Fast Error Recovery in CHORUS/OS: The Hot-Restart Technology

Vadim Abrossimov, Frédéric Herrmann, Jean-Christophe Hugly,
Frédéric Ruget, Eric Pouyoul, Michel Tombroff
Chorus Systems, Inc.

August 30, 1996

1 Introduction

Building large fault-tolerant systems is a very complex enterprise, and has led to important developments in hardware and software design, implementation and testing methodologies. Until recently, very few commercially available operating systems provided the appropriate level of support for building fault-tolerant software systems, which is why these systems have in general been implemented using ad-hoc and proprietary solutions.

Telecommunication and inter-networking manufacturers, in particular, are facing extremely severe availability and reliability requirements, often dictated by international standards and, of course, market pressure.

The hot-restart technology provided in CHORUS/ClassiX r3 has been designed and implemented to address the high-availability requirements of system builders, telecommunication manufacturers in particular. It complements the already available CHORUS fault-tolerance enablers, like dynamic binding for reconfiguration and migration of software components, fault isolation and confinement, and support for replicated services (see [4]), by adding a mechanism allowing to recover from errors in an efficient way. It also allows to use existing, third-party software packages which have not necessarily be designed to be fault-tolerant, and integrate them into a robust system.

In this paper we first describe the motivations and requirements for high-availability features in the context of a modular, componentized(TM) operating systems like CHORUS/ClassiX r3, and how these requirements have been addressed with the hot-restart technology. We then present in more details the core mechanisms and policies implemented by the kernel to support the hot-restart features.

1.1 Availability and Reliability

In order to clearly describe and discuss any fault-tolerant technology and avoid confusion between often close although different concepts, one has to adhere to a precise terminology.

- **Availability** is the quality or state of being available, *i.e.* present or ready for immediate use ([6]).
- **Reliability** is the quality or state of being reliable, *i.e.* suitable or fit to be relied on ([6]).

In other words, a system is said to be fully available if it provides uninterrupted services. To approach full availability, hardware and software redundancy, hot-standby and lock-step techniques have generally been used.
A system is fully reliable if it always provides accurate services (according to its specification), when it is available. In other words, a system can be highly-reliable, even though it is not highly-available.
To build such systems, techniques involving hardware and software redundancy, combined with voting mechanisms, have generally been used.
The scope of fault-tolerance is to build reliable and available systems.

1.2 Faults, Errors and Failures

The following definitions are extracted from [5].

- An **error** is a difference between the actual system behavior and its specification.
- A **failure** is an event which corresponds to the first occurrence of an error.
- A **fault** is a source which has the potential of generating errors.

The goal of an error recovery procedure is to bring the system to an error-free consistent state. To illustrate these concepts, let us look at the following code fragment:

```
char* ptr;

ptr = (char*)malloc(strlen("hello\n"));
sprintf(ptr, "hello\n");
```

In this code fragment, the fault is the fact that the value returned by malloc() is not checked for NULL value. The potential error this fault may generate is that the pointer ptr may, in some cases, be set to NULL. The corresponding potential failure would be the actual occurrence of that error, which would manifest itself as an exception.
A typical operating system like UNIX(TM) provides several basic mechanisms to detect such error and/or failures.

- Error detection via assertion mechanisms:

  ```
  char* ptr;

  ptr = (char*)malloc(strlen("hello\n"));
  assert(ptr);
  sprintf(ptr, "hello\n");
  ```

- Failure detection via exceptions and signaling mechanisms:

```
extern void func(int);
char* ptr;

signal(SIGSEGV, func);
ptr = (char*)malloc(strlen("hello\n"));
sprintf(ptr, "hello\n");
```

In these cases, the detection of an error leads to the abnormal termination of the faulty entity. Such detection mechanisms are sufficient for building simple applications, but cannot be used to build highly-available systems. This observation is the basis for the requirements that led to the design of the hot-restart technology, as will be presented in the rest of this paper.

1.3 Generated and Propagated Errors

In the previous examples, the failure of the program was directly generated by a fault in that program. However, a failure can be caused by a fault that is not in the program itself, but in another component of the system. For instance, a fault in the virtual memory module of the CHORUS microkernel may cause a direct error in that module, in turn causing an error in the IPC module, itself causing a failure in an application program. This type of error is said to have been propagated from it originating fault.

