Procedia Computer Science
Volume 66, 2015, Pages 625–634
YSC 2015. 4th International Young Scientists Conference on Computational Science

Dynamically Reconfigurable Distributed Modular Monitoring System for supercomputers (DiMMon)

Konstantin Stefanov, Vladimir Voevodin,
Sergey Zhumatiy and Vadim Voevodin[1]
[1]*Research Computing Center, Moscow State University*
{cstef, voevodin, serg, vadim}@parallel.ru

Abstract
A design for a new dynamically reconfigurable distributed modular monitoring system framework is proposed in this paper. The proposed design allows combining both monitoring tasks (supercomputer 'health' monitoring and performance monitoring) in one monitoring system. Our approach allows different parts of the monitoring system process only the data needed for the task assigned to these parts. This helps to process a lot of performance data and to get information about dynamic features of heavy parallel tasks. Another feature of our framework is the ability to calculate performance metrics on-the-fly, dynamically creating processing modules for every job or other objects of interest.

Keywords: monitoring; performance monitoring; supercomputer; BigData; monitoring system.

1 Introduction

Modern supercomputers demonstrate impressive performance, but in practice their capacity is severely underutilized. Everyone has heard or seen just how low supercomputer performance can be on real-life applications: in most cases it's just a small percent of peak performance indicators. But few really know exactly how low the efficiency of a supercomputing center generally is. While losses can be virtually invisible at each specific step of preparing and executing applications, they tend to snowball as they accumulate. Nothing is too small here, and every element of a supercomputing center must be thoroughly reviewed – starting from the accepted policy of task queuing and the dynamic application features to the system software setup and efficient infrastructure operation.

This is especially true for heavy science applications that need to run with high performance. It is very important to monitor their runtime metrics and detect inefficiency roots. For this purpose, monitoring data should be obtained with very high frequency, and in this case volume of this data can reach up to Terabytes per day per cluster, which makes its analysis a BigData task itself.

Besides efficiency, another important issue today is control over the proper operation of the hardware and software within the supercomputer systems themselves. The main reason is the

doi:10.1016/j.procs.2015.11.071

unprecedented growth in degree of parallelism. Thousands of users and applications; hundreds of thousands of computing nodes, processors, accelerators, ports, cables, software and hardware components; many millions of processing cores, processes, events, messages… Making sure all these components function together as a single system requires skillful control over the state of supercomputer components, and prompt detection and isolation of failures and errors.

However, monitoring a supercomputer system is a daunting task, and not just because of the sheer number of components. The main challenge is to ensure complete and constant monitoring. No human can handle this task, which begs to be automated – with monitoring systems capable of promptly monitoring the hardware and software status.

Ideally, monitoring systems should deal with both problems mentioned above, ensuring the correct operation of a supercomputer's subsystems and the efficiency of all applications being executed. The problems are formulated similarly, but their solutions require completely different features. In the first case, it is important to monitor the situation in its entirety and react promptly. In the second case, the biggest challenge is the huge volume of monitoring data. When dealing with the first issue, the key is to ensure the reliable operation of the monitoring system. Addressing the second issue requires focusing specifically on developing the monitoring system design principles, which help to collect, analyze and store this data.

In this paper, we propose an approach to building scalable supercomputer monitoring systems, both for current and future systems, with an extremely high degree of parallelism. Success in building a monitoring system can be achieved by reaching a reasonable compromise on two key issues: which data is stored in the database, and when and where it is analyzed. This approach formed the basis for a monitoring system that is currently being tested with the Chebyshev and Lomonosov supercomputers at Moscow State University.

2 Background and Related Work

Both hardware and software components can be the target of monitoring within supercomputer systems. Despite the diversity of data sources reporting their status and the different methods of collection, monitoring system design methods have much in common.

Fig. 1 shows the general architecture of monitoring systems. Agent processes operating on computing nodes are responsible for obtaining the information. Some systems claim to be 'agentless,' e.g. (Zenoss, 2015). In this case, the role of agents is played by certain operating system services running on the node. These can be SNMP agents, SSH access to gather information, etc. To get information on the status of other components of the computing system, SNMP agents or other components gathering data via different protocols can be used instead of agents running on the computing nodes. Data collected in this way is sent to the monitoring system server part, where it is processed, partially saved to the database, events are extracted and acted upon, and some parts are extracted for presentation to the user, particularly for visualization.

