A Software and Hardware Architecture for a Modular, Portable, Extensible Reliability Availability and Serviceability System

James H. Laros III, Sandia National Laboratories (USA) [1]

***Abstract*— This paper provides a very high level overview of a software and hardware architecture for a Reliability Availability and Serviceability system. One of the primary goals of this architecture is portability. The design of the architecture is intentionally modular to provide the extensibility necessary to allow the portions of the system that are not directly portable to be easily added or modified. This architecture is designed for use on systems ranging from commodity clusters to custom Massively Parallel Processing systems.**

I. Introduction

Reliability Availability and Serviceability (RAS) systems were commonly provided by vendors on mainframe class systems. Today RAS systems are still found on very high end custom systems like Red Storm [2] and BlueGene/L [3]. While these systems provide analogous functionality to the architecture proposed in this paper they lack both the portability and extensibility to allow them to be useful on other platforms. Commodity cluster systems have traditionally provided only a subset of the RAS capabilities found on the aforementioned custom architectures. Additionally when provided, these systems, more accurately categorized as management systems, are typically not portable. The purpose of this paper is to provide a very high level overview of a software and hardware architecture for a RAS system that is designed to be used on both commodity cluster systems and high end custom platforms.

Extensibility is key in this design not only for portability reasons, but to allow even a targeted implementation to be easily updated or modified. Systems evolve, sometimes rapidly at first, over their life cycle. This evolution can involve the addition of new hardware or software components, or simply a change in the monitoring or management approach in an effort to increase the RAS of the system. The extensibility required to allow the evolution of a targeted implementation is essentially the same as is required to achieve portability to new systems.

We will discuss a hardware architecture in section II that while not required can be leveraged to great benefit by the software design. The foundation of the software architecture is the System Description Language (SDL) discussed in section III. The section on Databases (section IV) describes two uses for information storage for this architecture. Some aspects of how the RAS components communicate are discussed in section V. The Device Interface discussion (section VI) briefly covers how the extensibility of the SDL is exploited to allow the majority of the RAS system software to be portable while also enabling it to quickly accommodate systems with new component interfaces. We present some of the ongoing work being accomplished at our site, in this and related areas, and some of our future plans in section VII. We conclude (section VIII) and recognize some contributions in section IX.

II. Hardware

The software architecture described in this paper is intended to have few if any specific hardware requirements, therefore the hardware architecture described in this section discusses general concepts. Figure 1 illustrates concepts that will be discussed here and in sections that follow. The hardware architecture will be described in hierarchical terms: the Top-Level node at the top of a downward growing tree.

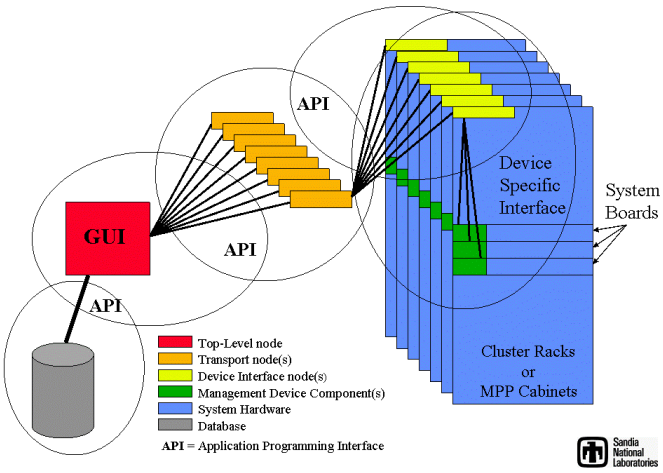

Fig. 1. RAS Hardware and Software Architecture

The Top-Level node (depicted in red) is the primary administrative interface of the RAS system. An administrator has the capability to control and monitor the entire system from this node. This node will also host, or interface to, the database system (the database will be covered in section IV). A properly designed hardware architecture should only require a single Top-Level node, however a second Top-Level node may be desired for fail-over in case of hardware malfunction.

