Red Storm Systems Management: Topics on Extending Current Capabilities

James H. Laros III, *Sandia National Laboratories* [I]

ABSTRACT: *Systems like the Red Storm system at Sandia Laboratories provide grand challenges for developers of systems software. While many of the challenges associated with systems like Red Storm are experienced on other systems, like commodity clusters, the sheer magnitude of the system magnifies issues that may seem trivial on less complex systems. The value of taking an integrated approach to many of the challenges faced in system software development and implementation on a system like Red Storm is presented in this paper in the form of topical example implementations.*

KEYWORDS: Red Storm, Cray XT3, Reliability Availability Serviceability, RAS, Object Oriented, OO, Systems Management, DDN, Database

1. Introduction

In this paper we will discuss two example implementations that leverage what we call the System Description Language (SDL) to extend some of the current capabilities of the Red Storm [II] system software. These implementations are distinct but complementary to the current system software. These example implementations are presented in an attempt to illustrate the benefits of approaching a problem, in this case systems management, using concepts like the SDL.

We will first provide a brief discussion about the SDL in Section 2 (System Description Language). This section should provide sufficient background about the concepts of the SDL that are leveraged in the specific examples discussed in this paper. Two examples of how these concepts can be applied are outlined in Section 3 (Describing a Red Storm System in Software) and Section 4 (Management of the DDN Subsystem). In Section 3 the discussion focuses on how a very complex system like the Red Storm XT3[III] system at Sandia Labs, can be described in software and stored in a database. The resulting database can form the basis of many and varied capabilities. In Section 4 we will build on the concepts described in Section 3 and add the expression of functional capabilities to the system description. This specific example will demonstrate interaction with the Data Direct Networks (DDN)[IV] controllers which form the basis for the storage system attached to Red Storm. Finally, conclusions and thoughts on future work are provided in Sections 5 and 6.

The examples provided in this paper will hopefully provide a small look at what we feel is a huge opportunity to improve the way systems software is architected for platforms like Red Storm especially in the area of Reliability Availability and Serviceability (RAS) systems.

2. System Description Language

The SDL provides a foundation that can be leveraged to provide a wide range of utility. The SDL provides the capability to produce a database which stores information about software and hardware components, relationships between components, and functional capabilities of components. (In general, when we use the term component we imply either software or hardware component. From the perspective of the SDL there is little or no difference in how each is described.) Relationships between components can be topological, for example, how components are physically connected to each other, or describe the path that components use to communicate with each other. Relationships can be straight forward or very complicated. The functionality of components can be leveraged to great advantage. In this area the SDL must understand a wide variety of the languages that components use to communicate.

We believe that SDL concepts map nicely to object oriented concepts. In practice we have found that using an object oriented approach results in a useful and extensible implementation. The specific implementations described in Sections 3 and 4 are implemented using the Perl [V] programming language. Perl supports most object oriented concepts and has proven to be useful for quickly implementing these concepts. We think it is likely that a higher level object oriented language like C++ is better suited to express some of the more abstract relationships that the SDL is conceptually capable of.

Using object oriented terminology (with some liberty), components map to classes. For each component type in the system a class in the SDL will describe the component type to the level of detail desired. Class attributes can be used to define informational characteristics about a component type or relationships between component types. Notice we have used the term component type rather than simply component. A class is used to describe a type of component. Whether there are 1 or 1000 components of the same type in the system we need only one class to describe that component type. For each component in the system we will instantiate an object of that component type (class) to specifically represent that component. For example, if there are 1000 components of type A we would instantiate 1000 objects of class A to represent each individual component. It is important to note that attributes provide one way to differentiate between instantiated components.

In the same way that information and relationships can be defined in classes, the functionality of a component can be encapsulated in a class. Adding class methods that implement specific functional capabilities of a component type allows objects instantiated based on the class to be capable of the desired functions.

Inheritance is another important object oriented concept that we leverage when implementing the SDL. Many components share common characteristics in the form of information, relationships and functionality. Leveraging inheritance allows us to very naturally express this commonality while not restricting us from differentiating where necessary. We will provide examples of how we leverage inheritance in Sections 3 and 4.

Once the SDL contains the classes necessary to describe a system, objects are instantiated to represent the components in the system and stored in a database. In Section 3 we will provide a real world example of how we leverage these concepts in practice to represent component information and relationships of the Red Storm system at Sandia Labs. In Section 4 we will provide an example that demonstrates the implementation of functional capabilities in addition to the information and relationship characteristics demonstrated in Section 3.

Many of the concepts employed in the SDL were originally conceived of and implemented as part of the Cluster Integration Toolkit (CIT)[VI] project at Sandia Labs. A detailed discussion of CIT can be found in *An Extensible, Portable, Scalable, Cluster Management Software Architecture*[VII]. A more implementation focused view of these concepts can be found in *The Cluster Integration Toolkit - An Extensible, Portable, Scalable Cluster Management Software Implementation* [VIII]. The concepts that are described in these papers have evolved into what we have described as the SDL. A discussion of an architecture that will leverage some of the more expanded capabilities of the SDL concept can be found in *A Software and Hardware Architecture for a Modular, Portable, Extensible Reliability Availability and Serviceability System*[IX].