Therefore, simplistic methods which only assume that the cause of a failure is directly linked to the failing entity cannot properly handle complex failure scenarios which arise in critical, real-time, distributed systems.

The study of this complex "error model", combined with the fact that systems based on the CHORUS/OS technology are often made up of a large set of closely interacting modules and actors, led us to design and implement specific features associated to the hot-restart technology, whose key characteristics are:

- Extension of the notion of fault confinement, normally associated to the CHORUS actor, to higher-level abstractions called "restart groups".
- Refinement of inter-actor invocation mechanisms, so to associate strong fault confinement semantics to them.
- Clear separation of the notions of "restart mechanism" from the one of "restart policy".

All these concepts are discussed in the rest of this paper.

2 Overview of The Hot-Restart Mechanism

The primary goal of the hot-restart features developed by Chorus Systems is mainly to address the high-availability problem (as opposed to the specific high-reliability problem), by providing a set of mechanisms allowing to:

- Capture failures and notify errors. These mechanisms are called "exception handlers" and "panic" system calls.

  These mechanisms have actually been provided by the CHORUS microkernel for a long time, but have been extended in the scope of the hot-restart framework to cope with the additional requirements imposed by the support of multiple, concurrent *subsystems* in a system like CHORUS/ClassiX r3.

- Recover from these failures and errors, and bring the system in an error-free state very rapidly, by “hot-restarting” the whole or portions of the system.

  In CHORUS/ClassiX r3, the CHORUS microkernel, as well as the ClassiX subsystems are ”hot-restartable”, *i.e.* restartable to their initial entry point without having to be reloaded from stable storage[1].

  The same mechanism can be applied to applications.

- Reconstruct the state of the hot-restarted portions of the system using a fine grain ”checkpointing” mechanism, based on a new type of CHORUS memory regions called “persistent regions”.

- Analyze errors using “error logging” and “core dumps” mechanisms, and repair faulty modules using a “patching” mechanism[2].

The combination of these features allows to construct highly-available systems and applications, by reducing dramatically the time it takes for a failed system or component to be back into service.

Before describing in more details the hot-restart mechanism, it is useful to rapidly recall some basic CHORUS concepts. The basic building block of CHORUS applications is the “actor” ([1], [3]). The actor represents the basic unit of resource allocation, and is the “shell” into which “threads” can execute. The actor also represents the basic error detection (exception handlers are attached to actors) and fault confinement (protected address space and execution boundary are associated to actors) abstraction. Quite naturally, the hot-restart technology also rests its foundations on the actor object.

2.1 Classification of Restart Actions

Failures can be caused by many types of faults, and all failures do not necessarily require the same recovery action. In order to provide enough flexibility to the system builder in choosing what best action is required, three types of ”restart actions” which can be applied to a failed actor are defined, listed below in decreasing order of “severity”.

- **Termination.**
  The failed actor is terminated. It is the most severe action in the sense that the actor is not given any ”second chance”, at least not until an external agent decides to recreate that actor. This simple action is similar

[1]Which, by contrast is often referred to as a “cold-restart”.

[2]The error logging, core dump and patching features are not described further in this paper. They will be the subject of a future companion paper.

to the abnormal termination mechanism described above for a traditional UNIX system.

- **Reload.**
  The failed actor is first stopped, its current state cleaned up, and finally, its code and data segments are reloaded from stable storage. The actor is therefore restarted again with a "fresh" code and data, possibly a new version of the program.

- **Hot-restart.**
  The failed actor is hot-restarted: all its non-persistent objects (threads, ports, stacks, private data, and non-persistent regions) are destroyed, its text and data segments are reinitialized to their original content without accessing the stable storage, and finally, the actor resumes its execution at its entry point.

