An Open Framework for Clustering

Alan Robertson − *IBM Linux Technology Center* − <alanr@unix.sh> / <alanr@us.ibm.com>

ABSTRACT

One of the most commonly identified features which is felt to be necessary for Linux^{TM} to be considered "enterprise−ready" is High−Availability. High−Availability (HA) systems provide increased service availability through clustering techniques. Clustering is also used to create High−Performance Clusters (HPC). These two different techniques have many things in common, but little has been done to unify them.

In the case of HA clusters, several open source high−availability projects have been created. These projects were created independently for complex historical reasons, not because of political, philosophical or licensing differences. Because of this, they originally shared no code at all. This minimized the benefits of the open source model, which both encourages and benefits from the sharing of common components. However, many of them share the need for a component for resetting cluster members. A component was created for filling this need with the specific intent of being a common component across open HA systems. This was successful beyond all expectations, and this component has become standard across most open source HA systems. All of the projects involved have benefitted from this commonality. Although this component is currently used primarily in HA clusters, HPC systems also have need of this capability.

In light of this success, the author began to search for more ways to extend these benefits across a broader set of cluster infrastructure components. Towards this end, we have created an architecture for an open cluster framework for both HA and HPC systems, and have begun to implement it. Although this framework has its origins in High−Availability systems, it is engineered to be applicable to high−performance clusters as well. This paper provides some background on open source clustering, defines this cluster framework, outlines its goals, describes key design elements of this framework, and details progress in implementing them.

Introduction

High−Availability failover techniques are commonly viewed as critical components in the set of enterprise capabilities of an operating system. In this regard, Linux is no exception, and it is well understood that these capabilities are essential for its wide acceptance in the enterprise server arena.

As a result, several different open source High−Availability projects have come into being. For the most part, these projects are separate for historical reasons, not for philosophical, political, or licensing reasons. Each of them has their own user community whom they faithfully support, and who expect to continue to receive support. They are willing to share components and software, but they find it difficult to do so, because they have no common infrastructure or assumptions which would enable this sharing. The STONITH module is an exception, since it was designed from the beginning to be a common component. It is the success of this common component which has prompted the work being described.

It should be understood that every cluster requires a common set of capabilities. These capabilities are often be implemented in different ways, but fundamentally serve the same purpose.

If these components were fit into a common component framework, then they could be assembled into many different cluster systems, each of which would fulfill different requirements in uniquely different ways. For example, one could imagine a small, embedded cluster system with minimal capabilities, or a large heavy−duty clustering system, and many in between which could be assembled out of components which fit into this framework.

In many respects, this creates a new class of configurable clustering systems, capable of meeting many different classes of needs well. Each of the users of the framework could select components from the project framework set, and assemble them into clusters which uniquely meet the needs of their particular customers. In many ways, this provides the best of the proprietary and open worlds − allowing competitors to differentiate themselves from each other, yet at the same time allowing them to share infrastructure for those components which do not have to be customized to meet the unique needs of their particular customer set. This maximizes both the benefits of the open development model, and the opportunities for competition in meeting the needs of unique market and technology niches

It also provides an ideal vehicle for research into cluster systems − since it allows researchers to concentrate on their area of interest and simply use components from the framework which they need for their research, but are not their area of concentration.

Background

As was mentioned earlier, there are several different Open Source clustering systems which were developed independently. However, they are all licensed under the GNU GPL. The author was involved with the development of two of them at one point in time, and it became clear that both needed a Linux implementation of a reset mechanism. So, a class of loadable modules was created for a reset mechanism. This proved to be quite successful, and has been contributed to by several different HA projects, not just the original two. The author actually only wrote one of the modules, and around a half−dozen more have been written and contributed by several different organizations.

This allows all the projects to avoid re−inventing the wheel, and maximizes the benefits of the open source model, as the "eyeballs" and developers of the community are spread across fewer lines of code, giving a more full−featured, better result. Although this subsystem was small and its goals modest, it was remarkably successful, and has become the de−facto Linux standard for resetting cluster nodes.