The next level, or levels, of the hardware architecture host the transport mechanism of the RAS system. This level may be absent depending on the size of the system. The transport hardware required (depicted in orange) will directly correspond to the size of the system and the capability of the transport hardware itself. Proper sizing of the transport hardware is critical to the proper functionality (and scalability) of the RAS system. The transport software design will be described in more detail in section V.

The level below the transport hardware, or below the Top-Level node in the absence of dedicated transport hardware, is the device interface hardware (depicted in yellow). While not a requirement, we envision this hardware will be located in the racks containing the system hardware. This hardware hosts the daemons that will interface with the actual devices that provide the command and monitoring capabilities and any other device included in the system. These devices also serve as an interface to transport devices or the Top-Level node. Like the transport devices the amount of device interface hardware is dependent on the size of the system and the capability of the device interface hardware itself.

The management device components (depicted in green) are system specific devices that are leveraged by the RAS system to control or monitor the system. These devices range widely in capability. It is important to point out that the proposed (or any) RAS system can only leverage the capabilities that the underlying devices (or software components) provide. On a commodity cluster system, for example, the management device components may be Intelligent Platform Management Interface (IPMI)[4] devices. On a custom Massively Parallel Processing (MPP) system they may be specially designed embedded systems with a proprietary interface. In either case, the RAS system must be able to communicate with them in their language to exploit their capabilities.

It is important to note that the RAS system may interact with any or all components in the system. It is of critical importance to design the system, including the RAS hardware, so that the RAS functionality has little or no impact on the primary purpose of the system. For example, if a layer of transport hardware is not provided the Top-Level node may be required to communicate directly with too many device interface nodes. This could cause the Top-Level node to be unresponsive to a critical request that could effect the functionality of the system. Likewise, if a transport layer is provided but is improperly sized, based on the size of the system and the capability of the transport hardware to accommodate the number of device interface nodes, the transport layer could delay the transmission of the same type of critical request. This delay could also adversely effect the functionality of the system. The number of device interface nodes can also be a potential bottleneck if too few are configured. For these reasons the RAS software design allows for a great level of flexibility which accommodates adding, or removing, hardware components to tune capabilities in the event of improper initial sizing or system growth.

III. Foundation

The foundation of the software architecture is a System Description Language (SDL) that will allow the RAS system to have detailed and specific knowledge of the underlying hardware and software components of the system, their roles, responsibilities, and relationships to other components (literally any information about the system or individual components). The SDL must be extensible. While the goal of the software architecture is to be portable, it is not possible to write device specific interfaces for devices that possibly have not yet been conceived of. While we will not be covering implementation details in general, this portion of the software architecture will likely be implemented in an Object Oriented Language. Regardless of the implementation language, we will use, with some liberty, object oriented terms to describe the structure of the SDL. Figure 2 pictorially represents some of the concepts that will be discussed. In the following paragraphs, for simplicity, we will discuss characteristics and functionality of hardware devices. Keep in mind that software components can also be described in the SDL.