The time it takes to hot-restart an actor is much shorter, compared to the time it takes to reload the same actor. Furthermore, the state of the actor is not entirely reinitialized after a hot-restart: persistent regions are kept intact, which is the basis for performing checkpointing actions and recreate the state of the failed actor to a state prior to the failure. This checkpointing mechanism is described in more details in Section 2.2 below.
In the case of a reload, obviously, the state of the actor prior to the failure is entirely lost. One may wonder what the distinction is between terminating and then "manually" recreating an actor, as opposed to reloading it. The difference lies in the fact that the reload action is performed automatically by the system, while an actor that has been terminated will not re-execute unless explicitly recreated by either another actor or a user.

2.2 Checkpointing

The goal of error recovery, as defined in the Introduction, is to bring the system to an error-free consistent state. The basic hot-restart mechanism described in the previous section defines this error-free consistent state to be the state corresponding to the initial loading of the failed component.
Figure 1 shows the state of an actor at its initialization, during execution, and after having been hot-restarted as a result of an error. In this first example, the data region of the actor is reinitialized to its initial content.

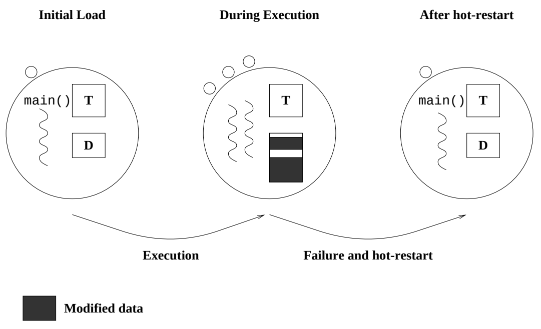

Figure 1: Basic actor hot-restart

Clearly, this is a very conservative approach, in general not sufficient to build usable fault-tolerant software. Therefore, CHORUS/ClassiX r3 provides a fine-grain "checkpointing" mechanism, based on the notion of "persistent memory region". An actor can allocate persistent memory regions which will stay intact after a hot-restart. These regions serve as checkpoint containers to the application, which can record into them the state information which will be required to reconstruct a consistent state in case the application is hot-restarted. Note that in any case, the code and non-persistent data regions are automatically re-initialized to the content corresponding to the initial load of the application.

Figure 2 shows the same scenario as Figure 1, except that the actor had declared a couple of persistent memory regions, and is therefore capable of quickly reconstructing its state.

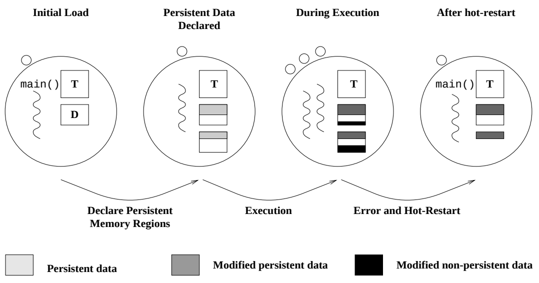

Figure 2: Persistent Actor Hot-Restart

These two characteristics (time to hot-restart versus time to reload, and persistence of designated memory regions) are the foundation of the hot-restart technology.

The attentive reader will probably ask the following question: *if the concept of hot-restart is to quickly restart execution with a restored version of its code and data, with possibly some memory regions kept intact, would it be possible to implement this feature in library, using for instance some kind of* `setjmp()/longjmp()` *mechanism?*

The negative answer to this question is justified as follows:

- CHORUS
  actors are not simple, mono-threaded execution objects. First, they can be multithreaded, but, more importantly, their code can be executed by "external" threads, which have invoked the actors via some invocation mechanism. Allowing such invocations to take place while guaranteeing sufficient fault confinement and isolation required support from the microkernel itself.

- CHORUS/ClassiX r3 supports, in addition to the concept of actor hot-restart, the one of "site restart". This means that the entire system - the site - can be hot-restarted, guaranteeing to each hot-restartable actor the same semantics as described above. Clearly, this mechanism relies on some low-level kernel primitives.

- Finally, the concept of actor itself is not rich enough to build highly-available, cooperative, distributed applications, and the hot-restart concept had to be extended to higher-level abstractions called *restart groups*.

These technical aspects are covered in the following sections.

2.3 Restart Groups

Many applications are made up of not one but several actors, which cooperate to provide certain type of services. As these actors cooperate closely together, any failure in one of them can have repercussions to the others.