There were several things which contributed to making it so strikingly successful. These include:

- Clean design. The design of the modules was clean, simple, easy−to−understand, and separate from involvement with the surrounding software. This made it an obvious choice to adopt, and easy to integrate into various cluster systems.
- Loadable modules (plugins). Since any given installation typically only used one reset mechanism at a time, having a module loading system made for cleaner system interactions, and has minimized system bloat. This has also led to a largely object−oriented approach to these modules, which enhanced the clarity of the design.

However, there is one notable weakness in the current STONITH system. The configuration of STONITH objects is awkward, and largely ill−suited for user−friendly configuration using a GUI. This results from a design decision to use an unstructured string to represent the data to configure the STONITH objects. This data required to configure STONITH objects (and indeed, most objects) requires richer semantics than a simple unstructured string to properly communicate the structure of this information to a GUI or other configuration system.

In addition to the lessons learned from the STONITH libraries, there is another attribute of the heartbeat software which was influential in this design as well. In heartbeat, all messages are sent as name−value pairs similar to the UNIX shell environment. As was explained in [ROB00], this helps significantly both with portability across machine types, and with cluster version management interactions. However, it is limited, in that it is incapable of dealing with more complex structured data such as lists. As a result, it is not powerful enough to serve as a unifying mechanism in a general cluster framework, or powerful enough that all clients could use it for their control messaging..

Creating A Framework

The term "framework" was chosen instead of the word "design" for this project. The term framework is used here to mean a collection of common infrastructure components and APIs which permit one to create a cluster system out of components which fit in (or conform to) the framework.

In my view these APIs and infrastructures should be highly neutral (or agnostic) towards all the following things:

- Processor architectures, Operating systems, environments, and versions.
- Programming languages
- Cluster implementation strategies (shared storage, shared−nothing, etc.).
- HA versus HPC clustering

It is important to note that the infrastructure itself should not implement any cluster capabilities − but provide a context for creating cluster capabilities. This is what allows the framework to be agnostic with respect to cluster implementation strategies. Note that some of the infrastructure may interact closely with or require certain cluster components. For example they might use or require basic communication services. These interactions are acceptable (and often necessary), but can increase the software size/weight/complexity of the smallest possible working cluster node.

Although the framework itself will maintain this strict neutrality, it is not necessary particular implementations of cluster components retain this neutrality.

For example, it is acceptable to implement a cluster component which uses a driver module which is only available on Linux on Power PC platforms. What is important is that the API is designed so that some form of its function can be performed in every target environment.

Some cluster components may only be usable in a master/slave environment, while others may assume an equal peerage arrangement. Such diversity is sometimes necessary in the implementation of components. However, it is not acceptable for these implementation choices to show through to APIs which define its interactions with the other cluster elements.

Goals

The goals of this framework have been heavily influenced by the experiences described earlier.

1. Encourage sharing of components, and increase the number of shared subsystems beyond the STONITH subsystem.
2. Allow each of the various HA projects to use components from the framework without requiring them to abandon their current customer set.
3. Allow the components to be adopted individually, with minimal expectations regarding the remainder of the framework.
4. Keep the design of individual component interfaces clean and "value−neutral". Make sure the APIs do not assume things about the rest of the cluster system and its implementation. In particular, avoid taking sides on implementation methods and techniques (like shared storage, versus shared−nothing clusters).
5. Make extensive use of loadable modules for framework components and provide a single, general module loading environment. This is essential given the desire to allow solutions to be assembled from sets of components which come from a diverse range of environments.
6. Provide a common infrastructure for configuring modules which does not require adding new code to the GUI to configure new types of plugins. I use the term self−configuring plugins for this property.
7. Use a simple XML subset for the external representation of all data. Any time data passes out of the memory of a process onto disk or across the network to another cluster member, XML will be the preferred data representation method. This provides the advantages attributed to name,value pairs, yet also allows for sending of much more complex structured data, at the cost of larger and more complex code for encoding and decoding of messages. This complexity is potentially formidable. If XML is used for all messages, then it will become necessary to lock the XML encoding/decoding libraries in memory. Most libraries which support full XML are many times larger than the entire heartbeat cluster system. Some are fully 10 times larger. Locking such a large piece of code into memory is undesirable. However, a simple subset of XML can be easily parsed in a few dozen kbytes of code.
8. All core software will be written in ’C’ − not C++ or an interpreted language. Heartbeat (which is written in ’C’) is small and lightweight, has been extremely stable with virtually no memory leaks or other stability issues.