The majority of existing monitoring systems (Zenoss, 2015), (Zabbix, 2015), (Cacti, 2015), (M. L. Massie, 2004), (Nagios, 2015), (Collectd, 2015) are built according to this principle. Let's take a look at their common characteristics.

First of all, no data are processed at the source computing node. The reasons for this are obvious for agentless systems. However, even for systems that install their agent software on the computing nodes, these agents are only used for collecting the relevant information and passing it on without processing. Usually this approach is explained by the need to reduce the agent's impact on the applications executed at this computing node.

Another feature of existing monitoring systems is a fixed configuration of data transmission routes. Data transmission routes are set during the initial configuration and cannot be changed during the course of its operation. Each agent is assigned a single address in the monitoring system server partition (or several addresses, if the server uses redundancy) to send collected data, and this address is also fixed throughout the system operation.

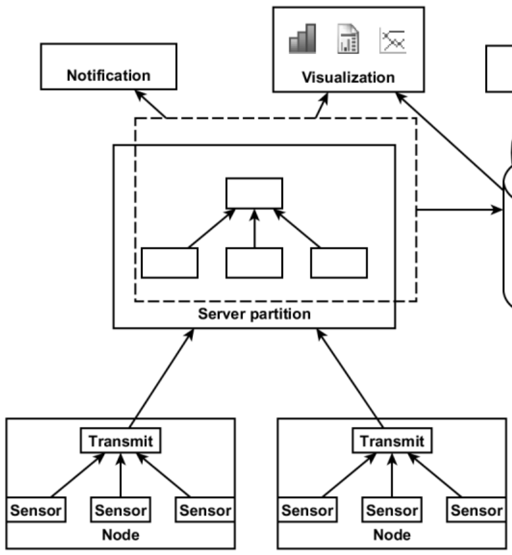

Figure 1: General monitoring system architecture

As a rule, to maintain performance, the monitoring system server partition has a distributed tree-like structure. Each leaf on the server partition is responsible for a given number of computing nodes. This component aggregates and processes the information, singling out data indicative of the events occurring in the system. The processed and streamlined flow of monitoring data is then sent on for further processing.

To ensure reliability, some components of the server partition (or the entire server partition) can be configured for redundancy, using several independent servers. In this case the entire data flow from computing nodes is sent to two hosts, where the data processing procedure is duplicated. If one component fails, the redundant component takes over its functions.

This architecture is particularly well-suited for tracking emergency situations (overheating, leaks, power failures, activated fire alarms), and the availability and operational status of individual components in the computing system. The data collection interval in this case varies from several dozen seconds to several dozen minutes, in addition to passive monitoring (reacting to event signals from external sources) used for tracking events that require immediate response.

The job of application performance monitoring is different from other types of monitoring. Data needs to be collected faster, within intervals of a few seconds, or even fractions of seconds. As a result, the volume of data to be processed is much larger. Besides, a parallel application is a dynamic object. Computing nodes and data sources associated with a specific job are determined when an application is launched.

Many approaches to monitoring job performance have been proposed (D. Gunter, 2002), (J. Mellor-Crummey, 2002), (H. Jagode, 2009), (L. Adhianto, 2010), (G. Eisenhauer, 1995), (M. Kluge, 2012), (R. Mooney, 2004), (B. Ries, 1993). Their common feature is that they are intended for monitoring the performance of specific jobs. A significant portion of these systems (D. Gunter, 2002), (J. Mellor-Crummey, 2002), (H. Jagode, 2009), (L. Adhianto, 2010), (G. Eisenhauer, 1995), (B. Ries, 1993) start agents at the respective computing nodes whenever a job is launched. Data from the agents are collected into a database then reviewed after the job is completed. Some systems, such as (M. Kluge, 2012) and (R. Mooney, 2004), constantly collect and save data from computing nodes. Data analysis is also performed by selecting part of the data related to a specific job.

This approach is quite reasonable when analyzing individual jobs. However, if we want to analyze the entire flow of jobs being executed by the supercomputer at any given moment, this approach would result in too much overhead.

