

Getting Started with HAFTA

Copyright ©2002-2003, 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/> .

1 Introduction

This document provides an introduction to using the High Availability and Fault Tolerant Architecture (HAFTA) Toolkit..

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

HAFTA provides two major components to support developing highly available and/or fault tolerant systems.

The first component is a checkpoint library which allows you to design and develop your application with the concept of sequenced operations, checkpoints and rollbacks built in. Because this is something that has to be designed in from the initial design, it is best applied when the initial low level design is conceived.

The checkpoint library includes:

- sequences – sequences of code to accomplish some specific functionality
- checkpoints – flags within a sequence to facilitate rollbacks
- rollbacks – user defined actions to take when sequences fail
- policies – user defined policies to handle rollbacks

The second component is a runtime watchdog process which reads and implements a system definition description. This description tells the runtime, called the overlord, which process depend on which resources and what actions to take if the dependancy fails. This component can be added after the systems has been developed but the most complete implementation would integrate it into each component of your system.

The overlord includes:

- olc – script compiler – produces people readable form of byte code.

- ola – byte code assembler – produces byte code.
- olrt – overlord runtime – reads byte code and interprets it.
- modules – user defined extensions – extends olrt to add custom functionality

HAFTA is implemented as a platform/os independant toolkit which currently runs on QNX6, QNX4, Linux and Solaris and should run on any Posix compliant OS which has the POSIX Realtime Extensions available.

We will take you through the initial creation of your own working copy of HAFTA, followed by a simple example to see how it can be used for your projects.

2 Getting Started

This section will describe how to get HAFTA up and running on your development system. It assumes you have already installed the distribution package into /opt/hafta using the appropriate mechanism for your platform.

2.1 Your working directory

HAFTA is distributed in source form and you are expected to customize and compile it for your particular application. This will include changing the directories to only include the modules you need, adding your own modules if you have needs which are not satisfied with the default modules and any other changes you deem necessary to make your system highly available.

The checkpoint library, is just a normal library and as such is just installed in the usual location for such libraries.

The overlord component is where all the customization is done. In order to help with this, you can create your own copy as follows:

```
$ mkdir ~/hafta
$ cd ~/hafta
$ cp -R /opt/hafta/src/overlord ./
$
```

2.2 Inital Compile

At this point, you can now make the default configuration for your platform and make sure everything compiles:

```
$ cd ~/hafta/overlord
$ make
```

3 Overlord

The overlord is the runtime component of the HAFTA 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 nebulooids. These concepts are defined in the following sections.

3.1.1 Nebulooids

Perhaps the most interesting concept of the Overlord language, nebulooids 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 nebulooid 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 nebulooid behaves is based entirely on the module it uses.

To help keep the terminology straight, a nebulooid is the object used with in the scripting language. A module is the source code used to implement the nebulooid.

Nebulooids can be assigned to and read from, though not every module supports these abilities. Every time a nebulooid is read from, it generally performs whatever function the module was designed to do. Many modules also require that the nebulooid 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 nebulooid 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. (As we will see later, there are many other structures which can contain Overlord scripting commands; procs can be called by these, as well.) Scripting commands are discussed later on in this article.

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 nebuloids 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 nebuloids (collectively known as *policy*). The procs and nebuloids defined here can be accessed system-wide by scripting commands. (Procs and nebuloids 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 The Compilation Steps

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 psuedo-code or pcode format.

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

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.

4 Checkpoint Library

4.1 Basic Client Program Structure

The checkpoint library allows you to design and develop your application with the concept of sequenced operations, checkpoints and rollbacks built in. Because this is something that has to be designed in from the initial design, it is best applied when the initial low level design is conceived.

A program written with the checkpoint library is structured into two levels, the sequence level, and the checkpoint level.

4.2 Sequences Overview

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 roll-back 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 client library.

4.3 Checkpoints Overview

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.

5 Simple Example

As an example of how the various components of the system work together, we will develop a data server and client app. The data server will simply provide a named pipe which contains a "known" amount of data and the client app which continually try to read this. If the server fails, the overlord will restart it. If the client notices, it will rollback and try again.

5.1 Data Server

The data server will be a simple program which runs on all the supported platforms (ie. no platform specific tricks). For this purpose we will create a named pipe, and loop through writing the alphabet to it.

