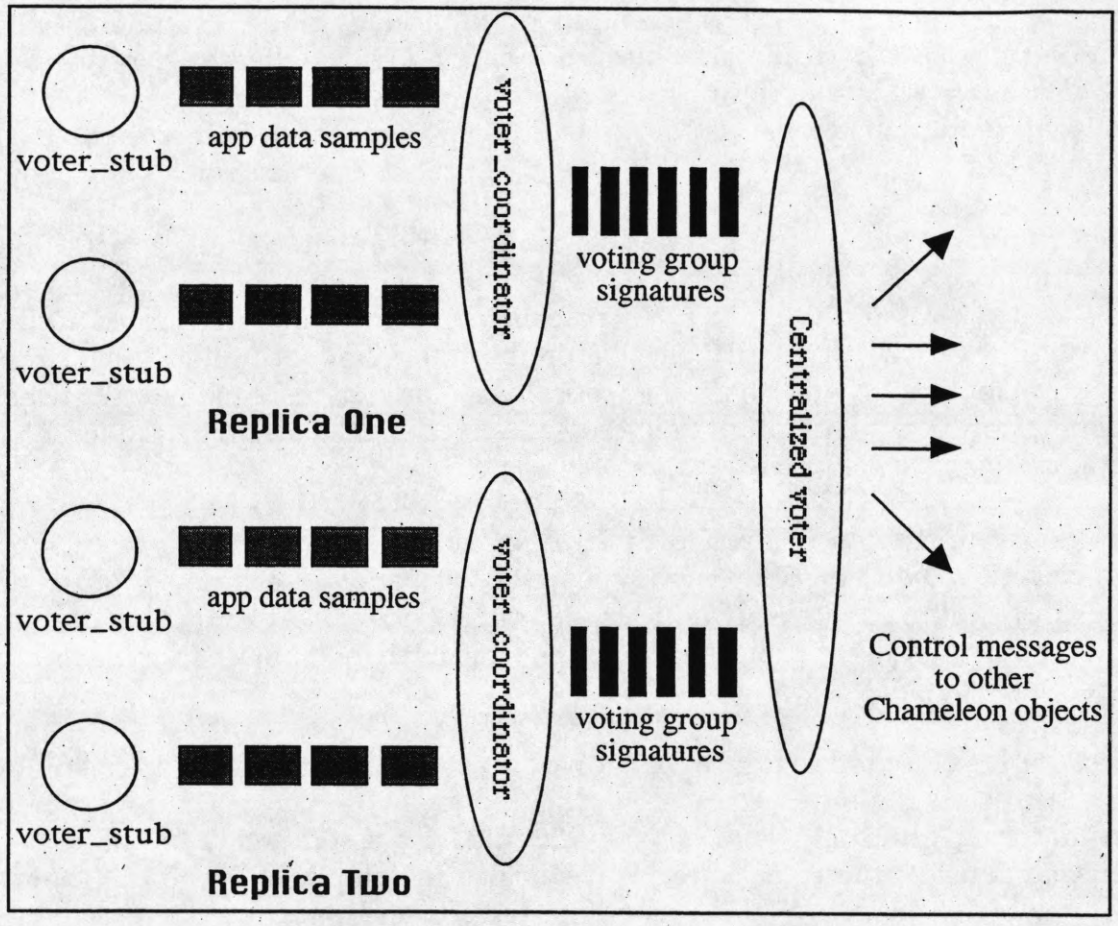

Figure 2.9: voting flow and signatures

Figure 2.9 explains the mechanism of signature generation referring to the first process of figure 2.8, which is replicated two times.

The voter executes its voting thread cycle, performs the required semaphore update depending on the voting policy, notifies the fault tolerance manager of partial agreements and requests the next voting sample from the periphery.