Obviously, highly detailed analysis of monitoring data is only required for certain specific jobs. This means that data for most jobs will be written to the database only once and read only once to calculate the respective integrated metrics which characterize the application in general. However, this mode of operation results in a strong degradation of the data storage system performance, or in unreasonable expense. Indeed, the most common storage systems – those using hard disk drives – show their top performance with linear reads or writes. However, the simultaneous writing of monitoring data and reading them to calculate metrics for completed jobs results in random access to the data, degrading the performance of HDD-based storage by two orders of magnitude or more. One possible solution is to use solid state drives (SSD), but these are way too expensive compared to HDD-based systems. In addition, they tend to fail quickly with many write cycles, which inevitably occurs when they are used for updating a database with performance information.

3 DiMMon system design principles

To address the entire range of supercomputer monitoring tasks, we are working on the DiMMon (Distributed Modular Monitoring) monitoring system framework. Its design is based on the following principles:

- Ability to direct different data flows along different routes, or copy the same data to several recipients for different processing functions.
- Support for the dynamic reconfiguration of the monitoring system operation modes (data transmission routes, data collection parameters, data processing rules).
- Ability to calculate performance metrics for individual jobs while collecting data, without writing it to disk and subsequently reading.
- Partial processing of monitoring data right on the computing nodes.

Sending different data along different routes allows metrics to be calculated both for the system in general and for a job being executed at a specific computing node. Data used to calculate performance metrics for the whole supercomputer are transferred to the components responsible for such calculation, while data needed to calculate individual job performance metrics are sent to the respective components.

To calculate metrics for individual jobs, it is not only necessary to create the respective components the moment a job is launched, but also to build data transmission routes. At the same time, it all has to be done in such a way that all the relevant data (and only relevant data) are received at the right point, i.e. the monitoring system components and data transmission routes need to be reconfigurable in the course of monitoring system operations. This reconfiguration allows metrics to be calculated during operations, without storing the data in the database. This capability means that the monitoring system needs to be connected to the computing system's resource manager.

It should be noted that complete performance monitoring requires data not just from the computing nodes running the job, but also from shared elements of the computing system: data storage, network switches, etc. (H. Jagode, 2009), (M. Kluge, 2012).

Finally, moving some of the data processing over to computing nodes significantly enhances the monitoring system's performance and its ability to process information. Monitoring system makers usually avoid processing any information on the computing nodes, in an attempt to reduce the impact on the job performed by the computing nodes. However, using Simultaneous Multithreading (SMT) technologies allows CPU resources not utilized by the main job to be loaded with additional activities with minimum impact on the main job. Virtual cores, which become available when SMT is enabled, are not usually used to execute computational jobs: all threads of a job share the same processing resources, so launching additional threads on a virtual processing core does not translate to an actual performance increase. Even if we do not count on technologies like SMT and just use a limited number of computing node resources (so as to minimize the impact on the job), resources with substantial total performance will be allocated for the monitoring system at the supercomputer level. Obtaining comparable performance on dedicated hardware would require a great number of servers and accompanying infrastructure. The first experiments at the Moscow State University supercomputers have already confirmed the effectiveness and efficiency of this approach.

4 DiMMon system architecture

The DiMMon monitoring framework is being developed to run under Linux OS, so whenever we speak about an ‘operating system’ we mean a Linux-kernel OS.

DiMMon is a distributed monitoring system composed of monitoring agents running on various computing nodes or other computers. The agents are connected to each other by exchanging messages via network.

A *monitoring agent* is an ordinary process from the operating system viewpoint. A monitoring agent is basically a runtime environment that supports the operation of the monitoring nodes, connections between them and message transmission.

All information on the status of the computing system is received and processed by the *monitoring system nodes* (any references to ‘nodes’ below mean specifically monitoring system nodes, not the supercomputer’s computing nodes). Data are sent from one node to another. Each node is configured independently from the others.

Two types of messages are passed inside the system. The primary type is messages containing monitoring data. They are sent from one node to another following *connection* between the nodes. Connections between nodes are created independently from one another. The second type is control messages, which are sent from one node to another regardless of any connections present between the nodes.

Each node is of a specific *type*. The node type is set when the node is created. The type defines functions called by the runtime system when the node needs to perform a certain action. The number of nodes of each type is not limited. One or more types are described in a *module*. A module is a dynamic-link library with a certain interface. The specific modules linked when a monitoring agent is launched are determined by its configuration. A set of node types which determine the monitoring agent functionality consists of the types defined in the loaded modules.

Each node can have an arbitrary number of named *inputs* and *outputs* (zero or more). Each node can receive and transmit control messages. Each node has a numeric ID assigned by the runtime system upon creation, and each node can also be assigned a name (a character string).