5.1.1 server.c source

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>

#define PIPENAME "/tmp/simple"

int main(int argc, char *argv[] )
{
    int pd; // fd for named pipe

    srand(time(NULL));
    if( mkfifo( PIPENAME, S_IRUSR | S_IWUSR ) )
    {
        if( errno != EEXIST )
        {
            printf( "Unable to make fifo \"%s\" (%s)\n",
                PIPENAME, strerror(errno));
            exit( EXIT_FAILURE );
        }
    }

    pd = open( PIPENAME, O_WRONLY );

    if( pd == -1 )
    {
        printf( "Unable to open fifo \"%s\" (%s)\n",
            PIPENAME, strerror(errno));
        exit( EXIT_FAILURE );
    }

    while(1)
    {
        if( (rand()%25) == 0 )
            write( pd,
                "This is bad data and has to be at least 25 chars!!!!",
                52 );
        else
            write(pd, "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 26 );
        sleep(1);
    }
    close( pd );

    exit( EXIT_SUCCESS );
}
```

You can copy this source from the distribution:

```
$ cd ~/hafta
$ cp /opt/hafta/examples/getstart/server.c .
```

```
$ cc -o server server.c
$
```

There is nothing special about this server to make it highly available. In a real world server, it would use the checkpoint library to handle unexpected errors. In our case here, we will use the overlord to restart it if it fails.

Note: This server will not fail if the pipe already exists. This is to allow the server to restart without failing.

The server also randomly inserts bad data into the outgoing data stream in order for the client to show how it can handle bad data.

5.2 Client Application

Now in the client case, we are going to add a few smarts. Although it will be as simple as the server, it will use the HAFTA checkpoint library to make the application roll back and try again when someone goes wrong.

The application will read data in 26 byte chunks and it “knows” that the first byte will always be an A. We will use this information to confirm that we are working and if we have a problem, we will roll back and try to open the pipe again.

The checkpoint library requires the application to consist of one or more series of sequences. Each sequence has one or more nodes which contain normal function to do whatever that node was supposed to do, a policy function to decide what to do when you succeed or fail, and rollback function to compliment the normal function in case you have to undo whatever the normal function did.

The client consists of one sequence. That sequence consists of two nodes. The first node initializes the system (opens the pipe) and the second loops while reading the data. It uses checkpoints to record it’s progress, so the rollback can properly handle what it needs to fix.

5.2.1 Normal Functions

In the first node, we try to open the pipe. If it fails we call `HC_NormalFail` with the reason it failed. If it succeeds, we pass the new node we want to proceed to.

Note: It is worth mentioning that the `policy_data` argument to `HC_NormalFail`, `HC_NormalSuccess`, `HC_RollBackFail` and `HC_RollBackSuccess` is completely defined by the particular policy. The default policy expects the Success calls to pass the new node number to proceed to, and the Fail calls to pass the error.

In the second node, we loop up to 25 times, reading the pipe, then validating the data, then repeating the loop. Of importance here, is the `HC_Checkpoint` calls. These are used in the rollback functions to understand where in the normal function the problem occurred.

If we succeed without problems through the 25 loops, we return success and finish the sequence.

5.2.2 RollBack Functions

In the first node, the RollBack function does three things, two obvious and one not so obvious. Obviously it prints a failure message, and then sleeps for a second to give the overlord a chance to restart the server. The non obvious thing it does is always succeed. On success we go to the node which is passed as an argument. The reason we always want to succeed, is because the default policy will rollback to the previous node after 5 consecutive failures and we really don't have anywhere to rollback to. Failure would try to rollback right away and make even less sense. Success has the effect of retrying forever – which may not be something you want to do in your application, but as a tutorial, we are showing how to do it.