For instance, let us assume that actors A and B cooperate closely (via CHORUS/IPC for instance), and that A fails. Simply terminating, reloading or hot-restarting A will probably not be sufficient, and will most certainly cause B either to fail itself, or to go through some special recovery action.

Recovery actions themselves, like roll-back, may in addition cause the well-known "domino effect" ([5]).

Building cooperating applications which can cope with the large number of potential fault scenarios is a very complex task, as the complexity grows exponentially with the number of actors.

Therefore, we have introduced the notion of "restart group", which is an abstraction allowing to group actors together and associate a common restart action to the group. In other words, when one actor of the group is submitted to a restart action (for instance a hot-restart), all the other actors of the group undergo the same action.

In addition to the notion of restart group, and in order to provide some form of inter-group restart semantics, we have implemented a "restart notification" mechanism, which allows an actor to be notified of the failure of a given restart group. This notification mechanism is capable of performing local and remote notification. It is not described further in this paper.

Finally, restart groups identify clear "failure domains", and extend therefore naturally the error confinement model generally associated to individual actors.

2.4 Restart Groups Hierarchy

2.4.1 Restart Groups and Subsystems

A system built on top of CHORUS/ClassiX r3 is often made up of a combination of multiple subsystems personalities running on the same CHORUS microkernel. Clearly, there exists a very close relationship between the notions of restart group and subsystem, for the following reasons:

- The actors implementing a subsystem cooperate closely together to provide a well-defined set of services. It is therefore natural to associate a common error recovery mechanism to all the actors implementing a given subsystem.

- The various subsystems do not have the same criticality. As these subsystems may be hosted on the same machine, it is important to isolate subsystems from each other, from the error confinement and recovery points of view.

2.4.2 Restart Groups Dependencies

We have seen in the previous section that there is a close relationship between the notions of restart groups and subsystems. There is clearly a dependency between the various subsystems, and between the subsystems and the applications. For instance, if the ClassiX subsystem fails (e.g. because of an exception in the Actor Manager), and is subsequently hot-restarted, all actors running at that time on top of ClassiX must also be at least hot-restarted. This dependency between one layer to the ones above it can be illustrated using a tree-like structure, as shown on Figure 3.

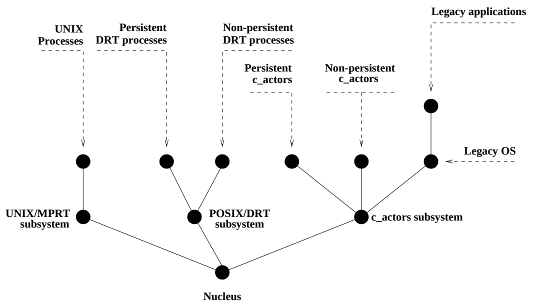

Figure 3: Restart group dependencies

These dependencies must be managed at the lowest level of the system. They represent the basic structure for analyzing error propagations and for activating the appropriate error recovery actions.

2.5 Site Restart

The restart group model is quite general, in the sense that the CHORUS microkernel itself, with the associated basic OS service actors, are also members of a restart group, the "kernel" restart group. This restart group has the particularity that it is the "root" of the restart group dependency graph. The various restart actions can also be applied to the kernel restart group. Because of the particular nature of that restart group, these actions are also called "site restart" (for a hot-restart), "site reboot" (for a reload).

2.6 Restart Mechanism and Policy

The three restart actions (termination, reload and hot-restart), taken individually, are the basic toolkit for dealing with a failed actor or more generally a failed restart group. However, when faced with a complex system like CHORUS/ClassiX r3, we had to address the following design issues:

- How do we categorize all the restart groups in the system using the above restart action severity paradigm? For instance, do we consider that a failure of a given actor is always more severe than the one of another?

- How do we cope with different availability requirements (e.g. very high for public switching equipment, less critical for private switching, irrelevant for a toaster)?

- How do we cope with changing conditions, such as variations in system loads, occurrences of consecutive or concurrent faults, etc.?

- And finally, how do we solve the above issues and still build a generic, multi-purpose operating system?