Infrastructure Components

There are several components of the infrastructure of this framework which need to be completed before ork on the components can take take place in full force.

XML marshalling / demarshalling

This subsystem takes data structures in memory, translates them into XML for transmission to another process or storing in a file, and conversely takes this XML and translates it back into data structures. The initial implementation is oriented towards the Glib collection data structures, but the design will allow plugging in encoders/decoders for any kind of data structure into the infrastructure.

The decoding (parsing) code will work only with a restricted subset of XML. In some ways, XML is a bit like Perl: "There’s more than one way to express it in XML". Our subset is grammatically simple, yet powerful enough, without so many possible ways of expressing information, yet retaining the ability to represent any kind of data which can be structured hierarchically (as XML requires).

Because the Glib collection data structures allow arbitrary types for their elements, it is necessary to use types which have a type wrapper around them in order to tell whether a list element is a string or an associative array (hash table) or another list. The term "wrapped structures" will be used here for this kind of structure.

The sample text in the paragraphs following illustrate two possible encodings of the data. The current alternative favorite is the data representation used by XML−RPC. Both encodings will be illustrated below.

Below is an example of a Glib *Glist* of strings as encoded into XML:

```
<L>
<LI>This one is first</LI>
<LI>This one is second</LI>
<LI>This is the last string</LI>
</L>
```

Below is an example of a Glib *GHashTable* associative array with strings for keys and values:

```
<AL>
<LI key="linux-ha"/>linux-ha.org/
<LI key="failsafe">
oss.sgi.com/projects/failsafe</LI>
<LI key="kimberlite">
oss.mclinux.com/projects/kimberlite/
</LI>
</AL>
```

Below is an example of a *GHashTable* with a string for one value, and a list as one of its values

```
<AL>
<LI key="tty">/dev/ttyS1</LI>
<LI key="ports">
<L>
<LI>node1</LI>
<LI>node2</LI>
</L>
</LI>
</AL>
```

Below is an example of a Glib *Glist* of strings as encoded into XML−RPC format:

```
<methodCall>
<methodName>subsystem/method
</methodName>
<parmams><param>
<array><value><string>first</string>
</value>
<value/><string>second</string>
</value>
<value/><string>last</string>
</value>
</array>
<param>
</params>
</methodCall>
```

Below is an example of a Glib *GHashTable* associative array with strings for keys and values. This example is encoded using the XML−RPC format:

```
<methodCall>
<methodName>subsystem/method2
</methodName>
<parmams><param><struct>
<member><name>linux-ha</name>
<value><string>linux-
ha.org/</string></value></member>
<member><name>failsafe</name>
<value><string>oss.sgi.com/projects/f
ailsafe</string></value></member>
<member><name>kimberlite"</name>>
<value><string>oss.mclinux.com/projec
ts/kimberlite/</string></value>
</member>
</struct>
</param>
</params>
</methodcall>
```

Below is an example of a Glib *GHashTable* with a string for one value, and a list as one of its values. It is encoded with the XML−RPC encoding.

```
<methodCall>
<methodName>subsystem/method3
</methodName>
```

```
<params><param><struct>
<member><name>tty</name>
<value><string>/dev/ttyS1</string>
<member><name>ports</name>
<value>
<array>
<data>
<value><string>node1</string></value
>
<value><string>node2</string></value
>
</data>
</array>
</struct>
</param>
</params>
</methodCall>
```

It seems likely that this encoding/decoding will be implemented as a loadable module, allowing for some choice in these matters, possibly including non−XML representations.