A connection between nodes is created by specifying the name or ID of the source node, name of the output on that node for establishing the connection, data recipient and the name of the input to which the data will be sent. After the connection is created, the runtime system will send a copy of the data output by the node to all inputs connected to the respective output.

The data passed between nodes is a sequence of triplets <*ID*, *Len*, *Value*>, where *ID* is an unsigned integer determining the data type, *Len* is the size of the data (*Value* field) in bytes, and *Value* is the actual data. The contents of the Value field are processed by the nodes handling the data. The Value field can contain a single value, a vector of values (e.g., values of a certain parameter on all processing cores at a computing node) or any other data structure. The node can pass data from input to output, unchanged, even if it is unaware of the specific semantics of the Value field. One message with monitoring data can contain any number of triplets.

Control messages are used to create and destroy nodes, set their names, create and delete connections between nodes, and create and subscribe to timers; these messages are processed by the runtime system (monitoring agent). Messages used to modify node settings are processed by the actual addressee nodes. A control message can be sent by any node to any other node. The message recipient is determined by the name or number of the receiving node.

Another element required for the monitoring system operation is *timers*. A timer is needed to initiate certain actions at certain moments in time. When a timer is created, its name is defined along with the moments when the timer will be triggered. It can be one present moment in time (in absolute terms or relative to the current time), or the timer can go off at certain intervals. A node can be *subscribed* to a timer, or have another node subscribed to one. Control messages are used for subscribing. When a timer goes off, it sends a control message to all subscribed nodes. If several timers go off simultaneously, the nodes are sent one control message containing the list of timers triggered.

All interaction within the system is done in a functional way. The node type determines what functions are available for processing monitoring data, receiving control messages, initializing and destroying the node itself, building connections, etc. These functions can be redefined for a specific node, and functions related to processing monitoring data can be redefined for a specific input.

Several types of nodes are used to build the monitoring system. However, the only difference between these types is their functionality; the runtime system treats all nodes in the same way.

A node that receives monitoring data from the supercomputer hardware or software (equipment or operating system) is called a *sensor*. A sensor is a node that has no inputs (input in the sense of the DiMMon system). Its job is to wait for a timer signal, then obtain monitoring data by querying the hardware, OS services, etc. and send them to its output (sensors normally have only one output).

A node that has both inputs and outputs processes the monitoring data. Processing can take different forms: filtering data, calculating the rate at which a value is changing (e.g., calculating data send/receive speed from bytes sent/received counters), comparing to a certain threshold, etc. Inputs and outputs are named, in order to separate data flows. For example, a node that selects data based on a certain criteria can forward data matching the filtering criteria to an output called 'match', and those that fail the criteria to an output called 'notmatch'. By connecting these outputs to different nodes we have separate data flows which can be processed independently. Alternatively, an input can be named after the type of monitoring data to be sent. For example, in a node calculating a value's rate of change, an input name can include the type of value to be sent (uint32, int64, etc.). These speed calculations are applied to data presented in the form of counters, e.g. counters for bytes/packets passing through a network interface. Inputs and outputs are created on demand. When the first connection to a new input is created, the node gets the input name and may block the connection if it does not have information on how data received through this input can be processed, i.e. the node can decide which input names are allowed to be created. A similar check can be performed when creating a connection to a node's outputs. When the last connection to the given input or output is destroyed, that input or output is destroyed, too.

Nodes that have inputs but no outputs (output in the DiMMon sense again) are intended to send data outside the agent. This node will most likely be used to organize the interaction between agents. This node receives monitoring data at its inputs, serializes and sends it via network to another agent operating on the same or another computer. The receiving agent operates a receiver node, which does not have any inputs, like a sensor node. But unlike a sensor, it receives data not from the OS or hardware, but from another agent across the network. Senders and receivers are configured (where data should be sent, what network address connections it will be accepted at, what protocol should be used for exchange, etc.) with control messages. In our experience, using a UDP protocol for transmitting data over the network is preferred. The maximum size of a datagram (64 KB) is sufficient to transmit all the data collected on a node at once. Connections within the supercomputer LAN are usually very reliable, so no data are lost; meanwhile, using UDP saves on the overhead of maintaining transmission reliability.