This problem is a classical problem of system software design, which led to the separation of two notions: "mechanism" and "policy". In the same way CHORUS/ClassiX r3 separates the notion of scheduling mechanism from the one of scheduling policy, it makes the distinction between the restart mechanism and the restart policy.
The mechanism is represented by the restart actions, while the policy is represented by algorithm that decides which restart action to take under each failure situation. The restart policy, a replaceable module of CHORUS/ClassiX r3, is the "umpire" of the system as far error recovery is concerned; it drives all the restart actions when failures occur. It can be replaced by a customer specific policy, adapted to the specific needs of the applications.

2.7 Restart Escalation and Propagation

In addition to deciding which restart action needs to be taken for a failure of a given restart group, the restart policy can take higher level decisions, using complex state information and taking into account the dynamic nature of the system. In particular, it can decide to "escalate" or "propagate" the restart actions. These terms are defined below.

- **Escalation.**
  Let us suppose that an actor fails, and that the restart policy decides to hot-restart it. A short moment after, the same actor fails again. The restart policy, assuming that the fault was therefore not transient, may decide to reload the actor, hence causing a fresh binary image to be reloaded, which will, hopefully, repair the fault.

  This is called "restart action escalation": the restart action has been escalated from a hot-restart (less severe) to a reload (more severe). Figure 4 illustrates this case.

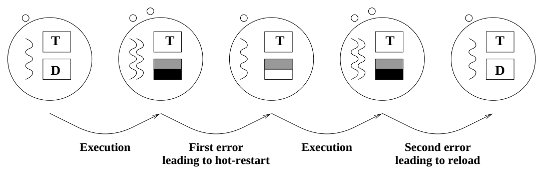

Figure 4: Restart escalation

- **Propagation.**
  Let us suppose that the same actor fails a third time, after having already been hot-restarted and then reloaded. The restart policy may in this case assume that the error is actually caused by a fault in another component of the system, for instance the kernel. It therefore decides to hot-restart the kernel. This is called "restart action propagation". As a result of this kernel hot-restart, the actor itself is of

course also hot-restarted. Figure 5 illustrates this case.

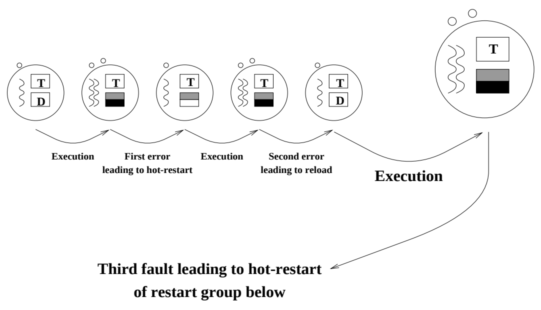
T
D
Execution
First error leading to hot-restart
Execution
Second error leading to reload
Execution
Third fault leading to hot-restart of restart group below

Figure 5: Restart escalation

The default restart policy implemented in CHORUS/ClassiX r3 contains several configurable parameters, allowing to tune the behavior of the restart policy in case of complex failure scenarios (how long before escalating, after how many faults, by which actor, *etc.*).

3 Advanced Topics

In Section 2 above, we have described the basic principles of the hot-restart feature. In order to implement these features, we had to develop a set of supporting modules, actors and APIs (Application Programming Interface). Three of the most interesting of these concepts are described in the Sections below.

3.1 Fault Isolation and Confinement

As we described in Section 2, the actor is the basic abstraction for fault isolation and confinement.

An operating system like CHORUS/ClassiX r3 consists of a set of actors and threads, each interacting closely, via various interaction mechanisms. The code and data of an actor (in particular supervisor actors) can be executed not only by the own threads of that actor, but also by external threads and other interrupt-type execution flows. These invocations are generally called "handlers", and can be categorized as follows:

- Traps handlers.
- Interrupt handlers.
- Timeout handlers.
- Timer and virtual timer handlers.
- Exception handlers.
- Abort handlers.

The features described in the following two sections have been developed to make sure these invocation mechanisms could still be used in the context of hot-restart, without affecting the system's performance.

