A Scalable and Flexible Monitoring System Framework for Supercomputers

Tong XIAO1,* and Kai LU1,2

1College of Computer, National University of Defense Technology, Changsha 410073, China
2Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha 410073, China

*Corresponding author

Keywords: Monitoring system, Framework, Supercomputer, Scalable, Flexible.

Abstract. Mankind’s demand for more powerful computing capabilities is never met, which has led to the continuous improvement of supercomputers’ performance. A more powerful supercomputer tends to have a larger system scale, which brings serious challenges to the system management, within which how to monitor the system’s state is a critical problem. To address this problem, a scalable and flexible monitoring system framework for supercomputers is brought forward in this paper which can monitor supercomputers with tens of thousands of nodes effectively and efficiently. In this paper, we firstly give an overview of the framework and then focus on the Super Computer System Description Language (SCSDL) which is key to the framework. In the end, we explain some techniques about implementing the framework, and the client GUIs of a job monitoring system and an error monitoring system for Tianhe-2 based on this framework are given, from which we can see that the framework is well scalable and flexible to monitor Tianhe-2 which has 16,000 nodes effectively and efficiently.

Introduction

It’s widely acknowledged that computational science has become the third pillar of the scientific enterprise, a peer alongside theory and physical experiment [1]. Computational science utilizes advanced computing capabilities to understand and solve complex problems. With the increase in size and complexity of the problems encountered, more powerful computing capabilities are always needed, which drives the continuous improvement of supercomputers’ performance.

As supercomputers are becoming more and more powerful, the system scale is rising rapidly. According to the latest Top500 list released in November 2016, the top 1 Sunway TaihuLight has 40,960 nodes, and the 2nd, Tianhe-2 has 16,000 nodes [2]. With such a large-scale supercomputer system, how to manage it effectively and efficiently to improve its reliability and usability is a very challenging problem.

Within the supercomputer’s system management, a very critical issue is to monitor the system’s state. The method of querying system state by command line tools is rather inefficient and unfriendly, as the administrator has to extract useful information from a large amount of plain text. Currently, there are a bundle of management systems for clusters and servers such as Ganglia [3], Nagios [4], Zabbix [5], etc. All of them provide GUIs to manage the target system, including monitoring the system’s state. However, they are not scalable and flexible enough to monitor supercomputers with tens of thousands of nodes like Tianhe-2.

To monitor Tianhe-2 and larger-scale supercomputers effectively and efficiently, we designed a scalable and flexible monitoring system framework based on which a job monitoring system and an error monitoring system for Tianhe-2 are implemented.

The rest of this paper is organized as follows. In section 2 we will give an overview of the framework. In section 3 we will describe in detail the Super Computer System Description Language (SCSDL) which is key to the framework. We will explain some implementation techniques in section 4 and conclude in section 5.
Overview of the Framework

Our framework is in C/S mode, as shown in Figure 1. The server is deployed on a management/login node of the target supercomputer system, and the client can be run on any ordinary PC or laptop connected to the management/login node through network. The client and server communicate with each other through SSH protocol.

An interaction between the client and server is as follows. First, the client sends a request of querying the system state to the server. After receiving the request, the server runs a set of scripts to obtain the system state information and format the raw data into an XML file which conforms to the schema of the Super Computer System Description Language (SCSDL) defined by us. The XML file is then returned to the client and the client parses it and visualizes its content on the GUI. This process keeps on periodically and the user can specify the interval time on the client GUI. Also, the user can start the process immediately by issuing a request manually on the client GUI in case of urgent needs.

Super Computer System Description Language (SCSDL)

Considering the large scale of a supercomputer, the amount of system state information is generally huge. Thus it would be quite convenient and friendly to the system administrator if we provide a GUI which reflects the physical layout, or more specifically, the node distribution of the supercomputer and associates state information with nodes on it. For example, it would help the system administrator to master the system usage if the GUI shows which nodes are occupied by each job and where the nodes are. Additionally, it would accelerate reaching unhealthy nodes if the GUI shows the health condition and location of each node. To get such a GUI and simplify the work of the client as well as reduce the data needed to be transferred from the server to the client, we designed the Super Computer System Description Language (SCSDL), which is a special form of XML, i.e., a subset of XML.

We use the W3C XML Schema Language [6] to define the schema of SCSDL since it’s much richer than DTDs [7]. It would be tedious to interpret the schema of SCSDL, so we merely take a demo document which conforms to the schema as an example and call it an SCSDL document. As shown in Figure 2 is a demo SCSDL document used in job monitoring, from which we can see the six important children of the root element of an SCSDL document: objects, details, table, tablelayout, systemview, systemviewlayout.