3. Describing a Red Storm System in Software

By leveraging the concepts discussed in Section 2 we can describe systems of great complexity like the Red Storm system at Sandia Labs. Red Storm is a very large system comprised of many components. Some of the components that we choose to represent are; Nodes (referred to as AMD_single), L0's[1], L1's and Seastars[2]. In the following sections we will describe the class structure used to describe these components, information about them and how relationships can be expressed in the classes. We will also describe how other concepts like grouping can be leveraged to establish relationships between instantiated objects.

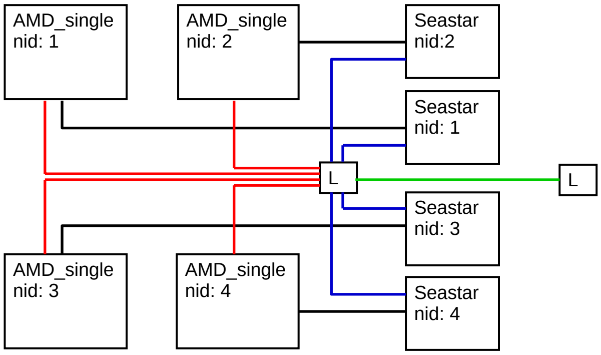

Figure 1 Red Storm Hardware Module (conceptual view)

3.1. Class Hierarchy

Figure 1 depicts a block diagram of some of the components on a Red Storm module. The lines drawn between the components represent relationships that will be established between the components. In this implementation each component type will have a corresponding class. The class name is comprised of a

[1] L0 and L1 in this paper refer to an embedded processor that serves as part of the Red Storm Reliability Availability and Serviceability system.

[2] Seastar is a high speed proprietary Network Interface Controller.

hierarchy of classes that combined provide the final expression of the class and later the objects instantiated from the class. The following is a list of the full Perl class names of the components we will discuss.

Device::Node::Cray::AMD_single[3]
Device::Node::Cray::L0
Device::Node::Cray::L1
Device::Network::Cray::Seastar
Device::SrvrMgmt::Cray::L0
Device::SrvrMgmt::Cray::L1

Notice that each class name begins with the class name **Device**. In this example we will be discussing hardware type components. Most if not all hardware components can be considered devices, therefore, the common base class for all of the components we list is the **Device** class. The second level class name (from the left) is **Node**, **Network** or **SrvrMgmt**. The significance of this class name is to structure the class hierarchy based on the general purpose that the device serves. You might also consider this a categorization of the type of device the component is. Notice that both the **Network** and **SrvrMgmt** class trees contain sub-classes **L0** and **L1**. In this implementation the L0 and L1 devices serve two roles (or can be categorized as two different types of device). Both exist in the system as nodes but they also serve a server management purpose in the overall system. This is one example of the flexibility this concept enables the implementer.

All classes in this device hierarchy contain a class type **Cray**. The primary purpose of this class is to encapsulate concepts that are potentially specific to this vendor's devices. Finally the terminal class in each of the listed classes is most specific to the individual device it is describing. Note that, potentially, these classes can be over-ridden further by sub-classing; for instance to specify a more specific Seastar component or a particular version of Seastar.

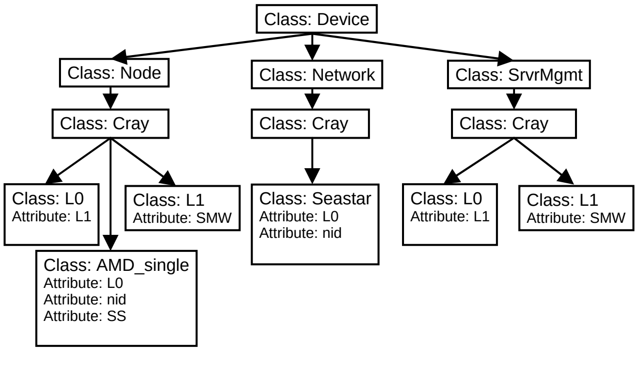

Figure 2 Red Storm Component Class Hierarchy

3.2. Component Information and Relationships

Figure 2 depicts both the inheritance path of the class hierarchy that will be used to represent the Red Storm system and attributes specific to terminal classes in the hierarchy. (Note: only a few of the many attributes are shown and will be discussed) The **AMD_single** class inherits from the **Cray**, **Node** and **Device** classes (traversing the class hierarchy from bottom to top). The **AMD_single** class represents a processor socket (or a node if you would rather, if the node is a single core socket node) on the Red Storm system. Figure 1 depicts a Red Storm compute module with four single core sockets. Notice the name selected for the class. While the current system has single core sockets it is likely that in the future this or other XT3 systems will contain dual core sockets (and before long, quad core). When this happens we can simply create a class to represent the dual core socket components called **AMD_dual**. This new class can potentially be based on the **AMD_single** class and can be designed to represent the differences between a single and dual core socket component. It is important to note that using the object oriented approach allows us great flexibility in how these classes are defined and how we construct an inheritance tree to express the system.

