|  |
| --- |
|  |
| BCP/DR Automated Failover and Test system  Technology EIS EP Engineering  Version 0.1 (Draft) |
| 15 jul 2011 |
| Copyright © 2011 Nomura  This document is the sole property of Nomura. No part of this document may be reproduced in any form or by any means - electronic, mechanical, photocopying, recording or otherwise - without the prior written permission of Nomura. |

# Introduction

This is a discussion document intended to present options for Automated Failover and Testing (otherwise referred to as ‘the system’) in order to achieve consensus on the manner in which the system should be implemented.

Nothing in this document should be considered to be ‘cast in stone’ although it is the authors belief that the approach recommended is the best one given the authors current knowledge.

The analysis carried out has considered the overall requirements of the system, has considered existing similar systems and has taken on-board the best parts of existing systems where appropriate.

Some of the higher level analysis; specifically the role of A2RM and how it integrates with the workflow of the system has not yet been carried out. This analysis is scheduled to be carried out before 6th August.

Much of the remainder of the system has been designed from a bottom-up approach, in a modular manner, with key components designed and implemented to a ‘proof of concept’ stage.

It is expected that numerous people will eventually need to be involved in the creation of plug-in components to the system, with that in mind the system has been designed to make it very easy to ‘encapsulate’ these components and make them easy to integrate into the system.

# The problem

The current BCP failover testing process is a heavily manual process requiring Infrastructure staff from several teams to manage the failover of various components including.

* SAN storage volumes
* Databases
* DNS service aliases
* Cluster Services

These manual tests can take several hours and will not scale to testing application failover frequently or in the event of a real disaster.

## Requirements

All tasks identified so far can be carried out on the command line. These commands can either be standard OS commands, third party tools or bespoke scripts.

There are a limited number of these tasks and so automation should be possible if certain conditions are met.

* All configuration data is available (for example in A2RM, either currently or by enhancing A2RM)
* Systems are ‘standardized’ so that we do not have to handle exceptions.
* That each task should have a testable ‘precondition’ which can be verified before the task is attempted.
* Each task should have a testable ‘success condition’ which can be verified after the task has been completed.
* Tasks should be co-ordinated from a central location or locations but run on remote systems.
* Linux, Solaris and Windows remote systems should be supported.
* The sequence of tasks should be organized into a workflow which takes into account the success of each preceding stage.
* On the failure of a task, the workflow should halt, produce an alert and allow for manual intervention before continuation of the workflow.
* Multiple workflows should be able to be run concurrently, both to allow multiple systems to be tested at the same time and to allow for a real disaster situation.
* The system should allow for a distinction between a real disaster recovery situation, (for example where the primary data centre is lost) and a test situation (where a production system can be stopped gracefully to avoid data loss).

## Configuration data

This is by no means an exhaustive list, it is simply data correlated from observation of a single DR test and is provided as an example of the type of information that would be required.

* The name of the application, e.g. **phoenix**
* The alias for the application, e.g. **phoenixdctmprd.nomura.com**
* The global container for the production application server, e.g. **lonux10052**
* The name of the production application zone, e.g. **lonux10425**
* The global container for the DR application server
* The zone for the DR application server, e.g. **lonux20405**
* The production DB host, e.g. **lonlx1024b06-c**
* The production cluster name, e.g. **SYSBPRD1009**
* The production database, e.g. **LN\_SA\_OTCDM\_PRD1**
* The DR DB host, e.g. **lonlx2029b05-c**
* The DR cluster name, e.g. **SYBPRD2008**
* Production VIP, e.g. **ln\_sa\_otcdm\_prd1-sg**
* DR VIP, e.g. ln\_sa\_otcdm\_dr
* CNAME, e.g. ln\_sa\_otcdm\_prd1

### Standardization

I noted during a DR Test session that some configuration files were changed manually (for DNS changes) but that in some cases the place where the new configuration was to go was blank, but in another case it was commented out. Automatic scripts to update configuration files would be much easier to maintain and more reliable if the configuration files they are editing are in a standard form.

### Pre-conditions

Most tests are likely to have pre-conditions that may either have to be met, or may be optional. For example, when copying a file, the source file must be present.