In the second node, we look at the checkpoint data to decide what to do with it. In this case we do nothing more than optionally print a message. We also return Success, but as you will see in the policies, we handle success differently in each node.

5.2.3 Policy Functions

In both these policy functions, we only replace the functionality in which we want to override the default. You can certainly replace the whole function if you need to. You can take a look at the default function in `/opt/hafta/src/checkpoint/lib/HC_Default.c`.

In the first node, all we want to override is the default behaviour on failure and never rollback to the previous node. This would allow us to return a failure from the normal function for as many times as you want.

In the second node, we differentiate why we failed from the error codes which we passed on the NormalFail call. In the case where we failed to read, we assume this means the server is dead and short circuit the default of 5 retries and immediately rollback to the previous node. In any other case, which is limited to bad data, we print the bad data and fall through to the default which will try to read the data again until it succeeds.

5.2.4 Creating and calling a sequence

Although sequences are nest-able, you can create and call new sequences from any of your current sequences functions, we kept it simple here and only have one sequence in the main function.

The process to call a sequence is fairly simple. You need a nodelist structure which contains the an array of the node number, the type of function, and the function pointer. You use this to create a sequence pointer which is passed to all the functions. When you want to “call” your sequence, you call the function `HC_CallSequence` and pass it the sequence you just created, some user specific data and the starting node number.

Once done with the sequence, it can be released with a called to `HC_DeleteSequence`.

5.2.5 client.c source

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>

#include "checkpoint.h"

#define PIPENAME "/tmp/simple"

#define CP_CLEAR      0
#define CP_RECV_DATA  1
#define CP_GOOD_DATA  2

#define ERR_OPEN_FAILED 100
#define ERR_READ_FAILED 101
#define ERR_BAD_DATA    102

#define NODE_EXIT      0
#define NODE_INIT      1
#define NODE_GETWORK   2

int
normalInit (HC_Sequence_t *sequence, void *userdata)
{
    int *fd = (int *) userdata;

    *fd = open (PIPENAME, O_RDONLY);
    if (*fd < 0)
    {
        HC_NormalFail (sequence, ERR_OPEN_FAILED);
    }
    return (HC_NormalSuccess (sequence, NODE_GETWORK));
}

int
normalGetWork (HC_Sequence_t *sequence, void *userdata)
{
    int *fd = (int *) userdata;
    int rc, i = 0;
    char buf[27];

    while( i < 25 )
    {
        HC_Checkpoint (sequence, CP_CLEAR); // clear checkpoints

        memset (buf, 0, sizeof (buf));

        rc = read (*fd, buf, 26);
    }
}
```

```

        if (rc != 26)
        {
            return (HC_NormalFail (sequence, ERR_READ_FAILED));
        }

        HC_Checkpoint (sequence, CP_RECV_DATA);
        if (buf[0] != 'A')
        {
            return (HC_NormalFail (sequence, buf[0]));
        }
        printf("%d successful read(s)\n", ++i );
    }
    return (HC_NormalSuccess (sequence, NODE_EXIT));
}

int
rollbackInit (HC_Sequence_t *sequence, void *userdata,
              unsigned long checkpoint)
{
    printf("Open failed! Perhaps the server died?\n");
    sleep (1); // open failed - wait and try again
    return (HC_RollBackSuccess (sequence, NODE_INIT));
}

int
rollbackGetWork (HC_Sequence_t *sequence, void *userdata,
                 unsigned long checkpoint)
{
    int *fd = (int *) userdata;

    switch (checkpoint)
    {
        case CP_RECV_DATA:
            printf("Some bad data\n" );
        case CP_CLEAR:
            // we don't do any thing different whether the read failed or the
            // data is bad.
            break;
        default:
            HC_Panic (sequence, "Invalid checkpoint %ul!\n", checkpoint);
            break;
    }
    // We say that we succeed so we try again. The policy will try 5 times,
    // then go to the previous node and reopen the file.
    return (HC_RollBackSuccess (sequence, NODE_GETWORK));
}

int
policyInit (HC_Sequence_t *sequence, void *userdata,
            HC_PolicyEvent_t event, long policy_data)
{
    switch (event)
    {