Module Loading

This subsystem has the job of loading and unloading modules, and registering and unregistering plugins.

The term module and plugin are often used inter−changably, but have distinct meanings in this document.

In this paper, we use the term module to mean a shared library which can be loaded at run time and in−voked. From the point of view of the module loading software, all modules are basically identical, and are treated as identical.

The term plugin is used here to denote a set of ca−pabilities which a module registers with the **PluginHandler** for their particular plugin type. This set of capabilities is the same for every plugin of a given plugin type, but different from the capabilities of a different plugin type. It is these capabilities (or APIs) which define the components of the system.

There is only one built−in plugin type, the **Plugin−Handler** plugin type. It registers **PluginHandlers.** Each **PluginHandler** then registers itself as being the handler for those types of plugins which it is prepared to manage. For example, if a module implements a STONITH plugin, it registers itself with the Plugin−Handler which manages STONITH plugins. If this PluginHandler is not already loaded, it is then loaded automatically. If there is no such PluginHandler, reg−istration of the plugin will fail.

Each loadable plugin exports only those functions which are defined by its API, because the module loading system implements explicit interface export−ing.

Self−Configuration

Many plugins will require configuration for proper operation. Most of these plugins will use the self−configuration API to obtain their configuration infor−mation. This API allows a plugin to present information to GUI to allow the GUI to collect the in−formation and provide it back to the plugin so it can be properly instantiated. Combined with the basic plugin capabilities, powerful sets of self−configuring objects can be added to the system without writing new user interface software.

The self−configuration API provides the following basic capabilities:

- Configuration Metadata query
- Configuration default query.
- Construct object with Configuration
- Current Configuration Query
- Modify object configuration

Each of these different capabilities will be discussed in turn.

Configuration Metadata query

This is the most complex and interesting of the ca−pabilities. The API return result is a set of metadata describing the information which must be provided to configure an object of the type in question.

From the top level view, the metadata is structured as a list of fields, each of which has a variable number of attributes.

For example, any given field will have some of these attributes:

- **fieldname** − the internal name of the field
- **label** − the user−visible label for the field. This is returned according to the requested locale
- **isarray** − true if the field is an array field
- **class** − simple or struct. Simple is the norm, but struct indicates that the field is a repeating field. In this case, the aggregatetype fieldset is an array of field values.

- **basictype** − the lowest−level (or most primitive) type of the field. Examples of basic types are string, boolean, integer, enumeration, etc.
- **specialtype** − the most semantic−rich type for the field
- **length** − number of displayable characters allowed in the field.
- **regex** − a regular expression for validating the field
- **short_text** − a short text explanation of the field, suitable for popping up automatically. It is pro−vided for the requested locale.
- **long_text**− a short text explanation of the field, suitable for bringing up on demand. It is provided for the requested locale.
- **minval** − the minimum allowable value for the field
- **maxval** − the maximum allowable value for the field
- **enumset** − a list of all possible values which the field is allowed to have.
- **fields** − an array of metadata fields making up the fields of a structured value.

As an example, if you have a field named IP which is an IPv4 address, it might have the following attrib−utes:

(fieldname, "IP")
(label, "IP address")
(basictype, "string")
(specialtype, "IPv4")
(length, 15)
(regex, "^[0−9]+\.[0−9]+\.[0−9]+\.[0−9]")
(short_text, "IP address of power switch")
(long_text, "Enter the IP address assigned to the BayTech power switch")

The purpose for having two different field types is to allow configuration programs to support configuring modules which use a newer version of the API spec than the GUI implements. The expectation, is that the GUI will fall back to the primitive type and other fields (such as the length, and regex) if it doesn't rec−ognize the high−level type to allow it to do the best job it can.