3.1.1 Local Access Points

Because of the above invocation mechanisms (or handlers), an error occurring while executing the code of an actor can be caused not only by the actor's own threads, but by many other execution flows.
In order to allow these interactions to be permissible in the context of an highly-available system, as well as providing the

required level of performance, we have designed and developed a new invocation mechanism called "Local Access Points", or LAPs.

LAPs are light-weight methods for accessing the services of an actor, while still guaranteeing the proper restart semantics when crossing inter actor boundaries. They allow the system to keep track of the "identity" of the various flows of control which traverse an actor, and to guarantee that an actor can be hot-restarted even thought is may currently be executing several "foreign" threads.

Figure 6 illustrates how threads and interrupts can execute the code of an actor by invoking its LAPs. It shows that an actor can be invoked from other actors, or from an interrupt level flow (the clock in this case). Note that a thread is allowed to block while executing a LAP.

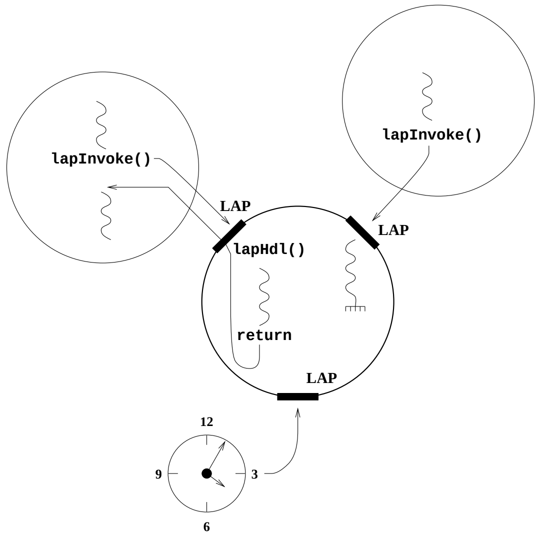

Figure 6: Local Access Points (LAPs)

3.1.2 Freezing and Restarting

Hot-restarting an actor is a complex operation. First, as we explained before, an actor may be hosting concurrent threads, not necessarily its own threads. If one of these threads fails while executing the code of that actor, the kernel may have to hot-restart that actor, without destroying threads which were temporarily executing in the actor.

Second, as the hot-restart operation is not atomic, and since we certainly do not want to lock the system while an actor is restarted so as to preserve the real-time characteristics of the system, the microkernel must make sure that no thread or interrupt will try to invoke an actor's code while this actor is being hot-restarted.

To solve these two problems, we have decomposed the operation of hot-restart of an actor into two distinct "restart phases":

- **Freeze.**

  The freeze operation stops all current execution flows in the target actor, and guarantees that all subsequent invocations will be forbidden. The microkernel achieves this by deconnecting any potential source of interrupt in that actor, and by deflecting any thread which may want to either penetrate or resume execution in the frozen actor.

  The freeze operation is, as it names indicate, the basic mechanism used to confine an error in a faulty actor. This is illustrated on Figure 7.

- **Restart.**

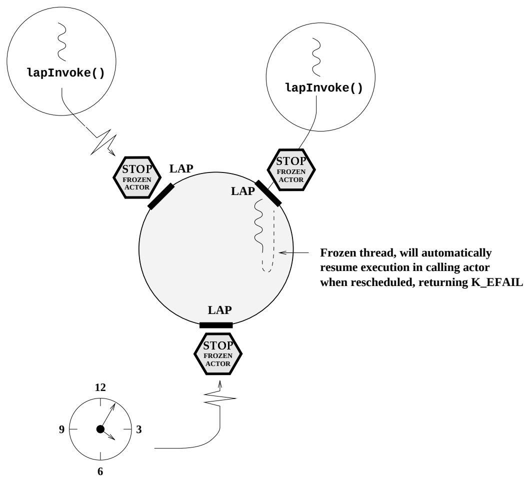
lapInvoke()
lapInvoke()
STOP FROZEN ACTOR
LAP
STOP FROZEN ACTOR
LAP
Frozen thread, will automatically resume execution in calling actor when rescheduled, returning K_EFAIL
LAP
STOP FROZEN ACTOR
12
9
3
6

Figure 7: Actor freeze

Once an actor has been frozen, it can be restarted. This means that all its non-persistent resources (ports, threads, private data, non-persistent regions) are destroyed, and the actor reincarnates with a new name[3].