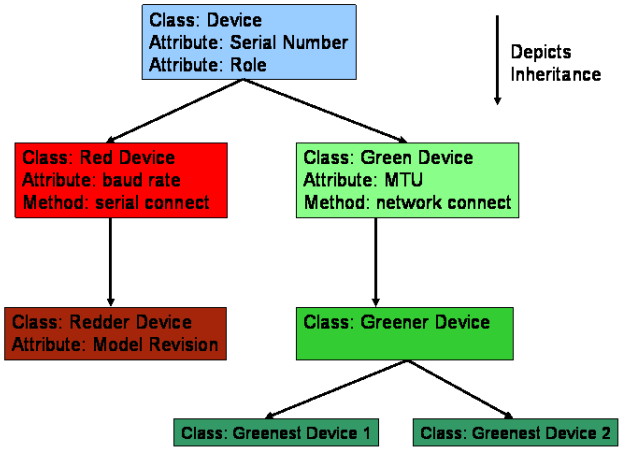

Fig. 2. SDL Class Hierarchy

Consider, for example, that systems are made up of devices. For the sake of this discussion we will define a device as any component of the system that you can touch (in practice this scheme can accommodate anything that can be described therefore is almost unlimited). There are many characteristics that all devices have; for instance most devices have a unique serial number. Since all devices share this characteristic, the Device class (pictured in blue in Figure 2), in our example the most generic of the classes, would define an attribute appropriate for assigning a serial number to objects of the Device class. While we feel it is important not to make assumptions, (in general, assumptions weaken portability) some concepts, especially those foundational to the software architecture, can be represented as attributes in generic classes like the Device class without adversely effecting portability. The concept of role, for example, can be used to describe the purpose that the device serves in the system. Abstract concepts, like role, do not impose requirements on the devices themselves and can therefore be used without fear of disrupting the portability of the system. If for an object of class Device the concept of role does not make sense, the role attribute for that object can simply be assigned a null value and dealt with appropriately by the RAS system.

In addition to simple descriptive attributes, a class will typically contain methods that can be used to exploit the specific capabilities of a device or a class of devices. (Recall that we mentioned that the RAS system will only be able to exploit capabilities of the underlying hardware.) If an interface

common to all devices exists it would be appropriate to include a method implementing the interface in the generic Device class. Typically, however, these types of methods are more commonly implemented in subclasses that describe groups of devices (like the Red and Green Device classes), or in subclasses specific to individual devices (the Redder, Greenest 1 and 2 Device classes).

As illustrated in Figure 2 (Red and Green Device classes), subclasses of the Device class can be created that include attributes and methods of devices that are more specific to the subclass but share (inherit) characteristics of the parent (Device class). Likewise, additional subclasses can be added to even more specifically describe categories of devices (Greener Device class). The final subclass descriptions should be the most specific. The Redder Device class illustrates an example where the implementor had the need to keep track of the model revision for this type of device and felt that it was not generally applicable enough to be included in the Red Device class. Even if no unique attribute or capability is defined in the class, terminal classes serve as a way to differentiate between nearly identical types of devices, if only by class name (Greenest 1 and 2 Device classes). Terminal classes also have the benefit of acting as a placeholder in the event that subtle differences emerge between very similar devices that share a parent class.

By taking this approach we are creating a foundation that is extensible and can evolve. With the addition of each new class the SDL becomes more expressive and capable of supporting systems with an increasingly large variety of devices. By using object oriented concepts we also leverage the commonality of these devices without sacrificing the granularity to express subtle differences. As the SDL evolves it becomes easier to add support for new device types (potentially as simple as copying and re-naming an existing class). To maximize the potential of this approach an object oriented language that supports complex inheritance relationships should be chosen.

While we have only provided simple examples of attributes and a general description of methods that define interaction with devices, it is hopefully easy to see how these concepts can be leveraged to describe almost anything. While devices are a very important part of the SDL, the SDL is not limited to describing hardware. As mentioned previously, aspects of software systems and components can be represented using similar concepts. More abstract concepts can also be expressed.

Figure 3 illustrates an example of a more abstract concept that can be represented in the SDL, the concept of a point. A point, in this context, is simply a location on the system. For example, one point on a system could be defined as the Network Interface Controller (NIC) of a node in a cluster. Another point on a system could be defined as the NIC of a different node on the same cluster system. In addition to defining these points we can also define a specific diagnostic program that is valid for the type of NIC that is pointed to by the points that we have defined. Imagine that we want to determine if these two NICs on the cluster can communicate with each other. The concept of points could be leveraged by a utility (possibly initiated from a Graphical User Interface (GUI)) that recognizes that the two points selected share a common diagnostic program. Based on this information, the utility could run the diagnostic program and report success or failure. If we expand on this concept by defining points that represent every NIC of this type on the system, connectivity can be tested between any node that has a NIC of this type. Consider how powerful this concept can be if any location in the system can be represented by a point and associated with an appropriate diagnostic. The concept of a point is just another example of a class that can be defined in the SDL. In implementation the point class could be associated with other classes by leveraging multiple inheritance. Objects of class point could also be associated with objects instantiated from other classes defined in the SDL using other mechanisms.