The objects element may contain several or a lot of children named object, each of which is used to identify a piece or a class of state information by giving it a unique id and color, and is called a state information object. The id is used to reference it elsewhere and the color is used to do painting that we’ll mention later. The object element normally contains some brief information. Figure 3 is a demo objects element used in job monitoring and each child element object represents a job information object. If we need more details of the state information, then we can add the details element which provides a child element detail for each object element to offer some details of the state information object.
The `table` and `tablelayout` elements are used in pair to control a table widget on the client GUI. A `tablelayout` element is bound to a `table` element by setting its `gid` attribute to the `id` attribute of the `table` element. Figure 2 shows two pairs of bound `table` and `tablelayout` elements. The `table` element defines the columns of the table and provides table content in rows. The `tablelayout` element gives some layout information about the table when displayed on the client GUI, such as which columns are to be visible and the percentage of each column’s width to the table’s total width.

The `systemview` and `systemviewlayout` elements are used in pair as well and are bound in the same way as the `table` and `tablelayout`, which can be seen from Figure 2.

The `systemview` element has two children: `scheme` and `data`. The `scheme` element divides the target supercomputer into several layers, and the lower the layer is, the less nodes it covers. As shown in Figure 4, we divide the target supercomputer into five layers: Row, Rack, Frame, NodeBoard, ComputeNode, and we can interpret it as follows: the supercomputer consists of two Rows (numbered from 0 to 1), each Row consists of 6 Racks (numbered from 0 to 5), each Rack contains 2 Frames (numbered from 0 to 1), each Frame contains 32 NodeBoards (numbered from 0 to 31), and each NodeBoard has 1 ComputeNode on it (numbered 0). After this division, we can convert the serial No. of each node into a layered No., for example, the first ComputeNode cn0 has a layered No. Rw0-Rk0-Fr0-Nb00-Cn00 according to the `scheme` element in Figure 4 (formatted using each layer’s mask attribute). We borrow this idea from the school’s management of students, a school usually divides students into several grades, each grade has several classes and each class consists of some students.

The `data` element is used to associate state information with nodes. With the help of the `objects` element and `scheme` element, we can associate state information with a bound of nodes at a time without needing to iterate each node thus reducing data size. As shown in Figure 5, we associate a `layer` element (layer1, layer2, etc.) with a job information object by setting its `oid` attribute to the `id`
attribute of the *object* element representing the job information object. If all the nodes covered by a *layer* element are associated with the same state information, here, occupied by the same job, then its children can be pruned. On the other hand, the children of a *layer* element inherit its *oid* attribute by default, so we only need to specify those children who are associated with different state information objects, which reduces unnecessary redundancy. The *layer* elements with identical *oid* attributes can be expressed more compactly using the *min* and *max* attributes.

```xml
<scheme>
  <layer1 tagname="Root" min="0" max="1" mask="Rw%01d">
    <layer2 tagname="Pack" min="0" max="5" mask="-Rk%01d">
      <layer3 tagname="Frame" min="0" max="1" mask="-Fr%01d">
        <layer4 tagname="NodeBoard" min="0" max="31" mask="-Nb%02d">
          <layer5 tagname="ComputeNode" min="0" max="0" mask="-Cn%01d">
            ...
          </layer5>
        </layer4>
      </layer3>
    </layer2>
  </layer1>
</scheme>
```

Figure 4. A demo *scheme* element.

```xml
<data>
  <layer1 oid="empty" min="0" max="0">
    <layer2 oid="empty" min="1" max="1">
      <layer3 oid="empty" min="0" max="0">
        <layer4 oid="j000003" min="16" max="16">
        </layer4>
      </layer3>
    </layer2>
  </layer1>
</data>
```

Figure 5. A demo *data* element used in job monitoring.

The *systemviewlayout* element is used to provide layout information of the system view on the client GUI. We want the system view to reflect the actual node distribution of the target supercomputer system so that we can locate any node exactly at the position indicated by the view, which is of great use when doing hardware replacement. The *systemviewlayout* element splits the display area of the system view into grids level by level until a rectangular area is designated for each node. To explain this procedure, let’s take Figure 6 as example:

The *level0* element represents the entire display area, and its *cols* attribute equals 1 indicates that it is split into one column, i.e., no split in the vertical direction. We do not specify how many rows to split the area into, so arbitrary number of rows can be added when needed. In this way the area is split into grids and we number them starting at 0 from left to right in each row and from top row to bottom row. The *level1* element’s *posstart* and *posend* attributes indicate taking the grids of *level0* numbered
from 0 to 1. Since level0 has only one column, so it must be split into two rows, the first row is grid 0 and the second row grid 1. The min and max attributes of level1 reference the layer1 elements numbered from 0 to 1 defined by the scheme element mentioned above. Let’s take the scheme element in Figure 4 as example in which the layer1 element represents Row. Then grid 0 of level0 is allocated to Row 0 and grid 1 of level0 to Row 1. The cols attribute of level1 indicates that the two grids allocated to it are both split into 8 columns further, so the first row contains grids 0 to 7. The first level2 element’s posstart attribute indicates that it takes grid 0 of level1. Similarly, the third level2 element takes grid 7 of level1. These two level2 elements only have a child element img which is used to specify the background image, so the grids allocated to them will not be split further. To the second level2 element, its posstart and posend attributes indicate that it takes grids 1 to 6 of level1, and its min and max attributes reference the layer2 elements numbered from 0 to 5 defined by the scheme element. Again, take the scheme element in Figure 4 as example in which the layer2 element represents Rack, then grids 1 to 6 of level1 are allocated to Rack 0 to 5 sequentially. The second level2 element also specifies a background image through its child element img, and the fact that it has a child element level3 means that the grids allocated to it will be split further. The split operation is just similar and we will not repeat it. Finally, when we get to level5 which is determined by the maxlevel attribute of level0, each node is allocated with a grid, i.e., a rectangle area. In systemview’s child element data, each node is associated with a state information object and we can use the color of the object to fill the node’s rectangle area. By this way we show the association between nodes and state information objects, i.e., association between nodes and state information on the client GUI.

```xml
<systemviewlayout id="DemoLayout" grid="Demo">
  <level0 maxlevel="5" cols="1" hgap="5" vgap="25" marginwidth="10" marginheight="5">
    <level1 posstart="0" posend="1" min="0" max="1" cols="8" hgap="0" vgap="0"> <!-- Row -->
      <level2 posstart="0">
        <img src="/images/row_header.png" />
      </level2>
    </level1>
  </level0>
  <level2 posstart="1" posend="6" posstep="1" min="0" max="5" cols="1" marginproportion="0.0879" marginbottomproportion="0.0114" marginleftproportion="0.1134" marginrightproportion="0.1055" hgap="0" vgap="0"> <!-- Rack -->
    <level3 posstart="0" posend="1" min="0" max="1" cols="16" hgap="0" vgap="0"> <!-- Frame -->
      <level4 posstart="0" posend="31" min="0" max="31" cols="1" vgap="0"> <!-- ComputeNode -->
        <level5 posstart="0" min="0"> <!-- NodeBoard -->
          <level6></level6>
          <level7></level7>
          <level8></level8>
          <level9></level9>
          <level10></level10>
        </level5>
      </level4>
    </level3>
  </level2>
  <level12 posstart="7">
    <img src="/images/computation.png" />
  </level12>
</level10>
</systemviewlayout>
```

Figure 6. A demo systemviewlayout element.

To understand the above procedure better, let’s consider the situation where we want all the students of a school to gather on the playground orderly. Generally, we first divide the whole playground into several regions and allocate each grade with a region. Then the region of each grade is further divided into smaller regions each of which is allocated to a class. Finally, each class specifies a position for each of its students. By this way, the location of every student is specified clearly. The systemviewlayout element works in a similar way.

To make the system view richer and nicer, we provide the img element which can be used to specify a background image for a grid. What’s more, when splitting a grid into smaller grids, we can set the horizontal and vertical gaps between the smaller grids through hgap and vgap attributes respectively, and margins on four sides of the grid through margintop, marginbottom, marginleft, marginright attributes. Besides setting these attributes’ values to absolute pixels, they can also be set...
to proportions of the grid’s width or height through \textit{hgappropportion}, \textit{vgappropportion}, \textit{marginintopproportion}, \textit{marginbottomproportion}, etc.

With the help of \textit{systemview} and \textit{systemviewlayout}, we can build a “map” that reflects the node distribution of the target supercomputer system, by which it’s easy to find any specific node. There is a critical problem we have to mention here, that is how to divide the target supercomputer into layers, i.e., how to define the \textit{scheme} element. It’s common to do the division according to the target supercomputer’s physical hierarchy, just as what is done in Figure 4. But that’s not necessary, the core principle is to make both \textit{systemview} and \textit{systemviewlayout} as simple and clear as possible after the division.