### Success-criteria

Tests to confirm that the task has been carried out successfully must always be performed. For example on changing a cname entry, sufficient time must be allowed for the change to propagate and for the system to see the new entry.

### Central co-ordination

If there needs to be a central system to co-ordinate the operation of the DR Test and Recovery system, this implies that in the event of a real disaster situation this central system would be brought on-line in the DR environment first. In any case, the A2RM system is already a pre-requisite which would need to be brought on line early (and probably manually).

### Remote running

Scripts need to be run on remote systems, probably under a specific user account or root and co-ordinated from the central system.

### Support for multiple platforms

The system needs to be able to run commands on remote Linux, Solaris and Windows servers. It is likely that different versions of the remote code will be needed for different environments, certainly for Windows.

### Workflow

Every DR Test schedule will be organised as a workflow, with checkpoints along the way where manual testing can take place (for example DR testing). This workflow should be automatically generated as far as possible, perhaps from a list of templates for common operations.

### Alerts and logging

Each task has both a pre-requisite and a success/fail status, in the event of any failure the system should halt and not continue with the workflow but flag an error allowing for manual intervention to either correct the problem or to abort the DR test. On correcting a fault the system should either be allowed to continue to the next task or to repeat a task.

Logging of individual tasks, down to the level of command input and output should be provided so that in the event of a failure detailed diagnostics can be obtained. These log files should be accessible from a central server and aggregated from the various remote systems.

### Concurrent operation

The system should be designed so that several DR Tests/recoveries can be carried out at the same time. This is so that in the event of a real DR there will not be a bottleneck caused by the recovery system.

However care needs to be taken in some areas where co-operation may be needed between tasks, a case in point might be DNS updates where it would make sense for DNS changes to be aggregated rather than have every DR task initiate its own update.

However I understand that the DNS system is being modified soon to use QUIP which will likely remove even this issue so any implementation of DR Testing will assume the use of QUIP.

### Distinguish between real DR and test DR

Tasks need to be able to be run whether or not the production system is currently running/accessible or not. For example in a DR test it might be advisable to power down a production system gracefully to simulate it going of-line. This could be handled by one of the tasks themselves, but the same task must also allow for the fact that in a real DR situation the production system may not be available to shut down and it should continue without failing.

# Existing systems.

## Database Management Toolkit (DMT)

The DMT system provides core maintenance of databases (backup, load, reorgs/re-index, statistics etc.), database copy and restore, reporting and audit features.

Of special interest is the way that the DMT system is constructed, it consists of a suite of programs and perl modules that are installed on any server. Once installed this suite provides a set of command line tools that allow for the control and operation of the database activities from any machine to any other by means of Socket connections. This means the system is distributed and resilient.

Although it may not be appropriate to modify DMT to be the DR system, it is possible that we can extract the key components of DMT (the secure socket connections and the authentication) to be used in DR.

A further aspect of DMT is that it is extendable by means of plugins allowing the core functionality to be extended by adding new modules without having to re-write the key components. The DR system should also adopt a similar Plugin approach with a well defined API so that Plugins conformant to the API can easily be written by members of the various teams using the tool.

## CA Workload Automation AE (AutoSys)

AutoSys is used extensively in Nomura, primarily as a job scheduler (a crontab replacement).

It is possible that we could base a DR Testing system on AutoSys although this is not what AutoSys was designed to do.

Also, basing a DR system on AutoSys means that we are creating a dependency upon another system, albeit a tier 1 highly redundant system. The fewer dependencies we have the easier it will be to initiate Disaster Recovery in the event of a real emergency.

## Marionette Collective (MCollective) from Puppet Labs.

PuppetLabs have a system called MCollective which allows for the parallel execution of jobs on remote systems.

Although it is designed for the execution of commands on clusters of servers by means of a publish/subscribe middleware, there is no reason why it could not be used to administer individual remote systems.

While not required for DR failover this could allow the framework to be extended to other uses where parallelism would be advantageous (e.g. reporting or patching)

It provides methods to create simple RPC style agents and clients and Web UIs in an easy to understand language.