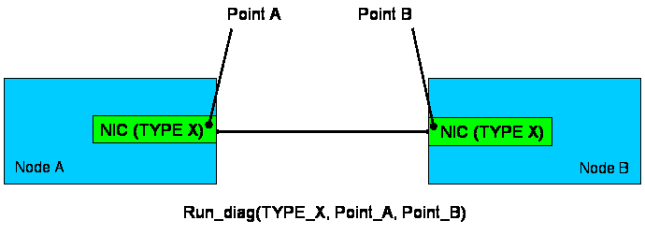

Fig. 3. Points on a System

Once the SDL is capable of describing the system, a database, or databases, can be generated and used by the RAS components to perform their duties (section IV describes more about the database system).

IV. Database

In the following discussion we will use the term database broadly to mean a place where information is stored in some structured way. We will discuss two databases but depending on how this architecture is implemented it may make sense to use more than two, possibly optimized for specific access methods, to satisfy information storage and retrieval needs.

Each system is unique in some way. Even if two systems are comprised of identical hardware the serial numbers or naming scheme, for example, of the individual components would be different. The first database describes each unique system using the SDL. This database, more accurately a Persistent Object Store (POS), contains objects representing hardware devices, software components, and any other abstract characteristic (like points) of the system that can be leveraged by the RAS software. Additionally, information about relationships between these objects, like topological information, is also stored. Once instantiated, this POS is a complete software description of a unique platform. The POS is used by the RAS system for any and all information about how to accomplish its purpose. Attribute information is leveraged for names, roles, IP addresses, simple and complex relationships and groupings (to name a few). Methods are used for specific device interaction and can leverage any capability implemented by a device.

Methods can also be used to describe how to construct messages before transmission and interpret them upon receipt (see section V). Additionally, methods can describe class unique schemas that can enable storage of information in creative ways that allow leveraging the strengths of relational databases without incurring limitations like strict pre-defined schemas. This information will typically be stored in a database (or databases) separate from the POS.

The second database that we will discuss is used for storage of information about the state of the system. This information would likely be stored in some type of relational database. The information contained in this database would have many uses for both real-time monitoring and historic calculation of system metrics (to name a few). The information in this database could be directly accessed using whatever interface the selected database exposes. In this way we can leverage the power of, for instance, advanced relational database languages. By using the information in conjunction with the POS and interpreting it using the SDL, the same information can be processed in other creative ways. If new ways of processing the information stored is desired, the best approach can be chosen from one or a combination of the available methods, or a new interpretation process can be defined in the SDL and then leveraged by the system.

Informational databases, such as these, could also be leveraged by other system components like run-time systems or schedulers. Run-time components can use a database for storing any number of data-points like job runtime or error codes. Scheduling systems could check the current status of nodes on the system before allocating them to a user. In cooperation with the run-time system, the scheduler could also mark nodes as unavailable that reported error conditions on previous runs. A GUI could use this database as a source for presenting last known state of components and update the information in the database by instructing the RAS system appropriately. A node marked as unavailable in the database could be returned to service through the GUI interface after it is determined to be healthy.

These are just a few examples of how database functionality could play a role in the RAS architecture. We anticipate that additional uses for these and other types of databases will emerge during implementation of this architecture.

V. Transport

It is essential for a RAS system to be capable of transporting information from one point to another. In this architecture the transport mechanism fills this role (see Figure 1). Information can be, for example, commands initiated by the administrator on the Top-Level node destined for a specific device or devices, or possibly a report of a failure originating from a device destined for the Top-Level node. Messages can originate and be instructed to terminate at any point in the RAS system.