An example of an informational attribute is the *nid*[4] (node id) attribute defined in the **AMD_single** class (Figure 2). This attribute is used to store information about a specific object that is instantiated based on this class. (Other attributes can exist to specify information specific to an entire class that is shared by all objects instantiated from the class.) An important piece of information about a processor on a Red Storm system from the run-time perspective is the *nid* number. This number is used, for example, for allocating the processors that will be assigned to an application.

The **AMD_single** class also specifies an attribute *L0*. This attribute contains the name of an instantiated object of type **Device::Node::Cray::L0** representing the relationship between the L0 processor on the module and the processor socket. In fact the same L0 object will be specified in every object of type AMD_single that is on the same physical module (board). Figure 1 depicts this relationship. Notice that each AMD_single component is connected to the L0 component by a red line. This line depicts the relationship that is established between these objects based on the attribute definition in each of the instantiated AMD_single objects.

The final attribute depicted in Figure 2 associated with the **AMD_single** class is the *SS* (Seastar) attribute. This attribute enables an object that is instantiated based on the **AMD_single** class to know which Seastar component it is associated with. Figure 1 depicts four Seastar components on a Red Storm compute module.

[3] Full or partial class names appear in bold text.

[4] Attribute names will appear in italics.

Each AMD_single component has a dedicated Seastar component. This relationship can be established by defining the *SS* attribute of an AMD_single object to the specific Seastar object that provides its network interface. Figure 1 depicts the relationship this attribute establishes with black lines between each of the AMD_single components and a Seastar component.

The **Seastar** class also contains a *nid* attribute. This attribute definition depicts another way to represent information while also representing a relationship between instantiated objects. Notice that in Figure 1 the AMD_single component whose *nid* attribute is set to one is related to the Seastar component whose *nid* attribute is set to one. The black lines represent the relationships formed as described previously by defining the *SS* attribute. This relationship can also be inferred by the common attribute definition in each object. This concept can be leveraged as necessary to form many simple or complex types of relationships.

The **Seastar** class, like the **AMD_single** class, contains an *L0* attribute. This attribute serves the same purpose in the **Seastar** class as it does in the **AMD_single** class. Notice that this is an example of forming a many to one relationship. On a Red Storm module the L0 is an important management component. Specifying which L0 is responsible for the management of components is a valuable piece of information that can be leveraged for many purposes, such as a RAS [IX] system.

The management hierarchy is defined further by the attributes specified in the **L0** and **L1** classes. The Red Storm management system is hierarchical in topology. In brief, each Red Storm module has an L0 (as shown in Figure 1). Each Red Storm cabinet contains one L1. The L0's on each of the 24 modules in a Red Storm cabinet communicate with the L1 in the cabinet. In turn the L1's in each Red Storm cabinet communicate with the System Management Workstation (SMW) at the top of the RAS hierarchy. The *L1* attribute in the **L0** class defines the relationship between the L0 and L1, and the *SMW* attribute in the **L1** class defines the relationship between the L1 and the SMW. Forward and backward relationships in a topology can be established using these same concepts.

By choosing attributes that represent information and relationships that are meaningful for a particular system, a system description can be generated and stored in a persistent manner that can be leveraged to provide great utility.

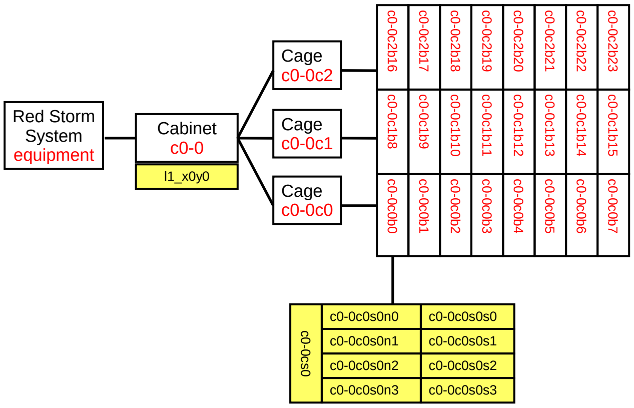

Figure 3 Red Storm Component Collections and Objects

3.3. The Red Storm Database

We have discussed how we use the SDL to describe components of the Red Storm system (based on classes), information about these components and relationships between components. In producing the Red Storm database we have another opportunity to express relationships between components (instantiated objects). Using the Red Storm naming convention we can leverage a concept we refer to as collections, or groups, to represent the physical topology of the system. (Other topological relationships, like network topologies, that exist in a system can be established using these concepts or in combination with other concepts previously discussed) Figure 3 depicts a portion of the physical topology of a Red Storm system. Each block in the diagram represents a more granular level of the physical topology. At the top (left in Figure 3) is the entire Red Storm system. In the database, the system is represented by a collection called equipment. This name is only meaningful in the sense that it was chosen to generically represent any system we wish to describe. We have total flexibility in naming collections and could have chosen to name the collection redstorm for example. Note that the collection name equipment is in red in Figure 3. Each name in red represents a collection in the database. The boxes in yellow represent objects instantiated based on the classes that we discussed previously.