Other nodes without outputs include those nodes that save monitoring data to an external database or launch external responses: notifying the administrator via e-mail or SMS, logging an emergency or initiating shutdown of affected computing components.

Since every sensor receives messages from the timer independently from others and sends collected monitoring data in a separate message, those messages must be consolidated into one before being sent via the network. This consolidation is performed by a *consolidation node*. This node saves any message with monitoring data it receives to its internal buffer and sends a special request (special control message) to the runtime system. Upon request, the runtime system creates a list of all currently active timers (sending messages to subscribed nodes) and responds to the consolidation node with a control message once all active timers stop notifying the subscribed nodes. Until this message is received, the consolidation node stores all the messages with monitoring data it receives in a buffer. When the message confirming the end of the timer notification is received, all sensors that were to generate data by a signal from the active timers have already responded with their data; now the consolidation node consolidates all the buffered data into a single message and sends it on for further processing (e.g., for transmission via network).

When a monitoring agent begins its operation, it reads the configuration file for a list of modules to load. In the current implementation all modules are loaded solely at the start of a monitoring agent. A set of node types which determine the monitoring agent functionality is defined by the types described in the loaded modules. The restriction with modules only being loaded at the start of the monitoring agent is not mandatory and can eventually be removed. After the modules are loaded, the agent prepares a list of all node types known to the runtime system, which will be available in the course of operation. Next, the agent reads the type of node to be created first (*starting node*) from the configuration file. This node is created and receives a control message containing the file descriptor pointing to the remaining part of the configuration file. Subsequently the starting node reads and interprets the remaining part of the configuration file. In particular, the file can contain commands to launch other nodes, add connections between them, create timers, have the nodes subscribe to the timers and perform other configuration tasks. After the starting node completes processing the control message, the runtime system completes initialization and switches to a normal operating mode. Further configuration changes can be initiated by any node. This can include a response to an event revealed from the monitoring data, or a node can accept network connections and respond to commands received from the network. This functionality is not included in the runtime system and must be implemented in the nodes.

Based on these principles, we can configure literally any configuration of data routes for different monitoring systems.

5 Using the DiMMon approach to design various monitoring systems

Here are a few examples of possible monitoring system configurations based on DiMMon principles.

First, let's consider using DiMMon as a module being executed on a supercomputer's computing node and supplying information to a single destination for further processing. This configuration is presented at Fig. 2.

Sensors (*Sensor1 ... SensorN*) read information on the status of the computing node where the agent is operating, at the timer (*Timer*) signal, and send it on to the consolidation node (*Consolidation*). After information is received from all of the sensors, the consolidation node releases it for further transmission via the network. Please note that the format in which the data are sent via the network is determined solely by the transmitting node, and by replacing this node it is possible to send data in a format suitable for various recipients, be it other DiMMon agents, other monitoring system server partitions or data recipients.

For simplicity, the figure does not show the starting node, which is created first and is responsible for building the entire configuration, since this node does not take part in the system operation after the configuration is created.

As another example, let's take a look at a monitoring system providing three types of data. The first type is for events that call for a guaranteed response, such as the failure of infrastructure equipment. The second and third data types are related to calculating the performance metrics for the entire computing system and individual jobs, respectively. An example of that configuration is shown in Fig. 3.

Data is sourced from agents operating on computing nodes. Data on infrastructure status can be collected by agents working on computing nodes or on dedicated servers. Primary data processing, such as comparing to set thresholds and calculating counter value change rates, is performed by the computing nodes. Next, data related to hardware operation status are sent for further processing to another part of the system, operating on dedicated servers. For reliability, this part has a redundant configuration, and the data are processed in the same way on two independent servers. Agents working on these servers also monitor the status of the other server in the pair.

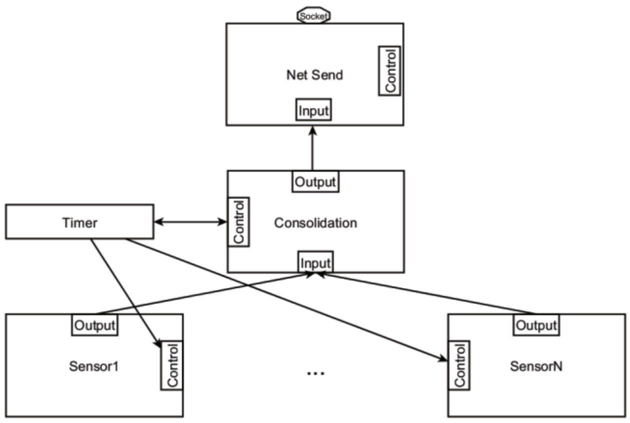

Figure 2: Example DiMMon agent configuration for obtaining information on the status of a computing node