As for the io subsystem, we give the list of the voting elements and their message subscription list:

voter_stub : is the sofware probe, which collects data sent from the application and forwards it to the *voter_coordinator* object. Actually a stub simply aliases the messages supplied from an io subsystem stub and performs the job subscribing to the following messages:

• MSG_ADD_VOTER_ELEMENT	• DYNA_TRANSMIT_FLOW
• MSG_REMOVE_VOTER_ELEMENT	• DYNA_RECEIVE_FLOW

Figure 2.10: *voter_stub* message subscription list

fan_out : this element simply forwards a single input flow to multiple application io_stubs, in order to drive the various replicas of a redundant execution with the same input source; the element is coordinated by the central voter element, which sets and resets an internal semaphore to provide synchronization with the matched execution step. It subscribes to the following messages:

• MSG_REGISTER_VOTER_GROUP	• DYNA_TRANSMIT_FLOW
• MSG_UNREGISTER_VOTER_GROUP	• DYNA_RECEIVE_FLOW
• MSG_NOTIFY_AGREE	• (MSG_LOCK)
• MSG_NOTIFY_PARTIAL_AGREE	• MSG_SIGNAL_RECEIVE

Figure 2.11 : *fan_out* message subscription list

app_voter_mgmt : is the interface element, which receives the configuration messages from the interface API at the very beginning of user application; this element builds up the overall voting subsystem allocating the objects to perform the requested fault tolerance policy.

• MSG_REGISTER_VOTER_GROUP	• MSG_APP_SIGNATURE
• MSG_UNREGISTER_VOTER_GROUP	• MSG_CONNECT_VOTER_ELEMENT
• MSG_ADD_VOTER_ELEMENT	• MSG_DISCONNECT_VOTER_ELEMENT
• MSG_REMOVE_VOTER_ELEMENT	

Figure 2.12: *app_voter_mgmt* message subscription list

armor_voter_cfgt : this element is under developement and acts as counterpart of the *app_voter_mgmt* for any voting policy internal to the armor structure. It configures the elements required in a voting system internal to the ARMOR and instructs each element of the different behavior required in such context.

voter_coordinator : a coordinator acts as mediator between the interface stubs and the centralized voter; it adapts the application flow rate to the voting check-rate and

performs the important operation to assemble all the voting data delivered from the application stubs in a whole chunk of data, upon which a digital signature is generated. The coordinator delivers only the digital signature to the central voter, because it reflects the actual output of the whole voting group. Before sending the signature the coordinator inserts the time-stamp, which marks the sample in the continous flow supplied from the application front-end. This element subscribes to the following messages:

• MSG_REGISTER_VOTER_GROUP	• MSG_ENABLE_VOTER_GROUP
• MSG_UNREGISTER_VOTER_GROUP	• MSG_DISABLE_VOTER_GROUP
• MSG_ADD_VOTER_ELEMENT	• MSG_LOCK_VOTER_GROUP
• MSG_REMOVE_VOTER_ELEMENT	• MSG_UNLOCK_VOTER_GROUP
• MSG_CONNECT_VOTER_ELEMENT	• MSG_GET_SAMPLE
• MSG_DISCONNECT_VOTER_ELEMENT	• MSG_GET_NEXT_SAMPLE

Figure 2.13 : *voter_coordinator* message subscription list

armor_voter : is the central object, where actually voting is performed. This element handles the voting, coordinates the incoming message queues coming from the voting coordinators and broadcasts messages to inform the party elements of the result upon complention of voting operation. Here is its message subscription list:

• MSG_ENABLE_VOTER	• MSG_NOTIFY_AGREE
• MSG_DISABLE_VOTER	• MSG_NOTIFY_PARTIAL_AGREE
• MSG_ARMOR_VOTER	• MSG_NOTIFY_MANAGER
• MSG_VOTER_CONNECT	• MSG_VOTER_FAILURE
• MSG_VOTER_DISCONNECT	• MSG_VOTER_CHANGE_MODE

Figure 2.14 : *armor_voter* message subscription list

voter_topology : this element is not yet implemented and will allow in future the creation of finite state voting automata, which converge to the bounded result. The configuration and theory that supports this element will be part of a separate paper.

3 - Chameleon Messsage Dispatching As in previous sections, we have extended the message addressing space alloving the registration of dynamic messages, i.e. of messages that are defined at run time in the normal environment operation and may be revocated, upon completion of the intended service. Chameleon messages are defined as unsigned integer: we have restricted the addressing space to integers having the first bit set for dynamic messages.