The Red Storm system (equipment) is made up of many cabinets. Figure 3 represents a single cabinet of the Red Storm system labeled c0-0. This name represents the physical position of the cabinet in the Red Storm system (c or cabinet in x position 0, y position 0). Note that all of the names have an x and y coordinate in their name. These names were chosen by Cray but the SDL can accommodate any naming scheme that is chosen. To view this grouping on an actual system, commands are provided to display collections or objects in the database. Since equipment is a collection in the database we can use the collection_mgr command to view what is contained in the equipment collection.

```
# collection_mgr equipment
c0-0
c0-1
c0-2
    .
    .
```

The collection_mgr command will list every object and collection that is in the specified collection which in this case would be every cabinet in the Red Storm system. (For space considerations we have truncated the actual output.) Note that the equipment collection contains, among others, a collection named c0-0 or cabinet located in x position 0 and y position 0. If we execute the collection_mgr command on this collection we will see what it contains.

```
# collection_mgr c0-0
c0-0c0
c0-0c1
c0-0c2
l1_x0y0
l1_x0y0-mgt
```

Notice that the cabinet c0-0 contains three cage names, c0-0c0, c0-0c1 and c0-0c2, representing the three cages in a Red Storm cabinet. These are also collections as depicted in Figure 3. There are two actual objects in the cabinet collection, l1_x0y0 and l1_x0y0-mgt. Both objects represent the same physical device in the cabinet. Recall from our previous discussion that we have the flexibility to describe different purposes for the same device. If we examine these two objects more closely we will see what the differences between the objects are:

```
# device_mgr l1_x0y0
name => l1_x0y0
interface => 0
        nic => 0
        name => eth0
        address => 10.1.100.100
        net_mask => 255.255.0.0
        boot_if => 1
        hostname => l1_x0y0
        is_primary => 1
role => RAS
vmname => Management
x_pos => 0
y_pos => 0
isa => Device::Node::Cray::L1
```

```
# device_mgr l1_x0y0-mgt
name => l1_x0y0-mgt
interface => 0
        nic => 0
        name => eth0
        address => 10.1.100.100
        net_mask => 255.255.0.0
        boot_if => 1
        hostname => l1_x0y0-mgt
        is_primary => 1
role => RAS
vmname => Management
x_pos => 0
y_pos => 0
isa => Device::SrvrMgmt::Cray::L1
```

Note that there are many attributes defined in each object that we have not discussed. Some will be mentioned later but some are beyond the scope of this paper. Suffice it to say that each attribute serves a purpose in describing the object that is intended to be leveraged by the user for some reason.

If we examine the output from the two device_mgr commands we notice more similarities than differences. Recall that both of these objects represent the same physical device. The class name for each object is different. Additionally, each object requires a unique name to be stored in the database (reflected by the *name* attribute). The *isa* attribute of the object shows the full class name of each object. The l1_x0y0 object is of type **Device::Node::Cray::L1** while the l1_x0y0-mgt object is of type **Device::SrvrMgmt::Cray::L1**. Even though we see little difference in the information defined for each object, the class types could encapsulate different functional capabilities that are not shown in this output. In this sense the objects, while representing the same device, could function very differently from each other and express different capabilities.

We can continue to drill further down into the hierarchy of collections using the collection_mgr command on the cage names displayed from exploding the cabinet collection. In this example we will expand the cage 0 collection, c0-0c0, which is contained in the cabinet c0-0 collection.

```
# collection_mgr c0-0c0
c0-0c0b0
c0-0c0b1
c0-0c0b2
c0-0c0b3
c0-0c0b4
c0-0c0b5
c0-0c0b6
c0-0c0b7
```

From this output we can see that cage 0 of cabinet 0 contains eight boards number from 0-7, which represent the eight boards in a Red Storm cage. These board designations are also collections intended to provide a more granular grouping of components just like the previous collections. We can again drill further down into this hierarchy of collections.

```
# collection_mgr c0-0c0b0
c0-0c0s0n0
c0-0c0s0n1
c0-0c0s0n2
c0-0c0s0n3
c0-0c0s0s0
c0-0c0s0s1
c0-0c0s0s2
c0-0c0s0s3
c0-0c0s0
c0-0c0s0-mgt
```