Performance-related data are sent to the part responsible for calculating system-wide performance metrics. Additionally, whenever a new job is launched, components are created automatically to monitor the performance metrics for that specific job. Monitoring agents on the nodes performing this job can be reconfigured without interrupting their operation, so as to send copies of the performance data to the newly created monitoring system components to process data for the new job. This way, some of the data transmission and processing routes (monitoring operational status of the hardware and calculating system-wide performance metrics) are maintained throughout the system operation time, while others (for data related to the performance of an individual job) are created and destroyed dynamically, as needed.

6 Performance of the proposed solution

We are currently working on implementing a system with the architecture described. A prototype has been created and performance tests have been run. Experiments were conducted on the Moscow State University Lomonosov Supercomputer (1.7 Petaflops peak performance) (V. Sadovnichy, 2013).

To study the maximum data rate the prototype system can handle, we implemented a sensor that responded to each control message received (a call to the respective function) with a packet of data from one node, containing a 32-bit integer value. The number of data packets received and the time passing since receiving the first packet were calculated. The result was a speed of 350,000 packets per second on a Gigabit Ethernet network (which roughly corresponds to 230 Mbps). The receiving server was not fully loaded, and the flow speed was determined by the sending party's generation capacity.

In another experiment, we checked the system's ability to process data from multiple sources, by examining the average CPU load for about 5,500 computing nodes. The data speed at the server partition reached about 47,500 values per second, with the monitoring system generating less than 3%