A *dynamic message* gives the ability to address specific elements, without increasing the overhead as the number of subscribing elements grows. This characteristic has been felt necessary to implement the io subsytem interface stubs and the voter stubs.

Moreover, the ability to register dynamic messages moves the responsibility of addressing from the receiving ends to the sources securing the delivered information against errors, which could arise from the same fault tolerance environment due to the coupling of elements sharing the same messages.

A very strong encapsulation of receiving parties is performed by the way of *private messages*, which are delivered only to a restricted group of end parties.

The above functionality is performed in differed ways: marking a message *exclusive* delivers it only to those elements, who previously registered the message; a *reserved* message is intended for the elements of a certain group, while a *private* message specifies in advance the list of receiving parties.

If an element attempts to subscribe to a message, which is not allowed, it gets an error reply.

The functions for handling dynamic messages are *RegisterMessage*(), *InvalidateMessage*(), *ReservedMessage*(), *MessageExclusive*(), *DefineGroup*(), whose calls are translated in the following list of statically defined messages:

MSG_REGISTER_MESSAGE
MSG_INVALIDATE_MESSAGE
MSG_RESERVED_MESSAGE
MSG_MESSAGE_EXCLUSIVE
MSG_DEFINE_GROUP

to access the functionality from the Chameleon application interface.

The dynamic message dispatching is currently under developement.

Conclusions

Chameleon framework aim is not to replace conventional fault-tolerance techniques, which can be still employed and are sometimes mandatory to assure hardware availability [VOAS98b]; the main goal of Chameleon and other similar dependable software architectures [CUKI98] is to increase the availability of services provided via untrusted networked systems, or using components-off-the-shelf (COTS), which most of the times cannot supply the same reliability of custom designed mission-critical softwares [IYER99], but are globally available at reasonable prices.

With the above intention, we designed some components, which can be easily integrated in existing software architectures to increase the reliability of the whole system; such components have been thought to guarantee at a time the availability of the basic "dependable" framework, which provides the fault-tolerance services, such as the smart voting subsystem.

The introduction of "virtual flows" provides a way to easily collect source data flow from existing applications, not specifically designed to be failure-tolerant, in order to submit it to the smart-voting system.

References

J. H. Wensley, "SIFT Software Implemented Fault Tolerance," presented at FJCC, 1972.

J. H. Wensley, Langley Research Center., United States. National Aeronautics and Space Administration. Scientific and Technical Information Office., and SRI International., *Design study of software-implemented fault-tolerance (SIFT) computer*. Washington, D.C.
[Springfield, Va.: National Aeronautics and Space Administration Scientific and Technical Information Office ;
For sale by the National Technical Information Service], 1982.

J. P. S. Eifert J.B., "Processor monitoring using asynchronous signatured instruction streams," presented at FTCS-14, 1984.

D. F. Green, D. L. Palumbo, D. W. Baltrus, and Langley Research Center., *Software implemented fault-tolerant (SIFT) user's guide*. Hampton, Va.: National Aeronautics and Space Administration Langley Research Center, 1984.

Y.-K. P. Thambidurai P., "Interactive Consistency with Multiple Failure Modes," presented at SRDS-7, 1988.

D. F. McAllister, M. A. Vouk, and United States. National Aeronautics and Space Administration., *Experiments in fault tolerant software reliability*. Raleigh, NC: North Carolina State University, 1989.

M. Malek, M. Pandya, K. Yau, and United States. National Aeronautics and Space Administration., *Redundancy management for efficient fault recovery in NASA's distributed computing system*
report for NASA grant NAG9-351. Austin, Tex.: Department of Electrical and Computer Engineering University of Texas, 1991.

K. P. Birman and United States. National Aeronautics and Space Administration, "The process group approach to reliable distributed computing," . Ithaca, N.Y.
Springfield, Va.: Dept. of Computer Science Cornell University ;
National Technical Information Service distributor, 1991.

D. D. Amir Y., S.Kramer, D. Malki, "Transis: A Communication Sub-System for High Availability," presented at FTCS-22, 1992.