As shown in Figure 3 all of the entries in a board collection are actual devices. The c0-0c0s0 and c0-0c0s0-mgt objects represent the L0 on board c0-0c0b0. The reason for the two objects representing the same physical device is equivalent to the example given for the L1 device at the cabinet level. In addition to the L0 object there are four node objects and four Seastar objects defined for each compute module in the system. These are each represented by an object in the database. The following device_mgr command shows some of the attributes that are associated with node 0 on this board.

```
# device_mgr c0-0c0s0n0
name => c0-0c0s0n0
l0 => c0-0c0s0
power => c0-0c0s0-mgt
power_port => 0
cage => 0
slot => 0
x_pos => 0
y_pos => 0
nid => 0
role => compute
vmname => catamount
leader => c0-0c0s0
SS => c0-0c0s0s0
isa => Device::Node::Cray::AMD_single
```

As mentioned previously there are many more attributes defined for this object than we will be able to describe. Notice however some of the attributes that we mentioned previously define relationships between objects. This node object specifies which L0 it is related to on the module. In addition, it specifies the Seastar object that it will use for communication. We can see what class the object is specified by the *isa* attribute and what node id is assigned by the *nid* attribute. Many of the other attributes are likely self explanatory including the *role* attribute that specifies that this node is a compute node. Other valid roles could include IO, or RAS as we saw previously when we expanded an L1 type object. An example of a more software related attribute is the *vmname* attribute. In this case *vmname* is set to catamount specifying that this compute node is running the catamount kernel. The *vmname* attribute for an IO node could be set to Linux since the Linux kernel is typically used on the IO partition. We could even include the *vmname* attribute in the class definition for L0's and L1's and set the object attribute to embedded_linux or even more specifically the type of embedded Linux that is used. There is virtually no limit to the information or relationships that this methodology can represent.

Hopefully, it is obvious by this point that many benefits can be realized by storing this wealth of information and relationships between components in a system. Some simple examples are generating host tables or converting between node names and node id's (*nid*). These are simple examples and generating a complex database like the one we have discussed seems to be a bit drastic simply to generate host files. But consider the challenges and amount of information necessary for a RAS system. Now consider how much of the same information a scheduling system needs. If we consider the sheer number of configuration files and separate definitions of information and complex relationships that exist in the software necessary to run a system like Red Storm the task of establishing a central storage of system information like we have described becomes trivial in comparison. We hope that using concepts like the SDL described in this paper to generate a central repository of information will become a cornerstone of future RAS systems and provide benefit to other system software components like schedulers and runtime systems.

4. Management of the DDN Subsystem

In this section we will discuss a targeted implementation of the concepts outlined in previous sections for the purpose of managing the DDN subsystem which provides the storage for the Red Storm system. In this implementation we will not only leverage the informational and relational aspects that the SDL can express but also implement the functional capabilities of the DDN controllers. In this section we will build on the concepts presented in previous sections and focus on the implementation of functional capabilities.

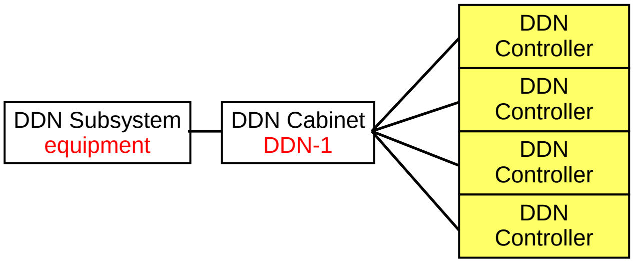

Figure 4 DDN Component Collections and Objects

4.1. The SDL and Class Hierarchy for the DDN subsystem

The DDN subsystem, like the Red Storm system itself, can physically be viewed in a hierarchical manner. Figure 4 represents the hierarchy that we have implemented (from left to right) to represent the physical configuration of the DDN subsystem. Each box in the diagram depicts a collection or object. White boxes are collections with the collection name in red. Yellow boxes are objects that represent hardware components. As with the previous Red Storm system example, the top (left most) collection which in this case represents the DDN subsystem is called equipment. The equipment collection itself contains collections representing each of the cabinets in the DDN subsystem. Figure 4 depicts the cabinet collection DDN-1. Each cabinet collection in this implementation contains four DDN controllers. Figure 4 depicts the four DDN controller objects in yellow that are contained in the DDN-1 cabinet collection.