The job of the microkernel is finished at this point, and it is the responsibility of the appropriate subsystem to reactivate the actor by creating a new thread into it.

3.2 The Hot-Restart Managers

3.2.1 The Site Personality Manager

In this paper, we have explained that applications, subsystems and the entire site itself can be subject to several types of restart actions. In particular, in CHORUS/ClassiX r3, it is possible to capture failures of individual subsystems, and to terminate, reload or hot-restart them independently of each other.

In order to support these mechanisms, subsystems themselves have to be considered as some kind of "application", in the sense that some entity must be responsible for detecting their errors, capturing their faults and, finally, apply the restart actions on these subsystems.

We have therefore implemented a new module of the CHORUS Nucleus, called the "Site Personality Manager", or SPM. The SPM is the module responsible for loading subsystems, detecting errors and capturing failures generated in these subsystems, and apply the recovery actions on these subsystems.

3.2.2 The Restart Manager

As we said before, we separated the notions of restart mechanism and restart policy. One module of the CHORUS Nucleus, called the "Restart Manager" (or RM), is responsible for decomposing the various restart actions (for instance, a hot-restart is composed of a freeze phase, followed by a restart phase, and finally an activate phase which actually reactivates the hot-restarted actor).

It is also responsible for managing the dependency relations between the various restart groups, in the form of the restart group tree discussed above.

Finally, it is the module into which replaceable restart policy modules can be "plugged".

[3]Actor names are called "capabilities" in the CHORUS terminology.

4 Conclusion and Future Work

The hot-restart feature is now implemented and available as early access of the CHORUS/ClassiX r3 product.
This work has raised a lot of extremely interesting topics, which we did not envision when we first embarked on this project.
First, after using the hot-restart features for several months now, we have realized that it is much more difficult to analyze and debug a system that never stops, even after it has failed. This will certainly lead us to explore new methods of monitoring and debugging.
Second, applying the hot-restart concept to emerging markets, like multi-media and inter-networking, will certainly allow us to enrich the technology and analyze new restart policies.
And finally, the prospect of applying these mechanisms to objects (CHORUS/COOL) and richer invocations mechanism (COOL-ORB) is an open door to many tempting projects.

References

[1] *Cooperative Operating System Kernel Architecture*, M. Gien, Chorus Systems Technical Report, CSI/MR-94-22, March 1994.

[2] *Evolution of the CHORUS Open Microkernel Architecture: The STREAM Project*, M. Gien, Proceedings of the 5th. IEEE Workshop on Future Trends in Distributed Computing Systems, Cheju Island, Korea, August 28-30, 1995.

[3] *The CHORUS Microkernel*, Dick Pountain, Byte Magazine, pp 131-139, January 1995.

[4] Fault Tolerance Enablers in the CHORUS Microkernel, J. Lipkis, M. Rozier, Chorus Systems Technical Report, CS/TR-93-45, June 1993.

[5] *Fault-Tolerant System Design*, S. Levi, A.K. Agrawala, McGraw-Hill Series on Computer Engineering, 1994.

[6] *Webtser's Ninth New Collegiate Dictionary*, Merriam-Webster, 1995.

[7] *Restart Manager Requirement Specifications and High-Level Design*, Eric Pouyoul, Chorus Systems Technical Report, CSI/TR-95-7.

[8] *c_actors Restart Requirement Specifications and High-Level Design*, Jean-Christophe Hugly, Chorus Systems Technical Report, CSI/TR-95-9.

[9] *Site Personality Manager Requirement Specifications and High-Level Design*, Eric Pouyoul, Michel Tombroff, Chorus Systems Technical Report, CSI/TR-95-8.

[10] *CHORUS/Nucleus v3 r6.1 Actor Restart Requirement Specifications and High-Level Design*, Frederic Ruget, Chorus Systems Technical Report, CS/TR-95-167.