P. S. Miner and Langley Research Center., *An extension to Schneider's general paradigm for fault-tolerant synchronization*. Hampton, Va.
[Springfield, Va.: National Aeronautics and Space Administration Langley Research Center ;
For sale by the National Technical Information Service, 1992.

J. G. S. Madeira H., "On-line Signature Learning and Checking," in *Dependable Computing for Critical Applications*, DCCA-2, Ed.: Springer-Verlag, 1992, 1992.

M. K. Reiter, "Distributing Trust with the Rampart Toolkit," *Comm. of the ACM*, vol. 36, pp. 71-74, . 1993.

P. S. Miner and United States. National Aeronautics and Space Administration. Scientific and Technical Information Program., *Verification of fault-tolerant clock synchronization systems*. [Washington, DC]
[Springfield, Va.: National Aeronautics and Space Administration Office of Management Scientific and Technical Information Program ;
National Technical Information Service distributor], 1993.

C. K. Huang Y., "Software Implemented Fault Tolerance: Technologies and Experience," presented at FTCS-23, 1993.

C. A. Liceaga, D. P. Siewiorek, and United States. National Aeronautics and Space Administration. Scientific and Technical Information Program., *Automatic specification of reliability models for fault-tolerant computers*. [Washington, DC]
[Springfield, Va.: National Aeronautics and Space Administration Office of Management Scientific and Technical Information Program ;
National Technical Information Service distributor], 1993.

K. P. Birman and R. Van Renesse, *Reliable distributed computing with the Isis toolkit*. Los Alamitos, Calif.: IEEE Computer Society Press, 1994.

M. R. Ohlsson J., "Implicit Signature Checking," presented at FTCS-25, 1995.

D. M. Dolev D., "The Transis Approach to High Availability Cluster Communication," *Communications of the ACM*, vol. 39, pp. 64-70, 1996.

K. P. Birman, *Building secure and reliable network applications*. Greenwich: Manning, 1996.

R. v. Renesse, K. P. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Communications of the ACM*, vol. 39, pp. 76-83, 1996.

V. H. Chandra T.D., S. Toueg, B. Charron-Bost, "On the Impossibility of Group Membership," presented at ACM Symposium on Principles of Distributed Computing, 1996.

D. Powell and e. al., "GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems," *IEEE Transactions on Parallel and Distributed Systems*, 1998.

[AGRA91] a. A. J. B. Agrawal D., "A Nonblocking Quorum Consensus Protocol for Replicated Data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 171-179, 1991.

[ASPR97] W. Aspray, "The Intel 4004 Microprocessor: What Constituted Invention?," *IEEE Annals of the History of Computing*, vol. 19, pp. 4-15, 1997.

[AVIZ77] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault-tolerance during execution," presented at IEEE COMPSAC '77, 1977.

[BIRM87] K. P. Birman, T. A. Joseph, and United States. National Aeronautics and Space Administration, "Reliable communication in the presence of failures," . Washington, DC
Springfield, Va.: National Aeronautics and Space Administration ;
National Technical Information Service distributor, 1987.

[BIRM87] K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems*, pp. 47-76, 1987.

[BORG89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault-Tolerance under UNIX," *ACM Transactions on Computer Systems*, vol. 7, pp. 1-24, 1989.

[BRIE93] D. Briere and P. Traverse, "AIRBUS A320/A330/A340 electrical flight controls - a family of fault tolerant systems.," presented at FTCS-23, Toulouse, 1993.

[CASN1] G. Neufeld and Y. Yang, "An ASN.1 to C Compiler," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1209-1220, 1990.

[CRIS91] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM*, vol. 34, pp. 56-78, 1991.

[CUKI98] M. Cukier and e. al., "AQUA: An Adaptive Architecture that Provides Dependable Distributed Objects," presented at SRDS-17, 1998.