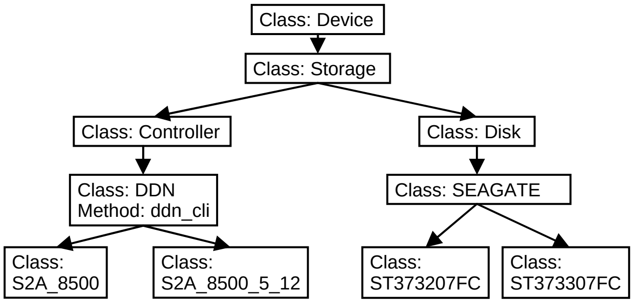

Figure 5 DDN Component Class Hierarchy

Figure 5 outlines the portion of the class hierarchy that we implemented to represent the DDN subsystem. We have applied the same principles discussed in previous sections such as the use of attributes to express information about components and relationships between components. In this implementation we will add the use of methods to express the functional capabilities of the DDN controllers. Our decision process of where to place methods in the class hierarchy is basically the same as the decision of where to place attributes. Our goals are to maximize code sharing and leverage inheritance as much as possible. It is important to note that these decisions are important but we are not permanently locked into our first instincts. The object oriented concepts used in the SDL allow us great flexibility to relocate the capabilities that are implemented into new or existing classes as the system, or how we view the system, evolves.

4.2. The DDN Command Line Interface

The DDN controllers provide a Command Line Interface (CLI) to the user. Typically the CLI is accessed by the user via a Telnet session. Once the user is logged into a DDN controller they have access to a large number of capabilities. For the purposes of the paper we will categorize the DDN CLI commands into two categories, 1st and 2nd level commands. By our definition a 1st level command is the execution of a DDN CLI command without any associated parameters. A 2nd level command is defined as the use of a 1st level command with additional parameters. For example the DDN CLI provides a command named “disk”. By our definition the use of this command by itself is categorized as a 1st level command. If the “disk” command is used with the optional parameter list, the command “disk list” would be categorized as a 2nd level command (by our definition). The consistency of the DDN CLI allowed us to abstract nearly all of the DDN CLI with a single method named ddn_cli located in the DDN class (see Figure 5). The methods implemented in the SDL can be loosely thought of as device drivers. The methods are intended to implement the capabilities of the component (in the case of a hardware component). It should be noted that these methods can only implement what a component is capable of. This does not mean, however, that we cannot develop methods that combine capabilities of a component in creative ways to allow new capabilities. As long as the new capabilities are built on top of basic functionality there are no limitations (See Section 4.4 Custom Commands).

4.3. Our Command Line Interface

We first implement methods in the appropriate classes to provide basic component functionality. To make use of this functionality a user interface must be provided that will accept input from the user and execute the appropriate methods to accomplish the desired task. We will briefly describe the user interface that we have developed in the form of a series of commands that represent the 1st and 2nd level DDN CLI.

The 1st level DDN CLI commands in our implementation are executed with a command named ddn. As with all of the commands we will describe adding the flag –help will result in a description of the available capabilities of the command. The ddn command abstracts the 1st level DDN CLI commands using the following command syntax.

```
# ddn <flag> <component or collection>
```

The flag is simply the 1st level DDN CLI command proceeded by two dashes (--). For example if you want to execute the “faults” command you would use the flag –faults. The component or collection portion of the command allows the user to specify a single component (object) which in this implementation would be a single DDN controller, or a collection of components. Note that the user may also provide a list comprised of multiple components and or collections. The following examples may help illustrate this concept.

```
# ddn –faults BLK-C1-U-1
```

This command would execute the "faults" command on the DDN controller BLK-C1-U-1 and display the output as if the user went through the process of logging into this controller and executing the "faults" command.

```
# ddn –faults DDN-1
```

This command would execute the "faults" command on each of the DDN controllers in the collection DDN-1 (BLK-C1-U-1, BLK-C1-U-2, BLK-C1-L-1, BLK-C1-L-2 from Figure 4) and display the output (Note that this is done in parallel, see Section 4.5). This applies to every 1st level DDN CLI command that is implemented.

The 2nd level DDN CLI commands are executed by combining the prefix ddn_ with the 1st level DDN CLI command, followed by a flag that indicates additional parameters to be used with the command. The additional parameters conform to the DDN CLI format. Examples will provide a better description of this. Suppose the user wants to execute the DDN CLI command "disk list" on controller BLK-C1-U-1. The user would execute the following command.

```
# ddn_disk –list BLK-C1-U-1
```

In this command the 1st level DDN CLI command "disk" is prepended by ddn_ to form the ddn_disk command. The DDN CLI command "disk" has various parameters that can be supplied along with the "disk" command (list is one of them). The resulting command ddn_disk –list will effectively execute the DDN CLI command "disk list" on controller BLK-C1-U-1. This approach was taken in an attempt to make the interface we present familiar to users of the DDN CLI. Note that users do not have to know every DDN CLI command available. Executing the command ddn –help will display a list of the 1st level DDN CLI commands that have been implemented. The user can then construct the 2nd level command by simply prepending ddn_ and adding the –help flag to view the available options for that 2nd level command.