```

```

        case HC_NORMALFAIL:
            // Rollback forever
            return (HC_RollBackCurrent (sequence));
        default:
            return (HC_DefaultPolicy (sequence, userdata, event, policy_data));
    }
}

int
policyGetWork (HC_Sequence_t *sequence, void *userdata,
               HC_PolicyEvent_t event, long policy_data)
{
    int *fd = (int *)userdata;

    switch (event)
    {
        case HC_NORMALFAIL:
            // override default in case of failed read - there is no point in
            // trying this 5 times.
            if (policy_data == ERR_READ_FAILED)
            {
                close( *fd );
                return (HC_RollBackPrev (sequence));
            }
            printf("policy_data = %c\n", policy_data );
        default:
            return (HC_DefaultPolicy (sequence, userdata, event, policy_data));
    }
}

int
main (int argc, char *argv[])
{
    HC_NodelistNode_t nodes[] = {
        {NODE_INIT,    HC_NORMALFUNC, normalInit},
        {NODE_GETWORK, HC_NORMALFUNC, normalGetWork},
        {NODE_INIT,    HC_ROLLBACKFUNC, rollbackInit},
        {NODE_GETWORK, HC_ROLLBACKFUNC, rollbackGetWork},
        {NODE_INIT,    HC_POLICYFUNC, policyInit},
        {NODE_GETWORK, HC_POLICYFUNC, policyGetWork},
        {0,            HC_LISTEND, NULL}
    };

    HC_Sequence_t *sequence;
    int fd;
    int rc;

    sequence = HC_NewSequence (nodes);

    rc = HC_CallSequence (sequence, &fd, NODE_INIT);

    fprintf (stderr, "Sequence was: %s\n", HC_Strerror (rc));

    HC_DeleteSequence( sequence );
}

```

```
    return (EXIT_SUCCESS);  
}
```

You can copy this source from the distribution:

```
$ cd ~/hafta  
$ cp /opt/hafta/examples/getstart/client.c .  
$ cc client.c -lcheckpoint -o client  
$
```

6 Overlord Script

Once we have a couple programs we want to describe, we need to create the Overlord Runtime (olrt) script which will describe the system. In this case we have a simple server and client, but the concept can be extended as far as is required to support your system.

6.1 Choosing modules to use

To keep this example fairly simple, we will only monitor the existence of the two processes. In other words, we will only react if the process dies for some reason, and will not worry about a process getting “stuck” or in some other unexpected state.

For this example, we will use four modules. Remember modules extend the functionality of the language in the same way that the C library adds to the C language.

The following modules will be used:

- `exec` – used to start processes.
- `getpidbyname` – returns the first pid which matches the name.
- `log` – outputs a string.
- `logf1` – outputs a formatted value.

6.2 Examining the script

In the same way we include headers in C, there are include files which are generated when we created the modules and we want to include them.

```
// Prototype modules required by this script.  
  
include "exec.include";  
include "getpidbyname.include";  
include "log.include";  
include "logf1.include";
```

Again for all the same reasons we want macros in C, we have a define operator which allows us to define constants.

Each process in the system can be polled to check to see if it is still responding, you define what this means. In this case we want to check the system every 100 milliseconds.

```
define POLL_INTERVAL 100
```

The actual system definition starts with the keyword `system`, followed by a list of it's dependencies. In this case, we only depend on "client".

```
system: client
{
```

Using the modules we have included, we want to declare global nebulooids which we will use in the various actions.

```
    // Create global instances of the modules required
    exec launch();
    getpidbyname getpid();
    log logerr(0, "test");
    logf1 logserverpid(0, "test", "server pid is %s");
    logf1 logclientpid(0, "test", "client pid is %s");
```

Without getting into alot of detail of how the various modules work, you can see that we can initialize the various nebulooids with different parameters.

Some nebulooids are very generic, for example, `launch()` which just takes the executable name and can be used to launch any executables. Other's like the `log` nebulooids are initialized with the format string, which makes each nebulooid specific to a particular use.

The next step is to define to the system how you want to monitor the process called "client" and what you want to do when starting, or restarting it.

```
    // check once per POLL_INTERVAL milliseconds to see if client is alive
    process client POLL_INTERVAL: server
    {
```

This says we have a process in the system which we will call "client". This name does not have to be the same as the executable name and if you had more than one client, you could call the first `client1` and the second `client2`.

This line also says we want to check this process every `POLL_INTERVAL` milliseconds and that it depends on something called "server".

```
        // run this once to start it
        start
        {
            // check to see if it was started by someone else in which case we
```

```
        // just record the pid.
        client = getpid( "client" );
    if( client == 0)
    {
        launch("./client");
        logerr(5, "Starting client");
        client = getpid("client");
    }
    endif;
    logclientpid(5, client );
}
```

OK, this action, called the start action, is executed the first time the overlord checks and fails it's test for client.