For example, an administrator issues a command on the Top-Level node (possibly from a GUI) to obtain the status of a system component. As a result of the request a message will be created. To preserve extensibility the only mandatory portions of the message are its destination and the object name associated with the message. (Note that the destination can be a single location, a group of locations or everywhere in the RAS system. The object name is necessary so any component of the RAS system can retrieve information about the object from the POS.) Once the message is created it is introduced into the transport mechanism. Recall from our discussion in section II that on a very small system it is possible that the Top-Level node will communicate directly with a device interface node. If so, it uses the same Application Programming Interface (API) as it would if it were talking to a component of the transport mechanism. Typically, however, the Top-Level node will pass the message off to the first of possibly several transport nodes that lie in the path to the final destination of the message. An important aspect of the transport mechanism is that it does not care about the payload of the message. Once the message reaches its destination, usually one of the device interface nodes, the SDL provides the key to interpret the payload based on the class or classes associated with the object the message is linked to. It is important to point out that objects can refer to, or in a sense contain, other objects of the same or different classes. The payload of a message can therefore be formatted and interpreted in many different ways. This allows the message itself to contain data in many different forms and not be limited by a scheme that will likely prove to be inadequate shortly after implementation. This object defined format of a message is one concept that allows the transport mechanism to be highly portable. By encapsulating the details the transport mechanism can largely ignore what it is transporting and worry about its responsibility of getting the message to where it is going. At this point the device interface node, after receiving the message, interpreting it and acting on it, will likely construct a message in response and insert it back into the transport mechanism for delivery.

Transporting a single message may seem uncomplicated but consider that the transport mechanism will be responsible for dealing with large numbers of messages in parallel. For a 10,000 node cluster system, if we only account for the nodes themselves, the transport mechanism would potentially have to deal with 10,000 messages at the same time. In reality a 10,000 node cluster system has many times more hardware and software components, all of which add to the number of messages that the transport mechanism must support. Considerations like these require the transport mechanism, and the RAS system as a whole, to be scalable.

The transport mechanism scales with the size of the system by using the SDL to understand its role in information transport. While the transport mechanism has knowledge of all aspects of the system the most important (in the area of scalability) is information about its location in the hierarchy. Interaction between transports (instances or daemons in the overall transport mechanism) is specifically described by an API. Figure 1 illustrates some of the potential areas of interaction. Notice that the Top-Level node will communicate with the transport mechanism using this API as will the device interface components. The same API may be extended to communicate with other resources, even resources external to the RAS system. Run-time and scheduling systems are examples of components that would potentially interact with

the RAS system to obtain information using this API. Typically, however, the device interface components of the RAS system will be responsible for inserting the largest number of messages into the transport mechanism.

VI. Device Interface

The device interface components of the RAS system interact with, potentially, all components (software or hardware) in the system. Consider that whether the component the RAS system is required to interact with is hardware or software, there is fundamentally no difference from the RAS systems perspective. As an example, whether the device interface component is talking to a low level firmware interface through a serial connection, or communicating to a persistent daemon with a full featured API, it uses methods defined in the SDL (as part of new or existing classes) to communicate with the component in a language that it understands. These methods can easily be added to the SDL so that the device interface components, or any other part of the RAS system, then understands the new languages required to communicate with components either introduced to an existing system, or a different system entirely. We have stressed the importance of extensibility throughout this paper. The extensibility of the SDL is leveraged the most in the interaction between the device interface components and the system components themselves.

Device interface components can act on their own based on predefined configurations. Monitoring functions frequently fall into this category. These operations can be performed based on configurations defined in various places. Configuration options can be part of the object that represents the component being monitored. Configurations can also be contained in objects themselves and simply related to the object being monitored. In the same way groups of configuration objects can be related to an object, or groups of objects, that are being monitored. This allows for great flexibility in configuring these or any other type of operations. This type of monitoring is frequently used for persistently monitoring the status of components. For example, a device interface component may be required to monitor a temperature sensor on the mother board of a node. The configuration associated with the object being monitored (the temperature sensor) defines a range. When the reading is within the defined range no action is taken. If the reading is outside of the defined range an action, also described in the configuration, is taken. Typically an out of range condition like this would result in the device interface component constructing a message destined for the Top-Level node to notify the administrator of the condition. The same condition could alternatively be configured to be handled by the device interface component which, for example, in response might increase the speed of a fan that cools that area of the mother board. If the temperature does not return to the normal range, in a time also specified in the configuration, the device interface may then notify the administrator by sending a message through the transport mechanism of the RAS system.