\textbf{Implementation}

We implement the client based on Eclipse RCP which can provide rich GUI experience and is cross-platform. To parse the SCSDL document received from the server, we employ Oracle’s Java Architecture for XML Binding (JAXB) technology \cite{7}. By this technology, a set of Java classes are generated according to SCSDL’s schema and an SCSDL document is unmarshalled into objects of them, so we can access the SCSDL document just by accessing the Java objects, which greatly simplifies the operation of accessing an SCSDL document. The client then generates several table widgets each of which is generated according to a pair of \textit{table} and \textit{tablelayout} elements, and paints the system view according to the \textit{systemview} and \textit{systemviewlayout} elements. Other widgets can also be added if needed.

In order to make our framework flexible and easy to be transplanted to monitor other supercomputer systems, the server produces the SCSDL document which contains state information based on a template rather than in a hardcoding style. The SCSDL document template is specified through the request sent to the server and it contains all the \textit{–layout} elements as well as the \textit{systemview} element with only child element \textit{scheme}. The server generates other elements and the child element \textit{data} of \textit{systemview} on the basis of the template. When we want the framework to monitor a different supercomputer system, all we need to do is providing an SCSDL document template and specifying it in the request sent from the client to the server.

Figure 7 and Figure 8 show the client GUIs of Tianhe-2 Job Monitoring System and Tianhe-2 Error Monitoring System based on the framework respectively.

In Figure 7, the upper left area is the system view which reflects the physical layout of Tianhe-2, below which there are two tables: the left table is called Active Jobs table and lists some important information of all the jobs currently running; the right table is called Inactive Jobs table and lists some important information of all the jobs waiting for resources. An \textit{object} element is generated by the server for each job in the SCSDL document, so each job information object is allocated with a unique color. In the system view, all the rectangle areas representing the nodes occupied by a specific job are filled with the color allocated to the job information object. In the Active Jobs table, each row contains a job’s information and is marked with a colored block using the color allocated to the job information object. The jobs listed in the Inactive Jobs table do not occupy any nodes and all the rows are simply marked with gray blocks. Because of Tianhe-2’s large scale, the system view is split into two tabs in case the rectangle area of each node is too small to be distinguished. If the system scale is even larger, more tabs will be added by the client automatically when needed. Another technique to cope with the large system scale is the mouse interaction between the system view and the Active Jobs table: if we press the mouse down on a row of the Active Jobs table, the rectangle areas of the nodes occupied by the corresponding job will all be highlighted with bold borders; if the mouse hovers over a node’s rectangle area in the system view, and the node is currently occupied by a job, then the row containing the job’s information will be selected automatically and all the rectangle areas of the nodes occupied by this job will also be highlighted. Other areas of the client GUI are used to display other useful information and are not the focus of this paper.

Figure 8 is similar to Figure 7 and we will not detail it any more. In Tianhe-2 Error Monitoring System, all the errors are classified into several ranks and an \textit{object} element is generated by the server
for each rank in the SCSDL document. Then, in the system view, the rectangle area of each node can be filled with a color according to the rank of the error which occurs in the node, for example, green stands for normal and red represents serious error. The tables list some important information about the errors which have occurred in the target supercomputer system and each row is marked with a colored block according to the error’s rank. Mouse interaction between the system view and the tables is also supplied in Tianhe-2 Error Monitoring System.

Figure 7. Client GUI of Tianhe-2 Job Monitoring System.

Figure 8. Client GUI of Tianhe-2 Error Monitoring System.

**Conclusion**

To monitor large-scale supercomputers effectively and efficiently, we propose a scalable and flexible monitoring system framework for supercomputers in this paper. The framework is in C/S mode: the client is based on Eclipse RCP and can run on any ordinary PC or notebook, the server is deployed on a management/login node of the supercomputer, the client and server communicate with each other through SSH protocol. The framework is easy to deploy and has little impact on the target supercomputer.
We focus on the Super Computer System Description Language (SCSDL) since it plays a key role in making our framework scalable and flexible. With the help of SCSDL, we can build a system view reflecting the actual node distribution of the target supercomputer and associate state information with nodes on the client GUI, which brings great convenience to the management of large-scale supercomputer systems. By implementing a job monitoring system and an error monitoring system for Tianhe-2 based on the framework, we verify that it’s scalable and flexible enough to monitor a supercomputer with tens of thousands of nodes effectively and efficiently.

Acknowledgement
This work is partially supported by the The National Key Research and Development Program of China (2016YFB0200401), by program for New Century Excellent Talents in University, by National Science Foundation (NSF) China 61402492, 61402486, 61379146, by the laboratory pre-research fund (9140C810106150C81001).

References