It is extremely pluggable and adaptable with rich authentication and authorization models using SSL, RSA, Authorization and Auditing.

# Concerns and Issues

There are a number of issues with an implementation that still need to be addressed in detail.

* **Authorisation and Authentication** – how do we carry out authentication of users in order to ensure that the user has the authority to execute commands on remote systems? It is likely that some commands on the remote system will need root access. This will require the setting up of trusted users with Kerberised Access to Root and a means of transmitting this information from the Local machine to the Remote machine. The DMT system has an authentication system but we would need to review it to see if it is secure. MCollective is highly configurable but it is likely that we would still need to write a customised authorisation module. AutoSys has authorisation and authentication built in.
* **Security** –since it is possible that we will be executing arbitrary commands on remote machines, in user accounts with a high level of access, it is important that it is not possible to ‘inject’ arbitrary commands into that communication stream.
* **Perl version** – The ‘standard’ installation of perl is 5.8.8 which is now about five years old and is in need of updating. It would be appropriate to base any new development on version 5.12. Unless there is a mechanism put in place to upgrade all servers to a more modern version of Perl it would probably be best to package Perl together with the application.
* **CPAN modules** – It is not clear (at least to me) how systems are updated with a new CPAN module, or with a more up-to-date version of a CPAN module. For that reason we should package any required CPAN modules together with the application code to make it independent of any system libraries.

# Proposal

It is proposed that the system is designed using the principles adopted in the DMT system, that is, a server component communicating to a client over a socket. This is known to work with all platforms, Linux, Solaris and Windows (using winsock).



This design splits the system into two virtual components a Server and a number of remote Clients in the same way as DMT. The Server and Client can either be on different physical machines or the same physical machine.

On the Server, the Workflow layer interfaces to A2RM and co-ordinates sequences of commands required to carry out an operation, for example the controlled shut-down of a production system and switch over to the DR system during DR testing.

Above the Workflow layer sit interfaces to the system. The Command Line Interface is included to provide the same sort of functionality as DMT offers at the moment. The Web API is included so that external systems can interface with the system and control it.

The Application Layer organises and mediates requests to the various plugin components.

Plugin Components in the Server are very simple, they define the main component (Application, Database and Storage are shown on the diagram, but there can be any number of these main components ) and the methods performed on that component. (For example, Database backup)

The transport layer of the Server and Client carries out the two way transfer of data in a manner which is transparent to the higher layers. The transport layer could be replaced by a different mechanism without affecting any of the plugin components. Mechanisms anticipated at the moment include the Sockets method exemplified by DMT and direct access if the client/server are running on the same machine.

Plugin Components in the Client echo the organisation of the plugin components on the server, however whereas the server plugin components simply defined which methods were available, the client versions contain the bulk of the code.

Where methods need to support multiple operating systems or external systems, such as databases or storage systems, different versions of those methods need to be provided. For example, an Application Control method (which can be used to start or stop applications) would need to have a Linux/Solaris version and a Windows version. Similarly a Database Backup method would need to have both a MySQL and an Oracle version.

The System Interface Layer provides common routines to facilitate communication with the client operating system.

## Workflow

Initial analysis of how the workflow integrates with A2RM is scheduled to be carried out by the 6th August.

However, it is expected that workflows will generally fall into a smallish number of typical scenarios, each of which can be templated with key data being obtained from A2RM.

TBD.

## Components

The design of the system ensures that it is easily extensible, allowing for the addition of new components with the minimum of change to the existing code.

This is achieved by means of a pluggable construction. To add a new component it is sufficient that a new set of Perl modules are written and tested and added to the directory structure.

Each component is written to conform to a standard interface definition using a core set of features. These core features provide the communication between the server and client via the communication layer and allow the component to be driven from the command processor module.

A pluggable architecture provides the following advantages.

* Adding a new module does not require changing existing code, this reduces the chance that new code will break existing code, ensures a more robust system and one which is easier to maintain and debug.
* A well defined interface for the component means that other people should be able to write and test new components as required.
* The system will automatically detect new pluggable components and add them into the code base automatically without the need for a manual code change or configuration changes.