In addition to the simpletypes described above, a data element can be an array of simpletypes, or an ar−ray of structures. If an information item is an array, the following attributes will be supplied.:

- minelem − the minimum number of elements which allowed in the array.
- maxelem − the maximum number of elements al−lowed in the array.

If an item has a class of struct, then only the field−name, label, isarray, class, shorttext, longtext, and fields attributes will be used. If the item is also an ar−ray (meaning an array of structures), then the array attributes will also be used.

Remote Procedure Calls

In a cluster, there is a need for a higher−level para−digm for communication than simply sending messages. The applications have many such forms as described by the Application−Level APIs. However, the internals of the cluster itself also need to commu−nicate with each other, and with administration utilities etc.

The framework will use several related forms of re−mote procedure calls to provide this structure. However, we still want all the version−compatibility features that XML provides. We will use the data for−mats described by the XML−RPC specification, and adapt them for use in a cluster. There will be three supported forms of RPC in this cluster.

- "Normal" RPC − a single request sent to a single node in the cluster with a mandatory response with a completion result − as described in the XML−RPC specification. Rather than relying on HTTP for the transport as described in the specification, we will use the cluster basic communication serv−ices for message transport.
- "Multicast" RPC. In this case, a single request is sent to a set of machines in the cluster (or the whole cluster) for each to interpret.
- "Normal" RPC through an separately authenticated external communication channel to a process which can be outside the cluster. This will allow us things like remote administration and monitor−ing capabilities.

For RPC within the cluster, some kinds of calls will not require a mandatory response with a completion result. This is an essential extension to be able to per−form certain kinds of cluster operations (notably leader election).

These few simple extensions will allow cluster components and cluster−aware applications to use a simple RPC paradigm for their operations. This will give a simple conceptual model to use as the basis for implementing higher−level APIs and services.

Local Client−Server API

This local client−server API provides authenticated access to such services as cluster components wish to provide to other local applications. It provides client registration services, authentication services, and integrates with the basic RPC services as described above. It enables building of APIs which allow processes to receive services from cluster components.

Cluster System Components

General API goals

The design of the good APIs is probably the most difficult and underestimated activity in the project. It is the authors' experience, that there are very few really good APIs.

The following goals are common to the design of all the APIs implemented by all the components:

- Implementation hiding − APIs should generally not require any particular method of implementation, or unnecessarily reveal or restrict properties of the components which implement it or other APIs.
- Generality − the API itself should reflect the nature of the concept it embodies, not the restrictions of a particular implementation.
- Simplicity − Each API should be as simple as it can to provide the needed function, but no simpler. This is parallel to Albert Einstein's famous quote about theories: A theory should be as simple as it can be, but no simpler.
- Data (structure) hiding − APIs should not reveal the contents of data structures unnecessarily to their users.
- Object orientation − Most APIs should reflect natural and obvious system concepts and objects and the operations which are useful to perform on them. If a set of interfaces does not, it may not deserve to be elevated to a documented API.
- Decomposition − Sometimes an object should be decomposed into two layers, so that the bottom level layer can be easily understood and implemented. For example, in the system reset feature, two layers help the implementation − one to directly reflect the hardware capabilities, and one to reflect the needs of the cluster system.
- Request identifiers − some APIs functions can easily be changed from being several different requests with the same parameter to a single request with a function code (opcode). This makes the API extensible, and easily added to in the future.
- Binary encoding − APIs should avoid encoding information into bits in the interface. Of course, component implementations are free to do this as much as they wish.
- Language/machine independence − APIs should be independent of OS, language and machine type, and the API itself should not assume a homogeneous cluster configuration.