Note: Think of client as a variable and it passes if it is non-zero.

The first thing we do it test to see if it is already running. This is **VERY** important as if you don't do this, you will not be able to start or restart the overlord without it failing to work as you expect – ie. it will do your actions regardless if they have been done before or not. It is up to you, the script writer to make sure your scripts work as you expect.

If the client is not running, ie. getpid could not find a pid to “client”, we then call launch to start it. We log this fact and then check to see that it actually did start. We don't test to see if that failed, because the overlord will notice and call the restart action if that is true.

```
        restart
    {
        client = getpid("client");
        if( client == 0 )
        {
            launch("./client");
            logerr(5, "Restarting client");
            client = getpid("client");
        }
        else
        {
            logerr( 5, "No restart required");
        }
        endif;
        logclientpid(5, client );
    }
```

In a similar way, this action is called when the overlord notices the process has failed after it has run the start action.

```
        dependfail
    {
        logerr(5, "client needs server running");
    }
```

This action is called if something you depend on fails. In this case if the server fails, this action is called.

```

process server POLL_INTERVAL:
{
    // run this once to start it
    start
{
    // check to see if it was started by someone else in which case we
    // just record the pid.
    server = getpid( "server" );
    if( server == 0 )
    {
        launch("./server");
        logerr(5, "Starting server");
        server = getpid("server");
    }
    endif;
    logserverpid(5, server );
}

    restart
{
    server = getpid("server");
    if( server == 0 )
    {
        launch("./server");
        logerr(5, "Restarting server");
        server = getpid("server");
    }
    else
    {
        logerr( 5, "No restart required");
    }
    endif;
    logserverpid(5, server );
}
}

```

This section describes the server process and is similar to the client description. The only real differences are the server doesn't depend on anything and therefore having a dependfail action doesn't make sense.

In either case the actions are optional, although there would be some question as to why you want to watch the process if you are not going to do anything if it fails.

6.3 Compiling the Script

The overlord runtime (olrt) is really a virtual machine (vm) which runs a psuedo code we created for this purpose. In order for you to "run" the script you just created, you need to compile it. The compilation is a two step process:

- Compile source to psuedo-assembly
- Assemble into pcode

We made the default installation install a precompiled version of the binary versions of the compiler, assembler and runtime. Obviously as you modify the runtime you will want to use your version.

The script from above is available in the same place as the other example code. You can copy it over and compile it as follows:

```
$ cd ~/hafta
$ cp /opt/hafta/examples/getstart/example.script .
$ olc -f example.script | ola -q
$
```

This produces a binary version of the pcode which the runtime uses. You will see a file called p.out which was produced by the assembler.

7 Testing the System

At this point we can now try it out. In one window, you can simply run olrt:

```
$ olrt
$
```

The overlord will notice that the server isn't running and start it, followed by the client. At this point the client should talk to the server and it should happily print out it's success.

The first test will be to kill the client. In a second window, do a "ps" and get the pid of the client. Use this to kill the client. You will notice the overlord reacts by restarting the client.

Next we repeat the same process with the server. This time the client notices and rolls back to reopen the file. Since there is no delay in the client rollback it tries several times before the overlord and restarts the server. Obviously all of this can be tuned to provide the behaviour you would want in your application.

Now we will try something different. Instead of killing the client or server, we will kill olrt. Nothing obvious happens. You can now look to see if it is still running (it shouldn't be). Obviously killing the client or server will make them stop. The trick is you can now restart olrt and because we did such a good job of writing our script, it all just works.

8 Conclusion

Obviously this is just the tip of the iceberg. There are many different things you can do with HAFTA because you control both the scripts and the modules.

You can write modules to access your hardware, or detect heartbeat messages from your applications.

The checkpoint library allows you to have control of your code both as it works in it's normal mode and how the system can handle various failures.

The software which makes up both the checkpoint library and the overlord components are shipping in source form so you can both inspect how they are made and extend them as required for your application.

We hope you come up with some inovative uses for it and let us know what you created. We would love it if you contributed back any modules you have created so they can be shared with other users.

The HAFTA Development Team