

HAFTA Architecture Overview

Copyright ©2001-2003, Astra Network Inc. All Rights Reserved

January 3, 2003

This is documentation for a BETA version of HAFTA. Please be advised that you may encounter problems; please report them so that they may be fixed. The latest information about HAFTA is always available at <http://www.astranetwork.com/hafta/>.

Contents

1	Introduction	1
2	Checkpoint Library	2
2.1	Sequences	2
2.2	Checkpoints	3
3	Overlord	3
3.1	Language	3
3.1.1	Nebuloids	3
3.1.2	Procedures	4
3.1.3	Processes	4
3.1.4	The System	4
3.2	Compiler	4
3.3	Runtime	5
3.4	Nebuloid Modules	5

1 Introduction

This document provides an overview of the initial release of the High Availability and Fault Tolerant Architecture (HAFTA) Toolkit. The toolkit provides both a library providing architectural support for the development of fault tolerant programs and a runtime which maintains the system in a certain state.

This document assumes a working knowledge of the requirements for high availability and fault tolerance.

The HAF TA product is divided into two major components: A checkpoint library to aid in the development of your application and a runtime manager called the overlord, which ensures the system stays in the state you define.

This product has been developed in a very platform non-specific manner and currently runs on QNX4, QNX6, Linux and Solaris.

The product is provided in open source form, so you can see exactly how the system works and you are encouraged to write your own plug-in modules to support any proprietary hardware or software you may use.

2 Checkpoint Library

The client application can use this library to help ensure the flow of the program is developed in such a way in which it can recover from any type of unexpected problem.

This is accomplished by dividing the program structure into two levels, the sequence level, and the checkpoint level.

2.1 Sequences

A sequence consists of an associative array style data structure containing a list of functions that the sequence will call. Each set of functions in the list, known as a node, contains a **normal** function, a **rollback** function, and a **policy** function.

A sequence is invoked similarly to the way one might invoke a single function, except that it is done through the API of the checkpoint library.

The important features of the sequence structure

- The progression through the nodes in the sequence is linear or branching. It is not a dependancy map.
- The nodes are anonymous and uniform. There is no need to differentiate between setup, running, or teardown functions.
- The nodes are self-directing. The execution sequence is indicated to the checkpoint library by the **policy** function, which is described below.
- Sequences are independant and nestable. Inter-sequence dependancies are left to the domain of the developer's code.
- When a node is invoked, it will be pushed onto the stack as it is called. This allows a node's policy function to roll back to a previous node should it determine that an error can't be corrected in the context of the current node.
- Errors encountered during the execution of the sequence are reported by the developer's code to the checkpoint library.

2.2 Checkpoints

Checkpoints are a facility for allowing the rollback of a partially completed function. Checkpoints are a strictly linear enumeration of the resources allocated by the programmer's code.

The value of the last checkpoint set by a particular **normal** or **rollback** function is stored, then presented to the **rollback** function so that the programmer's rollback code can know what resources it needs to free and/or deconfigure.

The important features of the checkpoint structure

- The progression through the checkpoints is strictly linear, and as such is 100% predictable. Branching is handled in the scope of the sequence structure, not the checkpoints.
- The checkpoints are maintained by both the **normal** function and the **rollback** function. This is necessary in case the **rollback** function fails and needs to be re-executed.
- The checkpoint system is designed to be inobtrusive and lightweight. It does not directly affect the flow of execution.

3 Overlord

The overlord is the runtime component of the HAF TA package. It is broken into a series of programs, only one of which actually needs to run on the target system.

The overlord system consists of a scripting language and script compiler, a psuedo-code interpreter and main loop, and a series of plug-in modules.

These plugin modules provide complete user flexibility as they provide the software and hardware specific features you need for your system.

3.1 Language

The Overlord language is designed to allow the user to define a system—what processes should be running, how to start them, what to do if they fail, etc.—and to tell the Overlord what it needs to do in order to keep this system running smoothly. To this end, it utilizes the concept of processes, procedures (procs), and nebuloids. These concepts are defined in the following sections.

3.1.1 Nebuloids

Perhaps the most interesting concept of the Overlord language, nebuloids are a common, simple interface to Overlord plugin modules, and provide most of the power of Overlord language scripts. They can be thought of much like variables—in fact, one of the major nebuloid modules *is* the **VAR** module—except with the ability to perform functions when read, such as checking the memory usage of

a process. How exactly a nebuloïd behaves is based entirely on the module it uses.

Nebuloïds can be assigned to and read from, though not every module supports these abilities. Every time a nebuloïd is read from, it generally performs whatever function the module was designed to do. Many modules also require that the nebuloïd be instantiated, in order to give the module context about what task exactly it is to be performing. For example, when the LOG module is instantiated, it is told where it will be logging to (stderr, a file, or the syslog). When the LOG nebuloïd is read from, it takes the message to be logged and the severity of the log message as arguments.

3.1.2 Procedures

Procs are simply a collection of Overlord scripting commands which can be called at specific intervals or by other procs, etc.

3.1.3 Processes

A process definition in the Overlord language corresponds directly with a process running in a UNIX-like environment. The process definition tells the overlord how to start a process, what to do when it dies, and contains nebuloïds and procs to aid in determining whether or not that process is misbehaving.

Processes also have the concept of dependencies, which are simply other processes which must be running in order for that process to function properly. It is ensured that these dependencies are running when the process is started, and the process is notified if one of the other processes it depends on fails. The user can determine the best course of action in this case, whether it be to simply kill off the process and let the Overlord restart it, to simply ignore it and let the process itself deal with it, or to attempt to notify the process in some way that its dependency is temporarily unavailable.

3.1.4 The System

A system simply is a collection of processes, procs, and nebuloïds instances (collectively known as *policy*). The procs and nebuloïds defined here can be accessed system-wide by scripting commands. (Procs and nebuloïds defined in processes can only be accessed within their own process.) The system also has the concept of dependencies; it must be told what processes must be running for the system to function.

3.2 Compiler

The script compiler system consists of a compiler, *olc*, which creates readable assembly language form and an assembler, *ola*, which converts the assembly into a binary pcode format.

script source $\xrightarrow{\text{olc}}$ script assembly $\xrightarrow{\text{ola}}$ binary pcode

This process can occur on your development platform and only the pcode binary and olrt, the overlord runtime needs to be placed on the target hardware.

A further advantage is the compiler could run on one architecture and the runtime on another. The binary pcode format is architecture independent.

3.3 Runtime

The runtime component is responsible for reading the binary pcode, resolving any module dependencies, loading the modules and then monitoring the system to ensure the dependencies are maintained. If a dependency fails, the runtime takes whatever action was defined in the scripts. These actions are usually implemented using modules. For example, if the script called for restarting a program and logging the failure, the **EXEC** and **LOG** modules would be used to provide this functionality.

3.4 Nebuloid Modules

The purpose of the nebuloid modules is several fold. The modules allow you to only include the features you need, thus keeping the runtime size as small as possible. The modules allow the compiler and interpreter to be written in pure C with no implementation specific features. All the hardware and os specific code is in the modules.

The system can be compiled with either static modules linked into the runtime or with the modules provided as shared objects. This allows for easier field upgrading of modules.

Modules can be as simple as the **VAR** module which provides the concept of a variable to the scripting language or slightly more complicated with os specific features such as **MEM** which gives the current memory usage of a particular process or as complicated as you need to talk to your particular hardware.