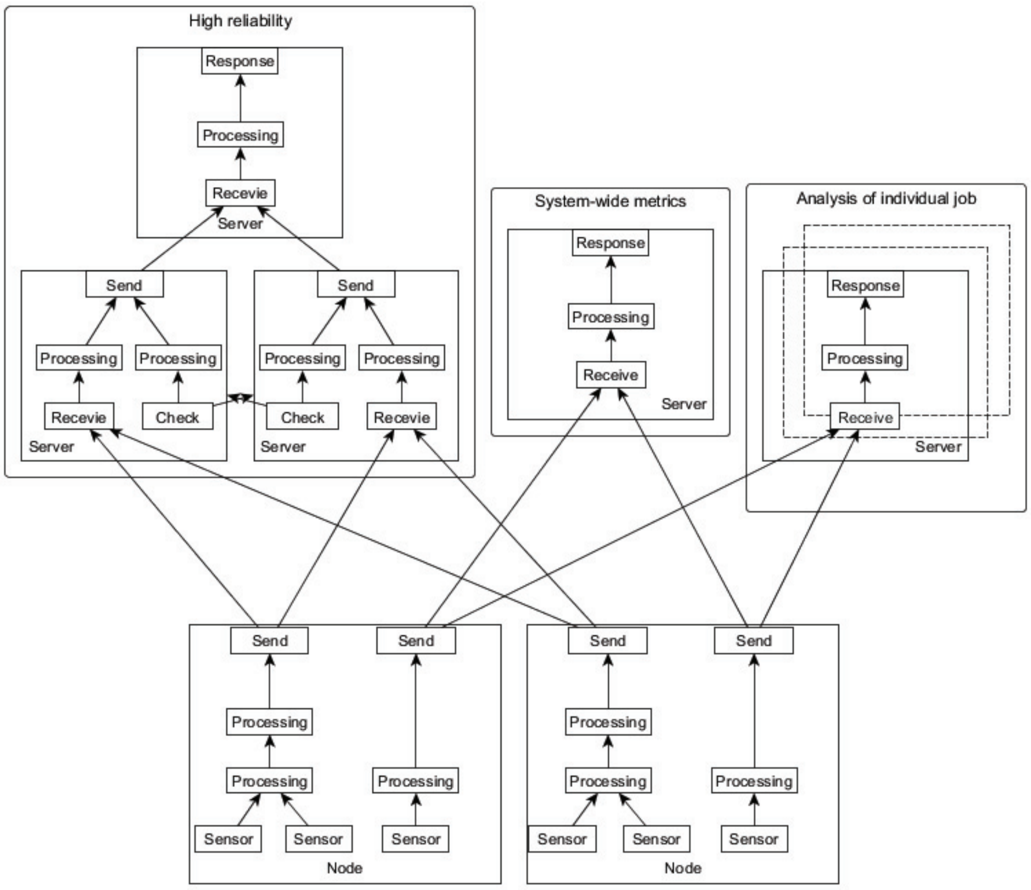

Figure 3: Example configuration of a monitoring system performing several tasks

load on one processing core on an Intel Xeon X5670 CPU running at 2.93 GHz. This experiment was also intended to check the stability of system operations. The system was perfectly stable for 35 days, without any signs of memory leaks, providing processed data all the time.

7 Conclusion and Future Work

The key contribution of this work is a new approach to building monitoring systems. The underlying principles used in the design of the DiMMon framework enable the building of monitoring systems suitable for processing data from existing and future computing systems. The system's modular design helps adapt it to new data sources and processing methods. Dynamic reconfiguration capabilities allow the creation of data processing components the moment a new object for processing appears, such as a new user application. Choosing the location for creating these components, depending on the current load of the computing system, helps ensure load balancing. Partially moving the data processing to computing nodes without serious overhead helps engage additional processing resources for handling monitoring data, thus ensuring monitoring system scalability. These proposed solutions help to solve BigData issues on processing big stream of monitoring data.

Currently work is in progress to implement data retrieval from sources necessary to use the system in production mode. In particular, there are plans for the system to be used as the source of data for the

OctoTron supercomputer autonomous maintenance system (A. Antonov, 2014) and for studying dynamic characteristics of application software when building Job Digests (A. Adinetz, 2013).

Acknowledgment. The reported study was partially funded by RFBR according to the research project No. 13-07-0775. This work was funded in part by the Ministry of Education and Science of the Russian Federation, Agreement No. 14.607.21.0006 (unique identifier RFMEFI60714X0006).

References

A. Adinetz, P. B. (2013). Job Digest – approach to jobs dynamic properties investigation on supercomputer systems. *Vestnik UGATU (scientific journal of Ufa State Aviation Technical University), 17*(2), 131-137.

A. Antonov, V. V. (2014). Securing of reliable and efficient autonomous functioning of supercomputers: basic principles and system prototype. *Vestnik UGATU (scientific journal of Ufa State Aviation Technical University), 18*(2), 227-236.

B. Ries, R. A. (1993). The paragon performance monitoring environment. *Proceedings of Supercomputing '93*, (pp. 850-859).

Cacti. (2015). Retrieved from Cacti - The Complete RRDTool-based Graphing Solution: http://www.cacti.net/

Collectd. (2015). Retrieved from Collectd – The system statistics collection daemon: https://collectd.org/

D. Gunter, B. T. (2002). Dynamic monitoring of high-performance distributed applications. *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, (pp. 163-170).

G. Eisenhauer, E. K. (1995). Falcon: on-line monitoring and steering of large-scale parallel programs. *Proceedings Frontiers '95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*, (pp. 422-429).

H. Jagode, J. D. (2009). A Holistic Approach for Performance Measurement and Analysis for Petascale Applications. *Computational Science*, pp. 686-695.

J. Mellor-Crummey, R. J. (2002). HPCVIEW: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing, 23*(1), pp. 81-104.

L. Adhianto, S. B.-C. (2010). HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience, 22*(6), pp. 685-701.

M. Kluge, D. H. (2012). Collecting Distributed Performance Data with Dataheap: Generating and Exploiting a Holistic System View. *Procedia Computer Science, 9*, pp. 1969-1978.

M. L. Massie, B. N. (2004). The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing, 30*(7), pp. 817-840.

Nagios. (2015). Retrieved from Nagios - The Industry Standard in IT Infrastructure Monitoring: http://www.nagios.org/

R. Mooney, K. P. (2004). NWPerf: a system wide performance monitoring tool for large Linux clusters. *IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, (pp. 379-389).

V. Sadovnichy, A. T. (2013). 'Lomonosov': Supercomputing at Moscow State University. In *Contemporary High Performance Computing: From Petascale toward Exascale* (pp. 283-307).

Zabbix. (2015). Retrieved from http://www.zabbix.com/

Zenoss. (2015). Retrieved from Zenoss Community - Open Source Network Monitoring and Systems Management: http://www.zenoss.org/