It should be noted that all commands that we have discussed and will discuss are actually implemented in a single executable. The commands ddn, ddn_disk and every other 2nd level command interface are actually links to the ddn_cli command (not meant to be called natively). This command acts much like the popular BusyBox [X] in that how it executes is dependent on what command it is called by.

4.4. Custom Commands

Once we implemented the basic capabilities of the DDN CLI we used these capabilities as building blocks to create additional commands based on user requirements. This is probably best illustrated with a simple example. The DDN CLI "faults" command when executed on a DDN controller will output about seven lines for a controller without faults and quite a few more if the controller has faults. Suppose you are only interested in a response if the controller has an error. By building on the basic DDN "faults" command a method can be constructed to examine the output and only notify the user if a fault is detected. The following command will return the controller name if it determines that there is a fault present. If no fault is detected nothing is returned.

```
# ddn –check_faults BLK-C1-U-1
```

Another value of a command like this is that it can be combined with other commands. Since the command listed will only return the controller name of a controller that has a fault we can pipe the output of this command into a subsequent command.

```
# ddn –check_faults DDN-1 | xargs ddn –faults
```

This command will query all of the controllers in the DDN-1 collection and return only those controllers that show a fault. By piping the output into the second ddn command we will display details about the fault for only the controller that had a fault. It is hopefully not a common situation that a controller displays faults. By using this command we have the ability to execute a single command that scans all of the controllers in the system for faults and displays details for any controller that is in a faulted state. By using a simple abstraction of a basic capability we can now check the health of a large number of controllers with a single command where previously we would be required to log into each controller to accomplish the same task. Note that all custom commands are currently implemented using the ddn command combined with a custom command flag such as –check_faults.

More complex capabilities can be implemented in much the same way. Each DDN controller has attached storage or disks. As you can see in Figure 5 we have included classes to represent disks in the DDN class hierarchy. For a large system like the DDN subsystem on Red Storm it would be very tedious and error prone to account for every disk during the database creation process. Since each DDN controller knows what disks are attached to it, this information can be discovered and entered into the database automatically rather than using a more manual process. This is accomplished with a custom method that leverages the "disk list" DDN CLI command. The "disk list" command returns a wealth of information about each disk that is attached to a DDN controller. We can simply develop a method that parses this output and instantiates an object to represent each disk component attached to that controller. By using this method for each controller in the DDN subsystem our database will contain an object representing every disk component in the system. The following command is the user interface to this capability.

```
# ddn –populate_disks <controller or component>
```

This command will populate the database with the disks that are attached to the controller specified. Note that if a collection of controllers is specified the process will be accomplished for every disk on every controller in the collection. By extension, every disk in the entire system can be discovered using a single command by listing a collection that contains every controller in the system.

While we have only discussed two examples of what can be accomplished we hope it is evident that any capability the end user desires can be implemented as long as it is based on the native functionality provided by the DDN controller.

4.5. Applying Parallelism

The benefits of the concepts presented in this paper become much more evident when dealing with large numbers of components. In the Red Storm example we hope to have demonstrated value in producing a central storage of information about the huge number of components in that system that can be leveraged for many purposes. In the DDN example we additionally implemented functional interaction with components. To be effective for large numbers of components we must apply parallelism. Consider the process of executing a single DDN CLI command without using any of the capabilities that we are providing. A user must telnet to a DDN controller, log in with a user name and password, execute the command and view the results. While this might be adequate for a system with a single or very few controllers it can quickly become very time-consuming for a large installation. So far we have shown that we can save the user the time it takes to log into the controller by automating the process as in the previous example of checking to see if a device has any faults.

```
# ddn –faults BLK-C1-U-1
```

By providing this interface we save the user a bit of trouble and a few seconds of time depending on how fast they can type. Recall that the user has the option of specifying a collection of controllers on the command line. If the collection only contains a few controllers and the command executes fairly quickly this may still not present an issue. But what if, for example, one of the controllers in the list is unavailable and the timeout time for the action that the user required is 30 seconds. If we are executing this process in a serial manner we have just added 30 seconds to the total execution time. Again for a few controllers this might not be a problem. What if your installation has a very large number of controllers? Even if it only takes five seconds for each operation if your installation has 88 controllers (like the Red Storm system) the command would take 440 seconds or over seven minutes. While this is probably faster than a user can log into each of 88 controllers and perform the same task it is in our opinion unacceptable. To remedy this we can simply apply some parallelism to the user interface so that the required command is executed on each controller in the specified collection at the same time. More about performance will be reported in Section 4.6.