## Example. Application - Control

This example describes a simple component, one that can be used to start/stop or retrieve the status of an application or process running on Linux, Solaris or Windows. This is represented on the above system diagram by the Application – Control component.

### Server Component

The server component is simple, it just needs to define the data shared between the server and the client and the methods that the component supports.

package Nomura::DRAFT::Plugin::App::Control;

#

# Generic methods and attributes for a plugin

#

use Moose::Role;

### The data shared between the server and the client

# Name of application

has 'app\_name' => (

is => 'rw',

isa => 'Str',

required => 1,

);

sub \_build\_shared\_data {

return [qw(app\_name)];

}

### The methods supported by this plugin

sub \_build\_methods {

return [qw(start stop status)];

}

### Mandatory methods

sub usage {

my ($self) = @\_;

print "USAGE: app\_\_control --app\_name <application name> <command>\n";

print " e.g. app\_\_control --app\_name my\_app start\n";

}

1;

This is a simple application, the data shared between the server and the client is just the name of the client application/process to be controlled, **app\_name**.

The methods supported by the component are **start**, **stop** and **status**

In addition, some mandatory methods are required, such as **usage** which provides help text for any command line use of the system.

It should be emphasised that all of the heavy lifting and shifting carried out by the application is handled in a generic manner either in the Application Layer, or in the Transport Layer.

### Client Component

The Client Component carries out the methods defined in the Server Component, i.e. **start, stop** and **status**. It also has to understand the difference between a linux/solaris and a Windows environment so we create two modules.

Here is the generic component (simplified for demonstration purposes).

package Nomura::DRAFT::Generic::Plugin::App::Control;

#

# Plugin to control applications, start/stop/status etc.

#

use Moose::Role;

sub start {

my ($self) = @\_;

my $command = "\etc\init.d\".$self->app\_name." start";

return `$command`;

}

sub stop {

my ($self) = @\_;

my $command = "\etc\init.d\".$self->app\_name." stop";

return `$command`;

}

sub status {

my ($self) = @\_;

# To be defined

return 1;

}

1;

There would be an equivalent component for Windows,

package Nomura::DRAFT::Windows::Plugin::App::Control;

....

Which I will not attempt to demonstrate at this point except to say that it will implement the same **start,stop** and **status** methods but in a Windows environment.

The choice of which component is used (in this case) is determined by the client recognising which operating system it is running on or being told by the server.

## Client/Server protocol

The communication protocol between the server and the client is based loosely on the telnet protocol where the server opens the connect ith the server, sends a number of commands which the client responds to and then when finished the server closes the connection.

An example of a typical session between the server and the client is shown below. For illustration purposes (not part of the protocol), the protocol exchanges are prefixed with s: for the server and c: for the client.

s: HELLO lonlx20575

c: OK HELLO

s: COMMAND App::Control->start

c: OK COMMAND

s: DATA

s: bless( {'app\_name' => 'foo'},

s: 'Nomura::DRAFT::Plugin::App::Control::SharedData')

s: .

c: OK DATA

c: RESPONSE

c: bless( {'success' => 1, 'app\_name' => 'foo'},

c: 'Nomura::DRAFT::Plugin::App::Control::SharedData')

c: .

s: OK RESPONSE

s: QUIT

c: OK QUIT

(The server closes the connection)

The session is opened by the server sending the **HELLO lonlx20575** which informs the client of the name of the server. All commands are acknowledged with an **OK**, in this case with an **OK HELLO** response.

Next the server sends the command that it wishes to be executed on the client, **COMMAND App::Control->start** this will be the module name and method of one of the plugin components installed on the client. If the plugin and method are found the client will respond with an **OK COMMAND**.

Data shared between the server and the client is sent next, this data is assumed to be multi-line data so it is prefixed with a **DATA** command and several lines until a line consisting of a single period ‘**.**’ is sent. The client will then respond with an **OK DATA**.

At this point the client has enough information to carry out its function, the command is executed and the result of the command is returned in a **RESPONSE** block which the server is waiting for. This is encoded in the same manner as the **DATA** and the server acknowledges receipt of it with an **OK RESPONSE**.

The server can now terminate the session with a **QUIT** command, the client responds with **OK QUIT** and the server can close the connection.