A few additional thoughts on writing good interfaces can be found in the libtool manual: : http://www.gnu.org/software/libtool/manual.html#Library%20tips

Initialization / Configuration

Basic Communication

The basic communication services will provide guaranteed packet delivery and packet content authentication to cluster modules, along with basic node status services. In the normal course of events, packets are encoded by the encoding plugin, and signed using an authentication plugin before being sent over the wire. The initial communication plugin module will likely be based on the heartbeat code.

Authentication Services

The authentication services which the cluster requires are to be able to digitally sign packets, and also to authenticate them. Several plugins of authentication services will likely be based on heartbeat's authentication code.

Marshalling/Demarshalling services

Packet Encoding (compression/encryption) services

Membership Services

Group Services

Cluster Management

Resource Management

Resource Monitoring

Application−Level APIs

GUI Configuration / Monitoring

Higher−Level cluster APIs

Several different higher−level cluster APIs will be available for cluster−aware applications to use. This will likely include:

- Ordered messaging − guarantee that every member of the cluster receives the messages in a message stream in the same order
- Barrier services − guarantees that every member of the cluster has acknowledged arriving at a barrier before any are allowed to pass it.
- Transactions − guarantees that the entire cluster performs a transaction together or not at all. Variations include 2−phase commit transactions, and n−phase transactions.
- RPC − client−level cluster RPC will also be provided.

Implementation Plan

The current thinking about the general plan of attack for completing the implementation of this framework is as follows:

1. Implement Infrastructure Components
2. Convert STONITH modules to self−configuring plugins.
3. Implement FailSafe (RHINO) interface to self−configuration API
4. Tune and adjust infrastructure as a result of experience gained above. This will act as a proof−of−concept for the framework infrastructure.
5. Define APIs for other components, and implement them − starting with communication and cluster membership.

Implementation Status

At this writing, the module−loading infrastructure and the XML encoding/decoding are well underway, and should be completed before the final copy of this paper is submitted. The design of the self−configuring object system is underway, and should be completed, and implementation begun before the final copy of this paper is submitted, and hopefully a lot more as well.

Future Plans

Talk here about the plan to involve the community, and briefly discuss the project plan highlight the URL, and to do list on the web and how it’s open to everyone, and we hope to get lots of people participating in the project.

Conclusions

Acknowledgments

Colorado School of Mines, the XML students, and lots and lots of other folks... David Brower initially advocated RPC as a basic paradigm and helped clarify the issues surrounding basic communication and the RPC paradigm. Also thanks to Rusty of SGI.

Everything from here to the end is not right yet ;−) This is mostly text from my last year's ALS paper.

To Learn More

The Linux−HA web site can be found at **[Rob01]**. *Heartbeat* can be downloaded (in source or RPM format) from the Linux−HA web site download page at: http://linux−ha.org/download/. Information on subscribing to the various Linux−HA mailing lists can be found on the contact page at: http://linux−ha.org/contact/. The Linux FailSafe project is described in detail in **[Vas00]**.

References

[Milz99] Milz, Harald: "The Linux High Availability HOWTO". http://metalab.unc.edu/pub/linux/ALPHA/linux−ha/High−Availability−HOWTO.html

[Phi98] *In Search of Clusters*, by Gregory F. Pfister, 2nd Edition 1998, Prentice Hall PTR

[Ball00] Ballinger, Rusty, "The RHINO GUI Infrastructure. http://oss.sgi.com/projects/rhino/

[Rob01] Robertson, A. L.,: "The High−Availability Linux Project". http://linux−ha.org/

[Twe00] Tweedie, S. C.,: "Barrier Operations". http://linux−ha.org/PhaseII/WhitePapers/sct/barrier.txt

[Vas00] Vasa, M.,: "The Linux Fail Safe Project". http://oss.sgi.com/projects/failsafe/

Need to add references for XML−RPC, for OMS plugins, Kimberlite, Glib, etc.