4.6. Some Examples of the Resulting Capabilities

The DDN subsystem that supports the unclassified side of the Red Storm system is comprised of 22 cabinets each with four controllers (88 controllers total). Perhaps the best display of some of the capabilities this implementation provides is to show some timing examples from our production system. To add some perspective to these results we measured, with reasonable accuracy, the time required to telnet into a controller, execute the command “disk status” and log out. The total time for this operation was just over six seconds. Your results may vary. In contrast here is the result of using our command to accomplish the same operation.

```
# time ddn_disk –status BLK-C1-U-1
real    0m1.315s
user    0m0.190s
sys     0m0.000s
```

While this is relatively a very significant difference the real benefit comes, as mentioned previously, when dealing with large numbers of components. On the production system we have added a collection called all_controllers that contains the name of every controller in the system. In the following example we will use the –check_connection option of the ddn command which logs into the specified controller(s) verifies that it has reached the correct destination and logs out. The first command will time this operation for a single controller. The second command will time the operation for all 88 controllers in the system using the all_controllers collection.

```
# time ddn –check_connection BLK-C1-U-1
real    0m1.290s
user    0m0.190s
sys     0m0.010s

# time ddn –check_connection all_controllers
real    0m2.420s
user    0m0.910s
sys     0m0.430s
```

As you can see from these results we can perform this operation on all 88 controllers in the system in less time than it takes to log into a single controller manually.

One of the typical operations that administrators at our site accomplish is executing the DDN CLI "stats" command. This command produces a moderate amount of output. The following two commands time how long it takes to execute the "stats" command on a single controller followed by all 88 controllers.

```
# time ddn –stats BLK-C1-U-1
real    0m1.291s
user    0m0.190s
sys     0m0.010s

# time ddn –stats all_controllers
real    0m5.447s
user    0m1.130s
sys     0m0.590s
```

Even using a command that produces significant output, the time to execute a command on all of the controllers in the system is very small. We should note that when a command is executed on multiple components the output for that component is prepended by the controller name. This allows us to redirect large amounts of output to a file and quickly find the controller in question while viewing the file with an editor or using your favorite UNIX utility to extract the information you desire. We should also note that the times listed are including the time to return and display all output from the commands. Each command was executed multiple times to verify that the results were consistent and reproducible.

5. Conclusion

We hope that the examples provided in this paper provide enough information for the reader to imagine other uses for these concepts. The implementations described originated as proof of concept exercises but quickly proved to be useful on a production system. All of the software used in these examples is freely available[VI]. Please contact the author for further information.

6. Future Work

One of the values of expressing information as we have described is that it can be abstracted in different ways to serve the many varying needs of the end user. The DDN example abstracted the administrative interface to the DDN controllers. This interface was targeted at system administrators responsible for the task of managing the DDN subsystem itself. Much of the same information could be leveraged to provide important configuration information to someone interested in supporting Lustre[XI]. Relationship information, like which nodes are connected to which DDN controller or what disks are combined to form a file-system, could be extremely valuable. By combining the information described in the examples into a single database all of the necessary information could be stored and retrieved in numerous creative ways. We are in the process of providing an abstraction of this combined information for the I/O team at Sandia Labs.

Many of the concepts discussed in this paper are part of an ongoing research project at Sandia Labs investigating RAS software architectures. Implementations such as these provide valuable proof of concept exercises.

Acknowledgments

We would like to thank the Computer Science Research Foundation (CSRF)[XII] at Sandia National Laboratories for funding much of the work that lead to the development of the concepts described in this paper.

About the Author

James H. Laros III is a Principle Member of the Technical Staff at Sandia National Laboratories, currently involved in research and development of Reliability Availability and Serviceability (RAS) software architectures.

References

I. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. Contact: jhlaros@sandia.gov

II. RedStorm http://www.cs.sandia.gov/platforms/RedStorm.html

III. Cray XT3 http://www.cray.com/products/xt3/index.html

IV. Data Direct Networks DDN http://www.datadirectnet.com/company.html

V. Perl – http://www.cpan.org

VI. Cluster Integration Toolkit CIT - http://www.cs.sandia.gov/CIT

VII. An Extensible, Portable, Scalable, Cluster Management Software Architecture. James H. Laros III, Lee Ward, Nathan W. Dauchy, Ron Brightwell, Trammell Hudson, Ruth Klundt, Proceedings of the 2002 IEEE International Conference on Cluster Computing, 23-26 Sept. 2002. Copyright 2002 IEEE

VIII. The Cluster Integration Toolkit - An Extensible, Portable, Scalable Cluster Management Software Implementation. James H. Laros III, Lee Ward, Nathan W. Dauchy, James Vasak, Ruth Klundt, Glen Laguna, Marcus Epperson, Jon R. Stearley Proceedings of the 1st Cluster World Conference and Expo 23-26 June, 2003.

IX. A Software and Hardware Architecture for a Modular, Portable, Extensible, Reliability Availability and

Serviceability System - James H. Laros III, presented at the 2nd Workshop on HIgh Performance Computing Reliability Issues in conjunction with the 12th International Symposium on High Performance Computer Architecture, February 11th, 2006.

X. BusyBox – http://www.busybox.net

XI. Lustre – http://www.clusterfs.com

XII. CSRF - Computer Science Research Institute http://www.cs.sandia.gov/CSRI/