[ESCH97] B. Eschermann, P. Terwiesch, A. M. AG, K. Scherrer, and A. N. P. AG, "Dependable High-Voltage Substation Protection," in *Dependable Computing for Critical Applications 5*, vol. 10, *Dependable Computing and Fault-Tolerant Systems*, M. M. Ravishankar K. Iyer, W. Kent Fuchs, Virgil Gligor, Ed. Los Alamitos, CA: IEEE Computer Society, 1997, pp. 19-34.

[IYER99] R. K. Iyer and A. Avizienis, "COTS Hardware and Software in High-Availability Systems," presented at FTCS99, 1999.

[JOHN89] B. W. Johnson, *Design and analysis of fault-tolerant digital systems*. Reading, Mass.: Addison-Wesley Pub. Co., 1989.

[KALB99] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *Transactions On Parallel and Distributed Systems*, vol. 10, pp. 560-579, 1999.

[KOEN89] A. Koenig and A. T. B. Laboratories, *C Traps and Pitfalls*, 1989.

[LESL84] L. Leslie, "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Transactions on Programming Languages and Systems*, pp. 254-280, 1984.

[LION96] J. L. Lions, "Ariane 5 flight 501 failure: Report of the inquiry board," , Paris July 19, 1996 1996.

[LITT91] B. Littlewood and D. Miller, "Software reliability and safety," . New York: Elsevier Applied Science, 1991, pp. x, 216.

[MAFF97] S. Maffeis, "Piranha: A CORBA Tool for High Availability," *IEEE Computer*, vol. 30, pp. 59-66, 1997.

[MARC94] J. J. Marciniak, *Encyclopedia of Software Engineering*: Wiley, New York, 1994.

[MICHE91] T. Michel, R. Leveugle, and G. Saucier, "A New Approach to Control Flow Checking without Program Modification," presented at FTCS-21, 1991.

[MS97] Microsoft, "Microsoft Clustering Architecture "Wolfpack"," , White Paper May 1997 1997.

[OSBO78] A. Osborne, *Z80 programming for logic design*. Berkeley, Calif.: Osborne, 1978.

[PHAM95] H. Pham, *Software reliability and testing*. Los Alamitos, Calif.: IEEE Computer Society Press, 1995.

[PIER65] W. H. Pierce, *Failure-tolerant computer design*. New York,: Academic Press, 1965.

[RAND75] B. Randell, "System structure for software fault-tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 220-232, 1975.
[SIWI98] D. P. Sieworek and R. S. Swarz, "Faults and Their Manifestations," in *Reliable Computer Systems: Design and Evaluation*, 3rd ed: A K Peters, Natick, Massachusetts, 1998.
[SUN97] Sun, "Sun RAS solutions for Mission-critical Computing," Sun, White Paper October 1997 1997.
[VOAS97a] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting How Badly 'Good' Software Can Behave," *IEEE Software*, pp. 73-83, 1997.
[VOAS97c] J. Voas, "Software fault injection: growing 'safer' systems," presented at IEEE Aerospace Confence, 1997.
[VOAS98a] J. M. Voas and G. McGraw, "Applied Safety Assessment," in *Software Fault Injection*, M. Spencer, Ed.: John Wiley & Sons, 1998, pp. 205-225.
[VOAS98b] J. M. Voas and G. McGraw, "Software Safety (Hyding Fault with EPA)," in *Software Fault Injection*, M. Spencer, Ed.: John Wiley & Sons, 1998, pp. 159-203.
[WHIS98] K. Whisnant, S. Bagchi, B. Srinivasan, Z. Kalbarczyk, and R. K. Iyer, "Incorporating Reconfigurability, Error Detection and Recovery into the Chameleon ARMOR Architecture," Center for Reliable and High Perforance Computing, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, Technical Report CRHC-98-13 UILU-ENG-98-2227, December 1998 1998.
[YEH97] Y. C. B. Yeh, B. C. A. Group, and F. S. Electronics, "Dependability of the 777 Primary Flight Control System," in *Dependable Computing for Critical Applications 5*, vol. 10, *Dependable Computing and Fault-Tolerant Systems*, M. M. Ravishankar K. Iyer, W. Kent Fuchs, Virgil Gligor, Ed. Los Alamitos, CA: IEEE Computer Society, 1997, pp. 3-17.