Device interface components can be instructed to accomplish tasks by other parts of the RAS system. An example of this is a control function initiated by a user on the Top-Level node and passed to the device interface component through the transport mechanism. For example, it is common for a user to query the status of a component, or components, on the system, possibly in response to a warning condition displayed on a GUI. In this case a message is generated by the Top-Level node containing a status instruction and passed to the transport mechanism which delivers it to the device interface component to act on the instruction. The device interface component accomplishes the required task based on its knowledge of how to interact with the component as defined in the object representing the component. Based on the response from the system component, the device interface component constructs a response message and passes it to the transport mechanism. The transport mechanism delivers the message to the Top-Level node where the user can act on the information. These types of control functions can also result in the status of a component being updated in a database that tracks this type of information.

Device interface components can be instructed to accomplish virtually anything described in the SDL. Operations that can be accomplished by the RAS system are only limited by the capabilities of the components that the RAS system interacts with.

VII. Future and Associated Work

While a complete formal implementation of this design has not yet begun, test implementations of portions of this system have been accomplished for purposes of verifying the feasibility of the design. The concept of the SDL is, for example, a much expanded design based on work done for the Cluster Integration Toolkit (CIT)[5] project at Sandia Labs. Test implementations have been written using Intelligent Platform Management Interface (IPMI)[4] devices to represent the system components. These test implementations were representative of the SDL in structure and tested the functionality of both directly including interface code in the classes and also leveraging external libraries such as the open source IPMI library FreeIPMI[6]. Research into capabilities of embedded systems and how they can be leveraged for the hardware architecture has also been accomplished.

Since the SDL is such a foundational concept in this design more formal documentation of the design must be accomplished to flush out potential problems that might be encountered and addressed before implementation. Specifically, the more abstract portions of the design (for example, the concept of points touched on in this paper) should be detailed and possibly tested with skeletal implementations. Some of the more complex relationships envisioned between classes should also be detailed and tested.

Only high level design work has been accomplished in the database area. Much research has yet to be done, specifically into how capabilities of modern databases can be leveraged to accomplish some of the more complex data mining capabilities that we have envisioned.

While we do not expect the transport mechanism to be a difficult portion of the implementation, the importance of this

component requires that it is not taken lightly. The free form composition of the messages may present some challenges like excessive overhead in build up and tear down of messages by the sender and receiver. The portions of the API that are exposed outside of the RAS system itself will also have to be well thought out. Changing the API within a controlled system causes work, but changing an API intended to be a "standard" interface can discourage others from using the interface.

VIII. Conclusion

While we touched on many of the high level concepts that describe this design which will guide further specifications and eventually implementation, this clearly is not a complete discussion of the many challenges in designing a RAS system. RAS can be a very broad concept. Even if considered in the most narrow of scopes RAS is a large topic that is difficult to address. We feel work in this area is important and impacts many if not all areas related to computing. The benefits of mature RAS systems have yet to be realized in many areas of computing. Hopefully in time, demand for more maintainable systems will drive concepts that have been common on high end platforms into other areas of computing.

IX. Acknowledgments

This research was funded by the Computer Science Research Foundation [7] at Sandia National Laboratories. We appreciate their continued support in funding RAS research at the Laboratories. Thank you to Cynthia Segura and others who reviewed this paper and provided valuable feedback.

References

[1] 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
[2] RedStorm - http://www.cs.sandia.gov/platforms/RedStorm.html
[3] BlueGene/L - http://www.llnl.gov/asci/platforms/bluegenel/bluegene_home.html
[4] Intelligent Platform Management Interface (IPMI) - http://www.intel.com/design/servers/ipmi/
[5] Cluster Integration Toolkit (CIT) - http://www.cs.sandia.gov/cit
[6] FreeIPMI - http://www.gnu.org/software/freeipmi/
[7] Computer Science Research Institute - http://www.cs.sandia.gov/CSRI/